PetPS: Supporting Huge Embedding Models with Persistent Memory

Minhui Xie
Youyou Lu*
Tsinghua University
xmh19@mails.tsinghua.edu.cn
luyouyou@tsinghua.edu.cn

Qing Wang
Yangyang Feng
Tsinghua University
{q-wang18,fyy21}@mails.tsinghua.edu.cn

Jiaqiang Liu
Kai Ren
Kuaishou
liujiaqiang@kuaishou.com
kair@alumni.cmu.edu

Jiwu Shu
Tsinghua University
shujw@tsinghua.edu.cn

ABSTRACT

Embedding models are effective for learning high-dimensional sparse data. Traditionally, they are deployed in DRAM parameter servers (PS) for online inference access. However, the ever-increasing model capacity makes this practice suffer from both high storage costs and long recovery time. Rapidly developing Persistent Memory (PM) offers new opportunities to PSs owing to its large capacity at low costs, as well as its persistence, while the application of PM also faces two challenges including high read latency and heavy CPU burden. To provide a low-cost but still high-performance parameter service for online inferences, we introduce PetPS, the first production-deployed PM parameter server. (1) To escape with high PM latency, PetPS introduces a PM hash index tailored for embedding model workloads, to minimize PM access. (2) To alleviate the CPU burden, PetPS offloads parameter gathering to NICs, to avoid CPU stalls when accessing parameters on PM and thus improve CPU efficiency. Our evaluation shows that PetPS can boost throughput by 1.3 – 1.7× compared to PSs that use state-of-the-art PM hash indexes, or get 2.9 – 5.5× latency reduction with the same throughput. Since 2020, PetPS has been deployed in Kuaishou, one world-leading short video company, and successfully reduced TCO by 30% without performance degradation.

PVLDB Reference Format:
Minhui Xie, Youyou Lu, Qing Wang, Yangyang Feng, Jiaqiang Liu, Kai Ren, Jiwu Shu. PetPS: Supporting Huge Embedding Models with Persistent Memory . PVLDB, 16(5): 1013 - 1022, 2023.
doi:10.14778/3579075.3579077

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/thustorage/PetPS.

1 INTRODUCTION

Recently, embedding models have shown the superiority of learning on high-dimensional input data (such as user IDs, item IDs and user-item interactions). These models leverage the *embedding* technique, which projects the raw feature IDs into low-dimensional vectors, to capture the underlying semantic meanings. Such vector representations are then fed into various DNNs [16, 24, 31, 39, 48, 54–56] to predict user behaviors. Embedding models are widely used in high-revenue business scenarios such as search, recommendation and advertising. Specifically, in datacenters of giant Internet companies, embedding models take up to 80% of all AI inference cycles [25].

Due to the ever-growing corpus size of high-dimensional features and the wide use of the Feature Cross technique [16], the capacity of productional embedding models [32] has exploded unprecedentedly, from the scale of 1 billion (Google, 2016) to over 12 trillion (Meta, 2022), increasing for over four orders of magnitude. This rapid growth in parameters brings better model accuracy [52], but commensurately puts forward higher requirements for memory capacity. The common practice is storing these huge models in the DRAM of parameter servers (PS) [19, 30] to enable real-time access to parameters for inference. Unfortunately, huge embedding models make this practice suffer from two downsides. First, such practice imposes high storage costs, not only because DRAM is an expensive medium, but also because DRAM accounts for almost half of the total system power consumption [23]. At Kuaishou, one world-leading company in the short video area, there are thousands of dedicated servers in the datacenters for storing various embedding models, causing high capital expenditures and operating expenses. Second, after a PS outage, reloading parameters into the DRAM takes considerable recovery time, which may violate the service-level agreement (SLA) of online inferences.

Rapidly developing Persistent Memory (PM), also termed as Non-Volatile Memory, provides new opportunities for parameter servers. First, PM is byte-addressable like DRAM but can offer 8× memory capacity than DRAM at a lower cost. Second, it provides data persistence and can deliver quicker recovery and less downtime.

However, despite these attractive benefits, building parameter servers on PM still faces two challenges.

- *High PM read latency*: PM endures 3× higher latency than DRAM. If not managed well, the high latency could be fully exposed to applications and cause a decline in inference performance.
- *Heavy CPU burden*: By embracing PM, the number of parameters stored on a single PS becomes 8× larger than that with DRAM, which means we have to serve more parameter requests on a slower media, but with the same CPU resources. This makes the CPU a performance bottleneck for parameter servers on PM.

This paper focuses on building parameter servers of huge embedding models with PM, to provide a low-cost but still high-performance parameter service for online inferences. Specifically, we present the first production-deployed PM parameter server called PetPS (Persistent Embedding Table Parameter Server).

*Youyou Lu is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 16, No. 5 ISSN 2150-8097.
doi:10.14778/3579075.3579077

PetPS follows the process flow of conventional PSs: 1) *Indexing*, responsible for identifying the sizes and memory addresses of requested parameters, and 2) *Gathering*, which gathers (serializes) these parameters together to reply to clients. Our key contribution is the PM-suited design and implementation of these two steps to effectively tackle the aforementioned challenges, based on our analysis of real-world model service workloads.

For indexing, we propose PetHash, a highly-optimized PM hash index that greatly reduces PM access. Its key design principle is to minimize PM reads, by embracing the unique workload characteristics of embedding models. First, the parameter capacity of PSs is usually stable, which eliminates rehashing. Different from existing PM hash indexes [33, 37, 57] that use multi-level structures to trade multiple reads for cheap rehashing, PetHash uses a *single-level structure* to locate one bucket with only one PM read. Second, the access distribution of parameters is skewed. For hot parameters, PetHash leverages *hotness-aware placement* to avoid probing the hash index and thus provides fast access to them. Third, a request contains the query of hundreds of parameters. PetHash appropriately prefetches possible locations to be visited to further hide the high read latency of PM.

For gathering, we find that over 29% CPU cycles are spent on gathering due to the high read latency of PM. To improve CPU efficiency, PetPS introduces *NIC Gathering*, which offloads CPU's gathering operations to NIC. Specifically, we notice that modern NICs have the scatter-gather direct memory access (DMA) functionality, which can be re-purposed for gathering parameters on PM. With this design, the CPU is freed from waiting for slow PM when reading parameters. Such reduction in CPU stalls directly translates to performance improvements. Despite these benefits, the introduction of NIC Gathering may cause parameter inconsistency if there are concurrent writes to parameters being DMAed. We use a combination of the copy-on-write mechanism and our proposed *epoch-list-based space reclamation* scheme to carefully protect parameters under DMA.

Evaluation with real-world service trace shows that: PetPS outperforms PSs using state-of-the-art PM hash indexes by up to 1.7× (from 1.3×) in throughput, or achieves up to 5.5× (from 2.9×) latency reduction with the same throughput. PetPS can also deliver comparable or even better latency and only 12% throughput degradation to the in-house DRAM PS of Kuaishou.

Since 2020, PetPS has been deployed in the production video recommendation services of Kuaishou. It successfully reduced the total cost of ownership (TCO) by 30% without affecting the original service performance.

Overall, this paper makes the following contributions:

- We give a workload analysis of real-world huge embedding model services (§2.3) and identify two key challenges that arise when PM meets huge embedding models (§2.4).
- Guided by our analysis, we build the first production-deployed PM parameter server, PetPS, featured with PetHash (§3.2), a PM hash index tailored for the embedding model workloads, and NIC Gathering (§3.3), a mechanism which retrofits the DMA engine on NICs for gathering parameters.
- We perform evaluation with production workloads that validates the effectiveness and efficiency of PetPS (§4).

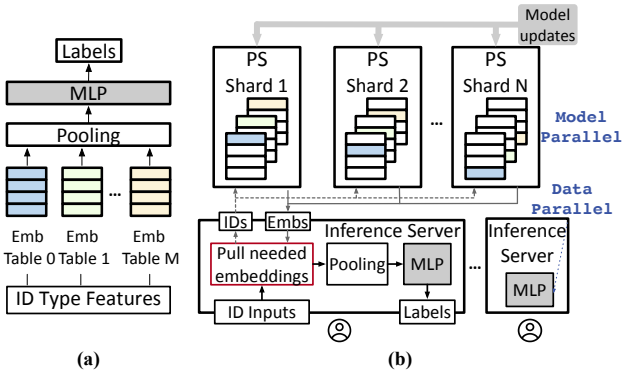

Figure 1: (a) Typical architecture of embedding models. *We omit non-ID type features (i.e., continuous features) for simplicity.* **(b) Production serving system for huge embedding models.** *Embedding vectors are sharded to PSs, whereas MLP is replicated to each inference server's GPU memory. The inference server requests PSs for needed embedding vectors.*

2 BACKGROUND & MOTIVATION

2.1 Persistent Memory

Rapidly developing Persistent Memory (PM) acts as a new tier between DRAM and SSD in today's storage hierarchy. It provides a large capacity at low costs with both DRAM-comparable access performance and SSD-like persistence (i.e., data will be retained when the system is shut down). PM is byte-addressable, and can be accessed by the CPU directly with load/store instructions.

Intel Optane DC Persistent Memory [8] (DCPMM) is the first and currently the only commercially-available PM product. This paper builds PS on DCPMM, but we only rely on its PM-common features (byte addressability, low latency, high capacity at low costs, persistence); hence, our PS can still work on future PM (such as CXL attached storage device [3]). The key obstacle for PM-enabled PS is PM's high read latency, which, without suitable system designs, could greatly drag down the performance of the inference service.

2.2 Embedding Models

Embedding models [16, 24, 31, 39, 48, 54–56] are a dominating type of deep learning models widely used in the core businesses of Internet companies such as recommendation systems and advertisements. Unlike traditional deep learning scenarios like computer vision or natural language processing, the input features of these core businesses contain a large number of ID-type features (e.g., user ID, video ID). These ID-type features are extremely high-dimensional and sparse, and cannot be directly learned by the neural network. To solve this problem, embedding models usually follow the Embedding-MLP architecture [26], which leverages the *embedding* technique to bridge the gap between ID-type features and neural network. Specifically, the model contains multiple *embedding tables*, each of which maps feature IDs of one specific ID type into low-dimensional vectors (called *embedding vectors*, or *embeddings* for short). Usually, the dimension of embeddings is between 16 and 256.

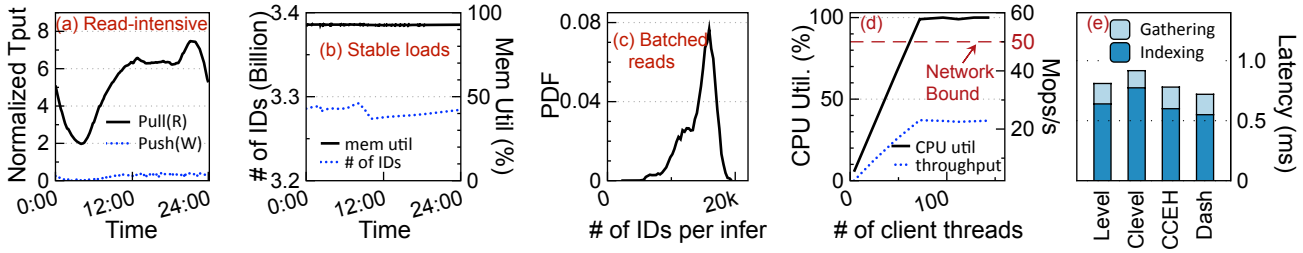

Figure 2: (a-c) Characteristics of one embedding model service in production. *(a) normalized throughput of read/write requests, (b) daily loads and memory utilization of PS, (c) PDF of feature ID count per inference request.* **(d-e) Motivation.** *(d) Throughput and CPU utilization of PS. (e) CPU time breakdown of PS.*

Fig. 1a shows the inference process of embedding models. ID-type features are first mapped to embeddings by looking up embedding tables. Then these embeddings are *pooled* (e.g., sum or average pooling), and finally fed into MLP to get the final label.

Modern embedding models demand a large number of parameters for embeddings. This is resulted by the combination of the large-scale cardinality in ID-type features and the wide utilization of feature cross technique (i.e., cartesian product) [16]. For example, the billion-scale video ID feature introduces an embedding table with trillions of parameters, while the feature cross of the user ID feature and video ID feature generates a new ID type, which leads to a much gigantic embedding table. As a result, embeddings usually account for over 99.9% of parameters in one embedding model [25, 32, 41]. With the increase in the number of IDs and more ID-type features introduced by model engineers, embedding models are racing to a scale of trillion parameters [32].

2.3 Huge Embedding Model Serving at Kuaishou

To serve with such giant embedding models, our serving system follows a disaggregated parameter server schema [49] (Fig. 1b). Specifically, there are two groups of servers: inference servers for the computation of neural networks with GPUs, and PSs for sharding the storage of giant models [19, 30]. Since MLP is usually small (a few hundred MBs), a common practice is to cache it on the GPU memory of inference servers, while store all embeddings on PSs. When receiving an inference task, the inference servers first acquire the corresponding embeddings of feature IDs from the PSs, and then continue to perform the forwarding of neural networks. PSs process requests with indexing and gathering, as stated in §1.

At Kuaishou, there are nearly one thousand models of various kinds, running on thousands of dedicated PS servers. Since inference is latency-sensitive, traditional PSs store parameters in DRAM to enable real-time access for inference servers [42, 49]. However, they have two drawbacks. First, the DRAM capacity required for storing huge embedding models introduces high CapEx and OpEx. Second, when PS outages, it needs to pull the complete model from the backend storage system. The several minutes of recovery time may violate the SLA of online inferences, affecting user experience.

This paper focuses on storing embeddings[1] on PS with PM, to enjoy its DRAM-like performance, persistence and high capacity at low costs. To better understand the characteristics of PS, we collect the accessing trace of one major video recommendation service at Kuaishou; see Fig. 2(a-c). The embedding model in this service achieves a scale of a trillion parameters. We draw three insights that are useful for the design of PetPS as follows:

- **Extremely-intensive reads.** Most operations (~98%) in PSs are read-only, with non-frequent writes (~2%) for model freshness.
- **Stable loads.** The parameter capacity and memory usage of PSs are stable. Although models continuously accept new coming IDs, we commonly eliminate old embeddings for accommodating new embeddings. As a result, the total loads keep stable.
- **Batched IDs in one read request.** A model inference request usually involves up to thousands of IDs. They are split into multiple read requests that are sent to PSs. In our setting, the batch size of IDs in one request is 500.

2.4 Motivation

To understand the basic performance and bottlenecks of PM-enabled PSs, we build a PS system using existing PM hash indexes and profile it. The index's keys are feature IDs and the values are the corresponding embeddings. The PS is exposed to clients via an RPC interface. We investigate four PM indexes, including Level [57], Clevel [15], CCEH [37], and Dash [33]. We use a machine with 6 DCPMMs as the PS and keep sending Pull requests to it. Each request contains 500 IDs, where IDs follow the Zipfian distribution ($\alpha = 0.99$).

CPU is the primary throughput bottleneck. Fig. 2d shows the throughput (in *Mops/s*, each parameter access is considered as an operation) and CPU utilization of PS with increasing client thread counts. We only show the result of Dash since other indexes share similar results. At the peak throughput, the system gets a 100% utilization of all 72 CPU cores, but only 46% of the available NIC bandwidth. Because the PM random read bandwidth (36 GB/s) is much higher than the NIC bandwidth (25 GB/s), we conclude that the CPU is exhausted and becomes the throughput bottleneck. The underlying reason is that, the CPU is stalled fetching the slow PM inefficiently, due to the inherent random access pattern of PS.

Most CPU time is spent on indexing, but non-negligible time on gathering. Fig. 2e further decomposes the CPU time of PS. First, across different PM hash indexes, over 84% of the CPU time is consumed on indexing. Thus, avoiding even one PM read per query in index can result in a significant latency reduction (e.g., the

[1]In the rest of paper, we use the term *parameter* and *embedding* interchangeably, since all parameters stored on PS are embeddings.

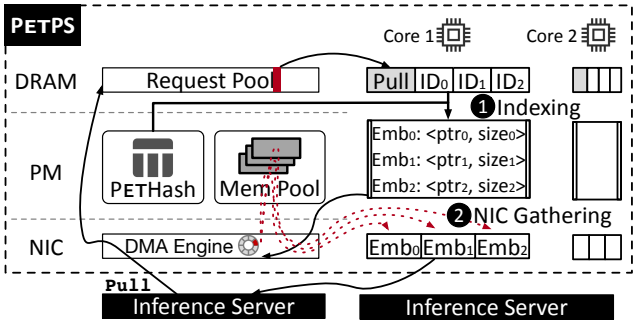

Figure 3: PETPS architecture.

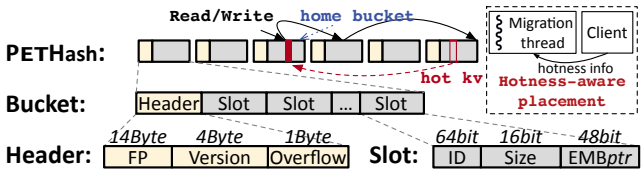

Figure 4: The structure of PETHash.

end-to-end latency can be reduced by 150 µs theoretically when the batch size is 500 and PM read latency is 300 ns). Second, aside from indexing, the CPUs of PSs spend up to 29% of time on gathering embeddings. We attribute it to the large batch size for gathering and the slow read latency of PM.

3 DESIGN & IMPLEMENTATION

3.1 PETPS Architecture

PETPS is a PM-enabled parameter server that enjoys PM's instant recovery and low storage costs. PETPS supports two interfaces: Pull (IDs) for getting the corresponding parameters of a batch of IDs (used by inference servers), and Push (IDs, embeddings) for updating a batch of parameters (used for model updates). Both interfaces are exposed via RPC.

PETPS follows the run-to-completion paradigm [43]. Each CPU thread of PSs undertakes the entire RPC lifetime, from receiving the request to sending the response back. Clients (inference servers) split all feature IDs of an inference task into a series of requests (empirically 500 IDs per request) to gain the concurrency benefits of multiple threads in PSs.

Fig. 3 illustrates the architecture of PETPS. It includes the following two key components: 1) PETHash, a persistent hash table highly optimized for parameter servers. Its keys are feature IDs, while the values point to the corresponding positions (e.g., memory addresses) of embeddings. 2) NIC Gathering, a mechanism that retrofits the DMA engine on NIC for gathering parameters and replying them back to clients. In addition, PETPS contains a slab-based memory allocator, Memory Pool, which manages the PM space for storing embeddings.

Fig. 3 also shows the workflow of a Pull request. Upon receiving a Pull request from an inference server, PETPS first looks up PETHash to get the addresses of parameters in Memory Pool, and then leverages the NIC Gathering mechanism to gather the parameters from Memory Pool and reply them to the inference server.

Next, we will introduce each of the key components in PETPS.

3.2 PETHash

Many prior PM hash indexes [15, 33, 37, 51, 57] commonly adopt the multi-level structure for ease of rehashing but pay the cost of multiple PM reads. For example, a state-of-the-art hash table, Dash [33], requires multiple pointer-chasing operations to locate one KV pair (directory → segment → bucket → KV pair). On the other hand, the key design principle of PETHash is to reduce the number of PM reads for most indexing operations to one, by embracing the unique workload characteristics in PSs.

Single-level structure. As our PSs have a stable capacity (§2.3), PETHash chooses the single-level structure of classic hash tables to reduce the read cost. The most obvious benefit is that we can locate a bucket directly by its bucket number, with only one PM read.

Fig. 4 shows the structure of PETHash. PETHash contains a single level of buckets. The bucket size is configurable, and is set to 256 bytes in PETPS, which is exactly the internal block size of PM. Borrowing the idea of a high-performance DRAM hash table, F14 [6], we do not choose chaining to resolve conflicts (due to the memory under-utilization in chained buckets), but use the open-address method. Each KV pair has a corresponding bucket (called *home bucket*) determined by the key's hash value. The KV pair is preferred to be placed in its home bucket, but it can also be displaced to other buckets when the home bucket is full.

Each bucket contains the following fields of metadata. A 14-byte *fingerprint* [38] enables fast intra-bucket search. A 4-byte *version* enables bucket-level write lock and lock-free search like Dash [33]. A 1-byte *overflow* counter indicates the number of KVs which should be placed in this bucket, but are displaced to other buckets since this one is full.

For insert operations, PETHash first locates the home bucket. If there are free slots, PETHash inserts the KV pair to it directly. If not, a displacement occurs. We probe the backward buckets with linear probing until we find an available one and insert to it. The probing step is set according to the hash of keys to avoid continuous full buckets. During the probing, the overflow counters of probed buckets are increased atomically to record this displacement.

For search/delete/update operations, PETHash probes buckets until the key is found, or it encounters a bucket whose overflow counter is 0, which means that the key does not exist in the hash table. Then it performs search/delete/update operations in the bucket. For delete operations, PETHash also needs to decrease the overflow counters in the probing path.

Note that the probing paths are expected to be short, if we set the hash table size appropriately. Theoretically, with a load factor of 0.8, over 99.99% of KVs can be found within three probes, and the mathematical expectation of probing count is 1.05. In practice, since the inference model is a replication of the training model, we can then empirically predetermine the hash table size, by considering the performance-and-space tradeoff.

Hotness-aware placement. Hotness-aware placement is designed based on the skew access pattern we often observe in real-world embedding models. The core idea is trying to place hot KV

pairs in their own home buckets, so that they can be found with only one PM read (in line with our design principle). Tracking the hotness of KV pairs should not cause interference to the service of PSs. Thus, instead of PSs, PetPS makes clients responsible for identifying hot sets because PSs' CPUs are precious while clients usually have spare CPU resources. Periodically (e.g., every 10 s), one client generates its hot sets by sampling and sends them to the corresponding PSs as the final hot set. Note that even only one client's hotness statistics are sufficiently accurate since all clients share a similar distribution of key accesses.

PetPS uses a dedicated *migration* thread to move hot keys. Upon receiving the new hot set, PetPS invokes the migration thread, which checks whether each hot key is in its own home bucket. If not, the migration thread first inserts the hot KV to its home bucket, in which case a cold victim may be migrated for making room (recursively), and then deletes it from the original bucket.

Prefetching. A Pull request often carries hundreds of keys, and all these keys are sequentially queried by one service thread of PS. This batched manner gives the opportunity of prefetching to accelerate the last mile of "one PM read principle". Before indexing the current key, PetPS initiates a prefetching instruction for the home bucket of the next key. With a high probability, the PM accesses for the indexing of the next key can be fully satisfied by the CPU cache. In this way, indexing and PM fetching are pipelined, and we can hide the latency of most PM reads in PetHash.

3.3 NIC Gathering

After indexing, the next step of PS is to gather the parameters based on the lookup results of the index, and send them back to the client. The conventional way of gathering by CPU wastes CPU cycles due to massive high-latency PM reads of parameters. PetPS proposes *NIC Gathering*, based on the observation that modern NICs already have the capability of *scatter-gather DMA*, which can be re-purposed for performing gathering embeddings. Fig. 5 compares the conventional CPU Gathering and PetPS's NIC Gathering.

In this subsection, we first briefly introduce the scatter-gather DMA on NIC, and then present how PetPS tames it for gathering embeddings. Since DMA is performed asynchronously by hardware, any concurrent updates or memory deallocation to the embeddings being DMA-ed may cause data inconsistency; we show how PetPS avoids such inconsistency and ensures correctness.

A primer on scatter-gather DMA on NIC. The lightweight DMA engines of modern NICs can perform zero-copy data scattering and gathering at low runtime costs. In this paper, we focus on the gathering operation. The programming interface is a *descriptor list* provided by user-space networking libraries such as Intel DPDK [5] and libibverbs [10]. Each descriptor includes the source address of a memory block along with its size. The software submits a list of descriptors to the NIC's DMA engine to initiate DMAs, and the DMA engine coalesces the memory blocks specified in the descriptor list together for network transmission. Scatter-gather DMA was originally designed for optimizing MPI collective communication to eliminate the additional copy of many large data blocks in the HPC scenario [22]. PetPS retrofits it for gathering embeddings without making modifications to the hardware.

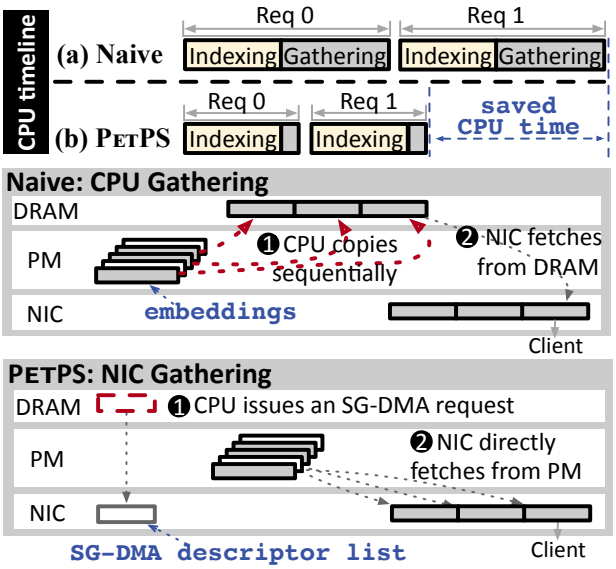

Figure 5: Comparison between conventional CPU Gathering and PetPS's NIC Gathering. *The red dashed line illustrates what the CPU does in both gathering schemes.*

Offloading embedding gathering to the NIC. To leverage the DMA engine for transferring embeddings with different sizes and dealing with missing embeddings, PetPS provides a compacted DMA-capable message layout. Each message contains two fields: the header and the payload. The header includes the number of missing embeddings and their feature IDs, while the payload compacts all the requested embeddings that are available in this PS.

If there are n embeddings in the payload field, assembling a message in PetPS will require $1 + n$ DMAs. One is for fetching the message header, which is initialized in a pre-allocated page-locked DRAM buffer. The other DMAs read n embeddings directly from PM. Leveraging the doorbell batching mechanism [28], PetPS needs notifying the NIC for only once to launch all these DMAs.

We also use domain-specific knowledge to optimize the metadata size of the message layout. This is particularly useful for reducing network traffic when transferring small embeddings (like low-dimensional embeddings or quantized embeddings). First, instead of explicitly concatenating each feature ID and its corresponding embedding like conventional PSs [30], we organize embeddings in order of feature IDs in the client's request, which implies the mapping information between IDs and embeddings. This can save the space of a feature ID (usually 8 bytes) for each embedding. Second, the message layout does not include the sizes of each embedding. This is because features of the same type share the same embedding size. We can maintain a type-to-size mapping table (with dozens of entries) and get the corresponding embedding size easily by looking up it with a specific feature type.

Protecting embeddings being gathered by DMA. Since DMA is done asynchronously by the NIC, PetPS must ensure two invariants during the DMA process: (i) the original embeddings must not be modified, and (ii) their memory must not be freed. Otherwise, partially-updated or wrong embeddings may be sent to the client.

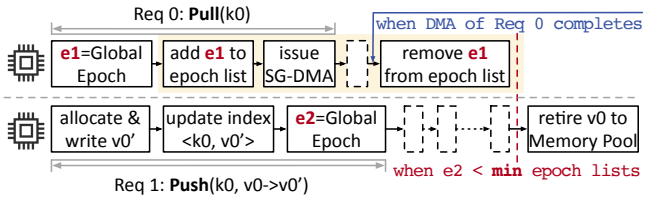

Figure 6: How the copy-on-write mechanism and our epoch-list-based reclamation safely protect NIC Gathering. *The upper part is a reader thread, while the lower part is a concurrent update thread. The shaded part shows the protection domain.*

For (i), PetPS uses the copy-on-write mechanism to ensure no in-place writes. When receiving a Push request (i.e., updating parameters), PetPS first allocates a batch of buffers with the same sizes as the to-be-updated embeddings from Memory Pool and writes the updates to them. Then, PetPS updates the new locations of these embeddings to PetHash atomically.

For (ii), the core obstacle of safely recycling old-version parameters is that: the updater threads are unaware of whether there are DMAs reading them concurrently and whether there will be upcoming ones. To record these information of reader threads for updater threads, PetPS proposes an *epoch-list-based space reclamation* scheme. Different from the classic epoch-based space reclamation [21], which sets a single epoch for a thread, PetPS assigns each reader thread an epoch list, which bookkeeps the epoches of all ongoing and forthcoming DMAs. Specifically, for the reader thread, each time it starts processing a Pull request, it acquires a global epoch and adds the epoch to its epoch list. The list will delete this epoch after the completion of scatter-gather DMA. For the update thread, it first logically frees the old-version embedding (i.e., only updating the index), and records the current global epoch as e. The old-version embedding can be freed physically (i.e., recycled to Memory Pool) only if the smallest one of all threads' epoch lists is greater than e, in which case all reader threads and DMA requests have no references to it. Fig. 6 shows a concrete example.

Advantages. Offloading the embedding gathering process to NIC brings three advantages. First, it is well suited for PM. It eliminates CPU stalls caused by bulk of high latency PM reads, alleviating the CPU bottleneck and thus improving the performance of PS. Second, the NIC-based gathering does not disturb CPU cache [7], while CPU Gathering incurs cache pollution, which affects the performance of indexing. According to our evaluation, CPU Gathering leads to 11% more LLC misses in indexing. Third, it does not rely on customized hardware and can be deployed directly on off-the-shelf NICs in our datacenter. Scatter-gather DMA, as a basic feature, has already been supported by most vender NICs, including Intel, Mellanox, and Broadcom.

3.4 Recovery

There are two phases for the recovery of PetPS: (i) recovering local PM, and (ii) catching up on unfinished model updates from training clusters. For (i), it is similar to existing PM KV stores. For (ii), we record recent model updates in a Kafka[1]-like message queue and the reading status of each PS persistently. Thus, after restarting, the outaged PSs can continue to consume model updates.

4 EVALUATION

4.1 Experimental Setup

Testbed. We run experiments on three machines, one as the PS, and two as the clients. All the machines are equipped with two Intel Xeon Gold 6240M CPUs at 2.6 GHz, 64GB of DRAM, and two 100Gbps Mellanox ConnectX-5 NICs. The PS machine has an additional 1.5TB of Intel Optane DCPMM (6 × 256GB).

Although we employ DCPMM for evaluation, the assumptions to PM in PetPS are completely based on its generic properties (byte addressability, access latency between DRAM and SSD, high capacity at low cost, persistence), independent of specific characteristics of Optane's 3D XPoint media. Therefore, we believe that PetPS remains effective for any future forms of PM (e.g., CXL storage devices like Samsung Memory-Semantic SSD [11]).

We configure the PS to use all cores as the RPC threads, and let clients continuously send requests to it. Each client thread issues up to 2 concurrent Pull requests, with 500 feature IDs per request. We adjust the client thread count to change the test pressure.

Workloads. We use the following two workloads.

Production. We use a production workload to evaluate the performance of PetPS and other competitor systems. It is a day-long trace collected from one major video recommendation service in Kuaishou production. The model running behind contains 100 billion parameters in total. The embedding dimension is set to 128. After sampling (for confidentiality), the trace contains 6 million samples, and involves 1 billion feature IDs.

YCSB. We use a synthetic workload YCSB [17] for the sensitivity study, since it can easily change the access pattern (such as feature ID distribution, skewness, and read/write ratio). Unless otherwise specified, we use YCSB-C (100% read) and a Zipfian feature ID distribution with the default parameter 0.99. The model is set to the same as that of the Production workload.

Competitors. We compare PetPS with the following three systems.

PSLite [4] is the implementation of classic PSs [30]. It leverages C++ unordered_map as the index. PSLite is used by popular systems like MXNet [2] and BytePS [27]. The original PSLite is single-threaded and we reimplement it to support multiple threads by maintaining 32 shards of indexes and using per-shard reader-writer locks for concurrency control.

DashPS replaces PSLite's index with the state-of-the-art PM hash index, Dash [33].

KuaiPS is Kuaishou's in-house implementation of DRAM PSs. It uses a concurrent chaining-based hash index. Like PetPS, it supports a variety of features required by production (e.g., parameter admission, retirement).

For fair comparisons, we make the following modifications to competitor systems. 1) For the systems which do not support PM natively, we simply modify them by allocating both the index and parameters on PM with our Memory Pool. 2) To avoid side effects of rehashing, we reserve sufficient memory for indexes like [14]. 3) We port the RPC of PetPS to all competitor systems to eliminate the performance bias caused by network stacks.

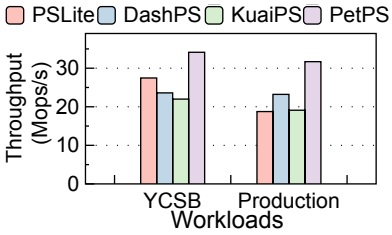

Figure 7: (Exp #1) Peak throughput.

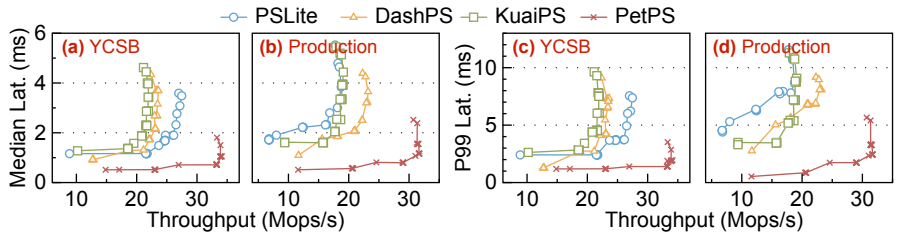

Figure 8: (Exp #1) Throughput v.s. latency.

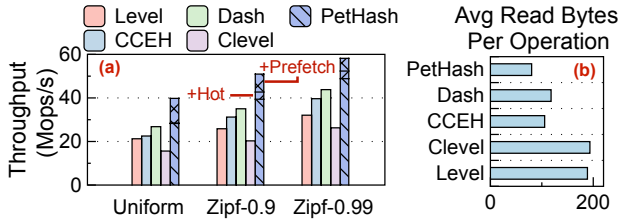

Figure 9: (Exp #2) Performance of PetHash. *(a) YCSB with three key distributions. (b) YCSB with Zipfian distribution ($\alpha = 0.99$).*

4.2 Overall Performance

Exp #1: Overall performance. Fig. 7 shows the peak throughput of each system. The unit *op* refers to an operation of pulling or pushing a parameter, the same below. PetPS yields 1.3 – 1.6× throughput gains on YCSB-C and 1.4 – 1.7× on Production respectively compared with other systems. It justifies the design of PetPS.

To illustrate more clearly, we collect the median / 99th percentile (P99) latencies and the corresponding throughput of clients under different test pressures; see Fig. 8. We find that PetPS consistently exhibits higher throughputs and lower latencies than other systems. At a low request pressure (such as 20Mops/s), PetPS respectively achieves 2.9 – 5.5× and 3.1 – 5.1× reductions in median and P99 latencies. At peak throughput, the reduction of the two latencies reaches 2.6 – 3.1× and 2.2 – 3.3×. All systems are bottlenecked by CPUs. KuaiPS first encounters the bottleneck since its index designed for DRAM is not suitable for PM, and its CPU is also responsible for other extra work such as hotspot identification and parameter retirement. The chained hashing of PSLite and the extendible hashing of Dash all introduce multiple reads to the indexing process (see Exp #2 for details), thus degrading the system performance. Our PetPS performs better by leveraging PetHash to greatly reduce the indexing time, with NIC Gathering to further unlash CPU's gathering.

4.3 Techniques

Exp #2: PetHash. We use the YCSB-C workload to evaluate the performance of PetHash and existing PM hash indexes. The key and value sizes are set to 8 bytes, which is the same with the setting of PS. We evaluate three key distributions, including uniform, Zipfian-0.9, and Zipfian-0.99 (a larger value means a more skewed distribution). Fig. 9a presents their throughput. We make the following observations: 1) PetHash enjoys 1.3 – 2.5× higher throughput than existing PM hash indexes because its PS-suited design greatly reduces the amount of PM reads (up to 2×, see Fig. 9b). 2) Compared with a raw PetHash, enabling hotness-aware placement can promote 8% throughput, mainly stemming from the query path reduction of hot KVs. This technique brings more improvement under a more skewed distribution. 3) Prefetching can further enhance the throughput by 11-40%. This benefit mainly comes from the hiding latency of most PM reads.

Exp #3: NIC Gathering. Fig. 10 illustrates the CPU time breakdown of PetPS, with two gathering schemes, at their peak throughput. We find that: 1) NIC Gathering reduces the gathering time from 180 µs to 14 µs (about 12.8×). It justifies that NIC Gathering eliminates the CPU overhead in gathering. 2) With NIC Gathering, the indexing part also gets a speedup of 22% since it avoids cache pollution caused by CPU Gathering. 3) The CPU saved from NIC Gathering can then improve the peak throughput by 1.2×.

Note that NIC Gathering is also applicable to DRAM. We repeat the above experiments by substituting PM with DRAM. Interestingly, we find that NIC Gathering instead causes a 30% decline of throughput. The underlying reason we find is that the low latency of DRAM makes the DMA engine a performance bottleneck.

Exp #4: Contributions of techniques to performance. Fig. 11 shows the contributions of techniques to the final performance of PetPS. We start with DashPS. By substituting Dash with PetHash, PetPS gets 1.2 – 1.3×, 1.3 – 1.5×, and 1.2 – 1.7× improvement in terms of throughput, median latency and P99 latency, since it reduces PM reads and speeds up the indexing step. Introducing NIC Gathering further delivers up to around 1.2× higher throughput, 2.3× lower median latency and up to 2.0× lower P99 latency, mainly coming from lifting the burden of gathering off of CPUs.

4.4 Sensitivity Study

Exp #5: Impact of write ratios. We use YCSB to test the sensitivity of PetPS to writes. We set different update ratios, the remaining proportion is for reading. As shown in Fig. 12, PetPS consistently achieves similar or better performance than any other system, even with high update ratios. This is because the single-level structure of PetHash is also friendly to updates. With a higher read ratio, PetPS can enhance more performance (thanks to NIC Gathering).

Exp #6: Impact of embedding dimensions. We use the Production workload and equip all systems with different embedding dimensions. Fig. 13 shows the CPU time breakdown at peak throughput of Dash-PS and PetPS. Due to space limitations, we omit other competitors with similar performance. The increased embedding dimension makes other systems take more time in the gathering

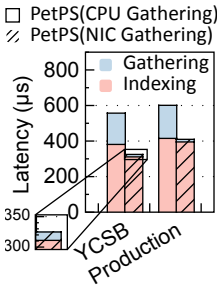

Figure 10: (Exp #3) Benefits of NIC Gathering.

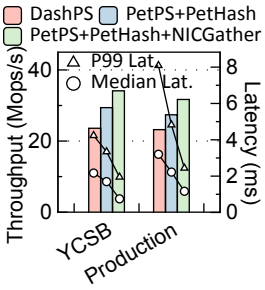

Figure 11: (Exp #4) Contribution of techniques to performance.

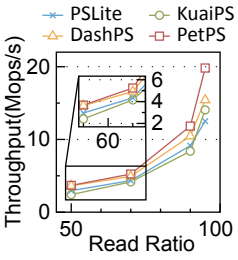

Figure 12: (Exp #5) Performance impact of write ratios.

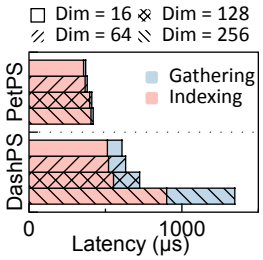

Figure 13: (Exp #6) Performance impact of embedding dimensions.

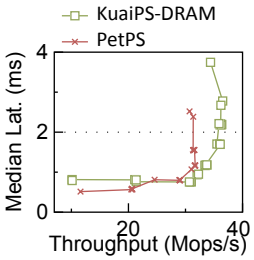

Figure 14: (Exp #7) Performance comparison with DRAM PS.

step and consequently affects the indexing step. However, PetPS keeps CPU efficiency even under large embedding dimensions.

4.5 Comparison with DRAM PS

Exp #7: Performance comparison with DRAM PS. We run KuaiPS on DRAM (called KuaiPS-DRAM) and compare it with PetPS on PM. Fig. 14 shows the throughput-latency curves of two systems. We only show the median latency due to space limitation, and the results of P99 latency are similar. 1) When the throughput is below 28Mops/s, PetPS reaches similar or even lower latency. 2) PetPS achieves only 12% lower peak throughput than KuaiPS-DRAM. Taking into account that the price of DRAM is 1.9× higher than that of PM, PetPS yields much better cost-efficiency gains than KuaiPS. According to our cost analysis [9], PetPS can reduce the TCO of PSs by 30% without performance degradation.

Exp #8: Recovery time. We evaluate the recovery time of PetPS on PM and KuaiPS on DRAM. For the model with 1 billion feature IDs, PetPS only needs 5 seconds of recovery time with the instant recovery of PM, while KuaiPS requires nearly 7 minutes to reload the model from remote storage.

5 RELATED WORK

Parameter servers for embedding models. PS architecture [18, 19, 27, 30, 40, 50] decouples model storage from computation, and eases training and serving of large-scale embedding models [35, 36, 42]. Traditional PSs are DRAM-only, introducing high storage costs when facing at-scale embedding models. There are several works extending the memory hierarchy of PS beyond DRAM with SSDs.

In the training scenario, AIBox [53] and HierPS[52] equip PS with SSDs. Their intuition is that we can prefetch parameters of the next training step to hide the latency of low-speed media, as the training dataset is all pre-known. Compared with them, the inference scenario of PetPS is more challenging due to the pre-unknown inference requests.

In the inference scenario, Bandana [20] stores embedding models with NVMe SSDs. The main challenges Bandana deals with are the limited read bandwidth and read amplification of SSDs. SDM [12] extends Bandana with a customized NVMe driver and a sophisticated DRAM cache. PetPS distinguishes from them in two aspects. 1) Bandana and SDM with slow SSDs are device-bottlenecked, while PetPS with fast PM is CPU-bottlenecked. This leads to different considerations and designs. 2) With a low performance, Bandana and SDM only suit for non-performance-critical embeddings (e.g., user embeddings) [20], which only take a small proportion of all embeddings. However, PetPS has a broader suitability since it delivers significantly better latency and throughput with PM.

Persistent memory indexes. PM is a promising low-cost substitute for DRAM. In the last few years, there is a wealth of work on PM indexes [13, 14, 29, 34, 46, 47]. The most relevant work with PetPS is PM hash indexes [15, 33, 37, 51, 57]. They mainly focus on optimizing writing, but are not suitable in the PS scenario with massive reads. Level and Clevel [15, 57] apply the two-level scheme to reduce PM writes and only need to rehash entries in one level. However, under the read-intensive scenario, Level suffers additional conflicts for its reader-writer locks for concurrency control, while Clevel needs to probe all levels for reads inefficiently. CCEH [37] and Dash [33] follow the extendible hash schemes. They can achieve a finer resizing granularity by only rehashing segments. However, they suffer from multiple pointer-chasing operations when locating one KV pair. For the reading path optimization, Dash employs fingerprint [38] to avoid unnecessary in-chunk PM reads and alleviate CPU stalls. PetPS also leverages it to speed up in-chunk probing. Coroutine-based approaches [44, 45] try to hide the high PM latency by scheduling another task (coroutine). PetPS achieves similar effects by customizing the batch query pipeline.

6 CONCLUSION

PetPS, as the first PM-enabled PS, achieves both fast recovery and low storage costs for huge embedding models compared with traditional DRAM PSs. First, based on the unique workload characteristics, PetPS designs a hash index tailored for PSs and greatly reduces PM reads. Second, PetPS offloads gathering from the CPU to NICs and thus promotes CPU efficiency. Evaluation with production workloads shows that PetPS outperforms PSs using state-of-the-art PM hash indexes. Our PetPS provides a prime example of PM's industrial application; we expect further efforts on this emerging hardware from both academia and industry.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China (Grant No. 2021YFB0300500), the National Natural Science Foundation of China (Grant No. 62022051) and Kuaishou.

REFERENCES

[1] 2022. Apache Kafka. https://kafka.apache.org/.

[2] 2022. Apache MXNet | A flexible and efficient library for deep learning. https://mxnet.apache.org/versions/1.9.1/.

[3] 2022. Compute Express Link. https://www.computeexpresslink.org/.

[4] 2022. dmlc/ps-lite: A lightweight parameter server interface. https://github.com/dmlc/ps-lite.

[5] 2022. DPDK. https://www.dpdk.org/.

[6] 2022. facebook/folly. https://github.com/facebook/folly/blob/main/folly/container/F14.md.

[7] 2022. Intel(r) Data Direct IO A Primer. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf.

[8] 2022. Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[9] 2022. Kuaishou: Storage Upgrade for Short Video Services. https://www.intel.com.au/content/www/au/en/customer-spotlight/stories/kuaishou-customer-story.html.

[10] 2022. rdma-core/ibverbs.h at master · linux-rdma/rdma-core. https://github.com/linux-rdma/rdma-core/blob/master/libibverbs/ibverbs.h.

[11] 2022. Samsung Electronics Unveils Far-Reaching, Next-Generation Memory Solutions at Flash Memory Summit 2022 – Samsung Global Newsroom. https://news.samsung.com/global/samsung-electronics-unveils-far-reaching-next-generation-memory-solutions-at-flash-memory-summit-2022.

[12] Ehsan K Ardestani, Changkyu Kim, Seung Jae Lee, Luoshang Pan, Valmiki Rampersad, Jens Axboe, Banit Agrawal, Fuxun Yu, Ansha Yu, Trung Le, et al. 2021. Supporting Massive DLRM Inference Through Software Defined Memory. *arXiv preprint arXiv:2110.11489* (2021).

[13] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: an efficient hybrid PMem-DRAM key-value store. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1544–1556.

[14] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1077–1091.

[15] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. 2020. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 799–812.

[16] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, DLRS@RecSys 2016, Boston, MA, USA, September 15, 2016*, Alexandros Karatzoglou, Balázs Hidasi, Domonkos Tikk, Oren Sar Shalom, Haggai Roitman, Bracha Shapira, and Lior Rokach (Eds.). ACM, 7–10. https://doi.org/10.1145/2988450.2988454

[17] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[18] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. 2016. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the eleventh european conference on computer systems*. 1–16.

[19] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger (Eds.). 1232–1240. https://proceedings.neurips.cc/paper/2012/hash/6aca97005c68f1206823815f66102863-Abstract.html

[20] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim M. Hazelwood, Asaf Cidon, and Sachin Katti. 2019. Bandana: Using Non-Volatile Memory for Storing Deep Learning Models. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*, Ameet Talwalkar, Virginia Smith, and Matei Zaharia (Eds.). mlsys.org. https://proceedings.mlsys.org/book/277.pdf

[21] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory.

[22] Ana Gainaru, Richard L Graham, Artem Polyakov, and Gilad Shainer. 2016. Using infiniband hardware gather-scatter capabilities to optimize mpi all-to-all. In *Proceedings of the 23rd European MPI Users' Group Meeting*. 167–179.

[23] Saugata Ghose, Abdullah Giray Yaglikçi, Raghav Gupta, Donghyuk Lee, Kais Kudrolli, William X. Liu, Hasan Hassan, Kevin K. Chang, Niladrish Chatterjee, Aditya Agrawal, Mike O'Connor, and Onur Mutlu. 2018. What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 3 (2018), 38:1–38:41. https://doi.org/10.1145/3224419

[24] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, Carles Sierra (Ed.). ijcai.org, 1725–1731. https://doi.org/10.24963/ijcai.2017/239

[25] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*. IEEE, 488–501. https://doi.org/10.1109/HPCA47549.2020.00047

[26] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*. 173–182.

[27] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 463–479.

[28] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design guidelines for high performance {RDMA} systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 437–450.

[29] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 462–477.

[30] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 583–598.

[31] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. 2018. xDeepFM: Combining Explicit and Implicit Feature Interactions for Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Yike Guo and Faisal Farooq (Eds.). ACM, 1754–1763. https://doi.org/10.1145/3219819.3220023

[32] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, et al. 2021. Persia: A Hybrid System Scaling Deep Learning Based Recommenders up to 100 Trillion Parameters. *arXiv preprint arXiv:2111.05897* (2021).

[33] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (apr 2020), 1147–1161. https://doi.org/10.14778/3389133.3389134

[34] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. {ROART}: Range-query Optimized Persistent {ART}. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 1–16.

[35] Xupeng Miao, Yining Shi, Hailin Zhang, Xin Zhang, Xiaonan Nie, Zhi Yang, and Bin Cui. 2022. HET-GMP: A Graph-based System Approach to Scaling Large Embedding Model Training. In *Proceedings of SIGMOD Conference*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 470–480. https://doi.org/10.1145/3514221.3517902

[36] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. 2022. HET: Scaling out Huge Embedding Model Training via Cache-enabled Distributed Framework. *Proc. VLDB Endow.* 15, 2 (2022), 312–320.

[37] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. 2019. {Write-Optimized} Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 31–44.

[38] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*. 371–386.

[39] Qi Pi, Guorui Zhou, Yujing Zhang, Zhe Wang, Lejian Ren, Ying Fan, Xiaoqiang Zhu, and Kun Gai. 2020. Search-based user interest modeling with lifelong sequential behavior data for click-through rate prediction. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 2685–2692.

[40] Alexander Renz-Wieland, Rainer Gemulla, Zoi Kaoudi, and Volker Markl. 2022. NuPS: A Parameter Server for Machine Learning with Non-Uniform Parameter Access. In *Proceedings of the 2022 International Conference on Management of Data*. 481–495.

[41] Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. 2022. RecShard: statistical feature-based memory optimization for industry-scale neural recommendation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 344–358.

[42] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. 2022. Ekko: A Large-Scale Deep Learning Recommender System with Low-Latency Model Update. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 821–839. https://www.usenix.org/conference/osdi22/presentation/sima

[43] Akshitha Sriraman and Thomas F Wenisch. 2018. {μTune}:{Auto-Tuned} Threading for {OLDI} Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 177–194.

[44] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Building blocks for persistent memory. *The VLDB Journal* 29, 6 (2020), 1223–1241.

[45] Marina Vemmou and Alexandros Daglis. 2021. COSPlay: Leveraging Task-Level Parallelism for High-Throughput Synchronous Persistence. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 86–99.

[46] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. 2022. Pacman: An Efficient Compaction Approach for {Log-Structured}{Key-Value} Store on Persistent Memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 773–788.

[47] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. 2021. Nap: A {Black-Box} Approach to {NUMA-Aware} Persistent Memory Indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 93–111.

[48] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & Cross Network for Ad Click Predictions. In *Proceedings of the ADKDD'17, Halifax, NS, Canada, August 13 - 17, 2017*. ACM, 12:1–12:7. https://doi.org/10.1145/3124749.3124754

[49] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. 2020. Kraken: memory-efficient continual learning for large-scale real-time recommendations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 21. https://doi.org/10.1109/SC41405.2020.00025

[50] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 181–193.

[51] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: a key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 194–209.

[52] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *Proceedings of Machine Learning and Systems* 2 (2020), 412–428.

[53] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. Aibox: Ctr prediction model training on a single node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 319–328.

[54] Guorui Zhou, Weijie Bian, Kailun Wu, Lejian Ren, Qi Pi, Yujing Zhang, Can Xiao, Xiang-Rong Sheng, Na Mou, Xinchen Luo, et al. 2020. CAN: revisiting feature co-action for click-through rate prediction. *arXiv preprint arXiv:2011.05625* (2020).

[55] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. 2019. Deep Interest Evolution Network for Click-Through Rate Prediction. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 5941–5948. https://doi.org/10.1609/aaai.v33i01.33015941

[56] Guorui Zhou, Xiaoqiang Zhu, Chengru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep Interest Network for Click-Through Rate Prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Yike Guo and Faisal Farooq (Eds.). ACM, 1059–1068. https://doi.org/10.1145/3219819.3219823

[57] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. {Write-Optimized} and {High-Performance} Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 461–476.