Prime Path Coverage in the GNU Compiler Collection

Jørgen Kvalsvik
j@lambda.is

March 27, 2025

Abstract

We describe the implementation of the prime path coverage support introduced the GNU Compiler Collection 15, a structural coverage metric that focuses on paths of execution through the program. Prime path coverage strikes a good balance between the number of tests and coverage, and requires that loops are taken, taken more than once, and skipped. We show that prime path coverage subsumes modified condition/decision coverage (MC/DC). We improve on the current state-of-the-art algorithms for enumerating prime paths by using a suffix tree for efficient pruning of duplicated and redundant subpaths, reducing it to $O(n^2m)$ from $O(n^2m^2)$, where n is the length of the longest path and m is the number of candidate paths. We can efficiently track candidate paths using a few bitwise operations based on a compact representation of the indices of the ordered prime paths. By analyzing the control flow graph, GCC can observe and instrument paths in a language-agnostic manner, and accurately report what code must be run in what order to achieve coverage.

1 Introduction

A major limitation in functional testing and dynamic software analysis is the *path coverage problem* [1, 17, 9], i.e. problems can only be detected in executed paths. Fuzzing [29] has proven to be an effective technique for exploring paths and detecting bugs, and there are algorithms [18] that try to generate minimal inputs for coverage. There have been proposed hardware extensions for dynamically expanding into untested paths [20], and automatic high coverage test generation [6]. Structural coverage analysis on its own remains a useful tool as it provides increased visibility into the code exercised when testing [14], an objective exit-criterion [7] for manual testing and test writing, and is powerful for checking assumptions. Ball and Larus [3] presented efficient path profiling on SPARC systems in the 1990s, but the industry has never widely adopted path coverage. They note that the cost of path profiling can be comparable to block- and edge profiling while providing far more information, and can form a basis for profile-guided optimizations. In this paper we describe our implementation of the prime path coverage support in the GNU Compiler Collection version 15.

While Li, Praphamontripong, and Offutt [19] found the cost/benefit of prime path coverage worse than edge-pair and definition-use, it is worth noting that prime path coverage still found more defects than the other coverage criterion used in the study. Durelli et al. [8] also found prime path coverage more effective at finding defects than edge-pair coverage at a moderate increase in test cases. Prime path coverage *almost* subsumes edge-pair coverage [1, 21]; self-edges require test paths with repeated vertices which are not simple. Since prime paths almost subsume edge-pairs it is reasonable that it is sensitive to the same problems, which seems supported by Li et al. [19]. Stronger coverage criteria have worse cost/benefit ratio in terms of number-of-bugs, but are more suited to detect *deeper* defects.

John Regehr [23] demonstrates a bug in Figure 1 that branch coverage, and even edge-pair coverage, will not detect, which he names *path sensitive condi-*

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

Figure 1: Assume `x` is a global int. Testing with 5 and 25 satisfies branch coverage, but does not trigger the bug (dereferencing `*p` when `p` is `NULL`). Prime path coverage would require both decisions to be `true`. The example is from Regehr [23].

tionals. These subtle interactions between conditionals is not uncommon in real-world programs. Regehr also notes that due to the *path explosion* problem [4] it is infeasible to achieve 100% path coverage. Even full path coverage is not, in itself, sufficient to find *all* bugs in real programs [22].

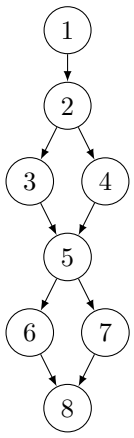
Even considering all of the limitations noted above, prime path coverage remains a powerful tool for software testing; it is a strong criterion that is simple to describe and understand, while simultaneously subsuming statement, branch, definition-use, and mostly edge-pair coverage [21, 1], which allows testing and development to focus on a single criterion. Path coverage, even limited forms such as simple path- and prime path coverage, is also able to observe *data dependent* infeasible paths, i.e. paths that cannot be taken due to contradictions or dependent values, defensive guards, and similar constructs. This makes path coverage a useful measurement even when not aiming for full coverage. While the main objective of testing should be to verify that the program complies with the functional requirements, prime path provides strong evidence that the program meets the requirements *and* that the requirements are complete.

2 Background

The tooling and algorithms in this space tend to work on a finite state machine (FSM) or graph representation of the programs [21, 1]. A control flow graph (CFG) is a graph representation of computation and control flow for a program module, e.g. a function in C. In the CFG, the vertices, or *basic blocks*, represent uninterruptible streams of computation while the edges characterize the control flow between the basic blocks. The CFG is the connected and possibly cyclic directed graph $G = (V, E, v_0, v_x)$ where V is a non-empty finite set of vertices, $E \subseteq \{ (u, v) \mid u \in V, v \in V \}$, and v_0, v_e are the entry and exit vertices so that for all vertices $v \in V$ there is a path $(v_0, \ldots, v)$ and $(v, \ldots, v_e)$. The entry- and exit vertices do no represent any computation, v_0 has no incoming edges, and v_e has no outgoing edges.

A *simple path* is a sequences of vertices $(v_1, \ldots, v_k)$ where all vertices are distinct. A *simple cycle* is a sequence of vertices $(v_1, \ldots, v_k)$ where $v_1 = v_k$ and all other vertices are distinct. Let P be simple paths, C simple cycles, and $R = P \cup C$; *prime paths* are the maximal objects of R. A prime path is *covered* if its vertices were visited in sequence during testing. Figure 2 shows the simple- and prime paths for a small function with no cycles. Counting simple paths is known to be #P-complete [27], so enumerating them is at least as hard, and finding prime paths from the set of simple paths quickly becomes impractical. Note that while there are fewer prime paths than simple paths, even small graphs may have a large number of prime paths [15]. Prime path coverage strikes a good balance between defect sensitivity and the number of required test paths. Notably, prime path coverage require loops to be taken, taken more than once, and skipped [1].

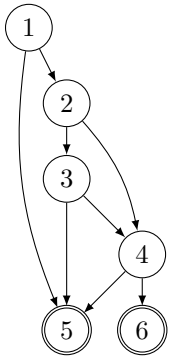
A coverage criterion C_1 *subsumes* C_2 if satisfying C_1 also guarantees satisfying C_2 [1]. For example, *vertex coverage*, or node coverage, is the criterion that each vertex in the CFG shall be visited at least once, and *edge coverage* is the criterion that each *edge* shall be taken at least once. Edge coverage subsumes vertex coverage because taking every edge in a connected graph guarantees visiting every vertex at least once. Prime path coverage subsumes most commonly used coverage metrics; statement, branch, condition, decision, node, edge, and definition-use [21, 8].



Simple paths [1 2] [1 2 3] [1 2 3 5]
[1 2 3 5 6] [1 2 3 5 6 8] [1 2 3 5 7]
[1 2 3 5 7 8] [1 2 4] [1 2 4 5]
[1 2 4 5 6] [1 2 4 5 6 8] [1 2 4 5 7]
[1 2 4 5 7 8] [2 3] [2 3 5] [2 3 5 6]
[2 3 5 6 8] [2 3 5 7] [2 3 5 7 8] [2 4]
[2 4 5] [2 4 5 6] [2 4 5 6 8] [2 4 5 7]
[2 4 5 7 8] [3 5] [3 5 6] [3 5 6 8]
[3 5 7] [3 5 7 8] [4 5] [4 5 6]
[4 5 6 8] [4 5 7] [4 5 7 8] [5 6]
[5 6 8] [5 7] [5 7 8] [6 8] [7 8]

Prime paths [1 2 3 5 6 8]
[1 2 3 5 7 8] [1 2 4 5 6 8]
[1 2 4 5 7 8]

Figure 2: The 41 simple paths and 4 prime paths for the CFG for two sequential decisions. Executing the prime paths would cover all simple paths.



(a) CFG

P_1	1 2 3 4 5
P_2	1 2 3 4 6
P_3	1 2 3 5
P_4	1 2 4 5
P_5	1 2 4 6
P_6	1 5

(b) Prime paths

	1	2	3	4	Path
1	0	0	0	0	P_5
2	0	0	0	1	P_4
3	0	0	1	0	P_5
4	0	0	1	1	P_4
5	0	1	0	0	P_2
6	0	1	0	1	P_1
7	0	1	1	0	P_3
8	0	1	1	1	P_3
9	1	0	0	0	P_6
10	1	0	0	1	P_6
11	1	0	1	0	P_6
12	1	0	1	1	P_6
13	1	1	0	0	P_6
14	1	1	0	1	P_6
15	1	1	1	0	P_6
16	1	1	1	1	P_6

(c) Truth table

Figure 3: BDD for the Boolean expression `a or (b and c) or d`. The double circle vertices 5 and 6 are the true and false outcome, respectively. Rows 1, 2, 5, 7, 9 (paths P_5, P_4, P_2, P_3, P_6) would achieve MC/DC, but row 6 (path P_1) would have to be included for prime path coverage.

3 Prime path coverage subsumes MC/DC

The unique property of MC/DC is the independence criterion [14], which states that each condition must be shown to take on both true and false while independently affecting the decision's outcome. A condition can independently affect the outcome if changing it while keeping the other conditions fixed also changes the outcome. Because the independence is shown from the *combination* of inputs, vertex coverage of a Boolean expression is not enough to satisfy MC/DC. A vertex can be visited and still not contribute towards MC/DC, and it is not obvious that it is subsumed by prime path coverage.

Proposition. *Every test vector required for MC/DC is tested with prime path coverage.*

Observation. *Every simple path is a subpath of a prime path.*

Proof. Any Boolean function has a canonical representation as a reduced ordered boolean decision diagram (BDD) [5], which is a rooted acyclic graph. Each combination of inputs map to a path in the BDD, which and MC/DC is achieved by taking a subset of the paths in the BDD. The BDD is acyclic so all paths are simple, and since prime path coverage requires taking all maximal simple paths it subsumes MC/DC. □

Which subset of paths MC/DC requires depends on the kind of MC/DC. For unique-cause MC/DC it is a subset where between pairs of inputs the final vertex changes when a only a single input condition differs. For masking MC/DC it is a set of paths that would visit all vertices while visits are filtered with a masking function [16]. The inverse is not true; MC/DC for a Boolean expression does not imply prime path coverage for the BDD, as shown in the counter example in Figure 3.

4 Enumerating prime paths

In this section we describe the algorithm in GCC that enumerates the prime paths. An efficient algorithm is important as the number of paths grows very fast with program complexity; for example, a sequence of n if-then-else with no nesting, as in Figure 2, has 2^n prime paths as every subsequent if-then-else would add another two prime paths to each of the 2^{n-1} prime paths up to it, and the relatively short function in Section 4 has 17 prime paths.

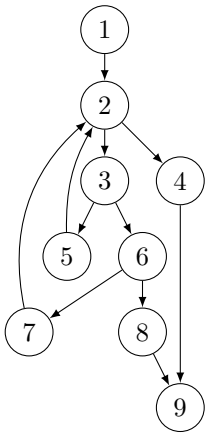
We build on two algorithms for prime path enumeration, described by Ammann and Offutt, and Fazli and Afsharchi. The algorithm by Ammann and Offutt [1] finds the maximal simple paths and simple cycles by starting with all single-vertex candidate paths and progressively extending each path with successors while maintaining the simple path- and cycle properties, When no more paths can be extended, the set of paths is pruned by removing all paths that also appear as subpaths, leaving only the paths that satisfy the *prime* criterion. Fazli and Afsharchi's compositional method [11] improve on this for many real-world programs. The high level steps of their algorithm are to (1) compute the component graph of the CFG; (2) generate the prime paths for each component and the component graph; (3) extract different intermediate paths from the components, and (4) merge intermediate paths to form the prime paths of the CFG. Fazli and Ebnenasir [12] propose a way to use the GPU to accelerate this design.

The effectiveness of the compositional method depends on the structure of the CFG, as it relies on building on the intermediate solutions from solving the smaller subgraphs inside the strongly connected components (SCC). This is ineffective when the component graph and the CFG are isomorphic, as in Figure 2, or when most of the CFG is within a single component, as in Figure 5. The former happens when there are no loops, and the latter when most of the function is inside a loop. In both cases GCC essentially falls back to the algorithm given by Ammann and Offutt.

Both algorithms need to filter non-prime paths, which can be done efficiently with a variant of generalized suffix trees [28, 10, 26, 13] over an integer

```
int search (int a[], int len, int key) {
  int low = 0;
  int high = len - 1;                    1
  while (low <= high) {                  2
    int mid = (low + high) / 2;
    if (a[mid] < key)                    3
      low = mid + 1;                     5
    else if (a[mid] > key)               6
      high = mid - 1;                    7
    else
      return mid;
  }                                      8
  return -1;                             4
}                                        9
```

(a) Code with the basic blocks annotated on the right



(b) CFG

1 2 3 5	3 6 7 2 4 9
1 2 3 6 7	5 2 3 5
1 2 3 6 8 9	5 2 3 6 7
1 2 4 9	5 2 3 6 8 9
2 3 5 2	6 7 2 3 5
2 3 6 7 2	6 7 2 3 6
3 5 2 3	7 2 3 6 7
3 5 2 4 9	7 2 3 6 8 9
3 6 7 2 3	

(c) Prime paths

Figure 4: A binary search with its control flow graph and 17 prime paths, showing that even short and relatively simple functions can have many prime paths. Note that block 9 is the *action* of the return, the transfer of control back to the caller.

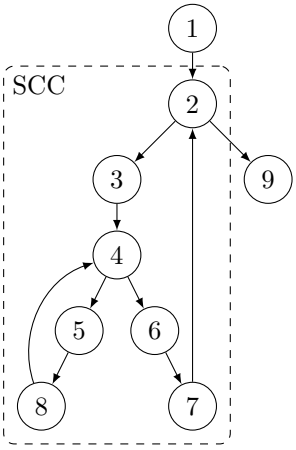


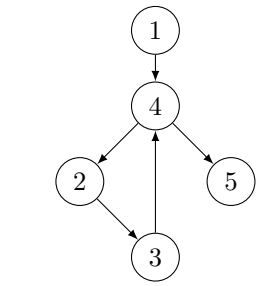
Figure 5: Most of the CFG is inside a single component.

alphabet of the vertex IDs. A generalized suffix tree is a suffix tree for a set of strings and can be constructed as a suffix tree of the concatenated strings and a special end-of-string character. Since prime paths are maximal simple paths or simple cycles they correspond to the strings in the suffix tree that do not prefix-match any suffix in the tree. Most traditional implementations of suffix trees store offsets into a string, but the prime paths are generated from the CFG and there is no explicit string. It is trivial to construct a string from candidate paths by concatenation. Alternatively, the tree can store substrings directly and not depend on an explicit string. This is how we implemented the suffix tree in GCC.

Figure 6 shows the suffix tree for a simple program and demonstrates the most important properties: path insertion, subpath detection, and path reconstruction. A path p is inserted by following the path from the root until either p is exhausted or the current vertex is a leaf, in which case the *final* mark on the leaf is cleared, the path is extended with the remaining vertices of p, and the new leaf is marked *final*. The height of the suffix tree is determined by the *longest* prime path and grows with the length of the paths, not with the *number* of paths. A suffix $p' = \textbf{tail}\ p$ is created and inserted, with the difference that the final vertex of p' is *not* marked final. This is repeated until p' is empty, or until p' fails to create a new vertex, as it means the remaining suffixes have already been inserted. Subpath detection is simple insertion; if a path is extended it is not prime and the final mark is removed, and if a superpath already existed p will be exhausted before reaching a leaf. For example, in Figure 6, inserting the path [1 4 2] would not reach the leaf 4, and inserting the path [1 2 5 6] would extend [1 2 5] and clear the final mark. Note that not all leafs are final; [3 4 5] is a suffix of [2 3 4 5] and not prime. Finally, there is path reconstruction; prime paths are the paths that end in a final vertex, and since the tree is *ordered* the paths can be enumerated in lexicographical order with a depth-first traversal of the tree.

Inserting a path P in the suffix tree is done by inserting the suffixes $P[1..n], P[2..n], \ldots, P[n]$ where $n = |P|$. Given m prime path candidates we need find the suffix tree, and by extension set of prime paths and eliminate redundant subpaths, in $O(n^2m)$. This is an improvement on the pruning step of the algorithm by Fazli et al [11, 12], which filters using a nested loop over the set of candidates in $O(n^2m^2)$. Note that paths tend to be short and n small; but the *number* of candidates m grows very fast.

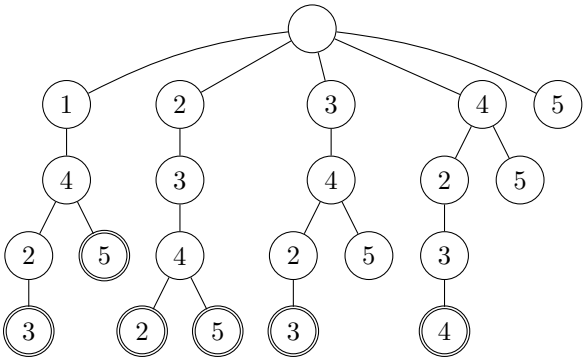
The path explosion problems makes it a practical necessity to limit analysis on programs that are too complex, i.e. programs with too many paths. The cost of enumerating the prime paths and emitting instructions grows with the number of prime paths, as well as the compiled object size. While there are techniques for estimating the number of $s - t$ paths [24] that could be used for estimating the prime path count, we employ a simple pessimistic heuristic; we maintain a running count of the number of paths and abort whenever it exceeds the given threshold. The heuristic is pessimistic as it counts inserts into the suffix tree without applying corrections for when paths are later subsumed. This is a pragmatic and fast solution that slightly over-counts paths and adds very little overhead, only an increment and a limits check, but might stop analysis on programs where the total path count is just under the threshold. The threshold default is quite high, 250000, and can be set by the user with a flag. This is deemed an acceptable solution to the path explosion problem given the remote likelihood



(a) CFG

1 4 2 3
1 4 5
2 3 4 2
2 3 4 5
3 4 2 3
4 2 3 4

(b) Prime paths



(c) Suffix tree

Figure 6: A CFG, its prime paths, and the suffix tree after all simple paths have been inserted. Every path from the root to a *final* (double) vertex is a prime path, and the ordered set of prime paths can be found by an ordered depth-first traversal of the tree left to right. Testing for any subpath is done with insertion: if no new vertices are created, the path is a subpath.

of any tester wanting to write that many test cases. As with any complexity problem, the better solution may be to refactor the program.

5 Instrumentation

It is necessary to keep track of both completed paths and partially taken paths. This can be done efficiently using only a few bitwise operations, similar to how GCC measures MC/DC [16]. The prime paths can be ordered lexicographically and a numerical identifier is assigned to each prime path based on its index in the ordered set. For each function, we add the persistent set P, which will be initialized with an empty set the first time the program is run in an auxiliary file called the counts or .gcda file. When the program is run it will, in the function prelude, initialize the function-local function set L. Each CFG vertex v is extended with three steps: recording/flushing, discarding, and initializing, in the order listed below.

Recording is updating the persistent counters P in the .gcda file with some of the paths in L. There may be more than one path which ends in the vertex v, and there may be paths that go through v which should not be recorded. $R(v)$ is the set of paths that end, and should be recorded, in v.

Discarding is removing the diverged-from paths from the candidate set L when taking an edge. $D(v)$ is the set of paths that should be discarded in v.

Initializing is starting the tracking of paths that begin in v by adding them to the current candidate set L. $I(v)$ is the set of paths that start, and should be initialized, in v.

These steps are set operations executed immediately upon entering the vertex v. **Recording** is $P \leftarrow P + (L \cap R(v))$, **discarding** is $L \leftarrow L - D(v)$, and **initializing** is $L \leftarrow L + I(v)$. For example, for the function `gnu_getcwd` in Figure 8, the vertex 3 will be transformed as in Figure 7. Note that for loops the recording and initialization will happen in the same vertex. This is fine; since L is initialized to the empty

```
P.add(intersection(R(3), L))
L.remove(D(3))
L.add(I(3))
val = getcwd (buffer, size)
if (val != 0)
```

Figure 7: Basic block 3 from Figure 8 and Figure 9a with prime path recording and tracking as functions on sets.

set, $L \cap R(v)$ will be empty on the first visit of v and adding it to P becomes a no-op.

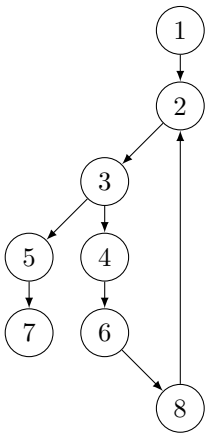
The paths and vertices in these examples are taken from Figure 8, and the full table of $R(v)$, $D(v)$ and $I(v)$ is shown in Figure 9c. Prime paths are recorded when entering the last vertex, e.g. $R(7) = \{ P_1, P_5 \}$, so P_1 and P_5 are recorded when entering vertex 7. Prime paths are discarded when they contain the predecessor p but not the vertex v. For example, $D(4) = \{ P_1, P_5 \}$ as there is an edge $(3, 4)$ and the vertex 4 is not in P_1 and has no predecessor in P_5. Finally, paths are initialized when entering the first vertex, so $I(4) = \{ P_5, P_6 \}$. These examples demonstrate why the order listed above is important for these operations. If discarding happened before recording, P_5 would never be covered as it is both discarded and recorded in 4. Similarly, if initialization happened before discarding then P_5 would be initialized in 4 before being immediately discarded.

With the set of prime paths enumerated, the functions over sets can be efficiently implemented with bitwise operations as follows; let B be the bitset representation of the set P where the the nth bit $B(n)$ corresponds to the path P_n, and $B_R(v), B_D(v), B_I(v)$ maps to $R(v), D(v), I(v)$ respectively. We have that $B_R(n) = 1$ if $P_n \in R(n)$, $B_D(n) = 1$ if $P_n \in D(n)$, and $B_I(n) = 1$ if $P_n \in I(n)$. All other bits are set to 0. A complete table of the bitset representations of the functions in Figure 9c is shown in Figure 10. B_L and B_P are both initialized to all zeros. Note that for the bitwise operations it is practical to invert B_D so applying it as a bitmask *preserves* non-discarded paths.

We can translate the set functions to work on

```
void *gnu_getcwd () {
  int size = 100;
  void *buffer = alloc (size);                1
  while (1) {                                 8
    void *val = getcwd (buffer, size);        2
    if (val != 0)                             3
      return buffer;                          5
    size *= 2;
    release (buffer);                         4
    buffer = alloc (size);                    6
  }                                           8
}                                             7
```

Figure 8: `gnu_getcwd` from unix-tree 2.0.4 [2] with the basic blocks annotated on the right.



(a) CFG

P_1	1 2 3 5 7
P_2	1 2 3 4 6 8
P_3	2 3 4 6 8 2
P_4	3 4 6 8 2 3
P_5	4 6 8 2 3 5 7
P_6	4 6 8 2 3 4
P_7	6 8 2 3 4 6
P_8	8 2 3 4 6 8

(b) Prime paths

	1	2	3	4	5	6	7	8
P_1	I			D			R	
P_2	I				D			R
P_3		RI			D			
P_4			RI		D			
P_5				DI			R	
P_6				RI	D			
P_7					D	RI		
P_8					D			RI

(c)

Figure 9: Instrumenting `gnu_getcwd` (Figure 8). The table in Figure 9c shows which paths are recorded (R), discarded (D), and initialized (I) when the vertex is visited.

v	$B_R(v)$	$B_D(I)$	$B_I(v)$
1	00000000	00000000	00000011
2	00000100	00000000	00000100
3	00001000	00000000	00001000
4	00100000	00010001	00110000
5	00000000	11101110	00000000
6	01000000	00000000	01000000
7	00010001	00000000	00000000
8	10000010	00000000	10000000

Figure 10: Bitset representation of Figure 9c

bitsets, so that for each vertex v; (1) **record** $P \leftarrow P + (L \cap R(v))$ becomes `P = P | (L & R[v])`; (2) **discard** $L \leftarrow L - D(v)$ becomes `L = L & ~D[v]`, and (3) **initialize** $L \leftarrow L + I(v)$ becomes `L = L | I[v]`. Full coverage is achieved when all bits in P are set.

To reduce the runtime and size overhead of instrumentation, certain unnecessary operations are eliminated. For example, if no paths are discarded in v then $B_D(v) = \emptyset$, which means $L = L - B_D(v)$ will have no effect and we do not need to emit instructions for updating L. The same techniques apply to the recording and initializing steps. It can be seen from the sparseness of the table in Figure 9c that many operations can be elided in practice; instructions have to be emitted if $R(v)$, $D(v)$, or $I(v)$ is non-empty, but only the instruction corresponding to that step. For example, the vertex 4 will be extended with all the steps, 2 would only need **record** and **discard**, and 5 only needs a single **discard** instruction to discard multiple prime paths A complete example of `gnu_getcwd` with instrumentation as if it was written in C is shown in Figure 11.

The number of prime paths of a function is usually much larger than the native word size or instruction operand sizes. In GCC 14, released in 2024, the size of the gcov data type used in the .gcda file is 32 or 64 bits, depending on the target architecture. The bitset of n bits can be partitioned into $k = \lceil \frac{n}{w} \rceil$ bins of w bits, where w is the number of bits in the gcov type. It is not necessary to emit instructions to update bins that are not affected. For example, for the `gnu_getcwd` in Figure 10 assuming $w = 4$ would split all bitsets in two. To perform the initialization `L = L | I[1]` where `I[1] = [0000, 0100]` it is sufficient to only apply the bitwise-or to the lower half. This optimization can *greatly* reduce the size of the compiled object files and improve runtime performance; in the the tree.c file in unix-tree 2.0.4 [2], each vertex typically interacted with up to 10% of the paths.

6 Reporting coverage

The coverage report is printed by the **gcov** program, which produces a report from the auxiliary notes and counts files [25]. The notes file, with the extension .gcno, is created by GCC when the program is compiled, and the counts file, with the extension .gcda, is created and updated by the instrumented program. The notes file stores information about the CFG, function names, line information, etc., and the counts file store the counters and coverage measurements. The prime paths are not stored explicitly in the notes file as it would be very large, but recomputed from the recorded CFG. If computing the prime paths for a function was aborted due to exceeding the path count threshold it is marked as such and gcov will not attempt to recompute the prime paths for that function. By using this information gcov can accurately report on prime path coverage and describe precisely how to cover the uncovered prime paths. Figure 12 shows an excerpt of a report on the `gnu_getcwd` function. This is the path $P_2 = [1\ 2\ 3\ 4\ 6\ 8]$ in Figure 9b, and gcov prints the lines of source source, and the basic block associated with it, in the order they must be executed to cover the prime path. GCC also offers a condensed line-oriented format intended for machine processing.

7 Conclusion and future work

This paper describes the implementation by the author of the prime path coverage support in GCC. We improve on the algorithms in this space by utilizing a suffix tree, which ensures fast removal of subsumed paths as well as providing a compact representation of the prime paths of a graph. The instrumented program is reasonably efficient; it only needs 1 bit per

```
extern uint P;
void *gnu_getcwd () {
  uint L = 0;
  L |= 00000011;
  int size = 100;
  void *buffer = alloc (size);                    1
  while (1) {                                     8
    P |= L & 00000100;
    L |= 00000100;
    void *val = getcwd (buffer, size);            2
    P |= L & 00001000;
    L |= 0001000;
    if (val != 0)                                 3
      L &= ~11101110;
      goto _return;                               5
    P |= L & 00100000;
    L &= ~00010001;
    L |= 00110000;
    size *= 2;
    release (buffer);                             4
    P |= L & 01000000;
    L |= 01000000;
    buffer = alloc (size);                        6
    P |= L & 10000010;
    L |= 10000100;
  }                                               8
_return:
  P |= L & 00010001;
  return buffer;
}                                                 7
```

Figure 11: `gnu_getcwd` with instrumentation as-if it was written in C. `P` is the persistent bitset, and the constant bitmasks are taken from the table in Figure 10. The label on each block is the is the CFG vertex ID as shown in Figure 8 and Figure 9a.

```
path 1 not covered:
BB 2:          5:void *getcwd (void *, int)
BB 2:          7:  int size = 100;
BB 2:          8:  void *buffer = alloc (size);
BB 3:         11:    void *value = getcwd (
    buffer, size);
BB 4:(false) 12:    if (value != 0)
BB 6:         14:    size *= 2;
BB 6:         15:    release (buffer);
BB 7:         16:    buffer = alloc (size);
BB 8:         10:  while (1) {
```

Figure 12: Path coverage report for `gnu_getcwd` the prime path $P_2 = [1\ 2\ 3\ 4\ 6\ 8]$ in Figure 9b. The columns are the basic block IDs, the edge/transition kind (decision), and the source code. Note that the basic blocks start at 2 as GCC reserves 0 and 1 for the entry- and exit blocks.

prime path, and the runtime bookkeeping only needs a few fast bitwise operations.

Finding the prime paths of a graph is an actively researched topic [11, 12], and improved algorithms would relax the current limitations and increase path count threshold and support more complex functions. Notably, GCC performs worse on graphs with large SCCs. Still, the practical limit is not computing the prime paths; the major slowdown are the later passes where GCC processes the extra instructions emitted by the instrumentation. Improvements in later compiler passes would greatly raise the prime path threshold.

Ball and Larus [3] show applications of *path profiling*, which can guide optimization similar to how GCC already uses edge- and block frequencies in profile-guided optimization. For both space and time efficiency the coverage instrumentation uses bitsets, but can be extended to also record path frequencies.

Our implementation uses a very simple heuristic to determine when coverage is too expensive, which is inaccurate as an upper bound, and computationally intensive as it enumerates prime paths up to the threshold. Good heuristics and estimation of graph complexity could be designed to efficiently reject functions. Fast approximations, even with reduced accuracy, would improve the responsiveness of the compiler

when faced with large complex programs with many functions that exceed the threshold.

Prime path coverage is not yet a widely used metric in the industry. While we have found prime path coverage to be an excellent tool to evaluate the complexity of code and as driver on where to spend effort during unit testing, further experiments could measure its effectiveness at finding defects, cost/benefit ratio, and relationship with testing the functional requirements.

References

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. 2nd ed. Cambridge University Press, 2016. ISBN: 978-1107172012.

[2] Steve Baker. *unix-tree*. Version 2.0.4. Sept. 6, 2022. URL: http://oldmanprogrammer.net/source.php?dir=projects/tree.

[3] Thomas Ball and James R. Larus. "Efficient path profiling". In: *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 29. Paris, France: IEEE Computer Society, 1996, pp. 46–57. ISBN: 0818676418.

[4] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. "RWset: attacking path explosion in constraint-based test generation". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Budapest, Hungary: Springer-Verlag, 2008, pp. 351–366. ISBN: 3540787992.

[5] Randal E. Bryant. "Symbolic Boolean manipulation with ordered binary-decision diagrams". In: *ACM Comput. Surv.* 24.3 (Sept. 1992), pp. 293–318. ISSN: 0360-0300. DOI: 10.1145/136035.136043. URL: https://doi.org/10.1145/136035.136043.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[7] John Chilenski. *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*. Tech. rep. DOT/FAA/AR-01/18. Apr. 2001. URL: https://rosap.ntl.bts.gov/view/dot/42764/dot_42764_DS1.pdf.

[8] Vinicius H.S. Durelli, Marcio E. Delamaro, and Jeff Offutt. "An experimental comparison of edge, edge-pair, and prime path criteria". In: *Science of Computer Programming* 152 (2018), pp. 99–115. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2017.10.003.

[9] Dawson Engler and Ken Ashcraft. "RacerX: effective, static detection of race conditions and deadlocks". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: Association for Computing Machinery, 2003, pp. 237–252. ISBN: 1581137575. DOI: 10.1145/945445.945468. URL: https://doi.org/10.1145/945445.945468.

[10] Martin Farach. "Optimal suffix tree construction with large alphabets". In: *Proceedings 38th Annual Symposium on Foundations of Computer Science*. 1997, pp. 137–143. DOI: 10.1109/SFCS.1997.646102.

[11] Ebrahim Fazli and Mohsen Afsharchi. "A Time and Space-Efficient Compositional Method for Prime and Test Paths Generation". In: *IEEE Access* 7 (2019), pp. 134399–134410. DOI: 10.1109/ACCESS.2019.2941429.

[12] Ebrahim Fazli and Ali Ebnenasir. "TPGen: A Self-Stabilizing GPU-Based Method for Prime and Test Paths Generation". In: (2022). arXiv: 2210.16998 [cs.SE]. URL: https://arxiv.org/abs/2210.16998.

[13] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. USA: Cambridge University Press, 1997. ISBN: 0521585198.

[14] Kelly J. Hayhurst et al. *A Practical Tutorial on Modified Condition/Decision Coverage*. Tech. rep. 20010057789. May 2001. URL: https://ntrs.nasa.gov/citations/20010057789.

[15] Garrett Kaminski et al. "An Evaluation of the Minimal-MUMCUT Logic Criterion and Prime Path Coverage". In: Jan. 2010, pp. 205–211.

[16] Jørgen Kvalsvik. *Modified Condition/Decision Coverage in the GNU Compiler Collection*. 2025. arXiv: 2501.02133 [cs.SE]. URL: https://arxiv.org/abs/2501.02133.

[17] Eric Larson and Todd Austin. "High Coverage Detection of Input-Related Security Faults". In: *12th USENIX Security Symposium (USENIX Security 03)*. Washington, D.C.: USENIX Association, Aug. 2003. URL: https://www.usenix.org/conference/12th-usenix-security-symposium/high-coverage-detection-input-related-security-faults.

[18] Nan Li, Fei Li, and Jeff Offutt. "Better Algorithms to Minimize the Cost of Test Paths". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 2012, pp. 280–289. DOI: 10.1109/ICST.2012.108.

[19] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. "An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage". In: May 2009, pp. 220–229. DOI: 10.1109/ICSTW.2009.30.

[20] Shan Lu et al. "PathExpander: Architectural Support for Increasing the Path Coverage of Dynamic Bug Detection". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. Jan. 2007, pp. 38–52. DOI: 10.1109/MICRO.2006.40.

[21] Vaclav Rechtberger, Miroslav Bures, and Bestoun S. Ahmed. *Overview of Test Coverage Criteria for Test Case Generation from Finite State Machines Modelled as Directed Graphs*. 2022. arXiv: 2203.09604 [cs.SE]. URL: https://arxiv.org/abs/2203.09604.

[22] John Regehr. *A Few Thoughts About Path Coverage*. URL: https://blog.regehr.org/archives/386 (visited on 07/30/2024).

[23] John Regehr. *Hiding Bugs from Branch Coverage*. URL: https://blog.regehr.org/archives/872 (visited on 07/30/2024).

[24] Ben Roberts and Dirk Kroese. "Estimating the Number of s-t Paths in a Graph". In: *J. Graph Algorithms Appl.* 11 (Jan. 2007), pp. 195–214. DOI: 10.7155/jgaa.00142.

[25] Richard M. Stallman et al. *GNU Compiler Collection 14 Manual, GCov section*. https://gcc.gnu.org/onlinedocs/gcc-14.2.0/gcc/Gcov-Data-Files.html. 2024.

[26] Esko Ukkonen. "On-line construction of suffix trees". In: *Algorithmica* 14.3 (Sept. 1995), pp. 249–260. ISSN: 0178-4617. DOI: 10.1007/BF01206331. URL: https://doi.org/10.1007/BF01206331.

[27] Leslie G. Valiant. "The Complexity of Enumeration and Reliability Problems". In: *SIAM Journal on Computing* 8.3 (1979), pp. 410–421. DOI: 10.1137/0208032. URL: https://doi.org/10.1137/0208032.

[28] Peter Weiner. "Linear pattern matching algorithms". In: *14th Annual Symposium on Switching and Automata Theory (swat 1973)*. 1973, pp. 1–11. DOI: 10.1109/SWAT.1973.13.

[29] Michał Zalewski. *Technical "whitepaper" for afl-fuzz*. URL: https://lcamtuf.coredump.cx/afl/technical_details.txt (visited on 07/30/2024).