ArchAgent: Agentic AI-driven Computer Architecture Discovery

Raghav Gupta[1†], Akanksha Jain[2], Abraham Gonzalez[2], Alexander Novikov[3], Po-Sen Huang[3], Matej Balog[3], Marvin Eisenberger[3], Sergey Shirobokov[3], Ngân Vũ[3], Martin Dixon[2], Borivoje Nikolić[1], Parthasarathy Ranganathan[2], Sagar Karandikar[1†]

[1]University of California, Berkeley
[2]Google [3]Google DeepMind

raghavgupta@berkeley.edu,{bora,sagark}@eecs.berkeley.edu
{avjain,abegonzalez,mgdixon,parthas}@google.com
{anovikov,posenhuang,matejb,meisenberger,shirobokov,nganvu}@google.com

Abstract

Agile hardware design flows are a critically needed force multiplier to meet the exploding demand for compute. Recently, agentic generative artificial intelligence (AI) systems have demonstrated significant advances in algorithm design, improving code efficiency, and enabling discovery across scientific domains.

Bridging these worlds, we present ArchAgent, an automated *computer architecture discovery* system built on AlphaEvolve. We show ArchAgent's ability to automatically design/implement state-of-the-art (SoTA) cache replacement policies (architecting new mechanisms/logic, not only changing parameters), broadly within the confines of an established cache replacement policy design competition.

In two days and without human intervention, ArchAgent generated a policy achieving a 5.3% IPC speedup improvement over the prior SoTA on public multi-core Google Workload Traces. On the heavily-explored single-core SPEC 2006 workloads, in only 18 days ArchAgent generated a policy showing a 0.9% IPC speedup improvement over the existing SoTA (a similar "winning margin" as reported by the existing SoTA). Comparing against the effort involved in developing the prior SoTA policies, ArchAgent achieved these gains 3-5× faster than humans.

Agentic flows also create an opportunity once a hardware system is deployed, which we call "post-silicon hyperspecialization". This means having the agent tune runtime-configurable parameters exposed in hardware policies to further align the policies with a specific workload (mix). Exploiting this, we demonstrate a 2.4% IPC speedup improvement over prior SoTA on the SPEC 2006 workloads.

Since ArchAgent is a first-of-its-kind architectural discovery system, we also outline lessons learned and broader implications for computer architecture research in the era of agentic AI. For example, we demonstrate the phenomenon of "simulator escapes", where the agentic AI flow discovered and exploited a loophole in a popular microarchitectural simulator—a consequence of the fact that these research tools were designed for a (now past) world where they were exclusively operated by humans acting in good-faith.

1 Introduction

The computing industry continues to battle the rift between the exploding demand for compute and the need for human-time-intensive per-domain specialization to make up for the plateau in classical hardware scaling techniques. Continued efforts in agile hardware design, more recently driven by the infusion of machine learning (ML), are welcomed to help bridge this gap. While inroads have been made in applying ML to RTL-to-chip flows, there is less focus on enabling automated "discovery" in earlier stages of the hardware design process, beyond using ML to explore parameterized design spaces. In this paper, we answer the following: Can modern generative AI tools help computer architects in early discovery, ideation, and pathfinding, so that we can *more quickly* architect *a greater number* of specialized architectures?

Recently, agentic flows based on large language models (LLMs) have emerged as useful tools to automate algorithm design, software systems design, software efficiency, and scientific discovery [11, 47]. Guided by high-level prompting from a human, such a flow can generate, implement, and evaluate ideas much faster

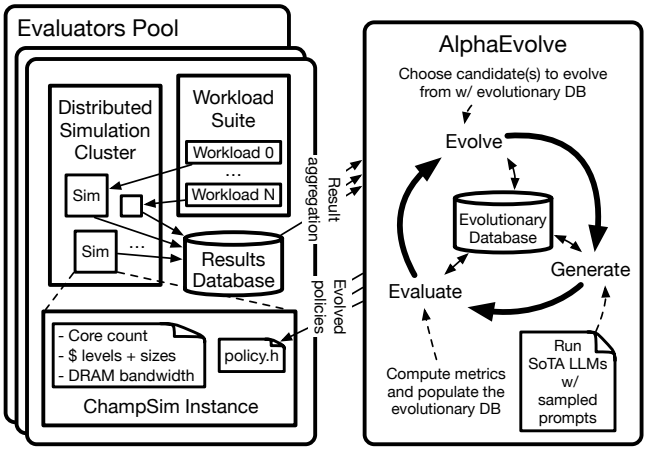

Figure 1: High-level system diagram of ArchAgent, our agentic-AI-based computer architecture discovery system. In this example, novel cache replacement policy candidates are automatically designed/implemented by AlphaEvolve in ChampSim, a popular trace-based microarchitectural simulator. ChampSim is then compiled and run with a specified workload suite (e.g., SPEC) to evaluate the new policy on a target metric (e.g., IPC). This process continues iteratively, with ArchAgent continually proposing and evaluating new logic/mechanisms within the policy.

[†]Work done while the author was also affiliated with Google.

than a human designer. This shifts the designer's focus to problem formulation, creative ideation, and experimental design.

The basic premise of ArchAgent is straightforward: Agentic, evolutionary ML-based tools can automatically and iteratively express and evaluate new concepts as code, so we should have these tools write code to express new architectures in the context of our community standard (micro)architectural simulators. Closing the loop, since our simulators can provide quantitative feedback, these agents can work iteratively to improve their solutions with new mechanisms/logic developed on-the-fly by an LLM. In the rest of this paper, we discuss the benefits and tradeoffs of this approach and express a call-to-action for our community infrastructure to enable us to get the most out of these agentic AI tools.

More concretely, we make the following key contributions:

1.0.1 The ArchAgent system. We design and implement ArchAgent, an agentic AI system that automates computer architecture discovery. ArchAgent expresses architectures as C++ models in ChampSim [19] and builds on AlphaEvolve [47], including a highly distributed evaluation setup to discover novel architectures. An outline of ArchAgent is shown in Figure 1.

1.0.2 Automatically designing state-of-the-art cache replacement policies. We show that ArchAgent can discover state-of-the-art (SoTA) cache replacement policies, broadly adhering to the experimental design of established community cache replacement policy design championships, such as the Cache Replacement Championship [20], most recently held in early 2017. The discovered policy improves IPC speedup normalized to LRU over the prior SoTA policy (published in followup work [56] in 2022) by 0.7% on single-core SPEC 2006 [21] in the absence of prefetching, and by 0.9% in the presence of prefetching. These gains are similar to prior "winning margins" reported by championship winners/new SoTA work in this domain. Notably, comparing against the effort involved in developing the prior SoTA policies, ArchAgent achieved these gains 3-5× faster than humans.

1.0.3 Policy deep-dive. To demonstrate the kinds of mechanisms/logic that ArchAgent can design, we perform an ablation study on `Policy31`, one of the winning replacement policies generated by ArchAgent. We break down the novel policy mechanisms created by ArchAgent qualitatively and quantitatively.

1.0.4 Designing beyond SPEC. In addition to discovering new policies for SPEC workloads, which is known to not be representative of cloud workloads [15, 16, 59], we use the publicly-available Google Workload Traces [1] to discover high-performance policies for hyperscale workloads, and we show that ArchAgent can design a policy that produces a 5.3% IPC speedup improvement over prior SoTA policies within two days. Previously designing a handcrafted hyperscale-centric policy would require a concerted, human-intensive effort spanning months.

1.0.5 Post-silicon "hyperspecialization". Design flows like ArchAgent represent a new paradigm in automated discovery. Although we have only explored pre-silicon discovery thus far, we can now ask the question: Could ArchAgent further improve its generated policy post-silicon, by configuring the policy specifically for each workload at runtime? While this process would traditionally be extremely human-intensive, we show that ArchAgent can further improve IPC speedup by 2.4% on average (up to 8.1% on `mcf_46B`) over the prior state-of-the-art by customizing post-silicon runtime-configurable policy parameters on memory-intensive SPEC 2006 workloads. *It is important to note that this is the only part of this work where we only tune parameters; all other uses of ArchAgent are for devising new policy logic and mechanisms.*

1.0.6 Strengths, pitfalls, and a call-to-action. ArchAgent provides a proof-of-concept that ML-assisted design flows can enable architectural discovery. However, this leaves many open questions still to be answered. We take an experiential *and* evidence-based deep-dive on the strengths and weaknesses of such systems today, including critical issues around about overfitting versus specialization, extrapolation from representative workloads, and the implications for our community evaluation infrastructure in the era of agentic AI.

2 Background

To set the context, we briefly cover very recent advances in two core areas: (1) agentic LLM-based evolutionary discovery tools and (2) cache replacement policy design and competition-based evaluation. A more substantial discussion of related work can be found in Section 8.

2.1 LLM-based Evolutionary Agents

Recently, there has been an emergence of coding agents designed for algorithmic and scientific discovery, such as Google DeepMind's AlphaEvolve. These agents use LLMs in combination with evolutionary search to discover new solutions across varied domains such as their use in mathematical proofs [18] and software systems research [11]. Such agentic systems allow for the evolution of an entire code file (e.g., hundreds of lines of code) in any programming language through automatic generation and evaluation of code. This is done by a human programmer first providing setup instructions (e.g., task directives, evaluation criteria, and background knowledge) to the system and an initial solution to seed an evolutionary database. Next, the setup instructions and prior code blocks from the evolutionary database are sampled to create prompts, which are fed to LLMs to generate new code blocks. These new code blocks are fed into an evaluator, then ranked and fed into the evolutionary database before continuing on iteratively. Such ranking and evolution is done by way of evolutionary algorithms such as island-based evolution [52, 62] or MAP-Elites evolution [44], which balance genetic diversity based on behaviors (e.g., code size and complexity) as well as performance on a specified metric (i.e., fitness score).

2.2 Cache Replacement Championships

With the goal of incentivizing the community to develop novel cache replacement policies, The 1st Journal of Instruction-Level Parallelism (JILP) Cache Replacement Championship (CRC-1) [3] was held in 2010 to compare different last level cache (LLC) cache replacement algorithms in a common evaluation framework. Given a fixed storage budget and predefined single- and multi-core configurations, competitors could submit cache replacement policies that would then be ranked by a common set of workloads. The latest championship, CRC-2 [20], had four tracks: (1) single-core

without prefetching, (2) single core with prefetching, (3) multi-core without prefetching, and (4) multi-core with prefetching. Each track was ranked separately, and the winner was decided based on the aggregate score across all tracks. The single-core track score was determined by the geometric-mean of replacement algorithm speedups across a set of single-threaded workloads, while in the multi-core track, scores were determined by the weighted speedup across a set of multi-program and multi-threaded workloads.

Looking across CRC-1, CRC-2, and other non-championship publications in the cache replacement policy domain [17, 23, 45, 56, 67, 71], margins of instructions-per-cycle (IPC) improvement normalized to LRU compared to prior winners are typically reported to be in the range of 1% to 3%, for single-core ChampSim configurations across various mixes of workloads (including SPEC, GAP [6], CVP1 [49], and more). For direct comparison, Mockingjay [56], the prior state-of-the-art cache replacement policy, saw a 1.6% and 1.2% increase in single-core performance normalized to LRU over Hawkeye [23] across a memory-intensive subset of SPEC 2006 workloads, without prefetching and with prefetching, respectively. We make two observations across the winning policies:

- The winning margins are typically small and are getting smaller, which attests to the difficulty of the problem domain and the expectation that the headroom is shrinking.
- Nevertheless, the field of cache replacement has progressed gradually with wins in the 1%-3% range accumulating over time. However, each advance requires a concerted effort, with (usually) multiple researchers studying the problem over a long period of time.

3 ArchAgent System Design

ArchAgent harnesses an agentic generative AI system to automate computer architecture discovery. At its core, we utilize the fact that a common mechanism for expressing new computer architectures is writing code in a software (micro)architectural simulator, which is a good match for an evolutionary large language model-based code-authoring agent such as AlphaEvolve. ArchAgent integrates ChampSim, a widely used trace-based microarchitectural simulator written in C++, with AlphaEvolve and a distributed evaluation backend. AlphaEvolve is capable of rewriting any C++ code file in ChampSim, but for the results presented in this paper, we restrict it to designing last-level cache replacement policies. ArchAgent works iteratively: automatically implementing candidate policies in ChampSim and running large-scale, distributed ChampSim simulations to evaluate quality (primarily on instructions per cycle (IPC) achieved). The candidate policies and evaluation feedback guide an evolutionary search algorithm to propose changes for a new generation of candidates. Figure 1 shows a high-level overview of ArchAgent. In the rest of this section, we show the key components of ArchAgent in the context of using it to design new cache replacement policies.

3.1 Prompting an LLM to be an Architecture Research Agent

To automatically generate new policies, we provide AlphaEvolve with an instruction prompt to describe the task and provide relevant background and context. Figure 2 lists a simplified example

```
Act as an expert software developer and computer
architect. The codebase that you are working on is a
simulator for an out-of-order superscalar processor
running a program trace. Your task is to iteratively
improve the indicated section of the codebase, which
models a replacement policy for the caches in the
simulated processor and win the Cache Replacement
Championship. That is, you want to design the best
possible cache replacement policy for an out-of-order
superscalar processor and implement it in the simulator.
The primary goal is to increase the scores on the
provided evaluation metrics, where larger values are
better. One of these metrics is the number of
instructions-per-cycle (IPC) that the processor executes.
 A better processor will achieve a higher IPC.

[...]

Ensure the code you introduce is realizable in hardware.
Ensure you don't use more than 48KB state for your
replacement policy. This is separate from the size of the
 cache itself. This is a strict limit and you must adhere
 to it honestly. Otherwise you will be disqualified.

[... context about simulator caveats/APIs, max lines to
change, workloads, system configs, prior working programs
, and prior literature ...]
```

Figure 2: Simplified example of a prompt given to the AlphaEvolve used in ArchAgent including persona, background information, guidance.

of a prompt used, that we aimed to be an expert computer architect persona. Combinations of prior policies (such as DRRIP, SHiP, Mockingjay), literature references, expectations of IPC headroom, and simulator interface information were provided to help an ensemble of fast and efficient (Gemini 2.5-Flash) and deep reasoning (Gemini 2.5-Pro) LLMs to generate varied policies. After each run, the prompt is automatically configured to provide solutions and metadata from previous runs to create the next evolution to test. Importantly, optimizations added included prompting the models to have more variation, respect hardware budget constraints (e.g., state size) required by the prior cache replacement championships, and reduce code complexity (e.g., upper bound of 1K lines of code). This included variations of word-play to “entice" the models to generate sufficiently different responses. The prompt described above is templatized, with various components probabilistically sampled and inserted to generate a variety of prompts at runtime to improve the volume of ideas explored.

3.2 Starter Code

We provide ChampSim’s default replacement policy C++ implementation files as starter code for AlphaEvolve to add new policy logic and change/remove ineffective logic. While we experimented with simple policies as starting points (e.g., LRU), we eventually converged on providing the Mockingjay policy source code, for faster progress and convergence due to slow simulation speeds. Similar to detailed documentation in the prompt, the file was also annotated extensively with additional API information, assumptions, and more.

Table 1: Simulated ChampSim memory system configurations obtained from the 2nd Cache Replacement Championship.

Parameter		Single-Core Config.		Multi-Core Config. (4 Cores)	
		Prefetch		Prefetch	
		Disabled	Enabled	Disabled	Enabled
Cache Sizes	L1 Data	48 KiB	48 KiB	48 KiB	48 KiB
	L1 Instruction	32 KiB	32 KiB	32 KiB	32 KiB
	L2 (Per Core)	512 KiB	512 KiB	512 KiB	512 KiB
	LLC (Shared)	2 MiB	2 MiB	8 MiB	8 MiB
Prefetchers	L1 Data	-	Next Line	-	Next Line
	L2 (Per Core)	-	PC Stride	-	PC Stride
DRAM	Bandwidth	25.6 GB/s	25.6 GB/s	25.6 GB/s	25.6 GB/s

3.3 Workloads and ChampSim Hardware Configurations

The choice of workload that can be fed to ChampSim in ArchAgent is flexible. In this case, we selected both SPEC 2006 and Google Workload Traces Version 2, representing a mix of publicly-available workloads with a variety of memory-traffic characteristics. These SPEC 2006 traces, obtained from CRC-2, use SimPoints [50] on multiple high LLC misses-per-kilo-instruction (MPKI) workloads [56]. We use the publicly-available Google Workload Traces Version 2 and convert into a ChampSim-compatible format. For ChampSim microarchitectural configurations, we also used the default CRC-2 single- and multi-core configurations as seen in Table 1. In Section 3.5, we describe the process of running ChampSim on our distributed backend to maximize parallelism on long-running evaluations.

3.4 Evaluation Metrics

To match CRC-2, ArchAgent optimizes for the IPC of the cores running the given workloads. For single-core ChampSim configurations, this amounts to the geometric-mean IPC speedup over the baseline ChampSim LRU policy. For multi-core configurations, this is the weighted IPC speedup over the same LRU policy.

While the initial LLM prompt was given a hardware budget and tips to improve explainability of generated responses, the underlying LLMs occasionally ignored instructions given. This resulted in combinations of unrealistic hardware implementations, circumventing the championship constraints, and complex and difficult to understand policies.

In addition to stronger prompt verbiage (e.g. explicit lines of code wanted), we added the ability to measure the number of lines of code generated and negatively rewarded the system for longer responses. Once a final candidate policy was identified (i.e., after sufficient iterative progress on target metrics), additional manual checking was also done to verify that output policies represented realistic, hardware-implementable designs.

3.5 Challenges in Using a Microarchitectural Simulator as an AI Evaluator

Evaluation of each policy change takes a significant amount of time because microarchitectural simulations are slow and ChampSim itself is single threaded. For example, the evaluation for single-core SPEC 2006 using ChampSim in the CRC-2 competition framing requires running workload traces for 1B instructions. Depending on the host machine running a simulation, the ChampSim configuration, and the benchmark being simulated, this can result in a single simulation taking over 12 hours on a SoTA server-class CPU. Multi-core evaluations are even slower, often taking 2-4 days for some workload mixes since simulation is single threaded, resulting in at least a proportional increase in runtime relative to the number of cores simulated.

To reduce ArchAgent's overall iteration time and avoid overfitting, we reduced the number of instructions run when evaluating proposed policies in each iteration of the evolutionary discovery process. While this resulted in some generalization issues—where results on shorter runs did not generalize to longer, representative runs—we took a cascaded approach wherein policies were created on shorter runs (i.e., 100M instructions) then later validated on longer runs (e.g., 1B instructions for SPEC results to match competition framing).

To maximize iteration speed, we also added the capability to run parallel long-running simulations on a distributed cluster. As we began to run microarchitectural simulations in this fashion, we experienced issues common to building distributed systems, including seeing simulations get canceled due to maintenance operations (e.g., kernel updates) and other hardware instability. Typical scale-out software workloads would avoid this issue by periodically checkpointing intermediate state to persistent storage to restart from. However, the ChampSim simulator does not support this functionality, resulting in failed multi-day simulations (used for validating proposed policies) having to be restarted from scratch. Thus, additional infrastructure was built to automatically restart jobs and a locally managed persistent cluster was set up to handle long running jobs susceptible to interruption. In Section 7.3 we discuss augmenting computer architecture evaluation infrastructure to avoid these issues.

3.6 Putting Together the Pieces

After the above inputs are provided and the environment configured, ArchAgent controls the discovery process, automatically generating novel policies, testing them against the evaluation environment, and continually improving them using evolutionary search guided by quantitative evaluator feedback. Due to prompt randomization, LLM entropy, and evolutionary algorithm sampling, evolution speed varies as the system explores the design space and reaches a state-of-the-art solution.

4 Using ArchAgent to Automatically Design LLC Replacement Policies for Single-Core Systems

In this case study, we use ArchAgent to design a last-level cache replacement policy, `Policy31`, to compete in the single-core portion of the CRC-2.

4.1 Methodology

`Policy31` is constructed by ArchAgent optimizing for SimPoint [50] traces from the SPEC 2006 workload suite on single-core ChampSim system configurations with both prefetching and no prefetching, as shown in Table 1. To align with recent cache replacement policy

Table 2: Comparison of Policy Evolution Setups. In `Evaluation Metric`, `geomean` is geometric-mean across benchmarks while `mean(IPC)` refers to the arithmetic mean of IPC across all cores.

Policy	Workload	Simulation Instructions		Evaluation Metric	Evolution Time
		Evolution	Validation		
`Policy31`	19 Memory Intensive SPEC06 Benchmarks	max(50M, 100M) within 1 hr	1B	$geomean\left(\frac{\text{IPC}_{\text{policy}}}{\text{IPC}_{\text{lru}}}\right)$	18 days
`Policy31-Tuned`	19 Memory Intensive SPEC06 Benchmarks	1B	1B	$geomean\left(\frac{\text{IPC}_{\text{policy}}}{\text{IPC}_{\text{lru}}}\right)$	< 8 days
`Policy61`	11 Google Workload Traces	50M	75M	$geomean\left(\frac{\text{mean(IPC}_{\text{policy}})}{\text{mean(IPC}_{\text{lru}})}\right)$	4 days
`Policy62`	11 Google Workload Traces	20M	75M	$geomean\left(\frac{\text{mean(IPC}_{\text{policy}})}{\text{mean(IPC}_{\text{lru}})}\right)$	2 days

evaluations [56], we only use memory-intensive SPEC workloads that have LLC MPKI > 1 with the LRU replacement policy. Workload-level speedup is measured as the IPC ratio of the generated policy over an LRU baseline ($\frac{IPC_{policy}}{IPC_{lru}}$) while suite-level speedup is measured as the geometric-mean of workload-level speedups as seen in Table 2.

For speedy evolution and to prevent overfitting, we provide feedback to ArchAgent by simulating a relatively small number of instructions (i.e., upto 100M instructions for each individual workload trace) to allow the simulations to complete within one hour. However, for validation, we collect final results by executing 1B instructions for each workload trace to comply with the CRC-2 competition framing. Quicker feedback allows ArchAgent to explore more solutions in the given time frame (further discussed in Section 7.3).

4.2 `Policy31` Description

Starting from the Mockingjay codebase, ArchAgent experimented for 18 days by adding (or changing/removing) code for new policy logic/mechanisms to create `Policy31`.

Mockingjay predicts the future reuse distance of each line using a PC-based predictor and then evicts the line whose predicted reuse is furthest in the future. ArchAgent augmented the policy with several new techniques discussed below.

4.2.1 Insertion Quality. `Policy31` includes an Insertion Quality Predictor (IQP) to identify PCs that bring in dead blocks (i.e., blocks that are never used), and it penalizes reuse distance prediction for dead blocks by inflating their predicted reuse distance (which makes them more likely to get evicted).

4.2.2 Hawks and Doves. `Policy31` tracks the usage intensity of each block with a 2-bit saturating counter. Blocks with a higher usage count are considered more valuable and are less likely to be evicted. Figure 3 shows the logic to adjust ETR (Estimated Time of Reuse) based on usage.

4.2.3 Prefetch-Aware Retention. `Policy31` introduces logic to de-prioritize easy-to-prefetch sources by giving them a high ETR and prioritize difficult-to-prefetch sources by giving them a low ETR.

4.2.4 Cache Pressure-Aware Adaptive Throttling (CPAAT). `Policy31` introduces a new mechanism to dynamically adjust the bypass aggressiveness based on the overall miss rate (i.e., cache pressure). Under high pressure, it's more conservative with new insertions, preserving the existing working set.

```
+ // --- HAWKS AND DOVES STATE ---
+ // A bit-packed 2b sat. ctr per cache line
+ // Tracks usage intensity and follows state budget
+ // Each byte holds 4 counters
+ std::vector<uint8_t> packed_usage_counter;

/* find a cache block to evict */
long policy31::find_victim(...) {
  for (...) {
+    // Retrieve H&D usage and use for ETR
+    current.usage = get_usage(set, way);
+    current.effective_etr = abs(current.etr_val) - (
current.usage * BONUS_PER_USE);
  }
}

/* called on every cache hit and cache refill */
void policy31::update_replacement_state(...) {
+  if (hit) {
+    // Inc. usage counter (saturating at 3)
+    // This makes frequently-used blocks stickier
+    increment_usage(set, way);
+  } else {
+    // Fill on a miss (a new line inserted)
+    // The new block starts with a usage of 0
+    reset_usage(set, way);
+  }
}
```

Figure 3: Example `Policy31` modifications in the form of a diff to implement the Hawks and Doves mechanism. The `packed_usage_counter` and corresponding `get-`, `increment-`, and `reset_usage` setter/getters are used to help determine eviction candidates.

4.3 Results

We compare suite-level IPC speedup improvement normalized to the standard LRU baseline. We find that `Policy31` improves IPC speedup normalized to LRU by 12.2% and 8.0% in the no prefetch and prefetch configurations, respectively (versus 11.4% and 7.0% for Mockingjay). Figure 6 shows the workload-level improvement across the suite in the prefetch-enabled case, demonstrating overall

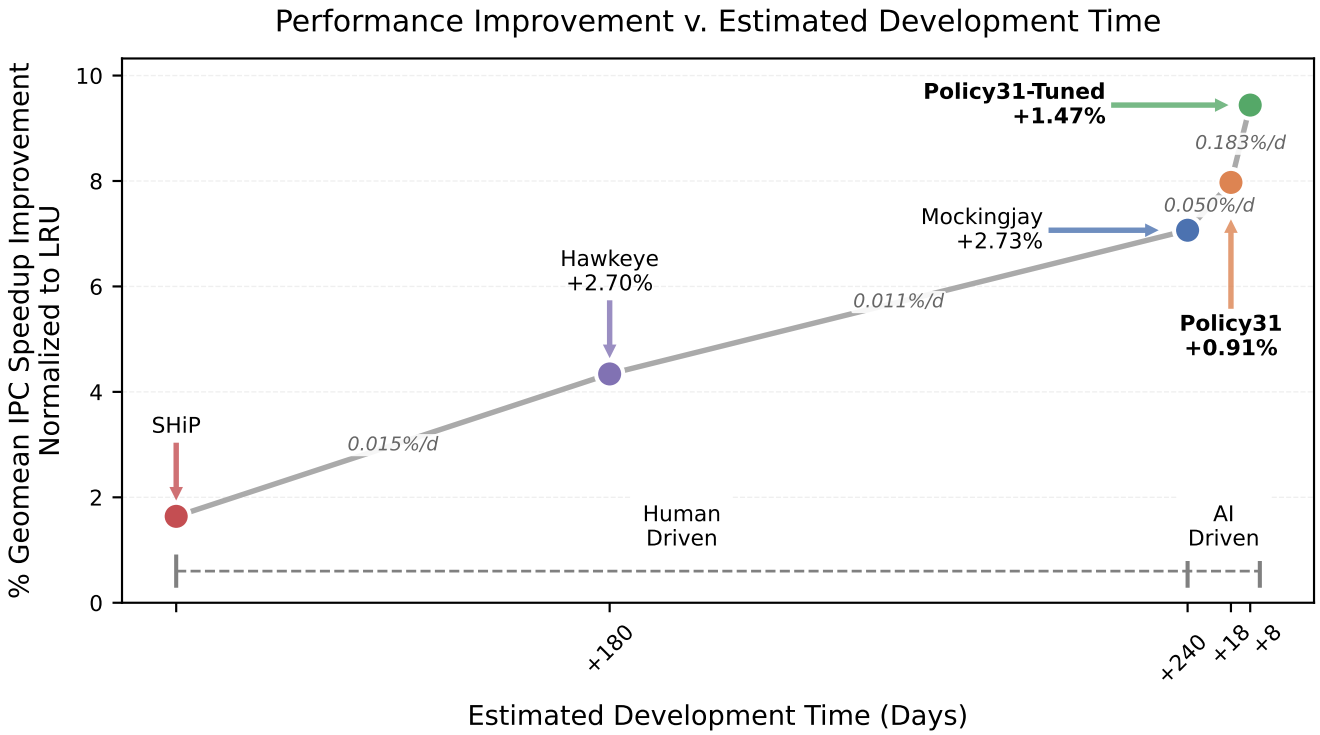

Figure 4: Performance improvement (suite-level geomean IPC speedup normalized to LRU) compared to estimated development time of replacement policies for the single-core prefetch-enabled ChampSim configuration on memory-intensive SPEC06 workloads. Slope (grey, italics) denotes percentage point improvement per day.

improvement and no significant outliers.

To put these improvements in context, Section 2.2 describes that the historical rate of improvement for cache replacement for single-core ChampSim configurations is reported to be in the 1-3% range per solution. Each improvement in prior work has needed significant effort by multiple researchers over several months, whereas here we have demonstrated that ArchAgent can achieve similar gains in less than 3 weeks.

Figure 4 compares the performance improvement achieved by successive SoTA cache replacement policies compared to the estimated time of developing these policies for the single-core prefetch-enabled ChampSim configuration on memory-intensive SPEC06 workloads. The estimated development time was obtained from their creators. These reported times include only time to develop the policies, excluding, for example, paper writing time. When comparing the effort involved in developing `Policy31` and the prior SoTA policies, we find that ArchAgent achieved its gains 3-5× faster than humans. Furthermore, replacement policy development for SPEC workloads is well explored as prior work has demonstrated that existing solutions come within 90% of the hit rate optimal solution [56]. This points to the increasing difficulty of finding performance improvements for this problem domain.

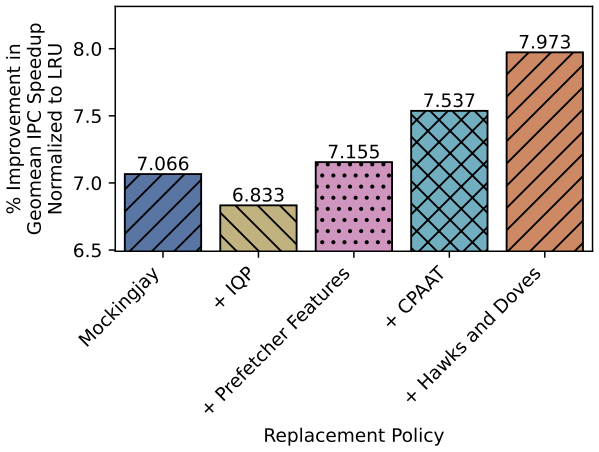

Figure 5: Ablation study measuring improvement in suite-level geomean IPC speedup with each new technique that composes `Policy31` for the single-core prefetch-enabled ChampSim configuration running SPEC06 memory intensive workloads.

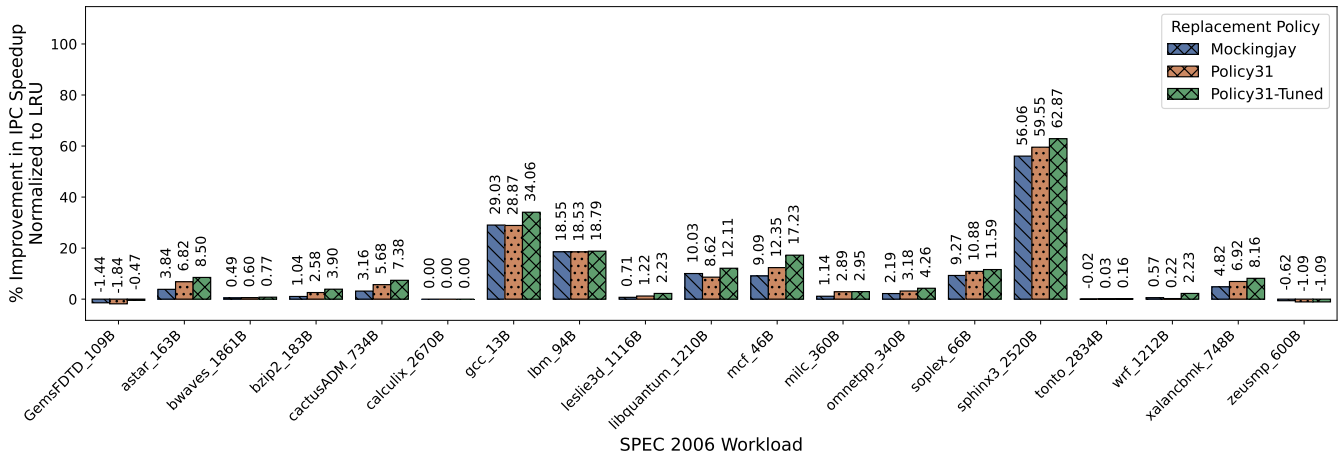

Figure 6: Per-workload improvement in policy IPC speedup normalized to LRU for the single-core prefetch-enabled ChampSim configuration running SPEC 2006 memory intensive workloads.

4.4 Ablation Study

To better understand the source of performance improvement for `Policy31`, we run an ablation study breaking down the impact of key policy features created by ArchAgent (described in 4.2). Figure 5 shows the ablation study results when running on the default single-core prefetch-enabled ChampSim configuration running workloads from SPEC 2006 suite for 1B instructions. Selectively, the study re-enables the various mechanisms that comprise `Policy31` one-by-one, until the complete `Policy31` is evaluated. As seen in the figure, most of the improvement is coming from CPAAT and Hawks and Doves, providing a 0.8% improvement, combined. IQP and prefetch management result in marginal gains and only show benefits when combined with all other mechanisms.

This highlights a challenge with machine-generated policies, where manual work is still needed to not only validate the performance improvement but also understand the root cause. To have high confidence in machine-generated policies, the process of ablation and verification will remain a key bottleneck that could also benefit from automation, as discussed further in 7.1.

5 Using ArchAgent for Runtime-Configurable, Workload-Specific Hyperoptimization

We observe that ArchAgent-generated policies such as `Policy31` introduce new microarchitectural techniques that make the policies flexible and amenable to runtime tuning. These runtime parameters are expressions in the policy that do not affect storage size and are instead either constants or simple arithmetic expressions derived from constants. We identify 13 runtime parameters in `Policy31` that fall into the following two categories:

- Constants used in scores such as ETR or bonuses/penalties for particular categories of accesses (e.g., prefetches).
- Thresholds such as those used to detect long ETR blocks or initiate adaptive decay of cache counters.

5.1 Methodology

Since we are exploring runtime tuning, both evolution and validation are performed on the memory intensive SPEC 2006 workload subset mentioned in Section 4.1 running for 1B instructions (Table 2). ArchAgent optimizes each workload individually and we enforce that it only changes runtime parameters instead of the entire replacement policy.

5.2 Results

ArchAgent creates a new runtime parameter-tuned variant of `Policy31`, called `Policy31-Tuned`, tuned for individual SPEC 2006 workloads on the championship single-core ChampSim prefetch-enabled configuration. When comparing suite-level speedups, `Policy31-Tuned` achieves an additional 1.5% overall geometric-mean IPC improvement as compared to `Policy31` and 2.3% overall improvement compared to Mockingjay, normalized to LRU. When analyzing workload-level speedups, Figure 6 shows that workloads such as `gcc_13B` and `mcf_46B` show >5.0% gain while others show limited improvement (e.g., `calculix_2670B`).

To put these results in context, Figure 4 shows that ArchAgent's rate of improvement with `Policy31-Tuned` is over 10× faster than prior SoTA policies. Notably, this improvement was achieved in less than eight days.

These results highlight the potential of AI-driven runtime specialization of hardware to specific workloads and we envision that such post-silicon hyperoptimizations could be commonplace, similar to automatic profile-guided optimizations prevalent in hyperscalers [4, 10, 36].

6 Using ArchAgent to Automatically Design Multi-Core LLC Replacement Policies for Google Workload Traces

With `Policy31`, we show ArchAgent can win within an established competition environment on SPEC 2006. However, it is widely established that SPEC 2006 is not representative of hyperscale cloud

workloads [15]. For example, SPEC workloads have much smaller instruction footprints and significantly lower instruction cache pressure as compared to hyperscale workloads. While Mockingjay shows clear wins on both single- and multi-core SPEC, Figure 8 shows it performs much worse than even LRU on Google Workload Traces.

Thus, in this case study, we use ArchAgent to generate multi-core cache replacement policies, `Policy61` and `Policy62`, specialized for publicly-available Google Workload Traces Version 2.

6.1 Methodology

`Policy61` and `Policy62` are generated by running 11 workloads from the Google Workload Traces Version 2 suite across both prefetch and non-prefetch multi-core ChampSim configurations stated in Table 1. The DynamoRIO trace scheduler [8], similar to an operating system scheduler, is used to schedule thread-level traces from each workload onto the four cores of the multi-core system.

Final validation is done by simulating the two policies for 75M instructions as seen in Table 2. Here, workload mix speedup is computed as the arithmetic mean of IPC across cores normalized to the arithmetic mean of IPC across cores using the LRU replacement policy. We find this to be a suitable metric because all four cores are executing threads from the same workload. Suite-level speedup is measured as the geometric-mean of workload-level speedups.

For speed, we provide evolution feedback by simulating 50M and 20M instructions for `Policy61` and `Policy62`, respectively. The feedback metric averages suite-level speedup scores for both multi-core configurations.

We provide the Mockingjay source code as the starter code for both policies, and we allow ArchAgent to change the entire replacement policy code.

6.2 Policy Description

Two different ArchAgent runs on these traces resulted in two vastly different policies, but both are building on the premise that code characteristics of hyperscale workloads are different, so the use of PC as a feature needs to be revisited.

`Policy61`. `Policy61` was generated over the course of four days. It maintains the core algorithm of Mockingjay, but makes one critical change: it enriches the signature used for prediction with information about the path taken to reach the current instruction as seen below.

```
// XOR prev. hist. with PC to make a uniq. signature
core_pc_history[cpu] = ((history << 1) | (history >> 63))
 ^ instr_pc;
```

This is expected to help in scenarios where the calling context of a PC is important for its caching behavior. For example, a generic `memcpy` routine might be called to copy small, frequently-reused data structures in one part of a program, and to perform large, non-temporal streaming copies in another. A predictor that only looks at the PC of the `memcpy`'s load/store instructions will conflate these distinct behaviors, leading to an average prediction that is optimal for neither case.

Similar ideas have been proposed in the literature before [41, 57], but not in the context of Mockingjay.

`Policy62`. `Policy62` was generated over the course of two days and uses a completely different approach from Mockingjay. It first removes all the key components of Mockingjay (including reuse prediction and eviction based on estimated time of reuse) and then evolves into something that is very close to SHiP, but is much more adaptive and sensitive to larger code footprints.

In particular, two key ideas make `Policy62` different:

- *Tagged Predictor Table*: PC-based cache predictors usually allow for aliasing, and the tables are simply indexed by a hash without any tag matching. `Policy62` explicitly stores a 3-bit tag inside the predictor entry. Predictions are retrieved only if the tags match. If the tags don't match, it resets the entry. This prevents "destructive aliasing," making the predictor more precise.
- *Learning Signal*: SHiP typically trains its predictor at the time of eviction. When a block is kicked out of the cache, SHiP checks whether the block was used. If yes, the counter for the PC that inserted it is incremented. If not, the counter is decremented. By contrast, `Policy62` updates the predictor when the line is accessed. If the line hits, the counter for the PC corresponding to the access is incremented. If the line misses, the counter for the PC corresponding to the access is decremented.

While subtle, the second difference is significant because it allows `Policy62` to learn much quicker than SHiP or Mockingjay. Waiting for cache eviction to learn can prolong learning feedback. The key idea here is that some PCs have a small working set that can be cached, while most others will not see reuse due to the large working set. Thus, PCs that miss frequently (and fetch a lot of data) are penalized and PCs that hit frequently are prioritized for cache residency.

6.3 Results

Figures 7 and 8 compare suite-level speedups on multi-core configurations with prefetching disabled and enabled, respectively. We find a major inversion in trends here with Mockingjay performing much worse than even LRU and SHiP emerging as the prior SoTA policy.

- On the non-prefetch configuration, we find that SHiP, `Policy61`, and `Policy62` achieve a 3.0%, 4.7%, and 6.1% improvement over LRU, respectively.
- On the prefetch-enabled configuration, we find that SHiP, `Policy61`, and `Policy62` achieve a 2.9%, 5.4%, and 8.2% improvement over LRU, respectively.
- On the prefetching-enabled configuration, Mockingjay, the starter code for ArchAgent policies, shows a slowdown of 9.5% over LRU. Thus, ArchAgent recovers a performance deficit of 17.8% with a simpler policy.

Figure 9 shows workload-level speedups across the Google Workload Traces suite in the prefetch-enabled case with `Policy61` and `Policy62` showing consistently strong performance compared to prior work.

In our understanding, the trend inversion between Mockingjay, SHiP, and LRU, and the strong performance of `Policy61` and `Policy62`—simpler policies derived from Mockingjay—can be attributed to the unique characteristics of these hyperscale workloads,

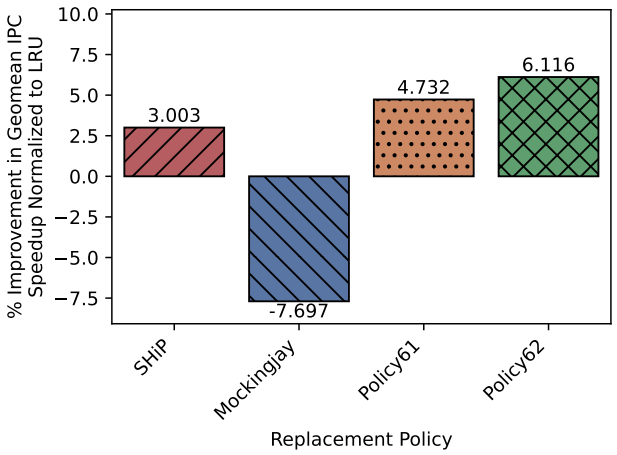

Figure 7: Improvement in geomean IPC speedup normalized to LRU for the multi-core prefetch-disabled ChampSim configuration running Google Workload Traces.

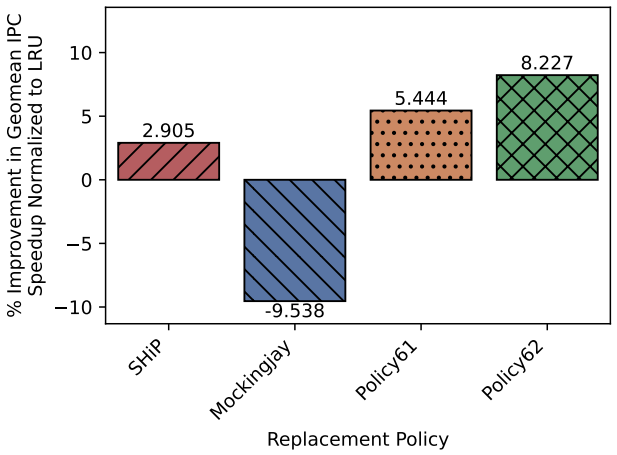

Figure 8: Improvement in geomean IPC speedup normalized to LRU for the multi-core prefetch-enabled ChampSim configuration running Google Workload Traces.

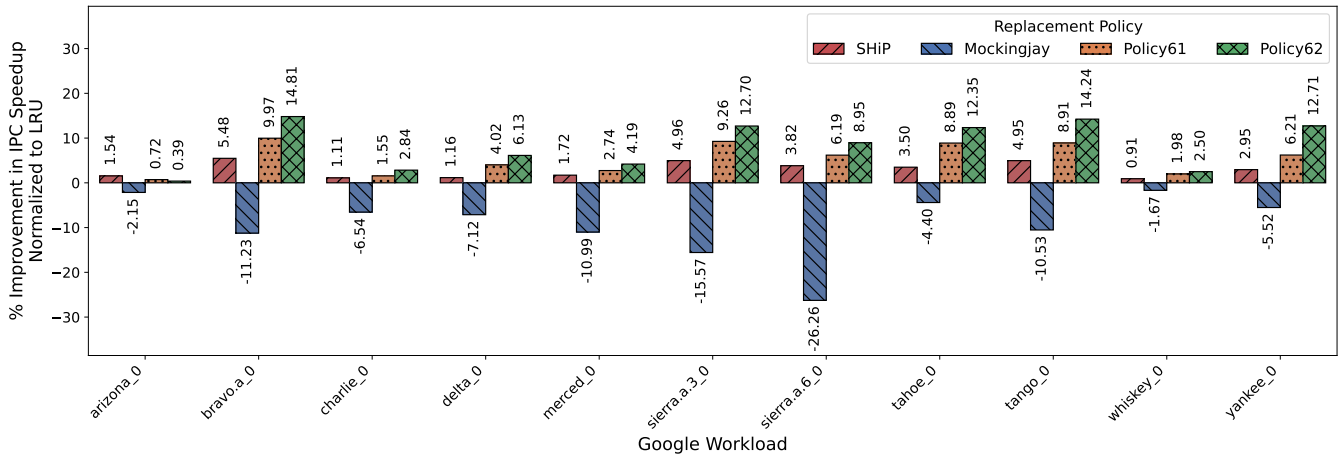

Figure 9: Per-workload improvement in policy IPC speedup normalized to LRU for the multi-core prefetch-enabled ChampSim configuration running Google Workload Traces.

such as deep call stacks, high degree of multithreading, and high context switch rates.

We show that ArchAgent enables designers to rapidly customize replacement policies to previously unexplored workload classes.

7 Discussion, Future Work, and a Call-to-Action

The use of evolutionary coding agents in computer architecture research shows great promise, but will require significant improvements to our community research infrastructure, as well as continued experimentation with AI tools. In this section, we outline some of the key takeaways and potential next steps in this area.

7.1 LLM Capabilities and Constraints

A central finding of this study is the remarkable capability of LLMs to generate new microarchitectural designs, including both microarchitectural technique discovery and parameter optimization. Given a rich context, our evolutionary agent automatically generated novel, complex, and functionally correct hardware policies that improved on the performance achieved by state-of-the-art human-designed policies for both classical workloads (SPEC) and hyperscale workloads (Google Workload Traces).

Given the current approach however, confirming the realizability of generated policies remains a human intensive process, due to the lack of ASIC quality-of-result data (e.g., frequency and area) in most microarchitectural simulators. In our case, while we found specialized prompts to be effective at generally guiding the LLMs

along these lines, significant human-driven verification was still needed to guarantee compliance. For example, to avoid complex, difficult-to-understand policies with thousands of lines of changes, we combined the use of additional prompting to limit code size (a proxy for code complexity and hardware realizability) as well as a rule-based verifier that calculated the number of total/added lines of code and discarded samples violating constraints. This significantly improved the explainability of solutions found, reducing the time-consuming manual ablation studies needed to understand any improvement and estimate its true hardware overhead. However, writing rule-based verifiers is not feasible for all constraints. As an example, inferring physical storage size from a microarchitectural model is non-trivial and thus we still rely on prompting and manual pruning/assessment for such a constraint. This highlights a critical and interesting avenue for future work: developing principled, automated methods to inject these hard architectural constraints directly into the design process, moving beyond simple prompting or post-generation manual verification.

Additionally, despite the high quality of generated solutions evidenced in this work, there remains significant opportunity in pushing the limits of the type of solutions ArchAgent can generate. For one, there is an opportunity to collect a dataset representing a much greater sample of prior art in the cache replacement space, both in terms of implementations (software code, RTL) and descriptions (documentation, academic papers, industrial reports). Adding additional agents in the loop is another opportunity. Each of a group of multiple agents can employ a different persona, e.g., a microarchitect, an SRAM design/process design expert, a physical design expert, and a workload expert, each of which are responsible for providing different (and perhaps competing) feedback on the design, potentially using industry standard EDA tools.

7.2 Hardware/Software Co-Design For Hyperspecialization

The automated nature of evolutionary coding agents will have significant implications for computer architecture and systems research. Historically, the immense human effort and simulation time required to design and validate even a single new heuristic has biased the field toward generalist solutions; policies that perform adequately, but rarely optimally, across all workloads. The sheer productivity of an automated parallelizable agent, which can generate and evaluate thousands of policy variants in a short time, shatters this barrier and makes specialization computationally feasible, especially if deployed systems support experimentation.

This capability can usher in a new era of hardware-software co-design, moving beyond the "one-size-fits-all" paradigm. Instead of a static, general-purpose cache policy, a system built with even greater runtime configurability (vs. the simpler runtime parameter tuning we explored in Section 5) could deploy purpose-built heuristics per workload. For example, a heuristic evolved specifically for a critical database, a high-priority AI model, or a latency-sensitive video pipeline could yield significant efficiency benefits. This requires the right low-cost flexible hardware-software interfaces that allow an operating system or hypervisor to securely and efficiently specialize hardware components, a promising and critical direction for future systems research.

7.3 A Call-to-Action: Evaluation Methodology Improvements

Evaluation methodology, namely computer architecture simulator performance and quality, needs to be re-evaluated in the new era of (potentially adversarial) AI agent-driven execution.

In our early experiments, we found that ArchAgent was "willing" to take any shortcut made available by the simulation environment, unlike a human researcher acting in good faith. We describe one such "simulator escape"—a situation where the agentic AI flow discovered and exploited a loophole in the simulator used for evaluations.

The ChampSim simulator does not support write-bypassing in the LLC. Verification that a policy is not performing write-bypassing only takes place *in an assertion that is eliminated by the compiler when ChampSim is compiled with optimizations enabled*. Given the simulator performance concerns described in Section 3.5 and below, building with these optimizations enabled is crucial. In this case, ArchAgent developed a policy called `Policy12`, that appeared to be beating Mockingjay on the SPEC06 workloads in the single-core no-prefetch and prefetch scenarios by 3% and 4%, respectively. However, ArchAgent won by taking advantage of the fact that a bypassed write in the LLC would disappear from the system entirely. This not only avoided evicting a potentially more useful line, *but also eliminated the corresponding DRAM write pressure entirely*, drastically improving IPC. Since there is no notion of "correct computation" in such a trace-driven simulator, there was no further way to detect this issue.

Building simulators that are either better verified or closer to the actual hardware design (e.g., RTL simulation or hardware-accelerated emulation) and thus less susceptible to "abuse" by the AI agents is crucial. Incorporating simulators that are closer to the actual hardware design (e.g., RTL-derived) would also have the added advantage of providing additional feedback to ArchAgent-like tools, including frequency, area, and power data collected from ASIC EDA tools.

As discussed briefly in Section 3.5, simulation speed also limits ArchAgent's rate of discovery. ArchAgent's ability to reach creative solutions is inversely proportional to the evaluation latency. This becomes untenable for long, complex simulations which are commonplace in computer architecture. For example, evaluating a single design point for a multi-core system using ChampSim, executing hundreds of millions of instructions, can require a 12- to 24-hour turnaround. An evolutionary search, which needs to evaluate thousands of candidates for enough variation, can be severely limited by the simulation speed, reducing the chances for the tool to try sufficiently interesting and "risky" ideas. Similarly, this simulation speed also hinders evaluation of longer representative workloads (e.g., instead of simulating 1B instructions, simulating hundreds of billions of instructions representative of production programs).

As established in this discussion, to truly unleash the power of AI-driven architecture discovery, computer architecture simulators must become both more accurate and orders of magnitude faster, warranting more research into higher fidelity frameworks and simulators. Hardware-accelerated simulators such as FireSim [28] could provide solutions to this problem.

8 Related Work

There has been decades of research in designing efficient cache replacement policies and applying broader automated techniques for hardware-software discovery.

8.1 Designing Cache Replacement Policies

Prior work on cache replacement policy design can be viewed from the lens of different design methodologies, which can be broadly categorized into three main paradigms: handcrafted heuristics, ML-based policies, and evolutionary parameter tuning.

8.1.1 Heuristic-based Replacement. Over three decades, handcrafted heuristics have changed from simple, static rules to highly sophisticated, adaptive algorithms. Early work proposed static heuristics [17, 26, 29, 35, 48, 51, 54, 58, 66] to avoid pathologies of LRU and LFU. More recent heuristics take a predictive approach [2, 12, 14, 22, 23, 27, 30–34, 39, 61, 63, 67]. They leverage past behavior to predict future caching priorities. The Mockingjay policy [56] use predicted reuse distances to mimic Belady's optimal caching solution [7]. More recently, Mostofi et al. use offline profiling to determine Insertion and Promotion Vectors for an unseen trace, however, this solution does not outperform Mockingjay [42]. We use Mockingjay as the initial code that the AI agent starts from for better performance.

Another line of work recognizes that cache replacement policies must operate in the context of the system they are in. For example, prefetch-aware policies [24, 68, 72] recognize the distinction between demand and prefetch accesses to make replacement decisions; other policies make a similar distinction for instruction accesses [42] or TLB accesses. There is also work that recognizes the need for adapting replacement policies to the nature of the cache hierarchy, such as inclusive caches [25] or sliced caches [60].

From a methodological perspective, all these solutions are based on human intuition and insight. These works do not leverage any automated methods to search the design space of replacement policies.

8.1.2 ML-based Replacement. One branch of research deploys ML models directly in hardware, ranging from lightweight online learning [75] to full deep learning inference [37, 64]. For example, PARROT [38] casts cache replacement as an imitation learning problem to approximate the "oracle" decisions of Belady's optimal policy. These policies result in high implementation complexity and overhead, as they require online neural network inference.

Therefore, a second line of research uses powerful, unconstrained ML models in an offline setting to discover features and insights, which are used to design hardware-friendly online policies [55, 57]. For example, the Glider policy [57] leveraged a powerful, attention-based Long Short-Term Memory (LSTM) network to discover novel features for a simple online model. In this case, the ML model serves as a "discovery engine," but interpreting the model and designing the final simple policy remains a manual process.

Our work is the first to bridge the divide between these two schools. We introduce an evolutionary coding framework that automates the discovery process of the "Offline Insight" school, while explicitly optimizing for the simplicity and hardware-efficiency that is the key bottleneck for the "Online Learning" school.

8.1.3 Evolutionary Computation for Replacement Policy Tuning. Prior work that has used evolutionary computation to discover replacement policies has focused on using genetic algorithms to tune parameters within a predefined policy structure [9, 43, 73]. While these works established a precedent for using evolutionary search, they were limited to optimizing parameters within a fixed-model, rather than evolving the structure of the algorithm itself. More recently, an evolutionary coding framework was used to build web-based caching policies [13]. This work uses an approach that is similar to ours, but applies it to software caches, where the constraints and performance tradeoffs are different.

8.2 ML For Other Design Tasks

Beyond predictive policies like cache replacement or branch prediction, ML has been used across a wide-variety of hardware-software co-design domains.

With the growing importance of ML accelerators, multiple prior works have focused on hardware-software co-design of neural network accelerators through various search space optimization techniques [46, 53, 69, 70, 74]. These works often formulate the search problem to explore both hardware mixes and software mappings to newly generated hardware, leveraging advanced analytical modeling to quickly iterate through the design space. Our work differs from these by focusing on *policies*, with a well defined set of championship constraints that resulting policies need to be evaluated with a detailed microarchitectural simulator running a mix of standard and realistic hyperscale CPU workloads.

Other techniques focused on optimizing microarchitectures [5], instead focus on other reinforcement learning techniques to explore pre-silicon parameters of microarchitectures or in the case of software, optimizing existing codebases/systems [11, 40, 65]. Our work is different from these approaches as it tackles the whole-scale design (of a cache replacement policy) rather than parametric optimization of an existing system using LLM-based discovery tooling.

9 Conclusions

In this paper, we have introduced ArchAgent, a first step towards building an agentic artificial intelligence system for automatic novel computer architecture discovery. Using ArchAgent, we automatically designed and implemented four novel cache replacement policy algorithms, `Policy31`, `Policy31-Tuned`, `Policy61`, `Policy62`, that beat SoTA cache replacement policies tested in various cache replacement championships. On the public multi-core Google Workload Traces, ArchAgent achieved a 5.3% better IPC speedup over the prior SoTA within two days, and on the single-core SPEC06 workloads, it achieved a 0.9% better IPC speedup over the prior SoTA in 18 days. Comparing against the effort involved in developing the prior SoTA policies, ArchAgent achieved these gains 3-5× faster than humans. We identified that agentic flows such as ArchAgent enable post-silicon hyperspecializtion of hardware systems to a specific workload (mix) demonstrating a 2.4% IPC speedup improvement over prior SoTA on single-core SPEC06 workloads. While we highlight that the use of these agentic systems allows for a dramatic increase in computer architecture discovery, further improvements are needed to improve simulator fidelity and speed in the era of agentic-AI-assisted research.

10 Acknowledgments

This paper has used generative AI technologies in the following ways. First, the core work done by this paper uses AI tooling to develop novel cache replacement policies. Minor use of generative AI tooling was used to generate tables, plot graphs, and assist with clarity and flow, limited to phrases and single sentences. The authors have reviewed, verified, and edited all such generated content and take full responsibility for the content of the paper, including any errors or omissions.

References

[1] [n. d.]. Google Workload Traces Version 2. https://console.cloud.google.com/storage/browser/external-traces-v2. Accessed: 2025-11-13.

[2] Jaume Abella, Antonio González, Xavier Vera, and Michael FP O'Boyle. 2005. IATAC: a smart predictor to turn-off L2 cache lines. *ACM Transactions on Architecture and Code Optimization (TACO)* 2, 1 (2005), 55–77.

[3] Alaa R. Alameldeen, Aamer Jaleel, and Moinuddin Qureshi. 2010. Cache Replacement Championship. https://jilp.org/jwac-1/

[4] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 513–526. doi:10.1145/3373376.3378498

[5] Chen Bai, Jianwang Zhai, Yuzhe Ma, Bei Yu, and Martin DF Wong. 2024. Towards automated risc-v microarchitecture design with reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 12–20.

[6] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[7] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* (1966), 78–101.

[8] D. Bruening, T. Garnett, and S. Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* 265–275. doi:10.1109/CGO.2003.1191551

[9] Fouad Butt and Abdolreza Abhari. 2010. Genetic algorithms: an approach to optimal web cache replacement. In *Proceedings of the 2010 Spring Simulation Multiconference*. 1–4.

[10] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 12–23. doi:10.1145/2854038.2854044

[11] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. 2025. Barbarians at the Gate: How AI is Upending Systems Research. (2025). arXiv:2510.06189 [cs.AI] https://arxiv.org/abs/2510.06189

[12] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving Cache Management Policies Using Dynamic Reuse Distances. In *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 389–400.

[13] Rohit Dwivedula, Divyanshu Saxena, Aditya Akella, Swarat Chaudhuri, and Daehyeok Kim. 2025. Man-Made Heuristics Are Dead. Long Live Code Generators! *arXiv preprint arXiv:2510.08803* (2025).

[14] Priyank Faldu and Boris Grot. 2017. Leeway: Addressing Variability in Dead-Block Prediction for Last-Level Caches. In *26th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 180–193.

[15] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 37–48. doi:10.1145/2150976.2150982

[16] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. doi:10.1145/3297858.3304013

[17] Hongliang Gao and Chris Wilkerson. 2010. A dueling segmented LRU replacement algorithm with adaptive bypassing. In *JWAC 2010-1st JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship*.

[18] Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. 2025. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864* (2025).

[19] 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 preprint arXiv:2210.14324* (2022).

[20] Paul V. Gratz, Jinchun Kim, and Gino Chacon. 2017. The 2nd Cache Replacement Championship. https://crc2.ece.tamu.edu/?page_id=53. Accessed: 2025-11-13.

[21] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. doi:10.1145/1186736.1186737

[22] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. 2002. Timekeeping in the memory system: predicting and optimizing memory behavior. In *29th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 209–220.

[23] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *43rd Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 78–89.

[24] Akanksha Jain and Calvin Lin. 2018. Rethinking Belady's Algorithm to Accommodate Prefetching. In *45th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 110–123.

[25] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C Steely Jr, and Joel Emer. 2010. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 151–162.

[26] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *45th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 60–71.

[27] Daniel A. Jiménez and Elvira Teran. 2017. Multiperspective Reuse Prediction. In *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 436–448.

[28] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. 2018. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 29–42.

[29] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. 1994. Caching strategies to improve disk system performance. *Computer* 3 (1994), 38–46.

[30] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. 2001. Cache decay: exploiting generational behavior to reduce cache leakage power. In *18th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 240–251.

[31] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. 2007. Cache replacement based on reuse-distance prediction. In *25th International Conference on Computer Design (ICCD)*. 245–250.

[32] Samira Khan, Yingying Tian, and Daniel A Jiménez. 2010. Sampling dead block prediction for last-level caches. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 175–186.

[33] Mazen Kharbutli and Yan Solihin. 2005. Counter-Based Cache Replacement Algorithms. In *23rd International Conference on Computer Design (ICCD)*. 61–68.

[34] An-Chow Lai, Cem Fide, and Babak Falsafi. 2001. Dead-block prediction & dead-block correlating prefetchers. In *28th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 144–154.

[35] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 1999. On the existence of a spectrum of policies that subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) policies. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 27. 134–143.

[36] Heiner Litz, Grant Ayers, and Parthasarathy Ranganathan. 2022. CRISP: Critical Slice Prefetching. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 300–313. doi:10.1145/3503222.3507745

[37] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. 2020. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*. 6237–6247.

[38] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. 2020. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*. PMLR, 6237–6247.

[39] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. 2008. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 222–233.

[40] Henry Massalin. 1987. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News* 15, 5 (1987), 122–126.

[41] Samira Mirbagher-Ajorpaz, Elba Garza, Gilles Pokam, and Daniel A Jiménez. 2020. Chirp: Control-flow history reuse prediction. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 131–145.

[42] 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*. 1005–1019.

[43] Amr Abdelhai Mourad, Saleh Mesbah, and Tamer F Mabrouk. 2020. A novel approach to cache replacement policy model based on genetic algorithms. In *2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*. IEEE, 405–411.

[44] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909* (2015).

[45] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 62–73.

[46] Gauthaman Murali, Min Gyu Park, and Sung Kyu Lim. 2024. 3DNN-Xplorer: A Machine Learning Framework for Design Space Exploration of Heterogeneous 3-D DNN Accelerators. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2024).

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

[48] Elizabeth J O'Neil, Patrick E O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. In *ACM SIGMOD Record*. 297–306.

[49] Arthur Perais, Rami Sheikh, Eric Rotenberg, Chris Wilkerson, Vinesh Srinivasan, Alla R. Alameldeen, and Mikko Lipasti. 2018. https://www.microarch.org/cvp1/cvp1/index.htm

[50] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using simpoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review* 31, 1 (2003), 318–319.

[51] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In *34th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 381–391.

[52] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. 2024. Mathematical discoveries from program search with large language models. *Nature* 625, 7995 (01 Jan 2024), 468–475. doi:10.1038/s41586-023-06924-6

[53] Chirag Sakhuja, Zhan Shi, and Calvin Lin. 2023. Leveraging domain information for the efficient automated design of deep learning accelerators. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 287–301.

[54] Vivek Seshadri, Onur Mutlu, Michael A Kozuch, and Todd C Mowry. 2012. The Evicted-Address Filter: A unified mechanism to address both cache pollution and thrashing. In *the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 355–366.

[55] 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)*. IEEE, 291–303.

[56] 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)*. IEEE, 558–572.

[57] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying Deep Learning to the Cache Replacement Problem. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 413–425.

[58] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. 1999. EELRU: simple and effective adaptive page replacement. In *ACM SIGMETRICS Performance Evaluation Review*. 122–133.

[59] Wei Su, Abhishek Dhanotia, Carlos Torres, Jayneel Gandhi, Neha Gholkar, Shobhit Kanaujia, Maxim Naumov, Kalyan Subramanian, Valentin Andrei, Yifan Yuan, and Chunqiang Tang. 2025. DCPerf: An Open-Source, Battle-Tested Performance Benchmark Suite for Datacenter Workloads. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 1717–1730. doi:10.1145/3695053.3731411

[60] Sweta, Prerna Priyadarshini, and Biswabandan Panda. 2025. Drishti: Do Not Forget Slicing While Designing Last-Level Cache Replacement Policies for Many-Core Systems. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture®*. 519–532.

[61] Masamichi Takagi and Kei Hiraki. 2004. Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)*. 20–30.

[62] Reiko Tanese. 1989. *Distributed genetic algorithms for function optimization*. University of Michigan.

[63] Elvira Teran, Zhe Wang, and Daniel A Jiménez. 2016. Perceptron learning for reuse prediction. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12.

[64] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving cache replacement with {ML-based}{LeCaR}. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.

[65] Rich Vuduc and James Demmel. 2000. Code Generators for Automatic Tuning of Numerical Kernels: Experiences with FFTW. In *Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG '00)*. Springer-Verlag, Berlin, Heidelberg, 190–211.

[66] Wayne A Wong and Jean-Loup Baer. 2000. Modified LRU policies for improving second-level cache behavior. In *6th International Symposium on High-Performance Computer Architecture (HPCA)*. 49–60.

[67] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. 2011. SHiP: Signature-based Hit Predictor for High Performance Caching. In *44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 430–441.

[68] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. 2011. PACMan: Prefetch-aware Cache Management for High Performance Caching. In *44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 442–453.

[69] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. 2021. Hasco: Towards agile hardware and software co-design for tensor computation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1055–1068.

[70] Yifeng Xiao, Yurong Xu, Ning Yan, Masood Mortazavi, and Pierluigi Nuzzo. 2025. CORE: Constraint-Aware One-Step Reinforcement Learning for Simulation-Guided Neural Network Accelerator Design. *arXiv preprint arXiv:2506.03474* (2025).

[71] Vinson Young, Chia-Chen Chou, Aamer Jaleel, and Moin Qureshi. 2017. Ship++: Enhancing signature-based hit predictor for improved cache performance. In *The 2nd Cache Replacement Championship (CRC-2 Workshop in ISCA 2017)*.

[72] Samuel Yuan, Divyanshu Saxena, Jiayi Chen, Nihal Sharma, and Aditya Akella. 2025. A Joint Learning Approach to Hardware Caching and Prefetching. *arXiv preprint arXiv:2510.10862* (2025).

[73] Martin Zadnik and Marco Canini. 2010. Evolution of cache replacement policies to track heavy-hitter flows. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. 1–2.

[74] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. 2022. A full-stack search technique for domain optimized deep learning accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 27–42. doi:10.1145/3503222.3507767

[75] Yang Zhou, Fang Wang, Zhan Shi, and Dan Feng. 2022. An end-to-end automatic cache replacement policy using deep reinforcement learning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 32. 537–545.