

A new era in code coverage in GCC

Jørgen Kvalsvik <j@patch.no>
patch.no

October 2026

GCC 2.8, EGCS 1.0

gcc/profile.cc has copyright headers dated 1996-1997

commit 86144b75fef3ae7025d389ea751422cbca1285d5

Author: Doug Evans <dje@gnu.org>

Date: Fri Mar 28 22:43:13 1997 +0000

Initial revision

Development picks up steam ca. 2017 with a steady stream of improvements

- ▶ Bugfixes
- ▶ Quality of implementation
- ▶ Output features (json, hotness colours, demangling)

The underlying framework is shared with profile-guided optimization (PGO)

Modified Condition/Decision Coverage

commit 08a52331803f66a4aaeaedd278436ca8eac57b50

Author: Jørgen Kvalsvik <j@lambda.is>

Date: Tue Dec 5 12:59:40 2023 +0100

Add condition coverage (MC/DC)

MC/DC - simplified

- ▶ Every condition has taken on true and false
- ▶ Every condition has been shown to *independently* affect the outcome

Widely used criterion

- ▶ Avionics and aerospace
- ▶ Space flight
- ▶ Automotive, trains

$$(a \ \&\& \ b) \ || \ (c \ \&\& \ d)$$

$$\begin{array}{cccc|c} 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{array}$$

a independently affects the outcome

$$\begin{array}{cccc|c} 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{array}$$

```
CFLAGS="-fcondition-coverage -ftest-coverage" LDFLAGS="--coverage
```

```
int test(int a, int b, int c, int d) {  
    if ((a && b) || (c && d))  
        return a*b;  
    else  
        return 1;  
}
```

```
int main() {  
    test(1, 0, 1, 1);  
    test(1, 1, 0, 0);  
}
```

```
make CFLAGS="-fcondition-coverage --coverage" test  
./test  
gcov --conditions test
```

```

2:      1:int test(int a, int b, int c, int d) {
2:      2:      if ((a && b) || (c && d))
condition outcomes covered 4/8
condition 0 not covered (false)
condition 1 not covered (false)
condition 2 not covered (false)
condition 3 not covered (false)
2:      3:          return a*b;
-:      4:      else
#####:      5:          return 1;
-:      6:}

```

Prime path coverage

commit 8ed2d5d219e999aee42015a0db38612011c2c507

Author: Jørgen Kvalsvik <j@lambda.is>

Date: Tue Jun 4 14:13:22 2024 +0200

Add prime path coverage to gcc/gcov

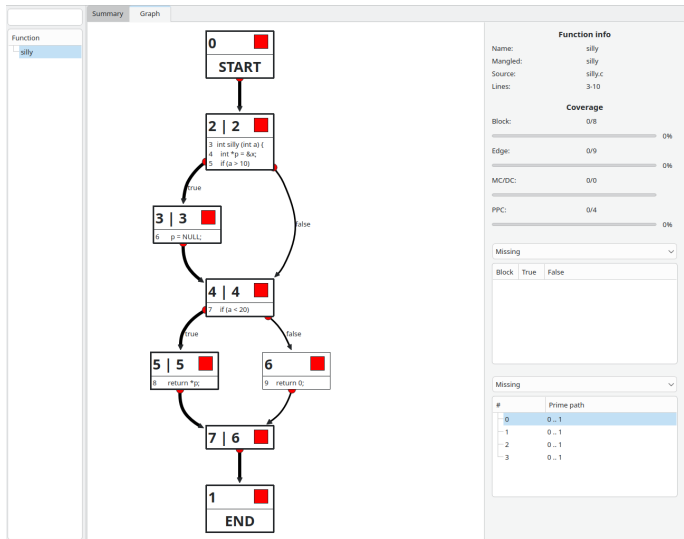
```
int silly (int a) {  
    int *p = &x;  
    if (a > 10)  
        p = NULL;  
    if (a < 20)  
        return *p;  
    return 0;  
}
```

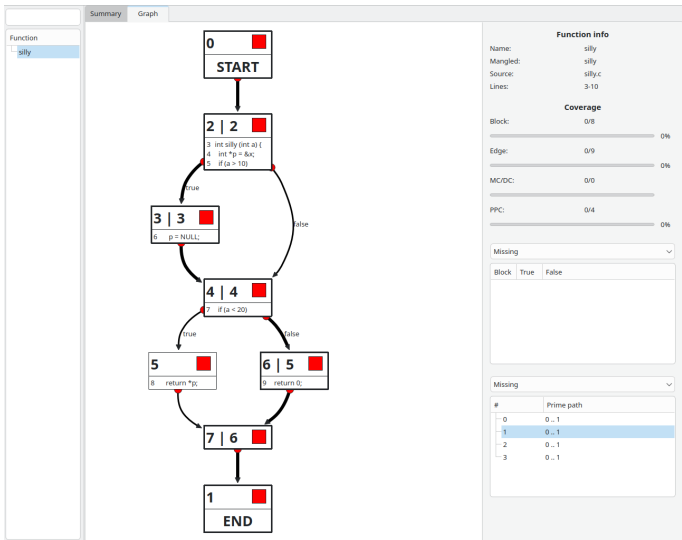
Source: John Regehr <https://blog.regehr.org/archives/872>

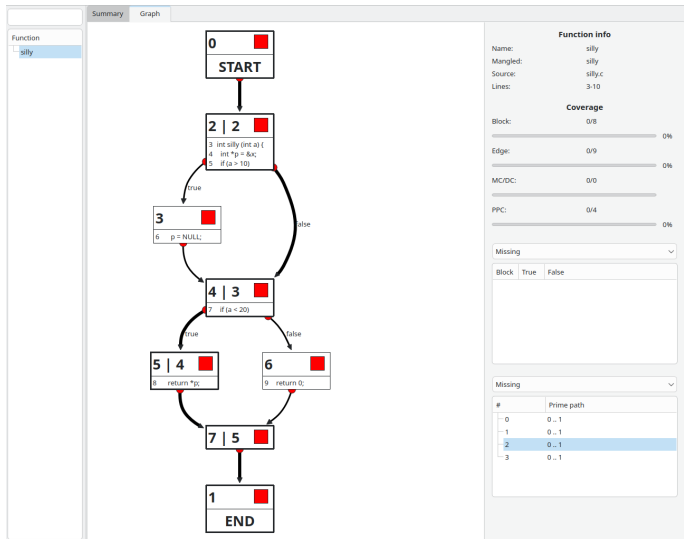
Take every **prime** path

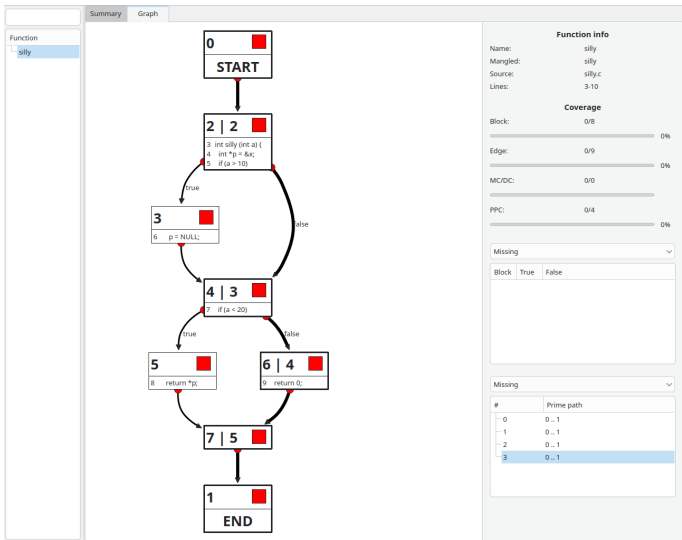
A **prime path** is a maximal-length simple path or simple cycle

```
int silly (int a) {  
    int *p = &x;  
    if (a > 10)  
        p = NULL;  
    if (a < 20)  
        return *p;  
    return 0;  
}
```









Prime paths and loops

Loops must be both *taken*, *skipped*, and *exited*

```
make CFLAGS="-fpath-coverage --coverage" test
./test
gcov --conditions test
```

path 0 not covered:

```
BB 2:          3: int silly (int a) {
BB 2:          4:   int *p = &x;
BB 2: (true)    5:   if (a > 10)
BB 3:          6:     p = NULL;
BB 4: (true)    7:   if (a < 20)
BB 5:          8:     return *p;
```

path 1 not covered:

```
BB 2:          3: int silly (int a) {
BB 2:          4:   int *p = &x;
BB 2: (true)    5:   if (a > 10)
BB 3:          6:     p = NULL;
BB 4: (false)   7:   if (a < 20)
BB 6:          9:   return 0;
```

Coverage suppression

commit f6a1e69999d7b614cbcedff6ecaa01b713f31266

Author: Jørgen Kvalsvik <j@lambda.is>

Date: Tue Sep 30 11:23:56 2025 +0200

Add attribute and pragma for coverage suppression

```
int fn(int *non_null, int positive) {  
    if (!non_null)  
        util_abort("non_null is NULL");  
    if (positive < 0)  
        util_abort("positive is not positive");  
    ...  
}
```

```
int fn(int *non_null, int positive) {  
#pragma GCC suppress_coverage begin  
    if (!non_null)  
        util_abort("non_null is NULL");  
    if (positive < 0)  
        util_abort("positive is not positive");  
#pragma GCC suppress_coverage end  
    ...  
}
```

Contributing to GCC

grep and messing around

The most precious resource are maintainers

Extras

Masking is when the right operand **alone** decides the outcome

$x \ \&\& \ 0$

$x \ || \ 1$

Short-circuiting is when the left operand **alone** decides the outcome

0 && x

1 || x

Masking would be short-circuiting if you **reversed** the expression