VRIJE
UNIVERSITEIT
BRUSSEL

USING MACHINE LEARNING TO OPTIMIZE THE WORKLIST ALGORITHM OF STATIC ANALYSES

Bachelor's Thesis

Aïko Schürmann

2025 - 2026

Promotor: prof. dr. Coen De Roover
Advisors: Bram Vandenbogaerde, Noah Van Es, Sarah Verbelen

sciences and bioengineering sciences

Abstract

Static program analysis universally relies on worklist algorithms to schedule the evaluation of program components, making the execution sequence a primary factor in the computational cost required to reach a least fixed point. In the context of modular frameworks analyzing higher-order functional languages, traditional deterministic heuristics suffer from rigid trade-offs: dependency-driven structural sorting frequently triggers highly inefficient "interleaving" cycles in recursive data structures, while naive queues like First-In-First-Out (FIFO) bypass these cycles through argument batching but ignore semantic dependencies altogether. Consequently, no single static scheduling policy is universally optimal.

This thesis proposes a dynamic, data-driven approach that frames worklist component selection as a supervised Learning-to-Rank (LTR) problem. To overcome the delayed and sparse rewards inherent to static analysis, we introduce a novel target formulation based on "Lattice Progression" (a quantifiable proxy for mathematical progression) combined with multi-step simulated lookahead. By training a Gradient Boosted Machine (XGBoost) on a 19-dimensional feature vector capturing both static graph topology and dynamic runtime state, the scheduler learns to adaptively balance structural momentum with strategic workload batching.

Empirical evaluation across a diverse suite of Scheme benchmarks demonstrates that the ML-guided scheduler successfully avoids local optima and generalizes to less training-similar programs. Compared to a robust FIFO baseline, the top-performing lookahead model reduced total algorithmic iterations by 35.8% (weighted). While the high-dimensional feature extraction currently introduces wall-clock overhead, these results provide empirical evidence that context-aware machine learning models can reduce the iteration cost of worklist scheduling on the evaluated benchmarks, offering a foundation for future adaptive static analyzers.

Keywords: *Static Analysis, Abstract Interpretation, Worklist Algorithm, Machine Learning, Learning-to-Rank, Scheduling Heuristics, Scheme.*

Contents

1 Introduction **3**
1.1 Context and Motivation . . . 3
1.2 Problem Statement . . . 4
1.3 Research Questions . . . 5
1.4 Proposed Solution . . . 6
1.5 Contributions . . . 6
1.6 Thesis Outline . . . 7

2 Background **8**
2.1 Principles of Abstract Interpretation . . . 8
2.1.1 Lattices and Abstract Domains . . . 8
2.1.2 Monotone Constraints and Fixed Points . . . 9
2.2 The ModF Analysis Framework . . . 9
2.2.1 The global store . . . 10
2.2.2 The Effect-Driven Worklist Algorithm . . . 10
2.3 Worklist Scheduling Heuristics . . . 11
2.3.1 Naive Approaches . . . 11
2.3.2 Structural Approaches . . . 12
2.4 The Interleaving Problem . . . 12
2.5 Machine Learning for Scheduling . . . 13
2.5.1 The Learning-to-Rank Paradigm . . . 13
2.5.2 Gradient Boosted Machines (GBMs) . . . 14
2.5.3 The Feature Space . . . 14

3 Formulating the Learning Target **16**
3.1 Establishing the Optimization Potential (The Landscape) . . . 16
3.1.1 The State Space of the Analysis Process . . . 16
3.1.2 Distributional Validation and Log-Normality . . . 17
3.1.3 Uncertainty Estimation via Non-Parametric Bootstrapping . . . 18
3.1.4 Structural Sensitivity Analysis (The T-Test) . . . 18
3.1.5 The Sampling Budget . . . 20
3.2 Approach 1: Global Trajectory Mimicry (Empirical Oracle) . . . 21
3.2.1 Trajectory Search and the Empirical Oracle . . . 21

3.2.2 Target Formulation: Reciprocal Rank Decay . . . 23
3.2.3 Flaws and Limitations of Imitation Learning . . . 23
3.3 Approach 2: Local Greedy Progression (Lattice Target) . . . 24
3.3.1 Quantifying Lattice Progression and Discovery . . . 24
3.3.2 Mitigating Myopia via Simulation Strategies . . . 26
3.3.3 Strategic Divergence: Local vs. Global Optimality . . . 27
3.3.4 Target Superiority and Selection . . . 28

4 Machine Learning Pipeline **30**
4.1 Feature Engineering . . . 30
4.1.1 Temporal and Progress Metrics . . . 30
4.1.2 Static and Syntactical Features . . . 31
4.1.3 Dependency Graph Topology and Centrality . . . 32
4.2 Data Generation and Deterministic Replay . . . 33
4.3 Data Preprocessing and Balancing . . . 33
4.3.1 Relative Scaling and Log-Transformation . . . 34
4.3.2 Ranking Objective and NDCG Evaluation . . . 34
4.3.3 Query Grouping and Data Leakage Prevention . . . 35

5 Evaluation and Discussion **36**
5.1 Experimental Setup . . . 36
5.2 RQ1: Scheduling Landscape and Optimization Potential . . . 37
5.3 RQ2: Impact of Target Formulation . . . 37
5.4 RQ3: Algorithmic Performance and Generalization . . . 38
5.4.1 Generalization Across Training Similarity Splits . . . 39
5.4.2 Ranking Metrics and Accuracy . . . 41
5.5 RQ4: Feature Impact and Interpretability . . . 42
5.6 RQ5: Practical Overhead and Wall-Clock Trade-off . . . 45
5.7 Discussion . . . 46
5.8 Threats to Validity and Limitations . . . 47

6 Related Work **50**
6.1 Worklist Scheduling and State Exploration . . . 50
6.2 Machine Learning in Static Analysis . . . 51
6.3 Summary . . . 51

7 Conclusion and Future Work **52**
7.1 Summary of Findings . . . 52
7.2 Future Directions . . . 53

Chapter 1

Introduction

1.1 Context and Motivation

Static program analysis is a fundamental technique in software engineering, used to verify program correctness and detect potential vulnerabilities without executing the source code. Because the exact runtime behavior of non-trivial programs cannot in general be predicted due to computability limits such as Rice's Theorem, static analyzers must trade absolute precision for guaranteed termination. One prominent technique to achieve this is abstract interpretation. Instead of executing a program concretely (exactly, as a standard interpreter would), this approach evaluates the program abstractly, operating over approximated abstract domains rather than exact concrete values. For example, an analyzer may track that a variable is an "integer" instead of the specific value "5". This yields a conservative over-approximation of program behavior and keeps the analysis computationally feasible while preserving mathematical soundness [4].
To address this complexity, frameworks such as ModF adopt a modular architecture [12], breaking the program into smaller, independently analyzed components that communicate through a shared abstract memory representation called the global store. This design improves scalability but introduces a key challenge: components must be incrementally discovered and repeatedly re-analyzed as shared state evolves. The engine coordinating this process is the worklist algorithm.
The worklist acts as a dynamic scheduler, maintaining a queue of components that currently require analysis or re-evaluation. As a component is processed, it may update the abstract values in the global store. If these updates change the abstract information stored at an address, any other components that depend on those specific memory addresses are automatically pushed back onto the worklist for re-analysis. This cycle of evaluation and dependency-triggering continues until the system reaches a steady state, known as a least fixed point.
The specific order in which components are selected from the worklist does not affect the final analysis result. Assuming abstract domains that enforce finite convergence (e.g., via finite-height lattices or widening operators [4]), the monotone equations converge to the same sound fixed point regardless of the specific processing order. However, the

schedule does affect the number of iterations required for convergence [9]. An unsuitable strategy may analyze components prematurely and cause many redundant re-analyses as dependent values continue to change. As a result, the worklist algorithm can be a major performance bottleneck in the framework. Optimizing it does not change the outcome of the analysis, but reduces the computational effort needed to reach that outcome.

1.2 Problem Statement

The efficiency of the worklist algorithm depends entirely on its scheduling heuristic. Historically, static analyzers have relied on rigid, predefined strategies to manage pending work, which can broadly be categorized into naive and structural approaches. Naive strategies, such as First-In-First-Out (FIFO) or Last-In-First-Out (LIFO), utilize basic data structures to queue components. FIFO, for instance, naturally enforces a breadth-first exploration by treating the worklist as a standard queue. While these methods introduce virtually no computational overhead, they remain agnostic to the program's underlying semantic structure and data flow [7].

To address the limitations of blind scheduling, structural or dependency-driven heuristics attempt to order the worklist using the relationships between components. By constructing a dependency graph and applying techniques such as topological sorting, often with Tarjan's algorithm to condense recursive cycles into Strongly Connected Components (SCCs), these heuristics aim to process components only after their prerequisites have stabilized. In theory, prioritizing callees over callers reduces redundant re-evaluations. By ensuring that a callee reaches a stable state before its return values are propagated upward, the analyzer prevents the caller from performing premature computations on incomplete data. This approach works well for programs with imperative updates and relatively straightforward call graphs, where data dependencies flow predictably [7].

However, this structural prioritization is less effective in higher-order functional languages that make heavy use of recursive data structures such as lists and closures. This issue appears as the "interleaving problem" [7]. When a structural heuristic strictly prioritizes a dependency, it shifts focus to the callee as soon as a single new input is discovered, rather than waiting for the caller to finish generating all inputs.

Consider a `map` function (the caller) recursively applying a closure `f` (the callee) to a list of elements. Under a structural heuristic, the analyzer evaluates `map`, which extracts the first list element, issues a call to `f`, and completes its current evaluation step. Because `f` is a newly discovered callee, the heuristic immediately selects it from the worklist next. The callee `f` processes this single element and writes its return value to the global store. Because this memory update changes a value that `map` depends on, `map` is automatically pushed back onto the worklist. Once re-selected, `map` extracts the second element and triggers `f` again, causing the heuristic to immediately prioritize `f` once more. This results in the analyzer rapidly "ping-ponging" between caller and callee for every individual element, preventing the analysis from aggregating information efficiently.

In scenarios dominated by the interleaving problem, the naive FIFO strategy can outperform more sophisticated structural heuristics. Because FIFO does not immediately

switch to newly discovered dependencies, it allows the caller to finish processing its current execution layer and accumulate a batch of arguments in the global store. When the callee is eventually selected from the queue, it can evaluate that batch of inputs together, which reduces the total number of context switches [7].
This contrast highlights the main problem addressed in this thesis: no single static heuristic is universally optimal. Structural heuristics perform well in imperative contexts, but they are less effective in recursive, data-heavy scenarios where FIFO's batching effect is beneficial. Achieving good convergence across diverse programs therefore requires a dynamic scheduling strategy that can adapt to the current analysis context.

1.3 Research Questions

To systematically address the limitations of static worklist heuristics and evaluate the viability of a machine-learning-driven alternative, this thesis is guided by the following primary research questions:

- **RQ1 (The Scheduling Landscape):** Do highly optimized scheduling trajectories exist within the state space of modular static analysis that significantly outperform robust deterministic heuristics like First-In-First-Out (FIFO)?
- **RQ2 (Impact of Target Formulation):** What is the impact of the training target formulation (global empirical trace mimicry versus local simulated Lattice Progression) on the scheduler's ability to discover iteration-reducing trajectories?
- **RQ3 (Algorithmic Performance & Generalization):** To what extent does the Learning-to-Rank (LTR) scheduler reduce the total number of worklist iterations required to reach a least fixed point compared to the standard FIFO baseline, and how well do these iteration reductions generalize to less training-similar programs?
- **RQ4 (Feature Impact and Interpretability):** What is the relative impact of dynamic runtime metrics versus static structural topology features on the scheduling decisions made by the learned policy?
- **RQ5 (Practical Overhead):** What is the real-world computational trade-off between the algorithmic iteration reductions achieved by the ML scheduler and the wall-clock overhead introduced by high-dimensional feature extraction?

These research questions also structure the remainder of the thesis. Chapter 3 addresses RQ1 by characterizing the statistical scheduling landscape, and investigates RQ2 by formally comparing the performance outcomes of the empirical Oracle and Lattice Progression targets. Chapter 5 then evaluates the integrated ML pipeline to answer the remaining questions, establishing its algorithmic performance and generalization (RQ3), interpreting the model's feature reliance (RQ4), and measuring the final wall-clock trade-offs (RQ5).

1.4 Proposed Solution

To bridge the gap between structure-agnostic naive queues and rigid dependency graphs, this thesis proposes a dynamic, data-driven scheduling approach. Instead of relying on a fixed heuristic, we frame worklist component selection as a machine learning problem, specifically Learning-to-Rank (LTR) [8].
Instead of using Reinforcement Learning, which is less attractive here because rewards are sparse and delayed and the action space changes with the worklist state [11], supervised LTR allows us to evaluate the worklist state directly at each step. By training a Gradient Boosted Machine (XGBoost), which is well suited to heterogeneous tabular features [2, 5], we can score every pending component against a diverse set of engineered features. These features capture both the static properties of the code (such as syntactic size and arity) and the dynamic, runtime state of the analysis (such as the number of pending updates, time spent waiting in the queue, and current visit counts).
By observing these indicators, the model learns to independently decide when it is beneficial to prioritize a callee to stabilize data flow, and when it is better to delay execution to allow for the batching of arguments. This effectively grants the analyzer the ability to adapt its scheduling policy dynamically, mitigating the interleaving problem while preserving the same least fixed point.

1.5 Contributions

The primary contributions of this thesis are as follows:

- **Statistical Characterization of the Scheduling Landscape:** We establish a rigorous mathematical foundation for the variability in worklist performance by showing that iteration costs are well approximated by a log-normal distribution for the studied benchmarks. Through large-scale random state-space exploration, we provide evidence for highly optimized execution trajectories that significantly outperform robust deterministic baselines like First-In-First-Out (FIFO).
- **Formulation of Multimodal Training Targets:** We propose and rigorously evaluate two distinct methodologies for defining an optimal scheduling decision:
  - *The Empirical Oracle:* A global, temporal approach leveraging Monte Carlo Tree Search (MCTS) to discover the shortest sampled path to convergence, framed as an imitation learning problem utilizing Reciprocal Rank Decay (RRD).
  - *The Lattice-Progression Target:* A local, semantic approach that assigns immediate and lookahead-simulated rewards derived from quantifiable mathematical progress within the abstract domain (Lattice Progression).
- **Contextual Feature Engineering for Static Analysis:** We developed a high-dimensional feature extraction pipeline capable of translating the complex, symbolic state of the analyzer into a numerical format suitable for machine learning.

This captures both static structural properties and dynamic runtime states, including dependency graph metrics like PageRank, component arity, and temporal queue wait times.

- **Implementation of an Automated Policy-Learning Pipeline:** We engineered an end-to-end framework integrating the effect-driven ModF static analyzer with a Gradient Boosted Machine (XGBoost) to execute Learning-to-Rank (LTR).
- **Evaluation of Specialization and Generalization:** We present a comprehensive evaluation of the learned scheduling policies across a diverse benchmark suite. We explicitly distinguish between the models' capacity to reduce iterations on training-similar versus less training-similar programs.

1.6 Thesis Outline

The remainder of this thesis is structured as follows:

- **Chapter 2: Background** reviews the foundational concepts required to understand the proposed approach. This includes the principles of abstract interpretation, the modular architecture of the ModF framework, traditional worklist scheduling heuristics, and a brief overview of Learning-to-Rank (LTR).
- **Chapter 3: Formulating the Learning Target** establishes empirical evidence for highly optimized scheduling trajectories within the state space. It then details the design of our machine learning targets, contrasting a global imitation learning approach (the Empirical Oracle) with a localized, semantic progression target (Lattice Progression).
- **Chapter 4: Machine Learning Pipeline** discusses the end-to-end engineering and integration of the predictive model. This chapter details the extraction of high-dimensional state features, the deterministic trace-and-replay architecture used to overcome data-generation bottlenecks, and the underlying model architecture.
- **Chapter 5: Evaluation and Discussion** presents a comprehensive empirical analysis of the learned scheduling policies. It benchmarks the model's performance against standard deterministic strategies, evaluates its capacity to generalize to less training-similar programs, and investigates feature importance to understand the model's decision-making process.
- **Chapter 6: Related Work** contextualizes this research within the broader landscape of static analysis optimization, dynamic scheduling algorithms, and the integration of machine learning into software engineering tools.
- **Chapter 7: Conclusion and Future Work** summarizes the primary findings and contributions of the thesis and proposes potential avenues for further research and architectural refinement.

Chapter 2

Background

2.1 Principles of Abstract Interpretation

The core motivation for abstract interpretation arises from fundamental limits in computability theory. According to Rice's theorem, deciding any non-trivial semantic property of a program is generally impossible. To address this undecidability, abstract interpretation trades absolute precision for guaranteed termination. Instead of tracking exact runtime values, it evaluates the program over an abstract domain, for instance, approximating a numerical variable as a bounded interval (e.g., $[0, 10]$) or a mathematical sign (positive/negative) rather than tracking its exact concrete state. This approach yields a conservative over-approximation of program behavior, ensuring the analysis remains computationally feasible while maintaining mathematical soundness [4].

2.1.1 Lattices and Abstract Domains

To formalize this approximation and bridge the gap between executing actual source code and mathematical reasoning, the properties of the program's variables are mapped onto a complete lattice. This algebraic structure allows the analyzer to systematically track, compare, and merge the possible values a variable might hold at any given program point. A complete lattice is typically denoted as $\mathcal{L} = \langle D, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ [10]:

- D represents the set of abstract states (e.g., the collection of all possible mathematical intervals a numerical variable could evaluate to at runtime).
- $\sqsubseteq$ is the partial order denoting relative precision. The relation $x \sqsubseteq y$ implies that x is more precise (i.e., it encompasses fewer concrete runtime values) than y.
- $\bot$ (bottom) represents the empty set of possible concrete values. Mathematically, it is the unique least element in the domain such that $\forall x \in D : \bot \sqsubseteq x$. Technically, it does not represent unreachable code itself, but rather the *result* of evaluating unreachable code. Because execution never reaches that program point, there are no concrete values produced, yielding $\bot$.

- $\top$ (top) represents a complete lack of precision, or "any possible value" a variable could hold. Mathematically, it is the unique greatest element such that $\forall x \in D : x \sqsubseteq \top$.
- $\sqcup$ (join, or least upper bound) merges information from converging control-flow paths in the code (such as the convergence point after an `if-else` block). Mathematically, $z = x \sqcup y$ is the least element such that $x \sqsubseteq z$ and $y \sqsubseteq z$; for any other upper bound w, $z \sqsubseteq w$. It yields the least element that safely over-approximates both operands. For example, in interval analysis: $[0, 5] \sqcup [3, 10] = [0, 10]$.
- $\sqcap$ (meet, or greatest lower bound) computes the intersection of states, which is utilized to refine information (e.g., filtering out impossible values within the specific branches of a conditional check). Mathematically, $z = x \sqcap y$ is the greatest element such that $z \sqsubseteq x$ and $z \sqsubseteq y$; for any other lower bound w, $w \sqsubseteq z$. For example: $[0, 5] \sqcap [3, 10] = [3, 5]$.

2.1.2 Monotone Constraints and Fixed Points

The execution semantics of the program are encoded as a set of transfer functions. For any given program component u, a transfer function $f_u : D \to D$ models how the component's instructions modify the abstract state. Monotonicity is the key condition that makes fixed-point iteration well-defined:

$$x \sqsubseteq y \implies f_u(x) \sqsubseteq f_u(y) \tag{2.1}$$

Monotonicity ensures that as the analyzer accumulates abstract information, it only moves upwards in the lattice, generally representing a less precise but more inclusive over-approximation, and never invalidates prior safe assumptions. A least fixed point (LFP) exists for monotone functions over complete lattices. In practice, the termination of the computation of this least fixed point additionally depends on using finite-height domains, widening, or another mechanism that prevents infinite ascending chains [4].

2.2 The ModF Analysis Framework

Traditional static analyzers often rely on a complete, pre-computed Control Flow Graph (CFG) of the entire application. This monolithic approach is less suitable for higher-order functional languages such as Scheme. Because functions are first-class values that can be passed as arguments, the specific function invoked at a given call site is often only known dynamically at evaluation time. Consequently, the call graph cannot be fully known in advance and must instead be discovered incrementally. To address this complexity and improve scalability, frameworks such as ModF use a modular architecture [12]. Instead of treating the whole program as a single, large component, ModF partitions the execution into multiple isolated components. A component serves as a conservative over-approximation for a set of runtime calls to a specific function. To distinguish between different invocations of the same function, ModF pairs the target

function with an abstract *context*. Consequently, multiple dynamic calls to a function are only merged into a single shared component if they occur within the exact same context. Because these components are cheaper to analyze and are evaluated in isolation, they communicate indirectly through a shared abstract memory representation, called the global store.

2.2.1 The global store

Because ModF evaluates components in isolation, they cannot communicate through direct value passing or registers. Instead, communication is orchestrated through a shared data structure known as the global store. The global store serves as an abstract representation of the program's memory heap and stack, mapping abstract addresses to their corresponding values in the abstract domain D:

$$\sigma : \text{Addr} \to D \tag{2.2}$$

When a caller invokes a callee, it writes the abstract argument values to specific addresses in this store, which the callee subsequently reads.

2.2.2 The Effect-Driven Worklist Algorithm

Unlike standard algorithms that propagate data directly along static CFG edges, ModF utilizes an effect-driven worklist. Analyzing a component does not immediately trigger its successors; rather, it emits "effects" that the framework monitors to schedule future work.
When a component u is processed, it can emit the following effects:

- **Read Effect ($R(\alpha)$):** If component u reads an address α from the global store, the framework registers a dependency edge ($\alpha \to u$).
- **Write Effect ($W(\alpha, v)$):** If component u writes a computed value v to address α, the store updates its state using the lattice join operator: $\sigma(\alpha) \leftarrow \sigma(\alpha) \sqcup v$. Crucially, if this update changes the abstract value at α, all components that previously read from α are automatically enqueued back onto the worklist for re-analysis.
- **Call Effect ($C(f)$):** If a call to a new function f is encountered, a corresponding component c_f is instantiated and added to the worklist.

Algorithm 1 illustrates how these effects drive the core execution loop. Because the call graph is not known in advance, the analyzer dynamically maps dependencies via Read effects, and triggers re-evaluations purely through lattice-state changes via Write effects.

Algorithm 1 The Effect-Driven Worklist Algorithm

```
1: W ← {c_main}                          ▷ Initialize worklist with the entry point
2: σ ← ∅                                              ▷ Initialize global store
3: Deps ← ∅              ▷ Initialize dependency map (Address → Components)
4: while W is not empty do
5:     c ← select and remove a component from W
6:     Effects ← evaluate(c, σ)
7:     for each e ∈ Effects do
8:         if e = Read(α) then
9:             Deps(α) ← Deps(α) ∪ {c}
10:        else if e = Write(α, v) then
11:            v_new ← σ(α) ⊔ v
12:            if v_new ≠ σ(α) then
13:                σ(α) ← v_new
14:                W ← W ∪ Deps(α)          ▷ Re-enqueue dependent components
15:            end if
16:        else if e = Call(f) then
17:            W ← W ∪ {c_f}
18:        end if
19:    end for
20: end while
```

This effect-driven architecture exposes the critical vulnerability of worklist scheduling on line 5 of Algorithm 1. A suboptimal heuristic may repeatedly prioritize a dependent component before the component providing its required data has stabilized, leading to substantial performance overheads and interleaving cycles.

2.3 Worklist Scheduling Heuristics

The efficiency of a modular static analysis is heavily dependent on its worklist scheduling heuristic, which dictates the order in which pending components are selected for evaluation. Historically, frameworks have relied on either naive data structures or rigid structural algorithms to manage this queue.

2.3.1 Naive Approaches

The simplest iteration strategies rely on basic data structures that remain agnostic to the underlying semantic structure or data flow of the program.

- **Random:** This heuristic selects an arbitrary component from the worklist at each step. While it introduces no computational overhead, its lack of predictability and failure to exploit program structure make it highly inefficient for complex analyses.

- **FIFO (First-In, First-Out):** By treating the worklist as a standard queue, FIFO enforces a breadth-first exploration order. It ensures that components added earlier are processed first, which tends to advance the analysis of the whole program uniformly.
- **LIFO (Last-In, First-Out):** Utilizing a stack, LIFO enforces a depth-first exploration. While intuitively appealing because it prioritizes the most recently discovered work, such as the immediate dependents of the current component, empirical evaluations often demonstrate that it performs worse than FIFO in modular analyses [7].

2.3.2 Structural Approaches

To address the limitations of blind scheduling, structural heuristics attempt to align the processing order with the flow of data dependencies. The objective is to process a component only after the prerequisites it depends on have stabilized, thereby preventing redundant re-evaluations.
This is typically achieved through dependency-driven topological sorting [7]. The heuristic constructs a graph where nodes represent components and edges represent dependencies (read, write, or call effects). Because static analysis dependencies often form cycles, especially in the presence of recursion, the algorithm applies Tarjan's algorithm to condense Strongly Connected Components (SCCs) into single nodes, forming a Directed Acyclic Graph (DAG). A topological sort then dictates the processing order, generally prioritizing components with fewer dependencies (callees) over those that depend on them (callers).
While topologically sorting this graph can theoretically minimize re-analyses, it introduces significant computational overhead. Because the dependency graph changes dynamically with every iteration, repeatedly recalculating the topological sort has complexity $O(V + E)$ per recalculation. This often negates the speed gains achieved by a better ordering [7].

2.4 The Interleaving Problem

While dependency-driven heuristics aim to minimize redundant computations by strictly prioritizing callees, prior research has demonstrated that this strategy frequently breaks down in higher-order functional languages that heavily utilize recursive data structures, such as lists and closures [7]. This breakdown manifests as the "interleaving problem".
When a dependency-driven heuristic strictly prioritizes a callee, it shifts focus to the callee as soon as a new dependency is discovered, rather than waiting for the caller to generate all potential inputs. In scenarios where a caller recursively iterates through a data structure and generates unique abstract inputs (e.g., individual `cons` cells uniquely identified by their allocation site), the following inefficient cycle emerges:

1. **Caller Execution:** The caller generates a unique argument (e.g., the first list element) and emits a call effect to invoke the callee.
2. **Heuristic Selection:** Because the structural heuristic strongly prioritizes dependencies, it immediately selects the newly discovered callee from the worklist for the very next iteration.
3. **Callee Execution:** The callee processes this single new argument and updates its return value in the global store.
4. **Re-triggering:** The update to the return value alters the abstract memory that the caller depends on, triggering a re-analysis. The caller is automatically re-enqueued, selected, generates the *next* unique argument, and invokes the callee again.

This results in the analysis bouncing rapidly between caller and callee for every single element in the data structure, preventing the analysis from aggregating information efficiently.
As Kolozyan observed, in these specific scenarios, the naive FIFO strategy substantially outperforms sophisticated structural heuristics [7]. Because FIFO does not immediately prioritize the dependency, it allows the caller to continue executing, accumulating a large set of argument values in the global store. When the callee is eventually selected from the back of the queue, it processes this accumulated "batch" of arguments in a single pass, substantially reducing the total number of context switches and analysis iterations.

2.5 Machine Learning for Scheduling

The contrasting performance between structural sorting and FIFO reveals that no single static heuristic is universally optimal. Achieving optimal performance requires a dynamic scheduling strategy capable of adapting to the current analysis context. To bridge this gap, we frame worklist component selection as a machine learning problem.
While one could theoretically model this as a Reinforcement Learning (RL) problem where an agent selects an action and receives a reward [11], the reward signal in static analysis is extremely sparse, appearing only at the very end of the analysis when convergence is achieved. Furthermore, the action space (the pending components) changes dynamically at each step. Instead, we frame the problem as a supervised *Learning-to-Rank* task.

2.5.1 The Learning-to-Rank Paradigm

Learning-to-Rank (LTR) is a class of supervised machine learning techniques originally developed for Information Retrieval (IR) systems, such as web search engines [8]. In a traditional search engine, the model receives a "query" and must sort a list of "documents" so that the most relevant documents appear at the top.
We map the worklist scheduling problem directly onto this IR paradigm:

- **The Query:** The current state of the static analyzer at a specific scheduling step.
- **The Documents:** The set of candidate components currently waiting in the pending worklist.
- **The Relevance Score:** A quantifiable measure of how much analytical progress evaluating a specific component will yield (discussed in detail in Chapter 3).

Rather than predicting an absolute numerical value (regression) or categorizing a component (classification), the LTR model is trained specifically to output an optimal *relative ordering* of the worklist. By selecting the component ranked highest by the model, the analyzer dynamically chooses the most profitable execution step.

2.5.2 Gradient Boosted Machines (GBMs)

To act as the ranking model, we utilize a Gradient Boosted Machine, specifically the XGBoost implementation [2]. GBMs are highly effective for tabular data with heterogeneous features [5]. The architecture relies on two core machine learning concepts: Decision Trees and Boosting.

Decision Trees: The fundamental building block of the model is a decision tree. A decision tree makes predictions by passing a data point through a series of binary splits based on feature thresholds (e.g., "Has this component been waiting in the queue for more than 10 steps?" or "Is its structural in-degree greater than 5?"). Because they split the data space orthogonally, decision trees are robust against outliers and naturally capture non-linear relationships between variables without requiring complex mathematical transformations.

Gradient Boosting: A single decision tree is often prone to overfitting or lacks predictive power. Gradient Boosting overcomes this by training an *ensemble* of multiple shallow trees sequentially. Instead of training all trees independently, each new tree is trained to predict the *residual errors* (the mistakes) made by the combination of all previous trees. In the context of LTR, the loss function being minimized by this gradient descent process is a pairwise ranking loss. The model learns to penalize situations where a low-relevance component is scored higher than a high-relevance component.

2.5.3 The Feature Space

For the decision trees to accurately rank the candidates, the complex, symbolic state of the static analyzer must be translated into a numerical format. At every scheduling step, we extract a feature vector for every pending component. These engineered features (detailed extensively in Chapter 4) capture three distinct aspects of the analysis:

- **Static Characteristics:** The intrinsic syntactic complexity of the code, such as syntactic size and function arity [7].
- **Dynamic State:** The runtime behavior of the analyzer, including how many times a component has been visited, how long it has been waiting in the queue, and the volume of incoming memory updates.

- **Graph Topology:** The component's position within the dynamically unfolding dependency graph, measured through standard network metrics such as in-degree, out-degree, and PageRank [7].

By observing these indicators and processing them through the gradient boosted ensemble, we aim to train a model that learns to independently decide when it is beneficial to prioritize a callee (to resolve dependencies) and when it is better to delay execution (to allow for the batching of arguments).

Chapter 3

Formulating the Learning Target

3.1 Establishing the Optimization Potential (The Landscape)

While previous chapters established that worklist scheduling fundamentally alters the flow of an analysis, it remains an open question whether highly optimized, "shortcut" trajectories actually exist within a program's state space, and whether they can significantly outperform a robust deterministic baseline like FIFO. Before attempting to train a machine learning model to predict a high-quality schedule, we must empirically establish that these fast trajectories exist and are worth finding (RQ1).

3.1.1 The State Space of the Analysis Process

Before evaluating this potential, it is necessary to formally define the "state space" of the static analysis process. In the context of a modular worklist algorithm, the execution is not a single linear progression, but rather a traversal through a vast, combinatorial landscape of possible scheduling decisions.
We define a single analysis state as a tuple $S = \langle \sigma, W, \Delta \rangle$, where:

- σ represents the current state of the global store (the abstract memory mapping addresses to lattice values).
- W represents the current worklist (the set of pending components ready for evaluation).
- Δ represents the dynamically discovered dependency graph (mapping memory addresses to the components that read them).

The analysis begins in an initial state $S_0 = \langle \emptyset, \{c_{main}\}, \emptyset \rangle$. At any given state S_i, the available "action space" for the scheduler is defined by the components currently waiting in the worklist (W). Selecting a component $c \in W$ triggers an evaluation step. As the component computes, its resulting read, write, and call effects deterministically update the store and worklist, transitioning the analyzer to a new state S_{i+1}. This iterative

process continues until the worklist is entirely empty ($W = \emptyset$), landing the analyzer in a terminal state S_∞ that represents the least fixed point.
The overall state space of the scheduling problem is therefore the Directed Acyclic Graph (DAG) of all valid transitions from S_0 to S_∞. A specific scheduling strategy—whether a static heuristic like FIFO or a trained machine learning model—produces a single trajectory $\tau = (S_0, c_0, S_1, c_1, \ldots, S_\infty)$ through this landscape.
Because a worklist can frequently contain dozens or hundreds of components simultaneously, the branching factor at each step is massive. The core optimization problem of this thesis can thus be formally defined: finding the specific trajectory τ through this state space that minimizes the total number of transitions (iterations) required to reach S_∞.

3.1.2 Distributional Validation and Log-Normality

Before comparing the performance of random trajectories to our deterministic baselines, we must first determine the underlying mathematical shape of the resulting iteration distributions. Because scheduling performance is strictly bounded at zero and often highly right-skewed by a few exceptionally poor trajectories, standard arithmetic means and thresholds (such as $\mu \pm 3\sigma$) can be deeply misleading or yield physically impossible negative iteration counts. Establishing whether the data follows a log-normal distribution is therefore a critical first step, as it formally justifies the use of logarithmic transformations to ensure robust, outlier-resistant statistical analysis later on.
To formalize this, we define the computational cost of a worklist algorithm as a random variable $\mathcal{X}$, representing the total iterations required to reach convergence. We hypothesize that $\mathcal{X}$ is reasonably approximated by a log-normal distribution. To evaluate this, we apply a logarithmic transformation $Z = \ln(\mathcal{X})$ to a sample of $N = 500$ independent random schedules generated for each benchmark.
The transformed data Z was subjected to Kolmogorov-Smirnov and Shapiro-Wilk tests, evaluating the null hypothesis H_{0_dist}: *The transformed data is normally distributed.* Testing at $\alpha = 0.05$ failed to reject normality for most non-trivial benchmarks. While failing to reject a null hypothesis is not a mathematical proof of strict normality, the context of the sample size is critical. With small sample sizes, these tests lack statistical power, making it trivially easy to fail to reject H_{0_dist} simply due to a lack of evidence. Because our large sample ($N = 500$) provides high sensitivity to distributional deviations, the failure to reject H_{0_dist} under these strict conditions provides strong empirical justification for using the log-normal approximation.
Thus, for a given set of observed iteration counts $X = \{x_1, \ldots, x_n\}$, we estimate the log-space parameters using the unbiased sample mean and standard deviation:

$$\hat{\mu}_{log} = \frac{1}{n} \sum_{i=1}^{n} \ln(x_i) \tag{3.1}$$

$$\hat{\sigma}_{log} = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (\ln(x_i) - \hat{\mu}_{log})^2} \tag{3.2}$$

3.1.3 Uncertainty Estimation via Non-Parametric Bootstrapping

Once the mathematical shape of the landscape is established, our ultimate objective is to calculate exactly how many random runs (N_{req}) are required to confidently locate a highly optimized trajectory for ML training (detailed in Section 3.1.5). To guarantee that this sampling budget is sufficiently large, we must base it on a strictly conservative, worst-case estimate of the distribution's spread. If we rely on an overly optimistic (wide) standard deviation, we risk overestimating the probability of finding a fast trajectory, leading to a sampling budget that is dangerously small.

Because of the high kurtosis inherent in program analysis costs, standard error approximations can be unreliable. To establish a statistically cautious, lower-bound estimate for the standard deviation, we employ Non-Parametric Bootstrapping.

From the original log-transformed dataset Z, we generate $B = 5000$ bootstrap samples $Z^{*(1)}, \ldots, Z^{*(B)}$ by resampling with replacement. By calculating the sample standard deviation $\hat{\sigma}_{log}^{*(b)}$ for each set, we derive an empirical distribution of the variance estimator. To obtain our cautious spread estimate, we extract a "Safety Margin" σ_{safe} based on the lower bound of the 95% confidence interval:

$$\sigma_{safe} = \hat{Q}_{0.025}\left(\{\hat{\sigma}_{log}^{*(1)}, \ldots, \hat{\sigma}_{log}^{*(B)}\}\right) \tag{3.3}$$

where $\hat{Q}_{0.025}$ represents the 2.5^{th} percentile. Using this lower bound yields an intentionally narrower fitted spread. By assuming this "thinner" tail during our later calculations, we force a conservative underestimate of the success probability (p_{tail}), which safely inflates the required sampling budget (N_{req}) to ensure we do not terminate our search prematurely.

3.1.4 Structural Sensitivity Analysis (The T-Test)

With the shape of the landscape established, we can objectively evaluate whether random trajectories can outperform our baseline. As established in Chapter 2, sophisticated structural heuristics can suffer catastrophic performance degradation on recursive, higher-order Scheme programs due to the interleaving problem [7]. Consequently, the naive FIFO strategy serves as the most robust, high-performing deterministic baseline for this dataset.

We quantify the performance of the random state-space exploration against this baseline using a One-Sample T-Test. This test evaluates the null hypothesis H_{0_perf}: *The expected cost of a random schedule is equal to the cost of the standard FIFO baseline ($\mu_{log} = \ln(L_{FIFO})$).*

By calculating the p-value for statistical significance ($\alpha = 0.05$) and observing the sign of the t-statistic (which subtracts the FIFO baseline from the random distribution), we can assess whether random trajectories are, on average, faster or slower than FIFO for each benchmark.

Benchmark	FIFO Steps	Random Mean	t-statistic	t-test p	N_{req}	Performance Bias
boyer.scm	$1,421$	$1,303$	-30.927	< 0.001	4803	Random is Faster
mceval.scm	$1,395$	$1,416$	3.637	< 0.001	4670	FIFO is Faster
church.scm	129	137	17.868	< 0.001	4549	FIFO is Faster
SICP-compiler	$1,925$	$1,941$	1.047	0.295	4464	Statistically Tied
four-in-a-row.scm	237	232	-32.028	< 0.001	4401	Random is Faster
regex.scm	181	177	-8.167	< 0.001	4330	Random is Faster
quasiquoting.scm*	53	54	4.972	< 0.001	4297	FIFO is Faster
grid.scm*	63	63	-3.718	< 0.001	4048	Random is Faster
work.scm*	61	59	-31.414	< 0.001	4035	Random is Faster

Table 3.1: Statistical landscape of benchmark programs. To provide an intuitive comparison, performance is contextualized using raw iteration counts (FIFO baseline vs. the average of the random explorations). The sampling budget N_{req} is derived from the underlying log-normal spread parameters and represents the estimated random runs required to observe one highly optimized trajectory with 95% confidence. Benchmarks marked with (*) exhibit slight deviations from strict log-normality under separate tests ($p < 0.05$) but are retained to prevent selection bias.

As shown in Table 3.1, analyzing the random walks yields three distinct performance profiles:

- **Random is Faster ($t < 0, p < 0.05$):** In programs such as `boyer.scm` and `regex.scm`, random exploration frequently uncovers schedules that substantially outperform FIFO. This provides statistical evidence that highly optimized paths through the state space exist, and that the rigid FIFO heuristic often fails to exploit them.
- **FIFO is Faster ($t > 0, p < 0.05$):** In programs like `mceval.scm` and `church.scm`, the average random trajectory is significantly slower. These programs rely heavily on the structural locality preserved by FIFO's breadth-first batching.
- **Statistically Tied ($p \geq 0.05$):** In complex programs like `SICP-compiler`, the difference between average random exploration and FIFO is not statistically distinguishable at $\alpha = 0.05$ in this sample. This suggests that FIFO provides little measurable advantage over random exploration in this context, although it does not establish formal non-inferiority.

This supports our central premise: very fast trajectories exist that can substantially outperform static heuristics, but relying on FIFO can miss them. Because no single static algorithm can reliably capture these fast paths across all topologies, a dynamic, data-driven approach is well justified.
To visually ground these statistical profiles, Figure 3.1 illustrates the empirical distribution of iterations for the `mceval.scm` benchmark. The density plot clearly exhibits the right-skewed, heavy-tailed shape characteristic of the scheduling landscape. While most random trajectories cluster around a central peak, a long tail extends far to the right,

representing exceptionally poor scheduling paths that drastically inflate the iteration count.

In this specific program, the rigid FIFO baseline (1,395 steps) sits slightly to the left of the random mean (1,416 steps). This visualizes exactly why the t-test categorizes this benchmark as a scenario where FIFO outpaces average random exploration, while simultaneously highlighting the vast spread of alternative trajectories that exist within the state space.

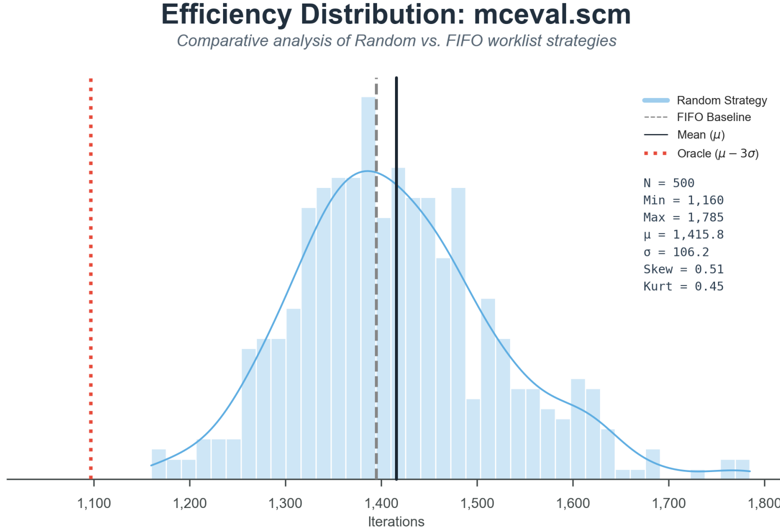

Figure 3.1: Empirical distribution of required iterations for the `mceval.scm` benchmark across random scheduling trajectories. The pronounced right-skewness illustrates the heavy-tailed nature of worklist performance. The relative placement of the standard FIFO baseline and the random mean visually grounds the statistical analysis.

3.1.5 The Sampling Budget

The statistical framework allows us to estimate N_{req}, the number of random samples required to observe at least one highly optimized trajectory with 95% confidence. We explicitly define a highly optimized trajectory as one falling three sample standard deviations ($3\hat{\sigma}_{log}$) below the empirical mean. We denote this target threshold as T_{log}:

$$T_{log} = \hat{\mu}_{log} - 3\hat{\sigma}_{log} \tag{3.4}$$

To calculate N_{req}, we model the random search as a sequence of independent Bernoulli trials. Let p_{tail} denote the probability of a single random schedule reaching this target threshold. To ensure our budget is mathematically conservative, we evaluate this

probability under the worst-case distribution model defined by the lower-bound safety margin (σ_{safe}) derived from our bootstrap analysis. Using the Cumulative Distribution Function (CDF) of the standard normal distribution (Φ), this probability is:

$$p_{tail} = \Phi\left(\frac{T_{log} - \hat{\mu}_{log}}{\sigma_{safe}}\right) \tag{3.5}$$

Because the safety margin is smaller than the sample standard deviation ($\sigma_{safe} < \hat{\sigma}_{log}$), the resulting Z-score evaluated here is strictly more negative than -3. This conservatively shrinks the expected tail probability, thereby safely inflating the estimated number of trials required.

Under this conservative model, the probability of failing to observe the event after N independent trials is $(1 - p_{tail})^N$. To achieve a target confidence level $C = 0.95$ of observing at least one such highly optimized trajectory, we bound the failure probability:

$$1 - (1 - p_{tail})^{N_{req}} \geq C \tag{3.6}$$

Solving for N_{req} yields the exact sampling budget formulation:

$$N_{req} = \left\lceil \frac{\ln(1 - C)}{\ln(1 - p_{tail})} \right\rceil \tag{3.7}$$

Applying this formulation illustrates the large size of the state space. For a benchmark like `boyer.scm`, isolating a single highly optimized trajectory requires an estimated minimum of $N = 4,803$ independent runs.

This mathematical reality establishes a critical baseline for our machine learning pipeline. Before an adaptive model can be trained to navigate this space, we must first extract the best sampled paths to serve as empirical training labels. The magnitude of N_{req} dictates that generating this initial dataset cannot be trivial; it demands a large computational search to overcome the statistical scarcity of highly optimized schedules.

3.2 Approach 1: Global Trajectory Mimicry (Empirical Oracle)

Having established the existence of optimized execution paths, our initial approach aimed to train a machine learning model to mimic them globally. We framed this as a supervised Learning-to-Rank problem in which the objective was to predict the sequence of component selections that minimizes the total iteration count needed to reach the fixed point.

3.2.1 Trajectory Search and the Empirical Oracle

To generate the necessary training data, we conducted a large-scale stochastic search of the scheduling state space. While our statistical framework in Section 3.1 demonstrated that capturing a generic fast trajectory (3σ below the mean) requires 4,803 random runs

for the boyer benchmark, training an imitation learning model benefits from the shortest path found by the available search budget, not just a generic fast run.

For each benchmark program in our training dataset, the search was systematically constrained by a strict wall-clock time budget rather than a fixed iteration limit. Because complex programs (such as `SICP-compiler`) require significantly more computational time to complete a single execution than simpler programs, the total number of completed search iterations naturally varied. This time-bounded approach yielded between 50,000 and 200,000 independent analysis runs per benchmark. To reach the fast tail of the distribution within this allocated time frame, the search phase could not rely on pure random walks; instead, it heavily utilized Monte Carlo Tree Search (MCTS) [1] to actively bias the exploration toward highly optimized paths.

MCTS is a heuristic search algorithm commonly used in artificial intelligence and decision processes. Rather than exploring the state space completely blindly, MCTS incrementally builds a search tree by mathematically balancing *exploration* (sampling previously unvisited scheduling choices) with *exploitation* (focusing the search budget on branches that have historically yielded fast iteration counts). By dynamically learning which paths through the worklist landscape are most promising, MCTS can drill down into the optimized tail of the distribution much faster than pure random sampling.

Instead of aggregating multiple fast trajectories, we retained only the single shortest sampled path from these extensive searches. This sequence of decisions was designated as the *empirical Oracle trace* for that program.

Program	**FIFO**	**Best Trace**	**Improvement**	**Deviation (σ)**
`church.scm`	129	77	40.3%	-8.92σ
`regex.scm`	181	112	38.1%	-7.75σ
`SICP-compiler`	1,925	1,248	35.2%	-5.27σ
`boyer.scm`	1,421	1,007	29.1%	-4.19σ
`mceval.scm`	1,395	1,024	26.6%	-4.66σ
`grid.scm`	63	50	20.6%	-4.11σ
`quasiquoting.scm`	53	45	15.1%	-4.33σ
`four-in-a-row.scm`	237	205	13.5%	-8.13σ
`work.scm`	61	56	8.2%	-3.03σ

Table 3.2: Comparison of total iterations between the standard FIFO heuristic and the best sampled empirical Oracle trace discovered via large-scale MCTS. The Deviation column quantifies how many standard deviations the Oracle trace falls below the expected random mean in the log-normal landscape.

As shown in Table 3.2, this best sampled path yielded substantial reductions in iterations. For complex benchmarks such as `SICP-compiler`, the best sampled trajectory required 1,248 iterations, more than 35% fewer analysis steps than the standard FIFO approach. Under our log-normal model, 1,248 iterations represents an event 5.27 stan-

dard deviations below the mean. For a standard normal distribution, the probability of a single random walk reaching this extreme threshold is approximately 6.82×10^{-8} (roughly 1 in 14.6 million). Even if we assume the maximum time budget of 200,000 pure random runs, the cumulative probability of observing at least one such highly optimized trace remains a mere 1.35%. This mathematical reality demonstrates exactly how unlikely it is to discover optimal trajectories via unguided exploration, strongly justifying the use of MCTS to actively navigate the state space and generate a high-quality training target.

3.2.2 Target Formulation: Reciprocal Rank Decay

In this approach, the target label Y_{RRD} is completely agnostic to semantic analytical progress; it is purely temporal. We assigned a numerical target score, $Y_{RRD}(u)$, to every candidate component u present in the pending worklist W at any given step t based entirely on when it appeared in the empirical Oracle trace.
We employed a Reciprocal Rank Decay formulation [3]. Let $\Delta(u)$ represent the "delay", namely the number of discrete scheduling steps that occur in the future empirical Oracle sequence before component u is selected. The target score is defined as:

$$Y_{RRD}(u) = \frac{1}{\Delta(u) + 1} \tag{3.8}$$

Under this formulation, the component chosen immediately by the empirical Oracle receives a score of 1.0, the next best option receives 0.5, and the score decays non-linearly for deferred components.

3.2.3 Flaws and Limitations of Imitation Learning

Although this approach could identify optimized paths, treating scheduling as a strict imitation learning problem based on a single temporal trace introduced several theoretical and practical limitations:

1. **Data Generation Bottleneck:** Executing 50,000 runs per training program simply to find one highly efficient trace is computationally explosive. There is also no guarantee of optimality.

2. **Compounding Errors (Covariate Shift):** Because the model is trained exclusively on states derived from the empirical Oracle trajectory, it has no knowledge of how to recover from mistakes. If the ML model makes a single suboptimal prediction at runtime and deviates from the oracle, it lands in a worklist state it has never seen during training. This leads to cascading failures where the model's performance rapidly degrades.

3. **Spurious Strictness and Tie-Breaking:** In many analysis states, the order of independent components (e.g., processing A before B, or B before A) may have no impact on the final iteration count. However, the empirical Oracle trace arbitrarily

selects one order. The target formulation aggressively penalizes the model for choosing B before A, forcing the learning algorithm to waste capacity by modeling meaningless tiebreaker noise rather than meaningful structural dependencies.

4. **Lack of Semantic Progress:** Reciprocal Rank Decay relies entirely on the temporal trace rather than actual state progression. It provides no mechanism for the model to learn *why* a component is useful, only that the empirical Oracle happened to pick it at that moment.

3.3 Approach 2: Local Greedy Progression (Lattice Target)

To resolve the data generation bottleneck and provide the learning algorithm with a dense, semantically meaningful reward signal, we abandoned the global temporal trace. Instead, we shifted the formulation toward local, immediate progression. The question becomes: *"Which component provides the strongest local proxy for progress toward convergence at the current step?"*

In the framework of abstract interpretation, one observable proxy for analytical progress is movement up the mathematical lattice. When a component is analyzed, it evaluates its internal semantics and potentially updates the abstract values in the global store. If this update results in a higher, more inclusive abstract state, the analysis has made measurable progress in the monotone iteration, although this is a proxy for convergence rather than a direct distance to the final fixed point.

3.3.1 Quantifying Lattice Progression and Discovery

We redefine our training target, $Y_{LP}(u)$, in terms of a quantifiable "Lattice Progression." For a given worklist state W, we simulate the execution of a pending component $u \in W$ and measure the resulting marginal change to the global store.

To formalize this, we first define the absolute progression of a global store snapshot σ. Mathematically, the store is defined as a total function $\sigma : \text{Addr} \to D$. This means that any previously unallocated address does not fall outside the domain, but rather inherently maps to the bottom element ($\bot$). To compute the total analytical progress, we evaluate the finite subset of actively allocated addresses—specifically those where the lattice state has strictly ascended above bottom ($\sigma(\alpha) \sqsupset \bot$)—using a lattice-specific scalar progress function $v(\sigma(\alpha))$.

This progress relies on tracking an element's structural position within the complete lattice $\mathcal{L} = \langle D, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$. We mathematically define the height function $\mathcal{H}(a, b)$ as the maximum length k of a strictly ascending chain $a = y_0 \sqsubset y_1 \sqsubset \cdots \sqsubset y_k = b$ between two elements in the domain. Using this concept, we formally define the two core metrics of lattice state:

- $level(x) = \mathcal{H}(\bot, x)$: The maximum number of structural updates required to reach x from the bottom element $\bot$, the least lattice element representing no observed

concrete values in the may-analysis ordering.

- $distanceToTop(x) = \mathcal{H}(x, \top)$: The maximum number of remaining updates required to push x to the top element (fully generalized state).

For finite lattices, the progress proxy $v_{finite}(x)$ is calculated as the linear ratio of the current lattice level against the total maximum path length through x:

$$v_{finite}(x) = \frac{level(x)}{level(x) + distanceToTop(x)} \tag{3.9}$$

Crucially, to guarantee analysis termination, all abstract domains employed in this framework theoretically possess a finite height. However, for certain complex structural domains (such as sets of abstract addresses or closures), dynamically calculating the exact *distanceToTop* is computationally prohibitive or practically undefined. In the framework's implementation, this unknown upper bound is often artificially represented by `Int.MaxValue`. For these specific lattices, we bypass the unknown distance parameter by applying a logarithmic dampening function strictly to the current level:

$$v_{dampened}(x) = 1.0 - \left(\frac{1}{1.0 + \ln(1.0 + level(x))}\right) \tag{3.10}$$

This score measures saturation toward $\top$ and is used as a proxy for convergence progress, not as a direct distance to the final fixed point.
The absolute progression proxy of a store snapshot is the finite sum of these values:

$$Progression(\sigma) = \sum_{\alpha \in \mathrm{Addr} | \sigma(\alpha) \sqsupset \bot} v(\sigma(\alpha)) \tag{3.11}$$

The immediate Lattice Progression $Gain(u)$ is strictly defined as the marginal difference between the post-update store and the pre-update store caused by analyzing component u:

$$Gain(u) = \max(0, Progression(\sigma^u_{post}) - Progression(\sigma_{pre})) \tag{3.12}$$

Theoretically, because abstract interpretation strictly relies on monotone join operations ($\sqcup$), the store's progression can never decrease ($\sigma_{post} \sqsupseteq \sigma_{pre}$). The $\max(0, \dots)$ operation serves purely as a defensive computational bound to sanitize infinitesimal negative values caused by floating-point rounding errors during the calculation of the non-linear progression proxies, ensuring a strictly non-negative reward signal for the machine learning model. When a simulated execution allocates a new abstract address, that address enters the active subset ($\sigma(\alpha) \sqsupset \bot$) and structurally contributes to the post-state progression.
Furthermore, semantic progress is not limited to just updating existing store values. Discovering entirely new, unanalyzed code paths is a critical aspect of static analysis. Therefore, the immediate reward function for selecting component u is calculated as the sum of its immediate Lattice Progression plus a Discovery Bonus:

$$Reward(u) = Gain(u) + (\lambda \times |NewComponents|) \tag{3.13}$$

where $NewComponents$ denotes the set of previously unseen components discovered during the simulated execution of u, and $\lambda = 0.1$ is a static weight assigned to newly discovered components.
The discovery bonus is intentionally kept small relative to the final per-state target scale. The combined rewards are later min-max normalized across all candidates in the current worklist state before training, so λ controls the within-state preference for newly discovered components but does not change the final label range. Because this weighting remains a heuristic design choice, we revisit it as a construct-validity limitation in Chapter 5.

3.3.2 Mitigating Myopia via Simulation Strategies

A purely greedy one-step lookahead approach, which always selects the component with the highest immediate $Reward(u)$, risks trapping the analyzer in local optima. The model may exhaust immediate gains while neglecting components that enable important structural updates later.
To mitigate this myopia while preserving the density of the reward signal, we expanded the local target formulation by simulating future trajectories. To calculate the target score for a candidate component u, we simulate its execution and then a sequence of $H - 1$ subsequent steps, where H denotes the lookahead horizon.
We evaluated two distinct simulation strategies to traverse this lookahead horizon, balancing computational overhead and structural awareness:

- **Greedy Lookahead ($K = 1$):** To evaluate a candidate, we simulate selecting it and then, for the remaining $H-1$ lookahead steps, greedily select the single component with the maximum immediate Lattice Progression. This strategy, evaluated at a deep horizon of $H = 25$, represents a linear "steepest ascent" simulation and tests whether component u unlocks a direct sequence of high-reward updates.
- **Beam Search Lookahead ($K > 1$):** To avoid the rigid, single-path nature of a purely greedy lookahead, we also employed a breadth-aware simulation. The beam search explores multiple branching lookahead trajectories simultaneously, maintaining the top K parallel states at each depth up to horizon H. We evaluated configurations balancing simulation depth and width, specifically $H = 15$ with $K = 3$, and $H = 10$ with $K = 5$.

For all lookahead variants, let $P^*(u) = (c_0, c_1, \ldots, c_{H-1})$ represent the sequence of components selected along the highest-yielding simulated trajectory, strictly anchored by the initial candidate choice $c_0 = u$. The cumulative target score $Y_{LP}(u)$ assigned to this initial candidate is the discounted sum of the immediate rewards evaluated along this specific path:

$$Y_{LP}(u) = \frac{1}{H} \sum_{d=0}^{H-1} \gamma^d \cdot Reward(c_d) \tag{3.14}$$

where $Reward(c_d)$ represents the immediate reward—comprising the marginal Lattice Progression and discovery bonus defined in Section 3.3.1—yielded strictly by executing component c_d on the intermediate simulated global store at depth d. The parameter γ acts as a discount factor (e.g., $\gamma = 0.9$) that prioritizes immediate gains while still incorporating longer-term structural progression. Finally, the scores calculated across all candidates in the current worklist are min-max normalized between 0.0 and 1.0 whenever the raw scores are not all equal; equal-score states are treated as ties. This provides the gradient-boosted machine with a scaled regression target.

3.3.3 Strategic Divergence: Local vs. Global Optimality

A critical methodological question is whether the local Lattice Progression target merely approximates the global empirical Oracle, or if it constitutes a fundamentally different scheduling strategy. To evaluate this, we deterministically replayed the empirical Oracle trace for each benchmark. At every individual scheduling step, we paused the analysis to calculate the potential Lattice Progression for all candidate components currently in the pending worklist. We then measured the *alignment percentage*, defined as the frequency with which the component actually selected by the empirical Oracle matched the candidate maximizing the target score $Y_{LP}(u)$.

Program	**Alignment %**
`work.scm`	96.43%
`quasiquoting.scm`	77.78%
`church.scm`	50.65%
`four-in-a-row.scm`	49.27%
`grid.scm`	48.00%
`regex.scm`	34.82%
`boyer.scm`	14.03%
`mceval.scm`	9.08%
`SICP-compiler`	5.93%

Table 3.3: Alignment between the empirical Oracle choice and the maximum local Lattice Progression choice at each step of the replay trace. The values shown reflect the primary Beam Search configuration ($H = 15, K = 3, \gamma = 0.9, \lambda = 0.1$).

As Table 3.3 demonstrates, alignment is heavily dependent on program complexity. In structurally simple benchmarks like `work.scm`, the local progression choice and the best sampled trace overlap almost entirely (96.43%). However, in complex analyses like `SICP-compiler`, the alignment drops to just 5.93%. Globally, the Pearson correlation between the continuous Reciprocal Rank Decay target assigned by the Oracle (Y_{RRD}) and the Lattice Progression target score (Y_{LP}) across all decision points is extremely weak ($r = 0.0602$). This confirms that rather than capturing the same underlying sig-

nal, the two target formulations encode fundamentally distinct scheduling strategies. To understand where this divergence occurs, we expanded our analysis to track the contextual state of the analyzer during each disagreement.

- **Sensitivity to Worklist Complexity:** Strict Top-1 alignment is highly inversely correlated with the size of the pending worklist. When the worklist is "Tiny" (1–5 candidates), the strategies agree 54.49% of the time. However, when the worklist expands into "Large" bottleneck states (50+ candidates), binary alignment plummets to a mere 1.71%.
- **Temporal Degradation:** The strategies align most frequently during the early phase of the analysis (25.36% alignment in the first third of iterations), where initial discovery heavily dictates progress. As the global store populates and the dependency graph matures, alignment drops significantly, reaching 10.31% in the final third of the analysis.
- **The Oracle's Discrepancy:** When the two strategies disagree, the empirical Oracle routinely selects components that offer poor immediate progression. For example, during disagreements in `SICP-compiler`, the component chosen by the Oracle yielded an average normalized Lattice Progression of just 0.266 (compared to the 1.0 maximum available).

3.3.4 Target Superiority and Selection

To determine the optimal training formulation, we directly compared the execution trajectories generated by the purely statistical empirical Oracle against those generated by the Lattice Progression target (utilizing simulated semantic lookahead).

Program	FIFO Baseline	Empirical Oracle	Lattice Progression	Improvement (LP)
`SICP-compiler`	$1,925$	$1,248$	752	60.9%
`boyer.scm`	$1,421$	$1,007$	563	60.4%
`mceval.scm`	$1,395$	$1,024$	521	62.7%
`four-in-a-row.scm`	237	205	197	16.9%
`regex.scm`	181	112	123	32.0%
`church.scm`	129	77	81	37.2%
`grid.scm`	63	50	50	20.6%
`work.scm`	61	56	55	9.8%
`quasiquoting.scm`	53	45	48	9.4%

Table 3.4: Comparison of target superiority and selection metrics. The Lattice Progression targets reflect the lookahead simulation parameters $H = 25$ and $K = 1$. The results demonstrate how the structural awareness of the lookahead strategy compares against the purely empirical sequences generated by the Oracle.

As detailed in Table 3.4, the Lattice Progression target consistently discovered superior, more highly optimized routes through the state space for complex program topologies.

For instance, in `SICP-compiler` and `mceval.scm`, the Lattice-guided trace reduced the required iterations by roughly 60% compared to the FIFO baseline, substantially outperforming the empirical Oracle.
Because the Lattice Progression target provides both a denser, more learnable reward signal and a superior iteration efficiency, we exclusively utilize it as the target formulation to train our predictive model for the remainder of this thesis.

Chapter 4

Machine Learning Pipeline

4.1 Feature Engineering

The performance of the learning algorithm depends on its capacity to evaluate the structural significance and analytical progression of a pending component. To achieve this, we extract a comprehensive 19-dimensional feature vector Φ for every candidate in the worklist at each scheduling step. These 19 dimensions are derived from 17 distinct core metrics. Because two of these highly skewed dynamic metrics, Wait Time and Pending Updates, are supplied to the model in two mathematically distinct formats (both normalized and log-scaled), the original 17 metrics expand into the final 19-dimensional vector. These features are rigorously defined mathematically and categorized into temporal progression, structural topology, and graph-theoretical centrality.

4.1.1 Temporal and Progress Metrics

These features track a component's position relative to the analysis timeline and its mathematical progress within the abstract lattice.

- **Wait Time (`norm_wait`, `log_wait`):** Evaluates the potential for argument batching by measuring how long a component has been delayed in the queue.

$$Wait(c) = t_{current} - t_{enqueued}(c) \tag{4.1}$$

  where $t_{enqueued}$ is the discrete step count when the component was most recently triggered. Due to its heavy-tailed distribution, this single core metric accounts for two dimensions in the final feature vector: a relatively normalized version (`norm_wait`) and a logarithmically scaled version (`log_wait`).

- **Age (`norm_age`):** Tracks the total lifespan of the component within the dynamic analysis.

$$Age(c) = t_{current} - t_{discovery}(c) \tag{4.2}$$

  where $t_{discovery}$ represents the exact step the component was first instantiated.

- **Average Input Convergence (`norm_avg_input_levelToTop`):** Quantifies the stability of the data supplied to the component. In this context, the triggering inputs $Vals(c)$ are not strictly limited to formal function arguments; $Vals(c)$ encompasses the complete set of abstract values upon which component c possesses read dependencies (including accessed global variables and environment states).

  To measure the saturation of these inputs without redundancy, we directly apply the lattice-specific progress proxies (v_{finite} and $v_{dampened}$) formally established in Section 3.3.1. Let $v(x)$ denote the application of the appropriate proxy based on the structural domain of input x. The convergence proxy is then calculated as the mean saturation of all incoming dependencies:

$$AvgLevelToTop(c) = \frac{1}{|Vals(c)|} \sum_{x \in Vals(c)} v(x) \tag{4.3}$$

  If $Vals(c) = \emptyset$ (i.e., the component has no tracked dependencies), this feature defaults to 0.

- **Pending Updates (`norm_pending_updates, log_pending_updates`):** The number of distinct producer components that have triggered updates for this component and have not yet been ingested. As with Wait Time, this metric accounts for two dimensions in the final vector to capture both relative scale (`norm_pending_updates`) and absolute magnitude orders (`log_pending_updates`).

- **Visit Count (`norm_visits`):** The historical frequency of a component's evaluation, enabling the model to identify and appropriately penalize deep recursive cycles that exhibit slow convergence.

- **Delta Change (`norm_delta_change`):** The quantitative magnitude of incoming lattice state alterations received by this component, serving as a momentum indicator.

- **Selection Status (`was_selected`):** A binary flag indicating whether the component was the exact candidate chosen in the immediately preceding scheduling step, providing necessary context for consecutive execution chains.

4.1.2 Static and Syntactical Features

These features describe the fixed syntactic complexity and inherent interface of the analyzed component, entirely independent of its dynamic execution state within the worklist:

- **Component Size (`norm_size`):** A proxy for computational evaluation cost, calculated recursively over the sub-expressions $e \in Sub(exp)$:

$$Size(exp) = 1 + \sum_{e \in Sub(exp)} Size(e) \tag{4.4}$$

- **Arity (`norm_arity`):** The exact number of formal parameters expected by the component, indicating its potential data-dependency footprint.
- **Entry Point Flag (`is_main`):** A strict binary indicator denoting whether the component serves as the primary entry point (the main function) of the analyzed program.

4.1.3 Dependency Graph Topology and Centrality

To capture both the local execution context and pinpoint global bottlenecks, we calculate topological and centrality metrics over the dynamically unfolding dependency graph $G = (V, E)$.

All dependency-graph features use a single directed convention. An edge $p \to c$ is created when component p spawns component c or triggers an update in c. Edges therefore point from producer to consumer, or equivalently from caller to callee in the component graph. Outgoing edges represent downstream dependents that can be affected by the current component, while incoming edges represent prerequisite producers that provide input to the current component.

- **Average Neighbor Convergence (`norm_avg_neighbor_conv`):** Measures the execution readiness of a component based on the stability of its direct predecessor producers, $Pred(c) = \{p \mid p \to c\}$. Let $[\![p]\!]$ denote the evaluated abstract value produced by predecessor p. By applying the structural progress proxy $v(x)$ defined in Section 3.3.1 to these incoming values, we compute:

$$AvgNeighborConv(c) = \frac{1}{|Pred(c)|} \sum_{p \in Pred(c)} v([\![p]\!]) \tag{4.5}$$

  If $Pred(c) = \emptyset$, this average is set to 0.

- **Out-Degree (`norm_out_degree`):** The number of direct downstream dependents, or consumer components, triggered by this component through outgoing edges.
- **In-Degree (`norm_in_degree`):** The number of prerequisite producer components that provide input to this component through incoming edges.
- **DAG Depth (`norm_dag_depth`):** The maximum structural depth of the component from the entry point, calculated within the current directed acyclic graph representation of the dependencies.
- **PageRank (`norm_pagerank`):** Identifies major sink components and information hubs by allowing rank to flow into a component along incoming producer-to-consumer edges. Using the power iteration method with damping factor $d = 0.85$, N total components, and the set of predecessors $In(u) = \{v \mid v \to u\}$:

$$PR(u) = \frac{1-d}{N} + d \sum_{v \in In(u)} \frac{PR(v)}{OutDeg(v)} \tag{4.6}$$

- **Betweenness (`norm_betweenness`):** Quantifies the component's structural role as a bridge between disparate execution paths:

$$BC(v) = \sum_{\substack{s \neq v \neq t \\ \rho_{st} > 0}} \frac{\rho_{st}(v)}{\rho_{st}} \tag{4.7}$$

  where ρ_{st} is the total number of shortest paths from s to t, and $\rho_{st}(v)$ is the number of those paths that strictly pass through v. Disconnected pairs with $\rho_{st} = 0$ are excluded from the sum.

- **Transitive Dependents (`norm_transitive_deps`):** Represents the theoretical downstream impact of evaluating the component, defined as the number of other components reachable from c in the transitive closure E^+ of the outgoing dependent relation:

$$TD(c) = |\{v \in V \setminus \{c\} \mid (c, v) \in E^+\}| \tag{4.8}$$

4.2 Data Generation and Deterministic Replay

Extracting the complex, 19-dimensional feature vectors (Φ) detailed in the previous section—such as dynamically recalculating PageRank or scanning the global store for pending updates—is computationally expensive. Attempting to extract these features live during the thousands of exploratory runs required to find an optimized global trajectory (such as the 50,000 MCTS runs evaluated for the Empirical Oracle) would be computationally impractical and create a severe data generation bottleneck.
To solve this, we decoupled the execution search phase from the feature extraction phase via a trace-and-replay architecture. During the initial search phase, the analyzer operates with minimal overhead, tracking only standard execution metrics and disabling the ML feature extraction pipeline entirely.
Once a highly optimized target trajectory is identified, we extract its "trace"—a lightweight, ordered sequence of the unique component IDs selected at each step. The analyzer then enters a deterministic replay phase. By overriding the heuristic queue and forcing the worklist to strictly follow the recorded sequence, the analyzer safely pauses at every execution step. In this frozen state, it extracts the full 19-dimensional feature matrix for every candidate in the worklist and calculates the corresponding training targets.

4.3 Data Preprocessing and Balancing

Raw analysis metrics inherently exhibit extreme variance and heavy-tailed distributions; for instance, a component might wait for 10 steps in a trivial program but 10,000 steps in a complex interpreter. To ensure these values remain comparable across diverse benchmarks, the raw features (f_{raw}) are transformed through a strict preprocessing pipeline before model ingestion.

4.3.1 Relative Scaling and Log-Transformation

To prevent the model from memorizing absolute thresholds that would fail to generalize to larger unseen programs, we utilize *Relative Normalization.* All 17 core features are normalized relative to the maximum value currently present within the active worklist, yielding metrics prefixed with `norm_`:

$$f_{norm}(c) = \frac{f_{raw}(c)}{\max_{u \in W} f_{raw}(u) + \epsilon} \tag{4.9}$$

where ϵ is a small constant introduced to prevent division by zero. This process establishes the first 17 dimensions of our feature vector.
Simultaneously, highly skewed dynamic features specifically measuring time and volume (Wait Time and Pending Updates) exhibit a power-law distribution. To compress this range and provide the model with a sense of absolute magnitude without breaking generalization, we apply an additional logarithmic scaling transformation to just these two metrics. This yields two supplemental features prefixed with `log_`:

$$f_{log}(c) = \ln(1 + f_{raw}(c)) \tag{4.10}$$

By appending these two log-scaled metrics (`log_wait` and `log_pending_updates`) to the 17 relatively normalized metrics, we arrive at the final, 19-dimensional feature vector Φ supplied to the machine learning model.

4.3.2 Ranking Objective and NDCG Evaluation

To align our model with the task of scheduling component execution, we formulate the learning problem as a ranking objective. Rather than regressing toward a single absolute target, we optimize for the relative quality of candidates within a single worklist step, using the Normalized Discounted Cumulative Gain (NDCG) as our objective function.

Target Discretization

We define the relevance gain $g(u)$ of a candidate component u based on its simulated Lattice Progression $Y_{LP}(u)$. To facilitate the ranking objective, we discretize the continuous $[0, 1]$ target into $k = 10$ relevance levels using quantile binning:

$$g_{discrete}(u) = \lfloor \text{min-max}(Y_{LP}(u)) \cdot (k - 1) \rfloor \tag{4.11}$$

where $\lfloor \cdot \rfloor$ denotes the floor function. This transformation allows the model to prioritize relative quality across discrete tiers, effectively mitigating the impact of simulation noise while emphasizing the distinction between high-value and low-value scheduling paths.

NDCG Formulation

We utilize the NDCG metric to evaluate the quality of the ranked candidates. Given a set of candidates U at a specific decision point, we calculate the Discounted Cumulative

Gain (DCG) as:

$$DCG_p = \sum_{i=1}^{p} \frac{2^{g_i} - 1}{\log_2(i+1)} \tag{4.12}$$

where g_i is the relevance level of the component at rank i. The final score is normalized by the Ideal DCG (IDCG), obtained by sorting all candidates in U by their optimal relevance gain:

$$NDCG_p = \frac{DCG_p}{IDCG_p} \tag{4.13}$$

This formulation ensures that the model is penalized more severely if it misranks high-utility components near the top of the worklist, which is essential for maintaining the structural progression of the analyzer.

4.3.3 Query Grouping and Data Leakage Prevention

Because worklist states can grow exceptionally large, sometimes containing hundreds of competing candidates simultaneously, we treat each scheduling step as an isolated *query group* analogous to how search engines group documents for a specific query.
During cross-validation, we employ a *GroupShuffleSplit* strategy partitioned by these query groups. This guarantees that all candidate components competing within the same specific scheduling step are kept together in either the training or validation set, preserving the integrity of the pairwise ranking objective. Because the split is performed at the query level rather than the program level, longer-running benchmarks naturally contribute more queries to both sets. While this step-level split allows different temporal states from the same program to appear in both training and validation sets, the model's capacity to truly generalize to unfamiliar code is rigorously and independently evaluated in Chapter 5 using a separate suite of completely unseen benchmark programs.

Chapter 5

Evaluation and Discussion

This chapter answers the research questions explicitly. RQ1 and RQ2 are primarily supported by the scheduling-landscape and target-formulation experiments from Chapter 3, while RQ3–RQ5 evaluate the integrated Learning-to-Rank scheduler. Each section below restates the corresponding research question, summarizes the relevant evidence, and gives a direct answer.

5.1 Experimental Setup

To evaluate the effectiveness of the Machine Learning (ML) driven scheduler, we benchmarked its performance against the deterministic FIFO heuristic. Performance is strictly measured by the total number of worklist iterations required to reach a least fixed point. We evaluated four distinct model configurations, trained using different simulation strategies (defined by their lookahead horizon H and beam width B):

- **L25_B1 (Greedy Lattice Lookahead):** The champion configuration. It clones the analysis state, simulates a 25-step rollout, and uses beam width $B = 1$, making the rollout greedy after the first candidate choice.
- **L15_B3 (Balanced Beam):** A moderate search exploring a depth of 15 steps while maintaining the top 3 parallel trajectories.
- **L10_B5 (Broad Beam):** A shallow, wide search exploring a depth of 10 steps across 5 parallel trajectories.
- **L1_B1 (Purely Greedy):** A baseline configuration with no future rollout. This model is trained to maximize immediate Lattice Progression on the next step.

All configurations use the Lattice Progression target with discovery weight $\lambda = 0.1$ and discount factor $\gamma = 0.9$. Ties in normalized target scores are treated as equal-score states during training; at execution time, exact prediction ties fall back to the worklist's deterministic iteration order.

The evaluation suite consists of 32 diverse benchmark programs, all distinct from the programs used to train the model. We partition these evaluation programs by their similarity to the training set. The *training-similar* group ($N = 3$) contains distinct programs whose control-flow shape, recursion patterns, and dependency structure closely resemble a program observed during training. The *less training-similar* group ($N = 29$) contains the remaining distinct programs, which do not have such a close training-set counterpart. These labels describe similarity to the training data; they do not indicate train/test leakage.

5.2 RQ1: Scheduling Landscape and Optimization Potential

Research question. Do highly optimized scheduling trajectories exist within the state space of modular static analysis that significantly outperform robust deterministic heuristics like FIFO?

Analysis. Chapter 3 first characterizes the scheduling landscape through large-scale random state-space exploration. The statistical landscape in Table 3.1 shows that the relative strength of FIFO varies by benchmark: random trajectories are significantly faster than FIFO for some programs, FIFO is significantly faster for others, and at least one complex benchmark is statistically tied. This already indicates that FIFO is robust but not universally optimal. The empirical Oracle results in Table 3.2 then show that targeted search can discover much shorter sampled traces, with improvements ranging from 8.2% to 40.3% over FIFO on the training benchmarks.

Answer to RQ1. Yes. The explored state spaces contain highly optimized trajectories that can substantially outperform FIFO, but these trajectories are sparse and benchmark-dependent. This supports the central premise that scheduling is a meaningful optimization target: the final fixed point is unchanged, but the path taken to reach it can vary dramatically in cost.

5.3 RQ2: Impact of Target Formulation

Research question. What is the impact of the training target formulation (global empirical trace mimicry versus local simulated Lattice Progression) on the scheduler's ability to discover iteration-reducing trajectories?

Analysis. The global empirical Oracle target provides an intuitive imitation-learning baseline, but it is data-hungry and brittle: it depends on expensive search, records only one sampled trajectory, and penalizes harmless deviations from that trace. The alignment analysis in Table 3.3 shows that Oracle choices and local Lattice Progression choices diverge strongly on complex programs, indicating that the Lattice Progression

target is not merely copying the same policy. Table 3.4 further shows that Lattice Progression with semantic lookahead finds shorter trajectories than the empirical Oracle on the most complex benchmarks, reducing FIFO iterations by roughly 60% on `SICP-compiler`, `boyer.scm`, and `mceval.scm`.

Answer to RQ2. The target formulation has a decisive impact. Global trace mimicry is useful for demonstrating that fast paths exist, but it is too sparse and trajectory-specific to serve as the final learning target. Local simulated Lattice Progression provides a denser and more semantically meaningful signal, and the lookahead variant discovers stronger iteration-reducing schedules. This is why the final ML scheduler is trained against the simulated Lattice Progression target rather than the empirical Oracle trace.

5.4 RQ3: Algorithmic Performance and Generalization

Research question. To what extent does the LTR scheduler reduce the total number of worklist iterations required to reach a least fixed point compared to FIFO, and how well do these reductions generalize to less training-similar programs?

Analysis. The aggregated performance of the models across the entire 32-program benchmark suite is detailed in Table 5.1.

Config	N	W. Ratio	Geo Mean Ratio	Net Steps Saved	Wins	Ties	Losses
`L25_B1`	32	0.642	0.839	8,133	26	2	4
`L15_B3`	32	0.714	0.881	6,504	24	1	7
`L10_B5`	32	0.743	0.879	5,847	24	2	6
`L1_B1`	32	1.084	0.954	-1,917	20	2	10

Table 5.1: Global performance across all 32 benchmark programs. The Weighted Ratio and Geometric Mean Ratio indicate the proportion of steps taken relative to the FIFO baseline (lower is better). A "Win" indicates the ML model required fewer iterations than FIFO.

The weighted ratio in Table 5.1 is computed as $\sum_p I_{ML}(p)/\sum_p I_{FIFO}(p)$, so `L25_B1`'s value of 0.642 corresponds to a 35.8% weighted iteration reduction on this 32-program evaluation. The geometric mean ratio instead summarizes the typical multiplicative per-program effect and corresponds to a 16.1% reduction. These weighted, geometric, and unweighted summaries answer different questions and should not be compared as if they were the same statistic.

The data supports the theoretical risk of "myopia" outlined in Chapter 3. The purely greedy configuration (`L1_B1`), which selects components solely on the basis of their immediate Lattice Progression, performs worse than FIFO in the weighted aggregate, adding a net $1,917$ steps across the benchmark suite. This happens despite winning on many small benchmarks: its losses occur on sufficiently costly programs to dominate the weighted

total. By chasing immediate, localized mathematical progress, the greedy model can trap the analyzer in local optima.

In contrast, the inclusion of simulated lookahead substantially improves the outcome. The greedy lattice lookahead configuration (`L25_B1`) emerged as the most effective global strategy. By simulating 25 steps into the future on a cloned analysis state, the model learns to defer immediate gratification in favor of components that unlock substantial structural updates later in the execution. It achieved a geometric mean iteration ratio of 0.839 against the baseline, securing 26 wins and saving a net total of 8,133 analysis steps.

5.4.1 Generalization Across Training Similarity Splits

A central challenge in applying machine learning to program analysis is preventing the model from merely memorizing the abstract syntax trees of its training set. A viable scheduler must perform both on distinct programs that closely resemble a training benchmark and on distinct programs that are less similar to the training set. To evaluate this, we separate the 3 training-similar programs from the 29 less training-similar programs.

Config	N	W. Ratio	Geo Mean Ratio	Net Steps Saved	Wins	Ties	Losses
`L25_B1`	3	0.385	0.363	7,098	3	0	0
`L15_B3`	3	0.488	0.484	5,908	3	0	0
`L10_B5`	3	0.594	0.588	4,682	3	0	0
`L1_B1`	3	1.027	0.912	-316	1	0	2

Table 5.2: Model specialization performance on the 3 training-similar programs, which are distinct evaluation programs that closely resemble programs in the training set.

As demonstrated in Table 5.2, models equipped with lookahead exhibit a strong capacity for specialization. When analyzing training-similar programs, the `L25_B1` model substantially outperforms the baseline, achieving a geometric mean ratio of 0.363 (an average iteration reduction of nearly 64%). Meanwhile, the greedy `L1_B1` model fails to specialize, losing to FIFO on 2 out of 3 familiar programs, reaffirming that optimal scheduling requires multi-step structural awareness.

Config	N	W. Ratio	Geo Mean Ratio	Net Steps Saved	Wins	Ties	Losses
`L25_B1`	29	0.907	0.915	1,035	23	2	4
`L10_B5`	29	0.896	0.917	1,165	21	2	6
`L15_B3`	29	0.947	0.938	596	21	1	7
`L1_B1`	29	1.143	0.958	-1,601	19	2	8

Table 5.3: Model generalization performance on the 29 less training-similar programs.

Table 5.3 provides evidence that the predictive scheduler can generalize to less training-similar programs, provided it is trained with future state simulation. Across the 29 less

training-similar programs, the L25_B1 configuration maintained a strong win rate (23 wins, 2 ties, 4 losses) and a favorable geometric mean ratio of 0.915. While the magnitude of the step reduction naturally decreases when encountering unfamiliar programs, the lookahead models remain highly competitive and outperform the naive baseline in aggregate and on most individual benchmarks.

Program	**Similarity to Training**	**Baseline Steps**	**ML Steps**	**Steps Saved**	**Ratio**
`compiler.scm`	Similar	3,557	880	2,677	0.247
`sboyer.scm`	Similar	4,257	1,462	2,795	0.343
`nboyer.scm`	Similar	3,722	2,096	1,626	0.563
`peval.scm`	Less similar	2,280	1,406	874	0.617
`cat.scm`	Less similar	9	7	2	0.778
`sum.scm`	Less similar	10	8	2	0.800
`puzzle.scm`	Less similar	130	107	23	0.823
`slatex.scm`	Less similar	1,319	1,090	229	0.826
`ctak.scm`	Less similar	35	29	6	0.829
`array1.scm`	Less similar	53	44	9	0.830
`tak.scm`	Less similar	15	13	2	0.867
`sumloop.scm`	Less similar	31	27	4	0.871
`wc.scm`	Less similar	24	21	3	0.875
`lattice.scm`	Less similar	320	281	39	0.878
`destruc.scm`	Less similar	101	89	12	0.881
`string.scm`	Less similar	42	38	4	0.905
`trav1.scm`	Less similar	223	202	21	0.906
`nqueens.scm`	Less similar	44	40	4	0.909
`matrix.scm`	Less similar	936	862	74	0.921
`fibc.scm`	Less similar	28	26	2	0.929
`paraffins.scm`	Less similar	154	146	8	0.948
`diviter.scm`	Less similar	128	123	5	0.961
`primes.scm`	Less similar	69	67	2	0.971
`earley.scm`	Less similar	895	880	15	0.983
`deriv.scm`	Less similar	64	63	1	0.984
`triangl.scm`	Less similar	164	163	1	0.994
`graphs.scm`	Less similar	2	2	0	1.000
`tail.scm`	Less similar	53	53	0	1.000
`mazefun.scm`	Less similar	403	421	-18	1.045
`perm9.scm`	Less similar	82	86	-4	1.049
`scheme.scm`	Less similar	3,303	3,511	-208	1.063
`browse.scm`	Less similar	261	338	-77	1.295

Table 5.4: Detailed performance breakdown per program for the top-performing L25_B1 configuration, sorted by iteration ratio. The similarity-to-training column uses Similar for distinct programs close to the training set and Less similar for the remaining distinct programs.

5.4.2 Ranking Metrics and Accuracy

To understand the predictive quality of the learned scheduling policy independently of the final iteration counts, we evaluate the underlying Gradient Boosted Machine using standard Information Retrieval (IR) metrics. Because we framed component selection as a Learning-to-Rank (LTR) problem, we assess the model's ability to correctly sort the pending worklist candidates at any given step.

Table 5.5 presents the split evaluation results for our highest-performing model configuration (`L25_B1`). The dataset of extracted worklist states was partitioned into an 85% training set and a 15% validation set at the query-group level, matching the grouping strategy described in Chapter 4. This keeps all candidates from the same scheduling decision together in either the training or validation split, but it does not by itself constitute a program-level generalization test; that stronger cross-program evaluation is provided separately by the 32-program benchmark suite.

Metric	Training (85%)	Validation (15%)
NDCG	0.9637	0.9187
Top-1 Accuracy	68.00%	29.87%
Top-3 Accuracy	88.68%	63.84%
MRR	0.7789	0.4367

Table 5.5: Split evaluation metrics for the `L25_B1` Learning-to-Rank model. The metrics quantify the model's ability to correctly rank the worklist candidates compared to the target scores derived from the lookahead simulation.

Analysis of Ranking Quality (NDCG)

The primary optimization objective of the model is the Normalized Discounted Cumulative Gain (NDCG). NDCG evaluates the overall quality of the sorted worklist, heavily penalizing models that place high-value targets near the bottom of the queue. The model achieved an exceptional validation NDCG of 0.9187. This indicates that the feature extraction pipeline provides sufficient resolution for the XGBoost ensemble to reliably separate high-impact structural updates from low-value, isolated components, generalizing well to less training-similar programs.

Precision and the Generalization Gap

While the NDCG remains high across the split, the exact classification metrics exhibit a noticeable generalization gap. **Top-1 Accuracy** measures the frequency with which the model's absolute highest-ranked prediction exactly matched the highest-scoring candidate under the lookahead Lattice Progression target. On the validation set, the model selected this proxy-best candidate 29.87% of the time.

In a traditional classification task, a 30% accuracy might indicate poor performance. However, in the context of modular static analysis, worklists frequently contain dozens

or hundreds of pending components simultaneously. Furthermore, many analysis states exhibit broad "ties" where selecting component A or component B yields identical structural progression. Consequently, predicting the exact highest-scoring proxy target out of a large queue is an extremely strict constraint.
A more representative metric for practical scheduling is **Top-3 Accuracy**, which measures how often the highest-scoring lookahead-target component appeared within the model's top 3 predictions. On less training-similar validation states, the model achieves a Top-3 Accuracy of 63.84%.
This capability is further supported by the **Mean Reciprocal Rank (MRR)**. An MRR of 0.4367 on the validation set means that the reciprocal-rank average is comparable to placing the highest-scoring lookahead-target component around rank 2.3 in a harmonic sense. It should not be interpreted as the arithmetic mean rank of the best target.

Conclusion of Metric Performance

These metrics contextualize the iteration reductions observed in the RQ3 evaluation. The model does not need to act as a perfect, deterministic oracle to outperform static heuristics like FIFO. By consistently placing components scored as profitable by the lookahead target within the top ranks of the worklist (Top-3 Accuracy $>$ 63%) and maintaining a structurally sound overall queue (NDCG $>$ 0.91), the predictive scheduler often avoids costly interleaving cycles and efficiently drives the analysis toward the least fixed point.

Answer to RQ3. The LTR scheduler reduces iterations substantially when trained with simulated lookahead. Across all 32 evaluation programs, `L25_B1` reaches a weighted step ratio of 0.642, corresponding to a total iteration reduction of 35.8%, and a geometric mean ratio of 0.839. On the 29 less training-similar programs, it still achieves a favorable geometric mean ratio of 0.915 with 23 wins, 2 ties, and 4 losses. The purely greedy `L1_B1` model performs worse in the weighted aggregate, showing that lookahead is necessary to avoid myopic schedules.

5.5 RQ4: Feature Impact and Interpretability

Research question. What is the relative impact of dynamic runtime metrics versus static structural topology features on the scheduling decisions made by the learned policy?

Analysis. To thoroughly understand the underlying decision-making process of the learned scheduling policy, we utilized SHapley Additive exPlanations (SHAP). SHAP values provide a game-theoretic approach to interpreting machine learning models by quantifying the marginal contribution of each individual feature to the final prediction score. Figure 5.1 presents the SHAP summary beeswarm plot extracted from the best-performing configuration (`L25_B1`).

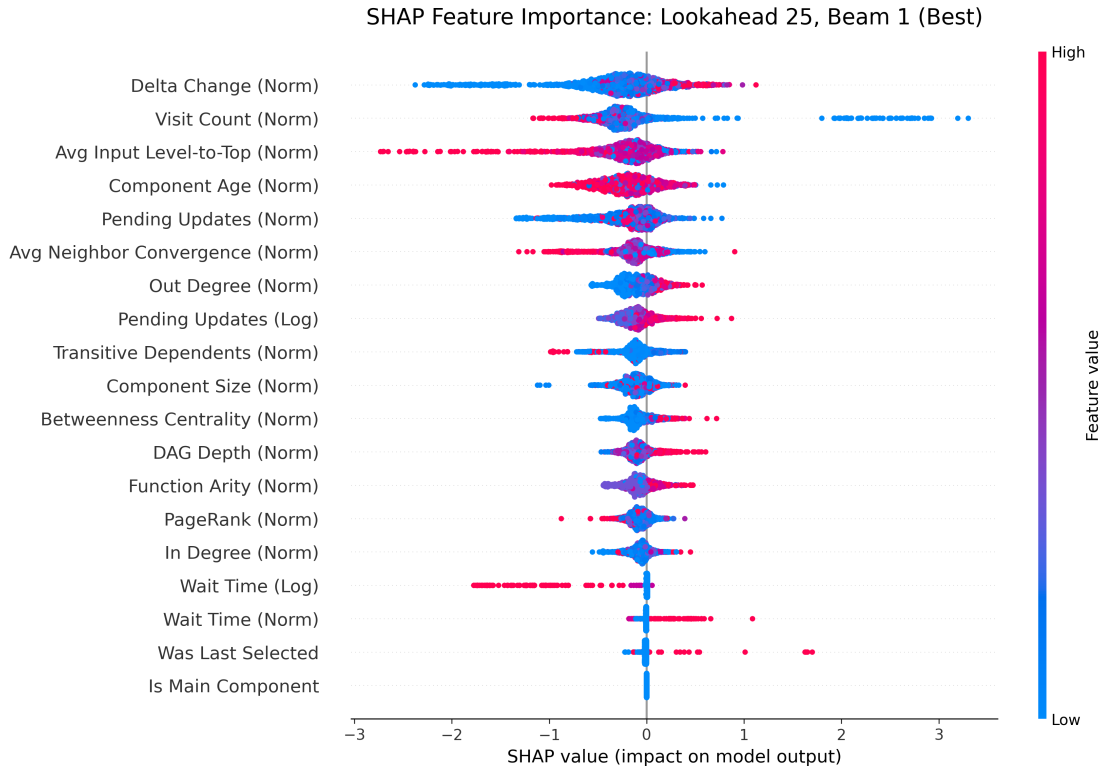

Figure 5.1: SHAP summary beeswarm plot for the L25_B1 model. Each point represents a single candidate component during a scheduling step. The color indicates the raw feature value (red for high, blue for low), and the horizontal axis represents the feature's impact on the model's ranking prediction (positive values increase priority).

The visualization provides several interpretive clues about how the predictive model optimizes the analysis trajectory:

Dynamic State Dictates Priority

The most striking observation from the SHAP analysis is that dynamic, state-aware metrics entirely dominate the top of the feature hierarchy. Static structural features derived from the dependency graph, such as **Betweenness Centrality**, **DAG Depth**, **Function Arity**, and **PageRank**, play only a secondary, supporting role in fine-tuning the lower ranks. This supports our core hypothesis from Chapter 2: rigid, static heuristics are fundamentally constrained because optimal scheduling requires continuously reacting to the live mathematical progression of the abstract interpretation.

Momentum and Cycle Avoidance

The two most universally impactful features governing the model's decisions are **Delta Change** and **Visit Count**, which act as opposing forces:

- **Delta Change (Norm):** This feature functions as a momentum indicator. High values (represented by the red cluster extending far to the right) strongly increase the component's prediction score. The model strongly prioritizes candidates that recently caused large shifts in the lattice state, choosing to ride the wave of active data flows to maximize immediate analytical progress.
- **Visit Count (Norm):** Conversely, high visit counts strongly penalize a component's ranking (evidenced by the dense red cluster pushed to the far left). This is consistent with the model learning to identify and break out of slow-converging recursive cycles, thereby mitigating the interleaving problem that traps standard structural heuristics.

Convergence and Workload Batching

The model's lookahead training also taught it to evaluate the readiness and accumulated workload of a candidate before execution:

- **Avg Input Level-to-Top (Norm):** Components with high values for this metric (represented by the red dots) are heavily penalized. In abstract interpretation, as values accumulate information, they ascend the lattice toward the top ($\top$), losing precision until they reach a fully converged, generalized state. By deprioritizing components whose prerequisite inputs are already close to $\top$, the model appears to encode a diminishing-returns pattern. It tends to avoid scheduling components that would mostly propagate highly generalized values, preferring instead to execute components whose inputs reside lower in the lattice and still possess the potential for meaningful structural refinement.
- **Pending Updates and Age:** High volumes of **Pending Updates** and elevated **Component Age** consistently push SHAP values to the right. This suggests that the model learned to exploit the batching effect. By occasionally delaying components and allowing their queues to fill up, the model can increase the effective Lattice Progression achieved per context switch.

Overall, the SHAP interpretation suggests that the `L25_B1` model does not blindly memorize structural graphs. Instead, its predictions appear to reflect a context-aware scheduling policy that balances momentum exploitation, workload batching, and strict cycle avoidance.

Answer to RQ4. Dynamic runtime metrics dominate the learned policy. The SHAP analysis indicates that features such as *Delta Change*, *Visit Count*, *Average Input Convergence*, *Pending Updates*, and *Age* have the strongest influence on ranking decisions, while static graph features such as PageRank, Betweenness Centrality, DAG depth, and arity play a secondary supporting role. The learned scheduler therefore appears to rely primarily on live analysis-state signals rather than memorizing static program structure.

5.6 RQ5: Practical Overhead and Wall-Clock Trade-off

Research question. What is the real-world computational trade-off between the algorithmic iteration reductions achieved by the ML scheduler and the wall-clock overhead introduced by high-dimensional feature extraction?

Analysis. While the predictive scheduler successfully reduces the total number of worklist iterations required to reach a fixed point, it is critical to distinguish between algorithmic efficiency (iterations) and practical system performance (wall-clock execution time). To evaluate this trade-off, we measured execution time and Time-Per-Iteration (TPI) for the `L25_B1` configuration on the 32-program Gambit benchmark set. For each program, we computed the iteration reduction $(1 - I_{ML}/I_{FIFO}) \times 100$ and wall-clock time reduction $(1 - T_{ML}/T_{FIFO}) \times 100$. We also group programs by FIFO iteration count: Small (< 100), Medium ($100 \leq$ steps < 1000), and Large (> 1000). Table 5.6 reports three complementary summaries: the weighted mean effect, the typical multiplicative effect, and arithmetic per-program averages.

Scope	**Summary**	N	**Iter. red.**	**Time red.**	**TPI cost**
All programs	Weighted mean	32	35.81%	−5.14%	1.64×
All programs	Geometric mean	32	16.13%	−195.01%	3.52×
All programs	Arithmetic mean	32	12.84%	−278.91%	4.24×
Small (< 100)	Arithmetic mean	15	9.36%	−290.93%	4.28×
Medium (100–1000)	Arithmetic mean	11	3.32%	−374.17%	5.09×
Large (> 1000)	Arithmetic mean	6	39.00%	−74.24%	2.60×
Large (> 1000)	Weighted mean	6	43.35%	0.58%	1.76×

Table 5.6: Iteration and timing summaries for `L25_B1` on the Gambit benchmark set. "Weighted mean" computes ratios after summing measurements first (e.g., $\sum_p I_{ML}(p)/\sum_p I_{FIFO}(p)$ and $\sum_p T_{ML}(p)/\sum_p T_{FIFO}(p)$), so larger baseline workloads receive more weight. Arithmetic and geometric means are computed over per-program ratios. Negative time reduction denotes slowdown relative to FIFO.

The data reveals a scaling-dependent trade-off. Across the full benchmark set, `L25_B1` saves 8,133 worklist steps and achieves a 35.81% weighted mean iteration reduction. However, after feature-extraction and inference overhead, the weighted mean wall-clock reduction is still −5.14%. The per-program arithmetic mean is more pessimistic: the average benchmark has a −278.91% time reduction because many small and medium programs pay the ML overhead without enough iteration savings to compensate. The large-program subset is the only case where the overhead is nearly amortized: the weighted mean iteration reduction reaches 43.35%, and the weighted mean wall-clock reduction improves slightly to 0.58%. Even there, the arithmetic mean time reduction remains negative (−74.24%), showing that the gains are concentrated in the largest workloads rather than uniformly distributed.

This overhead is primarily driven by the Time-Per-Iteration (TPI). In a standard FIFO queue, selecting the next component is an $O(1)$ operation. In the ML-driven framework, selecting the next component requires:

1. Iterating over every pending candidate in the worklist.
2. Extracting a 19-dimensional feature vector for each candidate, which involves calculating complex graph-theoretical metrics like PageRank and dynamically scanning the global store for pending updates.
3. Passing this feature matrix through the XGBoost ensemble to generate ranking scores.
4. Sorting the worklist based on these continuous predictions.

Its algorithmic value is clearest on the large benchmarks that dominate the real-world cost of static analysis, but deploying this specific pipeline broadly still requires systems-level optimization to reduce feature-extraction and inference overhead.

Answer to RQ5. The current scheduler achieves clear algorithmic savings but does not yet deliver broad wall-clock speedups. Across all programs, the 35.81% weighted mean iteration reduction is not enough to offset the ML overhead, resulting in a 5.14% weighted mean wall-clock slowdown relative to FIFO. On large programs, however, the 43.35% weighted mean iteration reduction is just enough to produce a small 0.58% weighted mean wall-clock improvement. This suggests that the algorithmic savings can amortize the overhead on sufficiently large workloads, but the current implementation still requires substantial systems-level optimization before it can provide reliable wall-clock speedups across the full benchmark suite.

5.7 Discussion

A combined analysis of the empirical evaluations, ranking metrics, and feature interpretability yields several cohesive insights into the mechanics and viability of machine-learning-driven static analysis scheduling. The results consistently support the hypothesis that dynamic, data-driven policies can successfully navigate the vast scheduling state space to find highly optimized execution trajectories.

The Necessity of "Patience" in Worklist Scheduling

The most prominent finding of this evaluation is the practical importance of simulated lookahead. The weighted-aggregate failure of the purely greedy configuration (`L1_B1`) demonstrates that analytical progress in static analysis is highly non-linear. Selecting a component that yields immediate, localized Lattice Progression can lead the analyzer into a local optimum, triggering for example the interleaving cycles discussed in Chapter 2. Conversely, the success of the `L25_B1` configuration shows that effective scheduling

requires a degree of "patience." The SHAP analysis is consistent with this interpretation because it shows that the model explicitly values *Wait Time*, *Component Age*, and *Pending Updates*. By simulating future states, the model learned that intentionally delaying certain components allows arguments to batch in the global store. This implicitly replicates and enhances the primary advantage of the FIFO queue, but applies it selectively based on the dynamic state rather than blindly to every component.

Dynamic Features over Structural Memorization

The robust performance of the predictive model on less training-similar programs (a geometric mean ratio of 0.915 across 29 programs) indicates that the model did not merely memorize the abstract syntax trees of its training data. The SHAP values suggest that the fitted model relies heavily on dynamic execution metrics (e.g., *Delta Change*, *Visit Count*, *Average Input Convergence*) rather than only on static graph topology. This provides evidence that the learned scheduler reacts to live analysis-state signals, such as recent state changes and slow-converging recursive loops, instead of depending solely on structural memorization.

The Generalization-Specialization Gap

While the model successfully generalizes, there remains a distinct performance gap between its execution on less training-similar programs and training-similar programs (where it achieved a striking 0.363 iteration ratio). This disparity highlights both the strength and the limitation of the current feature representation.
The relative scaling applied during data preprocessing appears to help the model transfer its learned bottleneck and batching patterns to less training-similar programs, securing wins against FIFO on most benchmarks in this split. However, reaching the ultra-fast "tail" of the log-normal landscape requires deep specialization. When the model encounters familiar recursive motifs and data-flow patterns, it leverages its underlying structural features (such as PageRank and DAG Depth) to find highly optimized shortcut trajectories. When presented with less training-similar architectures, it falls back on its generalized dynamic rules, yielding solid but less extreme iteration reductions.
Ultimately, these results show that selecting a good schedule is not a strict classification problem requiring a global shortest-path oracle. By maintaining a structurally sound queue (evidenced by an NDCG > 0.91) and consistently placing high-scoring lookahead-target components near the top of the worklist, the LTR framework successfully mitigates the rigid limitations of static heuristics.

5.8 Threats to Validity and Limitations

While the empirical evaluation provides strong evidence for ML-guided worklist scheduling, several methodological threats and practical limitations remain. We categorize these into construct, internal, external, and practical threats.

Construct Validity

Construct validity considers whether the experimental metrics genuinely measure the intended theoretical concepts.

- **The Lattice Progression Proxy:** The training target relies on simulated "Lattice Progression" (saturation toward $\top$) as a measure of analytical progress. However, moving upwards in the abstract lattice is a heuristic proxy for convergence. It does not mathematically equate to minimizing the remaining distance to the Least Fixed Point (LFP). The model may occasionally prioritize components that rapidly degrade precision rather than unlocking the critical structural updates required for global termination. In addition, the fixed discovery weight $\lambda = 0.1$ mixes a store-wide progression change with a count of newly discovered components; although per-state min-max normalization bounds the final labels, the relative influence of this bonus should be validated through sensitivity analysis.
- **Finite Lookahead Horizons:** The top-performing model (`L25_B1`) relies on a finite simulation horizon of 25 steps to overcome scheduling myopia. Consequently, the model is fundamentally blind to structural bottlenecks or batching opportunities that require more than 25 steps to resolve. It remains an approximation of optimal scheduling rather than a mathematically guaranteed shortest path.

Internal Validity

Internal validity concerns the causal relationships established during the experiment and the potential for confounding variables.

- **Tie-Breaking and Determinism:** In many worklist states, multiple components possess identical feature vectors or yield identical lookahead target scores. In these scenarios, the model predicts identical ranks, and the actual selection falls back on the underlying data structure's inherent iteration order. Because static analysis topologies are highly sensitive, different arbitrary tie-breaking outcomes could cause the analyzer to branch into different execution trajectories, introducing a degree of variance not entirely controlled by the ML policy.
- **Hyperparameter Selection:** The XGBoost hyperparameters and the specific lookahead simulation parameters (e.g., $H = 25, B = 1$) were selected based on iterative tuning. An exhaustive grid search across all possible beam widths and depths was computationally infeasible. Different configurations might yield different generalization gaps between training-similar and less training-similar programs.

External Validity

External validity addresses the extent to which these findings can be generalized beyond the specific context of this study.

- **Language and Paradigm Constraints:** The evaluation suite consists exclusively of higher-order functional Scheme programs. These programs do not provide a fully precise call graph in advance because higher-order calls are discovered through analysis, and they heavily utilize recursive data structures, which uniquely exacerbates the interleaving problem. It remains unverified whether this Learning-to-Rank scheduling approach would yield comparable iteration reductions in purely imperative or object-oriented languages (e.g., C or Java), where traditional, dependency-driven topological heuristics already perform reasonably well.
- **Analyzer Architecture:** The engineered features rely heavily on the effect-driven architecture of the ModF framework and its shared global store. Translating this predictive scheduler to a traditional data-flow analyzer would require a fundamental redesign of the feature extraction pipeline.

Chapter 6

Related Work

This thesis sits at the intersection of programming language theory and applied machine learning. To contextualize our contributions, this chapter reviews existing literature across two primary domains: traditional optimization strategies for static analysis worklists, and the emerging trend of applying machine learning to software analysis.

6.1 Worklist Scheduling and State Exploration

The impact of exploration order on the efficiency of static analysis has been a long-standing topic of research. As established by Cousot and Cousot [4], while the theoretical framework of abstract interpretation guarantees soundness and convergence under monotone conditions, it does not dictate the operational path taken to reach that fixed point.

Lyde and Might [9] extensively explored the consequences of state exploration choices in small-step abstract interpreters. They demonstrated that while the final analytical result remains invariant under different scheduling policies (assuming fair execution), the computational cost varies wildly. Their work highlights that poor scheduling leads to a combinatorial explosion of intermediate states, reinforcing our premise that the worklist algorithm is the primary performance bottleneck in non-trivial analyses.

Within the specific context of modular, effect-driven analysis, Kolozyan [7] directly investigated the efficacy of dependency-driven topological sorting. Kolozyan's work identified the "interleaving problem," demonstrating that strict structural prioritization fails catastrophically in higher-order functional languages due to recursive data structures. While Kolozyan showed that naive First-In-First-Out (FIFO) queues can bypass this issue through argument batching, the heuristic remained static. Our thesis directly builds upon this foundation. Rather than accepting the rigid trade-off between structural sorting and FIFO batching, we utilize Kolozyan's findings as the baseline to motivate a dynamic, machine-learning-driven scheduler capable of switching between these behaviors on the fly.

6.2 Machine Learning in Static Analysis

In recent years, the programming languages community has increasingly turned to data-driven techniques to tune the highly sensitive heuristics inherent in static analysis.
The most closely related approach to our work is the application of Machine Learning to parameterize analyzers. Heo et al. [6] proposed a method for adaptive static analysis using Bayesian Optimization. In their framework, machine learning is used to learn a policy that selects the optimal level of context-sensitivity (e.g., allocating higher precision to complex functions and lower precision to simple ones) based on the syntactic features of the target program.
While Heo et al. successfully demonstrated that ML can optimize static analysis, their approach is fundamentally *global and pre-emptive*. The machine learning model analyzes the code before execution and sets global parameters that remain fixed for the duration of the analysis.
In contrast, our approach is *local and reactive*. Although our model is trained offline and its weights remain static during deployment, it uses dynamic features. We utilize a Gradient Boosted Machine [2] to make scheduling micro-decisions at every individual step of the execution. By formulating the problem as a Learning-to-Rank (LTR) task [8], our model continuously responds to the live, unfolding mathematical state of the global store.

6.3 Summary

Existing research has thoroughly documented the vulnerabilities of traditional worklist heuristics and has begun exploring ML for pre-analysis parameter tuning. However, the use of supervised Learning-to-Rank algorithms to dynamically navigate the abstract interpretation state space during runtime remains largely unexplored. By extracting high-dimensional graph features and targeting simulated lattice progression, this thesis provides a data-driven methodology for mitigating the interleaving problem in the evaluated setting.

Chapter 7

Conclusion and Future Work

7.1 Summary of Findings

This thesis investigated the viability of using machine learning to dynamically optimize the worklist scheduling algorithm in modular static analysis. Traditional static heuristics, such as First-In-First-Out (FIFO) and dependency-driven topological sorting, are fundamentally constrained. While structural sorting minimizes re-evaluations in imperative contexts, it frequently triggers catastrophic interleaving cycles in higher-order functional programs. Conversely, while FIFO naturally mitigates these cycles through argument batching, it remains entirely blind to the semantic structure of the program. Our research successfully demonstrated that this rigid dichotomy can be overcome through a dynamic, data-driven scheduling policy. The primary findings of this thesis are as follows:

- **The Statistical Reality of the State Space:** By conducting large-scale random walks, we found empirical evidence consistent with the computational cost of worklist schedules being approximately log-normal on the studied benchmarks. This statistical framework also showed that highly optimized execution trajectories exist and can substantially outperform the standard FIFO baseline, validating the premise that adaptive scheduling is a worthwhile optimization target.
- **The Superiority of Local Semantic Targets:** We established that formulating the scheduling problem as an imitation learning task based on a global, empirical Oracle trace is highly susceptible to data scarcity and covariate shift. Instead, defining a local target based on simulated "Lattice Progression" provided the machine learning model with a dense, continuous reward signal. We demonstrated that this local progression target, while diverging from the Oracle's temporal path, constitutes a distinct, highly effective, and fundamentally more learnable scheduling strategy.
- **The Necessity of Lookahead Simulation:** The empirical evaluation showed that worklist scheduling is highly susceptible to myopia. A purely greedy model

that optimized only for immediate lattice updates performed worse than FIFO in the weighted aggregate, even though it still won on many smaller benchmarks. However, equipping the model with a 25-step simulated lookahead horizon (`L25_B1`) allowed it to achieve a geometric mean iteration ratio of 0.915 against FIFO on less training-similar programs.

- **Learned Dynamic Behavior:** Interpretability analysis via SHAP suggests that the Learning-to-Rank (LTR) model did not simply memorize structural graphs. Rather, its predictions reflect a balance between momentum exploitation and intentional argument batching, effectively combining strengths of both structural and naive heuristics on the fly.

In conclusion, this thesis successfully bridges the gap between machine learning and static analysis by demonstrating that worklist scheduling is not strictly bound to rigid, pre-defined heuristics. By redefining the scheduling problem as a local, reactive ranking task driven by structural and semantic features, we developed a predictive model capable of adapting to the unfolding complexities of higher-order functional programs. While achieving strict wall-clock improvements remains an ongoing systems-engineering challenge, the algorithmic reduction in analysis iterations—driven by the combination of a Lattice Progression target and lookahead simulation—supports the viability of this approach as an algorithmic optimization, pending substantial systems-level overhead reductions.

7.2 Future Directions

While the algorithmic success of the predictive scheduler is well supported by the data, the practical deployment of this system remains constrained by the computational overhead of the feature extraction pipeline. Transitioning this research from a theoretical proof-of-concept to a production-ready optimization requires significant systems-level engineering. Based on our findings, we propose the following avenues for future work:

Systems-Level Optimization and Hybrid Scheduling

The current framework recalculates expensive graph-theoretical metrics (such as PageRank and Betweenness Centrality) from scratch at every scheduling step, resulting in a wall-clock execution slowdown. Future implementations must focus on incremental feature extraction. Because the dependency graph in static analysis evolves by adding edges rather than completely restructuring, maintaining these metrics dynamically could substantially reduce the Time-Per-Iteration (TPI) overhead.
Furthermore, the predictive scheduler does not need to govern every single step of the analysis. A highly promising direction is the development of a *Hybrid Scheduler*. Such a system would utilize an $O(1)$ heuristic like FIFO during periods of low worklist contention, and only query the ML model when the worklist expands beyond a specific threshold, indicating a complex structural bottleneck.

Lightweight Model Architectures

While XGBoost provided exceptional accuracy and interpretability for this foundational study, gradient boosted trees are computationally expensive to evaluate compared to simple linear models. Future research should investigate whether the high-dimensional feature space can be distilled into a lightweight linear ranker or a highly optimized, shallow neural network, reducing inference latency without significantly degrading scheduling quality.

Reinforcement Learning (RL) Re-evaluation

At the outset of this research, Reinforcement Learning was deemed unsuitable due to the extreme sparsity of the final iteration-count reward. However, the successful formulation of the local "Lattice Progression" target fundamentally alters this landscape. Because Lattice Progression provides a dense, step-by-step measure of analytical progress, it can serve as a useful reward signal for an RL agent, although it remains a proxy for convergence. Training an RL agent in this newly defined environment could allow the scheduler to learn longer-horizon strategies without relying on expensive, discrete beam-search simulations.

Cross-Paradigm Generalization

Finally, the evaluation suite in this thesis was restricted to higher-order functional Scheme programs. A necessary next step is to integrate this predictive pipeline into static analyzers targeting imperative and object-oriented languages (such as C, Java, or Rust). Evaluating whether the learned policies transfer across fundamentally different programming paradigms will be critical in establishing predictive scheduling as a universal optimization technique for static analysis.

Bibliography

[1] Cameron B. Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[2] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 785–794. ISBN: 9781450342322. DOI: 10.1145/2939672.2939785. URL: https://doi.org/10.1145/2939672.2939785.

[3] Gordon V Cormack, Charles LA Clarke, and Stefan Buettcher. "Reciprocal rank fusion outperforms condorcet and individual rank learning methods". In: *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*. 2009, pp. 758–759.

[4] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1977, pp. 238–252.

[5] Leo Grinsztajn, Edouard Oyallon, and Gael Varoquaux. "Why do tree-based models still outperform deep learning on typical tabular data?" In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 507–520. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/0378c7692da36807bdec87ab043cdadc-Paper-Datasets_and_Benchmarks.pdf.

[6] Kihong Heo et al. "Adaptive static analysis via learning with bayesian optimization". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 40.4 (2018), pp. 1–37.

[7] Arman Kolozyan. *Dependency-driven worklist heuristics for modular analyses*. 2023.

[8] Tie-Yan Liu. "Learning to Rank for Information Retrieval". In: *Foundations and Trends in Information Retrieval* 3.3 (June 2009), pp. 225–331. ISSN: 1554-0669. DOI: 10.1561/1500000016. eprint: https://www.emerald.com/ftinr/article-pdf/3/3/225/11024564/1500000016en.pdf. URL: https://doi.org/10.1561/1500000016.

[9] Steven Lyde and Matthew Might. “State exploration choices in a small-step abstract interpreter”. In: *Scheme and Functional Programming Workshop*. 2015.

[10] Anders Møller and Michael I Schwartzbach. “Static program analysis”. In: *Notes. Feb* (2012).

[11] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction.* Vol. 1. 1. MIT press Cambridge, 1998.

[12] Noah Van Es et al. “MAF: A Framework for Modular Static Analysis of Higher-Order Languages”. In: *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2020, pp. 37–42. DOI: 10.1109/SCAM51674.2020.00009.