Accurate coverage metrics for compiler-generated debugging information

EuroLLVM 2024

J. Ryan Stinnett
🏠 convolv.es
💼 King's College London

Stephen Kell
🏠 humprog.org
💼 King's College London

Debugging example

```c
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Debugging example

```c
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Debugging example

```
 1 void example() {
 2   int number;
 3
 4   printf("Enter a number: ");
 5   scanf("%d", &number);
 6
 7   int sum = 0;
 8   int i = 1;
 9   while (i <= number) {
10     sum += i;
11     i++;
12   }
13
14   printf("Sum is %d.\n", sum);
15 }
```

Explore value of sum in debugger

```
    7    int sum = 0;
 -> 8    int i = 1;
    9    while (i <= number) {
(db) print sum
```

What will we see…?

Debugging example

```c
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Explore value of sum in debugger

```
   7    int sum = 0;
-> 8    int i = 1;
   9    while (i <= number) {
(db) print sum
```

If debug info is present and correct:

```
sum = 0
```

Debugging example

```
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Explore value of sum in debugger →

```
   7    int sum = 0;
-> 8    int i = 1;
   9    while (i <= number) {
(db) print sum
```

If debug info is present and correct:

```
sum = 0
```

Otherwise we may see any of:

```
sum = <variable not available>
sum = <optimised out>
sum = <garbage value>
```

Debugging example

```
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Explore value of sum in debugger

```
    7   int sum = 0;
 -> 8   int i = 1;
    9   while (i <= number) {
(db) print sum
```

If debug info is present and correct:

```
sum = 0
```

Otherwise we may see any of:

```
sum = <variable not available>
sum = <optimised out>
sum = <garbage value>
```

Quick intro to debug info

source	instructions	line table	location exprs
7 `int sum = 0;`	`mov eax, 0`	`ln 7`	`sum: (value 0)`
8 `int i = 1;`	`mov ecx, 1`	`ln 8`	`sum: (value 0)`
9 `while (i <= number) {`	`cmp eax, 1`	`ln 9`	`sum: (value 0)`
	`jl .loop_exit`	`ln 9`	`sum: (value 0)`
	`mov edx, 1`	`ln 9`	`sum: (value 0)`
	`.loop_body:`		`sum: (value 0)`
	`mov edi, esi`		`sum: (reg edi)`
10 `sum += i;`	`mov esi, ecx`	`ln 10`	`sum: (reg edi)`
	`add esi, edi`	`ln 10`	`sum: (reg esi)`
11 `i++;`	`add edx, 1`	`ln 11`	`sum: (reg esi)`
12 `}`	`cmp ecx, eax`	`ln 12`	`sum: (reg esi)`
	`mov ecx, edx`	`ln 12`	`sum: (reg esi)`
	`jl .loop_body`	`ln 12`	`sum: (reg esi)`
	`.loop_exit:`		

Quick intro to debug info

source	instructions	line table	location exprs
`7 int sum = 0;`	`mov eax, 0`	`ln 7`	`sum: (value 0)`
`8 int i = 1;`	`mov ecx, 1`	`ln 8`	`sum: (value 0)`
`9 while (i <= number) {`	`cmp eax, 1`	`ln 9`	`sum: (value 0)`
	`jl .loop_exit`	`ln 9`	`sum: (value 0)`
	`mov edx, 1`	`ln 9`	`sum: (value 0)`
	`.loop_body:`		`sum: (value 0)`
	`mov edi, esi`		`sum: (???)`
`10 sum += i;`	`mov esi, ecx`	`ln 10`	`sum: (???)`
	`add esi, edi`	`ln 10`	`sum: (???)`
`11 i++;`	`add edx, 1`	`ln 11`	`sum: (???)`
`12 }`	`cmp ecx, eax`	`ln 12`	`sum: (???)`
	`mov ecx, edx`	`ln 12`	`sum: (???)`
	`jl .loop_body`	`ln 12`	`sum: (???)`
	`.loop_exit:`		

Good properties of a metric

- Goal
  - Find a good debug info coverage metric
    to focus compiler efforts on truly missing coverage
- Good coverage metric should be
  - Independent of compiler used, options specified, etc.
  - Possible to achieve 100%
  - Free of anomalies (more on this later)

Debug info coverage challenges

```c
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Ideal compiler measured by existing tools →

Imaginary ideal compiler emits debug info for entire defined range of `sum`

Measure coverage with existing tools (`debuginfo-quality`, `llvm-dwarfdump`)...

Debug info coverage challenges

```
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Ideal compiler measured by existing tools →

Imaginary ideal compiler emits debug info for entire defined range of `sum`

Measure coverage with existing tools (`debuginfo-quality`, `llvm-dwarfdump`)...

$$\text{coverage} = \frac{\text{described}_{ib}}{\text{scope}_{ib}}$$

Debug info coverage challenges

```
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Sco
112 B

Des
63 B

Ideal compiler measured by existing tools →

Imaginary ideal compiler emits debug info for entire defined range of `sum`

Measure coverage with existing tools (`debuginfo-quality`, `llvm-dwarfdump`)...

$$\text{coverage} = \frac{\text{described}_{ib}}{\text{scope}_{ib}}$$

Debug info coverage challenges

```
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Sco
112 B

Des
63 B

Ideal compiler measured by existing tools →

Imaginary ideal compiler emits debug info for entire defined range of `sum`

Measure coverage with existing tools (`debuginfo-quality`, `llvm-dwarfdump`)...

$$\text{coverage} = \frac{\text{described}_{ib}}{\text{scope}_{ib}}$$

63 / 112 instr. bytes covered
56% coverage

Debug info coverage challenges

```
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Sco 112 B

Des 63 B

Ideal compiler measured by existing tools →

Imaginary ideal compiler emits debug info for entire defined range of `sum`

Measure coverage with existing tools (`debuginfo-quality`, `llvm-dwarfdump`)...

$$\text{coverage} = \frac{\text{described}_{ib}}{\text{scope}_{ib}}$$

63 / 112 instr. bytes covered

56% coverage

- ❌ Independent of compiler
- ❌ Possible to achieve 100%
- ❌ Free of anomalies (e.g. stack vs. reg)

Existing coverage tools

Tools such as `debuginfo-quality`, `llvm-dwarfdump`

- Measure in instruction bytes instead of source lines
  - Coverage in bytes is not comparable across compilers
    or even different options passed to the same compiler
  - Doesn’t line up with user experience stepping through source lines

Existing coverage tools

Tools such as `debuginfo-quality`, `llvm-dwarfdump`

- Measure in instruction bytes instead of source lines
    - Coverage in bytes is not comparable across compilers or even different options passed to the same compiler
    - Doesn’t line up with user experience stepping through source lines
- Use scope instead of defined region as denominator
    - Full coverage becomes impossible to achieve with register allocation
    - Accidentally favours unoptimised approach of placing all variables on the stack

Evolution of our approach

Our coverage approach

We construct a more accurate coverage metric:

- Measure coverage in terms of source lines
- For each variable, only expect coverage in the variable's defined region

$$\mathbf{coverage} = \frac{\mathbf{described}_{sl} \cap \mathbf{defined}_{sl}}{\mathbf{scope}_{sl} \cap \mathbf{defined}_{sl}}$$

Our coverage approach

```
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Def Des

Ideal compiler measured by our approach →

Imaginary ideal compiler emits debug info for entire defined range of sum

Measure coverage with our approach using source lines in sum defined region…

$$\text{coverage} = \frac{\text{described}_{sl} \cap \text{defined}_{sl}}{\text{scope}_{sl} \cap \text{defined}_{sl}}$$

Our coverage approach

```
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Def Des

Ideal compiler measured by our approach →

Imaginary ideal compiler emits debug info for entire defined range of `sum`

Measure coverage with our approach using source lines in `sum` defined region…

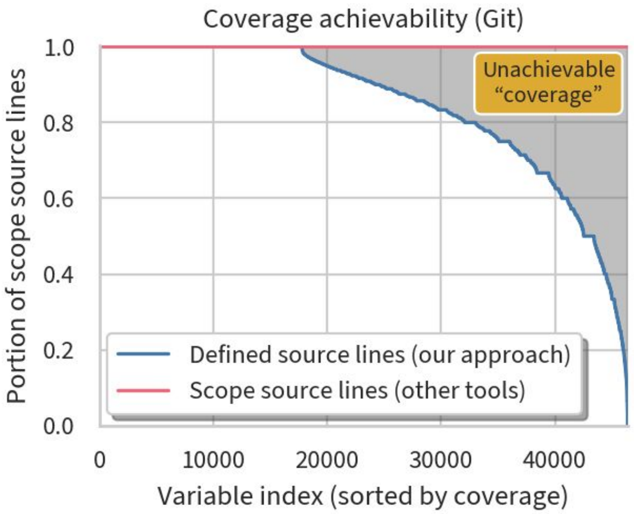
$$\text{coverage} = \frac{\text{described}_{sl} \cap \text{defined}_{sl}}{\text{scope}_{sl} \cap \text{defined}_{sl}}$$

8 / 8 source lines covered
100% coverage

- ✅ Independent of compiler
- ✅ Possible to achieve 100%
- 🤔 Free of anomalies

Coverage achievability

- Other tools (`llvm-dwarfdump`, `debuginfo-quality`) measure coverage using parent scope which includes points at which the variable is undefined
- Our approach starts tracking coverage from point of first definition
- Coverage in terms of source lines instead of instruction bytes
- Unlike past metrics, full coverage is actually attainable with our approach



Measuring real compilers

```
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Def Des

Real compiler measured by our approach →

Real compiler A emits debug info for its view of sum

Measure coverage with our approach using source lines in sum defined region...

$$\text{coverage} = \frac{\text{described}_{sl} \cap \text{defined}_{sl}}{\text{scope}_{sl} \cap \text{defined}_{sl}}$$

6 / 8 source lines in sum defined region covered

75% coverage

🤔

Measuring real compilers

```
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Def Des

Real compiler
measured by
our approach

Real compiler B emits debug info for its view of sum

Measure coverage with our approach using source lines in sum defined region...

$$\text{coverage} = \frac{\text{described}_{sl} \cap \text{defined}_{sl}}{\text{scope}_{sl} \cap \text{defined}_{sl}}$$

6 / 8 (different!) source lines in sum defined region covered

75% coverage

🤔

Measuring real compilers

```c
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Def Des

Current compilers only emit debug info for source lines where computation happens!

Measuring real compilers

```
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Def Des
Comp

Current compilers only emit debug info for source lines where computation happens!

After noticing this, we added a static source analysis pass to find these lines and then filter coverage to only lines with computation

Measuring real compilers

```
void example() {
  int number;

  printf("Enter a number: ");
  scanf("%d", &number);

  int sum = 0;
  int i = 1;
  while (i <= number) {
    sum += i;
    i++;
  }

  printf("Sum is %d.\n", sum);
}
```

Def Des
Comp

Real compiler measured by our approach

Real compiler emits debug info for its view of `sum`

Measure coverage with our approach using source lines with computation in `sum` defined region…

5 / 6 src. lines with comp. covered

✅ Independent of compiler
✅ Possible to achieve 100%
✅ Free of anomalies

Our revised coverage approach

We construct a more accurate coverage metric:

- Measure coverage in terms of source lines
- For each variable, only expect coverage in the variable’s defined region
- Filter source lines to only those with computation

$$\text{coverage} = \frac{\text{described}_{sl} \cap \text{defined}_{sl} \cap \text{computation}_{sl}}{\text{scope}_{sl} \cap \text{defined}_{sl} \cap \text{computation}_{sl}}$$

Experiments

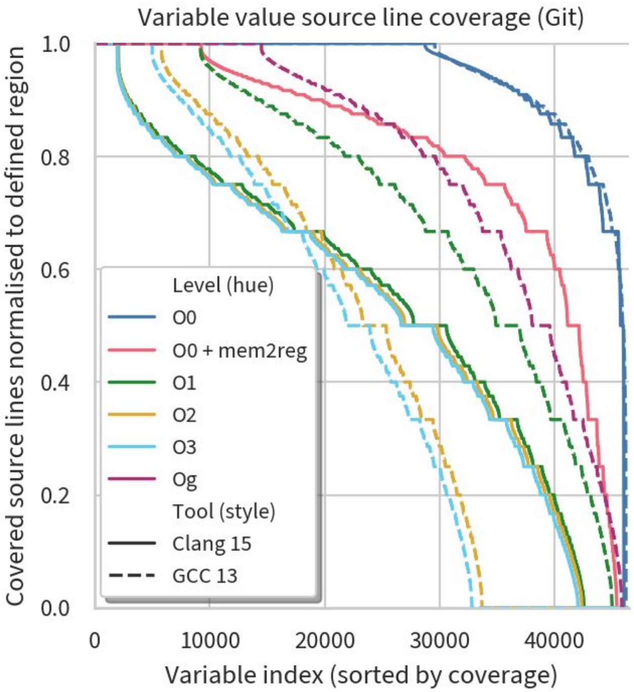
Current status of tooling

- Implemented tool to calculate our “defined source lines” coverage metric
  - Uses compiler-emitted DWARF as coverage input
- Built better baseline via source analysis
  - Program source provides the baseline input
- Research prototype available as artifact with our CC 2024 paper

Measuring coverage

- In programs optimised by common compilers, most variables are missing values for ≥1 line where they are defined
- Some variables are entirely inaccessible

- A large chasm remains uncovered
- Lots of room for future debug info coverage improvements

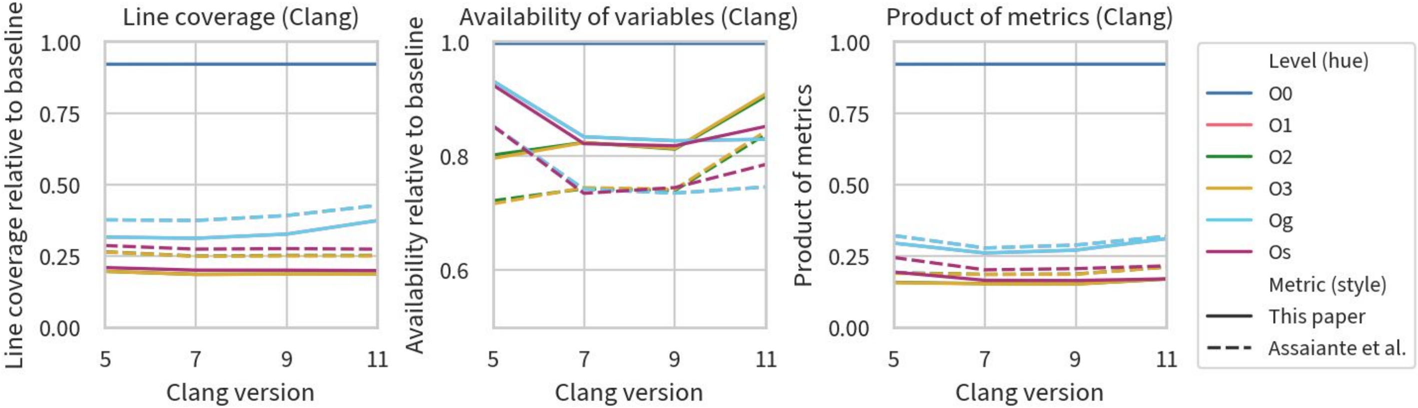


Replication study

Assaiante et al. (ASPLOS 2023) examined debug info coverage across 5,000 Csmith-generated programs

In replicating their experiment using an adjusted version of our approach, we found similarities that validate our work and also expected differences

Replication study



- Similar trends appear across both metrics
  - Our line coverage is slightly lower due to ours being relative to source vs. theirs relative to `O0`
  - Our availability is higher due to expected improvement from counting only defined region
- Validates our approach
  - Our source-relative optimised values ≅
    their `O0`-relative optimised values × our `O0` value (which they used as a baseline)

More detail in our CC 2024 paper

- Coverage for inlined functions
- Detailed description of replication study
- Case studies measuring specific compiler issues
- Knowledge extension: techniques to improve coverage
- Location views: debug info for source positions without instructions

Future improvements

- Planning to continue development towards a version for regular use
    - Likely rebuilding this as a new coverage mode for `llvm-dwarfdump`
    - Will start an RFC thread to discuss the best path with LLVM community
- Aim to make this more easily accessible on an ongoing basis
    - Integrate debug info metrics into Compiler Explorer
    - Add to existing metrics in LLVM nightly testing (LNT)
    - Create pre-merge metrics comparison similar to compile-time tracker
    - If you have feedback on these ideas, please let us know!

Summary

- Debug info often gets lost during optimisation
- This work focuses on improved **coverage metrics (completeness)** for *source coordinates* and *variable location* information
  to measure what is currently being lost
- In future work, we aim to also check consistency (correctness) of this debug info as well

Thanks!

J. Ryan Stinnett
🏠 convolv.es
💼 King's College London

Stephen Kell
🏠 humprog.org
💼 King's College London

Replication study

- Assaiante et al.
    - Dynamic approach using coverage by running debugger over program
        - Counting described variables on each line
    - Calculates average fraction of variables at `O1+` relative to `O0`
        - Only at lines common to both runs
- Our work
    - Static approach using coverage via DWARF debug info and static analysis
        - Counting described lines for each variable
        - For this replication, added a simple binary analysis step based on Valgrind to ensure we only examined reachable lines like Assaiante et al.
    - Reports coverage relative to static analysis baseline for all optimisation levels

Variable locations in DWARF

DWARF debug info generated by compiler (which we want to test) describes source variables via Turing-powerful stack machine with registers and memory as inputs

```
DW_TAG_variable
  DW_AT_name          ("y")
  DW_AT_decl_line     (3)
  DW_AT_type          (0x000000d5 "int")
  DW_AT_location
    [0x3f74, 0x3f7d):
      DW_OP_fbreg -12
🟥  [0x3f7d, 0x3f90):
🟥    <no location emitted>
    [0x3f90, 0x3f94):
      DW_OP_breg5 RDI+0, DW_OP_constu 0xffffffff, DW_OP_and,
      DW_OP_lit1, DW_OP_shl, DW_OP_stack_value
```

Simulated output illustrating expressivity of DWARF locations

Similar stack machine value expressions also appear in LLVM IR debug mappings

Locations are like a **symbolic mapping** of source variables to storage

Ways of thinking about debug info

- If a variable is eliminated, it is not necessarily the case that it is absent from the debug illusion: some debug info formats can describe how to reconstruct it from state that remains
  - DWARF supports expressions in an interpreted stack machine language that the debugger can use to compute functions of program state
- The more thoroughly a variable is “eliminated” from the emitted program, the more it needs to be described in the debug info
- Rather than viewing optimisations and debugging as mutually excluding, it is more accurate to see debug info as **residualising** the eliminations or simplifications made during optimisation
  - Run-time program gets shorter during optimisation, debug info grows to maintain illusion

Priority of debug info for compiler authors

- Passes do try to preserve debug info…
  - e.g. LLVM’s How to update debug info guide for optimisation pass authors
- Incentives not aligned for correct and complete debug info
  - Extra work to produce debug info on top of fast, correct run-time code
- No standard metrics for comparing debug info quality
  - Our own metric tracking coverage over variable’s defined range (instead of scope) may help move the conversation forward here

Disabling optimisation is not always an option

- Real scenarios for optimised debugging
  - Core dumps collected in production
  - Resource heavy programs (e.g. video games) which are too slow without optimisation
  - Programs whose behavior depends on optimisation (e.g. Linux kernel)
  - Tracing unwanted behaviours (e.g. race conditions, memory errors) which may only occur with optimisation
  - **Any program … if you want to debug what actually ran!**
- Poor developer experience has trained many programmers to assume optimised debugging is somehow insurmountable
  - Some may avoid using debuggers entirely
  - In some cases, you can rebuild without optimisation and try debugging again…