# Tiered Live Variable Analysis for Heavy-Tailed CFG Distributions

Frank Kuehnel

December 21, 2025

**Abstract**

Live-variable analysis is typically taught as a monotone fixed-point computation with pessimistic worst-case iteration bounds. This paper shows that, on real compiler workloads, both control-flow structure and convergence behavior are far more regular than theory suggests.

Across 290,000 functions from the Go toolchain, CFGs fall into three structural regimes: 68% are acyclic, 24% contain exactly one loop kernel (median 6 blocks), and only 8% exhibit multiple cyclic regions. Sizes are heavy-tailed with dispersion index ≈175.

More strikingly, *every* strongly connected component in this dataset reaches a liveness fixed point in exactly three passes when using alternating traversal order: postorder, then a modified reverse postorder starting from an exit-like node, then postorder again (PO–MRPO–PO). This modified order is computed within each SCC rather than globally, and we lack a formal proof of why three passes suffice universally.

These observations motivate a tiered solver: a one-pass fast path for acyclic code, SCC decomposition to isolate cyclic regions, and the three-pass scheme only inside non-trivial SCCs. Our contribution is empirical—a large-scale CFG characterization and a convergence regularity that enables simple, predictable implementation.

## 1 Introduction

Live-variable analysis is the canonical backward data-flow pass underlying interference graphs, spill decisions, and many SSA-based optimizations. In principle it is a monotone fixed-point problem; in practice, its cost depends on how often information must circulate around cycles in the control-flow graph (CFG).

Textbook treatments reason about arbitrary CFGs and worst-case iteration [7, 1, 5]. Production compilers, however, run the same solver millions of times on a *workload distribution*. When that distribution stratifies into distinct structural regimes, a monolithic solver design leaves performance on the table.

**Iteration bounds and reducibility.** For *reducible* CFGs—those with well-nested loops and single loop headers—iterative data-flow solvers converge in

$d + 1$ passes, where $d$ is the loop nesting depth [1, 2]. *Irreducible* CFGs lack this structure: cycles may have multiple entry points, and no canonical "depth" exists. The classical analysis offers only the worst-case $O(|B|)$ bound, and compilers typically apply expensive node splitting to restore reducibility [3]. Our empirical finding that *three passes suffice even for irreducible SCCs* (Section 3.1) suggests that real-world irreducible regions are far more tractable than worst-case theory predicts.

**The core observation.** We analyzed 290,334 CFG instances from a full build-and-test of the Go toolchain. CFGs fall into three structural regimes based on cyclic complexity:

- **68% acyclic:** All SCCs are singletons—no iteration needed.
- **24% single-loop:** Exactly one non-trivial SCC (median 6 blocks).
- **8% complex:** Multiple non-trivial SCCs.

The dispersion index is ≈175 (vs. 1 for Poisson)—there is no "characteristic scale."

**Contributions.** This paper makes two contributions:

1. **Structural characterization of real-world CFGs.** We provide the first large-scale empirical analysis of CFG cyclic structure, showing that production workloads fall into distinct regimes that demand tiered treatment.
2. **Three-pass convergence for SCCs.** We report that data-flow equations over strongly connected components converge in exactly three passes when using alternating traversal order (postorder → reverse postorder → postorder). This is significantly better than the theoretical $O(|B|)$ bound.

**Positioning.** Live-variable analysis sits in the classic monotone-framework tradition [5, 4]. Worklist iteration and SCC decomposition are textbook tools [1, 10]. What we add is empirical: (i) real workloads mix regimes with different solver needs, and (ii) every SCC in our dataset converges in exactly three alternating passes—a regularity not previously documented for production workloads.

**Paper organization.** Section 2 presents the empirical CFG distribution and structural regimes. Section 3 presents the tiered solver with three-pass convergence. Appendices provide data-flow background, full statistics, and algorithm pseudocode.

| Metric | Median | $p_{90}$ | Max | Mean | Var | Var/Mean |
|---|---|---|---|---|---|---|
| Blocks | 9 | 44 | 12,676 | 20.10 | 3521 | 175 |
| SCCs | 7 | 31 | 12,676 | 15.31 | 2749 | 180 |

Table 1: CFG statistics from 290,656 functions in Go build-and-test workload.

# 2 The Statistical Reality of CFG Structure

Table 1 summarizes the distribution. The dispersion index $D = \mathrm{Var}(X)/\mathbb{E}[X]$ quantifies over-dispersion: $D \approx 1$ for Poisson processes, but we observe $D \approx 175$–$180$. This indicates the workload contains many distinct "regimes" of CFG complexity, not a single characteristic scale.

Attempting to fit a negative binomial model (the standard extension for over-dispersed counts) yields shape parameters near $r \approx 0.1$, indicating an extremely heavy tail. Yet even this model fits poorly—the empirical distribution shows multi-modal structure from distinct code populations (small leaf functions, medium inlined code, large state machines). Full details appear in Appendix B.

**Key takeaway.** The distribution is *heterogeneous* and *heavy-tailed*. Solver design must account for both the common case (small, simple CFGs) and the tail (large, complex CFGs that dominate worst-case time and memory).

## 2.1 Structural Regimes: Where Cycles Live

Size statistics alone do not capture what matters for iterative solvers: *cyclic structure*. A CFG with 100 blocks but no back edges needs only one pass; a 20-block CFG with nested loops may need many. We classified each CFG by its non-trivial SCC count (SCCs with >1 block).

| Regime | Count | Fraction |
|---|---|---|
| Acyclic (0 non-trivial SCCs) | 197,091 | 67.9% |
| Single-loop (1 non-trivial SCC) | 68,357 | 23.5% |
| Multi-loop (2+ non-trivial SCCs) | 24,886 | 8.6% |

Table 2: CFG structural regimes. Over two-thirds of CFGs have no cycles at all.

Figure 1 shows the distribution of largest SCC size across all CFGs. The dominant spike at size 1 represents the acyclic majority—nearly 200,000 CFGs where every block is its own SCC.

For the 24% of CFGs with exactly one non-trivial SCC, Figure 2 shows the size distribution of that single loop kernel. The median is just 6 blocks, with 90% under 27 blocks—but the tail extends to 376 blocks.

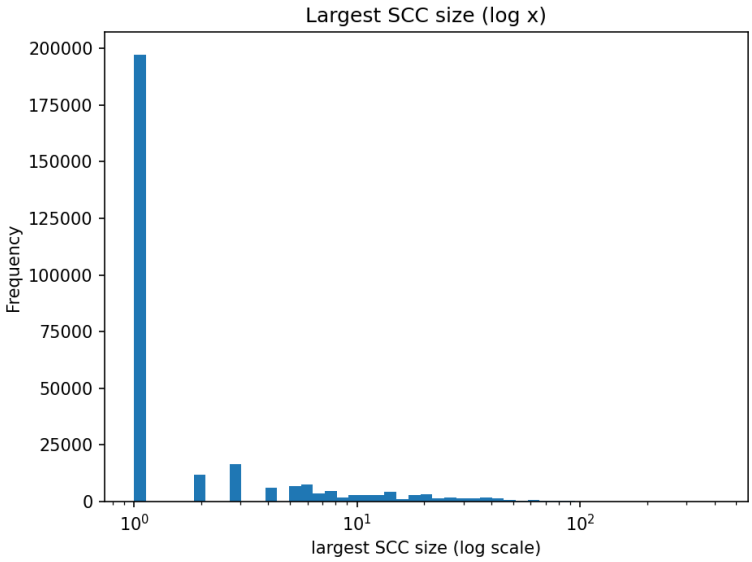



Figure 1: Distribution of largest SCC size (log scale). The spike at 1 represents the 68% of CFGs with no cycles. The tail extends to SCCs with hundreds of blocks.

**Implications for iteration.** These regimes have direct algorithmic consequences:

- **Acyclic (68%):** One pass suffices. No worklist, no convergence check needed.
- **Single-loop (24%):** Iteration confined to one small region (median 6 blocks). The rest of the CFG is processed in a single pass.
- **Multi-loop (8%):** Multiple iteration regions, but still isolated from each other by SCC decomposition.

A tiered solver that exploits this structure can avoid iteration overhead for 68% of CFGs, minimize it for another 24%, and reserve full generality for the remaining 8%. The key insight is that SCC decomposition handles all three regimes *without explicit case analysis*: acyclic CFGs yield all-singleton SCCs, single-loop CFGs yield one non-trivial SCC plus singletons, and multi-loop CFGs yield multiple independent SCCs.

# 3 A Tiered Solver Architecture

The structural regimes motivate a solver that adapts to CFG structure automatically. Algorithm 1 presents the complete approach: SCC decomposition

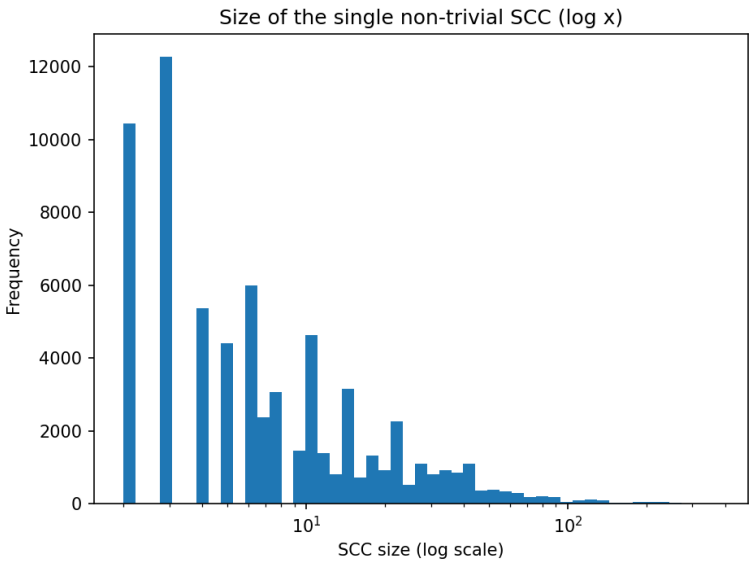



Figure 2: Size of the single non-trivial SCC (when exactly one exists). Median 6 blocks, $p_{90} = 27$, max = 376.

partitions the CFG, and a three-pass scheme with alternating traversal order solves each cyclic component.

**Algorithm 1** Tiered Liveness Solver

**Require:** CFG $G$ with blocks $B$, edges $E$, entry node $e$
**Ensure:** LiveIn($b$) and LiveOut($b$) for all $b \in B$

```
1: function ComputeLive(G)
2:     if G has no loops then                          ▷ 68% of CFGs: fast path
3:         Process all blocks in postorder (one pass)
4:         return
5:     end if

6:     sccs ← SCCPartition(G)                                  ▷ Kosaraju–Sharir
7:     for each K ∈ sccs in reverse topological order do
8:         if |K| = 1 then
9:             ProcessBlock(K[0])                          ▷ Singleton: one pass
10:        else
11:            SolveThreePass(K)                           ▷ Cyclic: three passes
12:        end if
13:    end for
14: end function

15: function SolveThreePass(K)
16:    po, mrpo ← AlternatingOrders(K)
17:    for b ∈ po do ProcessBlock(b)
18:    end for                                                    ▷ Pass 1: PO
19:    for b ∈ mrpo do ProcessBlock(b)
20:    end for                                                   ▷ Pass 2: MRPO
21:    for b ∈ po do ProcessBlock(b)
22:    end for                                                    ▷ Pass 3: PO
23: end function

24: function ProcessBlock(b)
25:    LiveOut(b) ← ⋃_{s∈Succ(b)} LiveIn(s)
26:    LiveIn(b) ← Use(b) ∪ (LiveOut(b) \ Def(b))
27: end function
```

### 3.1 Traversal Orders

The three-pass algorithm uses two traversal orders computed by DFS within each SCC:

**Postorder (PO):** DFS from an entry-like node, recording each node *after* visiting its successors. Result: exit-like nodes appear first, entry last.

**Modified reverse postorder (MRPO):** A *second* postorder traversal, but starting from an exit-like node (the first element of PO). Within an SCC,

this produces a traversal in the opposite direction—not simply the reversed list.

The key distinction from standard RPO: we compute a fresh DFS from a different starting point rather than reversing the original order. This only works within SCCs, where every node is reachable from every other (see Appendix D).

**Why alternation accelerates convergence.** Consider a cycle $b_1 \to b_2 \to b_3 \to b_1$. Suppose PO from $b_1$ yields $[b_3, b_2, b_1]$, and MRPO from $b_3$ yields $[b_1, b_2, b_3]$:

1. **Pass 1 (PO):** Facts flow $b_3 \to b_2 \to b_1$.
2. **Pass 2 (MRPO):** Facts flow $b_1 \to b_2 \to b_3$—the opposite direction.
3. **Pass 3 (PO):** Reconciles any remaining cross-cycle dependencies.

Single-direction iteration requires $n$ passes for an $n$-block cycle. Alternating directions propagate information both ways, converging in constant passes.

**Empirical result.** All 290,000 functions in our dataset converge in exactly three passes—including irreducible control flow and complex loop nests. While we lack a formal proof, this empirical bound is robust across diverse code patterns.

### 3.2 Complexity

Each ProcessBlock call is $O(|\text{Succ}(b)|)$. Within an SCC of size $|K|$, three passes cost $O(|K| + |E_K|)$. Summing over all SCCs (which partition the CFG):

$$\sum_K O(|K| + |E_K|) = O(|B| + |E|)$$

The algorithm is linear in CFG size, regardless of cyclic structure. The key wins are:

- **Acyclic CFGs (68%):** Skip SCC computation entirely; single pass suffices.
- **Fixed iteration count:** No convergence checking, predictable performance.
- **Sequential access:** Cache-friendly traversal, no dynamic worklist overhead.

## A Solving the Data-Flow Equations

This appendix reviews the standard formulation of live-range analysis as a data-flow problem and the iterative algorithm used to solve it [7, 5].

## A.1 The Data-Flow Equations

Let the CFG have basic blocks $b \in B$ with edges $b \to s$ to successors. For each block, define:

- $\mathrm{Def}(b)$: variables defined (written) in $b$
- $\mathrm{Use}(b)$: variables used (read) in $b$ before any local definition

Live variable analysis computes, for each block $b$:

$$\mathrm{LiveOut}(b) = \bigcup_{s \in \mathrm{Succ}(b)} \mathrm{LiveIn}(s) \tag{1}$$

$$\mathrm{LiveIn}(b) = \mathrm{Use}(b) \cup (\mathrm{LiveOut}(b) \setminus \mathrm{Def}(b)) \tag{2}$$

This is a *backward* data-flow problem: information flows from successors to predecessors, opposite to control flow [1].

## A.2 The Monotone Framework

The equations above instantiate a *monotone framework* [4, 5]. The key properties ensuring a well-defined solution are:

1. **Lattice structure.** The domain is the powerset $2^V$ of program variables, ordered by subset inclusion. The bottom element is $\emptyset$; the top is $V$.
2. **Monotone transfer functions.** The transfer function $f_b(X) = \mathrm{Use}(b) \cup (X \setminus \mathrm{Def}(b))$ is monotone: $X \subseteq Y \Rightarrow f_b(X) \subseteq f_b(Y)$.
3. **Finite height.** The lattice has height $|V|$, bounding the number of times any fact can change.

These properties guarantee that iterative application of the equations converges to the *least fixed point*—the smallest solution satisfying all constraints [8].

## A.3 The Standard Iterative Algorithm

The textbook algorithm processes blocks in reverse postorder (RPO) of a depth-first traversal [7, 1, 9]:

1. Compute a DFS postorder of the CFG.
2. Initialize $\mathrm{LiveIn}(b) = \mathrm{LiveOut}(b) = \emptyset$ for all $b$.
3. Repeat in reverse postorder until no changes:
   (a) Compute $\mathrm{LiveOut}(b)$ from successors via Equation 1.
   (b) Compute $\mathrm{LiveIn}(b)$ via Equation 2.

For a backward analysis, reverse postorder processes blocks "close to exits first," propagating information in the predominant direction of data flow.

## A.4 Complexity and Convergence

**Per-iteration cost.** Each pass visits every block and, for each block, examines its successors and performs set operations. With $|B|$ blocks and $|E|$ edges, one iteration costs $O(|B| + |E|)$ assuming set operations are $O(|V|)$ or use efficient representations (bit vectors, sparse sets).

**Number of iterations.** In the worst case, a single new fact may propagate one block per iteration. For a CFG with a long chain or deep loop nest, this yields $O(|B|)$ iterations, giving overall worst-case complexity:

$$O(|B| \cdot (|B| + |E|)) = O(|B|^2 + |B| \cdot |E|)$$

On reducible CFGs (most structured programs), convergence is typically much faster—often 2–3 passes suffice in practice [1]. However, irreducible control flow or adversarial loop structures can approach the worst case.

**The ordering assumption.** The standard algorithm assumes a *fixed* DFS-based ordering computed once before iteration. This ordering is optimal for acyclic graphs (one pass suffices) and near-optimal for reducible graphs with natural loops.

Notably, **there has been little research on dynamic or adaptive ordering schemes** that might accelerate convergence on complex CFGs. The literature universally adopts DFS postorder [7, 1, 9, 8].

**Alternating order: an empirical improvement.** Our empirical study (Section 3.1) reveals that *alternating* between postorder and reverse postorder dramatically improves convergence. Specifically, processing blocks in the sequence postorder $\rightarrow$ reverse postorder $\rightarrow$ postorder achieves convergence in exactly three passes for *all* SCCs in our 290,000-function dataset—including irreducible control flow and complex loop structures.

This is significantly better than the theoretical $O(|B|)$ worst case. While we lack a formal proof that three passes suffice universally, the empirical evidence suggests this alternating strategy exploits structure in real-world CFGs that single-direction iteration misses.

## A.5 Distance-to-Next-Use Extension

For register allocation, we often want not just *whether* a variable is live, but *how soon* it will be used [6]. We generalize to a map $L_b : V \rightarrow (\mathbb{N} \cup \{\infty\})$ where $L_b(v)$ estimates instructions until next use. At joins:

$$L_b(v) = \min_{s \in \mathrm{Succ}(b)} \left( \delta(b, s) + L_s^{\mathrm{in}}(v) \right)$$

where $\delta(b, s) \geq 1$ is an edge cost (modeling branch likelihood, transfer overhead, etc.).

This remains a monotone framework: the lattice is $(\mathbb{N} \cup \{\infty\})^V$ ordered pointwise by $\geq$, and distances decrease monotonically from $\infty$ toward smaller values as uses become reachable. The same iterative algorithm applies, with the same complexity bounds.

### A.6 Motivation for SCC-Based Solving

The theoretical $O(|B|^2)$ worst case motivates the SCC-based approach in Section 3. By decomposing the CFG into strongly connected components:

- Acyclic portions (68% of CFGs in our study) require exactly one pass.
- Cyclic SCCs are solved independently, confining iteration to the cyclic subgraph.
- The condensation DAG ensures no redundant reprocessing across SCCs.

Combined with the three-pass alternating-order algorithm (Section 3.1), this approach achieves $O(|B| + |E|)$ complexity on *all* CFGs in our study—not just typical ones, but including the complex tail cases that would otherwise dominate compile time.

## B Statistical Analysis Details

This appendix provides the full statistical analysis of CFG structure from our Go toolchain study.

### B.1 Dataset

Each line of the collected `*_scc.csv` files corresponds to one analyzed CFG instance, recording the number of basic blocks and the number of SCC "kernels" (SCCs in the condensation). Across 240 input files, the dataset contains $n =$ 290,656 CFG instances.

### B.2 Heavy Tails and Over-Dispersion

| Metric | $n$ | Min | Median | $p_{90}$ | Max | Mean | Var | Var/Mean |
|---|---|---|---|---|---|---|---|---|
| Blocks | 290,656 | 3 | 9 | 44 | 12,676 | 20.10 | 3521.00 | 175.14 |
| SCCs | 290,656 | 2 | 7 | 31 | 12,676 | 15.31 | 2749.30 | 179.59 |

Table 3: Complete CFG statistics. The dispersion index is ≈175–180, far exceeding Poisson ($D = 1$).

The dispersion index $D = \mathrm{Var}(X)/\mathbb{E}[X]$ diagnoses over-dispersion. For a Poisson model, $D \approx 1$. Observing $D \approx 175$ implies variance two orders of magnitude larger than a homogeneous process predicts.

One interpretation: if $X \mid \Lambda \sim \text{Poisson}(\Lambda)$ with latent rate $\Lambda$ varying across CFG instances, then

$$D = 1 + \frac{\text{Var}(\Lambda)}{\mathbb{E}[\Lambda]}.$$

A dispersion index near 175 implies $\text{Var}(\Lambda) \approx 174\,\mathbb{E}[\Lambda]$: the workload spans many "regimes" of CFG complexity.

## B.3 Why Negative Binomial Fits Poorly

The negative binomial (NB) extends Poisson for over-dispersed counts via a Gamma–Poisson mixture. Moment-matching yields extremely small shape parameters ($r \approx 0.115$ for blocks, $r \approx 0.086$ for SCCs), indicating a very heavy tail.

Yet this simple NB model remains a poor global fit for several reasons:

1. **Structural constraints.** CFGs have minimum sizes and discrete construction artifacts (entry/exit blocks, synthetic lowering blocks) that create mass at specific small counts not captured by continuous mixtures.

2. **Multi-modal mixtures.** Small leaf functions, medium inlined functions, generated code, and large dispatcher functions form distinct populations. A single Gamma mixing distribution cannot capture multiple modes.

3. **Different tail mechanisms.** The far tail often arises from specific sources (parser tables, regexp engines, large switch lowering) that may follow lognormal or power-law-like behavior rather than NB.

4. **SCC structure is not independent.** SCC count correlates with blocks but depends on loop structure, irreducibility, and canonicalization—additional heterogeneity beyond simple count models.

## B.4 Implications

The poor global NB fit is *good news*: it indicates exploitable structure. Stratifying by package, compilation stage, or structural features (presence of large switch lowering, irreducible loops, inlining depth) often yields per-stratum distributions far less pathological. Those strata-level models are the right abstraction for predicting and improving register-allocation performance.

## B.5 Structural Regime Analysis

Beyond size distributions, we analyzed the cyclic structure of each CFG by counting non-trivial SCCs (those with more than one block).

For the 68,357 CFGs with exactly one non-trivial SCC, Table 5 characterizes the size of that single loop kernel.

We also measured what fraction of each CFG's nodes participate in non-trivial SCCs:

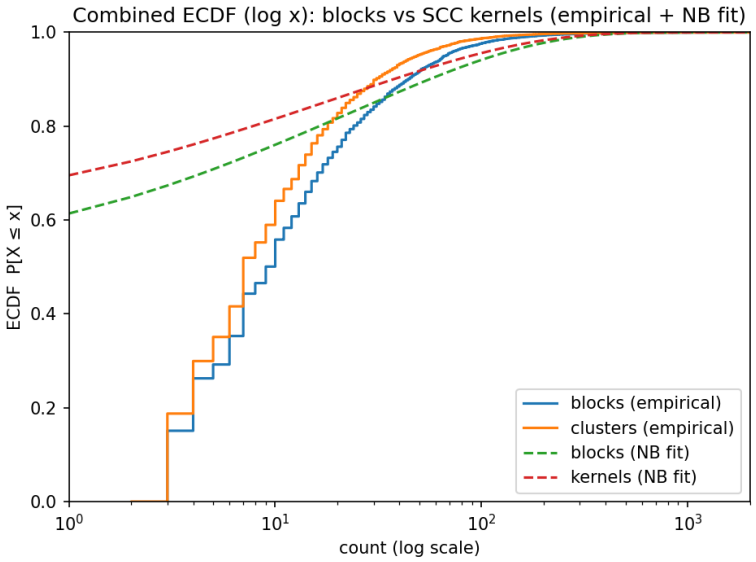



Figure 3: Empirical ECDFs (log $x$-axis) with moment-matched NB overlay. The poor fit confirms multi-regime workload structure.

| Non-trivial SCC count | CFGs | Fraction |
|---|---|---|
| 0 | 197,091 | 67.9% |
| 1 | 68,357 | 23.5% |
| 2 | 14,918 | 5.1% |
| 3 | 5,273 | 1.8% |
| 4 | 1,884 | 0.6% |
| 5+ | 2,811 | 1.0% |

Table 4: Distribution of non-trivial SCC counts. The tail drops off rapidly.

- **Median:** 0% (the acyclic majority)
- $p_{90}$**:** 60%
- $p_{99}$**:** 87.5%
- **Max:** 99.1%

Even in CFGs with cycles, the cyclic portion is often a minority of the total blocks. This reinforces the value of SCC decomposition: iteration is confined to a (usually small) subset of the CFG.

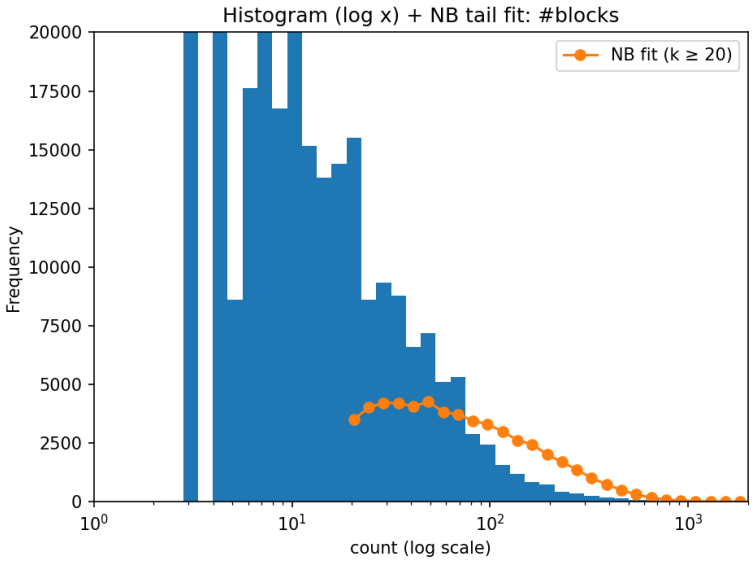



Figure 4: Block count histogram (log $x$-axis) with NB tail overlay. Even in the tail region, the fitted curve diverges from empirical counts.

| Metric | $n$ | Min | Median | $p_{90}$ | $p_{99}$ | Max |
|---|---|---|---|---|---|---|
| Single non-trivial SCC size | 68,357 | 2 | 6 | 27 | 86 | 376 |

Table 5: Size statistics for the single non-trivial SCC (conditional on exactly one existing).

# C SCC Algorithm Pseudocode

We use Kosaraju–Sharir [10] for SCC decomposition. Compared to Tarjan's algorithm [11]:

- Straightforward iterative implementation without explicit stack management.
- No auxiliary data (lowlink, index) required on graph nodes.
- The postorder from the first pass is typically already cached by the compiler, making that phase effectively free.

Using BFS instead of DFS for the second pass simplifies implementation while maintaining correctness.

## C.1 Algorithm Overview

1. **Forward pass**: Compute postorder traversal via DFS on forward edges.

2. **Reverse pass**: Process blocks in reverse postorder, performing BFS on predecessor edges to discover each SCC.

Processing in reverse postorder on the transposed graph ensures that starting a new component cannot reach any previously discovered component.

## C.2 Pseudocode

**Algorithm 2** Kosaraju–Sharir SCC Partition

**Require:** CFG $G = (B, E)$ with blocks $B$ and edges $E$
**Ensure:** List of SCCs in topological order of the condensation DAG

```
1:  function SCCPartition(G)
2:      po ← Postorder(G)                          ▷ DFS postorder on forward edges
3:      seen ← ∅
4:      reachable ← {b.ID : b ∈ po}
5:      result ← [ ]

6:      for i ← |po| − 1 downto 0 do               ▷ Reverse postorder
7:          leader ← po[i]
8:          if leader.ID ∈ seen then
9:              continue
10:         end if

11:         scc ← BFSReversed(G, leader, seen, reachable)
12:         Append(result, scc)
13:     end for

14:     return result
15: end function
```

**Algorithm 3** BFS on Reversed Edges

**Require:** Block *leader*, sets *seen* and *reachable*
**Ensure:** SCC containing *leader*; updates *seen* in place

```
1: function BFSReversed(G, leader, seen, reachable)
2:     queue ← [leader]
3:     scc ← [ ]
4:     seen ← seen ∪ {leader.ID}

5:     while queue ≠ ∅ do
6:         b ← Dequeue(queue)
7:         Append(scc, b)

8:         for e ∈ Preds(b) do                    ▷ Traverse reversed edges
9:             pred ← e.block
10:            if pred.ID ∈ reachable and pred.ID ∉ seen then
11:                seen ← seen ∪ {pred.ID}
12:                Enqueue(queue, pred)
13:            end if
14:        end for
15:    end while

16:    return scc
17: end function
```

### C.3 Properties

1. The first SCC contains only the entry block (assuming no predecessors).
2. Unreachable blocks are excluded.
3. SCCs are returned in topological order of the condensation DAG.
4. Block order within each SCC is deterministic for a given input.

**Complexity.** Time: $O(|B| + |E|)$—each block and edge visited once per pass. Space: $O(|B|)$ for visited sets and queue.

## D Why Modified Reverse Postorder Requires SCC Isolation

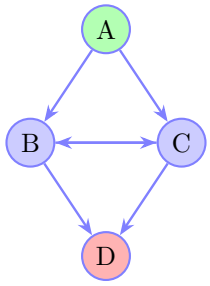
Postorder is commonly considered the most advantageous traversal for solving backward data-flow equations on general CFGs. A postorder traversal performs depth-first search from a designated start node, recording each node *after* its successors have been visited. This aligns with the natural program structure:

functions have a single entry, loops have a single header, and control flows "downward" through the CFG.

Reverse postorder (RPO)—simply the reversed list—places the entry first and exits last, approximating topological order for acyclic regions. For backward analysis, RPO ensures successors are processed before predecessors, which is optimal when no cycles exist.

**The problem with global RPO on irreducible CFGs.** Our three-pass algorithm uses a *modified* reverse postorder (MRPO) that differs from simply reversing the global postorder. The modification: compute postorder starting from an *exit-like* node (the first element of the original postorder) rather than the entry. This produces a traversal that naturally moves "toward" the entry, complementing the original postorder's "away from entry" direction.

Crucially, this only works within an SCC. Consider the following irreducible CFG:




Here A is the entry (green) and D is the exit (red). Nodes B and C form an irreducible cycle with two entry points.

**Global postorder from A:** $[\mathrm{D}, \mathrm{C}, \mathrm{B}, \mathrm{A}]$

The first element is D. If we compute postorder starting from D on the *full graph*, we get only [D]—D has no successors to explore. The "modified reverse postorder" degenerates to a single node, losing the alternating-direction benefit entirely.

**Within the SCC {B, C}:** Starting postorder from B within the restricted subgraph yields $[\mathrm{C}, \mathrm{B}]$. Starting from C (the first element) yields $[\mathrm{B}, \mathrm{C}]$. Now we have two genuinely alternating orders that cover the entire cycle.

**The key insight.** The modified reverse postorder works because, within an SCC, every node is reachable from every other node. Starting DFS from an "exit-like" node (one that appeared early in the original postorder) produces a traversal that explores the SCC in the opposite direction. On the full CFG, exit nodes may have no successors at all, making the second DFS trivial.

This is why our tiered architecture isolates SCCs before applying the three-pass scheme: the alternating orders are only meaningful within strongly connected regions.

# References


[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

[2] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, 1977.

[3] Johan Janssen and Henk Corporaal. Controlled node splitting. In *Proceedings of the 10th International Conference on Compiler Construction (CC)*, pages 44–58. Springer, 2001.

[4] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[5] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206. ACM, 1973.

[6] D. Ryan Koes and Seth Copen Goldstein. Register allocation deconstructed. Technical Report CMU-CS-09-131, Carnegie Mellon University, 2009.

[7] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[8] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[9] Fabrice Rastello. *SSA-based Compiler Design*. PhD thesis, INRIA, 2013. Habilitation thesis.

[10] Micha Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[11] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.