Source-Level Debugging of Compiler-Optimised Code: Ill-Posed, but Not Impossible

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

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

Abstract

Debuggability and optimisation are traditionally regarded as in fundamental tension. This paper disputes that idea, arguing instead that it is possible to compile programs such that they are both fully source-level-debuggable and fully optimised, and that the essential problem to be solved is loss of state. Although these two properties are usually not achievable at the same time, it argues the feasibility of providing the desired one 'on demand', and that metadata-based approaches extended with residual state can do so in a manner that generalises beyond dynamic deoptimisation. Correctness of debugging metadata is introduced as an ill-posed problem, a partial correctness criterion is proposed, and further approaches are discussed.

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

Keywords: compilers, debug information, optimisation

ACM Reference Format:
Stephen Kell and J. Ryan Stinnett. 2024. Source-Level Debugging of Compiler-Optimised Code: Ill-Posed, but Not Impossible. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '24), October 23–25, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3689492.3690047

1 Introduction

Programming language implementers must grapple with an essential conundrum. Humans write source code, while machines execute something lower-level. Yet to understand the program they *intend* to write, mortal programmers need to *see their code execute.* How can the implementation enable this?

Source-Level Debugging is a phrase partially capturing this requirement. It calls to mind a useful but narrow class of stepping-based interactive debugging tools. However, many further tools embed the same requirement: those for source-level tracing, profiling, crash reporting, introspection, visualisation, security auditing, and in general any tool offering a *source-level understanding* of an object-level execution. Put differently, this includes any dynamic analysis whose results are in source-level terms.[1] We will say 'debugging', but intend it as a shorthand.

Compiler optimisation and source-level debugging have long been seen as in fundamental tension. To a rough approximation, unoptimised code debugs well and optimised code doesn't. A workaround is to disable optimisations, but the optimised code is often what matters: it is what is deployed (e.g. consider the need to trace a production system), it may be the only usable option (e.g. a game or other real-time system that is too slow unless optimised), and understanding its performance may be the goal (e.g. reviewing a source-level performance profile).

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, 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.

Onward! '24, October 23–25, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1215-9/24/10
https://doi.org/10.1145/3689492.3690047

[1] Even this may be too narrow. While 'analysis' implies a 'read-only' tool, that observes but does not mutate execution state, certain tools do mutate it. This is familiar even in debuggers, such as with 'edit-and-continue' features or the ability to update program variables. In this paper we will restrict ourselves to observation-only tools, but acknowledge that the mutating kind can bring additional value.

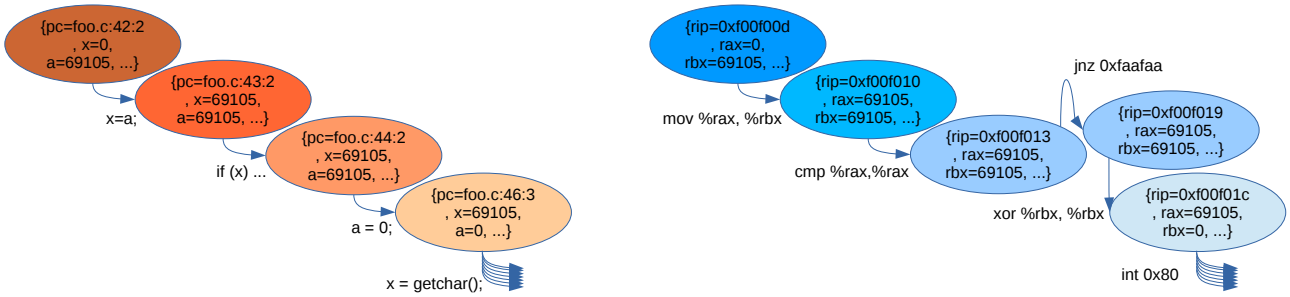


Figure 1. A piece of the concrete transition system (left) for a simple source program (to appear in Fig. 2), and the corresponding piece of the object program's transition system (right). *State mapping* means capturing the correspondence between these states. Here, this includes that `a` is in `rbx`, `x` is in `rax`, and that `pc` maps to `rip` as shown by the shades of each bubble.

These issues arise to an degree correlated with the level of optimisation, again appearing to affirm the tension.

This paper explores and explodes that tension. It need not be true that 'more debuggability means less optimization'; rather, we posit that the true trade-offs are better seen as over a different and larger set of variables. Holding optimization constant, full source-level debuggability *is* achievable—if one is prepared to allow the debugger to be more stateful, more complex, or in some senses less timely. The issues arising are confounding to the 'classically trained' language implementer, in that adequate vocabulary is lacking and the overlap with better-understood problems, such as verified compilation, is less than it might seem. Our contributions are as follows.

- We elaborate source-level debugging in terms of *state mapping*, further developing the idea of *residual computation* found in recent literature, and proposing the mapping's *density*, *height* and *fidelity* as a better framing of the classical optimisation–debuggability tension.
- Using the same framing, we consider the *temporal imprecision* of metadata-based debugging under code motion optimisations, and sketch what we believe to be the first automatable correctness criterion for local variable information as emitted by compilers performing such optimisations. Although only a partial correctness property, we argue that it is respected by most compiler passes—but not all, and we identify the necessary criteria.
- Having motivated the view of debugging in *concurrent* terms, where observation occurs in parallel with the optimised computation, we argue that debugging tools can only be called correct up to some defined fidelity property. We suggest how fidelity might be specified by borrowing ideas from concurrent memory models.

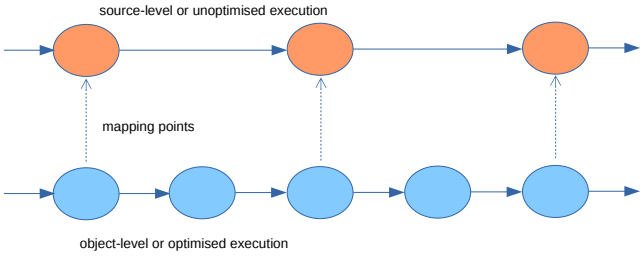
2 State Mapping: Two Approaches

We first view debugging in terms of transition systems.

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 our diagrams) stack and heap. Each state space is then, roughly, a Cartesian product of these many components. Fig. 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).

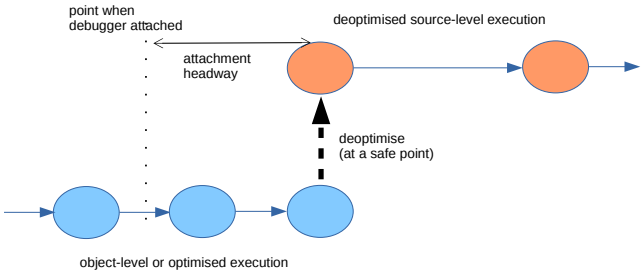
Since the target (object) machine is lower-level than the source (abstract) machine, the state spaces are not in bijection. There may be *mapping points* where the object program's state corresponds precisely to a source-level state.



2.2 Approach 1: Dynamic Deoptimisation

Dynamic deoptimisation [17] '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.



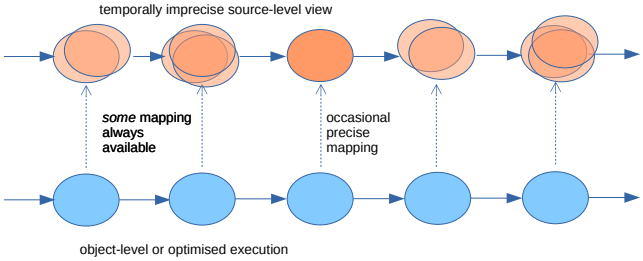
The *density* of these mapping points defines the granularity of *simulation* that must exist between optimised program and source program. Here it is clearly denser (finer-grained) than the input–output relation, but not so fine-grained as to offer a mapping at every instruction.

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.

2.3 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 [21] or the more modern DWARF [13] and CodeView [22]. 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 our 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 [30]. 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-powerful.

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, which we revisit in §7). The problems are not limited to temporal precision, however: in practice local variables are often missing or simply incorrect [1, 28]. 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. As we will explore, there is a fundamental issue relating to *retention of state*, but when considering fidelity generally, other more practical engineering issues appear comparable in significance. Metadata is of unwieldy design, comes with no strong correctness or

```
    ...
42: x = a;
43: if (x) {
44:     a = 0;
45:     x = getchar();
        ...
```

```
(dbg) break foo.c:43
(dbg) run
Hit breakpoint 1
43: if (x) {
(dbg) print x
69105
```

```
(dbg) break foo.c:43
(dbg) run
Hit breakpoint 1
43: if (x) {
(dbg) print x
<variable optimized out>
```

Figure 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 20210110
 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
...
```

Figure 3. DWARF value ('info') information (left, middle) and line table (right) for a plausible compilation of the code in Fig. 2, encoding the correspondence between the two sides of Fig. 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.

completeness 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-powerful 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 [13] format, the most expressive in widespread use.

3 Understanding Debug Information

Debug info has two main parts, which we will call *line* and *value* information.[2] Both encode *mappings* between states of the source program and states of the object program. Strictly we should say 'components of states', recalling our 'Cartesian product' from earlier (§2.1).

Line 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.

Value information addresses the local and global variable components: it includes where variables are located, and how the bits and bytes encode its source value. The latter information is called 'type information' but takes a form largely agnostic to the source language. 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. 3 shows a very simple example in DWARF based on the code in Fig. 2.

DWARF-generating compilers consider it a bug if compiling with debug info enabled (`-g`) changes the generated code [33]. In other words, enabling debugging does not curtail optimisation in any way.

[2] A third part, frame information, differs in that it is purely a machine-level construct: it maps a callee's stack and registers to their state at the time of entry to the current call, and so enable stack-walking without frame pointers. We do not consider it here, but it has been studied recently [2].

```
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

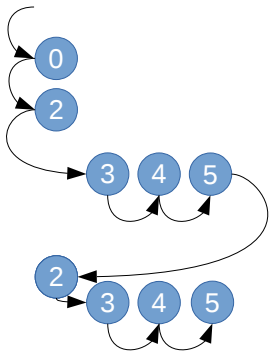


Figure 4. Left: a simple example function in C. Middle: simple debug info for a relatively unoptimised compilation on a fictional architecture; we assume locals are allocated to registers which are plentiful. Right: the *source-level* concrete progression of variable `i`, for an execution where `MAX` is 5 and `get_start()` returns 2.

In totality, the debug info maps states to states, as we have been conceptualising in terms of transition systems. However, each record of line or value 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 the trivial function `OP_reg0` but 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-powerful stack machine language. This allows it 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`.

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 [1]. Previous work has characterised debug info as capturing a *recovery function* [36] or *residual computation* [28]. 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 our *state-mapping* framing further in order to answer these questions.

4 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 fictitious simple piece of code, shown in Fig. 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.

```
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 or value	range
data	in r1	at all points
arg	r2	at all points
i	value 0	from 3 to 5
	r3	from 5
tmp	r4	from 5
out1	r5	from 3
out2	r6	from 3

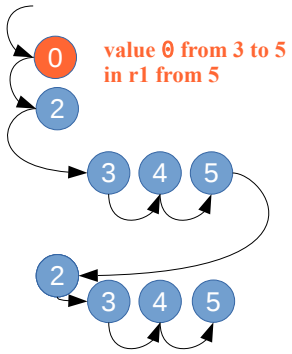


Figure 5. Left: a simple 'zeroing elimination' optimisation on the code of Fig. 4. Middle: how debug info can *residualise* the zeroing. Right: the *source-level* concrete view of variable `i` recoverable at debug time using residual computation, for the same execution as Fig. 4. Orange denotes residual artifacts.

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

local	location or value	range
data	in r1	at all points
arg	in r2	at all points
i	value 0	from 3 to 5
	value (r7-r1/4)	from 5
tmp	in r4	from 5
out1	in r5	from 3
out2	in r6	from 3
p	in r7	(synthetic)

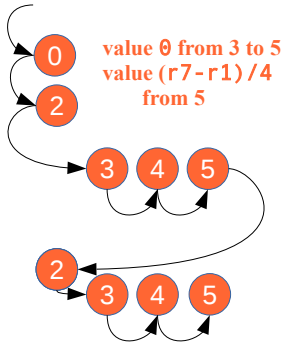


Figure 6. Left: a more complex optimisation (strength reduction) on the code of Fig. 5. Middle: how debug info can residualise the strength reduction, resurrecting `i` which is no longer directly present in the object program. Right: the *source-level* concrete view of variable `i` again recoverable at debug time using residual computation. The synthetic temporary `p` is assumed to reside in register `r7`. Orange denotes residual artifacts.

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. 5 with a simple optimisation to remove the redundant zeroing of `i` on line 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, in our example, 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.

Next, in Fig. 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 our 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`).

5 The Essential Problem: Loss of State

Our 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 statements or sequence points of the source language.) Secondly, not only are the control states injectively mappable, but execution continues to visit these in the same order: 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* to replace it with other. Of course non-control state is commonly also dropped, such as when local variables found to be 'dead' during register allocation are discarded from the program's run-time state (as with our `err` example in §3).

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

```
if (cond) {
    ...
    ++x;
} else {
    ...
    ++x;
}
```

… to become:

```
if (cond) {
    ...
} else {
    ...
}
++x;
```

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 [36] 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 our case `cond`. Dwarf's current line table happens not to be so indexable. (This recalls our 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 our earlier `i = 0` example. Suppose:

```
21: t = f();
22: int i = t + 1;
23: for (i = x; ...)
```

… is optimised to:

```
21: t = f();
22: int i;
23: for (i = x; ...)
```

… 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 will 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 [31] of `t+1`. We can stop adding to our 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-powerful metadata 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.

Is there a way between the horns of this dilemma? State-dropping optimizations, 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 that pithily summarises many aspects of the problem, Brender et al. [6] wrote as follows.

> Unfortunately, code-motion-related optimizations 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?

Our 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 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.

In such an approach, there is no longer a trade-off between optimization and debuggability! *In extremis*, *both* fully optimized and fully unoptimized 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 [8], and there is a literature on faster approaches to communication than the standard `ptrace()`-based trap approach [18].

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.

In the rest of this paper, we survey the consequences of this. Firstly, we argue that correctness of debug metadata becomes an ill-posed problem, in need of further specification regarding dimensions such as its *density* and *height* of mapping. Secondly, we note that although DWARF-based debugging has not yet reached this full stateful design, it is well on the way, suggesting that our design is reachable at least somewhat incrementally; we survey the relevant foreshadowings. Thirdly, what can be done to move current infrastructure closer towards it, especially starting from the current *temporally imprecise* behaviour? Finally, what further understanding must be gained, and connections developed, to make this part of a predictable and productive development infrastructure?

6 Correct Debug Info: An Ill-Posed Problem

Given the complexity of both current and hypothetical debugging systems, and the patchy state of present ones, a correctness property is highly desirable. It is a tempting to try to borrow this from more familiar program analysis problems relating multiple realisations of a program. However, this does not work—or, at least, debug info correctness is *not* equivalent or reducible to any of the following such problems.

Decompilation means recovering an equivalent source representation of an object program. Since a source program does not include a state mapping, even a hypothetical perfect decompiler that could recover the original source code would not help to enable debugging at all. Indeed the original source code *is* routinely available at debug time, but only *in addition to* debug info providing the state mapping.

Verified compilation means showing that a compiler always generates an object program whose input/output behaviour correctly refines that of a valid source program. However, such a compiler need not produce or retain any mapping between source and object states, nor prove anything about such a mapping except at input/output actions.

Translation validation means showing the input/output behaviour of one program is preserved or refined by a transformed or translated program. It can be done end-to-end [23] or at pass-by-pass level [20]. Again, such a tool need not produce or retain any mapping between source and object states, nor prove anything about such a mapping except at input/output actions.

The essential reason is that these problems are framed in terms of the program's input/output behaviour. However, a 'debugger' that could only step between input or output actions, not any intervening points in execution, would not be considered worthy of the name. Put differently, it's axiomatic that *a denser mapping* is needed for debugging. What should this mapping be? When is it correct? This cannot be answered in input/output terms, since it concerns the *incidentals* of how the programmer realised their source-level version. These incidentals—computation the object program *does not do* but the source program does, internal states that are not observable from the object program's input/output behaviour—are defined to be irrelevant in all the above problems. Yet it may be only from observing these 'incidentals' that the programmer can gain the source-level insight they seek. (Whereas, to paraphrase Perlis, a low-level programming language requires attention to the irrelevances of the machine, one could say high-level debugging *must enable* attention to the irrelevances of the human.)

We want a *dense* mapping, that can relate back to source level more than just input/output events. We also ideally want a *tall* mapping, reaching all the way up to source level, e.g. showing us the index variable `i` not the synthetic pointer `p` in our earlier example. *Adding density* risks losing temporal precision (one may see a composite of many source states, perhaps together with compilation artifacts). *Adding height* may compute a closer-to-source view but departing further from what actually runs, making it less appropriate for some tools' observation purposes (a profiler that counts zero-initialization overhead would not want to count the residual zeroing of our variable `i`) or even just for debugging some bugs (if we suspect a compiler bug, we want to focus on what actually happens).

Standard debugger features become underspecified in this light. Should a watchpoint on `i`, in our example from Fig. 5, fire when it is residually zeroed, or only when it is assigned 'for real'? And since, after strength reduction, it never *is* assigned 'for real'—only `p` is—should the watchpoint ever fire and, if so, what should the debugger show? If a compiler has optimised:

`x++; x++; x++;` … into: `x += 3;`

… should the watchpoint fire one or three times?

All these are questions that *can* be answered reasonably in many ways. Without explicitly specifying the desired view of execution, in terms of height and density and temporal precision, correctness is ill-posed. We view this as a more essential issue than 'source-level debuggability trades off against optimization': the real issue is that a *single* view cannot be fully faithful both to source and object program at the same time. But multiple views can be available! We should not suppose otherwise.

Another way to view this is that what is debugged is *always* a fiction, and one in need of careful specification. Indeed this fiction stretches down to the hardware level: mapping at instruction granularity relies on a hardware-provided illusion of single-stepping. Our notion of temporally precise(-or-not) mappings has an analogue in computer architecture's notion of *precise exceptions*, and there are interesting parallels in how modern out-of-order processor cores wrangle this. We explore these and other similarities next.

7 Statefulness Foreshadowed

Although a debugging toolchain based on stateful residual computation is yet to be *designed* per se, this does not mean it does not exist. Debugger toolchains have already departed on several fronts from a purely stateless 'recovery function' paradigm. We survey these here.

Record–Replay and Omniscience. Record–replay systems and omniscient debuggers necessarily retain state, to allow past program states to be recovered. Replay is fairly widespread, thanks both to basic instruction-wise emulators (stock `gdb` has included one since 2009 [15]) or more efficient and sophisticated recording including that of `rr` [25] and UndoDB [14]. Whereas replaying or reversible debuggers show only a single state at a time, a more fully 'omniscient' system such as Pernosco [24] supports queries over an entire execution trace. (By contrast, in a replay or reversible setting, revisiting the past remains an indirect process of setting breakpoints or watchpoints and collecting the results at specific states.) The stateful residual computation we have envisaged, with its 'attachment headway', can be seen as an 'attach-on-demand' version of record–replay, whereas existing implementations require attachment from program start. Although we framed residual state around recording state dropped by the object program, the dual approach of recording nondeterministic input as it *enters*, and using replay to regain subsequent states, allows greater efficiency. Still-greater efficiencies can be obtained from system-wide approaches [11]. An efficient implementation of residual state would likely employ some hybrid of these approaches

to minimise the communication overheads, similar to the parallelised approaches to dynamic analysis mentioned earlier [8].

Location Views. Given a compilation which optimises (for example) `x++; x++; x++;` into `x+=3;`, the *location views* extension [26] allows the debugger to step over the elided states. This requires the debugger to track state that the object program does not, roughly as 'fractional program counter' values. By this approach, any statically enumerable straight-line sequence of states that have folded to a single object program state can be reconstructed in the debugger. However, it can *not* reconstruct a branching computation (`if (x % 2) x--;` becoming `x &= ~1`) nor a sequence of data-dependent length (e.g. after 'arithmetic progression' optimisation eliminating a summation `for`-loop). These would be useful test cases for a fully realised residual state approach.

Entry Values. A common debugging need is to see a value as it was on entry to a call, noting that the current value may have changed. This motivated the specification of a DWARF operation `DW_OP_entry_value` whose specification is that it materialises such an 'entry value' on the stack—without saying how this is done. We do not know of a debugger which currently implements it in non-trivial cases, but the vagueness of the specification alludes to the possibility of additional state being kept to support the operation in other cases. Indeed, an omniscient debugger could already do so.

Knowledge Extension (a.k.a. Eviction Recovery). By analogy with all-knowing 'omniscience', an enhanced 'knowledge-extending' debugger was hypothesised in our recent paper about how to measure debug info's coverage of local variables [28]. This debugger would retain certain local variables for longer than the object program does; we noted as follows.

> a hypothetical debugger could be implemented today that performs this knowledge extension, by saving a variable's value at its last moment of knowability…

This envisages a state-saving mechanism based on hidden breakpoints, set at the last instruction(s) where a value is available. In that paper, this technique was merely simulated as a way to calculate the coverage gains it could enable. We later discovered that Tice [29] implemented exactly this technique in the Optdbx debugger (based on SGI `dbx`) and noted the performance trade-off faced especially when using conventional slower (trap-based) breakpoints.[3] A fast implementation would likely improve on this using some mixture of fast breakpoints [18], input-recording and parallel-analysis techniques—although even the slower implementation would still only cause slowdown on demand, and would usefully allow a deployed, fully optimised binary to be much more debuggable than at present. (This contrasts with how the LLVM community has recently proposed a reduction in optimisation as the default for the `-Og` optimisation level.[4] Although incurring only minor slowdown, the slowdown affects all runs of the deployed binary. Breakpoint-based communication overheads, by contrast, although typically greater, are incurred only when attached or enabled—this is the same trade-off adopted by DTrace's 'probes' design [7].)

Learning from Hardware. State dropped can be compensated by state kept. Modern high-performance CPUs exploit this, in order to residualise code motion perfectly: despite performing copiously out-of-order execution internally, a trap at any instruction will expose the registers in a sequentially consistent state, having been 'fixed up' on the trap using information kept in the CPU's reorder buffer. The distinction between 'precise' and 'imprecise exceptions' was a central feature of the 1967 algorithm of Tomasulo [32]. Reorder buffers [27] hold the state necessary to add to present the precise illusion (also allowing rollback of speculative executions). It would appear that a stateful debugger *could* residualise code motion, and any other state-dropping optimisation, by retaining the state in an analogous way. For example, a stateful debugger might include the equivalent of a reorder buffer: tracking 'source constructs in flight', retaining the necessary amount of past state, and using this to provide the illusion of source-level 'precise interrupts', such that pending operations were rolled back and a fully consistent source state were always shown.

8 Correct Modulo Temporal Imprecision

As a consequence of defining a mapping at every instruction, yet permitting the compiler to perform *any* code motion within the input/output behavioural envelope, current debug info is temporally imprecise. The interrupt-inspired solution we just sketched is an ambitious goal that would require extensive new DWARF metadata describing what state to save. Instead, an intermediate solution would be to define a partial correctness property that factors out temporal imprecision and detects only other infidelities, such as outright incorrect or missing information. Such a property would be expected to hold even under the imprecise, artificially dense 'full-instruction' mapping of existing debug info, and would fix a 'mapping height': at source level, modulo allowable temporal imprecisions. Such a property would provide an automatable test oracle useful for catching incidental fidelity problems in compiler-generated debug info. In order to sidestep the ill-posedness we remarked on earlier, we accept that such a property will necessarily be both specialised and partial.

In this section we sketch such a property and why we believe it can be the basis of an automatable test oracle for

[3] There it was called 'eviction recovery'—although we prefer to reserve 'recovery' for stateless pure functions, consistent with our framing earlier that contrasted 'recovery' with stateful 'residual computation'.

[4] See https://discourse.llvm.org/t/rfc-redefine-og-o1-and-add-a-new-level-of-og/72850 as retrieved on 2024/9/1.

debug info. Our property concerns all source variables, and expects full fidelity in the view of them that the debug info offers—modulo, of course, the temporal imprecision that is permitted. We introduce the property by returning to our example from earlier and considering some less 'convenient' optimisations that perform code motion.

8.1 Per-Variable Histories

The property is one we have already witnessed: in Fig. 4 and Fig. 5, the right-hand part of the figure showed a concrete history of values for a single variable (in this case `i`). Although not shown, we can also consider the arcs of this history to be labelled with the source coordinates where each update occurs, according to debug info, thereby testing both line and value information (§3).

Our property is simply that the histories observed for all local variables, on any given execution, are the same as in the source (or unoptimised) program. In other words, in the optimised code as viewed through its debug info, each source variable is seen *individually* to progress through the same sequence of values as in the source program, including that each change of value occurs at the correct source file and line. It does not capture any kind of coherence *across* variables.

8.2 Optimisations Tolerated

The property is invariant under dead code elimination (it concerns only reachable paths). It is also invariant under common subexpression elimination (CSE) since by definition subexpressions output to temporaries rather than source variables. If two source variables are themselves commoned (merged), they are can remain distinct in debug info (but still share machine state). Code folding, such as our `x++` example, must be residualised by location views in order to recover folded assignments (the `x+=3` substitution). This does mean we expect to flag as missing any assignments occurring in *eliminated branching control flow*, such as our `x&=1` example. We view this as an incidental limitation of current DWARF and therefore fair to report as a bug.

Note that the property is not affected by dropping of dead local variables, such as by a register allocator, because the history consists of assignment events; it does not matter if the assigned value is eliminated later on becoming dead. If the assignment is elided entirely, because it is never read (or read only by folded or CSE'd code), we expect it to be residualised by a location view (recalling §7); our hypothesised tool will flag a problem if the intermediate state is not made visible in this way. A flip side is that our property does not capture whether a given assignment remains available for its full source-level lifetime. Debug info which drops a variable 'early' would arguably be a missed bug—although subject to the knowledge extension or 'eviction recovery' solutions mentioned earlier (§7) and already captured reasonably well by a coverage metric we have defined in earlier work [28].

Since each variable is checked individually, code motion is tolerated: if one variable's assignments are hoisted or sunk past another's, each variable's individual history graph is unchanged. In this way, the compiler gains a lot of latitude in code scheduling, but may still be held to a high standard of debug info.

Although the property is also not affected much by control state dropping, such as merging identical code paths, the matching of source coordinates may need to be fuzzy in order to accommodate 'control merging' (§5). Again, this is a limitation of current DWARF. Both alternation-based fuzziness ('either at line 3 or line 10') and range-based fuzziness ('between lines 5 and 7') appear necessary in practice.

8.3 Optimisations Barely Tolerated

What about all the other compiler optimisations? We now consider one that does *not* respect the property. Fig. 7 shows a loop fusion transformation applied to our example from earlier. This has truncated the history of local variable `i`, since it no longer progresses through the index sequence of the second loop. (There is now only one loop!) Brender et al. [6] again provide a pithy insight.

> A variable is said to have split lifetimes if [its] set of fetches and stores … can be partitioned such that none of the values stored in one subset are ever fetched in another subset. When such a partition exists, the variable can be 'split' into several independent 'child' variables…

The context of the quotation was to motivate debug info that can capture *independent allocation*, also known as 'wandering variables', where a variable moves among many locations in its lifetime. An extreme kind of partitioning (albeit semantically different from that described) is the well-known Static Single Assignment (SSA) form, where a distinct 'subvariable' is created at each assignment. Whereas the SSA transformation can be seen as 'breaking to remake' the source program's structure, our desired subvariable property is as quoted above, and mirrors debugging in seeking instead to preserve the source structure—relaxing to break it 'just enough' to match legitimate code motion, and accepting the detriment to debug-time temporal precision. Only a very particular kind of code motion requires this relaxation at all: when assignment to a 'subvariable' is moved past that to another subvariable of the same origin variable. When two such peer 'subvariables' are scheduled independently by the compiler in this way, the origin variable's history may become 'interleaved with itself', meaning its source-level history is not preserved. However, separating the subvariables disentangles this interleaving; we *do* still expect to see each subvariable's history preserved. Fig. 8 illustrates. This case of code motion is somewhat uncommon because more often, a

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

local	location or value	range
data	in r1	at all points
arg	in r2	at all points
i	value 0	from 3 to 5
	value (r7-r1/4)	from 5
tmp	in r4	from 5
out1	in r5	from 3
out2	in r6	from 3
p	in r7	(synthetic)

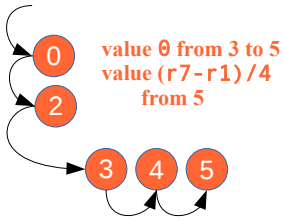


Figure 7. Left: adding a loop fusion optimisation to the code of Fig. 6. Middle: one plausible but problematic version of the debug info (same as Fig. 6). Right: the truncated *source-level* concrete history of variable `i` recovered using this (unchanged) debug info. Orange denotes residual artifacts.

```
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 (int j = tmp; j < MAX; ++j)
12  {
13    out2 &= data[j];
14  }
15  g(out1, out2);
16  return tmp;
17}
```

local	location or value	range
data	in r1	at all points
arg	in r2	at all points
i (1)	value 0	from 3 to 5
	value (r7-r1/4)	from 5
i (2) as j	value 0	from 3 to 5
	value (r7-r1/4)	from 5
tmp	in r4	from 5
out1	in r5	from 3
out2	in r6	from 3
p	in r7	(synthetic)

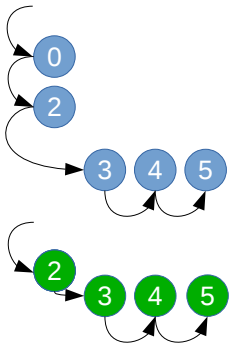


Figure 8. Left: the example code again, but with a substitution of `j` as a 'sub-variable' of `i`. After optimisation (not shown here) the two loops are still fused. Right: the two per-subvariable histories for subvariables of the original `i`, now `i` and `j`, restoring the source-level correspondence. Middle: a 'subvariable'-aware version of the debug info, showing that `i` and `j` are modelled simultaneously by the same object program state. Their separate identities ensure their histories are calculated separately, and the debug-correctness property can be expected to hold of the loop-fused code. The new subvariable is shown in green.

variable will be assigned a new value that depends on its previous value. It is the lack of such a dependency that enables the loop fusion in this example, despite the re-use of `i`. Indeed a more idiomatic version of the second loop would use a fresh index variable, say `j`, rather than re-use `i` from earlier, effectively performing the subvariable transformation in the original source.

8.4 Optimisations Not Tolerated

Are there examples of transformations that intentionally *do not* satisfy our property? One such is loop tiling. Consider a nest of two loops iterating over a potentially large amount of data.

```
int total = 0; unsigned i, i;
for (i = 0; i < big_number; ++i)
{
    for (j = 0; j < another_big_number; ++j)
    {
        total += a[j][i];
    }
}
```

This code models a classic inefficiency relative to C's row-major layout of a rectangular array. The code will visit memory in a strided fashion, accessing one element from a row and then moving to the next row which is distant in memory, eventually returning to visit the second column position of the first row, and so on. This could generates the (i, j)

(column–row) sequence $(0,0)$, $(0,1)$, $(0,2)$, …, $(1,0)$, $(1,1)$, and so on, where `i` progresses monotonically up whereas `j` visits a given index value many times. A compiler could 'block' or 'tile' the two loops so that array elements are visited in a more cache-optimal order: $(0,0)$, $(1,0)$ and so on. Under plausible debug info for `i` and `j`, it would be `j` that progresses monotonically up and `i` that visits its values many times. If the object program is transformed in this way, there is no way to recover `i` or `j` such that they progress through the same sequence of values as in the source program. Things simply happen in another order in the transformed program. In such cases it would be preferable for the compiler to *flag the intentional omission of* one or both of these variables. Since it's inherent to the optimisation that the index variable is destroyed—it is no longer modelled in the object program—one sensible outcome might be for a debugger to find this recorded in metadata, so that it can display a helpful message, ('variable expressly removed by compiler pass `tile`') rather than the current untrustworthy guess ('variable optimized out?'). An automated test would expect this explicit metadata, or, on its absence, flag the apparent failure to recover `i` or `j`.

Of course, all this simply revives the trade-off we claim can be eliminated: the optimised code is less debuggable. To correct this, we require stateful residual computation: the entire original (untiled) loop order would be emitted residually. In this way, the programmer could still optionally observe execution progressing the unoptimised way, and the view of execution would rejoin the 'base' (optimised) execution at a subsequent mapping point. The trade-off is that the user has observed a substantially different execution to what actually occurred in the base program. Assuming the user sought to understand the meaning of their code, this switch is appropriate. If they wished instead to understand its cache behaviour, it would not be. Hence we come back to the ill-posedness of the 'source-level debugging' problem: each tool, user or bug may have different requirements. However, *multiple* views can be made available at once; we need not fix just one, and need not choose ahead of time.

8.5 Practical Experience

Although ill-posedness means there are many 'right answers' to what debug info could be called correct, there are also plenty of wrong answers. Compilers frequently generate them! Unlike miscompilation bugs of the kind caught with the help of Csmith [34] and similar tools, which are usually the result of many compute-hours spent hammering the compiler with unusual programs, incorrect or incomplete debug info is simply the norm. We believe this is familiar to most developers who debug the optimised output of GCC, LLVM or similar compilers. Although the viability of our property as an automated test oracle remains a hypothesis, our experience to date includes building a tool checking a similar property (a precursor of the one we have described here). This tool has significant power to find confirmed bugs in the LLVM compiler. The remaining challenge is to do this in a precise way that avoids overapproximation, and therefore provides a low rate of false positive reports—noting that our property is a dynamic one, over execution traces, so should only be flagged for feasible executions.

9 Choose Your Illusion

If each tool, user or bug may have different requirements, how can we state these precisely? Whereas our 'temporally imprecise' property sidestepped many such questions, tackling them is a prerequisite to obtaining any thorough assurance about debugging infrastructure. We also believe it is likely to illuminate better debugging tool designs and more usable and reliable compilers.

Levels of Illusion. Past work has identified 'truthful' versus 'expected' as a binary distinction between approaches to source-level debugging [35]: a debugger can either provide a 'truthful' view, reflecting object code behaviour (but perhaps projected opportunistically up to source level where possible), or it may provide an 'expected' view that directly mirrors a naive implementation of the source language. Toolchains largely adhere to the 'truthful' model, whereas virtual machines use dynamic deoptimisation to achieve the 'expected' view. In effect we are motivating a more fine-grained and specific framing: *which source program properties* need to be preserved in a given debug-time view, and conversely, *which object program properties*?

The closest specifications of such properties we are aware of are memory models for languages and architectures, such as the C++ memory model [3]. In the case of limiting code motion, it seems likely that some ideas can be lifted directly: our per-variable property is essentially a 'needlessly partial' partial order on updates, retaining only the edges connecting accesses to a given variable (and even then allowing splitting to 'subvariables' when a certain partitioning was feasible).

A debugging tool that can switch between such multiple 'illusions' at debug time largely refines existing features for switching language: the 'bottom-most' illusion is only of assembly-level debugging, and the 'top' of a perfect naïve source view. A valid 'intermediate' debug-time view of execution might preserve some properties from the optimised object program and some from the naive source semantics. For example, a user may wish to see or not see compilation artifacts of multi-instruction stores (e.g. the 'adjacent data' scenario described by Boehm [5, §4.2]), may wish to follow or not follow folded or CSE'd control flow, may wish to remember program-dead local variables or not (noting the on-demand slowdown for enabling this) and so on.

Compiler Shifts. As optimised and unoptimised executions diverge, we cannot preserve both at once, in the sense of *observing* a single execution faithful to both. We have relied

heavily, however, on the possibility of switching between multiple views. To a debugger user, a 'default setting' that errs on the source-level side is likely to suffice in common cases; non-defaults might suit advanced users working with performance-sensitive workloads, those hunting compiler bugs, and so on. An important wider motivation for characterising these 'intermediate illusions', however, could be to achieve a shift from characterising degrees of optimisation in terms of what passes are performed, instead into terms of how far the object program is allowed to stray from a naïve source semantics—both in what it *does* at the object level, and what it can be observed to do at debug time (perhaps with the help of residual computation).

This is not merely a debugging concern; to systems programmers, it affects programs' semantic correctness, especially at the 'edges' of a language where memory and/or assembly code are used for communication with hardware or other-language code. Current approaches offer workarounds rather than solutions: a compiler officially holds only the input/output behaviour sacred, but the user may still tweak the active passes, or add compilation options (e.g. `-fno-strict-aliasing`)—in the hope that these changes add up to the particular effect they require (e.g. to tolerate type-violating aliasing that is intentionally present in their code) but with no guarantee. The existence of `volatile` in C is testament to this: it is essentially an inhibitor of optimisations, limiting the compiler to a naïve execution semantics, the 'abstract machine'. The shift we seek is specifying optimisations not pass-by-pass but as an envelope. This would be a major shift among compiler implementations, but plenty of motivation for this is already well-known especially in real-time, safety-critical and cryptographic (e.g. timing-sensitive, zeroing-sensitive) use cases. Such users often fall back on 'big-hammer' restrictions, such as 'use only a certified compiler' or even 'program only in assembly code', precisely owing to compilers' lack of contractual optimisation envelopes.

10 Related Work

Our discussion so far has surveyed much prior work; we briefly mention some further notable work here.

The earliest research into debugging optimised programs takes the general approach of propagating additional mapping information through the compiler [10, 16]. The value expressions (§3) of DWARF version 3 [9] and similar formats, extend this idea further, by allowing arbitrary functions to be computed in the debugger to recover values eliminated from the object program. This escalation in expressiveness was only later met (partially) with improved testing approaches, such as Dexter [4] used in the LLVM codebase, a system for manually annotating test cases with expected debugger-visible states.

Certain forms of automated testing have been proposed. The 'actionable programs' technique [19] generates many program variants, each selectively inhibiting optimisations at some point by adding an optimisation-inhibiting `printf`-like output action for many combinations of variables and source coordinates, then automatically driving a debugger to check agreement between less-optimised and more-optimised variants. The narrow range of these variants limits the depth of testing. Another approach [12] collects 'debugging traces' by single-stepping unoptimised and optimised binaries; 'trace invariants' are in effect a sanity cross-check of line, frame, variable scope, and function argument information, ensuring metadata is plausible but only providing a weak correctness property. The 'conjectures' technique [1] avoids the need for an oracle by seeking to instead find bugs via three heuristics, two reflecting ABI constraints on separate compilation (call sites and global linkage) and one that a variable's debug-availability should only decrease over each live range (reflecting a transition from live to dead but not vice-versa) This heuristic approach offers a useful 'lower bound' of debug info sanity.

11 Conclusions

Beliefs about debugging infrastructure arise as much from folklore as from analysis, just as its technologies arise as much from happenstance and accretion as from design. We have sought to correct a common perception that optimisation and debuggability necessarily negate one another, in favour of more nuanced views. These include that *state-dropping*, not 'optimisation' per se is what necessarily lessens debuggability, that state recovery and retention techniques are known, that new ones are possible, and therefore that fixed optimisation 'levels' and concomitant debugging penalties are not necessary. They can be supplanted by a more flexible system of dynamic mappings and opt-in communication overheads, not unlike dynamic deoptimisation but exceeding its range of debug-time views.

While we have shown also that these concepts are largely proven in one system or another, making them widely available brings technical problems of its own. Improving the testable quality of debugging metadata is one obvious place to push, and we have outlined one direction for doing so. Lessening the burden of metadata creation is a wider problem than testability, and goes potentially deep into compiler design; we have argued that *residualisation* not elimination is a lens that could be applied in optimisations from the ground up, and that the user-facing contract of optimisation would be better captured as properties of 'allowable envelopes' than a list of passes applied.

Analogies with hardware have informed our views, we used the mental models of concurrency. However, we have

mostly ignored concurrent hardware and concurrent programs. That is mainly because sequential debugging infrastructure is already gnarly enough! However, it is significant that arguably the area of greatest recent practical advance, record–replay, has been described as 'skirting the edge of feasibility' on much current hardware [25], requires incidental performance sacrifices even so, and is impossible on other hardware (ARM). This owes, broadly speaking, to omissions of observability in hardware concurrency primitives. This should be a call to action: just as infrastructure design can provide impressively strong debuggability properties at some turns (e.g. noting precise exceptions even on out-of-order CPUs), it can be alarmingly weak at others. Good designs rest on careful co-design of software infrastructure, compilers and other tools, with debugging firmly in mind—not as a bringer of unwanted compromise, but as an essential facility for the mortal programmer.

Acknowledgments

The authors are grateful to the many people who have commented on presentations of this and related material, including Adrian Prantl, John Regehr, Michael Norrish, Mike Dodds, Thomas Sewell, Peter Sewell, Al Grant, Robert O'Callahan and Kyle Huey. This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) via grant EP/W012308/1.

References

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

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

[3] Mark Batty. 2014. *The C11 and C++11 Concurrency Model*. Ph. D. Dissertation. University of Cambridge.

[4] Greg Bedwell. 2018. Measuring the User Debugging Experience. Presented at European LLVM Developers' Meeting. https://llvm.org/devmtg/2018-04/slides/Bedwell-Measuring_the_User_Debugging_Experience.pdf

[5] Hans-J. Boehm. 2005. Threads Cannot Be Implemented as a Library. In *Proc. of PLDI '05*. https://doi.org/10.1145/1065010.1065042

[6] 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.

[7] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. 2004. Dynamic Instrumentation of Production Systems. In *Proc. of USENIX ATC '04*. https://www.usenix.org/legacy/publications/library/proceedings/usenix04/tech/general/cantrill.html

[8] Jim Chow, Tal Garfinkel, and Peter M. Chen. 2008. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference (ATC'08)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/2008-usenix-annual-technical-conference/decoupling-dynamic-program-analysis-execution

[9] DWARF Debugging Information Format Committee. 2005. *DWARF Debugging Information Format version 3*. Free Standards Group.

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

[11] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. 2014. Eidetic Systems. In *Proc. of OSDI '14*. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/devecsery

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

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

[14] Jakob Engblom. 2012. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*. IEEE, 1–6.

[15] GDB Project. 2009. GDB and Reverse Debugging. Web page. Snapshot dated 2009/9/6, archived at https://web.archive.org/web/20100127100159/http://sourceware.org/gdb/news/reversible.html as retrieved on 2024/9/2.

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

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

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

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

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

[21] Julia Menapace, Jim Kingdon, and David MacKenzie. 2017. The 'stabs' debug format. Online documentation. Available at https://sourceware.org/gdb/current/onlinedocs/stabs.pdf as retrieved on 2024/9/2.

[22] Microsoft Corporation. 1995. CodeView 4 Symbolic Debug Information Specification. Product documentation.

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

[24] Robert O'Callahan, Kyle Huey, Devon O'Dell, and Terry Coatta. 2020. To Catch a Failure: The Record-and-Replay Approach to Debugging: A discussion with Robert O'Callahan, Kyle Huey, Devon O'Dell, and Terry Coatta. *Queue* 18, 1 (March 2020), 61–79. https://doi.org/10.1145/3387945.3391621

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

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

[27] J.E. Smith and A.R. Pleszkun. 1988. Implementing precise interrupts in pipelined processors. *IEEE Trans. Comput.* 37, 5 (1988), 562–573. https://doi.org/10.1109/12.4607

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

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

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

[31] Frank Tip. 1995. A survey of program slicing techniques. *J. Program. Lang.* 3, 3 (1995).

[32] R. M. Tomasulo. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33. https://doi.org/10.1147/rd.111.0025

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

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

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

[36] Lawrence Walter Zurawski. 1990. *Interactive Source-Level Debugging of Globally Optimized Code with Expected Behavior*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign.

Received 2024-04-25; accepted 2024-08-08