

000  
001  
002  
003 

# ALGORITHM GENERATION VIA CREATIVE IDEATION

  
004  
005  
006  
007  
008  
009  
010  
011  
012  
013  
014  
015  
016  
017  
018  
019  
020  
021  
022  
023  
024  
025  
026  
027  
028  
029  
030  
031  
032  
033  
034  
035  
036  
037  
038  
039  
040  
041  
042  
043  
044  
045  
046  
047  
048  
049  
050  
051  
052  
053

**Anonymous authors**

Paper under double-blind review

## ABSTRACT

Designing system algorithms remains challenging, where the discontinuous nature of the solution space often forces system engineers to rely on generic heuristics at the expense of performance. We study whether LLMs can practically drive algorithm generation, and find that they are biased towards well-known generic designs, rather than making the creative leaps needed to navigate the discontinuous solution space. To address this limitation, we introduce MetaMuse, a framework for creative ideation built on three self-reflection principles: (1) quantifying solution diversity and usefulness in measurable performance space, rather than abstract idea space, (2) steering ideation through external stimuli, rather than internal randomness, and (3) constructing executable solutions using waypoint reasoning, rather than free-form chain-of-thought. Considering two critical online problems at a global cloud provider, extensive evaluations show that MetaMuse can generate high-performing solutions: cache replacement (reducing cache misses by up to 35.76%) and online bin packing (reducing bin usage by up to 30.93%).

## 1 INTRODUCTION

Designing system algorithms continues to be a central challenge in computing systems. Traditionally, the development of such algorithms has been a manual and labor-intensive process. Our experience at a global cloud provider indicates that even seemingly simple algorithms used in production — such as cache replacement for data storage or bin packing for job scheduling — can require tens of thousands of engineering hours to design. As a result, practitioners often resort to generic heuristics from the literature, e.g., least-recently used (LRU) and least-frequently used (LFU) for cache replacement, and first-fit for bin packing, which frequently result in suboptimal performance.

This paper asks whether large language models (LLMs) can practically drive **algorithm generation**, *with an emphasis on principles to transform this task into a systematic process*. The core challenge in system algorithm design arises from the nature of its solution space: it is an inherently discontinuous space, where even a small change in algorithm design (e.g., data structure or control flow) can lead to sharp and non-linear changes in performance. Although it is sometimes possible to estimate the upper-bound performance, searching for practical solutions that approach this bound remains non-trivial. Furthermore, the discontinuous solution space does not provide sufficiently predictable patterns or a smooth landscape to guide the search.

Due to this discontinuity, we approach the algorithm generation task from a different angle, and frame it as a sampling process in the solution space. LLM attempts to generate distinct solutions at each step. This generative process represents a sequence of leaps in discontinuous solution space (Bubeck et al., 2023), which we formulate as **creative ideation** for LLMs. In fact, the systems community has long hypothesized algorithm design as a discovery process of ideas (Kant, 1985).

To study the algorithm generation task, we focus on high-impact problems at a global cloud provider: cache replacement and online bin packing. Our initial attempts of repeatedly sampling GPT-4o, Llama3.3-70B, and DeepSeek-V3 show that LLMs are fundamentally hindered by **availability bias** (Tversky & Kahneman, 1973) — LLMs are trained to output the most likely sequence of words, according to training datasets. As a result, solutions tend to cluster around well-known heuristics in the literature, e.g., LRU and LFU for caching. Furthermore, we find that this bias cannot be practically addressed through LLM hyperparameters like temperature (Ackley et al., 1985).

The key to creative ideation is exploiting knowledge that LLMs assume to be probabilistically irrelevant to the given problem. What is missing is a self-reflection process, which thinks how to

054 generate subsequent solutions by inspecting what solutions have been generated so far. In realizing  
 055 such a self-reflection framework, MetaMuse, we see three model-agnostic principles surfacing to best  
 056 guide this self-reflection. First, evaluating diversity of generated solutions should be grounded in  
 057 the measurable *feedback space* (e.g., simulation performance of system algorithms), rather than the  
 058 abstract idea space (Sankar & Sen, 2024). Second, steering the ideation is achieved through *external*  
 059 *stimuli* (e.g., keywords), rather than internal randomness (Honghua Chen and Nai Ding, 2023). Third,  
 060 developing executable solutions from external stimuli takes structured checkpoint-based steps, or  
 061 *waypoint reasoning*, rather than free-form chain-of-thought (Wei et al., 2023; Mehrotra et al., 2024).

062 Our work makes the following contributions:  
 063

- 064 1. Empirical analysis of LLMs’ fundamental limitations in the algorithm generation task (§2) —  
 065 shedding light on the impracticality of relying on LLMs’ internal randomness for ideation, based  
 066 on two online problems at a global cloud provider: cache replacement and online bin packing.
- 067 2. Principled framework for creative ideation, MetaMuse (§3) — systematically combining three  
 068 self-reflection principles, to guide the discontinuous leaps necessary to overcome LLMs’ bias.
- 069 3. Practical evaluation of the online algorithm generation task (§4) — showing that (1) MetaMuse  
 070 generates better cache replacement and bin packing solutions, outperforming LLM-based baselines  
 071 (by up to 9.89% fewer cache misses, and up to 21.06% less bin usage) and human heuristics (by up  
 072 to 35.76% fewer cache misses, and up to 30.93% less bin usage); (2) MetaMuse has up to 1.78×  
 073 more distinct cache replacement solutions and 1.80× more distinct bin packing solutions, than  
 074 LLM-based baselines; (3) MetaMuse has a low per-solution cost, up to 2.16 cents with GPT-4o.

## 076 2 BACKGROUND AND MOTIVATION

### 077 2.1 SYSTEM ALGORITHM DESIGN

078 System algorithms define how computing systems behave. They are typically designed to optimize  
 079 some performance objectives (e.g., cache hit ratio), for some scenarios or workloads (e.g., web  
 080 servers). The difficulty of designing such algorithms arises from the nature of their solution space.  
 081 It is discontinuous, where even a small change in algorithm design (e.g., the use of data structures)  
 082 can lead to sharp and non-linear changes in performance. In addition, the discontinuity does not  
 083 provide sufficiently predictable patterns or smooth landscape, to guide the search. This is a significant  
 084 departure from prior efforts on auto-tuning system config parameters (Alipourfard et al., 2017; Cortez  
 085 et al., 2017; Liang et al., 2020), which can be formulated as numerical optimization in most cases.

086 To this end, our work explores the use of LLMs to design heuristic algorithms, i.e., the algorithm  
 087 generation task. Due to the discontinuous solution space, we approach this task as a sampling process  
 088 in this space. At each step, LLMs attempt to generate distinct solutions. Conceptually, solving such a  
 089 discontinuous task requires certain “Eureka” ideas that constitute leaps in the progress towards the  
 090 final solution (Bubeck et al., 2023). Here, we refer to these leaps as creative ideation.

### 091 2.2 CREATIVE IDEATION

092 The goal of creative ideation is to discover useful solutions to a user-given problem, through a process  
 093 of generating diverse solutions over time. The first requirement is *usefulness* — a generated solution  
 094 should be relevant to the problem. The second requirement is *diversity* — a generated solution  
 095 should produce feedback (or an outcome) that is unseen in the current process (Boden, 1998). As we  
 096 accumulate diverse and useful solutions, the process advances towards finding the optimal solution.

097 For the algorithm generation task, creative ideation can be formulated as the following iterative  
 098 process. Each iteration takes in problem statements of the algorithm, along with the set of functions to  
 099 be implemented. For example, most caching algorithms can be abstracted into `insert` and `evict`  
 100 functions (Yang et al., 2020). We can also take in metadata and feedback of previous solutions, as  
 101 optional inputs. At iteration  $i$ , the output is an executable cache solution ( $c_i$ ) with all functions coded.  
 102 We can optionally provide feedback to iteration  $i + 1$ , by measuring hit ratio of previous solutions  
 103 ( $c_1^{hit}, \dots, c_i^{hit}$ ) on user-given workload trace, in an environment such as simulators. After  $n$  cache  
 104 solutions, we select the best-performing one from  $(c_1, \dots, c_n)$ .

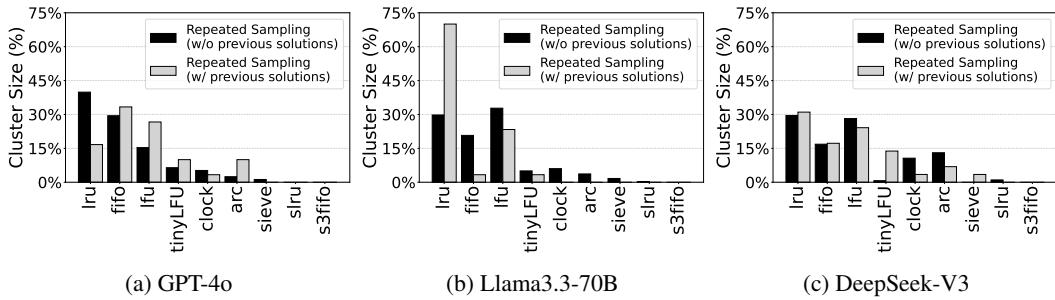


Figure 1: Repeatedly sampling LLMs can generate biased solutions. In the cache replacement problem, solutions tend to cluster around well-known ones in literature, e.g., LRU, LFU, and FIFO.

### 2.3 LLM LIMITATIONS IN CREATIVE IDEATION

Our work is motivated by the observation, where LLMs’ creative ideation capability is fundamentally limited by the same mechanism that enables it — the next-token prediction. In cognitive terms, responses are influenced by what one has frequently or recently seen, i.e., availability bias. For LLMs, this bias arises from the data frequency distribution in the training dataset.

**Impacts on algorithm generation.** Due to this bias, LLMs tend to generate certain designs. We illustrate with cache replacement, using GPT-4o (version: 1120), Llama3.3-70B (version: Instruct), and DeepSeek-V3 (version: 0324).

Figure 1 shows our attempts with *Repeated Sampling* (Brown et al., 2024; Snell et al., 2024). At iteration  $i$ , each LLM is prompted to create a new design  $c_i$ , with the default temperature of 1. Then,  $c_i$  is implemented by GPT-4o in Python, and benchmarked in the simulator to measure hit ratios on 30 synthetic traces. We try two approaches of Repeated Sampling, and they differ in whether each iteration’s prompt includes all previous solutions (i.e., solution descriptions and implementations). Figure 1 clusters 1,200 solutions into centroids of human heuristics, according to the similarity of hit ratios. The cluster sizes should ideally be uniform, but LLMs tend to design caches behaving similarly to well-known heuristics: LRU, LFU, and first-in-first-out (FIFO). Another observation is that making Repeated Sampling aware of previous solutions does not mitigate this bias.

Even if LLMs are instructed to generate new solutions by mutating previous solutions (Novikov et al., 2025; Sharma, 2025), we observe a bias towards tweaking the solution’s scoring function. LLMs tend to ignore other design dimensions such as data structure, hierarchical architecture, etc.

**Is adjusting LLM hyperparameters all you need?** One such parameter is temperature (Ackley et al., 1985), which internally regulates randomness in the generative process. Temperature smooths the probability distribution of next-token candidates, as computed by the softmax of their logits. At higher temperatures, high probabilities are decreased, and low probabilities are increased.

Unfortunately, high temperatures only mitigate availability bias, without practically addressing it. The reason is that increasing temperature smooths the probability distribution, which is a monotonic transformation retaining the relative ranking of output token candidates. Even at extreme temperatures, the distribution would approach flat and result in incoherent LLM outputs.

### 2.4 RELATED WORK

The first category is automatic heuristic design (Liu et al., 2024b; Ye et al., 2024; Zheng et al., 2025; Romera-Paredes et al., 2024; Novikov et al., 2025; Sharma, 2025). These efforts mostly rely on LLMs to evolve an initial population of candidate designs, through mutations and crossovers. This dependence on the initial population inherently constrains the reachable solution space. In addition, availability bias can impact how the population is evolved. Our case study highlights a bias towards frequently tweaking the cache algorithm’s scoring function, rather than leaping to other design dimensions. MetaMuse addresses these limitations by making leaps in the solution space.

There are also efforts exploring human-LLM collaborations to produce diverse outputs (Liu et al., 2024c; Wan et al., 2024; Vaccaro et al., 2024). However, they require human efforts and can be

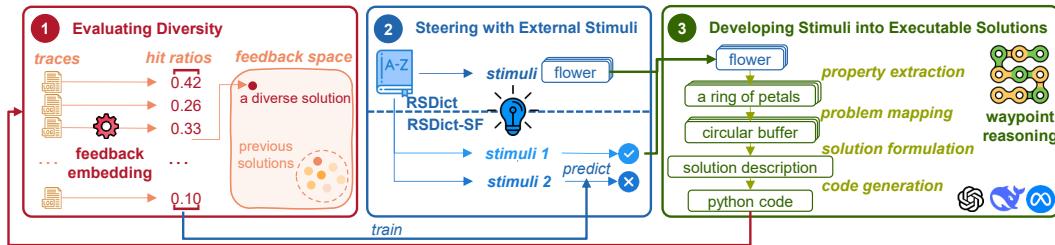


Figure 2: MetaMuse reflects on previous solutions, in order to think about how to generate subsequent solutions. Each iteration goes through three steps: evaluating the diversity of generated solutions (§3.1), steering ideation with external stimuli (§3.2), and developing executable solutions (§3.3).

subject to human bias. Instead, MetaMuse posits that LLMs themselves are capable of thinking “outside the box” (Mehrotra et al., 2024), and proposes to combine three self-reflection principles to achieve this capability (§3).

### 3 METAMUSE

Towards effective creative ideation, we address availability bias through a principled framework of self-reflection, *MetaMuse*. The key is to enable LLMs to think “outside the box,” i.e., exploiting knowledge that otherwise seems probabilistically irrelevant to solving the user problem. Figure 2 shows the overview of an iteration, which outputs one executable solution. The first step is to evaluate the diversity of generated solutions so far (§3.1). This evaluation is based on the embedding computed from each solution’s feedback (e.g., cache hit ratios in the figure). The second step is to steer ideation, by selecting a set of stimuli (e.g., the “flower” keyword in the figure) based on the diversity evaluation. §3.2 describes two stimuli selection strategies in the current implementation: RSDict and RSDict-SF. Finally, MetaMuse develops stimuli into executable solutions (§3.3), through four waypoints of reasoning: property extraction, problem mapping, solution formulation, and code generation.

#### 3.1 EVALUATING DIVERSITY

MetaMuse grounds diversity evaluation in feedback space, rather than idea space. In our case study, the former refers to performance measurements such as cache hit ratio, and the latter refers to the semantic embedding of algorithm descriptions or code implementations. MetaMuse represents each solution with a **feedback embedding**, which is a vector  $[p_1, p_2, \dots, p_n]$  of feedback measurements on  $n$  workload traces. Ideally, these traces should be different, and one way is to randomly synthesize traces with tools from the systems community (e.g., libCacheSim (Yang et al., 2020) for cache workloads).

This principle is motivated by the observation that semantic differences do not necessarily imply distinct solutions. Considering a case where GPT-4o generated functionally equivalent LFU-based solutions twice from different design descriptions: “*The cache evicts the object that is least frequently accessed*,” and “*The cache evicts the object with the lowest priority. Upon a cache hit, the object’s priority is incremented*.” Code semantics exhibit similar shortcomings, where a solution implemented with a linked list or a priority queue can have the same hit ratio.

Furthermore, feedback embedding has two desirable properties. First, since each dimension is a quantitative metric conveying magnitude, the Euclidean distance of two embeddings is a direct indication of how different corresponding solutions are. For the LFU case above, those two solutions would be considered equivalent because their distance is 0. Second, since feedback metrics typically pre-define a fixed range (e.g., 0–100% for cache hit ratio), the  $n$ -dimensional embedding space is inherently structured and bounded. This allows us to compute the steering direction (§3.2).

#### 3.2 STEERING WITH EXTERNAL STIMULI

The objective of steering is to guide LLMs to generate solutions, which targets regions of the feedback embedding space. Rather than relying on LLMs’ internal randomness, MetaMuse uses **external**

216 **stimuli** as the starting point for ideation. Stimuli can be unbiased to the problem, forcing LLMs to  
 217 associate with knowledge that seems probabilistically irrelevant. One domain-agnostic instantiation  
 218 of stimuli is keywords from an English dictionary.

219 Realizing this principle of steering has practical challenges. The first is the source of stimuli. In the  
 220 case of keywords, a large dictionary offers a broader keyword coverage, but it also contains many  
 221 keywords of little value. An example is technical jargon, e.g., “prolamin” in biology. In our case  
 222 study, if LLMs are forced to associate these keywords, they would frequently use them as variable  
 223 names, rather than inspiration to ideate new solutions. The second challenge is selecting  $s$  stimuli at  
 224 each iteration. §3.2.1 next presents selection strategies in our current implementation.

### 226 3.2.1 STIMULI SELECTION STRATEGIES

228 Our current implementation of MetaMuse includes two strategies: RSDict and RSDict-SF.

229 **RSDict.** RSDict is a stateless strategy that always randomly selects  $s$  keywords from the dictionary.  
 230 Since RSDict does not rely on evaluation results of previous solutions, it can steer scenarios where  
 231 evaluation is infeasible or costly (e.g., human judges).

233 **RSDict-SF.** Unlike RSDict, RSDict-SF considers the feedback embeddings of previous solutions,  
 234 to compute the steering direction at each iteration. Steering direction is specified by a feedback  
 235 embedding, which serves as the objective that RSDict-SF should aim to achieve in subsequent  
 236 solutions. In our case study, steering direction enables both exploration (for diversity) and exploitation  
 237 (for usefulness). For the former, the steering direction should be the farthest point from all previous  
 238 solutions, i.e., the feedback embedding that has the greatest Euclidean distance from all other  
 239 embeddings. For the latter, we set all dimensions of the target embedding vector to be a high value  
 240 (e.g., 100% cache hit ratio).

241 Given an embedding of steering direction, RSDict-SF computes the set of  $s$  stimuli that would likely  
 242 develop into solutions close to it. Our current implementation formulates this step as a prediction  
 243 problem, and solves it with Gaussian Process Regression (GPR) models. Given the sum of semantic  
 244 embeddings of  $s$  stimuli, we use  $n$  GPR models ( $\mathcal{M}_1, \dots, \mathcal{M}_n$ ), to predict each of the  $n$  dimensions  
 245 in embedding. Having ( $\mathcal{M}_1, \dots, \mathcal{M}_n$ ) allows RSDict-SF to predict the expected feedback embedding  
 246 of any set of stimuli. GPR models are fitted with all previous solutions at each iteration. We use  
 247 the dot-product kernel for GPR models. Dot-product kernel  $\mathcal{K}$  can capture the pairwise semantic  
 248 similarity between all observations across two solutions and is invariant to their ordering. For two  
 249 solutions  $c_i$  and  $c_j$ , their similarity captured by  $\mathcal{K}$  is

$$250 \quad 251 \quad 252 \quad \mathcal{K}(c_i, c_j) \propto \sum_{p=1}^s \sum_{q=1}^s \text{embedding}(o_{i,p})^\top \text{embedding}(o_{j,q}),$$

253 where  $o_{i,p}$  denotes the  $s$  observations derived from  $c_i$ ’s  $s$  stimuli, and similar for  $o_{j,q}$ .

254 We highlight two implementation details. First, predicting for *all* possible sets of stimuli is not  
 255 feasible in practice, and a dictionary with 3,000 common English words would already have 3,000<sup>s</sup>  
 256 sets. To this end, MetaMuse exercises the power-of-two random choices (Mitzenmacher et al.,  
 257 2001). It randomizes two sets of stimuli from the dictionary, and predicts their expected feedback  
 258 embeddings. Then, MetaMuse selects the one closest to the target embedding. Second, RSDict-SF is  
 259 bootstrapped, with  $w$  solutions generated by RSDict, in order to start training GPR models.

### 261 3.3 DEVELOPING STIMULI INTO EXECUTABLE SOLUTIONS

263 Third, developing executable solutions from  $s$  stimuli involves structured **waypoint reasoning**.  
 264 Waypoints are intermediate checkpoints, which develop seemly unrelated stimuli (c.f. §3.2.1) into  
 265 solutions for the problem at hand. In contrast to free-form chain-of-thought (Wei et al., 2023;  
 266 Mehrotra et al., 2024), we observe that waypoints prevent LLMs from superficially developing  
 267 solutions, e.g., simply turning stimuli into variable names in code.

268 MetaMuse currently specifies the following waypoints. The first waypoint focuses on *property*  
 269 *extraction*, where LLMs are prompted to associate the given stimuli, to related concepts and properties.  
 Considering the keyword “flower”, one associated property can be “a ring of petals”. These outputs

270 are then fed to the second waypoint, *problem mapping*. Here, LLMs are prompted to associate to  
 271 problem-related observations. Continuing our example of “a ring of petals”, LLMs can associate the  
 272 shape, to the observation of circular buffer in algorithm design. These observations are then fed to the  
 273 third waypoint, *solution formulation*. LLMs are prompted to combine observations, and synthesize  
 274 the complete description of a new solution. The final waypoint is *code generation*, where LLMs turn  
 275 the solution description into executable Python code. Waypoint prompts are included in §C.2.

## 277 4 EMPIRICAL RESULTS

278 **Baselines.** We have 21 extensive baselines. LLM-based heuristics design baselines include MCTS-  
 279 AHD (Zheng et al., 2025), ReEvo (Ye et al., 2024), OpenEvolve (Sharma, 2025), PlanSearch (Wang  
 280 et al., 2025), and Repeated Sampling (Brown et al., 2024). We also include human-crafted SOTA  
 281 heuristics for cache replacement problem (LRU, LFU, FIFO, Sieve (Zhang et al., 2024), S3FIFO (Yang  
 282 et al., 2023), TinyLFU (Einziger et al., 2017), SLRU (Huang et al., 2013), Clock (Corbato, 1969),  
 283 and ARC (Megiddo et al., 2003)), and online bin packing problem (Next Fit, Worst Fit, Almost  
 284 Worst Fit (AWF) (Johnson, 2010), First Fit (Dósa & Sgall, 2013), Best Fit (Dósa & Sgall, 2014),  
 285 Harmonic-k (Lee & Lee, 1985), and Refined First Fit (RFF) (Yao, 1980)). For Harmonic-k, we set  
 286  $k=4$  to align its number of bin categories with RFF.

287 **MetaMuse.** We take a dictionary of common English words and remove stop-words to get 2,899  
 288 keywords. Each solution is ideated from  $s=4$  keywords. The feedback embedding consists of  
 289 performance measurements on  $n=30$  traces. For cache replacement, these  $n$  traces are generated by  
 290 libCacheSim (Yang et al., 2020), from different Zipfian distributions. For bin packing,  $n$  traces are  
 291 from various Weibull distributions. The number of RSDict-SF warmup solutions is  $w=100$ , roughly  
 292 one-third of the total solutions in one experiment. Finally, prompts are included in §C.

293 **Experiment setup.** We focus on two high-impact problems at a global cloud provider: cache  
 294 replacement and bin packing. In each experiment, all methods aim to ideate and generate 350  
 295 executable solutions. To the best of our ability, we configure PlanSearch to output at least 15, 11, and  
 296 12 observations, while ideating with GPT-4o, Llama3.3-70B, and DeepSeek-V3, respectively. Many  
 297 baselines such as MCTS-AHD can tune parameters in their solutions. §5 discusses safeguards against  
 298 unsafe solutions, and environments are instrumented to catch errors, e.g., long-running execution.  
 299 Unsafe solutions are re-implemented. Coding prompts are in §C.2.

300 To evaluate cache replacement solutions, we use 96 real-world workload traces from four data access  
 301 scenarios (Table 1): RetrievalAttention (Liu et al., 2024a) (24 “ra-fwe” and 24 “ra-multikey” traces),  
 302 Tencent block storage (Zhang et al., 2020; 2018) (24 “tencent-storage” traces), and Alibaba cloud  
 303 storage (ali; Li et al., 2020; Wang et al., 2022) (24 “alibaba-storage” traces). The cache capacity is  
 304 set to 10% of the number of distinct objects in each trace.

305 To evaluate online bin packing solutions, we use 288 workload traces: BPPLIB library (Delorme  
 306 et al.) (72 “Falkenauer-U” and 72 “Scholl-1” traces), Weibull distribution with parameter ( $shape=3$ ,  
 307  $scale=45$ ) (Castineiras et al., 2012) (72 “Weibull”), and Gaussian distribution with parameter  
 308 ( $mean=0.3662$ ,  $std=0.1416$ ) (Yan et al., 2022) (72 “Gaussian”).

### 312 4.1 METAMUSE GENERATES HIGH-PERFORMING SOLUTIONS

313 **Cache replacement.** Figure 3 compares top solutions, as selected by the average cache miss ratio  
 314 over all 96 workload traces. Box plots show their miss ratio reduction, with respect to FIFO heuristics.

315 We first delve into results from GPT-4o. At the 90<sup>th</sup>-percentile trace, MetaMuse achieves 5.17%–  
 316 9.89% lower miss ratio than LLM-based baselines, and 1.75%–13.03% lower than human heuristics.  
 317 At the 75<sup>th</sup>-percentile trace, MetaMuse achieves 3.62%–6.39% lower miss ratio than LLM-based  
 318 baselines, and 6.62%–35.76% lower than human heuristics. We also see improvement with Llama3.3-  
 319 70B — at the 90<sup>th</sup>-percentile trace, MetaMuse achieves 2.41%–7.20% lower miss ratio than LLM-  
 320 based baselines, and 0.94%–10.34% lower than human heuristics; at the 75<sup>th</sup>-percentile trace,  
 321 MetaMuse achieves 3.67%–5.17% lower miss ratio than LLM-based baselines, and 5.48%–34.62%  
 322 lower than human heuristics. On DeepSeek-V3, at the 90th-percentile trace, MetaMuse achieves up  
 323 to 6.05% lower miss ratio than LLM-based baselines, and up to 9.14% lower than human heuristics.

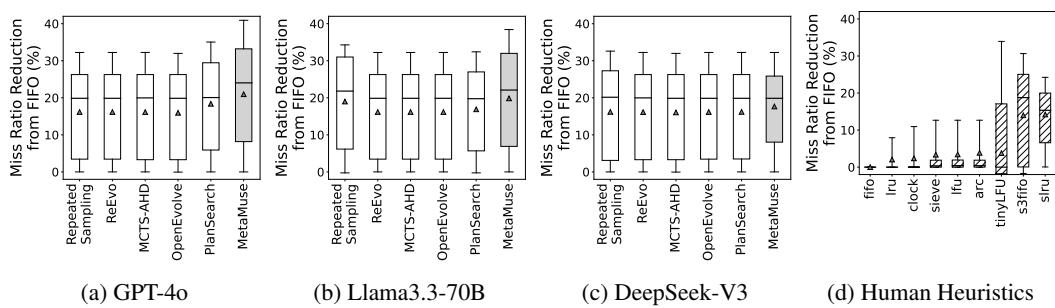


Figure 3: Comparisons of top cache solutions generated by *MetaMuse* and baselines. Each box plot represents the best solution from each model, and shows the miss ratio reduction (with respect to FIFO heuristics) achieved over 96 traces. *MetaMuse* has higher reduction across nearly all percentiles, on different LLMs.

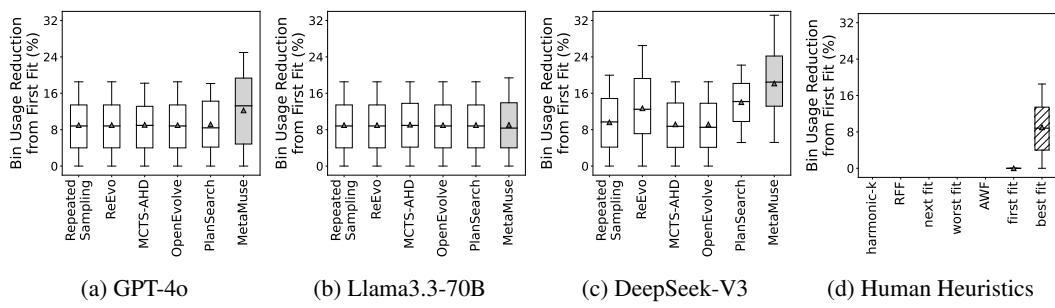


Figure 4: Comparisons of top online bin packing solutions generated by *MetaMuse* and baselines. Each box plot represents the best solution from each model, and shows the bin usage reduction (with respect to First Fit heuristics) achieved over 288 traces. *MetaMuse* has higher reduction across nearly all percentiles, on different LLMs. We note that some human heuristics are not visible because they perform worse than First Fit.

**Online bin packing.** Figure 4 compares top solutions, as selected by the average bin usage over all 96 workload traces. Box plots show their bin usage reduction, with respect to First Fit heuristics.

We first delve into results from GPT-4o. At the 90<sup>th</sup>-percentile trace, *MetaMuse* achieves 9.25%–9.42% lower bin usage than LLM-based baselines, and 9.25%–20.59% lower than human heuristics. At the 75<sup>th</sup>-percentile trace, *MetaMuse* achieves 6.94%–7.75% lower bin usage than LLM-based baselines, and 7.56%–18.36% lower than human heuristics. With Llama3-7B, at the 90<sup>th</sup>-percentile trace, *MetaMuse* achieves up to 0.19% less bin usage than LLM-based baselines, and up to 12.50% lower bin usage than human heuristics. With DeepSeek-V3, at the 90<sup>th</sup>-percentile trace, *MetaMuse* achieves up to 21.06% less bin usage than LLM-based baselines, and up to 30.93% lower bin usage than human heuristics.

## 4.2 METAMUSE GENERATES THE MOST DIVERSE SET OF SOLUTIONS

**Cache replacement.** The discovery of useful solutions depends on having diverse solutions. We evaluate diversity by the number of solutions with a distinct feedback embedding. Across all LLMs, *MetaMuse* consistently achieves higher diversity over LLM-based baselines. From analyzing 350 solutions generated by each method on GPT-4o, *MetaMuse* has 1.47× more distinct solutions on average. On DeepSeek-V3, *MetaMuse* has 1.57× more distinct solutions on average. On Llama3.3-70B, results show that *MetaMuse* can have 1.78× more distinct solutions on average.

Furthermore, we observe that having a higher diversity translates into lower availability bias. We recall that Repeated Sampling tends to generate solutions that tightly cluster to well-known human heuristics such as LRU, LFU, and FIFO, due to availability bias (§2.3). Using the nine human heuristics as centroids, empirical results show that *MetaMuse* reduces the cluster density (computed by  $\frac{\text{num\_points\_in\_cluster}}{\text{max\_euclidean\_distance}}$ ), for nearly all centroids. In other words, *MetaMuse* solutions are less

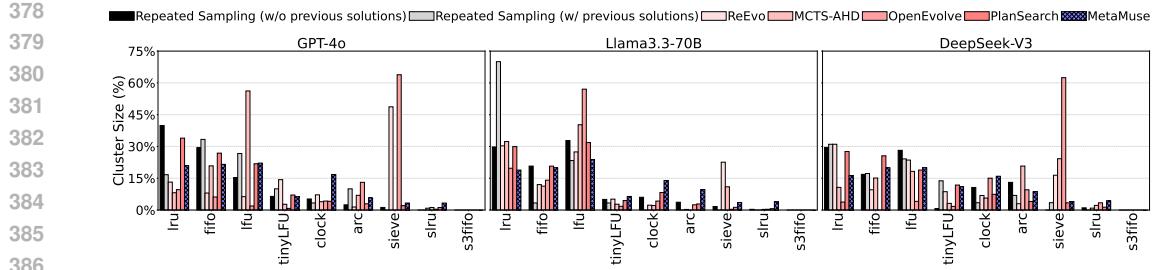


Figure 5: MetaMuse generates a more diverse set of solutions. Using human heuristics as centroids, results show that the cluster sizes for MetaMuse solutions have a lower standard deviation. In contrast, baselines can cluster to a subset of centroids.

tightly clustered than Repeated Sampling. For example, MetaMuse reduces the LRU cluster density by 35.25% (i.e., 6.15 down to 3.98). We see cluster density reduction even on the less common heuristics, e.g., the ARC cluster density reduces by 77.45% (i.e., 1.02 down to 0.23).

Finally, Figure 5 uses human heuristics as centroids, and evaluate whether generated solutions tend to cluster to a subset of centroids. Ideally, the cluster sizes should be uniform. The standard deviation of cluster size percentages for MetaMuse is 0.08/0.06/0.06, for GPT-4o/DeepSeek-V3/Llama3-70B. This is much lower than baselines: Repeated Sampling (0.14/0.11/0.11), Repeated Sampling with previous solutions (0.11/0.11/0.11), ReEvo (0.14/0.10/0.10), MCTS-AHD (0.17/0.08/0.08), OpenEvolve (0.19/0.19/0.19), and PlanSearch (0.12/0.10/0.99).

**Online bin packing.** We evaluate diversity by the number of solutions with a distinct feedback embedding. Across all LLMs, MetaMuse consistently achieves a higher diversity over LLM-based baselines. From analyzing 350 solutions generated by each method on GPT-4o, MetaMuse has  $1.44 \times$  more distinct solutions on average. On DeepSeek-V3, it has  $1.80 \times$  more distinct solutions on average. On Llama3-70B, it has  $1.31 \times$  more distinct solutions on average.

### 4.3 METAMUSE HAS A LOW PER-SOLUTION COST

We see having a low per-solution cost, as a factor driving the adoption of LLMs for creative ideation. Considering the cache case study, the average cost for MetaMuse to generate *one* full solution is 2.16 cents with GPT-4o, 2.11 cents with DeepSeek-V3, and 2.35 cents with Llama3.3-70B. These costs include using GPT-4o for code generation. For reference, running the Repeated Sampling baseline (with previous solutions) costs an average of 3.38 cents per solution.

Specifically, property extraction and problem mapping consume an average of 589.39/141.54 input/output tokens per solution with GPT-4o, 982.60/233.75 input/output tokens with DeepSeek-V3, and 768.20/207.50 input/output tokens with Llama3.3-70B. Solution formulation consumes an average of 934.04/230.21 input/output tokens per solution with GPT-4o, 473.01/425.77 input/output tokens with DeepSeek-V3, and 409.53/353.10 input/output tokens with Llama3.3-70B. Code generation consumes an average of 1463.92/1190.39 input/output tokens per solution with GPT-4o, 2223.72/1937.29 input/output tokens with DeepSeek-V3, and 2259.62/1979.89 input/output tokens with Llama3.3-70B. We note that if an implementation is considered unsafe (§5), it is re-implemented.

### 4.4 SURPRISING DESIGNS FROM METAMUSE

In addition to quantitative evaluations, we highlight MetaMuse-generated designs that are not immediately obvious to engineers. To do so, we look at top-performing cache replacement solution: *MetaMuse-533* (§D.1) and *MetaMuse-488* (§D.2).

One is the counter named NSE in *MetaMuse-533*. NSE deviates from the common belief of favoring newly admitted objects. Instead, it tracks the number of times that a cached object has witnessed eviction events. It is used in the score calculation, along with recent access time and frequency counter. Effectively, *MetaMuse-533* behaves to favor objects that have remained longer in the cache (up to some thresholds). Engineers hypothesize that NSE helps to learn objects' access patterns and prevent thrashing (Denning & J., 1968).

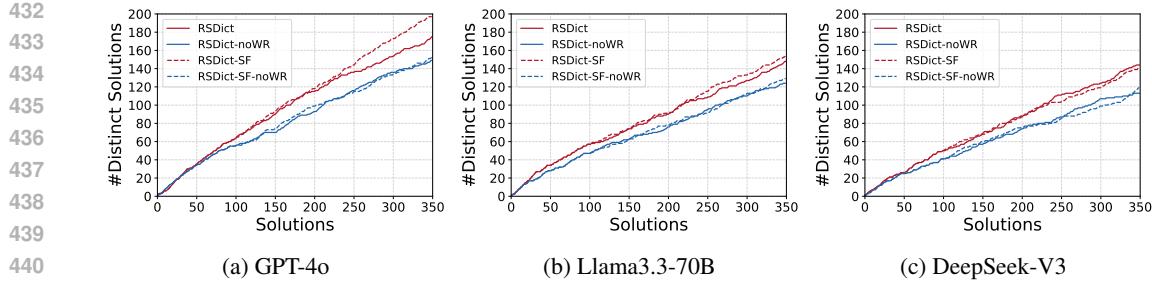


Figure 6: Comparisons of the diversity of cache replacement solutions, generated by different stimuli selection strategies: RSDict and RSDict-SF. “-noWR” means removing waypoint reasoning. We ideate with different LLMs, and code with GPT-4o. RSDict-SF (dashed lines) outperforms RSDict (solid lines). Removing waypoint reasoning (blue lines) reduces solution diversity (red lines).

Another is the use of hashing functions to segment the cache space in *MetaMuse-488*, which differs from the more common multi-tiered architecture. *MetaMuse-488* tries to maintain different groups of cache objects (as identified by their key hashes), and exercises per-group replacement policies in different segments. In contrast, engineers tend to use multi-tiered architecture, as a way to differentiate the importance of cache objects.

Finally, we highlight the use of saturating counters in *MetaMuse-533*. They increment until some pre-defined threshold. *MetaMuse-533* deviates from the typical usage scenarios, which are to reduce storage overhead by capping the maximum value of a counter (Yang et al., 2023; Corbato, 1969). Instead, *MetaMuse-533* seems to use saturating counters to accumulate meaningful usage history, but not so much as to mislead eviction decisions. For example, a saturating frequency counter allows *MetaMuse-533* to identify bursty objects, while preventing their counts from growing too large during the burst.

#### 4.5 ABLATION STUDY

We test key components in *MetaMuse*, with the case study of cache replacement algorithms.

**Stimuli selection strategies.** We test RSDict and RSDict-SF by running experiments on GPT-4o. First, the majority of RSDict and RSDict-SF solutions outperform Repeated Sampling solutions, and this demonstrates the value of having stimuli for steering ideation. Comparing solutions for the 90<sup>th</sup>-percentile workload trace, 67.20% of RSDict solutions and 75.60% of RSDict-SF solutions have a higher cache hit ratio than Repeated Sampling. In addition, these hit ratios can be up to 13.13% and 27.07% higher, respectively. Similarly, for the 75<sup>th</sup>-percentile workload trace, we see 66.80% of RSDict solutions and 72.40% of RSDict-SF solutions outperform Repeated Sampling. In addition, these hit ratios can be up to 15.43% and 21.93% higher, respectively.

Second, compared to RSDict, RSDict-SF has a greater number of solutions that can outperform Repeated Sampling. Empirical results show 12.50% and 8.38% more solutions, with respect to the 90<sup>th</sup> percentile and 75<sup>th</sup> percentile of workload traces, respectively. Furthermore, looking at solution diversity, RSDict-SF results in 13.17% more distinct solutions than RSDict.

**Waypoint reasoning.** Figure 6 shows that waypoint reasoning enables *MetaMuse* to generate more diverse solutions. We fix the coding agents to GPT-4o. At each iteration, we instrument *MetaMuse* to generate solutions with and without waypoint reasoning. When ideating with GPT-4o (Figure 6a), waypoint reasoning improves the number of distinct solutions from 149 to 175 for RSDict, and from 152 to 197 for RSDict-SF. When ideating with DeepSeek-V3 (Figure 6c), waypoint reasoning improves the number of distinct solutions from 113 to 144 for RSDict, and from 119 to 140 for RSDict-SF. When ideating with Llama3.3-70B (Figure 6b), waypoint reasoning improves the number of distinct solutions from 124 to 148 for RSDict, and from 129 to 154 for RSDict-SF.

## 486 5 DISCUSSION

488 **Safeguards against unsafe solutions.** In the case of system algorithms, three main errors result  
 489 in unsafe implementations: (1) long-running execution (e.g., more than 5 seconds), (2) excessive  
 490 memory usage over the user-specified requirement, and (3) illegal behaviors (e.g., falsely claiming  
 491 a cache hit when the requested object is not present). Our benchmark environments monitor the  
 492 per-benchmark execution time and peak memory usage, while running each solution. Both can be  
 493 collected from the GNU `time` command. For catching illegal behaviors, we check the returned  
 494 object value against the expected value in the trace. From our experience,  $\sim 2.28\%$  of solutions  
 495 generated by MetaMuse are classified as being unsafe, and subsequently discarded.

496 Finally, with Python scripts as the output, it is possible to leverage libraries in the Python ecosystem.  
 497 We discuss two on-going efforts. One is performing unit tests, with libraries such as `unittest` (uni  
 498 (2025)). Another is the design-and-code equivalence; we are exploring the use of LLMs to generate  
 499 state machines from design descriptions and code implementations, for the equivalence check.

500 **Limitations and future discussion.** First, our implementation explores an instantiation of external  
 501 stimuli (§3.2), which is specifically chosen for being general and problem-agnostic. However, the  
 502 MetaMuse framework is extensible to take in other instantiations, and future work will explore  
 503 possibilities such as long-term task-related memories. Second, our work currently focuses on fully  
 504 automated ideation, and an interesting direction is to explore whether subjective human guidance (e.g.,  
 505 human expertise) could further improve the solution usefulness. Third, we evaluate on real-world  
 506 workload traces (§4), and deploying LLM-generated algorithms to a business-critical production  
 507 environment remains an ongoing collaboration with engineers at a global cloud provider. We look  
 508 forward to leverage lessons learned for deeper practical impacts.

509 **Algorithm benchmarking costs.** The RSDict-SF strategy considers the feedback embeddings  
 510 of previous solutions (c.f. §3.2.1), and this requires benchmarking solutions on  $n$  workload traces.  
 511 Benchmarks can take a significant amount of time and resources, especially if they involve running  
 512 an end-to-end system with long workload traces. If costs are prohibitively high, RSDict-SF may  
 513 impact the practicality of MetaMuse. We note that benchmarking costs is a problem that the Systems  
 514 community has been actively addressing Akram & Sawalha (2019), with techniques such as discrete-  
 515 event simulations, hardware acceleration, parallelization, and so on. At the same time, we are  
 516 exploring stimuli selection strategies that have a lower requirement on running benchmarks.

517 **Synergy with existing approaches.** Although MetaMuse focuses on creative leaps, we recognize  
 518 the value in related approaches that iteratively refine algorithm designs (Liu et al., 2024b; Ye et al.,  
 519 2024; Zheng et al., 2025; Romera-Paredes et al., 2024; Novikov et al., 2025; Sharma, 2025). In  
 520 fact, both approaches are orthogonal, in a way similar to human’s divergent and convergent thinking  
 521 processes. MetaMuse can provide the initial population of solution candidates, which can then be  
 522 further improved by the iterative refinement approach.

## 523 6 CONCLUSION

525 To practically drive algorithm generation, MetaMuse is a framework of creative ideation that mitigates  
 526 LLMs’ availability bias. MetaMuse combines three self-reflection principles, guiding the stages of  
 527 creative ideation. Evaluations show that it can generate high-performing solutions for two high-impact  
 528 problems at a global cloud provider: cache replacement and bin packing algorithms. Furthermore,  
 529 we observe surprising design considerations not immediately obvious to human engineers.

## 531 7 REPRODUCIBILITY STATEMENT

533 We submit anonymized code in a zip file, as part of supplementary materials. Key prompts are  
 534 included in appendix (§C). After the paper is published, code and datasets will be made public in our  
 535 GitHub repository.

## 537 8 ETHICS STATEMENT

539 This work poses no ethical issues.

540 REFERENCES  
541

542 Alibaba block trace. URL <https://github.com/alibaba/block-traces>.

543 unittest — Python Unit Testing Framework, 2025. URL <https://docs.python.org/3/library/unittest.html>.

544

545 David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. A Learning Algorithm for Boltzmann Machines. *Cognitive Science*, 9(1), 1985.

546

547 Ayaz Akram and Lina Sawalha. A survey of computer architecture simulation techniques and tools. *IEEE Access*, 2019.

548

549

550 Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI*. USENIX, 2017.

551

552

553 Margaret A. Boden. Creativity and Artificial Intelligence. *Artificial Intelligence*, 1(2), 1998.

554

555 Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher R, and Azalia Mirhoseini. Large Language Monkey: Scaling Inference Compute with Repeated Sampling, 2024.

556

557

558 Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of Artificial General Intelligence: Early experiments with GPT-4, 2023.

559

560

561

562 Ignacio Castineiras, Milan de Cauwer, and Barry O’Sullivan. Weibull-based benchmarks for bin packing. pp. 207–222, 01 2012. ISBN 9783642335570. doi: 10.1007/978-3-642-33558-7\_17.

563

564

565 F. J. Corbato. A paging experiment with the multics system, 1969. URL <https://multicians.org/paging-experiment.pdf>.

566

567 Cortez, Eli, Bonde, Anand, Muzio, Alexandre, Russinovich, Mark, Fontoura, Marcus, Bianchini, and Ricardo. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *SOSP*. ACM, 2017.

568

569

570 M. Delorme, M. Iori, and S. Martello. BPPLIB. URL <https://site.unibo.it/operations-research/en/research/bplib-a-bin-packing-problem-library>.

571

572

573

574 Denning and Peter J. Thrashing: its causes and prevention. In *Proceedings of the December 9-11 and 1968 and Fall Joint Computer Conference and Part I*, AFIPS ’68 (Fall and part I), pp. 915–922, New York and NY and USA, 1968. Association for Computing Machinery. ISBN 9781450378994. doi: 10.1145/1476589.1476705. URL <https://doi.org/10.1145/1476589.1476705>.

575

576

577

578 György Dósa and Jiri Sgall. First Fit bin packing: A tight analysis. In Natacha Portier and Thomas Wilke (eds.), *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, volume 20 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 538–549, Dagstuhl, Germany, 2013. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-939897-50-7. doi: 10.4230/LIPIcs.STACS.2013.538. URL <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.STACS.2013.538>.

579

580

581

582

583

584 György Dósa and Jiří Sgall. Optimal analysis of best fit bin packing. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias (eds.), *Automata, Languages, and Programming*, pp. 429–441, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-43948-7.

585

586

587

588 Einziger, Gil, Friedman, Roy, Manes, and Ben. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Trans. Storage*, 2017.

589

590 Honghua Chen and Nai Ding. Probing the Creativity of Large Language Models: Can Models Produce Divergent Semantic Association. In *EMNLP*, 2023.

591

592

593 Huang, Qi, Birman, Ken, van Renesse, Robbert, Lloyd, Wyatt, Kumar, Sanjeev, Li, and Harry C. An Analysis of Facebook Photo Caching. In *SOSP*. ACM, 2013.

594 David Johnson. Near-optimal bin packing algorithms. 08 2010.  
 595

596 Elaine Kant. Understand and Automating Algorithm Design. In *Transactions on Software Engineering*.  
 597 IEEE, 1985.

598 C. C. Lee and D. T. Lee. A simple on-line bin-packing algorithm. *J. ACM*, 32(3):562–572, July  
 599 1985. ISSN 0004-5411. doi: 10.1145/3828.3833. URL <https://doi.org/10.1145/3828.3833>.

600

601 Li, Jinhong, Wang, Qiuping, Lee, Patrick P. C., Shi, and Chao. An In-Depth Analysis of Cloud Block  
 602 Storage Workloads in Large-Scale Production. In *IISWC*. IEEE, 2020.

603

604 Chieh-Jan Mike Liang, Hui Xue, Mao Yang, Lidong Zhou, Lifei Zhu, Zhao Lucis Li, Zibo Wang,  
 605 Qi Chen, Quanlu Zhang, Chuanjie Liu, and Wenjun Dai. AutoSys: The Design and Operation of  
 606 Learning-Augmented Systems. In *ATC*. USENIX, 2020.

607

608 Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chen-  
 609 gruidong Zhang, Bailu Ding, Kai Zhang, Chen Chen, Fan Yang, Yuqing Yang, and Lili Qiu.  
 610 RetrievalAttention: Accelerating Long-Context LLM Inference via Vector Retrieval. In *NeurIPS*  
 611 *Workshop on Efficient Natural Language and Speech Processing*, 2024a.

612

613 Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu  
 614 Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language  
 615 model. In *Forty-first International Conference on Machine Learning*, 2024b. URL <https://openreview.net/forum?id=BwAkaxqilB>.

616

617 Yiren Liu, Pranav Sharma, Mehul Jitendra Oswal, Haijun Xia, and Yun Huang. PersonaFlow:  
 618 Boosting Research Ideation with LLM-Simulated Expert Personas, 2024c.

619

620 Megiddo, Nimrod, Modha, and Dharmendra S. ARC: A Self-Tuning and Low Overhead Replacement  
 621 Cache. In *FAST*. USENIX, 2003.

622

623 Pronita Mehrotra, Aishni Parab, and Sumit Gulwani. Enhancing creativity in large language mod-  
 624 els through associative thinking strategies, 2024. URL <https://arxiv.org/abs/2405.06715>.

625

626 Michael Mitzenmacher, Andrea W. Richa, and Ramesh Sitaraman. The Power of Two Random  
 627 Choices: A Survey of Techniques and Results, 2001.

628

629 Alexander Novikov, Ngan Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wag-  
 630 ner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan  
 631 Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Push-  
 632 meet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery.  
 633 Technical report, DeepMind, 2025.

634

635 Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog,  
 636 M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang,  
 637 Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical Discoveries from Program  
 638 Search with Large Language Models. In *Nature*, 2024.

639

640 B. Sankar and Dibakar Sen. A Novel Mathematical Framework for Objective Characterization of  
 641 Ideas through Vector Embeddings in LLM, 2024.

642

643 Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025. URL <https://github.com/codelion/openevolve>.

644

645 Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM Test-Time Compute  
 646 Optimally can be More Effective than Scaling Model Parameters, 2024.

647

648 Amos Tversky and Daniel Kahneman. Availability: A Heuristic for Judging Frequency and Probabil-  
 649 ity. *Cognitive Psychology*, 5(2), 1973.

650

651 Vaccaro, Michelle, Almaatouq, Abdullah, Malone, and Thomas. When Combinations of Humans  
 652 and AI are Useful: A Systematic Review and Meta-analysis. *Nature Human Behaviour*, 2024.

648 Wan, Qian, Hu, Siying, Zhang, Yu, Wang, Piaohong, Wen, Bo, Lu, and Zhicong. It Felt Like Having  
 649 a Second Mind: Investigating Human-AI Co-creativity in Prewriting with Large Language Models.  
 650 *HCI*, 2024.

651

652 Evan Z Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, William Song, Vaskar Nath, Ziwen  
 653 Han, Sean M. Hendryx, Summer Yue, and Hugh Zhang. Planning in Natural Language Improves  
 654 LLM Search for Code Generation. In *ICLR*, 2025.

655

656 Qiuping Wang, Jinhong Li, Patrick P. C. Lee, Tao Ouyang, Chao Shi, and Lilong Huang. Separating  
 657 data via block invalidation time inference for write amplification reduction in Log-Structured  
 658 storage. In *FAST*. USENIX, 2022.

659

660 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc  
 661 Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models,  
 662 2023.

663

664 Jie Yan, Yunlei Lu, Liting Chen, Si Qin, Yixin Fang, Qingwei Lin, Thomas Moscibroda, Saravan  
 665 Rajmohan, and Dongmei Zhang. Solving the Batch Stochastic Bin Packing Problem in Cloud: A  
 666 Chance-constrained Optimization Approach. In *KDD*. ACM, 2022.

667

668 Yang, Juncheng, Zhang, Yazhuo, Qiu, Ziyue, Yue, Yao, Vinayak, and Rashmi. FIFO queues are all  
 669 you need for cache eviction. In *SOSP*. ACM, 2023.

670

671 Juncheng Yang, Yao Yue, and K. V. Rashmi. A Large Scale Analysis of Hundreds of In-memory  
 672 Cache Clusters at Twitter. In *OSDI*. USENIX, 2020.

673

674 Andrew Chi-Chih Yao. New Algorithms for Bin Packing. *Journal of ACM*, 1980.

675

676 Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park,  
 677 and Guojie Song. Reeve: Large language models as hyper-heuristics with reflective evolution. In  
 678 *Advances in Neural Information Processing Systems*, 2024. <https://github.com/ai4co/reevo>.

679

680 Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K.V. Rashmi. SIEVE is simpler than  
 681 LRU: an efficient Turn-Key eviction algorithm for web caches. In *NSDI*. USENIX, 2024.

682

683 Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. Tencent  
 684 block storage traces (SNIA IOTTA trace 27920). In Geoff Kuennning (ed.), *SNIA IOTTA Trace*  
 685 *Repository*. Storage Networking Industry Association, September 2018. URL <http://iotta.snia.org/traces/parallel/27917?only=27920>.

686

687 Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA:  
 688 An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems. In *ATC*.  
 689 USENIX, 2020.

690

691

692

693

694

695

696

697

698

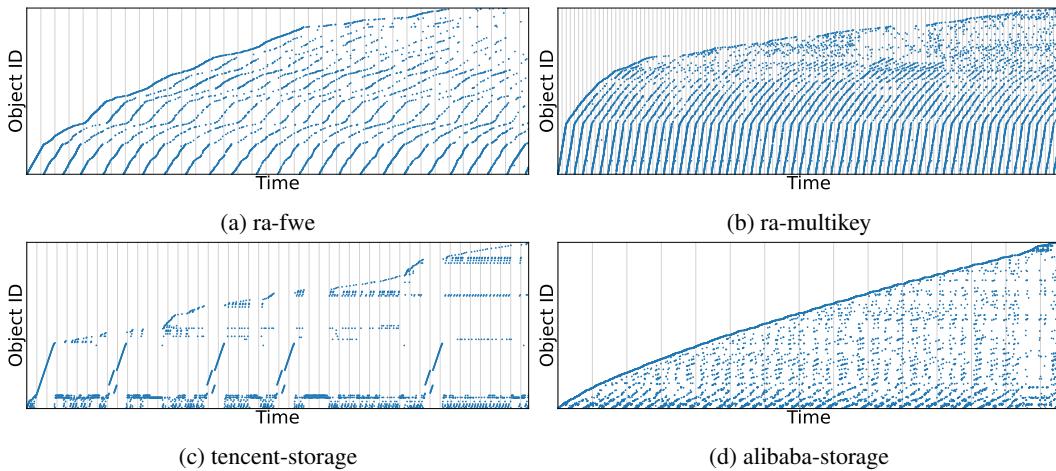
699

700

701

702  
703 A LLM USAGE STATEMENT704  
705 LLMs did not play a significant role in research ideation and writing, and they are not regarded as  
706 contributors.707  
708 B CACHE WORKLOAD TRACES709  
710 We detail the traces for evaluation on cache replacement in Table 1 (c.f. §4). To show their access  
711 patterns, we plot one trace from each data access scenario (Figure 7). The  $x$ -axis denotes a virtual  
712 timestamp, which is the number of cache accesses thus far. The  $y$ -axis denotes the accessed objects.  
713 Their IDs are sequentially assigned, following the chronological order of the time they appear for the  
714 first time. The intervals between the gray vertical lines equal the cache capacity, which is set as 10%  
715 of the trace footprint (i.e., the number of distinct object IDs).

scenario	ra-fwe	ra-multikey	tencent-storage	alibaba-storage
release year	2024	2024	2020	2024
avg #cache accesses	10598.42	27390.71	53503.79	7582.08
avg #distinct objects	938.54	1092.29	225.125	4108.83

721  
722 Table 1: The data access scenarios of cache replacement used in this work. Each scenario consists of  
723 24 cache traces.741  
742 Figure 7: The access pattern of one cache trace from each data access scenario.743  
744 C PROMPTS745  
746 C.1 IDEATION PROMPTS747  
748 Repeated Sampling uses the following prompt for solution formulation, generating its natural language  
749 description:750  
751 **The Ideation Prompt for Repeated Sampling**752  
753 You are an expert in computer systems. Your task is to create an innovative cache replacement  
754 policy.

755 Provide your creative policy using the following JSON structure:

```

756
757     "json
758     {
759         "metadata": "use a few sentences to summarize the metadata specifically maintained by the
760         policy here",
761         "evict": "use a few sentences to summarize how the policy chooses the eviction victim here",
762         "update_after_hit": "use a few sentences to summarize how the policy updates **each** of
763             the metadata it maintains immediately after a cache hit here",
764         "update_after_insert": "use a few sentences to summarize how the policy updates **each** of
765             the metadata it maintains immediately after inserting a new object into the cache here",
766         "update_after_evict": "use a few sentences to summarize how the policy updates **each** of
767             the metadata it maintains immediately after evicting the victim here"
768     }
769
770     Do not include any additional text or explanation in your response.

```

771 We use PlanSearch's original prompts (Wang et al., 2025) to generate observations. These observations  
 772 are then inserted into the placeholder [ [hints] ] of the following prompt for solution formulation:  
 773

#### The Ideation Prompt for PlanSearch

774 You are an expert in computer systems. Create an innovative cache replacement policy using  
 775 the following hints:  
 776 [ [hints] ]

777 Provide your creative policy using the following JSON structure:

```

778     "json
779     {
780         "metadata": "use a few sentences to summarize the metadata specifically maintained by the
781         policy here",
782         "evict": "use a few sentences to summarize how the policy chooses the eviction victim here",
783         "update_after_hit": "use a few sentences to summarize how the policy updates **each** of
784             the metadata it maintains immediately after a cache hit here",
785         "update_after_insert": "use a few sentences to summarize how the policy updates **each** of
786             the metadata it maintains immediately after inserting a new object into the cache here",
787         "update_after_evict": "use a few sentences to summarize how the policy updates **each** of
788             the metadata it maintains immediately after evicting the victim here"
789     }
790
791     Do not include any additional text or explanation in your response.

```

792 MetaMuse uses the following prompt for property extraction and problem mapping. They are the  
 793 first two waypoints (c.f. §3.3).

#### The Waypoint Reasoning Prompt for MetaMuse

794 You are an expert in computer systems. Your task is to use the concepts inspired by the given  
 795 word or phrase to design creative ideas for cache replacement policies.

796 Answer in the following format:

797 <The given word or phrase> relates to the concept of <concept 1>.

798 <Concept 1> relates to <Concept 2>.

799 ...

800 <Concept n-1> relates to <Concept n>

801 Inspired by <Concept n>, <your creative ideas for cache replacement policies in a few  
 802 sentences>.

803 \*\*Example 1\*\*:

804 The given word or phrase: angular momentum.

810  
 811 Answer:  
 812 “Angular momentum” relates to the concept of “rotation”.  
 813 “Rotation” relates to “cycle”.  
 814 Inspired by “cycle”, a cyclic pointer can be maintained to track cached objects and determine  
 815 eviction victims.  
 816 \*\*Example 2\*\*:  
 817 The given word or phrase: zebra.  
 818 Answer:  
 819 “Zebra” relates to the concept of “stripe”.  
 820 “Stripe” relates to “segmentation”.  
 821 Inspired by “segmentation”, a cache can be divided into segments with different eviction  
 822 priorities, and each segment can use a distinct eviction policy.  
 823 \*\*The given word or phrase\*\*: [[word]]  
 824 Do not include any additional text or explanation in your answer.

825 For solution formulation, MetaMuse uses the same ideation prompt as PlanSearch. The placeholder  
 826 [[hints]] will be replaced with the observations derived from the first two waypoints.  
 827

## 828 C.2 CODING PROMPTS

829 All methods use the following prompt to implement a solution in Python.  
 830 The placeholders, [[metadata]], [[evict]], [[update\_after\_hit]],  
 831 [[update\_after\_insert]], and [[update\_after\_evict]], will be filled in  
 832 with the corresponding fields in the JSON-formatted output of the ideation prompts. [[design]]  
 833 will be filled with a concatenation of these fields.  
 834

### 835 The Coding Prompt

836 You are an expert in Python. Your task is to implement a deterministic cache replacement  
 837 policy in Python without any randomness. You can only reference the attributes provided  
 838 below. You have read-only access to these attributes and no access to any functions.

839 [Begin of accessible attributes]

840 An “object” represents the unit of a request, such as inserting an object into the cache or  
 841 retrieving an object from the cache. Each object ‘obj’ provides the following \*\*read-only\*\*  
 842 attributes that you can reference:

843 - ‘obj.key’ (str): A string that uniquely identifies the object.

844 - ‘obj.size’ (int): A positive integer representing the size of the object in bytes.

845 You can also reference the following \*\*read-only\*\* attributes provided by a cache snapshots  
 846 ‘cache\_snapshot’:

847 - ‘cache\_snapshot.cache’ (dict): A dictionary containing the cached objects, where the keys  
 848 are the objects’ keys, and the values are the corresponding objects themselves.

849 - ‘cache\_snapshot.size’ (int): A non-negative integer representing the current total size of the  
 850 cache in bytes.

851 - ‘cache\_snapshot.capacity’ (int): A positive integer representing the maximum allowed size  
 852 of the cache in bytes.

853 - ‘cache\_snapshot.access\_count’ (int): The current total number of cache accesses. You can  
 854 also use this to represent current time.

855 - ‘cache\_snapshot.hit\_count’ (int): The current total number of cache hits.

856 - ‘cache\_snapshot.miss\_count’ (int): The current total number of cache misses.

857 [End of accessible attributes]

858 The cache replacement policy you need to implement is described below:

859 [Begin of cache replacement policy]

860 [[design]]

861 [End of cache replacement policy]

862 Implement this policy using the Python code framework below. Your implementation must  
 863 strictly follow the comments in this Python code framework.

```

864
865 [Begin of Python code framework]
866
867 # Import anything you need below. You must not use any randomness.
868 # For example, you cannot 'import random'. Also, you cannot use
869 # any function in 'numpy' that uses randomness, such as the
870 # functions in 'numpy.random'.
871
872 # Put tunable constant parameters below
873 # Put the metadata specifically maintained by the policy below. [[
874 # metadata]]
875
876 def evict(cache_snapshot, obj):
877     """
878     This function defines how the policy chooses the eviction victim
879     .
880     [[evict]]
881     - Args:
882         - 'cache_snapshot': A snapshot of the current cache state.
883         - 'obj': The new object that needs to be inserted into the
884             cache.
885     - Return:
886         - 'candid_obj_key': The key of the cached object that will
887             be evicted to make room for 'obj'.
888     """
889     candid_obj_key = None
890     # Your code below
891     return candid_obj_key
892
893 def update_after_hit(cache_snapshot, obj):
894     """
895     This function defines how the policy update the metadata it
896     maintains immediately after a cache hit.
897     [[update_after_hit]]
898     - Args:
899         - 'cache_snapshot': A snapshot of the current cache state.
900         - 'obj': The object accessed during the cache hit.
901     - Return: 'None'
902     """
903     # Your code below
904
905 def update_after_insert(cache_snapshot, obj):
906     """
907     This function defines how the policy updates the metadata it
908     maintains immediately after inserting a new object into the
909     cache.
910     [[update_after_insert]]
911     - Args:
912         - 'cache_snapshot': A snapshot of the current cache state.
913         - 'obj': The object that was just inserted into the cache.
914     - Return: 'None'
915     """
916     # Your code below
917
918 def update_after_evict(cache_snapshot, obj, evicted_obj):
919     """
920     This function defines how the policy updates the metadata it
921     maintains immediately after evicting the victim.
922     [[update_after_evict]]
923     - Args:
924         - 'cache_snapshot': A snapshot of the current cache state.
925         - 'obj': The object to be inserted into the cache.
926         - 'evicted_obj': The object that was just evicted from the
927             cache.
928     - Return: 'None'
929     """
930     # Your code below
931
932 [End of code framework]

```

You **must not** alter the provided code framework. Also, keep the two comments “# Put tunable constant parameters below” and “# Put the metadata specifically maintained by the policy below” unchanged. Wrap your code with “python and “ and include nothing else in your answer.

```

972         metadata['stochastic_model_outputs'].get(key, 0) +
973         metadata['data_entropy_values'].get(key, 0) +
974         metadata['neural_alignment_scores'].get(key, 0) +
975         metadata['access_frequency'].get(key, 0) +
976         metadata['recency'].get(key, 0) +
977         metadata['differential_privacy_noise_factors'].get(key, 0)
978     )
979
980     if score < min_score:
981         min_score = score
982         candid_obj_key = key
983
984     return candid_obj_key
985
986 def update_after_hit(cache_snapshot, obj):
987     """
988     This function defines how the policy updates the metadata it
989     maintains immediately after a cache hit.
990     After a cache hit, the policy increases the predictive likelihood
991     score, updates the stochastic model, recalculates data entropy,
992     adjusts the neural alignment score, updates access frequency and
993     recency, recalculates the differential privacy noise factor, and
994     updates the state-action values in the deep reinforcement
995     learning model.
996     - Args:
997         - 'cache_snapshot': A snapshot of the current cache state.
998         - 'obj': The object accessed during the cache hit.
999     - Return: 'None'
1000     """
1001     key = obj.key
1002     metadata['predictive_likelihood_scores'][key] = min(metadata['
1003         predictive_likelihood_scores'].get(key, 0) + 1,
1004         MAX_PREDICTIVE_SCORE)
1005     metadata['stochastic_model_outputs'][key] = min(metadata['
1006         stochastic_model_outputs'].get(key, 0) + 1, MAX_PREDICTIVE_SCORE)
1007     metadata['data_entropy_values'][key] = min(metadata['
1008         data_entropy_values'].get(key, 0) + 1, MAX_ENTROPY)
1009     metadata['neural_alignment_scores'][key] = min(metadata['
1010         neural_alignment_scores'].get(key, 0) + 1, MAX_NEURAL_ALIGNMENT)
1011     metadata['access_frequency'][key] = min(metadata['access_frequency'].get(key, 0) + 1, MAX_ACCESS_FREQUENCY)
1012     metadata['recency'][key] = cache_snapshot.access_count
1013     metadata['differential_privacy_noise_factors'][key] = min(metadata['
1014         differential_privacy_noise_factors'].get(key, 0) + 1,
1015         MAX_DIFFERENTIAL_PRIVACY_NOISE)
1016     metadata['deep_rl_state_action_values'][key] = min(metadata['
1017         deep_rl_state_action_values'].get(key, 0) + 1,
1018         MAX_PREDICTIVE_SCORE)
1019
1020 def update_after_insert(cache_snapshot, obj):
1021     """
1022     This function defines how the policy updates the metadata it
1023     maintains immediately after inserting a new object into the cache
1024     .
1025     After inserting a new object, the policy initializes the predictive
1026     likelihood score, updates the stochastic model, calculates
1027     initial data entropy, sets the neural alignment score,
1028     initializes access frequency and recency, assigns a differential
1029     privacy noise factor, generates quantum error correction codes,
1030     and updates the deep reinforcement learning model to include the
1031     new state.
1032     - Args:
1033         - 'cache_snapshot': A snapshot of the current cache state.
1034         - 'obj': The object that was just inserted into the cache.
1035     - Return: 'None'

```

```

1026     """
1027     key = obj.key
1028     metadata['predictive_likelihood_scores'][key] = 1
1029     metadata['stochastic_model_outputs'][key] = 1
1030     metadata['data_entropy_values'][key] = 1
1031     metadata['neural_alignment_scores'][key] = 1
1032     metadata['access_frequency'][key] = 1
1033     metadata['recency'][key] = cache_snapshot.access_count
1034     metadata['differential_privacy_noise_factors'][key] = 1
1035     metadata['quantum_error_correction_codes'][key] = 1
1036     metadata['deep_rl_state_action_values'][key] = 1
1037
1038     def update_after_evict(cache_snapshot, obj, evicted_obj):
1039         """
1040             This function defines how the policy updates the metadata it
1041             maintains immediately after evicting the victim.
1042             After evicting a victim, the policy removes its metadata, updates the
1043                 stochastic model, recalculates data entropy for remaining
1044                 entries, adjusts neural alignment scores, adjusts differential
1045                 privacy noise factors, updates quantum error correction codes,
1046                 and retrains the deep reinforcement learning model to adapt to
1047                 the new cache state.
1048             - Args:
1049                 - 'cache_snapshot': A snapshot of the current cache state.
1050                 - 'obj': The object to be inserted into the cache.
1051                 - 'evicted_obj': The object that was just evicted from the cache.
1052             - Return: 'None'
1053         """
1054
1055     evicted_key = evicted_obj.key
1056     if evicted_key in metadata['predictive_likelihood_scores']:
1057         del metadata['predictive_likelihood_scores'][evicted_key]
1058     if evicted_key in metadata['stochastic_model_outputs']:
1059         del metadata['stochastic_model_outputs'][evicted_key]
1060     if evicted_key in metadata['data_entropy_values']:
1061         del metadata['data_entropy_values'][evicted_key]
1062     if evicted_key in metadata['neural_alignment_scores']:
1063         del metadata['neural_alignment_scores'][evicted_key]
1064     if evicted_key in metadata['access_frequency']:
1065         del metadata['access_frequency'][evicted_key]
1066     if evicted_key in metadata['recency']:
1067         del metadata['recency'][evicted_key]
1068     if evicted_key in metadata['differential_privacy_noise_factors']:
1069         del metadata['differential_privacy_noise_factors'][evicted_key]
1070     if evicted_key in metadata['quantum_error_correction_codes']:
1071         del metadata['quantum_error_correction_codes'][evicted_key]
1072     if evicted_key in metadata['deep_rl_state_action_values']:
1073         del metadata['deep_rl_state_action_values'][evicted_key]
1074
1075     # Recalculate data entropy for remaining entries
1076     for key in cache_snapshot.cache:
1077         metadata['data_entropy_values'][key] = min(metadata['
1078             data_entropy_values'].get(key, 0) + 1, MAX_ENTROPY)
1079
1080     # Adjust neural alignment scores
1081     for key in cache_snapshot.cache:
1082         metadata['neural_alignment_scores'][key] = min(metadata['
1083             neural_alignment_scores'].get(key, 0) + 1,
1084             MAX_NEURAL_ALIGNMENT)
1085
1086     # Adjust differential privacy noise factors
1087     for key in cache_snapshot.cache:
1088         metadata['differential_privacy_noise_factors'][key] = min(

1089             metadata['differential_privacy_noise_factors'].get(key, 0) +
1090             1, MAX_DIFFERENTIAL_PRIVACY_NOISE)

```

```

1080     # Update quantum error correction codes
1081     for key in cache_snapshot.cache:
1082         metadata['quantum_error_correction_codes'][key] = min(metadata['
1083             quantum_error_correction_codes'].get(key, 0) + 1,
1084             MAX_PREDICTIVE_SCORE)
1085
1086     # Retrain the deep reinforcement learning model
1087     for key in cache_snapshot.cache:
1088         metadata['deep_rl_state_action_values'][key] = min(metadata['
1089             deep_rl_state_action_values'].get(key, 0) + 1,
1090             MAX_PREDICTIVE_SCORE)
1091
1092 D.2 METAMUSE-488
1093
1094 PARTITION_COUNT = 3 # Number of partitions based on usage patterns
1095 INITIAL_PRIORITY = 1
1096 INITIAL_RARITY = 1
1097
1098 metadata = {
1099     'partitions': [{} for _ in range(PARTITION_COUNT)], # List of
1100         # dictionaries for each partition
1101     'priority_scores': {}, # Dictionary mapping obj.key to priority
1102         # score
1103     'rarity_scores': {}, # Dictionary mapping obj.key to rarity score
1104     'timestamps': {}, # Dictionary mapping obj.key to access timestamps
1105 }
1106
1107 def evict(cache_snapshot, obj):
1108     candid_obj_key = None
1109     for partition in metadata['partitions']:
1110         # Find the least priority and rarity score item in the partition
1111         candidates = sorted(
1112             partition.items(),
1113             key=lambda item: (
1114                 metadata['priority_scores'][item[0]],
1115                 metadata['rarity_scores'][item[0]],
1116                 metadata['timestamps'][item[0]])
1117         )
1118         if candidates:
1119             candid_obj_key = candidates[0][0]
1120             break
1121
1122     return candid_obj_key
1123
1124 def update_after_hit(cache_snapshot, obj):
1125     metadata['timestamps'][obj.key] = cache_snapshot.access_count
1126     metadata['rarity_scores'][obj.key] += 1 # Increase rarity score
1127         # based on access frequency
1128     # Reevaluate priority within the partition
1129     # Here we assume a simple priority adjustment based on access
1130         # frequency
1131     metadata['priority_scores'][obj.key] = max(metadata['priority_scores'
1132             ][obj.key], metadata['rarity_scores'][obj.key])
1133
1134 def update_after_insert(cache_snapshot, obj):
1135     partition_index = hash(obj.key) % PARTITION_COUNT
1136     metadata['partitions'][partition_index][obj.key] = obj
1137     metadata['priority_scores'][obj.key] = INITIAL_PRIORITY
1138     metadata['rarity_scores'][obj.key] = INITIAL_RARITY
1139     metadata['timestamps'][obj.key] = cache_snapshot.access_count
1140
1141 def update_after_evict(cache_snapshot, obj, evicted_obj):
1142     partition_index = hash(evicted_obj.key) % PARTITION_COUNT

```

```

1134     if evicted_obj.key in metadata['partitions'][partition_index]:
1135         del metadata['partitions'][partition_index][evicted_obj.key]
1136         del metadata['priority_scores'][evicted_obj.key]
1137         del metadata['rarity_scores'][evicted_obj.key]
1138         del metadata['timestamps'][evicted_obj.key]
1139
1140     # Rebalance partitions based on current usage patterns
1141     for partition in metadata['partitions']:
1142         for key in partition:
1143             # Simplistic strategy to adjust priority and rarity scores
1144             metadata['priority_scores'][key] = max(INITIAL_PRIORITY,
1145                 metadata['priority_scores'][key] - 1)
1146             metadata['rarity_scores'][key] = max(INITIAL_RARITY, metadata
1147                 ['rarity_scores'][key] - 1)
1148
1149 D.3 COMPARE RSDICT AND RSDICT-NOWR USING AN EXAMPLE
1150
1151 We compare RSDict and RSDict-noWR with a concrete example on cache replacement to illustrate
1152 waypoint reasoning.
1153
1154 RSDict and RSDict-noWR both use the stimuli set {best, impose, extra, pale} to generate cache
1155 replacement policies.
1156
1157 D.3.1 PROPERTY EXTRACTION
1158
1159 RSDict reasons about the problem-related property of each stimulus:
1160
1161     • best → comparison → ranking
1162     • impose → authority → hierarchy
1163     • extra → surplus → overflow
1164     • pale → fading → decay
1165
1166 D.3.2 PROBLEM MAPPING
1167
1168 RSDict maps the extracted properties to the problem domain to generate observations:
1169
1170     • ranking → objects in the cache can be ranked based on access frequency or recency, with
1171         the lowest-ranked objects being evicted first
1172     • hierarchy → a hierarchical cache replacement policy can be developed where objects are
1173         ranked based on importance, and lower-ranked objects are evicted first
1174     • overflow → a cache replacement policy can focus on evicting the least valuable entries to
1175         make room for critical ones that exceed the cache's immediate capacity, essentially creating
1176         a priority-based overflow management system
1177     • decay → a cache replacement policy can use a decay function to gradually decrease the
1178         priority of cache entries, selecting the least 'bright' or least relevant items for eviction
1179
1180 D.3.3 SOLUTION FORMULATION
1181
1182 RSDict designs a solution using the observations:
1183
1184     • metadata: The policy maintains metadata including access frequency, recency of access,
1185         hierarchical importance, and a decay score for each cache entry. A dual ranking system
1186         assigns a composite score based on these factors.
1187     • update_after_hit: The policy calculates a composite score using access frequency, recency,
1188         importance rank, and decay values to identify the lowest scored entries for eviction, priori-
1189         tizing those near a threshold in the hierarchical cache structure.
1190     • update_after_insert: After a cache hit, the access frequency and recency values are incre-
1191         mented, the decay function is reset or slowed, and hierarchical importance may be updated
1192         to reflect improved relevance.
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3280
3281
3282
32
```

1188 • update\_after\_evict: Post eviction, the metadata recalibrates the hierarchical importance  
 1189 thresholds, updates decay dynamics, and modifies frequency/recency metrics for remaining  
 1190 entries to ensure balanced prioritization.

1191 RSDict-noWR directly uses the given stimuli to generate a solution:

1192 • metadata: The policy maintains metadata including a ‘Pale Access Frequency’ score for each  
 1193 entry, representing its likelihood of future accesses, and an ‘Extra Usage Counter’ to track  
 1194 rare intense usage patterns. Additionally, it includes ‘Best Recent Utilization’ timestamps,  
 1195 recording the time since the last access.

1196 • evict: The policy chooses the eviction victim by identifying the entry with the lowest ‘Pale  
 1197 Access Frequency’ score, the lowest ‘Extra Usage Counter’, and the oldest ‘Best Recent  
 1198 Utilization’ timestamp, focusing on entries least likely to be used in the near future.

1199 • update\_after\_hit: Upon a cache hit, the policy increases the ‘Pale Access Frequency’ score  
 1200 and ‘Extra Usage Counter’ for the accessed entry, boosting its likelihood to stay, and updates  
 1201 the ‘Best Recent Utilization’ timestamp to the current time.

1202 • update\_after\_insert: After inserting a new object, the policy initializes its ‘Pale Access  
 1203 Frequency’ score and ‘Extra Usage Counter’ to default low values, indicating an unbiased  
 1204 entry yet to prove its importance, and sets the ‘Best Recent Utilization’ timestamp to the  
 1205 current time.

1206 • update\_after\_evict: Upon eviction, the policy resets the metadata for the evicted entry,  
 1207 ensuring that new insertions don’t inherit old, irrelevant frequencies or utilization timestamps,  
 1208 and recalibrates other entries to maintain consistent scoring dynamics.

1209 This example shows that without waypoint reasoning, LLMs tend to use the given stimuli in a  
 1210 superficial way. In RSDict-noWR, LLMs uses the stimulus “pale” to name a metadata “Pale Access  
 1211 Frequency”, which is identical to a standard frequency counter. In RSDict, LLMs first derive the  
 1212 concept of “decay” from “pale”, then maps it to observation “decay function”, which is more difficult  
 1213 to come up with compared with frequency counting in cache replacement.

#### 1214 D.3.4 DIVERSITY EVALUATION

1215 After calculating the feedback embeddings of the two solutions, we find that RSDict-noWR’s solution  
 1216 is identical to LFU, while RSDict’s solution is different from any human heuristic.

### 1217 E IMPACTS OF USING TASK-RELATED STIMULI

1218 §3.2 mentions that stimuli can be unbiased to the problem, forcing LLMs to associate with knowledge  
 1219 that seems probabilistically irrelevant. And, one domain-agnostic instantiation of stimuli is keywords  
 1220 from an English dictionary. However, MetaMuse can also accommodate task-related stimuli. To this  
 1221 end, this section presents empirical results on the impacts of using task-related stimuli.

1222 Our experiments are based on designing cache replacement policies with GPT-4o. We start by  
 1223 prompting GPT-4o to output a list of cache-related stimuli, and GPT-4o generates 78 keywords:  
 1224 *lfru, consistency, ghost, clock, cache, optimal, associativity, s3-fifo, benchmark, bélády, policy, hit,*  
 1225 *temporal, memory, slru, queues, queue, arc, static, replacement, coherence, algorithms, access,*  
 1226 *plru, miss, brrip, probabilistic, caching, storage, sieve, prefetching, tree, lirs, analysis, timestamp,*  
 1227 *lifo, latency, policies, dynamic, recency, algorithm, reinsertion, performance, tlru, flash, clock-pro,*  
 1228 *ratio, discard, binary, fifo, expiration, mq, hawkeye, distance, srrip, pollution, ssds, locality,*  
 1229 *lfuda, eviction, reuse, drrip, mru, lfu, prediction, pointers, lru, inter-reference, rrip, approximation,*  
 1230 *distributed, throughput, data, aging, survival, metadata, streaming*. Then, we feed these stimuli  
 1231 to MetaMuse, and generate 350 cache replacement algorithm designs. Empirical results show that  
 1232 task-related stimuli reduce the number of distinct solutions by 11.

1233 Figure 8 compares top solutions, as selected by the average cache miss ratio over all 96 workload  
 1234 traces. Box plots show their miss ratio reduction, with respect to FIFO heuristics.

1235 At the 90<sup>th</sup>-percentile trace, MetaMuse using stimuli from an English dictionary achieves 3.24%  
 1236 lower miss ratio than MetaMuse using cache-related stimuli. At the 75<sup>th</sup>-percentile trace, MetaMuse

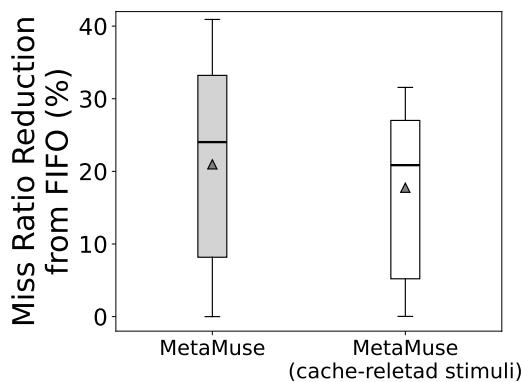


Figure 8: Comparisons of top cache solutions generated by *MetaMuse* using stimuli from an English dictionary and *MetaMuse* using cache-related stimuli. Each box plot represents the best solution from each model, and shows the miss ratio reduction (with respect to FIFO heuristics) achieved over 96 traces. Using stimuli from an English dictionary has higher reduction across nearly all percentiles.

using stimuli from an English dictionary achieves 6.99% lower miss ratio than *MetaMuse* using cache-related stimuli.

1242  
1243  
1244  
1245  
1246  
1247  
1248  
1249  
1250  
1251  
1252  
1253  
1254  
1255  
1256  
1257  
1258  
1259  
1260  
1261  
1262  
1263  
1264  
1265  
1266  
1267  
1268  
1269  
1270  
1271  
1272  
1273  
1274  
1275  
1276  
1277  
1278  
1279  
1280  
1281  
1282  
1283  
1284  
1285  
1286  
1287  
1288  
1289  
1290  
1291  
1292  
1293  
1294  
1295