

```
$ cc -O2 -g -o hello hello.c && ./hello
```

```
$ cc -O2 -g -o hello hello.c && ./hello
```

```
$ cat -n hello.c
```

```
1  #include <stdio.h>
```

```
2
```

```
3  int main(void)
```

```
4  {
```

```
5      const char *s = "Hello, world!";
```

```
6      printf("%s\n", s);
```

```
7      return 0;
```

```
8  }
```

```
$
```

```
$ cc -O2 -g -o hello hello.c && ./hello
$ cat -n hello.c
1  #include <stdio.h>
2
3  int main(void)
4  {
5      const char *s = "Hello, world!";
6      printf("%s\n", s);
7      return 0;
8  }
$ gdb hello
(gdb) break 6
(gdb) run
Breakpoint 1, main () at hello.c:6
6          printf("%s\n", s);
(gdb) print s
<optimized out>                <-- lies!
```

```
$ gdb hello
```


```
(gdb) break printf
```

```
(gdb) run
```

```
Hello, worlf!
```

```
[Inferior 1 (process 12795) exited normally]
```

Optimisers are good–bad. . .

☰ 

Filter by

- <> Code 0
- Issues 743**
- 🔗 Pull requests 63
- 💬 Discussions 0
- 🔗 Commits 2k
- 📦 Packages 0
- 📖 Wikis 0

State


- Open
- Closed

Advanced


- Owner
- State

743 results (90 ms) Sort by: Best match ▾ 🔖 Save ⋮


llvm/llvm-project

 **[DebugInfo@O2] SLP Vectorizer drops DebugInfo**



bugzilla

 wolfy1961 · Opened on 1 Jun 2020 · #45507


llvm/llvm-project

 **[DebugInfo][SelectionDAG] Missing bool assignment of re**


debuginfo **llvm:codegen**

 OCHyams ·  1 · Opened on 2 Jun · #63076

llvm/llvm-project

 **[DebugInfo@O1] SelectionDAG misses DebugLoc coverage**

bugzilla **llvm:codegen** **wrong-debug**

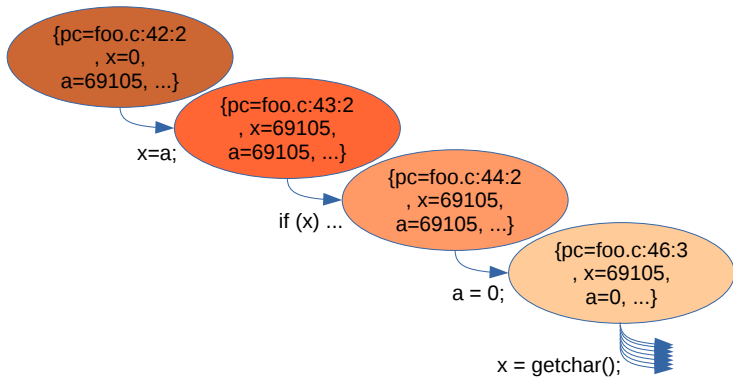
 jmorse · Opened on 21 May 2020 · #45365

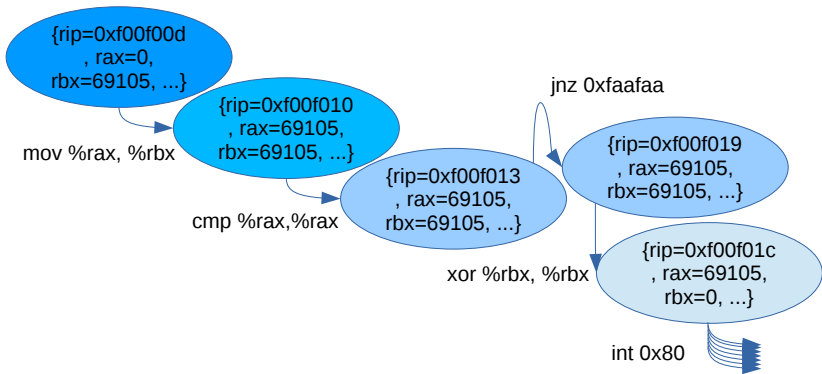
5

```

$ cc -O2 -g -o hello hello.c && readelf -wi hello | column
<b>: TAG_compile_unit          <6d>: TAG_subprogram
<c> AT_producer   : GNU C17 10.2.1 <6e> AT_name       : main
<10> AT_language  : ANSI C99   <72> AT_decl_file  : 1
<11> AT_name      : hello.c     <73> AT_decl_line  : 3
<1d> AT_low_pc    : 0x0         <75> AT_type      : <0x30>
<30>: TAG_base_type          <79> AT_low_pc    : 0x1050
<31> AT_byte_size : 4          <81> AT_high_pc   : 0x17
<32> AT_encoding  : signed     <8f>: TAG_variable
<33> AT_name      : signed int <90> AT_name      : s
<38>: TAG_base_type          <92> AT_decl_file  : 1
<39> AT_byte_size : 1          <93> AT_decl_line  : 5
<3a> AT_encoding  : signed char <95> AT_type      : <0x5b>
<3b> AT_name      : signed char <99> AT_location  : OP_addr: 2004; OP_stack_value
<48>: TAG_const_type        <c6>: TAG_subprogram
<49> AT_type      : <0x38>    <c7> AT_linkage_name: puts
<5b>: TAG_pointer_type      <cb> AT_name      : __builtin_puts
<5c> AT_byte_size : 8          <cf> AT_decl_file  : 2
<5d> AT_type      : <0x68>    <d0> AT_decl_line  : 0

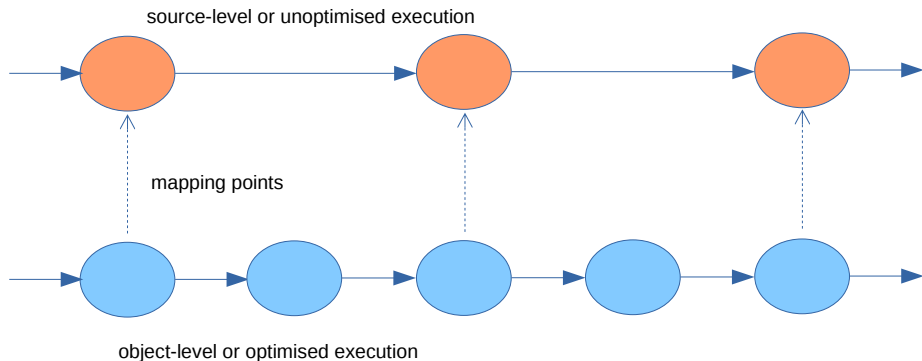
```





Debugging information encodes a function

- mapping *machine program state*
- ... up to *source program state*
- ... at run time
- ... ‘undoing’ **complex** transformations wrought by compilers



How good is it *possible* to make the debugging experience, given a program binary that has been optimised by the compiler?

In this work, we identify and study a related important problem: the completeness of debug information. Unlike correctness issues for which an unoptimized executable can serve as reference, we find there is no analogous oracle to deem when the cause behind an unreported part of program state is an unavoidable effect of optimization or a compiler implementation defect. In this scenario, we argue that empirically derived conjectures on the expected availability of debug information can serve as an effective means to expose classes of these defects.

Assaiante et al, ASPLOS 2023

```

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}

```

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

```

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}

```

data: in r1 at all points

arg: in r2 at all points

i: value 0 from 3 to 5
in r3 from 5

tmp: in r4 from 5

out1: in r5 from 3

out2: in r6 from 3

Don't say:

“Compilers may *eliminate* unused or redundant computation (code) and state (variables).”

Do say:

“Compilers must *residualise* into the debug info anything they want to leave out of the base program.”

```

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 <=MAX < &data[MAX]; ++i)p
7     {
8         out1 ^= data[i]*p;
9     }
10
11    for (i =tmp = &data[tmp]; i <=MAX
12        p<&data[MAX]; ++i)p {
13        out2 &= data[i]*p;
14    }
15    g(out1, out2);
16    return tmp;
17}

```

data: in r1 at all lines

arg: in r2 at all lines

i: value 0 at lines 3-5
value (r7-r1)/4
from line 5

tmp: in r4 from line 5

out1: in r5 from line 3

out2: in r6 from line 3

Zurawski⁷ develops the notion of a *recovery function* that matches each kind of optimization. As an optimization is applied during compilation, the compensating recovery function is also created and made available for later use by a debugger. If such a recovery function cannot be created, then the optimization is omitted. Unfortunately, code-motion-related optimizations generally lack recovery functions and so must be foregone. Taking this approach to the extreme converges with traditional practice, which is simply to disable all optimization and debug a completely unoptimized program.


```

(gdb) l
11 for (i = tmp; i < MAX; ++i)
12 {
13     out2 &= data[i];
14 }
15 g(out1, out2);
16 return tmp;           <== we are here
17 }
(gdb) print out1
<optimized out>       <-- not always a lie

```

... loss of (in-scope) data

```

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

```

→

```

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

```

... loss of control state

DWARF-style residual computation gives us. . .

. . . this:



. . . but not this:



Time-travel debugging is great, but. . .

- ‘record input from program start’

. . . takes preparation! + always-on overhead

What if we could *residualise state*, not just code?

- ‘retain otherwise-dropped information while attached’

. . . can’t travel back arbitrarily far, but debugging stays sane.

Residual computation: not a (stateless) *function* but a (stateful) *process*!

DWARF tacitly assumes *state lives only in the object program*.

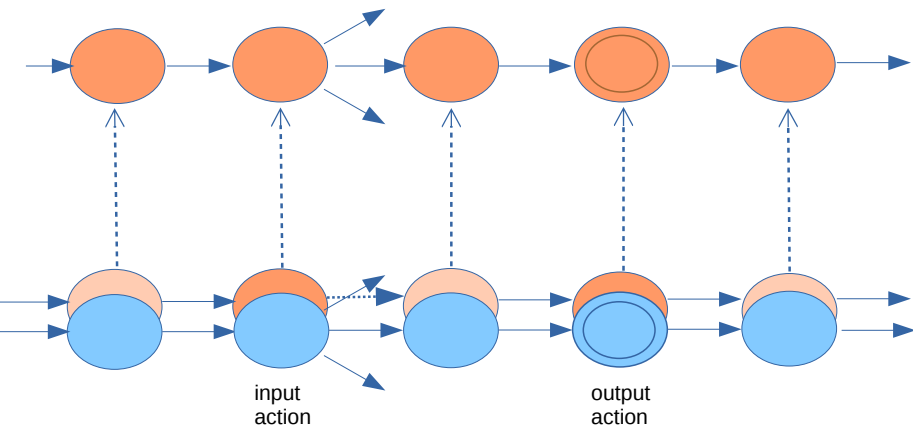
Why not also in the debugger?

Recent additions *already* do this in two small, ad-hoc ways.

- ‘location views’ – one PC, many source locations
- ‘[call] entry value’ operation – magic up a past value

See also ‘eviction recovery’ (in Caroline Tice’s 1999 thesis)

Wanted: *general* design for this. Does it generalise?



Limit case: consume same input, compute arbitrarily differently...

We were asking was the wrong question!

How good is it possible to make the *available debugging experiences*?

A program subject to compiler optimisation is no longer ‘a program’.
It’s a *family of program variants*.

‘Debugger, show me execution at source level’ is an *ill-posed* request!
The user should choose how ‘optimised’ a view they want to see!

But there’s no limit to how complete and correct an ‘illusion’ we can show, as long as we can *residualise* both computation and state.

Source-level debugging of optimised code → residual computation.

It fails ‘unavoidably’ only because it’s not stateful (enough, yet).

Aside from thought-stuff, some more specific work we’re doing:

Measuring local variables’ coverage in debug info

- CC ’24 paper, LLVM contribution ongoing. . .

Differential testing of local variable info, call tree recovery. . .

- ongoing

Debug info meets garbage collection: stack maps. . .

- my nutty side project

That’s it! Ask me questions! You can also read our Onward! 24 paper.

<https://humprog.org/~stephen/#onward24>
stephen@humprog.org jryans@gmail.com