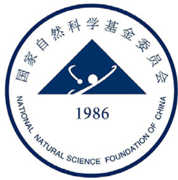
Fundamental Research 4 (2024) 651–659



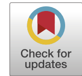
Contents lists available at ScienceDirect

Fundamental Research

journal homepage: http://www.keaipublishing.com/en/journals/fundamental-research/

Article

An efficient labeled memory system for learned indexes



Yuxuan Mo, Jingnan Jia, Pengfei Li, Yu Hua*

Huazhong University of Science and Technology, Wuhan 430074, China

ARTICLE INFO

Article history:
Received 5 December 2021
Received in revised form 7 May 2022
Accepted 9 May 2022
Available online 8 June 2022

Keywords:
Heterogeneous memory system
Cache hierarchy
Data movement
Resource contention
Learned index

ABSTRACT

The appearance and wide use of memory hardware bring significant changes to the conventional vertical memory hierarchy that fails to handle contentions for shared hardware resources and expensive data movements. To deal with these problems, existing schemes have to rely on inefficient scheduling strategies that also cause extra temporal, spatial and bandwidth overheads. Based on the insights that the shared hardware resources trend to be uniformly and hierarchically offered to the requests for co-located applications in memory systems, we present an efficient abstraction of memory hierarchies, called *Label*, which is used to establish the connection between the application layer and underlying hardware layer. Based on labels, our paper proposes LaMem, a labeled, resource-isolated and cross-tiered memory system by leveraging the way-based partitioning technique for shared resources to guarantee QoS demands of applications, while supporting fast and low-overhead cache repartitioning technique. Besides, we customize LaMem for the learned index that fundamentally replaces storage structures with computation models as a case study to verify the applicability of LaMem. Experimental results demonstrate the efficiency and efficacy of LaMem.

1. Introduction

Memory architectures are tightly related with significant changes in hardware ecosystems [1]. Memory-related hardware technologies such as Dynamic Random Access Memory (DRAM), and Non-Volatile Memory (NVM) have been introduced to meet the performance demands of systems and applications, which introduce more complex memory hierarchies. Two important issues need to be addressed to achieve resource optimization and energy saving in memory systems, i.e., the contentions for shared hardware resources and data movements among hierarchies.

Resource Contentions. Due to the wide use of heterogeneous memory devices, say DRAM, SSD and NVM, the significant changes of memory hierarchies increase the number of shared hardware resources, which intensifies the resource contention. The use of multi-core platforms is helpful to enhance the system performance and throughput, but it still suffers from the quality of service (QoS) challenge due to the contention for various shared hardware resources [2]. In order to guarantee the QoS of applications, we need to mitigate the resource contention that is non-trivial [3], due to the difficulties in achieving the resource isolation. Specifically, the contention causes unpredictable performance variability and uncertainty [4] due to the disability to accurately determine the system performance and the interference among co-located applications running on the same platform. Moreover, the shared memory hierarchies encounter the contention problem [5]. Existing way-based (e.g., Intel CAT technique [6]) and page-coloring partitioning techniques [7] are used to mitigate the resource contention. However, the way-based technique easily leads to performance degradation [3] and doesn't support cache capacity repartitioning, and the page-coloring partitioning technique incurs extra overheads of repartitioning.

Data Movements. The significant changes in memory hierarchies increase the complexity of data movements among hierarchies. To bridge the I/O performance gap between memory and CPU, the hierarchical levels in memory systems increase via adding new hardware devices [8]. However, this leads to expensive data movements among heterogeneous hardware devices [9]. In the meantime, each tier of the memory system is independent and requires case-by-case designs to manage. Data may be cached or replicated in multiple tiers, or stored only once in a single tier [1]. Hence, automatic data movements among tiers become important for both systems and users. To decrease unnecessary data movements between memory and CPU, recent advances in 3D-stacked memory technology [10,11] move processing elements closer to the data and use the logic layer to perform processing-in-memory (PIM). However, some design issues arise when adding compute logics to the memory device [12]. For example, the PIM processing logic fails to quickly access involving address translation and cache coherence in CPU, which are generally helpful for programmers.

It is challenging to achieve resource isolation and reduce data movements in memory systems. First, high-level QoS demands of applications and underlying hardware resources are unaware of each other in

* Corresponding author.
E-mail address: csyhua@hust.edu.cn (Y. Hua).

https://doi.org/10.1016/j.fmre.2022.05.016
2667-3258/© 2022 The Authors. Publishing Services by Elsevier B.V. on behalf of KeAi Communications Co. Ltd. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)

modern memory systems. This enables the shared hardware resources to be uniformly accessed by the requests of co-located applications, which however incurs resource contentions. Second, it is difficult to accurately determine the partition sizes for applications to achieve design goals (e.g., QoS or fairness). Third, existing schemes fail to fully exploit the shared resources to reduce the data movements.

In practice, due to the lack of considerations for memory hierarchies and the heterogeneous shared hardware resource contentions in multi-core systems, existing schemes fail to achieve both resource optimization and energy saving. To effectively schedule resources, a centralized platform resource manager [2] is proposed as an embedded system-on-chip that manages and monitors the hardware resources. A data placement [13] utilizes all layers to perform memory, metadata, and communication management in hierarchical buffering systems. Moreover, to convey critical messages from the application layer to the underlying hardware layer, existing designs [14,15] use labels to represent the deadline of a batch job, compute available resources (e.g., the number of CPU cores and the size of memories), or point to the data. However, these designs do not participate in handling complex data movements among memory hierarchies.

Unlike existing schemes, our paper proposes LaMem, a Labeled Memory system for alleviating shared hardware resource contentions and decreasing the data movements. LaMem presents an efficient software-hardware abstraction, called *label*, to represent the resource demands of applications. The label allows high-level applications to communicate with underlying hardware resources. Based on the labels, LaMem improves the memory system performance in two aspects, i.e., improving the execution performance of applications and reducing the number of data movements.

Specifically, LaMem leverages the way-based partitioning technique for shared resources to achieve resource isolation, while providing the on-demand cache resource offering scheme and shortcut design to deliver differentiated levels of performance. Moreover, LaMem presents a lazy-repartitioning technique to support dynamic resource adjustment with low overheads.

Hence, LaMem shows how to use the simple yet effective partitioning technique and resource allocation scheme to enable a labeled memory system to achieve high performance and reduced data movements. A learned index [16] is able to support fast query operations and delivers high performance by using trained models. The high performance actually depends on the efficient accesses upon multi-layer memory hierarchy. To verify the applicability, we exploit the characteristics of an advanced learned index named ALEX [17] and customize LaMem for improving system performance. Our paper makes the following contributions:

- ***Significant performance improvements.*** We leverage the way-partitioning to achieve resources isolation. Based on labels, we present an effective resource allocation scheme and a shortcut design to provide differentiated levels of performance and QoS guarantee. The above techniques deliver high performance in the labeled memory systems.
- ***Decreased data movements.*** We leverage the resources partitioning to decrease the data movements among co-located applications. Moreover, our proposed shortcut design is used for low-priority applications to mitigate the complexity of data movements among hierarchies. LaMem thus reduces the amounts of data movements.
- ***Real case study.*** We choose a real-world application, i.e., the learned index as a case study. We provide a shortcut scheme for ALEX in LaMem. LaMem efficiently improves the performance of ALEX according to experimental results.
- ***Implementation and evaluation.*** We implement a prototype of LaMem on GEM5 [18] simulator. Through extensive experiments, we demonstrate that LaMem achieves performance gains by about 2.1× on average, as well as reducing 43.2% of data movements compared with existing memory systems with unmanaged shared resources. LaMem thus achieves better performance and fewer data movements.

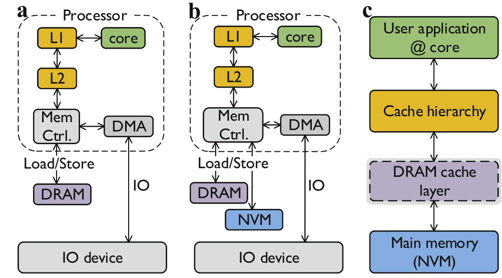


Fig. 1. The hierarchical memory systems. (a) DRAM-based memory system. (b) Hybrid DRAM/NVM memory system. (c) Logical memory hierarchy.

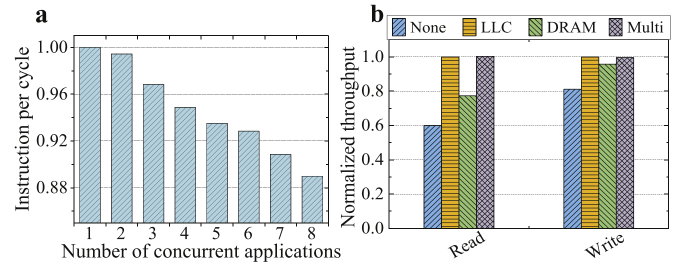


Fig. 2. The cross-application contentions and multi-tier data movements. (a) The IPC of gcc. (b) The throughput of learned indexes. *Multi* simulates both LLC and DRAM layers; *LLC* simulates only LLC layer; DRAM simulates only DRAM layer; *None* doesn't simulate any cache layers.

2. Background and motivation

2.1. Cross-application contentions

In Fig. 1a, 1b, we illustrate the DRAM-based memory system and the holistic DRAM/NVM memory architecture. It is necessary to fully exploit the benefits of DRAM and NVM technologies. There are two different architectures, including flat-addressable and hierarchical hybrid memory architectures [19]. The flat-addressable hybrid memory architecture contains DRAM and NVM in the flat-address space as the main memory [20]. In the hierarchical hybrid memory architecture, DRAM becomes a cache layer of NVM to deliver high performance [21].

Moreover, Fig. 1c illustrates the logical memory hierarchy: user applications - cache hierarchies - DRAM cache layer - main memory. In this hierarchy, the memory system needs to deal with interleaved requests from different applications, which yields more shared hardware resource contentions and data movements between adjacent layers (detailed in Section 2.2). In this paper, we mainly focus on hierarchical hybrid memory architecture to exploit performance gains, i.e., maximizing system throughput and guaranteeing the quality of service.

When applications are running concurrently in a multi-core system, they compete for various shared resources such as CPU cores, caches, memory bandwidths, and network, called *cross-application contention problem*. This problem causes severe performance degradation and resource wastes. We conduct a heuristic experiment to show how this issue occurs when various applications compete for the shared hardware resources in memory systems.

Motivating Example (i): To quantify the negative effects of cross-application contentions, we perform the motivating evaluation on fully shared LLC (Last Level Cache) that is widely used in multiple cores. We leverage the Gem5-NVMain [18,22] platform to evaluate the behaviors of LLC with the gcc application from SPEC CPU2006 [23]. We run multiple gcc applications at the same time and divide each instruction per cycle (IPC) by the ideal IPC. Fig. 2a shows the results when scaling the evaluation up to 8 concurrent applications, meaning that the sharing hurts the performance of gcc.

2.2. Data movements among multiple tiers

The wide use of heterogeneous devices, e.g., DRAM, NVM and SSD, aims to meet the demands of systems and applications. Meanwhile, the configuration space of multi-tier caching enables the design of complex systems but increases the complexity of data movements among multi-tier structures, called *multi-tier configuration problem*. Beyond the traditional layer-to-layer view of caching, we need to coordinate the access to these layers shared by multiple applications.

The learned index that supports insert operations needs to load the whole data node to execute the model retraining process, which brings significant data movements. The model retraining process can block the requests from other applications, since the large data stream fills up the cache space. With the increase of the hierarchical memory tiers, the model re-training process leads to many data movements. We evaluate the performance of different operations of the learned index including the lookup and insert under different multi-tier configurations.

Motivating Example(ii): To evaluate the performance of learned indexes under different multi-tier configurations, we use the same platform as Section 2.1 and carry out *read-only* and *write-only learned indexes* under the YCSB Benchmark [24]. Fig. 2b shows the lookup and insert throughputs of the learned index under different multi-tier configurations, which demonstrates that *adding a cache layer becomes helpful, but not always*. Therefore, in order to improve the overall performance, it is necessary for the learned index to explore an optimal multi-tier configuration.

2.3. The need for labeled memory systems

In general, *cross-application contentions* and *multi-tier configurations* are derived from the gap between high-level application semantics (e.g., resource demands) and underlying hardware in modern memory systems.

We argue that the cross-application contentions occur in the *horizontal* direction of the shared hardware resources, and multi-tier configurations occur in the *vertical* direction among the shared hardware resources. In both directions, there are complex data movements and severe performance degradation due to the unawareness of underlying shared hardware resources.

To overcome these limitations, we propose a new labeled memory system to convey high-level program semantics of applications to underlying hardware resources. In particular, we leverage labels to represent the resource demands of applications so that the shared hardware resources are aware of the high-level applications. Moreover, the label is related with the shared hardware resource partitioning, and each application acquires sufficient shared hardware resources to achieve performance improvements.

3. The design of LaMem

In order to efficiently support high-level applications, LaMem mitigates the resource contentions and expensive data movements among memory hierarchies. LaMem consists of three main components: label mechanism, resource partitioning, and dynamic resource adjustment.

3.1. The label mechanism

In this section, we discuss the semantics and the initialization of labels. Moreover, we study how to perform labels on requests and how labels propagate within the memory hierarchy.

The Initialization of Labels: We define a new hardware-software abstraction, called *label*, which serves as the basic unit of expressing the priority of application programs, to convey high-level resource demands to the low-level system hardware. The shared hardware resources are allocated for each application according to their labels. Hence, the label abstraction is the cornerstone of our labeled memory system.

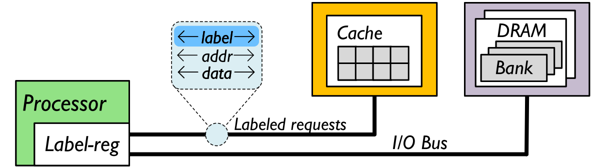


Fig. 3. Performing labels on requests.

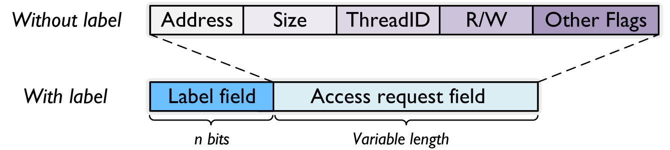


Fig. 4. The structure of a labeled request.

To allow underlying shared hardware resources to distinguish clients with different priorities, we add a label register into each CPU core to store its label (as shown in Fig. 3). The label register occupies small space (no more than 1B). Requests from each processor will be attached with corresponding labels before being sent to the cache layer.

The labels are initialized by the user in advance and written to the label-register through system software. The value of a label represents the priority of an application. To enable the memory system to provide the support for specified labels determined by users, we add one instruction into our labeled memory system, called *lstore*. This instruction is called via `<lstore reg, label>`, where *reg* is the destination label register in the core, and the *label* represents the priorities of applications from users. Before applications run, the label register of each core stores the corresponding labels via predefined settings, and the default value is zero.

The Labeled Requests: The labeled request is the basic component of LaMem. The individual fields of the labeled request are shown in Fig. 4. The conventional fields for memory access include virtual address, size, thread-id, read/write commands, and flags. Since the modern 64-bit memory bus has not been fully used up [25], there are 16 bits available in the virtual address, which can be exploited as the label.

Performing Labels on Requests: Once a core issues a request, the label stored in the label register will be appended to the corresponding request and travel along with entire lifetime. The appended label is not related with the execution correctness of requests, and only provides supplemental information to enable more flexible resource allocation (details shown in Section 3.2). As the priority of each workload may vary over time, the labeled memory system needs to provide a dynamic label update mechanism according to the characteristics of each workload for efficient service.

Propagation in Memory Hierarchies: We need to address the main design issue: *labeling multi-phase writeback requests in multi-tiered memory hierarchies*. In general, a write operation consists of multiple phases due to the multi-tiered memory hierarchies. For instance, a dirty block of the last level cache is selected to be evicted when a cache miss occurs. Moreover, the memory system writes this evicted block back to the next memory layer. During this writeback phase, we cannot determine which label is appended to the writeback request. To address this problem, we observe that it is necessary to store the label message of each request when there are multi-phase writeback requests. The label will be stored with each cache block, and these extra bits are negligible compared with the large size of cache blocks.

3.2. Shared resource allocation

Conventional LRU-based shared cache replacement does not distinguish the data blocks of each client, which potentially causes cache thrashing, unfairness and QoS problem, and increased data movements

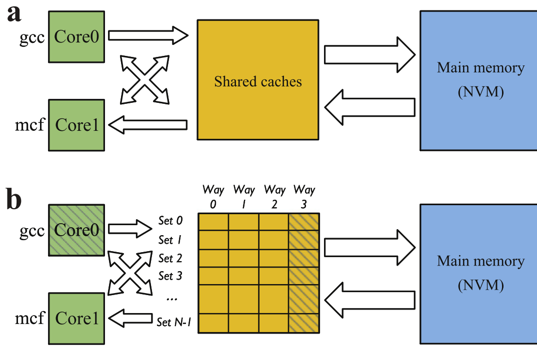


Fig. 5. Two partitioning schemes of the shared cache. (a) the traditional cache partitioning scheme. (b) the way-based cache partitioning scheme.

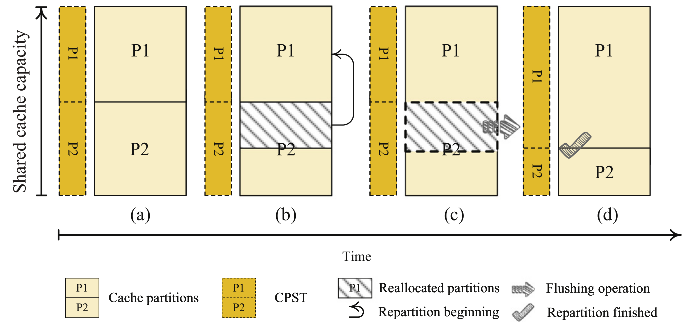


Fig. 6. The flush-based resource repartitioning process.

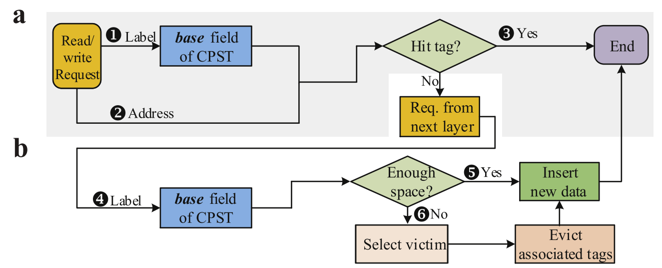


Fig. 7. Processing read/write requests before the dynamic adjustment.

among cache layers. We present an efficient cache isolation technology and on-demand resource offering scheme to mitigate the interference among different clients, thus obtaining energy savings and improving the quality of services.

We utilize the way-based partitioning scheme for the shared cache to guarantee the cache resource isolation among clients. As shown in Fig. 5, the cache partitioning scheme leverages the way (a way consists of a data block and the valid and tag bits), and we distribute ways in one cache set to different clients according to their labels.

In many real-world scenarios, there is a trade-off between offering differentiated services and fairness-oriented services for clients. We present an on-demand cache resource offering strategy to coalesce both clients' resource demands and their priorities. We first meet the demands of high-priority clients, and then process the requests from the lower priority clients. When the shared cache resources are insufficient to meet the demands for all applications, we provide an on-demand resource offering strategy by implementing different allocating policies.

Priority-based Design: Given a set of labels with different priorities, the resource allocator chooses applications to serve according to the high-priority-first principle. All applications obtain their required resource capacity in the priority order. The priority-based principle provides the flexibility to exhibit certain priorities for performance guarantees on the system. This design aims to satisfy the QoS demands of clients with high priority.

Proportion-based Design: For some applications with the same priority, if the remaining resources are insufficient to meet the clients' resource requirements, the remaining resource capacity will be divided among sharers in proportion according to their demands. This scheme aims to guarantee the fairness of equal-priority applications.

3.3. Dynamic resource adjustments

To guarantee real-time QoS, fast and low-overhead repartitioning techniques are required to meet different clients' timing requirements. We describe the following two approaches: the flush-based baseline repartitioning and our proposed low-overhead lazy-repartitioning techniques.

3.3.1. Flush-based repartitioning

Most of existing dynamic cache partitioning schemes are driven by the underlying hardware (e.g., PARD[2] and QoSMT [26]), while some are driven by the software (e.g., dCat [27]). However, they need to use the flush-based approach to remove a cache partition after changing the cache allocation explicitly. In flush-based resource repartitioning, the dynamic adjustment among cache partitions will cause extra time overhead and the performance slowdown. Fig. 6 shows the process of the adjustment.

(a) *Initial stage:* Assuming that there are two clients (e.g., A and B) co-located in the memory system, sharing the cache resource equally. They respectively occupy partitions P_1 and P_2 that are filled with their data.

(b) *Repartition beginning:* A needs more cache capacity to improve its performance and triggers the adjustment among P_1 and P_2.

(c) *Flushing operation:* When reallocating part of the P_2's capacity and distributing it to application A, we need to flush the dirty cachelines to the next cache layer or the backing memory, and then invalidate their tags. This operation will block the process of B and cause extra time overhead.

(d) *Repartition finished:* We finally modify the records of each applications' cache partition statistic table (CPST), which records the bitmask of each partition, and thus the flush-based resource repartitioning is completed.

3.3.2. Lazy-repartitioning

To address the limitations of the flush-based resource repartitioning scheme, we propose the lazy-repartitioning approach to avoid the extra time overhead caused by flush operations, which relies on the label mechanism and CPST.

As shown in Fig. 7, the cache utilizes the *base* field which records the initial cache partition of each application to process read and write requests before the dynamic adjustment. Fig. 7a illustrates the cache lookup operation when cache hits occur; each client correctly performs cache read and write within its respective initial cache partitions. ❶When the memory access request is received, the label of this request is used to index the cache partition recorded in the *base* field. ❷The address is searched in the tag array as a conventional cache. ❸If the tag hits, the request for data array will be served. Fig. 7b illustrates the cache eviction and insertion when cache misses occur; clients correctly perform possible data array evictions and the new entry insertion within their respective initial cache partitions. For example, ❹when the tag misses, a request is sent to the next cache layer. The label of the returned request is used to index the cache partition. ❺Meanwhile, the victim tag and its corresponding data array entry are evicted, and the new tag is inserted if there is not enough space there. ❻If not, the new tag is inserted immediately. The data is returned to the requested cache as soon as arriving. Inserting a new line into the cache increases the critical path since other cache access requests are involved.

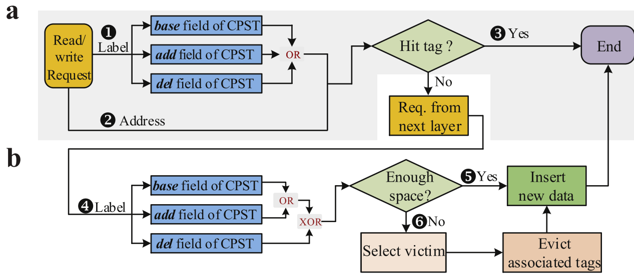


Fig. 8. Processing read/write requests during the dynamic adjustment period.

The cache access behaviors during the dynamic adjustment period are shown in Fig. 8. The *base* field records the initial cache partition of each client, and the *add* and *del* fields respectively record the increased and decreased cache partition ranges over initial values. The three records support our proposed Lazy-Repartitioning dynamic adjustment approach.

We first look up the data with the tag as shown in Fig. 8a. The difference between Figs. 8a and 7a is step❶. We adopt the lazy scheme, which first modifies the records of CPST while not achieving the real dynamic adjustment among cache partitions. Therefore, during the lookup phase, we need to check all possible areas (i.e. ***bitmask_base*** + ***bitmask_add*** + ***bitmask_del***, "+" means OR operation) indexed by the label. Fig. 8b illustrates the cache eviction and insertion when the tag misses the cache, and the difference between Figs. 8b and 7b is step❹.The newly arrived data entry needs to be handled in its adjusted cache partition (i.e.***bitmask_base*** + ***bitmask_add*** - ***bitmask_del***, "-" means XOR operation) indexed by the labels. If there is not enough space remaining, the victim segments are selected to make room for the new block, and their tags are evicted. The block is finally inserted.

4. Case study

We utilize the learned index as a case study to demonstrate the effectiveness of LaMem. We first introduce the learned index and exploit the characteristics of its different operations. We further demonstrate our implementation details and propose a shortcut scheme for the learned index.

4.1. The learned index

An index is generally used to reduce the high-latency I/Os, but the traditional index structures suffer from the performance bottleneck of frequently and inefficiently accessing indexes in large-scale memory systems. *Learned Index* [16] is proposed to address this bottleneck, which replaces the index structure with the computation models. The key insight is that the index structure can be viewed as a model that returns a predicted position with a given key. When the fixed-length key-position pairs are sorted by the keys, this model approximates the cumulative distribution function (CDF) that can be further represented by machine learning models. By using the learned index, we can significantly decrease the query latency.

ALEX is a state-of-the-art learned index that is able to support update operations, contains a tree-like structure, and preserves several gaps in advance to insert new data. ALEX structure is similar to the B+ tree, which only stores records in leaf nodes. When the filling ratio of the data node reaches the threshold, ALEX carries out the node split and expansion operations to acquire sufficient gaps. We choose the ALEX for the case study due to achieving high performance and allowing runtime insertion. Since most learned indexes have similar structures and data processing flow [28–30], LaMem is also applicable to other learned indexes.

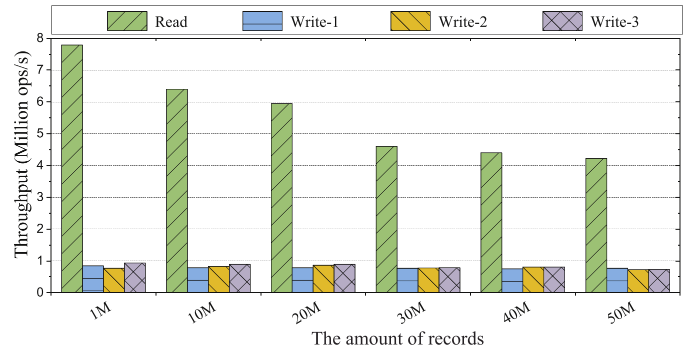


Fig. 9. The throughputs under different index sizes.

4.2. The learned index characteristics

We leverage the learned index ALEX [17] to explore the characteristics of different operations including lookup and insert. Specifically, we need to bulk load it with a batch of records first to initialize the index structure and further allow it to execute lookup or insert operations. The details about the experimental setup are shown in 5.1.

Observation 1: Insertion requires more data accesses. For insert operations, the learned index carries out the lookup operation with the given key. When failing to obtain the value, the index attempts to insert the records to the predicted position. As a result, the insert operation needs to read more data from the cache and write more data to cache, compared with lookup operations.

Observation 2: The lookup performance is related with the data size of the index. We load the ALEX with one million, ten million, twenty million, thirty million, forty million, and fifty million records. After the initialization, we send one hundred thousand lookup requests to the learned index continuously and calculate the throughput. Fig. 9 shows that the lookup throughput of the learned index decreases with the increase of database sizes.

Observation 3: The performance of insert is lightly related with operation amounts. For Write-1, Write-2 and Write-3, we first load ALEX with one thousand, ten thousand, and one hundred thousand records respectively, and then insert records into ALEX to reach the set index size. We observe that the insert throughput is smooth when the number of insert operations increases as shown in Fig. 9. We also observe that the insert throughput of Write 1 is slightly lower than others because its index structure is more possible to be changed.

4.3. Shortcut scheme

We use a shortcut scheme for the multi-client learned index based on the above observations. We assume that each client runs one learned index with a single operation in each core. Taking the following steps to implement LaMem for the learned index. (1) Before partitioning, LaMem first evaluates and collects the characteristics of the initial database of each client in an offline way. Specifically, LaMem gathers the throughput curves of clients with lookup operations under candidate cache allocations. The insert operation contains a similar process as the lookup at the beginning, and the insert performance is not affected by the operation amount. Hence, it is reasonable to profile the clients through executing lookup operations to obtain the approximate demands. (2) We distribute the higher priority to the client with high write frequency according to Observation 1. (3) For the read-only client with a smaller data size than the threshold, we bypass their data requests around the current cache layer to save resources. But for clients with high write frequencies and the initial data sizes are still under the threshold, we attach them with the modest priority. (4) Besides, if vacant space remains when meeting all the needs of clients, we will distribute it to the clients with high priority. After that, we finish the implementation process.

Table 1
The experimental configurations.

Hardware	Configurations
CPU	8 8-issue In-Order X86 cores, 2GHz
L1 cache	32KB 2-way, hit = 2 cycles
L1 cache	1MB 8-way, hit = 10 cycles
Shared LLC	8MB 16-way, LRU, hit = 20 cycles
DRAM	256MB: FR-FCFS request scheduling 1 channel, 1 rank/channel, 8 banks/rank Burst Length = 8, Row buffer = 1KB 70 ns read latency, 70 ns write latency
NVM-Memory	8GB: FR-FCFS request scheduling 4 channel, 1 rank/channel, 8 banks/rank Burst Length = 8, Row buffer = 1KB 150 ns read latency, 300 ns write latency

Table 2
Applications of the LaMem (CS = Cache-Sensitive, ST = Streaming).

Class	Applications
CS	soplex, omnetpp, lbm, gcc
ST	milc, libquantum, hmmer, namd, calculix, leslie3d, gromacs, bzip2

5. Evaluation and analysis

We evaluate the performance of the learned index implemented in LaMem compared with the performance in hierarchical memory structure with different resource management schemes. Besides, we also evaluate LaMem with 12 representative applications from SPEC benchmark, because LaMem is general enough and can be customized to other applications.

5.1. Experimental setup

We use Gem5-NVMain [18,22] platform to evaluate LaMem. NVMain is a cycle-accurate main memory simulator for NVM technologies. The NVM system consists of x86-64 processors running at 2 GHz, 32 KB L1 data and instruction caches, and 8 MB shared last level cache as described in Table 1. Without loss of generality, we model PCM technologies [31] with 8 GB capacity and DRAM cache with 256 MB [32].

Applications. We utilize the open-sourced learned index ALEX [17] as the application used for this evaluation. Each core runs a learned index executing different operations, including insert and lookup upon the YCSB dataset. We have also used 12 representative applications from the SPEC benchmark suite [23] to evaluate LaMem. As shown in Table 2, these applications are classified into cache-sensitive (CS) or streaming (ST), similar to prior works [3,33]. The performance of the CS applications changes significantly over the equal-share baseline as cache size varies. The applications belonging to the ST category still achieve the same level of performance as using all cache resources when the resource contention exists. We execute ten different multiple programmed workloads (i.e., coalescence). Each coalescence consists of 8 applications randomly selected (with replacement) from our pool of applications. They are co-located in the memory system, and each application runs with ten billion instructions.

Comparisons. To demonstrate the performance gains from the proposed techniques, we compare LaMem with the following schemes:

• *A memory system with unmanaged shared cache allocation (**SHARE**).* It refers to the baseline configuration, where the shared hardware resources are left unpartitioned and shared by all applications pinned into each core.
• *A scheme for equal-sharing cache allocation (SE)* [34]. It demonstrates an equally-partitioning technique of the shared cache capacity between on-chip cores. The scheme guarantees fairness because all applications obtain the same capacity across different schedules.

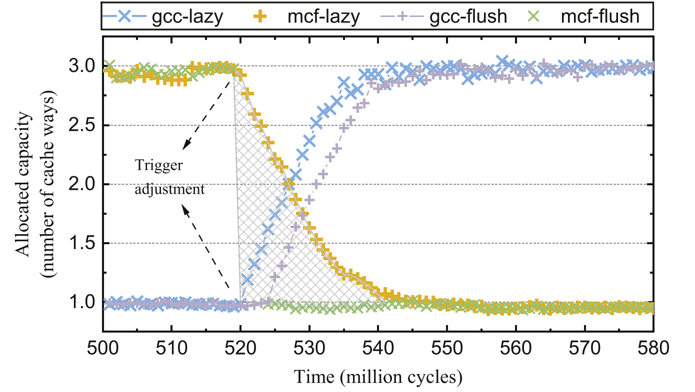


Fig. 10. Cache Allocation for `gcc-mcf`.

• *A scheme for proportional-sharing cache allocation (SP)* [35]. Applications with different priority levels provide service differentiation as opposed to fairness. Note that SE corresponds to a special case of proportional-sharing when all applications have the same priority.

Metrics. We measure system performance using the following metrics: throughput metric, QoS metric and data movements metric.

Throughput Metric. Throughput metric is the number of operations that the learned index can process per second. Prior works focusing on learned indexes view the throughput as the primary metric. In this work, we calculate the average throughput to measure the performance gains of the leaned index.

QoS Metric. We use equal-share cache allocation to define the performance bottom line for QoS, which provides baseline QoS results for multi-processor users. The QoS metric is defined as the sum of per-application performance slowdowns (as negative percentages) over this baseline [34]. We use the throughput to evaluate the performance of the learned index and the IPC (Instruction Per Cycle) to evaluate the performance of the SPEC benchmark.

Data Movements Metric. To quantify the impact of different schemes on the complexity of data movements, we use the BPI (Bytes Per Instructions) metric for each coalescence, where S_{inst} represents the sum of co-running instructions, $S_{cacheline}$ and S_{page} respectively represent the amounts of 64 B cacheline and 4 KB pages movements. 64 B cacheline movements occur respectively in LLC and DRAM cache, LLC and NVM. The 4 KB page movements occur between DRAM cache and NVM:

$$BPI = \frac{1}{S_{inst}}\left(S_{cacheline} \cdot 64 + S_{page} \cdot 4096\right) \tag{1}$$

All the experiments run on a Linux server (Ubuntu 18.04.3), which has two 26-core Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz CPUs (each core with 32KB L1 instruction cache, 32KB L1 data cache, and 1MB L2 cache), 32MB last level cache and 8GB DRAM.

5.2. Lazy-repartitioning

To guarantee real-time QoS, fast and low-overhead repartitioning is used for meeting different application's timing requirements.

We trigger the cache capacity adjustment between the application gcc and mcf. Before adjustment, `gcc` and `mcf` are allocated on average 3 ways and 1 way of cache capacity respectively to better fit their requirements. Temporal resource repartitioning occurs during simulation time 520 to 540 million cycles, when `gcc` enters a phase that needs more capacity. Fig. 10 shows the cache resources repartitioning process. Flush-based repartitioning produces extra time overhead, which means that `mcf` only executes cacheline flush operations and blocks program-mcf processing during this period, while producing extra time overhead and waste of cache resources. However, lazy-repartitioning adapts to this change quickly without time overhead, and the shaded

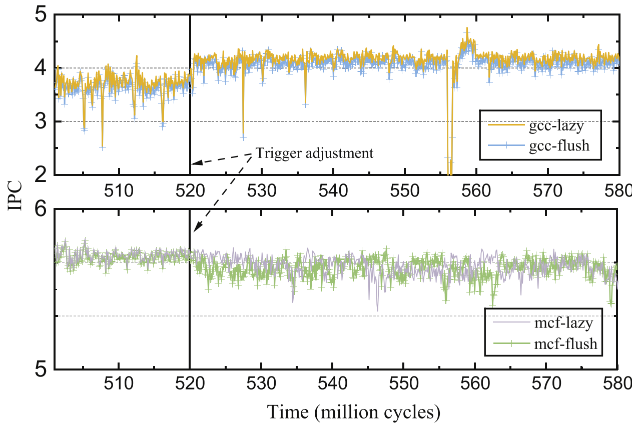


Fig. 11. The change of IPC curve.

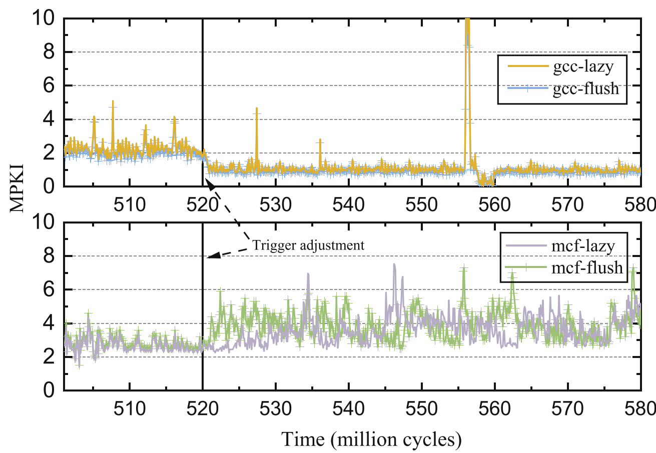


Fig. 12. The change of MPKI curve.

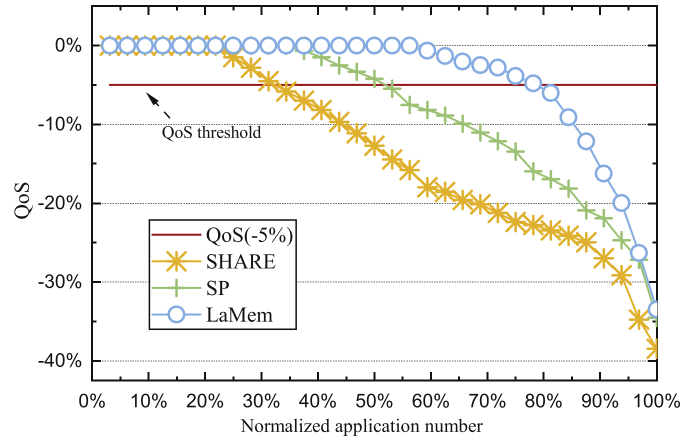


Fig. 13. The curves of the QoS guaranteed schemes.

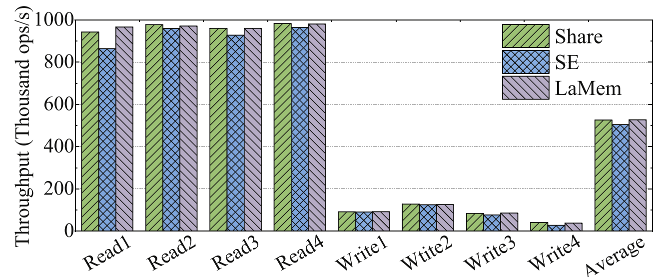


Fig. 14. The throughput of learned indexes.

area is the performance gains over flush-based repartitioning, since lazy-repartitioning makes full use of reallocated resources during the adjustment period.

The IPC curves under flush-based repartitioning and lazy-repartitioning schemes are shown in Fig. 11. Specifically, `gcc`-lazy and `gcc`-flush have almost the same IPC results because `gcc` is allocated more capacity without the interference of repartitioning. However, the IPC of `mcf`-lazy decreases more slowly than `mcf`-flush after triggering adjustment, since the reallocated capacity of `mcf` is fully used under the lazy-repartitioning scheme during the adjustment period. Similarly, Fig. 12 depicts that `mcf`-lazy and `mcf`-flush have almost the same MPKI (Misses Per Kilo-Instruction) results, and the MPKI of `mcf`-lazy increases more slowly than `mcf`-flush after triggering adjustment.

5.3. QoS Guarantee

QoS demonstrates the ability of the memory system to provide an application with guaranteed baseline performance, and we use equal-share cache allocation to define the performance bottom line for QoS within the user-specified threshold (-5% in this paper). For each scheme, we plot the percentage of applications or coalescences using QoS metric values (Fig. 13). These curves are essentially Cumulative Distribution Functions (CDF), so that a higher curve indicates a better performing scheme. We find that LaMem can guarantee the QoS for 80% of the applications, while only 30% in SHARE reach the QoS threshold, since LaMem can use labels to convey higher-level program semantics of applications to underlying hardware resources, to guarantee differentiated levels of performance.

5.4. System throughput

Eight learned indexes run concurrently, and each executes insert or lookup operations. Read 1–4 execute lookup operations and Write 1-4 execute insert operations. Fig. 14 shows the performance of each learned index under different schemes. SHARE has comparable performance to LaMem since not every learned index has been initialized at the same time. We observe that LaMem achieves the best performance and is much better than SE because the shared cache resources are divided according to their demands, and low priority requests bypass the LLC or DRAM cache layer. When the learned index becomes large-scale with frequent insert operations, it requires more memory resources and results in performance degradation. Simple resource partition scheme like SE can not meet their demands, while SHARE is unable to cope with resource contention problems.

5.5. Decreased data movements

We count the amounts of data movements for four coalescences under different schemes (Fig. 15). These results show that SE, SP and LaMem respectively reduce the amount of data movements by 21.5%, 27.3%, and 43.2% on average over SHARE scheme, since LaMem leverages the resources partitioning to decrease the data movements of co-located applications. Moreover, the proposed shortcut design is used for low-priority applications to mitigate the complexity of data movements among hierarchies. LaMem thus reduces the amounts of data movements.

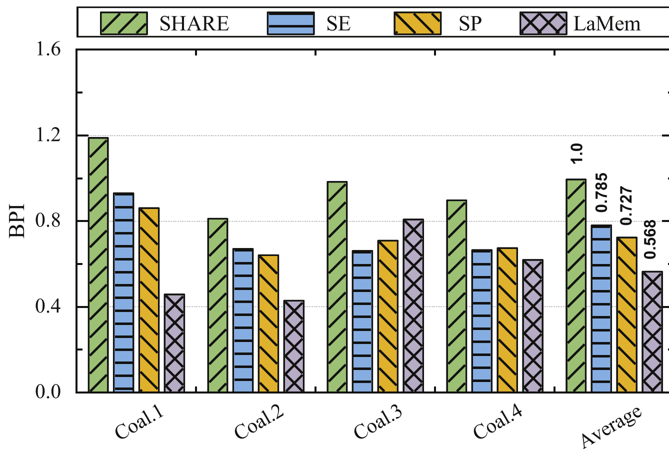


Fig. 15. The decreased data movements.

6. Related works

Resources-aware Scheduling: TPShare [14] proposes a vertically coordinated scheduling solution to enable the coordination of schedulers from different horizontal layers, which reduces resource waste and performance interference due to independent time-sharing or space-sharing scheduling of different layers under cloud application frameworks. LABIOS [15] presents a distributed, label-based I/O system to support many I/O workloads under a single storage system (unique I/O requirements lead to a proliferation of different storage devices and software stacks, many of which have conflicting requirements.). However, the scheduler in LABIOS fails to address the QoS problems. PARD [2] presents a centralized platform with a resource manager in storage systems to manage the shared resources. Unlike them, our work focuses on label-based resources management to decrease data movements in memory systems.

Data Movements in Memory Hierarchy: Given the high cost of data movements in modern memory systems, the energy and performance costs of moving data between memory and computation units are significantly higher than the computation costs [9,36]. Recent advances in 3D-stacked memory technology provide an opportunity to avoid unnecessary data movements between memory and the CPU [37]. The logic layer [37] is used to perform processing-in memory (PIM), also known as near-data processing [10,11], where some computations are transferred from the CPU to the logic layer underneath the memory layer. Promising progress has shown in in-memory and near-memory processing, reducing data movements by moving elements closer to the data. However, many important issues arise when adding compute logic to memory devices [12]. In our paper, LaMem presents an efficient resource allocation scheme and mitigates the complexity of data movements among hierarchies.

7. Discussion

While this paper demonstrates that LaMem can efficiently improve the performance and decrease the data movements from the underlying hardware layer, the following issues are needed to be discussed, including:

(i). The prototype of LaMem. Each application is pinned to the different core, and all requests from the located application of each core will be attached into the same label message, like [2,3,6]. The prototype of LaMem can be optimized to support multiple threads or attached to different operations in the learned index. A more intelligent label mechanism will be implemented in future work.

(ii). Memory overhead. The memory overhead comes from the label register of the CPU and the cache partition statistics table (CPST). As previously discussed, the label register is tiny (1B) in each core, and CPST needs only 1KB for eight applications. The cache storage overhead depends on the number of processor cores. For example, on a machine with a 16-cores processor, we only need 5 bits (4 bits for distinguishing different applications and 1 bit for setting priorities) to provide differentiated services, which introduces 0.98% ($5/(64 \times 8) = 0.0098$) extra cache resources compared with the original cache.

(iii). Comparisons with real hardware. PARD [2] and Labeled RISC-V [5] exist in the architecture level fully driven by the underlying hardware, which provide a new programming interface to convey an application's high-level information to the hardware to manage the shared resources, and introduce a per-computer centralized platform resource manager to control resource allocation and trigger resource adjustments. However, unlike them, our design in the system level enables new functionalities, like middleware-supported differentiated services and dynamic resource adjustments in memory systems. LaMem aims to build a resource-isolated and cross-tiered memory system with a fast and low-overhead cache repartitioning technique to reduce the number of data movements.

8. Conclusion

In order to mitigate resource contentions and expensive data movements among increasing memory hierarchies, we present a simple yet effective hardware-software abstraction, called *Label*, which bridges the semantic gap between the application layer and the underlying hardware layer. Moreover, our proposed LaMem presents a lazy-repartitioning technique to avoid the extra time overhead and performance slowdown during the dynamic adjustment period. We also customize LaMem for a learned index as a case study to demonstrate the efficiency and applicability of LaMem. Extensive experimental results show that LaMem efficiently improves the performance of applications with few data movements compared with existing memory systems.

Declaration of competing interest

The authors declare that they have no conflicts of interest in this work.

Acknowledgment

This work was supported in part by National Natural Science Foundation of China (62125202).

References

[1] G. Amvrosiadis, A. R. Butt, V. Tarasov, E. Zadok, M. Zhao, Data storage research vision 2025(2019). https://dl.acm.org/doi/book/10.5555/3316807.
[2] J. Ma, X. Sui, N. Sun, et al., Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (pard), in: ACM SIGPLAN Notices, 50, 2015, pp. 131–143.
[3] N. El-Sayed, A. Mukkara, P.-A. Tsai, et al., Kpart: A hybrid cache partitioning-sharing technique for commodity multicores, in: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2018, pp. 104–117.
[4] J. Mars, L. Tang, M.L. Soffa, Directly characterizing cross core interference through contention synthesis, in: Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, 2011, pp. 167–176.
[5] Z. Yu, J.M. Bowen Huang, N. Sun, et al., Labeled RISC-V: A new perspective on software-defined architecture, Workshop on Computer Architecture Research with RISC-V(CARVV), 2017.
[6] C. Intel, Improving Real-time Performance by Utilizing Cache Allocation Technology, Intel Corporation, 2015.
[7] L. Liu, Z. Cui, M. Xing, et al., A software memory partition approach for eliminating bank-level interference in multicore systems, in: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, 2012, pp. 367–376.
[8] J. Bent, G. Grider, B. Kettering, et al., Storage challenges at los alamos national lab, in: IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), 2012, pp. 1–5.
[9] A. Boroumand, S. Ghose, Y. Kim, et al., Google workloads for consumer devices: Mitigating data movement bottlenecks, in: ACM SIGPLAN Notices, 53, 2018, pp. 316–331.

[10] X. Tang, O. Kislal, M. Kandemir, et al., Data movement aware computation partitioning, in: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, 2017, pp. 730–744.
[11] C. Xie, S.L. Song, J. Wang, X. Fu, et al., Processing-in-memory enabled graphics processors for 3d rendering, in: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017, pp. 637–648.
[12] S. Ghose, K. Hsieh, A. Boroumand, et al., Enabling the adoption of processing-in-memory: Challenges, mechanisms, future research directions, arXiv:1802.00320(2018).
[13] A. Kougkas, H. Devarajan, X.-H. Sun, Hermes: A heterogeneous-aware multi-tiered distributed i/o buffering system, in: Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, 2018, pp. 219–230.
[14] Y. Wang, L. Li, Y. Wu, et al., TPShare: A Time-space sharing scheduling abstraction for shared cloud via vertical labels, 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), Phoenix, AZ, USA, 2019, pp. 499-512.
[15] A. Kougkas, H. Devarajan, J. Lofstead, et al., Labios: A distributed label-based i/o system, in: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, 2019, pp. 13–24.
[16] T. Kraska, A. Beutel, E.H. Chi, et al., The case for learned index structures, in: Proceedings of the 2018 ACM SIGMOD, 2018, pp. 489–504.
[17] J. Ding, U.F. Minhas, J. Yu, et al., Alex: An updatable adaptive learned index, in: Proceedings of the 2020 ACM SIGMOD, 2020, pp. 969–984.
[18] N. Binkert, B. Beckmann, G. Black, et al., The gem5 simulator, ACM SIGARCH Comput. Arch. News 39 (2) (2011) 1–7.
[19] H. Liu, Y. Chen, X. Liao, et al., Hardware/software cooperative caching for hybrid dram/NVM memory architectures, in: Proceedings of the International Conference on Supercomputing, 2017, p. 26.
[20] L.E. Ramos, E. Gorbatov, R. Bianchini, Page placement in hybrid memory systems, in: Proceedings of the international conference on Supercomputing, 2011, pp. 85–95.
[21] M.K. Qureshi, V. Srinivasan, J.A. Rivers, Scalable high performance main memory system using phase-change memory technology, in: ACM SIGARCH Computer Architecture News, 37, 2009, pp. 24–33.
[22] M. Poremba, T. Zhang, Y. Xie, Nvmain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems, IEEE Comput. Arch. Lett. 14 (2) (2015) 140–143.
[23] J.L. Henning, Spec cpu2006 benchmark descriptions, ACM SIGARCH Comput. Arch. News 34 (4) (2006) 1–17.
[24] B.F. Cooper, A. Silberstein, E. Tam, et al., Benchmarking cloud serving systems with YCSB, in: Proceedings of the 1st ACM Symposium on Cloud Computing, 2010, pp. 143–154.
[25] P. Guide, Intel® 64 and ia-32 architectures software developers manual, Volume 3B: System programming Guide, Part 2 (2011) 5.
[26] X. Jin, Y. Zhou, B. Huang, et al., Qosmt: Supporting precise performance control for simultaneous multithreading architecture, in: Proceedings of the ACM International Conference on Supercomputing, 2019, pp. 206–216.
[27] C. Xu, K. Rajamani, A. Ferreira, et al., dcat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service, in: Proceedings of the Thirteenth EuroSys Conference, 2018, pp. 14:1–14:13.
[28] C. Tang, Y. Wang, Z. Dong, et al., Xindex: A scalable learned index for multicore data storage, in: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2020, pp. 308–320.
[29] A. Kipf, R. Marcus, A. van Renen, et al., Radixspline: A single-pass learned index, in: Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, 2020, pp. 1–5.
[30] A. Galakatos, M. Markovitch, C. Binnig, et al., Fiting-tree: A data-aware index structure, in: Proceedings of the 2019 ACM SIGMOD, 2019, pp. 1189–1206.
[31] H.-S.P. Wong, S. Raoux, S. Kim, et al., Phase change memory, Proc. IEEE 98 (12) (2010) 2201–2227.
[32] N. Kurd, M. Chowdhury, E. Burton, et al., Haswell: A family of ia 22 nm processors, IEEE J. Solid-State Circt. 50 (1) (2014) 49–58.
[33] D. Sanchez, C. Kozyrakis, Vantage: Scalable and efficient fine-grain cache partitioning, in: Proceedings of the 38th Annual International Symposium on Computer Architecture, 2011, pp. 57–68.
[34] J. Chang, G.S. Sohi, Cooperative cache partitioning for chip multiprocessors, in: ACM International Conference on Supercomputing 25th Anniversary Volume, 2007, pp. 402–412.
[35] H. Cook, M. Moretó, S. Bird, et al., A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness, in: The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013, 2013, pp. 308–319.
[36] C. Lefurgy, K. Rajamani, F. Rawson, et al., Energy management for commercial servers, Computer 36 (12) (2003) 39–48.
[37] D. Lee, S. Ghose, G. Pekhimenko, et al., Simultaneous multi-layer access: Improving 3d-stacked memory bandwidth at low cost, ACM Trans. Arch. Code Optim. 12 (4) (2016) 63.

Yuxuan Mo received the B.E degree from Huazhong University of Science and Technology (HUST) in 2021. She is currently a PhD student majoring in computer system and architecture at HUST. Her current research interests include learned indexes, persistent memory systems and data deduplication. Her work has received the Best Paper Award in IEEE HPCC 2021.

Yu Hua (BRID: 09580.00.59361) is a professor in Huazhong University of Science and Technology. He obtained his B.E and Ph.D degrees respectively in 2001 and 2005. His research interests include storage systems, non-volatile memory architectures, etc. His papers have been published in major conferences and journals, including *OSDI*, *FAST*, *MICRO*, *ASPLOS*, *ACM TOS*, *ACM TACO*, *IEEE TC*, *IEEE TPDS*. He serves as PC (vice) Chairs in ICDCS 2021, ACM APSys 2019, and ICPADS 2016. He is the distinguished member of CCF, and senior member of ACM and IEEE. He has been selected as the Distinguished Speaker of ACM and CCF.