Two-Path Live Variable Analysis for Heavy-Tailed CFG Distributions

Frank Kuehnel

December 25, 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 291,000 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 9% exhibit multiple cyclic regions. Even within loopy CFGs, 93% of SCCs are singletons.

More strikingly, *every* strongly connected component in this dataset reaches a liveness fixed point in at most three passes when using alternating traversal order (entryward–exitward–entryward). This bound holds for both reducible and irreducible control flow, though we lack a formal proof.

These observations motivate a two-path solver: a one-pass fast path for acyclic CFGs (68%), and SCC decomposition with up to three passes for all loopy CFGs (32%). Benchmarks show improvements across the board: 7–15% on reducible loops, 1–4% on irreducible CFGs, and even 0.3–1.2% on acyclic code, yielding a 5.4% geometric mean improvement. The design is simpler than worklist-based approaches—no priority queues, no convergence checking, and uniform treatment of reducible and irreducible CFGs.

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 a worst-case $O(|B|)$ bound, and compilers often apply expensive node splitting to restore reducibility [3].

SCC decomposition sidesteps the reducibility question: it partitions *any* CFG into acyclic inter-SCC structure plus isolated cyclic kernels. Our empirical finding that *three passes suffice within each SCC* (Section 3.3) suggests that even complex cyclic regions are more tractable than worst-case theory predicts.

The core observation. We analyzed 290,904 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).
- **9% complex:** Multiple non-trivial SCCs.

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

Contributions. This paper makes two empirical contributions:

1. **Structural characterization of CFGs at scale.** We show that production workloads fall into distinct regimes (acyclic, single-loop, multi-loop) that warrant specialized solver design.
2. **Three-pass convergence within SCCs.** We observe that data-flow iteration over every SCC in our dataset converges in exactly three passes when using alternating traversal order (PO–MRPO–PO). We lack a formal proof, but this empirical bound is far better than the theoretical $O(|B|)$ worst case.

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 stratify into regimes with different solver needs, and (ii) every SCC in our dataset converges in exactly three alternating passes—a regularity not previously documented.

Paper organization. Section 2 presents the empirical CFG distribution. Section 3 presents the two-path 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	3516	175
SCCs	7	31	12,676	15.31	2747	179

Table 1: CFG statistics from 290,904 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 $D \approx 175$–179. 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	Characteristic
Acyclic	197,304	67.8%	All SCCs singletons; one pass suffices
Single-loop	68,426	23.5%	One non-trivial SCC (median 6 blocks)
Multi-loop	25,174	8.7%	Multiple non-trivial SCCs
Within the 93,600 loopy CFGs (total 2,107,393 SCCs):			
Singleton SCCs	1,958,878	92.95%	No iteration needed
Size-2 SCCs	22,817	1.08%	Trivial cycles
Size-3 SCCs	37,365	1.77%	Small loops
Larger SCCs	88,333	4.19%	Requires iteration

Table 2: CFG structural regimes and SCC composition. Even within loopy CFGs, 93% of SCCs are singletons.

Metric	n	Min	Median	p_{90}	p_{99}	Max
Single non-trivial SCC size	68,426	2	6	27	86	581
Fraction of CFG in cycles	290,904	0%	0%	60%	87.5%	99.4%

Table 3: Non-trivial SCC statistics. The median fraction of 0% reflects the acyclic majority; even at p_{90}, only 60% of nodes participate in cycles.

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.

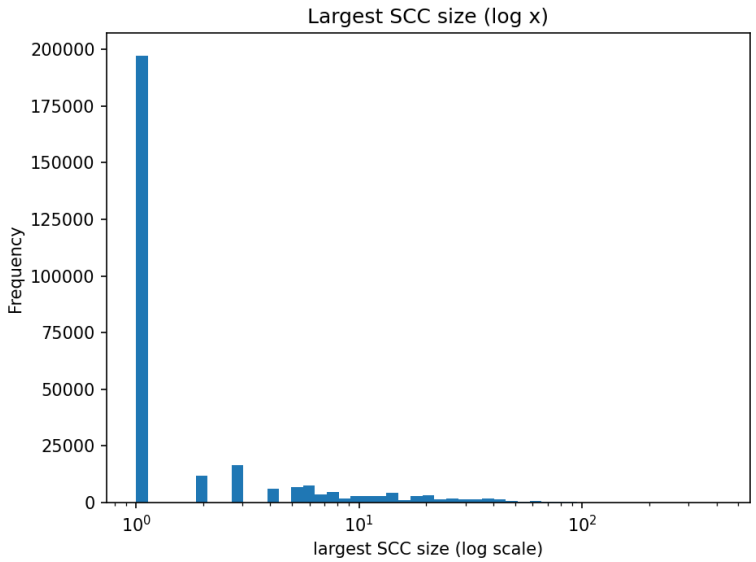


Figure 1: Distribution of largest SCC size (log scale). The spike at 1 represents the 68% of CFGs with no cycles. The tail extends to SCCs with hundreds of blocks.

Implications for iteration. These regimes have direct algorithmic consequences. Acyclic CFGs (68%) need only one pass. Single-loop CFGs (24%) confine iteration to one small region (median 6 blocks). Multi-loop CFGs (9%) have multiple iteration regions, but SCC decomposition isolates them. A two-path solver can avoid iteration overhead for 68% of CFGs, minimize it for another 24%, and reserve full generality for the remaining 9%.

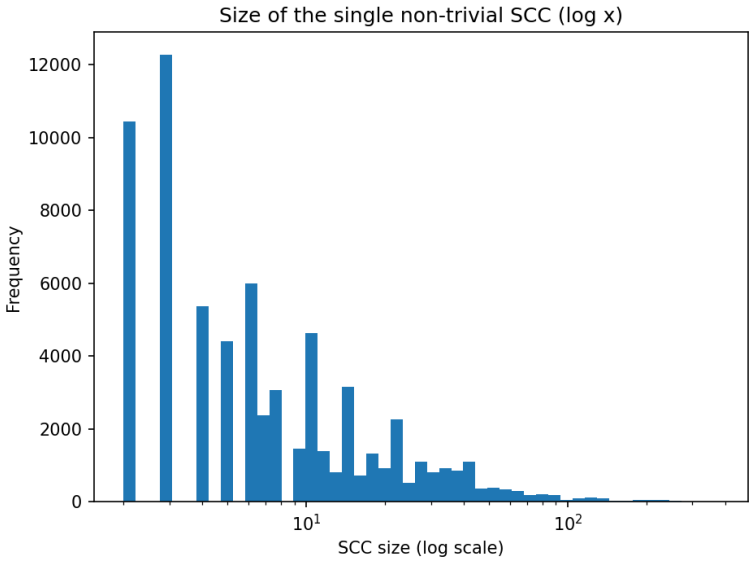


Figure 2: Size of the single non-trivial SCC (when exactly one exists). Median 6 blocks, $p_{90} = 27$, max = 581.

3 A Two-Path Solver Design

The structural regimes motivate a solver that adapts to CFG structure automatically. Algorithm 1 presents the implementation with two paths: a fast acyclic path and an SCC-based path for CFGs with loops.

Algorithm 1 Two-Path Liveness Solver

Require: CFG G with blocks B, edges E, loop nest information
Ensure: LiveIn(b), LiveOut(b), and Desired(b) for all $b \in B$

```
1:  function ComputeLive(G)
2:      if G has no loops then                          ▷ 68%: acyclic fast path
3:          for b ∈ PostOrder(G) do
4:              ProcessBlock(b)
5:              ProcessDesired(b)
6:          end for
7:          return
8:      end if

9:      ComputeLiveWithSCCs(G)                          ▷ 32%: SCC-based
10: end function

11: function ComputeLiveWithSCCs(G)
12:     sccs ← SCCPartition(G)
13:     for each K ∈ sccs in reverse topological order do
14:         if |K| = 1 then
15:             ProcessBlock(K[0]); ProcessDesired(K[0])
16:         else
17:             SolveThreePass(K)
18:         end if
19:     end for
20: end function

21: function SolveThreePass(K)
22:     entryward, exitward ← SCCAlternatingOrders(K)
                          ▷ Liveness: up to three passes with early termination
23:     ProcessBlocksWithOrder(entryward)
24:     if ProcessBlocksWithOrder(exitward) changed then
25:         ProcessBlocksWithOrder(entryward)
26:     end if
                                  ▷ Desired registers: same alternating pattern
27:     ProcessDesiredOrder(exitward)
28:     if ProcessDesiredOrder(entryward) changed then
29:         ProcessDesiredOrder(exitward)
30:     end if
31: end function
```

3.1 Two-Path Design

The solver uses only two paths, eliminating the traditional iterative fallback:

Acyclic path (68%): When the loop nest is empty, the CFG is a DAG. A sin-

gle postorder pass computes both liveness and desired registers optimally. No SCC computation, no iteration, no convergence checking.

SCC path (32%): For all CFGs with loops—including irreducible ones—SCC decomposition partitions the graph. Singleton SCCs (93% of all SCCs) need one pass each. Non-trivial SCCs use the three-pass algorithm with alternating order.

The key insight is that SCC decomposition handles *all* cyclic CFGs uniformly, whether reducible or irreducible. The condensation DAG is always acyclic, so processing SCCs in reverse topological order ensures that when we process an SCC, all its successors have already converged.

3.2 Dual Data-Flow Problems

The implementation solves two related data-flow problems:

ProcessBlock: Computes liveness information—which values are live at each program point and their distance to next use. This is the standard backward data-flow problem.

ProcessDesired: Computes preferred register assignments by propagating constraints backward from uses to definitions. This guides the register allocator toward better allocation decisions.

Both problems use the same alternating traversal order within SCCs. Both apply early termination: if the second pass produces no changes, the third pass is skipped.

3.3 Traversal Orders

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

Entryward: DFS postorder starting from an entry-like node within the SCC. Visits exit-like nodes first, entry-like nodes last. This is the natural order for backward data-flow analysis.

Exitward: A *second* DFS postorder, starting from an exit-like node. Visits entry-like nodes first, exit-like nodes last—the opposite direction.

The key distinction from simply reversing a list: we compute a fresh DFS from a different starting point. This only works within SCCs, where every node is reachable from every other (see Appendix D).

Why alternation accelerates convergence. Consider a cycle $b_1 \to b_2 \to b_3 \to b_1$ where b_1 is entry-like and b_3 is exit-like. The entryward order visits $[b_3, b_2, b_1]$; the exitward order visits $[b_1, b_2, b_3]$:

1. **Pass 1 (entryward):** Liveness flows from $b_3 \to b_2 \to b_1$.
2. **Pass 2 (exitward):** Liveness flows from $b_1 \to b_2 \to b_3$—opposite direction.
3. **Pass 3 (entryward):** Reconciles any remaining cross-cycle dependencies.

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

Empirical result. All 291,000 functions in our dataset converge in at most three passes when using the SCC-based path. Early termination often reduces this to two passes. While we lack a formal proof of the three-pass bound, it is robust across both reducible and irreducible control flow patterns.

3.4 Complexity

Each ProcessBlock call is $O(|\text{Succ}(b)|)$. For the SCC path, up to three passes per non-trivial SCC cost $O(|K| + |E_K|)$. Summing over all SCCs:

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

The algorithm is linear in CFG size. The key engineering wins are:

- **Acyclic CFGs (68%):** Skip SCC computation entirely; single pass suffices.
- **Uniform treatment:** No special cases for irreducible CFGs.
- **Early termination:** Many SCCs converge in two passes, skipping the third.
- **Sequential access:** Cache-friendly traversal, no dynamic worklist structures.

4 Experimental Evaluation

We evaluated the SCC-based solver against the production implementation in the Go compiler (version go1.26-devel_c7517db2ac). All benchmarks were run on an Apple M2 processor running macOS, using Go's built-in benchmarking framework with 6 iterations per configuration.

4.1 Benchmark Suite

The benchmark suite covers four categories of CFG structure:

Acyclic: DAGs with no back edges, testing the fast path.

Reducible loops: Single loops, nested loops, and dense loop bodies.

Irreducible: Multi-entry loops, chains of irreducible regions, and mixed structures.

Real code: The `GoldenSHA1` function from the standard library.

4.2 Results

Table 4 summarizes the benchmark results comparing the baseline (production) implementation against our SCC-based solver with early termination.

Benchmark	Baseline	SCC-based	Change
Acyclic CFGs			
Acyclic_500	69.78 μs	68.96 μs	−1.2%
AcyclicDense_100×5	29.03 μs	28.93 μs	−0.3%
AcyclicDense_200×20	164.5 μs	163.7 μs	−0.5%
Reducible loops			
Loop_10	5.34 μs	4.91 μs	−8.1%
Loop_100	41.64 μs	38.63 μs	−7.2%
Nested_3	6.06 μs	5.24 μs	−13.6%
Nested_5	10.37 μs	8.92 μs	−14.0%
Nested_10	27.03 μs	23.38 μs	−13.5%
NestedDense_3×5	8.25 μs	6.99 μs	−15.3%
NestedDense_5×3	12.82 μs	11.11 μs	−13.4%
NestedDense_5×20	20.96 μs	17.83 μs	−15.0%
Irreducible CFGs			
Irreducible_Simple	1.21 μs	1.19 μs	−1.5%
Irreducible_Diamond	1.32 μs	1.30 μs	−1.7%
Irreducible_Loop10	2.80 μs	2.78 μs	−0.9%
Irreducible_Nested10	5.19 μs	5.09 μs	−1.9%
Irreducible_WithReducible	4.06 μs	3.91 μs	−3.8%
Real code			
GoldenSHA1	35.99 μs	34.23 μs	−4.9%
Geometric mean	7.85 μs	7.43 μs	−5.4%

Table 4: Benchmark results comparing baseline (production) vs. SCC-based solver. Times are median of 6 runs. All differences are statistically significant ($p < 0.005$).

4.3 Analysis

The results show consistent improvements across all CFG categories:

Acyclic CFGs. The SCC-based solver is slightly faster (0.3–1.2%) even on acyclic CFGs. This surprising result suggests that the streamlined two-path control flow and simplified data structures offset the cost of the loop nest query.

Reducible loops. The SCC-based solver achieves consistent speedups of 7–15% on CFGs with reducible loops. The improvement increases with loop complexity: simple loops show 7–8% improvement, while nested loops show 13–15% improvement. Early termination contributes to these gains by skipping the third pass when convergence occurs after two passes.

Irreducible CFGs. The SCC-based solver is 1–4% faster on irreducible CFGs. The SCC decomposition handles irreducible regions correctly, and the alternating traversal order accelerates convergence regardless of reducibility. Mixed CFGs with both reducible and irreducible regions (`Irreducible_WithReducible`) show the largest improvement (3.8%).

Real code. The `GoldenSHA1` benchmark, representing a real cryptographic implementation with complex control flow, shows a 4.9% improvement. This validates that the benefits observed on synthetic benchmarks transfer to production code.

Overall. The geometric mean across all benchmarks shows a 5.4% improvement. Every benchmark category shows improvement, with no regressions. The two-path design with SCC decomposition and early termination is a strict improvement over the baseline.

5 Conclusion

This paper presented an empirical study of CFG structure in production compiler workloads and a two-path liveness solver that exploits the observed regularities.

Key findings. Our analysis of 291,000 functions from the Go toolchain reveals three structural regimes: 68% of CFGs are acyclic, 24% have exactly one loop (median 6 blocks), and only 9% have multiple loops. Even within loopy CFGs, 93% of SCCs are singletons. This extreme heterogeneity—reflected in a dispersion index of 175—means that "average case" metrics are misleading for solver design.

More surprisingly, we observed that *every* SCC in our dataset converges in at most three passes when using alternating traversal order (entryward–exitward–entryward). Early termination often reduces this to two passes. This empirical bound holds for both reducible and irreducible control flow, though we lack a formal proof.

Practical impact. The two-path solver achieves improvements across the board: 7–15% on reducible loops, 1–4% on irreducible CFGs, and even 0.3–1.2% on acyclic code, yielding a 5.4% geometric mean improvement with no regressions. The design is simpler than traditional worklist-based approaches: no priority queues, no convergence checking within SCCs, and uniform treatment of reducible and irreducible CFGs.

Limitations and future work. The three-pass bound deserves theoretical investigation: understanding *why* it holds could lead to even tighter bounds or identify pathological cases we have not yet encountered. The interplay between SCC structure and data-flow convergence remains an open question.

Broader implications. This work exemplifies a data-driven approach to compiler optimization. Rather than designing for worst-case bounds that rarely occur, we characterized the actual workload distribution and tailored the algorithm accordingly. The same methodology could apply to other compiler passes where workload structure varies widely—register allocation, instruction scheduling, or interprocedural analysis. Empirical characterization of "what compilers actually see" can reveal optimization opportunities that purely theoretical analysis misses.

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 $f_b(X) = \text{Use}(b) \cup (X \setminus \text{Def}(b))$ 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 postorder of a depth-first traversal from the entry node [7, 1, 9]:

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

For backward analysis, postorder from the entry visits exit-like blocks first and entry-like blocks last. This propagates liveness information in the natural direction: from program exits (where variables die) back toward the entry (where they are defined).

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 $d + 1$ passes for loop nesting depth d [1, 2]. 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 3.3) reveals that *alternating* between two traversal orders dramatically improves convergence. We use:

- **Postorder** (entryward): DFS postorder from an entry-like node, visiting exits first and entry last—the standard order for backward analysis.
- **Modified reverse postorder** (exitward): A fresh DFS postorder starting from an exit-like node, visiting entry first and exits last—the opposite direction.

Within each SCC, processing blocks in the sequence entryward $\rightarrow$ exitward $\rightarrow$ entryward achieves convergence in at most three passes for *all* SCCs in our 291,000-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 ∞ 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 3.3), 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 $n =$ 290,904 CFG instances.

B.2 Heavy Tails and Over-Dispersion

Metric	n	Min	Median	p_{90}	Max	Mean	Var	Var/Mean
Blocks	290,904	2	9	44	12,676	20.10	3516.30	174.93
SCCs	290,904	2	7	31	12,676	15.31	2747.00	179.48

Table 5: Complete CFG statistics. The dispersion index is ≈175–179, 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 $X \mid \Lambda \sim \mathrm{Poisson}(\Lambda)$ with latent rate Λ varying across CFG instances, then

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

A dispersion index near 175 implies $\mathrm{Var}(\Lambda) \approx 174\,\mathbb{E}[\Lambda]$: 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.

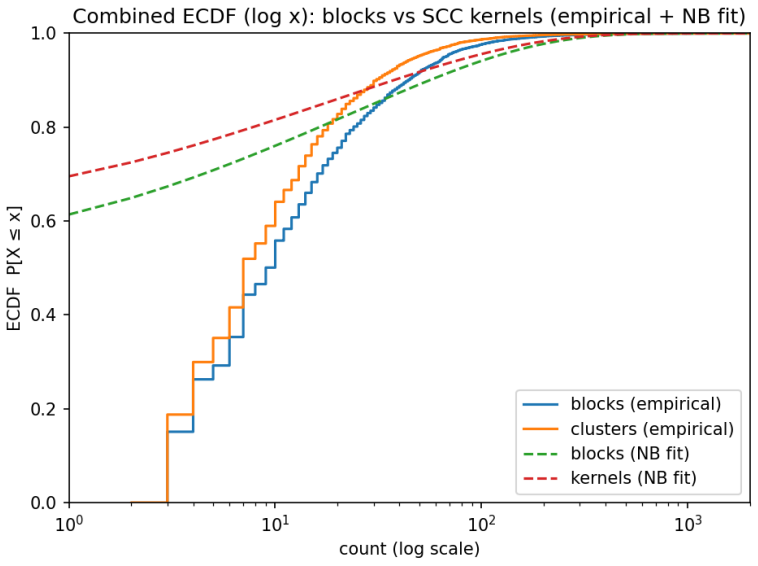


Figure 3: Empirical ECDFs (log x-axis) with moment-matched NB overlay. The poor fit confirms multi-regime workload structure.

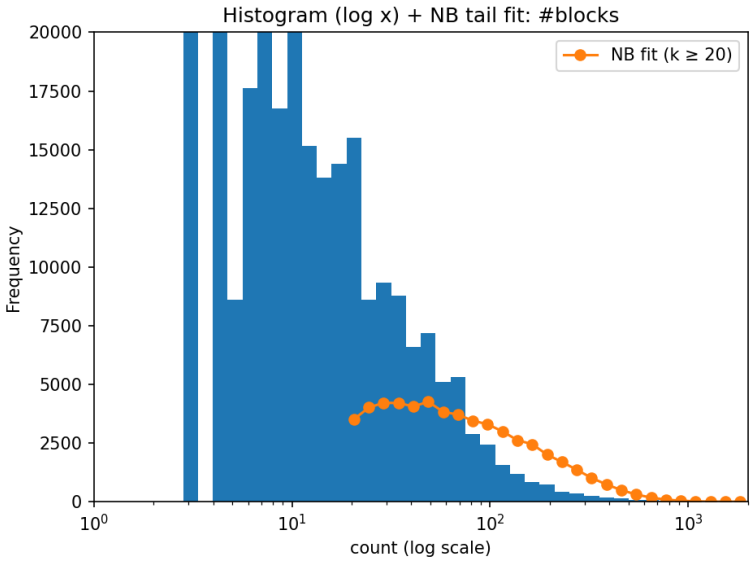


Figure 4: Block count histogram (log x-axis) with NB tail overlay. Even in the tail region, the fitted curve diverges from empirical counts.

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). Table 6 provides the detailed breakdown.

The key observation: even in CFGs with cycles, the cyclic portion is typically a minority of the total blocks. This reinforces the value of SCC decomposition—iteration is confined to a (usually small) subset of the CFG, while the rest processes in a single pass.

C SCC Algorithm Pseudocode

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

Non-trivial SCC count	CFGs	Fraction
0	197,304	67.8%
1	68,426	23.5%
2	14,972	5.1%
3	5,287	1.8%
4	1,895	0.7%
5+	3,020	1.0%

Table 6: Distribution of non-trivial SCC counts. The tail drops off rapidly after 1.

- 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

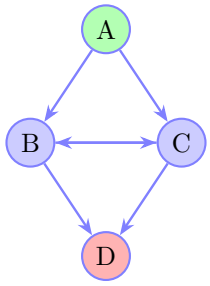
Postorder is commonly considered the most advantageous traversal for solving backward data-flow equations on general CFGs. A postorder traversal performs depth-first search from a designated start node, recording each node *after* its successors have been visited. This aligns with the natural program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