Tiered Live Variable Analysis for Heavy-Tailed CFG Distributions

Frank Kuehnel

December 20, 2025

Abstract

Live range 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 are far more regular than that view suggests.

Across 290,000 functions from the Go toolchain, CFGs fall into three regimes: 68% are acyclic, 24% contain exactly one loop kernel (median 6 blocks), and only 8% exhibit multiple cyclic regions; overall sizes are heavy-tailed (dispersion ≈175). More strikingly, *every* strongly connected component in this dataset reaches a liveness fixed point in exactly three traversals when we alternate postorder and reverse postorder (PO–RPO–PO).

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

1 Introduction

Live-variable (live-range) analysis is the ubiquitous backward data-flow pass that drives interference construction, spilling, and many SSA-based optimizations. In principle it is a monotone fixed-point problem; in practice, its cost is dominated by how often information must circulate around cycles in the control-flow graph (CFG).

The literature and textbooks largely reason about arbitrary CFG instances and worst-case iteration [8, 1, 4]. Modern compilers, however, run the same solver millions of times on a *workload distribution*. When that distribution has distinct structural regimes, a single "one-size-fits-all" solver leaves performance on the table.

This paper makes two empirical claims from a 290,000-function study of the Go toolchain: (i) CFGs are heavy-tailed and stratify into three regimes (68% acyclic, 24% single-loop, 8% multi-cycle), and (ii) within each strongly

connected component (SCC), liveness converges in exactly three alternating traversals (postorder, reverse postorder, postorder). Together, these facts justify a *tiered* architecture that optimizes the common case while remaining robust on the tail.

From examples to distributions. Figure 1 shows a textbook-sized CFG fragment. Such examples are invaluable for correctness arguments, but they hide the question that matters for compiler engineering: *what kinds of CFGs occur in bulk, and what solver structure matches that distribution?*

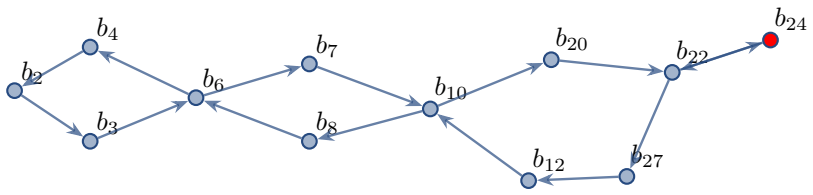

Figure 1: A typical CFG fragment used in algorithm exposition. Single-instance analysis reveals local structure but says nothing about the *distribution* of such structures a compiler must handle.

But this focus on individual instances is overrated for engineering purposes. A production compiler does not process one CFG—it processes hundreds of thousands, drawn from a distribution with complex structure. The question is not "how does the algorithm behave on *this* graph?" but "how does it behave across the full workload?" Recent data-driven work has begun to address this gap, using profiling and traces to guide allocation decisions in ways that outperform purely static analyses [16, 7, 12]. We contribute to this empirical turn by characterizing the *structural distribution* of CFGs in a production workload.

The core observation. We analyzed 290,334 CFG instances from a full build-and-test of the Go toolchain. The size distribution is striking:

- **Median:** 9 blocks, 7 SCCs
- **90th percentile:** 44 blocks, 31 SCCs
- **Maximum:** 12,676 blocks
- **Dispersion index:** $\approx$175 (vs. 1 for Poisson)

But the *structural* distribution is even more revealing. CFGs fall into three regimes based on their cyclic complexity:

- **68% acyclic:** All SCCs are singletons—no back edges, no iteration needed.
- **24% single-loop:** Exactly one non-trivial SCC (median size 6 blocks)—one localized region requiring iteration.

- **8% complex:** Multiple non-trivial SCCs—genuinely complex control flow.

The variance is 175× the mean—there is no "characteristic scale" for CFG complexity. Simple parametric models (Poisson, negative binomial) fail to capture this structure.

Design implications. This distribution fundamentally shapes how we should build live variable solvers:

1. **Average-case metrics mislead.** A mean of 20 blocks hides that end-to-end time is dominated by tail functions, not the median.

2. **Simple cases deserve simple treatment.** The 50% of CFGs with ≤ 9 blocks need minimal overhead—elaborate data structures hurt more than they help.

3. **Tail cases are more tractable than theory suggests.** While textbook worst-case analysis implies $O(|B|)$ iterations, our empirical study reveals that *all* SCCs in our dataset converge in exactly three passes with an alternating traversal order.

4. **Stratification reveals structure.** The poor fit of global models is *good news*: it indicates exploitable heterogeneity across packages, compilation stages, and code patterns.

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 (68% acyclic, 24% single-loop, 8% complex) that demand tiered treatment.

2. **Three-pass convergence for SCCs.** We report the empirical finding 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 and, to our knowledge, has not been previously documented.

1.1 Positioning Our Claims in the Literature

Our results are not a new formulation of data-flow analysis; they are measurements that sharpen what parts of the standard story matter in production, and they suggest a simpler solver organization than worst-case reasoning alone would imply.

What is standard. Live-variable analysis sits in the classic monotone-framework tradition [4, 3, 9]. Worklist iteration, and SCC decomposition to localize cyclic dependence, are textbook tools for obtaining linear-time behavior on acyclic regions and convergence on cyclic ones [1, 13, 14]. Traversal order (postorder, reverse postorder) is also a familiar engineering knob, typically discussed as a heuristic for faster convergence rather than as a property that can be stated at the workload level.

What we add empirically. First, we characterize CFG cyclic structure at scale and show that real workloads are not well summarized by a single "typical" CFG: they mix regimes with very different solver needs. Second, and more surprisingly, we observe a workload-wide convergence regularity: every SCC in our dataset reaches a fixed point in exactly three alternating traversals. To our knowledge, this "PO–RPO–PO in three" phenomenon has not been previously documented for liveness in production compiler workloads.

Why this matters for design. Because most CFGs are acyclic and most cyclic regions are small, a solver can be structured as a dispatcher rather than a monolith: a one-pass acyclic fast path, an SCC-local solver for reducible cycles, and a rare fallback for irreducible control flow. This paper develops that tiered architecture and evaluates how well the three-pass SCC solver matches the observed distribution.

Our approach. We advocate a *tiered architecture* that naturally adapts to the distribution: SCC decomposition [13, 14] partitions the CFG into acyclic regions that need only one pass each, plus cyclic kernels. Within each kernel, we apply the three-pass algorithm with alternating traversal order. This combination is fast on typical small CFGs, efficient on cyclic regions, and avoids the complexity of general worklist solvers—achieving both the speed linear-scan methods promise [10] and predictable behavior on complex cases.

The remainder of this paper develops this approach. Section 2 summarizes the empirical CFG distribution and structural regimes. Section 3 translates these statistics into design principles. Section 4 presents the tiered solver, including the three-pass algorithm. Appendix A details the data-flow equations and their iterative solution, Appendix B provides full statistical analysis, and Appendix C gives algorithm pseudocode.

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

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.

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 2 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 3 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.

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.

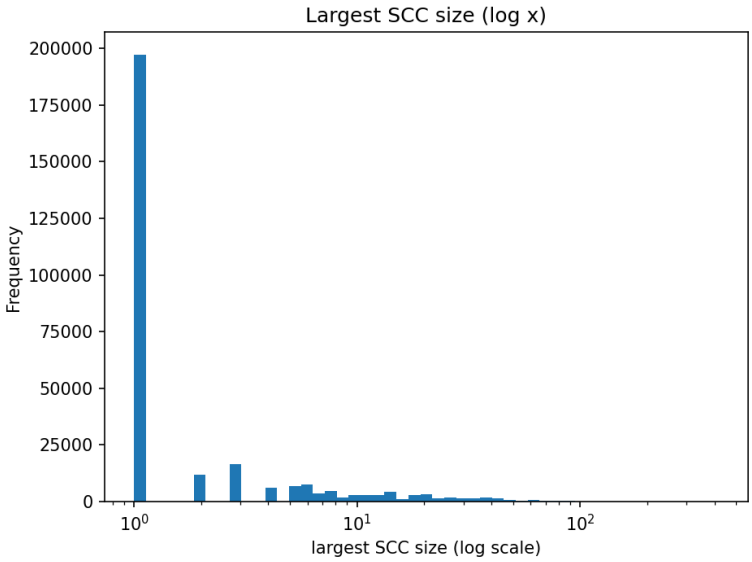

Figure 2: Distribution of largest SCC size (log scale). The spike at 1 represents the 68% of CFGs with no cycles. The tail extends to SCCs with hundreds of blocks.

- **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%.

3 Design Implications

The structural regime data directly informs solver architecture. Rather than designing for a generic "CFG," we design for three distinct populations:

Regime 1: Acyclic (68%). One backward pass suffices—no iteration, no SCC computation needed. Avoid allocating iteration-related structures for this majority case.

Regime 2: Single-loop (24%). One non-trivial SCC (median 6 blocks) amid otherwise acyclic code. Iteration is confined to a small region; the rest needs one pass.

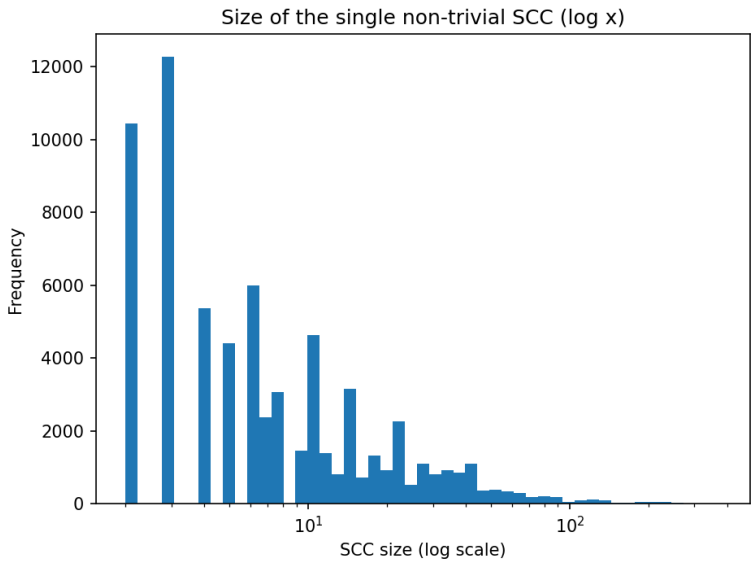

Figure 3: Size of the single non-trivial SCC (when exactly one exists). Median 6 blocks, $p_{90} = 27$, max = 376.

Regime 3: Multi-loop (8%). Multiple cyclic regions, including the largest functions. Each SCC is solved independently; complexity is additive, not multiplicative.

The key insight. SCC decomposition handles all three regimes *without explicit case analysis*. The algorithm naturally adapts: acyclic CFGs yield all-singleton SCCs (one pass each), single-loop CFGs yield one non-trivial SCC plus singletons, and multi-loop CFGs yield multiple independent SCCs. Section 4 presents the resulting algorithm.

4 A Tiered Solver Architecture

The design implications from Section 3 motivate a solver that adapts to CFG structure automatically. Algorithm 1 presents the complete approach: SCC decomposition 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, rpo ← AlternatingOrders(K)
17:    for b ∈ po do ProcessBlock(b)
18:    end for                                         ▷ Pass 1
19:    for b ∈ rpo do ProcessBlock(b)
20:    end for                                         ▷ Pass 2
21:    for b ∈ po do ProcessBlock(b)
22:    end for                                         ▷ Pass 3
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
```

4.1 Traversal Orders

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

Postorder (PO): Visit each node *after* all its DFS-tree successors. Result: exits appear first, entry last.

Reverse postorder (RPO): The reversed sequence—entry first, exits last. On a DAG, this is a topological order.

For backward data-flow, processing in RPO visits each node after its successors, propagating liveness "upward" optimally. But cycles break topological order: some successor is inevitably processed after its predecessor.

Why alternation accelerates convergence. Consider a cycle $b_1 \to b_2 \to b_3 \to b_1$ with $po = [b_3, b_2, b_1]$ and $rpo = [b_1, b_2, b_3]$:

1. **Pass 1 (PO):** Facts flow $b_3 \to b_2 \to b_1$.
2. **Pass 2 (RPO):** 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.

4.2 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 [8, 4].

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* [3, 4]. 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 [9].

A.3 The Standard Iterative Algorithm

The textbook algorithm processes blocks in reverse postorder (RPO) of a depth-first traversal [8, 1, 11]:

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 [8, 1, 11, 9].

Alternating order: an empirical improvement. Our empirical study (Section 4.1) 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 ∞ 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 4. 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 4.1), 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 Λ 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:

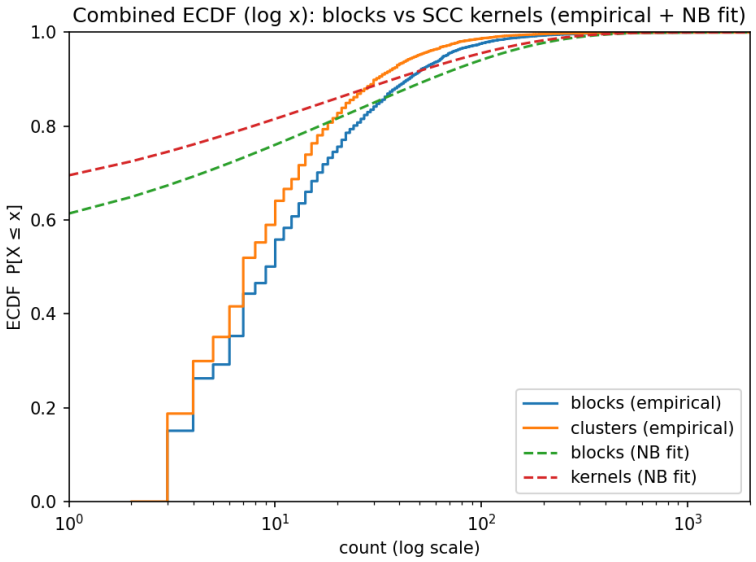

Figure 4: Empirical ECDFs (log x-axis) with moment-matched NB overlay. The poor fit confirms multi-regime workload structure.

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.

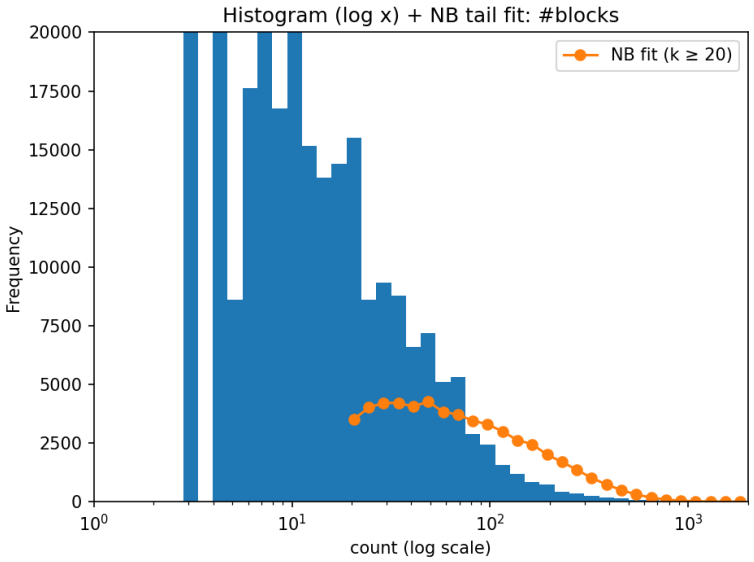

Figure 5: Block count histogram (log x-axis) with NB tail overlay. Even in the tail region, the fitted curve diverges from empirical counts.

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 [13] for SCC decomposition. Compared to Tarjan's algorithm [14]:

- 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.

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] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leiss, Alexandru Mallon, and Andreas Zwinkau. Simple and efficient construc-

tion of static single assignment form. *ACM Transactions on Programming Languages and Systems*, 40(3):1–30, 2018.

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

[4] 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.

[5] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. A liveness analysis for dead code elimination. In *Proceedings of the International Conference on Compiler Construction*, volume 641 of *LNCS*, pages 256–270. Springer, 1992.

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

[7] John McMillan and Thuy Nguyen. Data-driven optimization of register allocation. *ACM SIGPLAN Notices*, 2022.

[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] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

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

[12] Daniel Salomon et al. Empirical evaluation of register pressure under real workloads. *IEEE Transactions on Computers*, 2019.

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

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

[15] Christian Wimmer and Michael Franz. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 132–141. ACM, 2005.

[16] Christian Wimmer and Michael Franz. Linear scan register allocation on ssa form. In *Proceedings of the 2010 International Symposium on Code Generation and Optimization (CGO)*, pages 170–179. ACM, 2010.