Learned Prefetching with Delayed Utility

Anonymous authors
Paper under double-blind review

Abstract

While learned sequence models have proven highly effective at capturing complex memory access patterns, they are conventionally framed as short-horizon predictors and evaluated under idealized, instant-fill assumptions. The rapid shift toward disaggregated memory topologies—where memory is pooled across high-latency fabrics—fundamentally breaks this paradigm. In these environments, higher and variable latencies mean that timeliness dictates system performance: a highly accurate prediction that arrives too late fails to hide latency, while an early prediction wastes limited bandwidth and pollutes the cache. To address these disaggregated constraints, we argue that prefetching must be recast from next-step prediction to a delayed-utility learning problem. We propose a multi-horizon sequence model that predicts across several future lead times, exposing a richer set of candidate accesses tailored to the deployment's specific latency regime. Evaluated under a rigorous, stall-aware framework using real-world memory access traces, we demonstrate that standard single-horizon models frequently fall short in high-latency environments, whereas our multi-horizon approach successfully captures the deployment-dependent utility of different lead times at a modest inference cost.

1 Introduction

Cache prefetching aims to reduce memory stalls by bringing data into the cache before it is requested. In modern memory systems, however, the usefulness of a prefetch depends on more than whether the predicted access is eventually correct. A prefetch issued too late provides little latency hiding, while a prefetch issued too early can evict useful data, consume limited bandwidth, and increase cache pollution. This *timeliness* problem becomes even more important in emerging memory systems such as CXL-backed memory, where memory can be pooled or attached across devices (Wang et al., 2026; Berger et al., 2026) and accessed through a higher-latency fabric rather than only through local DRAM channels (Wang et al., 2025; Li et al., 2026). In these settings, misses are more expensive and latency variation is larger, making both the timing and the selectivity of prefetching correspondingly more important.

Recent work has shown that learned models can capture memory-access patterns that are difficult to encode using hand-designed heuristics, including multilayer perceptrons, sequence models, and graph-based predictors (Hashemi et al., 2018; Asgari et al., 2021; Yang et al., 2023). At the same time, classical prefetching and caching theory has long emphasized the importance of lookahead, timeliness, and the interaction between prediction and cache state (Albers, 1997; Cao et al., 1995). Despite this, most learned prefetching pipelines are still evaluated primarily through short-horizon prediction or simplified cache-hit measurements that do not fully account for prefetch latency, in-flight lines, or the cadence at which prefetch decisions are issued. As a result, there is a substantial gap between what learned prefetchers are trained to predict and what actually determines their utility at deployment time.

This mismatch is especially acute for horizon selection. Predicting the next access or the nearest future accesses may yield high accuracy, but such predictions can still be too late to hide memory latency. Predicting further into the future provides more lead time, but these targets are harder to model and can increase harmful early fetches. As we show empirically, no single fixed horizon reliably captures the useful predictive signal across a trace: the

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

information a prefetcher needs to act in time is distributed across several lead times at once, not concentrated in whichever one turns out best. In other words, the question in learned prefetching is not only *what* to predict, but *how many lead times to draw candidates from* and *under what timing model* those predictions will be consumed.

In this paper, we argue that learned prefetchers should be studied as *delayed-utility predictors*, not merely as next-step predictors. We introduce a more realistic evaluation framework for learned prefetching that explicitly models delayed fills, the temporary unavailability of in-flight prefetched lines, and stall-aware runtime cost under different prefetch cadences. Using this framework, we show that short-horizon predictive quality is not a reliable proxy for downstream cache performance: once timeliness is modeled explicitly, model rankings can change substantially, and in some settings even reverse.

Motivated by this observation, we adapt existing learned model of (Hashemi et al., 2018) from single-horizon prediction to a multi-horizon setting, borrowing from how multi-horizon forecasting has been formulated for recurrent and transformer-based sequence models (Wen et al., 2018; Lim et al., 2020). Our design uses a shared-trunk multihead architecture that predicts across several future lead times, allowing the prefetcher to expose a richer set of candidate accesses than a single-step model at modest additional inference cost. The purpose of this adaptation is not simply to add more predictions, but to provide access to multiple lead times whose utility depends on the deployment regime. Empirically, we find that next-step or short-horizon accuracy alone does not determine cache benefit, and that multi-horizon predictors are substantially more informative under realistic timing constraints.

Our contributions are as follows:

- We present a realistic evaluation framework for learned prefetching that incorporates delayed fills, in-flight prefetch unavailability, and stall-aware runtime cost under varying decision cadences.
- We show that useful prefetch candidates are distributed across multiple lead times rather than concentrated in a single best horizon: individual lead times perform nearly identically in isolation, and the benefit comes from aggregating across many of them at once—an effect standard simplified evaluation can miss entirely, since it collapses immediately under delayed utility.
- We adapt existing learned prefetching models to a multi-horizon setting through a shared-trunk multihead design, and demonstrate that multi-horizon prediction provides a stronger basis for prefetch decisions than single-horizon prediction under realistic deployment conditions.

2 Preliminaries

Before describing our approach, we introduce prefetching more concretely for readers less familiar with the area. A processor's memory accesses form a sequence over time: at each step t, some instruction (identified by its program counter, pc_t) touches a memory address a_t. A *prefetcher* tries to guess which address will be needed soon and bring it into the cache *before* it is actually requested, so that when the real access happens it is already available and the processor does not stall waiting on main memory.

Directly predicting raw addresses is difficult, since the space of possible addresses is enormous and largely arbitrary. In practice, prefetchers instead work with the *delta* (Chen & Baer, 1995; Joseph & Grunwald, 1997; Nesbit & Smith, 2004) between consecutive addresses, $\delta_t = a_t - a_{t-1}$ (formalized further in Section 3.2), because access patterns are often far more regular in delta space than in raw address space—for example, a program iterating over an array touches evenly spaced addresses.

A concrete example. Suppose a program has just touched addresses 100, 104, and 108 in sequence—a fixed delta of +4 each step, typical of an array traversal. A predictor that has picked up on this pattern can extrapolate the next address, $\hat{a} = 112$, and issue a prefetch for it *before* the processor actually asks for it.

Why correctness alone is not enough. A correct prediction is not, by itself, useful; *when* it is issued matters just as much as whether it is right. Figure 1 shows the same correct prediction issued too early, at the right time, and too late, with three different outcomes despite identical accuracy. This is the *timeliness* property that motivates the rest of this paper: a prefetcher's value depends not only on whether it predicts correctly, but on whether its predictions arrive close enough to when they are needed—relative to how often prefetch decisions are made and how long a prefetch takes to land once issued. Section 3.1 formalizes this into the delayed-utility framing at the core of our evaluation.

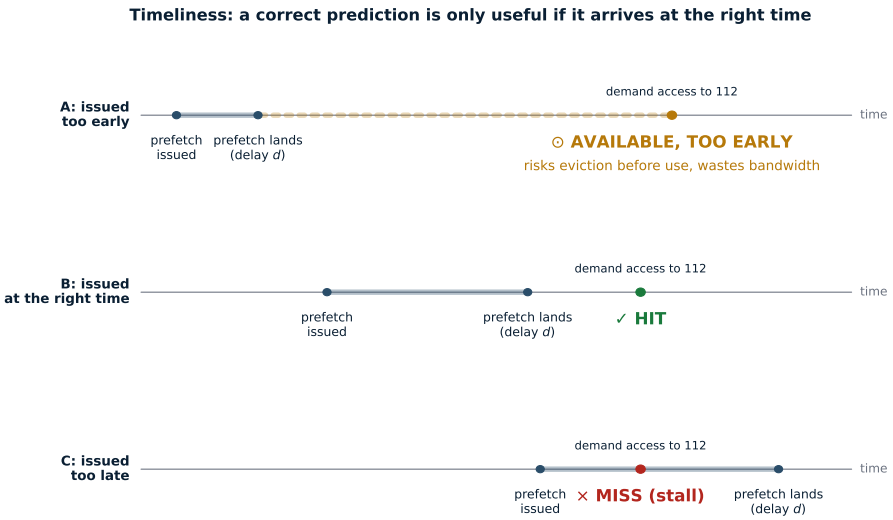

Figure 1: **The same correct prediction, three different outcomes.** The processor will next demand address 112; a prefetcher has correctly predicted this and issues a prefetch for it, but at a different time in each scenario. **(A)** Issued too early: the line lands long before it is needed and sits idle, risking eviction and wasting bandwidth and cache capacity. **(B)** Issued at the right time: the line lands just before the demand access, a hit. **(C)** Issued too late: the line has not yet landed when the demand access occurs, a miss despite the correct prediction.

We refer to how many accesses ahead a prediction looks as its *lead time* (e.g. LSTM-Step50 predicts 50 accesses ahead); our multi-horizon models predict across several lead times at once rather than committing to one. They do this with a *shared trunk*: a single LSTM processes the access history once, and each lead time is read off that same hidden state through its own small linear head, so adding another lead time costs one more head, not another LSTM.

3 Prefetching Models

3.1 Problem Definition

Before describing the model, we state the decision problem it is deployed to solve. This is deliberately not the same object as the supervised prediction task the model is trained on; we return to that distinction at the end of this subsection.

Trace and observed history. The system observes a sequence of memory accesses $(a_1, \mathrm{pc}_1), (a_2, \mathrm{pc}_2), \ldots$, where $a_t \in \mathbb{Z}$ is the byte address touched at step t and pc_t is the issuing instruction pointer (Section 5, Address and delta granularity). At step t a policy may condition only on the causal history observed so far; in practice every learned policy conditions on a bounded window of the L most recent accesses, $H_t = \big((a_{t-L+1}, \mathrm{pc}_{t-L+1}), \ldots, (a_t, \mathrm{pc}_t)\big)$.

Decision timing. Prefetch decisions are made only at a fixed cadence τ, measured in accesses (τ counts steps of the access index t, not the cycle count c_t introduced below) – this is not an idealized modeling assumption but exactly what the evaluation harness does (Section 5, Equal-budget protocol). Decisions occur at times $\mathcal{T} = \{\tau, 2\tau, 3\tau, \ldots\}$; between decision points no new prefetch is issued, though every policy's internal state (context window, per-PC tables) still advances on every access. At each $t \in \mathcal{T}$ the policy π maps H_t to a candidate set $K_t = \pi(H_t) \subseteq \mathbb{Z}$ with $|K_t| \leq k$.

Two clocks: accesses and cycles. The access index t counts *accesses*: it advances by exactly 1 every time the trace touches an address, regardless of whether that access hits or stalls. Delay, however, is a statement about elapsed *time*, not elapsed accesses, so we track a second quantity, the cycle count c_t: the cumulative simulated runtime through the t-th access, $c_t = c_{t-1} + \text{cost}_t(\pi)$ with $c_0 = 0$, where $\text{cost}_t(\pi)$ is defined below (Objective). Unlike t, c_t does *not* advance by a fixed amount per access: it advances by 1 cycle on a hit but by more than 1 whenever an access stalls, since a stall is exactly the extra cycles the objective is charging for. The two clocks coincide only when every access is a hit; in general $c_t \geq t$.

Delay. A candidate issued at access $t \in \mathcal{T}$, when the cycle count stands at c_t, does not land immediately: it occupies a cache slot right away but only becomes hittable once the cycle count reaches $c_t + d$ for a fixed delay d. Because c_t advances non-uniformly (previous paragraph), the delay d is d *cycles* of simulated latency, not d *accesses*: since each stalling access already consumes more than one cycle, the same candidate spends fewer accesses in flight during a heavily-stalling stretch of the trace and more during a heavily-hitting one, where each access advances c_t by only one cycle. An access to that address before its cycle-count target is reached still misses (Section 4). The delayed-availability hit-rate metric (Section 4) uses a simpler, uniform variant of this same rule instead: landing is measured directly against the access index ($t + d$ accesses) rather than against c_t, deliberately decoupling whether a policy's predictions are timely from the cost accounting itself. You can only gain a "hit" if the prefetch is completed before the access is requested.

Cache capacity. The cache holds a set of resident addresses $S_t \subseteq \mathbb{Z}$, $|S_t| \leq C$, updated by both demand accesses and landed prefetches, with the oldest entry evicted under LRU whenever a new address would exceed capacity C (Section 5, Cache model).

Objective. Let $\text{cost}_t(\pi)$ be 1 cycle if $a_t \in S_t$ (a hit), $1 + d$ cycles if a_t is neither resident nor pending (a clean miss), or $1 + r_t$ cycles if a_t is pending with r_t cycles remaining until it lands (Section 4) – r_t is measured against the cycle count c_t defined above, not against the access index, so it can exceed the number of accesses remaining before landing. The prefetching decision problem is to choose π minimizing total runtime over a trace of length T:

$$J(\pi) = \sum_{t=1}^{T} \text{cost}_t(\pi) = c_T. \tag{1}$$

This closes the loop with the Delay paragraph above: c_t is simultaneously the clock that determines when a pending prefetch lands and the running total of the objective itself, which is what makes $J(\pi)$ a genuine *stall-aware runtime* rather than a count of accesses.

Why this is not the supervised task. Framed this way, prefetching is an online sequential decision problem: a bounded action from a bounded observation, whose consequences are felt d cycles later and mediated by a capacity-C cache carrying state across every earlier decision. This is a different object from the *supervised* objective each model is actually trained on (Section 3.2): a per-step cross-entropy loss against a single true future delta, evaluated independently at every t with no reference to τ, d, C, or the cost incurred anywhere else in the trace. Training a model that decides which pages to prefetch based on these would still heavily depend on actually predicting future accesses. A model that minimizes the supervised loss is not thereby guaranteed to minimize $J(\pi)$ – the gap between these two objectives is exactly what Sections 4 and 5 measure empirically.

3.2 Model Architecture

We build on Long Short-Term Memory (LSTM) networks (Hochreiter & Schmidhuber, 1997), following Hashemi et al. (2018)'s use of LSTM embeddings for memory-access prediction. An LSTM processes a sequence with a gated cell state updated additively rather than multiplicatively, which lets it retain information over long context windows without the vanishing-gradient problem that limits standard RNNs; we use the standard formulation throughout and do not modify the cell itself. Given an input x_t at each step t, the LSTM produces a hidden state h_t; we describe next what x_t is in our setting and how the final hidden state h_L over a context window of length L is used.

From accesses to inputs. In our setting the input at step t is the memory access itself: the issuing instruction pointer pc_t and the delta $\delta_t = a_t - a_{t-1}$ between consecutive addresses. Both are embedded and combined into a single input vector $x_t = E_{\mathrm{pc}}(\mathrm{pc}_t) \oplus E_\delta(\delta_t)$, where E_{pc} and E_δ are learned embedding tables and $\oplus$ denotes concatenation. The LSTM consumes a context window of the L most recent accesses and produces a final hidden state h_L. For a single-horizon model predicting the cumulative delta N steps ahead, h_L is projected through a linear layer and a softmax over a fixed vocabulary of the V most frequent cumulative deltas observed in training, with a catch-all out-of-vocabulary class for everything else:

$$p(\Delta_N = \delta_j \mid h_L) = \mathrm{softmax}(W_{\mathrm{out}} h_L + b_{\mathrm{out}})_j, \qquad \delta_j \in \mathcal{V}. \tag{2}$$

LSTM-Step50 is exactly this model with $N = 50$.

Multi-horizon extension. The shared-trunk multi-horizon model reuses the same LSTM trunk and final hidden state h_L, but attaches one independent output head per horizon n in the target span $\{n_{\min}, \ldots, n_{\max}\}$ (e.g. 50–70 for Multihead 50–70):

$$p_n(\Delta_n = \delta_j \mid h_L) = \mathrm{softmax}(W_n h_L + b_n)_j, \qquad n \in \{n_{\min}, \ldots, n_{\max}\}, \tag{3}$$

trained jointly by summing the cross-entropy loss of every head, $\mathcal{L} = \sum_n \mathrm{CE}\big(p_n(\cdot \mid h_L), y_n\big)$, where y_n is the true cumulative delta n steps ahead. Sharing the trunk means the marginal cost of an extra horizon is one more linear head, not another LSTM.

From per-head distributions to prefetch candidates. At inference time, each active head n decodes its top-m predicted deltas into a set of candidate addresses,

$$\mathcal{C}_n = \big\{\, a_t + \delta_j \;:\; \delta_j \in \mathrm{top}_m\, p_n(\cdot \mid h_L) \,\big\}. \tag{4}$$

Because nearby horizons often predict the same or nearby addresses, candidates are pooled across heads into a single ranked list before the top-k budget is applied, with farther horizons discounted by $\gamma \in (0, 1)$ so that nearer-term, typically more confident, predictions are preferred when addresses collide:

$$\mathrm{score}(a) = \sum_n \gamma^{\,\mathrm{rank}(n)}\, p_n(a - a_t \mid h_L) \cdot \mathbb{1}[a \in \mathcal{C}_n], \tag{5}$$

where $\mathrm{rank}(n)$ orders horizons from nearest (0) to farthest within the span, and $p_n(a - a_t \mid h_L)$ is head n's probability for the specific delta that produces address a – well defined whenever $a \in \mathcal{C}_n$, since a fixed current address a_t makes the map $\delta_j \mapsto a_t + \delta_j$ one-to-one. The k addresses with the highest $\mathrm{score}(a)$ are issued as the model's prefetch candidates for that decision point. Figure 2 illustrates this pipeline end to end, from the raw access stream through per-horizon prediction to the final top-k candidate list.

4 Evaluation Models

Most previously proposed ML-based prefetchers are evaluated purely on next-step prediction accuracy—precision and recall over the top-k most probable candidates—which implicitly assumes a correct prediction is useful the instant it is made. Section 3.1 formalizes why that assumption breaks under a real delay d: a prefetch takes real time to arrive, and a correct-but-late prediction is no more useful than a wrong one. We report results under two views of that same decision problem, which differ in how they measure delay.

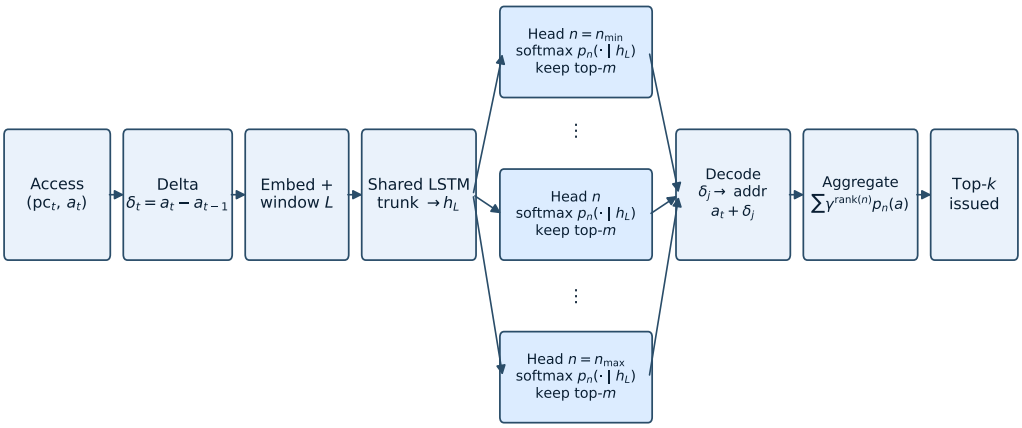

Figure 2: **Multi-horizon prefetch pipeline.** Each access updates the running delta δ_t, which feeds a context window into the shared LSTM trunk. Every horizon n in the target span has its own head: a softmax over the delta vocabulary, truncated to its top-m most confident predictions. Predictions from all heads are decoded into candidate addresses, pooled with a $\gamma^{\text{rank}(n)}$ discount that favors nearer-term horizons on collision, and the top-k highest-scoring addresses are issued as prefetches.

Delayed-availability hit rate is the fraction of accesses served from cache under the simplified, access-indexed delay rule of Section 3.1 (Delay): a candidate issued at access t lands at access $t + d$, so d counts accesses rather than cycles—the same hit-rate metric used throughout prior work, but computed honestly under $d > 0$ rather than assuming instant fill. *Stall-aware runtime* instead reports the full objective $J(\pi)$ (Section 3.1, Objective): total simulated cycles, where a candidate's d-cycle delay is measured against the same cumulative cycle count c_t that the objective itself accumulates, so a policy is credited not just for eventually landing a correct prediction but for how much stall time it actually removes. We report both because hit rate alone can look nearly flat across policies even when the runtime difference is large (compare Figures 3 and 4); stall-aware runtime, with its stall-aware cycle-based notion of delay, is what actually determines deployment value.

5 Experiments

The focus of our work has been to train models that capture the practicality of prefetching and simulating real life scenarios. Instead of focusing on the accuracy of the models, we see what impact prefetching can have over the previous state-of-the-art models and non ML based prefetchers.

5.1 Datasets

We evaluate memory-access traces generated with CXL-DMSim (Wang et al., 2026), a full-system cycle-accurate simulator. We use five workloads from SPEC CPU 2017 benchmarks (Bucek et al., 2018): blender, imagick, lbm, parest and povray. We evaluate two simulated CXL device configurations for each workload: an ASIC-based configuration with a 154 ns memory-access overhead relative to conventional DDR memory, and an FPGA-based configuration with a 244 ns overhead.

5.2 Experimental Setup

We split each trace into a training and test set using the first 70% of accesses for training and the final 30% for evaluation, preserving temporal order rather than shuffling, and train every

model independently on each trace rather than pooling across traces. All learned policies are trained with Adam (Kingma & Ba, 2015); the graph-based SGDP (Yang et al., 2023) baseline instead uses a 90%/10% split for its own training and dictionary construction, but is evaluated downstream over the same window as every other policy (Appendix B). We report the full training and evaluation hyperparameters in Appendix B.

Equal-budget protocol. Every policy—learned or classical—issues at most k prefetch candidates at the same decision points, with cadence τ (Section 3.1): one decision every τ demand accesses. We vary τ across experiments, as specified with each result below (τ=30 for Figures 3–4, τ=10 for Figure 5). Classical prefetchers update their tables on *every* access (as in hardware) but may only *issue* at those shared decision points. Learned models are held to the same rule: their input context window advances by one access on every demand access, but the model is only invoked to produce a new candidate list at a decision point, and never issues more than k candidates there. Any difference we measure is therefore attributable to prediction quality, not to one policy being permitted to see more of the trace or to prefetch more aggressively than another.

Address and delta granularity. Every trace address is the raw byte address CXL-DMSim recorded (Section 5, Datasets); $\delta_t = a_t - a_{t-1}$ (Section 3) is a byte-address difference, and every learned model predicts and outputs at this same granularity. The cache below allocates and evicts one entry per distinct byte address, with no cache-block-size quantization; we use *line* throughout for its storage unit, but it should be read as *one raw address*, not a fixed-size aligned block.

Cache model. Every policy owns an independent cache instance, so no policy's misses or evictions affect another's; the only thing every policy shares is the trace itself. Each cache is fully associative, address-keyed, and holds 1 000 lines under LRU replacement, with no explicit modeling of cache-block size or set associativity—every distinct address in the trace occupies exactly one line. Capacity, associativity, and replacement policy are identical across every policy in every experiment, governed by the delayed-availability and stall-aware rules of Section 4. All reported numbers are single deterministic runs: every model is evaluated in inference mode against a fixed trace and a fixed train/test split, so we do not report variance across random seeds.

5.3 Policies Considered

LRU is a no-prefetch control that isolates how much of the hit rate is attributable to reuse alone. Stride is a per-PC stride predictor with a saturating confidence counter; Berti-like approximates the multi-delta local-history mechanism of Berti (Navarro-Torres et al., 2022) by retaining several confident deltas per PC; and Best-Offset follows (Michaud, 2016), with the candidate-offset list drawn from the most frequent training-region deltas. LSTM-Step50 is our single-horizon ML baseline: an LSTM trained to predict the cumulative delta 50 accesses ahead. Multihead 50–55 and Multihead 50–70 are the proposed shared-trunk multi-horizon models, which expose candidates from every horizon in their span under the same top-k budget. Horizon-GNN is a session-graph GNN following SGDP (Yang et al., 2023), adapted to predict the same N-step-ahead cumulative delta directly in a single forward pass rather than through the original model's autoregressive top-1 rollout, under an identical top-k budget.

5.4 Evaluation Models

We report both metrics defined in Section 4: delayed-availability hit rate and stall-aware total cycles, the objective $J(\pi)$ from Section 3.1.

Figures 3 and 4 give a per-trace view of six policies at a single operating point (k=20, cadence 30, d=30) across all ten workload/configuration traces (blender, imagick, lbm, parest, and povray, each × ASIC/FPGA). Of the three classical baselines, we plot only Stride, which is the strongest of the three at this operating point on both metrics. Both multi-horizon variants are shown alongside LSTM-Step50: Multihead 50–55 sits between

the single-horizon baseline and MULTIHEAD 50–70 on most traces, though not uniformly—on LBM FPGA it falls slightly below LSTM-STEP50 on hit rate (0.861 vs. 0.862), the one trace where the narrower six-horizon span does not pay for itself. HORIZON-GNN never beats STRIDE, LSTM-STEP50, or either MULTIHEAD variant on either metric at this operating point, and only edges past the LRU no-prefetch control on LBM, IMAGICK, and PAREST; on BLENDER and POVRAY it underperforms even doing nothing. One possible contributing factor is that Horizon-GNN is optimized for single-step prediction, whereas our evaluation requires direct long-horizon predictions.

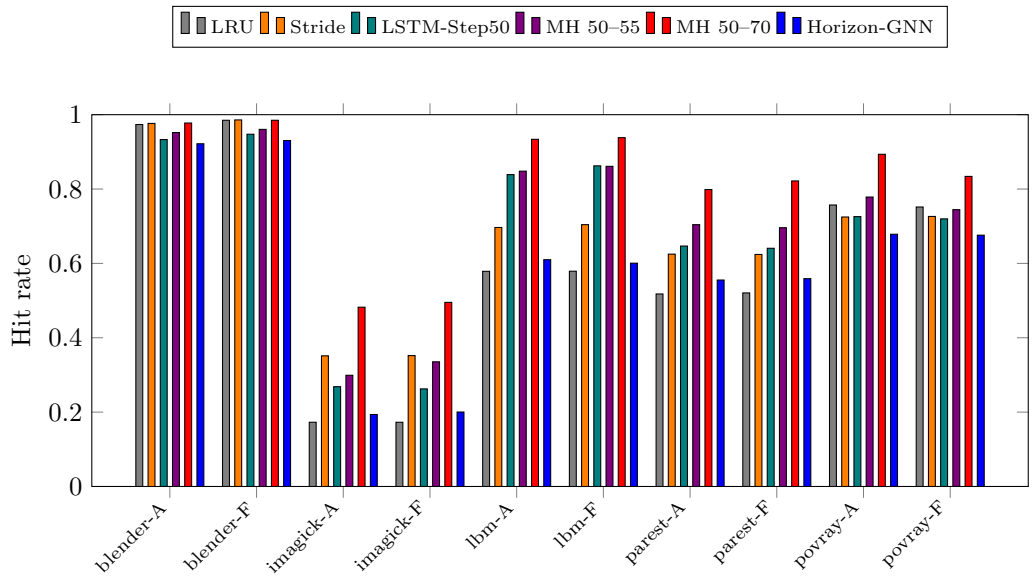

Figure 3: **Delayed-availability hit rate**, per trace, at k=20, cadence 30, d=30.

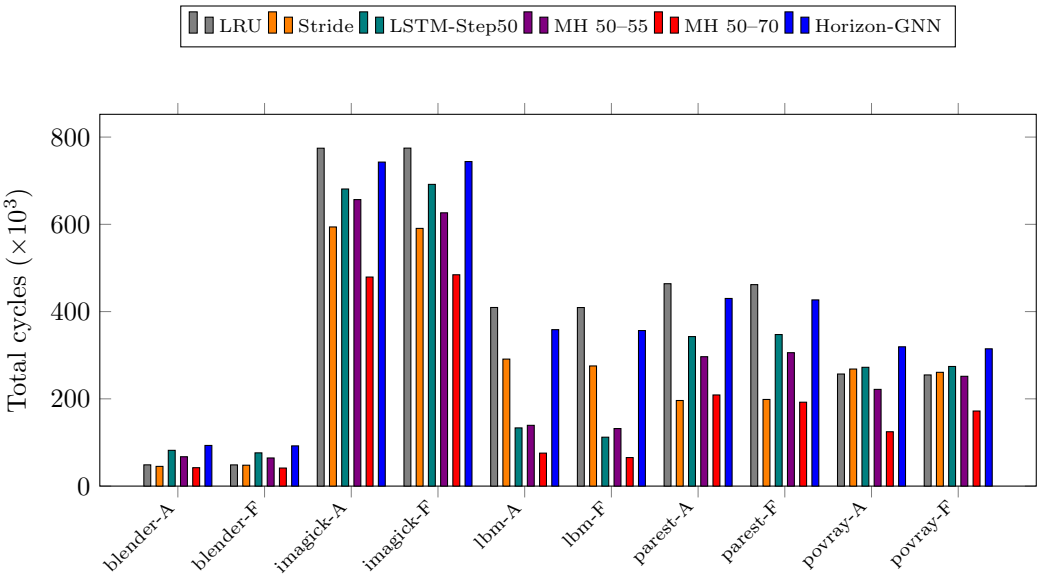

Figure 4: **Stall-aware runtime** (total cycles, lower is better), per trace, at k=20, cadence 30, d=30.

Figure 5 shows the effect of increasing prefetch latency at k=20 and cadence 10, averaged over all ten traces. MULTIHEAD 50–70 reduces simulated cycles relative to LSTM-STEP50

by 47.7% at d=10, increasing to 67.1% at d=100, after which the improvement plateaus. This plateau follows from the cost model itself: at large d a clean miss's cost (1+d cycles) dominates a hit's cost (1 cycle), so each policy's runtime scales roughly with its own miss rate times d, and the relative gap between two policies converges toward the ratio of their miss rates rather than continuing to widen. Because CXL-attached memory already operates in the 100–300-cycle range where this plateau sets in, the advantage MULTIHEAD 50–70 shows here reflects the steady-state regime a real deployment would see, not a transient effect confined to small delays. MULTIHEAD 50–55 also outperforms STRIDE on average, although STRIDE performs better on IMAGICK, illustrating that a narrow prediction range is not uniformly effective across workloads (Appendix A.1). The substantially higher costs of HORIZON-GNN are reported in Appendix A.1.

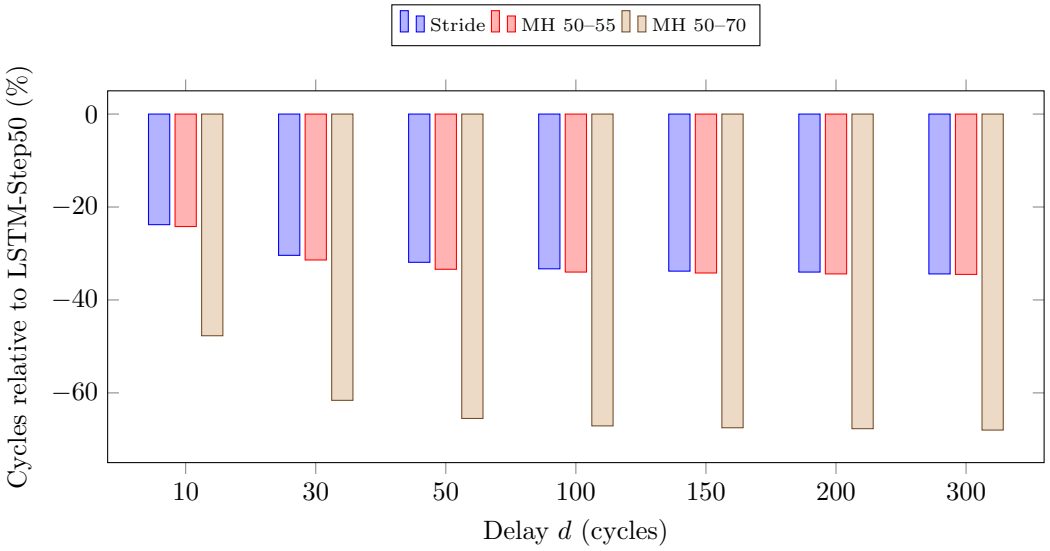

Figure 5: **Stall-aware advantage grows with delay.** Mean cycles relative to LSTM-Step50 (0%, not plotted) at k=20, cadence 10, averaged over the ten traces from Figures 3–4. All three policies cost fewer cycles than the single-horizon baseline on this ten-trace mean; MULTIHEAD 50–70's margin widens sharply through d=100 and then plateaus, while STRIDE and MULTIHEAD 50–55 track each other closely, a pattern driven by IMAGICK (see discussion above).

6 CONCLUSION

We study learned prefetching under delayed availability and stall-aware runtime, showing that predictive accuracy alone is not a reliable proxy for downstream prefetching utility. We extend LSTM-based prefetching to a multi-horizon setting using a shared sequence encoder with separate prediction heads for different future lead times. Our experiments demonstrate that aggregating predictions across multiple horizons substantially improves simulated runtime compared with single-horizon prediction and classical prefetching baselines. An ablation of individual prediction heads further shows that the benefit arises from combining information across lead times rather than selecting a particularly effective individual horizon. These findings highlight the importance of evaluating learned prefetchers under explicit latency constraints.

AI USE STATEMENT

We used ChatGPT and Claude to obtain feedback on manuscript presentation and for generating the code required for the experiments. The authors independently reviewed all AI-assisted suggestions, verified the reported experimental results and technical claims, and take responsibility for the final content of this work.

References

Susanne Albers. On the influence of lookahead in competitive paging algorithms. *Algorithmica*, 18(3):283–305, 1997.

Alitaghi Asgari, Andrew Gunter, Mehdi Saeidi, Mieszko Lis, and Prashant J Nair. Mpmlp: A case for multi-page multi-layer perceptron prefetcher. In *The 2021 Workshop on ML for Computer Architecture and Systems (MLArchSys 2021), held in conjunction with ISCA*, 2021.

Daniel S Berger, Karthik Kumar, Midhul Vuppalapati, Chet Douglas, Jesse Sathre, Ian Robinson, Prateek Tandon, and Mark D Hill. Cxl in cloud practice: Practical lessons for incrementally scaling deployment. *IEEE Transactions on Computers*, 2026.

James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 41–42, 2018.

Pei Cao, Edward W Felten, Anna R Karlin, and Kai Li. A study of integrated prefetching and caching strategies. *ACM SIGMETRICS Performance Evaluation Review*, 23(1):188–197, 1995.

Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995. doi: 10.1109/12.381947.

Kenneth M Curewitz, P Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pp. 257–266, 1993.

Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. In *International Conference on Machine Learning*, pp. 1919–1928. PMLR, 2018.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9 (8):1735–1780, 1997.

Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture*, pp. 252–263, 1997.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

P Krishnan and Jeffrey Scott Vitter. Optimal prediction for prefetching in the worst case. *SIAM Journal on Computing*, 27(6):1617–1636, 1998.

Nanqinqin Li, Yuhong Zhong, Asaf Cidon, and Michael J. Freedman. Harvesting Sub-Microsecond CXL memory stalls with LiteSwitch. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 903–920, Seattle, WA, July 2026. USENIX Association. ISBN 978-1-939133-55-7. URL `https://www.usenix.org/conference/osdi26/presentation/li-nanqinqin`.

Bryan Lim, Sercan O. Arik, Nicolas Loeff, and Tomas Pfister. Temporal fusion transformers for interpretable multi-horizon time series forecasting, 2020. URL `https://arxiv.org/abs/1912.09363`.

Pierre Michaud. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 469–480. IEEE, 2016.

Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Víctor Viñals-Yúfera, Daniel A Jiménez, and Alberto Ros. Berti: an accurate local-delta data prefetcher. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 975–991. IEEE, 2022.

Kyle J Nesbit and James E Smith. Data cache prefetching using a global history buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pp. 96–96. IEEE, 2004.

Mehran Shakerinava, Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Multi-lookahead offset prefetching. 2019. URL https://api.semanticscholar.org/CorpusID:199570386.

Jeffrey Scott Vitter and P Krishnan. Optimal prefetching via data compression. *Journal of the ACM*, 43(5):771–793, 1996.

Xi Wang, Jie Liu, Jianbo Wu, Shuangyan Yang, Jie Ren, Bhanu Shankar, and Dong Li. Performance characterization of cxl memory and its use cases. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1048–1061. IEEE, 2025. doi: 10.1109/ipdps64566.2025.00097. URL http://dx.doi.org/10.1109/IPDPS64566.2025.00097.

Yanjing Wang, Lizhou Wu, Wentao Hong, Yang Ou, Zicong Wang, Sunfeng Gao, Jie Zhang, Sheng Ma, Dezun Dong, Xingyun Qi, Mingche Lai, and Nong Xiao. Cxl-dmsim: A full-system cxl disaggregated memory simulator with comprehensive silicon validation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 45(4): 1787–1801, 2026. doi: 10.1109/TCAD.2025.3607145.

Ruofeng Wen, Kari Torkkola, Balakrishnan Narayanaswamy, and Dhruv Madeka. A multi-horizon quantile recurrent forecaster, 2018. URL https://arxiv.org/abs/1711.11053.

Yiyuan Yang, Rongshang Li, Qiquan Shi, Xijun Li, Gang Hu, Xing Li, and Mingxuan Yuan. Sgdp: A stream-graph neural network based data prefetcher. In *2023 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE, 2023.

Pengmiao Zhang, Ajitesh Srivastava, Anant V Nori, Rajgopal Kannan, and Viktor K Prasanna. Transformap: Transformer for memory access prediction. *arXiv preprint arXiv:2205.14778*, 2022.

A Additional Results

A.1 Raw Stall-Aware Cycles Across Delay Configurations

Figure 5 plots Stride, Multihead 50–55, and Multihead 50–70 as cycles *relative to* LSTM-Step50 at k=20, cadence 10. Table 1 gives the underlying raw numbers behind that figure — mean absolute stall-aware cycles at the same operating point, across the same delay range, over the same ten-trace set used throughout the main text (Figures 3–5) — including the two policies the figure omits, LRU and Horizon-GNN, for reference. Every row here is averaged over all ten traces, including imagick.

Table 1: **Raw stall-aware cycles by delay.** Mean total cycles ($\times 10^3$, lower is better) at k=20, cadence 10, averaged over all ten traces from Figures 3–5.

Prefetcher	d=10	d=30	d=50	d=100	d=150	d=200	d=300
LRU (no prefetch)	150.1	390.2	630.4	1230.8	1831.2	2431.6	3632.4
Stride	68.0	145.4	224.4	422.3	620.2	818.0	1213.8
LSTM-Step50	89.3	208.9	329.8	633.1	936.4	1239.9	1849.0
Multihead 50–55 *(ours)*	67.7	143.3	219.7	417.6	615.8	814.0	1210.4
Multihead 50–70 *(ours)*	46.7	80.2	113.9	208.3	304.4	400.4	592.4
Horizon-GNN	135.7	347.4	560.1	1099.9	1638.9	2174.7	3248.8

Why the ten-trace mean is close. Multihead 50–55 edges out Stride by only 0.3–2.1% overall because the two traces pull in opposite directions at very different strengths. On the eight traces excluding imagick, Multihead 50–55 beats Stride by 21–39% at every delay — and by 48–65% on povray ASIC specifically, the single widest margin in the set. On imagick alone the ordering reverses: Stride costs 28–34% *fewer* cycles than Multihead 50–55, because a 50–55 horizon span is not wide enough to reliably beat a fixed stride once imagick's delay must be covered, while Multihead 50–70's wider span still handles it. The ten-trace mean only tips toward Multihead 50–55 because the other eight traces collectively outweigh imagick's pull, not because the narrower span is a robust win.

A.2 Do Individual Heads Matter, or Only the Aggregate?

Section 3.2 pools candidates across every head in Multihead 50–70's target span with a $\gamma^{\mathrm{rank}(n)}$ discount. This raises a natural question: does that pooling add value, or would a single well-chosen head do just as well? Because every head is trained jointly with the shared trunk, we can answer this directly from the existing Multihead 50–70 checkpoint—no retraining required—by running the model with only one active head at inference time and evaluating it exactly as a single-horizon prefetcher, through the same stall-aware harness used everywhere else in this paper. For this ablation, each active head is reconstructed with its per-head candidate cap m set to k=20 (rather than the trained m=10; Table 2, footnote), so a lone head exposes at least k candidates and both the single-head and 21-head aggregate policies may issue the same maximum k=20 prefetches per decision—otherwise a lone head would be capped at 10 candidates while the aggregate draws from 210, and the comparison below would demonstrate the value of a larger candidate pool rather than cleanly isolating the value of multiple horizons.

Figure 6 reports this for five representative heads ($n \in \{50, 55, 60, 65, 70\}$, evenly spaced across the trained span) against the full 21-head aggregate, averaged over the same ten-trace set as everywhere else in this paper, at k=20, cadence 10. Every individual head performs almost identically to every other head—at d=10 the five heads span 88.5–93.9 $\times 10^3$ cycles, a 6.1% range—and all of them cost roughly twice what the full aggregate costs (46.0×10^3 at d=10), a gap that widens to over 3× by d=300 (1801–1983 $\times 10^3$ per head vs. 571.4×10^3 for the aggregate). This holds consistently across all ten traces, not just as an artifact of one workload: the multi-horizon model's advantage comes almost entirely from pooling predictions across many horizons at each decision point, not from any individual head happening to predict a particularly useful lead time. A deployment restricted to a

single-horizon model would gain little from choosing one head over another, since the benefit lives specifically in evaluating several and aggregating.

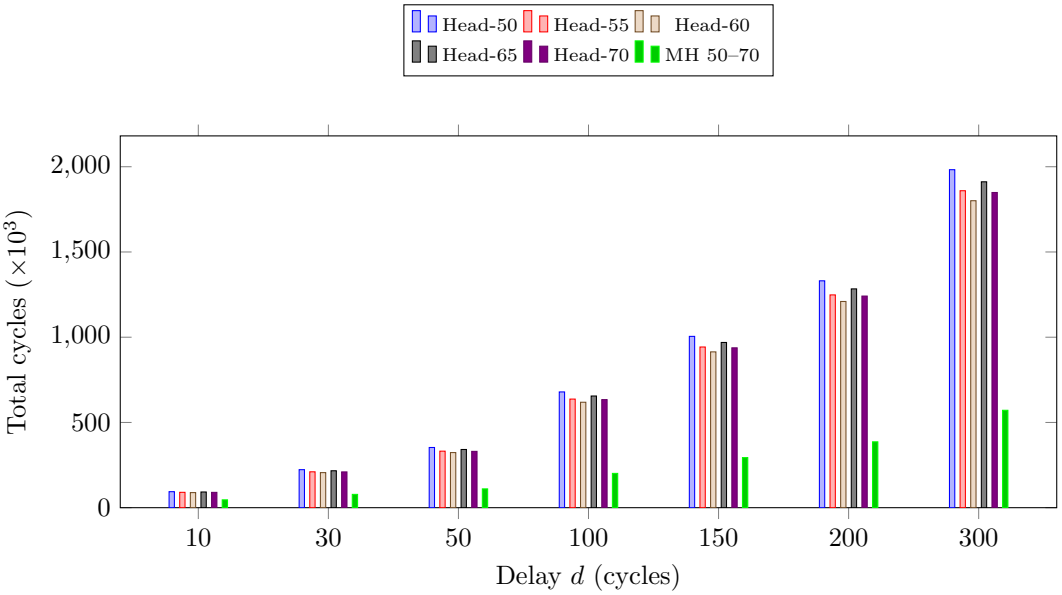

Figure 6: **No single head matches the aggregate.** Stall-aware total cycles (lower is better) for five individual heads of the MULTIHEAD 50–70 checkpoint, each reconstructed with its per-head candidate cap raised to m=k=20 so a lone head can fill the same budget as the aggregate, evaluated alone as single-horizon prefetchers against the full 21-head aggregated policy, averaged over all ten traces at k=20, cadence 10. The five heads are nearly indistinguishable from each other and all cost roughly 2–3× what the aggregate costs, growing with delay.

B Training and Evaluation Hyperparameters

Table 2 gives the full training and evaluation configuration for every learned policy, referenced from Section 5.2.

C Sensitivity to γ and Cache Capacity

Two knobs in our setup are fixed throughout the main text: the cross-head discount γ=0.9 and the cache capacity C=1 000 lines. We check both on all ten traces at k=20, cadence 10, d=30; neither check requires retraining, since both parameters are applied at inference time. Table 3 summarizes the range of MULTIHEAD 50–70's stall-aware cycles across each sweep.

Cache capacity. Varying $C \in \{250, 500, 1\,000, 2\,000, 5\,000\}$ – a 20× range – MULTIHEAD 50–70 remains the lowest-cycle policy of the four tested (LRU, STRIDE, LSTM-STEP50, MULTIHEAD 50–70) at every capacity on eight of the ten traces. On PAREST (both configurations), STRIDE is cheaper than MULTIHEAD 50–70 at every capacity in this sweep – consistent with PAREST ASIC already being an exception in Figure 4 at the main text's cadence-30 operating point.

Cross-head discount. Varying $\gamma \in \{0.5, 0.7, 0.9, 0.95, 0.99\}$, MULTIHEAD 50–70's own cycle count is nearly flat on BLENDER and LBM (under 13% range), but varies substantially more on IMAGICK (43–59% range), PAREST (27–58%), and POVRAY (28–98%), growing worse as γ decreases. This is not noise: a low γ heavily discounts the farther heads in the 50–70 span, and these are the traces where that far-horizon information matters most. At the lowest γ=0.5 tested, MULTIHEAD 50–70 loses to STRIDE (at C=1 000) on IMAGICK and PAREST

Table 2: **Training and evaluation hyperparameters.** LSTM-STEP50 and both MULTIHEAD variants share the same recipe on nine of the ten traces; IMAGICK uses a reduced recipe (footnote below). HORIZON-GNN's *training* follows the published SGDP (Yang et al., 2023) setup, including a wider held-out split; its *downstream evaluation* (Figures 3–5, Table 1) instead uses the identical access window as every other policy (footnote below).

	STEP50 / MULTIHEAD	HORIZON-GNN
Train/test split	70% / 30% (temporal)	90% / 10% (temporal, training only[3])
Optimizer	Adam, lr = 0.01	Adam, lr = 0.001
Context window	128 accesses	32 accesses
Embedding / hidden size	128 / 128	100
Batch size	128	128
Train steps	up to 200 000[1]	3 epochs
Early stopping	30 evals, no improvement	—
Output vocabulary cap	50 000 per horizon	top 1 000 deltas
Per-head candidate cap m	10[2]	—
Cross-head discount γ	0.9 (Multihead only)	—
Prefetch budget k	10 or 20, stated per result	10 or 20, stated per result
Eval. window (hit rate)	50 000 accesses/trace	50 000 accesses/trace[3]
Eval. window (stall-aware)	30 000 accesses/trace	30 000 accesses/trace[3]

Notes: [1]IMAGICK's raw input-delta vocabulary (≈200 000 distinct values) makes this recipe impractically slow specifically on that trace; its LSTM-STEP50 and MULTIHEAD checkpoints (both ASIC and FPGA variants) instead use a capped vocabulary of 20 000 (vs. 50 000) and 60 000 train steps (vs. up to 200 000). All other hyperparameters and the evaluation protocol are unchanged.
[2]Each head (and LSTM-STEP50) keeps its own top-m=10 predicted deltas before pooling; LSTM-STEP50 and every classical baseline are always evaluated with m (or an unbounded candidate list, for classical baselines) reconstructed to match the budget k at inference time, so they can fill the full k slots. The MULTIHEAD aggregate pools m=10 candidates from every one of its (6 or 21) heads before applying the $\gamma^{\text{rank}(n)}$ discount and the final top-k cut, so it draws from 60–210 raw candidates and is never limited by any single head's cap. The one place this cap directly binds is the single-head ablation of Appendix A.2, where by construction only one head is active; there we reconstruct that head with m=k=20 specifically so a lone head can also fill the full budget, keeping that comparison budget-fair against the 21-head aggregate.
[3]HORIZON-GNN's own checkpoint is trained on the first 90% of each trace, following Yang et al. (2023)'s 90%/10% split. To avoid evaluating it on any data it was trained on, the shared downstream comparison in Figures 3–5 and Table 1 evaluates every policy – LRU, STRIDE, LSTM-STEP50, MULTIHEAD, and HORIZON-GNN – inside the same loop, over the final 10% of each trace, using the same 50 000- or 30 000-access window for every policy: no policy in those results is scored over a different number of demand accesses, or over data it was trained on, than any other.

(both configurations). At the γ=0.9 used throughout this paper, it recovers the lead on IMAGICK (220,692 vs. 289,591) but *not* on PAREST, where STRIDE stays cheaper regardless of γ or cache capacity in this sweep. $\gamma \geq 0.9$ is sufficient to keep MULTIHEAD 50–70 ahead of STRIDE on eight of the ten traces tested here.

D RELATED WORK

Prefetching and caching have been studied from several perspectives: systems and architecture, learned prediction, and theory. This section surveys the classical and theoretical strands and situates our delayed-utility framing relative to them.

Classical prefetchers. Berti Navarro-Torres et al. (2022) selects the best delta for a given instruction from its own local access history, improving on Best-Offset (Michaud, 2016) and Multi-Lookahead Offset Prefetching (Shakerinava et al., 2019), which instead prefetch from a single offset table shared across instructions, blurring together per-instruction patterns that Berti's local-history tracking keeps separate.

Table 3: **Sensitivity of Multihead 50–70's stall-aware cycles to γ and cache capacity** (k=20, cadence 10, d=30). Min–max cycles across the swept values; the main text uses γ=0.9 and C=1 000.

Trace	Cache capacity $C \in \{250, \ldots, 5\,000\}$	Discount $\gamma \in \{0.5, \ldots, 0.99\}$
BLENDER ASIC	33,600–38,590	34,590–36,376
BLENDER FPGA	39,313–70,178	44,732–50,179
IMAGICK ASIC	215,892–225,231	216,316–310,320
IMAGICK FPGA	203,430–206,970	203,310–322,230
LBM ASIC	31,066–31,066	30,966–31,077
LBM FPGA	30,840–30,840	30,801–30,850
PAREST ASIC	47,940–65,040	46,140–72,900
PAREST FPGA	48,420–55,590	48,900–61,953
POVRAY ASIC	33,532–53,958	45,314–57,831
POVRAY FPGA	43,634–89,652	66,046–130,890

Theoretical foundations. Prefetching has also long been studied through competitive analysis. Vitter and Krishnan (Vitter & Krishnan, 1996) showed a compression-based predictor achieves an asymptotically optimal fault rate for Markov-source access sequences; Krishnan and Vitter (Krishnan & Vitter, 1998) extended this to worst-case guarantees that hold without assuming any particular source model, and Curewitz et al. (Curewitz et al., 1993) turned the same idea into a practical algorithm evaluated on real traces. Cao et al. (Cao et al., 1995) folded delayed utility directly into this framework, proving that integrated prefetching-and-caching strategies stay within a factor of two of the offline optimum.

Machine learning based prefetchers. Learned predictors have increasingly replaced hand-designed heuristics for capturing memory-access patterns. Hashemi et al. (2018) were among the first to predict future addresses directly from LSTM embeddings of the access stream, without hand-crafted features. The ISCA 2021 ML-Based Data Prefetching Competition spurred lighter-weight architectures for real-time deployment, including a multi-layer perceptron over page contexts (Asgari et al., 2021) and a transformer-based predictor (Zhang et al., 2022). SGDP (Yang et al., 2023) models the access stream as a graph to capture dependencies a purely sequential model can miss, but produces multi-step predictions only by rolling its single-step output forward autoregressively – a strategy that compounds error with lead time and, as Section 5 shows, cannot substitute for genuinely modeling multiple horizons directly.

Together, this work establishes timeliness and delayed utility as first-order theoretical concerns and shows learned models can capture patterns beyond hand-designed heuristics – but the two threads remain separate: learned prefetchers are still evaluated as if every correct prediction were instantly useful. We close this gap empirically, evaluating multi-lead-time predictors under a realistic delayed-availability and stall-aware framework (Section 4).