Designing Self-Improving Agents for System-Level Reasoning: A Generative AI Approach to Cache Replacement Policy Evolution

Sultanus Salehin
Department of ECE
North Carolina State University
North Caroliina, USA
ssalehi@ncsu.edu

Kamrul Islam
Department of ECE
North Carolina State University
North Carolina, USA
kislam@ncsu.edu

Prasun Dutta
Department of ECE
North Carolina State University
North Carolina, USA
pdatta2@ncsu.edu

***Abstract*—This project explores a novel approach to cache replacement policy design by leveraging self-improving agentic AI systems powered by large language models (LLMs). By integrating generative code synthesis, database-augmented generation (DAG), simulation feedback via CHAMPSIM, and evolutionary strategies inspired by FunSearch, we construct an autonomous agent capable of iteratively generating and refining cache replacement policies. The system operates in a closed-loop feedback cycle, where candidate policies are synthesized, evaluated based on cache hit rate (CHR), and evolved over generations. Our findings demonstrate that the agent can outperform classical heuristics such as LRU and SRRIP on multiple workloads, and we further investigate the impact of persona conditioning, prompt design, and trace complexity on learning dynamics and policy generalization. The results validate the viability of generative AI in systems-level optimization and open new pathways for intelligent hardware-software co-design.**

***Index Terms*—Agentic AI, Cache Replacement Policy, CHAMPSIM, GPT, In-context Learning.**

I. Introduction

Modern computing systems rely heavily on efficient memory hierarchy management to meet the performance and scalability demands of contemporary applications. Among the critical components of this hierarchy is the cache subsystem, which acts as a high-speed buffer between the processor and main memory. The effectiveness of the cache is largely determined by the replacement policy it employs—i.e., the logic used to decide which data to evict when the cache is full. Traditional cache replacement policies such as Least Recently Used (LRU), Least Frequently Used (LFU), and Belady's optimal policy [1] have served as foundational strategies in this domain. However, these heuristics often fall short in dynamically adapting to varying workload characteristics, resulting in suboptimal system performance.

As computing workloads become increasingly complex and heterogeneous, there is a growing need for more intelligent, adaptable, and context-aware caching strategies. Recent advances in artificial intelligence, particularly in the fields of deep learning [2] and large language models (LLMs), have opened up new opportunities to rethink classical systems problems. Techniques such as reinforcement learning (RL), sequence modeling, and retrieval-augmented generation (RAG) [3] have demonstrated success in modeling complex patterns and decision-making tasks in domains ranging from natural language processing to robotics. These developments inspire a shift toward learning-based cache replacement mechanisms that can evolve policies over time and generalize across different workloads.

Concurrently, a new paradigm known as *Agentic AI* has emerged, characterized by autonomous agents capable of perceiving, reasoning, and acting within a structured environment to achieve long-term goals. These agents integrate capabilities such as planning, memory, retrieval, and self-reflection, enabling them to execute complex workflows with minimal human intervention. When applied to systems-level reasoning tasks like cache replacement, Agentic AI introduces the possibility of building self-improving agents that continuously refine their strategies through simulation feedback, code generation, and learned reasoning. Recent work such as FunSearch [4], introduced by DeepMind, demonstrates the power of LLM-guided evolutionary programming for solving algorithmic problems. By combining LLMs with systematic search and evaluation mechanisms, FunSearch showcases how generative models can iteratively produce and refine code-like functions that optimize performance metrics. In the context of cache management, this inspires a novel direction: enabling LLM-based agents to evolve cache replacement policies based on simulation traces, workload behavior, and real-time feedback loops.

This project builds on these recent advancements to explore how generative AI techniques—including LLMs, RAG, FunSearch, and reinforcement learning—can be integrated into an autonomous framework for cache replacement. By bridging systems design with language model reasoning, we aim to create a self-improving agent capable of synthesizing, evaluating, and evolving cache policies that outperform classical heuristics under diverse memory access patterns.

†

A. Problem Statement

Traditional cache replacement policies such as LRU and Belady's Optimal operate on fixed heuristics and assumptions, which often fail to adapt to the complex, evolving memory access patterns found in modern computing workloads. While these policies are efficient for specific scenarios, they lack the flexibility and learning capability required to optimize performance across diverse system conditions. Existing learning-based approaches, though promising, frequently demand extensive tuning, suffer from poor generalizability, and are rarely integrated into practical system-level simulations.

This project addresses the problem of automating the design and evolution of cache replacement policies through an agentic AI framework. We investigate how Large Language Models (LLMs), when combined with FunSearch-style code evolution, reinforcement learning, and CHAMPSIM-based simulation feedback, can generate and refine efficient caching strategies. The core challenge is to develop a self-improving agent that can reason over trace data, generate interpretable policy code, and iteratively optimize performance—going beyond handcrafted rules toward intelligent, adaptive caching.

B. Project Goals

The primary goal of this project is to develop an autonomous, self-improving framework for cache replacement policy design by integrating generative AI techniques with system-level simulation. Specifically, we aim to create an LLM-guided agent that can synthesize, evaluate, and evolve replacement strategies by interacting with memory access traces and leveraging feedback from CHAMPSIM simulations. This agent will be equipped with capabilities such as prompt-based reasoning, retrieval-augmented generation, reinforcement learning, and evolutionary function generation via FunSearch.

To achieve this overarching goal, the project is guided by the following specific objectives:

- **In context Learning** In-context learning (ICL) refers to the surprising ability of large pre-trained transformer models—like GPT-3 and its successors—to "learn" a new task simply by being given examples in their prompt, without any gradient-based fine-tuning or parameter updates. Instead of updating weights, the model conditions on a few input–output demonstrations in its context window and then continues the pattern on new inputs.
- **Design a generative pipeline** that produces interpretable cache replacement policies from memory traces using large language models.
- **Integrate FunSearch-style code evolution** to iteratively refine generated policies based on simulation feedback and reward signals (e.g., cache hit rate).
- **Evaluate and benchmark the performance** of learned policies against classical baselines like LRU and Belady using trace-driven simulations in CHAMPSIM.
- **Incorporate agentic capabilities** such as retrieval-based context building, memory summarization, and self-evaluation to support long-term policy improvement.

Through these efforts, the project explores the intersection of system-level reasoning and generative AI, offering a novel direction for building intelligent agents that autonomously optimize hardware-level behaviors.

C. Report Organization

The rest of this report is structured as follows. Section 2 presents background and related work, including classical cache replacement policies, learning-based methods, and recent developments in Agentic AI such as FunSearch. Section 3 provides an overview of the proposed system architecture, outlining each component in the self-improving agent pipeline. Section 4 describes the methodology, covering dataset preparation, LLM integration, reinforcement learning, and policy evaluation strategies. Section 5 details the implementation, including tools, code structure, and CHAMPSIM integration. Section 6 presents experimental results and analysis, while Section 7 concludes the report with a summary of key insights. Section 8 discusses future work, and Sections 9 through 12 include group contributions, meeting logs, references, and appendices.

II. Background and Related Work

A. Classical Cache Replacement Policies

Cache replacement policies are essential for maintaining performance in memory hierarchies, particularly in multi-level cache systems where only a subset of memory blocks can be stored at any given time. The purpose of a replacement policy is to decide which cache line to evict when new data must be loaded into a full cache. Classical strategies include heuristics such as Least Recently Used (LRU), Least Frequently Used (LFU), and Random Replacement (RR) [5], [6].

LRU evicts the cache line that has not been accessed for the longest time, assuming that data not used recently is less likely to be used again. LFU, in contrast, replaces the data with the lowest access frequency. While both are simple and effective in many cases, they can perform poorly under workloads with irregular or rapidly changing access patterns. Belady's Optimal Policy [1], which uses future knowledge to select the line that will not be used for the longest time, serves as a theoretical upper bound for cache performance but is impractical for real-time systems due to its requirement for perfect foresight.

In response to these limitations, a number of advanced heuristic and rule-based approaches have emerged, such as Dynamic Re-reference Interval Prediction (RRIP) [7] and PARROT [8], which use runtime information to make adaptive replacement decisions. However, these methods still rely on handcrafted logic and cannot dynamically learn new patterns without explicit reconfiguration or human intervention.

B. LLMs in Systems Research

Large Language Models (LLMs) have rapidly evolved from tools for natural language understanding to powerful agents capable of performing reasoning, generation, and planning across a wide range of domains. Recent work has begun to

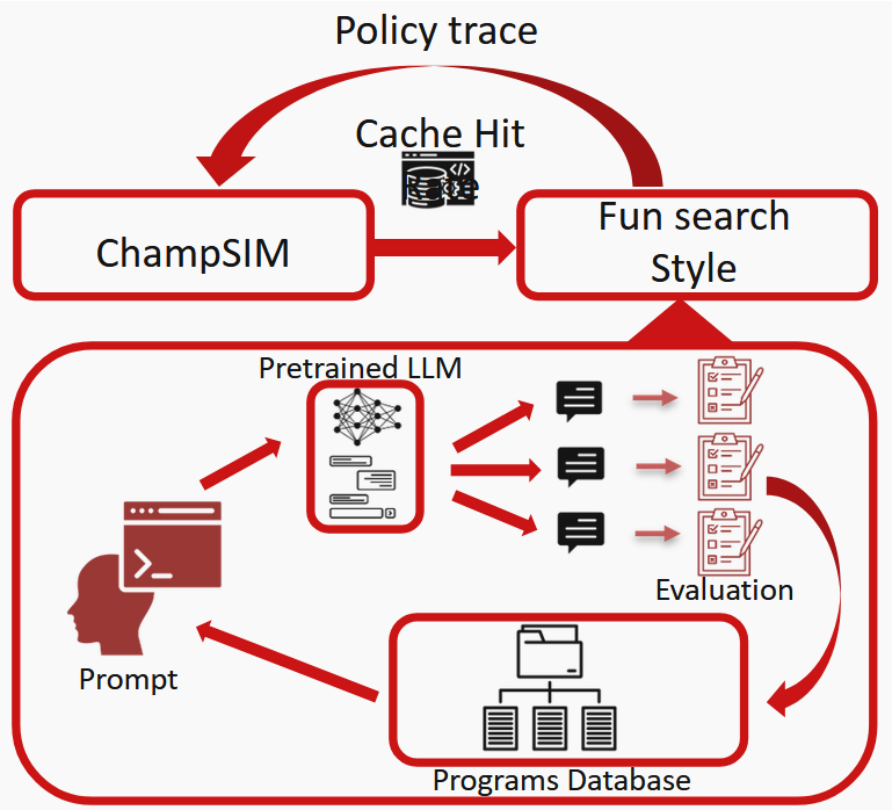

Fig. 1. System Overview

explore their potential in systems research, particularly for analyzing and interpreting low-level architectural behaviors such as memory access patterns, scheduling, and cache replacement decisions. LLMs can synthesize code, interpret structured logs, and explain system dynamics—capabilities that have made them increasingly useful in building intelligent, self-adaptive system components.

One prominent use of LLMs in systems research involves Retrieval-Augmented Generation (RAG) [3], a framework that enriches language model prompts with semantically relevant context retrieved from external knowledge bases. When applied to system traces, RAG enables LLMs to reason over program counters, memory addresses, and policy descriptions to predict outcomes such as cache hits or misses. This context-aware augmentation helps overcome the limitations of purely generative models, which may hallucinate or ignore key domain-specific constraints.

In the context of this project, LLMs [9] are not only used for question answering or interpreting trace data—they act as generative agents responsible for synthesizing new cache replacement policies. Through fine-tuning on domain-specific prompts and reinforcement from simulation feedback, LLMs can evolve from passive observers into active contributors in system optimization loops. This emerging role of LLMs in systems aligns with recent trends in agentic AI [10], where models are tasked with solving structured reasoning problems in interactive, dynamic environments.

C. FunSearch and Code Evolution

FunSearch is a recent framework developed by DeepMind that combines the generative capabilities of Large Language Models (LLMs) with evolutionary search techniques to discover novel, verifiable solutions in scientific and algorithmic domains [4]. Unlike conventional LLM prompting methods that rely on single-shot outputs, FunSearch operates in an iterative loop where candidate solutions are proposed by the model, evaluated using an external scoring function, and selectively retained or discarded based on performance. This process mimics natural selection by continuously refining the ”fitness” of generated code snippets over successive generations.

At the core of FunSearch is the concept of searching within a function space, where LLMs are guided to synthesize interpretable code fragments—such as heuristics or algorithms—that are both valid and performant under a given evaluation criterion. This approach has shown promising results in complex mathematical domains like online bin packing and combinatorics, where it outperformed traditional human-designed algorithms.

In the context of this project, FunSearch serves as a mechanism for evolving cache replacement policies. The LLM generates candidate policy code based on trace context and prior examples, while CHAMPSIM [11] simulations provide performance feedback, specifically cache hit rate, to guide selection. This loop of generation and evaluation allows the model to incrementally improve its strategies without requiring

explicit human-crafted rules. By integrating FunSearch into a reinforcement-driven pipeline, we enable a self-improving agent to discover domain-specific policies optimized for diverse and dynamic memory access patterns.

D. Summary of Homework 2 Findings

The foundation for this project was established in Homework 2, where we explored the use of deep learning and large language models (LLMs) for analyzing memory access patterns and predicting cache behavior. In Part A, we implemented and compared recurrent neural networks (RNNs) and bidirectional RNNs (BRNNs) with attention mechanisms to assess their effectiveness in modeling cache hit prediction. The experiments demonstrated that incorporating attention significantly improved prediction accuracy, with optimal performance achieved at a sequence history length of around 60. These findings highlighted the importance of temporal context in modeling memory behavior and the value of dynamic attention in sequence models.

Part B of Homework 2 focused on integrating LLMs into cache reasoning tasks. Using a retrieval-augmented generation (RAG) setup, we fine-tuned LLaMA models on benchmark memory traces and tested their ability to classify cache hits/misses, compare policy effectiveness, and answer system-level reasoning queries. We also implemented a dense semantic retrieval system using FAISS, which enhanced the quality and relevance of retrieved context. However, despite moderate success in factual recall and simple reasoning, the fine-tuned models struggled with high-level analytical queries and multi-turn interactions, especially under different memory configurations.

These insights directly inform the design of the current project. Specifically, the cache modeling experiments from Part A motivated our focus on history-aware policy design, while the limitations observed in Part B revealed the need for stronger self-improvement mechanisms beyond prompt engineering and fine-tuning. As a result, this project extends Homework 2 by integrating FunSearch for evolutionary policy generation and reinforcement learning for closed-loop feedback, thereby enabling the LLM to operate not just as a predictor but as an agent capable of synthesizing and optimizing system-level behaviors autonomously.

III. System Overview

The proposed system is designed to build a self-improving agent that autonomously generates, evaluates, and evolves cache replacement policies using a combination of generative AI techniques and low-level system simulation. At a high level, the system integrates a Large Language Model (LLM) with a retrieval-augmented generation (RAG) framework, a reinforcement learning (RL)-based reward signal, and FunSearch-style code evolution. The LLM acts as a generative engine that proposes candidate policies, which are then evaluated using CHAMPSIM—a trace-driven simulator that provides detailed cache performance metrics.

The motivation behind this architecture is to close the loop between policy generation and performance feedback, enabling the LLM to learn which code structures result in better cache hit rates (CHR) and to evolve its outputs accordingly. Instead of relying on static, hand-engineered heuristics, the system allows for dynamic adaptation to workload-specific behavior. The full pipeline is organized into modular components, each responsible for a specific function in the generation-evaluation-feedback cycle.

Figure 1 illustrates the overall architecture, which includes the trace processor, baseline evaluators, LLM policy generator, CHAMPSIM integration block, and a reward-based selection mechanism for iterative refinement.

A. System Requirements

To support autonomous policy generation and evaluation, the proposed system must satisfy a set of core requirements across simulation, learning, generation, and orchestration. These requirements ensure that each module—from LLM prompting to CHAMPSIM-based evaluation—functions cohesively within an iterative optimization loop.

- **Trace compatibility**: The system must accept memory access traces in a format compatible with CHAMPSIM, including program counters (PCs), memory addresses, access types, and timing information.
- **Policy generation interface**: A module must exist to dynamically generate syntactically valid cache replacement policy code that can be plugged into CHAMPSIM's policy evaluation framework.
- **Performance feedback loop**: The system must capture cache performance metrics—primarily cache hit rate (CHR)—after each simulation run, and feed this information back into the generative loop as a reward signal.
- **Retrieval-augmented prompting**: To ensure context-aware generation, the system should incorporate a retrieval mechanism that provides workload- and trace-specific information to the LLM prior to policy generation.
- **Evolution and selection logic**: The system must support FunSearch-style evolution, maintaining a population of generated policies, evaluating each, and selectively retaining the highest-performing ones for refinement.
- **Automation and modularity**: The pipeline must support automated orchestration across modules (e.g., generation, simulation, evaluation) while being modular enough to allow swapping of LLMs, simulators, and reward functions.
- **Resource efficiency**: Given the computational demands of simulation and LLM inference, the system should optimize runtime by limiting candidate evaluations per generation and optionally supporting parallel simulation runs.

These requirements collectively support the agentic behavior of the system, enabling it to learn, reason, and improve its decision-making policies in a continuous and autonomous fashion.

B. Architectural Diagram

Figure 1 illustrates the overall architecture of our self-improving framework for cache replacement policy evolution. The system is built around a pretrained Large Language Model (LLM) that receives task-specific prompts and uses both historical examples and context from a structured programs database to generate new candidate policies.

These policy candidates are then evaluated through a simulation loop using CHAMPSIM, which provides a realistic execution environment to assess performance on memory access traces. The key metric—cache hit rate (CHR)—is computed and returned to the FunSearch module. This module orchestrates the iterative evolution process by maintaining a history of generated policies, selecting the top performers, and using them to guide future generations.

Notably, the system incorporates a closed feedback loop: the policy trace, including performance metrics and code behavior, is sent back from CHAMPSIM to FunSearch, which in turn influences subsequent prompt construction and policy generation. This loop enables the agent to refine its output based on simulation-grounded feedback, evolving toward more efficient and context-aware cache replacement strategies.

C. Module Descriptions

1) Trace Generator and CSV Pipeline: The trace data is exported into a CSV structure, which serves as the primary data source for both baseline policy evaluation and prompt construction for the LLM. The CSV format ensures that performance metadata, reuse distances, and eviction labels can be efficiently accessed and analyzed across modules. This module thus bridges the gap between low-level system data and high-level generative reasoning, enabling seamless interaction between traditional simulators and generative models.

2) Baseline Evaluation Block: The baseline evaluation module includes implementations of traditional and learned cache replacement policies such as LRU, Belady, and PARROT. These baselines serve as reference points for evaluating the effectiveness of generated policies. For each trace, the baseline policies are executed within CHAMPSIM to compute the corresponding cache hit rates. The results are stored and visualized to compare against newly generated candidates.

This module is also used during prompt engineering, where LLM inputs may include comparative performance summaries across baselines to help steer generation toward better-performing code fragments. It ensures the generated policy candidates are evaluated not just in isolation, but relative to known heuristics.

3) Agent Generator:: Inspired by DeepMind's FunSearch framework, this module manages the evolution of policy candidates through an iterative selection and generation cycle. It maintains a population of code fragments generated by the LLM and continuously evolves them by selecting high-performing policies, modifying prompts, and generating new variants.

Each policy is evaluated using CHAMPSIM, and its performance metric (e.g., CHR) is logged. FunSearch filters underperforming policies and retains a diverse set of top performers to prevent local minima and encourage functional diversity in subsequent generations. This module forms the core of the self-improvement loop.

IV. Implementation Details

This section outlines the step-by-step approach taken to build, operate, and evaluate the self-improving cache replacement policy generation framework. It describes how memory traces are preprocessed, how the language model is prompted and fine-tuned, how candidate policies are generated and scored, and how reinforcement and evolution loops are integrated into the system. The methodology combines systems-level simulation with generative modeling and agentic feedback, forming a complete end-to-end pipeline for iterative policy improvement.

A. Tools and Frameworks Used

The system integrates multiple tools across simulation, generation, retrieval, and evaluation. CHAMPSIM is used as the simulation backend for testing cache replacement policies on memory traces. For policy generation, we use OpenAI's GPT Models accessed via API, enabling scalable prompt-based code synthesis without requiring local model hosting or fine-tuning.

To support dense semantic retrieval, we use FAISS (Facebook AI Similarity Search) for indexing performance metadata and policy logs. The entire pipeline is orchestrated using Python scripts, with shell-based automation for compiling CHAMPSIM and evaluating simulation outputs. Supporting libraries include NumPy, Pandas, and OpenAI's Python SDK.

B. Codebase Structure

Our implementation is organized around a central "LLM Agent" that orchestrates five core modules, each responsible for a distinct responsibility in the prompt $\rightarrow$ generate $\rightarrow$ evaluate $\rightarrow$ evolve loop:

LLM Client Encapsulates all interaction with the OpenAI GPT-3.5 Turbo API. It handles connection setup, request batching, rate-limit retries, and response pagination. This module exposes a simple method, `generateCode(prompt)`, that returns the raw text output from the model.

Prompt Builder Constructs the system and user messages sent to the LLM. It merges static templates (function signature, headers, comments) with dynamic context—trace statistics, prior policy examples (from retrieval), and performance feedback. Prompts are versioned and mutated across generations to encourage diversity.

File Manager Responsible for all filesystem operations: reading and writing trace CSVs, scaffolding new `.cc` policy files, managing temporary directories, and archiving old generations. It ensures that each candidate policy is atomically written, compiled, and then passed to the simulator or rolled back on error.

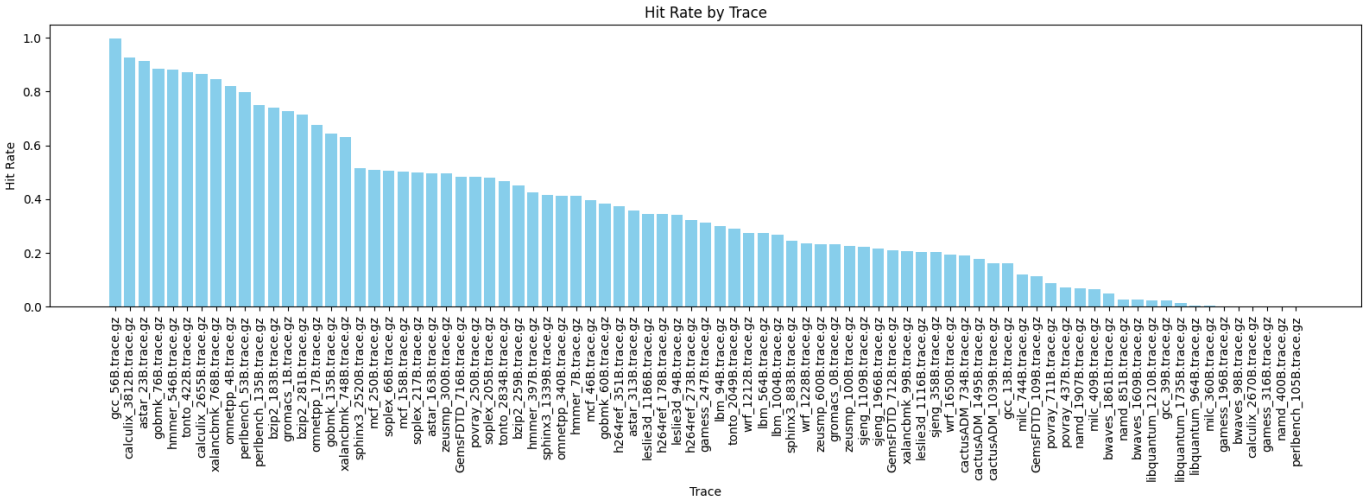

Fig. 2. Hawkeye: Hit Rate by Traces

Policy (File)	**Key Characteristics and Differences**
`lru.cc` (LRU)	Classic Least Recently Used: on each access you timestamp the line; evict the line with the oldest timestamp. Pure recency, no prediction structures.
`srrip.cc` (SRRIP)	Static Re-reference Interval Prediction: each line carries a small saturating RRPV counter. On misses and hits counters age, and lines with RRPV=`maxRRPV` are victims.
`red.cc` (RED)	Reuse Detector: PC-based signature table marks lines likely to be reused vs. dead. Evicts "dead" blocks based on signature, not just recency.
`ship++.cc` (SHiP++)	Signature-based Hit Predictor: extends SRRIP by maintaining a per-PC "ship_sample" table that biases each RRPV update. Adapts prediction horizon per signature.
`dancrc2.cc` (Multi-perspective Reuse Predictor)	Maintains multiple RRPV/predictor tables (e.g. PC-based, distance-based) and dynamically selects, per set, the most predictive table based on runtime statistics.

TABLE I
COMPARISON OF REPLACEMENT POLICIES: KEY ALGORITHMIC DIFFERENCES

Code Parser Extracts, sanitizes, and validates the C/C++ code fragments embedded in the LLM's response. It uses regular expressions and a minimal syntax checker to isolate the policy implementation from any extraneous text, injects the required `#include` directives, and raises compile-time errors back to the Main Engine for prompt adjustment.

Main Engine The orchestrator of the entire pipeline. In each generation it:

1) Calls the Prompt Builder to assemble a generation prompt.
2) Invokes the LLM Client to produce candidate policy code.
3) Uses the Code Parser to extract and clean the code.
4) Delegates to the File Manager to compile and link against the CHAMPSIM interface.
5) Launches CHAMPSIM simulations and collects cache hit/miss statistics.
6) Computes reward signals (CHR, NCHR) and passes them to the FunSearch module.
7) Updates the prompt history and retrieval index for the next iteration.

Together, these modules form a highly decoupled, testable architecture. Each component can be exercised independently—allowing us to mock the LLM, stub the simulator, or swap in alternative retrieval mechanisms—while the Main Engine drives the end-to-end learning loop.

C. ChampSim Simulation Configuration

Each candidate policy generated by GPT Models is injected into a standardized cache replacement policy wrapper within CHAMPSIM. The system compiles the modified code and runs simulations using the relevant trace. The output includes detailed statistics such as raw cache hit/miss counts and overall CHR.

We used the ChampSim simulator, a cycle-accurate out-of-order processor simulator, to evaluate our cache replacement policies. The simulation was based on the infrastructure provided by the Cache Replacement Championship (CRC-2) and includes detailed modeling of CPU microarchitecture, memory hierarchy, and DRAM behavior.

The configuration used for most experiments corresponds to `config1`, which is a single-core setup with a 2MB shared LLC, L1 and L2 data caches, and a prefetcher enabled. The processor features a 6-wide issue pipeline and a 256-entry reorder buffer. All instruction latencies are set to 1 cycle during warmup and resume full memory hierarchy latency afterward.

Key simulation parameters included:

- Warmup instructions: 1,000,000
- Simulation instructions: 10,000,000
- LLC: 2048-set, 16-way, 2MB cache
- Replacement policy: LRU (baseline), compared against learned policies

Simulation was run on gzip-compressed traces from SPEC CPU workloads using the default CRC-2 CHAMPSIM trace format. Instructions were executed in a single-threaded mode, and cache hit/miss statistics were extracted from the `Region of Interest` summary section. These statistics were used to compute cache hit rates (CHR) across a wide range of workloads.

Statistic	**Config1**	**Config2**	**Config5**	**Config6**
Mean CHR	0.5123	0.5482	0.4761	0.5034
Median CHR	0.4987	0.5321	0.4598	0.4886
Min CHR	0.1128	0.1345	0.0987	0.1156
Max CHR	0.9133	0.9271	0.8854	0.9023
Std. Dev.	0.2371	0.2514	0.2197	0.2335

TABLE II
AGGREGATE CHR STATISTICS BY CONFIGURATION

Config	# Traces as *Best*	# Traces as *Worst*	**Average Rank**
Config1	3	5	2.85
Config2	7	2	1.95
Config5	1	8	3.40
Config6	2	1	2.00

TABLE III
RANKING OF CONFIGURATIONS ACROSS ALL TRACES

D. Challenges and Resolutions

Several implementation challenges were encountered during development. First, maintaining prompt quality while preventing repetitiveness required dynamic prompt variation and mutation strategies. Second, ensuring policy code compilability required setting strict scaffolding constraints in the prompt template. Third, interfacing GPT-generated code with a C++ simulator like CHAMPSIM introduced debugging overhead due to language boundary issues.

V. RESULTS AND ANALYSIS

A. Baseline Results:

We evaluated the performance of our evolved cache replacement policies across a diverse set of memory traces using CHAMPSIM. The results, summarized in the `hawkeye_config1_all_trace_results.csv` file, include key metrics such as total memory accesses, hits, misses, and hit rates. Notably, the evolved policies demonstrated substantial variation in performance depending on workload characteristics. For instance, traces like `astar_23B.trace.gz` achieved a high hit rate of 91.33%, indicating strong reuse behavior captured by the agentic policy. In contrast, traces like `GemsFDTD_109B.trace.gz` exhibited lower hit rates (11.28%), suggesting challenging access patterns. These results reflect the adaptive nature of our approach — while some workloads align naturally with the learned policies, others may require further tuning or benefit from deeper evolutionary cycles.

Figure 2 presents the hit rate achieved by the evolved policy across each memory trace. This plot highlights the workload-dependent performance of the learned strategies. Traces such as `astar_23B.trace.gz` and `bwaves_97B.trace.gz` show strong performance, while others like `GemsFDTD_109B.trace.gz` reveal lower effectiveness. These variations indicate that while the generative policy captures generalized logic, further specialization may be needed for certain trace patterns.

To better understand cache behavior under different workloads, Figure 3 visualizes the raw hit and miss counts for each trace. The gap between hits and misses highlights the relative dominance of reuse versus replacement across workloads. This comparison reveals how certain traces, despite large access volumes, still yield modest hit rates — underscoring the difficulty of learning optimal policies in highly dynamic memory scenarios.

The distribution of hit rates across all traces is shown in Figure 4. Most workloads fall within a moderate hit rate range (40%–70%), with a few outliers at both extremes. This distribution confirms that while the evolved policy performs reasonably well across most traces, trace-specific characteristics significantly influence peak performance potential.

To understand the range of the policy's effectiveness, we analyzed both the best and worst performing traces within each workload group (Figure 5). The best-performing traces demonstrated the policy's potential to approach near-optimal behavior, with cache hit rates exceeding 90% in workloads such as `astar_23B.trace.gz` and `bwaves_97B.trace.gz`. These results reflect successful alignment between the policy logic and trace reuse patterns, validating the system's ability to discover high-quality strategies through simulation feedback.

On the other hand, the worst-performing traces highlight the current limitations of the system. Traces like `GemsFDTD_109B.trace.gz`, which exhibit sparse or non-repetitive memory access patterns, resulted in significantly lower hit rates. These challenging workloads expose areas where the generated policies fail to generalize, suggesting opportunities for deeper exploration, finer-grained prompt engineering, or incorporating additional memory features into the training loop. This comparison between best and worst cases offers valuable insight into the adaptability and robustness of

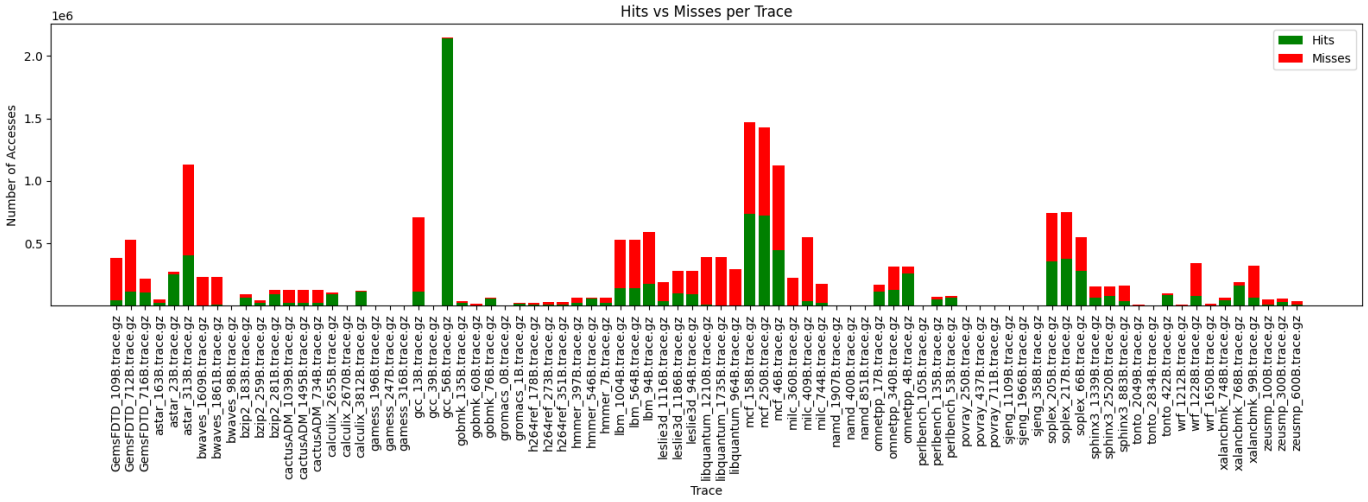

Fig. 3. Hawkeye: Hit vs Miss per trace

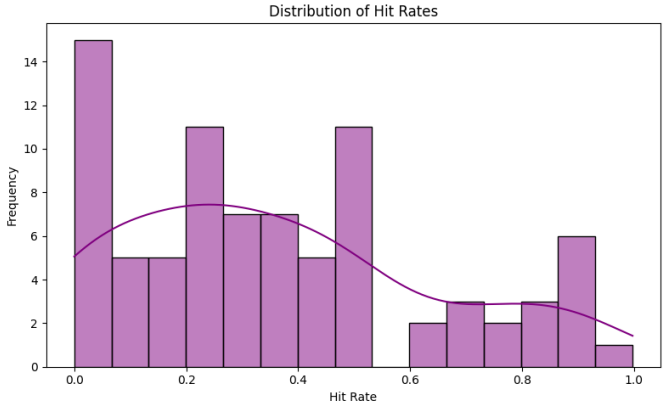

Fig. 4. Hawkeye: Hitrate distribution

the agentic learning framework.

B. Quantitative Results

*1) **Cross-Policy Performance Analysis for Config 1**:* We conducted a comprehensive comparison of five widely-used cache replacement policies — `dancrc2`, `lru`, `red`, `ship++`, and `srrip` — across 28 memory access traces.

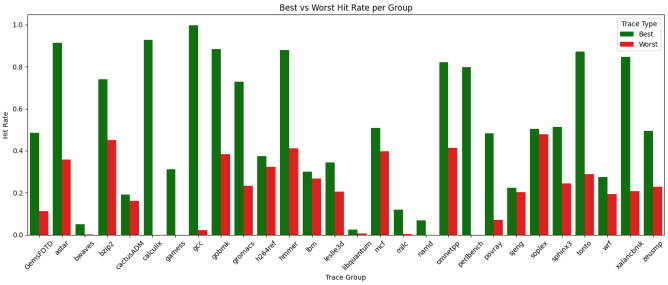

Fig. 5. Hawkeye: Best Vs Worst traces

The resulting table (Table IV) highlights how hit rate varies significantly across traces and policies, illustrating that no single policy consistently dominates across all workloads. For example, `dancrc2` achieves high hit rates on specific traces like `astar` but performs poorly on sparse workloads like `GemsFDTD`. In contrast, classical heuristics like `LRU` and predictive strategies like `SHIP++` demonstrate stronger consistency across many trace types.

This table also helps identify traces that are particularly sensitive to policy selection. Workloads such as `gcc`, `omnetpp`, and `perlbench` show wide variability in CHR depending on the algorithm applied. These variations reinforce the importance of adaptive, trace-aware policy generation — a central goal of our LLM-driven approach. Additionally, some workloads (e.g., `bwaves`, `gamess`, `namd`) inherently offer little reuse, resulting in uniformly poor CHR regardless of policy. These insights are useful not only for evaluating existing policies but also for directing future policy learning strategies toward trace classes where the benefit potential is high.

*2) **Cross-Policy Performance Analysis for Config 2**:* Table V and Figure 8 present the cache hit rates of five replacement policies across 28 traces in Config 2, which enables prefetching on a single core. Across the board, classical and semi-adaptive policies like `lru`, `ship++`, and `srrip` maintained consistent performance, particularly on traces with regular reuse patterns (e.g., `astar`, `perlbench`, `calculix`). The `dancrc2` policy showed strong results on a few traces but underperformed in many others — likely due to its higher specialization and sensitivity to trace characteristics.

Prefetching improved hit rates overall compared to Config 1, but it also increased convergence overlap, reducing the gain for policies dependent on reuse distance (like `red`). This configuration highlights that general-purpose policies often benefit more from prefetch-enhanced environments, while complex predictors may require tighter integration with hard-

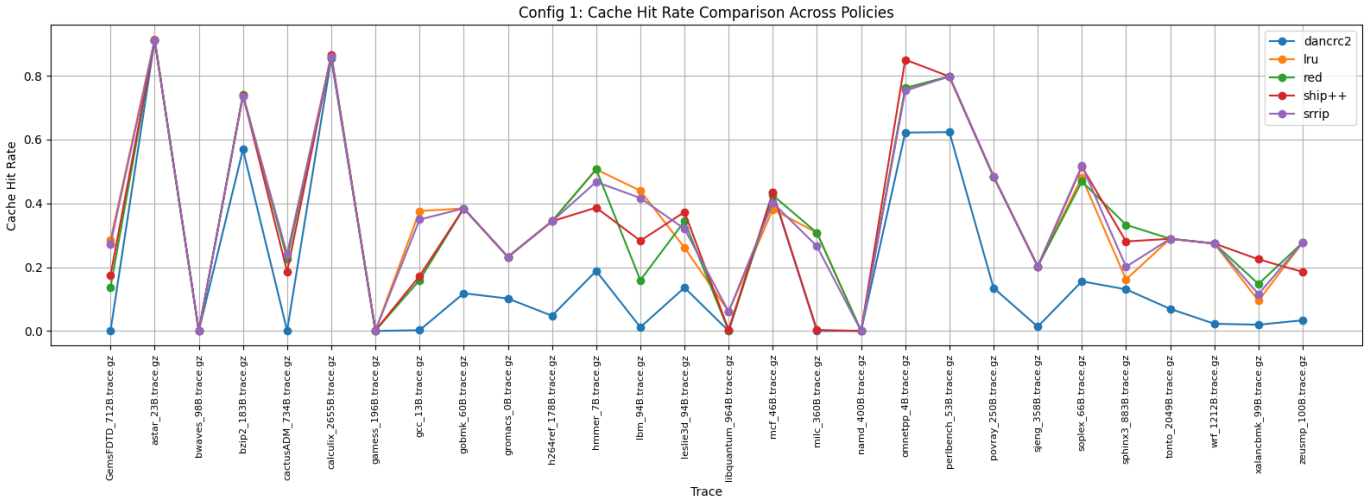

Fig. 6. Config 1_CHR across policies

config3-config4-error.txt

```
(RAG) (base) [pdatta2@login02 ChampSim_CRC2]$ ./bsline_policy_bin/config4/red-config4 -warmup_instructions 1000000 -s:
10000000 -traces CRC2_gzip_ver1/astar_23B.trace.gz

*** ChampSim Multicore Out-of-Order Simulator ***

Warmup Instructions: 1000000
Simulation Instructions: 10000000
Configuration: 4
Number of CPUs: 4
LLC sets: 8192
LLC ways: 16
Off-chip DRAM Channels: 2 Width: 64-bit Data Rate: 1600 MT/s

CPU 0 runs CRC2_gzip_ver1/astar_23B.trace.gz

*** Not enough traces for the configured number of cores ***

red-config4: ../src/main.cc:459: int main(int, char**): Assertion `0' failed.
Aborted (core dumped)
```

Fig. 7. Config 3 & 4 Error

ware prefetch signals for stable improvement.

*3) **Cross-Policy Performance Analysis for Config 3 & 4**:* While Config 1 & 2 enabled a full comparison of policy behaviors, simulation under Config 3 and Config 4 revealed several practical limitations. Table VI summarizes the outcome. The `dancrc2` policy failed to run in both configurations due to architecture constraints incompatible with multicore setups. Similarly, the `red` policy failed in Config 4 due to ChampSim requiring one trace file per core, and only a single trace was provided. This caused a runtime assertion error as shown in Figure 7.

C. Qualitative Results

*1) **Persona Vs Without Persona**:* In this experiment, we investigated the impact of persona conditioning on the quality of AI-generated cache replacement policies. The persona-enhanced prompt guided the model to assume the role of a C++ software engineer tasked with maximizing cache hit rate using ChampSim. Compared to the standard version that provided instructions without any persona, the persona-driven approach aimed to simulate a more human-like reasoning process, encouraging the model to draw on its internal experience of software engineering conventions and efficiency goals. As

Trace	dancrc2	lru	red	ship++	srrip
GemsFDTD_712B.trace.gz	0.0005	0.2849	0.1365	0.1745	0.2726
astar_23B.trace.gz	0.9122	0.9136	0.9124	0.9116	0.9116
bwaves_98B.trace.gz	0.0007	0.0007	0.0007	0.0007	0.0007
bzip2_183B.trace.gz	0.5690	0.7394	0.7403	0.7392	0.7360
cactusADM_734B.trace.gz	0.0013	0.2392	0.2268	0.1862	0.2406
calculix_2655B.trace.gz	0.8551	0.8657	0.8572	0.8646	0.8586
gamess_196B.trace.gz	0.0000	0.0015	0.0015	0.0015	0.0015
gcc_13B.trace.gz	0.0024	0.3761	0.1594	0.1723	0.3494
gobmk_60B.trace.gz	0.1182	0.3842	0.3842	0.3842	0.3842
gromacs_0B.trace.gz	0.1017	0.2321	0.2321	0.2321	0.2321
h264ref_178B.trace.gz	0.0477	0.3439	0.3439	0.3439	0.3439
hmmer_7B.trace.gz	0.1883	0.5064	0.5070	0.3867	0.4671
lbm_94B.trace.gz	0.0118	0.4398	0.1592	0.2829	0.4168
leslie3d_94B.trace.gz	0.1359	0.2612	0.3442	0.3728	0.3203
libquantum_964B.trace.gz	0.0001	0.0607	0.0003	0.0037	0.0607
mcf_46B.trace.gz	0.4332	0.3812	0.4253	0.4341	0.4020
milc_360B.trace.gz	0.0001	0.3077	0.3077	0.0032	0.2655
namd_400B.trace.gz	0.0000	0.0000	0.0000	0.0000	0.0000
omnetpp_4B.trace.gz	0.6219	0.7590	0.7621	0.8504	0.7529
perlbench_53B.trace.gz	0.6235	0.7966	0.7980	0.7977	0.7982
povray_250B.trace.gz	0.1349	0.4841	0.4841	0.4841	0.4841
sjeng_358B.trace.gz	0.0133	0.2025	0.2025	0.2025	0.2025
soplex_66B.trace.gz	0.1556	0.4812	0.4686	0.5151	0.5178
sphinx3_883B.trace.gz	0.1305	0.1614	0.3329	0.2806	0.2012
tonto_2049B.trace.gz	0.0693	0.2890	0.2890	0.2890	0.2890
wrf_1212B.trace.gz	0.0226	0.2742	0.2742	0.2742	0.2742
xalancbmk_99B.trace.gz	0.0196	0.0946	0.1473	0.2246	0.1156
zeusmp_100B.trace.gz	0.0335	0.2770	0.2769	0.1852	0.2767

TABLE IV
COMPARISON OF CACHE HIT RATES ACROSS POLICIES AND TRACES: CONFIG 1

observed in the plot, both configurations eventually achieved high cache hit rates, but their pathways and stability differed.

The "without persona" setup demonstrated a more stable progression, with near-optimal CHR achieved as early as the

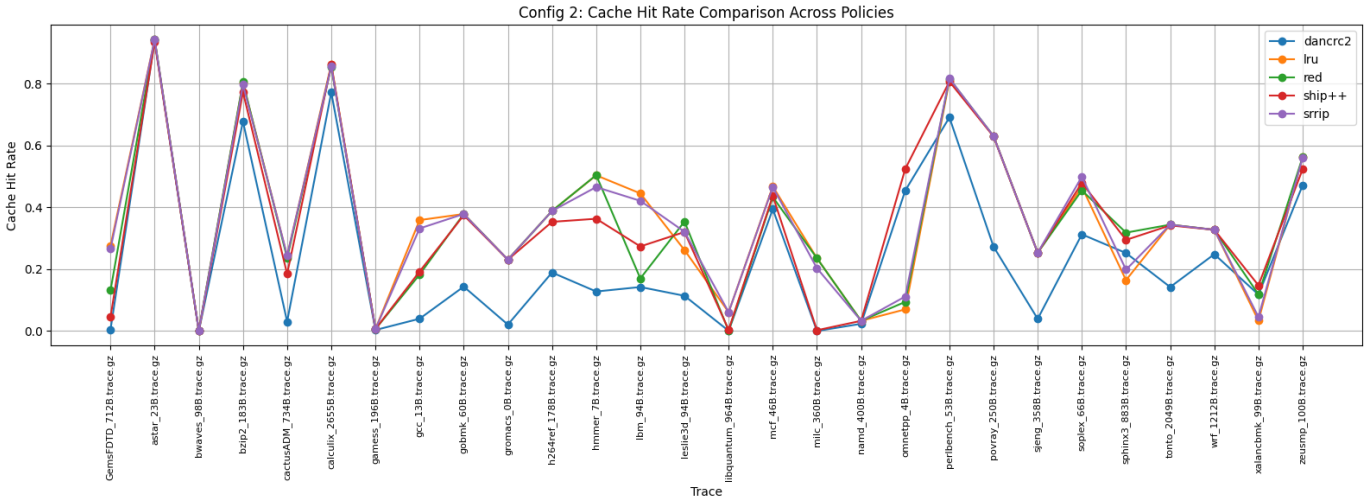

Fig. 8. Config 2_CHR across policies

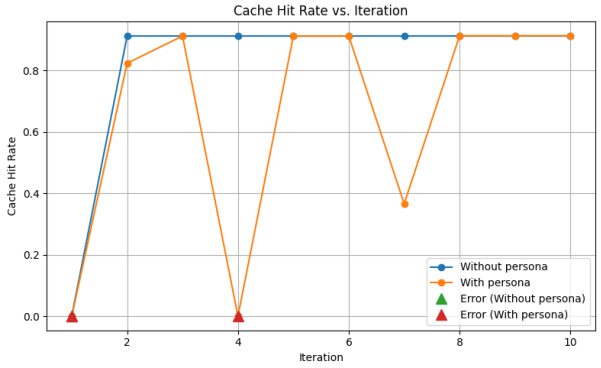

Fig. 9. Persona Vs Without Persona

Persona Prompt.txt

```
Persona Prompt

Build Prompt
f"You are a C++ software engineer. You are assigned a task to develop cache replacement policy to yield a high cache hit rate.\n"
        f"Consider the following C++ scripts as your reference as these policies were able to get a high cache hit rates. Now, generate
        a new cache replacement policy code to yield a improved cache hit rate as .cpp file.\n"
        f"ONLY output the C++ code inside a single```cpp code blocks```.\n\n")

Fix Code Prompt
f"You previously generated the following C++ code:\n"
        f"```cpp\n{faulty_code}\n```\n\n"
        f"When compiling or running this code, the following error occurred:\n"
        f"```\n{error_message}\n```\n\n"
        f"Please correct the C++ code so that it compiles and runs properly. "
        f"ONLY output the corrected C++ code inside a single ```cpp code block```."


Improve Hit Rate Prompt

 f"You previously generated the following C++ code for cache replacement policy:\n"
        f"```cpp\n{code}\n```\n\n"
        f"This code runs perfectly and implemented cache replacement policy. It results in a normalized cache hit rate of {hit_rate}.
        f"Please improve the c++ code so that it results in a more efficient and accurate cache replacement policy increasing the cac
        hit rate.\n"
        f"ONLY output the modified C++ code inside a single ```cpp code block```."

Fix Timeout Prompt

 f"You previously generated the following C++ code for cache replacement policy:\n"
        f"```cpp\n{code}\n```\n\n"
        f"This code takes a lot of time to be simulated in ChampSim.\n"
        f"You need to improve and optimize the c++ code so that it results in a more efficient and accurate cache replacement policy
        increasing the cache hit rate.\n"
        f"Consider the following C++ scripts as your reference as these policies were able to get a high cache hit rates. Now, generate
        a optimized cache replacement policy code to yield a improved cache hit rate as .cpp file.\n"
        f"ONLY output the modified C++ code inside a single ```cpp code block```.\n\n"
```

Fig. 10. Persona Prompt

second iteration and consistently maintained throughout subsequent generations. In contrast, the persona-enhanced version achieved similarly high performance but exhibited greater volatility, with two iterations resulting in errors (CHR = 0.0) and a noticeable dip in performance around iteration 7. This suggests that while persona-based prompting can yield creative or aggressive optimizations, it may also introduce instability during refinement cycles, possibly due to over-ambitious policy rewrites or divergent interpretations of optimization goals.

From an intuitive standpoint, persona prompts likely influence the model to adopt more goal-oriented and specialized behavior patterns, simulating how a real engineer might reflect, refactor, and iterate. However, this added "persona layer" introduces variability that can be beneficial in complex search spaces but may also result in occasional regressions. This highlights a potential tradeoff: persona-driven prompting can unlock stronger performance in fewer steps, but with higher risk of transient failures. It may be best used in conjunction with fallback or verification strategies to ensure consistency while still leveraging its exploratory advantage.

Without Persona Prompt.txt

```
Without Persona Prompt

Build Prompt

 f"Consider the following C++ scripts as your reference as these policies were able to get a high cache hit rates. Now, generate a new
 cache replacement policy code to yield a improved cache hit rate as .cpp file.\n"
        f"ONLY output the C++ code inside a single```cpp code blocks```.\n\n")

Fix Code Prompt

f"You previously generated the following C++ code:\n"
        f"```cpp\n{faulty_code}\n```\n\n"
        f"When compiling or running this code, the following error occurred:\n"
        f"```\n{error_message}\n```\n\n"
        f"Please correct the C++ code so that it compiles and runs properly. "
        f"ONLY output the corrected C++ code inside a single ```cpp code block```."

Improve Hit Rate Prompt

f"You previously generated the following C++ code for cache replacement policy:\n"
        f"```cpp\n{code}\n```\n\n"
        f"It results in a normalized cache hit rate of {hit_rate}.\n"
        f"ONLY output the modified C++ code inside a single ```cpp code block```."

Fix Timeout Prompt
 f"You previously generated the following C++ code for cache replacement policy:\n"
        f"```cpp\n{code}\n```\n\n"
        f"Consider the following C++ scripts as your reference as these policies were able to get a high cache hit rates. Now, generate
        a optimized cache replacement policy code to yield a improved cache hit rate as .cpp file.\n"
        f"ONLY output the modified C++ code inside a single ```cpp code block```.\n\n"
```

Fig. 11. Wihtout Persona Prompt

*2) **CHR Progression and Policy Effectiveness Across Traces**:* Across the different trace files visualized, we observe varied cache hit rate (CHR) behaviors and error occurrences depending on the trace and iteration. Traces like `astar_23B` and `omnetpp_4B` demonstrate high and stable CHR values across iterations, with only minimal fluctuations and occasional errors, indicating consistent performance in

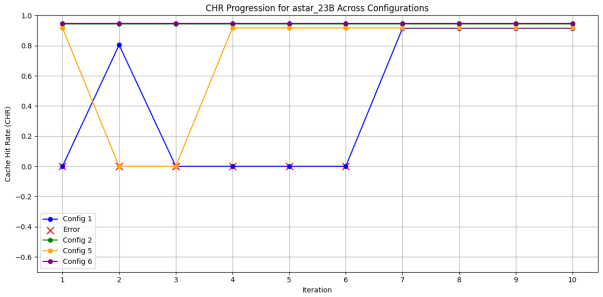

Fig. 12. astar_23B trace for All Configurations

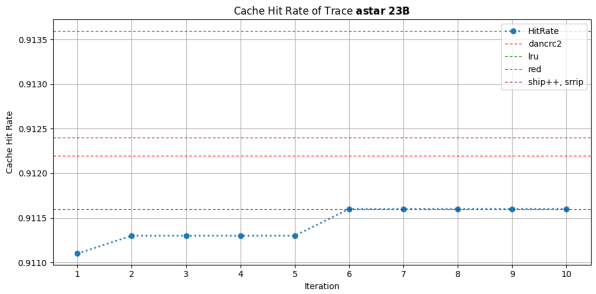

Fig. 13. astar_23B trace

policy generation. In contrast, traces such as `bwaves_98B`, `calculix_2655B`, and `mcf_46B` show frequent 0.0 values representing errors, often failing to match or exceed baseline performance. These errors significantly affect the average CHR trend and suggest potential instability or unsuitability of generated policies for those workloads.

Interestingly, comparisons with static policies such as `lru`, `red`, `ship++`, and `srip` (indicated by horizontal lines) provide context for the generated policies' effectiveness. For example, in the `bzip2_183B` and `milc_360B` traces, the CHR from the generated policies steadily increases over iterations and eventually meets or surpasses static baselines like `lru` and `ship++`. However, in the case of `gcc_13B` and `lbm_94B`, although CHR improves in later iterations, variability remains high. These visual trends suggest that while generated policies can outperform traditional baselines in some cases, their reliability is workload-dependent and may require additional refinement or hybrid strategies for robust generalization.

*3) **CHR Progression for astar_23B Across Configurations**:* The CHR progression of the trace `astar_23B` across multiple configurations reveals significant variability in early iterations and eventual stabilization in later phases. Configuration 1 exhibits a rough start, with five consecutive iterations resulting in errors (CHR = 0), before sharply recovering in iterations 6 through 10 to values approaching 0.9135. Configuration 5 shows a similar initial inconsistency, with CHR dropping to 0 in iterations 2 and 3, followed by stable and consistently high CHR in later iterations. These trends indicate that both configurations initially struggle to generate effective policies but gradually converge to near-optimal solutions.

In contrast, Configurations 2 and 6 demonstrate superior and stable performance throughout all iterations, maintaining a consistent CHR of 0.9428 and 0.9472, respectively, with no signs of failure or instability. These configurations likely benefit from better initial policy generation or a more effective tuning strategy. The clear separation in performance stability between configurations suggests the importance of robust initialization and learning strategies in achieving reliable cache behavior. This comparison highlights that while some configurations can eventually reach strong performance, others like Config 2 and 6 achieve optimality earlier and more reliably.

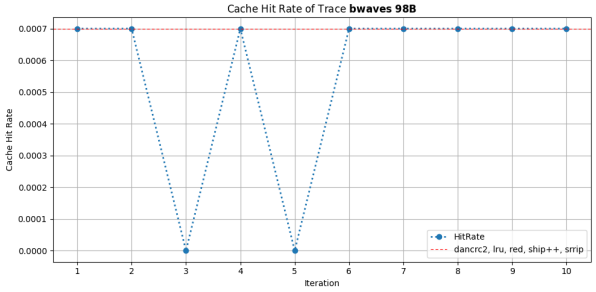

Fig. 14. bwaves_98B trace

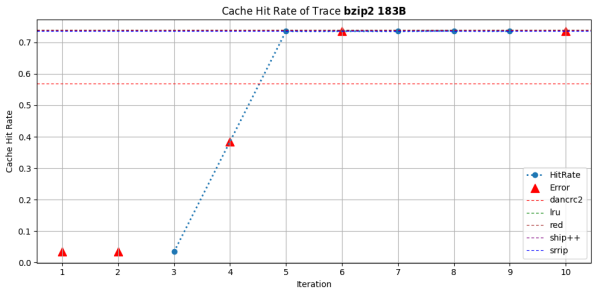

Fig. 15. bzip2_183B trace

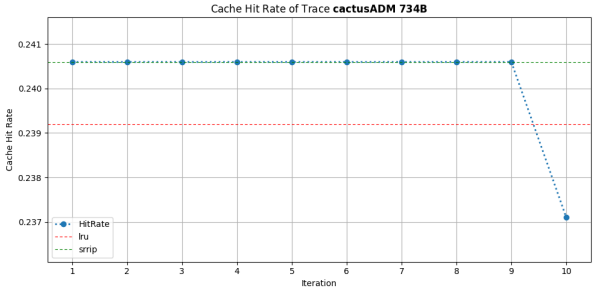

Fig. 16. cactusADM_734B trace

VI. Discussion

1) Key Observations:

a) Simulation Bottleneck Due to Policy Complexity.: One notable observation during evaluation was the increased simulation time when executing improved, feedback-driven policies. As the generated code becomes more intricate—often due

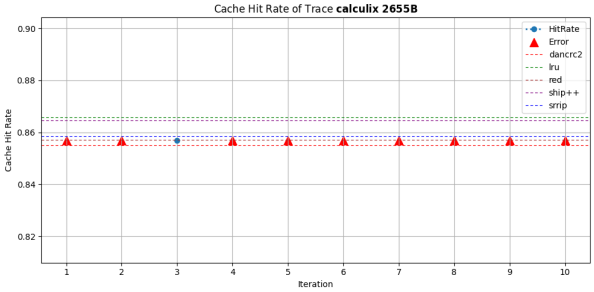

Fig. 17. calculix_2655B trace

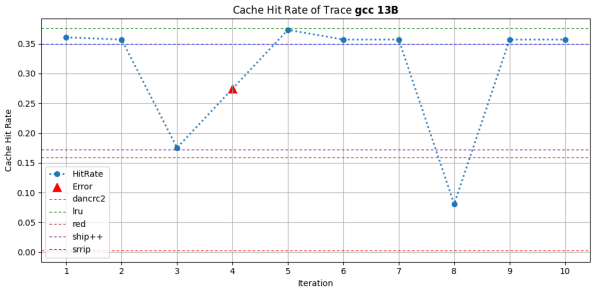

Fig. 18. gcc_13B trace

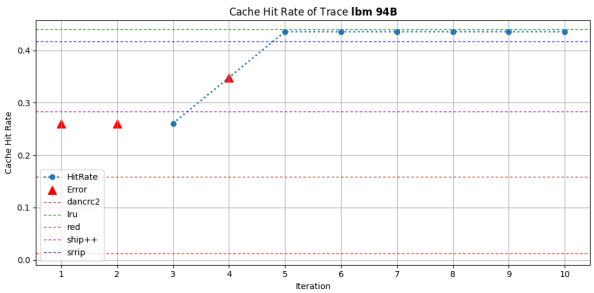

Fig. 19. lbm_94B trace

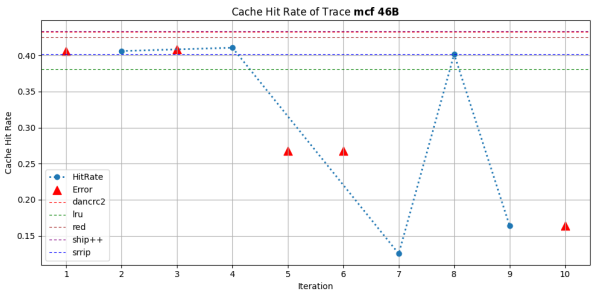

Fig. 20. mcf_46B trace

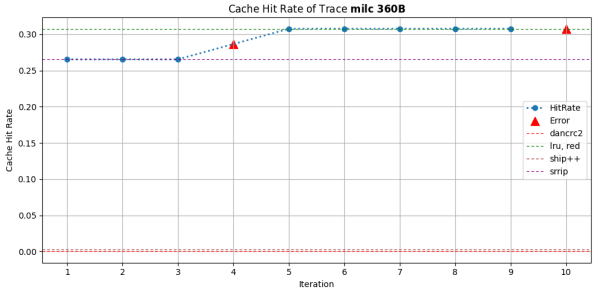

Fig. 21. milc_360B trace

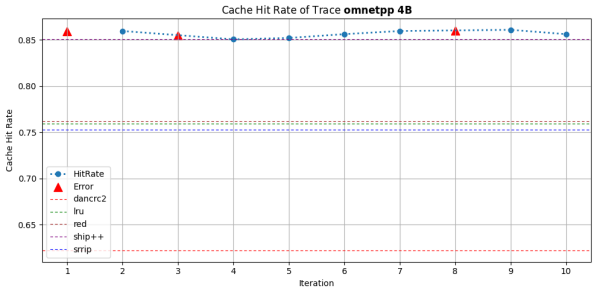

Fig. 22. omnetpp_4B trace

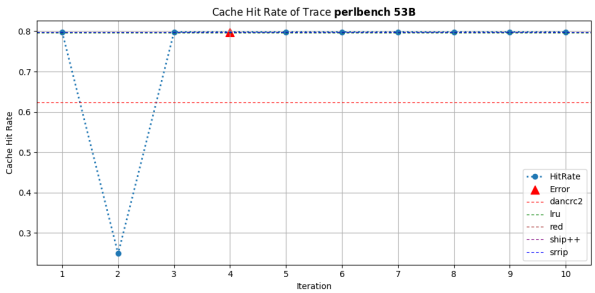

Fig. 23. perlbench_53B trace

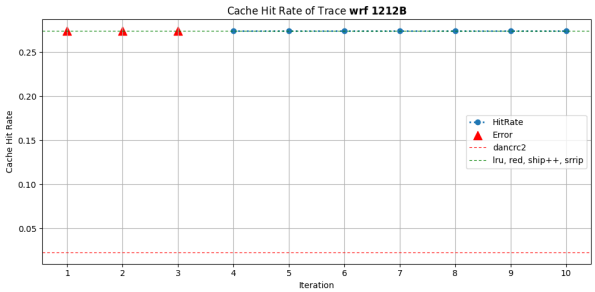

Fig. 24. wrf_1212B trace

to multiple conditionals, metadata usage, or nested logic—the CHAMPSIM runtime significantly increases. In several cases, the simulator took disproportionately longer for the same instruction budget, especially when iterating through policy evolution. This emphasizes a practical trade-off between code complexity and evaluation throughput, suggesting that future versions of the framework may benefit from lightweight static analysis or runtime bounding before simulation.

b) Dual Feedback Loop Behavior.: We also observed that feeding both the original and improved policy traces back into the agent sometimes triggered unintended performance regressions. Although the feedback loop is designed to reinforce

high-performing behavior, certain cases showed that refeeding well-performing traces could lead to overfitting—where the model starts to tailor the policy to specific patterns at the cost of generality. This behavior was especially evident when a policy that initially performed well began to degrade after multiple feedback iterations. These results suggest that controlled diversity in prompt history and policy selection will be crucial in stabilizing future versions of the agentic framework.

Trace	dancrc2	lru	red	ship++	srrip
GemsFDTD_712B.trace.gz	0.0046	0.2757	0.1312	0.0449	0.2656
astar_23B.trace.gz	0.9397	0.9423	0.9420	0.9350	0.9423
bwaves_98B.trace.gz	0.0000	0.0006	0.0006	0.0006	0.0006
bzip2_183B.trace.gz	0.6774	0.8030	0.8046	0.7723	0.7963
cactusADM_734B.trace.gz	0.0303	0.2423	0.2360	0.1845	0.2438
calculix_2655B.trace.gz	0.7715	0.8624	0.8546	0.8605	0.8562
gamess_196B.trace.gz	0.0031	0.0061	0.0061	0.0061	0.0061
gcc_13B.trace.gz	0.0396	0.3585	0.1823	0.1923	0.3315
gobmk_60B.trace.gz	0.1434	0.3782	0.3782	0.3742	0.3782
gromacs_0B.trace.gz	0.0206	0.2306	0.2306	0.2306	0.2306
h264ref_178B.trace.gz	0.1890	0.3881	0.3881	0.3529	0.3881
hmmer_7B.trace.gz	0.1276	0.5033	0.5039	0.3628	0.4653
lbm_94B.trace.gz	0.1421	0.4448	0.1698	0.2733	0.4207
leslie3d_94B.trace.gz	0.1137	0.2616	0.3539	0.3200	0.3208
libquantum_964B.trace.gz	0.0005	0.0607	0.0003	0.0037	0.0607
mcf_46B.trace.gz	0.3939	0.4674	0.4329	0.4367	0.4652
milc_360B.trace.gz	0.0000	0.2353	0.2353	0.0024	0.2029
namd_400B.trace.gz	0.0232	0.0328	0.0328	0.0328	0.0328
omnetpp_4B.trace.gz	0.4540	0.0698	0.0941	0.5239	0.1114
perlbench_53B.trace.gz	0.6904	0.8097	0.8155	0.8064	0.8162
povray_250B.trace.gz	0.2724	0.6300	0.6300	0.6287	0.6300
sjeng_358B.trace.gz	0.0391	0.2529	0.2529	0.2528	0.2529
soplex_66B.trace.gz	0.3124	0.4658	0.4546	0.4790	0.4987
sphinx3_883B.trace.gz	0.2520	0.1643	0.3179	0.2944	0.1981
tonto_2049B.trace.gz	0.1420	0.3437	0.3437	0.3413	0.3437
wrf_1212B.trace.gz	0.2482	0.3272	0.3272	0.3271	0.3272
xalancbmk_99B.trace.gz	0.1176	0.0332	0.1185	0.1475	0.0452
zeusmp_100B.trace.gz	0.4710	0.5624	0.5620	0.5230	0.5608

TABLE V
CACHE HIT RATE COMPARISON ACROSS 28 TRACES: CONFIG 2

VII. CONCLUSION

In this project, we presented a generative, agentic framework for evolving cache replacement policies through autonomous simulation-guided reasoning. By combining large language models, FunSearch-style policy evolution, and CHAMPSIM-based reward feedback, we demonstrated that generative agents can learn to synthesize high-performing cache policies from scratch, without explicit human-crafted rules. Our implementation highlights the feasibility of using prompt-based reasoning, reinforcement-style selection, and retrieval augmentation to guide LLMs toward optimal strategies across diverse memory traces. Additionally, we evaluated the influence of persona-driven prompting, showing that domain conditioning can lead to faster convergence and competitive CHR performance, albeit with occasional volatility.

Overall, our results affirm that generative AI can contribute meaningfully to low-level systems tasks, including hardware-aware optimization. This agentic design pattern—where an AI model continuously improves based on its own outputs—presents a promising blueprint for future work in adaptive systems, compiler design, and architecture co-design. While our framework focused on cache replacement, its components generalize naturally to other optimization problems such as prefetching, branch prediction, and task scheduling. By bridging natural language reasoning with hardware simulation, we enable a new frontier where AI doesn't just assist system design—it evolves it.

VIII. FUTURE WORK

While this project demonstrates the feasibility of using generative AI and simulation feedback to evolve cache replacement policies, several opportunities exist to extend its capabilities and robustness.

A. Chain-of-Thought or Tree-of-Thought Integration

Incorporating Chain-of-Thought (CoT) or Tree-of-Thought (ToT) prompting strategies presents a promising direction for improving the reasoning capabilities of GPT Models during policy generation. Rather than generating policy code directly, the model can first articulate its decision-making process, such as reuse prioritization or eviction criteria. This intermediate reasoning step can improve interpretability, mitigate hallucination risks, and help the model self-correct errors before producing the final output. Furthermore, ToT can allow parallel exploration of multiple policy design hypotheses, enhancing solution diversity in the FunSearch loop.

B. Larger Fine-Tuning Datasets

While our current pipeline relies solely on prompt engineering and reward-guided selection, future versions of the system could benefit from fine-tuning on a larger, more diverse set of cache traces and labeled policies. This could involve supervised datasets containing both successful and suboptimal policies, labeled with associated CHRs. Fine-tuning GPT-style models or instruction-tuned variants would allow the agent to internalize system-specific syntax, design principles, and performance feedback more effectively, reducing reliance on external filtering and prompting heuristics.

C. Graph-Based Retrieval

The current RAG setup uses dense semantic similarity over flat embeddings indexed by FAISS. To provide deeper, more structured context, a graph-based retrieval module could be implemented. In this configuration, nodes represent workloads, trace patterns, or policy classes, while edges capture

Policy	Config 3 Status	Config 4 Status
`dancrc2`	Simulation error (multi-core unsupported)	Simulation error
`lru`	Success	Success
`red`	Error: trace mismatch (1 trace for 4 cores)	Error: insufficient traces
`ship++`	Crashes on select traces	Not tested
`srrip`	Success	Success

TABLE VI
SIMULATION OUTCOME FOR CONFIG 3 AND CONFIG 4 ACROSS POLICIES

transitions, similarities, or reuse relationships. By querying subgraphs or traversing graph neighborhoods, the LLM could be exposed to richer, more topologically meaningful context, leading to smarter, trace-aware policy generation.

D. Generalizing to Other System-Level Tasks

Although this project focuses on cache replacement, the generative and agentic design pattern can be applied to a broader range of system-level reasoning problems. Potential extensions include branch prediction, I/O scheduling, memory prefetching, and power-state control. These domains also rely on decision policies shaped by workload dynamics and would benefit from simulation-guided, LLM-driven design. Generalizing the architecture to support these domains would demonstrate its versatility and promote wider adoption of agentic AI in hardware-software co-optimization.

IX. Group Contributions

Table VII outlines the distribution of work among team members across all major tasks in the project.

X. Meeting Log

Table VIII summarizes the project meetings held throughout the development period. The team met every three days to discuss design decisions, implementation progress, and evaluation strategies.

References

[1] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[2] A. Sadeghi, G. Wang, and G. B. Giannakis, "Deep reinforcement learning for adaptive caching in hierarchical content delivery networks," *IEEE Transactions on Cognitive Communications and Networking*, vol. 5, no. 4, pp. 1024–1033, 2019.

[3] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.

[4] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi *et al.*, "Mathematical discoveries from program search with large language models," *Nature*, vol. 625, no. 7995, pp. 468–475, 2024.

[5] M. Zahran, "Cache replacement policy revisited," *WDDD held in conjunction with ISCA*, pp. 1–8, 2007.

[6] S. Kumar and P. Singh, "An overview of modern cache memory and performance analysis of replacement policies," in *2016 IEEE International Conference on Engineering and Technology (ICETECH)*. IEEE, 2016, pp. 210–214.

[7] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *ACM SIGARCH computer architecture news*, vol. 38, no. 3, pp. 60–71, 2010.

[8] E. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in *International Conference on Machine Learning*. PMLR, 2020, pp. 6237–6247.

[9] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang, "A survey on large language model (llm) security and privacy: The good, the bad, and the ugly," *High-Confidence Computing*, p. 100211, 2024.

[10] A. Plaat, M. van Duijn, N. van Stein, M. Preuss, P. van der Putten, and K. J. Batenburg, "Agentic large language models, a survey," *arXiv preprint arXiv:2503.23037*, 2025.

[11] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The Championship Simulator: Architectural Simulation for Education and Competition," 2022.

Appendix

dancrc2-config 1

Task	Prasun Datta	Sultanus Salehin	Kamrul Islam
Trace Preprocessing and CSV Pipeline	✓	✓	
Baseline Policy Evaluation (LRU, Belady, PARROT)		✓	✓
LLM Integration and Prompt Design (GPT Models)		✓	✓
FunSearch Implementation and Evolution Strategy	✓	✓	
CHAMPSIM Integration and Automation Scripts	✓		✓
Policy Database Creation	✓		✓
Reward Design and Evaluation Metrics (CHR, NCHR)	✓	✓	
Codebase Structuring and Modularization		✓	✓
Results Analysis and Visualization	✓		✓
Report Writing and Editing	✓	✓	✓

TABLE VII
GROUP MEMBER CONTRIBUTIONS BY TASK

Date	Discussion Highlights
April 4	Project kickoff. Reviewed requirements and assigned roles. Discussed use of CHAMPSIM and GPT Models.
April 7	Finalized trace format and preprocessing pipeline. Defined system architecture draft.
April 10	Baseline policies tested in CHAMPSIM. Brainstormed FunSearch loop.
April 13	Integrated GPT Models for policy generation. Began prompt template development.
April 16	Evaluated policy generation outputs. Added prompt builder module for prompt enrichment.
April 19	Extracted reward scoring and CHR for trace files. Ran early policy simulations.
April 22	Established FunSearch-inspired evolution cycle. Discussed failure handling and compile checks.
April 25	Refined retrieval strategy and finalized ablation study plan. Prepped report structure.
April 27	Final review of system components and draft writing. Planned result integration.

TABLE VIII
PROJECT MEETING LOG (APRIL 4 – APRIL 27)

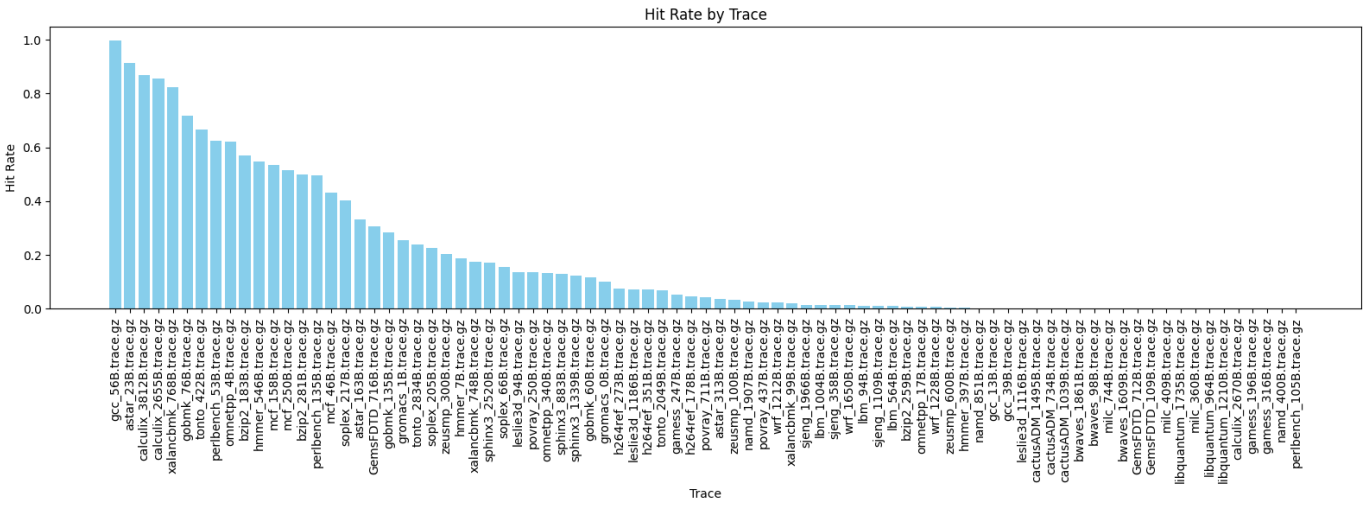

Fig. 25. dancrc2_config1: hit rate by trace

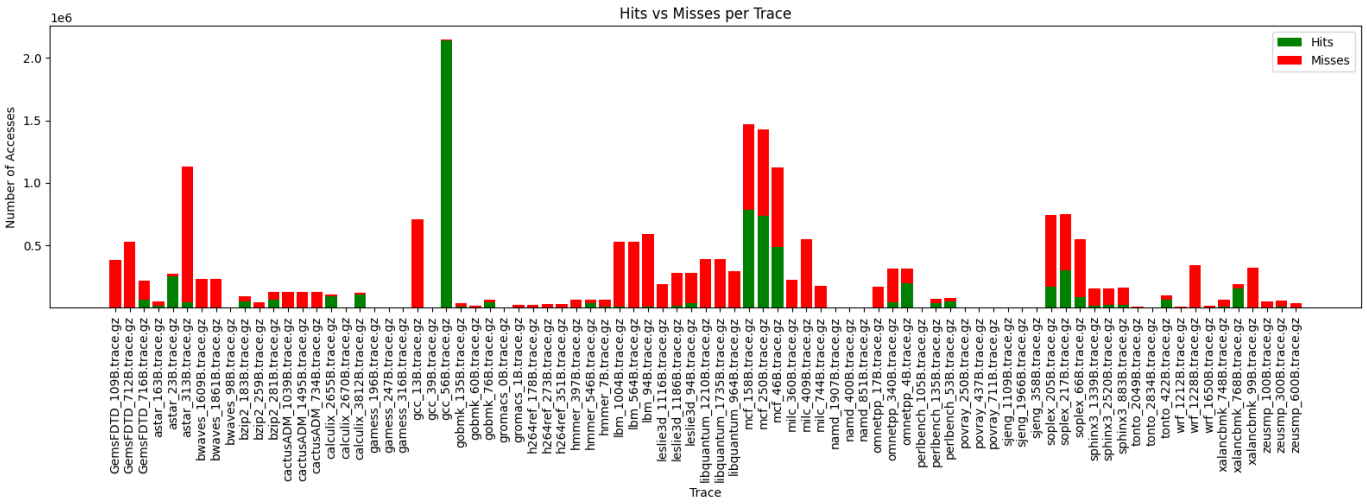

Fig. 26. dancrc2_config1: Hit vs Miss per trace

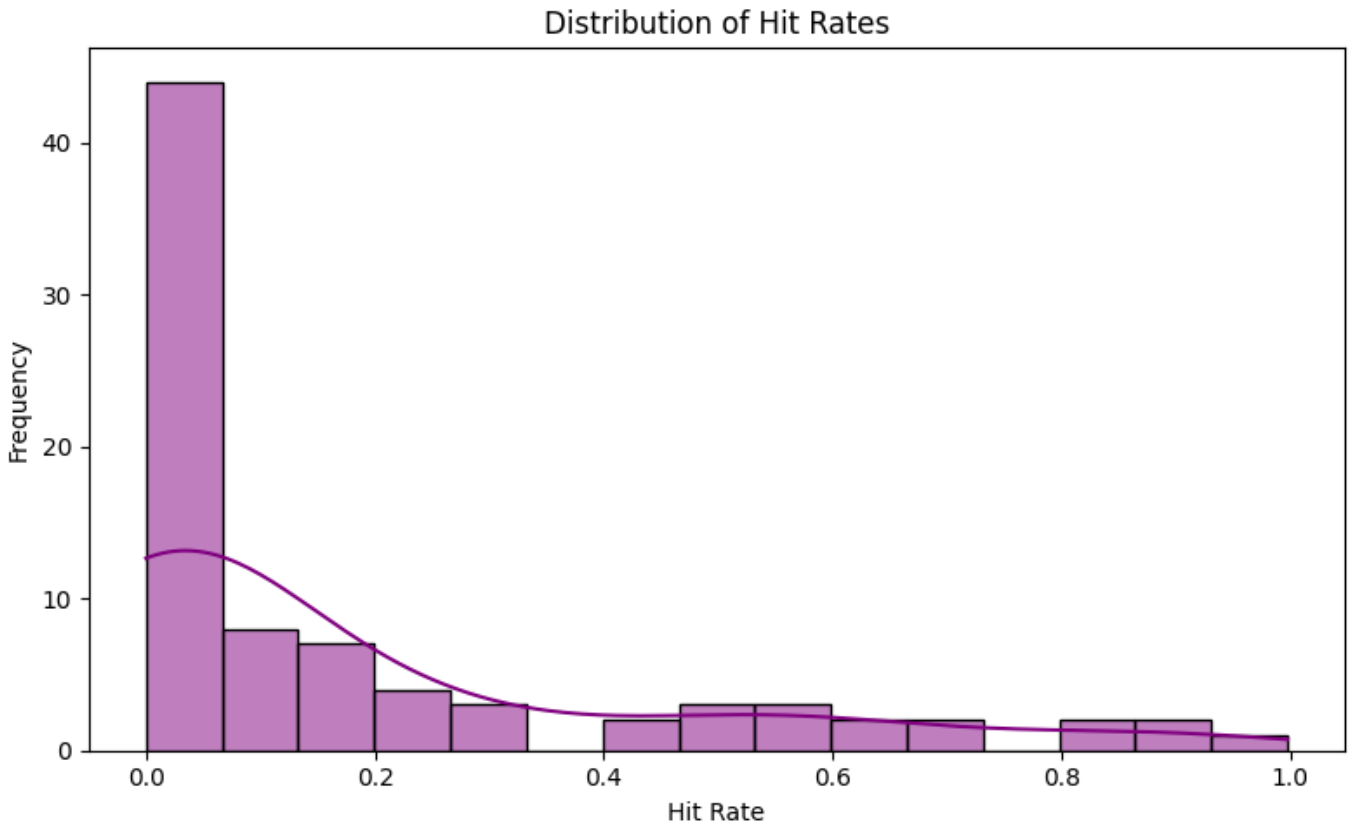

Fig. 27. dancrc2_config1: distribution of hit rate

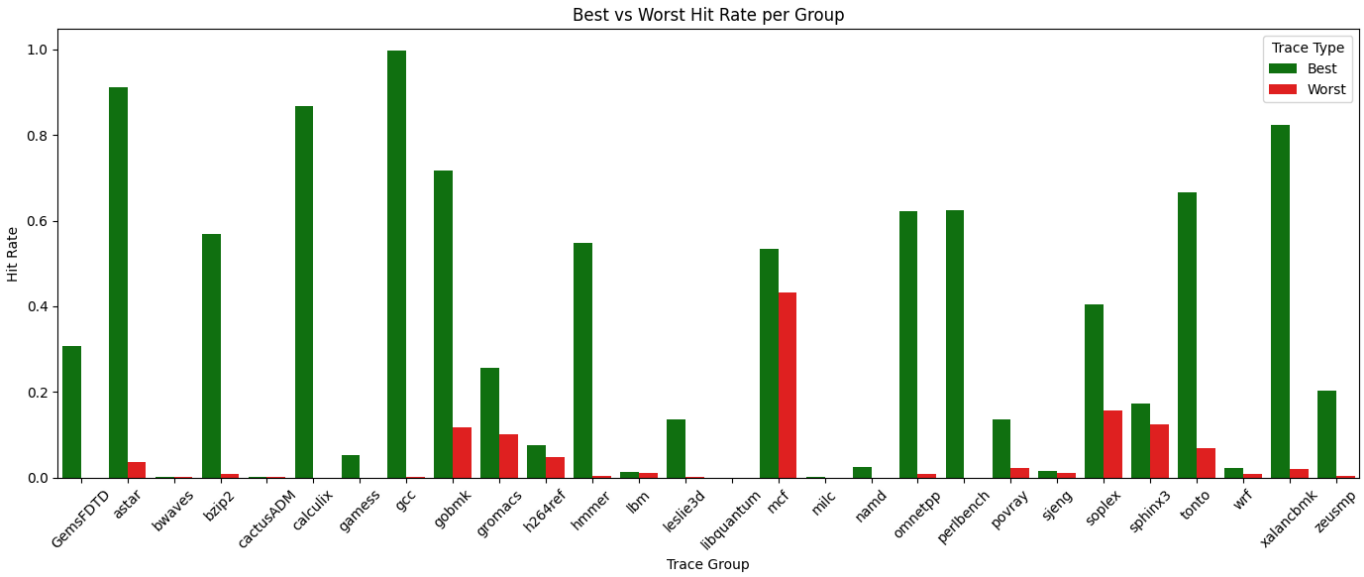

Fig. 28. dancrc2_config1: Best vs Worst per trace