CacheMind: From Miss Rates to Why — Natural-Language, Trace-Grounded Reasoning for Cache Replacement

Kaushal Mhapsekar
North Carolina State University
Electrical and Computer Engineering
Raleigh, North Carolina, USA
kmhapse@ncsu.edu

Azam Ghanbari
North Carolina State University
Electrical and Computer Engineering
Raleigh, North Carolina, USA
aghanba2@ncsu.edu

Bita Aslrousta
North Carolina State University
Electrical and Computer Engineering
Raleigh, North Carolina, USA
baslrou@ncsu.edu

Samira Mirbagher-Ajorpaz
North Carolina State University
Electrical and Computer Engineering
Raleigh, North Carolina, USA
smirbag@ncsu.edu

Abstract

Cache replacement remains a challenging problem in CPU microarchitecture, often addressed using hand-crafted heuristics that limit cache performance. Cache data analysis requires parsing millions of trace entries with manual filtering, making the process slow and non-interactive.

To address this, we introduce CacheMind, a conversational tool that uses Retrieval-Augmented Generation (RAG) and Large Language Models (LLMs) to enable semantic reasoning over cache traces. Architects can now ask natural language questions like, "*Why is the memory access associated with PC X causing more evictions*?", and receive trace-grounded, human-readable answers linked to program semantics for the first time.

To evaluate CacheMind, we present CacheMindBench, the first verified benchmark suite for LLM-based reasoning for the cache replacement problem. Using the Sieve retriever, CacheMind achieves 66.67% on 75 unseen trace-grounded questions and 84.80% on 25 unseen policy-specific reasoning tasks; with Ranger, it achieves 89.33% and 64.80% on the same evaluations. Additionally, with Ranger, CacheMind achieves 100% accuracy on 4 out of 6 categories in the trace-grounded tier of CacheMindBench. Compared to LlamaIndex (10% retrieval success), Sieve achieves 60% and Ranger achieves 90%, demonstrating that existing Retrieval-Augmented Generation (RAGs) are insufficient for precise, trace-grounded microarchitectural reasoning.

We provided four concrete actionable insights derived using CacheMind, wherein bypassing use case improved cache hit rate by 7.66% and speedup by 2.04%, software fix use case gives speedup of 76%, and Mockingjay replacement policy use case gives speedup of 0.7%; showing the utility of CacheMind on non-trivial queries that require a natural-language interface.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Computing methodologies** → **Information extraction**; *Causal reasoning and diagnostics*; *Interactive simulation*; • **Software and its engineering** → *Software notations and tools.*

Keywords: Cache replacement policy, natural-language-guided ChampSim, gem5, large language models, retrieval-augmented generation, microarchitecture-directed prefetching, compiler optimization, hardware-software co-design, machine learning for systems, LLM reasoning benchmark.

ACM Reference Format:
Kaushal Mhapsekar, Azam Ghanbari, Bita Aslrousta, and Samira Mirbagher-Ajorpaz. 2026. CacheMind: From Miss Rates to Why — Natural-Language, Trace-Grounded Reasoning for Cache Replacement. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3779212.3790136

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790136

1 Introduction

Traditional cache replacement policies are predominantly crafted through manual design processes guided by empirical workload characterization, or brute-force exploration of the architectural parameter space [18, 30, 37]. In contrast, recent state-of-the-art work learns replacement behavior directly from traces - *Hawkeye* uses Belady-guided labels with an online classifier [17], *Glider* distills deep offline models into compact online policies [34], and *PARROT* frames cache replacement as imitation learning with an offline solution [24]. Reinforcement-learning approaches such as *RLR* and *Stormbird* extend this line by incorporating bypass and access-type awareness [32, 39]. Complementary efforts study policy–prefetch interactions and cost effectiveness [27] and redesign at page granularity with translation-aware metadata (Genie Cache [23]). Whereas, predictors like Hermes target long-latency loads that indirectly affect cache bottleneck [3]. Despite this progress, learned cache policies remain difficult to deploy in practice, often due to brittle contexts, opaque reasoning, and operationally heavy, limiting adoption in production systems.

Cache replacement analysis typically relies on fixed metrics (e.g., hit/miss rates) derived from the state-of-the-art processor simulators such as trace-driven ChampSim or cycle-accurate gem5, and manual, offline trace inspection [17, 18, 34, 37] In practice, architects must sift through millions of events to answer simple "why" questions—*which PC missed on which data, under which policy, and why*. This workflow is slow, non-interactive, and hard to trust when comparing policies across workloads.

Reasoning-focused evaluation relies on curated suites with unambiguous answers, such as *GSM8K* (grade-school math), *MATH* (competition math), *DROP* (discrete reasoning over text), and *ARC* (commonsense science) [6–8, 15]. In architecture, *ChampSim* and *CRC-2* standardized trace-based evaluation for LLC policies [12], but there is no community benchmark for *trace-grounded reasoning*. Such a benchmark should: (i) ask questions about specific microarchitectural events (per-PC, per-address), (ii) tie answers to ground-truth traces, and (iii) score factual, comparative, and causal reasoning. This motivates our verified suite for cache-replacement analysis.

We argue that the next generation of microarchitectural simulators must move beyond fixed metric outputs (e.g., hit/miss rates) to engines that can answer arbitrary, interactive questions about any event in the simulated execution. CacheMind demonstrates this concept for cache replacement analysis, acting as a microarchitectural microscope that enables free-form, per-PC, per-data exploration.

We present CacheMind, the first conversational, retrieval augmented system that turns raw ChampSim traces into a *microarchitectural microscope*. Given a natural-language query, CacheMind retrieves a narrow, verifiable slice of the trace (per-PC, per-address, per-event), augments it with policy/workload context and code snippets, and produces a grounded, human-readable answer. The goal is to support interactive, explainable reasoning about replacement behavior—not just aggregate reporting.

Large language models (LLMs) are text predictors trained on vast corpora. In systems, they are increasingly used as *tool users*: given a natural-language request, the model produces structured actions (e.g., code, queries, API calls) that read data, compute, and return an answer. Prior work (e.g., Toolformer, programmatic prompting frameworks) shows how to (i) ground an LLM with external tools and data, (ii) constrain its outputs to schemas or code, and (iii) verify the result against sources [22, 31]. For architecture tasks, this means translating questions like "Which PC is causing most evictions under policy X?" into precise database slices and simple analyses that can be checked.

RAG follows a "*search-then-write*" recipe. Instead of asking a model to answer from memory, we first *retrieve* a small set of relevant facts from an external store and then *generate* the answer using only those facts. Retrieval can be keyword-based (sparse) or embedding-based (dense) [21]. Some architectures even retrieve during generation to keep long-range context fresh (e.g., RETRO) [4]. Self-RAG variants let the model request or verify evidence mid-answer [1]. In our setting, RAG boils millions of memory-trace entries down to a *tiny*, verifiable window (e.g., all accesses to a specific PC and address under a given policy) that fits within the LLM's context. This reduces hallucinations and makes answers checkable.

LLMs have limited context windows. To carry information across turns, systems maintain: (i) a *sliding buffer* of recent messages; (ii) *summaries* of older turns; and/or (iii) a *vector store* (embedding index) of past facts that can be re-retrieved when similar questions arise. In effect, this is a *conversation memory* layer. For trace analysis, buffers keep recent PCs, addresses, and policies handy; summaries help retain findings; and vector memory lets the system quickly rediscover relevant slices. External-memory LMs like RETRO perform a related trick during decoding by consulting an index rather than storing everything in the prompt [4, 28]. We augmented the Generator LLM with conversation memory buffer, turning it into an assistive chat tool. This enables reasoning across multiple queries by retaining intermediate results, previous contexts, and trace-level insights.

In CacheMind, a user query is first parsed and routed to the *retriever*, which searches an external store of trace-derived artifacts (e.g., per-PC slices, reuse statistics, and metadata). The retriever applies symbolic filters (policy, workload, PC, address) and, when applicable, semantic ranking to assemble a compact context bundle. This retrieved context conditions the *generator* (an LLM), which produces a trace-grounded, context-aware response. Figure 1 presents a detailed overview of the architecture of the proposed retrieval system.

We adopt *retrieval-augmented generation (RAG)* to ensure answers are *trace-grounded*. RAG serves two roles: (i) it fetches the exact slice(s) of simulator output from an external database needed to answer a query, and (ii) it supplies this retrieved context to the generator LLM so the response is informed by verifiable evidence rather than prior knowledge.

In ablations where the generator is held fixed and only the retriever is toggled, enabling retrieval produces quantitatively grounded answers. All results reported in this paper are evaluated under the trace-grounded (RAG-enabled) setting.

To make progress measurable, we introduce CacheMind Bench, a verified suite of 100 questions spanning factual retrieval, policy/workload comparison, arithmetic, adversarial "trick" checks, and high-level semantic analysis. Each item is trace-grounded with a single source of truth.

Full fine-tuning updates all model weights on domain data but is expensive. *Parameter-efficient fine-tuning (PEFT)* (e.g., LoRA) inserts small adapters while freezing the base model, cutting trainable parameters by orders of magnitude [16]. PEFT can inject domain style and terminology. However, if the new data are narrow, the model can become overconfident outside its niche (i.e., hallucinate more), so regularization and validation matter. We fine-tune GPT-4o-mini on domain-specific traces and also on prompts to improve precision for cache-specific queries while maintaining inference efficiency.

Instead of updating weights, we can show 1–3 short examples in the prompt that demonstrate the task format or reasoning steps. This *in-context learning* often boosts accuracy on structurally similar queries [5]. *Chain-of-thought* prompting asks the model to write intermediate steps, which can help verifiability on some tasks (with model- and task-specific gains) [36]. CacheMind performs one-shot and few-shot prompt engineering to improve LLM adaptability by passing one (one-shot) or three (few-shot) context-response example pairs to the Generator LLM.

Large Language Models (LLMs) exhibit *emergent abilities*—capabilities that are not present in smaller models and do not result from explicit supervision, but appear once the model surpasses a critical threshold in scale [35] These include multi-step mathematical reasoning, logical deduction, and causal inference across benchmarks like GSM8K [7], ARC [6], and DROP [8]. Emergence allows models to generalize to tasks they were *never explicitly trained to perform.*

Cache replacement policy reasoning exemplifies such a task. It requires tracing causality between temporally distant memory events, evaluating policy decisions across reuse distances, and linking low-level microarchitectural signals (e.g., PC, address, miss type) to high-level program semantics (e.g., function behavior, loop locality). This goes well beyond pattern classification or retrieval—it demands abstraction, explanation, and insight. No fixed heuristic or shallow model can answer arbitrary reasoning questions across diverse workloads, replacement policies, and memory access patterns without supervised training on that.

CacheMind is the first to explicitly enhance multi-source, software-to-hardware causal reasoning, unleashing generative-AI to assist architects in understanding interactions that are currently invisible and enabling the first steps toward causal, phase-aware explanations of cache behavior. CacheMind does not focus on simple facts that humans already know; but tasks that are hard for even experts to do. Modern caches exhibit complex cross-effects, prefetcher–replacement interactions (e.g., PACIPV ISCA'25) [27], consistency/coherence behavior, inclusiveness, and page-size, that defy manual reasoning. In contrast, existing reasoning benchmarks such as QuArch [29] ask static questions like "what translates a logical address into a physical?", which reflect textbook knowledge rather than multi-layer causal analysis of caches, a complex dynamic system.

In CacheMind, the PC is a pointer to the line of code that must change in software or by the compiler, or to a common pattern that can be cheaply detected and handled in hardware e.g, a perceptron. PC-information assists with designing effective hash/signature functions of useful histories given the pool of addresses with minimal hardware overhead. This paper shows such actionable insights including software and hardware fixes such as how CacheMind identifies bypass candidates and dead-on-arrival blocks and produces improved signatures to predict and bypass them, identifying stable PCs with low reuse-distance variance to improve reuse-distance predictor training, guiding targeted software prefetch insertion by pinpointing dominant miss-causing instructions, and classifying hot and cold cache sets to support focused sampling and more effective replacement policy design.

We evaluate five variants of the proposed CacheMind system, pairing its retrieval engine with different LLM backends: GPT-3.5-Turbo (legacy model), o3 (reasoning model), GPT-4o (flexible, general-purpose model), GPT-4o-mini (smaller, cheaper model), and a fine-tuned GPT-4o-mini(finetuned model). Performance is assessed across all eleven benchmark categories. Our results show CacheMind framework successfully utilizes reasoning capability of LLMs for processor design queries.

Contribution. This paper makes the following contributions:

- **CacheMind:** a query-first, RAG-based framework for cache replacement analysis that answers per-event, per-PC questions with verifiable evidence. It introduced a *Dual Retrieva*l design, *Sieve* (fast symbolic –semantic filtering) and *Ranger* (LLM-based code retrieval) to balance precision and flexibility.
- **Benchmark:** CacheMindBench, 100 verified questions evaluating factual, comparative, arithmetic and semantic reasoning over traces.
- **Evaluation:** Using CRC-2 traces and multiple policies, CacheMind achieves (90%) accuracy on generic trace-based questions and (74.9%) in reasoning tasks; *Ranger* notably improves open-ended and arithmetic cases and provides fine-grain insights. CacheMind also shows 9X better retrieval accuracy compared to LlamaIndex.

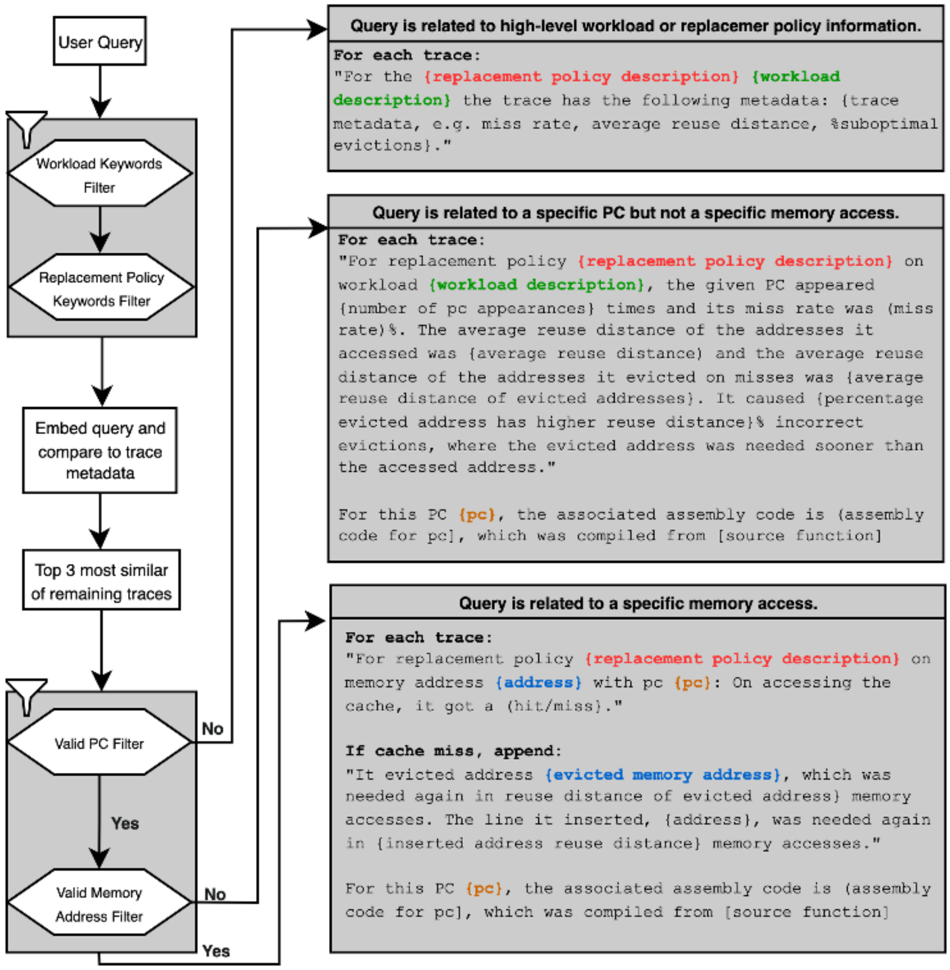


Figure 1. The method filters raw traces to a task-specific slice and returns the most informative evidence for the user's query. Old ChampSim could tell you a miss; CacheMind shows which PC missed on which data, under which policy, and why, for every event, acting as a microarchitectural *microscope* that turns raw traces into per-PC, cross-policy answers.

- **Actionable Insights:** CacheMind demonstrates utility by giving actionable insights such as examples provided: 1) bypass prediction, increasing hit rate by 7.66% and IPC by 2.04%, 2) enhanced Mockingjay RDP training, resulting in 0.7% speedup and 3) software fixes giving 76% speedup.

We argue that next-generation simulators should evolve from fixed reports to engines that can *answer arbitrary, interactive questions* about any simulated event. CacheMind demonstrates this direction for cache replacement. We plan to open-source CacheMindBench, the retrieval pipeline, and artifacts to support reproducibility and follow-up work in the community.

Paper organization. Section 2 presents an overview of the retrieval-augmented system design and its two retrievers - CacheMind-Sieve (symbolic–semantic filtering) and CacheMind-Ranger (LLM-guided generation/execution)- along with the retrieval flow.

Section 3 introduces CacheMindBench, detailing categories, scoring, and verification, and describes the external database/metadata that successfully ground queries (Figure 3, Figure 2). Section 4 outlines methodology, including simulator configuration, workloads, and policies. Section 5 reports results: category-wise accuracy, the impact of retrieval quality, Sieve vs. Ranger comparison, LLM ablations (fine-tuning; one/few-shot prompting), model sensitivity, and qualitative observations. Section 6 discusses related work. Section 7 concludes and highlights future directions.

2 Background and Related Works

Heuristics-based policies. Classical cache replacement uses simple rules that approximate locality with tiny state and constant-time decisions. The most common is *Least Recently Used (LRU)*, which evicts the line that has gone unused for the longest time; it works well when recently used data tends to be reused but breaks down on scans and weak temporal locality. *Belady's optimal* policy is an offline oracle that evicts the line whose *next use* is farthest in the future; it defines the upper bound on hit rate but requires knowledge of the future and thus is not implementable in hardware [2]. Practical online improvements include *DIP* (Dynamic Insertion Policy), which uses *set dueling* to mix insertion depths based on a few leader sets [30]; *RRIP/DRRIP*, which predicts a line's re-reference interval with small saturating counters to resist scans and adapt across phases [18]; and *SHiP*, which augments recency with simple program-context signatures to bias re-reference predictions [37]. These families remain attractive due to their small area, simple control, and predictable timing.

Learning-based policies. Newer designs learn signals that better approximate Belady while staying deployable. *Hawkeye* uses Belady-guided labels in sampled sets to classify cache-friendly lines and then applies that classifier online [17]. *Glider* mines long-range structure with deep models offline and distills the behavior to a compact online policy [34]. *Mockingjay* extends learning-based replacement by predicting continuous reuse distance using PC-indexed temporal-difference learning, enabling eviction decisions that closely track Belady's optimal ordering [33]. *PARROT* treats replacement as *imitation learning*, training an LSTM on traces to mimic an oracle and deploying a lightweight predictor [24]. These works show that information beyond pure recency (e.g., reuse-distance patterns and program context) helps at LLC scale.

Machine-learning–inspired policies. Work adjacent to replacement has shaped model and feature choices for microarchitectural predictors. The *perceptron branch predictor* demonstrated that linear models can exploit long histories efficiently in hardware [20]. *Multiperspective reuse prediction* aggregates features from program context, address bits, and control flow to guide bypass/promotion [19]. Genetic search has also been applied to feature selection and weight tuning for compact classifiers [10]. In parallel, *Reinforcement Learned Replacement (RLR)* shows how ML can craft competitive yet hardware-conscious replacement policies [32], while *Stormbird* recasts replacement as reinforcement learning, augmenting Belady-style guidance with bypass and access-type signals [39]. These lines motivate compact models and robust featurization under tight latency/area budgets.

LLM- and RAG-augmented tools. Beyond replacement policies, a growing body of systems work augments domain tools with LLMs via retrieval-augmented generation. Healthcare assistants retrieve EHR/chart fragments to answer clinician queries with verifiable context [9, 14]; enterprise pipelines build graph-structured contexts prior to generation (GraphRAG) to improve faithfulness on large corpora [26]. MLOmics provides a curated, model-ready multi-omics repository with standardized schemas and programmatic access—well suited for RAG-style assistants that need verifiable, provenance-linked data [38]. These systems share our recipe: a domain index, a retriever that assembles task-specific evidence, and a generator constrained to ground its output in the retrieved context.

3 CacheMind

3.1 System Overview

At the core of CacheMind are two complementary retrievers and a generator: (i) *Sieve*, a symbolic–semantic filter that quickly isolates the right trace window from policy/-workload/PC/address cues; and (ii) *Ranger*, an LLM-guided retriever that translates queries into executable code to fetch complex evidence when templates fall short. A generator LLM then synthesizes a concise, trace-grounded explanation.

Retrieval. CacheMind employs two complementary retrieval strategies:

1. CacheMind-Sieve, a filter-based retriever that combines symbolic and semantic filtering to efficiently extract relevant trace entries for LLM interpretation (Figure 1), and
2. CacheMind-Ranger, an LLM-based retriever that leverages large language models to dynamically generate and execute retrieval queries against the external database (Figure 3).

3.2 CacheMind-Sieve (Symbolic-Indexed Entries for Verifiable Extraction)

The CacheMind-Sieve retriever uses a multi-stage process tailored for ChampSim traces.

Figure 1 depicts CacheMind-Sieve architecture and the retrieval flow. It illustrates how CacheMind, filters raw traces to a task-specific slice and returns the most informative evidence for the user's query. It contains the block diagram of the retrieval pipeline: query parsing, policy/workload selection, symbolic PC/address filtering, optional semantic ranking, aggregation of per-PC statistics and metadata, and assembly of a trace-grounded context for the LLM.

3.2.1 Trace-Level Filtering Module. The first stage applies a sentence embedder to extract workload and replacement policy names from the user query. These are ranked by similarity and matched against the database keys to narrow the search space. For example:

- Input query: "What is the miss rate for PC 0x4037ba on the mcf workload with PARROT replacement policy?"

- Translated query: filter by `policy=PARROT`, `workload =mcf`
- Output: corresponding trace slice from the key `mcf _evictions_parrot`

3.2.2 PC and Address Filtering Module. The second stage applies symbolic filters on program counters (PCs) and memory addresses if present in the query. For example:

- Query: "Does the memory access with PC 0x401e31 and address 0x35e798a637f result in a cache hit or miss for the lbm workload under PARROT?"
- Filters applied: `program_counter == 0x401e31`, `memory _address == 0x35e798a637f`
- Result: isolates a compact trace slice from millions of entries in `lbm_evictions_parrot`.

Each database entry stores PC, address, workload, policy, eviction type, and metadata such as recency, reuse distance, and cache state.

3.2.3 Cache Statistical Expert. For PCs present in the retrieved slice, helper functions compute statistics including miss rate, access and eviction reuse distances, and percentage of bad evictions.

3.2.4 Response Generation LLM Module. The final response is assembled by combining workload information, policy description, PC-level context (function name, assembly, cache statistics), and trace metadata. This structured response grounds the subsequent LLM interpretation. An example of the prompt is shown in Figure 2

Cache Access Trace
- **PC:** 0x405832
- **Address:** 0x2a9e6a48d9d
- **Set ID:** 0b10110011101
- **Evict:** true

Cache Lines
- {"0x2c919839d9d", "0x405832"}
- {"0x2c91983e59d", "0x405832"}
- . . .

Access History
- {"0x3528e1a7d9d", "0x409228"}
- {"0x286af5fd59d", "0x409270"}
- . . .

Cache Line Scores
- {3062739738013, 180}, {3062739756445, 181}, {3707401110941, 195}

Assembly (mainSimpleSort)

```
405821:  84 c0    test   %al,%al
405832:  0f 85    jne    4032d7 <mainSimpleSort+0xbd>
405839:  eb 01    jmp    40336d <mainSimpleSort+0x153>
40583b:  90       nop
40583f:  8b 45 ecmov     -0x14(%rbp),%eax
```

Figure 2. Example trace excerpt retrieved by CacheMind

3.3 CacheMind-Ranger (Retrieval via Agentic Neural Generation and Execution Runtime)

While CacheMind-Sieve achieves high precision for structured queries, it is less effective for open-ended or dynamic queries. To address this limitation, CacheMind introduces CacheMind-Ranger, an LLM-based retriever. The Retrieval LLM is provided with:

- The retrieval objective (e.g., generate structured code to query the database),
- A detailed schema of the external database, and
- Task-specific instructions and formatting rules.

The model uses the system prompt, shown in Figure 3, to generate executable Python code that extracts relevant entries from the external trace database. This design enables handling of queries that deviate from predefined templates and leverages the LLM's ability to map natural language requests into structured data retrieval. We have used OpenAI's GPT-4o model for this purpose.

4 CacheMindBench: Novel Verified Benchmark Suite for AI Evaluation

CacheMindBench comprises 100 verified, trace-grounded questions in two tiers designed to isolate retrieval accuracy from higher-level reasoning:

- **Trace-Grounded (TG-QA) Questions (75).** Assess precise retrieval and basic logic, e.g., hit/miss classification, per-PC miss rates, cross-policy comparisons, counting, and simple arithmetic. Example: "Does the access with PC `0x401dc9` and address `0x47ea85d37f` result in a cache hit under `LRU` on `lbm`?"
- **Architectural Reasoning and Analysis (ARA) Questions (25).** Evaluate analytical and reasoning capability when provided trace context (e.g., policy/workload descriptions, per-PC slices, statistics). Example: "Why does PC `0x401e31` perform worse under `PARROT` than `Belady` on `lbm`?"

4.1 Trace-Grounded tier.

All questions are grounded in simulator outputs and verified against trace logs. This tier targets fine-grained retrieval fidelity at PC/address granularity and includes:

- **Per-access verification:** Hit/miss classification for specific {`PC`, `address`, `policy`, `workload`} tuples.
- **Per-PC miss rate analysis:** Miss-rate breakdowns per PC, often across multiple policies.
- **Cross-policy ranking:** Ordering policies by hit/miss behavior per PC or workload.
- **Counting:** Frequency of events (e.g., accesses, evictions) under specified filters.
- **Arithmetic:** Simple computations over retrieved fields (e.g., mean evicted reuse distance).

Benchmark	Category	Description	#	Representative Example
Trace-Grounded Questions (75)	Cache Hit/Miss	Determine if a specific access results in a hit or miss	30	"Does PC 0x401dc9 and address 0x47ea85d37f result in a cache hit in `lbm` under `PARROT`?"
	Miss Rate	Compute miss rate of a specific PC or workload	10	"What is the miss rate for PC 0x4037ba in `mcf` with `PARROT`?"
	Policy Comparison	Compare miss behavior across policies	15	"Which policy has the lowest miss rate for PC 0x409270 in `astar`?"
	Count	Count frequency of events (e.g., accesses, evictions)	5	"How many times did PC 0x405832 appear in `astar` under `LRU`?"
	Arithmetic	Perform arithmetic over trace statistics	10	"What is the average *evicted reuse distance* of PC 0x40170a for the `lbm` workload with `MLP`?"
	Trick Question	Catch inconsistencies or invalid assumptions	5	"Does PC 0x4037aa in `lbm` access address 0x1b73be82e3f?" (Answer: *TRICK*)
Architectural Reasoning and Analysis (25)	Microarchitecture Concepts	Answer general cache questions with logic	5	"How does increasing cache size affect miss rate? Compare increasing #sets vs. #ways."
	Code Generation	Generate code to analyze specific trace conditions	5	"Write code to compute hits for PC 0x4037ba and address 0xa3a0df3d9d in `mcf` under `LRU`."
	Replacement Policy	Analyze behavior of policies across contexts	5	"Why does Belady outperform `LRU` on PC 0x409270 in `astar`?"
	Workload Analysis	Reason about entire workload characteristics	5	"Which workload has the highest cache miss rate under `MLP`?"
	Semantic Analysis	Connect PCs to high-level program behavior	5	"Why does PC 0x4037ba have a high hit rate? Examine the assembly context and analyze."

Table 1. CacheMindBench categories and representative queries with counts and example questions for each.

- **Trick questions:** Premise checks that should be rejected (e.g., mismatched PC/workload), probing hallucination and uncertainty handling.

Scoring is binary (0/1) using exact-match verification against the trace.

4.2 Architectural Reasoning and Analysis (ARA) tier.

This subset targets trace-informed analysis and explanation, evaluating how well the system synthesizes retrieved and is categorized into:

- **Microarchitecture concepts:** Retrieval-light questions probing general understanding of cache behavior.
- **Code generation:** Requests for brief analysis code operating on the specified trace slice.
- **Policy analysis:** Explanations that link policy mechanics (e.g., recency vs. reuse distance) to observed PC-level effects.
- **Workload analysis:** Comparisons across workloads given their descriptions and retrieved metadata.
- **Semantic analysis:** Linking trace statistics to disassembly/context at a PC to explain higher-level behavior.

Answers are rubric-graded on a 0–5 scale for correctness, use of evidence, and clarity.

Table 1 summarizes the 11 categories across both tiers and their sizes in the validated dataset.

4.3 External Database and Metadata

Data organization. The external store is a Python dictionary, `loaded_data`, whose keys are trace identifiers of the form `<workload>_evictions_<policy>` (e.g., `lbm_evictions_lru`). Each key maps to a dictionary with three fields:

- `data_frame`: a *pandas DataFrame* containing per-access records (not a list of dicts).
- `metadata`: a single descriptive string summarizing whole-trace statistics (e.g., totals, rates, correlations).
- `description`: a short human-readable summary of the workload and policy.

DataFrame schema. Each `data_frame` follows a uniform schema; below we list each column and its purpose:

- `program_counter` - Instruction identity (e.g., `0x401d9b`).
- `memory_address` - Accessed memory location (e.g., `0x35e798a637f`).
- `cache_set_id` - Target cache set.
- `evict` - Access outcome (*Cache Hit*/*Cache Miss*).
- `miss_type` - Miss taxonomy (e.g., *Capacity*, *Conflict*).
- `evicted_address` - Line evicted by this access (if any).
- `accessed_address_recency` - Textual recency descriptor.
- `accessed_address_reuse_distance` - Reuse distance for the accessed line.
- `evicted_address_reuse_distance` - Reuse distance for the evicted line.

```
SYSTEM PROMPT
You are a Python code-writing assistant for analyzing cache memory
trace data. Your task is to generate Python code that extracts
string-formatted answers from a dictionary named loaded_data.

Data Structure Overview
  • loaded_data: a dictionary with keys like lbm_evictions_lru.
  • Values: "data_frame" (pandas DataFrame), "metadata" (string),
    "description" (string).
  • Workloads: astar, lbm, mcf.
  • Policies: belady, lru, mlp, parrot.
Dataframe Structure (data_frame)
Columns include (non-exhaustive):
  • program_counter, memory_address, cache_set_id, evict (Cache
    Hit/Cache Miss), miss_type
  • reuse/recency features, function_code, assembly_code
  • current_cache_lines, recent_access_history,
    cache_line_eviction_scores

Metadata (metadata)
  • A single string summarizing trace stats (accesses, misses,
    evictions, miss rate, correlations, etc.).
  • Access via loaded_data[trace_id]["metadata"].
  • Extract numbers with simple matching or regex, e.g.
    re.search(r"total misses", metadata).
  • Example:
      Cache Performance Summary: 140704 total accesses,
      133542 total misses, 94.91% miss rate,
      100.00% capacity misses, 0.00% conflict misses, 133478
      total evictions,
      87085 (65.24%) wrong evictions where evicted line has
      lower reuse distance.
      The correlation between accessed address recency and
      cache misses is 0.18.

Task Instructions
  • First check matching workload/policy; then check PC/address;
    finally fall back to metadata.
  • Return a single result string with hit/miss, reuse/recency,
    relevant metadata summary, and assembly context.
  • If nothing is found, return a clear message.
Output Rules
  • Must set result = "..." (a Python string).
  • No markdown, explanations, print, or comments.
Valid Examples
  result = f"The miss rate for PC 0x401e31 is 44.69%."
Invalid Examples
  return df["miss_rate"], print(result), result = df
```

Figure 3. Formatted system prompt summarizing the data container, dataframe schema, metadata string, task flow, and strict output rules for code generation with CacheMind-Ranger. The ability to compare, find the root cause of performance differences among designs, and to judge replacement policies accurately, is the essential first step toward automating cache policy debugging and design assistant using AI.

- `function_name` - Source-level function name mapped from PC.
- `function_code` - Short source snippet around the PC.
- `assembly_code` - Disassembly around the PC.
- `current_cache_lines` - Snapshot of (PC, address) pairs resident in the set at access time.
- `recent_access_history` - Recent (PC, address) tuples for context.
- `cache_line_eviction_scores` - Per-line scores used by the policy to decide evictions.
- `current_cache_line_addresses` - Addresses resident in the set at access time.
- `evicted_address_reuse_distance_numeric` - Reuse distance for the evicted line.
- `accessed_address_reuse_distance_numeric` - Reuse distance for the accessed line.
- `accessed_address_recency_numeric` -Access recency (number of intervening accesses).
- `is_miss` - Indicator for miss/hit (1 = miss, 0 = hit).

Trace-level metadata. The `metadata` field is a single free-form string summarizing the entire trace (e.g., total accesses/misses/evictions, miss rate, miss breakdowns, wrong-eviction ratios, correlations). It is accessed as `loaded_data[trace_id]["metadata"]` and parsed with string matching or regular expressions when numeric values are needed. A representative snippet: *"Cache Performance Summary: 140,704 total accesses, 133,542 total misses, 94.91% miss rate; 100.00% capacity misses, 0.00% conflict misses; 133,478 total evictions; 87,085 (65.24%) wrong evictions where the evicted line has lower reuse distance. Correlation between accessed-address recency and cache misses: 0.18."*

5 Methodology

Simulator. For LLC trace generation, we use ChampSim [12], a trace-based simulator widely used in prior work and by CRC-2. Building on the PARROT infrastructure [13, 24], we replay LLC accesses and emit eviction-annotated traces under Belady's optimal, LRU, and PARROT. To broaden the comparison within the same framework, we implemented and integrated an MLP-based replacement policy ourselves. The resulting traces are normalized and stored as the external database used in our experiments. The processor and memory hierarchy used in all experiments is summarized in Table 2.

We also extended CacheMind to the cycle-accurate gem5 simulator to support a broader range of software interventions as use cases for CacheMind, as well as testing other workloads and trace sources, enabling access to richer microarchitectural features such as complex instructions, access types, and cache-level information. This gem5-based version of CacheMind is used to evaluate software-level interventions, including software prefetching for high missed PCs, as reported in Section 6.3.

Workloads. We evaluate the `astar`, `lbm` and `mcf` workloads derived from SPEC CPU 2006 benchmarks used in CRC-2, producing detailed memory traces. Each trace records program counters (PCs), memory addresses, cache hit/miss outcomes, and eviction decisions. Simulations are executed for each workload with 0 warm-up instructions and 1 billion simulation instructions. We set the warm-up phase to zero and simulate the entire workload traces, as our goal is to

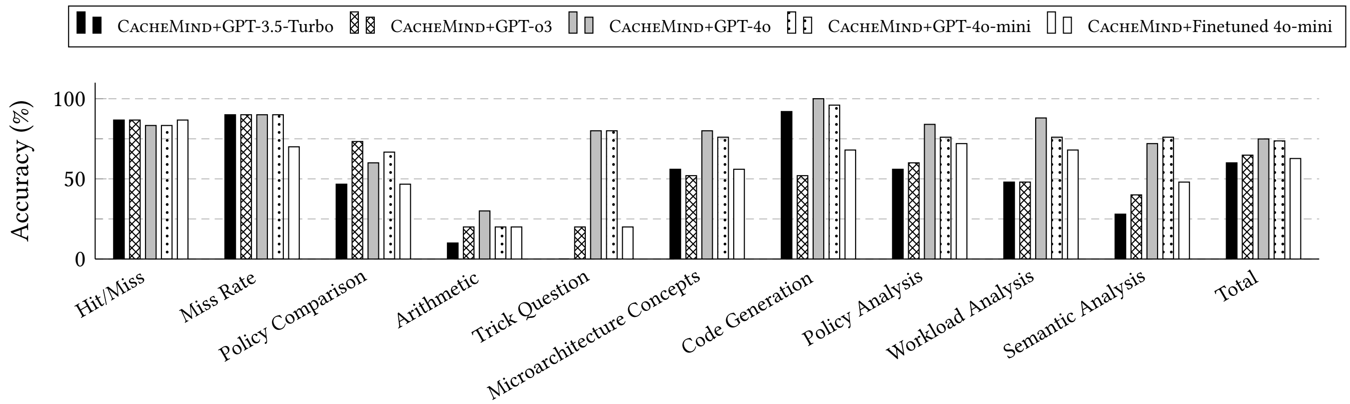


Figure 4. Accuracy of CacheMind with different LLM backends across CacheMindBench categories.

Component	**Configuration**
Processor	1 core; 4 GHz; 6-wide fetch/decode/execute; 4-wide retire; 352-entry ROB; 128-entry LQ; 72-entry SQ; bimodal branch predictor
L1 I-Cache	32 KB, 64 sets, 8 ways; 4-cycle latency; 8-entry MSHR; LRU
L1 D-Cache	32 KB, 64 sets, 8 ways; 4-cycle latency; 16-entry MSHR; LRU
L2 Cache	512 KB, 1024 sets, 8 ways; 12-cycle latency; 32-entry MSHR; LRU
LLC	2 MB, 2048 sets, 16 ways; 26-cycle latency; 64-entry MSHR; Replacement Policies - Belady's optimal, LRU, PARROT, Multi-layer Perceptron
DRAM	4 GB; DDR4-3200MT/s; 1 channel; 1 rank/channel; 8 banks/rank; 512 MB/bank; 8-bit channel; tRP = tRCD = tCAS = 12.5 ns

Table 2. Processor and Memory Configuration.

analyze cache-access events (hits/misses, evictions, reuse/recency) rather than steady-state performance statistics.

Replacement Policies. Cache behavior is measured under both baseline and learned policies. Baseline policies include Least Recently Used (LRU) and Belady's optimal, while learned policies include PARROT [24] and a Multi-Layer Perceptron (MLP) [19]. The OpenAI Gym framework [13] is used to simulate these policies and generate metadata. We extend the framework to support MLP in addition to the existing LRU, Belady, and PARROT implementations, enabling direct comparison of heuristic, oracle, and neural approaches in a unified environment.

Traces and Metadata. To enrich trace semantics, we augment ChampSim outputs with source-level metadata. Each PC is linked to its corresponding assembly and source code, while additional cache parameters (e.g., recency, reuse distance, and PC–address correlations) are extracted to form feature sets for machine learning. The resulting database comprises 4.52 GB of data, covering three workloads and four replacement policies, yielding twelve dataframes in total, which is approximately 375 MB per dataframe (1 workload and 1 policy). An extended database with potentially 8-10 replacement policies and 30 workloads could be roughly 100 GB large to be analyzed.

6 Results

6.1 Evaluation by Query Category

Figure 4 shows the breakdown results by the category of queries in CacheMindBench.

Cache Hit/Miss. All models are strong when the retrieval context is high. With 30 queries, GPT-4o and GPT-4o-mini reach 83.3%, while GPT-3.5, o3, and finetuned 4o-mini each score 86.7%. These are direct lookups once the PC+address+policy+workload tuple is isolated.

Miss Rate. On 10 queries, GPT-3.5/o3/GPT-4o/GPT-4o-mini achieve 90.0%; the finetuned 4o-mini yields 80.0%. Computing a per-PC or per-workload miss rate remains easy when the relevant slice is retrieved.

Policy Comparison. When asked to compare policies for a given PC or workload (15 queries), performance diverges: o3 leads at 73.3%, GPT-4o-mini at 66.7%, GPT-4o at 60.0%, while GPT-3.5 and finetuned 4o-mini are at 46.7%. These require selecting and ranking across multiple retrieved statistics rather than returning a single value.

Count. All models score 0/5. Pure counting over low-context windows is brittle: a single missed filter or failure to iterate the entire slice yields an incorrect result.

Arithmetic. Numerical aggregation beyond simple rates remains challenging with low context (10 queries): GPT-4o reaches 30.0%, GPT-3.5 10.0%, o3/GPT-4o-mini/finetuned 4o-mini each 20.0%. These typically require multiple filtered values and a secondary computation (e.g., averaging reuse distances).

Trick Questions. Epistemic checks (5 queries) separate robust models: GPT-4o and GPT-4o-mini each reach 80.0%, o3 20.0%, and GPT-3.5 0.0%. Example: *"Does PC 0x4037aa in* `lbm` *access 0x1b73be82e3f under PARROT?"* is invalid because

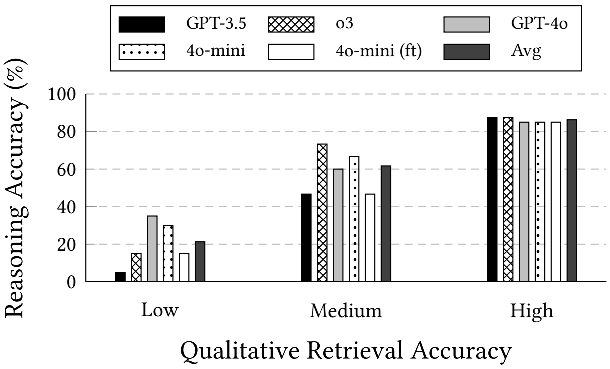


Figure 5. Accuracy across retrieval-context quality (Low/Medium/High) for each backend paired with CacheMind. It shows that the retrieval quality is the precondition for cache replacement policy high level reasoning.

that PC appears only in `mcf`; models must reject the false premise rather than guess.

Simple Reasoning. On short, concept-style microarchitecture questions (5 items, scored 0–5 each), GPT-4o reaches 80.0% (20/25), GPT-4o-mini 76.0%, GPT-3.5 56.0%, and o3 52.0%. These do not strictly need retrieval but reward stable domain knowledge.

Code Generation. For retrieval-and-analysis code (5 items, 0–5 scoring), GPT-4o attains 100.0% (25/25) and GPT-4o-mini 96.0%. GPT-3.5 is surprisingly strong at 92.0%, while o3 and finetuned 4o-mini trail at 52.0% and 68.0%. Clear schemas and narrow APIs appear to favor code synthesis.

Replacement Policy Analysis. Causal questions (5 items, 0–5) such as *"Why does Belady outperform LRU on PC 0x409270 in* `astar`*?"* require linking reuse patterns to policy rules. GPT-4o leads at 84.0%, followed by GPT-4o-mini 76.0%, finetuned 4o-mini 72.0%, and o3 60.0%; GPT-3.5 is at 56.0%. Larger models better connect retrieved evidence to policy logic.

Workload Analysis. Whole-workload behavior (5 items, 0–5) shows GPT-4o at 88.0%, GPT-4o-mini 76.0%, finetuned 4o-mini 68.0%, while GPT-3.5 and o3 both reach 48.0%. Summarizing patterns across many PCs remains sensitive to retrieval coverage.

Semantic Analysis. Linking trace events to code intent (5 items, 0–5) is hardest: GPT-4o-mini attains 76.0%, GPT-4o 72.0%, finetuned 4o-mini 48.0%, o3 40.0%, and GPT-3.5 28.0%. Models must connect PC-level locality with disassembly/-function context, going beyond symbolic lookups.

Finetuning. Contrary to expectation, Figure 4 shows that fine-tuned GPT-4o-mini did not outperform the non-fine-tuned variant. In reasoning categories (Trick Question, Semantic Analysis), fine-tuning amplified hallucinations (20% vs. 80% and 48% vs. 76% respectively). This aligns with recent findings [11] showing that fine-tuning narrows generalization and increases hallucination risk in scientific domains.

Question Category: Cache Hit/Miss

User Query:
`Does the memory access with PC 0x401dc9 and address 0x47ea85d37f result in a cache hit or cache miss for the lbm workload and PARROT replacement policy?`

One-shot Example provided to the Generator LLM via prompt:

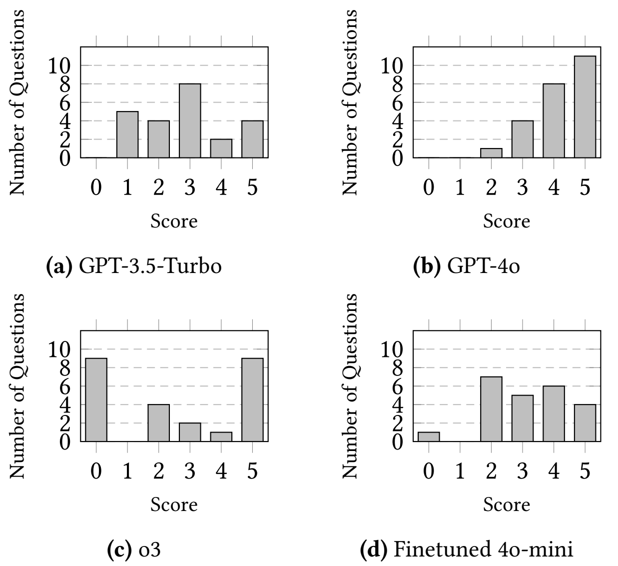
Context:
```
For policy LRU on workload lbm . . . at PC
0x401dc9 and address 0x47ea85d37f:
Cache result: Cache Miss
Evicted address: 0x19e02d19b7f (needed again in
2304 accesses), Inserted address needed again
in 3132 accesses.
Answer the following question: Does the
memory access with PC 0x401dc9 and address
0x47ea85d37f result in a cache hit or cache miss
for the lbm workload and LRU replacement policy?
The correct answer is,
```
Response: `Cache Miss`

LLM Output: `Cache Hit` or `Cache Miss`, grounded in the retrieved trace context.

Figure 6. A one-shot prompt example passed along with a Cache Hit/Miss user query to the generator LLM and the correct CacheMind's output.

One and Few-shot Prompting. Figure 6 shows an example of the CacheMind one-shot prompt. Based on our results, we observed that overall, one or few-shot prompting does not improve system performance significantly. For few questions, when the generator LLM is not provided with sufficient context, it takes the context from the example as its own and gives incorrect answers. On the other hand, the given examples help the generator identify and assess trick questions better than zero-shot prompting.

LLM Variation. Across all 11 benchmark categories, GPT-4o paired with CacheMind achieves the highest weighted total score of 74.9%, followed by o3 (64.8%), finetuned 4o-mini (62.7%), and GPT-3.5 (60.0%). GPT-3.5 serves as a weak but useful baseline, while o3 shows strong reasoning but inconsistent response coverage (Figure 7).



(a) GPT-3.5-Turbo **(b)** GPT-4o

(c) o3 **(d)** Finetuned 4o-mini

Figure 7. Distribution of reasoning scores by model. Each panel shows the frequency of scores {0,…,5} for the corresponding backend when paired with CacheMind: showing how many questions each model scored from 0 to 5

.

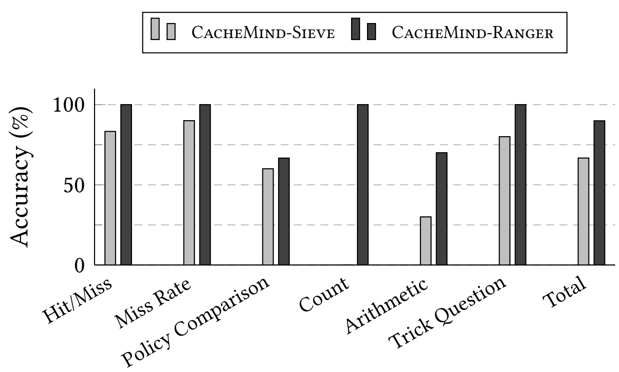


Figure 8. Accuracy of the two proposed Retrievers - CacheMind-Sieve and CacheMind-Ranger across CacheMindBench categories. CacheMind-Ranger achieves an average score of 89.33% compared to 66.67% of CacheMind-Sieve on Trace-Grounded Questions. This highlights the potential of dynamic LLM-based retrievers.

6.2 CacheMind vs. Conventional Retrieval/RAGs

Sieve exemplifies a conventional retrieval approach, it performs poorly even on primitive tasks (Figure 8), confirming that a static program cannot generalize to the diverse, arbitrary decomposition-based questions CacheMind is designed to handle. Figure 9 compares retrieval accuracy across ten evaluation queries spanning five categories of trace-grounded questions. LlamaIndex achieves 10% accuracy, Sieve achieves 60%, and Ranger achieves 90%. These results show that CacheMind significantly improves retrieval accuracy for non-trivial natural-language queries over microarchitectural traces.

This failing of prior RAGs stems from fundamental limitations of embedding-based RAGs. Conventional frameworks such as LlamaIndex [25] rely on c*osine similarity* over embeddings, which works well for text-heavy datasets but fails for microarchitectural traces where records differ only by small numerical or bit-level changes. As a result, embedding-based retrievers often return imprecise or incorrect context. To address this issue, we first designed CacheMind-Sieve, a filter-based retriever that performs explicit checks on structured attributes such as program counter (PC), memory address, replacement policy, and workload. While Sieve improves accuracy, it requires anticipating query patterns in advance, limiting its flexibility similar to conventional database retrievals.

CacheMind-Ranger removes this limitation by using the code-generation capability of large language models to dynamically generate database queries at inference time, enabling accurate retrieval for previously unseen query formulations.

6.3 Actionable Insights

This section provides concrete actionable insights derived using CacheMind showing the utility of CacheMind on non-trivial queries that require a natural-language interface directly providing speedup.

Mockingjay Use Case. The Mockingjay [33] replacement policy approximates Belady's Optimal (MIN) policy by predicting the reuse distance of cache lines in accessed cache set and evicting the line with the furthest reuse distance in the future. It uses a multiclass PC-based reuse distance predictor and orders cache lines by their estimated time of reuse (ETR). Figure 10 shows the chat session using which we can identify PCs with different amounts of variance in their ETR value, which in turn helps us identify PCs that have more predictable reuse distance values. We can use these "stable" PCs to train the Reuse Distance Predictor with less noisy samples. Based on the above insight, we changed the Mockingjay source code to train only on the list of stable PCs identified by CacheMind. We compared the results of both modified and unmodified policies on ChampSim, keeping everything else the same, to observe that stable training increased IPC from 0.47698 to 0.480307 (0.7% speedup) over milc workload. We chose milc because Mockingjay does worse than Hawkeye.

Signature Optimization for Bypass Logic. On the *mcf* workload, bypassing the PCs identified by CacheMind improves performance under an LRU policy. Specifically, bypassing ten PCs increases the cache hit rate from 25.06% to 26.98% (+7.66% relative) and improves IPC from 0.047905 to 0.048809, corresponding to a 2.04% speedup. All other simulation parameters are kept unchanged.

LlamaIndex	SIEVE	RANGER
Example Query: *When PC* `0x409270` *and address* `0x2bfd401c63f` *is accessed on the astar workload with LRU policy, does the cache hit or miss?*		
Example Retrieved Context: 0.6483328952344215 {'trace_id': 'astar_evictions_lru', 'doc_type': 'trace_summary'} DESCRIPTION: Replacement Policy: <Policy Description> Workload: <Workload Description> – 0.6140775745472963 TRACE_ID: astar_evictions_lru **program_counter=0x409538, memory_address=0x2bfd401b693,** evict=Cache Miss, – 0.6140775745472963 TRACE_ID: astar_evictions_lru **program_counter=0x4090c3, memory_address=0x2bfd401caf2,** evict=Cache Miss, . . . **Exact PC, Memory Address match not found**	Workloads involved: <Workload Description> Policies involved: <Policy Description> Assembly code snippet for PC 0x409270 (representative instructions): <Assembly Code> Source Function: _ZN7way2obj11createwayarERP6pointtRi **LRU + astar @ PC 0x409270, addr 0x2bfd401c63f:** **Cache result: Cache Miss** Belady + astar: . . . MLP + astar: . . . Evicted address: 0x31232a3ee3f (needed again in 3648 accesses). Inserted address needed again in 6287 accesses.	**Result: Cache Miss for PC 0x409270 and addr 0x2bfd401c63f** (workload: astar, policy: LRU). Function: _ZN7way2obj11createwayarERP6pointtRi Assembly: <Assembly Code> Cache Performance Summary: 140704 total accesses; 133542 total misses; 94.91% miss rate; . . . Replacement Policy: <Policy Description> Workload: <Workload Description>
Evaluation: *Queries with correct retrieved context = **1/10 (10%)*** *Average time for retrieval = **36.62 seconds***	*Queries with correct retrieved context = **6/10 (60%)*** *Average time for retrieval = **3.65 seconds***	*Queries with correct retrieved context = **9/10 (90%)*** *Average time for retrieval = **4.43 seconds***

Figure 9. Comparison of retrieved context across LlamaIndex, SIEVE, and RANGER for the same query.

User: Mockingjay uses PC-based reuse-distance prediction; suggest ideas to improve performance.
Assistant: *Summary of ideation: cluster PCs by ETR variance; train RDP on stable samples.*

User: List all unique PCs in the trace.
Assistant: `4184b0, 4184c0, 418502, . . . , 417f58`.

User: Compute mean ETR per PC.
Assistant: `PC_i` → `mean_ETR(PC_i)` (collapsed).

User: Compute standard deviation of ETR per PC.
Assistant: `PC_i` → `std_ETR(PC_i)` (collapsed).

User: Group PCs by ETR variance.
Assistant: LowVar: {`413930, . . .`}, MedVar: {`41391c, . . .`}, HighVar: {`413948, . . .`}.

Figure 10. Identifying PCs with low ETR variance (more predictable reuse) using `CacheMind` for enhancing Mockingjay.

Figure 11 shows the CACHEMIND queries used to identify PCs in the *astar*, *lbm*, and *mcf* traces that exhibit low hit rates and large reuse distances under Belady's optimal policy. Such PCs are frequently evicted even by the optimal policy and compete with other PCs that demonstrate higher reuse. This suggests that bypassing these accesses can reduce cache pollution.

To validate this insight, we ran ChampSim simulations using the same workloads and simpoints used to generate the CACHEMIND traces. We implemented a simple conditional bypass in the LRU replacement logic that skips cache insertion for the identified PCs. CACHEMIND also reveals complex access behavior in the *lbm* workload, where PCs with high reuse are interleaved with PCs exhibiting poor reuse, as shown in Figure 11. This observation helps understand why PC-signature based policies such as SHiP [37] outperform their predecessor policies like RRIP [18] on *lbm*, as they learn reuse behavior directly at PC granularity.

PC-Information Applications, Software Intervention and Prefetcher Use Case. Using the PC identified by CACHEMIND, adding a software prefetch to a pointer-chasing microbenchmark increases IPC from 0.131452 to 0.231261. Figure 12 shows the CACHEMIND queries used to identify the PC responsible for the majority of cache misses in the

User: List all PCs in the mcf workload.
Assistant: `0x401380, 0x401384, 0x40138f, . . .`

User: For mcf + Belady, compute average accessed-address reuse distance, cache hit rate and hit count per PC; sort in descending order in terms of reuse distance.
Assistant: `PC` → `(AvgReuseDist, HitRate, HitCount)` (e.g., `0x402ea8`, `0x4037aa`, ...).

User: Identify PCs suitable for bypassing to improve IPC.
Assistant: Bypass candidates: `0x4037aa, 0x402ea8, 0x401380, 0x40138f, . . .`
(high reuse distance and/or near-zero hit rate).

. . .

User: Based on the reuse and hit statistics for lbm workload, what insights can you derive?
Assistant: In lbm, interleaved streaming accesses appear to push useful lines to LRU positions long before reuse. Implications and how to approximate Belady for lbm:

- Use PC-aware reuse prediction.
- Favor RRIP/DIP-like insertion/aging.
- Consider bypass and scan detection.

Net insight: lbm's working set interleaves strong temporal reuse with heavy scan interference. Belady's advantage comes from separating these classes at fine granularity and preserving the long-horizon reuses that recency- and heuristic-driven policies fail to capture.

Figure 11. Condensed transcript of an LLM-assisted analysis for identifying bypassable PCs in the `mcf` workload. Long numerical outputs are collapsed for clarity.

microbenchmark. The benchmark is designed to generate misses from a single dominant load instruction at an initially unknown PC, which is recovered using CacheMind. Based on this insight, we modified the microbenchmark to insert a built-in C software prefetch instruction that prefetches future addresses in the pointer-chasing array according to the observed access pattern, leading to significant performance gains. Simulating this modified source code demonstrates that CacheMind can identify actionable PCs and directly guide performance-improving software-hardware co-design, intervention and fixes.

Set Hotness Analysis Use Case. Figure 13 shows how CacheMind is used to identify hot and cold cache sets from access traces. This analysis allows users to quickly localize sets that dominate cache activity, which is difficult to infer directly from raw traces. Knowledge of set hotness is important for understanding cache behavior and replacement effectiveness. In sampled-set LLC policies, such as those used in prior work, learning eviction behavior from hot sets is more effective than uniform random sampling, since hot sets generate more misses and replacement decisions.

User: List all unique PCs in the given trace.
Assistant: `0x400444, 0x400701, 0x400709, . . .`

User: From the unique PCs, identify the PC causing the most cache misses.
Assistant: `0x400512`.

User: What is the miss rate of PC `0x400512`?
Assistant: **74.73%** miss rate.

Figure 12. Condensed transcript of an interactive analysis identifying the most miss-prone PC in a cache access trace. Long PC lists are collapsed for clarity.

By providing a natural-language interface to query set-level activity, CacheMind supports targeted analysis of cache behavior and enables more informed sampling and replacement policy design.

User: For astar workload and Belady replacement policy, could you list unique cache sets in ascending order?
Assistant: `6, 35, 38, 53, 67, . . . , 1940, 1967, 1973.`

User: Based on the list of cache sets accessed for astar + Belady, could you find the number of cache hits and hit rate?
Assistant: `Set S_i: hits(S_i), hit rate(S_i)` for $i = 1 \ldots K$ (full per-set table).
e.g., Set 6: 863 hits, 38.32%.

User: Identify 5 hot and 5 cold sets by hit rate.
Assistant: `Hot Sets = {332, 1424, 1604, 1086, 1456}, Cold Sets = {35, 53, 143, 362, 1211}.`

User: For astar workload and LRU replacement policy, could you list unique cache sets in ascending order?

. . .

Assistant: `Hot Sets = {332, 1604, 1424, 1456, 1086}, Cold Sets = {287, 1789, 113, 157, 488}.`

User: Compare hot sets (LRU vs Belady) and derive insights.
Assistant: Hot set identity likely overlaps, but Belady amplifies hotness:

- Hot sets arise from intrinsic workload locality.
- Belady avoids premature evictions, increasing hit concentration.

Figure 13. Demo chat session on CacheMind for identifying hot and cold cache sets and deriving insights.

Beyond aggregate accuracy, the experiments reveal deeper findings:
Belady vs. PARROT. Across the three benchmarks, astar, lbm, and mcf, PARROT outperformed Belady for 2, 5, and 3 PCs respectively, in terms of hit rate. Although this may

seem counterintuitive, it highlights a key distinction: Belady (OPT) makes globally optimal decisions over the entire trace, whereas PARROT learns PC-local heuristics. Consequently, OPT exploits structure beyond PC identity that PARROT and other PC-based policies fail to capture. OPT provides an upper bound on the total cache hit rate, implying that any practical replacement policy achieves a hit rate less than or equal to OPT when aggregated over all accesses. However, this global guarantee does not extend to individual program counters (PCs): the hit rate of OPT for accesses associated with a given PC may be lower than that of a practical replacement policy, as observed.

Distribution of Model Behavior. Score distributions (Figure 7) reveal that o3 is bimodal, excelling or failing completely, while GPT-4o shows consistent competence across categories. GPT-3.5-Turbo and Finetuned GPT-4o-mini also show uniform distribution with a lower mean.

Context can suppress latent knowledge. In a *Microarchitecture Concepts* question that required decomposing a memory address into *offset/index/tag* given the cache geometry, all Generator models answered incorrectly when paired with CacheMind-Sieve (which surfaced partial/ambiguous metadata). When paired with CacheMind-Ranger, which retrieved the correct line size and set count, the same models produced the correct decomposition. This suggests the LLMs possess the requisite background knowledge but are highly sensitive to retrieval precision: inappropriate context can override otherwise correct internal reasoning.

Trust and Epistemic Robustness. Trick Question performance suggests that only GPT-4o achieves the rejection behavior necessary for deployment in hardware verification workflows. Other models hallucinate under adversarial phrasing.

Semantic Recovery. Linking low-level cache events to high-level code structures remains the hardest benchmark. This opens new opportunities for applying CacheMind as a co-analysis tool in architecture research (Figure 2 shows an example of such capability).

7 Conclusion

CacheMind reframes cache analysis as *query-first* reasoning over traces, coupling a symbolic–semantic retriever with LLMs to turn raw events into verifiable answers. CacheMind is a microarchitectural microscope that magnifies cache events to per-PC, per-data resolution. The links each access to its program semantics and shows exactly which data each PC missed or evicted under every replacement policy. This fine-grained lens reveals patterns and behaviors that aggregate metrics obscure, enabling cross-policy comparisons and targeted design evaluation.

Across 100 verified questions, CacheMind achieves strong accuracy, especially on hit/miss and miss-rate queries, and reveals that retrieval precision is the dominant factor, while large, reasoning-capable LLMs further lift performance. Our LLM-based retriever (CacheMind-Ranger) improves open-ended tasks and mitigates failures seen with a purely filter-driven pipeline (CacheMind-Sieve). Fine-tuning offered limited benefit, whereas grounded context consistently reduced hallucinations and enabled policy/workload analysis.

Code and Data Availability

The code for CacheMind and the suite of benchmark questions, CacheMindBench are open-source and available at: https://github.com/kaushal1803/cachemind

References

[1] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection. (2023). arXiv:2310.11511 [cs.CL] https://arxiv.org/abs/2310.11511

[2] L. A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* 5, 2 (1966), 78–101. doi:10.1147/sj.52.0078

[3] Rahul Bera, Konstantinos Kanellopoulos, Shankar Balachandran, David Novo, Ataberk Olgun, Mohammad Sadrosadati, and Onur Mutlu. 2022. Hermes: Accelerating Long-Latency Load Requests via Perceptron-Based Off-Chip Load Prediction. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Chicago, Illinois, USA). IEEE/ACM, 1–18. doi:10.1109/MICRO56248.2022.00015

[4] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. 2022. Improving language models by retrieving from trillions of tokens. arXiv:2112.04426 [cs.CL] https://arxiv.org/abs/2112.04426

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL] https://arxiv.org/abs/2005.14165

[6] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge. (2018). arXiv:1803.05457 [cs.AI] https://arxiv.org/abs/1803.05457

[7] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. (2021). arXiv:2110.14168 [cs.LG] https://arxiv.org/abs/2110.14168

[8] Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. DROP: A Reading Comprehension Benchmark Requiring Discrete Reasoning Over Paragraphs. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran,

and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 2368–2378. doi:10.18653/v1/N19-1246

[9] Farieda Gaber, Maqsood Shaik, Vedran Franke, and Altuna Akalin. 2024. Evaluating large language model workflows in clinical decision support: referral, triage, and diagnosis. *medRxiv* (2024). doi:10.1101/2024.09.27.24314505

[10] Elba Garza, Samira Mirbagher-Ajorpaz, Tahsin Ahmad Khan, and Daniel A. Jiménez. 2019. Bit-level perceptron prediction for indirect branches. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/3307650.3322217

[11] Zorik Gekhman, Gal Yona, Roee Aharoni, Matan Eyal, Amir Feder, Roi Reichart, and Jonathan Herzig. 2024. Does Fine-Tuning LLMs on New Knowledge Encourage Hallucinations?. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 7765–7784. doi:10.18653/v1/2024.emnlp-main.444

[12] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. arXiv:2210.14324 [cs.AR] https://arxiv.org/abs/2210.14324

[13] Google Research. 2025. google-research: cache_replacement. https://github.com/google-research/google-research/tree/master/cache_replacement GitHub repository.

[14] Stefan Hegselmann, Georg von Arnim, Tillmann Rheude, Noel Kronenberg, David Sontag, Gerhard Hindricks, Roland Eils, and Benjamin Wild. 2025. Large Language Models are Powerful Electronic Health Record Encoders. arXiv:2502.17403 [cs.LG] https://arxiv.org/abs/2502.17403

[15] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring Mathematical Problem Solving With the MATH Dataset. (2021). arXiv:2103.03874 [cs.LG] https://arxiv.org/abs/2103.03874

[16] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs.CL] https://arxiv.org/abs/2106.09685

[17] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. IEEE Press, Seoul, Republic of Korea, 78–89. doi:10.1109/ISCA.2016.17

[18] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 60–71. doi:10.1145/1816038.1815971

[19] Daniel A. Jiménez and Elvira Teran. 2017. Multiperspective reuse prediction. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 436–448. doi:10.1145/3123939.3123942

[20] Daniel A. Jiménez and Calvin Lin. 2001. Dynamic Branch Prediction with Perceptrons. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 197–206. doi:10.1109/HPCA.2001.903263

[21] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. arXiv:2004.04906 [cs.CL] https://arxiv.org/abs/2004.04906

[22] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. arXiv:2310.03714 [cs.CL] https://arxiv.org/abs/2310.03714

[23] Youngin Kim and William J. Song. 2024. Genie Cache: Non-Blocking Miss Handling and Replacement in Page-Table-Based DRAM Cache. In *Proceedings of the 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 983–996. doi:10.1109/MICRO61859.2024.00076

[24] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. 2020. An imitation learning approach for cache replacement. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. JMLR.org, Article 579, 11 pages. doi:10.1145/3352460.3358319

[25] Jerry Liu. 2022. *LlamaIndex*. https://github.com/jerryjliu/llama_index

[26] Congmin Min, Rhea Mathew, Joyce Pan, Sahil Bansal, Abbas Keshavarzi, and Amar Viswanathan Kannan. 2025. Efficient Knowledge Graph Construction and Retrieval from Unstructured Text for Large-Scale RAG Systems. arXiv:2507.03226 [cs.AI] https://arxiv.org/abs/2507.03226

[27] Saba Mostofi, Setu Gupta, Ahmad Hassani, Krishnam Tibrewala, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. 2025. Light-weight Cache Replacement for Instruction Heavy Workloads. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 1005–1019. doi:10.1145/3695053.3730993

[28] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560 [cs.AI] https://arxiv.org/abs/2310.08560

[29] Shvetank Prakash, Andrew Cheng, Jason Yik, Arya Tschand, Radhika Ghosal, Ikechukwu Uchendu, Jessica Quaye, Jeffrey Ma, Shreyas Grampurohit, Sofia Giannuzzi, Arnav Balyan, Fin Amin, Aadya Pipersenia, Yash Choudhary, Ankita Nayak, Amir Yazdanbakhsh, and Vijay Janapa Reddi. 2025. QuArch: A Question-Answering Dataset for AI Agents in Computer Architecture. arXiv:2501.01892 [cs.AR] https://arxiv.org/abs/2501.01892

[30] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. *SIGARCH Comput. Archit. News* 35, 2 (June 2007), 381–391. doi:10.1145/1273440.1250709

[31] Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: language models can teach themselves to use tools. 13 pages. doi:10.5555/3666122.3669119

[32] Subhash Sethumurugan, Jieming Yin, and John Sartori. 2021. Designing a Cost-Effective Cache Replacement Policy using Machine Learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 291–303. doi:10.1109/HPCA51647.2021.00033

[33] Ishan Shah, Akanksha Jain, and Calvin Lin. 2022. Effective Mimicry of Belady's MIN Policy. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 558–572. doi:10.1109/HPCA53966.2022.00048

[34] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying Deep Learning to the Cache Replacement Problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO-52)*. Association for Computing Machinery, New York, NY, USA, 413–425. doi:10.1145/3352460.3358319

[35] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent Abilities of Large Language Models. arXiv:2206.07682 [cs.CL] https://arxiv.org/abs/2206.07682

[36] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL] https://arxiv.org/abs/2201.11903

[37] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. 2011. SHiP: signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO-44)*. Association for Computing Machinery, New York, NY, USA, 430–441. doi:10.1145/2155620.2155671

[38] Ziwei Yang, Rikuto Kotoge, Xihao Piao, Zheng Chen, Lingwei Zhu, Peng Gao, Yasuko Matsubara, Yasushi Sakurai, and Jimeng Sun. 2025. MLOmics: Cancer Multi-Omics Database for Machine Learning. arXiv:2409.02143 [q-bio.GN] https://arxiv.org/abs/2409.02143

[39] Ho Jung Yoo, Jeong Hun Kim, and Tae Hee Han. 2023. RL-Based Cache Replacement: A Modern Interpretation of Belady's Algorithm With Bypass Mechanism and Access Type Analysis. *IEEE Access* 11 (2023), 145238–145253. doi:10.1109/ACCESS.2023.3346790