ICGMM: CXL-enabled Memory Expansion with Intelligent Caching Using Gaussian Mixture Model

Hanqiu Chen[1§], Yitu Wang[2§], Luis Vitorio Cargnini[3], Mohammadreza Soltaniyeh[3], Dongyang Li[3], Gongjin Sun[3], Pradeep Subedi[3], Andrew Chang[3], Yiran Chen[2] and Cong Hao[1]
[1]Georgia Institute of Technology, [2]Duke University, and [3]Samsung Semiconductor, Inc.
{hanqiu.chen, callie.hao}@gatech.edu, {yitu.wang, yiran.chen}@duke.edu
{v.cargnini, m.soltaniyeh, dongyang.li, gongjin.s, prad.subedi, andrew.c1}@samsung.com [*]

Abstract

Compute Express Link (CXL) emerges as a solution for wide gap between computational speed and data communication rates among host and multiple devices. It fosters a unified and coherent memory space between host and CXL storage devices such as such as Solid-state drive (SSD) for memory expansion, with a corresponding DRAM implemented as the device cache. However, this introduces challenges such as substantial cache miss penalties, sub-optimal caching due to data access granularity mismatch between the DRAM "cache" and SSD "memory", and inefficient hardware cache management. To address these issues, we propose a novel solution, named ICGMM, which optimizes caching and eviction directly on hardware, employing a Gaussian Mixture Model (GMM)-based approach. We prototype our solution on an FPGA board, which demonstrates a noteworthy improvement compared to the classic Least Recently Used (LRU) cache strategy. We observe a decrease in the cache miss rate ranging from 0.32% to 6.14%, leading to a substantial 16.23% to 39.14% reduction in the average SSD access latency. Furthermore, when compared to the state-of-the-art Long Short-Term Memory (LSTM)-based cache policies, our GMM algorithm on FPGA showcases an impressive latency reduction of over 10,000 times. Remarkably, this is achieved while demanding much fewer hardware resources.

Keywords: Gaussian Mixture Model (GMM), Compute Express Link (CXL), DRAM Cache, Memory Expansion

1 Introduction

Memory wall [1] is a critical performance bottleneck significantly impacts the hardware efficiency in memory-intensive tasks [2–4]. Compute Express Link (CXL) [5] has emerged as a viable solution to this challenge. Built upon the serial

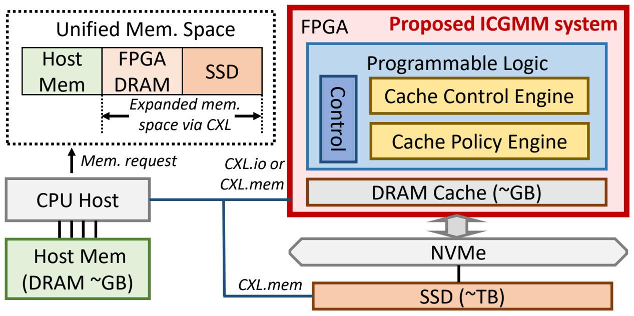

Figure 1. CXL-enabled memory expansion. SSD serves as an extension of host main memory. FPGA DRAM is used as a cache to facilitate memory access to SSD via CXL. FPGA programmable logic is used for intelligent cache management.

PCI Express (PCIe) infrastructure, CXL offers a low-latency, high-bandwidth interconnect technology. It facilitates the direct sharing of memory and cache resources among devices, thus greatly enhancing the performance of data-intensive applications [6–9].

Memory coherence provided by CXL facilitates memory expansion to the storage of space of CXL devices, such as SSD. Yang et al. [10] proposed using DRAM as a "cache" for storage in CXL-enabled memory expansion systems, demonstrating improved SSD access efficiency, however, DRAM "cache" still faces several challenges. ❶ **Large cache miss penalties.** Compared with using DRAM as main memory and SRAM as cache, SSD data access latency is in microseconds [11], significantly higher than DRAM access in nanoseconds. This leads to substantial cache miss penalties. ❷ **Sub-optimal caching.** Suboptimal caching arises due to a mismatch in data access granularity between DRAM (64B) and SSD (4KB) [12]. This necessitates a minimum cache block of one page size (4KB), often resulting in the caching of extraneous data. ❸ **Hardware-inefficient cache policy designs.** Prior learning-based cache policies [13–16] are managed by the software, inducing high overhead of executing the corresponding algorithm and overlooking the hardware performance considerations for CXL-device cache, i.e., the DRAM.

Motivated by these challenges, in this paper, we propose **ICGMM, a hardware-managed DRAM caching system for CXL-enabled memory expansion** prototyped on FPGA. ICGMM incorporates a hardware-efficient cache policy engine based on the Gaussian Mixture Model (GMM) for *improved cache hit rate and reduced average memory access*

[*][§]Equal contribution. Hanqiu Chen is responsible for prototype design on FPGA and paper writing, and Yitu Wang is responsible for GMM-based cache policy design. This work has been done as part of Hanqiu's and Yitu's internships in the Memory Solution Lab at Samsung Semiconductor, Inc.

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3656239

latency. Our contributions can be summarized as follows in four aspects: system, algorithm, hardware, and evaluation.

1. **System – Hardware-managed DRAM cache system design.** ICGMM is an end-to-end DRAM cache management system prototyped on FPGA. It features a hardware-implemented cache controller that functions independently of the host control. ICGMM also incorporates the GMM algorithm implemented on hardware for intelligent caching and eviction.
2. **Algorithm – GMM-based cache policy for intelligent caching and eviction.** We develop a two-dimensional GMM to predict the access frequency of requested SSD pages by leveraging physical addresses and timestamps from host memory access requests as the inputs. Based on GMM prediction, only frequently accessed pages will be cached in DRAM to reduce cache miss penalty, and the least frequently accessed pages will be evicted if needed.
3. **Hardware – Optimized GMM hardware design with dataflow architecture.** Our GMM design is hardware-friendly because GMM computation is highly pipelineable, as each Gaussian function operates independently. Compared to LSTM-based cache policies, small GMM size and the capability of GMM to predict SSD page access frequency on-the-fly using current status trace information without tracing back previous information offers significant memory advantages. To further minimize GMM overhead and resolve the complex control, we also designed a dataflow architecture ensuring that GMM computation can be overlapped with SSD access.
4. **Evaluation – On-board measurement.** We evaluate the performance of ICGMM end-to-end using the Alveo U50 FPGA. Compared to the traditional Least Recently Used (LRU) cache policy, ICGMM achieved a 0.32% to 6.14% decrease in cache miss rates, resulting in a 16.23% to 39.14% reduction in average SSD access latency across seven mainstream benchmarks encompassing various application types. Furthermore, GMM implementation outperforms LSTM-based policies on FPGA, reducing latency by over 10,000× while requiring fewer hardware resources.

2 Preliminary and Motivations

2.1 CXL-enabled memory expansion

Compute Express Link (CXL) [5] is a novel interconnect protocol built on PCIe that connects heterogeneous devices within a unified memory space. Its capability of simply using host load/store instructions via CXL.io and CXL.mem protocol accelerates device memory access from the host. Using DRAM as a cache for storage devices like SSDs is a feasible memory expansion option for high-efficiency SSD access in CXL-enabled systems [10]. However, as explained in Sec. 1, the DRAM cache is inefficient in facilitating data transfer. It is necessary to align the DRAM cache block size to 4KB to be consistent with SSD minimum access granularity. This granularity mismatch can decrease cache hit rates due to the inclusion of infrequently accessed data.

Motivated by the **suboptimal coarse-grained caching and large cache miss penalties**, we propose to design a more powerful cache policy engine by predicting the frequently accessed data at the page level (4KB), to **improve the cache hit rate and thus largely reduce the average SSD access latency** from the host via CXL. Fig. 1 illustrates an overview of CXL-enabled memory extension, where our proposed ICGMM prototyped on FPGA is highlighted.

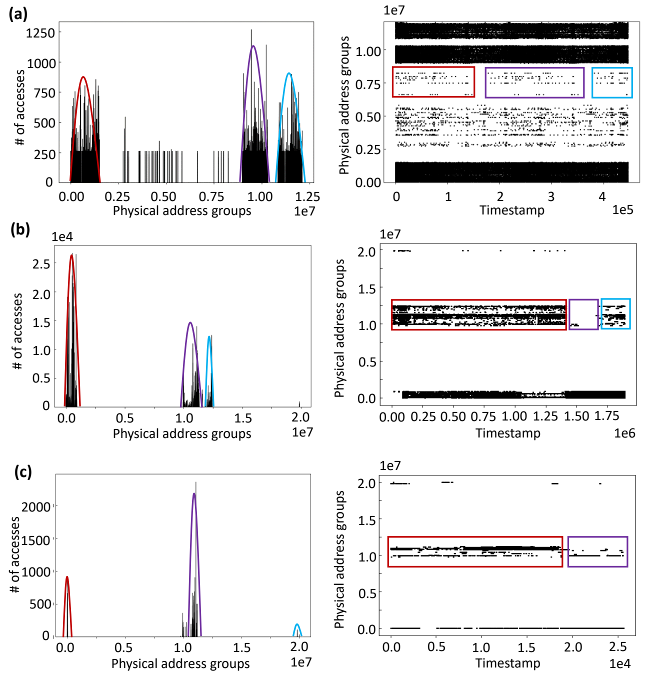

Figure 2. Memory access spatial distribution (left) and temporal distribution (right) from three benchmarks: (a) dlrm [17], (b) parsec [18], and (c) sysbench [19]. Spatial distribution can be fitted with different Gaussian functions; temporal distribution shows uneven access frequency within a specific range of addresses (see colored annotations).

2.2 Learning-based cache policy

Learning-based cache policies utilize historical caching data to forecast future caching priorities to boost cache hit rates. Integrating machine learning into cache policy design is currently a prominent research area. For example, DeepCache [13] and Glider [14] innovate in smart caching with an LSTM-based cache line predictor. LeCaR [20] and PARROT [15] adopt imitation (IL) and reinforcement learning (RL) for more complex policies. GL-Cache [16] groups similar cache objects for collective learning and eviction decisions.

While existing learning-based cache policies enhance cache hit rates, they are predominantly software-centric, neglecting the hardware cost for actual execution. In contrast, we propose a **hardware-friendly and lightweight** cache policy engine, aiming for **reduced hardware overhead and resource utilization** compared to existing cache policies.

2.3 Gaussian Mixture Model (GMM)

The GMM [21] is a flexible clustering model for capturing the underlying structure of data through a combination of Gaussian distributions. We conduct a preliminary

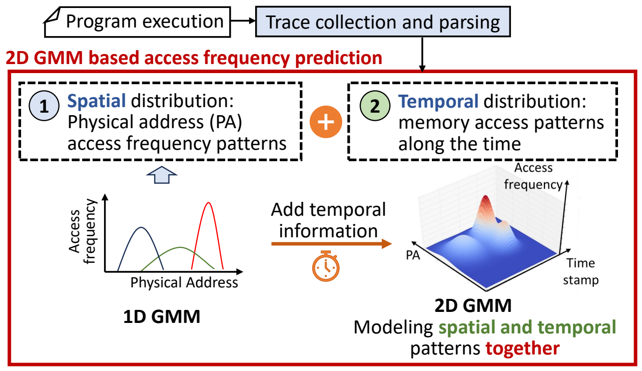
Program execution
Trace collection and parsing
2D GMM based access frequency prediction
1 Spatial distribution: Physical address (PA) access frequency patterns
2 Temporal distribution: memory access patterns along the time
Access frequency
Physical Address
1D GMM
Add temporal information
Access frequency
PA
Time stamp
2D GMM
Modeling spatial and temporal patterns together

Figure 3. We propose a two-dimensional GMM to capture both spatial and temporal memory access patterns.

study on three trace benchmarks: *dlrm* [17], parsec [18] and sysbench [19] with memory access spatial and temporal distribution patterns shown in Fig. 2: the *spatial distribution* describes the memory access frequency associated with data location, while the *temporal distribution* describes memory access location associated with time. We find that *spatial distribution* aligns with a mixture of Gaussian distributions, each having distinct means and co-variances. *Temporal distribution* is non-random and can be clustered into multiple groups, which is suitable to use GMM for clustering. We also observe similar spatial and temporal patterns on other trace benchmarks. To obtain the optimal accuracy, GMM needs to combine these two distributions for prediction because although some specific address ranges have higher access frequency than others, however, the access frequency distribution is *uneven* in temporal. Only considering spatial distribution will degrade GMM prediction performance.

Motivated by the **natural fit for employing GMM in modeling both spatial and temporal memory access patterns together**, we propose to use a **two-dimensional GMM** for memory access modeling, using transformed physical address (P) and timestamp (T) as inputs, as shown in Fig. 3. When extending to 2D, the Gaussian distribution can be expressed using the following equations:

$$\mathcal{N}(\mathbf{x}, |\boldsymbol{\mu}_k, \Sigma_k) = \frac{1}{2\pi|\Sigma_k|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_k)^T \Sigma_k^{-1}(\mathbf{x}-\boldsymbol{\mu}_k)\right) \quad (1)$$

$$\mathbf{x} = \begin{pmatrix} P \\ T \end{pmatrix} \quad \boldsymbol{\mu}_k = \begin{pmatrix} \mu_P \\ \mu_T \end{pmatrix}_k \quad \Sigma_k = \begin{pmatrix} \sigma_{PP} & \sigma_{PT} \\ \sigma_{TP} & \sigma_{TT} \end{pmatrix}_k \quad (2)$$

where $\boldsymbol{\mu}_k$ is a 2D mean vector and Σ_k is a 2×2 co-variance matrix of each 2D Gaussian function. By mixing K Gaussian functions together using different normalized weights π_k ($0 \leq \pi_k \leq 1, \sum_{k=1}^{K} \pi_k = 1$), our 2D GMM outputs a score $\mathcal{G}$ that predicts the future access frequency of each physical address, as shown in the following equation:

$$\mathcal{G}(\boldsymbol{\pi}, \boldsymbol{\mu}, \Sigma) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \Sigma_k) \quad (3)$$

where $\boldsymbol{\pi}$, $\boldsymbol{\mu}$, and Σ are trainable parameters for encoding different types of memory access traces.

Algorithm 1 Trace timestamp transformation for GMM

```
1:  $timestamp \leftarrow 0$
2:  $index \leftarrow 0$
3:  while a memory request comes do
4:      if $index \geq$ len_window then
5:          $timestamp \leftarrow timestamp + 1$
6:          $index \leftarrow 0$
7:      end if
8:      if $timestamp \geq$ len_access_shot then
9:          $timestamp \leftarrow 0$
10:     end if
11:     $index \leftarrow index + 1$
12: end while
```

3 Cache Policy Engine Design with GMM

The cache policy engine, integrated with GMM, determines whether the current requested SSD page will be cached and which page will be evicted based on GMM prediction score for access frequency. A higher score indicates a higher likelihood of future page access. As shown in Fig. 4, when the score is higher than the threshold, the page will be cached in the DRAM. To gather sufficient data for GMM training, each program runs for a long time, enough until passing the warm-up stage and the memory access pattern is stable. We use an open-sourced tool [10] for trace collection, including the read/write information, physical address, and access time. Then the trace will be parsed and processed (Sec. 3.1) to break down into meaningful time windows and exact inputs for GMM. The input of GMM includes the page index calculated from the physical address and transformed access timestamp. Next, GMM will make caching and eviction decisions based on the GMM score (Sec. 3.2). GMM is trained using the Expectation-Maximization (EM) algorithm (Sec. 3.3).

3.1 Trace processing

We do a preprocessing for the trace to facilitate GMM training. To mitigate program warmup biases, we discard the initial 20% and final 10% of traces. Since SSD access granularity is in 4KB pages, differing from host access granularity, we consolidate addresses into pages by assigning a page index (PI) computed from the physical address (PA) as $PI = PA << 12$.

To help GMM capture memory access locality better, we first partition the whole trace into small segments named *access shots*. The number of traces inside one access shot is represented by *len_access_shot*. We then continue to divide the traces inside one access shot into much smaller segments named *time window*. The number of traces inside a time window is represented by *len_window*. As outlined in Algorithm 1, we assign the same timestamp to traces inside the same time window, and the time window can be indexed by this timestamp. Different time windows are indexed with an incremental timestamp, which will be reset to zero if a trace reaches the end of the access shot. In our experiments, we empirically choose $len\_window = 32$ and

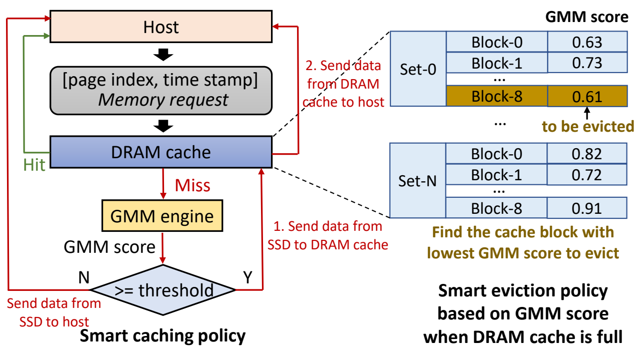

Figure 4. Intelligent caching and eviction with GMM.

len_access_shot = 10, 000 for optimal GNN training performance.

3.2 Intelligent caching and eviction with GMM

As shown in Fig. 4, to utilize GMM for intelligent caching, when an application is running on our system, if the page index of a memory request hits in the DRAM cache, the data is sent directly to the host, bypassing the GMM. On a cache miss, the GMM calculates the score for that page; if the score falls below a certain threshold, which indicates infrequent future access, then the page won't be cached and is sent directly from the SSD to the host. In contrast, scores above the threshold trigger caching those pages to the DRAM, anticipating frequent access in the future.

We also utilize GMM for intelligent eviction by substituting the LRU counter with the GMM score within cache blocks. Upon a full DRAM cache necessitating eviction, we sort the blocks by GMM score within the relevant set and evict the block with the lowest score, indicative of infrequent future access. After eviction, we update the cache with a new page from SSD, along with its corresponding GMM score.

3.3 GMM training

GMM training is conducted through unsupervised learning, utilizing the Expectation-Maximization (EM) algorithm. This process comprises two main steps. In the first expectation step, the probability of each trace belonging to each Gaussian function is calculated based on Bayes' theorem. In the second maximization step, GMM parameters ($\boldsymbol{\pi}$, $\boldsymbol{\mu}$ and Σ) are updated, ensuring that GMM parameters better represent input trace memory access patterns. After each iteration, the convergence is checked by calculating the change in maximum likelihood estimate (MLE) of $\boldsymbol{\pi}$, $\boldsymbol{\mu}$ and Σ. If the change is below the predefined threshold, GMM is converged and parameters will be saved for inference.

4 ICGMM Hardware Architecture Design

We prototype ICGMM on FPGA not only to demonstrate its hardware friendliness but, more importantly, ICGMM opens a new opportunity of utilizing the SmartSSD [22] as a CXL-enabled memory expansion device: SmartSSD has a small FPGA attached to its storage, which can function as a cache and controller for the SSD. Since host-based DRAM cache management is inefficient due to the high latency from frequent data exchanges between the host and SmartSSD, a hardware-managed cache control through programmable logic in the SmartSSD can significantly improve the efficiency of using DRAM as a cache for SSD.

However, the hardware implementation of ICGMM is nontrivial because of the complex control required over various modules and blocks. For example, the cache policy engine must be constantly active to wait for signals, triggering computations only upon cache miss. This run-time dynamic control cannot be determined during the hardware design phase. Therefore, we propose dataflow architecture using "free running kernels" for data-driven control and high parallelism between different modules and blocks.

ICGMM is prototyped on Xilinx Alveo U50 FPGA, utilizing its high-bandwidth memory (HBM) as the DRAM cache. As shown in Fig. 5, the system comprises three main modules: ❶ **a cache policy engine** with an optimized GMM kernel for predicting the likelihood of future page access frequency; ❷ **a cache control engine** responsible for cache management, hit/miss determination, and cache replacement, which also includes a SSD access latency emulator; ❸ **a signal controller** for interfacing with cache control and policy engines and managing data flow between HBM, on-board buffers, and different modules and blocks.

4.1 GMM policy engine design

Our GMM-based cache policy engine is an independent, data-driven module that can continuously process data without external intervention, which is also known as a "free-running kernel". Upon detecting a new trace from trace FIFO, the engine retrieves and decodes the trace to GMM inputs format and executes GMM inference. A dedicated control block oversees this policy engine, awaiting directives from a central signal controller. If the signal controller activates the cache policy engine, the trace FIFO and response (rsp) FIFO are activated, and GMM scores will be returned from the rsp FIFO to intelligently manage the cache. Otherwise, two FIFOs will be closed, and the system will run a default traditional Least Recently Used (LRU) strategy for cache replacement.

For optimized hardware performance, we exploit the feature of independent Gaussian functions in GMM by constructing a deep computation pipeline that processes different stages of the computation simultaneously with an initiation interval (II) = 1. For the task of score accumulation from different Gaussian functions, which requires sequential processing, we implement a shift register that holds temporal values during accumulation to resolve the data dependency. Furthermore, the GMM size is small enough to be stored within an on-board weight buffer, which avoids continuous data exchanges between the HBM and weight buffer.

4.2 Cache control engine design

The cache control engine consists of two modules, one for cache management, and another for emulating the SSD access latency. Within the cache management module, incoming traces from the trace FIFO queue are decoded to extract the set index, which is used to identify the appropriate cache set within the HBM. Instead of transferring the actual data

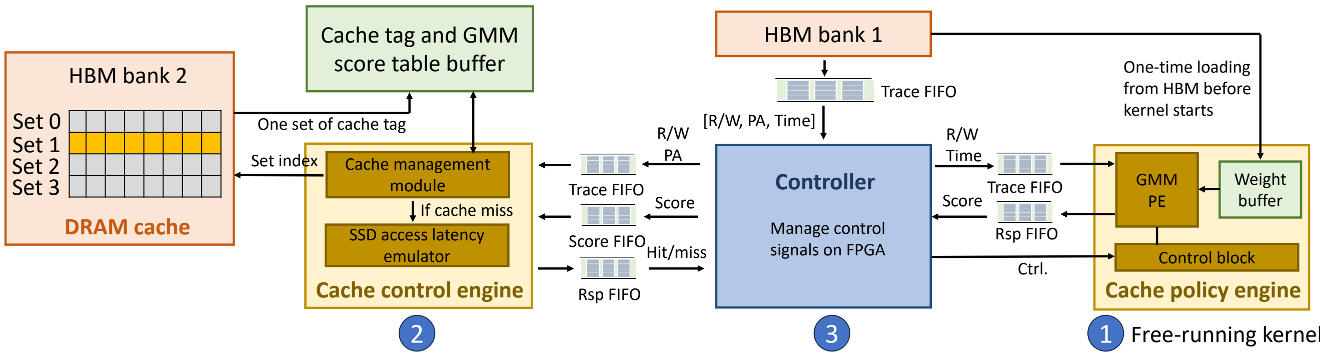

Figure 5. ICGMM hardware architecture design with three main modules: cache control engine, cache policy engine, and signal controller. ICGMM is designed as a dataflow architecture with FIFO interfaces between different modules for high parallelism and efficient data-driven control. PA means the physical address.

in the cache, only the cache tags and GMM scores are transferred to the on-board buffer. Partitioning the cache tag and GMM score table buffer allows for simultaneous comparison of all tags from various blocks with the target tag of the incoming memory request, as opposed to the traditional sequential comparison. This parallel processing significantly accelerates cache hit/miss determination.

For more precise end-to-end ICGMM performance evaluation on FPGA, we also incorporate an SSD access latency emulator within the cache control engine. This emulator operates during a cache miss event, pausing the dataflow in the cache control engine for a set duration to emulate SSD response times. The parameters for response time vary according to the type of SSD or other storage devices.

4.3 Dataflow architecture

We employ a dataflow architecture to minimize the additional overhead brought by the cache policy engine and trace loading from HBM. When the cache control engine is conducting tag comparison, new traces can be fetched from HBM into on-board registers, allowing simultaneous trace loading and cache management. Moreover, during a cache miss, the cache policy engine and SSD access emulator are triggered concurrently, significantly reducing extra delays caused by the cache policy engine. Moreover, dataflow architecture simplifies the complex control as data-driven control for "free-running" cache policy engine, resolving the non-deterministic behavior during run-time.

5 Experiment

5.1 Experiment setup

Trace collection source. For a comprehensive evaluation of ICGMM, we use different benchmarks, including both synthetic traces and traces from real-world applications. The synthetic trace benchmarks we choose are *hashmap* and *heap* [10]. The real-world trace benchmarks are from different domains, including *dlrm* [17] from deep learning recommendation systems, *parsec* [18] and *stream* [23] from high-performance computing, *memtier* [24] and *sysbench* [19] from database systems.

Hardware deployment. As a case study, the cache configuration we chose is shown as follows: cache size = 64 MB, block size = 4 KB, and associativity = 8. The number of Gaussian functions for GMM is 256. The target SSD for access latency emulation is fabricated using TLC technology, with average read latency = 75μs and write latency = 900μs [11]. ICGMM is deployed on Xilinx Alevo U50 FPGA running at 233MHz, using High-Level Synthesis (HLS) by Vitis HLS and Vivado 2023.1, with only 190 (14%) BRAM and 117 (2%) DSP consumption.

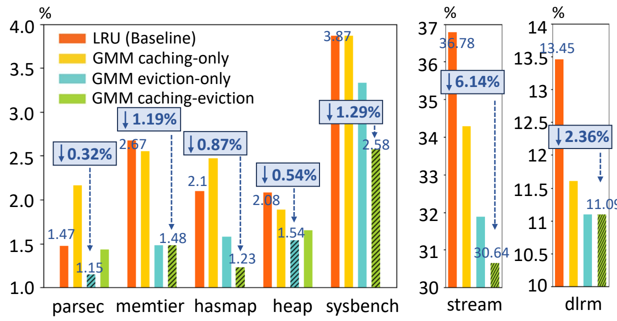

Figure 6. Comparison of cache miss rates between baseline LRU and three distinct strategies using GMM as the policy engine. Values the lower the better. Dashed bars indicate the best policy results.

5.2 Evaluation on GMM for caching and eviction

We evaluate the effectiveness of GMM as a cache policy engine by comparing the cache miss rate of GMM against the baseline LRU policy with three strategies: GMM for smart caching, GMM for smart eviction, and a combination of both. We pick the strategy yielding the lowest cache miss rate, as shown in Fig. 6. Our results indicate that GMM reduces cache misses across all traces. Specifically, GMM eviction alone is the best for *parsec* and *heap*, while a combined approach achieves optimal performance for other traces.

5.3 Hardware performance evaluation on FPGA

Average memory access latency reduction. Utilizing GMM for cache policy significantly lowers average memory access time by mitigating the substantial latency penalties of SSD access when cache miss happens. We conduct end-to-end on-board measurements, which reveal that DRAM cache hit time is 1μs. Upon a cache miss, GMM inference latency is 3μs, which is quick enough to be overlapped with the SSD read (75μs) or write (900μs) request latency. In this case, cache miss penalties come from SSD access latency, with a total penalty reaching 75μs if SSD read is required and 975μs for dirty cache block writing back to SSD upon eviction. As shown in Table. 1, GMM achieves a 16.23% to 39.14% reduction in average memory access time across seven benchmarks, compared to LRU.

Table 1. Comparison of average SSD access time using different cache policies: LRU and GMM.

Benchmark	LRU	**GMM**	**Reduction (%)**
parsec	3.92μs	**3.29μs**	**16.23**
memtier	2.98μs	**2.09μs**	**29.87**
hashmap	18.10μs	**11.02μs**	**39.14**
heap	16.48μs	**12.46μs**	**24.39**
sysbench	3.87μs	**2.91μs**	**24.79**
stream	156.39μs	**125.71μs**	**19.62**
dlrm	70.65μs	**58.43μs**	**17.30**

Table 2. Resource utilization and latency comparison between LSTM and GMM for cache policy engine.

	BRAM	DSP	LUT	FF	Latency
LSTM	339	145	85029	103561	46.3ms
GMM	**8**	**113**	**58353**	**152583**	**3μs**
GMM gain	2%	78%	69%	147%	15433×

GMM latency and resource utilization benefits. To highlight the hardware efficiency of GMM as a policy engine, we contrast it with an LSTM-based policy engine, which is commonly used in previous research [13, 14]. We design a three-layer LSTM model as a baseline with hidden dimension = 128, input sequence length = 32, and deploy the LSTM on the same FPGA platform with reasonable optimizations and similar DSPs utilization to ensure comparison fairness. During training, the LSTM is hard to converge across the same traces used for GMM using the same inputs, because it is unable to encode extensive temporal information in long traces with a lightweight design. A heavier LSTM model has a better encoding quality but at the cost of hardware efficiency. In comparison, as shown in Table 2, our LSTM baseline design is over 10,000× slower and consumes over 40× more BRAM than GMM, underscoring the superior hardware efficiency of GMM.

6 Conclusion

In this paper, we propose **ICGMM**, an intelligent hardware caching solution using a Gaussian Mixture Model (GMM) as a policy engine for CXL-enabled memory expansion systems. The optimized GMM-based policy engine smartly manages caching and eviction while minimizing overhead through dataflow architecture. ICGMM reduces cache miss rates by 0.32% to 6.14% and average SSD access time by 16.23% to 39.14% across diverse benchmarks. Furthermore, compared to the commonly used LSTM-based policy engine, the GMM-based policy engine achieves only 2% on-chip memory usage and over 10,000× quicker inference speed.

7 Acknowledgements

This work is partly supported by the Samsung Memory Solution Lab University Program. Andrew Chang provided supervision. Luis Vitorio Cargnini guided the design of a GMM-based cache policy engine, and Mohammadreza Soltaniyeh guided the implementation of a hardware prototype. Dongyang Li, Gongjin Sun, Pradeep Subedi, Yiran Chen, and Cong Hao contributed to insightful discussions and assisted in paper writing.

References

[1] Amir Gholami et al. Ai and memory wall. *RiseLab Medium Post*, 2021.
[2] Paresh Kharya and Ali Alvi. Using deepspeed and megatron to train megatron-turing nlg 530b, the world's largest and most powerful generative language model. *NVIDIA Developer Blog*, 2021.
[3] Minxue Tang et al. Fedcor: Correlation-based active client selection strategy for heterogeneous federated learning. In *CVPR*, 2022.
[4] Yitu Wang et al. Rerec: In-reram acceleration with access-aware mapping for personalized recommendation. In *ICCAD*, 2021.
[5] Compute Express Link (CXL). https://www.computeexpresslink.org/, 2022. Accessed: 2023-11-14.
[6] Debendra Das Sharma. Compute express link (cxl): Enabling heterogeneous data-centric computing with heterogeneous memory hierarchy. *MICRO*, 2022.
[7] Huaicheng Li et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *ASPLOS*, 2023.
[8] Yitu Wang et al. Ems-i: An efficient memory system design with specialized caching mechanism for recommendation inference. *ACM Transactions on Embedded Computing Systems*, 2023.
[9] Shiyu Li et al. Ndrec: A near-data processing system for training large-scale recommendation models. *IEEE Transactions on Computers*, pages 1–14, 2024.
[10] Shao-Peng Yang et al. Overcoming the memory wall with {CXL-Enabled}{SSDs}. In *USENIX ATC*, 2023.
[11] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces.* 2018.
[12] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *CSUR*, 2005.
[13] Chen Zhong et al. A deep reinforcement learning-based framework for content caching. In *CISS*, 2018.
[14] Zhan Shi et al. Applying deep learning to the cache replacement problem. In *MICRO*, 2019.
[15] Evan Liu et al. An imitation learning approach for cache replacement. In *ICML*, 2020.
[16] Juncheng Yang et al. {GL-Cache}: Group-level learning for efficient and high-performance caching. In *FAST*, 2023.
[17] Udit Gupta et al. The architectural implications of facebook's dnn-based personalized recommendation. *CoRR*, 2019.
[18] Christian Bienia et al. The parsec benchmark suite: Characterization and architectural implications. In *PACT*, 2008.
[19] https://github.com/akopytov/sysbench, 2023. Accessed: 2023-11-15.
[20] Giuseppe Vietri et al. Driving cache replacement with {ML-based}{LeCaR}. In *HotStorage*, 2018.
[21] Douglas A Reynolds et al. Gaussian mixture models. *Encyclopedia of biometrics*, 2009.
[22] Joo Hwan Lee et al. Smartssd: Fpga accelerated near-storage data analytics on ssd. 2020.
[23] John McCalpin. Stream: Sustainable memory bandwidth in high performance computers. *http://www. cs. virginia. edu/stream/*, 2006.
[24] memtier_benchmark, 2023. Accessed: 2023-11-15.