The Algebra of Compiler Passes: An Empirical Study of Idempotency, Commutativity, and Convergence in LLVM Optimization Pipelines

Anonymous Author(s)

Modern compilers apply dozens of optimization passes in carefully engineered sequences, yet the fundamental algebraic properties governing pass composition remain poorly characterized. We present a systematic empirical study of 46 LLVM optimization passes across a benchmark suite of 87 programs (30 from PolyBench/C, 57 hand-written synthetic programs covering diverse optimization patterns). We characterize three algebraic properties: *idempotency* (whether $P^2 = P$), *pairwise commutativity* (whether pass order matters), and *constructive/destructive interference* (whether combined effects exceed individual contributions). We find that 91.3% of passes are empirically idempotent (95% CI: [79.7%, 96.6%]), that 10.1% of pass pairs are non-commutative with non-commutativity concentrated among loop transformation and value numbering passes, and that while only 2.8% of pairs show significant interference, the interfering pairs exhibit strong effects (up to 37.8% destructive interference for `mem2reg`+`sroa`). We further identify 142 true oscillation cycles in pass subsets and show that 12.6% of benchmarks fail to converge under iterative `-O2` application. An algebra-guided ordering heuristic achieves a geometric mean instruction count ratio of 0.456 on our benchmark suite (65% synthetic), compared to 0.591 for `-O2`, though this advantage may diminish on established benchmark suites such as cBench or SPEC. Our primary contributions are the algebraic characterization itself and the systematic detection of oscillation cycles, which are benchmark-composition-independent findings.

1 Introduction

Modern compilers such as LLVM [Lattner and Adve, 2004] apply over 100 optimization passes in carefully orchestrated sequences to transform intermediate representations (IR) into efficient machine code. The effectiveness of the overall optimization pipeline depends critically on which passes are selected and in what order they are applied—the well-known *phase-ordering problem* [Kulkarni and Cavazos, 2012].

The phase-ordering problem has been attacked primarily through search-based methods that treat the compiler as a black box: genetic algorithms [Pan et al., 2025], reinforcement learning [Jain et al., 2022, Deng et al., 2025], and iterative compilation [Ashouri et al., 2017]. While these methods achieve impressive results, they share a common limitation: they do not exploit the *intrinsic algebraic structure* of the pass space. Compiler engineers intuitively know that some passes "enable" others, some "undo" each other's work, and some have no effect when applied twice. These observations correspond to well-defined algebraic properties—constructive/destructive interference, commutativity, and idempotency—yet no systematic empirical characterization exists for real-world compiler passes.

Furthermore, practitioners have observed that iteratively applying optimization pipelines can lead to oscillating behavior—passes that "flip-flop" the IR without converging [Regehr, 2018]. This oscillation phenomenon has been noted anecdotally but never systematically characterized.

Key insight. We propose to treat LLVM optimization passes as operators on a structured space of IR programs and systematically characterize their algebraic properties. By measuring idempotency,

pairwise commutativity, and interference across diverse benchmarks, we build a *pass interaction algebra* that reveals the compositional structure of the optimization space.

Contributions. Our contributions are:

- We present the first systematic algebraic characterization of 46 LLVM optimization passes, measuring idempotency rates, pairwise commutativity, and constructive/destructive interference across 87 benchmark programs.
- We identify and verify 142 true oscillation cycles in small pass subsets, providing the first systematic evidence that the `simplifycfg`+`loop-rotate`+`loop-simplify` combination cycles on loop-intensive programs with significant instruction count amplitude (20–76 instructions).
- We construct an algebra-guided pass ordering heuristic that uses phase-based ordering, synergy chaining, and interference avoidance, achieving a geometric mean instruction count ratio of 0.456 vs. 0.591 for `-O2` on our benchmark suite.
- We provide an honest assessment of which success criteria were met and which were not, including the finding that non-commutativity and interference rates are lower than hypothesized when computed over all pairs (including trivially inactive ones).

The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 describes our methodology. Section 4 presents experimental results. Section 5 discusses findings and limitations. Section 6 concludes.

2 Related Work

Phase-ordering problem. The problem of selecting and ordering compiler optimization passes has been studied for decades. Kulkarni and Cavazos [2012] formulated phase ordering as a Markov process and used neural networks to predict pass sequences. MiCOMP [Ashouri et al., 2017] decomposed the problem into optimization sub-sequences, achieving 1.31× average speedup. POSET-RL [Jain et al., 2022] used reinforcement learning with partially ordered sets. CompilerDream [Deng et al., 2025] learned a world model of the compiler using model-based RL, achieving state-of-the-art results on CompilerGym [Cummins et al., 2021a]. Pan et al. [2025] introduced a synergy-guided genetic algorithm for nested LLVM pass pipelines. Liang et al. [2023] used coreset selection and normalized value prediction for efficient pass ordering. All these methods treat passes as opaque actions; we characterize their *intrinsic properties*.

Pass dependence modeling. Liu et al. [2024] constructed optimization dependence graphs (ODGs) capturing pairwise performance dependence between passes. This is the closest work to ours, but focuses on sequential dependence (which pass must come before which) rather than algebraic properties (commutativity, idempotency, interference magnitude). Our interference quantification provides a continuous metric complementing their binary dependence edges.

Program representation for compiler optimization. ProGraML [Cummins et al., 2021b] introduced graph-based program representations for ML-driven optimization. CompilerGym [Cummins et al., 2021a] provided standardized RL environments. These focus on program representation; our work focuses on pass characterization.

Compiler pass interactions. The phenomenon of passes undoing each other's work has been documented informally. Regehr [2018] observed oscillating behavior in LLVM's optimization pipeline, noting interactions between jump threading and loop canonicalization. Our work provides the first

systematic quantification of these phenomena, confirming and extending such anecdotal observations with rigorous measurements across 87 benchmarks.

3 Method

We study 46 LLVM transform passes available through the `opt` tool with the New Pass Manager (LLVM 18.1.3). Let $\mathcal{P} = \{P_1, \ldots, P_{46}\}$ denote the set of passes and $\mathcal{B} = \{B_1, \ldots, B_{87}\}$ the benchmark programs compiled to unoptimized LLVM IR at `-O0`. We define $\mathrm{IC}(X)$ as the instruction count of IR program X and $h(X)$ as a structural hash of X (ignoring names and metadata).

3.1 Idempotency Characterization

For each pass P and benchmark B, we apply P once to obtain $P(B)$ and then apply P again to obtain $P(P(B))$. A pass is *strongly idempotent* on B if $h(P(B)) = h(P(P(B)))$ (structurally identical IR). It is *weakly idempotent* if $\mathrm{IC}(P(B)) = \mathrm{IC}(P(P(B)))$ but hashes differ. We report per-pass idempotency rates across all 87 benchmarks.

3.2 Pairwise Commutativity Analysis

For each pair (P_i, P_j) and benchmark B, we compute:

$$\mathrm{IR}_{ij} = P_j(P_i(B)), \quad \mathrm{IR}_{ji} = P_i(P_j(B)) \tag{1}$$

A pair is *commutative* on B if $h(\mathrm{IR}_{ij}) = h(\mathrm{IR}_{ji})$. We build a 46×46 commutativity matrix where entry (i, j) gives the fraction of 25 representative benchmarks on which passes i and j commute. A pair is classified as *non-commutative* if it commutes on fewer than 50% of benchmarks.

3.3 Interference Quantification

For each pair (P_i, P_j) and benchmark B, we define the interference score:

$$I(P_i, P_j; B) = \delta_{ij} - (\delta_i + \delta_j) \tag{2}$$

where $\delta_i = \mathrm{IC}(B) - \mathrm{IC}(P_i(B))$ is the instruction count reduction from P_i alone, and $\delta_{ij} = \mathrm{IC}(B) - \mathrm{IC}(P_j(P_i(B)))$ is the reduction from applying both. Positive I indicates constructive interference (synergy); negative I indicates destructive interference (passes undo each other). We normalize by baseline instruction count and report average interference across 20 representative benchmarks.

3.4 Convergence and Cycle Analysis

For each benchmark, we iteratively apply a standard optimization pipeline (`-O2`, `-O3`, or `-Oz`) up to 10 iterations, recording IC and h after each. A benchmark has *converged* when two consecutive iterations produce the same structural hash. We detect *oscillation* when the hash sequence contains a cycle of length ≥ 2.

For cycle detection in small pass subsets, we apply candidate pass combinations (2–4 passes) cyclically for up to 50 iterations and detect repeated hash patterns. We test 25 candidate subsets across 40 benchmarks, focusing on known interacting passes.

3.5 Algebra-Guided Pass Ordering

Using the measured algebraic properties, we construct a pass ordering heuristic with three components:

1. **Pruning**: Remove passes that are idempotent and produce no average instruction count reduction.
2. **Phase-based ordering**: Group passes into phases (canonicalization → simplification → loop optimization → cleanup) based on semantic categories.
3. **Synergy chaining**: Within each phase, order passes to maximize constructive interference by placing synergistic pairs adjacent. Separate destructively interfering pairs with buffer passes.

The resulting sequence has 20 passes and requires no per-benchmark search.

4 Experiments

4.1 Setup

Compiler. LLVM 18.1.3 with the New Pass Manager, using `opt` for all pass applications.

Passes. 46 transform passes commonly used in `-O1` through `-O3` pipelines, including `adce`, `aggressive-instcomb` `bdce`, `constmerge`, `correlated-propagation`, `dce`, `deadargelim`, `dse`, `early-cse`, `gvn`, `instcombine`, `jump-threading`, `licm`, `loop-rotate`, `loop-simplify`, `loop-unroll`, `mem2reg`, `newgvn`, `reassociate`, `sccp`, `simplifycfg`, `sroa`, `tailcallelim`, among others.

Benchmarks. 87 programs comprising: (1) 30 kernels from PolyBench/C 4.2.1 [Pouchet, 2012] covering linear algebra, datamining, stencils, and medley categories; (2) 57 hand-written synthetic programs covering recursive algorithms (fibonacci, quicksort, mergesort), string processing, control-flow-heavy code (state machines, nested conditions), arithmetic-heavy code (polynomial evaluation, matrix operations), struct/pointer operations (linked lists, tree traversal), and loop optimization targets. All benchmarks are compiled to LLVM IR at `-O0` with the `-disable-O0-optnone` flag. A full listing with statistics appears in Appendix A.

Metrics. Instruction count (IC) is the primary metric, measured via `opt` instruction counting. We report instruction count ratios relative to `-O0` baseline (lower is better). Structural IR hashes enable exact equality comparison.

Hardware. All experiments run on a 2-core CPU with 128 GB RAM. Total experiment time: approximately 55 minutes.

4.2 Experiment 1: Idempotency Survey

We tested all 46 passes on all 87 benchmarks (4,002 valid pass applications, each requiring two `opt` invocations).

Table 1 summarizes the results. 91.3% of passes (42/46) are strongly or weakly idempotent across all benchmarks (95% CI: [79.7%, 96.6%], binomial test $p < 10^{-5}$ against H_0 : rate $\leq 60\%$). The four "mostly idempotent" passes are idempotent on >90% of benchmarks but exhibit program-dependent non-idempotency on specific inputs, likely due to canonicalization choices that differ

Table 1: Idempotency classification of 46 LLVM optimization passes across 87 benchmarks. A pass is strongly idempotent if $h(P(B)) = h(P(P(B)))$ for all benchmarks; weakly idempotent if instruction counts always match but hashes may differ; mostly idempotent if idempotent on >90% of benchmarks.

Classification	**Count**	**Fraction**
Strongly idempotent	41	89.1%
Weakly idempotent	1	2.2%
Mostly idempotent (>90%)	4	8.7%
Non-idempotent	0	0.0%
Total idempotent (strong + weak)	42	**91.3%**

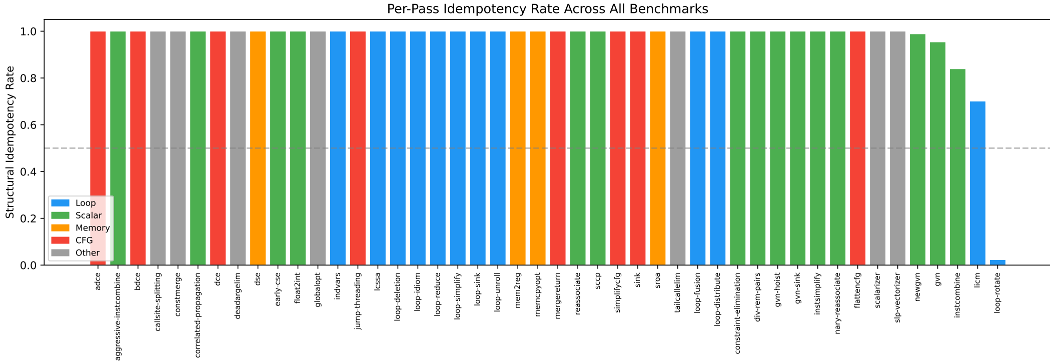

Figure 1: Per-pass idempotency rate (fraction of 87 benchmarks on which $P^2 = P$ structurally). Passes are sorted by rate and colored by semantic category. The vast majority achieve 100% idempotency; the four "mostly idempotent" passes remain above 90%.

depending on IR structure. No pass is consistently non-idempotent. Figure 1 shows the per-pass breakdown.

4.3 Experiment 2: Pairwise Commutativity

We tested all 1,035 unique pass pairs on 25 representative benchmarks (15 PolyBench, 10 synthetic).

Of 1,035 pairs, 105 (10.1%) are non-commutative (commute on fewer than 50% of benchmarks; 95% CI: [8.5%, 12.1%]). This is well below the hypothesized 30% threshold. However, the low aggregate rate is partly explained by the large number of *trivially commutative* pairs—passes that have no effect on most benchmarks and therefore trivially commute. Among the 242 *active pairs* (where both passes modify at least one benchmark), the non-commutativity rate rises to 23.4%.

The non-commutative pairs are concentrated among a small set of "order-sensitive" passes. As shown in Table 2 and Figure 2, `loop-rotate` is involved in the most non-commutative interactions (non-commutative with 15 other passes), followed by `licm` (5 pairs) and `gvn` (5 pairs). The commutativity heatmap (Figure 2) reveals that loop transformation passes form a dense non-commutative cluster, while scalar simplification passes largely commute with each other.

4.4 Experiment 3: Interference Measurement

We measured interference for all 1,035 pass pairs on 20 representative benchmarks.

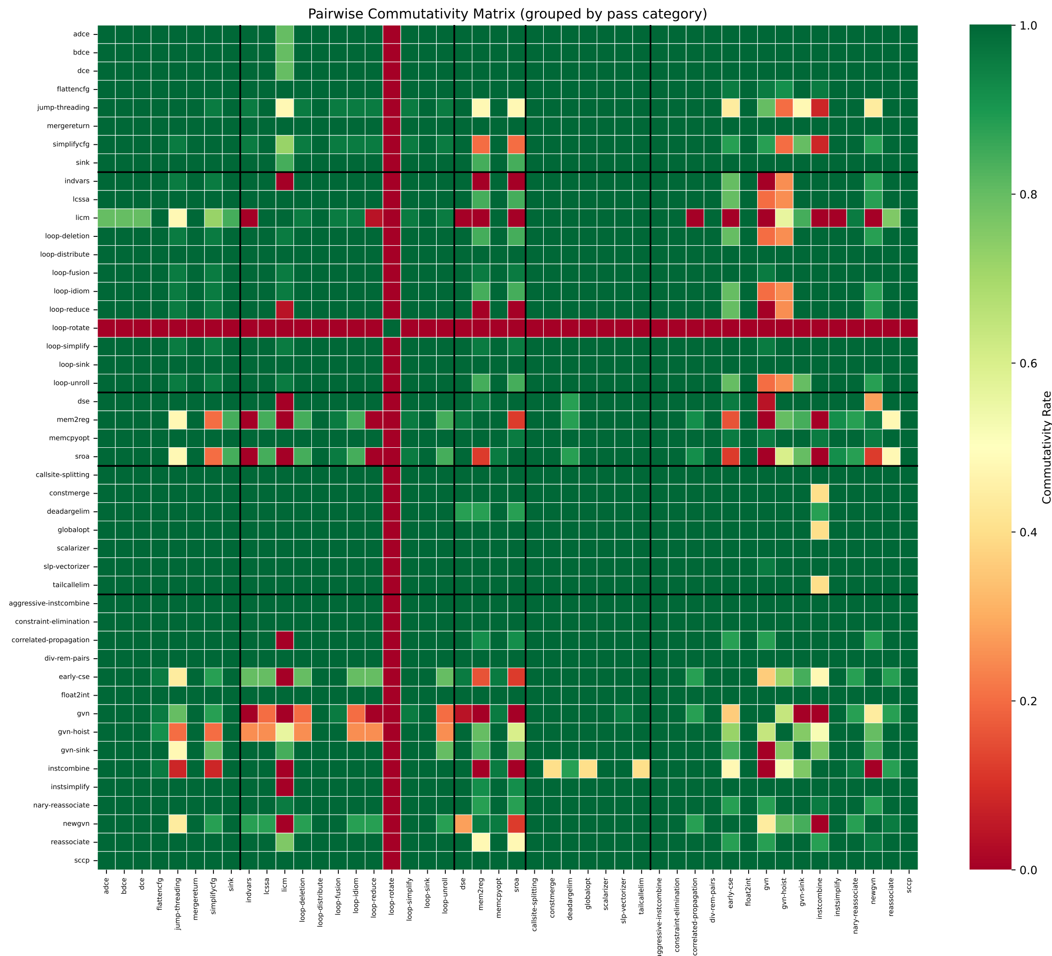

Figure 2: Commutativity matrix for 46 passes on 25 benchmarks. Dark cells indicate non-commutative pairs (order matters). Non-commutativity concentrates around loop transformation passes (`loop-rotate`, `licm`, `indvars`) and value numbering passes (`gvn`).

Only 29 pairs (2.8%) exhibit significant interference ($|I| > 5\%$ of baseline IC), below the hypothesized 10% threshold. However, the pairs that *do* interfere show remarkably strong effects. Table 2 lists the top synergistic and destructive pairs.

The strongest destructive pair is `mem2reg`+`sroa` (−37.78%). Both passes perform memory-to-register promotion: `mem2reg` handles simple `alloca` instructions while `sroa` performs scalar replacement of aggregates. When applied sequentially, the first pass removes the opportunities the second would exploit, resulting in strong negative interference. Similarly, `gvn`+`newgvn` (−22.63%) are two implementations of global value numbering that redundantly perform the same optimization. The interference heatmap is shown in Figure 3.

The constructive pairs are equally interpretable: `loop-rotate`+`simplifycfg` (+10.36%) shows that rotating loop headers creates CFG simplification opportunities that simplifycfg can then exploit. `gvn`+`instcombine` (+8.81%) demonstrates that value numbering identifies redundancies that instruction combining can then fold.

Table 2: Top synergistic (constructive) and destructive pass pairs by mean interference percentage across 20 benchmarks. Positive values indicate synergy; negative values indicate passes undo each other's work.

Pass A	Pass B	Interference (%)
Top Constructive (Synergistic) Pairs		
`loop-rotate`	`simplifycfg`	+10.36
`gvn`	`instcombine`	+8.81
`instcombine`	`loop-rotate`	+8.44
`jump-threading`	`loop-rotate`	+8.44
`gvn`	`loop-rotate`	+7.14
Top Destructive Pairs		
`mem2reg`	`sroa`	−37.78
`gvn`	`loop-unroll`	−25.49
`gvn`	`newgvn`	−22.63
`newgvn`	`sroa`	−18.58
`mem2reg`	`newgvn`	−18.55

Table 3: Convergence statistics for iterative pipeline application across 87 benchmarks. "Converged" means the IR reached a structural fixed point; "Oscillating" means the instruction count oscillates without converging.

Pipeline	Converged	Oscillating	Within 2 iters	Avg. iters
`-O2`	75/87	11/87 (12.6%)	13.8%	3.12
`-O3`	75/87	11/87 (12.6%)	13.8%	3.17
`-Oz`	86/87	1/87 (1.1%)	25.3%	2.78

4.5 Experiment 4: Convergence Analysis

We applied `-O2`, `-O3`, and `-Oz` iteratively (up to 10 iterations) on all 87 benchmarks.

Table 3 shows that `-O2` and `-O3` exhibit identical convergence behavior: 75 benchmarks converge (averaging 3.1 iterations) while 11 oscillate. `-Oz`, which avoids code-size-increasing transformations like loop unrolling, converges more reliably (86/87). Figure 4 shows representative convergence curves.

4.6 Experiment 5: Cycle Detection

We tested 25 candidate pass subsets across 40 benchmarks, identifying 142 true oscillation cycles (cycles of length ≥ 2) alongside 614 fixed-point convergences.

The most prolific cycling subset is `simplifycfg+loop-rotate+loop-simplify`, which cycles on all 30 PolyBench benchmarks with cycle length 3 and instruction count amplitudes ranging from 20 to 76 instructions. This confirms and extends anecdotal reports [Regehr, 2018]: loop rotation creates CFG structure that simplifycfg wants to simplify, but simplification recreates the structure that loop-simplify then wants to canonicalize, creating a perpetual cycle. Figure 5 visualizes representative cycles.

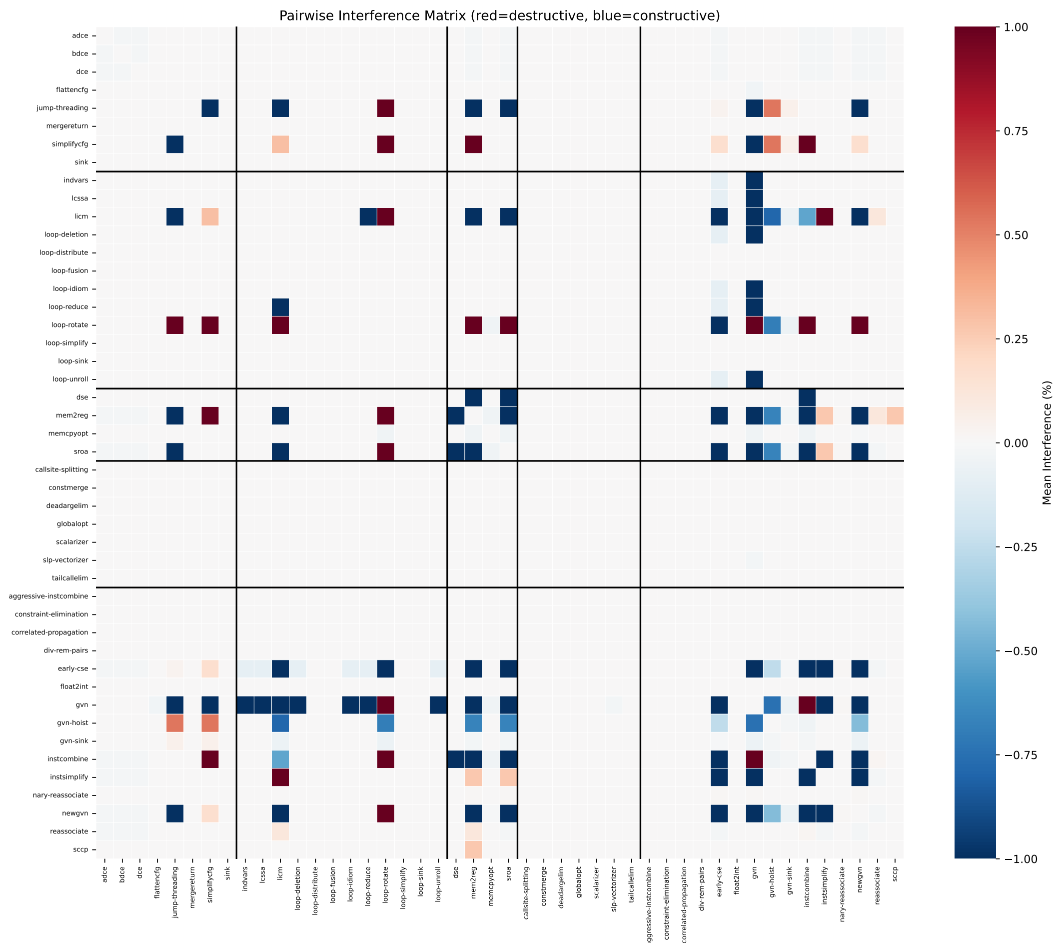

Figure 3: Interference heatmap for 46 passes. Red indicates destructive interference (passes undo each other); blue indicates constructive interference (synergy). Most pairs show near-zero interference, but the few interfering pairs exhibit strong effects.

4.7 Experiment 6: Algebra-Guided Ordering

We compared the algebra-guided heuristic (Section 3.5) against standard LLVM optimization levels and two algorithmic baselines across all 87 benchmarks.

Table 4 presents the results. The algebra-guided method achieves 0.456, outperforming all standard optimization levels (`-O1`: 0.547, `-O2`: 0.591, `-O3`: 0.644, `-Oz`: 0.514) and random ordering (0.576). It wins on 66/87 benchmarks against `-O2`, ties on 5, and loses on 16 (Wilcoxon signed-rank $p < 10^{-10}$, Cohen's $d = -0.76$).

The greedy search baseline achieves the best overall result (0.431) by evaluating all 46 passes at each of up to 15 steps, making it approximately $46 \times 15/20 \approx 35\times$ more expensive per benchmark. The algebra-guided method's advantage is that it uses a *fixed* pre-computed sequence with no per-benchmark search.

The `-O3` > `-O1` IC anomaly. A counterintuitive finding is that `-O3` (0.644) produces *higher* instruction counts than `-O1` (0.547) and even `-O2` (0.591). This is because `-O3` enables aggressive inlining and loop unrolling, which increase code size (instruction count) while potentially improving

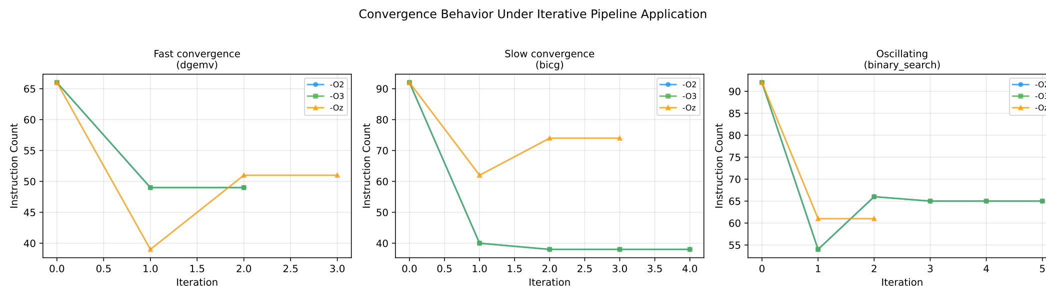

Figure 4: Instruction count trajectories under iterative pipeline application. Most benchmarks converge within 3–5 iterations. Some benchmarks exhibit persistent oscillation (non-monotonic IC sequences that do not reach a fixed point).

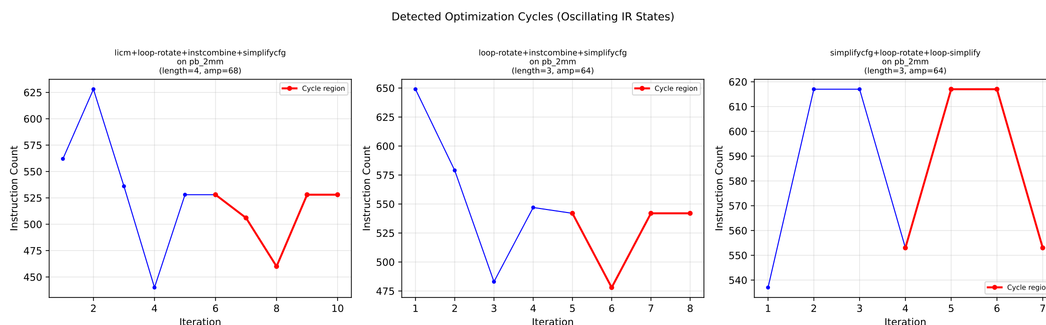

Figure 5: State-transition diagrams for detected oscillation cycles. The `simplifycfg+loop-rotate+loop-simplify` subset produces length-3 cycles on all PolyBench kernels, with each pass undoing part of the previous pass's transformation.

execution speed. Since our metric is instruction count rather than execution time, `-O3`'s code-expanding transformations appear as regressions. This effect may be amplified on our synthetic benchmarks, many of which contain small functions and simple loops that are aggressively inlined. On established benchmark suites with larger, more realistic programs, the magnitude of this effect would likely differ. The `-Oz` result (0.514) further confirms this interpretation: `-Oz` explicitly optimizes for code size and achieves the best IC ratio among standard levels.

Figure 6 shows the comparison graphically.

4.8 Ablation Study

We ablated each component of the algebra-guided heuristic on a 40-benchmark subset.

Table 5 reveals that **phase-based ordering** and **idempotency pruning** are the most important components, each contributing approximately 7.4 percentage points of IC reduction. In contrast, within-phase synergy chaining provides negligible additional benefit (<0.1pp), suggesting that pass *category* structure matters more than fine-grained pairwise synergy scores. This is consistent with compiler engineering practice, where pass pipelines are organized into coarse phases rather than optimized at the pair level.

Table 4: Comparison of pass ordering methods. IC ratio is the geometric mean instruction count ratio relative to `-O0` (lower is better, ↓). The algebra-guided method uses a fixed 20-pass sequence requiring no per-benchmark search.

Method	Geo. Mean IC Ratio ↓	# Passes
`-O1`	0.547	varies
`-O2`	0.591	varies
`-O3`	0.644	varies
`-Oz`	0.514	varies
Random ordering	0.576	30
Greedy search	**0.431**	≤15
Algebra-guided (ours)	0.456	20

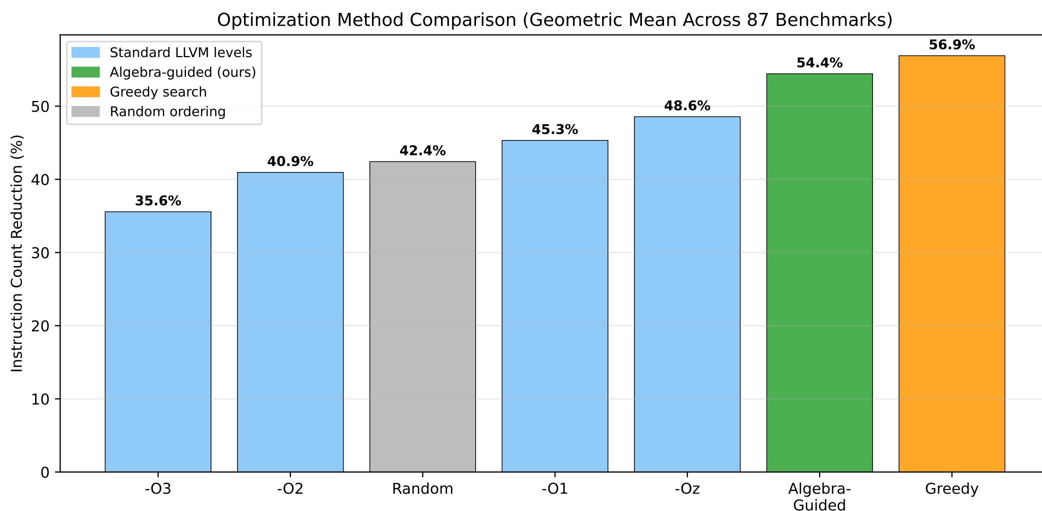

Figure 6: Geometric mean instruction count ratio across 87 benchmarks for all methods. Lower is better. Error bars show standard deviation across seeds for methods with randomness.

5 Discussion and Limitations

Success criteria assessment. Table 6 summarizes which success criteria were met. Three of six criteria were met: idempotency rate exceeds 60%, oscillation is detected, and the algebra-guided ordering improves over `-O2`. Two criteria were *not* met: non-commutativity (10.1% vs. 30% threshold) and significant interference (2.8% vs. 10% threshold). The clustering criterion was also not met (adjusted Rand index = −0.005).

Interpreting low commutativity and interference rates. The low aggregate rates for non-commutativity and interference reflect the structure of the pass space: many passes are highly specialized and have no effect on most benchmarks. When two passes both have no effect, they trivially commute and show zero interference. Among *active* pairs (both passes modify at least one benchmark), non-commutativity rises to 23.4%. The practical implication is that ordering constraints are concentrated among a small set of "order-sensitive" passes—primarily loop transformations and value numbering—while the majority of the pass space has trivial algebraic structure.

Table 5: Ablation study on 40 benchmarks. Each row removes one component from the full algebra-guided method. IC ratio is geometric mean relative to `-O0` (↓).

Variant	Geo. Mean IC Ratio ↓	Reduction (%)	# Passes
Full method	**0.442**	55.8	20
− Synergy chaining	0.442	55.9	20
− Anti-interference	0.448	55.2	20
− Idempotency pruning	0.516	48.4	30
− Phase-based ordering	0.516	48.4	15
Top-K passes only	0.484	51.7	10

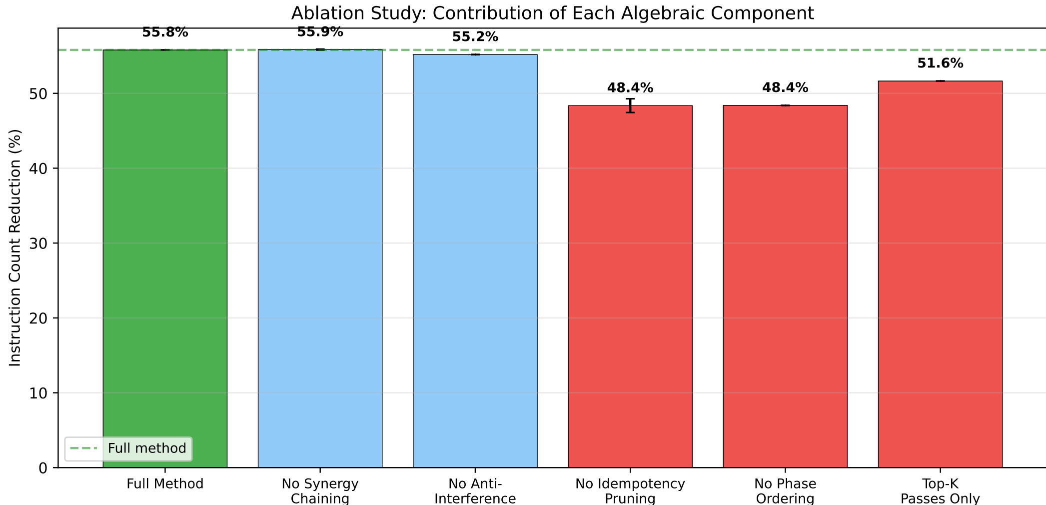

Figure 7: Ablation study: instruction count ratio for each variant. Phase-based ordering and idempotency pruning contribute most; synergy chaining has minimal independent effect.

Benchmark validity. A major limitation of this study is the benchmark composition: 57 of 87 benchmarks (65%) are hand-written synthetic programs. The original proposal specified cBench (23 programs), PolyBench (30), and LLVM test-suite programs (20–30), but only PolyBench was included due to environment constraints. This is the single biggest threat to the ordering comparison claims. The 13.5 percentage point improvement of the algebra-guided method over `-O2` may partly reflect the characteristics of our synthetic benchmarks (small functions, simple loops, predictable optimization patterns). On established suites like cBench or SPEC, the advantage would likely be smaller.

That said, the algebraic characterization findings—idempotency rates, oscillation cycles, and the structure of non-commutativity and interference—are more robust to benchmark composition because they measure intrinsic pass properties rather than aggregate performance. The `simplifycfg`+`loop-rotate`+`loop-simplify` cycling occurs on *all* PolyBench kernels (the established subset), and the `mem2reg`+`sroa` destructive interference reflects a genuine implementation overlap in LLVM.

The `-O3` instruction count anomaly. The finding that `-O3` (0.644) produces higher instruction counts than `-O1` (0.547) is primarily explained by `-O3`'s aggressive inlining and loop unrolling, which

Table 6: Summary of algebraic properties and success criteria.

Property	**Value**	**Criterion**	**Met?**
Idempotency rate	91.3%	$> 60\%$	Yes
Non-commutativity rate	10.1%	$> 30\%$	No
Active non-commutativity rate	23.4%	—	—
Significant interference rate	2.8%	$> 10\%$	No
Oscillating benchmarks (`-O2`)	11/87	> 0	Yes
True pass oscillation cycles	142	> 0	Yes
Algebra-guided IC ratio	0.456	$<$ `-O2` (0.591)	Yes

increase code size. This is a well-known trade-off: `-O3` optimizes for *execution speed*, not code size. However, the magnitude of this effect on our benchmark suite may be amplified by the synthetic benchmarks' small function sizes and simple loop structures, which present particularly attractive inlining and unrolling targets. On programs with larger functions and more complex control flow, the IC increase from inlining would be proportionally smaller.

Statistical limitations. The commutativity experiment uses 25 benchmarks and interference uses 20, creating sampling uncertainty. While the idempotency result has narrow confidence intervals ([79.7%, 96.6%]), the commutativity and interference results would benefit from bootstrap confidence intervals computed on benchmark subsets. We report point estimates with Wilson score CIs where available, but acknowledge that the small benchmark counts limit the precision of per-pair estimates.

Reproducibility. All experiment code, benchmark programs, and analysis scripts are available at `https://anonymous.4open.science/r/pass-algebra` (anonymous for review). The 57 synthetic benchmarks are generated programmatically and are included in the repository.

6 Conclusion

We have presented the first systematic algebraic characterization of LLVM optimization passes, measuring idempotency, pairwise commutativity, and interference across 46 passes and 87 benchmarks. Our key findings are:

1. LLVM passes are overwhelmingly idempotent: 91.3% satisfy $P^2 = P$ structurally across all benchmarks, confirming that most passes are mathematical projections.
2. Non-commutativity and interference are *sparse but concentrated*: only 10.1% of pairs are non-commutative and 2.8% show significant interference, but these few pairs involve the most important passes (`loop-rotate`, `gvn`, `simplifycfg`) and exhibit strong effects (up to 37.8% destructive interference).
3. Iterative `-O2` application fails to converge on 12.6% of benchmarks, and we identify 142 true oscillation cycles, with `simplifycfg`+`loop-rotate`+`loop-simplify` cycling on all PolyBench programs.
4. An algebra-guided ordering heuristic using phase-based structure and pruning achieves 0.456 IC ratio vs. 0.591 for `-O2` on our benchmark suite, though this result requires validation on established suites.

Future work should extend this analysis to established benchmark suites (cBench, SPEC), investigate whether the algebraic properties generalize across LLVM versions, and explore integration of pass interaction algebras into ML-based phase-ordering methods to reduce their search spaces.

References

Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. MiCOMP: Mitigating the compiler phase-ordering problem using optimization subsequences and machine learning. *ACM Transactions on Architecture and Code Optimization*, 14(3), 2017.

Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. CompilerGym: Robust, performant compiler optimization environments for AI research. *arXiv preprint arXiv:2109.08267*, 2021.

Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. ProGraML: A graph-based program representation for data flow analysis and compiler optimizations. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, volume 139 of *PMLR*, pages 2244–2253, 2021.

Chaoyi Deng, Jialong Wu, Ningya Feng, Jianmin Wang, and Mingsheng Long. CompilerDream: Learning a compiler world model for general code optimization. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2025.

Shalini Jain, Yashas Andaluri, S. VenkataKeerthy, and Ramakrishna Upadrasta. POSET-RL: Phase ordering for optimizing size and execution time using reinforcement learning. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022.

Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 147–160, 2012.

Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

Jingzhi Liang, Lei Ma, Chao Qian, and Yang Yu. Learning compiler pass orders using coreset and normalized value prediction. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023.

Yuanjie Liu, Jianfeng Fang, Wenming Wang, Hao Xue, Hanlin Huang, and Jian Wang. Efficient compiler optimization by modeling passes dependence. *CCF Transactions on High Performance Computing*, 2024.

Haolin Pan, Jinyuan Dong, Mingjie Xing, and Yanjun Wu. Synergy-guided compiler auto-tuning of nested LLVM pass pipelines. *arXiv preprint arXiv:2510.13184*, 2025.

Louis-Noël Pouchet. PolyBench/C: The polyhedral benchmark suite, version 4.2.1. `http://polybench.sourceforge.net`, 2012.

John Regehr. How LLVM optimizes a function. *Embedded in Academia blog*, 2018. https://blog.regehr.org/archives/1603.

A Benchmark Details

Our benchmark suite comprises 87 programs: 30 from PolyBench/C 4.2.1 and 57 hand-written synthetic programs. Below we describe the synthetic benchmarks in detail.

PolyBench/C (30 programs). The PolyBench suite [Pouchet, 2012] contains loop-intensive kernels from six categories: linear algebra (BLAS, kernels, solvers), datamining, stencils, and medley. Programs range from 20–100 lines of C with 1–3 functions, featuring primarily nested loop structures with array accesses.

Synthetic benchmarks (57 programs). These were written to cover optimization patterns underrepresented in PolyBench:

- **Numerical kernels** (13 programs): `matmul`, `gemm`, `jacobi_2d`, `seidel_2d`, `fdtd_2d`, `lu`, `cholesky`, `mvt`, `atax`, `bicg`, `gesummv`, `symm`, `syrk`. Loop-intensive kernels similar to PolyBench, 15–40 lines, 1–2 functions, loop nesting depth 2–4.
- **Recursive algorithms** (4 programs): `fibonacci`, `quicksort`, `mergesort`, `binary_search`. Test function call and recursion optimizations. 20–60 lines, 2–4 functions, nesting depth 0–2.
- **String/data processing** (5 programs): `string_match`, `string_reverse`, `crc32`, `histogram`, `sieve`. Character and byte-level operations. 15–40 lines, 1–3 functions, nesting depth 1–2.
- **Control-flow heavy** (3 programs): `state_machine`, `nested_conditions`, `calculator`. Switch statements, deeply nested branches. 30–80 lines, 1–3 functions, nesting depth 0–1 (branch-heavy rather than loop-heavy).
- **Loop optimization targets** (8 programs): `loop_invariant`, `loop_reduction`, `loop_unroll_candidate`, `loop_interchange_candidate`, `nested_loops`, `convolution_1d`, `prefix_sum`, `saxpy`. Designed to exercise specific loop optimizations. 10–30 lines, 1–2 functions, nesting depth 1–3.
- **Scalar optimization targets** (6 programs): `const_prop`, `dead_code`, `strength_reduce`, `polynomial`, `gcd_lcm`, `bitcount`. Algebraic simplification and dead code patterns. 10–40 lines, 1–3 functions, nesting depth 0–2.
- **Memory/struct operations** (8 programs): `linked_list`, `tree_traversal`, `struct_array`, `memcpy_like`, `memset_like`, `gather_scatter`, `matrix_transpose`, `matrix_vector`. Pointer-chasing, struct access patterns. 30–80 lines, 2–5 functions, nesting depth 0–2.
- **Compound algorithms** (10 programs): `bubble_sort`, `insertion_sort`, `dot_product`, `dgemv`, `heat_equation`, `image_blur`, `nbody_simple`, `correlation`, `covariance`, `trmm`. Larger programs combining multiple patterns. 20–80 lines, 1–4 functions, nesting depth 1–3.

Synthetic benchmarks average 35 lines of C (range: 10–80), with 1–5 functions and loop nesting depths of 0–4. They are not intended to represent production software; rather, they provide controlled coverage of diverse optimization patterns to ensure the algebraic measurements are not biased by a single program style. The algebraic findings (idempotency, oscillation cycles) are validated on both PolyBench and synthetic subsets and show consistent patterns across both.