TAMING EQUALITY SATURATION BY SYNTHESIZING REUSABLE STRATEGIES

Anonymous authors
Paper under double-blind review

ABSTRACT

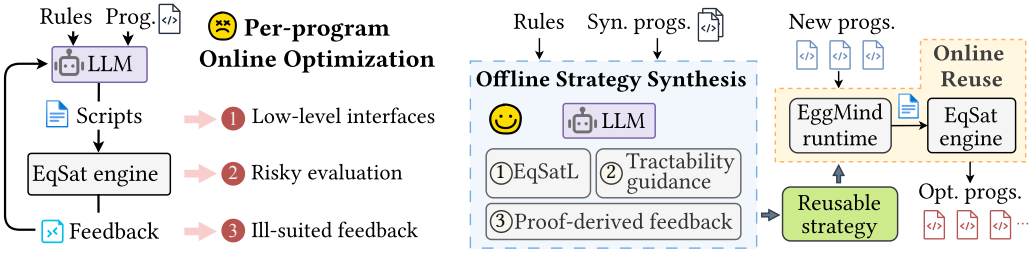
Equality Saturation (EqSat) compactly represents equivalent program variants in an e-graph and explores them through rewrite rules, exposing optimization opportunities overlooked by heuristic pass pipelines in conventional compilers. However, retaining these alternatives can cause exponential e-graph growth, incurring prohibitive runtime and memory overhead. Consequently, practical EqSat requires carefully engineered search control to orchestrate rewrite applications and constrain e-graph growth, which demands extensive manual effort. Large Language Models (LLMs) offer a promising way to automate this process, yet existing formulations typically optimize EqSat control separately for each input, causing domain-level optimization insights to be repeatedly rediscovered. Moreover, LLM-based EqSat search remains difficult because of the low-level controls, risky candidate evaluation, and weak refinement feedback exposed by existing systems. We present EGGMIND, a framework that addresses these issues by reformulating EqSat optimization as offline synthesis of reusable strategies followed by online reuse. EGGMIND introduces EqSatL, an executable structured representation of saturation strategies that exposes high-level decisions over rewrite interactions and simplification. Its synthesis process combines proof-derived structural feedback with tractability guidance to efficiently refine candidate strategies across representative program instances. The resulting strategies can be reused for unseen related programs without LLM calls. Evaluations show that EGGMIND consistently improves both solution quality and resource efficiency. Compared with unguided EqSat, it reduces expression cost by 54.6%, while achieving a $69.5\times$ runtime speedup and a 91.8% reduction in peak memory. It also achieves an $8.7\times$ speedup over an expert-designed strategy while matching or improving its solution quality.

1 INTRODUCTION

Compiler optimizations are always indispensable for unlocking peak performance across diverse modern computing systems. However, conventional compilers (Lattner et al., 2021) apply destructive optimization passes sequentially, introducing a phase-ordering problem that often yields suboptimal results. Equality saturation (EqSat) (Tate et al., 2009; Willsey et al., 2021; Zhang et al., 2023) addresses this problem by iteratively applying rewrite rules to accumulate equivalent program variants within a shared e-graph, ultimately extracting a highly optimized candidate guided by a specified cost model. By preserving these alternatives without premature commitments, EqSat enables global exploration of the optimization space under a given cost model, establishing itself as a premier optimization paradigm across machine learning compilation (Yang et al., 2021; Vohra et al., 2025; Park et al., 2026), program vectorization (VanHattum et al., 2021; Thomas & Bornholt, 2024), and hardware design (Hofmann et al., 2025; Kong et al., 2026; Xiao et al., 2026).

Despite this elegance, EqSat's expansive search space frequently triggers combinatorial state-space explosions. Unconstrained rewrites cause e-graphs to grow exponentially, rapidly exhausting memory and runtime budgets. Scalable EqSat therefore demands meticulously engineered search control, including rewrite organization and selection (Thomas & Bornholt, 2024; He et al., 2023; Zhang et al., 2025) and intermediate optimization guidance (Koehler et al., 2024). We unify these decisions under a common abstraction termed an *EqSat strategy*. Currently, designing robust strategies remains a largely heuristic and trial-and-error endeavor demanding substantial domain expertise and manual effort, posing a practical bottleneck to broader EqSat adoption.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



(a) Per-program online optimization by LLM (b) Overview of EGGMIND

Figure 1: Reformulating EqSat control around reusable strategy synthesis.

Recent LLM advancements in automating algorithm optimization (Chen et al., 2026; Novikov et al., 2025; Mikek et al., 2026; Qiu et al., 2026) suggest a natural pathway for EqSat strategy design. A straightforward formulation treats EqSat control as a *per-program online optimization* problem (Zhang et al., 2025). Given a target program and rewrite rules, the LLM iteratively constructs backend-specific control scripts to orchestrate rewrite interactions and execution order, runs them with the EqSat engine, and refines them using evaluation feedback, as illustrated in Figure 1a. Such per-program optimization confines the resulting experience to a single case. However, programs within the same domain often share similar optimization patterns and objectives, benefiting from recurring control logic. This mismatch causes substantial overhead through repeated rediscovery of similar control decisions. Beyond this lack of reusability, LLM-based EqSat strategy search itself presents several challenges that constrain its practicability. First, existing EqSat systems expose low-level control interfaces instead of strategy-level abstractions, forcing the LLM to encode rewrite organization and search control in backend-specific scripts. Second, candidate evaluation is risky, as permissive control can trigger rapid e-graph growth that consumes gigabytes of memory, while restrictive control can prematurely eliminate essential optimization paths. Third, EqSat provides ill-suited feedback for iterative refinement (Deo et al., 2026). Aggregate metrics are too coarse to isolate useful rewrite interactions, while raw e-graphs exceed LLM context bounds. Together, these challenges make LLM-based EqSat strategy search costly and its refinement unreliable.

In this paper, we present EGGMIND, a framework that reformulates EqSat control from per-program online optimization into *offline strategy synthesis* and *online reuse*. As Figure 1b depicts, EGGMIND synthesizes a reusable strategy offline from representative programs and rewrite rules in the target domain. The strategy captures reusable control decisions over rewrite organization, execution flow, and simplification, allowing optimization experience to be shared across related programs. During online reuse, these synthesized strategies can be lowered by the EGGMIND runtime to underlying EqSat scripts to optimize new programs without further model invocations.

Realizing this offline synthesis effectively requires addressing the practical challenges of LLM-based EqSat search. To this end, EGGMIND introduces three core mechanisms. First, **EqSatL** (Section 3) establishes a high-level language encapsulating rewrite organization and e-graph simplification, allowing the model to navigate strategy-level decisions rather than wrestling with low-level execution scripts. Second, **tractability guidance** (Section 4.2) steers search away from volatile strategies likely to cause resource exhaustion or over-constrained search. Third, **proof-derived feedback** (Section 4.1) isolates compact, rule-level experiences extracted directly from successful equivalence proofs. By recording exactly how the input program connects to the optimized result, this mechanism supplies the LLM with highly precise, structural signals for iterative refinement. Together, these mechanisms transform online per-program control into tractable offline synthesis, yielding reusable strategies that can be applied to other programs in the same domain.

In summary, this paper makes the following contributions:

- We introduce *reusable EqSat strategy synthesis* as a novel optimization paradigm, shifting expensive per-program LLM search from online deployment to offline strategy synthesis.
- We propose EqSatL, a structured search abstraction that decouples high-level EqSat control from low-level backends to enable systematic synthesis of reusable EqSat strategies.
- We implement EGGMIND, a strategy synthesis framework that efficiently navigates this search space with proof-derived structural feedback and tractability guidance to refine reusable strategies.

We evaluate EGGMIND across diverse domains and find that it consistently improves solution quality and resource efficiency. Compared with unguided EqSat, it reduces expression cost by 54.6% while achieving a 69.5× runtime speedup and a 91.8% reduction in peak memory. It also achieves an 8.7× speedup over an expert-designed strategy while matching or improving solution quality.

2 BACKGROUND AND PROBLEM FORMULATION

2.1 EQUALITY SATURATION

EqSat (Tate et al., 2009; Willsey et al., 2021) represents equivalent programs in an *e-graph* that compactly shares common substructures. An e-graph groups equivalent expressions into *e-classes*, each containing *e-nodes* that represent operators over child e-classes. Figure 2a shows an example e-graph $\mathcal{G}_1$, with dotted containers for e-classes and solid boxes for e-nodes. EqSat proceeds in two stages, *saturation* and *extraction*. Saturation applies rewrite rules to add equivalent expressions while preserving existing alternatives, until reaching a fixed point or a resource limit. Here, saturating rule r_1 ($m+m \rightsquigarrow 2\times m$) expands $\mathcal{G}_0$ to $\mathcal{G}_1$ by adding $2\times a$ and $2\times b$ to the corresponding e-classes.

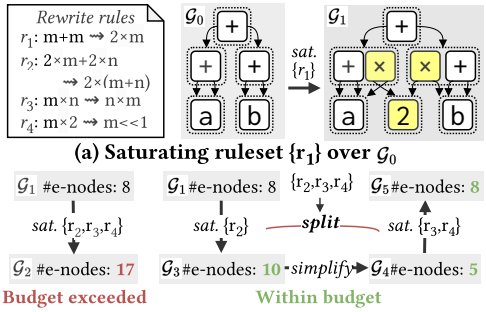


Figure 2: EqSat search control.

Extraction then selects a program according to a user-defined cost model. In practical EqSat engines such as egglog (Zhang et al., 2023), an optimization is specified through a backend script that defines the input program and rewrite rules, organizes rewrites into rulesets, schedules their execution during saturation, and extracts the optimized result. Appendix A.1 shows a representative example.

During saturation, newly added expressions can trigger further cascading rewrites and rapidly expand the e-graph. As shown in Figure 2b, saturating with all rewrites $\{r_2, r_3, r_4\}$ causes the number of e-nodes to more than double and exceed the memory budget. In realistic workloads, larger rewrite sets and denser rewrite interactions can amplify this effect into combinatorial e-graph explosion. Prior EqSat systems (Thomas & Bornholt, 2024) therefore control e-graph growth by partitioning rewrites and introducing *simplification* steps that reduce e-graph size by pruning less useful alternatives. In our example, separating the rewrites into groups with simplification between them keeps the intermediate e-graphs within the budget while preserving valuable optimization pathways.

2.2 EQSAT SEARCH CONTROL AND REUSABLE STRATEGY SYNTHESIS

We unify these forms of search control into an *EqSat strategy*, a set of control decisions governing saturation. It determines rewrite interactions and e-graph simplification without altering rule semantics or extraction. Existing systems expose varying subsets of this control space, yet differ substantially in how strategies are specified and whether they can be reused across programs.

Table 1: Comparison of representative EqSat control approaches. △ denotes limited reuse.

Approach	Control	Design	Reuse
Isaria	Schedule	Offline expert	✓
Guided EqSat	Targets	Offline expert	△
MCTS	Rules	Online search	×
ASPEN	Rules	Online LLM	×
EGGMIND	Strategy	Offline LLM	✓

Table 1 compares representative approaches by their control scope, design method, and reusability. Isaria (Thomas & Bornholt, 2024) relies on an expert-designed schedule with limited automation for rule grouping and simplification. Guided EqSat (Koehler et al., 2024) steers search using expert-provided intermediate targets that offer limited cross-search reuse. Monte Carlo Tree Search (MCTS)-based EqSat (He et al., 2023) performs online tree search over rewrite sequences for each input. Similarly, ASPEN (Zhang et al., 2025) employs an LLM for online rewrite selection per program. However, none of these approaches combines broad saturation control, automated offline design, and cross-program reuse. While recent engine-level advancements expose customizable saturation loops (Van der Cruysse et al., 2026), they still require strategies to be manually specified. EGGMIND therefore studies *reusable EqSat strategy synthesis*. Given syn-

Artifact

$$\textit{strategy} ::= \textit{strategy}(name,\ rulesets,\ flow)$$

Ruleset Partitioning

$$\textit{rulesets} ::= \textit{ruleset}^{+}$$
$$\textit{ruleset} ::= \textit{ruleset}(name,\ tags = [tag^{*}])$$

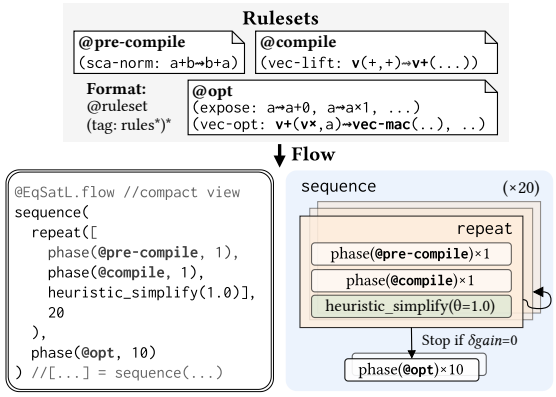
Schedule Construction

$$\begin{aligned} \textit{flow} &::= \textit{node} \\ &\mid \textit{sequence}(node^{+}) \\ \textit{node} &::= \textit{phase}(ruleset,\ iter_limit) \\ &\mid \textit{repeat}(flow,\ rounds) \\ &\mid \textit{simplify} \end{aligned}$$

Simplification Control

$$\begin{aligned} \textit{simplify} &::= \textit{heuristic_simplify}(\theta) \\ &\mid \textit{knowledge_guided_simplify}(\theta,\ poset) \\ \textit{poset} &::= \textit{pref_pair}^{*} \\ \textit{pref_pair} &::= (\textit{preferred},\ \textit{disfavored}) \end{aligned}$$

(a) Formal syntax of EqSatL.



(b) A compact EqSatL view of an expert-designed strategy.

Figure 3: EqSatL: (a) formal syntax and (b) an example strategy.

thesis programs, rewrite rules, a cost model, and resource constraints, the goal is to autonomously synthesize a single saturation strategy that can be reused effectively across unseen related programs without modifying the underlying rewrite semantics.

3 EqSatL: A Language for EqSat Strategies

Synthesizing reusable EqSat strategies requires transforming open-ended backend control into a structured space. EggMind achieves this through EqSatL, a domain-specific language (DSL) representing strategies as explicit artifacts. By decoupling strategic intent from concrete execution, EqSatL acts as an orchestration layer over engines like *egglog* (Zhang et al., 2023). It delegates rewrite execution to the backend while preserving the structural abstraction necessary for cross-program synthesis and reuse. Consequently, its design should balance three competing requirements: compactness for stable LLM generation, explicitness for reliable validation, and expressiveness to capture EqSat's full optimization potential. Our formulation relies on two observations regarding the trade-off between optimization quality and e-graph tractability. First, while one rewrite exposes opportunities for another, unconstrained interactions trigger explosive e-graph growth. Second, controlling this growth necessitates contraction: under-pruning leaves the search intractable, whereas over-pruning destroys promising optimization pathways. Thus, an effective strategy must dictate three core decisions: which rewrites interact, how these interactions unfold over time, and when and how the e-graph undergoes simplification. EqSatL explicitly encodes these decisions through ruleset partitioning, schedule construction, and simplification control, as Figure 3a illustrates.

Ruleset Partitioning. Ruleset partitioning controls rewrite interactions by determining which rules apply together during saturation. Rather than enumerating explicit rule lists, EqSatL abstracts rewrites using semantic *tags*, each representing a cluster of related rules. These tags provide a compact vocabulary for strategy construction, shielding the strategy space from low-level rewrite details. Derived once per domain through an initial LLM-driven categorization step, these tags are fixed and reused throughout strategy synthesis.

Schedule Construction. Schedule construction governs how ruleset interactions unfold over time. EqSatL represents execution as a *flow tree* built from `phase`, `sequence`, and `repeat`. A `phase` applies one ruleset for a bounded number of iterations, while `sequence` composes multiple flows in order. `repeat` revisits an entire subflow for up to a bounded number of rounds, terminating early when a round yields no further cost improvement. Each round ends with a mandatory `simplify` step to control e-graph growth during repeated interaction, as detailed in the following paragraph. This exposes more optimization opportunities across rounds without substantially increasing e-graph size.

Simplification Control. Even when rewrite interaction is well structured through partitioning and scheduling, it can still cause substantial e-graph growth. EqSatL therefore introduces `simplify` as a first-class strategy node, providing an interface for e-graph contraction instead of requiring it to be assembled from low-level backend primitives. Its default heuristic mode is controlled by a strength

parameter θ and is implemented using either extract-and-rebuild contraction or within-e-class cost-based pruning. EqSatL also supports *knowledge-guided simplification*, where structural knowledge is encoded as preferred and disfavored pattern pairs that guide pruning when equivalent forms arise in the same context.

EqSatL is expressive enough to capture expert-designed strategies such as Isaria (Thomas & Bornholt, 2024), with one such encoding illustrated in Figure 3b. The top panel organizes rewrites into semantic rulesets, while the bottom panel depicts the corresponding EqSatL control flow and its tree structure. Specifically, a `repeat` block sequentially applies the `pre-compile` ruleset, the `compile` ruleset, and a `heuristic_simplify` step for up to 20 rounds, concluding with a final phase over the `opt` ruleset. This example demonstrates how ruleset partitioning, repeated interaction, and explicit simplification seamlessly coalesce within a single EqSatL strategy.

4 Reusable Strategy Search

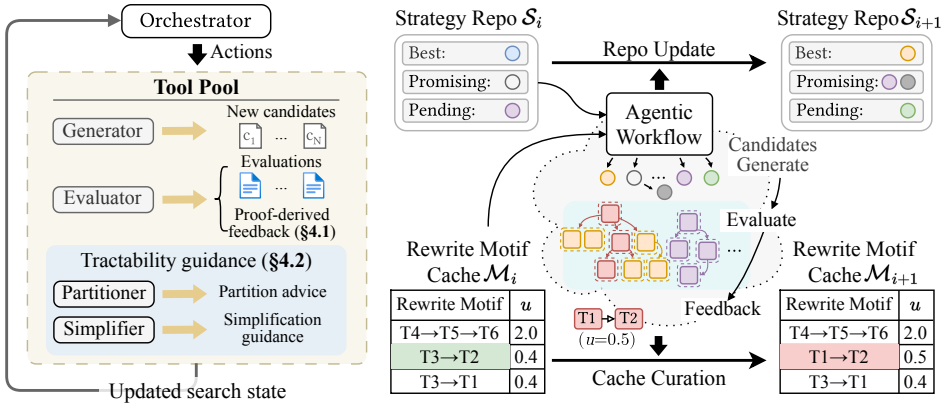


Figure 4: At each iteration, the agentic workflow in (a) proposes and evaluates EqSat strategies with proof-derived feedback and tractability guidance, updating the strategy repository and rewrite motif cache in (b) from i to $i+1$.

With EqSatL defining the strategy space, the remaining challenge is to search this space effectively while keeping candidate evaluations tractable. As shown in Figure 4, EggMind performs offline synthesis as an adaptive agentic search over EqSatL strategies. In the agentic workflow (Figure 4a), EggMind introduces a dedicated *Orchestrator* agent as a high-level decision maker. Its action space consists of the operations supported by the specialized agents in the tool pool. At each iteration, the Orchestrator observes the current search state and selects the next action and the corresponding agent to execute it. The *Generator* proposes or revises EqSatL strategies, while the *Evaluator* executes candidate strategies on the synthesis programs. Successful evaluations can further provide proof-derived structural feedback, described in Section 4.1. The *Partitioner* and *Simplifier* provide tractability guidance, detailed in Section 4.2. The outputs of these tools then update the search state for the next iteration, as shown in Figure 4b. At iteration i, the state consists of a strategy repository $\mathcal{S}_i$ and a rewrite motif cache $\mathcal{M}_i$. The repository $\mathcal{S}_i$ maintains the best strategy so far, promising strategies worth further refinement, and pending strategies awaiting evaluation, while the motif cache $\mathcal{M}_i$ stores proof-derived feedback from successful runs. Their updates produce $\mathcal{S}_{i+1}$ and $\mathcal{M}_{i+1}$, which form the search state for subsequent Orchestrator decisions. We next describe the structural feedback and tractability guidance that support this adaptive search.

4.1 Proof-Derived Structural Feedback

Backends such as egglog can provide a *proof tree* (Flatt et al., 2022; Zhang et al., 2023) showing how local equalities induced by rewrite applications are composed into a proof of equivalence between the input expression and the optimized result, as illustrated in Figure 5b. A rewrite application $r_i : \ell \Rightarrow \rho$ establishes one such local equality between its left- and right-hand sides, as represented by nodes such as r_1 and r_2. Within an EqSat run, rewrite applications interact with one another, and the local equalities they introduce are therefore interconnected. The proof tree captures these

relationships through several types of auxiliary reasoning nodes: *Congruence* (*Congr*) lifts an equality from a subexpression to its enclosing expression; *Transitivity* (*Trans*) chains equalities through a shared intermediate expression; and *Symmetry* (*Sym*) reverses an equality. Other auxiliary proof nodes are omitted here for clarity and described in Appendix B.2.

Unlike aggregate execution metrics, the proof tree reveals which rewrites contribute to the optimized result and how their effects are connected, providing structural feedback for strategy refinement. As shown in Figure 5a–b, when the Evaluator observes a measurable cost reduction $g = 1 - \text{cost}_{\text{after}}/\text{cost}_{\text{before}}$, EGGMIND requests the corresponding proof tree from the backend. The cost here comes from the problem-specific cost model. However, rewrite applications are interleaved with numerous auxiliary reasoning nodes and unrelated branches, making the raw tree difficult to use directly as feedback.

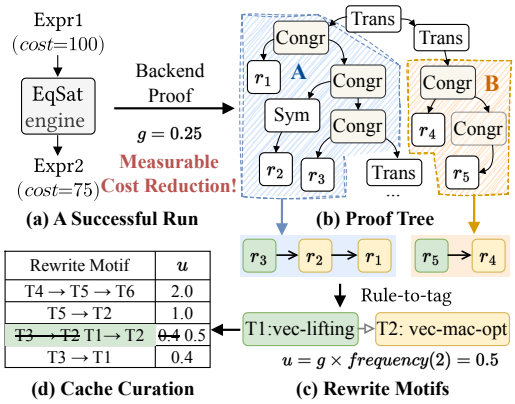


Figure 5: Proof-derived feedback in EGGMIND.

To isolate the relevant interactions, EGGMIND follows *Congr* paths in the proof tree. Because these paths track how local equalities propagate through enclosing expression contexts, they help identify the local regions in which related rewrite applications take effect, as highlighted in Figure 5b. Within each such region, EGGMIND retains the ordered rewrite applications and removes the remaining auxiliary proof structure, yielding a *rewrite chain*. EGGMIND then maps each rule in the rewrite chain to its semantic tag, removing rule-level details. We call the resulting tag sequence a *rewrite motif*. Because multiple rules may share a tag, different rewrite chains can collapse to the same motif. As illustrated in Figure 5c, one such chain is lifted to `vec-lifting`→`vec-mac-opt`, capturing the recurring interaction in which vector lifting enables a subsequent `vecmac`-related optimization. The resulting motifs are aggregated into the rewrite motif cache in Figure 5d. Each motif is scored as $u = g \cdot f$, where f denotes its frequency across successful cases. EGGMIND retains the top-scoring motifs according to u and exposes a small set of them as reusable context for proposing subsequent EqSatL strategies.

4.2 TRACTABILITY GUIDANCE

To keep strategy search tractable, EGGMIND augments the search with two complementary forms of guidance. Dependency-based partition advice steers rewrite grouping away from problematic interactions, while knowledge-guided simplification reduces the risk of prematurely pruning intermediate forms that remain useful for later optimization.

4.2.1 DEPENDENCY-BASED RULESET PARTITIONING

Large rewrite vocabularies make ruleset partitioning inherently difficult, as language models often fail to identify safe rule combinations reliably. Although exhaustive evaluation could eventually correct poor assignments, this approach becomes prohibitively expensive when adverse rewrites induce rapid e-graph blowup. To mitigate this costly exploration, EGGMIND scores candidate partitions using a dependency-based risk model and uses these scores to steer the strategy search.

We model the rewrite vocabulary as a directed dependency graph $G = (R, E, w)$, where each node $r \in R$ is a rewrite rule and each edge weight w_{ij} estimates how strongly r_i may enable r_j. The weight combines three signals of potential interaction: direct enablement when the right-hand side (RHS) of r_i matches the left-hand side (LHS) of r_j, operator and argument overlap, and subtree-level matchability. A phase assignment $\phi : R \to \{1, \ldots, k\}$ places each rule into one of k phases.

We score each assignment according to four dependency patterns. The objective rewards *forward* dependencies from phase t to phase $t + 1$, encouraging one phase to expose rewrite opportunities for the next. It penalizes *backflow* so that later phases do not reactivate earlier ones, and separately penalizes *phase-1 inflow* to keep the first phase locally coherent. It also penalizes *same-phase* dependencies, discouraging strongly interacting rewrites from being concentrated within a single phase.

Together, these terms favor forward activation while reducing uncontrolled rewrite interaction and e-graph growth. The term weights are fixed per domain through a lightweight grid sweep over the synthesis programs. We give the full objective in Appendix B.1.

4.2.2 KNOWLEDGE-GUIDED SIMPLIFICATION

Standard heuristic simplification contracts the e-graph by ranking e-nodes with a local cost model and pruning high-cost alternatives. This criterion can be myopic, as an intermediate form with poor local cost may still enable profitable rewrites later. Moreover, what should be pruned can differ across phases, making a fixed heuristic prone to removing forms that later optimization may still need. To avoid such premature pruning, EGGMIND employs a *Simplifier* agent that guides simplification in each phase using domain knowledge and experience accumulated from search feedback.

For each phase t, the Simplifier represents this guidance as a set of preferred and disfavored pattern pairs (p^+, p^-). These pairs define a partial order over patterns, which we term a *simplification poset*. We write $p^- \prec_t p^+$ to indicate that, when e-nodes matching p^+ and p^- coexist within the same e-class, nodes matching p^- should be deprioritized in favor of those matching p^+. If no matching p^+ is present, the preference does not penalize p^-. Instead of enforcing these preferences as hard deletions, EGGMIND incorporates them as additive penalties to the base cost. Specifically, the simplification score of an e-node e is $s(e) = c(e) + p^{(t)}_{\text{pos}}(e)$, where $c(e)$ is the base cost and $p^{(t)}_{\text{pos}}(e)$ is a bounded penalty induced by the simplification poset for phase t. Within each e-class, e-nodes are ranked by $s(e)$, and the top θ fraction is pruned. This biases simplification toward preferred forms while θ controls pruning aggressiveness. If the resulting guidance harms downstream optimization, evaluation feedback allows the Simplifier to revise the poset in subsequent search iterations.

5 EVALUATION

5.1 EXPERIMENTAL SETUP

We evaluate EGGMIND across three distinct domains including vectorization, tensor-graph optimization, and logic synthesis. Vectorization serves as the primary domain for end-to-end evaluation and architectural ablations, while XLA-based (OpenXLA Project, 2026) tensor-graph optimization and EqMap-based (Hofmann et al., 2025) logic synthesis evaluate the methodology across additional domains. Domain-specific configurations are detailed in their respective subsections.

We compare EGGMIND against unguided EqSat, expert-designed control strategies, and alternative synthesis baselines. Across workloads, we report final extraction cost under the domain-specific cost model, evaluator wall-clock time, and peak Resident Set Size (RSS). All evaluated configurations use *egglog*'s default greedy extractor (Zhang et al., 2023). To mitigate algorithmic variance, each strategy-search method is executed across three independent trials. We present cost reductions and evaluator speedups as ranges across completed runs, whereas offline search latency and total token consumption are reported as mean $\pm$ sample standard deviation. Unless otherwise specified, all model-based components use DeepSeek-V4.1-Flash (DeepSeek-AI, 2026). All experiments run on a dual-socket machine with Intel Xeon Gold 6348 CPUs (56 physical cores total) and 2 TiB of RAM. EqSat execution is CPU-only, with each evaluation limited to 600 seconds and 25 GB of memory.

5.2 EFFECTIVENESS ON VECTORIZING COMPILERS

In the vectorization setting, we follow prior EqSat-based work (VanHattum et al., 2021; Thomas & Bornholt, 2024) and evaluate on 2D-convolution (2D) and matrix-multiplication (MM) kernels across different problem sizes. We use 8 programs for offline synthesis and 14 held-out programs for evaluation, comparing against unguided EqSat and the expert-designed Isaria (Thomas & Bornholt, 2024) strategy using its original schedule. For the main end-to-end comparison, we report results from the strategy obtained in the initial synthesis run; two additional runs assess cross-run stability in Appendix C.3. Across the three runs, EGGMIND requires 15.1$\pm$2.4 minutes and 4.94$\pm$0.75M LLM tokens per run, corresponding to an estimated API cost of \$0.78$\pm$0.11 off-peak (\$1.57$\pm$0.23 peak) under cache-miss input pricing. The initial-run strategy is reused unchanged on all held-out programs without further LLM calls.

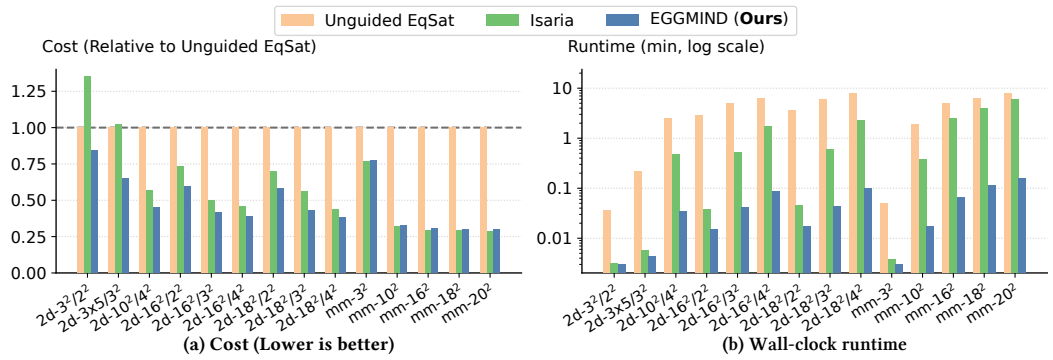


Figure 6: Performance of one synthesized strategy on 14 held-out vectorization programs: (a) normalized extraction cost and (b) evaluator wall time (log scale). Lower is better.

As shown in Figure 6, EGGMIND consistently improves both optimization quality and online efficiency over the two baselines. For vectorization, optimization quality is measured by final extraction cost under the benchmark's latency-oriented cost model. As shown in Figure 6a, EGGMIND reduces cost by 54.59% relative to unguided EqSat and by 14.30% relative to Isaria, based on geometric means of per-program cost ratios. It improves over unguided EqSat on all 14 programs and achieves lower cost than Isaria on 9 programs while matching it on the remaining 5. These results show that the synthesized strategy remains effective when reused on unseen programs. Figure 6b shows a similar advantage in online efficiency. EGGMIND is faster than both unguided EqSat and Isaria on all 14 held-out programs, with geometric-mean evaluator speedups of $69.49\times$ and $8.70\times$, respectively. It also reduces peak sampled RSS by 91.76% relative to unguided EqSat and by 72.43% relative to Isaria. On the largest matrix-multiplication case, evaluator wall time decreases from 472.7 s for unguided EqSat and 359.4 s for Isaria to 9.7 s with EGGMIND.

A qualitative inspection of the synthesized strategy helps explain these gains. EGGMIND places expansive *expose* rewrites, such as $a \rightsquigarrow a + 0$, inside bounded `repeat` regions together with vector lifting, local vector normalization, and simplification. In contrast, the expert-designed Isaria strategy in Figure 3b places several high-growth rewrites in a late `opt` phase with a large iteration budget, where repeated activation can amplify e-graph growth without comparable improvement in solution quality. By organizing these interactions more selectively, EGGMIND preserves useful optimization opportunities while limiting uncontrolled search expansion.

5.3 COMPARING STRATEGY SEARCH METHODS

We next compare four strategy search methods in the vectorization domain to disentangle the benefits of EqSatL from those of EGGMIND's synthesis procedure. A coding agent either synthesizes backend-specific *egglog* artifacts per program or explores EqSatL for reusable strategies. An evolutionary baseline navigates the EqSatL space without language models, whereas EGGMIND integrates EqSatL with its dedicated agentic search loop. All LLM-based methods use DeepSeek-V4.1-Flash across three independent trials. We assess optimization quality, online execution efficiency, search completion rates, and offline latency.

Coding-agent search can occasionally find strong solutions, but varies substantially across runs. Over raw *egglog*, complete runs achieve 17.54–42.19% cost reduction and 9.32–$16.93\times$ evaluator speedup. Over EqSatL, the best run reaches 50.74% cost reduction, while another complete run achieves only 4.59% and one run is incomplete. Evolution completes all runs and reaches 43.09–$112.48\times$ evaluator speedup, but achieves only -1.06–10.03% cost reduction. Appendix C.2 describes the evolutionary search algorithm and evaluation protocol.

By contrast, EGGMIND consistently achieves 54.39–54.59% cost reduction together with 27.03–$69.49\times$ evaluator speedup across all three runs. It completes every search run while requiring 64.3–70.4% less offline search time than the three alternative methods. Among the LLM-based methods, EGGMIND also has the lowest mean token consumption, using 23.3% fewer tokens than raw-*egglog* search and a comparable amount to EqSatL coding-agent search. Detailed per-run results are reported in Appendix C.3. Overall, EqSatL provides a reusable strategy space, while EGGMIND makes its search substantially more reliable and effective.

Table 2: Comparison of strategy-search methods on vectorization. Cost reduction and evaluator speedup are ranges over complete runs; search time and token usage are mean ± SD over all runs. Incomplete runs fail to finish all programs or exceed resource limits.

Method	Strategy Space	Complete Runs	Cost Reduction ↑	Evaluator Speedup ↑	Search Time (min) ↓	LLM Tokens (M) ↓
Coding agent	*egglog*	2/3 ✗	17.54–42.19%	9.32–16.93×	48.4±6.0	6.44±1.48
Coding agent	EqSatL	2/3 ✗	4.59–50.74%	1.16–5.87×	42.3±4.6	5.08±2.75
Evolution	EqSatL	3/3 ✓	−1.06–10.03%	43.09–112.48×	51.0±15.2	–
EGGMIND	EqSatL	**3/3 ✓**	**54.39–54.59%**	**27.03–69.49×**	**15.1±2.4**	**4.94±0.75**

5.4 ABLATION STUDY

Beyond baseline comparisons, we conduct an ablation study to isolate the performance contributions of individual EGGMIND components. We evaluate four structural variants across the 14 held-out vectorization programs against the complete system. All comparisons report the geometric mean of per-program ratios. Ablating proof-derived feedback severely limits search efficiency, inflating final cost by 118.2% while driving evaluator wall-clock time and peak RSS to 24.61× and 6.70× their reference baseline values. Disabling the `repeat` construct within EqSatL triggers the most drastic quality degradation, increasing cost by 155.2% and execution time to 8.96×. Removing the Orchestrator raises extraction cost by 32.0% and wall time by 82.4%, penalizing 13 of 14 benchmarks and confirming the necessity of high-level synthesis management. Finally, omitting the Partitioner increases cost by 15.3% and runtime by 26.3%, proving that strictly structured rewrite organization benefits both optimization quality and execution speed. Detailed per-benchmark metrics and a targeted analysis of knowledge-guided simplification are provided in Appendix C.4.

5.5 EVALUATIONS ON DIVERSE DOMAINS

We further evaluate whether the same strategy-synthesis methodology applies beyond vectorization. For each domain, EGGMIND synthesizes a new EqSatL strategy using the domain's own rewrite vocabulary, cost model, and workloads. Equality saturation has previously been applied to tensor-graph optimization and search control (Yang et al., 2021; Hartmann et al., 2024). We consider algebraic simplification in XLA HLO (OpenXLA Project, 2026; Arora et al., 2025) and an EqMap-based logic-synthesis case study (Hofmann et al., 2025; Pal et al., 2023).

On the XLA-based benchmark, EGGMIND achieves lower final cost than unguided EqSat on 13 of 17 programs and matches it on the remaining four, while reducing evaluator wall time by a geometric mean of 11.89×. In the EqMap (Hofmann et al., 2025) case study, we additionally consider an augmented rewrite vocabulary synthesized with Enumo (Pal et al., 2023). EGGMIND operates over the same augmented rewrite space as the unguided baselines and achieves the lowest cost on 8 of 9 carry-chain benchmarks. It reduces cost by 33.76% relative to the standard EqMap configuration and by 8.75% relative to the higher-budget baseline, while reducing peak sampled RSS by 51.94% relative to unguided search in the augmented rewrite space. Detailed setup, rewrite augmentation, and per-benchmark results for the EqMap case study are provided in Appendix C.5. Together, these results show that EGGMIND's strategy-synthesis methodology can be effectively applied across distinct EqSat domains.

6 CONCLUSION

We introduced EGGMIND, a framework for synthesizing reusable EqSat strategies to broaden the practical applicability of equality saturation. EGGMIND formalizes search control into a structured, executable strategy space via EqSatL, leveraging proof-derived structural feedback and tractability guidance to enable adaptive strategy refinement. Once synthesized, strategies deploy onto new programs with zero online inference overhead. Across diverse domains including vectorization, tensor-graph optimization, and logic synthesis, empirical evaluations demonstrate that EGGMIND consistently optimizes the trade-off between solution quality and resource efficiency, enabling effective reuse on unseen programs while outperforming expert-designed baselines.

AI USE STATEMENT

In this work, we used generative AI tools as part of the proposed methodology, specifically for the LLM-based strategy synthesis described in the paper. Additionally, we used generative AI tools to assist with manuscript editing to improve writing quality and readability, and to assist with the creation and refinement of scientific figures. We have reviewed all AI-assisted work and take responsibility for the final content of this work, including text, figures, and artifacts produced with the aid of generative AI.

REFERENCES

Jai Arora, Sirui Lu, Devansh Jain, Tianfan Xu, Farzin Houshmand, Phitchaya Mangpo Phothilimthana, Mohsen Lesani, Praveen Narayanan, Karthik Srinivasa Murthy, Rastislav Bodik, Amit Sabne, and Charith Mendis. Tensorright: Automated verification of tensor graph rewrites. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. doi: 10.1145/3704865. URL https://doi.org/10.1145/3704865.

Hongzheng Chen, Alexander Novikov, Ngân Vũ, Hanna Alam, Zhiru Zhang, Aiden Grossman, Mircea Trofin, and Amir Yazdanbakhsh. Magellan: Autonomous discovery of novel compiler optimization heuristics with alphaevolve, 2026. URL https://arxiv.org/abs/2601.21096.

DeepSeek-AI. Deepseek-v4.1-flash: Pushing the limits of kv cache compression, 2026. URL https://arxiv.org/abs/2609.19969.

Akash Deo, Simone Campanoni, and Tommy McMichen. AI coding agents need better compiler remarks, 2026. URL https://arxiv.org/abs/2604.13927. Presented at the Workshop on Co-Design for Agentic and Multimodal AI (CoDAIM) 2026.

Oliver Flatt, Samuel Coward, Max Willsey, Zachary Tatlock, and Pavel Panchekha. Small proofs from congruence closure. In *2022 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 75–83, 2022. doi: 10.34727/2022/isbn.978-3-85448-053-2_13.

Jakob Hartmann, Guoliang He, and Eiko Yoneki. Optimizing tensor computation graphs with equality saturation and monte carlo tree search. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, PACT '24, pp. 40–52, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706318. doi: 10.1145/3656019.3689611. URL https://doi.org/10.1145/3656019.3689611.

Guoliang He, Zak Singh, and Eiko Yoneki. Mcts-geb: Monte carlo tree search is a good e-graph builder. In *Proceedings of the 3rd Workshop on Machine Learning and Systems*, EuroMLSys '23, pp. 26–33, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700842. doi: 10.1145/3578356.3592577. URL https://doi.org/10.1145/3578356.3592577.

Matthew Hofmann, Berk Gokmen, and Zhiru Zhang. Eqmap: Fpga lut remapping using e-graphs. In *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, 2025. doi: 10.1109/ICCAD66269.2025.11240672.

Thomas Koehler, Andrés Goens, Siddharth Bhat, Tobias Grosser, Phil Trinder, and Michel Steuwer. Guided equality saturation. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi: 10.1145/3632900. URL https://doi.org/10.1145/3632900.

Sijie Kong, Jingtao Xia, Daniel Ruelas-Petrisko, Zachary D. Sisco, Jonathan Balkind, and Gus Henry Smith. Improving equality saturation for EDA via semantic e-graphs. *Proc. ACM Program. Lang.*, 10(PLDI), June 2026. doi: 10.1145/3808299. URL https://doi.org/10.1145/3808299.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, February 2021. doi: 10.1109/CGO51591.2021.9370308. URL https://ieeexplore.ieee.org/abstract/document/9370308.

Benjamin Mikek, Danylo Vashchilenko, Bryan Lu, and Panpan Xu. Agentic code optimization via compiler-llm cooperation, 2026. URL https://arxiv.org/abs/2604.04238.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

OpenXLA Project. Xla: Optimizing compiler for machine learning. https://openxla.org/xla/tf2xla, 2026. Accessed: 2026-04-14.

Anjali Pal, Brett Saiki, Ryan Tjoa, Cynthia Richey, Amy Zhu, Oliver Flatt, Max Willsey, Zachary Tatlock, and Chandrakana Nandi. Equality saturation theory exploration à la carte. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023. doi: 10.1145/3622834. URL https://doi.org/10.1145/3622834.

Jaehyeong Park, Youngchan Kim, Haechan An, Gieun Jeong, Jeehoon Kang, and Dongsu Han. Trinity: Three-dimensional tensor program optimization via tile-level equality saturation. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26, pp. 2079–2107, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400723599. doi: 10.1145/3779212.3790240. URL https://doi.org/10.1145/3779212.3790240.

Lei Qiu, Zi Yang, Fang Lyu, Ming Zhong, Huimin Cui, and Xiaobing Feng. Beyond pass-by-pass optimization: Intent-driven ir optimization with large language models, 2026. URL https://arxiv.org/abs/2602.18511.

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pp. 264–276. ACM, 2009. doi: 10.1145/1480881.1480915.

Samuel Thomas and James Bornholt. Automatic Generation of Vectorizing Compilers for Customizable Digital Signal Processors. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, volume 1 of *ASPLOS '24*, pp. 19–34, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 979-8-4007-0372-0. doi: 10.1145/3617232.3624873. URL https://dl.acm.org/doi/10.1145/3617232.3624873.

Jonathan Van der Cruysse, Abd-El-Aziz Zayed, Mai Jacob Peng, and Christophe Dubach. Parallel and customizable equality saturation. In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction*, CC '26, pp. 94–105, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400722745. doi: 10.1145/3771775.3786266. URL https://doi.org/10.1145/3771775.3786266.

Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pp. 874–886, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8317-2. doi: 10.1145/3445814.3446707. URL https://dl.acm.org/doi/10.1145/3445814.3446707.

Arya Vohra, Leo Seojun Lee, Jakub Bachurski, Oleksandr Zinenko, Phitchaya Mangpo Phothilimthana, Albert Cohen, and William S. Moses. Mind the abstraction gap: Bringing equality saturation to real-world ML compilers. *Proc. ACM Program. Lang.*, 9(OOPSLA2):329–356, October 2025. doi: 10.1145/3763062. URL https://doi.org/10.1145/3763062.

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi: 10.1145/3434304. URL https://doi.org/10.1145/3434304.

Youwei Xiao, Chenyun Yin, Yitian Sun, Yuyang Zou, and Yun Liang. Finding reusable instructions via e-graph anti-unification. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26, pp. 749–763, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400723599. doi: 10.1145/3779212.3790162. URL https://doi.org/10.1145/3779212.3790162.

Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. In A. Smola, A. Dimakis, and I. Stoica (eds.), *Proceedings of Machine Learning and Systems*, volume 3, pp. 255–268, 2021. URL https://proceedings.mlsys.org/paper_files/paper/2021/file/cc427d934a7f6c0663e5923f49eba531-Paper.pdf.

Niansong Zhang, Chenhui Deng, Johannes Maximilian Kuehn, Chia-Tung Ho, Cunxi Yu, Zhiru Zhang, and Haoxing Ren. Aspen: Llm-guided e-graph rewriting for rtl datapath optimization. In *2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD)*, pp. 1–9. IEEE Press, 2025. doi: 10.1109/MLCAD65511.2025.11189222. URL https://doi.org/10.1109/MLCAD65511.2025.11189222.

Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. Better together: Unifying datalog and equality saturation. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi: 10.1145/3591239. URL https://doi.org/10.1145/3591239.

A Background and Problem Formulation

A.1 Backend-Level EqSat Control in egglog

An egglog script specifies the rewrite system together with how the EqSat engine executes saturation. Beyond defining the input expression and rewrite rules, controlling the search requires explicitly organizing rewrites into rulesets and composing scheduling primitives to determine their execution order and iteration budgets. Listing 1 shows a schematic example corresponding to the type of staged EqSat control used in our workloads. Type declarations and rewrite details unrelated to search control are omitted for brevity.

Listing 1: Schematic backend-level EqSat control in egglog.

```
; Explicit rule-to-ruleset assignments
(ruleset pre-compile)
(rewrite (Add a b) (Add b a) :ruleset pre-compile)
(rewrite (Mul a 1) a :ruleset pre-compile)

(ruleset compile)
(rewrite (VecAdd (VecMul v a) v) (VecMac v a) :ruleset compile)
(rewrite (...) (...) :ruleset compile)

(ruleset opt)
(rewrite (Add a 0) a :ruleset opt)
(rewrite (Mul a 2) (Shl a 1) :ruleset opt)

; Explicit execution schedule
(run-schedule (repeat 20 (seq (run pre-compile 1) (run compile 1))) (run opt 10))

; Extract the current best expression
(extract root)
```

This representation exposes several backend-level details that must be handled explicitly when constructing EqSat control. First, rewrite organization is encoded through individual rule-to-ruleset assignments. The script does not provide a semantic categorization of rewrites, so constructing or modifying a partition requires manipulating the membership of individual rules. Second, execution control is expressed by composing low-level scheduling primitives. More involved phase structures and interactions must therefore be translated manually into a concrete schedule with explicit ordering and iteration bounds. Third, the e-graph simplification required by our workflow is not exposed as a first-class scheduling operation. For example, implementing extract-and-rebuild requires the surrounding runtime to extract the current best expression, construct a fresh e-graph using this expression as the new input, and then invoke the subsequent schedule.

These operations are necessary for backend execution, but they obscure the higher-level strategy decisions relevant to synthesis. EqSatL lifts these details into explicit abstractions for ruleset partitioning, schedule construction, and simplification control.

B Methodology

B.1 Dependency-Based Ruleset Risk Model

We model the rewrite vocabulary as a directed dependency graph $G = (R, E, w)$, where each node $r \in R$ denotes a rewrite rule and each weighted edge w_{ij} estimates the potential interaction from r_i to r_j. Specifically, w_{ij} combines three syntactic signals: (1) direct enablement, where the right-hand side (RHS) of r_i matches the left-hand side (LHS) of r_j; (2) operator and argument overlap between the two rules; and (3) subtree-level matchability, where the RHS of r_i can match a subtree of r_j.

A phase assignment $\phi : R \to \{1, \ldots, k\}$ places each rewrite rule into one of k phases. We score a candidate assignment using

$$
\begin{aligned}
\max_{\phi}\ & \alpha \sum\nolimits_{(i,j)\in E} w_{ij}\, \mathbf{1}[\phi(i)+1=\phi(j)] && \text{(forward)}\\
& -\beta \sum\nolimits_{(i,j)\in E} w_{ij}\, \mathbf{1}[\phi(i)>\phi(j)] && \text{(backflow)}\\
& -\gamma \sum\nolimits_{(i,j)\in E} w_{ij}\, \mathbf{1}[\phi(j)=1, \phi(i)\neq 1] && \text{(phase-1 inflow)}\\
& -\delta \sum\nolimits_{(i,j)\in E} w_{ij}\, \mathbf{1}[\phi(i)=\phi(j)] && \text{(same-phase).} \qquad (1)
\end{aligned}
$$

The first term rewards dependencies that move from phase t to phase $t+1$. The remaining terms penalize backflow to earlier phases, inflow into the first phase, and strongly interacting rewrites placed within the same phase, respectively. The coefficients $\alpha, \beta, \gamma, \delta$ control the relative strengths of these terms and are fixed per domain through a lightweight grid sweep over the synthesis programs.

B.2 Proof-Tree Representation

Egglog can reconstruct the reasoning that connects two equivalent expressions as a proof composed of rewrite applications and auxiliary equality steps (Flatt et al., 2022; Zhang et al., 2023). For analysis, we materialize this proof as a tree whose nodes record how individual equalities are established and combined. This representation allows us to recover not only which rewrite rules participate in a successful optimization, but also how their applications interact through the surrounding expression.

The proof trees encountered in our workloads contain several recurring node types. A *Rule* node records the application of a concrete rewrite rule and therefore directly identifies an optimization step. A *Congruence* node propagates an established equality from a subexpression to the larger expression that contains it. For example, given

$$a + b = b + a,$$

congruence allows this equality to be used inside a surrounding expression, establishing

$$(a + b) \times c = (b + a) \times c.$$

A *Transitivity* node connects two equality steps through a shared intermediate expression: from $e_1 = e_2$ and $e_2 = e_3$, it establishes $e_1 = e_3$. A *Symmetry* node reverses an established equality, deriving $e_2 = e_1$ from $e_1 = e_2$.

Egglog additionally supports proof steps arising from function merging. A *MergeFn* node records the application of a user-defined merge function. In egglog, when the same function input is associated with multiple outputs, its merge function combines the old and new values into a single result. The corresponding proof node records this merge as part of the derivation. Such nodes can therefore appear between rewrite applications even though they do not themselves represent rewrite rules.

Overall, *Rule* nodes identify the concrete transformations performed during optimization, while the remaining nodes describe how the equalities established by these transformations propagate through expressions and compose with one another. This distinction is important for EggMind: the raw proof tree contains the rewrite interactions of interest, but these interactions may be separated by auxiliary proof structure. Our proof-derived feedback therefore focuses on recovering local relationships among rewrite applications rather than exposing the complete raw proof tree to the LLM.

C Evaluation

C.1 Coding-Agent Search Details

The coding-agent baselines use DeepSeek-V4.1-Flash with the same one-hour total search budget, eight physical cores, and per-evaluation limits described in the experiment setup. Raw *egglog* directly optimizes a backend artifact for each of the 22 programs, whereas EqSatL searches for one reusable strategy using eight synthesis programs. For comparison, we report raw-*egglog* results on the same 14 programs used for EqSatL held-out evaluation, although raw *egglog* is allowed to optimize these programs directly.

Table 3 reports all three independent runs for each baseline. Raw *egglog* completes 22/22 programs in Runs 1 and 3, while Run 2 completes 17/22 and leaves five large cases unfinished after early termination when additional schedules exceeded resource limits. EqSatL completes the held-out evaluation in Runs 1 and 3. Run 2 produces a strategy that is usable on 8/14 held-out programs; the other six evaluations terminate at resource limits, including five RSS-limit terminations and one wall-clock termination. We omit aggregate gain and speedup for incomplete runs rather than treating missing cases as zero improvement. Search time includes model latency and strategy evaluation, while token counts are provider-reported totals.

C.2 Evolutionary Search over EqSatL

To evaluate whether EqSatL alone is sufficient for effective strategy synthesis, we implement a conventional evolutionary search directly over the same EqSatL strategy space used by EggMind. The baseline uses the same rewrite vocabulary, semantic tags, EqSatL validator and lowering pipeline, synthesis programs, and EqSat execution limits. It does not use an LLM, proof-derived rewrite motifs, dependency-based ruleset partitioning, or knowledge-guided simplification.

We use a mutation-based evolutionary search with a population size of four. Each individual is a valid EqSatL strategy. Mutations modify ruleset membership, flow structure, and bounded execution parameters, including moving or swapping tags between rulesets, splitting or merging rulesets, inserting, removing, or reordering phases, introducing or removing `repeat` and heuristic `simplify` nodes, and adjusting phase iteration limits, repeat rounds, and simplification strength. Evolution invokes no LLM during search.

To reduce evaluation cost, we use a two-stage protocol over the eight synthesis programs. Candidates are first screened on four programs and are promoted to the remaining four when they remain promising. Only candidates evaluated on all eight synthesis programs are eligible to become the final strategy. Candidate selection prioritizes feasible strategies with better extraction cost, while runtime and peak sampled RSS are used to distinguish strategies with similar optimization quality. Held-out programs are never used during search.

We allow at most 80 underlying EqSat executions in each evolutionary search run. Failed candidate evaluations consume this budget. We repeat the search three times with independent random seeds and evaluate the final strategy from each run on the 14 held-out programs. Its offline cost consists entirely of strategy evaluation and search orchestration.

C.3 Per-Run Strategy Search Results

Table 4 reports the three EggMind runs summarized in Table 2. Each run uses the same eight representative programs and a budget of 50 search steps. The resulting strategy is then reused unchanged on the same 14 held-out programs. Run identifiers abstract away the underlying random seeds.

Across the three runs, EggMind achieves a cost reduction of $54.52 \pm 0.12\%$. Offline synthesis takes 15.07 ± 2.41 minutes and consumes 4.94 ± 0.75M LLM tokens. These values report means and sample standard deviations across runs. Search time includes model latency and strategy evaluation but excludes held-out evaluation.

C.4 Ablation Study

We compare the full EggMind system with four variants on the same 14 held-out vectorization programs. *No Orchestrator* disables high-level decision making, *No Feedback* removes proof-derived feedback, *No Repeat* disables the `repeat` construct in EqSatL, and *No Partitioner* removes dependency-based ruleset partitioning. Figure 7 shows cost and complete evaluator wall-time ratios for six held-out programs spanning both workload families and small, medium, and large sizes. Aggregate statistics use geometric means of per-program ratios over all 14 held-out programs.

Removing proof-derived feedback or `repeat` increases cost, runtime, and peak sampled RSS on every held-out program. Without proof-derived feedback, cost increases by 118.2%, while runtime and peak sampled RSS reach $24.61\times$ and $6.70\times$ those of the full system, the largest aggregate resource overhead. On 2D $18^2 \times 2^2$, runtime rises from 1.03 s to 98.4 s, a $95.1\times$ increase. Disabling

Table 3: Per-run results for the DeepSeek-V4.1-Flash coding-agent baselines. Coverage is the number of completed programs in the search cohort: 22 programs for raw *egglog* and 14 held-out programs for EqSatL. Cost reduction and speedup are reported only for complete runs.

Method	Run	Coverage	CostReduction	EvaluatorSpeedup	SearchTime (min)	Requests /Tokens (M)
Raw *egglog*	1	22/22	17.54%	9.32×	43.18	81 / 6.19
Raw *egglog*	2	17/22	–	–	54.91	91 / 8.03
Raw *egglog*	3	22/22	42.19%	16.93×	47.16	66 / 5.10
EqSatL	1	14/14	4.59%	5.87×	47.60	90 / 8.19
EqSatL	2	8/14	–	–	39.02	73 / 4.11
EqSatL	3	14/14	50.74%	1.16×	40.39	56 / 2.95

Table 4: Per-run results for EGGMIND. Cost reduction and evaluator speedup are geometric means over the 14 held-out programs. Search time and LLM tokens measure offline synthesis. SD denotes sample standard deviation across runs.

Run	Coverage	Cost Reduction	Evaluator Speedup	Search Time (min)	LLM Tokens (M)
Run 1	14/14	54.59%	69.49×	13.17	4.15
Run 2	14/14	54.59%	52.36×	14.27	5.64
Run 3	14/14	54.39%	27.03×	17.78	5.02
Mean ± SD	–	$54.52 \pm 0.12\%$	–	15.07 ± 2.41	4.94 ± 0.75

`repeat` produces the largest aggregate cost increase of 155.2%, with runtime and peak sampled RSS reaching 8.96× and 2.11×.

Removing the Orchestrator increases cost on 13 of 14 programs and matches the full system on the remaining program. Its aggregate cost and runtime increase by 32.0% and 82.4%, respectively. Removing the Partitioner increases aggregate cost by 15.3% and runtime by 26.3%. It produces higher cost on 10 programs and longer runtime on 12 programs. Peak sampled RSS increases by 6.1%. These results show that partitioning improves the aggregate quality and runtime of the synthesized strategy, although its effect is not uniform across individual programs.

Role of Simplification. On the vectorization benchmark, simplification is a core component of successful strategies. Both Isaria and EGGMIND place explicit `simplify` nodes at phase boundaries and inside refinement loops, reflecting the need to control e-graph growth while preserving useful intermediate forms. For most workloads, EGGMIND's ruleset partitioning and schedule construction already steer e-graph growth in useful directions. Heuristic simplification therefore does not need to identify which forms remain valuable later and can mainly focus on contraction, so disabling knowledge-guided simplification changes little. However, simplification acts as a crucial safeguard when rulesets are arbitrary and disjoint. To evaluate the effectiveness of posets more directly, we study a targeted high-growth setting derived from Isaria's `opt`×10 block (Figure 3b), where *expose*-style and vector-optimization rewrites are mixed in one coarse phase. In this setting, heuristic simplification can become myopic because it contracts locally without recognizing which intermediate forms still matter for later progress. Figure 8 compares simplification controls under a fixed iteration budget. Skipping `opt` reduces runtime (0.4 s) but incurs a +55 cost delta. *Heuristic simplification* with varying strengths already retains most of the benefit, keeping results closer to baseline (+38) while reducing runtime to 0.5–0.8 s, where *light* and *hard* correspond to the strength parameter θ. We further evaluate *knowledge-guided simplification* through two simplification posets proposed by the Simplifier. *Poset 1* orders commutative variants and reaches +15 in 2.2 s. *Poset 2* extends it with a `vecmac`-specific preference and recovers baseline quality in 3.54 s. These results show that simplification itself is essential, and that posets are most useful when coarse phase structure makes heuristic contraction less reliable.

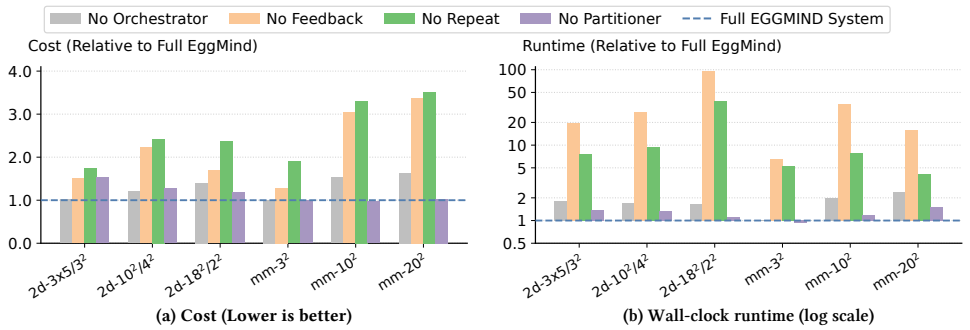


Figure 7: Component ablations on six representative held-out vectorization programs. Cost and evaluator wall time are normalized to the full EGGMIND system (dashed line at 1); lower is better. Complete evaluator wall time uses a logarithmic axis, with bars extending from 1 to each ratio. Aggregate statistics in the text use all 14 held-out programs.

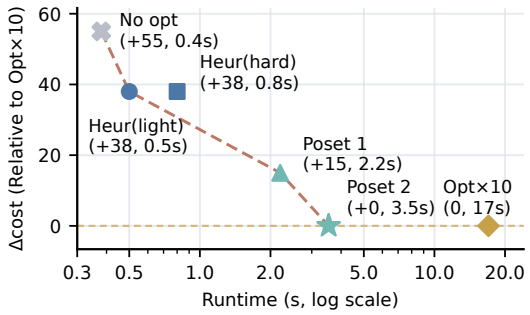


Figure 8: Trade-off among simplification controls.

C.5 Case Study on Logic Synthesis

In this case study, we examine EGGMIND with technology mapping in logic synthesis. Our rewrite space, derived from the ASIC EqMap flow (Hofmann et al., 2025), combines Boolean equivalence rewrites for logical exploration with lifting rules for technology mapping, and is evaluated on carry-chain benchmarks of increasing bit-widths. While expressive rewrites and large budgets can, in principle, expose lower-cost cell mappings, many profitable mappings require long rewrite chains. Under the original EqMap flow, such mappings typically emerge only with large global iteration budgets, which become computationally prohibitive as circuit width grows. The key bottleneck is therefore not expressiveness, but practical reachability under bounded search.

Rule Augmentation with Enumo. We use Enumo (Pal et al., 2023) to synthesize interaction rewrites between Boolean logic and physical cells. Two representative rules are shown below:

$$\mathsf{Or}\big(\mathsf{AOI21}_{\mathsf{X1}}(c,b,a),\ \mathsf{AOI21}_{\mathsf{X1}}(d,b,a)\big) \Longrightarrow \mathsf{AOI22}_{\mathsf{X1}}(d,c,b,a),$$
$$\mathsf{And}\big(\mathsf{OAI21}_{\mathsf{X1}}(d,b,a),\ \mathsf{OAI21}_{\mathsf{X1}}(c,b,a)\big) \Longrightarrow \mathsf{OAI22}_{\mathsf{X1}}(b,a,c,d).$$

These rewrites collapse long chains of Boolean normalization, factoring, and lifting into single-step transformations, enabling profitable higher-level cell mappings to be reached earlier in the search. However, they also enlarge the matching space and make the search more susceptible to e-graph explosion. Rule augmentation, therefore, improves reachability while making the expanded rewrite space harder to exploit in practice. EGGMIND addresses this with schedule construction and simplification in EqSatL.

End-to-End Results. We compare EGGMIND against the original and augmented rulesets evaluated at two distinct EqSat iterations ($i = 4$ and $i = 5$), where EGGMIND operates over the same

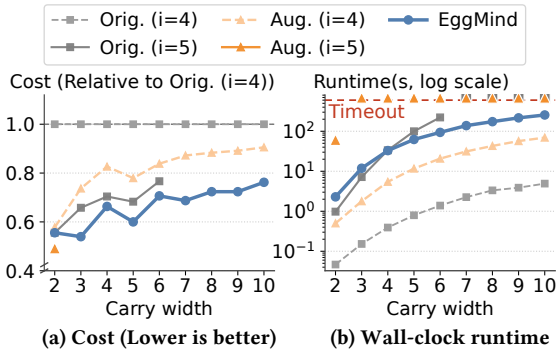


Figure 9: EqMap carry-benchmark comparison.

augmented rewrite space. This allows us to differentiate the gains from search budget, rewrite-space reachability, and strategic organization. Figure 9 shows that EGGMIND achieves the best quality-efficiency trade-off. Increasing the budget for the original ruleset ($i = 4 \rightarrow 5$) improves cost but causes four large cases to exceed the 600 s timeout. Augmented Rules at $i = 4$ improve reachability, but at $i = 5$ expose the inherent limits of unguided search. The e-graph expansion here becomes so explosive that only the smallest test cases finish within 600 s, and reach a peak sampled RSS of approximately 4 GB. Thus, rule augmentation alone is insufficient, and enlarging the matching space makes brute-force search even less attractive. By contrast, EGGMIND converts the same augmented space into a stable scheduled optimization process, achieving the lowest cost on 8/9 benchmarks (with unguided Augmented Rules ($i = 5$) winning only on the smallest case), reducing cost by 33.76% and 8.75% compared to Original Rules with $i = 4$ and $i = 5$, respectively, while maintaining stable evaluator wall time (2.3–255.4 s). It also reduces peak sampled RSS by 51.94% compared to the unguided Augmented Rules ($i = 5$). These results underscore the synergy between rule and strategy synthesis: the former expands reachability, while the latter determines whether that expanded space is practically exploitable.