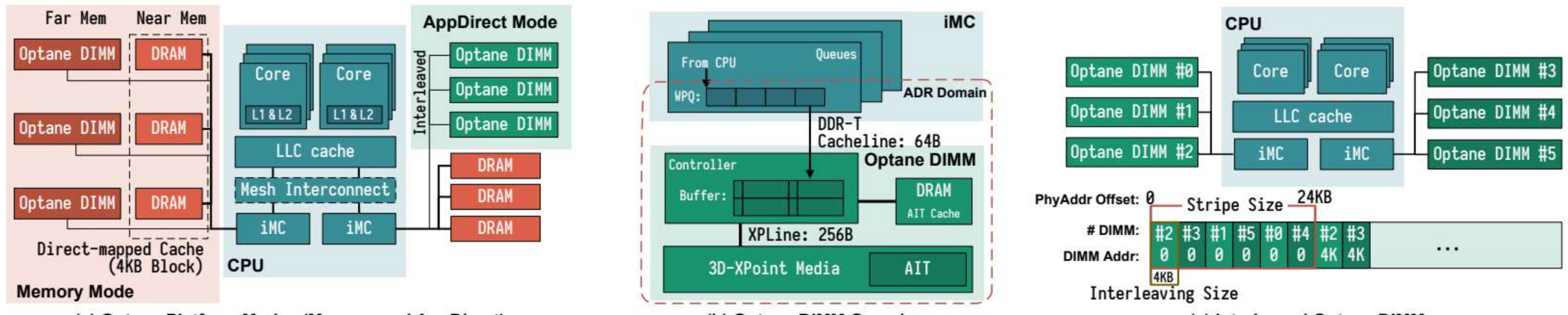
SwapKV: A Hotness Aware In-memory Key-Value Store for Hybrid Memory Systems

Journal：TKDE 2021

Authors: Lixiao Cui, Kewen He, Yusen Li, Peng Li, Jiachen Zhang, Gang Wang, and Xiaoguang Liu
University: Nankai University

Background

- **Persistent Memory**
  - Intel's Optane DIMM



(a) Optane Platform Modes (Memory and AppDirect) (b) Optane DIMM Overview (c) Interleaved Optane DIMMs

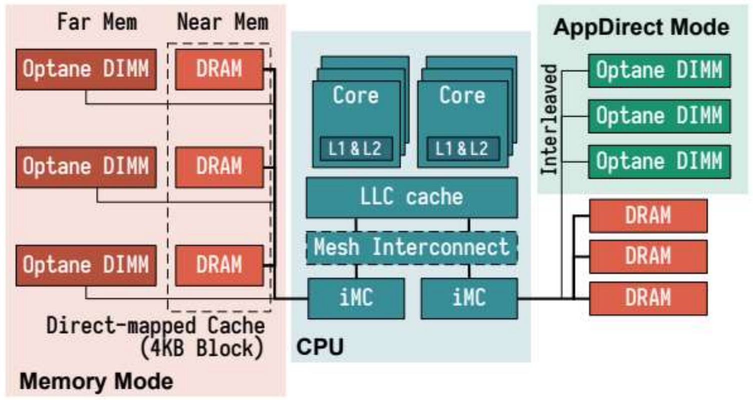
- ✓ iMC: integrated memory controller (sitting within the asynchronous DRAM refresh(**ADR**) domain)
- ✓ The iMC maintains **RPQs** and **WPQs** for each of the Optane DIMMs.

- ✓ **Memory Mode**: Optane DIMM as a larger volatile portion of main memory
- ✓ **AppDirect Mode**: Optane DIMM as a separate persistent memory device

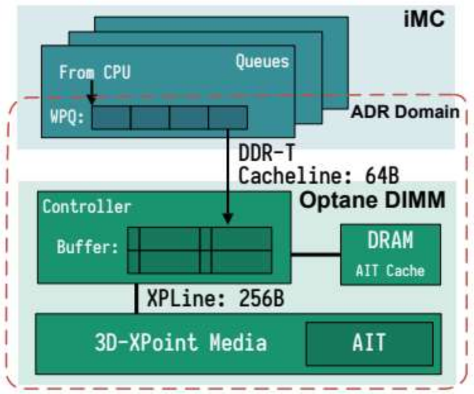
Background

➢ Persistent Memory

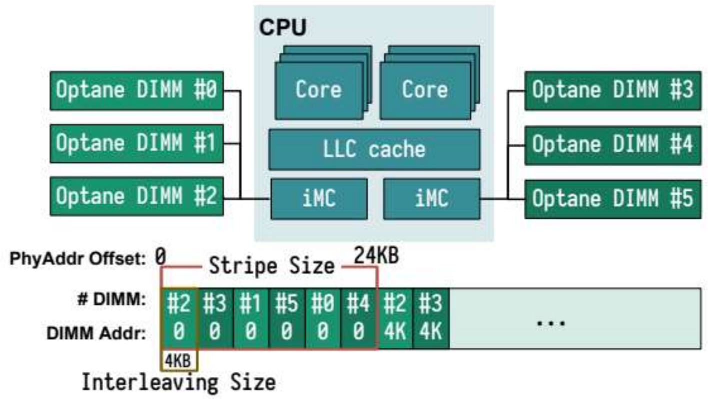
- Intel’s Optane DIMM



(a) Optane Platform Modes (Memory and AppDirect)



(b) Optane DIMM Overview



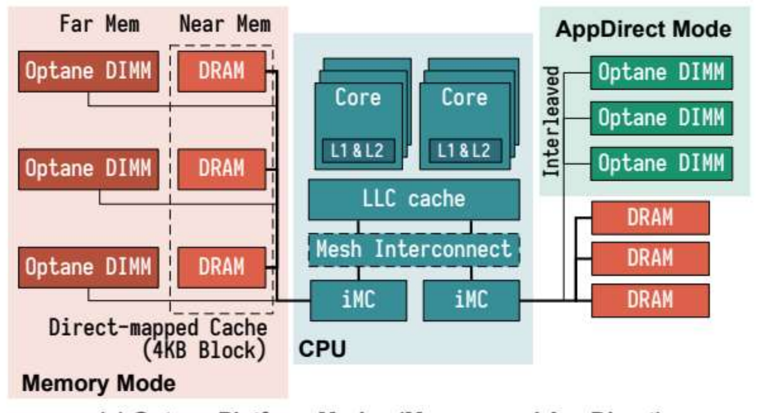
(c) Interleaved Optane DIMMs

- ✓ Memory accesses to the NVDIMM arrive first at the **on-DIMM controller**, which coordinates access to the Optane media. Similar to SSDs, the Optane DIMM performs an internal address translation for wear-leveling and bad-block management, and maintains an **address indirection table (AIT)** for this translation.
- ✓ 3D-XPoint physical media access granularity is **256 bytes**.
- ✓ On-DIMM controller has a small write-combining **buffer** to merge adjacent writes.

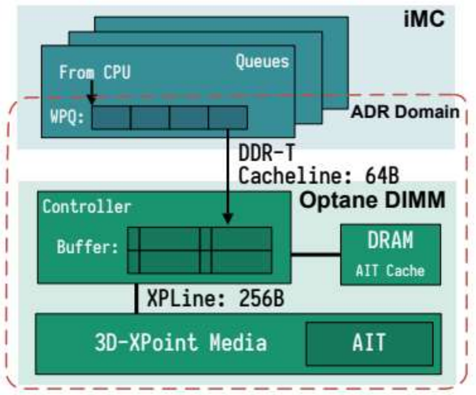
Background

➢ Persistent Memory

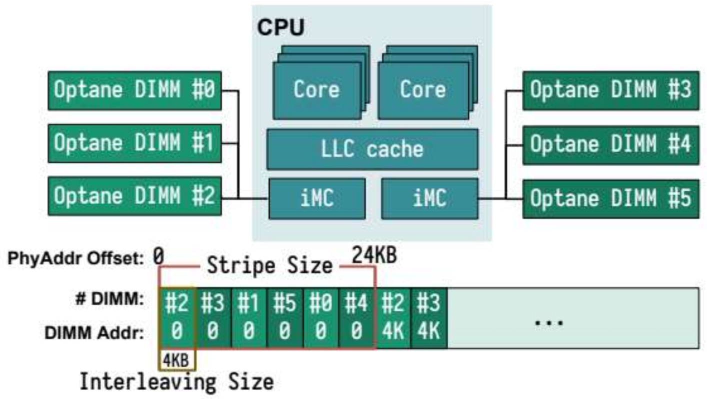
- Intel’s Optane DIMM



(a) Optane Platform Modes (Memory and AppDirect)



(b) Optane DIMM Overview



(c) Interleaved Optane DIMMs

✓ CPUs supporting Optane DIMM:
  - ☐ Intel’s Cascade Lake processors (first)
  - ☐ Intel’s Xeon Platinum/Gold processors (current)

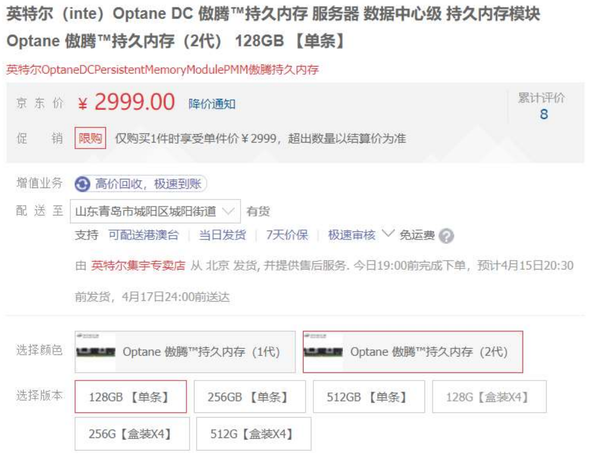
✓ Each processor contains **one or two** processor dies which comprise separate NUMA nodes. Each processor die has **two** iMCs, and each iMC supports **three** channels.

✓ The only supported interleaving size is 4 kB, which ensures that accesses to a single page fall into a single DIMM. With six DIMMs, an access larger than 24 kB will access all the DIMMs.

Background

➢ Persistent Memory

- Intel’s Optane DC PMEM


英特尔（inte）Optane DC 傲腾™持久内存 服务器 数据中心级 持久内存模块 Optane 傲腾™持久内存（2代） 128GB 【单条】

英特尔OptaneDCPersistentMemoryModulePMM傲腾持久内存

京 东 价 ￥2999.00 降价通知 累计评价 8

促　　销 限购 仅购买1件时享受单件价￥2999，超出数量以结算价为准

增值业务 高价回收，极速到账

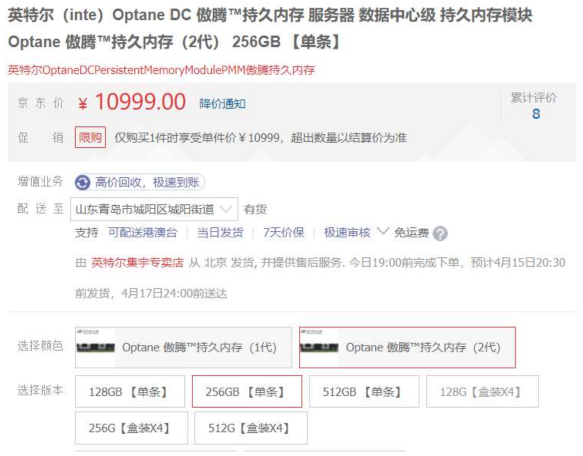
配 送 至 山东青岛市城阳区城阳街道 有货

支持 可配送港澳台 | 当日发货 | 7天价保 | 极速审核 免运费

由 英特尔集宇专卖店 从 北京 发货, 并提供售后服务. 今日19:00前完成下单，预计4月15日20:30前发货，4月17日24:00前送达

选择颜色 Optane 傲腾™持久内存（1代） [✓ Optane 傲腾™持久内存（2代）]

选择版本 [✓ 128GB 【单条】] 256GB 【单条】 512GB 【单条】 128G【盒装X4】 256G【盒装X4】 512G【盒装X4】


英特尔（inte）Optane DC 傲腾™持久内存 服务器 数据中心级 持久内存模块 Optane 傲腾™持久内存（2代） 256GB 【单条】

英特尔OptaneDCPersistentMemoryModulePMM傲腾持久内存

京 东 价 ￥10999.00 降价通知 累计评价 8

促　　销 限购 仅购买1件时享受单件价￥10999，超出数量以结算价为准

增值业务 高价回收，极速到账

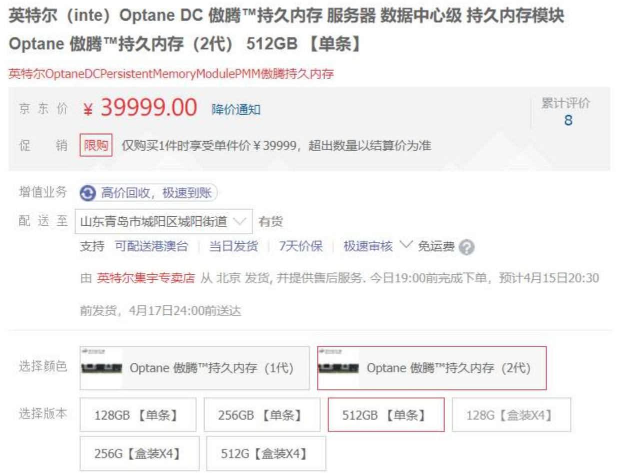
配 送 至 山东青岛市城阳区城阳街道 有货

支持 可配送港澳台 | 当日发货 | 7天价保 | 极速审核 免运费

由 英特尔集宇专卖店 从 北京 发货, 并提供售后服务. 今日19:00前完成下单，预计4月15日20:30前发货，4月17日24:00前送达

选择颜色 Optane 傲腾™持久内存（1代） [✓ Optane 傲腾™持久内存（2代）]

选择版本 128GB 【单条】 [✓ 256GB 【单条】] 512GB 【单条】 128G【盒装X4】 256G【盒装X4】 512G【盒装X4】


英特尔（inte）Optane DC 傲腾™持久内存 服务器 数据中心级 持久内存模块 Optane 傲腾™持久内存（2代） 512GB 【单条】

英特尔OptaneDCPersistentMemoryModulePMM傲腾持久内存

京 东 价 ￥39999.00 降价通知 累计评价 8

促　　销 限购 仅购买1件时享受单件价￥39999，超出数量以结算价为准

增值业务 高价回收，极速到账

配 送 至 山东青岛市城阳区城阳街道 有货

支持 可配送港澳台 | 当日发货 | 7天价保 | 极速审核 免运费

由 英特尔集宇专卖店 从 北京 发货, 并提供售后服务. 今日19:00前完成下单，预计4月15日20:30前发货，4月17日24:00前送达

选择颜色 Optane 傲腾™持久内存（1代） [✓ Optane 傲腾™持久内存（2代）]

选择版本 128GB 【单条】 256GB 【单条】 [✓ 512GB 【单条】] 128G【盒装X4】 256G【盒装X4】 512G【盒装X4】

Background

- **Persistent Memory**
  - Performance

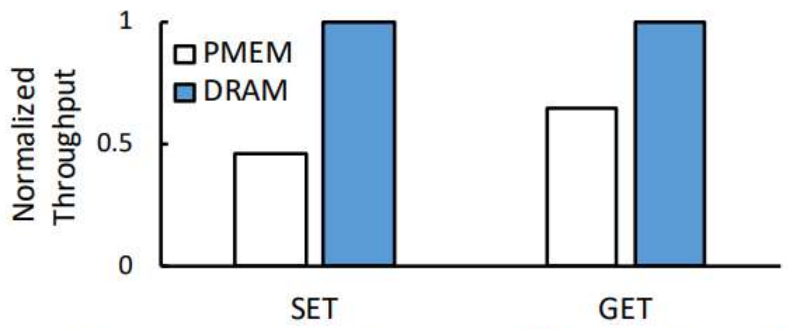


Fig. 1: Performance gap between DRAM and PMEM

Table 1: Bandwidth (in GB/s) and Latency (in ns) of DRAM, PMem, and SSD for 32 threads measured on our server[1].

	READ			WRITE		
	BW	Latency		BW	Latency	
	max	Seq	Rnd	max	Seq	Rnd
DRAM	100	40	190	70	110	170
PMem	40	50	450	13	230	900
SSD	1	115k	130k	1	125k	125k

Background

Persistent Memory

- Performance

Property	DRAM	PMEM
Read latency(ns)	101	305
Write latency(ns)	86	90
Read bandwidth(GB/s)	115	39
Write bandwidth(GB/s)	79	14

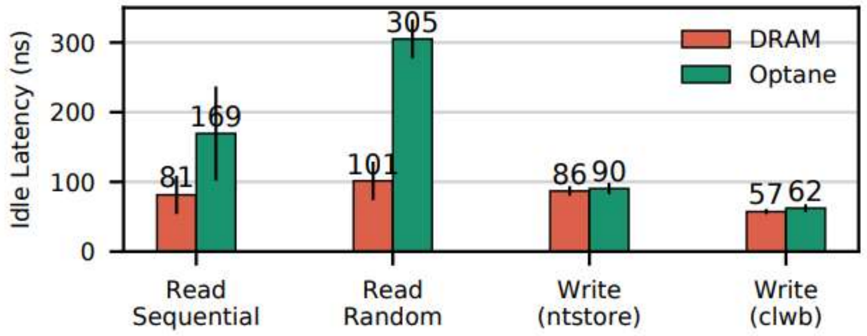


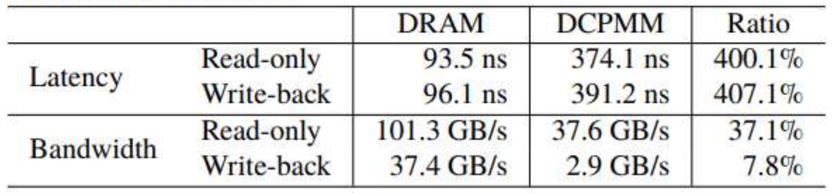
Figure 2: **Best-case latency** An experiment showing random and sequential read latency, as well as write latency using cached write with `clwb` and `ntstore` instructions. Error bars show one standard deviation.

We measured read latency by timing the average latency for individual 8-byte load instructions to sequential and random memory addresses. To eliminate caching and queuing effects, we empty the CPU pipeline and issue a memory fence (mfence) between measurements (mfence serves the purpose of serialization for reading timestamps).
For writes, we load the cache line into the cache and then measure the latency of one of two instruction sequences: a 64-bit store, a clwb, and an mfence; or a non-temporal store followed by an mfence.

Background

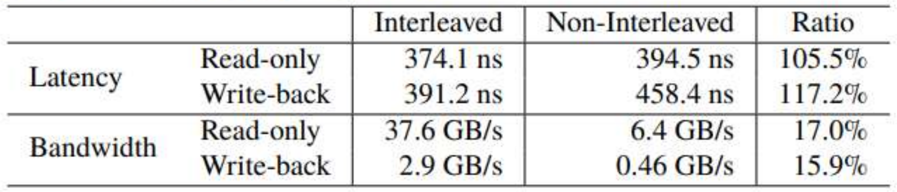
- **Persistent Memory**
  - Performance

Table 2: The obtained performance numbers of interleaved DRAM and DCPMM



		DRAM	DCPMM	Ratio
Latency	Read-only	93.5 ns	374.1 ns	400.1%
	Write-back	96.1 ns	391.2 ns	407.1%
Bandwidth	Read-only	101.3 GB/s	37.6 GB/s	37.1%
	Write-back	37.4 GB/s	2.9 GB/s	7.8%

Table 3: The obtained performance numbers of interleaved and non-interleaved DCPMM



		Interleaved	Non-Interleaved	Ratio
Latency	Read-only	374.1 ns	394.5 ns	105.5%
	Write-back	391.2 ns	458.4 ns	117.2%
Bandwidth	Read-only	37.6 GB/s	6.4 GB/s	17.0%
	Write-back	2.9 GB/s	0.46 GB/s	15.9%

Problem

- Many prior studies strive to build hybrid memory systems to retain both the advantages of DRAM and PMEM. However, they are either application agnostic or simply take DRAM as a cache, which are both not efficient for in-memory KV stores.

Motivation

➢ General Hybrid Memory Systems

- Compare Memory Mode with DRAM-only, PMEM-only

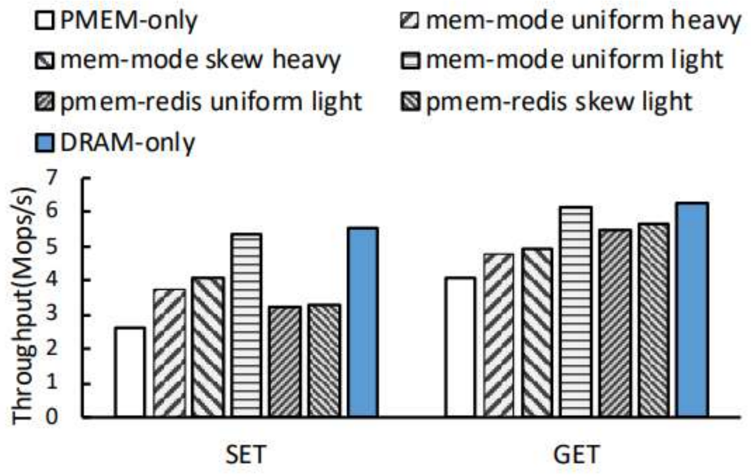


Fig. 2: Throughput of Redis in different situations. (The size of key is 16B and the size of value is 256B. *PMEM-only*: hosted entirely in PMEM; *mem-mode*: hosted in `Memory Mode`. The ratio of DRAM to PMEM is 1: 8; *skew*: the key distribution conforms to the zipfian distribution. The skewness is 0.71, in this case, 30% of data receive 70% of accesses; *uniform*: the key distribution is random; *heavy*: the dataset size is 2x larger than the DRAM cache; *light*: the dataset size is smaller than the DRAM cache)

1. Memory Mode suffers significant performance degradation compared to DRAM-only for the SET operations.
2. The read performance of Memory Mode is much better than PMEM-only.

- This observation implies that keeping hot data in DRAM is very important for improving system responsiveness and throughput.

Motivation

- **Hybrid Memory Oriented KV Systems**
  - Compare pmem-redis with DRAM-only

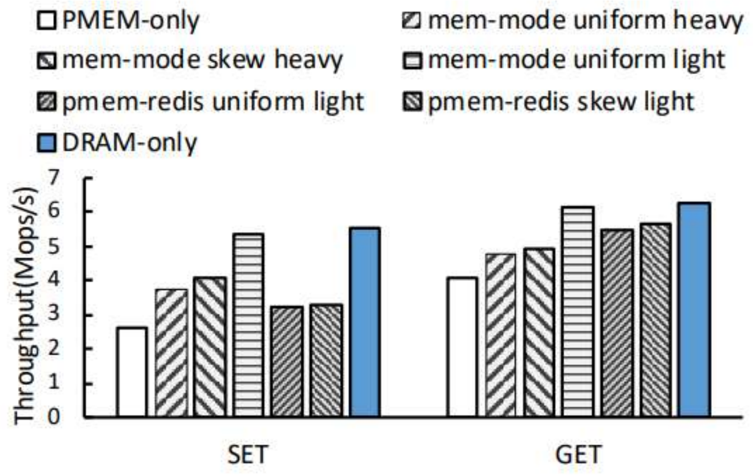


Fig. 2: Throughput of Redis in different situations. (The size of key is 16B and the size of value is 256B. *PMEM-only*: hosted entirely in PMEM; *mem-mode*: hosted in `Memory Mode`. The ratio of DRAM to PMEM is 1: 8; *skew*: the key distribution conforms to the zipfian distribution. The skewness is 0.71, in this case, 30% of data receive 70% of accesses; *uniform*: the key distribution is random; *heavy*: the dataset size is 2x larger than the DRAM cache; *light*: the dataset size is smaller than the DRAM cache)

- pmem-redis stores the small but frequently accessed data (e.g. index structures and key) in DRAM and puts large size data (e.g. values) in PMEM.
  1. 41% (SET) and 13% (GET) gap compared with DRAM-only
  2. similar performance in uniform and Zipfian distribution workloads
- This observation implies that in addition to the keys, hot values alsa need to be maintained in DRAM.

Solution

- This paper presents SwapKV, which strives to retain both the advantages of DRAM and PMEM, aiming to achieve both high performance and large capacity simultaneously.

Challenge Issues & Techniques

- **How to fully utilize the bandwidth of PMEM?**
- **Hybrid Memory Manager**
  - Page Structure (the unit for migration and hotness recording)
    - ✓ A page is composed of 4KB by default, which is larger than 256B, the access granularity of PMEM.
    - ✓ A page is divided into multiple **chunks** with equal size. We categorize pages into multiple **classes** according to their chunk size. We set the minimum chunk size to 16B by default and the growth exponential is 1.25^n. KV items that fit the chunk size are inserted to the page.
    - ✓ The maximum chunk size is limited by the page size, while also limits the maximum KV item size. To address this problem, SwapKV considers page size and minimum chunk size as tunable parameters. For the KV items whose size is larger than page size, SwapKV allocs space for them from memory pool directly. In this case, the *page_addr* points to the KV items, and SwapKV treats these pages as a special class.

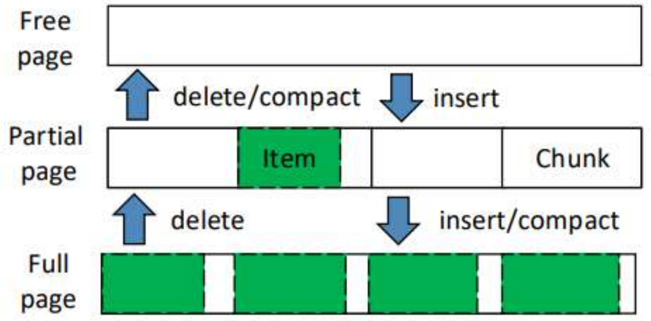
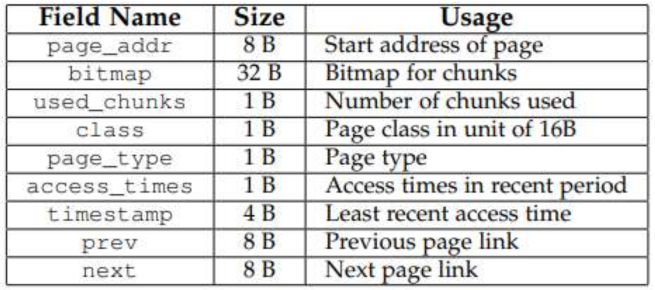


Fig. 3: Structure of three types of pages (*free page*, *partial page*, and *full page*)

Challenge Issues & Techniques

- **How to fully utilize the bandwidth of PMEM?**
- **Hybrid Memory Manager**
  - Page Structure (the unit for migration and hotness recording)
    - ✓ The overall size of all metadata fields is equal to 64B (cacheline size), which is CPU cache friendly.

TABLE 2: Metadata layout of page.



Field Name	Size	Usage
page_addr	8 B	Start address of page
bitmap	32 B	Bitmap for chunks
used_chunks	1 B	Number of chunks used
class	1 B	Page class in unit of 16B
page_type	1 B	Page type
access_times	1 B	Access times in recent period
timestamp	4 B	Least recent access time
prev	8 B	Previous page link
next	8 B	Next page link

    - ✓ The ***bitmap*** records space usage in the page, so that we can quickly find available chunks.
    - ✓ The ***used_chunks*** represents the number of chunks used.
    - ✓ We can calculate the chunk size by ***class*** and distinguish memory type a page belongs to by ***page_type***.
    - ✓ There are some fields for efficient management and hotness recording. We organize pages into multiple types of doubly-linked lists so every page has ***prev*** and ***next*** links. The ***page_type*** also records the list type that page belongs to. The ***timestamp*** and ***access_times*** are used to indicate activity level.

Challenge Issues & Techniques

- **How to fully utilize the bandwidth of PMEM?**
- **Hybrid Memory Manager**
  - Memory Allocation and Recycle
    - ✓ When processing **insert** operations, SwapKV selects a *partial page* fit for the inserted KV item in DRAM, and write the item to page and update the bitmap. If we can’t find a suitable *partial page*, a *free page* will be allocated and transformed to a *partial page*.
    - ✓ As for **update** operations, SwapKV checks the location of the old item. If the old item is stored in DRAM, and the new item can also be placed in the chunk where the old item is located, SwapKV overwrites the old item directly. Otherwise, SwapKV sets a tombstone flag in the bitmap of the old item and executes processes the operation as an insert. When processing data migration, SwapKV allocates a page-sized space in the target memory and frees up the origin space of the migrated page.

Challenge Issues & Techniques

- **How to fully utilize the bandwidth of PMEM?**
- **Hybrid Memory Manager**
  - Memory Allocation and Recycle
    - ✓ The DELETE and UPDATE operations will delete old data, increasing the number of partial pages. The partial pages in DRAM can be reused when handling SET operations.
    - ✓ However, we avoid writing items to partial pages in PMEM to maintain the performance of SET, leading to wasted space of partial pages in PMEM. To recycle the space in PMEM, we compact underutilized pages periodically.
    - ✓ SwapKV starts compacting PMEM pages when the capacity of PMEM drops below a threshold value. The worker thread checks the utilization of a partial page by scanning the bitmap. If the utilization of a page is below 50%, SwapKV will write all live items to other underutilized pages with the same chunk size. Meanwhile, the hash index is modified to point to the new location of the items. After that, SwapKV can recycle a free page and reuse it to store new data. We implement synchronous page compaction in this work.

Challenge Issues & Techniques

- **How to keep hot data in DRAM?**
- **Hotness Awareness Mechanism**
  - Page Activity Level
    - ✓ Each page has two fields, **timestamp** and **access_times**, in its metadata to indicate activity level.
    - ✓ The **timestamp** records the least recent access time for this page. The GET and INSERT operations will update this field, while the DELETE and UPDATE operations keep it unchanged. We set the initial value of access_times to 0.
    - ✓ When we GET and INSERT items in this page, the interval between timestamp and current time will be calculated. If the time interval is shorter than the time threshold, the **access_times** is increased by one until the value becomes out of range.
    - ✓ The timestamp is cleared if the page is not accessed for a long time. A page is regarded as active when the value of access_times is larger than the threshold. With the access_times, we can estimate the activity level of pages.

Challenge Issues & Techniques

- **How to keep hot data in DRAM?**
- **Hotness Awareness Mechanism**
  - Page Activity Level
    - ✓ Using a one-size-fits-all threshold for access_times is unreasonable, because the pages that contain more chunks may be accessed more and turns to be active faster. For fairness reason, the reduced cost of migrating pages to DRAM should be at least $latency_{page}$ (the latency of reading a complete in-PMEM page) regardless of the chunk size.
    - ✓ We define the cost of a in-PMEM page to be $access\ times * latency_{chunk}$, where the $latency_{chunk}$ is the PMEM latency of reading a chunk. The latency of PMEM increases linearly with access size if the size is larger than 256B.
    - ✓ Therefore, for the pages that contain chunk larger than 256B, we let the threshold of access_times be equal to the number of chunks in one page for different classes. The PMEM latency when access size is less than 256B is roughly equal to the latency at 256B, we set the threshold of pages that contain chunk smaller than 256B to be the same as pages that contain 256B chunk. Therefore, the active pages have the same reduced cost.

Challenge Issues & Techniques

- **How to keep hot data in DRAM?**
- **Hotness Awareness Mechanism**
  - Page Filtering Mechanism (i.e. filtering cold pages in DRAM and hot pages in PMEM)
    - ✓ We apply **LRU** policy directly to filter the cold pages in DRAM, and design a **multi-level hotness mechanism** to pick out hot pages in PMEM. Each page **class** implements the filtering mechanisms independently, and the number of hotness levels for filtering hot pages are different in different classes.
    - ✓ In DRAM, the partial pages are responsible for receiving newly written KV items. If a page is full, SwapKV inserts it into an LRU list. In the LRU list, once a page is accessed, it will be moved to the head of the list. If the DRAM capacity is insufficient, the tail page in the LRU list will be migrated to PMEM.

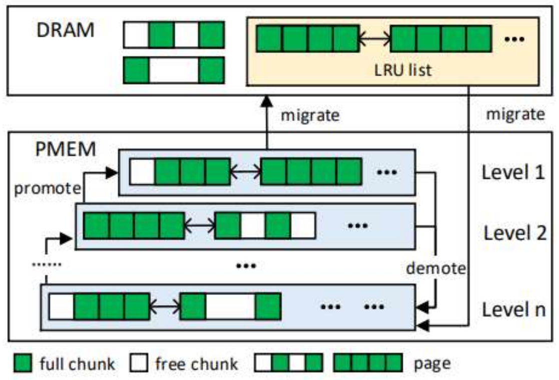


Fig. 4: Procedure of pages filtering. (The LRU list will filter cold pages in DRAM. The hotness level will filter hot pages in PMEM)

Challenge Issues & Techniques

- **How to keep hot data in DRAM?**
- **Hotness Awareness Mechanism**
  - Page Filtering Mechanism (i.e. filtering cold pages in DRAM and hot pages in PMEM)
    - ✓ In PMEM, in order to select frequently accessed pages, we organize the pages in the hierarchy. The pages on the same level are linked. The top level stores the hottest pages and the bottom level stores the coldest pages. A page swapped from DRAM is inserted into the bottom level first. If the page is found to be active when it is accessed, we move the page to the upper level. Otherwise, we move the inactive page back to the bottom level.
    - ✓ A page in the top level becomes a candidate hot page. Once a candidate hot page is accessed, the page will be migrated to DRAM. Currently, the number of hotness levels is a parameter, which could be adjusted according to the workloads. The default number of hotness levels is equal to the number of chunks for one page in the corresponding page class.

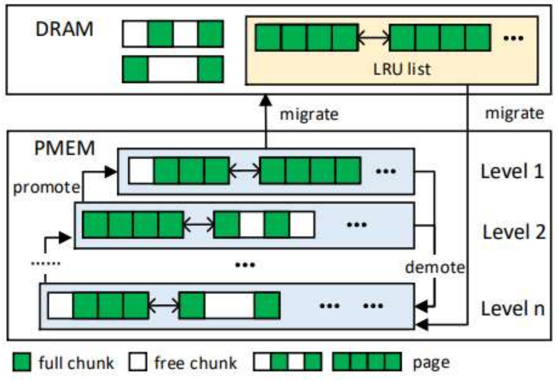


Fig. 4: Procedure of pages filtering. (The LRU list will filter cold pages in DRAM. The hotness level will filter hot pages in PMEM)

Challenge Issues & Techniques

- **How to reduce the overhead of data migration?**
- **Asynchronous Page Migration**
  - To reduce the performance loss caused by thread synchronization, a dedicated **background thread** is only responsible for copying pages between DRAM and PMEM. The **worker thread** is used to select candidate pages and update metadata information. To eliminate the synchronization overhead among thread, We apply the lock-free swap queues and do not lock the page during migration.
  - Step1: The worker thread selects a candidate page and moves it into the **swap list**. Meanwhile, the worker thread pushes a tuple into the **incomplete swap queue**. The tuple contains the address of the candidate page metadata, the destination memory address, and the opcode (i.e. different copy instructions like memcpy and non-temporal store---adopted).

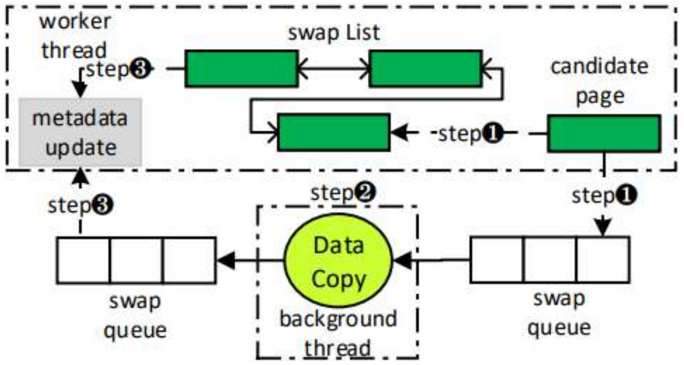


Fig. 5: Procedure of pages migration

Challenge Issues & Techniques

- **How to reduce the overhead of data migration?**
- **Asynchronous Page Migration**
  - Step2: The background thread picks requests from the **incomplete swap queue** and copy the data of the candidate page to the destination address. After that, the tuple is popped from the incomplete swap queue and is pushed to the **complete swap queue**.
  - Step3: The worker thread updates the metadata of the complete pages, including the *page_addr* and the *page_type* fields, and recycle the memory space addressed by the old *page_addr*. The metadata of the page will be moved from the swap list to the corresponding list based on its type.

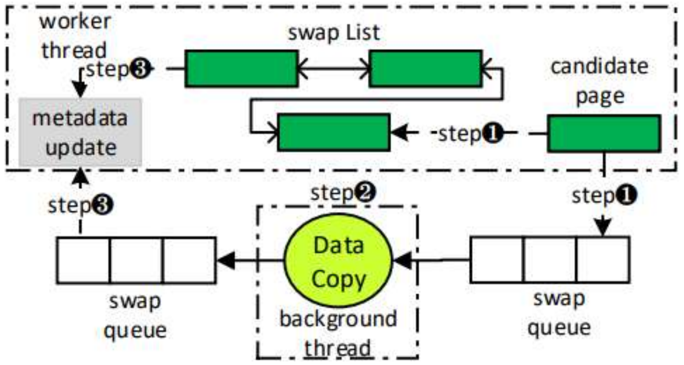


Fig. 5: Procedure of pages migration

Challenge Issues & Techniques

- **How to reduce the overhead of data migration?**
- **Asynchronous Page Migration**
  - Data consistency

SwapKV can process page migration without blocking requests from clients. Instead of locking the page during migration, we mark it with an *occupancy* flag by reusing *page_type* field. However, if we only use one flag to distinguish pages being migrated, the worker thread cannot look for candidate pages efficiently since the worker thread needs to look up the LRU list from the tail to find a cold candidate page not marked occupancy. SwapKV addresses this issue by removing the occupied pages from the original lists and link it to the swap list. For the operations not modifying the data, such as GET and DELETE, SwapKV can handle them on occupied pages directly. For SET operation, SwapKV selects another page to store the new KV item.

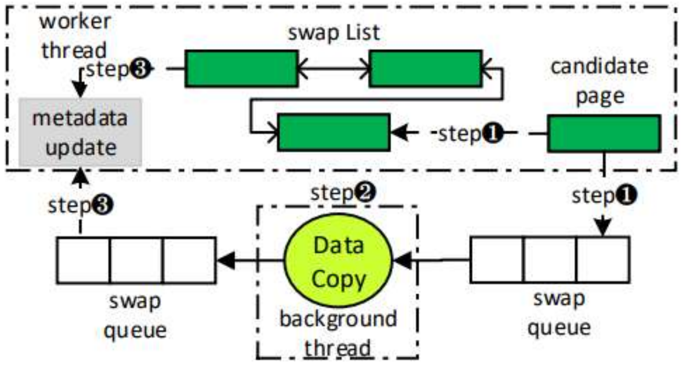


Fig. 5: Procedure of pages migration

Challenge Issues & Techniques

- **How to reduce the overhead of data migration?**
- **Hash Index Design**
  - SwapKV relies on a hash index to look up KV items. We still use the chain hash table in Redis. To combine the page structure and hash index better, we modify the entry structure as shown in Figure 6. Each bucket may have multiple entries. An entry contains an 8B pointer of the page metadata, the 1B chunk id of KV item, and a 4B signature of the key. We can calculate the start address of items according to the page address, chunk size, and chunk id. In the item, 1B key len and 3B value len are reserved to support variable key and value.
  - Using a page metadata pointer instead of an item pointer can avoid the hash index update overhead during page migration. For each page migration, SwapKV only needs to modify the page_addr field of the metadata, not all the entries of the KV items in the page.

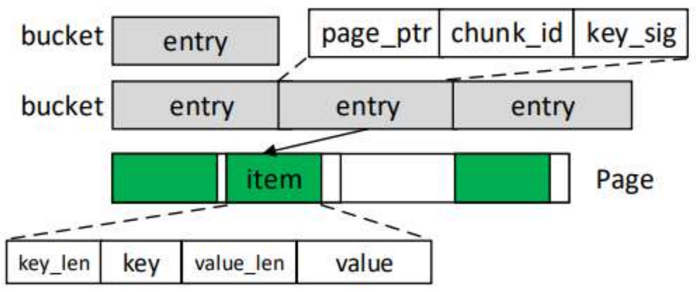


Fig. 6: Hash index structure

Challenge Issues & Techniques

➢ **Implementation**

- We implement SwapKV in C on top of **Redis** v4.0. We use *pthreads* to implement the background thread. We utilize PMDK, a mature PMEM development library, to manage PMEM. When the system starts, SwapKV uses *pmem_map_file* function to create a PMEM file, then map the file to the virtual memory region. The DRAM-to-PMEM migration uses *pmem_memcpy* which supports non-temporal store instructions to bypass the CPU cache. We use *jemalloc* library to allocate memory for the hash index and other metadata. For both DRAM and PMEM, the maximum capacity that can be used by the hybrid memory system is controlled by the hybrid memory manager.

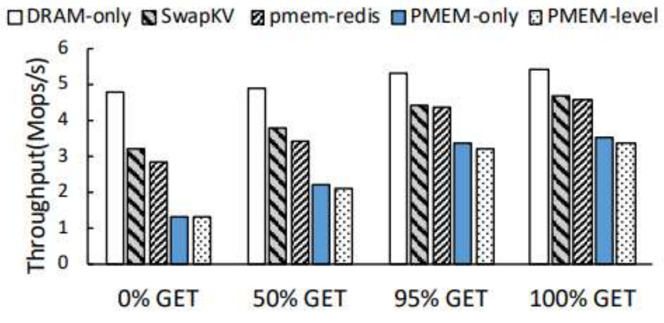
Evaluation

➢ Experimental Setup

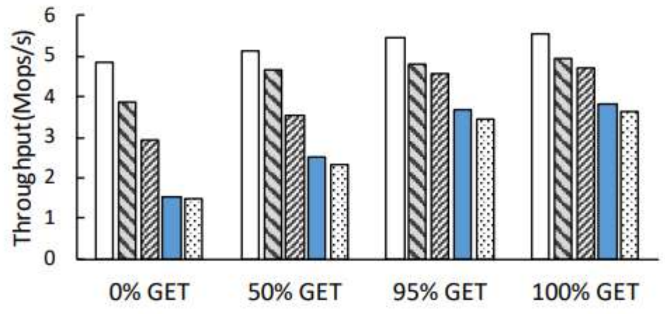
- We evaluate SwapKV on a dual-socket x86 server, which is equipped with two Intel Xeon Gold 5220 processors. Each processor runs at 2.2GHz frequency and has 18 cores with 32KB L1i cache, 32KB L1d cache and 1MB L2 cache in each core. The shared last level cache of each processor is 25 MB. The total DRAM size is 128GB, with 64GB in each socket. The server has **4 128GB Intel Optane DC persistent memory** installed in socket 0, the total size of PMEM is 512GB. The ext4-DAX file system is mounted on the PMEM devices.
- **System configuration.** We set the maximum DRAM capacity and PMEM capacity in the hybrid memory system to 512MB and 10GB per instance. Coupled with other parts (e.g. hash index), the total DRAM consumption of 36 instances is about 40GB. To avoid the performance loss caused by accessing PMEM across sockets, all experiments are conducted on socket 0.
- **Compared Systems**
  - ✓ DRAM-only Redis v4.0
  - ✓ PMEM-only Redis v4.0
  - ✓ PMEM-level, which replaces hash table of Redis with level hashing (a persistent hashing based on PMEM). Every hash bucket has 2 entries in it.
  - ✓ pmem-redis, an improved version for PMEM based on Redis v4.0. We set the maximum PMEM capacity that can be used in pmem-redis to 10GB, as that in SwapKV.
  - ✓ Memory Mode Redis v4.0

Evaluation

- **System Throughput and Latency**
  - Throughput

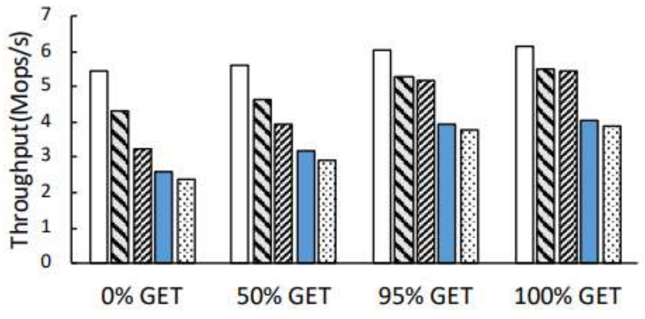


(a) Uniform distribution.

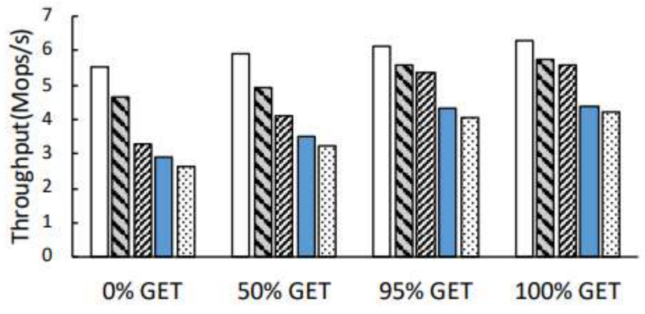


(b) Zipfian distribution.

Fig. 7: Throughput of 1000B value



(a) Uniform distribution.



(b) Zipfian distribution.

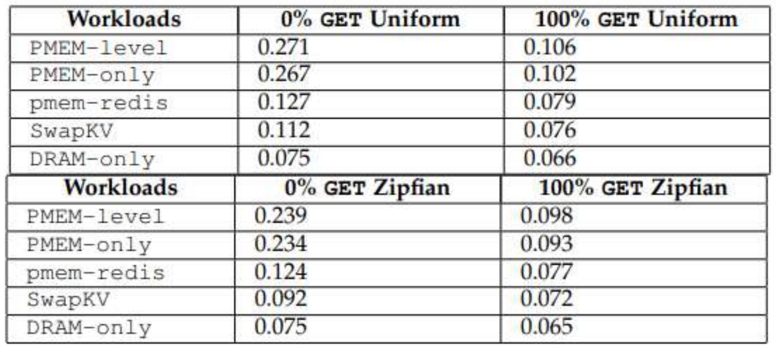
Fig. 8: Throughput of 256B value

Evaluation

➢ System Throughput and Latency

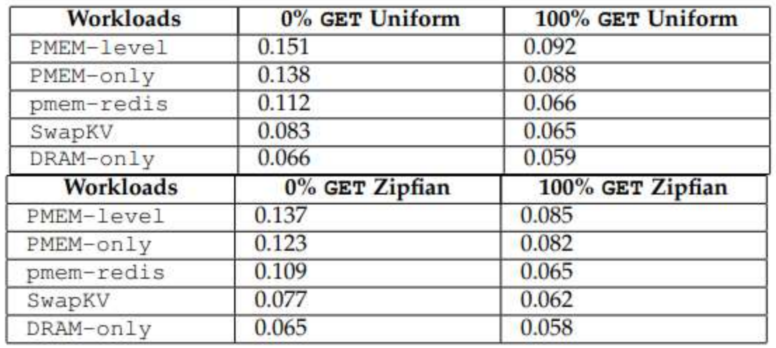
- Latency

TABLE 3: Latency of 1000B (ms)



Workloads	0% GET Uniform	100% GET Uniform
PMEM-level	0.271	0.106
PMEM-only	0.267	0.102
pmem-redis	0.127	0.079
SwapKV	0.112	0.076
DRAM-only	0.075	0.066
Workloads	**0% GET Zipfian**	**100% GET Zipfian**
PMEM-level	0.239	0.098
PMEM-only	0.234	0.093
pmem-redis	0.124	0.077
SwapKV	0.092	0.072
DRAM-only	0.075	0.065

TABLE 4: Latency of 256B (ms)



Workloads	0% GET Uniform	100% GET Uniform
PMEM-level	0.151	0.092
PMEM-only	0.138	0.088
pmem-redis	0.112	0.066
SwapKV	0.083	0.065
DRAM-only	0.066	0.059
Workloads	**0% GET Zipfian**	**100% GET Zipfian**
PMEM-level	0.137	0.085
PMEM-only	0.123	0.082
pmem-redis	0.109	0.065
SwapKV	0.077	0.062
DRAM-only	0.065	0.058

Evaluation

➢ Scalability

- For 0% GET, the write-only workload, the background thread is busy for data migration. In this situation, the increased factor of SwapKV is still higher than pmem-redis but it is slightly lower than DRAM-only. The CPU utilization of one instance are 55%, 69%, 66% and 72% for DRAM-only, pmem-redis, SwapKV and PMEM-only respectively. These results confirm that SwapKV does not use more CPU resources than others.
- The CPU utilization of background thread is only 8% in 0% GET workload.

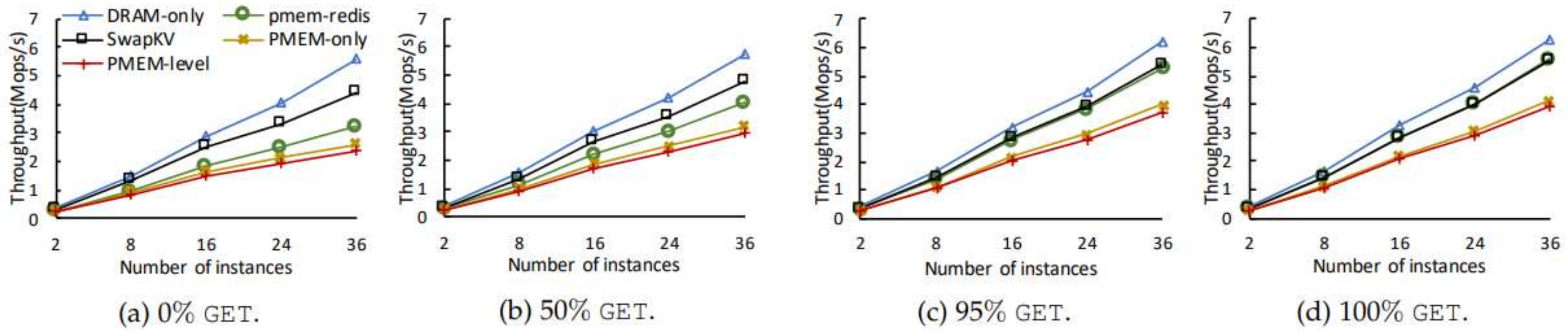


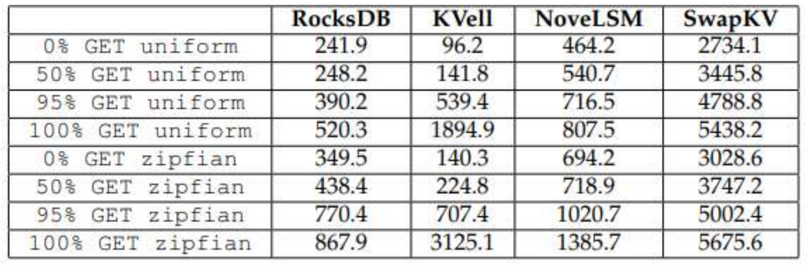
Fig. 9: Scalability on uniform distribution workloads

Evaluation

➢ Comparison to RocksDB, NoveLSM and KVell

- The compared systems could ensure data durability. For a fair comparison, we evaluate SwapKV with opening **AOF** (logging mechanism). The AOF files are in PMEM. The sstable of RocksDB and NoveLSM, and key-values of KVell are placed into PMEM.

TABLE 5: Throughput of RocksDB, KVell and NoveLSM (Kops/s)



	RocksDB	KVell	NoveLSM	SwapKV
0% GET uniform	241.9	96.2	464.2	2734.1
50% GET uniform	248.2	141.8	540.7	3445.8
95% GET uniform	390.2	539.4	716.5	4788.8
100% GET uniform	520.3	1894.9	807.5	5438.2
0% GET zipfian	349.5	140.3	694.2	3028.6
50% GET zipfian	438.4	224.8	718.9	3747.2
95% GET zipfian	770.4	707.4	1020.7	5002.4
100% GET zipfian	867.9	3125.1	1385.7	5675.6

- It can be seen that in the zipfian distributed workloads, SwapKV outperforms RocksDB, NoveLSM and KVell significantly. The advantage is 1.8x ~ 28x over KVell, 6.5x ~ 11x over RocksDB and 4x ~ 6.3x over NoveLSM respectively.

Evaluation

The Benefit of Each Component in SwapKV

- There are three main components in SwapKV, hybrid memory pool (hm), hotness aware mechanism (hotness) and asynchronous migration (async).
- Based on PMEM-only, **+hm version** adds 512MB DRAM every instance and places index in DRAM. It adopts FIFO queue in DRAM to organize pages and migrates page synchronously.
- Based on +hm version, **+hotness** adds the hotness aware mechanism.
- The **+async** version further adds asynchronous migration, which is the full SwapKV version.

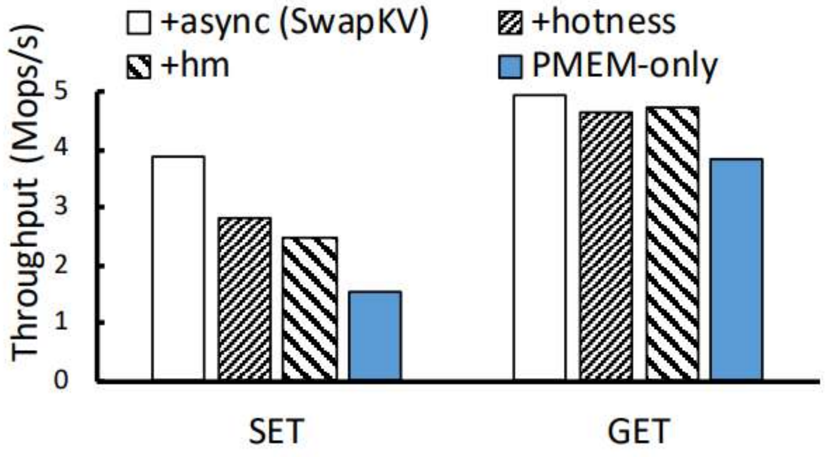


Fig. 10: Benefit of each component

Evaluation

- **Robustness of Hotness Awareness Mechanism**
  - Impact of Skewness (reflecting the efficiency of hotness awareness mechanism)
    - ✓ The locality 60/40 means that 40% of the data receive 60% of the accesses.
    - ✓ We conduct 50% GET workload with value size of 1000B under different skewness.

TABLE 6: Skewness parameter

Skewness	0.44	0.71	0.88	0.99
Locality	60/40	70/30	80/20	default of YCSB

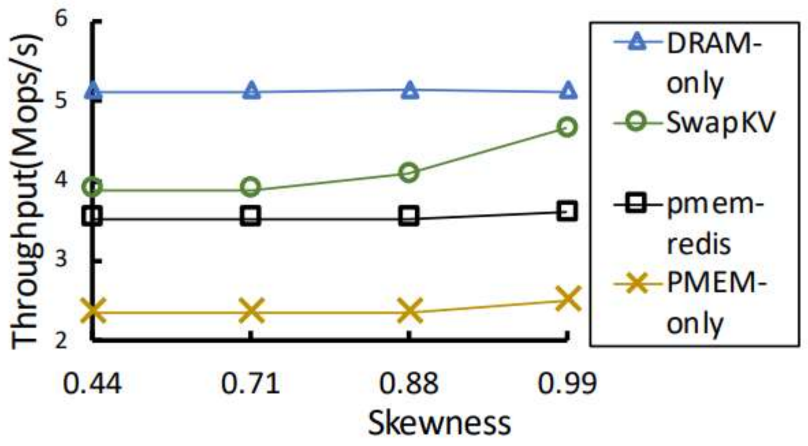


Fig. 11: Impact of Skewness

Evaluation

- **Robustness of Hotness Awareness Mechanism**
  - Impact of DRAM Capacity
    - ✓ Y-axis: The normalized access ratio of DRAM to PMEM
    - ✓ X-axis: The PMEM/DRAM ratio represents the ratio of KV items stored in PMEM and DRAM.

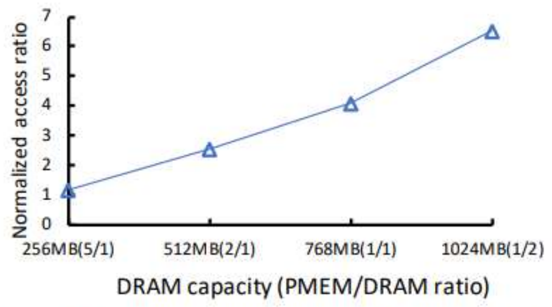
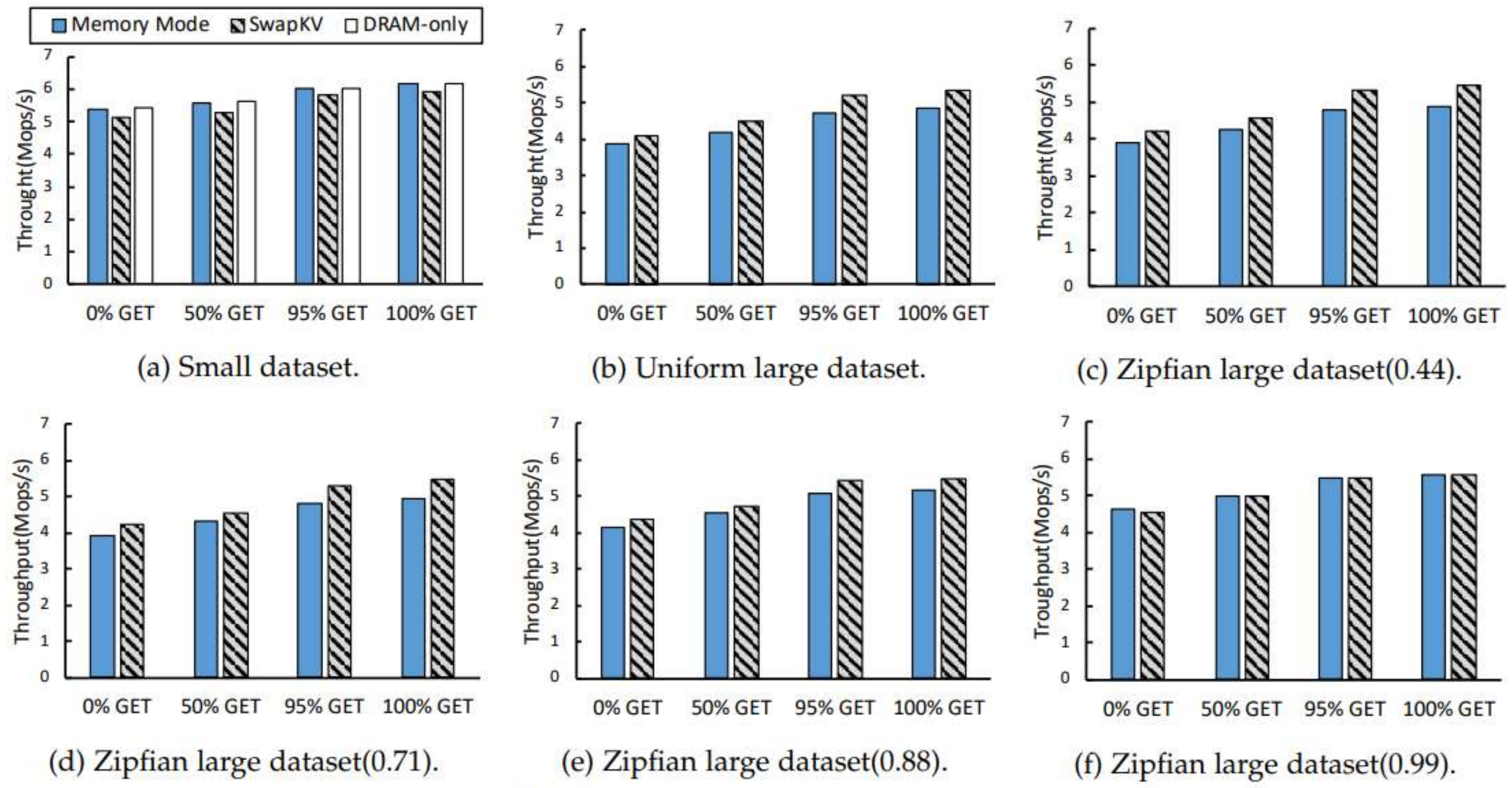


Fig. 12: Access ratio under different DRAM capacity

- ✓ By dividing the access ratio by the capacity ratio, we can calculate the ratio of the average number of access for each item in DRAM and PMEM. For the DRAM capacity of 256MB, the average access ratio of each item in DRAM and PMEM is 5.95:1, which implies that the hottest KV items have been collected into DRAM. The above results demonstrate that the hotness awareness mechanisms applied in SwapKV are efficient for skewed workloads. Even with very limited DRAM capacity, SwapKV can still keep the hottest data in DRAM.

Evaluation

➢ **Impact of Dataset Size**



(a) Small dataset. (b) Uniform large dataset. (c) Zipfian large dataset(0.44).

(d) Zipfian large dataset(0.71). (e) Zipfian large dataset(0.88). (f) Zipfian large dataset(0.99).

Fig. 13: The impact of dataset size

Evaluation

➢ Impact of Value Size

- As the value size decreases, the throughput of DRAM-only increases gradually.
- However, pmem-redis does not achieve the expected performance improvement for smaller value sizes, which has similar throughput for both large and small value sizes. This is mainly due to the negative impact of write amplification for small items.
- In contrast, SwapKV has a similar performance trend with DRAM-only, which confirms that organizing KV items into pages can efficiently alleviate the write amplification issue caused by small items.

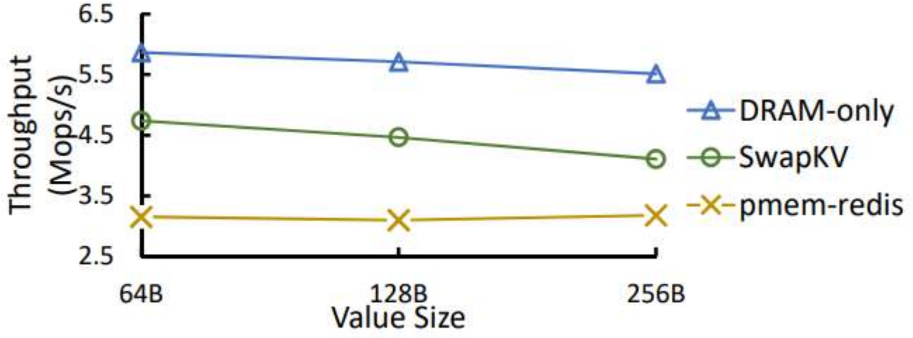


Fig. 14: SET throughput under different value size

Evaluation

➢ **DRAM Consumption**

- Sequentially inserting 15M key-values with 256B value size

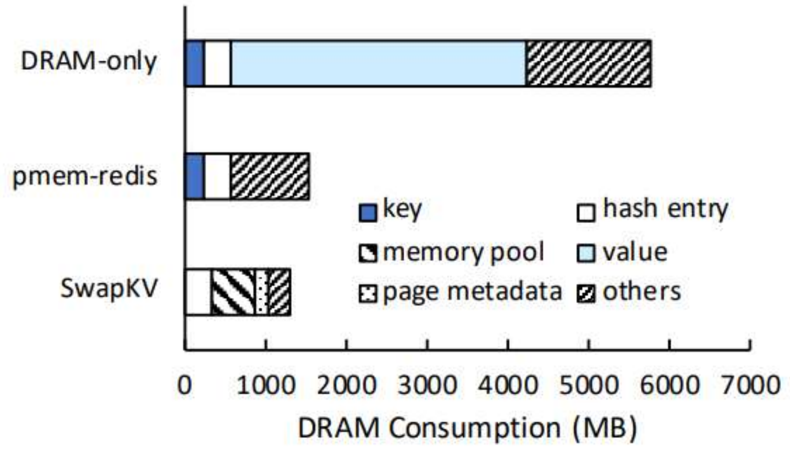


Fig. 15: DRAM consumption

Thank You!