Tiered Live Variable Analysis for Heavy-Tailed CFG Distributions

Frank Kuehnel

December 19, 2025

Abstract

Live range analysis is a foundational backward data-flow problem in register allocation. Textbook presentations suggest near-linear complexity, but this obscures two critical realities that emerge from empirical study of production workloads.

First, control-flow graph (CFG) structures follow a heavy-tailed distribution with distinct structural regimes. In a 290,000-function study of the Go toolchain, we find that 68% of CFGs are entirely acyclic (requiring no iteration), 24% contain exactly one loop kernel (median 6 blocks), and only 8% have multiple cyclic regions. CFG sizes span four orders of magnitude with a dispersion index of ≈175, demanding a tiered solver architecture.

Second, and more surprisingly, we find that *all* strongly connected components in our dataset converge in exactly three passes when using an alternating traversal order: postorder, then reverse postorder, then postorder again. This is significantly better than the $O(|B|^2)$ worst-case iterations suggested by standard complexity analysis, and to our knowledge has not been previously reported in the literature.

These findings motivate a tiered solver: SCC decomposition isolates cyclic regions, and within each SCC, a simple three-pass algorithm with alternating order suffices in practice. The key insight is to *optimize for the median, but design for the tail*—while recognizing that even the tail may be more tractable than theory predicts.

1 Introduction

Compiler back-ends must map an unbounded set of program temporaries to a finite set of hardware registers while keeping code fast and compact. Two core problems govern this mapping: *register allocation*, which decides where each variable resides during execution, and *live-range analysis*, which determines when a variable's value is actually needed [8]. The quality of this mapping significantly impacts performance and code size, especially on architectures with small register sets or tight memory hierarchies.

Over decades, researchers have developed a spectrum of allocation approaches—from graph-coloring to linear-scan [10] to more recent data-driven formulations—each balancing theoretical optimality against practical compile-time cost [6]. Modern compilers combine liveness information with interference graphs, SSA-based optimizations, and targeted spill code generation [2]. Yet most algorithmic work focuses on the *structure* of the problem (lattices, fixed points, graph properties) rather than the *distribution* of problem instances a compiler actually encounters.

The single-instance fallacy. Algorithm design typically focuses on individual problem instances. Consider the CFG fragment in Figure 1: a 12-block graph with a mix of forward edges, back edges, and nested structure. Textbooks analyze such examples to derive complexity bounds, prove correctness, and illustrate data-flow propagation [8, 5].

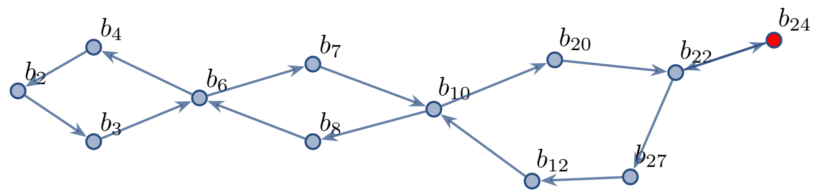

Figure 1: A typical CFG fragment used in algorithm exposition. Single-instance analysis reveals local structure but says nothing about the *distribution* of such structures a compiler must handle.

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:** ≈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 $\rightarrow$ reverse postorder $\rightarrow$ 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

This section contrasts our main empirical findings with established theory and prior empirical work on liveness analysis, CFG structure, and register allocation. Each claim is framed explicitly as a workload-specific observation against the broader background.

CFG distributions and structural regimes. Classical compiler texts treat CFGs largely through worst-case or stylized examples, with complexity arguments expressed in terms of $|B|$ and $|E|$ but without modeling the distribution of CFGs in large production workloads [8, 1, 9]. Our heavy-tailed, over-dispersed distributions for block and SCC counts (dispersion index ≈ 175) and the emergence of three distinct structural regimes ($\approx 68\%$ acyclic, $\approx 24\%$ single-loop, $\approx 8\%$ multi-loop) extend this picture by showing that real workloads are far from homogeneous in practice. While similar over-dispersion and regime effects are observed in empirical studies of program metrics and performance-critical tails, those works do not provide the CFG-focused, large-scale characterization reported here [12, 15]. Our contribution is thus complementary: it refines the practical backdrop against which worst-case CFG arguments are typically stated.

Worst-case bounds vs. observed convergence. The standard monotone-framework treatment of live-variable analysis guarantees convergence to a least fixed point on a finite lattice, with worst-case complexity up to $O(|B|^2)$ when $O(|B|)$ iterations are required [4, 3, 9]. Textbooks recommend DFS-based orders (postorder or reverse postorder) and note that, on structured code, convergence is usually fast in practice, often in a small number of passes [8, 1]. However, these works do not state a general constant-pass bound across large industrial workloads. Our empirical finding—that all SCCs in a 290k-CFG Go workload converge in exactly three passes under an alternating postorder $\rightarrow$ reverse-postorder $\rightarrow$ postorder schedule—does not contradict the theoretical $O(|B|)$ bound but instead narrows the gap between theory and practice for this workload. It shows that adversarial worst cases are not only rare but apparently absent in this setting, providing a stronger empirical guarantee than previously documented for liveness in production compilers.

Traversal order and solver design. Prior work on data-flow solvers broadly distinguishes between worklist-based "chaotic iteration" and fixed-order sweeps, with the common recommendation to use (reverse) postorder for backward problems on reducible CFGs [8, 9]. The literature offers little systematic exploration of alternating fixed orders within SCCs as a convergence accelerator; the default assumption is a single DFS-derived order repeated until a worklist stabilizes [1, 11]. In contrast, our alternating-order, three-pass scheme can be seen as a constrained, deterministic variant of chaotic iteration that empirically achieves convergence in a constant number of passes, with predictable memory access and without dynamic worklists. The novelty here is not the use of SCCs or postorder itself—both are well established [13, 14]—but the demonstration that a specific, simple order schedule is sufficient for all observed SCCs in a large industrial workload.

SCC decomposition and tiered architectures. Using SCC decomposition to localize iteration and obtain linear-time behavior on acyclic regions is classic in data-flow analysis and graph algorithms [13, 14, 3]. Modern SSA-based back ends build on these ideas but focus primarily on the representation and transformation of SSA programs, treating the data-flow solver as a largely standard component [11, 2]. Our tiered architecture sits at the intersection of these traditions: it leverages SCC decomposition in the textbook manner, but combines it with an explicitly regime-aware engineering stance ("optimize for the median, design for the tail") informed by the measured CFG distribution. This goes beyond generic worst-case-oriented designs by making the common acyclic and single-loop regimes first-class citizens in the solver architecture.

Relation to data-driven and trace-based allocation. Data-driven and trace-based approaches to register allocation—particularly in JIT and dynamic compilation settings—use profiling to focus compiler effort on hot paths and to guide allocation and spilling decisions [16, 7, 12]. These works operate mainly at the level of register pressure, spill heuristics, and trace selection rather than at the level of liveness solver internals. Our work complements that line by showing that the underlying live-variable analysis can itself be engineered with a workload-aware mindset: the heavy-tailed CFG structure and three-pass convergence suggest that a simpler, fixed-iteration solver suffices for a large class of real programs. In this sense, our contribution broadens the "data-driven" perspective downward into the data-flow layer, aligning empirical allocator research with equally empirical insights about the control-flow graphs those allocators depend on.

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.

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.

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

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

2.1 Structural Regimes: Where Cycles Live

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

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

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

Figure 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.

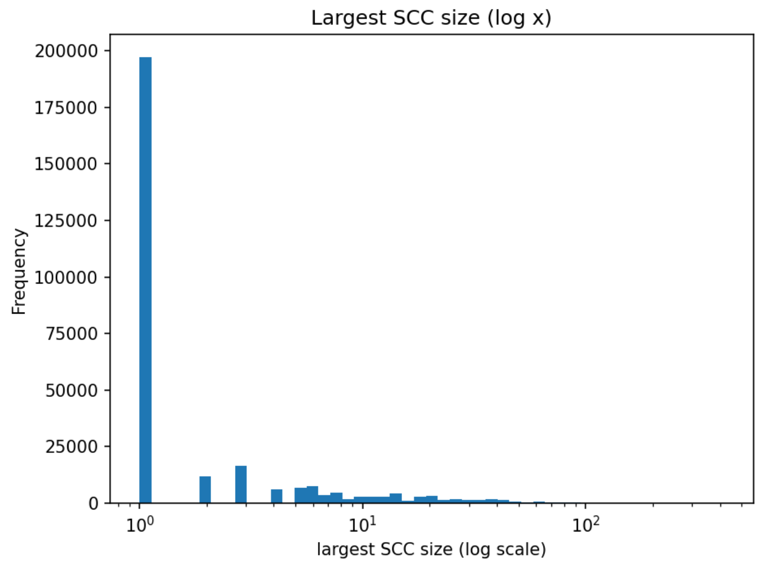

Figure 2: Distribution of largest SCC size (log scale). The spike at 1 represents the 68% of CFGs with no cycles. The tail extends to SCCs with hundreds of blocks.

- **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 CFGs (68%). These need exactly one backward pass—no iteration, no worklist, no convergence checking. The solver should recognize this case (all SCCs are singletons) and take the fast path. Setup overhead matters here: avoid allocating iteration-related data structures when they won't be used.

Regime 2: Single-loop CFGs (24%). One non-trivial SCC, typically small (median 6 blocks). The solver iterates within this single kernel while processing everything else in one pass. This is the "common loop" case—worth optimizing specifically because it covers a quarter of all functions.

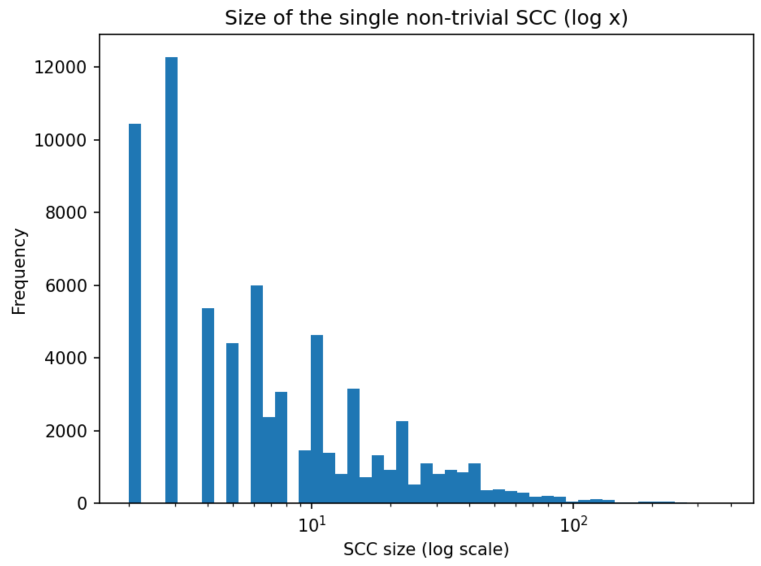

Figure 3: Size of the single non-trivial SCC (when exactly one exists). Median 6 blocks, $p_{90} = 27$, max $= 376$.

Regime 3: Multi-loop CFGs (8%). Multiple cyclic regions, potentially including the largest and most complex functions. Each SCC is solved independently, and our empirical finding (Section 4.2) shows that even these complex cases converge in exactly three passes with alternating order.

The SCC decomposition advantage. Remarkably, SCC-based processing handles all three regimes *without explicit case analysis*. Acyclic CFGs yield all-singleton SCCs processed in one pass. Single-loop CFGs yield one non-trivial SCC that iterates while singletons don't. Multi-loop CFGs yield multiple non-trivial SCCs, each solved independently. The algorithm naturally adapts to the regime.

Tail-aware engineering. The 8% complex regime includes functions with 12,000+ blocks and SCCs with hundreds of nodes. Our three-pass algorithm handles these efficiently:

- Fixed iteration count (three passes) gives predictable performance
- No dynamic worklist data structures needed
- Memory access patterns are sequential and cache-friendly

Metric selection. Use quantiles and regime fractions rather than means. The fraction of CFGs in each regime, the 99th percentile of non-trivial SCC size, and tail exceedance rates (e.g., fraction with SCC >100 blocks) are better stability indicators than averages.

4 A Tiered Solver Architecture

Our solver combines SCC decomposition with a novel three-pass iteration scheme to handle all three structural regimes efficiently and uniformly.

4.1 SCC Decomposition

We partition the CFG into strongly connected components using Kosaraju–Sharir (see Appendix C for pseudocode). Contracting each SCC yields a DAG—the *condensation graph*. For backward analysis, we process SCCs in reverse topological order: when solving any SCC, all successor SCCs are already complete.

Automatic regime adaptation. The key insight is that SCC decomposition *implicitly* handles all three regimes:

- **Acyclic CFGs:** All SCCs are singletons. Each is processed once, in topological order. No iteration occurs—the algorithm degenerates to a single backward pass.
- **Single-loop CFGs:** One SCC has multiple blocks; the rest are singletons. Singletons are processed in one pass; the non-trivial SCC iterates internally.
- **Multi-loop CFGs:** Multiple non-trivial SCCs, but each is solved independently. Inter-SCC dependencies are resolved by topological ordering, not iteration.

No explicit regime detection is needed—the structure emerges from the SCC partition.

4.2 Three-Pass Algorithm for SCCs

The standard approach for cyclic SCCs is either a worklist solver or repeated full passes until convergence. Textbook analysis suggests worst-case $O(|B|)$ iterations for a cycle of $|B|$ blocks.

However, our empirical study reveals a striking result: **all SCCs in our 290,000-function dataset converge in exactly three passes** when using an alternating traversal order—postorder, then reverse postorder, then postorder again.

Postorder and reverse postorder. A *postorder* traversal visits each node *after* visiting all its successors in the DFS tree. For a CFG, this means nodes with no outgoing edges (exits) appear early in the sequence, while the entry node appears last. Concretely, if DFS from entry e visits $e \to b_1 \to b_2 \to \cdots$, the postorder records nodes as their subtrees complete: exits first, entry last.

Reverse postorder (RPO) is simply this sequence reversed: entry first, exits last. For a DAG, RPO is a topological order. For backward data-flow analysis, RPO ensures that when we process a node, we have already processed its successors—ideal for propagating liveness information "upward" from exits toward the entry.

Why alternation helps on cycles. On acyclic graphs, a single RPO pass suffices. But cycles break the topological property: some successor will inevitably be processed after its predecessor, requiring iteration. The key insight is that *alternating directions* accelerates convergence:

Consider a cycle $b_1 \to b_2 \to b_3 \to b_1$. Suppose postorder is $[b_3, b_2, b_1]$ (exits-first) and reverse postorder is $[b_1, b_2, b_3]$ (entry-first).

- **Pass 1 (postorder $[b_3, b_2, b_1]$):** Information flows $b_3 \to b_2 \to b_1$. Facts from b_3 reach b_1, but b_1's facts don't yet reach b_3.
- **Pass 2 (reverse postorder $[b_1, b_2, b_3]$):** Information flows $b_1 \to b_2 \to b_3$. Now b_1's updated facts propagate forward to b_3.
- **Pass 3 (postorder):** Reconciles any remaining dependencies—facts that needed to "go around" the cycle twice.

With single-direction iteration, a fact at b_1 takes 3 passes to return to b_1. With alternating order, information flows both ways simultaneously, converging faster.

Algorithm inputs. The algorithm takes three inputs:

- **Initial facts** (called `start` in implementations): The data-flow values at SCC boundaries, inherited from successor SCCs that have already been solved. For liveness, this is the LiveIn of successor blocks outside the current SCC.
- **Entry node**: The block where DFS begins. For the full CFG, this is the function entry. Within an SCC, any block can serve as entry; the choice affects traversal order but not correctness.
- **The SCC subgraph**: The blocks and edges forming the strongly connected component.

Algorithm 1 gives the pseudocode. Each pass applies the transfer function f_b to every block, updating LiveIn from LiveOut.

Algorithm 1 Three-Pass SCC Solver

Require: SCC K (blocks and edges), entry node e, initial boundary facts from successor SCCs

Ensure: Converged LiveIn and LiveOut for all blocks in K

```
 1: function SolveThreePass($K$, $e$, $boundaryFacts$)
 2:     $po \leftarrow$ PostOrder($K, e$)                          ▷ DFS postorder starting from $e$
 3:     $rpo \leftarrow$ Reverse($po$)                                         ▷ Reverse postorder

 4:     // Initialize LiveOut from boundary (successor SCCs already
    solved)
 5:     for $b \in K$ do
 6:         LiveOut($b$) $\leftarrow \bigcup_{s \in \mathrm{Succ}(b)}$ LiveIn($s$)      ▷ Includes external successors
 7:     end for

 8:     ApplyPass($po$)                           ▷ Pass 1: postorder (exits $\rightarrow$ entry)
 9:     ApplyPass($rpo$)                  ▷ Pass 2: reverse postorder (entry $\rightarrow$ exits)
10:     ApplyPass($po$)                           ▷ Pass 3: postorder (exits $\rightarrow$ entry)
11: end function

12: function ApplyPass($order$)
13:     for $b \in order$ do
14:         LiveOut($b$) $\leftarrow \bigcup_{s \in \mathrm{Succ}(b)}$ LiveIn($s$)
15:         LiveIn($b$) $\leftarrow$ Use($b$) $\cup$ (LiveOut($b$) $\setminus$ Def($b$))
16:     end for
17: end function
```

Complexity. Each pass visits every block once: $O(|K|+|E_K|)$ per pass. With exactly three passes, the total cost is $O(|K|+|E_K|)$ per SCC. Summing over all SCCs (which partition the CFG), the total cost is $O(|B|+|E|)$—linear in CFG size.

Empirical validation. We validated convergence by comparing three-pass results against a reference worklist solver on all 290,000 functions. In every case, the three-pass algorithm produced identical results, confirming that three passes suffice for this workload. While we lack a formal proof of universal convergence, the empirical evidence—spanning reducible and irreducible control flow, simple loops and complex nests—suggests this is a robust practical bound.

Comparison to worklist solvers. A worklist solver guarantees convergence but incurs overhead: maintaining the queue, checking for changes, and reprocessing blocks in unpredictable order. The three-pass algorithm trades theoretical generality for simplicity: fixed iteration count, predictable memory access patterns, and no dynamic data structures. For the rare case where three passes might not suffice, a convergence check after pass 3 could trigger worklist fallback.

4.3 Complete Tiered Algorithm

Combining SCC decomposition with the three-pass solver yields the complete algorithm:

1. Compute SCCs of the CFG using Kosaraju–Sharir (Appendix C).
2. Process SCCs in reverse topological order of the condensation DAG.
3. For each SCC:
    - If singleton (acyclic): apply transfer function once.
    - If non-trivial (cyclic): apply three-pass algorithm.

This achieves $O(|B|+|E|)$ complexity for the entire CFG, regardless of cyclic structure.

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 $f_b(X) = \text{Use}(b) \cup (X \setminus \text{Def}(b))$ 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 reverse postorder (RPO) of a depth-first traversal [8, 1, 11]:

1. Compute a DFS postorder of the CFG.

2. Initialize $\text{LiveIn}(b) = \text{LiveOut}(b) = \emptyset$ for all b.

3. Repeat in reverse postorder until no changes:

   (a) Compute $\text{LiveOut}(b)$ from successors via Equation 1.

   (b) Compute $\text{LiveIn}(b)$ 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 4.2) 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 $L_b : V \to (\mathbb{N} \cup \{\infty\})$ 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 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 4.2), 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 $X \mid \Lambda \sim \mathrm{Poisson}(\Lambda)$ 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.

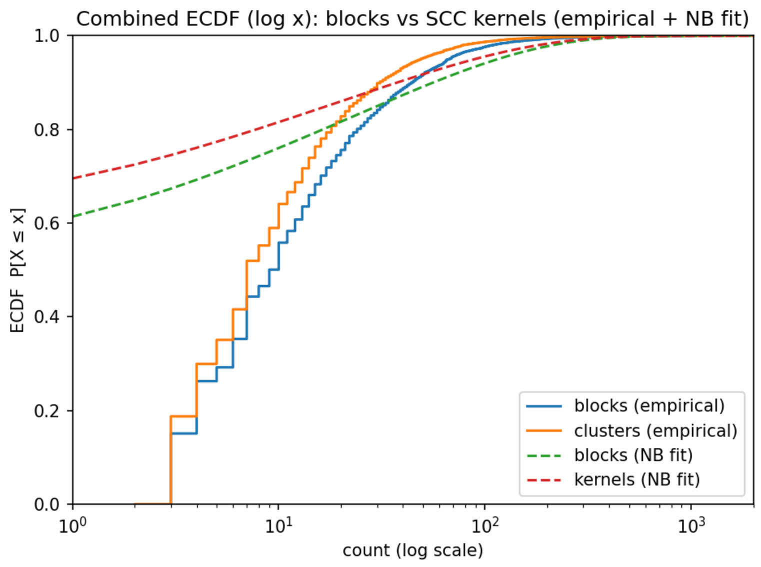

Figure 4: Empirical ECDFs (log x-axis) with moment-matched NB overlay. The poor fit confirms multi-regime workload structure.

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.

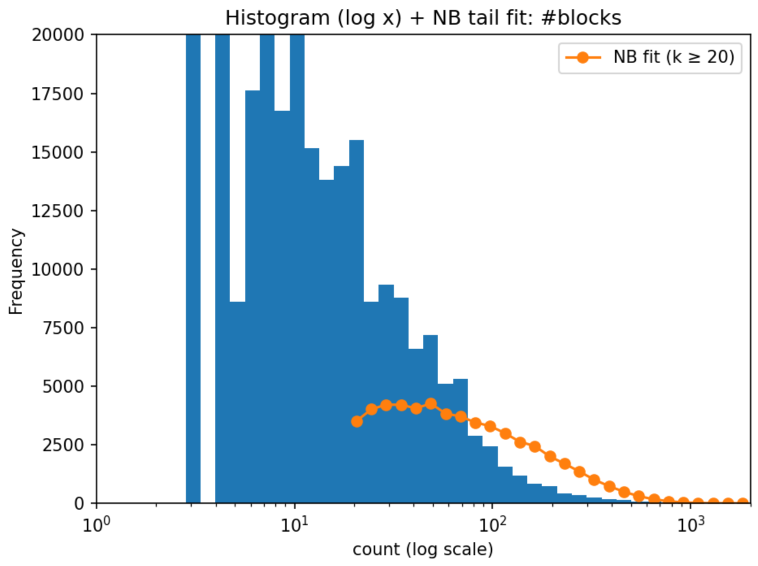

Figure 5: Block count histogram (log x-axis) with NB tail overlay. Even in the tail region, the fitted curve diverges from empirical counts.

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).

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.

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:

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

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).

- **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.

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:      $po \leftarrow$ Postorder($G$)                    ▷ DFS postorder on forward edges
3:      $seen \leftarrow \emptyset$
4:      $reachable \leftarrow \{b.ID : b \in po\}$
5:      $result \leftarrow [\,]$

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

11:         $scc \leftarrow$ 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:     $queue \leftarrow [leader]$
 3:     $scc \leftarrow [\,]$
 4:     $seen \leftarrow seen \cup \{leader.ID\}$

 5:     while $queue \neq \emptyset$ do
 6:         $b \leftarrow$ DEQUEUE($queue$)
 7:         APPEND($scc, b$)

 8:         for $e \in$ PREDS($b$) do                        ▷ Traverse reversed edges
 9:             $pred \leftarrow e.block$
10:             if $pred.ID \in reachable$ and $pred.ID \notin seen$ then
11:                 $seen \leftarrow seen \cup \{pred.ID\}$
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.