

Testing debuggability of compiler-optimised programs

Department of Informatics
King's College London

This dissertation is submitted for the degree of
Doctor of Philosophy

J. Ryan Stinnett

2026-04

Abstract

There are many scenarios where it is essential to debug the optimised version of a program (e.g. in-situ debugging of a production system or resource-heavy programs such as games which are unusable without optimisation). Debuggers for compiled programs often rely on *compiler-produced metadata* to reconstruct a familiar source-level view of program state. Reliable debug info enables not only interactive debugging, but also profiling, inspection, and modification abilities (such as program visualisation, tracing, live editing, and customisation) that assist developers and users throughout the software lifecycle.

This metadata describes various details of compilation including how data is represented (*type* information), how variables map to memory/register locations (*variable location* information), how binary code addresses map to source file/line/column positions (*source coordinate* information), how function calls have been affected by features such as inlining and tail calls (*function call* information), and how the stack may be “unwound” to walk the call chain (*frame* information). Unfortunately, current compilers often either eliminate this metadata entirely or produce erroneous metadata when optimisations are used.

The work in this dissertation focuses on the completeness and correctness of *variable location* and *function call* information. These metadata categories are frequently degraded by optimisations in mainstream compilers. With the goal of identifying gaps in debug info and exploring solutions which resolve entire classes of debugging issues, I examine four aspects of this problem space: classifying the experience expressible with current debug info formats, measuring coverage of variable location info, analysing function call consistency via call trees, and checking correctness of variable location info using a novel consistency property. Along the way, I define debug info correctness criteria for optimised programs using unoptimised compilation as an oracle, allowing the two compilations to differ within certain bounds only.

These approaches have discovered a variety of compiler bugs related to debug metadata handling. Further application of these techniques would enable compiler authors to improve the generation of program metadata and track debug info reliability over time to avoid regressions.

Acknowledgements

My supervisor, Stephen Kell, inspired me back in 2019 to investigate malleable software generally and debug info reliability more specifically. We began collaborating on the grant proposal which ultimately funded this work. I would like to thank him for taking a chance on me, especially given my somewhat unusual career path. He has always made time for our work and has helped me to improve my skills as a researcher and communicator.

Thanks to my second supervisor, Laurence Tratt, for helping me navigate the PhD process at King's. My colleagues at King's have made my day to day experience more enjoyable overall: Andrei Lascu, Tom Wallis, Edd Barrett, Lukas Diekmann, Jake Hughes, and Iti Shree.

Our industrial supporters at Arm and Pernosco have offered great feedback which has improved this work in various ways. Thanks in particular to Al Grant at Arm and Robert O'Callahan from Pernosco.

The LLVM debug info community has offered valuable feedback on our discovered issues and proposed improvements. The Sony / SN Systems team in particular has often made time to help: Orlando Cazalet-Hyams, Jeremy Morse, and Stephen Tozer.

Finally, thanks to my partner Maxine Champion for their support during this process. I do not think I would have had the courage to take up this challenge without them.

The work in this dissertation 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. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of any sponsor or supporter.

Contents

Abstract	1
Acknowledgements	2
1 Introduction	7
1.1 Background	8
1.1.1 Language implementations	8
1.1.2 Metadata vs. VM approaches	8
1.1.3 Kinds of metadata	12
1.1.4 Effects of optimisation	12
1.1.5 Causes of degraded metadata	13
1.1.6 Run-time optimisation prioritised	14
1.1.7 Complexity of mappings	15
1.1.8 Validation practices	17
1.2 Example	18
1.2.1 LLVM IR crash course	18
1.2.2 Debug info handling issues	20
1.3 Applications	22
1.3.1 Optimised programs	22
1.3.2 Offering users more abilities	23
1.4 Research questions	26
1.5 Thesis	26
1.6 Contributions	26
1.7 Publications	27
2 Defining the debugging illusion	28
2.1 Truthful vs. expected	28
2.2 State mapping: two approaches	29
2.2.1 The transition system model	30
2.2.2 Approach 1: dynamic deoptimisation	31
2.2.3 Approach 2: in-toolchain debug metadata	32

2.3	Debug info as residual computation	34
2.3.1	Debug info concepts	34
2.3.2	Residual computation at work	36
2.3.3	The essential problem: loss of state	38
2.3.4	Recovery functions vs. residual computation	39
2.3.5	Resolving the optimisation vs. debuggability dilemma	40
2.3.6	Optimisations that alter program semantics	41
2.4	Sub-properties of “expected”	42
3	Coverage metrics for variable debug info	44
3.1	Introduction	44
3.2	Understanding debug coverage	45
3.2.1	Debug info as residualised code	45
3.2.2	Research questions	46
3.3	A conceptual basis for coverage metrics	47
3.3.1	Liveness as a state space	47
3.3.2	A more realistic lifecycle	48
3.4	Defining a coverage metric	51
3.4.1	Naïve instruction-based metrics	51
3.4.2	Counting source lines, not instructions	52
3.4.3	Correcting accidents, permitting full coverage	53
3.5	Implementation	54
3.6	Evaluation	56
3.6.1	Experience with metrics	56
3.6.2	Case studies	57
3.6.3	Aggregate comparisons: a replication study	59
3.6.4	Experimenting with end-of-scope treatments	60
3.7	Conclusions and future work	61
4	Consistency checking of function call metadata using dynamic call trees	62
4.1	Introduction	62
4.2	Background	64
4.2.1	Dynamic call trees	64
4.2.2	Object- vs. source-level call trees	64
4.2.3	Limited differential testing of debug info	65
4.2.4	Compiler bugs vs. format bugs	66
4.3	Exploring divergent call trees	67
4.3.1	Collecting a call tree: the instruction-wise algorithm	67
4.3.2	Collecting a call tree: implementation	69
4.3.3	Exploratory study outline	70

4.3.4	Expected divergences	74
4.3.5	Unexpected divergences	78
4.3.6	Handling traces with inlining	80
4.3.7	Limitations of the study method	85
4.4	Quantifying divergence across compilers	85
4.4.1	Measuring divergence	87
4.4.2	Differences across compilers	88
4.4.3	Differences across optimisation levels	88
4.4.4	Differences across codebases	89
4.4.5	Compiler bugs found	90
4.5	Conclusions and future work	90
5	Consistency checking of variable debug info via symbolic execution	91
5.1	Introduction	91
5.1.1	Understanding local variable metadata	92
5.1.2	Problem statement	93
5.2	Illusion consistency up to local variables	94
5.3	Implementing a tool	95
5.3.1	Extending symbolic execution	96
5.3.2	Consistency checking in practice	99
5.3.3	Current limitations	100
5.3.4	Consistency checking step by step	102
5.3.5	Examples	106
5.4	Evaluation	107
5.4.1	Confirming existing issues	108
5.4.2	Finding new issues	109
5.4.3	False positives	110
5.5	Conclusions and future work	112
6	Related work	114
6.1	Early debugging techniques	114
6.2	Approaches to debugging optimised code	115
6.3	Print debugging	120
6.4	Functional language debugging	121
6.5	Applications of debug info	123
6.6	Debug info coverage	124
6.7	Compiler testing	124
6.8	Debug info validation	127
6.9	Improving debug info	129

7	Conclusions and future work	130
7.1	Summary	130
7.2	Further projects	131
7.2.1	Residualisation	131
7.2.2	Push beyond internal limits of debuggability	133
7.2.3	Explore external limits of debuggability	134
	Bibliography	135

Chapter 1

Introduction

When something unexpected happens with software, a wide variety of dynamic program analysis tools are available to help, including debuggers, profilers, tracers, and more. These tools assist investigation by helping developers to understand one or more aspects of program execution.

Software dynamic analysis tools depend on many pieces of a programming language implementation working together to provide a reliable experience for users. This is doubly true when one considers compiler-optimised programs, which have gone through many transformations to eke out additional run-time performance.

If something goes wrong along the way, tools may be unable to correctly identify the source-level program state of interest. In debuggers, a common failure of this kind is the message “variable optimised out” when attempting to print or evaluate a local variable. This message often occurs even when a variable remains represented in the program; it is triggered when the *debug info*, not the variable, is missing. Debug info is compiler-generated metadata that describes many aspects of the output object program and its relation to the input source program, as we will see later on this chapter. When coverage is lacking like this, the effectiveness of tooling is degraded.

The research described in this dissertation focuses on resolving foundational reliability problems that lead to failures like that “variable optimised out” message in dynamic program analysis tools that rely on compiler-generated metadata.

We start this chapter by examining programming language implementations and following several implementation pathways through to their impact on software tooling with several examples along the way. If you’d like a preview of where this is all heading first, feel free to jump ahead to the thesis statement (§1.5) and contributions (§1.6).

1.1 Background

1.1.1 Language implementations

Programming languages are typically implemented either as an ahead-of-time (AOT) compiler to some run-time object format or as a run-time interpreter from source to machine code. There are many further distinctions one could make about each.

For discussion purposes here in this dissertation, we will simplify and distinguish the two as follows:

- a *compiler* transforms the program ahead of time into object code for later execution, while also committing to some level of optimisation during transformation and discarding the input source program
- an *interpreter* executes the program directly from source at run time, permitting on-demand optimisation using various techniques and retaining access to the input source program

In some approaches, the compiler-interpreter distinction becomes a bit fuzzier e.g. when an interpreter uses just-in-time (JIT) compilation. When applying the above definitions, we view a JIT-compiling interpreter as primarily an interpreter for our purposes, as they typically use a similar debugging model to other interpreters (as explained further in the next section).

These two language implementation strategies lead naturally to two different approaches to enable software tooling, which includes debuggers, profilers, tracers, coverage tools, and any other dynamic analysis tool which may be interested in observing some aspect of program execution.

1.1.2 Metadata vs. VM approaches

For a compiler, the various transformations from source to object program occur at compilation time. Tools like debuggers need to be able to map between object and source program states, and this has long been achieved by *metadata* generated by the compiler in a standard format, making it cross-vendor, decoupled and tool-agnostic (e.g. useful also to profilers, tracers, postmortem debugging). In this metadata model, the compiler attempts to describe everything a tool might want to know, giving tool authors a high degree of flexibility in terms of how that info is applied and enabling novel applications not originally anticipated (more detail in §1.3). There can still be limitations however, as tools may want to know more than anticipated by compiler authors. Fig. 1.1 shows an example C function referenced throughout this chapter. Fig. 1.2 shows an example of metadata for that same function in the DWARF format [DWARF Debugging Information Format Committee 2017] commonly used on Unix-like platforms.

Forgoing these benefits, interpreters are often implemented as language “virtual machines” (VMs), e.g. for Java or JavaScript, that use in-VM debug servers and quite complex *dynamic*

```

1 int loop(int input) {
2     int current;
3     for (current = input; current < input + 10; ++current) {
4         if (current < 3)
5             return 0;
6     }
7     return 1;
8 }

```

Fig. 1.1: A simple C function with a loop and conditional.

DW_TAG_subprogram	Addr	Line	Column	Flags
DW_AT_low_pc (0x00)	----	-----	-----	-----
DW_AT_high_pc (0x25)	0x00	1	0	is_stmt
DW_AT_name ("loop")	0x03	3	3	is_stmt prologue_end
DW_AT_decl_line (1)	0x10	0	3	
	0x13	4	9	is_stmt
DW_TAG_formal_parameter	0x15	3	33	is_stmt
DW_AT_location (DW_OP_reg5 RDI)	0x1a	3	3	
DW_AT_name ("input")	0x1c	0	3	
DW_AT_decl_line (1)	0x21	8	1	is_stmt
	0x22	0	1	
DW_TAG_variable	0x24	8	1	
DW_AT_location	0x25	8	1	end_sequence
[0x00, 0x06):				
DW_OP_reg5 RDI				
[0x13, 0x18):				
DW_OP_breg2 RCX+1,				
DW_OP_plus_uconst 0x1,				
DW_OP_stack_value				
[0x22, 0x25):				
DW_OP_breg2 RCX+1,				
DW_OP_plus_uconst 0x1,				
DW_OP_stack_value				
DW_AT_name ("current")				
DW_AT_decl_line (2)				

Fig. 1.2: An example of debug info in the DWARF metadata format for the function in Fig. 1.1. On the left, various details of a function and its parameters and local variables are described. The DW_AT_location portions state the location of a source variable as a function of program state (memory, registers) using a Turing-complete stack machine language (§2.3.1). On the right, the line table maps PCs to source coordinates. There are also a few additional flags such as is_stmt, which denotes a recommended breakpoint location.

deoptimisation techniques [Zurawski and Johnson 1991; Hölzle et al. 1992]. Instead of supplying static metadata (produced by AOT compilation and existing alongside a program binary) for separate processing and consumption as needed by tools, this debug server approach requires tools to be built specifically for dynamic debug APIs unique to each VM. Fig. 1.3 demonstrates the complexity this creates for tool authors by comparing a small subset of the virtual machine debug APIs offered by two production-grade language implementations: the SpiderMonkey JavaScript implementation [Mozilla 2014] and the OpenJDK Java implementation [Oracle 2000]. Along with complexity, this model can introduce obstacles since the debug server model requires accessing program state through VM-vendor-defined APIs. Tool authors may be out of luck if the API does not offer all the abilities needed for a tooling feature. In recent years, the “integration matrix” posed by custom debug APIs (N debug servers $\times$ M tooling clients) has been somewhat ameliorated by efforts towards the Debug Adapter Protocol [Microsoft 2017], which has been adopted by commonly-used editors for their debugging interfaces and is supported via debug server adapters for many language implementations.

While interpreters *could* choose to emit the same standard metadata as compilers, in practice there are currently only a few experimental interpreter-based language implementations that do so [Kell and Irwin 2011]. It is unclear precisely why metadata is so rarely found in interpreter-based language implementations. It may be a cyclical cultural issue: most interpreters do not emit metadata, so authors of new interpreters may not consider it important. Interpreters are often making just-in-time program transformation decisions as quickly as possible (balancing the potential gains of further transformation at run time with its contribution to total program evaluation time), so it may be that interpreter authors deem emission of standard metadata to not be worth the cost to performance. A language VM API that emits standard metadata only upon request (avoiding concerns of adding fixed overhead to all evaluations) could be an interesting mix of approaches for future language implementers to investigate.

Comparison of virtual machine debug APIs

SpiderMonkey JavaScript VM	OpenJDK Java VM (JDI)
• Debugger context object - addDebuggee(...) - findScripts(...) - onEnterFrame = (frame) => { ... } • Debugger.Script objects - lineCount - source - setBreakpoint(...) • Debugger.Frame objects - arguments - environment - eval(...) • Debugger.Environment objects - names() - getVariable(...) - setVariable(...)	• VirtualMachine context interface - allClasses() - allThreads() - eventRequestManager() - redefineClasses(...) • EventRequestManager interface - createMethodEntryRequest() • ReferenceType interface (class, interface, etc.) - allFields() - allLineLocations() - allMethods() • ThreadReference interface - frames() • StackFrame interface - getArgumentValues() - getValue(...) - setValue(...) - visibleVariables(...)

Fig. 1.3: A comparison of the virtual machine debug APIs provided by the SpiderMonkey JavaScript implementation [Mozilla 2014] and the OpenJDK Java Debug Interface (JDI) implementation [Oracle 2000]. These listings show only a small subset of the properties and functions available in each VM debug API. Each API has its own feature set. While some high-level concepts are shared, there are also less common abilities that are only be found in debug APIs offered by specific language implementations. Other VMs also offer their own dynamic debug server APIs, but the sheer number of different APIs can pose an integration implementation burden for each potential tool wishing to be an API client.

1.1.3 Kinds of metadata

Metadata describes *how* the compiler has “realised” the program, including:

- *type* information:
how data is represented in terms of machine-level bytes,
along with source-level data types defined in the program
- *variable location* information:
how variables map to memory/register locations
- *source coordinate* information:
how binary code addresses map to source file/line/column positions
- *function call* information:
how function calls have been affected by features such as inlining and tail calls
- *frame* information:
how the stack may be “unwound” to walk the call chain

Type information is largely fixed for a given source program, but the others become more complex with greater optimisation. The simplest is frame information, which maps stack frames to a call chain or backtrace. Previous work has shown that both deep consistency checking of frame info and from-scratch *synthesis* are feasible [Bastian et al. 2019].

The DWARF metadata example in Fig. 1.2 offers a glimpse of how one metadata format stores source coordinate and variable location details. This example shows various details of a function and its parameters and local variables. The DW_AT_location portions state the location of a source variable as a function of program state (memory, registers) using a Turing-complete stack machine language (§2.3.1). The example also shows the line table which maps PCs to source coordinates.

The work in this dissertation is concerned with *variable location* and *function call* information. These metadata categories are frequently degraded by optimisations in mainstream compilers.

While DWARF aspires to support a large variety of source languages and programming paradigms via its extensible design, it is primarily used by imperative language compilers for C, Rust, etc. The experiments in this dissertation similarly focus on these C-like languages, as they have the most mature debug info emission tooling.

1.1.4 Effects of optimisation

Unfortunately, the metadata (also referred to as debug info) produced by compilers is often incorrect or missing after optimisation. For example, a compiler generating metadata (Fig. 1.2) for the function `loop` (Fig. 1.1) might neglect to cover some portion of PC range for variable

current even though it is still live, which would cause the value of this source variable to be inaccessible to debuggers and other tools, leading to failure messages like “variable optimised out”. A compiler might also fail to emit line table entries corresponding to some statements in the source program, such as the conditional on line 4. If there are no line table entries that map to a given source line, then tools can’t stop at that line (which they need in order to support e.g. breakpoints), as they must be able to map the source coordinate “line 4” to an instruction address in the running program in order to stop there.

On asking the debugger to pause at function `loop`, line 4, and inspect variable `current`, with optimised code one might see execution run past that line without pausing, or the debugger claim that no such line/variable exists if any debug info handling issues have occurred during optimisation.

Today’s dominant compiler design philosophy prioritises program optimisation above other concerns: any additional work to support debuggers and other tools should “not limit or compromise” run-time program performance [Brender et al. 1998]. This is an engineering tension but not a mutual exclusion, since (as we highlight in §1.1.7, with further details discussed in §2.3) metadata can be seen as a program itself, in principle capable of “undoing” arbitrary optimisation, if it uses a sufficiently expressive language.

Interpreters that involve JIT compilation typically use dynamic deoptimisation techniques [Zurawski and Johnson 1991; Hölzle et al. 1992] to enable debugging optimised programs. These approaches switch to an unoptimised version of the program on the fly when debugging is requested, performing low-level surgery to program state such as *on-stack replacement* [Guo 2014] to move between different execution modes. Many of these techniques only support switching to an unoptimised mode at specific program points. Those “safepoints” also act as optimisation barriers, reducing run-time performance and preventing the optimiser from achieving its full potential.

1.1.5 Causes of degraded metadata

While a few small segments of the compiler author community do focus on debug info (and even produce debug info updating guides for optimisation pass authors [LLVM Project 2020a]), compilers often end up degrading debug info due to a mixture of their own design choices and expressiveness limits in current debug info formats.

In the category of compiler design choices, a compiler’s intermediate representation (IR) will need to be able to express the sometimes complex relationships between source variable values and program IR values as optimisation proceeds. After some kinds of optimisations, it is necessary to make use of multiple values from the optimised program’s state to express a given source variable value. If the compiler’s IR is unable to keep track of these more advanced relationships between the source and object state spaces, then it is likely that such variable values will be lost. While compiler authors could choose to refrain from optimising in cases

```

1 int if_conversion(int n) {
2     int r;
3     if (n) {
4         r = n + 1;
5     } else {
6         r = n - 1;
7     }
8     return r;
9 }

```

Fig. 1.4: An example C function with an “if-else” conditional block which compilers can convert into a single branchless select instruction (as seen in e.g. LLVM IR) that covers both paths. Current debug info formats are unable to represent source coordinates after this transformation, as it would require source coordinates to depend on additional program state beyond the current PC.

where the compiler’s architecture complicates emission of correct debug info, in practice today’s compilers often allow a run-time performance optimisation to proceed even if it may lead to a loss in debug info quality (discussed further in §1.1.6).

In the category of debug info format expressiveness, a format should aim to express all of the transformations a compiler has made to a program. For example, current compilers can transform the two paths of an “if-else” conditional block (such as the one shown in Fig. 1.4) into a single branchless select IR instruction that covers both paths (typically emitted using predicated machine instructions where available, e.g. `cmov` on Intel architectures). To accurately represent the current source position, debug info would need to make the source coordinates for that single IR instruction depend on program state (effectively evaluating the conditional at debug time), as it corresponds to two distinct assignments to `r` in the source program. Current debug info formats do not allow source coordinates to depend on other program state like this. Instead, only the current program counter (PC) is allowed as an input when determining source coordinates. This means source locations for cases such as these branchless instructions will be lost and debugging users will be unable to examine these program points.

1.1.6 Run-time optimisation prioritised

Current compiler maintainer policies allow optimisation passes to change and evolve even if those compiler pass code changes reduce tooling reliability. This may at least partly be a problem of incentives: optimisation pass authors typically have a specific goal in mind to improve some aspect of run-time performance. Debugging and other software tools can feel like a far-away concern. Such tools are used well after compilation, and many times not at all, so it can feel difficult to justify investment in this area.

Thinking more holistically, this would seem to be a short-sighted view. Software defects are costly (both in time and money) to repair [Krasner 2022]. When viewed in this economic frame, it seems reasonable (at least to this author) that we should be doing everything we can

to improve software tool reliability so that bugs can be found and fixed more rapidly.

Despite earlier work to protect debuggability by restricting optimisations (e.g. ensuring “expected behaviour” [Zurawski and Johnson 1991] at certain “inspection points” [Zellweger 1983]), compiler authors consider such restrictions too limiting: they assume that most compiler users demand the best possible run-time performance even if debuggability is sacrificed to achieve it.

We can see this assumption at work in LLVM’s debug info updating guide [LLVM Project 2020a]. While the guide describes scenarios for preserving debug info across a variety of optimisation pass transformations, it also assumes debug metadata will be lost in some situations. There are even cases (such as complex relationships to source coordinates) where the current policy *requires* pass authors to delete debug info to avoid potential confusion (e.g. when hoisting some portion of a conditional block before the conditional that guards it), especially for more automated use cases like coverage and profiling that expect to know with certainty whether a given source line was reached.

For example, looking again at the conditional block in Fig. 1.4, an optimisation might hoist the `else` block above the conditional guard. LLVM’s policy requires the source coordinate info (e.g. “line 6” in this case) to be dropped. If the source coordinates for this now-hoisted line were instead preserved, it could lead to a confusing experience in e.g. a profiler or coverage tool where it may look like the `else` branch is always taken. This policy choice can however have unfortunate consequences for other tools, such as stepping debuggers, where it may no longer be possible to stop at the affected source line since the info needed to achieve has been removed.

1.1.7 Complexity of mappings

Metadata mappings between source and object program state spaces can be complex. Optimisations frequently *move* code (relative to its position in source) and variables (between multiple locations, in registers and memory), or may *eliminate* them entirely. This lessens the alignment between source and binary, complicating the mappings. As a result, metadata must be expressive. DWARF encodes mappings as bytecode programs for three different abstract machines: frame, source coordinate, and variable location. The latter of these is Turing-complete (§2.3.1). Debuggability is limited largely by the (“internal”) effort required of compiler authors to generate correct and complete metadata, including testing to catch when passes degrade it, but also sometimes by the metadata format’s (“external”) expressiveness. We will discuss the possibilities and trade-offs related to debug-time computation in more detail throughout §2.3.

The complexity of debug info mappings is often correlated with progress through the optimisation pipeline. An exploration of this relationship is depicted in Fig. 1.5. This data was collected from Clang 18 compilations of the Git codebase at -O2. Inlining was disabled (-fno-inline) for these compilations, as it complicates comparative data analysis across

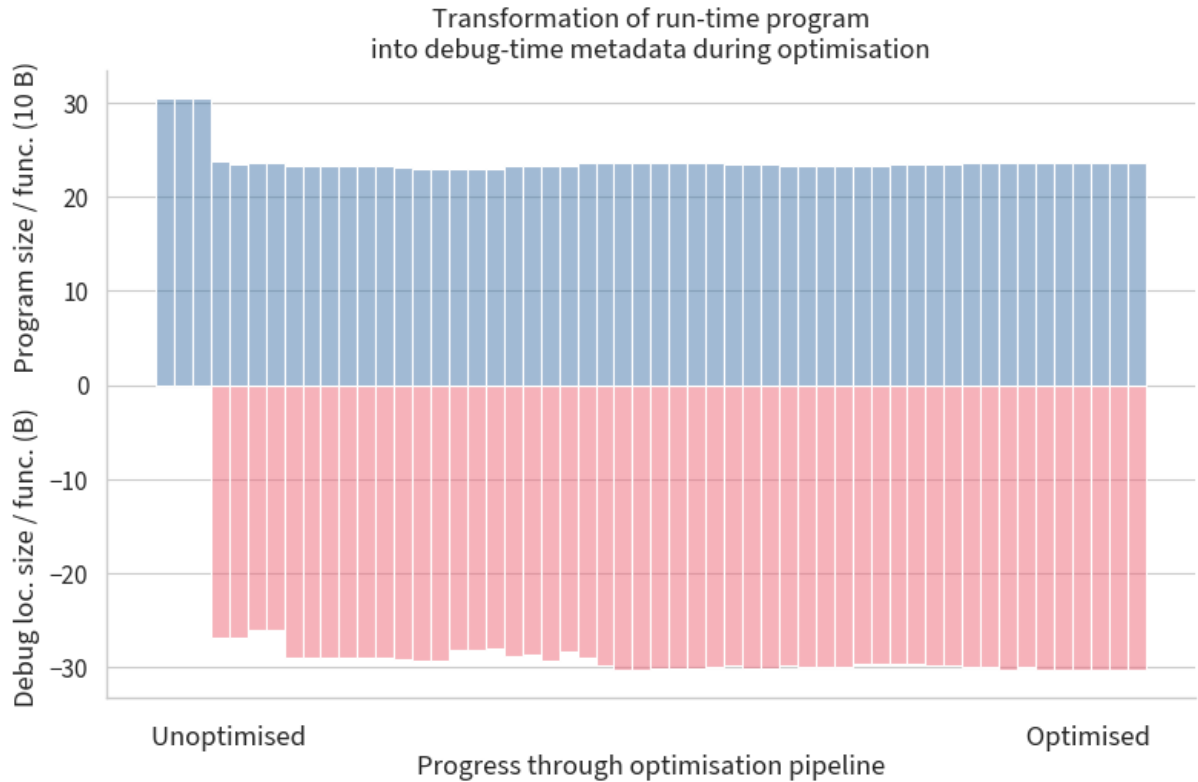


Fig. 1.5: Exploration of the transformation from run-time program instructions into debug-time metadata, which is correlated with the program’s progress through the optimisation pipeline. Data collected from Clang 18 compilations of the Git codebase at `-O2`. Inlining is disabled (`-fno-inline`), as it complicates comparative data analysis across passes. For each horizontal step from left to right, the compiler is allowed to complete slightly more of the optimisation pipeline, which can be controlled via `-passes`. Mean run-time program size per function is shown on the top, presented in 10s of bytes, allowing comparative trends to be seen more easily. Mean debug-time variable location expression (which describes where to find variable values in program state) size per function is shown on the bottom, presented in bytes.

passes. For each horizontal step from left to right, the compiler is allowed to complete slightly more of the optimisation pipeline, which can be controlled via `-passes`.

As more of the run-time program is transformed and optimised by each successive pass moving it further and further away from the original input, the debug-time description emitted in compiler metadata will typically grow to record additional steps needed to map between source and object program state spaces. This is indeed the trend we see in Fig. 1.5, which plots mean run-time program size per function on the top versus mean debug-time variable location expression (which describes where to find variable values in program state) size per function on the bottom.

```

1 void simple_memcpy_loop(int *dest, const int *src, unsigned nelems) {
2     for (unsigned i = 0; i != nelems; ++i)
3         dest[i] = src[i]; // DexLabel('target_simple_memcpy_loop')
4 }
5 // DexLimitSteps('i', 0, 4, 8, on_line=ref('target_simple_memcpy_loop'))
6 // DexExpectWatchValue('nelems', '16', on_line=ref('target_simple_memcpy_loop'))
7 // DexExpectWatchValue('src[i]', '3', '7', '1', on_line=ref('target_simple_memcpy_loop'))

```

Fig. 1.6: An interaction-based debugging test using the Dexter [Bedwell 2018] framework. The source is compiled and loaded into a debugger. The Dex-prefixed annotations can control movement through the program and add assertions that check program state as seen in the debugger. Example adapted from LLVM’s [cross-project tests](#).

1.1.8 Validation practices

Compiler authors use tests to identify *how much* debuggability is lost, in *which passes*, and perhaps to “blame” certain *changes*. Techniques include:

Synthetic coverage testing This approach attaches meaningless but complete metadata before a pass using e.g. LLVM’s debugify infrastructure, then tests how much remains after.

Interaction-based testing This technique drives a real debugger over test cases serving as “oracles” of expected behaviour (e.g. “on hits of line 10, variable x should be 1, then 2, then 3”). Here oracles are written by hand, so coverage is gated by human effort and limited to program locations covered by test inputs.

Dexter [Bedwell 2018] is one example of the latter interaction-based testing approach. It is integrated into the LLVM codebase and can drive both LLDB and GDB debuggers. Fig. 1.6 shows an example function with additional Dexter annotations added (comment lines with the Dex prefix) by the test author. The framework compiles test code with a given compiler and loads it into a debugger. Some Dexter annotations, such as `DexExpectWatchValue`, identify source variables to be examined in the debugger as well as asserting their expected values at different program points. Other Dexter annotations, such as `DexLimitSteps`, control which program points are examined and limit test execution time.

As of 2025 (7 years after Dexter appeared publicly), there are roughly 30 test files using Dexter in the LLVM code base compared to the thousands of debugging-focused test files using other techniques. A much larger amount of manual test author effort would be needed to cover all debugging integration aspects using Dexter, but so far this has not materialised in the LLVM community.

1.2 Example

The previous sections have sketched the basic outline of debug info. We now examine this metadata in context by returning to the example C function `loop` from [Fig. 1.1](#), which makes use of a few basic language features, including loops, conditionals, and assignments. Inspecting the compiler’s IR as the function passes through the optimisation pipeline highlights how debug info is represented by the compiler internally and also reveals how it can degrade.

1.2.1 LLVM IR crash course

Figures in this example ([Fig. 1.7](#), [Fig. 1.8](#), [Fig. 1.9](#)) show fragments of LLVM IR for the function in [Fig. 1.1](#). While we assume the reader is familiar with the general concepts of compiler intermediate representations, we will review the key elements of LLVM IR that we make use of here so that these figures are comprehensible.

LLVM IR is organised into functions, with each function containing one or more basic blocks. Each basic block has one or more instructions, and uses only sequential internal control flow. A basic block starts with a label and ends with a terminator instruction, e.g. `ret` (return) or `br` (branch). Block terminator instructions either return from the entire function or jump to one of the function’s basic blocks. Some LLVM IR instructions produce new values. Values are in static single assignment (SSA) form, meaning each IR value is assigned in exactly one statement [[Singer 2022](#)]. The `@` prefix is used before all function name identifiers. The `%` prefix appears before all function argument and SSA value identifiers.

The `add` instruction adds two values together. The `icmp` instruction performs one of several integer comparisons. The `icmp slt` variant (integer compare, signed less-than) seen in the figures here interprets operands as signed values and returns true if the first operand is less than the second. Basic blocks can optionally begin with one or more `phi` instructions. The `phi` instruction computes a value using different expressions depending on which of the 2 or more incoming edges was travelled to reach the block containing the instruction.

There are two major debug info aspects that we can see represented in LLVM IR:

Source coordinates LLVM IR operations may record the source coordinates they correspond to in the program source. For example in [Fig. 1.7](#), the `br` branch instruction on IR line 20 is annotated with `1: 4 c: 9` meaning source line 4, column 9, where we see a conditional in the original program ([Fig. 1.1](#)). These source coordinates flow through optimisation (with edits where required after moving, splitting, and merging instructions in various ways) and define the mapping from program counter to source coordinates in debug info.

Variable locations LLVM IR also describes the mapping from IR SSA values to source variables in program state through several *debug intrinsics*. Some debug intrinsics, such as `dbg.value`, describe a link between the *value* of a source variable and an IR

```

1  define i32 @loop(i32 %input) {
2  entry:
3      @dbg.value(i32 %input, name: "input")
4      @dbg.value(i32 %input, name: "current")
5      %cmp1 = icmp slt i32 %input, 3
6      %0 = add i32 %input, 9
7      br label %for.body, 1: 3 c: 3
8
9  for.cond:
10     %inc = add i32 %current.06, 1, 1: 3 c: 47
11     @dbg.value(i32 %current.06, name: "current",
12         !DIExpression(DW_OP_plus_uconst, 1, DW_OP_stack_value))
13     %cmp = icmp slt i32 %current.06, %0, 1: 3 c: 33
14     br i1 %cmp, label %for.body, label %cleanup, 1: 3 c: 3
15
16  for.body:
17     %current.06 = phi i32 [ %input, %entry ], [ %inc, %for.cond ]
18     @dbg.value(i32 %current.06, name: "current",
19         !DIExpression(DW_OP_plus_uconst, 1, DW_OP_stack_value))
20     br i1 %cmp1, label %cleanup, label %for.cond, 1: 4 c: 9
21
22  cleanup:
23     %retval.0 = phi i32 [ 0, %for.body ], [ 1, %for.cond ]
24     ret i32 %retval.0, 1: 8 c: 1
25 }

```

Fig. 1.7: LLVM 11 IR before the loop strength reduction optimisation pass for the function in Fig. 1.1. We can see several debug-info-related IR features describing *source coordinates* (the 1: and c: annotations at the end of various lines) and *variable locations* (via the debug intrinsic `dbg.value`, which links a source variable's value to an IR SSA expression).

SSA expression. Other debug intrinsics describe a connection between the *address* that a source variable occupies in memory and an IR SSA expression. (Other debug intrinsics beyond `dbg.value` will be described when they appear in examples.) These LLVM debug intrinsics borrow the stack machine expression language found in the DWARF variable location format. For example in Fig. 1.7, lines 18–19 state that from this program point forward, the value of the source variable `current` equals the value of the SSA-variable-referencing expression `%current.06 + 1`, until the next debug intrinsic that describes the same source variable (if any).

The snippets of LLVM IR presented here and elsewhere in this thesis have been edited and simplified to improve readability. In particular, source variable names have been inlined into debug intrinsics and source coordinates have been compactly displayed as 1: 4 c: 9 annotations (meaning source line 4, column 9). In actual LLVM IR, these details are recorded as separate metadata nodes and each of these usage sites references a corresponding metadata node identifier.

```

1  define i32 @loop(i32 %input) {
2  entry:
3      @dbg.value(i32 %input, name: "input")
4      @dbg.value(i32 %input, name: "current")
5      %cmp1 = icmp slt i32 %input, 3
6      %0 = add i32 %input, 9
7      %1 = add i32 %input, -1, l: 3 c: 3
8      br label %for.body, l: 3 c: 3
9
10 for.cond:
11     @dbg.value(i32 undef, name: "current",
12         !DIExpression(DW_OP_plus_uconst, 1, DW_OP_stack_value))
13     %lsr.iv.next = add i32 %lsr.iv, 1, l: 3 c: 33
14     %cmp = icmp slt i32 %lsr.iv.next, %0, l: 3 c: 33
15     br i1 %cmp, label %for.body, label %cleanup, l: 3 c: 3
16
17 for.body:
18     %lsr.iv = phi i32 [ %1, %entry ], [ %lsr.iv.next, %for.cond ]
19     @dbg.value(i32 undef, name: "current",
20         !DIExpression(DW_OP_plus_uconst, 1, DW_OP_stack_value))
21     br i1 %cmp1, label %cleanup, label %for.cond, l: 4 c: 9
22
23 cleanup:
24     %retval.0 = phi i32 [ 0, %for.body ], [ 1, %for.cond ]
25     ret i32 %retval.0, l: 8 c: 1
26 }

```

Fig. 1.8: LLVM 11 IR after the loop strength reduction pass for the function in Fig. 1.1. Several variable value debug intrinsics have become “disconnected” from the run-time IR, denoted here by the `undef` values (lines 11, 19) appearing where there should instead be a reference to an IR SSA value if the intrinsic were still correctly connected to the IR program. This `undef` value means the affected debug intrinsics are incomplete, meaningless statements that now fail to describe the link between a source variable and IR SSA values.

1.2.2 Debug info handling issues

Fig. 1.7 shows the same loop function from Fig. 1.1 represented as LLVM IR taken just before the loop strength reduction pass in the LLVM 11 optimisation pipeline.

In LLVM 11, this particular pass was manipulating run-time program IR values without considering the debug intrinsics that may depend on them (the debug intrinsics were entirely ignored by the pass). When it transformed various bits of the run-time loop IR, the debug intrinsics became “disconnected” from the IR SSA values they reference as they were not also transformed to match the new IR. We can see this disconnected state represented in Fig. 1.8 where several `dbg.value` intrinsics now state an IR input to the debug stack-machine expression of `undef` (undefined) instead of referencing a defined value as they did before this pass. This `undef` means these debug intrinsics are now incomplete, meaningless statements that fail to describe the relationship between a source variable and IR SSA values.

In LLVM 12, this issue was resolved, as we can see from IR extract in Fig. 1.9. The pass makes similar transformations to the run-time IR while also updating the debug intrinsics to

```

1  define i32 @loop(i32 %input) {
2  entry:
3      @dbg.value(i32 %input, name: "input")
4      @dbg.value(i32 %input, name: "current")
5      %cmp1 = icmp slt i32 %input, 3
6      %0 = add i32 %input, 9
7      %1 = add i32 %input, -1, l: 3 c: 3
8      br label %for.body, l: 3 c: 3
9
10 for.cond:
11     @dbg.value(i32 %lsr.iv, name: "current",
12         !DIExpression(DW_OP_plus_uconst, 2, DW_OP_stack_value))
13     %lsr.iv.next = add i32 %lsr.iv, 1, l: 3 c: 33
14     %cmp = icmp slt i32 %lsr.iv.next, %0, l: 3 c: 33
15     br i1 %cmp, label %for.body, label %cleanup, l: 3 c: 3,
16
17 for.body:
18     %lsr.iv = phi i32 [ %1, %entry ], [ %lsr.iv.next, %for.cond ]
19     @dbg.value(i32 %lsr.iv, name: "current",
20         !DIExpression(DW_OP_plus_uconst, 2, DW_OP_stack_value))
21     br i1 %cmp1, label %cleanup, label %for.cond, l: 4 c: 9
22
23 cleanup:
24     %retval.0 = phi i32 [ 0, %for.body ], [ 1, %for.cond ]
25     ret i32 %retval.0, l: 8 c: 1
26 }

```

Fig. 1.9: LLVM 12 IR after the loop strength reduction pass for the function in Fig. 1.1. The disconnected variable value debug intrinsics (lines 11, 19) seen in Fig. 1.8 that had undef inputs to their stack-machine expressions (rendering them incomplete and meaningless) have been repaired via additional debug handling code. They now correctly reference the IR SSA value `%lsr.iv` (“loop strength reduction index value”) introduced by the pass.

reference the IR SSA value `%lsr.iv` (“loop strength reduction index value”) introduced by the pass.

LLVM authors fixed this particular issue in LLVM 12 by adding roughly 60 lines of debug info handling code into the transformations applied by the loop strength reduction pass. Unfortunately, that additional debug manipulation code is both specific to that pass¹ and only handles a subset of the possible transformations the pass can perform. These partial, pass-specific debug info fixes were added to the loop strength reduction pass as part of LLVM 12 which was released in 2021.

Looking again at the code for this same pass in 2025, we see that after 4 more years of compiler development, pass-specific debug handling code has grown to roughly 700 lines (more than a 10x increase). The additional code attempts to handle further transformations the pass can perform, amounting to 10% of this particular pass’s code overall. Although this debug

¹From a compiler-maintenance perspective, it would be better if as much debug info handling code as possible was instead generalised across passes, unlike what was done here. A more general approach would enable pass authors to focus on their run-time optimisation while “automatically” getting debug info support from generalised infrastructure elsewhere in the compiler. See §7.2.2 for further notes on this.

handling code has grown to cover more cases over time, it may still not be complete: current approaches offer no assurance of debugging completeness, so this kind of code is instead only reactively improved when users complain via bug reports. Each of these lines of pass-specific debug handling code adds to the compiler maintenance burden and complicates the work of pass authors, most of whom are not debug info experts.

1.3 Applications

Debug info is used by many tools and workflows. In this section, we discuss just a few of the many user and developer workflows enabled by this program metadata.

1.3.1 Optimised programs

One common workaround to these debugging reliability issues with optimised programs is to simply avoid them entirely by rebuilding without optimisation. While this heavy-handed approach sometimes suffices, there are many use cases that depend on the debugging information for optimised code.

Some codebases cannot be built without optimisation (e.g. the Linux kernel which contains macros and inline assembly blocks that assume certain optimisations like inlining are always enabled [[Linux contributors 2007](#)]), and others cannot meaningfully be run without it (e.g. resource-heavy programs such as games which are unusable without optimisation [[Microsoft 2025a](#); [Pendleton 2023](#)]). Some bugs occur only on optimised “full speed” code (e.g. faulty optimisation passes producing invalid run-time programs), and some tools are only useful on the same (e.g. profilers). Deploying partial workarounds when facing difficulties debugging optimised code (rebuilding without optimisation) brings costs to developers (e.g. rebuilding takes time) and does not help in-the-field bug reporting by end users (being optimised, deployed binaries are frequently undebuggable).

Some optimisations also leverage the assumption that programs are free of undefined behaviour (UB). For example, compilers may remove live code blocks when working under the “anything goes” UB semantics of languages like C. After applying optimisations that make use of the UB-free assumption when transforming the program and associated debug info, tools like debuggers should continue to function as expected (assuming correct debug info), as long as the UB-free assumption truly holds. If instead undefined behaviour *is* present, then the debug info may be meaningless and tools may behave erratically. The same logic applies to the debug info checking approaches developed in this thesis when confronted with UB.

This does mean that investigating program bugs *caused* by UB remains particularly tricky, as both the run-time program and debug metadata may diverge from the programmer’s expectation. I hope the research community will investigate additional debug metadata checking techniques that are able to cope with UB. For other bug categories (logic errors, incorrect API usage, etc.),

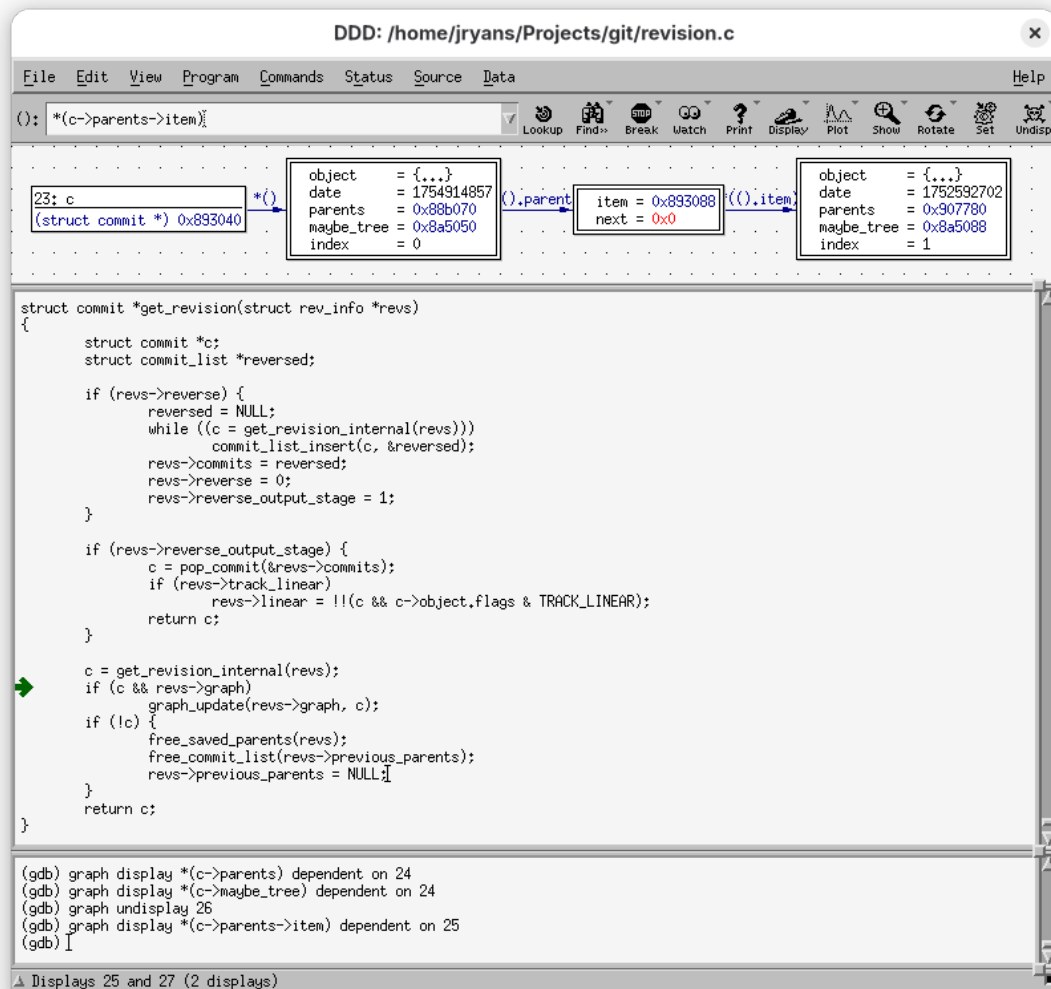


Fig. 1.10: Graphical debugger interface DDD [GNU Project 1995] enabled by debug info visualises connections between program data structures.

debuggers should function as expected, and the checking techniques described here can help by checking debug info reliability.

1.3.2 Offering users more abilities

Reliable program metadata has the potential to enable not only common dynamic program analysis tools like interactive debuggers, but also a whole spectrum of software inspection and modification abilities.

Visualisability Metadata makes it easier to construct program visualisations to aid in code navigation and archeology. Tools such as DDD [GNU Project 1995] make use of debug info to reveal connections between data structures as seen in Fig. 1.10.

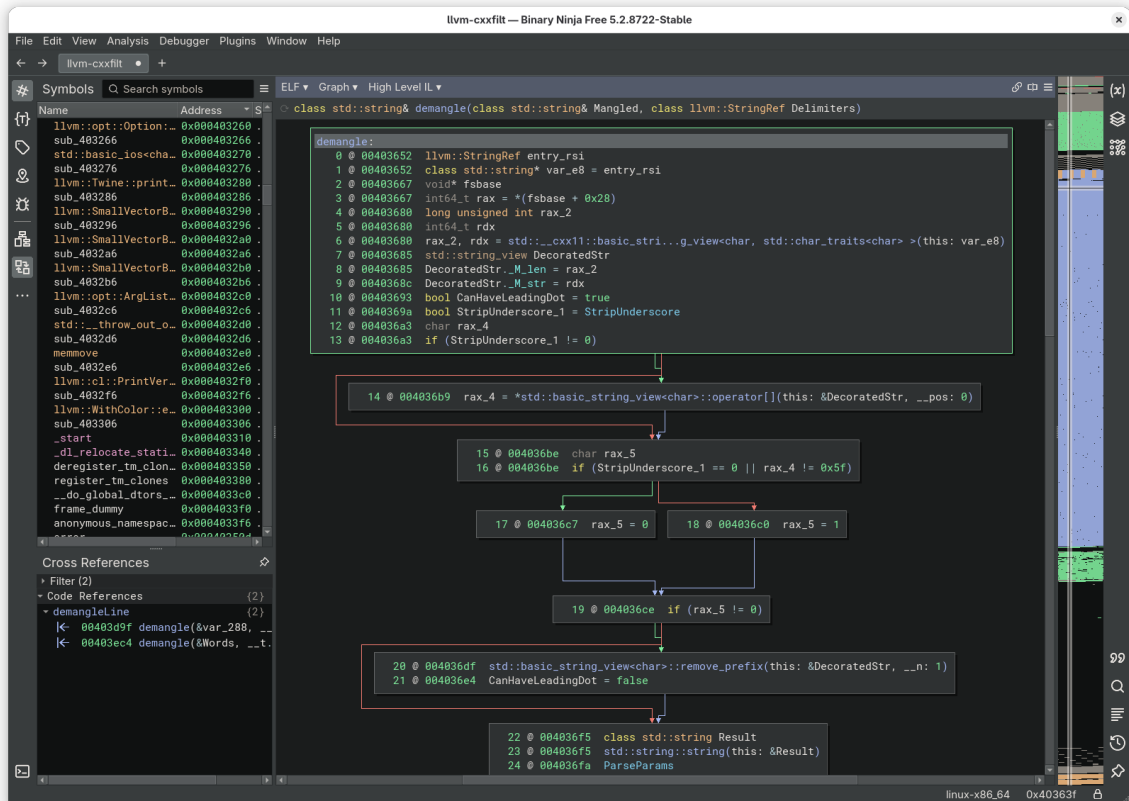


Fig. 1.11: Reverse engineering tool Binary Ninja [Vector 35 2015] reveals connections between binary code blocks. Visualisations are enhanced with details like types and variable names when debug info is available. In this figure, source-level types and variable names from debug info can be seen in the blocks that make up the graph view.

Reverse engineering tools such as Binary Ninja [Vector 35 2015] shown in Fig. 1.11 analyse compiled binaries and attempt to discover and reconstruct connections between blocks and functions. Program metadata can simplify analyses and enhance visualisations with various source-level details (e.g. types, variable names) that are typically omitted from binaries.

Explainability Tracing-based tools such as SystemTap [Eigler 2006] help users understand function calls and other program behaviour over time. Debug info assists these program traces by layering on source info like function names and source coordinates.

More experimental program investigation tools such as Whyline [Ko and Myers 2008] allow users to navigate and debug software across time (via an IO trace captured during execution) by asking questions like “why did color become blue?”. While the original prototype makes use of a VM debugging API in the Java ecosystem, debug info could enable similar techniques for compiled programs.

Live editability Debug info enables more than just observation: it can also be used to assist with modifying program behaviour. Some debuggers such as the one in Visual Studio [Microsoft

[2025b](#)] support “edit and continue” features for compiled languages like C and C++ that allow live code changes to existing functions in running programs. Debug metadata is used to find functions in the program and patch them while ensuring execution flows to the new version of each modified function.

Malleability In today’s computing environment, software developers build for imagined user needs, but in reality, each user has slightly different requirements and preferences. There is often a long tail of additional features and customisations desired by some subset of a program’s user base. For example, imagine there was a biologist who wanted to add a custom DNA widget to their data analysis notebook. If the notebook is a closed-source data analysis tool, they may simply be out of luck. Even if the source is available for reuse, they would need to recruit one or more software engineers to add their desired features as well as maintain them long-term if those changes are not accepted by the notebook’s upstream developers. *Malleability* explores techniques that would enable the biologist to construct a feature extension themselves (perhaps similar to local browser add-ons) implementing this functionality.

Software tooling is perhaps the most obvious consumer of debug info, but the potential applications for reliable debug info and other program metadata go far beyond such tools. It can also support software malleability which seeks to empower *domain experts* (non-programmers who are experts in some other field) to control and modify their digital workflows to better suit their needs as opposed to waiting (perhaps indefinitely) for an upstream vendor to consider adding another optional long-tail feature [[Kell 2009](#); [Basman and Tchernavskij 2018](#); [Beaudouin-Lafon 2019](#); [Tchernavskij 2019](#); [Stinnett and contributors 2019](#)].

Detailed program metadata that could be used by tools like `liballocs` [[Kell 2015](#)] for such an approach already exists: debug info produced by compilers. Unfortunately, this metadata is currently too unreliable to drive the low-level program modifications that malleability may need, such as extracting a feature from one codebase and wiring it up to another (in a semi-automated fashion without support from software experts). Currently, more basic uses of debug info, like breakpoints or tracing, are not yet 100% reliable. If we ever hope to use that same supporting infrastructure to guide complex modification of binary artifacts, we will need to have truly reliable program metadata.

Metaprogramming techniques may also be useful stepping stones towards malleability. In particular, reflection and aspect-oriented programming [[Kiczales et al. 1997](#); [Tanter 2004](#); [Wallis 2024](#)] show promise by enabling additional functionality to be knitted together with existing software, though these techniques are largely aimed at software developers rather than users.

Reliable debug info may assist in implementing these approaches by e.g. describing functions, variables, and call sites in compiled program binaries.

1.4 Research questions

We state here the primary set of research questions investigated by the work in this dissertation. Additional questions arise in subsequent chapters as we explore each debug info checking approach in more detail.

What are viable approaches to check *completeness* of debug info? There are surely various ratios of *covered* over *coverable* items we could measure to examine completeness. How do we define these terms precisely? What would such ratios mean at a different program points and across control flow? Do the definitions we make hold up in the face of common compiler optimisations?

How might we check the *correctness* of debug info? Although compiler testing in general has a long history in the literature, it has typically focused on behavioural correctness of the output program. There is very little prior work that explicitly tests compiler-emitted program metadata such as debug info. Which testing techniques could be adapted to check this additional compiler output?

What obstacles prevent achieving the ideal debugging illusion? Are there *internal* architectural challenges within popular compilers that add complexity to achieving ideal debug info? Do the *external* debug info format specifications imply any expressiveness limits that prevent compilers from fully describing program transformations?

1.5 Thesis

The thesis of this dissertation is:

The *completeness* and *correctness* of debug info generated by compilers can be measured against significantly stronger properties than have been demonstrated previously. These stronger properties can be realised through the application of testing approaches that use the connection between an optimised program’s source-level and object-level state spaces. The issues discovered via these techniques enable compiler authors to improve the generation of program metadata.

The contributions below sketch my pathway toward these goals.

1.6 Contributions

The major contributions of this dissertation include:

- a model of optimisation while also generating debug info as *residualising* computation ([Chapter 2](#))

- a debugging-centric analysis of local variables and their *lifecycle* within a program (Chapter 3)
- a discussion of a series of candidate *local variable coverage* metrics, culminating in an implemented coverage tool which improves on earlier metrics by obtaining an accurate and achievable “complete coverage” baseline (Chapter 3)
- a consistency checking approach that finds recurring *function call metadata* divergences, by constructing a dynamic analysis tool that can collect source-level *dynamic call trees* of real executions (Chapter 4)
- a differential checking technique examining *local variable correctness*, founded on a notion of illusion consistency (Chapter 5)

Each of these is expanded upon in the following chapters and supported by evaluations which test these methods on popular software packages. For each testing approach, I implemented them as usable tools, applied them to real compilers and codebases, and reported the resulting problems found by those tools to compiler authors as actionable issues.

1.7 Publications

Ideas from the following publication can be found in Chapter 2 and also throughout this thesis:

- Stephen Kell and J. Ryan Stinnett. 2024. Source-level debugging of compiler-optimised code: ill-posed, but not impossible. In *Proc. of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '24)*. <https://doi.org/10.1145/3689492.3690047>

A large portion of Chapter 3 has been adapted from the following publication:

- J. Ryan Stinnett and Stephen Kell. 2024. Accurate coverage metrics for compiler-generated debugging information. In *Proc. of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24)*. <https://doi.org/10.1145/3640537.3641578>

A large portion of Chapter 4 has been adapted from the following publication:

- J. Ryan Stinnett and Stephen Kell. 2026. Debugging debugging information using dynamic call trees. *Proc. ACM Program. Lang.* 10, OOPSLA1 (April 2026), 105:1–105:27. <https://doi.org/10.1145/3798213>

Chapter 2

Defining the debugging illusion

Software tools like debuggers and profilers help developers to understand incorrect programs by attempting to provide the “illusion” of inspecting and controlling a running *source* program, even when what runs is in fact a transformed program the language implementation has produced for the target machine. A complex interplay takes place between these tools and the language implementation in the process of assisting developers as they diagnose and resolve software issues. In this chapter, we explore the nature of this debugging illusion, discuss different forms the illusion may take, and describe several completeness and correctness properties of the illusion which we aim to check through approaches we detail in later chapters.

2.1 Truthful vs. expected

Prior work [[Zellweger 1984](#)] has identified two categories of *illusion*: **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” illusions are characteristic of compiler toolchains using debug info formats such as DWARF [[2017](#)].

An illustration of this is how Hotspot’s feature called “full speed debugging” [[Sun Microsystems 2001](#)] really means “slowed down, but only when the debugger is attached”, using dynamic deoptimisation. Since debugging users never actually observe execution of the optimised code, this is not a “truthful” approach. In LLVM, by contrast, it’s considered a bug if compiling with debug info enabled (`-g`) changes what code is generated [[Wang et al. 2023a](#)]. In this way, full-speed debuggable execution really is feasible, at a cost of often truthfully showing the unexpected.

Both “expected” and “truthful” approaches have merit. To approximate a “best of both” solution, previous efforts by some truthful approaches have attempted to move more closely

towards expected semantics, often by introducing more detailed metadata—such as allowing the debugger to split one machine-program state into many virtual states that can be stepped through separately, each presenting different source-level “location views” [Oliva 2017b]. Meanwhile, expected approaches often seek to more closely approximate the level of optimisation allowed by the truthful approach by inhibiting optimisation only “on demand” and “just enough” [Van De Vanter et al. 2018].

There are, however, fundamental limits to these trajectories. The “expected” approach is necessarily more constraining of optimisation because *at least* while execution is under observation, it must appear to preserve a simulation of the source program, *at least* up to the queries requested by the user (e.g. breakpoints, watchpoints, etc). The truthful approach, on the other hand, has by definition not enforced any such constraint during optimisation. This suggests an end-point where *both* illusions are available and explicitly distinguished: for example, a truthful implementation could offer expected semantics *additionally* by maintaining the code and program state necessary for modelling the less optimised execution separately somehow (i.e. code and state which would otherwise be discarded); the “location views” technique is a limited form of this. An expected debugging illusion could optionally be stepped down to truthful, analogous to how source-level debuggers may permit users to step down to an assembly-level view. Current debugging technologies are, however, far from this point: I know of no fully expected system that can also map real stack or register contents to their source-level views, for example, and no truthful system that can also fully simulate the unoptimised program if requested.

To my knowledge, the “expected” property is not offered by any optimising ahead-of-time compiler toolchain using debug info formats such as DWARF [2017]. Rather, it appears only in interpreters—either simple interpreters, or in sophisticated language virtual machines’ debug servers that use dynamic deoptimisation [Hölzle et al. 1992] to switch to an interpreter¹ on demand. In both cases, the debugging infrastructure never observes optimised code, unlike toolchains using DWARF or the like. We focus on the toolchain scenario in this dissertation.

2.2 State mapping: two approaches

Different systems and language implementations have approached debugging differently. Any approach must address two main aspects: a *state-mapping problem* (mapping some or all object states upwards) and constructing a *debug-time view* (what execution is observed to occur, in what terms?). Sophisticated language virtual machines (VMs) employ dynamic deoptimisation to avoid the full generality of the state-mapping problem and peg the debug-time view to an unoptimised source-level execution—by actually switching the VM to such a mode. This appears to confirm the tension that what is observable is not what is optimised. Meanwhile,

¹This extends to template-based just-in-time compilers, which, like interpreters, also preserve a direct simulation of the source program.

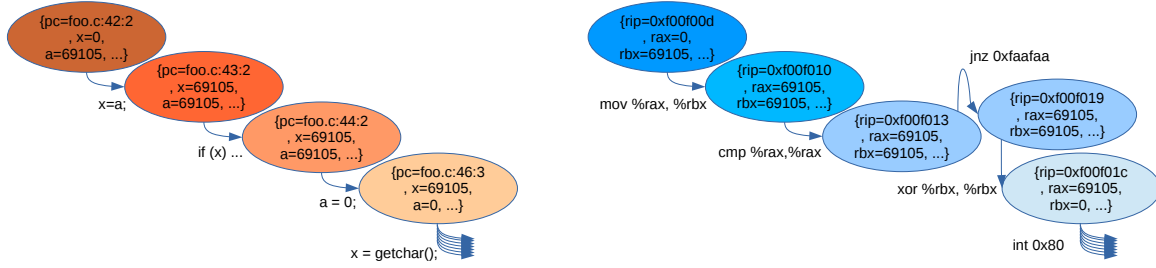


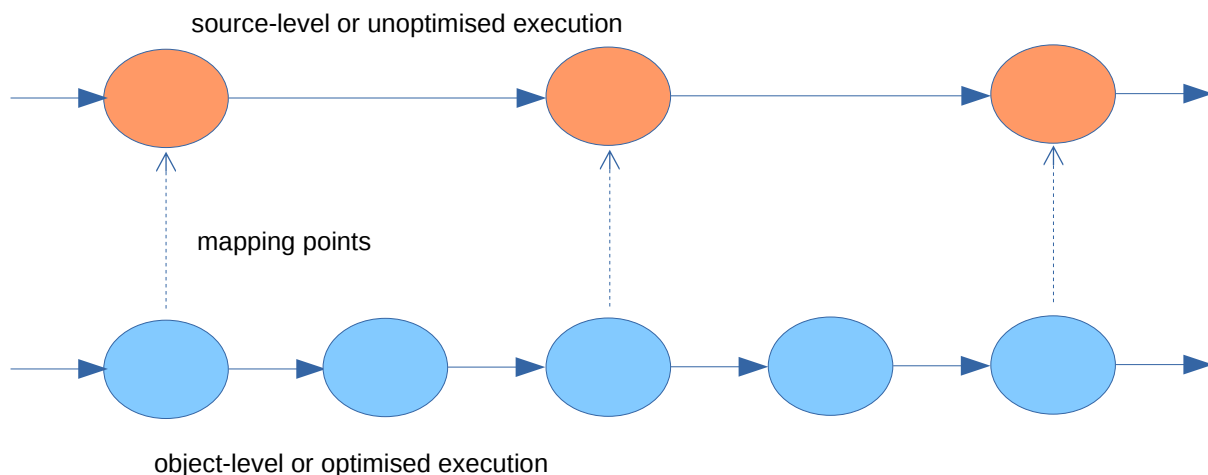
Fig. 2.1: A piece of the concrete transition system (left) for a simple source program (to appear in Fig. 2.2), and the corresponding piece of the object program’s transition system (right). *State mapping* means capturing the correspondence between these states. Here, this includes that `a` is in `rbx`, `x` is in `rax`, and that `pc` maps to `rip` as shown by the shades of each bubble.

ahead-of-time compiler toolchains instead appear to take on the complete state-mapping problem, generating extensive metadata which promises to map any object state back to source, but keeping the debug-time view reflecting a perhaps heavily optimised execution. The catch is that in practice the mapping is “best-effort”: it is invariably incomplete (e.g. some states are not mappable) and sometimes incorrect. The view, despite its best-effort mapping to source terms, still reflects the actions and sequencing of the optimised object code—sometimes enlightening, but more often confusing. These issues arise to a degree correlated with the level of optimisation, again appearing to affirm the tension.

2.2.1 The transition system model

We can view both source and object program as transition systems where each state is labelled with the value of all program variables, including the program counter and (although not shown in these diagrams) stack and heap. Each state space is then, roughly, a Cartesian product of these many components. Fig. 2.1 shows a simple example. State mapping in this case is almost one-to-one, but not quite: the object-level diagram has five states instead of four, because two instructions are required to realise the conditional branch of a single source-level `if` construct. Note also that as shown, each system is concrete: the distinct bubbles should not be seen as nodes in a control-flow graph, but rather as fully concrete program states, or collections of bits. The only branching is therefore on nondeterminism: this is a `getchar()` operation in the example, but could also include other input actions (or, in a concurrent program, scheduling decisions).

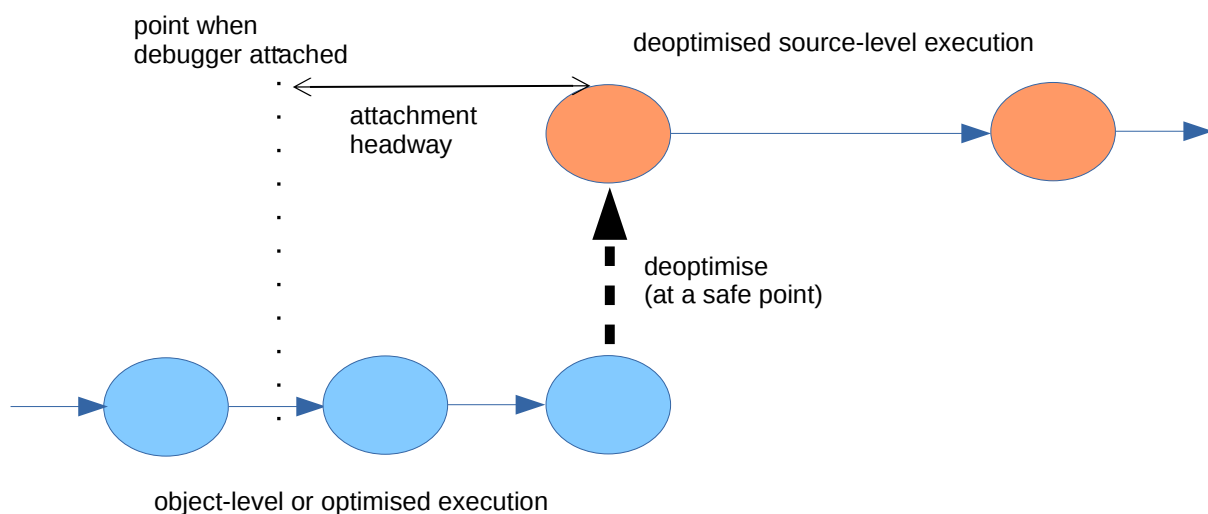
Since the target (object) machine is lower-level than the source (abstract) machine, the state spaces are not in bijection. There may be *mapping points* where the object program’s state corresponds precisely to a source-level state.



2.2.2 Approach 1: dynamic deoptimisation

Dynamic deoptimisation [Hölzle et al. 1992] “solves” the problem of debugging optimised code by an abdication. Only unoptimised execution is ever observed! This allows a faithful observation of source semantics, while reducing the penalties to performance and convenience, relative to running the entire program unoptimised from the start. However, the observed execution is always the slow, unoptimised one.

The design can be seen as limiting the state mapping problems that the implementation must solve. Only so many optimised-program states must be mappable to unoptimised ones—enough to meet a user’s *timeliness* expectation. A debugger might not attach immediately, but rather at the next “safe point”, such as a call or loop back edge—more than frequent enough in practice. Therefore, under such a scheme, only call sites and back edges need be mappable. The mapping is kept in private compiler-generated metadata, usually also used by the garbage collector.



The *density* of these mapping points defines the granularity of simulation that must exist between optimised program and source program. Here it is clearly denser (finer-grained) than

...	(dbg) break foo.c:43	(dbg) break foo.c:43
42: x = a;	(dbg) run	(dbg) run
43: if (x) {	Hit breakpoint 1	Hit breakpoint 1
44: a = 0;	43: if (x) {	43: if (x) {
45: x = getchar();	(dbg) print x	(dbg) print x
...	69105	<variable optimised out>

Fig. 2.2: Source listing and two possible debugging sessions for different compilations of the program. In the right-hand session, variable `x` is not available. The debugger assumes this is because it has been optimised out, but a more likely explanation is that the compiler incorrectly omitted to describe `x` in the debug info.

TAG_compile_unit AT_producer : GNU C17 10.2.1 AT_language : ANSI C99 AT_name : foo.c AT_comp_dir : /src/libfoo TAG_subprogram AT_name : f AT_type : <9> (int) AT_low_pc : 0xf00eef0 AT_high_pc : 0xf00f0fc TAG_formal_parameter AT_name : a AT_type : <9> (int)	(continued) AT_location : OP_reg3 (rbx) TAG_variable AT_name : x AT_type : <9> (int) AT_location : OP_reg0 (rax) 9: TAG_base_type AT_byte_size : 4 AT_encoding : signed AT_name : int TAG_subprogram AT_name : getchar	File name line start addr stmt? ----- ... foo.c 42 0xf00f00d y foo.c 43 0xf00f010 y foo.c 44 0xf00f019 y foo.c 45 0xf00f01c y ...
---	---	---

Fig. 2.3: DWARF value (“info”) information (left, middle) and line table (right) for a plausible compilation of the code in Fig. 2.2, encoding the correspondence between the two sides of Fig. 2.1. Inspecting local variables relies on correct and complete variable records including location expressions (which can instead compute a variable’s *value* if it is not stored at any location), and also on `low_pc` and `high_pc` (or other equivalent forms) to find the relevant record. The `OP_` tokens denote instructions in expressions for a stack machine, although the expressions here simply opaquely name a specific machine register.

the input–output relation, but not so fine-grained as to offer a mapping at every instruction.

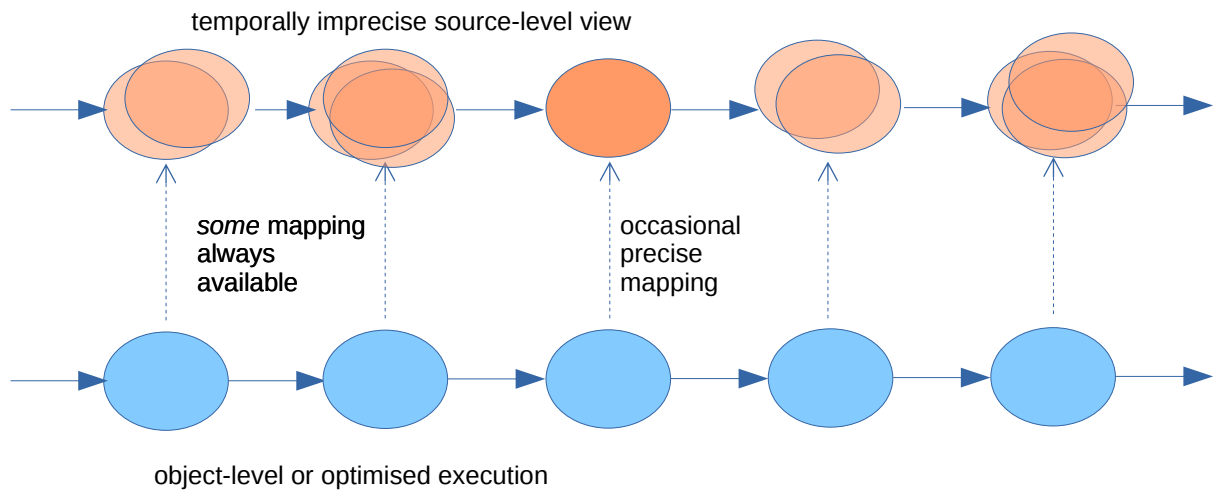
Deoptimisation is specialised to interactive debugging, which occurs at human speed. It is less applicable to tracing, and not at all suitable for profiling, where retaining optimised execution speed is desirable or essential. The contract between language implementation (compiler) and tools is therefore specialised to a class of tool: for example, on the Java platform the interface for interactive debugging (JDI) is separate from the bytecode instrumentation interface catering to profiling and tracing (JVM TI). None of these interfaces exposes the state mapping itself, even though the compiler must generate it—it exists as compiler-private metadata describing how to deoptimise at each safe point.

2.2.3 Approach 2: in-toolchain debug metadata

A contrasting approach is taken by ahead-of-time compilation toolchains in the tradition of Unix, C, C++, LLVM and so on. In principle, their compiler-generated debug metadata undertakes to map *any* binary state to a source state. A debugger can be attached at any time. There is no deoptimisation, so there are no safe points (and, not coincidentally, usually no garbage collector). Execution can proceed at full speed even with a debugger attached. However,

unlike with safepoints, the compiler is under no finer-grained simulation requirement than to preserve the input–output behaviour of the source program, i.e. the basic contract of correct compilation.

Metadata is public and standardised, in formats such as the venerable STABS [Menapace et al. 1992] or the more modern DWARF [2017] and CodeView [Microsoft 1995]. It is part of the contract between the tool and compiler, and the same metadata can be used by many tools. Operating system features, such as `ptrace()`, provide tool-specific foundations, while metadata remains tool-agnostic.



We observe that this mapping has two dimensions: it can vary in **density** (how many object states are mapped?) and also in **height** (what is the semantic distance between object and source? how complex is the mapping as a computation?). Density determines the granularity of source-level stepping, and also the attachment headway for deoptimisation. Height determines *what execution is actually being observed*. We have been thinking so far in terms of a single fixed height: we map back to “source level”. In reality, however, this is leaky: details of the object program are often betrayed “by accident”. At least one approach in the literature proposes to embrace this, providing a viewpoint short of the original source program, as a way to lessen mapping difficulties [Tice and Graham 1998]. This approach is described as “non-transparent”, meaning it does not attempt to create the illusion that the original source program is executing; we could see it as an intentionally “less tall” state mapping. The strength of the toolchain approach, relative to deoptimisation, is that it lets us observe the object execution, even at instruction granularity (its mapping appears *fully dense*), while at least attempting to view it at source level (i.e. it may aspire to a *fully tall* mapping, but often falls short). To make possible this tall mapping under complex optimisations, real metadata formats are Turing-complete (§2.3.1).

Despite the appearance of power, in practice the debug-time view is a shaky one: control flow “bounces” around lines of the optimised program, and observed values of variables need not coherently reflect any one source state. We say it is “temporally imprecise” (intentionally borrowing the word “precise” from hardware literature on exceptions). The problems are

not limited to temporal precision, however: in practice local variables are often missing or simply incorrect (explored further in [Chapter 3](#)). We can talk about a mapping’s **fidelity**, encompassing all issues of correctness and precision (when fixing height and density), with **temporal precision** as one important sub-dimension.

The claim of a fundamental tension is often used as mitigation for a lack of fidelity. Metadata is of unwieldy design, comes with no strong completeness or correctness criterion, and must be transformed by the optimiser just as it transforms the code. Therefore, for compiler authors, generating it is an onerous task, and corners are often cut.

The toolchain approach, although seemingly more powerful, is also more complex and more confounding than dynamic deoptimisation. What does it mean for this metadata to be correct? Can a “temporally imprecise” mapping nevertheless be called correct? How can its recurring infidelities be squared with the apparently huge possibilities of Turing-complete metadata? We explore this in the forthcoming sections, where “debug info” will refer to metadata in the ahead-of-time tradition—typified by the DWARF [\[2017\]](#) format, the most expressive in widespread use.

2.3 Debug info as residual computation

As we explore below, the DWARF [\[2017\]](#) debug info format makes use of Turing-complete “residual” computation abilities. When programs are transformed during optimisation, compilers can describe those changes in the emitted debug info and enable program states that were optimised away to be recovered by debuggers and other tools.

2.3.1 Debug info concepts

In the previous chapter, we briefly touched on the different kinds of metadata in debug info formats ([§1.1.3](#)). In this chapter, we focus primarily on *source coordinate* and *variable location* information. Both encode *mappings* between states of the source program and states of the object program. Strictly speaking, we should say “components of states”, recalling the “Cartesian product” from earlier ([§2.2.1](#)).

Source coordinate information addresses the program counter (PC) component: it maps program counter values to source coordinates (file, line, and sometimes columns or ranges thereof) and back, and enables display of source code, setting breakpoints, and so on.

Variable location information addresses the local and global variable components: it includes where variables are located, and how the bits and bytes encode its source value. Together, these enable debug-time inspection of program values: both directly in named variables, and indirectly by following pointers or references (e.g. into the heap). Metadata describes both data and code (“subprograms” in DWARF), including which lexical scopes are active at a given program counter, and where a called function’s body has been inlined. [Fig. 2.3](#) shows a very

```

1 int f(int *data, void *arg)
2 {
3   int i = 0, tmp, out1 = 0, out2 = 0;
4
5   i = tmp = get_start(arg);
6   for (; i < MAX; ++i)
7   {
8     out1 ^= data[i];
9   }
10
11  for (i = tmp; i < MAX; ++i)
12  {
13    out2 &= data[i];
14  }
15  g(out1, out2);
16  return tmp;
17}

```

local	location	range
data	in r1	at all points
arg	in r2	at all points
i	in r3	from 3
tmp	in r4	from 5
out1	in r5	from 3
out2	in r6	from 3

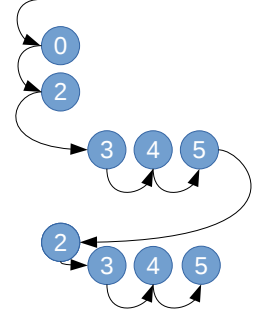


Fig. 2.4: Left: a simple example function in C. Middle: simple debug info for a relatively unoptimised compilation on a fictional architecture; we assume locals are allocated to registers which are plentiful. Right: the *source-level* concrete progression of variable *i*, for an execution where *MAX* is 5 and *get_start()* returns 2.

simple example in DWARF based on the code in Fig. 2.2.

DWARF-generating compilers consider it a bug if compiling with debug info enabled (`-g`) changes the generated code [Wang et al. 2023a]. In other words, enabling debugging does not curtail optimisation in any way.

In totality, the debug info maps states to states, as we have been conceptualising in terms of transition systems. However, each record of source coordinate or variable location information addresses a component of the state space, and therefore relates sets of many states at a time (e.g. “when *rip* is between `0xf00f010` and `0xf00f01c`, source variable *x* has the value in register *rax*” or “... execution is on line 42”). This “PC-keyed” factoring is one of many complexities of debug info formats such as DWARF, arising from a mixture of necessity and accident. To illustrate necessity: suppose variable *x* moves between registers and the stack. Debug info must describe, at each instruction, where such a “wandering” variable resides—no longer trivially mapped to a single register for the variable’s lifetime, but instead requiring a case split over the program counter. Similarly, consider some variable that is no longer modelled directly in the object program but can be computed from other program state, such as the index variable in a statically bounded loop that has been unrolled into a straight-line segment in the binary code. In the unrolled loop, the loop index will be implied by the program counter. The debug info may describe how to *recover* the index at each instruction.

Debug info is therefore a *computational artifact*: it encodes functions of the object program state. To do this, DWARF embeds a Turing-complete stack machine language [Oakley 2011]. This stack machine includes operations for arithmetic and conditional branching, reading and writing its own stack storage, as well as with read-only access to various kinds of program state (memory, registers). Debuggers and other debug-info-powered tools evaluate these expressions to recover source variable locations and values from remaining program state. As these DWARF programs cannot directly modify program state, they are akin to a Turing machine that burns its storage tape upon completion. These features give DWARF greater expressivity compared

to other debug info formats like CodeView [Microsoft 1995], which are far more restricted, supporting only a predefined set of options for variable locations, instead of utilising residual computation.

This computational power allows debug info to bridge the sometimes-large gap between what the object program internally *does compute* and what the source program *would compute*, perhaps encoding arbitrary pure functions. For example, consider a variable `err` that is frequently updated, by `err = f(state)`, where `f` denotes some complex expression (or inlineable pure function). If `state` is core to the program logic but `err` is referenced only in assertions, then when assertions are macro-deleted in a given build of the software, *both* the variable `err` *and* the computation `f(state)` might be eliminated from the optimised program. However, both could still be described in the debug info, including a residualised stack-machine representation of `f`.²

As compilers perform transformations at the code level, they generate functions *at the state level* which “undo” those effects. Note the difference in level: this reflects the difference between compile time (code), when transformation is performed, and run time (state) when debugging occurs. The computation performed by debug info is not simply the inverse transformation on code. These state-level computations are onerous for compiler developers to generate, hard to test, and their correctness often not prioritised. Unsurprisingly they are frequently incomplete and incorrect [Assaiante et al. 2023]. Previous work has characterised debug info as capturing a *recovery function* [Zurawski 1990]. In the next section we will review these, and ask more generally: what exactly is debug info is doing? What are its limitations, and what does it mean for it to be correct? We will develop this *state-mapping* framing further in order to answer these questions.

2.3.2 Residual computation at work

To illustrate state mapping and introduce the idea of *residual computation*, in this section we walk through some example optimisations on a contrived piece of code, shown in Fig. 2.4. An unoptimised compilation would result in a straightforward mapping from object to source states, which is shown in the middle of the figure. On the right, the valuations of variable `i` are shown for a particular concrete execution, where `MAX` is 5 and `get_start()` returns 2. In what follows, we will consider the effects on the variable `i` of various optimisations to this code, examining the same concrete execution each time.

Whereas classical compiler texts view optimisations as lossily reducing or eliminating computation, with debug info we argue it is better to see optimisations as losslessly “*residualising*” computation. Rather than eliminating computation, it is “moved into” debug info. This is illustrated in Fig. 2.5 with a simple optimisation to remove the redundant zeroing of `i` on line

²As suggested earlier (§1.3.1), if a program contains undefined behaviour (UB), both the run-time program and debug-time metadata produced by the compiler cannot be trusted. We can no longer make any claims that e.g. debug-time residual computation recovers source program attributes when UB is present.

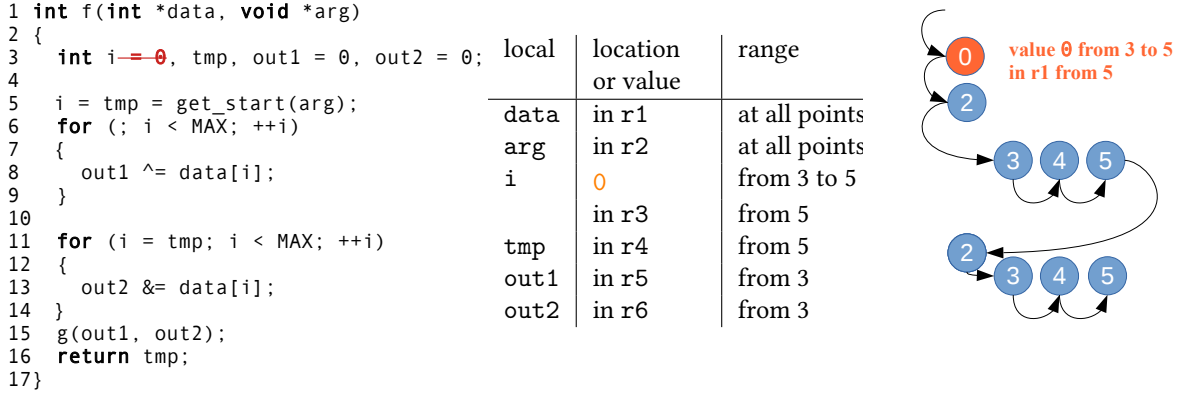


Fig. 2.5: Left: a simple “zeroing elimination” optimisation on the code of Fig. 2.4. Middle: how debug info can *residualise* the zeroing. Right: the *source-level* concrete view of variable *i* recoverable at debug time using residual computation, for the same execution as Fig. 2.4. Orange denotes residual artifacts.

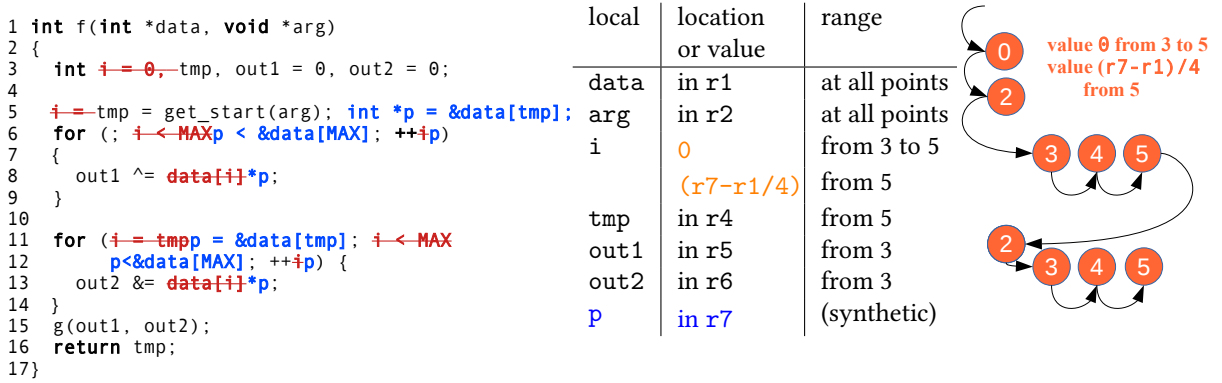


Fig. 2.6: Left: a more complex optimisation (strength reduction) on the code of Fig. 2.5. Middle: how debug info can residualise the strength reduction, resurrecting *i* which is no longer directly present in the object program. Right: the *source-level* concrete view of variable *i* again recoverable at debug time using residual computation. The synthetic temporary *p* is assumed to reside in register *r7*. Orange denotes residual artifacts.

3. While in the object code, the zeroing would appear to be removed (typically saving one instruction), we see it is not removed but rather moved into the debug info, which now records the fact that *i* is zero at line 3.

Similarly, while this initial assignment of zero to *i* would be called “dead” in classic compiler terms, in debugging we must beware two conflicting meanings of “dead”. “Dead code”, meaning *unreachable* code, can safely be eliminated with no loss of debuggability. In contrast, “dead stores” typically *are* reachable, just will not be read *by the program*. Since they might still be read *by the debugger*, these should be residualised into debug info rather than eliminated altogether. Again, this motivates how this zeroing is recorded in the debug info even though *i* is never zeroed in object code, since a user might nevertheless expect to see *i* initially equal to zero in the debugger.

Next, in Fig. 2.6, we consider a more expansive optimisation, strength-reducing the array indexing in the two loops by replacing the loop induction variables by a pointer. *Strength*

reduction refers to elimination of an expensive operation such as multiplication in an array-indexing calculation, with one or more cheaper operations such as addition. In this case, the indexing is replaced a pointer increment at each loop iteration, avoiding the need for a fresh index calculation each time. As before, in a debugger the developer could still observe the concrete execution seen on the right, with variable *i* appearing to progress through its source-level valuations, even though *i* is no longer directly represented in the object code. The compiler-generated temporary *p* is assumed to reside in register *r7*, and *i* can be computed as a function of this register and data (still in *r1*).

2.3.3 The essential problem: loss of state

These examples so far have been rather convenient in a few ways. Firstly, the control structure has been unchanged: there is an injective mapping from distinct points in the source program to distinct points in the object program. (We have identified source points only by line numbers, but this could be refined into terms of columns ranges, perhaps following the statement boundaries or sequence points of the source language.) Secondly, not only are the control states injectively mappable, but execution order in the object program is the same (after mapping) as the order in the source program. Thirdly, residual computations have been stateless pure functions: all the inputs they needed have been available in the current object program state.

Not all optimisations are so convenient: those in the categories of *code merging*, *code motion* and/or (what we will call) *state dropping* clearly do not satisfy these properties. These categories are inter-related: code merging can be seen as dropping control state, and code motion as refactoring the control state, i.e. dropping *some* control states to replace them with others. Of course, non-control state is commonly also dropped, such as when local variables are found to be “dead”: during register allocation, they are discarded from the program’s run-time state (as with the *err* example in §2.3.1).

Ostensibly dropped state might be *recoverable*. Consider an optimisation that merges identical tails of two basic blocks:

<pre> if (cond) { ... ++x; } else { ... ++x; } </pre>	... to become:	<pre> if (cond) { ... } else { ... } ++x; </pre>
---	----------------	--

The mapping of control positions is non-injective, since multiple points in the source program are realised by *the same* point in the object program. However, this could be disambiguated as long as *cond* is still available or computable. The framing by Zurawski [1990] of debug info as *recovery functions* captures exactly this intuition: a function over object program state can recover the source-level view—although only if there are sufficient redundancies in

the object program’s state space.

Since version 3 of DWARF in 2005, eliminated local variables can be recovered in this way. Control state, such as the relevant line number in this example, still cannot: it would require the line number table to be indexable not only by the program counter, but by other state that can tell us *where we came from*, in this case `cond`. DWARF’s current line table happens not to be so indexable.³ (This recalls the earlier description of DWARF as “PC-keyed”: the program counter occupies a special role.)

Unfortunately, `cond` might not always be available. This scenario is not limited to non-injective control mapping situations; it could also apply in a store elimination similar to the earlier `i = 0` example. Suppose:

21: <code>t = f();</code>		21: <code>t = f();</code>
22: <code>int i = t + 1;</code>	<code>... is optimised to:</code>	22: <code>int i;</code>
23: <code>for (i = x; ...)</code>		23: <code>for (i = x; ...)</code>

... and we would like to residualise `i` at line 22 as the computation `t+1`. What if `t` itself is not available? The transformation has potentially shortened the live range of `t`, since it is no longer used on line 22, meaning the compiler may attempt to drop its state.

Here, again, our definition of “available” does not match that of standard compiler texts. For us, a value is available either if it is stored in the program state, *or* if it is residually computable i.e. computable as some function of (recursively) available values. Constructing the residual computation for `i`, we are in effect elaborating the backward slice [Tip 1995] of `t+1`. We can stop adding to the backward slice when we hit values still stored in object program state, which are trivially available. but we are in trouble if we hit a value that is truly unavailable. These unavailable values consist of that subset of the *nondeterministic input* to the program that is no longer retained, either directly or inferrably, in object program state. Although the debugger can re-do computations, it cannot generally re-perform input actions of the debugged program. This is why Turing-complete metadata (§2.3.1) is not by itself enough to counter arbitrary compile-time transformations.

The compiler could conservatively assume the input is lost and so omit any residualisation, curtailing debuggability. Or, since it is in charge of generating code, it could refrain from discarding the input-dependent state at all, curtailing optimisation. This is the classic dilemma faced by even the most diligent compiler pass author.

2.3.4 Recovery functions vs. residual computation

State-dropping optimisations, of which code motion is one kind, have been viewed as “unavoidably” curtailing debuggability owing to an apparent lack of any recovery function. In an article

³Although we have emphasised the Turing-complete nature of DWARF (§2.3.1), that power is largely restricted to recovery of variable locations and values in current versions, and even then it cannot store its own state between uses. Control state (e.g. the line table’s PC to source coordinate mapping) is currently recorded using simpler machinery, with its abilities restricted even further.

that pithily summarises many aspects of the problem, [Brender et al. \[1998\]](#) wrote as follows.

Unfortunately, code-motion-related optimisations generally lack recovery functions and so must be foregone [sic].

But under what assumptions *do* these optimisations truly lack a means of recovery? Since state-dropping seems to be the central issue, what prevents us simply from *not* dropping the state, at least when a debugger is attached?

The short answer is that the assumption to date has been one where debug info retains no state. The framing of “recovery functions” conveys this implicitly: from some place where the base (non-residual) program is stopped, such a function may inspect program state as, input and produce (hopefully) a meaningful source-level output. Then, control passes back to the base program or, if the debugger user chooses, to another recovery function answering a separate query. Each recovery function is a stateless computation that is composed sequentially with elements of the base program.

The framing of *residual computation* instead of recovery functions, intends a contrast: the debug-time computation holds state, and therefore can be seen to run continuously *in parallel* with the base program. It is, in effect, a process consuming the same input as the base program, and frequently synchronising with it—a frequency determined by the density of mapping. Since compilers must preserve input/output behaviour, it seems feasible to require that all input action points be mappable. In between input actions, the computation has all the information it needs to construct an arbitrary debug-time view, including (but not limited to) one exhibiting the naïve source semantics.

2.3.5 Resolving the optimisation vs. debuggability dilemma

With a residual computation approach, there is no longer a trade-off between optimisation and debuggability. In extremis, *both* fully optimised and fully unoptimised versions of the code might run in parallel—the latter being the maximal residual computation, executed in the debugger, and the former being the “real” object program, executing on the target. Assuming this residual computation need only run “on demand”, when debugging operations are performed, the trade-off disappears: we can have *both* a real full-speed execution *and* a source-level view. In its place is left a *timeliness* concern, i.e. with how much headway the debugger must have been attached: a trade-off that is familiar from dynamic deoptimisation. Since input actions define the minimum *synchronisation* between these parallel computations, we expect the worst-case headway to be the time to the program’s next input action, but opportunistically there are likely to be many further synchronisation points.

There is also an efficiency concern, not only in terms of resources for the parallel computation but more significantly to provide communication. Conceptually, all input must still be

received via the object program, and communicated downstream to the debugger. Virtualisation-based approaches to dynamic analysis have encountered and addressed closely related challenges [Chow et al. 2008], and there is a literature on faster approaches to communication than the standard `ptrace()`-based trap approach [Kessler 1990].

How does this compare to dynamic deoptimisation? It is similar, in that the unoptimised view is restored by additional computation, activated on demand. However observation of just the base-level (object) computation remains possible, and the residual code “sits atop” the base code (e.g. may access its state) rather than replacing it. Residual computation is done only to the extent needed: where little optimisation was done, little residual code or state are needed. There need not be a just-in-time compiler or on-stack replacement engine, but the necessary debug info will be no less detailed than the metadata used by such systems, detailing the stack and register contents. The residual computation conceptually exists in the debugger, not inlined into the deoptimised program code. This separation may be helpful, but may add communication overhead. The essential outcome is that there is the option of not only executing a fully optimised, object-program execution but also observing that execution, where that degree of observation suffices, *or*, on demand, restoring the full source-level semantics. Both can even be available at once, rather than requiring a binary choice. The result is rather like the difference between deoptimisation and splitting fast from slow paths at compile time: under the latter approach both paths exist at once, and may even share code, but the slower path is used only when needed; for us it also executes “residually” in the debugger.

2.3.6 Optimisations that alter program semantics

Some optimisations, such as those in the “fast math” category for floating point calculations, imply altered program semantics and may produce different results when comparing the unoptimised and optimised programs. These semantics deviations during optimisation pose an additional challenge when emitting debug info, but all is not lost. This is, after all, another variant of the choice between truthful and expected debugging illusions, which we have discussed throughout this chapter.

By enabling these compilation modes that alter program semantics, the user is effectively stating that computational results are allowed to vary (within some envelope) in return for (hopefully) improved run-time performance. Similar to the previous section’s discussion of offering both full-speed execution and a source-level view, residualised computation allows both semantic variations to be described. This allows the user to choose at debug time whether they wish to closely follow the optimised program with its altered semantics and results, or if they would rather have the debugger recover the source-level program semantics, effectively undoing these optimisations.

2.4 Sub-properties of “expected”

In the last few sections, we have explored several key aspects of debug info formats. The residual computation over program state enabled by debug metadata (§2.3) allows software tools to present a source-level debugging illusion to the developer and will be referenced throughout the rest of this thesis. With the truthful vs. expected illusion framing in mind (§2.1), stateful residual computation can in principle enable the *expected* illusion. We identify the following practical, testable properties that are *necessary* (but not sufficient) for the expected illusion.

Control coverage It is possible to stop at all reachable points in the source program’s control flow. Put differently, if code actually executes, then it also appears to execute as observed from the debugger.⁴

Control consistency For any given reachable program point in the source program, it is possible to stop there in all compilations of the program.

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

Variable semantic consistency When examining a variable at any reachable point, its observed value is consistent with the source program’s operational semantics, in multiple ways. These can be split into *cross-consistency* properties such as “happens-before” relationships (e.g. that events occurring earlier in program order have all happened, while all later events have not) and *internal consistency* properties for single variables, e.g. observed values are explainable by the source semantics, not intermediate states that are merely compilation artifacts.⁵

Semantic consistency properties are the essential point of divergence between “truthful” and “expected” illusions: firstly because “best-effort” toolchains’ optimisations are written to preserve only input/output behaviour and not any finer-grained simulation of the source program, and secondly because truthful debuggers intentionally expose intermediate states (e.g. during two instructions doing a multi-word load, allowing stepping in between the two instructions).

⁴In this dissertation, we assume an imperative source language, i.e. one with explicit control flow, although we believe this could be generalised—perhaps in terms of a partial order on active program constructs. While DWARF aims to support many source languages and programming paradigms, it is primarily emitted by compilers for imperative languages.

⁵There is a link here to the consistency notions found in the memory model literature. Although this problem is orthogonal to concurrency per se, debug-time observations can be viewed as additional operations occurring in parallel composition with the program. One relevant kind of “compilation artifact” is the “adjacent data” scenario described by Boehm [2005, §4.2].

We can say an illusion is *stronger* or *weaker* to the extent that these properties do or do not hold; full expected behaviour is the strongest illusion we consider. Not all optimisations weaken the debugging illusion. For example, if a section of code is provably unreachable, eliminating it does not weaken the illusion, because even during debugging, by definition none of the eliminated control-flow points could be stopped at.

We assume a starting point of compiler toolchains targeting a largely “truthful” approach, and we apply several testing approaches to check how close those toolchains are to the “expected” debug illusion in the remainder of this dissertation. Subsequent chapters explore different aspects of this illusion through practical tool-building and evaluate many of the properties just mentioned on real compilers and codebases:

- In [Chapter 3](#), we focus on the variable coverage aspect, i.e. the problem of *missing* local variable information
- In [Chapter 4](#), we examine how the function call / return subset of control consistency has been affected by features such as inlining and tail calls during optimisation
- In [Chapter 5](#), we describe a specific consistency property that isolates the problem of *incorrect* local variable information

Chapter 3

Coverage metrics for variable debug info

With the debugging illusion concept from [Chapter 2](#) as our guide, we turn our attention to the first testing approach. Many debugging tools rely on compiler-produced metadata to present a source-language view of program states, such as variable values and source line numbers. While this tends to work for unoptimised programs, current compilers often generate only partial debugging information in optimised programs. Current approaches for measuring the extent of coverage of local variables are based on crude assumptions (for example, assuming variables could cover their whole parent scope) and are not comparable from one compilation to another. In this chapter, we propose some new metrics, computable by tools we have implemented, which could serve as motivation for language implementations to improve debugging quality.

3.1 Introduction

Whereas compiler benchmarks provide a basis for evaluating the optimisation benefit achieved for a program, there is no equivalent way to measure (or show absence of) the incurred *debuggability disbenefit* in the generated “program plus debug info”. Certain crude metrics do exist but, as we will survey, they have various flaws and, most notably, are not comparable across compilers. One reason for this is that although missing debug info is usually regarded as a compiler bug, it is currently not clear what it means for debug info to be fully complete and correct. Historically, a “best effort” approach has prevailed, pursuing specific improvements [[Brender et al. 1998](#)] but placing no strong correctness criterion on compilers.

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

- a debugging-centric analysis of local variables and their *lifecycle* within a program ([§3.3](#));
- a discussion of a series of candidate *local variable coverage* metrics ([§3.4](#)), with experience of applying them to complex cases found in real debug info, culminating in an imple-

mented coverage tool which improves on earlier metrics by obtaining an accurate and achievable “complete coverage” baseline (§3.5);

- experimental evidence showing that this metric reliably reflects changes in debuggability across compiler versions (§3.6.1), can explain the debuggability effects of compiler changes (§3.6.2) and can reproduce *with expected differences* findings of a prior study (§3.6.3).¹

3.2 Understanding debug coverage

With the debugging illusion concept and the sub-properties of the expected illusion from Chapter 2 in mind, this chapter focuses on measuring the *extent* of local variable debug info (coverage), but not its semantic consistency (correctness). Although this means a compiler could game this metric by adding spurious debug info, compiler authors are unlikely to add this intentionally since it would offer an especially bad user experience for end developers. In Chapter 2, we defined the variable coverage property as follows (§2.4):

Variable coverage At any reachable point, it is possible to examine all named values that the source program deems to be *in scope* at that point and also taking a well-defined value—roughly, it is not uninitialised (defined more precisely in §3.3.2).

3.2.1 Debug info as residualised code

As we explored earlier (§2.3.2), compilers may optimise code so that a variable is no longer explicitly represented. Such variables remain coverable at debug time in modern debug info formats like DWARF [2017], which describe them as functions to be computed by the debugger. In DWARF these are *expressions* in an interpreted stack machine language. Expressions can compute a variable’s *location* in memory or the register file—often a simple offset from the stack pointer suffices, but sometimes a more complex expression is needed (e.g. in a nested function, traversing a static-link pointer to reach locals in the lexically enclosing scope)—or they can compute a variable’s *value* directly. Fig. 3.1 shows an example.

¹Experiments and implemented tools are available in a deposited artifact [Stinnett and Kell 2024b].

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

Fig. 3.1: How DWARF might describe a local variable over four distinct address ranges within a function. Most expressions compute where it is *located*: in a register or (later) on the stack. Over the second range, however, it is not represented explicitly; its *value* is computed as a scaled difference of two registers. (The textual syntax is for illustration only.)

source	instructions	location views, value exprs
1: int x = 1;		view 1: ln 1, x: (value 1)
2: x++;		view 2: ln 2, x: (value 1)
3: x++;		view 3: ln 3, x: (value 2)
	0xd06: mov \$3, %rdi	view 4: ln 4, x: (value 3)
4: x = f(x);	0xd0d: callq f	ln 4, x: (value 3)
	0xd12: mov %rax, %rdi	ln 4, x: (reg RAX)
5: ...	0xd15: ...	ln 5, x: (reg RDI)

Fig. 3.2: How DWARF can conceptually residualise control-flow positions that were eliminated during optimisation, using the “location views” extension of [Oliva \[2010\]](#). A single program counter value (here 0xd06) can have a numbered sequence of “views” such that a single local variable is described differently for each view. At debug time, control appears to pass through each view in sequence, even though the program counter does not advance.

Compilers may also optimise code so that an intermediate control-flow position of the source program is elided, having no corresponding program counter position in the object code. Again, full variable coverage over these positions often remains feasible, as debug info can represent intermediate states to be synthesised by the debugger, effectively existing *in between* the instructions of the object program. [Fig. 3.2](#) shows an example. This facility is less well established but is implemented in (at least) extensions to DWARF that are proposed for a future standard and already used by GCC [[Oliva 2010, 2017b,a](#)]. Currently, eliminated *branching* control flow cannot be represented in DWARF or any proposed extension.

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

3.2.2 Research questions

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

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

question (§3.3).

RQ2. For measuring coverage, what are the necessary practical steps? How can we measure the extent of local variable coverage in debug info, in a way that fairly represents the effective debug-time availability of the program’s local variables? To answer this, we need to define an accurate baseline for when a variable *should* be present, i.e. one that accounts for residualisation opportunities. This is complicated by the sometimes unpredictable and counterintuitive behaviour of compilers and the partial information they generate. We define a measure of coverage with respect to a given local variable, capturing over how much of the program the debugger is able to show a value for it, relative to the achievable baseline. We improve on previous metrics by ensuring the baseline *is* achievable, using both conceptual insights (§3.4) and experiences during implementation (§3.5).

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

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

3.3 A conceptual basis for coverage metrics

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

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

This demands answers to three questions: what counts as “covered”, what counts as “coverable”, and in what unit these are counted. In this section we answer the first two of these, by proposing definitions for when a local variable is *covered* and *coverable*, made by analogy with a familiar liveness analysis using the data-flow method [Allen and Cocke 1976].

3.3.1 Liveness as a state space

Traditional liveness analysis [Allen and Cocke 1976] deems a variable to be live at a point if a *definition* reaches that point and has at least one later *use*. We can model each variable in such a scenario by two unary predicates: whether it has been initialised (“Defined” or *D*) and whether the current definition will be used again (“Live” or *L*). Although these two predicates

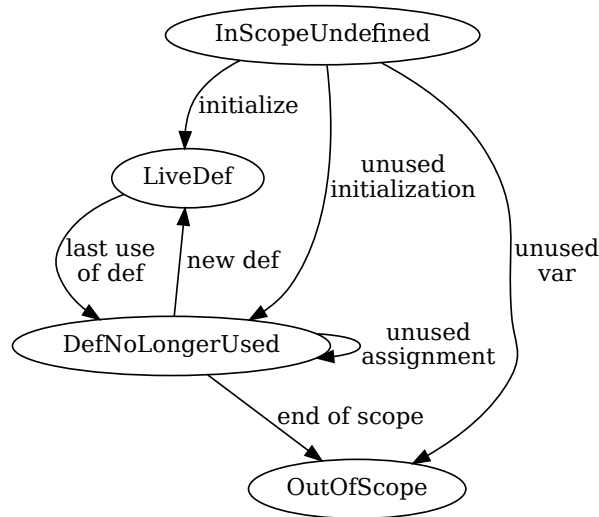


Fig. 3.3: Classical view of the lifecycle of a local variable.

are orthogonal, a reduction applies since conventionally “Live” is assumed to imply “Defined” (i.e. uninitialised reads are not considered), leaving three in-scope states. The resulting machine is shown in Fig. 3.3; for readability, an additional “out of scope” exit node is added.

3.3.2 A more realistic lifecycle

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

Multiple deaths A variable may be defined (initialised) or not, but “dead” is a less clear concept. An in-scope variable that is deemed dead by the optimiser may still be requested by the debugger’s user, even though the program does not need it. From here we qualify “dead” as “program-dead”, to highlight this. Consider the straight-line program shown in Fig. 3.4. On the right is the source code, and on the left is a flow chart of significant events in the variable’s lifecycle, showing its two additional deaths: when it is unrecoverable (unknowable), and when it is no longer in scope.

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

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

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

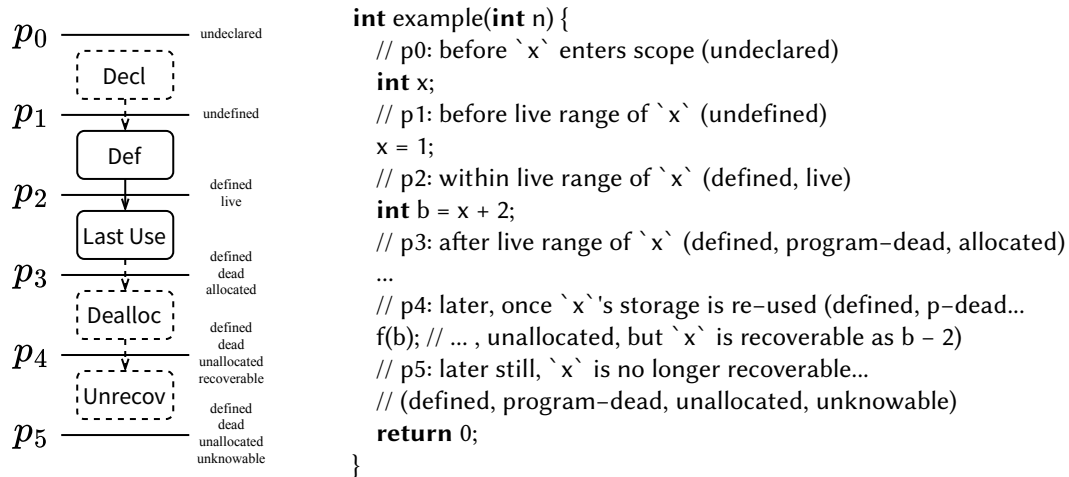


Fig. 3.4: Variable `x` is in different states across a range of program points p . The right-hand C source function is annotated with these points. The left-hand flow chart illustrates `x`'s lifecycle in a hypothetical compilation, from before `x`'s introduction (`Decl`), across its definition (`Def`), last use (`Last Use`) and the subsequent reclamation of its stack or register storage (`Dealloc`). A final `Unrecov` event models the point from which there is no longer enough state to reconstruct the value.

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

These four elements are again somewhat non-orthogonal. To be knowable and to be live both require being defined. (We regard uninitialised values as meaningless and therefore trivially unknowable.²) To be allocated and program-live is also to be knowable (simply by loading from the allocated storage; a DWARF “location expression” denotes this). After applying these reductions, eight states of a possible 16 remain; the state space is shown in [Tab. 3.1](#).

“Knowable” here means that reading from storage and/or evaluating *some* debug info expression *could* obtain the variable's value. This subsumes three sub-cases: the allocated case (“knowable” simply by loading from storage), the “recoverable” case when a variable is computable as a function of other (redundant) state (as the “scaled difference of pointers” in [Fig. 3.1](#)), but also the “literal value” case e.g. in

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

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

²There may be run-time language or program environment features that cause uninitialised variables to have values in certain cases, due to memory-mapped IO, signals, etc. The lifecycle discussed here focuses on statically-known variable states that would be encountered at compile time.

Table 3.1: Case analysis of in-scope variables, which at any point may (A) be allocated to a storage location, (D) take a defined value according to source semantics, (L) hold a program-live value, and (K) be possibly knowable given the right debug info.

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

The restriction of K to functions of the *current* object program state reflects how under current tools, when state is to be dropped from the object program, there is no mechanism that can cause it to be remembered e.g. by an attached debugger. Dropped state is not residualised. Whether there exists any such function appears undecidable in general, posing a problem for this metric—which seeks to determine whether each variable is possibly *coverable*. For now, we assume that all local variables are coverable at all points where they are defined, but with optional special handling for the positions after their last use (i.e. when they are finally program-dead). This can avoid penalising compilers for discarding state as a usual register allocator would, with the rationale that this *could* still be covered by a debugger. Prior work has done so [Tice 1999], and is similar to a technique we describe later (§3.6.4).

Finally we define two further predicates:

- $S(p)$ is true iff a variable is in scope at p ;
- $B(p)$ is true iff the debugging information describes a variable at p .³

Note that B is true when the debug info *actually does* describe the variable, either in its allocation or as a residualised expression—as distinct from whether it potentially could (K). As in Fig. 3.3, when a variable is out of scope, the other predicates are no longer of interest.

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

³We chose B to mean “described” only because we have already assigned D to “defined”.

3.4 Defining a coverage metric

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

3.4.1 Naïve instruction-based metrics

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

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

This is what is computed by existing tools such as `llvm-dwarfdump` [LLVM Project 2020b] and `debuginfo-quality` [O’Callahan 2018]. Unfortunately this results in four problems.

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

Secondly, in the presence of *location views*, as shown in Fig. 3.2, it simply **omits to count some coverage**. Location views can cover a source variable over a range of zero instructions, which by definition will not be counted.

Thirdly, in practice it **accidentally favours some compilations**, specifically unoptimised compilations which put local variables in a stack slot for their whole lifetime. Conversely it penalises optimised compilations using a register allocator. This is because a variable described as being located in a stack slot will be in B (it is described) over all program points in S , even those where the variable is not yet defined (points not in D). Conversely, a variable that is stored only in registers will be in B only where it is also in D because registers are allocated not over a whole function but over specific ranges of instructions—*live ranges*, which always start at a *definition*, so naturally exclude program points not in D .

To make this more concrete, we return to the example function from Fig. 3.4. Variable `b` is not defined until part way through the function. An unoptimised compilation may allocate stack storage for this variable across the entire function scope, while an optimised compilation likely uses a register only for region of the function where `b` is defined. Existing coverage tools take an overly simplistic stance that variables should be coverable for their entire containing scope, but the optimised version will not (and should not need to) cover `b` for the entire function, as coverage is only actually required within the variable’s defined region.

Put differently, the coverage approach used by existing tools overestimates coverage by

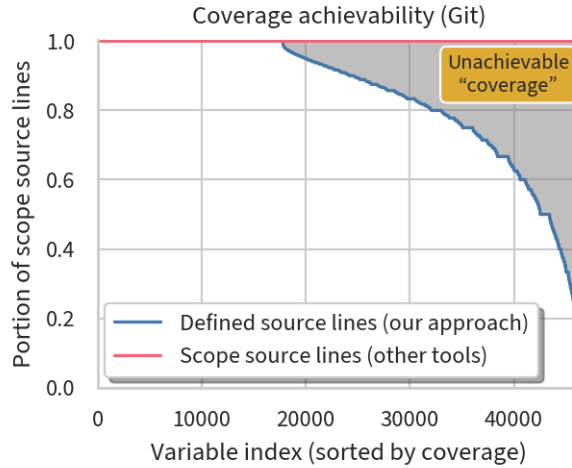


Fig. 3.5: Coverage achievability in the Git codebase. The gap between the lines represents “artificial” coverage that other tools (`llvm-dwarfdump`, `debuginfo-quality`) suggest exists but cannot be achieved as it is outside the variable’s defined ranges. The “other tools” line is always 1.0 by definition and included only to emphasise the difference between approaches.

counting as covered a variable in a described stack slot that holds only garbage (uninitialised) data ($B \supset D$). (Usually a “variable not defined” message would be preferable in these cases!) This overestimation means **full coverage becomes impossible to achieve** when using register allocation, since the denominator counts program points over which the variable will have no described value so cannot be counted in the numerator ($B \subset S$). Fig. 3.5 reveals the unachievable coverage which other tools (`llvm-dwarfdump`, `debuginfo-quality`) suggest exists but is ultimately unattainable after accounting for each variable’s defined region.

3.4.2 Counting source lines, not instructions

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

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

The debug info includes a line table, supplying a mapping such that any set of source lines can be projected to a set of instructions or vice-versa. By performing this projection, we obtain a set of source positions (e.g. pairs of (filename, line number)) that should be embodied by *any* compilation of the same code. When P is defined in this way, the number of instructions embodying any particular source location may vary arbitrarily without affecting the metric. This is intentional: we deem instruction counts not relevant to the experience of a developer debugging at source level.⁴

⁴The focus on lines does mean that very long source lines, e.g. as produced by some macro expansions, suffer limited resolution. However, this metric could work with any notion of *program points* that is reflected in the compiler’s line table. For example, source features could be correlated to DWARF column information, but compiler

Line tables’ mappings are not one-to-one: inlining leads to source lines mapping to many program counters, while various code-folding transformations do the converse. While we may still calculate sets of source lines or instruction bytes, a local variable may be covered or not covered at each. The simplest scheme (which is what we implement) calculates a variable’s coverage as an unweighted mean across all realised instances of a source line, e.g. inlined copies plus the out-of-line copy. Each source line continues to contribute at most 1 to the metric’s numerator, so in the case of such “multiply instantiated” lines, a contribution may be fractional rather than just 0 or 1.⁵ (One could also argue for unequal weighting among these instances, e.g. supposing the out-of-line copy is called more often than each inlined copy. Making a meaningful choice would require profile information.)

3.4.3 Correcting accidents, permitting full coverage

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

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

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

support for this information is patchy.

⁵For exposition we continue to use the set cardinality notation, in spite of this generalisation to fractional contributions. Another fractional case might be where only certain bytes of a value are available, as common with locals of `struct` or other composite type. The tool does not currently account for this, but could easily be extended to do so.

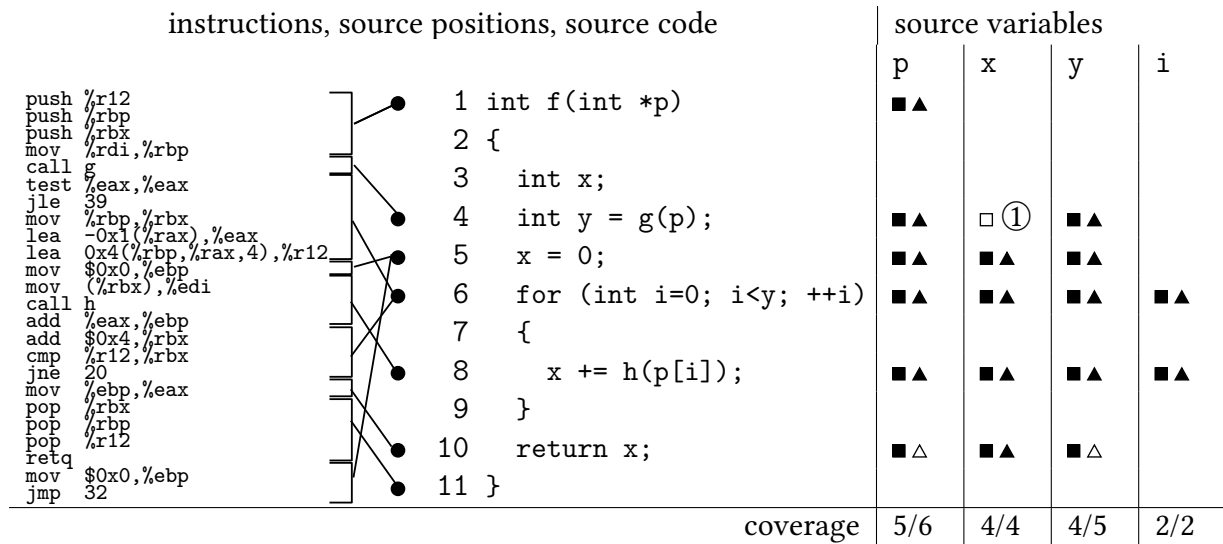


Fig. 3.6: An annotated view of how this coverage metric might view a simple compiled function. Circles ● denote source positions identified in the debug info. Square boxes mark those source positions for which a variable is in-scope and undefined (□) or defined (■). Triangles mark those for which the given source variable is covered by debug info (▲) or not (△); the markers shown are typical for a lightly optimised version of the code. ① marks a location where variable x is not yet initialised but, if allocated storage e.g. on the stack, would be erroneously counted as covered by a naive metric.

3.5 Implementation

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

Heuristic attempts To avoid the complexity of control-flow reconstruction, the developers of the `llvm-dwarfdump` tool trialled a modification where, essentially, an approximation of $|S \cap D|$ was used. If S consists of a set of instruction bytes, as offsets within a function, the denominator is $|S \setminus [0, n)|$ where byte n is the first over which *any* DWARF expression for the variable is defined. However, this is circular: it uses what is *covered* (the instructions covered by defined DWARF expressions) as a proxy for what is *coverable* (the instructions over which the variable *should* be covered), undermining the intention of a coverage metric to capture the gap between these. Indeed, the heuristic was found to be unreliable and removed [Bessonova 2019].

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

mem2reg as a baseline To approximate D by excluding those lines which *precede the variable's point(s) of first definition*, one method is to use the compiler itself. Although a fully unoptimised build does not help (because the compiler usually just places all variables on the stack for their entire scope), adding *just* LLVM's mem2reg pass⁶ to the unoptimised 00 build largely has the right effect: it moves variables into registers and describes them only from their point(s) of definition. Essentially, this exploits mem2reg's liveness analysis to approximate $S \cap D$. Since this pass works on virtual registers, it is not limited by the supply of physical registers, so is applied to any non-address-taken scalar local. We call this baseline 00-mem2reg. Unfortunately, various uncontrollable effects make 00-mem2reg unsuitable as a baseline. At different optimisation levels, we found that the line number tables map the same code to slightly different source ranges and that 00-mem2reg tends to yield a slight subset of the true D . Some 01 compilations report source lines absent from the 00-mem2reg baseline set, leading to coverage over 100%, which is clearly neither accurate nor acceptable.

Source analysis Instead of mem2reg, we use a source-level analysis to calculate S and D as sets of source line numbers. These sets are constructed by a series of “filters” over the source line number space. We classify AST nodes according to whether they *perform computation*—for example, declaration-only nodes do not perform computation, but nodes using variables and invoking operators do. (Note that this computation may later be residualised into debug info; there is no obligation for this computation to be embodied by *instructions* in the eventual binary.) We use the parser's retained source coordinates to map AST nodes to lines.⁷ Any *non-computational* lines (e.g. declarations, braces, and other lines that do not modify program state) that appear in the DWARF line table are ignored when calculating the measured binary's set of described lines (B), sidestepping the 00 and 01 variability seen with mem2reg. The source analysis similarly computes S and D as subsets of the file's computational lines. Since S and D are determined by source analysis, the denominator remains valid even when compilers' line tables are incomplete.

Static versus dynamic analysis The approach we have described is a static analysis examining all code; it does not rely on collecting execution traces. However, in the presence of unreachable code it may be fairer to filter out lines that are not executed across some collection of test inputs. This method is adaptable straightforwardly to this approach, simply by an addi-

⁶LLVM's mem2reg pass is one of the first passes in the optimisation pipeline. Language front-ends (like Clang) typically emit LLVM IR using stack allocations (alloca instructions) extensively. The mem2reg pass promotes stack allocations that are only referenced by loads and stores into registers via SSA values. It also includes the scalar replacement of aggregates (SROA) transformation, which breaks up structure allocations into allocations of each member where possible.

⁷While AST nodes do offer more precise source coordinates than just lines, the conversion to source lines is useful here as the other steps of this coverage approach are also tracking sets of line numbers, which we wish to filter here via set intersection. Future work could explore improving precision for the less common case of multiple meaningful expressions on a single line by e.g. first reformatting the source file to move each expression to its own line.

tional filter on the sets of lines in the numerator and denominator. We describe an experiment using this approach in the next section.

3.6 Evaluation

This chapter explores four research questions (detailed in §3.2.2 and also summarised here):

RQ1. For measuring coverage, what is a viable conceptual basis?

RQ2. For measuring coverage, what are the necessary practical steps?

RQ3. In aggregate, how does this metric’s picture of debuggability depart from those computed by naïve variations and/or previously proposed metrics?

RQ4. In detail, does this metric explain debuggability gaps in a way consistent with how these are understood by real compiler developers?

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

3.6.1 Experience with metrics

Fig. 3.7 shows coverage achieved for the Git codebase as compiled by Clang 15 and GCC 13, as measured using this metric. Coverage is computed per variable and plotted after sorting the variables by coverage. Due to this sorting, we can see the shape of the coverage distribution, but we cannot correlate individual variables across compilations, as each variable’s index may be different for each line. The “plateaus” seen in each line represent groups of variables that happen to have a coverage ratio equal to some common fraction (e.g. $\frac{1}{3}$, $\frac{1}{2}$, $\frac{2}{3}$, etc.). Each compiler is run at multiple optimisation levels. As one might expect, when comparing coverage across optimisation levels, we see that any optimisation beyond 00 significantly degrades coverage. GCC’s 0g optimisation level (which is a variant of 01 with some debug-info-interfering passes removed) offers the best coverage of the optimised runs. Clang does not currently (as of 2024) offer a meaningful 0g (it is merely an alias for 01), but the LLVM community is working on adding this in the near future [Livermore-Tozer 2023]. Recent versions of Clang enable nearly all of their implemented optimisations at 01, so the higher levels (02, 03) show only minor differences. The difference between 00 and 00-mem2reg (which moves most variables off the stack and into registers) is intriguing, since it shows a sizeable amount of coverage lost even after little optimisation. At 01, GCC shows greater coverage than Clang for every variable: GCC’s dashed green line in the figure is completely to the right of Clang’s solid green line. At higher optimisation levels (02, 03), GCC fully covers more variables than Clang at the same

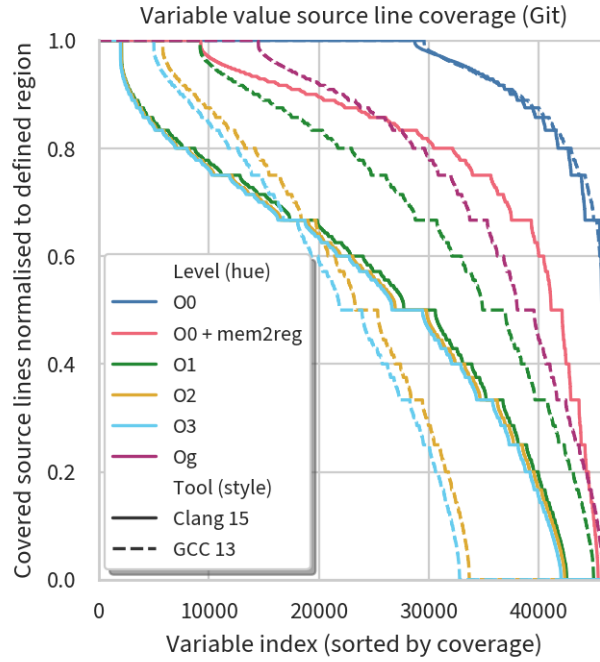


Fig. 3.7: Variable value source line coverage for Git codebase. For each optimisation level, variables are sorted by coverage. Due to this sorting, we can see the shape of the coverage distribution, but we cannot correlate individual variables across compilations, as each variable’s index may be different for each line.

setting, as seen by where their lines first appear at the top of the figure, but GCC’s coverage then drops more rapidly, intersecting Clang’s corresponding lines in the figure near the 0.65 coverage point on the vertical axis.

Fig. 3.8 shows coverage across 6 analysed packages. These packages were chosen to explore coverage across different application domains. The list includes Dash (shell), FFmpeg (media), Git (version control), libbz2 (compression), FLINT (numerical), and tar (archive). Compiled with Clang 15 at O2, we see a similar profile across all packages.

We compared this coverage metric approach with that of other tools (`llvm-dwarfdump`, `debuginfo-quality`) for the Git codebase using Clang 15 at O2 by computing the correlation between the two. The Pearson correlation coefficient is 0.656, which suggests only a moderately correlated positive relationship. Indeed, this approach often computes quite different coverage values than past approaches [LLVM Project 2020b; O’Callahan 2018]. Several factors explain this. As highlighted earlier (§3.4), we lift from bytes to lines, restrict to computational lines, and include only the defined region as covered and coverable. These improvements affect both numerator and denominator, and manifest differently for each variable depending on whether each line of its defined region is covered or uncovered.

3.6.2 Case studies

RQ4 asked: in detail, does this metric explain debuggability gaps in a way consistent with how these are understood by real compiler developers? We examine this through two case studies.

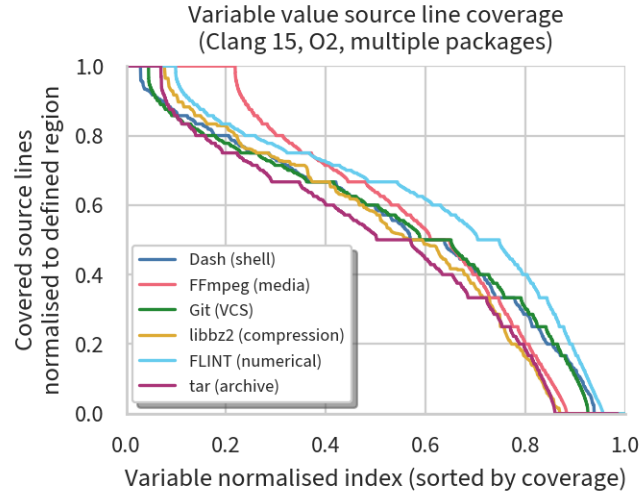


Fig. 3.8: Variable value source line coverage across several packages. Variables are sorted by coverage (see further sorting notes in Fig. 3.7).

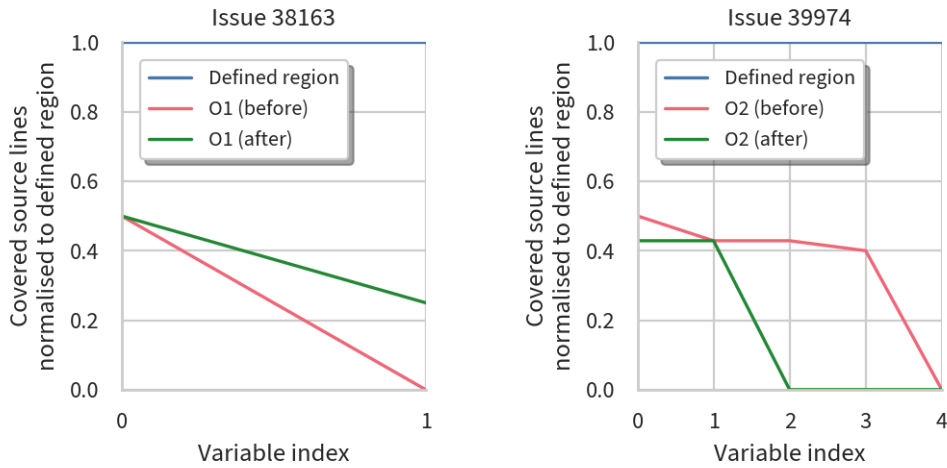


Fig. 3.9: Coverage before and after resolving LLVM issues 38163 (left) and 39974 (right). Each measured program is the (very small) bug reproduction example attached to the corresponding issue, with only 2 and 5 variables respectively.

Compiler changes can go both ways: even when an issue is *resolved*, coverage may go up or down. This is because compiler authors generally take the (quite reasonable) perspective that incorrect debug info is worse than none at all [LLVM Project 2022], so may sometimes remove coverage to avoid incorrectness. We study one case of each kind.

Debug info repaired LLVM issue 38163 (“Loop strength reduction preserves little debug info”) involves the LoopStrengthReduce pass which transforms loop induction variables into more efficient forms. Debug info describing these induction variables was being dropped. The issue was resolved via a change which residualises the variable in some cases. As seen in Fig. 3.9, this metric shows that source variable value coverage improved as a result.

Debug info dropped LLVM issue 39974 (“Salvaged memory loads can observe subsequent memory writes”) concerns a function that loads from memory into an unused local variable.

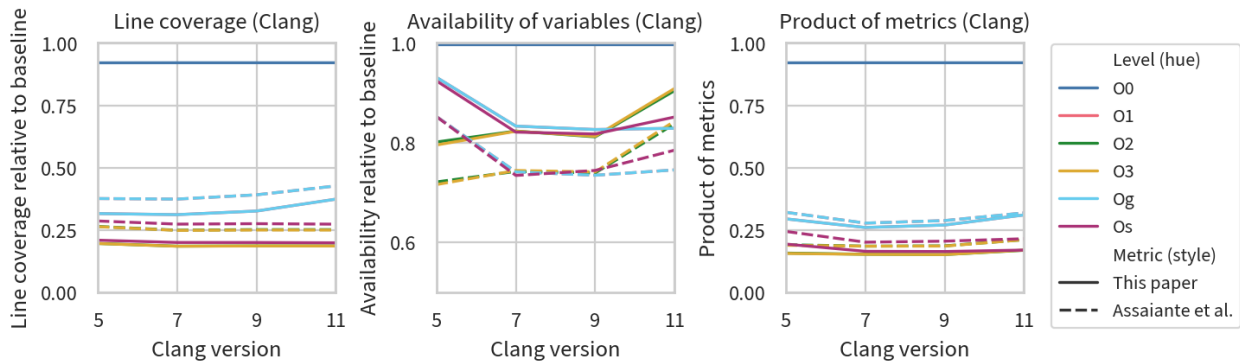


Fig. 3.10: Comparison of metric approach (using static source analysis as baseline) with Assaiante et al. [2023] (using O0 as baseline). Metrics computed using the same 5,000 Csmith-generated [Yang et al. 2011] programs as used by Assaiante et al. Of the two, only this metric is capable of measuring coverage at O0. The two metrics show the same trends across optimisation levels and versions but a markedly different absolute value, reflecting the fairer baseline computed using “scope shrinking” (§3.4.3).

The EarlyCSE pass eliminates the load but correctly residualises this as a dereference operation in debug info. However, after a later store to the pointer this expression yields an incorrect value: the newly stored value, not the value that would have been loaded earlier (which has become potentially unknowable). The compiler authors chose to remove the debug info for this variable. Fig. 3.9 shows that this metric confirms that several source variables lost all value coverage as expected for this compiler change.

3.6.3 Aggregate comparisons: a replication study

RQ3 asked: in aggregate, how does this metric’s picture of debuggability depart from those computed by naïve variations and/or previously proposed metrics? We answer this by a *replication study* in which we adapt these metrics to reproduce an experiment from recent literature. The similarities we find serve as validation of definitional and implementational correctness but we also find certain expected differences that we can explain in terms of this metric’s definition. The study reproduces the experiments of Assaiante et al. [2023] which generated the data plotted in Fig. 3.10. The metrics presented have been computed across the same 5,000 Csmith-generated programs from their published artifact.⁸ Their metrics examine local variables only, not formal parameters, so we created a similarly adapted version of ours.

While we have defined this metric in a per-variable fashion, aggregating across lines, Assaiante et al. did the converse, defining a metric per line and aggregating over that line’s variables. They first compute “line coverage” as a ratio of the lines present in the DWARF line table for a given compilation relative to the lines present in their O0 baseline. Next, they compute “availability of variables” by attempting to stop in the debugger on each such line,

⁸Csmith [Yang et al. 2011] (when using default settings without further tweaks) tends to produce programs with dead code that ends up getting optimised away, which ends up shrinking the region over which coverage can be explored. Future work could explore coverage in programs from other generators, such as YARPGen [Livinskii et al. 2020], as they may produce programs which retain more code across optimisation.

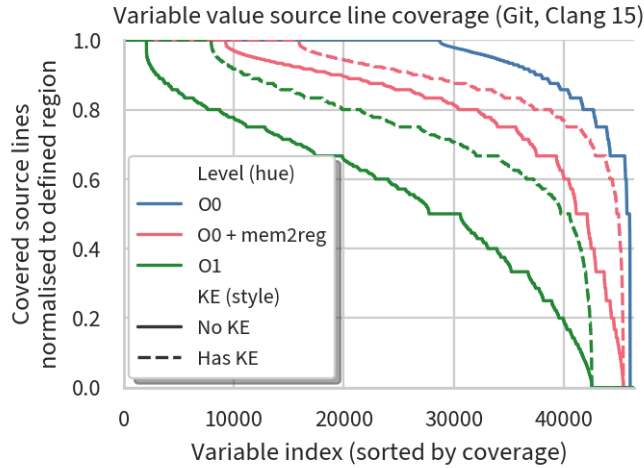


Fig. 3.11: Coverage for the Git codebase before and after end-of-scope “knowledge extension” (KE). Variables are sorted by coverage (see further sorting notes in Fig. 3.7).

and recording the number of variables accessible relative to O0. The overall availability score is the arithmetic mean. Fortunately, this method is straightforwardly adapted to a line-oriented approach, simply by enumerating at each line the familiar denominator $|S \cap D|$ (the variables that are both in-scope and defined) and numerator $|B \cap D|$ (the variables that are both described and defined). This approach is able to compute the metric for O0, since we use our familiar source analysis as a baseline, whereas Assaiante et al. use O0 as the baseline; we noted in §3.5 why O0 is unreliable.

Their work relies on a dynamic analysis and thus will not encounter unreachable lines, which these generated programs do contain. For this replication experiment, we added a binary analysis step using a simple custom Valgrind [Nethercote and Seward 2007] tool to enumerate executed lines, as anticipated earlier (§3.5). This allowed us to obtain similar line coverage.

Similar trends appear in both this metric and theirs when looking across compiler versions and optimisation levels. Line coverage using this metric is slightly lower owing to the different baseline. Since this metric can measure line coverage even at O0, we can additionally see that if their O_n coverage is multiplied by the O0 value (which is their baseline), we indeed arrive at (approximately) our own O_n line coverage. Meanwhile, the availability of variables is somewhat higher. This reflects the expected improvement of this approach: by limiting coverable lines to the defined region, we avoid the problem of the over-counting stack-based O0 baseline, which artificially prevents reaching 100% coverage (§3.4.1).

3.6.4 Experimenting with end-of-scope treatments

Scope shrinking (§3.4.3) disregards a variable’s region of undefinedness at the beginning of its scope, i.e. until it is defined. A near-dual scenario exists at the *end* of its scope: a defined variable may have a period of *unknowability* when its last value, although well-defined in source terms, is no longer recoverable from the object program state (seen in Fig. 3.4). Since DWARF expressions residualise computation, but do not have the ability to record additional

state, even the most sophisticated DWARF expression could not reconstruct the variable in this case. The variable’s last value would need to be saved just before becoming unknowable, but DWARF itself has no means to do so.

However, a debugger could be implemented today that performs this *knowledge extension*, by saving a variable’s value at its last moment of knowability (computed from the instruction ranges of debug info). Fig. 3.11 shows a simulation of the potential coverage gains after applying end-of-scope knowledge extension. Tice [1999] implemented a similar state-residualisation technique for local variables in the Optdbx debugger, remembering dead values while they are still in scope but no longer register-allocated. DWARF version 5 alludes to such possibilities (in the specification of DW_OP_entry_value, the operation to obtain *somehow* a value as it existed on entry to the function) and *omniscient debuggers* (e.g. Pernosco) take the extreme position of ensuring full knowability at all times.

3.7 Conclusions and future work

We have defined the variable coverage property of correct debug info, building on the idea of debug info as a “residualisation” of the optimised-away parts of the original program first mentioned in §2.3. We have elaborated this properties into a coverage criterion that we have embodied in an implemented tool and found useful in finding and explaining real debuggability problems. The residualisation perspective suggests future work that applies to this entire thesis. We have collected these thesis-wide project ideas in §7.2.

Chapter 4

Consistency checking of function call metadata using dynamic call trees

While [Chapter 3](#) explored variable completeness, we now examine a different testing approach focused on consistency of control metadata. 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. In this chapter, 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.

4.1 Introduction

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” 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, not only 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

```

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. 4.1: 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.

represent these. Over the past 30 years, the DWARF format has steadily accreted new features for handling various optimisations of these kinds (§2.3).

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.

With the debugging illusion concept and the sub-properties of the expected illusion from Chapter 2 in mind, this chapter focuses on how the function call / return subset of control consistency has been affected by features such as inlining and tail calls during optimisation. In Chapter 2, we defined the control consistency property as follows (§2.4):

Control consistency For any given reachable program point in the source program, it is possible to stop there in all compilations of the program.

This chapter explores how *dynamic call trees* may be used to define a strong correctness criterion. We make the following contributions:

- 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 debug info.

- 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 this 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 these classified divergences across various codebases, compilers, compiler versions, and optimisation levels. This data reveals certain patterns and trends across different compilers, while confirming that this categorisation explains most divergences seen.¹

4.2 Background

4.2.1 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.

4.2.2 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 this 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.

¹We have published an open-source artifact [[Stinnett and Kell 2026b](#)] containing both the tools we developed for this chapter and the experimental pipelines used to acquire, process, and display results.

Some existing tools can generate dynamic call trees, including `uftrace` and `funtrace` for user programs, or `ftrace` in the Linux kernel.² 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, instruments 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 (§4.3).

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, as highlighted earlier (§2.3). 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. Debug info formats (e.g. DWARF) can residualise computation and some parts of program state as well.

4.2.3 Limited differential testing 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 (Chapter 3).

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 for testing backtrace and other dynamic call tree-related features at debug time [Di Luna et al. 2021]:

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

²At 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>

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.

4.2.4 Compiler bugs vs. format bugs

In this chapter, we make a contribution towards enabling *strong* differential testing³ of debugging information. One 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 (§2.3), 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.⁴) This existence proof escapes from the horns of the dilemma mentioned in this chapter’s 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.

³Following the terminology from Weyuker [1982], we say that this chapter enables “testing” instead of “testability”. Under their model, software is testable if there exists any practical oracle at all. Finding a *better* oracle (as we have done here) enables stronger testing of something that was already testable.

⁴This “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.

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 unoptimised 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 chapter 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.

4.3 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? These answers are gathered from an exploratory experiment over a selection of real codebases and compilers.

4.3.1 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 source-level 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”. (A subprogram need not occupy a single contiguous “range” of addresses, hence the preference for “space”.)

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.

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.⁵

⁵A 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

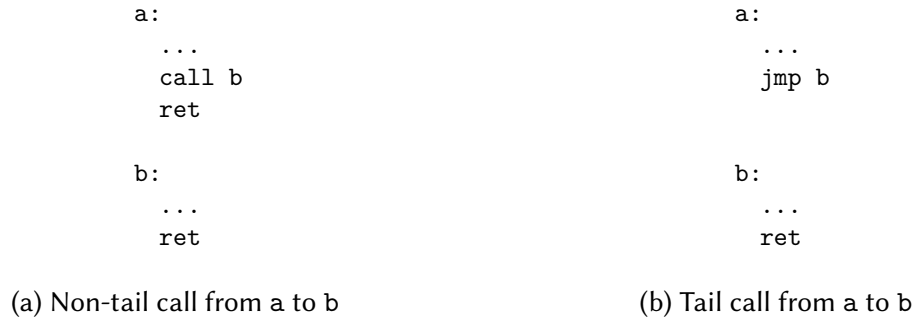


Fig. 4.2: Example machine instructions for non-tail vs. tail call scenarios. The callee b ends with a return in both cases (unless it performs its own tail call).

Out-of-line returns always occur as a return instruction, so they are simpler to recognise compared to out-of-line calls. As Fig. 4.2 shows, the callee ends with a return for both the non-tail and tail out-of-line cases (unless it performs its own tail call).

This 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. This 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 these call trees against frame information, although prior work has considered the testing and synthesis of frame information separately [Bastian et al. 2019].

not a source-level call. Such an annotation does *not* exist in current DWARF. Luckily this idiom is an assembly programmer’s trick and these experiments have not encountered it in any compiler-generated code. If they had, it would be featured in Tab. 4.2.

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 it is not collected separately. Therefore, this 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.

4.3.2 Collecting a call tree: implementation

We built tools to collect a dynamic call tree of executions of user-level binaries using the Quarkslab QBDI dynamic binary instrumentation framework [Hubain and Tessier 2017]. This is able to preserve the original instruction boundaries, and in our informal experiments it was found to be 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 these QBDI-based tools ran at ~270x native execution, while a `ptrace()` approach was far slower at ~1,700,000x native execution.⁶

The tracing and instrumentation features of the approach we implemented 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 §4.3.1), we push an 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 this experiment, we choose to observe a single executable binary at a time, since this is the granularity

⁶These informal experiments were performed on macOS 13. Other platforms (e.g. Linux) may have different performance characteristics.

at which we have been controlling compilation. Therefore, we do not collect traces within calls to shared libraries. The tool must handle specially the platform-specific mechanisms, such as procedure linkage tables, used to reach external code.

4.3.3 Exploratory study outline

What do we find when comparing call trees across unoptimised and optimised executions? The method of this 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 (§4.3.4.1).
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. 4.1](#) and [Tab. 4.2](#).

Following experience, in order to allow for optimised 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 (§4.3.6).

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

`< CT | CF | RF, function, coordinates >`

for example

`< CT, main, 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, the 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 these 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 The 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 §4.4.2.)

Expected and unexpected divergences Tab. 4.1 and Tab. 4.2 show the divergences resulting from this 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 (though we will discuss a reasonably straightforward workaround momentarily), 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.⁷

Separation criteria To separate “expected” from unexpected divergences, some judgement is required. Some recurring classes suggest obvious questions about 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 bug arguably lies not in current DWARF features but in how these are customarily interpreted

⁷Most of these enhancements have not yet been directly discussed with the DWARF community. Based on my past experience proposing similar debug info changes, there can be both engineering trade-offs as well as political nuances to navigate. Existing feature reuse in new ways can be acceptable if agreement can be reached among the community. In other cases, an extension of the standard ends up being the chosen path. The community can also be sensitive to debug info size increases. They may demand a stronger argument that a new feature is likely to deliver a sufficiently improved experience if it will cause a large increase in debug info size.

Table 4.1: 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 this exploratory experiment.

Category	Semantic preconditions	Path to resolution
Coordinates changed by small delta (≤ 3 lines) (§4.3.4.4)	Various	Locations views and/or dependent coordinates (see §4.3.4.4, §4.3.4.5)
Coordinates removed owing to control flow merging (§4.3.4.5)	Repeated steps across multiple control flow paths	Dependent coordinates (see §4.3.4.5)
Added library call (§4.3.4.1) (e.g. array initialisation loop $\rightarrow$ added <code>memset</code> 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 <code>memcpy</code> $\rightarrow$ converted to store)	Specialised transformations for known library functions	Call could be described in debug info like inlining
Replaced library call (e.g. simple <code>printf</code> $\rightarrow$ converted to <code>puts</code>)	Specialised transformations for known library functions	Call could be described as original call like inlining (§4.3.3)
Out-of-line call removed by constant propagation (§4.3.4.2)	<code>static</code> 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 (§4.3.4.3)	User <code>const/pure</code> 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 §4.3.4.2)
Spurious entries/exits of a single inlined callee (§4.3.6)	Callee body available	Inlining metadata could describe entries/exits along control-flow edges; see §4.3.6
Merged multiple inlined calls to the same callee (§4.3.6)	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 §4.3.4.2
Code motion of one non-inlined call site past another (§4.3.4.6)	User <code>const/pure</code> annotation or callee body available and provably effect-free	Possibly “reorder buffer”-like residual state to delay observation of hoisted call §4.3.4.6

Table 4.2: 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) (§4.3.5.1)	“Single return” or other exit path transformation	Lack of convention for coordinates of function exit
Tail call missing (§4.3.5.2)	Various (for calls in tail position)	Coding omission or limitation of compiler architecture
Inlined call missing	Various (callee body available, ≥ 1 identifiable instruction remaining)	Inlined call should be described; see §4.3.6
Coordinates removed despite absence of control flow merging	Various	Coding omission or limitation of compiler architecture

<pre> 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 </pre>	<pre> - 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 </pre>
--	--

Fig. 4.3: 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 these tools [Chawathe et al. 1996].

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. 4.3 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 [Chawathe et al. 1996] in the tools we implemented, 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.

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 (§4.3.6).

<pre> 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; } </pre>	<pre> 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 </pre>
--	--

Fig. 4.4: 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.

<pre> void read_very_early_config() { struct config_options opts = {0}; opts.respect_includes = 1; opts.ignore_repo = 1; } </pre>	<pre> 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 </pre>
---	---

Fig. 4.5: 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 it looks like a removal to the 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.

4.3.4 Expected divergences

4.3.4.1 Added library call

In Fig. 4.4, we have a loop that initialises an array element by element with a constant value. The LLVM 18 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. It could also be encoded by reusing existing debug info features in a new way, e.g. by a call site marked as `artificial`—whereas current DWARF reserves `artificial` for “objects and types”.

We also found a related-but-opposite scenario, which is shown in Fig. 4.5. 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 the trace comparison tools, this appears as a call being removed.

4.3.4.2 Out-of-line call removed by constant propagation

In Fig. 4.6, `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

<code>static int is_dir_sep(int c) {</code>	CT: is_absolute_path
<code> return c == '/';</code>	CF: is_absolute_path
<code>}</code>	CT: is_dir_sep
	RF: is_dir_sep
<code>static int has_dos_drive_prefix(const char *path) {</code>	- CF: is_absolute_path
<code> return 0;</code>	- CT: has_dos_drive_prefix
<code>}</code>	- RF: has_dos_drive_prefix
	RF: is_absolute_path
<code>int is_absolute_path(const char *path) {</code>	
<code> return is_dir_sep(path[0]) has_dos_drive_prefix(path);</code>	
<code>}</code>	

Fig. 4.6: 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.

value. 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. 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 therefore 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 between 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.

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” [Morse 2024].

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 Clang does not. A zero-length inline would never appear in a backtrace (in the absence of location views⁸) and would not show up in this 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.

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).*

4.3.4.3 Out-of-line call to multiply-called callee obviated by user annotation

In Fig. 4.7, 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 (as a zero-length `inlined_subroutine` record) 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).

4.3.4.4 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 (§4.3.5.1).

⁸GCC’s willingness to generate zero-length inlines is likely related to its support for location views, although these are a logically separate feature.

<code>__attribute__((pure)) int frame_samples(</code>	CT: receive_for_stream
<code>const SyncQueue *sq, SyncQueueFrame frame);</code>	CF: receive_for_stream
<code>void receive_for_stream(SyncQueue *sq,</code>	CT: frame_samples
<code>unsigned int stream_idx, SyncQueueFrame frame) {</code>	RF: frame_samples
<code>SyncQueueStream *st = &sq->streams[stream_idx];</code>	CF: receive_for_stream
<code>if (st->samples_queued >= frame_samples(sq, frame))</code>	CT: av_fifo_drain2
<code>av_fifo_drain2(st->fifo, 1);</code>	RF: av_fifo_drain2
<code>st->samples_queued -= frame_samples(sq, frame);</code>	- CF: receive_for_stream
<code>}</code>	- CT: frame_samples
	- RF: frame_samples
	RF: receive_for_stream

Fig. 4.7: Similar example to Fig. 4.1, 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 <code>uint64_t getnanotime(void) {</code>	CT: getnanotime at trace.c:1
2 <code>static uint64_t offset;</code>	- CF: getnanotime at trace.c:8
3 <code>if (offset > 1) {</code>	+ CF: getnanotime at trace.c:0
4 <code>return offset + highres_nanos();</code>	CT: gettimeofday_nanos at time.c:1
5 <code>} else if (offset == 1) {</code>	RF: gettimeofday_nanos at time.c:4
6 <code>return gettimeofday_nanos();</code>	RF: getnanotime at trace.c:13
7 <code>} else {</code>	
8 <code>uint64_t now = gettimeofday_nanos();</code>	
9 <code>uint64_t highres = highres_nanos();</code>	
10 <code>offset = highres ? (now - highres) : 1;</code>	
11 <code>return now;</code>	
12 <code>}</code>	
13 <code>}</code>	

Fig. 4.8: 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 only as a function of the program counter alone. (This is signified by “line 0”; bona fide line numbers start at 1.)

4.3.4.5 Coordinates removed by control flow simplification

In Fig. 4.8, 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 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.*

4.3.4.6 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.

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 is the call tree affected. 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*, as we described earlier (§2.3.3). 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 is “running ahead” of the committed program-order execution, 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 additional *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.*

4.3.5 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 a diff of a dynamic call tree traces. Refer to the earlier case study section (§4.3.3) for more detail on the syntax used.

4.3.5.1 Large coordinate changes

It is not expected for source coordinates to undergo large changes, which we define as larger than three lines. In Fig. 4.9, 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 on line 8, as the return coordinates, instead of the coordinates of the return statement. This issue is tracked upstream at the time of writing.⁹

⁹<https://github.com/llvm/llvm-project/issues/57567>

<pre> 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 }</pre>	<pre> CT: xopen at wrapper.c:1 - RF: xopen at wrapper.c:4 + RF: xopen at wrapper.c:8</pre>
---	--

Fig. 4.9: 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 (line 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 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 the 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?¹⁰ Intuitively, this convention should be invariant under optimisation.

At first glance, this 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 01, 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.

4.3.5.2 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 (§4.3.1), 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,

¹⁰A related (potentially) surprising debugging experience can occur with C++ destructors: when stepping past a return statement, the debugger may jump to the source of each (possibly inlined) destructor for the objects whose lifetime has now ended. After all necessary destructors have run for a given block, the debugger may then jump back to the end-of-block point before continuing onward.

<pre> 1 char *xdg_config_home_for(2 const char *subdir, const char *filename) { 3 const char *home; 4 home = getenv("HOME"); 5 if (home) 6 return mkpathdup("%s/.config/%s/%s", 7 home, subdir, filename); 8 return NULL; 9 }</pre>	<pre> - 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</pre>
---	---

Fig. 4.10: 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 the trace collector’s understanding of the current frame list as seen here.

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 the instruction-wise algorithm we described earlier (§4.3.1).

In Fig. 4.10, 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 the traces. Given complete and accurate info from the compiler, the traces do include these artificial tail calls.

4.3.6 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 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 multiple times (perhaps as part of a loop body), 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. For example, B might be inside a loop body that was unrolled. We have no way to distinguish a single activation of a longer B (intermixed with its caller A) from several activations of a shorter B.

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 1.¹¹

It is easy to reproduce this problem with a simple C program such as that of Fig. 4.11, where

¹¹This limitation arises in addition to the inability of `inlined_subroutine` records to address location views, identified earlier (§4.3.4.2).

```

1  int global;
2  void effect(void) { ++global; }
3  inline void do_it(void) { effect(); }
4  int main(void) {
5      int ctr = 0;
6      do { do_it(); ++ctr; } while (ctr <= 1);
7      return global;
8  }

```

Fig. 4.11: 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.

both GCC and Clang 02 will place the two `effect()` calls¹² 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 this 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. 4.12. Trace (i) was extracted from an unoptimised Git execution, while trace (ii) came from an optimised run (including inlining) applying only the usual instruction-wise 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

¹²Two `effect()` calls occur in the following way: the loop body on line 6 is visited twice (for `ctr` values {0, 1}), which means `do_it()` is called twice, and thus `effect()` is called twice as well.

the unoptimised view, but in reality inlining has merely allowed later code motion to interleave the instructions of the calls. In general, this 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 this dissertation’s naïve instruction-wise tracing method tends to overapproximate, debuggers’ tends to underapproximate.)

These observations suggest that a sequence of three transformations to the 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. 4.12](#) 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

CT: strbuf_vaddf at strbuf.c:390:0 ICF: strbuf_vaddf at strbuf.c:394:7 IRF: strbuf_avail at strbuf.h:139:0 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 CF: xrealloc at wrapper.c:135:2 ICT: 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	CT: strbuf_vaddf at strbuf.c:390:0 CF: strbuf_vaddf at strbuf.c:394:7 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 ICT: memory_limit_check at wrapper.c:17:0 RF: memory_limit_check 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:139:0 RF: strbuf_avail at strbuf.h:141:2 RF: strbuf_vaddf at strbuf.c:408:1	CT: strbuf_vaddf at strbuf.c:390:0 ICF: strbuf_vaddf at strbuf.c:394:7 IRF: strbuf_avail at strbuf.h:139:0 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 CF: strbuf_grow at strbuf.c:135:2 ICT: memory_limit_check at wrapper.c:17:0 RF: memory_limit_check 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	CT: strbuf_vaddf at strbuf.c:390:0 ICF: strbuf_vaddf at strbuf.c:394:7 IRF: strbuf_avail at strbuf.h:139:0 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 CF: strbuf_grow at strbuf.c:135:2 ICT: memory_limit_check at wrapper.c:17:0 RF: memory_limit_check 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	(i) unoptimised	(ii) naïve inlining	(iii) with clustering	(iv) also with merging, matching unoptimised
--	---	--	--	------------------------	----------------------------	------------------------------	---

Fig. 4.12: 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 these 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.

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. 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 it 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 this 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 later experiments (see §4.4) 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 subsequent experiments, we apply clustering only.

4.3.7 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, this 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 §4.3.4.1 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. This 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.

4.4 Quantifying divergence across compilers

Using the pattern base created during the exploratory study, we now conduct a larger experiment answering two questions. Firstly, to what extent do the issues identified in the 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. These 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

Table 4.3: 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 these 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 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 this 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 not interested in the benchmark score). [Tab. 4.3](#) details each package we examined and any systematic filtering we applied.

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

The tools detect the “expected” divergences of [Tab. 4.1](#) and report a deduplicated count of affected source positions. They also detect remaining “unexpected” issues of [Tab. 4.2](#); 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.

4.4.1 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. 4.3](#).

The following figures present divergence metrics by “bucket”. A bucket is a group of the trace patterns (as in [§4.3.3](#)). 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.

While most of the buckets have a clear correspondence with the categories we described earlier in [Tab. 4.1](#) and [Tab. 4.2](#), a few would benefit from further explanation.

The *inlined reentry added* and *inlined noise added* buckets represent slightly different variants of additional calls that appear in traces when inlining is used. The *inlined reentry added* variant is an unexpected divergence where the trace appears to leave and reenter an inlined frame because the compiler failed to carry inlining info for some instructions through the optimisation pipeline. We reported an example of this bucket to the LLVM community in issue [156700](#). The *inlined noise added* variant is an expected divergence where once again the trace is leaving and reentering inlined frames, but this time it is because of a group of two or more active calls that have all been inlined together into their caller (which is indeed an expected and beneficial outcome of inlining). While the inlining tree transformations we described earlier ([§4.3.6](#)) are able to cope with added leaf calls due to inlining, we do not currently apply those transformations to added multi-frame calls, as each such branch of the trace may have their own unique children. Future work may find a good transformation to remove this effect of inlining when desired.

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.

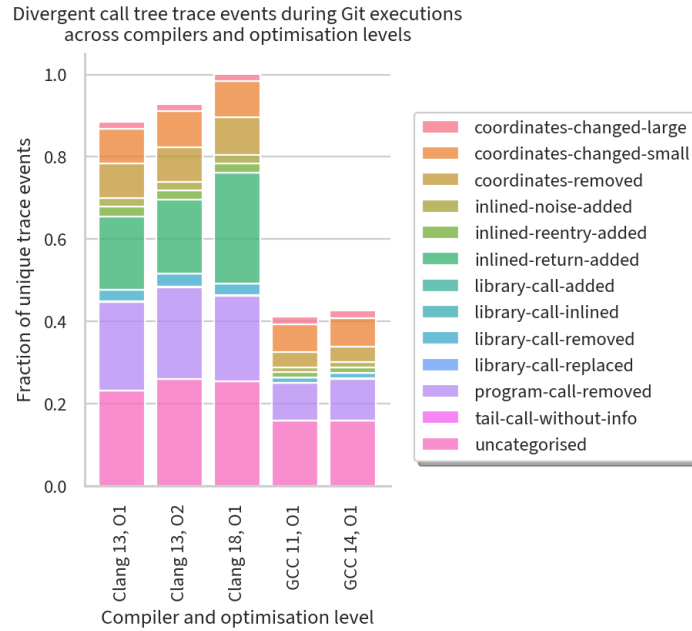


Fig. 4.13: Divergent call tree trace events seen from different compilers during executions of the Git codebase compiled at various optimisation levels.

4.4.2 Differences across compilers

Fig. 4.13 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 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 (difference in size of a given bucket) 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).¹³ 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.

4.4.3 Differences across optimisation levels

Fig. 4.13 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.

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 (i.e. the trace event’s PC address was not mapped in the DWARF line table), and any remaining

¹³There 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.

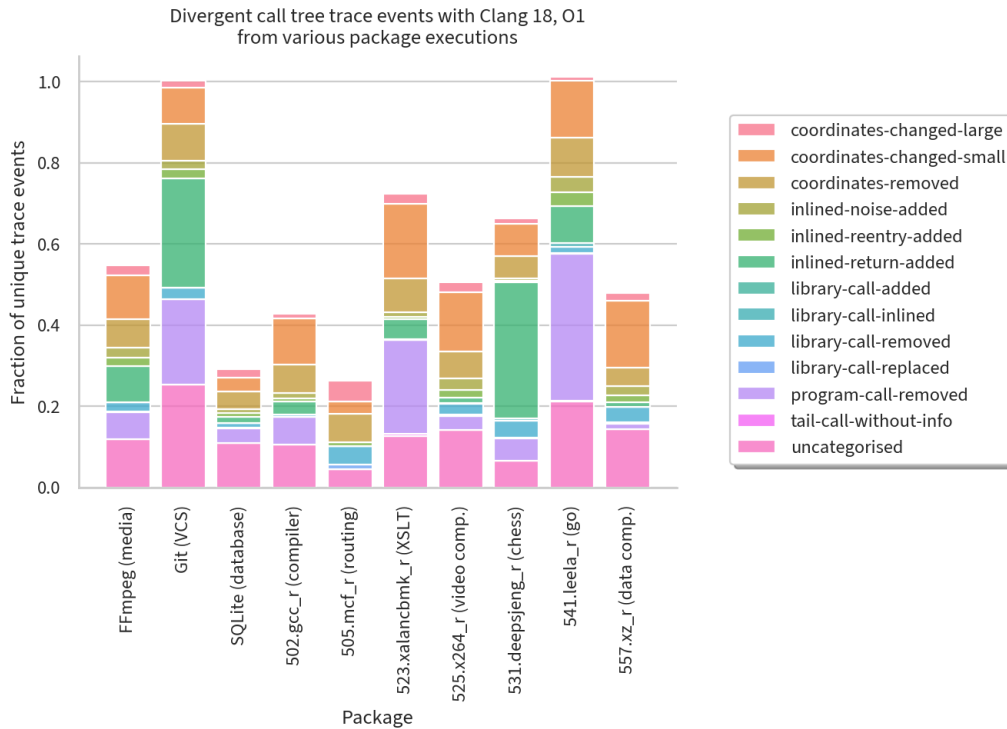


Fig. 4.14: Divergent call tree trace events during executions of various packages compiled with Clang 18 at O1.

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.

4.4.4 Differences across codebases

Fig. 4.14 shows divergences found while executing the 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. 4.3).

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 §4.3.4.2). 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.

Table 4.4: 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 decl. C++ member functions not in debug info	Fixed in 22.x
GCC	121424	Debug info associates return instruction with inlined function	Confirmed

4.4.5 Compiler bugs found

[Tab. 4.4](#) shows the compiler issues we have found and reported so far during this work. These were found by first running the automated divergence checking tool we implemented which highlights and summarises trace divergences. We then manually inspected a random sampling of these to better understand 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. The 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.

4.5 Conclusions and future work

This 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.

Chapter 5

Consistency checking of variable debug info via symbolic execution

Whereas [Chapter 4](#) explored control consistency, we now examine a final testing approach focused on variable consistency. Many debugging tools rely on compiler-produced metadata to present a source-language view of program states, such as variable values and source line numbers. While this tends to work for unoptimised programs, current compilers often throw away or corrupt debugging information in optimised programs. Current approaches for checking correctness rely on manual test cases or only cover a small portion of the compiler. In this chapter, we investigate a new approach in which unoptimised and optimised versions of a program are symbolically executed side-by-side as a way to check for consistency in their debugging information. We show that this allows checking correctness of debug info generation across a much larger portion of the compiler.

5.1 Introduction

Incorrect or missing debug info is usually regarded as a compiler bug. Current techniques used to test production compilers (e.g. Dexter [\[Bedwell 2018\]](#), used for LLVM) use manually written tests that drive a debugger to chosen points and check specific values, much like a real debugging user would. While this can increase assurance, writing such tests is onerous for compiler engineers. In the absence of a strong criterion for what level of debuggability to expect, compiler bugs are sometimes deemed “fixed” simply by suppressing incorrect debug info, rather than by correcting the problem [\[LLVM Project 2019\]](#). Experiments with both Clang and GCC in [Chapter 3](#) show that at the standard O2 optimisation level, more than half of variables are uninspectable over at least half of their live source lines.

This chapter focuses on how to replace this manual effort by automated consistency checking techniques. Currently it is not clear what it means for debug info to be fully correct, or even fully present. Historically, a “best-effort consensus” has prevailed [\[Brender et al. 1998\]](#)

that places no strong correctness criterion on compilers; recent work by Assaiante et al. [2023] has observed the difficulty distinguishing “unavoidable effect[s]” of optimisation from fixable bugs, and remarked that debug info completeness has “no reliable oracle”.

With the debugging illusion concept and the sub-properties of the expected illusion from Chapter 2 in mind, this chapter focuses on a specific consistency property that isolates the problem of *incorrect* local variable information. In Chapter 2, we defined the variable semantic consistency property as follows (§2.4):

Variable semantic consistency When examining a variable at any reachable point, its observed value is consistent with the source program’s operational semantics, in multiple ways. These can be split into *cross-consistency* properties such as “happens-before” relationships (e.g. that events occurring earlier in program order have all happened, while all later events have not) and *internal consistency* properties for single variables, e.g. observed values are explainable by the source semantics, not intermediate states that are merely compilation artifacts.

In this chapter we make the following contributions:

- a differential checking technique founded on a notion of *semantic consistency* for local variables (§5.2), where consistency is guided by an automated tool¹ built using symbolic execution techniques (§5.3)²
- experience showing that this check can correctly identify existing debug info issues (§5.4.1) and find valid new ones (§5.4.2)

5.1.1 Understanding local variable metadata

If a local variable³ is eliminated, it is not necessarily the case that it is absent from the debug illusion: some debug info formats can describe how to reconstruct it from state that remains, as we explored in Chapter 2. Which weakenings *must* happen is a function of the expressiveness of the debug info format. Since there is no correctness condition on compilers, any weakening *may* happen if the optimiser happens not to output the necessary metadata. Debug info metadata formats like DWARF [2017] are computationally expressive, in order to counter the complex code transformations of powerful optimisers. DWARF supports *expressions* in an interpreted stack machine language that the debugger can use to compute functions of program state. These functions compute either a variable’s *location* (when storage for the variable has been reserved in program state) or they compute a variable’s *value* directly, thereby preserving the

¹The prototype described in this dissertation does make some approximations (§5.3.3), and those necessitate manual filtering to remove false positives (§5.4.3). Future work that may resolve these issues is described at the end of the chapter (§5.5).

²The consistency check tool is available at <https://github.com/jryans/check-debug-info>.

³From here on, we consider formal parameters to be a kind of local variable.

```

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

```

Fig. 5.1: How DWARF might describe a local variable over four distinct address ranges within a function. Most expressions compute where it is *located*: in a register or (later) on the stack. Over the second range, however, it is not represented explicitly; its *value* is computed as a scaled difference of two registers. (The textual syntax is for illustration only. This figure previously appeared as Fig. 3.1. We have replicated it here for convenience.)

debugging illusion even for variables that are not explicitly represented at any storage location. Fig. 3.1 shows an example (which we have also replicated here as Fig. 5.1 for convenience).

5.1.2 Problem statement

Given the perspective of debug info *residualising* otherwise eliminated or simplified computation (§3.2.1), we can now state the specific problem we address in the remainder of the chapter:

Checking illusion consistency Local variable information should, at the very least, be *present*, and we discussed completeness metrics to check that in Chapter 3. How can we detect whether the local variable info is *correct*? To answer this, we need a systematic method for comparing the debug-time behaviour of a known-good, known-debuggable compilation against the alternative compilation under test. We define a property of *per-variable illusion consistency* relating local variables across a pair of executions of the same source program, e.g. as performed by an optimised and unoptimised compilation of the same code run under the same input, an approximation of which is testable using a form of symbolic execution (§5.3.1).

Inclusions and exclusions Full illusion coverage over local variables would make for debug-time abilities roughly “as good as printf debugging”: any in-scope variable could be correctly printed at any point. Currently, it is often remarked that printf is more reliable than debuggers [O’Callahan 2021] in part because variables can become unavailable to debuggers after optimisation. Roughly, the coverage and consistency illusion sub-properties (§2.4) capture how well this “gap to printf” has been closed. However, when using printf, optimisations are curtailed up to preservation of the printed variables (until after printing), whereas there is no such curtailment in effect in this scenario.

Print debugging requires recompilation every time you wish to change the variables or program points under investigation, which may add a considerable amount of time to the overall bug investigation process. Stepping debuggers and other program analysis tools using debug info do not necessitate additional build cycles before use.

Relationship to full semantic consistency Per-variable illusion consistency is a partial, “piecewise” form of semantic consistency, considering separately each individual small piece of program state (here, local variables). This contrasts with more general consistency properties definable only *across* many units of state; one example might be *snapshot consistency*, meaning that when debug-printing multiple variables, at some point the collection of values observed should be one also observable under a natural source semantics. As a shorthand, we refer to such *non*-piecewise consistency as “simulation” properties. This choice is pragmatic: as we will show, the piecewise approach offers a strong correctness criterion (albeit partial) that is achievable even in a “truthful” (§2.1) design, is feasible to check mechanically, and has bug-finding power. Of course, the flip side is that we overlook simulation-related compromises to the debug illusion. This includes the common experience of a stepping debugger “jumping around” between multiple source lines, deviating from program order, which is a problem we do not address.

5.2 Illusion consistency up to local variables

Here we consider how to define and test the correctness of the debug illusion for local variables. The best available oracle is the program before optimisation. This check consists of a tool taking as input two variants of the same program, each with its own debug info. We compare the illusion induced by debug info in the two programs. In practice these will be compiler intermediate representations (IR) before and after optimisation.⁴ Fig. 5.2 shows a high-level view of this approach.

What property is it reasonable to expect? This notion of consistency is defined *along pairs of paths*. By enumerating program inputs, we enumerate a set of pairs of paths: the same input drives each program variant down a path, according to the variant’s internal structure, but capturing the same portion of the program’s overall behaviour. The program variants may differ in a multitude of ways (due to transforms applied during optimisation), but we assume their observable input-output behaviour will remain the same.

The property we check can be thought of as placing a watchpoint on a single local variable at a time, and ensuring it progresses through the same sequence of values across both variants, and that changes in its value appear to occur at the same source program coordinates. This is intentionally without examining *other* variables at each such point (“simulation properties”; §5.1.2); instead we repeat the check for each variable in isolation. Fig. 5.3 shows an example of this variable value sequence property at the source level for loop variable *i*. This is a feasible correctness criterion for a “truthful” (§2.1) compiler—for example it allows great latitude to

⁴Instead of using unoptimised IR as presented in this chapter, one could also use a source language interpreter as the testing oracle, which would additionally enable testing of any potential transformations that may occur in the compiler’s front end stages before optimisation. In practice, debug-relevant compiler bugs are rare in the stages before optimisation, and using a separate program format for the unoptimised variant would add complexity at comparison time, so we have not gone down this road here.

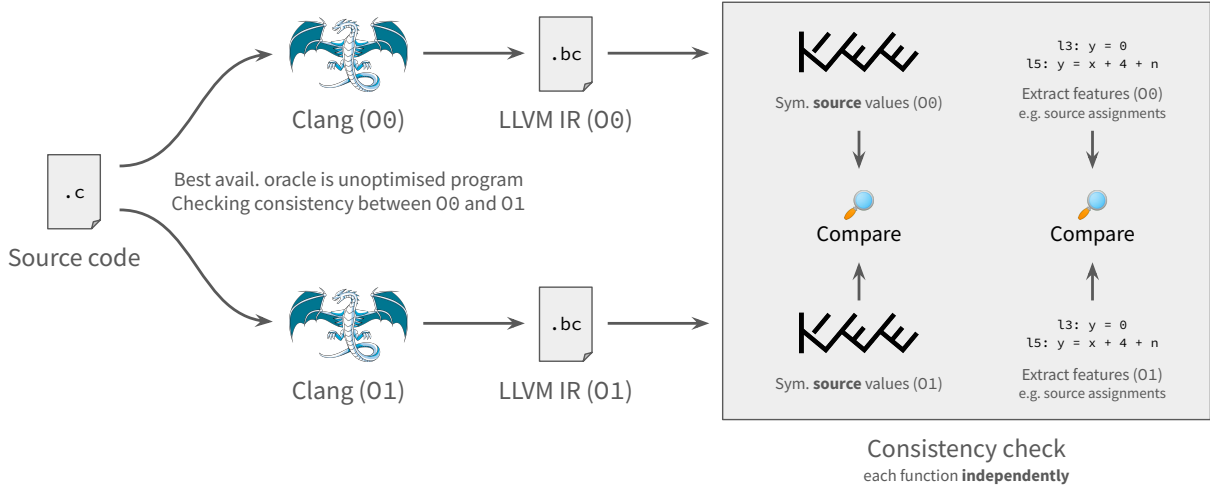


Fig. 5.2: The pipeline for illusion consistency checking. A C program is compiled by Clang into unoptimised and optimised variants as LLVM IR. These are then fed into a consistency check which explores the two variants using KLEE, collecting *source-level assignment operations* (§5.3.4.2). The operations occurring across the two variants are compared to detect inconsistencies in their debug illusions.

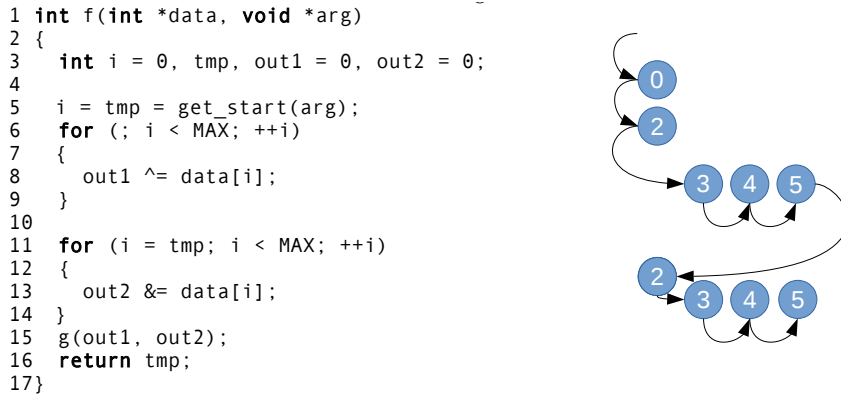


Fig. 5.3: Left: A simple example function in C. Right: The *source-level* concrete progression of variable *i*, for an execution where MAX is 5 and get_start() returns 2.

reorder and simplify code.

We have prototyped a bug-finding tool based on illusion consistency for finding bugs in the debug info for local variables. The core of the tool uses symbolic execution, and it works on LLVM IR, which in turn embeds a version of DWARF debug info. The following subsections describe how the tool is constructed and the challenges of applying illusion consistency to LLVM IR programs.

5.3 Implementing a tool

Constructing a tool to automate checking this property has involved a variety of challenges. The current implementation approximates several aspects of this checking task. These engineering choices and approximations are explained in further detail in the following sections.

5.3.1 Extending symbolic execution

We have implemented a tool that checks an approximate version (§5.3.3) of this illusion consistency property as a set of extensions to KLEE [Cadaru et al. 2008], a symbolic execution engine. KLEE’s symbolic execution is a good fit for several reasons. Firstly, it is an exploratory analysis which at all times characterises the explored path in terms of the input that leads down that path (in terms of a “path condition”, a formula over a set of symbolic input variables). This allows us to reason about state transitions occurring on “the same” paths through the code, without reference to the code’s internals—which the optimiser may have changed beyond recognition. Secondly, KLEE operates on LLVM IR which allows us to test LLVM-based compiler toolchains; it can be seen as a symbolic interpreter of LLVM IR. Thirdly, the symbolic formulae for IR variables enjoy a close correspondence with DWARF expressions and vice-versa; we use this to gain insight about whether a given DWARF expression is correct, and it may also allow reasoning about whether a source variable is knowable at all (§7.2.1).

KLEE is not a perfect fit as a base for this tool, so we have implemented some generic extensions. These extensions assisted us in the experiments described here and are also likely to be useful to the broader KLEE community.

5.3.1.1 Debug-awareness

We have extended KLEE to support mapping its symbolic LLVM-level state to source-level state by evaluating the stack-machine expressions in the debug info (§5.1.1). This also includes a translation from (a common subset of) DWARF expressions to SMT formulae, allowing a “symbolic source value” to be retrieved as an SMT formula in KLEE, just like an IR value, even though it has been “residualised out” of the base IR program and into debug info annotations.

For example, a symbolic array index that exists only residually in the debug info, computed from a symbolic pointer in the IR, can be retrieved as a formula. Fig. 5.4 and Fig. 5.5 show source and LLVM IR with debug annotations for this example of an array index which has been optimised out of the run-time program and residualised into debug annotations. The array index `i` in these figures is described by IR debug annotations that relate source-level assignment operations (§5.3.4.2) to SSA IR values. When compiling this down to native code, the value of `i` is optimised away and does not directly appear in program state. Debug info formats like DWARF can represent the value of `i` as function of the remaining run-time program state. The extended KLEE we describe here captures the value of `i` as an SMT formula in terms of the function’s inputs.

Stock KLEE does not use debug info at all; the features we added support may benefit KLEE users, e.g. enabling KLEE to report any of its findings in terms of both source coordinates and source variables.

```

1 void examine(unsigned *element);
2
3 void example(unsigned *array, unsigned size) {
4     for (unsigned i = 0; i != size; ++i)
5         examine(array + i);
6 }

```

Fig. 5.4: A simple C example that loops through an array argument, passing a pointer to each array element to another function defined elsewhere.

```

1 define void @example(i32* %array, i32 %size) {
2   entry:
3     @dbg.value(i32* %array, name: "array")
4     @dbg.value(i32 %size, name: "size")
5     @dbg.value(i32 0, name: "i")
6     %cmp.not4 = icmp eq i32 %size, 0, l: 4 c: 26
7     br i1 %cmp.not4, label %for.cond.cleanup, label %for.body.preheader, l: 4 c: 3
8
9   for.body.preheader:
10    %0 = zext i32 %size to i64, l: 4 c: 3
11    br label %for.body, l: 4 c: 3
12
13   for.body:
14    %indvars.iv = phi i64 [ 0, %for.body.preheader ], [ %indvars.iv.next, %for.body ]
15    @dbg.value(i64 %indvars.iv, name: "i")
16    %add.ptr = getelementptr inbounds i32, i32* %array, i64 %indvars.iv, l: 5 c: 19
17    call void @examine(i32* %add.ptr), l: 5 c: 5
18    %indvars.iv.next = add i64 %indvars.iv, 1, l: 4 c: 35
19    @dbg.value(i64 %indvars.iv.next, name: "i")
20    %cmp.not = icmp eq i64 %indvars.iv.next, %0, l: 4 c: 26
21    br i1 %cmp.not, label %for.cond.cleanup, label %for.body, l: 4 c: 3
22
23   for.cond.cleanup:
24     ret void, l: 6 c: 1
25 }

```

Fig. 5.5: LLVM IR for the example function from Fig. 5.4. Line 15 shows a debug annotation that represents the source variable `i` in terms of the SSA IR value `indvars.iv`, which is incremented each time through the loop body. When compiling this down to native code, the value of `i` is optimised away. Debug info formats like DWARF can represent the value of `i` in terms of other program state. The tool we describe here uses these debug annotations to discover SMT formulae for each source-level assignment operation (§5.3.4.2) during symbolic execution.

5.3.1.2 Abstract identity for heap allocation

When heap allocation occurs in KLEE, each distinct allocation context is modelled as a “memory object” which, in different states of exploration, can store many distinct values (“object states”). In mainline KLEE, the identity of an allocated object is represented simply using a pointer to a structure in the host `malloc` heap. This representation may be used in symbolic formulae, for example when assigning to a local pointer variable. However, since this identity may change from run to run, it does not allow us to correlate source-level allocation contexts across different runs of the program, or in this case runs of the different *variants*. Therefore, we generate a more abstract symbolic name for each allocation, using features of the source-level context of the allocation. For example, to identify successive allocations made for source variable `bob`, we may generate the names `bob_0`, `bob_1`, etc. This allows us to reason whether the value assigned to a pointer local at some source program point really is “the same allocation” across the two variants. In typical uses of symbolic execution, there is no need to perform this correlation, so an abstract identifier is not needed. (We suspect it might still prove useful, e.g. to understand formulae describing pointer values, or test cases generated using these.)

5.3.1.3 Function-wise operation

To allow high-coverage exploration of compiler outputs, which need not be complete programs, we implemented a “function-wise” mode in KLEE. This treats each function as if it were the program entry point, considers all function inputs as symbolic (including from memory objects not yet seen), and does not descend into other functions. This function-wise mode may also be useful to others building on top of KLEE. Other researchers have described similar partial-program symbolic execution modes [[Ramos and Engler 2015](#); [Barrett 2021](#)], though so far such features have not yet appeared in upstream KLEE.

Unlike typical applications of symbolic execution, for us it is not a concern whether the explored paths are whole-program-feasible. In effect, by using function-wise exploration we pursue “many small explorations”, each involving simpler constraints and shorter paths (from exploring a smaller state space) than a single big whole-program exploration. This mimics the static analysis performed by compilers during optimisation, but retains the concrete identification of paths appropriate to the differential approach. It is also, of course, underapproximate because we may not explore all paths; while this would not be appropriate in a compiler, it is appropriate for a tool intended to find inconsistencies (albeit incompletely). Below ([§5.3.3](#)) we discuss some additional approximations introduced by the actual tool.

5.3.1.4 Analysis window

We use an “analysis window” to control which function calls are explored and which are skipped. By default, only a single program function is inside the analysis window and all other calls are skipped. For each skipped function call, any non-constant object that is either global

```

1 int example(int n) {
2     int x = n * 2;
3     int y = 0;
4     for (unsigned int i = 0; i < n; i++) {
5         y += x + 4 + n;
6     }
7     return y;
8 }

```

Fig. 5.6: A simple C example with a loop and several variables.

or address-taken is (conservatively) reset to an unconstrained state. An address-taken local is thus modelled as a new symbolic variable after a function call. Any further modifications to that local in the function being checked will use formulae expressed in terms of that new symbolic variable. The return value of the skipped call is similarly represented by a new symbolic variable as well. “Adding symbolicness” in this way models the unknown effect of these calls to functions outside the analysis window. In the optimised variant, the analysis window used in function-wise exploration cannot be strictly limited to a single *source* function, since some calls may have been inlined. The analysis we perform can tolerate this (see §5.3.3).

5.3.1.5 Memory model

Function-wise mode bears some similarity to the *generalized symbolic execution* of [Khurshid et al. \[2003\]](#). That technique also models pre-existing heap environments, such that dereferencing a symbolic pointer forks not only one state per possibly-pointed-to “seen” object but *also* one state for a fresh, unconstrained (“previously unseen”) object.

For the experimental results presented in this chapter, we were able to make progress with a simpler model of the heap. Each pointer-type function argument is set to a concrete address pointing to a fresh, unconstrained symbolic object. This does underapproximate the heap, and may miss portions of control flow where e.g. two pointer-type arguments are checked for equality, as this approach assumes pointer arguments never alias. We prototyped initial work towards a more accurate memory model which is described later (§5.5).

5.3.2 Consistency checking in practice

The goal is to detect inconsistencies, including omissions, in the debug info that is present after optimisation, relative to that before. A high-level summary is as follows:

1. Compile the test program to LLVM IR with and without optimisation.
2. Enumerate, separately for each local source variable, the static IR-level *source-level assignment operations* that affect it. As we will see (§5.3.4.2), assignment operations may take many forms in an IR program.

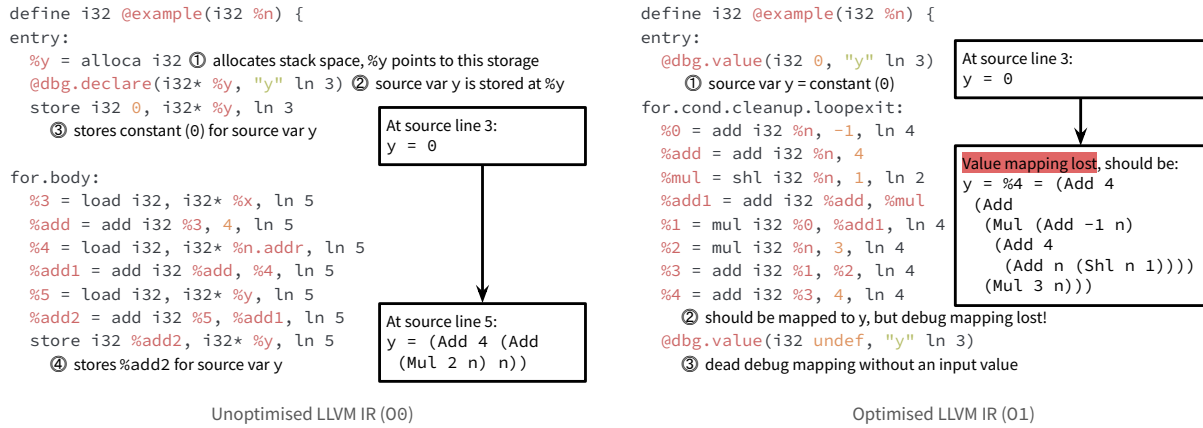


Fig. 5.7: A worked example shows the consistency check in more detail across unoptimised and optimised LLVM IR for the example function in Fig. 5.6. Source variable `y` is assigned on source lines 3 and 5. Using these source coordinates to match assignments across the two program versions, this approach finds that the assignment on line 5 in the optimised version was lost (undef should have been `%4`), and this indicates a bug in the optimiser’s debug info handling. The unoptimised and optimised formula pairs shown are equivalent, as we discuss in more detail (§5.3.4.7).

3. To identify where assignments may be lost in the induced debug illusion, statically compute a pairwise matching of these source-level assignment operations in the two compilations of the program. Roughly, assignments are paired if they are “the same operation” as identified by their source coordinates in debug info. We expect all assignments to have at least one match, but the matching is generally not one-to-one (e.g. if optimisation splits or merges paths). Unmatched assignments are flagged as an error, as it denotes a potential problem in the compiler’s variable location handling (false positives are possible when CFG transformations occur, as we discuss further in §5.3.3).
4. To identify where *assigned values* may be incorrect in the induced debug illusion, use KLEE with extensions (§5.3.1) to dynamically explore paths through the function and collect formulae for the IR values “assigned” at each identified assignment operation. For each source variable, for each pair of matched assignments to that variable, we use the SMT solver to check that the assigned formulae are equivalent.

5.3.3 Current limitations

Inlining In the optimised variant, some calls may have been inlined. This inlining does not pose a problem for this analysis, in the sense that the local variables of these inlined callees can be straightforwardly filtered out—the debug info cross-references their origin to the inlined callee. (Inlining’s main effect is to slightly reduce the extent of symbolic overapproximation, since it replaces function calls that require “adding symbolicness” with relatively concrete actual code by moving those functions inside the current analysis window.) However, it does mean that the debug illusion *within an inlined callee* is not currently tested; rather we would only be able to test its out-of-line copy.

```

1 int if_conversion(int n) {
2     int r;
3     if (n) {
4         r = n + 1;
5     } else {
6         r = n - 1;
7     }
8     return r;
9 }

```

Fig. 5.8: An example C function with an “if-else” conditional block which compilers can convert into a single branchless select instruction that covers both paths. This example also appeared earlier in [Fig. 1.4](#).

Approximations The tool precedes symbolic execution with a flow-insensitive *matching pass* (§5.3.2, step 3) which is effectively a static pre-filter to quickly detect consistency failures where a reachable assignment in one program variant has no possible analogue in the other. Assignments can be associated with an unambiguous source variable (or no source variable) and matched by their source coordinate information. However, it is vulnerable to false positives when there is dead code in the unoptimised program. Dead code (when present) might be symbolically explored, thus recording additional source-level assignment operations into the assignment sets for any encountered variables, and those assignment operations would then not be found when exploring the optimised program variant. To address simple cases of dead code, we currently use a basic local analysis to detect obviously dead code, in which assignments should be ignored; this could clearly be improved e.g. by using LLVM’s own dead code elimination passes.

Exploration is terminated when each statically identified assignment has been visited once in both variants *over any path*, rather than attempting to explore all paths. This approach of static assignment matching via source coordinates together with visiting basic blocks only once is also vulnerable to false positives when CFG transformations change the paths taken when comparing the two program versions. For example, if-conversion may transform a control flow diamond with two paths (one for the “if” branch and another for the “else” branch) into a single block, such as the one in [Fig. 5.8](#). Since we visit each basic block only once, we may encounter more source-level assignment operations for each source variable in the pre-if-conversion program (because there is an additional block to visit) than we find in the post-if-conversion program. The static assignment matching process would then report false positives, as it views the mismatched assignment count as a signal that debug info has been lost, when really this is an artifact of the exploration approach.

We discuss false positives that may result from these approximations later (§5.4.3). We also sketch initial work towards a more dynamic approach that avoids these approximations towards the end of this chapter (§5.5).

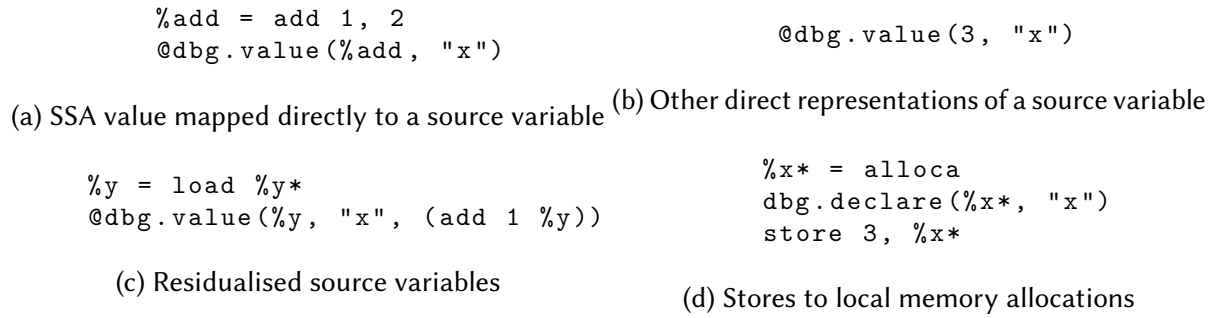


Fig. 5.9: Examples of source-level assignment operation scenarios in an SSA IR with debugging annotations.

5.3.4 Consistency checking step by step

The following sections give more details of each step in the consistency checking process as we have currently implemented it.

5.3.4.1 Compile to LLVM IR

We produce program variants by compilation of the same program firstly without optimisation (in the current implementation, LLVM IR emitted by Clang with 00) and secondly with optimisation (01 or higher).

5.3.4.2 Collect source-level assignment operations

Here we identify IR instructions that change source-level local variables which we refer to as *source-level assignment operations*. In the context of an static single assignment (SSA) IR with debugging annotations, such changes can occur in multiple ways as shown in [Fig. 5.9](#).

New SSA value, mapped directly to a source variable In SSA, only a freshly introduced local may be assigned to. If such a new IR local directly models a source variable, the IR should follow it with an “intrinsic” recording the mapping back to source; in LLVM this is a `dbg.value` intrinsic. Although the SSA value is necessarily new, these intrinsics do not distinguish whether the *source* variable is new; it may simply be “moving” to the new SSA value. In real IR, debug intrinsics are often repeated, so need not correspond to source-level assignment operations—they are as if to say “`x = x`”. This is especially common across phi nodes. These “redundant” assignments are filtered out after symbolic execution by comparing values between source-level assignment operations for each variable in the order in which we encountered them.

Binding change, without a new SSA value Conversely, a new mapping need not follow a new SSA value. For example, given source `y = x`, in IR the same SSA value previously modelling `x` might simply now be mapped to `y` by a “standalone” `dbg.value`. We also identify these binding changes as assignments.

Stores to local memory allocations Stores to memory may update local variables residing on the stack. In LLVM, the `dbg.declare` intrinsic states that the value stored at an *address* is the value of a given source variable; for locals, the address was produced by an earlier `alloca`. *Store-like* operations include `store` instructions as well as memory-writing intrinsics such as `memcpy`. Any store-like operation to a `dbg.declare`-mapped local's address is considered an assignment to the source variable. `dbg.value` intrinsics can also optionally describe an *address* instead of a value when `DW_OP_deref` is present in their value expression. We additionally track store-like operations to addresses referenced by for these “implicit” `dbg.value`-with-`DW_OP_deref` intrinsics.

Residualised source variables In “residualised” cases, the IR program does not represent a given source variable directly, but the debugging information records it as a function of other variables (§5.1.1) or (degenerately) a constant literal value. The `dbg.value` intrinsic supplies these expressions; complex expressions are referenced as IR nodes reachable only from debug intrinsics and which ultimately are residualised as DWARF expressions rather than object code.

For each source variable in the program, we build up a *list* (some parts of this approach do make use of order) of the source-level assignment operations indicated by these debug intrinsics (store-like operations for `dbg.declare` and the intrinsics themselves for `dbg.value`).

Fig. 5.7 shows an example of the identified LLVM debug intrinsics in both program versions. We will return to this example as we explain the later steps.

5.3.4.3 Static matching of assignments

For each source variable, we check that the set of unoptimised source-level assignment operations is contained within the set of optimised source-level assignment operations, matched by their source coordinates. While some optimisations (e.g. removing unreachable code) may legitimately reduce source coordinate coverage, in general we expect the optimised version to have assignments which cover the same lines as the unoptimised version. This does have some risk of false positives (§5.3.3).

The resulting pairwise matching of assignments is in general a many-to-many relation, since the unoptimised and optimised versions may each have a different number of source-level assignment operations for a given source variable, debug annotations may be duplicated, and so on.

Fig. 5.7 shows the unoptimised version assigns to source variable *y* on source lines 3 and 5. The optimised version there does indicate an assignment on line 3, but later `dbg.value` intrinsic has an input of `undef` which indicates the state of source variable *y* is no longer known. Since there is no second assignment for source line 5, we report a missing assignment for this optimised program.

5.3.4.4 Collect symbolic values

We now run each program version using the extended function-wise KLEE (§5.3.1). For each function in the program, we define the current analysis window to contain only that function while we explore it. All inputs to the current function are treated symbolically, including loads from non-local memory and values produced by calls to functions outside the analysis window (via both address-taken locals passed to the call as well as the function’s return value). KLEE builds up SMT formulae for each value in the IR program during execution. When KLEE encounters a previously identified source-level assignment operation in the IR, we save the formula for the value assigned. We also record the order in which KLEE first encountered each assignment operation via a per-variable sequential operation counter. This extended KLEE (§5.3.1) can capture SMT formulae equally well for both IR-level “bona fide” source-level assignment operations (stores or new SSA values) and debug info “residualised” ones (e.g. DWARF expressions).

As an approximation, exploration is terminated when each statically identified assignment has been visited once in both variants *over any path*, rather than attempting to explore all paths.

5.3.4.5 Filter away redundant assignments

Debug info intrinsics in LLVM IR sometimes restate the same the value for a variable that was already present. In particular, phi nodes used to join values from multiple incoming blocks in SSA form are a common source of such “redundant” intrinsics which effectively restate the value of a source variable via a new SSA value in the current block, even though the concrete value of source variable has not actually changed.

This analysis compares each variable’s sequence of source-level assignment operations across program versions. For a given source variable to be deemed consistent, we expect those assignment operations to occur in the same sequence of steps in both versions. For these reasons, any redundant operations found during execution must be eliminated before further analysis can proceed.

Symbolic execution had gathered SMT formulae for each encountered assignment operation that each source variable proceeded through. We loop through the assignments for each source variable (in the order in which they were encountered) and compare the current assignment operation with the previous operation by checking whether the value formulae are equivalent. If they are, we consider the current assignment operation to be redundant and remove it from the list of source-level assignment operations for that variable.

5.3.4.6 Check for lost assignments

After filtering away redundant assignments, we then match source-level assignment operations for each source variable pairwise across the program versions. The source coordinates of the

source-level assignment operations are used to match up the pairs.

This allows us to identify where assignments may be lost in the induced debug illusion. We expect all assignments to have a one-to-one match across program versions. If e.g. the optimised version has fewer source-level assignment operations, it indicates that debug info was lost during optimisation. If there is a different number of source-level assignment operations across versions, we flag an assignment count inconsistency. Note that assignments in statically eliminated dead code will by definition never be explored, so these do not cause false positives.

In Fig. 5.7, we see that for source variable y , there are two source-level assignment operations in the unoptimised program and only one assignment operation in optimised program (the undef input to the second `dbg.value` intrinsic in the optimised program effectively “terminates” the previous operation and does not create a second operation itself). The missing second operation in optimised version is due to a compiler bug dropping source value information at this point in the program. Since there is a mismatch in assignment count, we report an assignment count inconsistency for this example.

5.3.4.7 Check for consistent values

We then check the symbolic formulae that KLEE collected for these paired assignments (one unoptimised and one optimised) are equivalent by asking the SMT solver to prove ($\text{Eq } \langle \text{Unopt formula} \rangle \langle \text{Opt formula} \rangle$). If this fails, we flag a value inconsistency.

We turn our attention to the example in Fig. 5.7 one more time. For the single pair of matched assignments (source line 3), KLEE collected the constant 0 for both the unoptimised and optimised versions, which is trivially equivalent. For the second assignment on source line 5, there’s technically nothing to compare in this example, since we don’t have a matched pair of source-level assignment operations (due to a compiler bug dropping that info during optimisation). Using an SMT solver manually on this example (in order to show what this step would have done had the compiler preserved the debug info as we expect), we find that the unoptimised formula

$$(\text{Add } 4 \ (\text{Add } (\text{Mul } 2 \ n) \ n))$$

is equivalent to the optimised formula

$$(\text{Add } 4 \ (\text{Add } (\text{Mul } (\text{Add } -1 \ n) \ (\text{Add } 4 \ (\text{Add } n \ (\text{Shl } n \ 1)))) \ (\text{Mul } 3 \ n)))$$

for the previous SSA value %4. It appears the compiler has failed to keep track of the connection between the SSA value %4 and the source variable y in this case, despite this being clearly representable in debug info (*knowable*) at debug time. For assignments where we do have values to check from each program version, we report a source value error when they are not equivalent.

<pre> 1 int example(int n) { 2 int x = 1; 3 int y = 2; 4 return x + y + n; 5 } </pre>	<pre> define i32 @example(i32 %n) { @dbg.value(%n, name: n, ln: 1) @dbg.value(1, name: x, ln: 2) @dbg.value(2, name: y, ln: 3) %add = add 1, 2, ln: 4 %add1 = add %add, %n, ln: 4 ret %add1, ln: 4 } </pre>
(a) Original C source	(b) Unoptimised LLVM IR

Fig. 5.10: Initial program before optimisation as original source and abbreviated LLVM IR

<pre> - @dbg.value(1, name: x, ln: 2) - @dbg.value(2, name: y, ln: 3) + @dbg.value(1, name: x, ln: 3) + @dbg.value(2, name: y, ln: 2) </pre>	<pre> - @dbg.value(2, name: y, ln: 3) + @dbg.value(4, name: y, ln: 3) </pre>
(a) Corrupted source coordinates	(b) Corrupted source value
<pre> @dbg.value(2, name: y, ln: 3) + @dbg.value(2, name: y, ln: 4) </pre>	<pre> - @dbg.value(2, name: y, ln: 3) </pre>
(c) Redundant value added	(d) Value removed

Fig. 5.11: Various transformations of program from Fig. 5.10

5.3.5 Examples

In this section, we discuss a few example IR transformations that represent synthetic bugs and show how this approach fares for each one. To save space, the examples are presented as snippets of abbreviated LLVM IR with various unimportant attributes elided and debug info inlined. Fig. 5.10 shows the “starting point” for each example as both original C source and unoptimised⁵ IR. Each example will make a small transformation to the IR and work through how the tool would behave in that case.

In each example, added lines are displayed in **green text** and removed lines are displayed in **red text**. Lines of form `@dbg.value(%add, name: y, ln: 3)` are LLVM IR debug info intrinsics which link an SSA value or constant to a source variable at that point in the program, such as “the value of source variable *y* on source line 3 is the SSA value `%add`”.

Inconsistent source coordinates Fig. 5.11a shows a transformation which has flipped the source coordinates for the variables *x* and *y* as if the assignments occurred on different lines. We compare the source-level assignment operations across program versions to ensure they cover the same source lines, so this reveals the mismatched source coordinates here. The tool

⁵For the “unoptimised” version presented here, we have actually run one optimisation pass: the memory to (SSA) register (`mem2reg`) pass which (as a side effect) transforms the initial LLVM IR `dbg.declare` intrinsics into `dbg.value` intrinsics. This is done purely to keep the examples shorter and easier to read. The tool will accept either version.

reports this inconsistency successfully.

Inconsistent values Fig. 5.11b shows a transformation which changed the value of the source variable y from 2 to 4. We collect the values of all source-level assignment operations via symbolic execution and then compare them to their corresponding assignments in the other program version. The assignments describe the same source variable with same source coordinates, so we then compare their values. The tool reports this value inconsistency successfully.

Redundant values Fig. 5.11c shows a transformation which added an additional value intrinsic. The original IR states that source variable y is assigned the value 2 on source line 3. The added value intrinsic states that y also has the same value (2) on source line 4 as well. This is redundant information and results in the same expected debugging experience as the original IR when compiled. When we collect source-level assignment operations for comparison, we remove redundant operations such as this. The tool thus correctly reports these programs as consistent.

Removed values Fig. 5.11d shows a transformation which removes the value intrinsic that assigned a value to source variable y . A debugger will no longer be able to display the value of y in this function. We compare source-level assignment operations across program versions after symbolic execution. Since the unoptimised program has an assignment of source variable y on source line 3, we expect to find some assignment for y in the optimised program as well, but there no longer is one. The tool reports this missing debug info successfully.

5.4 Evaluation

Here we describe some practical experiences so far using this consistency check, in support of the following hypothesis:

Hypothesis. *The consistency check approach is a correctness criterion for local variable debug info, is feasible to check mechanically⁶, and has bug-finding power.*

The primary approach shown here is by producing and/or reproducing bug reports as confirmed by real compiler developers.

A key benefit of this approach, which cannot meaningfully be measured, is that unlike existing approaches to checking correctness of debugging information, we have no need to annotate the expected variable values [Bedwell 2018]. Unlike manual annotation, this approach requires only test programs as input, so it can be automated to explore a large corpus of test

⁶Mechanically feasible modulo acknowledged approximations (§5.3.3) in the prototype described by this dissertation that necessitate some manual filtering (§5.4.3).

Table 5.1: Summary of existing issues explored by pass

Pass	Issues
EarlyCSE	1
InstCombine	2
LCSSA	1
LoopStrengthReduce	1
LoopVectorize	1
Reassociate	1
SimplifyCFG	6
Total	13

programs. These in turn can be generated randomly (e.g. by Csmith [Yang et al. 2011] or YARPGen [Livinskii et al. 2020]). We can also repurpose existing compiler test cases to test debug info, in effect extending the functional correctness to a “debug-correctness” test, again without requiring annotation effort.

5.4.1 Confirming existing issues

To validate this approach, we gathered previously filed debug info issues and attempted to detect the original issue via the consistency check. For issues that were resolved, we also tried to verify the resolution as well. In 2019, we surveyed debug-info-related issues from the LLVM project which were part of a larger issue grouping for “poor debug experiences” as determined by the community. At the time of that survey, there were 51 such issues, and only 20 of those had associated fixes that we could attempt to verify. We filtered further to only issues in optimisation passes (as that is the portion of the compiler pipeline our approach investigates), leaving us with a sample of 13 issues. We worked down the list of matching issues ordered by ascending issue ID. Of these, 11 issues were resolved and 2 were open at the time of this analysis.

If we were to repeat this issue survey now in 2026, we would find many more issues to analyze, but investigating each one requires a non-trivial amount of manual effort interpreting tool output, so we have stuck with our earlier issue sample for now. (Future work in this direction should improve tooling efficiency, thus reducing the effort required to analyse additional issues.)

Tab. 5.1 shows these issues span a variety of optimisation passes. Tab. 5.2 gives more detail on each issue. For each issue, we successfully detected the original debug info failure using this approach. For the subset of issues with fixes, we further verified that this approach confirms the fix altered debug info in the intended way and the consistency check described in this chapter now passes. This gives us confidence that this approach detects debug issues of value to compiler developers.

Table 5.2: Details of existing issues explored. For each issue, we successfully detected the original debug info failure using this approach. For the subset of issues with fixes, we further verified that this approach confirms the fix altered debug info in the intended way.

ID	Summary	Category
38101	Stale debug variable value in optimised code	Variable
38110	Unreachable line stepped onto after speculating	Location
38111	Speculated BB presents illegal variable value to debugger	Variable
38163	Loop strength reduction preserves little debug info	Variable
38366	Vectorized loop presents constant-valued iterator to debugger	Variable
38367	Return value unprintable in func with vectorized loop	Variable
38535	Unreachable line stepped onto	Location
38591	Regression: Unreachable line stepped onto after SimplifyCFG	Location
38687	Misleading value for local variable at the return statement	Variable
39133	InstCombine pointer type-cast sinking reduces location ranges	Variable
39974	Salvaged memory loads can observe subsequent memory writes	Variable
41226	SimplifyCFG sink splitting drops variable info	Variable
43238	InstCombine leaves incorrect debug value when removing code	Variable

Table 5.3: Summary of new issues discovered by pass

Pass	Issues
EarlyCSE	1
IndVarSimplify	1
InstCombine	4
IPSCCP	1
Reassociate	2
Total	9

5.4.2 Finding new issues

We next sought to discover new debug info issues by applying the consistency check approach to a variety of input programs, including generated programs from tools like Csmith [Yang et al. 2011], production code bases for Git and LLVM, as well as programs hand-written by the authors.

We focused this evaluation on Clang 13, the latest stable version at the time we began this work. Problems identified by the tool were manually triaged to avoid creating duplicate issue reports. Before submitting any potential new issues upstream to LLVM, we retested them on the latest main branch and minimised code samples using C-Reduce [Regehr et al. 2012]. The optimisation pipeline feature [Rifkin 2022] of Compiler Explorer [Godbolt and contributors 2012] was used to narrow down the pass where debug info was lost in each case. We used Whole-program LLVM (WLLVM) [Ravitch 2011] to collect LLVM bitcode for entire codebases.

Tab. 5.3 shows the new issues we discovered come from a variety of different passes across the optimisation pipeline. Tab. 5.4 gives more detail on each issue. We have discovered a total of 9 debug info issues in LLVM so far. Of those, we have filed 8 with the LLVM project, and the community has confirmed 7 as valid issues. The 1 remaining filed issue is currently awaiting community confirmation. The 1 other issue we found in LLVM 13 was fixed coincidentally (not

Table 5.4: Details of new issues discovered. For each issue, we report its current status, e.g. whether it has been newly filed, confirmed by others, or fixed.

ID	Summary	Status
N/A	IPSCCP removes global array value debug info	Fixed in 15.x
60640	IndVarSimplify loses track of debug value from loop body	Confirmed
60532	Reassociate drops debug value info for decremented variable	Fixed in 23.x
61153	InstCombine drops debug value info towards end of function	Confirmed
61169	EarlyCSE drops debug info for pointer values derived from arguments	Confirmed
61261	InstCombine drops debug value info for start of live range	Confirmed
61266	InstCombine drops debug value after simplifying to boolean	Confirmed
61272	Reassociate stops tracking debug value after several steps	Fixed in 23.x
62051	InstCombine drops debug value when sinking select	Filed

following the filed report) in LLVM 15. The LLVM community’s confirmation of these issues gives us confidence that this approach finds real issues that are insightful for compiler authors.

5.4.3 False positives

As hinted earlier (§5.3.3), the current implementation of this consistency check makes various approximations that can lead to false positives. During the experiments described above (confirming existing consistency issues and finding new ones), manual review of the tool output was often needed to double-check the deductions made. With that in mind, the current tool implementation may be challenging for others to use confidently. Since these false positives stem from approximations made during tool building (and not from the consistency property as a concept), we believe these shortcomings can be addressed through additional tool engineering work. We outline further enhancements and improvements in later sections (§5.5) that should resolve these false positives.

CFG transformations (such as if-conversion) that change the set of paths between program versions can lead to false positives with the current tool implementation. Fig. 5.8 shows an example C function that goes through this kind of CFG transformation when comparing LLVM IR from early and late stages of the optimisation pipeline (Fig. 5.12, Fig. 5.13). With the current tool implementation, each basic block is explored only once. For variable `r`, the tool finds 3 source-level assignment operations in the early-stage program variant (Fig. 5.12), whereas it finds only 1 source-level assignment operation in the late-stage program variant (Fig. 5.13). The tool reports this as an error because the count of source-level assignment operations does not match across the two program variants. Future work using dynamic exploration (§5.5) should help to resolve the false positive in this example.

```

1  define i32 @if_conversion(i32 %n) {
2  entry:
3      @dbg.value(i32 %n, name: "n")
4      %tobool = icmp ne i32 %n, 0, l: 3 c: 7
5      br i1 %tobool, label %if.then, label %if.else, l: 3 c: 7
6
7  if.then:
8      %add = add nsw i32 %n, 1, l: 4 c: 11
9      @dbg.value(i32 %add, name: "r")
10     br label %if.end, l: 5 c: 3
11
12  if.else:
13     %sub = sub nsw i32 %n, 1, l: 6 c: 11
14     @dbg.value(i32 %sub, name: "r")
15     br label %if.end
16
17  if.end:
18     %r.0 = phi i32 [ %add, %if.then ], [ %sub, %if.else ]
19     @dbg.value(i32 %r.0, name: "r")
20     ret i32 %r.0, l: 8 c: 3
21 }

```

Fig. 5.12: LLVM 13 IR early in the optimisation pipeline for the function in [Fig. 5.8](#). There are separate blocks for the “if” and “else” conditional paths.

```

1  define i32 @if_conversion(i32 %n) {
2  entry:
3      @dbg.value(i32 %n, name: "n")
4      %tobool.not = icmp eq i32 %n, 0, l: 3 c: 7
5      %add = add nsw i32 %n, 1, l: 3 c: 7
6      %r.0 = select i1 %tobool.not, i32 -1, i32 %add, l: 3 c: 7
7      @dbg.value(i32 %r.0, name: "r")
8      ret i32 %r.0, l: 8 c: 3
9  }

```

Fig. 5.13: LLVM 13 IR later in the optimisation pipeline for the function in [Fig. 5.8](#). There is now only a single block, and the conditional paths have been transformed into a branchless `select` instruction.

5.5 Conclusions and future work

We have defined various properties of a correct debug illusion ([Chapter 2](#)), characterising debug info as a “residualisation” of the optimised-away parts of the original program. We have elaborated these properties into correctness criteria that we have embodied in implemented tools and found useful in finding and explaining real debuggability problems. The residualisation perspective suggests future work that applies to this entire thesis. We have collected these thesis-wide project ideas in [§7.2](#). The future work described below is specific to symbolic execution and seeks to address various limitations of the experimental tool described in this chapter.

Lazy object generation, improved memory model As discussed earlier ([§5.3.1.5](#)), the experimental results in this chapter made use of a simplified, underapproximate memory model. All pointer-type function arguments are set to distinct concrete addresses that point to unconstrained symbolic values. This enabled progress on simple examples without fear of path explosion due to address space searches (which can occur when pointer values are symbolic).

However, this simpler model is clearly underapproximate and leads to missed coverage in functions that e.g. compare two pointer-type function arguments for equality, as this approach gives each pointer a unique address. To resolve these issues, we also started work on a lazy object generation approach that more closely aligns with that of [Khurshid et al. \[2003\]](#). In this improved memory model, we lazily create multiple states for each pointer-type variable when it is first accessed during symbolic exploration. Each pointer is modelled as being any of:

1. the null pointer
2. a pointer to an unconstrained symbolic object
3. a pointer to an existing object of the expected type

This more accurate memory model enables improved function coverage for control flow that checks for such cases. We have partially implemented this memory model variant, but more work is needed to ensure it is performant and handles all edge cases correctly.

Dynamic exploration As mentioned in the approximations above ([§5.3.3](#)), we used static matching to pair off source-level assignment operations from each program version via their source coordinates information and only visited each basic block once. This can produce false positives when CFG transformations change the number of blocks, as it may mean we gather a different count of source-level assignment operations and appear to detect errors due to that mismatch (when in reality the debug info may actually be correct).

To resolve these issues, we also prototyped a more dynamic exploration approach within each function that visits blocks as many times as needed. Rather than using source coordinates

to match source-level assignment operations across program versions, we tracked the order in which encountered each assignment operation, as the order should be the same across program versions (once any redundancies have been filtered away). We partially implemented this variant, but further work is still needed to control the path explosions resulting from the larger symbolic execution search space.

Subset of language features Some source language features, such as loops, add complexity to symbolic execution analysis and can make it harder to achieve a complete view of variable assignment operations across all paths in a given function.

Future work could choose to avoid these challenges by selecting e.g. only loop-free functions for analysis. While this would implicitly test fewer compiler optimisations (as loop optimisations would not apply to the programs analysed), it should still reveal interesting debug info handling bugs while also simplifying the symbolic analysis process and associated tool implementation.

Chapter 6

Related work

6.1 Early debugging techniques

Debugging approaches in one form or another have been around since the early days of computing in the 1950s [[Satterthwaite 1975](#)]. These early techniques were very basic, giving only simple state dumps triggered at breakpoint locations and were often ad-hoc, proprietary, and undocumented. Early debugging abilities were embedded in the surrounding compiler or operating system (rather than being standalone debugging tools) and typically focused on the “base case” of debugging the executed program without considering how it may have been transformed by optimisations.

While some might assume interactive debuggers and debug info metadata formats are a recent construction, early precursors (such as symbol listings that map variables to address and specialised debugging programs to run together with your programs under analysis) were already around by the time of the PDP series [[Digital Equipment Corporation 1967](#); [Evans and Darley 1966](#)]. These environments supported basic features like breakpoints and viewing values at addresses, but mapping the machine-level view back to program text was left up to the debugging operator.

The debugging systems for higher-level languages are much more complex than those for assembly code. They must locate the symbol table, find the beginning and end of source-level statements, and determine some way to extract the dynamic information needed for debugging – about the program’s behavior, which is now hidden in a sequence of machine instructions rather than being the obvious result of one machine instruction. [[Balzer 1969](#)]

Early Lisp systems supported a higher-level approach [[Evans and Darley 1966](#)], allowing the user to to set breakpoints and patch the programs at the Lisp source level. The also offered support of checking which function you are currently within, a precursor of the modern debugger’s backtrace. Lisp systems often included both an interpreter and compiler. The

availability of an interpreter simplified the implementation of these higher-level debugging features compared to other language implementations of the time that often offered only a compiler.

6.2 Approaches to debugging optimised code

In the 1980s, we start to see efforts towards some level of debugging experience for optimised programs, but as we explain below, they don't quite fulfill the ideal debugging vision and each has different trade-offs.

“Expected” versus “truthful” Prior work [Zellweger 1984] has identified two categories of illusion: *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” illusions are characteristic of 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. 2023a]. In this way, full-speed debuggable execution really is feasible, at a cost of often truthfully showing the unexpected. Other approaches often seek to more closely approximate the level of optimisation allowed by the truthful approach by inhibiting optimisation only “on demand” and “just enough” [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 [Zurawski and Johnson 1991; Hölzle et al. 1992; Guo 2014; D'Elia and Demetrescu 2018].

A debugger can provide expected behavior if it can recover the source-level, unoptimized program state (current execution locations, variable values, and procedure traceback) from the low-level, optimized state whenever it is invoked. The compiler computes the set of *inspection points*, which are the points in the program where the debugger might be invoked. For each inspection point, the compiler computes *recovery information*, which describes the effects of optimization at that inspection point. Recovery information lets the debugger recover the values that the variables would have had in an unoptimized program. [Zurawski and Johnson 1991]

Zurawski and Johnson [1991] describe (in an unpublished draft) one of the earliest examples using this approach selecting transition to unoptimised code to achieve expected behaviour via their *context-oriented model* of debugging which tracks the source-level state for each stack

frame. The compiler finds inspection points where a debugger might want to attach and only optimises between those points, so there is some loss of optimisation.

Hölzle et al. [1992] is the first published work with this style of debugging. Their work is set in the context of the Self language and restricts optimisation across interrupt points, similar to Zurawski and Johnson. The implementation from Hölzle et al. *lazily* deoptimises (waiting until control returns to a frame), which simplifies their implementation and spreads deoptimisation work over time, avoiding potentially distracting pauses.

Guo [2014] shares details of dynamic deoptimisation for debugging in the context of SpiderMonkey (Firefox’s JavaScript engine) with multiple JITs to navigate. D’Elia and Demetrescu [2018] explore on-stack replacement at arbitrary program points and investigate the feasibility of recovering variable values using their techniques.

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.

An illustration of this contrast between truthful and expected attitude is how Hotspot’s feature called “full speed debugging” [Sun Microsystems 2001] really means “slowed down, but only when the debugger is attached”, using dynamic deoptimisation to provide the expected view when a debugger is attached. Since debugging users never actually observe execution of the optimised code, this is not a truthful approach.

“Best of both” Both expected and truthful approaches have merit. To approximate a “best of both” solution [Kell 2021], ongoing efforts among truthful approaches seek to approximate more closely the expected semantics, often by introducing more detailed metadata—such as allowing the debugger to split one machine-program state into many virtual states that can be stepped through separately, each presenting different source-level *location views* [Oliva 2017b], which roughly equate to “fractional” program counter values between actual machine instructions. Meanwhile, expected approaches often seek to more closely approximate the level of optimisation allowed by the truthful approach by inhibiting optimisation only “on demand” and “just enough” [Van De Vanter et al. 2018]. There are, however, fundamental limits to these trajectories. The expected approach is necessarily more constraining of optimisation because *at least* while execution is under observation, it must appear to preserve a simulation of the source program, *at least* up to the queries requested by the user (e.g. breakpoints, watchpoints, etc). The truthful approach, on the other hand, has by definition not enforced any such constraint during optimisation. This suggests an end-point where *both* illusions are available and explicitly distinguished: for example, a truthful implementation could offer expected semantics *additionally* by maintaining the code and program state necessary for modelling the less optimised execution separately somehow (i.e. code and state which would otherwise

be discarded); the location views technique is a limited form of this. An expected debugging illusion could optionally be stepped down to truthful, analogous to how source-level debuggers may permit stepping down to assembly. Current debugging technologies are, however, far from this point: I know of no fully expected system that can also map real stack or register contents to their source-level views, for example, and no truthful system that can also fully simulate the unoptimised if requested.

“Truthful” fidelity improvements This category includes any technique that treats the machine program state as the ground truth and tries to “lift” this up to a source-level view where possible; this includes the value expressions of DWARF [2017] and similar formats. The earliest research into debugging optimised programs takes a broadly similar approach of propagating mapping information through the passes of the compiler [Hennessy 1982; Coutant et al. 1988]. It aims for the more limited truthful behaviour: sometimes partial information is presented when the full context may be missing. One of the first efforts [Hennessy 1982] mainly focused on tracking the location of source variable values in optimised program state during execution. DOC [Coutant et al. 1988] stores debug info tracking locations of values from memory through to registers, resembling an early form of today’s debug info.

“Expected” up to queries To preserve expected behaviour, a system must either forgo non-illusion-preserving optimisations up-front, or must retain enough extra state to allow them to be disabled or undone dynamically. This state-retaining, less-optimised code can be viewed as an instrumented version of the base code. GraalVM lazily adds debug instrumentation to the program AST and recompiles (at the moment a breakpoint is set) with optimisations enabled [Van De Vanter et al. 2018]. As with `printf`, the resulting code remains optimised “up to” the active queries, which are embodied in ordinary code seen by the compiler, so may restrict optimisation.

Hybrid approaches [Zellweger 1984] describes how to compare unoptimised and optimised program data-flow and control-flow to untangle state changes of specific optimisations (Fig. 6.1) and recover variable values. OPTVIEW [Tice and Graham 1998] adds a modified source listing to the pure truthful illusion that models the optimised program and so may not match the original source. Their approach relies on the compiler to annotate *key instructions* in the binary for each preserved source statement.

A critical characteristic of the optimised source is that the order of statements in the source must reflect their order of execution. Determining the order in which the source statements will be executed becomes complicated after optimisation, as there is no longer a clear concept of individual source statements in the target code. The instructions for statements have been broken up, duplicated, recombined, and interleaved, making it difficult to state where one statement ends and another

Source fragment

```

10 x, count: INT;
11 SetX: PROC [i: INT] = INLINE
12   {count ← count + 1;
13    x ← i};
14   . . .
50 AdjustX: PROC [a,b: INT] =
51   {IF a < 10 THEN
52     {SetX [b+3];

53     a ← 100}
54   ELSE
55     SetX[b-5];

56   };

```

After expansion

```

AdjustX: PROC [a,b: INT] =
  {IF a < 10 THEN
    {<load b+3>;
     <store i>;
     count ← count + 1;
     x ← i;
     a ← 100}
  ELSE
    {<load b-5>;
     <store i>;
     count ← count + 1;
     x ← i};
  };

```

Navigator debugging tables

inline table

inline call id	inline name	call source
1	SetX	22
2	SetX	55

object table

source table

object	source	source	object
0	50	11	9,22
2	51	12	10,23
6	52	13	13,26
9	11 ¹	...	...
10	12 ¹	50	0
13	13 ²	51	2
15	53	52	6
		53	15
19	55	54	19
22	11 ²	55	19
23	12 ²	56	28
26	13 ²		
28	56		

Fig. 6.1: Inlining example in the Navigator debugger [Zellweger 1984] for optimised programs in the Cedar language used at Xerox PARC.

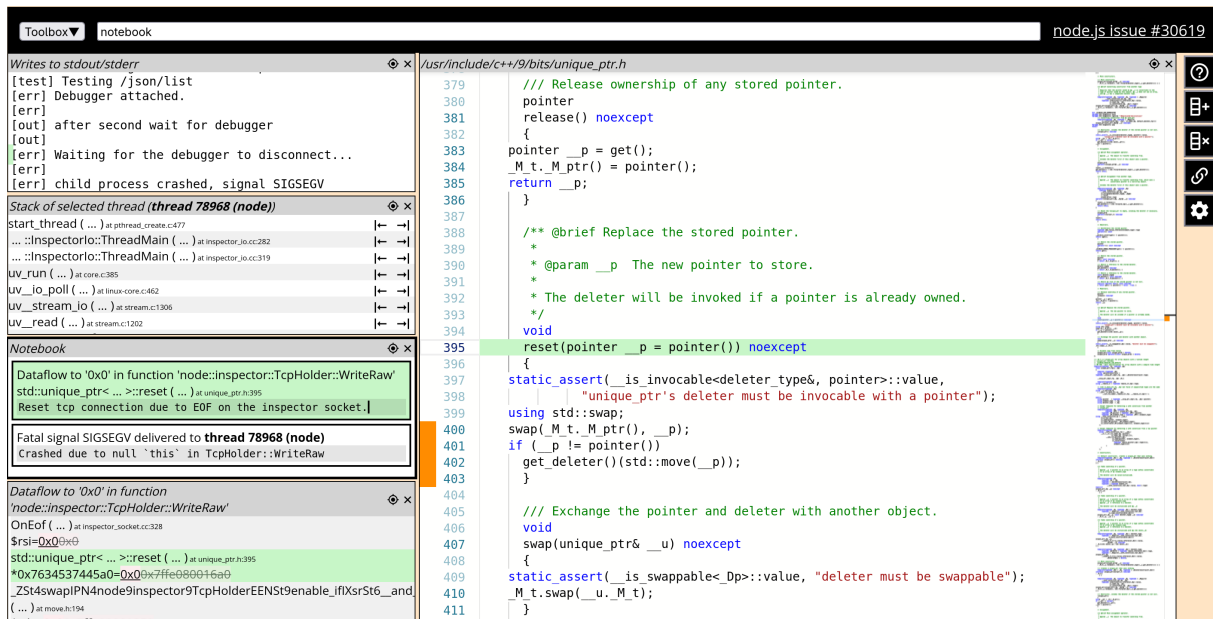


Fig. 6.2: Pernosco “notebook” during investigation of an intermittent race condition in a Node.js test. Values can be traced backwards through time and linked together via dataflow analysis features. Debug info enhances many of the views in the left column, allowing connections between program state across time to be shown in terms of source-level types and variable names.

begins. In order to deal with this problem we use key instructions. For any source statement, its key instruction is the single low level instruction that most closely embodies the semantics of that statement. [Tice 1999]

Omniscience or “knowledge levels” As discussed earlier (§3.6.4), a debugger could currently perform *knowledge extension* and more generally retain state, thereby improving the debugging illusion for any given metadata. While compilers often think of optimisations as lossily reducing or eliminating computation, with debug info we argue it is better to see optimisations as losslessly “residualising” computation. Rather than eliminating computation, it is “moved into” debug info. 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 (Fig. 6.2), 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 (when it was not anticipated that a debugger would be needed) and are arguably less truthful than a conventional debugger such as gdb over ptrace, which does not bring such constraints.

6.3 Print debugging

Interactive debuggers and related tool-based techniques are of course not the only way to find and fix software bugs. Nearly every language ecosystem offers some form of “print” statement that can be added directly to the program itself to trace specific values of interest.

Print debugging has the obvious downside that you must anticipate the variables of interest, change the program to report their values, recompile the program (if using a compiled language implementation), and only then start to see results. This can be quite cumbersome, especially in larger codebases. It does have several strengths as well. As we mentioned, print features are available in nearly all languages, while debugging tools can often be language-specific [Siebenmann 2022]. Use of print statements also happens in the same context as “normal” programming, so there are no separate tools or “mental context switches” required [Andersson 2022]. You can have confidence that you’ll see the correct values even with optimisation enabled, but the print statements may alter optimiser behaviour and change the performance profile of the program.

Interactive debugging (once you have it set up and running) can be used to navigate the state of the whole program, and with modern editor integrations, it can show many variables on screen in context with the program source. There is no need to modify the program at all, though as this dissertation discusses, current debuggers can struggle to present optimised programs in a sensible manner due to incorrect and incomplete debug info.

Both paths (print debugging and interactive debugging workflows) have advantages and disadvantages, so it becomes a matter of personal taste. It’s best to keep both options in mind, and evaluate each path when faced with a bug to fix, so one can choose tools that are most effective for the problem at hand. Further work in omniscient debuggers (built on record-replay systems) may be one way to combine the advantages of both of these paths [O’Callahan 2019].

There are many reasons why print debugging is still the best option for many developers. `rr`, `Pernosco` and similar tools can’t even be used at all in many contexts. However, most of the limitations of these tools (programming languages, operating systems, hardware platforms, overhead) could be mitigated with sufficient investment in engineering work and a modicum of support from platform vendors. It’s important to keep in mind that the level of investment in these tools to date has been incredibly low, basically just a handful of startups and destitute open source projects. If the software industry took debugging seriously — instead of just grumbling about the tools and reverting to print debugging (or, at best, building a polished implementation of the features debuggers have had since the 1980s) — and invested accordingly we could make enormous strides, and not many people would feel the need to resort to print debugging. [O’Callahan 2021]

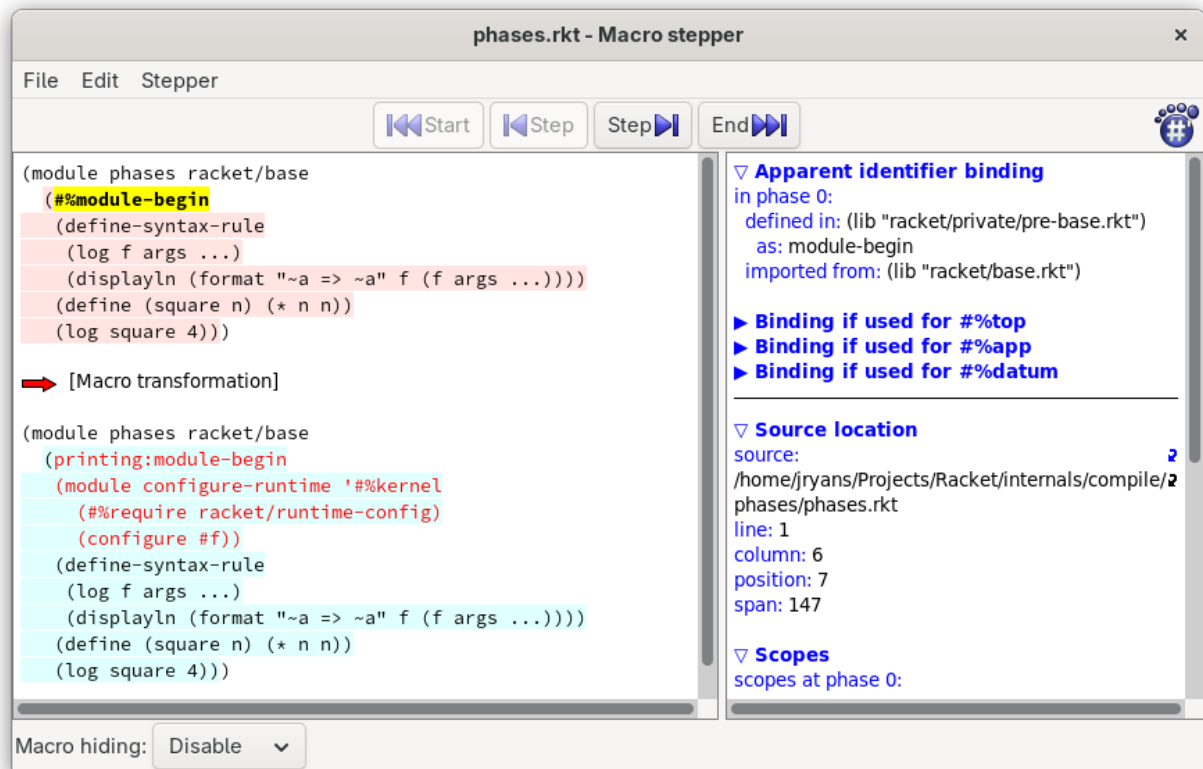


Fig. 6.3: Racket’s macro stepper [Culpepper and Felleisen 2010] showing program transformation during macro application.

6.4 Functional language debugging

Although we did not state it explicitly, most of the efforts cited thus far in this chapter are focused on debugging experiences and debug info for imperative programming languages. Other language families, such as functional languages, can also benefit from robust debugging tools as well. Unfortunately, debugging tools for functional languages are currently less well-developed, leaving functional ecosystems even more dependent on print debugging [Micinski 2021].

Functional language semantics are often further away from machine instructions than imperative languages, which makes it harder to reuse traditional binary debuggers (GDB, LLDB) which originated from imperative toolchains [Whittington 2020]. The mostly-pure value semantics of functional languages allows optimisers to transform programs quite thoroughly, leaving the user with an unrecognisable machine-level program that may be quite hard to manually map back to source.

Many functional language implementations offer a read-eval-print loop (REPL) interface that can be used for testing out an idea or light debugging [Whittington 2020]. The debugging tools we discuss here aim to offer more abilities beyond those found in a basic REPL.

Haskell Some functional language communities have developed tracing-focused debugging tools of varying levels of sophistication. Haskell has perhaps the most developed lineage of this

style of functional tracing work. As one of the few languages that defaults to lazy evaluation, it can be even more challenging to follow the steps taken in Haskell programs, so users need all the help they can get from debugging aids [Huang et al. 2023]. The Freja debugging system [Nilsson 2001] uses a separate compiler and builds a call-tree-like trace on-demand during debugging. The debugging user is prompted through a series of questions to check which reductions are correct, which then allows the system to pinpoint a problematic area of code. The Hood debugging tool [Gill 2000] allows observing and printing values at specific user-chosen program points, similar to the basic function of a stepping debugger. The Hat debugging tool [Chitil et al. 2003] rewrites Haskell programs to embed tracing logic which records each reduction step to a trace file during program execution. Hat provides various tools to explore these traces in various ways, including forward and backward stepping, semi-automatic error location (inspired by Freja), observe specific program points (inspired by Hood), etc. These tracing based tools enable some basic debugging abilities, but also have various drawbacks such as huge trace files and custom compilers that can make them harder to use. To ameliorate these availability issues, the Glasgow Haskell Compiler (GHC) now supplies an interactive debugging mode in the REPL [Marlow et al. 2007]. According to user surveys, current Haskell debugging tools are hard to understand and hard to apply in practice, so there is still much room for improvement [Huang et al. 2023].

Racket An interactive stepping debugger as well as an “algebraic stepper” [Clements et al. 2001] are available for Racket [Felleisen et al. 2015]. The debugger recompiles the user program with many annotations somewhat akin to debug info formats, but these annotations are instead part of the run-time program, and this transformation often slows programs down enough that the Racket debugger can be hard to use effectively. The algebraic stepper shows program execution step by step, with each sub-expression transforming into its result until the function’s final value is computed, similar to how one might compute a value by hand. The stepper only supports simpler teaching languages, and is not available for the full Racket language. A related “macro stepper” [Culpepper and Felleisen 2010] is also available, which shows each step of macro execution (Fig. 6.3) as expressions are transformed by these metaprograms.

OCaml Recent work in the OCaml ecosystem [Whittington and Ridge 2019] approaches these debugging challenges by offering an interpreter that can show the intermediate stages of program execution (Fig. 6.4), similar to Racket’s stepper. Further work may integrate this OCaml interpreter with the default compilation pipeline, somewhat like imperative VMs that can deoptimise at debugging time (at the moment, you must choose one or the other). Some have also experimented with a modified OCaml compiler that emits DWARF debug info, allowing traditional stepping debuggers to be used [Shinwell 2016]. Shinwell and their collaborators have continued those efforts in Jane Street’s recently open-sourced “OxCaml” fork of OCaml [Jane Street 2025].

```

$./ocaml examples/factorial.ml -show-all -side-lets -remove-rec factorial -no-if-bool -no-arith
factorial 4
n = 4 => if n = 1 then 1 else n * factorial (n - 1)
n = 4 => n * factorial (n - 1)
n = 4 => 4 * factorial (n - 1)
           => 4 * factorial (4 - 1)
           => 4 * factorial 3
n = 3 => 4 * (if n = 1 then 1 else n * factorial (n - 1))
n = 3 => 4 * (n * factorial (n - 1))
n = 3 => 4 * (3 * factorial (n - 1))
           => 4 * (3 * factorial (3 - 1))
           => 4 * (3 * factorial 2)
n = 2 => 4 * (3 * (if n = 1 then 1 else n * factorial (n - 1)))
n = 2 => 4 * (3 * (n * factorial (n - 1)))
n = 2 => 4 * (3 * (2 * factorial (n - 1)))
           => 4 * (3 * (2 * factorial (2 - 1)))
           => 4 * (3 * (2 * factorial 1))
           => 4 * (3 * (2 * factorial 1))
n = 1 => 4 * (3 * (2 * (if n = 1 then 1 else n * factorial (n - 1))))
           => 4 * (3 * (2 * 1))
           => 4 * (3 * 2)
           => 24

```

Fig. 6.4: OCaml interpreter [Whittington and Ridge 2019] showing each execution step. The underlined subexpression in each step shows the portion that will be reduced next.

6.5 Applications of debug info

While debugging tools are a major consumer of debug info, there are many other applications beyond this as well. Profiling and coverage tools leverage debugging metadata to map execution back to source coordinates for human-friendly reporting [LLVM Project 2023].

There is also a history of more surprising applications, such as exception handling [de Dinechin 2000], bounds checking [Avijit et al. 2004], link-time code generation [Kell 2010], and type checking [Banavar et al. 1994]. Assured metadata will also interest security researchers, since Turing-complete metadata has been shown to allow hiding exploits not detected by static analyses [Oakley 2011].

Malleable software [Stinnett and contributors 2019; Kell 2009; Basman and Tchernavskij 2018; Beaudouin-Lafon 2019; Tchernavskij 2019] explores techniques that empower domain experts (non-programmers who are experts in some other field) to control and modify their digital workflows. To give a few examples, a user may have a preferred colour picker tool from one application that they wish to use system-wide [Basman and Tchernavskij 2018], or they may want features of two apps to interoperate in some way not imagined in advance by the upstream vendors [Beaudouin-Lafon 2019]. These desires and more would ideally be possible for individual users to achieve, but current technological and economic practices make this quite hard to achieve for even expert programmers. There is still much work to do before this dream can be realised.

Metaprogramming techniques may help to enable malleability. In particular, reflection and aspect-oriented programming [Kiczales et al. 1997; Tanter 2004; Wallis 2024] show promise by enabling additional functionality to be knitted together with existing software, though these techniques are largely aimed at software developers rather than users. Reliable debug info could enable these approaches to be applied after compilation to binary programs in the wild.

Detailed debug info could also be used by tools like `liballocs` [Kell 2015] to enable customisation of running software, but this is only feasible with accurate and reliable metadata as we pursue here. VM-based language implementations often end up maintaining reliable metadata internally that tracks the process’s memory space (perhaps without realising the potential for other applications) to support features like garbage collection, dynamic deoptimisation, edit and continue, and more. Ahead-of-time compiled implementations, on the other hand, do not typically depend on the correctness of the metadata they produce, so further work is needed before one could rely on debug info to edit the behaviour of compiled programs in pursuit of malleability.

6.6 Debug info coverage

To my knowledge, relatively little work has addressed the problem of measuring coverage of variable location info. As covered earlier (§3.4.1) two existing tools `llvm-dwarfdump` [LLVM Project 2020b] and `debuginfo-quality` [O’Callahan 2018] compute essentially the same instruction-based metric, whose problems we surveyed. Assaiante et al. [2023] used a differential approach which tracks how many variables can be accessed in the debugger on each source line relative to the unoptimised version. The metric we’ve described offers more precision via a source analysis baseline and by considering only each variable’s defined region.

Rather more work has addressed other aspects of source-level debugging of optimised code (see approaches in §6.2). On-stack replacement systems [Hölzle et al. 1992; Guo 2014; D’Elia and Demetrescu 2018] sidestep the issues motivating the coverage metrics effort, but rely on just-in-time compilers, and also constrain the compiler to retain specific program points where it is safe to jump from optimised to unoptimised mode for debugging. By contrast, ahead-of-time toolchain compilers are traditionally implemented without such constraints, and thus can benefit from the metrics work in Chapter 3.

6.7 Compiler testing

Many techniques for testing compilers have appeared over the years [Chen et al. 2020]. We review a few highlights of this field, and attempt to relate them to the debug info focus of the work in this dissertation.

Translation validation This technique focuses on the input-output behaviour of two program versions, ensuring that IO semantics are preserved even when the program may be substantially transformed. There are several variants of this translation validation approach, including some that check the entire optimisation pipeline [Pnueli et al. 1998; Necula 2000; Stepp et al. 2011; Tristan et al. 2011; Sewell et al. 2013] and another that examines each pass individually [Lopes et al. 2021]. While this works well for checking run-time program behaviour

across optimisation, it does not check anything about the mapping between source and object programs that we find in debug info.

Pnueli et al. [1998] introduces the translation validation concept, posing it as an alternative to formally verifying the compiler in advance through proofs (in the style of e.g. CompCert [Kästner et al. 2017]). Nacula [2000] describes their implementation of translation validation for an optimising compiler (GCC). They make use of a *simulation relation* to guide their validation tooling.

A *simulation relation* between two IL [intermediate language] programs S and T (the source and the target) is a set of elements of the form $(PC_S, PC_T, \bar{E})$, where PC_S and PC_T are program points in S and T respectively and $\bar{E}$ is a sequence of boolean expressions referring to temporaries live in S and T at the respective program points. [Nacula 2000]

Checking equivalence is then a matter of checking each element of the simulation relation. Stepp et al. [2011] uses a validation approach based on *equality saturation* [Tate et al. 2009] where the optimiser adds equality information relating multiple program versions in a single intermediate representation. Tristan et al. [2011] improves upon these previous approaches in the context of LLVM by rewriting program *value graphs* to a *normalised form* (rather than finding possibly many equivalent programs through saturation). Sewell et al. [2013] describe translation validation techniques targetting the seL4 microkernel where they prove refinement between formal semantics at the C source level and the binary level.

Program generation Test programs can be generated through a variety of means, including manual and automated approaches. Csmith and its relatives [Eide and Regehr 2008; Yang et al. 2011; Livinskii et al. 2020; Even-Mendoza et al. 2022] randomly generate valid input programs and then make use of differential testing to check their run-time behaviour by compiling them with multiple toolchains and comparing the output after running each compiled program.

Eide and Regehr [2008] use their program generation tool *randprog* to test compiler handling of the C `volatile` qualifier. Yang et al. [2011] build on this work to create Csmith, which covers a large subset of C while also staying away from undefined and unspecified behaviours (UB) using a mixture of static and dynamic constraints. Livinskii et al. [2020] describes their tool YARPGen which supports *generation policies* to allow skewing the random choices made during program generation to increase output diversity. YARPGen uses only static analysis to avoid UB, which allows for more expressiveness than the Csmith approach, thereby potentially testing additional parts of the compiler. Even-Mendoza et al. [2022] alter Csmith to create CsmithEdge where UB constraints are relaxed to increase expressiveness and then any generated programs that do contain UB are filtered away using a separate analysis pass.

While this differential technique using generated programs could be aimed at a debug info (and indeed I even tried it early on), it is challenging to apply because:

1. Some generation tools produce programs that are largely dead code, which is entirely removed by optimisation, leaving no debug info to compare across program versions
2. The remaining variations between compilers are often small changes (e.g. slight source coordinate changes) that result from internal compiler implementation details and there are often good arguments for each version, leading to no clear resolution for such differences

Program reduction Work on program generation has also inspired the creation of test case reduction tools, such as C-Reduce [Regehr et al. 2012], which uses an extensible suite of source transformations to construct reduced program variants. C-Reduce takes a user-supplied “interestingness test” which constrains reduction to ensure essential test case properties are preserved. The debug info testing approaches described in this dissertation could be used as part of a reduction interestingness test. For example, one could test for changes in debug info coverage to identify program variants that may alter debug info quality.

Metamorphic testing This approach [Chen et al. 1998; Segura et al. 2016; Donaldson et al. 2017; Tolksdorf et al. 2019] is a technique that gets around the test oracle problem by enumerating “metamorphic relations” that describe how each change to an input can be reflected in the output. For the domain of compiler testing, metamorphic approaches produce two programs that should be equivalent for all inputs by e.g. injecting dead code into the program [Donaldson and Lascu 2016] or applying semantics-preserving transformations [Donaldson et al. 2017] such as $\sin(x + 2\pi) = \sin(x)$.

While I am not aware of any work applying metamorphic testing to debug info so far (and I believe it may encounter challenges similar to those mentioned in the program generation section above), recent work [Tolksdorf et al. 2019] does focus on a related problem of *interactive* metamorphic testing of debuggers. They vary the script of the debugger actions (e.g. add breakpoint, step to next line, etc.) using metamorphic relations and check that the same program state is presented to the debugging user, finding bugs in the *debugger itself* along the way.

Symbolic execution This is a program analysis technique [Boyer et al. 1975; King 1975; Clarke 1976; Khurshid et al. 2003; Cadar et al. 2008; Baldoni et al. 2018] that allows checking whether some property holds by systematically exploring program paths using symbolic values instead of concrete inputs. It has spawned a huge sub-field over the past five decades, with one popular symbolic execution tool (KLEE) [Cadar et al. 2008] currently tracking more than 300 papers that use or extend their system.

Early symbolic systems were fairly similar and focused on the core idea of representing program outputs as formulae with operations applied to program inputs. SELECT [Boyer et al. 1975] provides example program inputs that can serve as test data and output assertions match

the path. EFFIGY [King 1975] tracks path constraints using a series of algebraic operations. Clarke [1976] describes their own system that improves on these by analysing a standard language (Fortran) rather than a subset or dialect.

Hundreds of papers have improved on various aspects [Baldoni et al. 2018] of the core symbolic execution concept, including mixing in some amount of concrete execution, different forms of memory modeling, new ways of interacting with the program environment, approaches to tame path explosion, and more.

Lazy initialisation [Khurshid et al. 2003] is a particularly useful innovation that was applied in the symbolic testing work described here. When working with complex data structures (e.g. lists, trees, etc.), a naive approach might eagerly create a new object for every element, but this can lead to path explosion. Lazy initialisation defers object creation until the time of first access. Once accessed, the symbolic state is forked to cover several possibilities for the accessed reference: (a) empty / null, (b) a new symbolic object, (c) an existing concrete object (of the expected type).

KLEE [Cadar et al. 2008] captures symbolic “path constraints” during program execution, enabling assertions (and other operations of interest) to be checked automatically via a theorem prover. Any source program condition written as an assertion is converted to a proof and checked along each path that encounters it during execution.

Some usages of symbolic execution benefit from examining only a small slice of the program at a time as we did in the work described here by way of an *analysis window*. The window controls the set of functions to explore while skipping the rest and conservatively modelling the effects of skipped functions by resetting data they may have accessed to unconstrained symbolic values. This feature is similar to several “under-constrained” symbolic execution modes that have appeared previously [Ramos and Engler 2015; Barrett 2021], though so far this style of partial-program symbolic execution has not yet made its way into upstream KLEE.

Symbolic execution is most often used for bug finding by checking that e.g. no memory safety rules have been violated. Applications of symbolic execution to compiler testing are less common, though they can be found in the literature. Combining symbolic execution with forms of translation validation has been the primary route thus far, with an early application in Lisp compilers [Samet 1976] and more recent work [Wang et al. 2023b] that checks the correctness of Halide compiler transformations. There has also been work towards a symbolic-execution-based debugger [Hentschel et al. 2019] that combines typical stepping debugger actions with the ability to explore across many program paths starting from any program point of interest.

6.8 Debug info validation

As we’ve just reviewed, there are various approaches to compiler testing in the literature, and it would be nice if existing approaches for relating multiple versions of a program could be used to check debug info as well. Unfortunately, debug info correctness is not reducible to most of

those techniques as we mention above. In [Chapter 5](#), we were able to make use of symbolic execution to check variable value correctness in debug info across optimisation.

Existing approaches to validating debugging information, such as Dexter [[Bedwell 2018](#)] used in the LLVM codebase, rely on manually annotating individual test cases with the expected state at each program point of interest. While this provides some test coverage, it also consumes compiler engineer time as well.

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 check *correctness*, it is unable to check *completeness* since `printf` prevents removal by the optimiser. It also assumes that source coordinate info is correct, but this information is often dropped or miscalculated by optimisers. 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. Unlike the work in this thesis, both of these systems can cover the compiler “end to end” including front and back ends, rather than just the optimiser.

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) stores to globals if they are written and that (3) 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 that these conjectures would be applicable for a whole-program optimiser, for example, and the approach overall offers at best a lower bound of debug info sanity.

“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`). 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 the work in [Chapter 4](#), 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.

6.9 Improving debug info

LLVM compiler authors [Cazalet-Hyams 2024] have enhanced internal APIs for managing source coordinates so that optimisation pass author intent is tracked. If a pass e.g. drops source coordinates without using a new annotation to convey that intention, this can be detected by automated tooling and reports for review and potential fixes. They are also working on implementing the key instructions concept (which aims to preserve expected stepping behaviour even after program optimisation) [Tice and Graham 1998] in LLVM.

Recent research [Huang et al. 2025] proposes a technique to find and fix source coordinates in debug info. They define a “conformance relation” between unoptimised and optimised program versions. Their tooling collects source coordinate updates by observing program control flow during optimisation in LLVM, and these coordinate changes are then checked against those the compiler produced itself. For compiler passes manipulating IR using a limited set of primitives (Create, Clone, Move, and Replace) modelled on a subset of the LLVM API, they compute 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.

Chapter 7

Conclusions and future work

7.1 Summary

As described in the introduction ([Chapter 1](#)), the work in this dissertation focuses on checking the *completeness* and *correctness* of variable location and function call information in debug info.

In [Chapter 2](#), we introduced the debugging illusion concept which is implicitly used by software tools like debuggers and profilers to help developers to understand programs in source-level terms. Debug info formats such as DWARF [\[2017\]](#) enable residualised computation over program state to map the machine-level program back into the source-level program. We focused on the *expected* illusion (where tools present a view of the program as if it were unoptimised using operational semantics consistent with the source language) and identified control coverage, control consistency, variable coverage, and variable semantic consistency properties of that illusion. Each of the subsequent chapters explored approaches to check many of these properties.

In [Chapter 3](#), we focused on the variable coverage aspect, i.e. the problem of *missing* local variable information. Current approaches for measuring the extent of coverage of local variables are based on crude assumptions (for example, assuming variables could cover their whole parent scope) and are not comparable from one compilation to another. We proposed new metrics which could serve as motivation for language implementations to improve debugging quality. We have elaborated these properties into coverage criteria that we have embodied in an implemented tool and found useful in finding and explaining real debuggability problems. A variable coverage technique based on this work has been contributed upstream to LLVM, and LLVM compiler authors plan to make use of this to both check coverage of individual compiler changes and also to track coverage over time to avoid regressions.

In [Chapter 4](#), we examined how the function call / return subset of control consistency has been affected by features such as inlining and tail calls during optimisation. We conducted a study of the quality of debugging information in respect of the *source-level dynamic call trees*

it can recover. We compared the trees recovered from optimised and unoptimised versions of the same program, producing a classification of the observed divergences. For each class, we analysed whether format or compiler is to blame and identified specific ways to address these defects. This 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.

In [Chapter 5](#), we described a specific consistency property that isolates the problem of *incorrect* local variable information. We investigated a new approach in which unoptimised and optimised versions of a program are symbolically executed side-by-side as a way to check for consistency in their debugging information. We showed that this allows checking correctness of debug info generation across a much larger portion of the compiler than that of other approaches like manually-written tests.

Throughout this dissertation, we have also argued that compilers could and should adopt stronger criteria for completeness and correctness of their debugging information. We have elaborated these properties into correctness criteria that we have embodied in implemented tools and found useful in finding and explaining real debuggability problems.

7.2 Further projects

The following describes additional related projects that further improve aspects of debug info which will hopefully be explored in the future by myself or others in the community.

7.2.1 Residualisation

Debug info formats like DWARF [\[2017\]](#) make use of Turing-complete “residual” computation abilities, as we first explained in [§2.3.1](#). While working on the projects in this thesis, we identified several opportunities for debug info quality improvements by taking these residualisation ideas even further.

Designing for state retention A first, modest goal (out of those listed in this section) is to allow a conventional debugger to achieve a stronger illusion by “residualising state”, i.e. preserving local variables over ranges when they would otherwise be *unknowable*, as we

developed in §3.6.4. Current debug info does make this possible, for example by setting a “hidden breakpoint” on all local variables’ end-of-range locations and snapshotting the “last seen value” when these breakpoints are hit, retaining this value at least until exit from its defining scope. (This still requires the debugger be already attached at the point of last use, but not necessarily from the start of execution—in contrast to record/replay.) Tice [1999] implemented a related state-residualisation technique for local variables in the Optdbx debugger, confirming that this high-level idea is indeed viable, but this has so far not appeared in mainstream debugging tools. Doing this reliably would require control-flow reconstruction in the debugger, since the numerically “last” instruction in an address range might not be the last to execute. Doing this efficiently would require a faster breakpoint mechanism, such as that of Kessler [1990], rather than the usual hardware trap approach, since very frequent traps would cause noticeable slowdown. Compilers could help in both cases: they could explicitly record the end-of-range instructions requiring a breakpoint, as a new record in the debug info. For efficiency, they could even assist fast breakpointing at such locations, e.g. emitting a “nop sled” similar to DTrace probe points [Cantrill et al. 2004], although the code size cost could be prohibitive, and this would diverge from the usual practice that code generation is not affected by debugging support. A good design for addressing these problems therefore remains future work.

Design for residualisation We have argued that debug info can be seen as a “residual” destination for code and/or state eliminated from the emitted program. To date, compilers have not been engineered on this basis. Rather, the generation of debug info is an additional and “manual” task for the author of an optimisation pass. Could it instead be an automatic side-effect of the primitive IR transformations provided by a compiler framework? This stands to align debug info generation with verified compilation [Leroy 2009; Kästner et al. 2017] and translation validation [Necula 2000; Lopes et al. 2021].

Reasoning about knowability We have focused on techniques for detecting problems, rather than fixing them. Currently, if a compiler drops a local variable from the debug info over some range, perhaps because it lacked the ability to generate a sophisticated DWARF expression to recover that value, the tool we developed will flag an inconsistency if the unoptimised code updates the variable during this range—the optimised variant appears not to perform this update. However, it is likely that information gathered during consistency checking can be useful for suggesting the right value expression for such a variable—which although no longer *allocated*, not *known* according to current debug info, but nevertheless *knowable* as some function of program state; SMT techniques over these collected formulae may allow discovering this knowability, for example to infer the “ $b - 2$ ” formula in Fig. 3.4.

7.2.2 Push beyond internal limits of debuggability

Compilers are often poorly structured internally to recover debug info to the maximum extent possible with current debug info formats. In particular, debug info handling in optimisation passes is often pass-specific (as opposed to generalised debug-info-handling infrastructure that could act on behalf of all passes), and there is limited code reuse. If a pass has limited or no debug info handling support, then the metadata is likely to be lost by any transformations the pass makes.

To tackle this topic, one could survey compiler IR mutation APIs, with a focus on debug info handling in those APIs. Current IR mutation is quite low-level (e.g. “insert instruction *X* at point *Y*”) and leaves each individual optimisation pass responsible for updating debug info as needed. After exploring the IR mutation APIs as they are currently, it would be possible to build somewhat higher-level APIs (e.g. “replace block *A* with block *B*”) that would allow centralised code paths to do more of the work to transform debug info where needed. For example, replacing the low-level sequence of operations “remove instruction *X*, add instruction *Y*” with something more like “replace instruction *X* with instruction *Y*” would convey the pass author’s intention such that shared support code could do any debug info transformations to ensure variables that might have been described in terms of the old instruction *X* are updated to be in terms of *Y* (or other program state that remains). This should (if feasible) improve the quality of debug info across all passes and also reduce the effort asked of pass authors. Optimisation passes can then incrementally upgrade to these higher-level mutation APIs as desired, gaining better debug info quality with each pass that does so.

Rewriting compiler passes in terms of a more restricted domain specific language (DSL) could be another approach to this topic. For example, the nanopass compiler architecture as well as compilers using attribute grammars enable optimisation through the use of DSLs [Keep and Dybvig 2013; Kramer and Van Wyk 2020; Ringo et al. 2023]. To my knowledge, these compiler DSL techniques have thus far only explored the run-time optimisation aspect of pass construction, but they could be extended to also emit debug info as well. The restricted language of these DSLs may ease the creation of “debuggability contracts” describing debug info correctness guarantees offered at different optimisation levels. They could also aid formal verification of debug info pathways, in a similar vein to what CompCert has achieved for run-time program optimisation [Leroy 2009; Kästner et al. 2017]. This approach does of course imply a significant engineering cost if it were to be retrofitted into a production compiler, as there would be many existing optimisation passes to rewrite.

With either of the approaches mentioned in this section, their debug handling improvements can be evaluated by checking variable coverage both end-to-end and also pass by pass. More passes should be able to cope with more scenarios than they could before, leading to better coverage, which can be measured using the approach from Chapter 3. The reduction in code size can also be measured as a proxy for reduced developer effort. The correctness of these

changes can be checked with the consistency checking tools from [Chapter 4](#) and [Chapter 5](#).

7.2.3 Explore external limits of debuggability

Debug info formats have grown various features (e.g. DWARF value expressions) organically over time which attempt to capture enough info to reconstruct a source-level view of program state. As we have mentioned throughout this dissertation, the current level of expressivity in these debug info formats is not sufficient to convey all of the necessary program state for all of the source language features in use. For some cases which are not currently captured in debug info, it is hard to be sure whether existing metadata features could be used (potentially in some new way) or if there is in fact a fundamental limit in current debug info formats which would need to be extended.

One could exhaustively explore the limits of debug info expressiveness in current debug info formats. For example, as we have already described, there are various kinds of variable location situations which are not possible to express in current formats, but this has never been thoroughly explored to enumerate all such cases. The aim would be to pinpoint the fundamental expressiveness limits like this which could dramatically improve debug info coverage for wide swathes of variables and programs.

A recent “location views” [[Oliva 2017b](#)] effort demonstrates one potential expressiveness extension which expands the frontier of program states that can be represented in debug info: it allows the debugger to split one machine-program state into many virtual states that can be stepped through separately, each presenting different source-level states.

New extensions to debug metadata that arise from this exploration can be evaluated both conceptually in terms of language semantics supported and also quantitatively via variable location coverage improvements. Changes in debug info coverage can be measured as we have shown in [Chapter 3](#). In addition, the consistency checking approaches [Chapter 4](#) and [Chapter 5](#) can be applied here as well to check for correctness.

Bibliography

- F. E. Allen and J. Cocke. 1976. A program data flow analysis procedure. *Commun. ACM* 19, 3 (March 1976). doi:10.1145/360018.360025
- 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
- Rasmus Andersson. 2022. If debuggers could visualize my programs while they run. Retrieved 2022-06-06 from <https://twitter.com/rsms/status/1532485114638151680>
- 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
- Kumar Avijit, Prateek Gupta, and Deepak Gupta. 2004. TIED, LibsafePlus: Tools for runtime buffer overflow protection. In *Proc. of USENIX Security '04*. Retrieved 2023-07-18 from <https://www.usenix.org/legacy/events/sec04/tech/avijit.html>
- Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *Comput. Surveys* 51, 3 (May 2018), 50:1–50:39. doi:10.1145/3182657
- R. M. Balzer. 1969. EXDAMS: Extendable debugging and monitoring system. In *Proc. of Spring Joint Computer Conference '69*. doi:10.1145/1476793.1476881
- Guruduth Banavar, Gary Lindstrom, and Douglas Orr. 1994. Type-safe composition of object modules. In *Proc. of the International Conference on Computer Systems and Education '94*. Retrieved 2023-01-06 from <https://users.cs.utah.edu/docs/techreports/1994/pdf/UUCS-94-001.pdf>
- Langston Barrett. 2021. Under-constrained symbolic execution with Crucible. Retrieved 2021-11-29 from <https://galois.com/blog/2021/10/under-constrained-symbolic-execution-with-crucible/>

- Antranig Basman and Philip Tchernavskij. 2018. What lies in the path of the revolution. In *Proc. of PPIG '18*. Retrieved 2019-09-09 from https://github.com/amb26/papers/blob/main/ppig-2018/What_Lies_in_the_Path_of_the_Revolution.pdf
- Théophile Bastian, Stephen Kell, and Francesco Zappa Nardelli. 2019. Reliable and fast DWARF-based stack unwinding. *Proc. ACM Program. Lang.* 3, OOPSLA (Oct. 2019), 146:1–146:24. doi:10.1145/3360572
- Michel Beaudouin-Lafon. 2019. A world without apps. Retrieved 2021-03-31 from <https://www.youtube.com/watch?v=ntaudUum06E>
- 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
- Kristina Bessonova. 2019. [llvm-dwarfdump][Statistics] Unify coverage statistic computation. Retrieved 2023-03-22 from <https://reviews.llvm.org/D70548>
- Hans-J. Boehm. 2005. Threads cannot be implemented as a library. In *Proc. of PLDI '05*. doi:10.1145/1065010.1065042
- Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*. doi:10.1145/800027.808445
- Ronald F. Brender, Jeffrey E. Nelson, and Mark E. Arsenault. 1998. Debugging optimized code: Concepts and implementation on DIGITAL Alpha systems. *Digital Technical Journal* 10, 1 (1998), 81–99. Retrieved 2022-06-27 from <https://www.mrynet.com/FTP/operatingsystems/DEC/www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art7.pdf>
- Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of OSDI '08*. Retrieved 2018-10-01 from https://www.usenix.org/legacy/event/osdi08/tech/full_papers/cadar/cadar.pdf
- Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. 2004. Dynamic instrumentation of production systems. In *Proc. of USENIX ATC '04*. Retrieved 2023-04-22 from <https://www.usenix.org/legacy/publications/library/proceedings/usenix04/tech/general/cantrill.html>
- Orlando Cazalet-Hyams. 2024. Improving optimized code line table quality. Retrieved 2025-04-16 from <https://www.youtube.com/watch?v=yj1EUzXVP1I>
- 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

- Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *Comput. Surveys* 53, 1 (Feb. 2020), 4:1–4:36. doi:10.1145/3363562
- T. Y. Chen, S. C. Cheung, and S. M. Yiu. 1998. Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST-CS98-01. Hong Kong University of Science and Technology. Retrieved 2025-07-10 from <https://arxiv.org/abs/2002.12543>
- Olaf Chitil, Colin Runciman, and Malcolm Wallace. 2003. Transforming Haskell for tracing. In *Implementation of Functional Languages*, Ricardo Peña and Thomas Arts (Eds.). doi:10.1007/3-540-44854-3_11
- Jim Chow, Tal Garfinkel, and Peter M. Chen. 2008. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. of USENIX ATC '08*. Retrieved 2026-03-13 from <https://www.usenix.org/conference/2008-usenix-annual-technical-conference/decoupling-dynamic-program-analysis-execution>
- Lori A. Clarke. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering* SE-2, 3 (Sept. 1976), 215–222. doi:10.1109/TSE.1976.233817
- John Clements, Matthew Flatt, and Matthias Felleisen. 2001. Modeling an algebraic stepper. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, David Sands (Ed.). doi:10.1007/3-540-45309-1_21
- 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
- Ryan Culpepper and Matthias Felleisen. 2010. Debugging hygienic macros. *Science of Computer Programming* 75, 7 (July 2010), 496–515. doi:10.1016/j.scico.2009.06.001
- C. de Dinechin. 2000. C++ exception handling. *IEEE Concurrency* 8, 4 (Oct. 2000), 72–79. doi:10.1109/4434.895109
- 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
- Digital Equipment Corporation. 1967. PDP-8 DDT programming manual. Technical Report DEC-08-CDDA-D. Retrieved 2025-07-08 from <https://bitsavers.org/www.computer.museum.uq.edu.au/pdf/DEC-08-CDDA-D%20PDP-8%20DDT%20Programming%20Manual.pdf>

- Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017), 93:1–93:29. doi:10.1145/3133917
- Alastair F. Donaldson and Andrei Lascu. 2016. Metamorphic testing for (graphics) compilers. In *Proceedings of the 1st International Workshop on Metamorphic Testing - MET '16*. doi:10.1145/2896971.2896978
- DWARF Debugging Information Format Committee. 2017. DWARF debugging information format: Version 5. Retrieved 2019-07-29 from <https://dwarfstd.org/doc/DWARF5.pdf>
- Eric Eide and John Regehr. 2008. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM International Conference on Embedded Software (EMSOFT '08)*. doi:10.1145/1450058.1450093
- Frank Ch Eigler. 2006. Problem Solving With SystemTap. In *Proc. of Ottawa Linux Symposium*. Retrieved 2024-02-08 from <https://www.kernel.org/doc/ols/2006/ols2006v1-pages-261-268.pdf>
- Thomas G. Evans and D. Lucille Darley. 1966. On-line debugging techniques: A survey. In *Proc. of Fall Joint Computer Conference '66*. doi:10.1145/1464291.1464295
- Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2022. CsmithEdge: More effective compiler testing by handling undefined behaviour less conservatively. *Empirical Software Engineering* 27, 6 (July 2022), 129. doi:10.1007/s10664-022-10146-1
- Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket manifesto. In *Proc. of SNAPL '15*. doi:10.4230/LIPIcs.SNAPL.2015.113
- Andy Gill. 2000. Debugging Haskell by observing intermediate data structures. In *Proc. of Haskell '00*. Retrieved 2025-07-24 from https://web.archive.org/web/20170808193857id_/http://www.ittc.ku.edu/~andygill/papers/Hood00.pdf
- GNU Project. 1995. DDD - Data Display Debugger. Retrieved 2025-11-21 from <https://www.gnu.org/software/ddd/>
- Matt Godbolt and contributors. 2012. Compiler Explorer. <https://godbolt.org>
- 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

- Martin Hentschel, Richard Bubel, and Reiner Hähnle. 2019. The Symbolic Execution Debugger (SED): A platform for interactive symbolic execution, debugging, verification and more. *International Journal on Software Tools for Technology Transfer* 21, 5 (Oct. 2019), 485–513. doi:10.1007/s10009-018-0490-9
- Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *Proc. of PLDI '92*. doi:10.1145/143095.143114
- Ruanqianqian (Lisa) Huang, Elizaveta Pertseva, Michael Coblenz, and Sorin Lerner. 2023. How do Haskell programmers debug?. In *Proc. of PLATEAU '23*. doi:10.1184/R1/22277347.v1
- 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
- Jane Street. 2025. OxCaml. Retrieved 2025-07-24 from <https://github.com/oxcaml/oxcaml>
- Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. 2017. Closing the gap – the formally verified optimizing compiler CompCert. In *SSS'17: Safety-critical Systems Symposium 2017*. Retrieved 2022-08-01 from <https://hal.inria.fr/hal-01399482>
- Andrew W. Keep and R. Kent Dybvig. 2013. A nanopass framework for commercial compiler development. In *Proc. of ICFP '13*. doi:10.1145/2500365.2500618
- Stephen Kell. 2009. The mythical matched modules: Overcoming the tyranny of inflexible software construction. In *Proc. of OOPSLA '09*. doi:10.1145/1639950.1640051
- Stephen Kell. 2010. Component adaptation and assembly using interface relations. In *Proc. of OOPSLA '10*. doi:10.1145/1869459.1869487
- Stephen Kell. 2015. Towards a dynamic object model within Unix processes. In *Proc. of Onward! '15*. doi:10.1145/2814228.2814238
- Stephen Kell. 2021. The two cultures of programming language implementation. Retrieved 2022-03-15 from <https://www.humprog.org/~stephen/research/talks/kell21two-slides.pdf>
- Stephen Kell and Conrad Irwin. 2011. Virtual machines should be invisible. In *Proc. of VMIL '11*. doi:10.1145/2095050.2095099

- 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
- Peter B. Kessler. 1990. Fast breakpoints: Design and implementation. In *Proc. of PLDI '90*. doi:10.1145/93542.93555
- Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Hubert Garavel and John Hatcliff (Eds.). doi:10.1007/3-540-36577-X_40
- Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *Proc. of ECOOP '97*, Mehmet Akşit and Satoshi Matsuoka (Eds.). doi:10.1007/BFb0053381
- James C. King. 1975. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software*. doi:10.1145/800027.808444
- Amy J. Ko and Brad A. Myers. 2008. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proc. of ICSE '08*. 301. doi:10.1145/1368088.1368130
- Lucas Kramer and Eric Van Wyk. 2020. Strategic tree rewriting in attribute grammars. In *Proc. of SLE '20*. doi:10.1145/3426425.3426943
- Herb Krasner. 2022. The cost of poor software quality in the US: A 2022 report. Technical Report. Consortium for Information and Software Quality (CISQ). Retrieved 2026-04-21 from <https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf>
- Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. <https://doi.org/10.1145/1538788.1538814>
- 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
- Linux contributors. 2007. Inlining in Linux. Retrieved 2026-07-30 from <https://www.kernel.org/doc/local/inline.html>
- Stephen Livermore-Tozer. 2023. [RFC] Redefine Og/O1 and add a new level of Og. Retrieved 2023-11-07 from <https://discourse.llvm.org/t/rfc-redefine-og-o1-and-add-a-new-level-of-og/72850>
- Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (Nov. 2020), 196:1–196:25. doi:10.1145/3428264

- LLVM Project. 2019. [DebugInfo@O2] Salvaged memory loads can observe subsequent memory writes · Issue #39974 · llvm/llvm-project. Retrieved 2026-08-05 from <https://github.com/llvm/llvm-project/issues/39974>
- LLVM Project. 2020a. How to update debug info: A guide for LLVM pass authors. Retrieved 2025-10-18 from <https://llvm.org/docs/HowToUpdateDebugInfo.html>
- LLVM Project. 2020b. llvm-dwarfdump - Dump and verify DWARF debug information. Retrieved 2023-03-13 from <https://llvm.org/docs/CommandGuide/llvm-dwarfdump.html>
- LLVM Project. 2022. Source level debugging with LLVM. Retrieved 2023-01-05 from <https://llvm.org/docs/SourceLevelDebugging.html>
- LLVM Project. 2023. Profile Guided Optimization - Clang Compiler User's Manual. Retrieved 2023-07-18 from <https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization>
- Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded translation validation for LLVM. In *Proc. of PLDI '21*. doi:10.1145/3453483.3454030
- Simon Marlow, José Iborra, Bernard Pope, and Andy Gill. 2007. A lightweight interactive debugger for Haskell. In *Proc. of Haskell '07*. doi:10.1145/1291201.1291204
- Julia Menapace, Jim Kingdon, and David MacKenzie. 1992. The “stabs” debug format. Online documentation. Retrieved 2026-03-02 from <https://sourceware.org/gdb/onlinedocs/stabs.pdf>
- Kristopher Micinski. 2021. No functional language has a good debugger. Retrieved 2022-01-12 from <https://twitter.com/krismicinski/status/1417217349346897921>
- Microsoft. 1995. Codeview 4 symbolic debug information specification. Product documentation.
- Microsoft. 2017. Debug Adapter Protocol. Retrieved 2025-11-13 from <https://microsoft.github.io/debug-adapter-protocol/>
- Microsoft. 2025a. C++ dynamic debugging (preview) - Visual Studio (Windows). Retrieved 2026-08-04 from <https://learn.microsoft.com/en-us/visualstudio/debugger/cpp-dynamic-debugging?view=visualstudio>
- Microsoft. 2025b. Supported code changes (C++) - Visual Studio (Windows). Retrieved 2025-11-20 from <https://learn.microsoft.com/en-us/visualstudio/debugger/supported-code-changes-cpp?view=visualstudio>
- Jeremy Morse. 2024. Thought process for IPOs like constant propagation. Personal communication.
- Mozilla. 2014. Debugger API — Firefox Source Docs documentation. Retrieved 2025-11-12 from <https://firefox-source-docs.mozilla.org/js/Debugger/index.html>

- George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proc. of PLDI '00*. doi:10.1145/349299.349314
- Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of PLDI '08*. doi:10.1145/1250734.1250746
- Henrik Nilsson. 2001. How to look busy while being as lazy as ever: The implementation of a lazy functional debugger. *Journal of Functional Programming* 11, 6 (Nov. 2001), 629–671. doi:10.1017/S095679680100418X
- James Oakley. 2011. Exploiting the hard-working DWARF: Trojan and exploit techniques with no native executable code. In *Proc. of WOOT '11*. Retrieved 2023-07-18 from <https://www.usenix.org/conference/woot11/exploiting-hard-working-dwarf-trojan-and-exploit-techniques-no-native-executable>
- Robert O’Callahan. 2018. Comparing the quality of debug information produced by Clang and GCC. Retrieved 2021-11-10 from <https://robert.ocallahan.org/2018/11/comparing-quality-of-debug-information.html>
- Robert O’Callahan. 2019. Your debugger sucks. Retrieved 2021-11-10 from <https://robert.ocallahan.org/2019/11/your-debugger-sucks.html>
- Robert O’Callahan. 2021. Print debugging should go away. Retrieved 2021-11-10 from <https://robert.ocallahan.org/2021/04/print-debugging-should-go-away.html>
- 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. 2010. Consistent views at recommended breakpoints. In *Proc. of GCC Summit*. 6. Retrieved 2022-06-27 from <https://www.fsfla.org/~lxoliva/papers/sfn/gcc2010.pdf>
- Alexandre Oliva. 2017a. Location view numbering. Retrieved 2023-01-10 from <https://dwarfstd.org/ShowIssue.php?issue=170427.1>
- Alexandre Oliva. 2017b. 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>
- Oracle. 2000. Java Platform Debugger Architecture. Retrieved 2025-11-26 from <https://docs.oracle.com/en/java/javase/25/docs/specs/jpda/jpda.html>
- Noah Pendleton. 2023. Pocket article: Debug vs. Release builds considered harmful. Retrieved 2023-05-18 from <https://interrupt.memfault.com/blog/debug-vs-release-builds>

- A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*. doi:10.1007/BFb0054170
- David A Ramos and Dawson Engler. 2015. Under-constrained symbolic execution: Correctness checking for real code. In *Proc. of USENIX Security '15*. Retrieved 2019-11-03 from <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos>
- Tristan Ravitch. 2011. Whole-program LLVM (WLLVM). Retrieved 2023-04-07 from <https://github.com/travitch/whole-program-llvm>
- John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proc. of PLDI '12*. doi:10.1145/2254064.2254104
- Jeremy Rifkin. 2022. LLVM optimization pipeline viewer for Compiler Explorer. <https://github.com/compiler-explorer/compiler-explorer/pull/3769>
- Nathan Ringo, Lucas Kramer, and Eric Van Wyk. 2023. Nanopass attribute grammars. In *Proc. of SLE '23*. doi:10.1145/3623476.3623514
- Hanan Samet. 1976. Compiler testing via symbolic interpretation. In *Proceedings of the 1976 Annual Conference (ACM '76)*. doi:10.1145/800191.805648
- Edwin H. Satterthwaite. 1975. *Source language debugging tools*. Ph. D. Dissertation. Stanford University. Retrieved 2023-01-03 from <https://www.proquest.com/docview/302762122>
- Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on Software Engineering* 42, 9 (Sept. 2016), 805–824. doi:10.1109/TSE.2016.2532875
- Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. doi:10.1145/2491956.2462183
- Mark Shinwell. 2016. libmonda: Make OCaml native debugging awesome. Retrieved 2022-10-18 from <https://mshinwell.github.io/libmonda/>
- Chris Siebenmann. 2022. Print based debugging and infrequent developers. Retrieved 2022-08-01 from <https://utcc.utoronto.ca/~cks/space/blog/programming/PrintDebuggingAndInfrequentDevs>
- Jeremy Singer. 2022. Introduction. In *SSA-based Compiler Design*, Fabrice Rastello and Florent Bouchez Tichadou (Eds.). 3–12. doi:10.1007/978-3-030-80515-9_1

- 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
- Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-based translation validator for LLVM. In *Computer Aided Verification (Lecture Notes in Computer Science)*. doi:10.1007/978-3-642-22110-1_59
- J. Ryan Stinnett and contributors. 2019. Malleable Systems Collective. Retrieved 2022-02-23 from <https://malleable.systems/>
- J. Ryan Stinnett and Stephen Kell. 2024a. Accurate coverage metrics for compiler-generated debugging information. In *Proc. of CC '24*. doi:10.1145/3640537.3641578
- J. Ryan Stinnett and Stephen Kell. 2024b. Accurate coverage metrics for compiler-generated debugging information (artifact). doi:10.5281/zenodo.10568392
- J. Ryan Stinnett and Stephen Kell. 2026a. Debugging debugging information using dynamic call trees. *Proc. ACM Program. Lang.* 10, OOPSLA1 (April 2026), 105:1–105:27. doi:10.1145/3798213
- J. Ryan Stinnett and Stephen Kell. 2026b. Debugging debugging information using dynamic call trees (artifact). doi:10.5281/zenodo.18392053
- Sun Microsystems. 2001. JPDA enhancements in the Java 2 SDK, v1.4. Retrieved 2026-03-13 from <https://web.archive.org/web/20020612123152/http://java.sun.com/j2se/1.4/docs/guide/jpda/enhancements.html> Product documentation (archived).
- Éric Tanter. 2004. *From metaobject protocols to versatile kernels for aspect-oriented programming*. Ph. D. Dissertation. University of Nantes and University of Chile. Retrieved 2024-11-01 from <https://pleiad.dcc.uchile.cl/papers/2004/etanter-phd.pdf.zip>
- Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. doi:10.1145/1480881.1480915
- Philip Tchernavskij. 2019. *Designing and programming malleable software*. Ph. D. Dissertation. Université Paris Saclay. Retrieved 2019-11-29 from <https://theses.hal.science/tel-02612943/>
- Caroline Tice. 1999. *Non-transparent debugging of optimized code*. Ph. D. Dissertation. University of California, Berkeley. Retrieved 2024-08-27 from <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
- Frank Tip. 1995. A survey of program slicing techniques. *J. Program. Lang.* 3, 3 (1995).

- Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. 2019. Interactive metamorphic testing of debuggers. In *Proc. of ISSTA '19*. doi:10.1145/3293882.3330567
- Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating value-graph translation validation for LLVM. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. doi:10.1145/1993498.1993533
- 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
- Vector 35. 2015. Binary Ninja. Retrieved 2025-11-22 from <https://binary.ninja/>
- Tom Wallis. 2024. *Aspect-oriented modelling*. Ph.D. Dissertation. University of Glasgow. doi:10.5525/gla.thesis.84511
- Theodore Luo Wang, Yongqiang Tian, Yiwen Dong, Zhenyang Xu, and Chengnian Sun. 2023a. Compilation consistency modulo debug information. In *Proc. of ASPLOS '23*. doi:10.1145/3575693.3575740
- Yanzhao Wang, Fei Xie, Zhenkun Yang, Jeremy Casas, Pasquale Cocchini, and Jin Yang. 2023b. An automated verification framework for HalideIR-based compiler transformations. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. doi:10.23919/DATE56975.2023.10137308
- Elaine J. Weyuker. 1982. On testing non-testable programs. *Comput. J.* 25, 4 (Nov. 1982). doi:10.1093/comjnl/25.4.465
- John Whittington. 2020. *Debugging functional programs by interpretation*. Ph.D. Dissertation. University of Leicester. doi:10.25392/leicester.data.12696617.v1
- John Whittington and Tom Ridge. 2019. Direct interpretation of functional programs for debugging. *Electronic Proceedings in Theoretical Computer Science* 294 (May 2019), 41–73. doi:10.4204/EPTCS.294.3
- Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proc. of PLDI '11*. doi:10.1145/1993498.1993532
- 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 Walter Zurawski. 1990. *Interactive source-level debugging of globally optimized code with expected behavior*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign. Retrieved 2022-06-27 from <https://www.proquest.com/docview/303862035>
- Lawrence W Zurawski and Ralph E Johnson. 1991. Debugging optimized code with expected behavior. Retrieved 2026-08-12 from <http://www.cs.cmu.edu/afs/cs/project/clisp/hackers/ram/work/rtl/debugging-opt-code.ps> Unpublished draft.