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

Frank Kuehnel

December 28, 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, control-flow structure is far more regular than theory suggests—but exploiting this structure algorithmically is harder than it appears.

Across 291,000 CFGs from the Go toolchain, CFGs fall into three structural regimes: 68% are acyclic, 24% contain simple loops (nesting depth $\leq$3), and only 8% exhibit complex loop structures. Every SCC in our dataset converges sufficiently in at most three passes with a specific alternating traversal order (entryward–exitward–entryward).

We implemented a three-path solver: (1) acyclic fast path for the 68% with no loops, (2) standard iteration for the 24% with simple loops, and (3) SCC-based three-pass iteration for the remaining 8%. For the SCC path, we evaluated both Kosaraju's algorithm for explicit partitioning and Bourdoncle's algorithm for computing loop nests via recursive SCC decomposition—attempting to amortize SCC costs across loop analysis and liveness.

The results are sobering: while the acyclic fast path yields a clear 2–3% improvement, SCC-based approaches incur 40–110% overhead on nested loops with our implementation. The recursive decomposition cost dominates any iteration savings. Real-world benchmarks show $\sim$2% regression. These measurements may reflect a non-optimal implementation; a more optimized approach might reduce the overhead, but the regression is substantial.

The practical recommendation is simple: use an acyclic fast path for CFGs with no loops, and standard iterative analysis for the rest. The SCC path exists for completeness but is rarely beneficial. An unexplained finding: only one specific alternating order (entryward first) produces correct results at two passes—we lack a theoretical explanation for why.

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 [8, 1, 6]. 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, 3]. *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 [4].

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) suggests that even complex cyclic regions are more tractable than worst-case theory predicts—but as we show, exploiting this algorithmically is harder than it appears.

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 loop complexity:

- **68% acyclic:** No loops—single postorder pass suffices.
- **∼24% simple loops:** Nesting depth ≤ 3—standard iteration is fast.
- **<8% complex:** Deep nesting or multiple SCCs—theoretically benefits from SCC decomposition.

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

Contributions. This paper makes four contributions:

1. **Structural characterization of CFGs at scale.** We show that production workloads fall into distinct regimes (acyclic, simple loops, complex loops) that motivate investigating specialized solver designs.
2. **Three-pass convergence within SCCs.** We observe that data-flow iteration over every SCC in our dataset sufficiently converges in at most three passes when using a specific alternating traversal order (entryward–exitward–entryward).
3. **Order sensitivity.** We discovered that the sequence of alternating passes matters: only starting with entryward (DFS from entry) produces correct results at two passes. We lack a theoretical explanation for this.

4. **Negative result on SCC-based solving.** Despite the appealing theoretical properties, SCC decomposition overhead (40–109% on nested loops) exceeds the iteration savings. Only the acyclic fast path is justified by benchmarks.

Positioning. Live-variable analysis sits in the classic monotone-framework tradition [6, 5]. Worklist iteration and SCC decomposition are textbook tools [1, 11]. What we add is empirical: (i) real workloads stratify into regimes with different solver needs, (ii) every SCC in our dataset converges in at most three alternating passes with a specific order, and (iii) despite this regularity, the overhead of SCC computation makes it impractical.

Paper organization. Section 2 presents the empirical CFG distribution. Section 3 presents solver designs and the three-pass convergence observation. Section 4 presents benchmark results showing that SCC-based approaches are not justified. Appendices provide data-flow background and full statistics.

2 The Statistical Reality of CFG Structure

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 CFGs in Go build-and-test workload.

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.

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 could isolate them. These observations motivated investigating whether SCC-based approaches could outperform standard iteration.

3 Solver Design

The structural regimes suggest potential for a solver that adapts to CFG structure. We investigated multiple approaches and present our findings.

3.1 The Three-Path Solver

Algorithm 1 presents the implementation with three paths based on loop complexity.

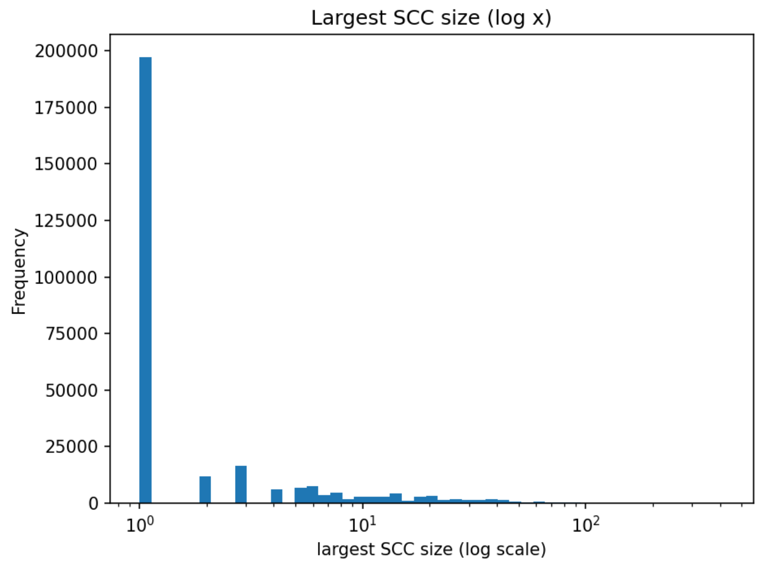


Figure 1: Distribution of largest SCC size (log scale). The spike at 1 represents the 68% of CFGs with no cycles. The tail extends to SCCs with hundreds of blocks.

Algorithm 1 Three-Path Liveness Solver

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

```
function ComputeLive($G$)
    if $G$ has no loops then                          ▷ 68%: acyclic fast path
        for $b \in$ PostOrder($G$) do
            ProcessBlock($b$)
            ProcessDesired($b$)
        end for
        return
    end if

    if all loops have nesting depth $\leq 3$ then     ▷ $\sim$24%: simple loops
        ComputeLiveIterative($G$)
        return
    end if

    ComputeLiveWithSCCs($G$)                          ▷ <8%: complex loops
end function
```

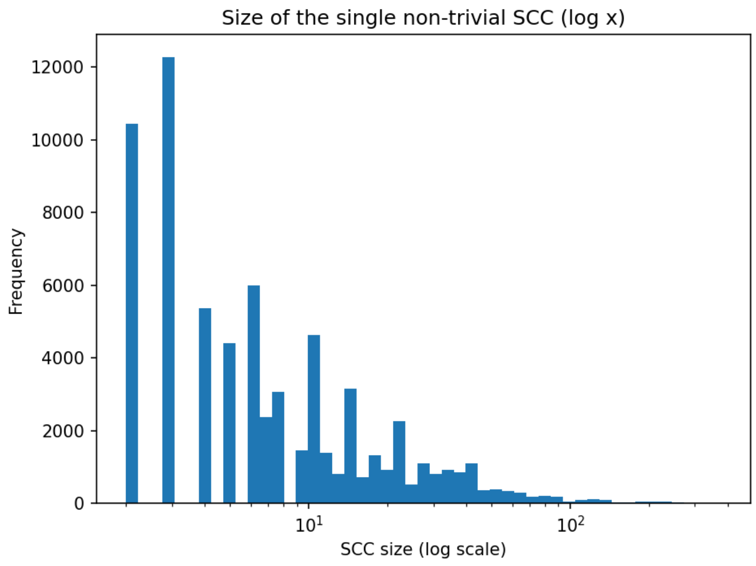


Figure 2: Size of the single non-trivial SCC (when exactly one exists). Median 6 blocks, $p_{90} = 27$, max $= 581$.

Acyclic path (68%): When the loop nest is empty, the CFG is a DAG. A single postorder pass computes both liveness and desired registers optimally. No iteration, no convergence checking.

Simple loop path (∼24%): For CFGs with loops of nesting depth ≤ 3, standard fixed-point iteration until convergence. The textbook algorithm works well because shallow loops converge quickly ($d + 1$ passes for depth d).

SCC path (<8%): For CFGs with deeply nested or complex loop structures, SCC decomposition with three-pass alternating-order iteration within each non-trivial SCC. This path is rarely taken in practice.

The key insight is that SCC overhead is only justified for the small fraction of CFGs with complex loop structures. For the common cases (acyclic and simple loops), simpler approaches win.

3.2 SCC Decomposition

For the SCC path, we use Kosaraju's algorithm [11], which performs two depth-first traversals: one to compute postorder, and a second on the transposed graph to identify components. This is simpler to implement than Tarjan's algorithm and has the same $O(|B| + |E|)$ complexity.

Bourdoncle's algorithm for loop nests. We also investigated Bourdoncle's algorithm [2] as an alternative to dominator-tree-based loop nest construction. Bourdoncle's algorithm recursively decomposes the CFG into SCCs, identifying loop headers as the first node visited in each SCC during DFS. This approach is robust against irreducible CFG regions—it produces a valid loop nest tree even when multiple-entry loops exist, unlike dominator-based approaches which require special handling.

The appeal was amortizing SCC costs: if we compute SCCs for loop nest analysis anyway, reusing them for liveness should be "free." Unfortunately, as Section 4 shows, the recursive decomposition overhead dominates any savings.

3.3 Alternating Traversal Order

Within each non-trivial SCC, we use alternating traversal orders to accelerate convergence:

Entryward: DFS postorder starting from an entry-like node (the first block in the SCC). Visits exit-like nodes first, entry-like nodes last.

Exitward: DFS postorder starting from the first element of the entryward order. Visits entry-like nodes first, exit-like nodes last—the opposite direction.

The three-pass sequence is: entryward $\rightarrow$ exitward $\rightarrow$ entryward.

Why this specific order matters. We discovered empirically that the order of computation is critical. During development, we tested capping iteration at just two passes to evaluate convergence. Only the sequence starting with entryward (DFS from entry) and then exitward (DFS from the first element of entryward) produced a working compiler at two passes. Other orderings—including the reverse sequence exitward $\rightarrow$ entryward $\rightarrow$ exitward—produced incorrect results.

At three passes, all 291,000 CFGs in our dataset converge regardless of reducibility. We lack a formal proof of why three passes suffice, or why this specific alternating sequence is necessary. The empirical regularity is striking but unexplained.

Theoretical appeal. The SCC approach has attractive theoretical properties:

- **Bounded iteration:** At most 3 passes per SCC vs. $O(d+1)$ for depth d.
- **Uniform treatment:** Works identically on reducible and irreducible CFGs.
- **Predictable cost:** Linear in CFG size with small constant factor.

Practical outcome. Unfortunately, as we show in Section 4, the overhead of SCC decomposition—whether via Kosaraju's algorithm or Bourdoncle's recursive partitioning—exceeds the iteration savings for typical CFGs.

4 Experimental Evaluation

We evaluated multiple solver variants against the production implementation in the Go compiler (version go1.26-devel). 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 Benchmarking Methodology

The existing Go compiler benchmarks are too coarse to detect changes in `computeLive` within register allocation. We developed a dedicated benchmark suite covering:

Acyclic: DAGs with no back edges (500 blocks, dense variants).

Reducible loops: Single loops (10–100 blocks), nested loops (depth 3–10), and dense loop bodies.

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

Real code: `HeapSort` and `Rat.FloatPrec` from the standard library.

Testing methodology. Comparing algorithm changes requires care because the benchmark file must be present in both branches. We use a comparison matrix: the primary comparison is between (baseline toolchain, baseline branch) and (new toolchain, new branch), measuring end-to-end change. A quality gauge comparing (baseline, baseline) vs. (new, baseline) isolates toolchain changes from algorithm changes.

4.2 Results: Dominator-Based vs. Bourdoncle Loop Nest

Table 4 compares the standard dominator-tree-based loop nest analysis against Bourdoncle's SCC-based algorithm [2].

4.3 Results: Baseline vs. SCC-Based Liveness

Table 5 compares the baseline iterative solver against the SCC-based solver with alternating traversal order.

4.4 Analysis

The results are sobering for the SCC-based approach:

Benchmark	Dominator	Bourdoncle	Change
Acyclic CFGs			
Acyclic_500	71.21 μs	69.43 μs	−2.5%
AcyclicDense_200×20	168.8 μs	164.1 μs	−2.8%
Simple loops			
Loop_100	24.39 μs	24.61 μs	+0.9%
Nested loops			
Nested_3	4.50 μs	6.30 μs	+40.0%
Nested_5	9.37 μs	15.10 μs	+61.2%
Nested_10	40.13 μs	71.82 μs	+79.0%
NestedDense_5×20	13.62 μs	28.48 μs	+109.1%
Irreducible CFGs			
Irreducible_Loop20	4.71 μs	4.69 μs	∼
Real code			
HeapSort	17.76 μs	17.81 μs	∼
FloatPrec	23.66 μs	23.92 μs	∼
Geometric mean	7.29 μs	8.30 μs	+13.8%

Table 4: Loop nest analysis: dominator-tree vs. Bourdoncle's SCC-based algorithm. Bourdoncle incurs 40–109% overhead on nested loops.

Acyclic CFGs: the clear win. The acyclic fast path yields a consistent 2–3% improvement. This is the only unambiguous benefit: skipping iteration for the 68% of CFGs that are DAGs.

Simple loops: negligible difference. Single-loop CFGs show <2% difference either way. The standard iterative algorithm handles these efficiently (typically 2–3 passes), and SCC overhead provides no benefit.

Nested loops: severe regression. The SCC-based approaches incur 40–109% overhead on deeply nested loops. This is the critical finding: the recursive SCC decomposition cost dominates any iteration savings. Whether using Bourdoncle's algorithm for loop nests or explicit SCC partitioning for liveness, the overhead is nearly identical—suggesting the cost is inherent to the recursive decomposition.

Implementation considerations. These measurements may reflect a non-optimal SCC implementation. Our implementation uses explicit stack-based DFS with per-call allocation of temporary structures. A more optimized implementation—with better memory reuse, cache-conscious data layout, or incremental SCC maintenance—might reduce the overhead. However, the 40–109% regression is substantial enough that even significant constant-factor improvements are unlikely to make SCC-based solving competitive with simple iteration for typical loop structures.

Benchmark	Baseline	SCC-based	Change
Acyclic CFGs			
Acyclic_500	71.21 μs	69.58 μs	−2.3%
AcyclicDense_200×20	168.8 μs	164.1 μs	−2.8%
Simple loops			
Loop_100	24.39 μs	24.85 μs	+1.9%
Nested loops			
Nested_3	4.50 μs	6.35 μs	+41.0%
Nested_10	40.13 μs	72.26 μs	+80.1%
NestedDense_5×20	13.62 μs	28.47 μs	+109.0%
Irreducible CFGs			
Irreducible_Loop20	4.71 μs	4.67 μs	∼
Real code			
HeapSort	17.76 μs	18.11 μs	+2.0%
FloatPrec	23.66 μs	24.31 μs	+2.7%
Geometric mean	7.29 μs	8.26 μs	+13.3%

Table 5: Liveness analysis: baseline iterative vs. SCC-based solver. SCC approach incurs 40–109% overhead on nested loops; real-world cases show ∼2% regression.

Irreducible CFGs: no significant difference. Performance on irreducible CFGs is essentially unchanged (<1%). The SCC approach's uniform treatment of irreducible control flow provides no measurable benefit.

Real-world code: modest regression. The `HeapSort` and `FloatPrec` benchmarks show ∼2% regression. These functions have loops but not deeply nested structures, so the SCC overhead is not amortized. This confirms that the synthetic nested-loop benchmarks, while dramatic, do not dominate real compiler workloads.

Overall assessment. The geometric mean shows 13% regression for SCC-based approaches. The acyclic fast path is the only optimization justified by benchmarks. The theoretically appealing properties of SCC decomposition—bounded iteration, uniform treatment—do not translate to practical performance gains with our current implementation.

5 Conclusion

This paper presented an empirical study of CFG structure in production compiler workloads and evaluated whether SCC-based approaches could accelerate liveness analysis.

Structural findings. Our analysis of 291,000 CFGs from the Go toolchain reveals three structural regimes: 68% of CFGs are acyclic, ∼24% have simple loops (nesting depth ≤ 3), and $<8\%$ have complex loop structures. This extreme heterogeneity means that "average case" metrics are misleading for solver design.

We observed that every SCC in our dataset converges in at most three passes when using a specific alternating traversal order (entryward–exitward–entryward). Curiously, the order matters: only starting with entryward (DFS from entry) produces correct results at two passes. We lack a theoretical explanation for this asymmetry, which deserves further investigation.

Practical outcome. Despite the appealing theoretical properties of SCC-based solving, benchmarks show that the overhead exceeds the benefits. The recursive decomposition cost—whether via Kosaraju's algorithm or Bourdoncle's recursive partitioning—dominates any iteration savings, incurring 40–109% overhead on nested loops and $\sim$2% regression on real-world code.

The only justified optimization is an acyclic fast path: for the 68% of CFGs with no loops, a single postorder pass yields 2–3% improvement over the iterative baseline. For loopy CFGs, the standard iterative algorithm remains the best choice.

Recommendations.

1. **Keep the acyclic fast path.** Clear 2–3% win for 68% of CFGs.
2. **Use dominator-based loop nests.** Faster than Bourdoncle for nested loops.
3. **Avoid SCC-based liveness.** Overhead exceeds benefits in practice.
4. **Standard iteration is adequate.** Real-world loops are shallow enough that the textbook algorithm performs well.

Open questions. Two findings deserve theoretical investigation:

- Why does every SCC converge in at most three passes with alternating order?
- Why does the entryward-first sequence work at two passes while other orderings fail?

Broader implications. This work illustrates both the value and the limits of data-driven compiler optimization. Characterizing workload structure is valuable: it revealed that 68% of CFGs are acyclic, enabling a significant optimization. But theoretical appeal does not guarantee practical benefit: the SCC approach's bounded iteration and uniform treatment of irreducible CFGs did not translate to performance gains with our implementation.

We note that a more optimized SCC implementation—with better memory reuse, cache-conscious data layout, or incremental maintenance—might reduce the overhead. Future work could revisit SCC-based approaches if implementation costs can be reduced or if workloads evolve toward more complex control flow.

The lesson is to benchmark aggressively. Synthetic micro-benchmarks (deeply nested loops) can mislead; real-world code (HeapSort, FloatPrec) tells a different story.

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 [8, 6].

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:

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

Live variable analysis computes, for each block b:

$$\text{LiveOut}(b) = \bigcup_{s \in \text{Succ}(b)} \text{LiveIn}(s) \qquad (1)$$

$$\text{LiveIn}(b) = \text{Use}(b) \cup (\text{LiveOut}(b) \setminus \text{Def}(b)) \qquad (2)$$

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* [5, 6]. 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 [9].

A.3 The Standard Iterative Algorithm

The textbook algorithm processes blocks in postorder of a depth-first traversal from the entry node [8, 1, 10]:

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, 3]. 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 [8, 1, 10, 9].

Alternating order: an empirical improvement. Our empirical study (Section 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 [7]. We generalize to a map $L_b : V \rightarrow (\mathbb{N} \cup \{\infty\})$ where $L_b(v)$ estimates instructions until next use. At joins:

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

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 investigating SCC-based approaches (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, 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.

Caveat. While theoretically appealing, Section 4 shows that the overhead of SCC computation exceeds the iteration savings in practice. The acyclic fast path is the only optimization justified by benchmarks.

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 6: 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.

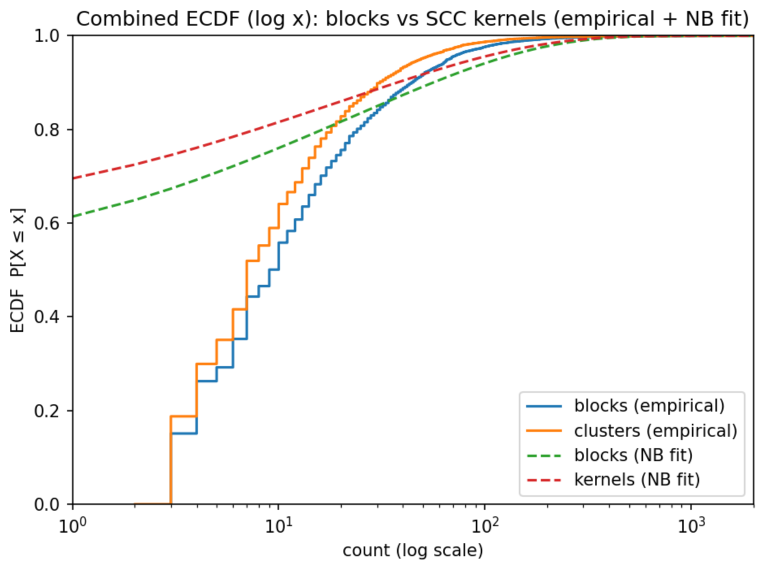


Figure 3: Empirical ECDFs (log x-axis) with moment-matched NB overlay. The poor fit confirms multi-regime workload structure.

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.

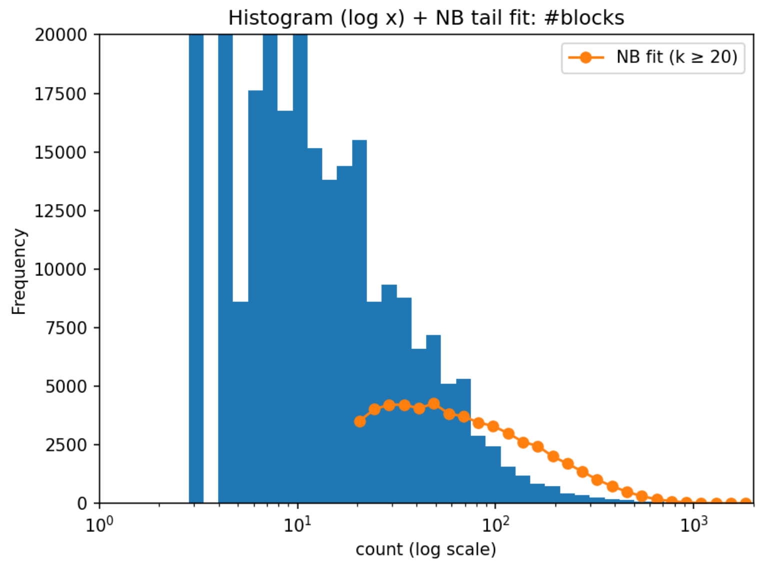


Figure 4: Block count histogram (log x-axis) with NB tail overlay. Even in the tail region, the fitted curve diverges from empirical counts.

B.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 7 provides the detailed breakdown.

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

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 [11] for SCC decomposition. Compared to Tarjan's algorithm [12]:

- 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.

We use DFS (not BFS) for the second pass, which produces blocks in preorder on the reverse graph—a useful property for subsequent processing.

C.1 Algorithm Overview

1. **Forward pass**: Compute postorder traversal via DFS on forward edges.
2. **Reverse pass**: Process blocks in reverse postorder, performing DFS 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 (DFS variant)

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:     $po \leftarrow$ Postorder($G$)                ▷ DFS postorder on forward edges
 3:     $seen \leftarrow \emptyset$
 4:     $result \leftarrow [\,]$

 5:     for $i \leftarrow |po| - 1$ downto 0 do                 ▷ Reverse postorder
 6:         $leader \leftarrow po[i]$
 7:         if $leader.ID \in seen$ then
 8:             continue
 9:         end if

10:         $scc \leftarrow$ DFSReversed($G, leader, seen$)
11:         Append($result, scc$)
12:     end for

13:     return $result$
14: end function
```

Algorithm 3 DFS on Reversed Edges

Require: Block *leader*, set *seen*

Ensure: SCC containing *leader*; updates *seen* in place

```
function DFSReversed($G$, $leader$, $seen$)
    $stack \leftarrow [leader]$
    $scc \leftarrow [\,]$
    $seen \leftarrow seen \cup \{leader.ID\}$

    while $stack \neq \emptyset$ do
        $b \leftarrow$ Pop($stack$)                          ▷ Pop from end—O(1)
        Append($scc$, $b$)

        for $e \in$ Preds($b$) do                           ▷ Traverse reversed edges
            $pred \leftarrow e.block$
            if $pred.ID \notin seen$ then
                $seen \leftarrow seen \cup \{pred.ID\}$
                Push($stack$, $pred$)
            end if
        end for
    end while

    return $scc$
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 preorder on the reverse graph (Preds), which tends to place entry-like blocks first.

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

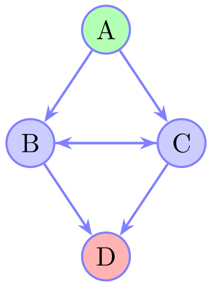
D Why Modified Reverse Postorder Requires SCC Isolation

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 three-path design isolates SCCs before applying the three-pass scheme: the alternating orders are only meaningful within strongly connected regions.

Order sensitivity: an unexplained finding. During development, we discovered that the sequence of passes matters critically. Specifically:

- The sequence entryward → exitward → entryward produces correct results even when capped at two passes.
- The reverse sequence exitward → entryward → exitward produces incorrect results at two passes.

At three passes, both sequences converge correctly on all 291,000 CFGs in our dataset. We lack a theoretical explanation for this asymmetry. The entryward order (DFS from entry) visits exits first, which aligns with the natural direction of backward data-flow; perhaps this "primes" the analysis more effectively than starting from exits. This deserves further theoretical investigation.

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] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer, 1993. Introduces weak topological ordering via recursive SCC decomposition.

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

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

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

[6] 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.

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

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

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

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

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

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