COSMOS: RL-Enhanced Locality-Aware Counter Cache Optimization for Secure Memory

Haoran Geng
University of Notre Dame
Notre Dame, IN, USA
hgeng@nd.edu

Xiaoyang Lu
Illinois Institute of Technology
Chicago, IL, USA
xlu40@illinoistech.edu

Yuezhi Che
Wuhan University
Wuhan, China
cheyuezhi@whu.edu.cn

Ziang Tian
Wuhan University
Wuhan, China
ziangtian@whu.edu.cn

Dazhao Cheng
Wuhan University
Wuhan, China
dcheng@whu.edu.cn

Xian-He Sun
Illinois Institute of Technology
Chicago, IL, USA
sun@illinoistech.edu

Michael Niemier
University of Notre Dame
Notre Dame, IN, USA
mniemier@nd.edu

X. Sharon Hu
University of Notre Dame
Notre Dame, IN, USA
shu@nd.edu

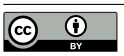
Abstract

Secure memory systems employing AES-CTR encryption face significant performance challenges due to high counter (CTR) cache miss rates, especially in applications with irregular memory access patterns. These high miss rates increase memory traffic and latency, as each CTR cache miss triggers additional DRAM accesses. To address these bottlenecks and adapt to diverse access patterns, we propose COSMOS (Counter Optimized Secure Memory Operation Scheme), a novel solution leveraging reinforcement learning to reduce long memory access latency. COSMOS integrates two RL-based specialized predictors: one for data location prediction and another for CTR locality prediction, each with a well-defined state space, action space, and reward function. The RL-based data location predictor determines whether data reside on-chip or off-chip after an L1 cache miss, enabling early CTR access for off-chip predictions with minimal changes to the existing cache hierarchy. The RL-based CTR locality predictor identifies CTRs with high locality, supporting a locality-centric CTR cache (LCR-CTR) to improve cache efficiency and reduce miss rates. COSMOS improves performance over MorphCtr by 25% in for irregular memory access applications, with minimal hardware overhead.

Keywords

secure memory systems; AES-CTR encryption; reinforcement learning; counter cache optimization; graph algorithms

ACM Reference Format:
Haoran Geng, Xiaoyang Lu, Yuezhi Che, Ziang Tian, Dazhao Cheng, Xian-He Sun, Michael Niemier, and X. Sharon Hu. 2025. COSMOS: RL-Enhanced Locality-Aware Counter Cache Optimization for Secure Memory. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3725843.3756047



This work is licensed under a Creative Commons Attribution 4.0 International License.
MICRO '25, Seoul, Republic of Korea
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1573-0/2025/10
https://doi.org/10.1145/3725843.3756047

1 Introduction

Many companies are migrating their data to the cloud to take advantage of potential cost savings. However, securing sensitive and private data remains a critical concern for various applications. In cloud environments, companies lose control over physical access to computing facilities, raising the risk of physical attacks where malicious individuals with access can steal or tamper with sensitive application data. To address these security challenges, leading companies like Intel and AMD have developed secure memory systems, such as Intel SGX [13] and AMD SEV [4]. Both system use the Advanced Encryption Standard Counter Mode (AES-CTR) [13] as their preferred memory encryption scheme to ensure data confidentiality. In AES-CTR, a counter (CTR) is assigned to each data block, which is incrementally updated and encrypted using AES. This encrypted CTR is then XORed with the plaintext data to produce the corresponding ciphertext for secure storage. The memory controller (MC) stores CTRs in DRAM. When a last-level cache (LLC) miss occurs, the MC fetches the CTR associated with the missing data block from DRAM. The CTR is then used to decrypt the block and verify its integrity upon arrival.

AES-CTR is complemented by Message Authentication Codes (MACs) and Merkle Trees (MTs) to prevent replay attacks [25, 71]. MACs ensure data authenticity, while MTs verify CTR integrity. Under this scheme, each data block is associated with a CTR and a MAC, both structured within a MT whose root is securely stored on-chip. While this combination of AES-CTR, MACs, and MTs provides robust security, it introduces significant memory access overhead. During each memory read access, the MAC and CTR are fetched and verified against the MT root to ensure data authenticity and CTR integrity. To prevent replay attacks, the CTR is incremented with each memory write, acting as a timestamp to ensure MT verification detects and blocks any attempt to reuse old data or CTRs.

Consequently, memory writes involve updating the MAC, incrementing the CTR for encryption, and accessing MT nodes for CTR integrity verification. These additional security-specific memory accesses significantly increase memory traffic, ultimately affecting overall system performance.

To mitigate this overhead, a CTR cache is implemented in the MC to reduce CTR DRAM accesses [13, 43, 47]. However, the large memory footprints and irregular access patterns of modern applications can result in a high CTR cache failure rate [3, 5, 28, 36, 41]. Consequently, even with the use of a CTR cache, CTR DRAM accesses remain largely unmitigated, and the CTR cache instead adds latency to the critical path for accessing CTRs. In order to improve CTR cache hit rates, previous work has proposed Morphable CTR (MorphCtr) [46], which allows a single CTR block to manage a larger number of data blocks, thus improving CTR cacheability. However, as we will later demonstrate, even with a 1:128 CTR-to-data block mapping ratio, applications with irregular access patterns continue to suffer from high CTR cache miss rates, resulting in underutilization of the CTR cache.

We observe that the high CTR cache miss rate in applications with irregular memory access patterns is primarily due to the CTR cache being accessed only after an LLC miss occurs. In such applications, the LLC miss rate is typically high, as hot data is already cached in L1 and L2, leading to cold CTRs flowing into the CTR cache. This leads to inefficient CTR cache utilization, as cold CTRs are less likely to be accessed with a small reuse distance.

A recently proposed solution, EMCC [65], suggests moving the CTR access earlier in the cache hierarchy, ideally within the L2 cache. EMCC demonstrates that placing the CTR cache at the L2 level allows more hot CTRs to enter the cache, significantly reducing CTR miss rates. However, this solution requires complex modifications to the L2 cache controller, which introduces implementation challenges.

We perform a comprehensive analysis of the performance impact of the CTR cache behavior in secure memory systems (see Sec. 3), which motivates the design of COSMOS (Counter Optimized Secure Memory Operation Scheme). COSMOS is specifically designed to: (1) reduce CTR access latency by predicting off-chip data requests and fetching their corresponding CTRs directly, (2) accurately identify CTRs with high locality, and (3) implement a CTR cache with a locality-centric replacement policy to increase CTR hit rates. The COSMOS design focuses on the following technical aspects.

First, we propose a novel approach to accelerate off-chip CTR accesses while maintaining the existing cache hierarchy. Specifically, we predict whether data is on-chip or off-chip immediately after an L1 cache miss. If the data is predicted to be off-chip, we proactively fetch the corresponding data and its CTR from the main memory and CTR cache. This approach not only accelerates the off-chip CTR accesses by removing the on-chip cache access latency from the critical path, but also provides the opportunity to access the CTR cache earlier after an L1 miss, ensuring that the CTR cache is populated with hot CTRs. Reinforcement learning (RL) provides a promising solution, enabling accurate predictions and adaptability to varying access patterns across diverse applications [7, 35]. Therefore, we propose a lightweight RL-based data location predictor that adapts to varying memory access patterns, enabling speculative fetching of data and CTRs directly from main memory and the CTR cache, while requiring minimal hardware modifications.

Moreover, we observe that applications with irregular memory access patterns continue to experience relatively high CTR cache miss rates when accessing the CTR after an L1 cache miss. To further optimize CTR access following data location prediction, we focus on fully leveraging CTR locality within the CTR cache. This requires a thorough understanding of CTR locality patterns, which we address by employing an RL-based CTR locality predictor to identify CTRs with high data locality after an L1 cache access. Based on the CTR locality predictions, we design a CTR cache with a locality-centric replacement (LCR) policy, which we call the LCR-CTR cache, that retains CTRs identified as having good locality for longer periods to improve the overall CTR cache hit rate. CTRs identified as having good locality are marked and stored in the LCR-CTR cache, while those with poor locality are prioritized for eviction. By dynamically refining CTR locality predictions through RL, the CTR cache efficiently retains high-locality CTRs, significantly improving CTR cache hit rates and reducing CTR access latency.

COSMOS outperforms MorphCtr [46] by 25% for irregular memory access applications. We also compare COSMOS with EMCC [65], an optimized extension of MorphCtr, and observe that COSMOS outperforms EMCC by 10%. In particular, COSMOS achieves these improvements with only 147KB of additional on-chip storage overhead in the MC.

In summary, COSMOS offers the following key contributions:

- We present a comprehensive analysis of the CTR cache hit behavior of applications with irregular memory access pattern in secure memory systems, evaluating its impact on performance, and exploring the potential of various optimization strategies.
- An RL-based data location predictor that predicts whether data is on-chip or off-chip immediately after an L1 cache miss. This enables earlier and more efficient CTR access in the cache hierarchy, while allowing the CTR cache to be populated with hot CTRs.
- An RL-based CTR locality predictor that identifies the locality characteristics of CTRs, distinguishing between those with good and poor locality.
- An LCR-CTR cache that leverages RL-based CTR locality predictions to prioritize retaining CTRs with good locality. The LCR-CTR dynamically adjusts CTR replacement decisions based on access patterns, improving CTR cache hit rates.

2 Background

This section reviews AES-CTR based secure memory, split CTR, MorphCtr, and RL.

2.1 AES-CTR Based Secure Memory

Memory encryption techniques use symmetric key encryption methods, notably AES-CTR, to ensure the confidentiality of off-chip data [17]. These methods are also combined with MAC and MT to verify the integrity of the data [13]. Figure 1 illustrates the AES-CTR encryption process. It combines a data block's physical address (PA) with a CTR to generate ciphertext [18, 62]. The

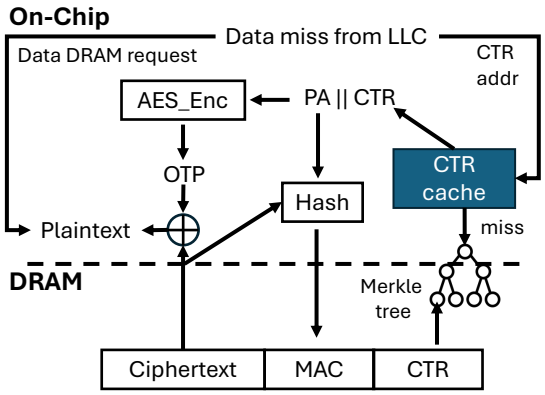


Figure 1: AES-CTR memory protection scheme.

encryption is expressed as:

$$Ciphertext = Plaintext \oplus \text{AES\_Enc}(PA \| CTR).$$

Here, $\|$ and $\oplus$ represent concatenation and XOR operations, respectively. AES_Enc denotes AES encryption, which produces a one-time pad (OTP) from the combination of the PA and CTR .

To ensure data integrity and authentication, an MAC is employed. The MAC verifies that the data remains unaltered and authenticates its origin. For a given ciphertext, the MAC is generated by hashing the concatenation of the ciphertext with the PA and CTR. This process is represented by the equation:

$$MAC = Hash(Ciphertext \| (PA \| CTR)),$$

where *Hash* denotes a cryptographic hash function that produces a fixed-size output. During DRAM reads, the MACs are cross-referenced to verify the integrity and authenticity of the data. However, MAC verification alone is not sufficient to prevent all attacks. For instance, replay attacks can insert outdated data with its corresponding CTR and MAC, evading detection. To address this vulnerability, an MT structure is utilized [58]. As illustrated in Figure 1, CTRs form the leaf nodes of the MT, with each internal node contains a hash derived from its children, and the root hash stored securely on-chip. This structure allows for the detection of any CTR discrepancies during tree traversal, thus protecting against tampering and replay attacks.

AES-XTS, used in Intel SGXv2 [72] and AMD SEV [1, 2, 4], eliminate the need for CTR, thus avoiding the overhead challenges present in AES-CTR [37]. It uses a two-key encryption model, one for ciphertext generation and one as a tweak derived from physical addresses. Although this offers efficiency, AES-XTS lacks inherent support for integrity verification and is vulnerable to ciphertext side channel attacks [32, 67], making it less secure than AES-CTR+MT. Given these limitations, we focus on the more secure AES-CTR+MT design, which supports replay protection - crucial for high-assurance systems.

2.2 Split CTR

The two main challenges with AES-CTR based secure memory are (1) the high CTR cache miss rate and (2) the CTR storage overhead. The high CTR cache miss rate occurs because each data block requires a unique CTR value, leading to frequent accesses to the CTR cache. Misses in the CTR cache can lead to high DRAM CTR access latency, which significantly impacts overall performance. The CTR storage overhead arises from the need to store an increasing number of CTRs as the number of data blocks grows.

To address the challenge of CTR management, Yan et al. [68] proposed a split CTR scheme for efficient memory encryption. In this scheme, the CTR block is divided into two parts: a small per-block minor CTR and a larger major CTR shared by multiple blocks within an encryption page. The split CTR scheme allows one CTR block to map to multiple data blocks (typically 64). This increased granularity of the CTR block leads to higher CTR cache hits and reduced CTR storage overhead compared to the standard CTR scheme [68].

The encryption process using split CTRs is similar to the standard AES-CTR mode, with concatenated major and minor CTRs used to generate the one-time pad:

$$Ciphertext = Plaintext \oplus \text{AES\_Enc}(PA \| CTR_M \| CTR_m),$$

where $\|$ represents the bitwise concatenation operation, $\oplus$ represents the XOR operation, CTR_M denotes the major CTR, and CTR_m denotes the minor CTR.

MorphCtr [46] improves split CTRs by offering a 1:128 ratio of CTR to data blocks, doubling the density. Each MorphCtr CTR block comprises a 57-bit major CTR, 7-bit format field, and space for 128 3-bit minor CTRs. MorphCtr switches between Zero CTR Compression (ZCC) for sparse CTR usage and a uniform format for dense CTR usage.

2.3 Reinforcement Learning

RL [45, 49] is a machine learning approach that enables an agent to autonomously learn how to maximize cumulative rewards by interacting with the environment and obtaining feedback from its actions. The interaction between the agent and the environment at timestep t is represented as a tuple (S_t, A_t, R_{t+1}), where the agent observes the state of the environment S_t, selects an action A_t, and the environment transitions to a new state S_{t+1} while providing a reward R_{t+1} to the agent. The goal of the agent is to discover an optimal policy that maximizes the total cumulative reward from the environment over the long term. To achieve this, the agent must consider the long-term impact of each action rather than focus solely on immediate rewards. The expected cumulative reward associated with the execution of an action A in a given state S is quantified as the Q-value of the state action pair, $Q(S, A)$ [24].

SARSA [45] is an on-policy RL algorithm used to learn a Markov decision process policy. The agent interacts with the environment and updates the Q-value based on the current state S_t, current action A_t, reward R_{t+1} received, next state S_{t+1}, and next action A_{t+1}:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)].$$

The learning rate α determines how quickly the Q-values are updated, and the discount factor γ influences how future rewards are weighed against immediate rewards. As γ approaches 0, the agent becomes more "opportunistic", prioritizing immediate rewards. As γ increases, the agent becomes more "far-sighted", aiming for higher long-term rewards.

RL has proven particularly effective for problems involving rapidly changing environments [38, 56], making it well-suited for

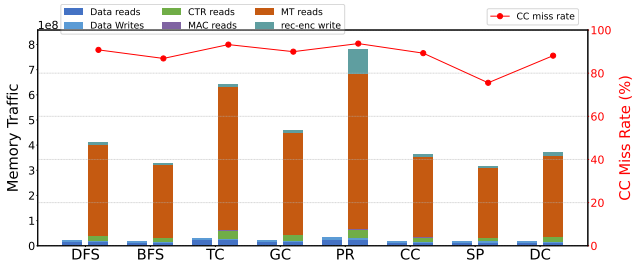


Figure 2: Comparison of memory traffic and CTR cache miss rate across various graph algorithms: non-protect (NP) vs. Secure Memory (w MorphCtr).

handling the CTR accesses of memory-intensive applications. The RL framework has been successfully applied to various memory system optimization tasks, demonstrating its versatility and effectiveness [7, 20, 26, 33, 35, 51, 60, 70].

3 Observations and Motivations

In this section, we first highlight the limitations of current memory encryption methods due to high CTR cache miss rates. Then, we discuss specific motivations for our proposed work.

3.1 Characterizing the CTR Cache Problem

Workloads with irregular memory access patterns often suffer from high memory traffic in secure memory systems [65]. While SGXv1 limited performance due to its small secure memory region (<128MB), real-world applications demand much larger memory footprints. As secure memory scales, CTR management under AES-CTR+MT becomes a key bottleneck, especially in irregular workloads, due to frequent CTR cache misses and the resulting MT traversals. To better understand the root cause, we perform experiments to characterize CTR access behavior.

We simulate a secure memory system [13] with the MorphCtr scheme [46] using Gem5 [8]. The system is modeled as a 4-core setup with a 8MB shared LLC and 128KB CTR cache per core (see Sec. 5 for further details). We use graph benchmarks from GraphBIG [39], evaluated on the GitHub developer social network dataset [44]. The evaluated algorithms include Depth-First Search (DFS), Breadth-First Search (BFS), Graph Coloring (GC), Page Ranking (PR), Triangle Counting (TC), Connected Components (CC), Shortest Path (SP), and Degree Centrality (DC), all executed using 4 threads.

Figure 2 shows that most memory traffic comes from MT nodes read, not re-encryption writes. In MorphCtr, re-encryption only occurs after 67 updates to the same counter—something rare in graph workloads with irregular access patterns. As a result, re-encryption traffic is negligible.

In contrast, CTR access overhead is significant. Each CTR retrieved from DRAM must be authenticated by traversing a MT from leaf to root[72], incurring multiple reads from MT nodes per access. For example, with a 32GB memory and 64B cache line size, there are approximately 537 million cache lines. Since each MorphCtr CTR covers 128 lines, verifying a single CTR requires access to the $\log_2(537M/128) \approx 22$ MT nodes. These MT nodes reads dominate the overall memory traffic, particularly when CTR cache miss rates

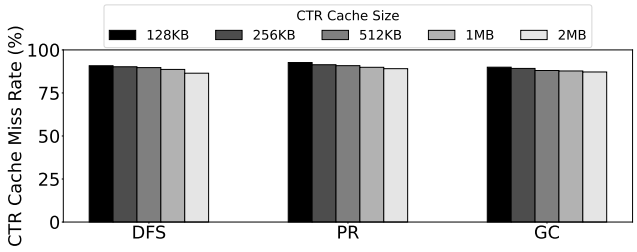


Figure 3: CTR cache size vs. miss rate for graph workloads with irregular access patterns.

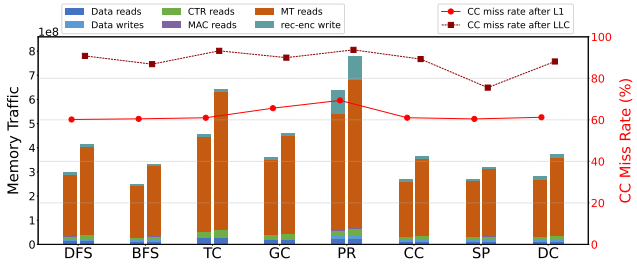


Figure 4: Comparison of memory traffic and CTR cache miss rate across various graph algorithms: Secure Memory access CTR after L1 vs. Secure Memory access CTR after LLC.

are high in these graph applications where the CTR cache miss rate is around 90%.

Key Insight: In secure memory, each CTR cache miss incurs not only DRAM access but also multiple integrity checks via MT traversal. For workloads with irregular access patterns, improving the CTR cache hit rate can significantly reduce total memory traffic and unlock large performance gains.

3.2 CTR Cache Hit is All You Need

To address the memory traffic bottleneck caused by CTR cache misses, we explore two key approaches: scaling the CTR cache size and modifying the CTR access point in the memory hierarchy.

3.2.1 Limited Gains from Scaling CTR Cache Size. If CTR cache misses are the main source of memory overhead, a natural question arises: why not simply increase the CTR cache size to improve hit rates? To explore this, we evaluated three graph applications—DFS, PR, and GC—with varying CTR cache sizes from 128KB to 2MB.

Figure 3 shows the results. Despite an 8× increase in cache capacity, the CTR cache miss rate decreases only by around 5%. This indicates a poor trade-off between hardware overhead and actual cache efficiency.

The primary reason for this behavior is that CTRs are accessed only after an LLC miss, meaning the CTR cache effectively behaves like an "L4" cache. For workloads with irregular access patterns, data with strong locality are already captured by L1, L2 and LLC caches. As a result, the CTR cache is predominantly populated with counters associated with data that exhibit poor reuse, making the traditional CTR cache scaling ineffective.

3.2.2 Enabling Early CTR Access to Improve Locality. Another solution is to allow CTRs to be accessed earlier in the cache hierarchy,

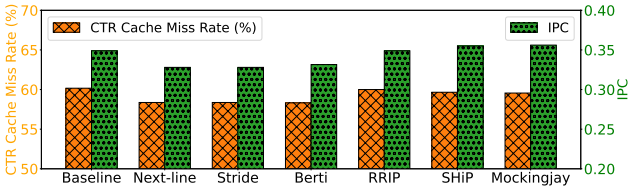


Figure 5: CTR cache miss rate and IPC for Intel SGX with MorphCtr on DFS, showing CTR access after L1 misses when using Next-Line, Stride, Berti prefetchers, and RRIP, SHiP and Mockingjay replacement policies.

enabling access streams with better locality to populate the CTR cache and improve hit rates.

To evaluate this idea, we conduct an experiment where we assume CTRs are accessed after an L1 miss instead of after the LLC, disregarding the hardware constraints required for such early access. As shown in Figure 4, this change reduces the CTR cache miss rate by an average of 25% across graph applications. Although total read and write activity increases slightly (by 5%), the number of reads from the MT nodes – triggered by CTR misses – drops significantly (by 25%).

This leads to the first key insight of this paper: enabling earlier access to CTRs within the cache hierarchy allows CTRs with better locality to populate the CTR cache sooner, improving cache hit rate and reducing secure memory traffic.

3.3 Cache Optimization in CTR Cache

Given the importance of CTR cache hits, a natural follow-up question is: in addition to enabling early CTR access, can existing cache optimization techniques effectively address CTR cache miss challenges?Traditional cache optimization methods have been successfully applied to various memory system problems. However, as we demonstrate in the following section, these conventional approaches face significant limitations when applied to CTR cache optimization, particularly for applications with irregular memory access patterns.

We evaluated the performance impact of these optimizations on the CTR cache within the Intel SGX system integrated with MorphCtr, using a DFS benchmark when accessing CTRs after L1 cache misses (§5 details the methodology). We evaluate six cache optimization techniques. These include three prefetchers: Next-Line, Stride [11, 12], and Berti [42], a state-of-the-art prefetcher. We also evaluate three replacement policies: RRIP [21], SHiP [9], and Mockingjay [50], a state-of-the-art replacement policy. We configure RRIP and SHiP with reference prediction value (RRPV) parameters, setting an initial value of 2 and a maximum value of 3 for RRIP. Additionally, SHiP uses a 16,384 entry SHCT table with a maximum RRPV of 7. To model Mockingjay [50], we maintain a sampled cache with 4,096 entries that dynamically learns reuse distances. For each cache access, Mockingjay updates the estimated time of arrival (ETA) values and evicts the block with the highest ETA. Figure 5 shows that, compared to the baseline using only the LRU policy, neither advanced prefetching nor replacement policies significantly reduced the CTR cache miss rate or improved IPC.

Both the Next-line, Stride and Berti prefetchers show only marginal reductions in the CTR cache miss rate while simultaneously lowering overall IPC. This ineffectiveness can be attributed to two primary factors: (1) Poor locality of CTRs in applications with irregular memory access patterns, leading to low prefetcher accuracy. Specifically, the next-line prefetcher achieved a prefetch accuracy of just 1.02%, while the Stride prefetcher performed even worse, with an accuracy of 0.54% and Berti with an accuracy of 5.43%. (2) The need for integrity checks on CTR accesses from DRAM, regardless of prefetch accuracy. This means that incorrect prefetches still trigger unnecessary integrity checks, introducing additional overhead and negatively impacting performance.

The smart replacement policies, RRIP, SHiP and Mockingjay, also failed to improve performance, resulting in slightly lower CTR cache miss rates compared to the baseline. The primary reason for their ineffectiveness is the irregular access patterns of the application, which significantly reduce the accuracy of RRIP's prediction and hinder SHiP's ability to identify and retain CTRs that are likely to be re-referenced. Mockingjay [50] behaves similarly to SHiP, as shown in Figure 5. While Mockingjay observes longer access patterns, its decision-making is governed by fixed heuristics rather than adaptive learning. Unlike traditional data cache accesses, where reuse distance predictions are effective, CTR accesses in irregular memory applications target a deeper memory hierarchy, exhibiting weak locality and highly variable reuse behavior. As a result, Mockingjay's heuristic-based eviction strategy is less effective. Automated techniques are needed that can accurately predict and adapt to complex access patterns in diverse workloads.

3.4 Leveraging Reinforcement Learning

To optimize the performance of CTR accesses in secure memory systems, we need to address two critical tasks: (1) enabling early CTR access after L1 cache misses, and (2) enhancing CTR cache efficiency by leveraging CTR locality. We design two RL-based predictors to tackle these tasks: (1) predicting whether data require DRAM access after L1 misses, enabling early CTR fetching when necessary, and (2) predicting and leveraging CTR locality to populate the CTR cache with high-locality CTRs. We argue that RL is inherently suited for these tasks due to the following advantages.

Adaptivity. Both predictors must adapt to the evolving data access patterns of memory-intensive applications. RL provides a natural solution to this challenge by continuously learning and adjusting based on real-time feedback. Unlike traditional prefetching [11, 12] or cache replacement schemes [9, 21], which rely on static rules or assumptions rooted in human intuition about application behaviors, RL dynamically adapts to varying and rapidly shifting patterns. This adaptability ensures that both predictors function as autonomous agents, learning through continuous interaction with the memory system.

Online learning. Unlike offline learning-based approaches, an RL agent uses an online learning approach. Online learning enables the RL agent to continuously adapt its decision-making by iteratively optimizing its policy based on rewards received from the environment, eliminating the need for expensive offline training and fine-tuning. Online learning ensures adaptability while simultaneously reducing the overhead and complexity of pre-trained models.

Ease of Implementation. Prior works [14, 34, 52, 69] have explored sophisticated machine learning models for access pattern

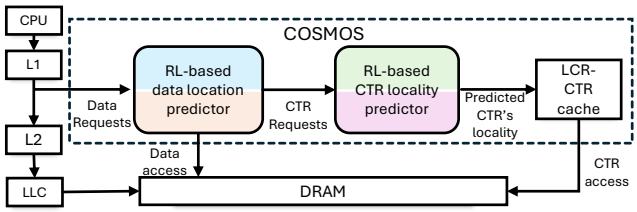


Figure 6: Overall Architecture of COSMOS.

recognition. While these models demonstrate promising accuracy, their large sizes often exceed on-chip storage limits, and their high computational demands result in unacceptable inference latencies for real-time systems. In contrast, RL can be efficiently implemented in hardware using lookup tables (Section 4), introducing minimal complexity and requiring only small hardware modifications, making it a practical choice for adoption in memory systems.

While perceptrons have been used in cache and memory optimizations [6, 22, 23, 27, 29, 61], RL offers unique advantages for access behavior prediction. First, RL models state transitions and capture access patterns over time, whereas perceptrons treat each access independently, missing temporal relationships. Second, RL optimizes cumulative reward by accounting for both immediate and delayed impacts of decisions, while perceptrons focus only on short-term, per-access classification. Finally, RL actively explores uncertain states, enabling robustness across changing workload phases. Perceptrons are deterministic and offer limited generalization under dynamic or irregular memory behaviors.

Based on the inherent qualities of RL, **our second key idea is to employ two separate RL predictors: one for data location prediction to enable early CTR access, and another for CTR locality prediction to optimize cache management. Each predictor is designed to solve a different optimization problem with its own well-defined state space, action space, and reward function.** By focusing on specific tasks, these RL-based predictors work autonomously while working collaboratively to improve the overall performance of secure memory systems.

4 COSMOS Design

Building on our key ideas from Section 3, we propose COSMOS, as illustrated in Figure 6. COSMOS implements these ideas through three core components: (1) a CTR locality predictor to identify CTRs with good locality, (2) a data location predictor to enable early CTR access by predicting data location after L1 misses, and (3) an LCR-CTR cache that optimizes CTR cache management based on the predicted CTR locality. In this section, we first define the RL primitives used in COSMOS and then detail the RL-based CTR locality predictor, the LCR-CTR cache, and the RL-based data location predictor. Finally, we discuss the hyperparameters chosen for the two predictors and analyze the hardware overhead of COSMOS.

4.1 RL Formulations for COSMOS

We formulate both CTR locality prediction and data location prediction as RL problems, each tailored to address a specific challenge in optimizing CTR accesses. RL predictors interact with the processor and the secure memory system, leveraging RL to dynamically adapt to memory access patterns.

4.1.1 CTR Locality Prediction RL Formulation. With each CTR access, the locality predictor observes the key memory access feature as the state to classify the locality of the CTR. Following each action, the predictor receives a reward based on the accuracy of its decision, allowing for continuous refinement and improvement in the precision of the classification. These CTR locality predictions provide valuable information for the LCR-CTR cache, which improves the overall CTR cache efficiency.

State. The state is defined by each CTR access. We construct the state space by hashing the physical address of the CTR to capture its locality characteristics. Specifically, bits 6 to 47 of the physical address, corresponding to the page number, are used as input for hashing. To achieve uniform state distribution, we employ a variant of the splitmix64 hashing function [63], leveraging prime multipliers to generate well-distributed state indices. This hashing approach creates a compact state representation that captures spatial locality patterns within a certain address range, whereas temporal locality is inherently captured through repeated accesses to the same hashed state.

Action. The action for the CTR locality prediction is to classify whether the accessed CTR has good locality or bad locality. Once the agent makes this determination, it tags the CTR's locality classification and sends this information to the LCR-CTR cache. CTRs identified as having good locality are prioritized and retained in the cache, while those with bad locality are deprioritized or marked for eviction.

Observable. The observable for the CTR locality prediction agent is derived from the CTR Evaluation Table (CET), a lightweight, LRU-managed buffer that tracks the CTR's state and access history over time. For each CTR access, its locality prediction (good or bad) and the current state are recorded in the CET. This observable data enables the evaluation of prediction accuracy and subsequent refinement of the RL policy.

Rewards. We define six rewards for the CTR locality predictor: R_{C_hg}, R_{C_hb}, R_{C_mb}, R_{C_mg}, R_{C_eg}, and R_{C_eb}. Rewards are assigned based on the CTR's observed behavior in the CET: **(1) CET hit (good locality):** A positive reward (R_{C_hg}) is given for correct good locality predictions, while a penalty (R_{C_hb}) is applied for incorrect bad locality predictions. **(2) CET miss (bad locality):** A positive reward (R_{C_mb}) is given for correct bad locality predictions, while a penalty (R_{C_mg}) is applied for incorrect good locality predictions. **(3) CET eviction (indicating bad locality):** A penalty (R_{C_eg}) is given for evicted entries that were misclassified as having good locality, while a reward (R_{C_eb}) is assigned for evicted entries with bad locality predictions.

4.1.2 Data Location Predictor RL Formulation. With every L1 cache miss, the data location predictor observes the key feature of the data access as the state to predict the location of the data block. For off-chip predictions, COSMOS speculatively accesses the CTR cache earlier, removing on-chip cache access latency from the critical path. After each prediction, the predictor receives a reward based on its accuracy, enabling continuous improvement in decision-making.

State. The state for the data location predictor is triggered by each L1 cache miss. Upon an L1 miss, the state space is constructed using the same address ranges and hashing mechanism detailed

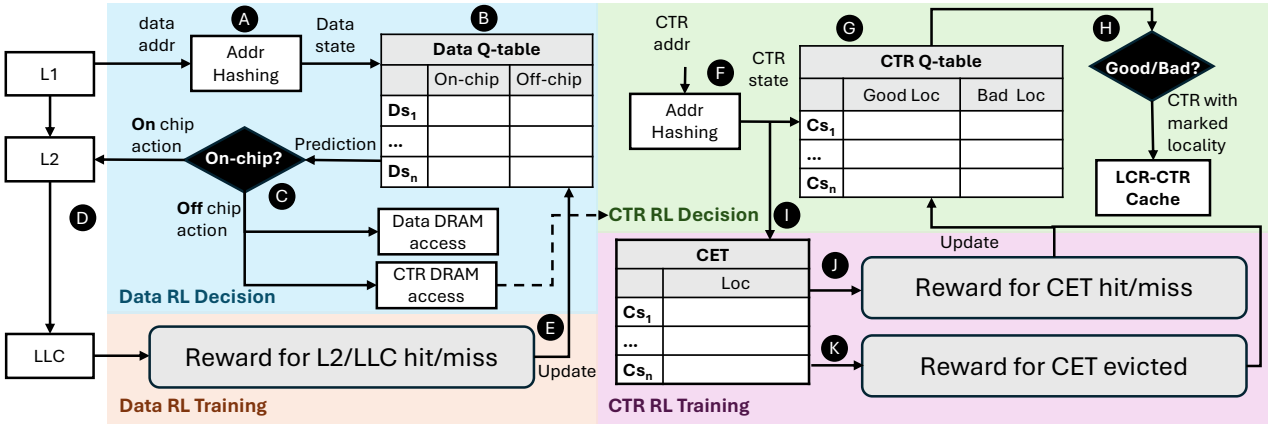


Figure 7: RL designs of COSMOS: (left) RL-based data location prediction; (right) RL-based CTR locality prediction

in the CTR locality predictor RL formulation, ensuring consistent capture of spatial and temporal characteristics.

Action. The action for the data location predictor is to determine whether the requested data block resides on-chip (in L2/L3 cache) or off-chip (in DRAM). Based on this classification, COSMOS performs a standard sequential L2, then L3 cache access for on-chip predictions or directly fetches the data from DRAM and retrieves the corresponding CTR from the LCR-CTR cache for off-chip predictions following an L1 miss.

Observable. Regardless of the data location prediction, COSMOS concurrently accesses the cache hierarchy for each data request to validate the prediction. The actual location of the data block—whether on-chip or off-chip—serves as feedback to evaluate prediction accuracy and dynamically refine the decision-making of the data location predictor.

Rewards. We define four rewards for the data location predictor: R_{D_hi}, R_{D_mo}, R_{D_ho}, and R_{D_mi}. Rewards are assigned to evaluate the correctness of the prediction by comparing it with the actual location of the data: **(1) Data is on-chip:** A positive reward (R_{D_hi}) is given when the predictor correctly classifies the data as on-chip. Conversely, a penalty (R_{D_ho}) is applied when the predictor incorrectly classifies the data as off-chip. **(2) Data is off-chip:** A positive reward (R_{D_mo}) is given when the predictor correctly classifies the data as off-chip. Conversely, a penalty (R_{D_mi}) is applied when the predictor incorrectly classifies the data as on-chip.

4.2 RL-based CTR Locality Predictor

The RL-based CTR locality predictor operates through two parallel processes: decision and training. These processes utilize two key hardware structures: a CTR Q-table for maintaining state-action Q-values and a CET, an LRU-managed buffer that evaluates the effectiveness of predictions over a defined temporal window. Figure 7 (right) shows the design, with Algorithm 1 providing details.

CTR RL Decision Process. Upon a CTR access, COSMOS creates a state representation by hashing specific bits of the physical address (Ⓕ) (Algorithm 1 line 2). Using the generated state, COSMOS queries the CTR Q-table to retrieve Q-values for all potential actions (Ⓖ) (Algorithm 1 line 3). The action with the highest Q-value is selected as the predicted locality of the CTR, with occasional random actions chosen to facilitate exploration and improve

Algorithm 1 COSMOS RL-based CTR Locality Prediction Algorithm

1: **procedure** COSMOS CTR PREDICTION(ctr_addr)
2: $S \leftarrow$ hash_address(ctr_addr)
3: $A \leftarrow \max_a Q(S,a)$ **or** random_action() **with** ϵ_C
4: **if** A **is good locality then**
5: Mark the CTR as good locality
6: **else**
7: Mark the CTR as bad locality
8: **end if**
9: $nearby\_S \leftarrow \{\text{hash\_address}(addr) \mid addr \in [ctr\_addr-32, ctr\_addr+32]\}$
10: $cet\_result \leftarrow$ Check CET for any state in $nearby\_S$
11: **if** cet_result **is hit then**
12: $R \leftarrow$ **if** A is good locality **then** R_{C_hg} **else** R_{C_hb}
13: **else**
14: $R \leftarrow$ **if** A is good locality **then** R_{C_mg} **else** R_{C_mb}
15: **end if**
16: $S2 \leftarrow$ CET.head.state; $A2 \leftarrow$ CET.head.action
17: **Q-table:** $Q(S,A) \leftarrow Q(S,A) + \alpha_C[R + \gamma_C \max_a Q(S2,A2) - Q(S,A)]$
18: Insert (S,A) into CET
19: **if** CET entry evicted **then**
20: $R \leftarrow$ **if** $evicted\_entry.A$ is good locality **then** R_{C_eg} **else** R_{C_eb}
21: $s \leftarrow$ evicted_entry.state; $a \leftarrow$ evicted_entry.action
22: **Q-table:** $Q(s,a) \leftarrow Q(s,a) + \alpha_C[R + \gamma_C \max_a Q(S2,A2) - Q(s,a)]$
23: **end if**
24: **end procedure**

long-term predictions (Algorithm 1 lines 4-8). The predicted locality is tagged and passed to the LCR-CTR cache for further cache management (Ⓗ).

CTR RL Training Process. COSMOS leverages the CET to continuously evaluate and refine CTR locality predictions (Ⓘ). For each new CTR request, COSMOS checks if the hashed address of the request or its nearby CTRs (within an address range of ±32) matches an entry in the CET (Algorithm 1 line 9). A match (CTR hit) indicates good locality, suggesting that the CTR or a nearby address with strong spatial locality has been accessed again within a defined temporal window. For such hits, COSMOS assigns a reward R_{C_hg} for correct good locality predictions or a penalty R_{C_hb} for incorrect bad locality predictions (Algorithm 1 line 12). Conversely, CET misses indicate poor locality. For such misses, COSMOS assigns a reward R_{C_mb} for correct bad locality predictions or a penalty R_{C_mg} for incorrect good locality predictions (Ⓙ) (Algorithm 1 line 14) . Once the reward is determined, the state, action, and reward of the current CTR request are used to update the CTR Q-table directly. Subsequently, the corresponding state-action pair is recorded in the CET.

When a CET entry is evicted, it indicates that neither the evicted CTR nor its nearby CTRs have been accessed for an extended period

Algorithm 2 LCR-CTR Cache Replacement Policy

```
1: procedure LCR-CTR Replacement(set)
2:     evict_candidate ← None
3:     max_bad_score ← −1
4:     min_good_score ← ∞
5:     for all cache_line in set do                    ▷ Only within the specific set
6:         if cache_line.locality_flag = 0 then        ▷ Bad locality
7:             if cache_line.locality_score >max_bad_score then
8:                 evict_candidate ← cache_line
9:                 max_bad_score ← cache_line.locality_score
10:            end if
11:        else                                        ▷ Good locality
12:            if evict_candidate = None then
13:                if cache_line.locality_score <min_good_score then
14:                    evict_candidate ← cache_line
15:                    min_good_score ← cache_line.locality_score
16:                end if
17:            end if
18:        end if
19:    end for
20:    evict evict_candidate
21: end procedure
```

Algorithm 3 RL-based Data Location Prediction Algorithm

```
1: procedure COSMOS Data Prediction(addr)
2:     S ← hash_address(addr)
3:     A ← max_a Q(S, a) or random_action() with ε_D
4:     Perform normal L2 and LLC cache access
5:     if A is off-chip then
6:         Initiate DRAM fetch and CTR access
7:     end if
8:     if L2 or LLC hit then
9:         R ← if A is on-chip then R_D_hi else R_D_ho
10:        if R = R_D_ho then
11:            Kill current data DRAM process and continues the CTR access
12:        end if
13:    else
14:        R ← if A is on-chip then R_D_mi else R_D_mo
15:        if R = R_D_mi then
16:            Perform DRAM fetch and CTR access
17:        end if
18:    end if
19:    a ← if (L2 or LLC hit) then ON_CHIP else OFF_CHIP
20:    Q-table: Q(S, A) ← Q(S, A) + α_D[R + γ_D max_a Q(S, a) − Q(S, A)]
21: end procedure
```

(Algorithm 1 line 19). In such cases, COSMOS assigns final rewards to the corresponding evicted state-action pair: R_{C_eb} for correct bad locality predictions or R_{C_eg} for incorrect good locality predictions (K) (Algorithm 1 line 20). This process further refines the CTR Q-table .

4.3 LCR-CTR Cache

The goal of the LCR-CTR cache is to ensure that CTRs predicted to have good locality remain in the cache for the longest duration, thereby increasing the likelihood of CTR cache hits. Each cache line in the LCR-CTR cache includes a 1-bit flag indicating the predicted locality: a value of 1 denotes good locality, while 0 denotes bad locality. Additionally, each line contains an 8-bit score representing the corresponding locality score derived from the CTR Q-table.

Algorithm 2 outlines the detailed replacement policy in LCR-CTR cache. The replacement policy is designed to prioritize CTR blocks with good locality within each cache set, ensuring optimal cache utilization. Within a specific set, the primary eviction targets are blocks predicted to have bad locality (indicated by a 0 in the 1-bit flag). Among these, the LCR-CTR cache first evicts the CTR with the highest bad locality score (Algorithm 2 lines 5-10). If all blocks in the set are marked as having good locality, LCR-CTR cache evicts the block with the lowest good locality score (Algorithm 2 lines 12-16). This hierarchical approach allows the LCR-CTR cache to retain high-scoring good locality CTRs for longer periods within each set, thereby maximizing cache utilization and improving overall performance.

4.4 RL-based Data Location Predictor

COSMOS employs an RL-based data location predictor to enable early CTR access by determining whether the requested data resides on-chip or in DRAM. The predictor utilizes a data Q-table to store state-prediction pairs and operates through parallel decision and training processes, as illustrated in Figure 7 (left) and detailed in Algorithm 3.

Data RL Decision. For decision-making, COSMOS hashes the data's physical address to create a state (A) (Algorithm 3 line 2), queries the Q-table (B) (Algorithm 3 line 3), and performs an action based on the prediction. For on-chip predictions, COSMOS executes standard cache accesses, whereas off-chip predictions trigger immediate DRAM data accesses and LCR-CTR retrievals (C), conducted concurrently with cache hierarchy checks (Algorithm 3 lines 4–6).

Data RL Training. Running parallel to the decision process, the training process updates the Q-table based on prediction accuracy. After each decision, COSMOS checks the actual location of the data block by concurrently accessing the cache hierarchy (D) and compares it with the initial prediction. Rewards are subsequently assigned based on this comparison (E) (Algorithm 3 line 2 8-18). Correct predictions receive positive rewards (R_{D_hi} and R_{D_mo}), while incorrect predictions incur penalties (R_{D_ho} and R_{D_mi}) (Algorithm 3 line 12, 14). The data Q-table is updated immediately after the reward for the current prediction is determined, using the Q-learning rule (Algorithm 3 line 20).

COSMOS manages data and CTR accesses based on prediction outcomes: (1) For correct on-chip predictions, data follows standard cache hierarchy access (Algorithm 3 line 4); (2) For incorrect on-chip predictions, after cache misses, COSMOS falls back to CTR cache and DRAM access (Algorithm 3 line 16); (3) For correct off-chip predictions, COSMOS bypasses the cache hierarchy for direct DRAM and CTR cache access (Algorithm 3 line 6); and (4) For incorrect off-chip predictions, COSMOS halts DRAM access but maintains CTR cache access to exploit potential locality (Algorithm 3 line 11). By integrating data location prediction with efficient management of data and CTR accesses, COSMOS enhances overall performance without incurring potential losses.

4.5 Hyperparameter and Reward Value Selection

COSMOS utilizes the ϵ-greedy [57] method to balance exploration and exploitation in both data and CTR locality prediction. With probability ϵ, it explores through random actions; otherwise, it exploits the learned policy by selecting the highest Q-value action.

Selecting hyperparameters involves defining reasonable ranges, performing initial tests with various combinations, and evaluating performance metrics to identify optimal values. The learning rate (α_D, α_C), the discount factor (γ_D, γ_C), and the exploration rate (ϵ_D, ϵ_C) -with α_D, γ_D, ϵ_D originating from the RL-based data location predictor (Algorithm 3 line 20), and α_C, γ_C, ϵ_C from the RL-based

Table 1: Reward values and hyperparameters.

Reward Values
$R_{D\_mo} = 12, R_{D\_mi} = -30, R_{D\_ho} = -20, R_{D\_hi} = 9$
$R_{C\_hg} = 13, R_{C\_hb} = -12, R_{C\_mg} = -16, R_{C\_mb} = 20$
$R_{C\_eg} = -22, R_{C\_eb} = 26$
Hyper-parameters
$\alpha_D = 0.09, \gamma_D = 0.88, \epsilon_D = 0.1$
$\alpha_C = 0.05, \gamma_C = 0.35, \epsilon_C = 0.001$

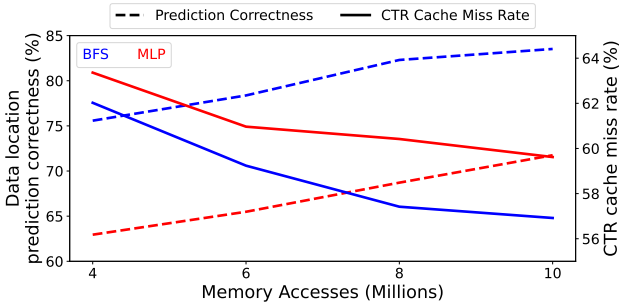


Figure 8: Data Location Prediction correctness (left y-axis) and CTR cache miss rate (right y-axis) for BFS (graph-based) and MLP (non-graph) workloads as memory accesses increase.

CTR locality predictor (Algorithm 1 line 22) significantly influence COSMOS's learning efficiency and prediction accuracy. We define the hyperparameter ranges as $\alpha_D, \gamma_D, \alpha_C, \gamma_C \in [10^{-3}, 1]$ and $\epsilon_D, \epsilon_C \in [0, 1]$, as these values are commonly used in the selection of hyperparameters in RL [30]. To perform fast evaluations, we implement the RL-based data location predictor and the RL-based CTR locality predictor in Python, using Pintool [19] to capture the GraphBIG DFS memory footprint as input. We test 1,000 hyperparameter combinations and select the optimal combination based on the maximum LCR-CTR cache hit rate after data location and CTR locality RL prediction, using fixed reward scores (positive rewards = 10 and negative rewards = -10).

With the best hyperparameter combination ($\alpha_D = 0.09$, $\gamma_D = 0.88$, $\epsilon_D = 0.1$, $\alpha_C = 0.05$, $\gamma_C = 0.35$, $\epsilon_C = 0.001$), we then tested 1000 reward combinations. Positive rewards (R_{C_hg}, R_{C_mb}, R_{C_eb}, R_{D_hi}, R_{D_mo}) ranged from 0 to 100, while negative rewards (R_{C_hb}, R_{C_mg}, R_{C_eg}, R_{D_ho}, R_{D_mi}) ranged from -100 to -1.

Table 1 presents the final chosen values. COSMOS requires only one-time tuning for a specific application domain, meaning that once tuned for irregular memory access workloads (e.g., graph processing), it generalizes well within that category. However, for workloads with different memory behaviors (e.g., machine learning), re-tuning is necessary to optimize performance, as data location and CTR locality predictions depend on access patterns. Since this work focuses on irregular memory applications, we tune COSMOS once using the GraphBIG DFS benchmark, ensuring optimal performance within this category.

To demonstrate the generalization capability of COSMOS, we evaluate the adaptability of its RL-based predictors using two distinct workloads: BFS, a graph-based workload similar to DFS, and a 3-layer MLP, a nongraph workload. Neither workload was used during RL hyperparameter tuning. We apply the same α, γ, and ϵ values tuned in DFS and evaluate both workloads over a range of total memory accesses from 4 million to 10 million. Figure 8 shows the prediction correctness of the data location predictor and the CTR cache miss rate. The reduction in the CTR cache miss rate reflects the effectiveness of COSMOS, which integrates cooperation between the data location predictor and the CTR locality predictor. As shown in Figure 8, the data location predictor quickly converges on BFS, reaching 83% prediction correctness, while COSMOS achieves a CTR cache miss rate of 60% at 10 million accesses. In contrast, MLP starts with lower prediction correctness because its access pattern was not seen during RL hyperparameter tuning. Nevertheless, the correctness of the data location predictor continues to increase, exceeding 70% after 10 million accesses, demonstrating that the RL agent can gradually adapt through online learning.

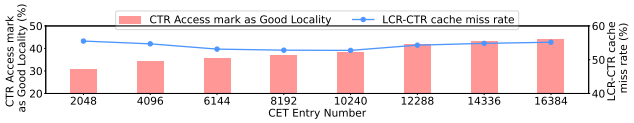


Figure 9: Comparison of CET entry number vs. percentage of CTR access marked as good locality by RL in COSMOS (left y-axis) and LCR-CTR cache miss rate of DFS (right y-axis).

Table 2: Storage overhead of COSMOS.

Component	Details	Overhead
Data Q-Table	16384 entries; 16 bits/entry	32KB
CTR Q-Table	16384 entries; 16 bits/entry	32KB
CET	8192 entries; 65 bits/entry (64 bits address, 1 bit prediction)	66KB
LCR-CTR cache	Extra 9 bits/cache line (8 bits reward, 1 bit prediction)	17KB
Total		**147KB**

4.6 Hardware Overhead of COSMOS

Table 2 summarizes the storage overhead for COSMOS, totaling 147KB, which represents only 1. 84% of an LLC capacity of 8MB. This overhead is distributed across several components. The Data Q-Table and the CTR Q-Table each contain 16,384 entries corresponding to 16,384 states for data and CTR blocks, respectively. We chose this size because 16,384 states are sufficient for 32GB memory addresses, and the 8 bit Q score is enough for binary predictions [35]. Both tables store two 8-bit Q-values for binary predictions (data on-chip/off-chip and good/bad CTR locality). The LCR-CTR cache requires an additional 9 bits per CTR cache line (1 bit for prediction and 8 bits for the corresponding Q values).

We conduct a design space exploration to analyze the relationship between CET size and LCR-CTR cache miss rate of DFS. Figure 9 shows CET entry numbers versus CTR good locality percentage (left y-axis) and LCR-CTR cache miss rate (right y-axis). As CET size grows, it classifies more CTR accesses as good locality. The LCR-CTR cache miss rate initially decreases with increasing CET size but begins to rise when the CET becomes very large. This is because a large CET will tend to allow the CTR locality prediction scheme to recognize an excess of CTRs as good locality, thus undermining the CTR cache replacement decisions. When CET

Table 3: Simulation settings

Processor Parameters	
Core	4 Cores, X86, OoO, 3GHz
L1 Cache	2 cycles, 32KB, 2-Way
L2 Cache	20 Cycles, 1MB, 8-Way
LLC	128 Cycles, 8MB, 16-Way
Memory Parameters	
Type	DDR4_2400_16x4
Size	32 GB
AES-CTR[13] Parameters	
AES Latency	128-bit, 40 Cycles
Authentication latency	40 Cycles
MAC	64 bits each
CTR cache	LRU, 512 KB
Morphctr[46] Parameters	
CTR combination	1 cycle
Re-encryption Latency	extra 64B DRAM request
COSMOS Parameters	
LCR-CTR cache	128 KB

entries equal 8,192, it shows a low LCR-CTR cache miss rate. As CET entries increase to 10,240, the reduction in miss rate becomes minimal. To balance the size and LCR-CTR cache performance, we chose the optimal configuration to be a CET with 8,192 entries, each containing a 64-bit CTR state value and a 1-bit prediction value.

We estimate the power and area of COSMOS components using SRAM macros generated by a commercial 28 nm memory compiler under standard industrial conditions (0.9 V, 25°C, 3 GHz). This compiler is derived from an industry-standard process design kit (PDK) and is widely used in commercial chip development, ensuring realistic modeling of SRAM structures. The Data Q-table and CTR Q-table each occupy an area of 0.057 mm^2, with a power consumption of 45.29 mW. The CET structure requires 0.116 mm^2 of area and consumes 92.00 mW of power. The LCR-CTR cache contributes an additional 0.030 mm^2 of area and 24.06 mW of power. In total, COSMOS introduces 0.260 mm^2 of area overhead and consumes 206.65 mW of additional power.

Regarding performance impact, the Q-table access is estimated to take 1 cycle (0.024ns) in a 3GHz system. Upon an L1 miss, the Data Q-table access and address hashing occurs in parallel with other data accessing functions (L2, LLC access), thus not affecting the critical path.The CTR Q-table access and CTR address hashing can be performed in parallel with LCR-CTR cache and CTR DRAM access, ensuring that their overhead does not impact the critical path of CTR access. CET access and address hashing can be performed in parallel with CTR access, so in our evaluation we do not consider the CET access and hashing latency.

5 EVALUATION METHODOLOGY

We evaluate COSMOS using Gem5[8], a cycle-accurate simulator, with the graph-based benchmarks discussed in Section 3 under multi-threading. In addition to graph workloads, we also evaluate the SPEC benchmarks mcf and canneal from SPEC CPU2006 [16] and omnetpp from SPEC CPU2017 [54], which are known for their low locality and irregular memory access patterns. The SPEC benchmarks are also executed using 4 threads. We used Gem5's Syscall Emulation (SE) mode to precisely measure DRAM traffic, cache hit rates, and MC overhead.

Table 4: COSMOS Design Variations

Design	Description
COSMOS-DP	Data predictor only
COSMOS-CP	CTR predictor + LCR-CTR cache
COSMOS	Full RL implementation

The simulation setup is detailed in Table 3, assumes a 4-core, 4-thread X86 architecture with 32KB L1 cache (per core), 1MB L2 cache (per core), 8MB shared LLC cache, and a 512KB CTR cache (per core), matching the on-chip storage cost of COSMOS. We model an AES-CTR-based secure memory system with a 64-bit MAC per 64B cache line, which requires one MAC access per eight data accesses for authentication [13]. Both MAC authentication and AES encryption/decryption incur a 40-cycle latency, as in [18, 62, 65, 66], with the CTR cache and AES operations integrated into the MC.

We use the MorphCtr design [46] as a baseline, with COSMOS applied to MorphCtr as an enhancement. [1] MorphCtr's CTR blocks contain minor CTRs and two major CTRs combined to form the actual CTR for AES processing. We assume a 1-cycle CTR combination latency in both MorphCtr and COSMOS. CTR integrity checks require accessing the MT nodes for each CTR DRAM access. The CTR integrity checking process runs in parallel with the OTP encryption/decryption, so we do not model this overhead explicitly.

For CTR overflow handling, COSMOS follows MorphCtr's approach, triggering re-encryption after 67 accesses to the same CTR location [46]. Like EMCC [65] and RMCC [66], overflows generate 64B requests processed in the background by the MC using dedicated queue slots. This parallel processing allows COSMOS's predictors and cache management to operate independently of re-encryption. Re-encryption events are rare in our target applications; according to our experiments, among 1 million memory writes, there are only 1000 CTR overflows.

As discussed in Section 4, the lookup process in the RL predictor is performed off the critical path, ensuring that no additional overhead is introduced. In our implementation, COSMOS includes a 128KB LCR-CTR cache. We intentionally choose a relatively small LCR-CTR cache because COSMOS introduces 147KB of additional storage in the memory controller compared to other designs [65, 66], ensuring a fair comparison.

For a complete evaluation of COSMOS we evaluated three variations of COSMOS (Table 4): COSMOS-DP with only the RL-based data location predictor, COSMOS-CP with only RL-based CTR locality predictor and LCR-CTR cache, and full COSMOS combining both components. This ablation study isolates the impact of each predictor while demonstrating their synergistic benefits.

6 Evaluation Results

Figure 10 illustrates the performance of MorphCtr (our baseline), COSMOS-DP, COSMOS-CP, and the full COSMOS implementation, with all results normalized to a non-protected (NP) memory system. We observe that COSMOS-DP delivers an 15% average performance

[1] As a flexible CTR cache optimizer, our solution can work with various designs. In this article, we choose MorphCtr as the baseline, although COSMOS could also be applied to other designs, such as MT tree optimizations in Synergy[48] and ITESP[59].

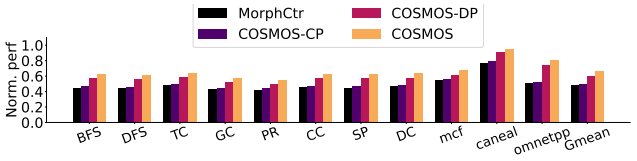


Figure 10: Performance of COSMOS-DP, COSMOS-CP, COSMOS and Morphctr, normalized to a non-protected memory system.

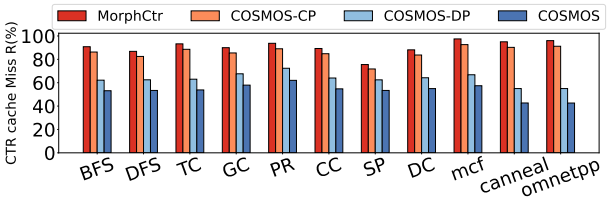


Figure 11: CTR cache miss rate of COSMOS-DP, COSMOS-CP, COSMOS and MorphCtr across different graph algorithms.

improvement over baseline, while COSMOS-CP achieves an 5% improvement. The complete COSMOS implementation, which combines both data location and CTR locality predictions, achieves an average performance gain of 25%. The average residual performance overhead compared to the NP system remains approximately 33%. This performance gap is primarily due to the remaining CTR cache misses, which trigger additional DRAM accesses for CTR integrity verification, including MT node retrievals.

In the following subsections, we explore how each component of COSMOS contributes to reducing CTR cache miss rates and improving overall system performance.

6.1 Performance Analysis

6.1.1 CTR Cache Performance. COSMOS's performance gains stem primarily from its ability to reduce CTR cache miss rates—a key contributor to secure memory overhead. Figure 11 presents the CTR cache miss rates in different variants of COSMOS. COSMOS-CP shows limited improvement, as it accesses CTRs only after LLC misses, causing the CTR cache to behave like an "L4" cache , where most high-locality data have already been filtered out. As a result, the CTR cache captures few reusable counters, limiting its effectiveness. In contrast, COSMOS-DP and the full COSMOS design access CTRs immediately after L1 misses, allowing high-locality CTRs to be inserted into the cache earlier. This early access significantly increases cache hit rates and overall performance.

6.1.2 Effectiveness of RL. We evaluate the effectiveness of the two RL predictors used in COSMOS. To assess the impact of the RL-based CTR locality predictor, we examine the percentage of CTR accesses classified as having good locality, as shown in Figure 13. For CTR accesses triggered after LLC misses, only about 5% are identified as having good locality. This low percentage limits the effectiveness of the LCR-CTR cache, which is designed to retain CTRs marked as high locality while evicting those deemed less useful.

For the RL-based data location predictor, Figure 12 shows that the prediction accuracy averages around 85% across all evaluated

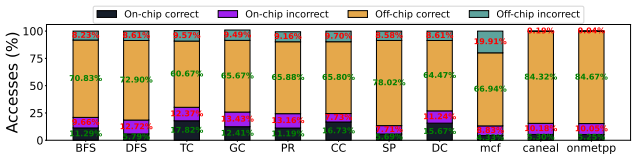


Figure 12: Data access prediction distribution and accuracy in COSMOS's RL-based data location prediction across different algorithms. Green text indicates correct prediction portions.

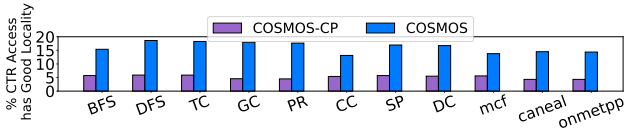


Figure 13: Comparison of percentage of CTR accesses considered as good locality by RL in COSMOS and COSMOS-CP across different algorithms.

benchmarks. This includes both correctly predicted on-chip and off-chip accesses. This level of precision is particularly impressive given the irregular memory access patterns in our target applications, where traditional pattern-based predictors typically achieve less than 60% precision. The RL predictor adapts quickly to changing access patterns, reaching stable prediction accuracy within the first 10 total million memory accesses. Accurate off-chip predictions help bypass unnecessary L2 and LLC accesses, thus reducing average memory latency.

Additionally, about 12% of off-chip predictions are misclassified—that is, they actually have good locality and should have been kept on-chip. However, these misclassified accesses are still directed to the CTR cache. This unintentionally benefits the system by allowing more high-locality CTRs to enter the cache, improving its utilization and effectiveness. Our analysis suggests that approximately 30% of the overall improvement in the CTR cache hit rate comes from this beneficial side effect of misclassification.

In the full COSMOS design, the combination of the two RL predictors leads to better CTR cache utilization. As shown in Figure 13, approximately 20% of the CTR accesses are identified to have good locality. Compared to COSMOS-DP, the full COSMOS design reduces the CTR cache miss rate by 14 percentage points (Figure 11). This improvement demonstrates the synergistic value of combining data location prediction with accurate locality prediction: the former enables early access to the CTR, while the latter ensures efficient utilization of the CTR cache space by identifying and prioritizing the CTRs that will be reused.

6.1.3 Secure Memory Access Time. The major benefit of using COSMOS in secure memory lies in its ability to reduce the overall memory traffic. To quantify this impact, we define Secure Memory Access Time (SMAT), which reflects the average latency per access in secure memory, combining traditional memory hierarchy access latencies with additional overhead specific to secure memory.

$$\text{SMAT} = L1 + MR_{L1}(L2 + MR_{L2}(LLC + MR_{LLC}(\text{CTR} + DRAM))) \quad (1)$$

$$\text{CTR} = \text{CTR}_{\text{hit}} + MR_{\text{CTR}}(\text{CTR}_{\text{DRAM}} + \text{CTR}_{\text{verify}}) \quad (2)$$

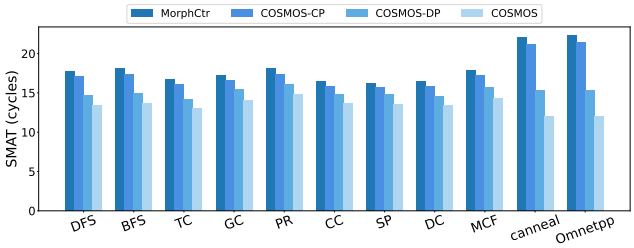


Figure 14: Secure Memory Access Time (SMAT) across different secure memory designs: MorphCtr, COSMOS-CP, COSMOS-DP, and COSMOS.

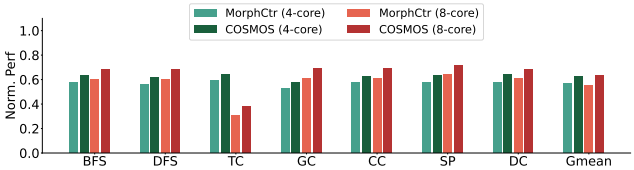


Figure 15: Performance comparison of COSMOS and MorphCtr, normalized to a non-protected memory system, in multi-core systems across various graph-based workloads.

Here, *MR* represents the miss rate at each cache level. The CTR term captures the secure memory overhead, where a CTR cache miss results in both CTR DRAM access and CTR verification.

Figure 14 presents the SMAT across all evaluated applications for MorphCtr, COSMOS-CP, COSMOS-DP, and the full COSMOS design. COSMOS achieves the lowest SMAT among all configurations, with improvements from two key mechanisms. First, it reduces the CTR cache miss rate, thus reducing expensive CTR DRAM accesses and MT verifications. Second, thanks to the RL-based data location predictor, approximately 70% of the memory accesses bypass the L2 and LLC caches and go directly from L1 to the CTR cache. This reduces unnecessary cache lookups and reduces the overall access latency.

Finally, to evaluate the scalability of COSMOS, we extend our analysis beyond the default 4-core configuration by incorporating an 8-core system with a 16 MB shared LLC. The evaluation spans seven representative graph-based workloads, including BFS, DFS, TC, GC, CC, SP, and DC. As shown in Figure 15, COSMOS consistently outperforms MorphCtr in both configurations, achieving a 26% performance gain in the 8-core system (compared to 25% in the 4-core setup), demonstrating that COSMOS maintains consistent performance benefits as the system scales.

6.2 Comparison with State-of-the-Art Designs

We compare COSMOS with EMCC [65], a state-of-the-art design that integrates CTR caching and AES/MAC logic into the L2 cache controller. Its key idea is to overlap CTR decryption and data access by embedding CTR management within the L2 pipeline. We implement an EMCC-like system in Gem5 that places the CTR cache at the L2 cache level. In this implementation, data access of L2, LLC, and DRAM occurs in parallel with CTR access in L2. To focus on evaluating the performance achieved by the core design of EMCC, we follow the original flow [65] while excluding additional overheads, such as the extra latency from AES computation in L2, L2 CTR cache access latency, and NoC latency between L2 and

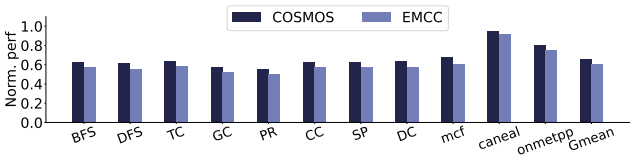


Figure 16: Performance of COSMOS and EMCC normalized to a non-protected memory system.

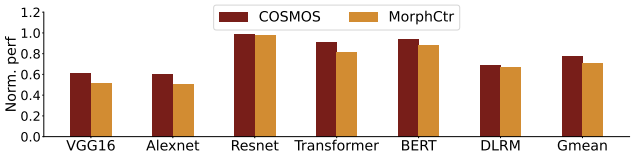


Figure 17: Performance comparison of COSMOS and MorphCtr, normalized to a non-protected memory system, in a 4-core configuration running machine learning workloads.

LLC. Figure 16 shows that EMCC achieves an average 12% performance improvement over MorphCtr. COSMOS provides a 10% performance gain over our ideal EMCC implementation. This improvement stems from the dual RL-based specialized predictors in COSMOS, which collaboratively reduce CTR cache misses and mitigate memory access latency without modifying cache controllers.

RMCC [66], another recently proposed secure memory optimization, retains frequently accessed counters near the memory controller to reduce CTR-related traffic. It reports performance gains comparable to EMCC over MorphCtr. Since COSMOS achieves an additional 10% improvement over the ideal implementation of EMCC, similar benefits are expected over RMCC. COSMOS provides stronger performance potential than RMCC, as it enables earlier CTR access through data location prediction, while RMCC applies remapping only after LLC misses. Furthermore, utilizing RL provides greater adaptability for COSMOS compared to the memoization-aware scheme employed by RMCC.

6.3 Workload on Regular Memory Access Pattern

Finally, we evaluate COSMOS on workloads with regular memory access patterns to assess whether it negatively impacts performance in such scenarios. We parallelize inference across four threads for six representative models: AlexNet [31], ResNet [15], VGG [53], BERT [10], Transformer [40], and DLRM [40].

For vision models (AlexNet, ResNet, VGG), we process 224×224×3 images, parallelizing convolutional output channels and fully connected neurons. Language models (BERT, Transformer) operate on sequences of length 128 with 768-dimensional embeddings, parallelizing self-attention and feed-forward layers. The recommendation model DLRM processes 13 dense features and multiple categorical embeddings, parallelizing embedding lookups and fully connected operations.

We use the same COSMOS hyperparameters tuned for irregular graph workloads, without re-tuning for these regular access patterns. This setup allows us to evaluate the generalizability of COSMOS rather than optimize for peak performance.

As shown in Figure 17, COSMOS yields only modest gains (~3%) over MorphCtr. This is due to three key factors: (1) suboptimal hyperparameters for these workloads reduce prediction accuracy;

(2) regular memory access patterns already achieve high cache hit rates, diminishing the benefit of CTR cache optimizations; and (3) re-encryption becomes a major bottleneck, since the same CTRs are repeatedly accessed, over 50% of accesses trigger re-encryption. Because COSMOS focuses on CTR cache efficiency rather than re-encryption handling, its impact is limited here.

However, this experiment confirms that COSMOS does not harm performance in regular workloads, demonstrating robustness and general applicability without introducing regressions.

7 Related Work

Previous memory encryption research explored countermode encryption and integrity trees [13, 48, 55, 59], with challenges in off-chip CTR storage. MGX [18] and SoftVN [62] introduced novel CTR generation methods. Split CTR [68] and Morphctr [46] improved storage efficiency through shared counters and adaptive representations. Although EMCC [65], RMCC [66], and Counter-light [64] built on MorphCtr, they fall short of COSMOS's performance.

ML-based schemes are designed to optimize cache performance and reduce cache errors using advanced prediction techniques. Hermes [6] employs a perceptron-based predictor for load miss predictions. TCP [23] uses similar predictors for off-chip access and prefetch filtering. Pythia [7] leverages online RL for data access-aware prefetching. CHROME [35] integrates cache replacement, bypass, and prefetching via RL online. However, none of these designs addresses the unique challenges of secure memory systems.

8 Conclusion

We introduced COSMOS, a novel approach to optimize secure memory systems that leverages RL for data location and CTR locality prediction. By enabling earlier CTR access in the cache hierarchy and better using CTR locality, COSMOS reduces CTR cache miss rates and overall memory access latency. With minimal hardware modifications, COSMOS achieves a 25% performance gain over MorphCtr, outperforming EMCC by 10% in applications with irregular memory access.

Acknowledgments

The authors sincerely thank the anonymous reviewers for their valuable comments and feedback. We are also grateful to members of the Hardware-Software Codesign Lab at the University of Notre Dame, the Gnosis Research Center at Illinois Insitute of Technology and the ICSLAB at Wuhan University for their insightful suggestions and experimental support throughout this work.

This research was supported in part by SUPREME, one of the six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Additional support was provided by the National Science Foundation (NSF) under grants CNS-2152497 and CNS-2310422. Dr. Dazhao Cheng was supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62341410. Dr. Yuezhi Che was supported by the NSFC under Grant No. 62402346.

References

[1] Advanced Micro Devices, Inc. 2020. *AMD Secure Encrypted Virtualization API Version 0.24.* Technical Report Publication #55766. AMD Corporation. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/programmer-references/55766_SEV-KM_API_Specification.pdf

[2] Advanced Micro Devices, Inc. 2020. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More.* Technical Report. AMD Corporation. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

[3] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2012. Revisiting hardware-assisted page walks for virtualized systems. *ACM SIGARCH Computer Architecture News* 40, 3 (2012), 476–487.

[4] AMD. 2023. AMD Memory Encryption White Paper. https://www.amd.com/system/files/TechDocs/memory-encryption-white-paper.pdf

[5] Thomas W Barr, Alan L Cox, and Scott Rixner. 2010. Translation caching: skip, don't walk (the page table). *ACM SIGARCH Computer Architecture News* 38, 3 (2010), 48–59.

[6] Rahul Bera, Konstantinos Kanellopoulos, Shankar Balachandran, David Novo, Ataberk Olgun, Mohammad Sadrosadat, and Onur Mutlu. 2022. Hermes: Accelerating long-latency load requests via perceptron-based off-chip load prediction. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–18.

[7] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1121–1137.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[9] Zehan Cui, Aamer Jaleel, Simon C. Steely, and Joel S. Emer. 2011. SHiP: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture (MICRO)*. IEEE/ACM, 430–441.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[11] John WC Fu and Janak H Patel. 1991. Data prefetching in multiprocessor vector cache memories. *ACM SIGARCH Computer Architecture News* 19, 3 (1991), 54–63.

[12] John WC Fu, Janak H Patel, and Bob L Janssens. 1992. Stride directed prefetching in scalar processors. *ACM SIGMICRO Newsletter* 23, 1-2 (1992), 102–110.

[13] Shay Gueron. 2016. Memory encryption for general-purpose processors. *IEEE Security & Privacy* 14, 6 (2016), 54–62.

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

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[16] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[17] Michael Henson and Stephen Taylor. 2014. Memory encryption: A survey of existing techniques. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 1–26.

[18] Weizhe Hua, Muhammad Umar, Zhiru Zhang, and G Edward Suh. 2022. MGX: Near-zero overhead memory protection for data-intensive accelerators. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 726–741.

[19] Intel Corporation. 2020. Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool. Accessed: 2024-06-17.

[20] Engin Ipek, Onur Mutlu, José F Martínez, and Rich Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. *ACM SIGARCH Computer Architecture News* 36, 3 (2008), 39–50.

[21] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel S. Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th annual international symposium on Computer architecture (ISCA)*. ACM, 60–71.

[22] Majid Jalili and Mattan Erez. 2022. Reducing load latency with cache level prediction. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 648–661.

[23] Alexandre Valentin Jamet, Georgios Vavouliotis, Daniel A Jiménez, Lluc Alvarez, and Marc Casas. 2024. A Two Level Neural Approach Combining Off-Chip Prediction with Adaptive Prefetch Filtering. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 528–542.

[24] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, and Jong Wook Kim. 2019. Q-learning algorithms: A comprehensive classification and applications. *IEEE access* 7 (2019), 133653–133667.

[25] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2017. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 377–390.

[26] Daniel A Jiménez and Elvira Teran. 2017. Multiperspective reuse prediction. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 436–448.

[27] Daniel A Jiménez and Elvira Teran. 2017. Multiperspective reuse prediction. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 436–448.

[28] Gokul B Kandiraju and Anand Sivasubramaniam. 2002. Going the distance for TLB prefetching: An application-driven study. *ACM SIGARCH Computer Architecture News* 30, 2 (2002), 195–206.

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

[30] Mariam Kiran and Melis Ozyildirim. 2022. Hyperparameter tuning for deep reinforcement learning applications. *arXiv preprint arXiv:2201.11182* (2022).

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).

[32] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. {CIPHERLEAKS}: Breaking Constant-time Cryptography on {AMD}{SEV} via the Ciphertext Side Channel. In *30th USENIX Security Symposium (USENIX Security 21)*. 717–732.

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

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

[35] Xiaoyang Lu, Hamed Najafi, Jason Liu, and Xian-He Sun. 2024. CHROME: Concurrency-aware holistic cache management framework with online reinforcement learning. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1154–1167.

[36] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. 2013. TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 1 (2013), 1–38.

[37] Luther Martin. 2010. XTS: A mode of AES for encrypting hard disks. *IEEE Security & Privacy* 8, 3 (2010), 68–69.

[38] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[39] Lifeng Nai, Yinglong Xia, Ilie G Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[40] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).

[41] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. 2002. Practical, transparent operating system support for superpages. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 89–104.

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

[43] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. 2007. Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 183–196.

[44] Benedek Rozemberczki, Carl Allen, and Rik Sarkar. 2019. Multi-scale Attributed Node Embedding. arXiv:1909.13021 [cs.LG]

[45] Gavin A Rummery and Mahesan Niranjan. 1994. *On-line Q-learning using connectionist systems*. Vol. 37. University of Cambridge, Department of Engineering Cambridge, UK.

[46] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, Jose A Joao, and Moinuddin K Qureshi. 2018. Morphable counters: Enabling compact integrity trees for low-overhead secure memories. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 416–427.

[47] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, and Moinuddin K Qureshi. 2018. Synergy: Rethinking secure-memory design for error-correcting memories. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 454–465.

[48] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, and Moinuddin K Qureshi. 2018. Synergy: Rethinking secure-memory design for error-correcting memories. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 454–465.

[49] Subhash Sethumurugan, Jieming Yin, and John Sartori. 2021. Designing a cost-effective cache replacement policy using machine learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 291–303.

[50] Ishan Shah, Akanksha Jain, and Calvin Lin. 2022. Effective mimicry of belady's min policy. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 558–572.

[51] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 413–425.

[52] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2019. A neural hierarchical sequence model for irregular data prefetching. In *ML For Systems Workshop, NeurIPS*.

[53] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[54] Standard Performance Evaluation Corporation. 2017. SPEC CPU2017 Benchmark Suite. https://www.spec.org/cpu2017/

[55] G Edward Suh, Dwaine Clarke, Blaise Gasend, Marten Van Dijk, and Srinivas Devadas. 2003. Efficient memory integrity verification and encryption for secure processors. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 339–350.

[56] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[57] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[58] Michael Szydlo. 2004. Merkle tree traversal in log space and time. In *Advances in Cryptology-EUROCRYPT 2004: International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004. Proceedings 23*. Springer, 541–554.

[59] Meysam Taassori, Rajeev Balasubramonian, Siddhartha Chhabra, Alaa R Alameldeen, Manjula Peddireddy, Rajat Agarwal, and Ryan Stutsman. 2020. Compact leakage-free support for integrity and reliability. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 735–748.

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

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

[62] Muhammad Umar, Weizhe Hua, Zhiru Zhang, and G Edward Suh. 2022. Softvn: Efficient memory protection via software-provided version numbers. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 160–172.

[63] Sebastiano Vigna. 2017. Further scramblings of Marsaglia's xorshift generators. *J. Comput. Appl. Math.* 315 (2017), 175–181.

[64] Xin Wang, Jagadish Kotra, Alex Jones, Wenjie Xiong, and Xun Jian. [n. d.]. Counter-light Memory Encryption. ([n. d.]).

[65] Xin Wang, Jagadish B Kotra, and Xun Jian. 2022. Eager memory cryptography in caches. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 693–709.

[66] Xin Wang, Daulet Talapkaliyev, Matthew Hicks, and Xun Jian. 2022. Self-reinforcing memoization for cryptography calculations in secure memory systems. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 678–692.

[67] Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. 2023. Cipherfix: Mitigating Ciphertext {Side-Channel} Attacks in Software. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6789–6806.

[68] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. 2006. Improving cost, performance, and security of memory encryption and authentication. *ACM SIGARCH Computer Architecture News* 34, 2 (2006), 179–190.

[69] Pengmiao Zhang, Neelesh Gupta, Rajgopal Kannan, and Viktor K Prasanna. 2024. Attention, Distillation, and Tabularization: Towards Practical Neural Network-Based Prefetching. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 876–888.

[70] Pengmiao Zhang, Rajgopal Kannan, Ajitesh Srivastava, Anant V Nori, and Viktor K Prasanna. 2022. Resemble: reinforced ensemble framework for data prefetching. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[71] Yang Zhao, Xing Hu, Shuangchen Li, Jing Ye, Lei Deng, Yu Ji, Jianyu Xu, Dong Wu, and Yuan Xie. 2019. Memory trojan attack on neural network accelerators. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1415–1420.

[72] Wei Zheng, Ying Wu, Xiaoxue Wu, Chen Feng, Yulei Sui, Xiapu Luo, and Yajin Zhou. 2021. A survey of Intel SGX and its applications. *Frontiers of Computer Science* 15 (2021), 1–15.