LLM-Enhanced Memory Prefetching: Evaluating Gemini's Capability for Program Counter and Delta Prediction

Jayden Lin
jaydenl@andrew.cmu.edu

ABSTRACT

Memory prefetching remains a bottleneck in modern computer systems, where traditional hardware prefetchers struggle with irregular access patterns characteristic of increasingly complex applications. While sequential and stride-based prefetchers effectively handle simpler memory patterns, they fail to predict certain nonlinear memory traversals, increasing latency from one to four cycles for a cache hit to tens to hundreds of cycles [5], which translates to an exorbitant speed reduction. A prefetcher aims to load data intended for future use before it is requested, so the problem revolves around predicted where the useful data lie.

This paper incorporates Gemini, a large language model-based prefetching system that leverages transformer architectures to predict memory access patterns from program execution traces. Unlike conventional approaches that rely on fixed algorithmic patterns, Gemini attempts to learn relationships between program counter sequences and memory address deltas by training on historical execution data. This system attempts to capture dependencies and context-sensitive behaviors that traditional prefetchers do not have access to, in hopes of accurate prediction of memory address accesses.

Our findings indicate that large language models have the capability to prefetch for certain types of program executions, and may have the capacity to be trained for more nuanced ones. However, they are far too slow and inconsistent in their current states to be viable. With further developments, there may avenues that adapt to evolving software complexity.

1 INTRODUCTION

Recent advances in large language models (LLMs) have demonstrated remarkable pattern recognition capabilities across diverse domains[1]. This paper explores the application of Google's Gemini model to the domain of memory prefetching, specifically investigating its ability to predict future memory access deltas based on program counter traces and historical access patterns.

Through systematic evaluation using program traces, we demonstrate that LLM-based prefetching can achieve impressive results for specific applications.

2 BACKGROUND AND MOTIVATION

2.1 The Memory Wall Problem

Moore's law, though not perfectly accurate, states that the number of transistors in a chip doubles every two years. Processor performance has kept with this rapid growth, but memory access latency remains stagnant, fostering an ever-increasing disparity. Processors executing billions of operations per second must wait hundreds of cycles for data from main memory to proceed.

Memory prefetching offers a solution by loading data into faster cache levels before explicit processor requests. Successful prefetching transforms costly memory stalls into productive computation time, eliminating higher cache accesses that can take upwards of 20 times longer to complete [5].

2.2 Limitations of Current Prefetching

Traditional hardware prefetchers detect patterns in simple access sequences but struggle with modern application complexity. Traditional approaches include [3]:

Sequential prefetchers: Detect consecutive accesses and fetch subsequent cache lines

Stride prefetchers: Identify regular intervals and predict addresses using constant offsets

Stream buffers: Record values from multiple concurrent access streams

TAGE prefetchers: Maintain limited historical context using tagged geometric history

These pattern-based methods perform worse on irregular access patterns increasingly common in modern software [2]. Applications featuring pointer-chasing, sparse matrices, tree traversals, and hash lookups exhibit complex, context-dependent behaviors that defy simple prediction.

2.3 Machine Learning's Advantages

The sequential nature of program execution traces aligns naturally with transformer architectures designed for sequential modeling. Key advantages include:

- Capturing long-range dependencies in memory access patterns
- Learning non-linear relationships between program context and memory behavior
- Adapting through fine-tuning for specific applications or domains

LLM-based prefetching could unlock significant performance gains for irregular workloads, while future work on fine tuning could allow programs to have individualized models, rather than a one-size-fits-all prefetching system. This approach provides program designers with significantly improved flexibility in data locality, highlighting machine learning's primary advantage in system design.

2.4 Google Gemini's API

Gemini API is free for the public use, which is why we have selected it for the trials in this paper. Other LLMs such as Claude, ChatGPT, DeepSeek, etc. either do not have an public API or are not applicable for research purposes.

3 METHODOLOGY AND EXPERIMENTAL DESIGN

This study was conducted using seven pairs of benchmark traces, each simulating a distinct program execution. Each trace consists of program counters (PC) and the associated memory address deltas (Δ), which are the differences between consecutive memory accesses. We systematically gather a number of consecutive PC-Δ pairs as a batch, then format it to fit into a question template that the Gemini API can respond to. In the request, Gemini is tasked to predict the subsequent Δ values without providing preamble or postamble. This response is defensively parsed to account for rare instances of unexpected tokens. These parsed Δ values are converted into a set of relative addresses where we can analyze them against the ground truth set.

There are many hyperparameters that can be varied during this process, which enables three types of experiments to determine optimal conditions for LLM prefetching:

(1) **Hyperparameter sweeps:** Given a specific dataset and prompt style, systematically test a matrix of input batch size and prediction lookahead size. For example, batch size can vary from 100–200 lines, while lookahead size can range from 30–80 lines.
(2) **Prompt comparison:** Given a specific parameter setting and dataset, make API calls with different prompts, ranging from minimal context to full explanations of the required task.
(3) **Dataset comparison:** Given a specific parameter setting and prompt style, test multiple datasets to determine which programs Gemini performs better on.

We calculate four metrics: accuracy, precision, recall, and prediction duration. Each address prediction is treated as a binary true/false value through set intersection with the subsequent ground truth Δ. Each configuration of hyperparameters undergoes 10 batch predictions to balance statistical reliability with computational feasibility. The metrics are then averaged and visualized through heatmaps and bar charts to facilitate comparison between parameter combinations.

3.1 Prediction Pipeline

Figure 1 illustrates our LLM-enhanced prefetching pipeline. The system operates in two main phases: (1) training data preparation and prompt engineering, and (2) inference and prediction evaluation.

3.2 Dataset Characteristics

Our evaluation utilizes 7 distinct datasets derived from program execution traces. In the 50-Cache versions, the cache allocated for storing memory is double the size of the 25-Cache Versions, which causes slight differences in trace sizes as seen in **Table 1**.

Each line contains a program counter and a memory access delta, representing the difference between consecutive memory address accesses.

3.3 Experimental Configuration

Table 2 presents our main adjustable hyperparameters.

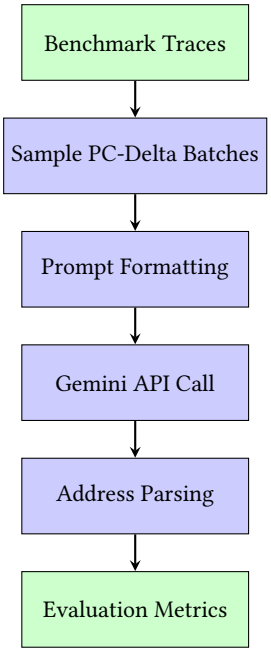

Figure 1: LLM Memory Prefetching System Pipeline

Table 1: Dataset Characteristics

Dataset	25-Cache Lines	50-Cache Lines	Program Use
cactu	6.6 M	2.2 M	??
lrange	121 M	127 M	??
mcf	13.1 M	1.3 M	??
numpy	17.6 M	884 K	??
omnetpp	154 M	62.4 M	??
pr	393 M	388 M	??
ycsb-c	42.3 M	22.7 M	??

3.4 Prompt Engineering Strategies

We developed four distinct prompt engineering approaches:

(1) **Original**: The prompt explains that PC and delta pairs are given sequentially.
(2) **Minimal**: No context is given, just values. The prompt solely states to predict the next values in the sequence, with no differentiation between PC and delta.
(3) **Contextual**: The prompt opens with describing the context: "You are analyzing memory access patterns from a computer program" and also explains the meaning of PC and deltas.
(4) **Expert**: The prompt gives Gemini a role and emphasizes the importance of accuracy: "You are an expert specializing in memory prefetching optimization. Your predictions are critical for system performance". Context is given as well.

Table 2: Experimental Test Configuration Hyperparameters

Dataset	Input Batch Size	Prediction Size	Cache Size	Gemini Version	Prompt Style	Tuned
cactu	100	10	25	2.0	Original	Yes
lrange	125	20	50	2.5	Minimal	No
mcf	150	30			Contextual	
numpy	175	40			Expert	
omnetpp	200	50				
pr		80				
ycsb-c		70				

Algorithm 1 Original Prompt Algorithm

1: **Input:** Historical PC-Delta pairs $(pc_1, \Delta_1), ..., (pc_n, \Delta_n)$
2: **Template:**
3: "Here are the most recent {batch size} program counter, delta pairs in sequential order (PC in hex, delta in decimal):"
4: "[PC-delta sequence]"
5: "Predict the next {prediction size} delta values with no other text or confirmations."
6: **Output:** Predicted deltas $\hat{\delta}_{n+1}, ..., \hat{\delta}_{n+N}$

4 RESULTS AND DISCUSSION

4.1 Model Version Comparison

Figure 2 compares the performance of Gemini 2.0 and 2.5 across metrics on the cactu dataset. Both models demonstrate similar performance in terms of accuracy, with hit rates of 89.7% and 89.6% respectively, precision scores of 90.6% and 91.2%, and recall values of 90.0% and 90.2%. The marginal differences in these core performance metrics are negligible.

However, a significant disparity emerges in inference time. Gemini 2.5 requires 25.2 seconds to complete 10 API calls, while Gemini 2.0 accomplishes the same task in only 3.7 seconds. Given that the performance improvements of Gemini 2.5 are minimal and do not justify the substantial increase in inference time, we selected Gemini 2.0 for all subsequent experiments.

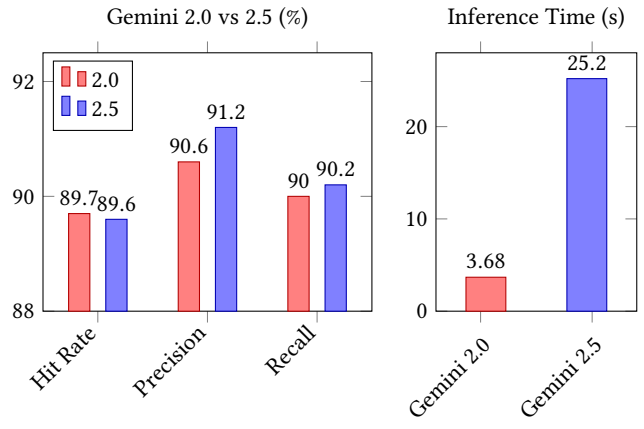

Figure 2: Gemini models 2.0 and 2.5 performance comparison

4.2 Prompt Configuration Analysis

Table 3 compares the prompt strategies, and reveals slight performance differences between the original, contextual, and expert prompt settings across all four metrics, notably, the minimal prompt has a higher accuracy by 6-12%, while also requiring less than half the time than all of the other prompt strategies.

This finding suggests that reducing prompt complexity allows the model to focus primarily on pattern detection without being distracted by extensive contextual information or domain-specific instructions. The simplified approach produces significant improvements in processing speed, indicating that overly complex prompts may introduce unnecessary cognitive overhead for the language model, and that the context provided is likely redundant and does not benefit in these small batch cases. These findings are consistent with other datasets as well.

Table 3: Prompt Strategy Comparison (lrange_50)

Prompt	Accuracy (%)	Precision (%)	Recall (%)	Time (s)
Original	49.7	63.1	66.5	4.71
Minimal	56.2	68.0	71.7	1.82
Contextual	44.5	60.4	59.0	4.72
Expert	51.6	64.9	67.5	4.62

4.3 Dataset Performance Analysis

Our evaluation across multiple program traces reveals that Gemini's performance varies dramatically depending on the specific workload characteristics. While the model performs well on many traces, it shows poor results on others, highlighting the importance of program behavior patterns for LLM-based prefetching.

Comparing cache sizes, programs generally show similar performance between 25-line and 50-line context windows, with one notable exception: the MCF trace. For MCF, the 25-line configuration achieves less than 0.5% accuracy, while the 50-line version reaches 97.4% accuracy.

This performance variability suggests that certain program types are well-suited for LLM-based prefetching, likely those with sequential patterns that fit within 100-200 line scopes and can be effectively captured in one-shot prompts. Conversely, programs with irregular or highly complex memory access patterns may not benefit from this approach. The cache size analysis further demonstrates that the granularity of context provided to the model significantly influences

Table 4: Dataset Average Performance Summary

Dataset	Accuracy (%)	Precision (%)	Recall (%)	Time (s)
cactu_25	78.7	83.8	83.2	3.57
cactu_50	95.3	97.2	97.2	4.01
lrange_25	41.6	59.1	57.8	4.14
lrange_50	46.3	61.4	62.4	4.16
mcf_25	0.4	0.9	0.8	5.01
mcf_50	97.4	100.0	97.4	3.55
numpy_25	89.4	100.0	89.4	4.00
numpy_50	99.4	100.0	99.4	3.81
omnetpp_25	20.6	37.9	27.8	4.19
omnetpp_50	11.3	19.7	19.2	3.87
pr_25	98.0	90.0	98.0	3.51
pr_50	100.0	100.0	100.0	3.64
ycsb_25	2.7	4.7	4.6	5.33
ycsb_50	1.6	3.1	3.0	5.34

the quality of generated predictions, with some programs requiring larger context windows to reveal their underlying patterns.

4.4 Prediction Path Visualization

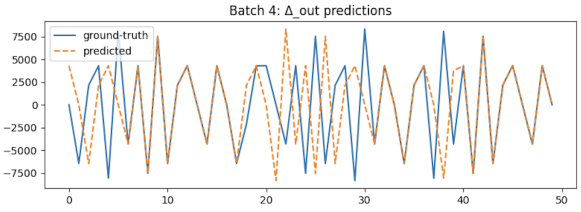

Figure 3: Prediction and Ground Truth

The prediction path visualization demonstrates the model's ability to align with and self-correct against ground truth patterns, even under complex and highly varying execution conditions. This adaptive behavior suggests that LLMs can capture nuanced program behaviors that might challenge traditional prefetching algorithms, though this capability is highly dependent on the specific program characteristics and context provided.

4.5 Hyperparameter Sweep Results

Figures 4 and 5 show the accuracy heatmap plotting batch and output size, where lighter colors indicate better model performance. Performance degrades significantly at batch size 200, which suggests a soft batch size ceiling beyond which the transformer struggles to maintain relationships within the data, likely due to memory limitations or attention constraints. Optimal performance occurs at moderate output sizes of 30-40 addresses, with the three best-performing configurations being (100,40), (125,30), and (175,30), achieving accuracies of 1.000, 1.000, and 0.987 respectively. Batch sizes below 200 demonstrate consistently strong performance, indicating robust scalability within this range. The timing analysis

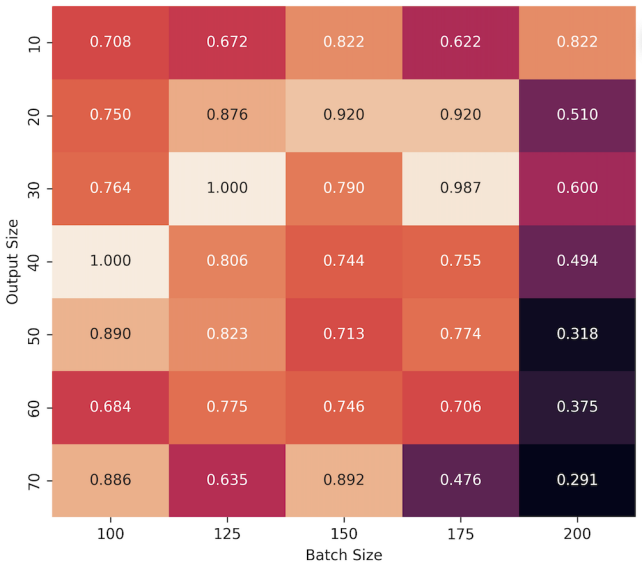

Figure 4: Caption

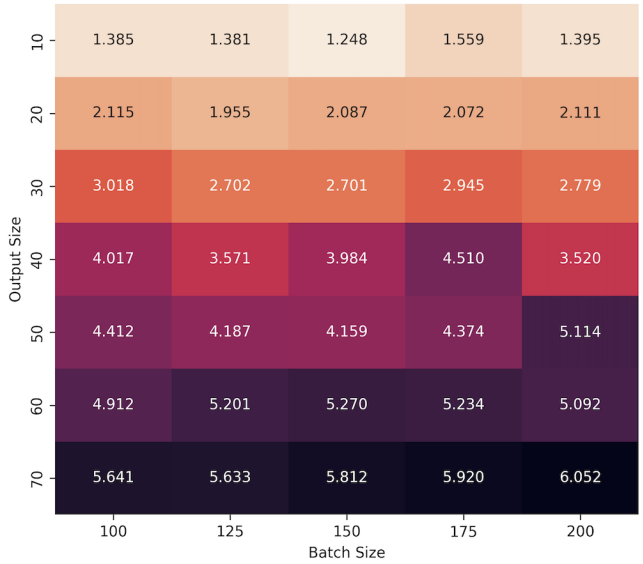

Figure 5: Caption

reveals that completion time is governed by output size rather than batch size. Average completion times increase approximately linearly with output size (from 1.4s at size 10 to 5.9s at size 70), while showing minimal variation across different batch sizes for equivalent output sizes. This pattern suggests that the bottleneck lies in the generation process rather than input processing, where each additional output token requires similar processing time regardless of batch size, consistent with autoregressive transformer behavior during inference. [4]

4.6 Limitations and Challenges

Fine tuning: While fine tuning was intended to be an important portion to this project, the tuning process was too costly (about $30 per trained model) and the results performed poorer than the one-shot prompts, suggesting that some part of the methodology is not properly implemented.

Inference Latency: Current inference times (130ms at the fastest) are too high for real-time prefetching applications when compared to normal . Significant optimization would be required for practical deployment.

Parameter-Dependent Reliability: Response reliability is highly sensitive to configuration parameters, with success rates varying dramatically from 0.05-100% depending on context window size, prompt complexity, and dataset characteristics. This parameter sensitivity makes robust deployment challenging and requires extensive tuning for each application scenario.

Economic Feasibility: After processing approximately 2000 batches, API costs reached $0.50, which may seem modest but becomes prohibitive when scaled to continuous prefetching requirements. For production systems requiring millions of predictions daily, current pricing models make LLM-based prefetching economically unfeasible.

5 FUTURE WORK

5.1 Model Optimization and Fine-tuning

Investigating fine-tuning approaches designed for memory access prediction could potentially improve both speed and accuracy. Future work should explore whether domain-specific training can reduce the model size requirements while maintaining prediction quality, addressing both latency and cost concerns.

5.2 Advanced Prompt Engineering

Our results demonstrate that prompt strategy significantly impacts both performance and inference time. Future research should systematically explore more sophisticated prompt engineering techniques, including few-shot learning approaches, chain-of-thought prompting, and adaptive prompt selection based on program characteristics. The minimal prompt performed best in these small batch scenarios, but this conclusion is not generalizable to every scenario.

5.3 Model Diversity and Scaling

Comprehensive evaluation across different Gemini versions and alternative LLMs (Claude, GPT-4, specialized smaller models) could reveal optimal model architectures for prefetching tasks. Additionally, investigating how input-output dimensions scale with different model capabilities could improve deployment strategies.

6 CONCLUSION

This paper presents an evaluation of large language models for memory prefetching, demonstrating both the remarkable potential and significant limitations of applying LLMs to low-level system optimization tasks. There is certainly more work to be done in this area, and it is likely that machine learning integrates into mainstream prefetching systems in the near future.

Alternative Value Proposition: Despite deployment challenges, LLMs could serve a valuable role in prefetching research and development. Their ability to identify complex, non-obvious patterns in memory access traces could be useful in the design of more efficient traditional prefetchers. By using LLMs as pattern discovery tools rather than production prefetchers, researchers could leverage their pattern recognition capabilities to develop insights that guide the creation of more specialized prefetching algorithms.

ACKNOWLEDGMENTS

I thank Systopia for providing the datasets that made this research possible. I am grateful to Shaurya Patel and Sasha Federova for providing the opportunity to work on this project and for their guidance throughout the research process. I also acknowledge the computational resources provided by institution's high-performance computing facility and Google's Gemini API.

REFERENCES

[1] Jesse Harte, Wouter Zorgdrager, Panos Louridas, Asterios Katsifodimos, Dietmar Jannach, and Marios Fragkoulis. 2023. Leveraging large language models for sequential recommendation. In *Proceedings of the 17th ACM Conference on Recommender Systems*. 1096–1102.

[2] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When prefetching works, when it doesn't, and why. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 1 (2012), 1–29.

[3] Sparsh Mittal. 2016. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)* 49, 2 (2016), 1–35.

[4] Ziyad Sheebaelhamd, Michael Tschannen, Michael Muehlebach, and Claire Vernade. 2025. Quantization-free autoregressive action transformer. *arXiv preprint arXiv:2503.14259* (2025).

[5] Steven JE Wilton and Norman P Jouppi. 2002. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of solid-state circuits* 31, 5 (2002), 677–688.