How debuggable is your (compiler-optimised) program?

Stephen Kell  J. Ryan Stinnett  
King’s College London  

stephen@humprog.org  

February 2024
(gdb) print i
<variable optimized out>
(gdb) print i
<no debug info for 'i' here; assume optimized out>
How debugging works (one of two ways)

ld.so
.c .f.c.f
link (ld)
.c.s
as
fc
as
as
.o.o .o.o .o.o
a.out
load (dyn. link)
operating system
hardware
cc
...
How debugging works (one of two ways)

ld.so
.c .f .c .f
link (ld)
.c .s
as
fc
as
.o .o .o .o .o .o
a.out
load (dyn. link)
operating system
hardware

e8 eb fe ff ff      callq 0x555555555030
8b 05 d1 2e 00 00   mov 0x2ed1(%rip),%eax
5d                     pop  %rbp
c3                     retq
How debugging works (one of two ways)

```bash
ld.so
.c .f .c .f
link (ld)
.c .s
as -g
fc -g
as as
.o .o .o .o .o .o
a.out
load (dyn. link)
operating system
hardware
cc -g
...
g(a);
return x;
```
How debugging fails to work from Assaiante et al, ASPLOS 2023

Product of Metrics

- Og
- O3
- O2
- Os

(clang)
What’s not happening

Our optimisers are just *soooo* good. . .

. . . that 70–80% of program state held in local variables. . .

. . . is eliminated from the program image at run time?
What is happening? Optimisers are good–bad...

(This is anecdata. How to be more rigorous? Ask me about this later.)
What’s happening

- no strong correctness criterion for compilers
- no metric for different compilers’ authors to compete on
- tension of coverage vs correctness
  - ‘if unsure, emit nothing just to be safe!’?
  - ‘if unsure, emit something – it might help’?
This talk in one slide

- a view of debug info as \textit{residualised computation}
- a proposed correctness criterion for \textit{local variable info}
- a tool that checks it(-ish)
- some preliminary results
- (optional: some pretty pictures)
- some discussion, to-do items, . . .
We say ‘debugging’, but...
A while back: dynamic languages + **DWARF** interop

- **Traditional approach: in-band**
  - Headers point to object metadata

- **DwarfPython approach:**
  - Metadata kept out-of-band and looked up associatively
  - Look-up function

---

**Out-of-band metadata**

- **my_ellipse**
  - **maj**
  - **min**
  - **ctr**

**Object metadata**

- **class point**
  - **field:** x ...
  - **field:** y ...

- **class ellipse**
  - **field:** min ...
  - **field:** maj ...
  - **field:** ctr ...
  - **method:** draw
  - **method:** move

**Object data**

- **my_ellipse**
  - **header**
  - **min:** 1.5
  - **maj:** 1.0
  - **ctr**

- **ell_centre**
  - **header**
  - **field:** x
  - **field:** y

- **DwarfPython**
  - Dynamic language implementation
Yet more: run-time checking in C, helped by Dwarf

if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker, (struct commit *)obj))
        return -1;
    return 0; unchecked
}

becoming...

if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker,
            (CHECK (is_a(obj, "struct commit")),
            (struct commit *)obj)))
        return -1;
}
$ cc -g -o hello hello.c && readelf -wi hello | column

<b>:TAG_compile_unit
  AT_language : 1 (ANSI C)
  AT_name : hello.c
  AT_low_pc : 0x4004f6
  AT_high_pc : 0x400516

<7ae>:TAG_pointer_type
  AT_byte_size: 8
  AT_type : <0x2af>
  AT_name : argc
  AT_byte_size: 8
  AT_type : <0xc5>
  AT_location : fbreg - 20

<76c>:TAG_subprogram
  AT_name : main

<c5>: TAG_base_type
  AT_type : <0xc5>
  AT_low_pc : 0x4004f6
  AT_high_pc : 0x400516
  AT_encoding : 5 (signed)
  AT_name : int

<791>: TAG_formal_parameter
  AT_name : argv

<2af>:TAG_pointer_type
  AT_byte_size: 8
  AT_type : <0x2b5>
  AT_location : fbreg - 20

<2b5>:TAG_base_type
  AT_byte_size: 1
  AT_encoding : 6 (char)
  AT_name : char

<79f>: TAG_formal_parameter
  AT_name : argv
  AT_type : <0x7ae>
  AT_location : fbreg - 32
AT_name : gconv_parse_code
AT_prototyped : 1
AT_low_pc : 0x31ba0
AT_high_pc : 0x158
AT_frame_base : {OP_call_frame_cfa}
AT_GNU_locviews: 0x1275c
AT_ranges : 0x19b7

AT_name : suffix
AT_type : <0x1dc62>

AT_name : pc
AT_type : <0x1e5be>
AT_abstract_origin: <0x1e64e>
AT_entry_pc : 0x31bf0
AT_low_pc : 0x31bf0
AT_GNU_locviews: 0x12737

AT_name : len
AT_type : <0x1daa9>
AT_entry_pc : 0x31c38
AT_low_pc : 0x31c38
AT_high_pc : 0x41
A Dwarf of a spec?
Mapping states

Let’s adopt a transition-system-like view of program execution

- **state**: a (varying) collection of name-value pairs
- **operations**: atomic changes to the state
Mapping states

Let’s adopt a transition-system-like view of program execution

- **state**: a (varying) collection of name-value pairs
- **operations**: atomic changes to the state

DWARF encodes a function

- mapping *machine program state*
- . . . up to *source program state*
- . . . at run time
- . . . ‘undoing’ **complex** transformations wrought by compilers
Tool wanted: debug consistency checker

Source code
.bc
LLVM IR
(-g -O0)
Clang
(-g -O1)
Clang
(-g -O0)
.bc
LLVM IR
(-g -O1)

the binary we want to check

the moral authority

a convenient proxy
Best avail. oracle is unoptimised program

Checking consistency between O0 and O1

Consistency check
each function independently

Sym. source values (O0)
e.g. source assignments

Sym. source values (O1)
e.g. source assignments

Compare

Compare

l3: y = 0
l5: y = x + 4 + n

l3: y = 0
l5: y = x + 4 + n

Extract features (O0)

Extract features (O1)
Some engineering later. . .

We’re checking some property to do with *consistency w.r.t. -*\textit{O}0*

But what exactly is the right property to check?

. . . given that most notions of *simulation* are off the table?
Let’s consider three optimisations of this code

```c
int f(int *data, void *arg) {
    int i = 0, tmp, out1 = 0, out2 = 0;
    i = tmp = get_start(arg);
    for (; i < MAX; ++i) {
        out1 ^= data[i];
    }
    for (i = tmp; i < MAX; ++i) {
        out2 &= data[i];
    }
    g(out1, out2);
    return tmp;
}
```
Optimisation 1: eliminate dead store (zeroing)

```c
int f(int *data, void *arg)
{
    int i = 0, tmp, out1 = 0, out2 = 0;
    i = tmp = get_start(arg);
    for (; i < MAX; ++i)
    {
        out1 ^= data[i];
    }
    for (i = tmp; i < MAX; ++i)
    {
        out2 &= data[i];
    }
    g(out1, out2);
    return tmp;
}
```
Optimisation 2: strength-reduce indexing, then eliminate i

```c
int f(int *data, void *arg)
{
    int i = 0, tmp, out1 = 0, out2 = 0;

    i = tmp = get_start(arg); int *p = &data[tmp];
    for (; i < MAX; ++i)
    {
        out1 ^= data[i] * p;
    }

    for (i = tmp; i < MAX; ++i)
    {
        out2 &= data[i] * p;
    }

    g(out1, out2);
    return tmp;
}
```
Optimisation 3: loop fusion

```c
int f(int *data, void *arg)
{
    int i = 0, tmp, out1 = 0, out2 = 0;

    tmp = get_start(arg);
    int *p = &data[tmp];

    for (i = tmp; i < MAX; ++i)
    {
        out1 ^= data[i] * p;
        out2 &= data[i] * p;
    }

    for (i = tmp; i < MAX; ++i)
    {
        out2 &= data[i];
    }

    g(out1, out2);
    return tmp;
}
```
In all cases, Dwarf lets us recover a source-level view...

```c
int f(int *data, void *arg)
{
    int i = 0, tmp, out1 = 0, out2 = 0;
    i = tmp = get_start(arg);
    for (; i < MAX; ++i)
    {
        out1 ^= data[i];
    }
    for (i = tmp; i < MAX; ++i)
    {
        out2 &= data[i];
    }
    g(out1, out2);
    return tmp;
}
```
int f(int *data, void *arg)
{
    int i = 0, tmp, out1 = 0, out2 = 0;
    i = tmp = get_start(arg);
    for (; i < MAX; ++i)
    {
        out1 ^= data[i];
    }
    for (i = tmp; i < MAX; ++i)
    {
        out2 &= data[i];
    }
    g(out1, out2);
    return tmp;
}
What just happened

What first looked like an *elimination* . . .

. . . is now a *residualisation*.

Debug info is a computational medium—a peer of the object program!
Different residualisations

Conceptually, we can residualise **computation**:  
- for any operation dropped from the program...  
- ... compute it instead in debug info  
- e.g. using a Dwarf expression...  

We can also residualise **state**:  
- as any piece of state is dropped from the program...  
- ... stash it in the debugger!  
- ... sadly not using Dwarf currently :-(

It’s not usual for compiler authors to think in these terms  
(... yet, but they should?)  
(... blame those academic textbook authors?)
Optimisation 2 → residualise i by computation

int f(int *data, void *arg) { int i = 0, tmp, out1 = 0, out2 = 0;
  i = tmp = get_start(arg); int *p = &data[tmp];
  for (; i < MAX; p < &data[MAX]; ++i) {
    out1 ^= data[i]*p;
  }
  for (i = tmp; p < &data[MAX]; ++i) {
    out2 &= data[i]*p;
  }
  g(out1, out2);
  return tmp;
}
int f(int *data, void *arg)
{
    int i = 0, tmp, out1 = 0, out2 = 0;

    i = tmp = get_start(arg); int *p = &data[tmp];
    for (; i < MAX; p < &data[MAX]; ++i)
    {
        out1 ^= data[i] * p;
        out2 &= data[i] * p;
    }

    for (i = tmp; i < MAX; ++i)
    {
        out2 &= data[i];
    }

    g(out1, out2);
    return tmp;
}
Harder to residualise: reordering!

Updates to out2 now happen in a different order, w.r.t. out1!

What would it mean to ‘residualise’ code motion like this?

Possibly: keep the entire unopt code around on the side (e.g. in debug info), and have the debugger switch to running *that* (… and back?)

(i.e. what language VMs’ debug servers do: dynamic deopt)

Doing this *or similar* is necessary if we expect the debug-time view of the program to *simulate* the source program

For toolchains, is this a reasonable expectation? Compilers simply aren’t written to be *simulation-preserving*…
Wanted: a correctness property

Not too strong that it can’t tolerate code motion

Not too weak, i.e. don’t tolerate missed opportunities to residualise
An oracle... for what question, exactly?

Unfortunately, for the many reasons we discussed in Section 2, at the moment there is instead no reliable oracle that may tell when (or where) a given piece of program state should be visible when debugging an optimized executable instance of that program.

(Assaiante et al again)

→ ‘give up’ on strong correctness, focusing rather on how to
  ■ identify coverage lower than one can reasonably ‘conjecture’
  ■ ≈ detect ‘debug info smells’

Alternatively: decree ‘it should always be available’, and make it so!
Let’s preview something more ambitious. . .

Variables *should* always be visible!

Just not necessarily updating ‘in the right order’

. . . *relative to one another*

i.e. not a simulation, but coherent per-variable
Let’s preview something more ambitious. . .

Variables *should* always be visible!

Just not necessarily updating ‘in the right order’

. . . *relative to one another*

i.e. not a simulation, but coherent per-variable

So our correctness property is something like:

- each variable goes through the same sequence of values
- . . . that it did in the source program order
- ( . . . *individually*, i.e. saying nothing about *co-occurrence*)

Want this even if we have to tweak Dwarf to do it. . .

- Dwarf can compute any function we like *of current state only*
- some ability to residualise control-flow vertices via *location views*
Suppose MAX is 5 and get_start returns 2. Over such a program path, what does i do?
Suppose MAX is 5 and get_start returns 2. What does r3 (holding i from line 5) do?

```c
int f(int *data, void *arg)
{
    int i = 0, tmp, out1 = 0, out2 = 0;
    i = tmp = get_start(arg);
    for (; i < MAX; ++i)
    {
        out1 ^= data[i];
    }
    for (i = tmp; i < MAX; ++i)
    {
        out2 &= data[i];
    }
    g(out1, out2);
    return tmp;
}
```
Suppose MAX is 5 and get_start returns 2.

What does `i` do?

```c
int f(int *data, void *arg)
{
    int i = 0, tmp, out1 = 0, out2 = 0;
    i = tmp = get_start(arg);
    for (; i < MAX; ++i)
    {
        out1 ^= data[i];
    }
    for (i = tmp; i < MAX; ++i)
    {
        out2 &= data[i];
    }
    g(out1, out2);
    return tmp;
}
```
Suppose \( \text{MAX} \) is 5 and \text{get\_start} \ returns 2. 

What does \( i \) do?

2
3 4 5
3 4
2
5

value 0 from 3 to 5
value \((r7-r1)/4\) from 5

1 int f(int *data, void *arg)
2 {
3   int i = 0, tmp, out1 = 0, out2 = 0;
4   i = tmp = get_start(arg); int *p = &data[tmp];
5   for (; i < \text{MAX} \land p < &data[\text{MAX}]; ++\( i \), \( p \)) {
6     out1 ^= data[i] * p;
7   }
8   for (i = tmp; p < &data[\text{MAX}]; ++\( i \), \( p \)) {
9     out2 &= data[i] * p;
10   }
11   g(out1, out2);
12   return tmp;
13}
So far so good

‘Convince yourself’ that it works for tmp, out1 and out2. . .

and for variants, e.g. do zeroing of i but later, or pick a sink/hoist to do

Strong as a bug-finding tool, albeit not a panacea:

- per-concrete-path approach can witness values ‘correct by accident’
- execution can still ‘jump around’ (i.e. no simulation)

Does it break?
Suppose MAX is 5 and get_start returns 2.
What does i do?

```c
int f(int *data, void *arg)
{
    int i = 0, tmp, out1 = 0, out2 = 0;
    i = tmp = get_start(arg); int *p = &data[tmp];
    for (; i < MAX; p < &data[MAX]; ++i)
    {
        out1 ^= data[i] * p;
        out2 &= data[i] * p;
    }
    for (i = tmp; i < MAX; ++i)
    {
        out2 &= data[i];
    }
    g(out1, out2);
    return tmp;
}
```
Fusing the loops broke our property over i!

The second iteration over the sequence no longer happens.

Medium-term fix: find ways to residualise the missing states
Short-term fix: relax our property
Possible relaxations

To unbreak loop fusion, we could try:

- ‘subsequence’: allow skipping ahead $n$
  - how to bound this? static $n$? dynamic?
- ‘set’: just treat updates as a set, discarding order + duplicates
- lifetime splitting: ‘i epoch 1’, ‘i epoch 2’ (these are being overlapped)
Subsequence relaxation

→ allow skipping ahead. It ‘works’...

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

Seems hard to make principled. But violations would clearly be bugs.
Set relaxation

Compare the set of updates, coordinatised but not ordered.

```c
int f(int *data, void *arg) {
    int i = 0, tmp, out1 = 0, out2 = 0;

    i = tmp = get_start(arg); int *p = &data[tmp];

    for (; i < MAX; ++i) {
        out1 ^= data[i] * p;
        out2 &= data[i] * p;
    }

    for (i = tmp; i < MAX; ++i) {
        out2 &= data[i];
    }

    g(out1, out2);
    return tmp;
}
```

Set is:
{3:undef→0, 5:0→2, 6:2→3, 6:3→4, 6:4→5}
Set relaxation

... a nice idea but unfortunately doesn’t work!

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

Sets differ!

Would need to know that lines 5–6 are ‘standing in’ for 11.
int f(int *data, void *arg) {
    int i = 0, tmp, out1 = 0, out2 = 0;

    i = tmp = get_start(arg);
    for (; i < MAX; ++i) {
        out1 ^= data[i];
    }

    for (int j = tmp; j < MAX; ++j) {
        out2 &= data[j];
    }

    g(out1, out2);
    return tmp;
}
Splitting source variables

```c
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     out1 ^= data[i];
8 
9 }
10
11   for (int j = tmp; j < MAX; ++j)
12     { out2 &= data[j];
13   }
14   g(out1, out2);
15   return tmp;
16 }
```

After fusion, i and j will be modelled by the same machine variable...
Splitting source variables

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

Our cue to split `i` into two epochs
- for which our property is evaluated separately
- run concurrently (interleaved)...
- the same machine variable models both!
Fusing the loops broke our property over $i$!

Hypothesis: most passes do preserve our per-variable property . . .
. . . maybe sometimes needing the epoch-forking trick

Full disclosure: loop tiling also breaks even if we split!

Hypothesis: some kind of 80–20 or (90–10?) split?

‘Relatively few passes are this ornery.’ So for an ornery pass:

- have the pass annotate the vars it screws up
- $\rightarrow$ picked up by a checking tool (‘relax on this var. . . ’)

Hit me with your ornery passes! (Someone mentioned GVN?)
ineffectual <-> unreachable

Elimination of unreachable code does not affect our property.
Reachable code sometimes called ‘dead’, e.g. ‘dead stores’...

s = 42;
s = f(x);  // breakpoint here...

... but dead stores are reachable! Just not effectual...

We would reasonably want them to appear in the ‘debug illusion’!
Limits of residualisation

**DWARF** can residualise only

- state that is *redundant*
- *statically enumerable* operations
- ... forming a straight-line control flow segment

So a ‘straight line’ of unobserved intermediate states can be residualised

*Branching control flow* is trickier, e.g. . . .

```java
if (cond) {
    s = <pure expr 1>;  // dead store
} else {
    s = <pure expr 2>;  // dead store
}
s = 0;  // the reason the above are dead
```

... although in this case, if-conversion suffices
Dead values

- ask me about resurrection pill *knowledge extension*