HMEH: write-optimal extendible hashing for hybrid DRAM-NVM memory

Xiaomin Zou [1], Fang Wang[1*], Dan Feng[1], Janxi Chen[1], Chaojie Liu[1], Fan Li[1], Nan Su[2]

Huazhong University of Science and Technology[1], China

Shandong Massive Information Technology Research Institute[2], China

Outline

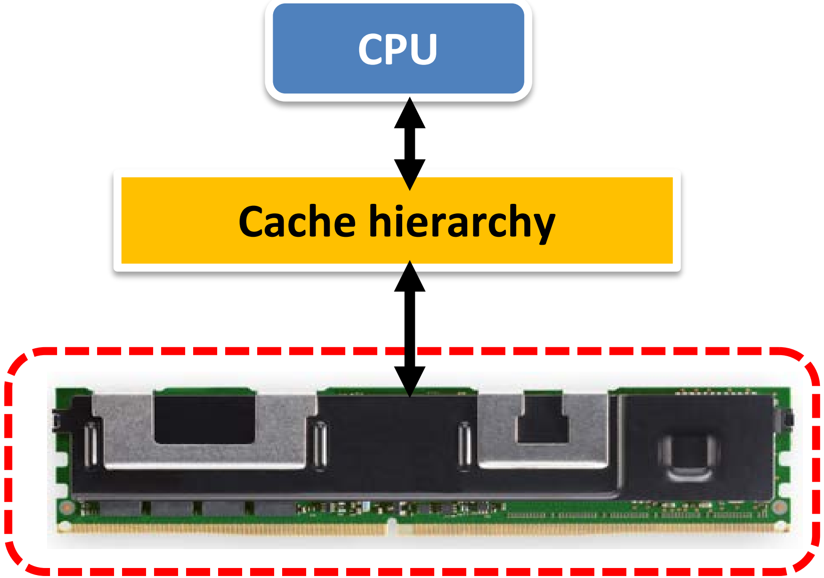
- **Background and motivation**
- **Our Work: HMEH**
- **Performance Evaluation**
- **Conclusion**

Background : Non-Volatile Memory (NVM)

➢ NVM is expected to complement or replace DRAM as main memory

☺
- ✓ non-volatile
- ✓ large capacity
- ✓ high performance
- ✓ low standby power

☹
- limited write endurance
- asymmetric properties



Intel Optane DC Persistent Memory

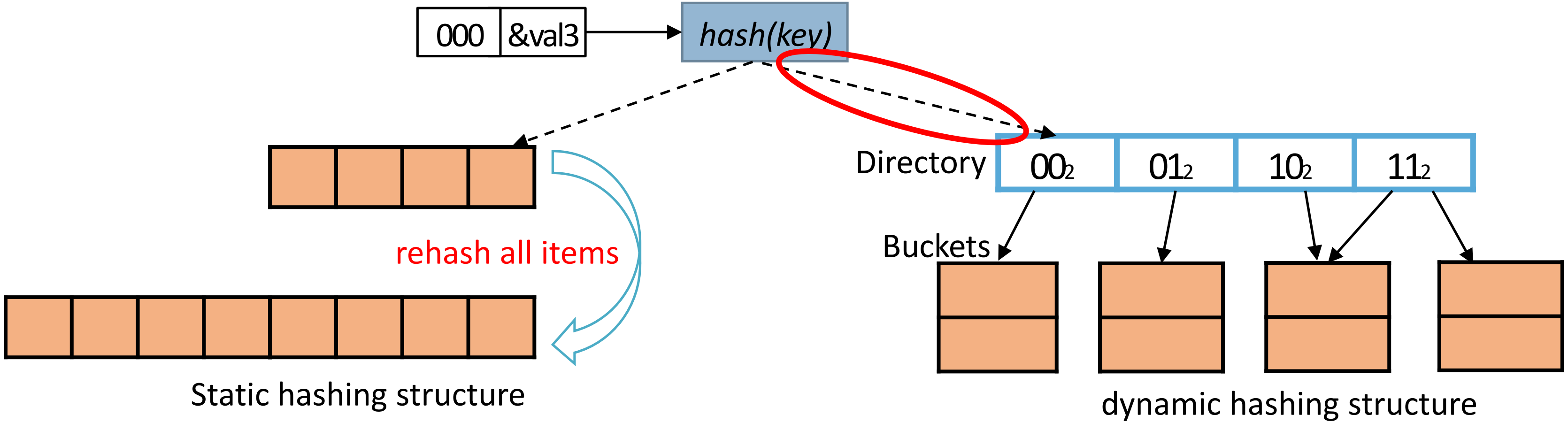
Background : NVM-based hash structures

- **Hashing structures are widely used in storage systems**
  - ✓ main memory database
  - ✓ in-cache index
  - ✓ in-memory key-value store

- **Previous work is insufficient for real NVM device**
  - PFHT [INFLOW 2015]
  - Path hashing [MSST 2017]
  - Level hashing [OSDI 2018]
  - CCEH [FAST 2019]

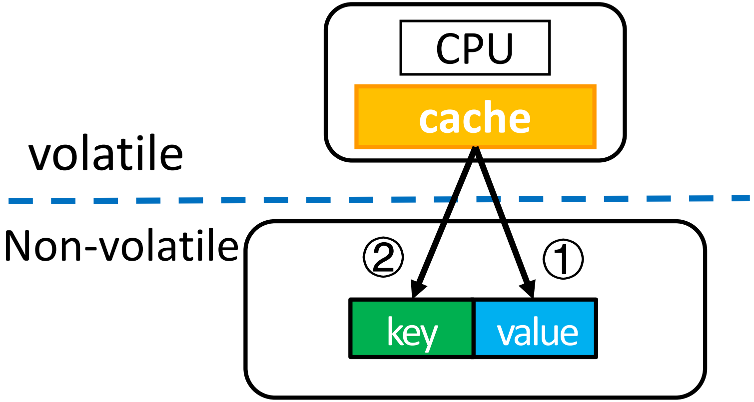
Motivation : The design of hashing structure

- **Static hashing structure vs Dynamic hashing structure**
  - **Static hashing: Cost inefficiency for resizing hash table**
  - **Dynamic hashing: need extra directory access and the read latency of optane DCPMM is higher**



Motivation : High overhead for data consistency

- **Data consistency guarantee**
  - **The volatile/non-volatile boundary is between CPU cache and NVM**
  - **Arbitrarily-evicted cache lines → memory writes reordering**



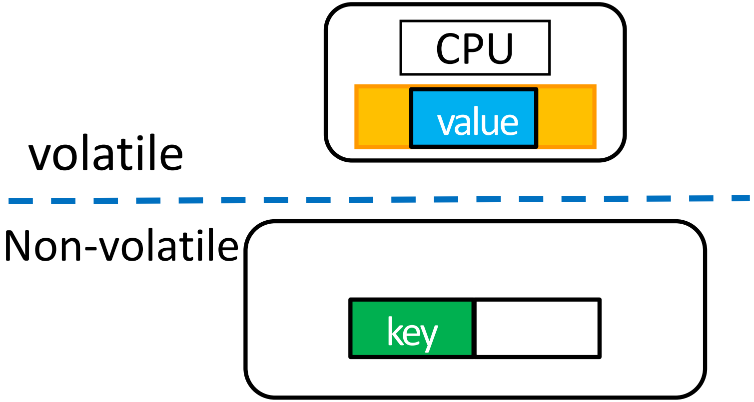
Program reordering

St value;

St key;

Motivation : High overhead for data consistency

- **Data consistency guarantee**
  - **The volatile/non-volatile boundary is between CPU cache and NVM**
  - **Arbitrarily-evicted cache lines → memory writes reordering**



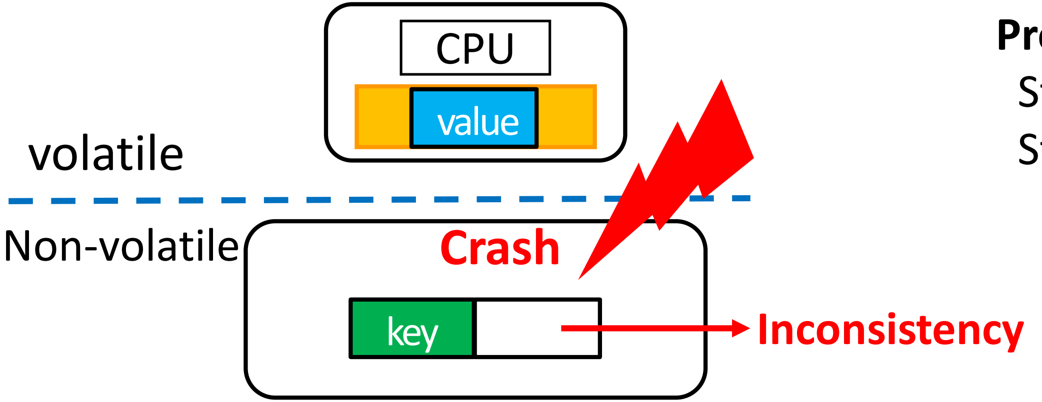
Program reordering

St value;
St key;

reordered

Motivation : High overhead for data consistency

- **Data consistency guarantee**
  - **The volatile/non-volatile boundary is between CPU cache and NVM**
  - **Arbitrarily-evicted cache lines → memory writes reordering**



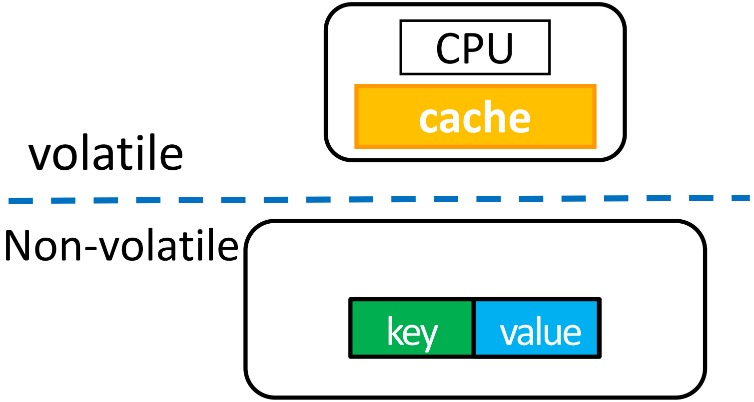
Program reordering

St value;
St key;

Motivation : High overhead for data consistency

➢ Data consistency guarantee

- The volatile/non-volatile boundary is between CPU cache and NVM
- Arbitrarily-evicted cache lines → memory writes reordering
  - ✓ Flush: flush cache lines
  - ✓ Fence: order CPU cache line flush



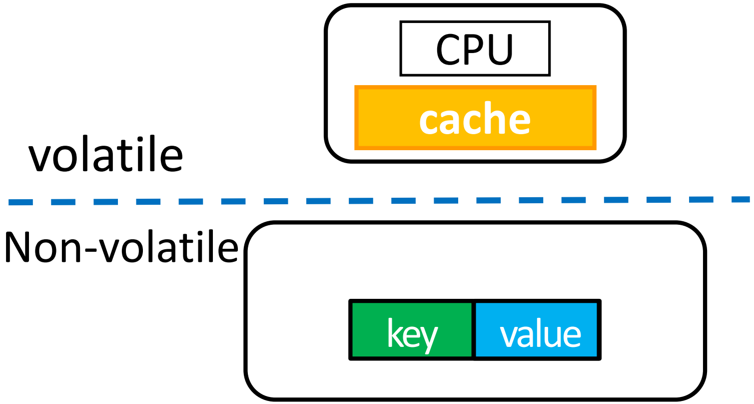
Program reordering

```
St value;
Fence();
St key;
Flush()
```

Motivation : High overhead for data consistency

- **Data consistency guarantee**
  - **The volatile/non-volatile boundary is between CPU cache and NVM**
  - **Arbitrarily-evicted cache lines → memory writes reordering**
    - ✓ **Flush: flush cache lines**
    - ✓ **Fence: order CPU cache line flush**

Expensive !



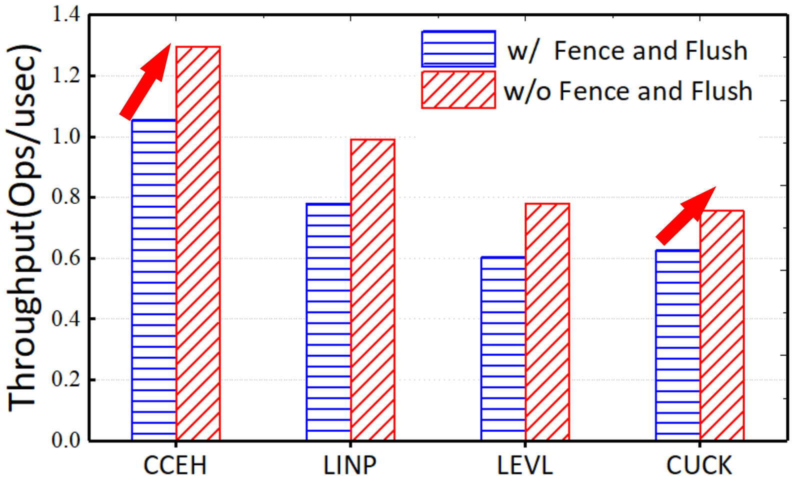
Program reordering

St value;
Fence();
St key;
Flush()

Motivation : High overhead for data consistency

- **Data consistency guarantee**
  - **the evaluation with/without Fence and Flush in optane DCPMM**
  - ✓ **CCEH[FAST 2019], LEVL[OSDI 2018], linear hashing, and cuckoo hashing**

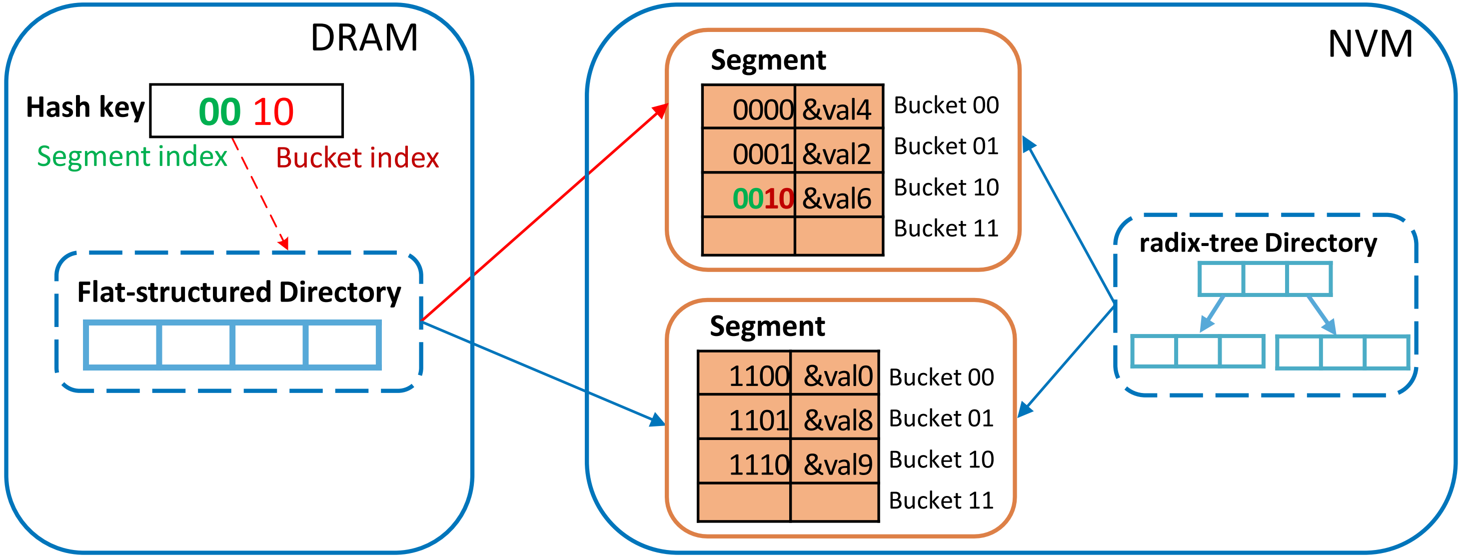
without Fence and Flush instructions, the throughputs of these hashing schemes are improved by **20.3% to 29.1%**



- **Our goals**
  - ✓ **high-performance dynamic hashing with low data consistency overhead and fast recovery**

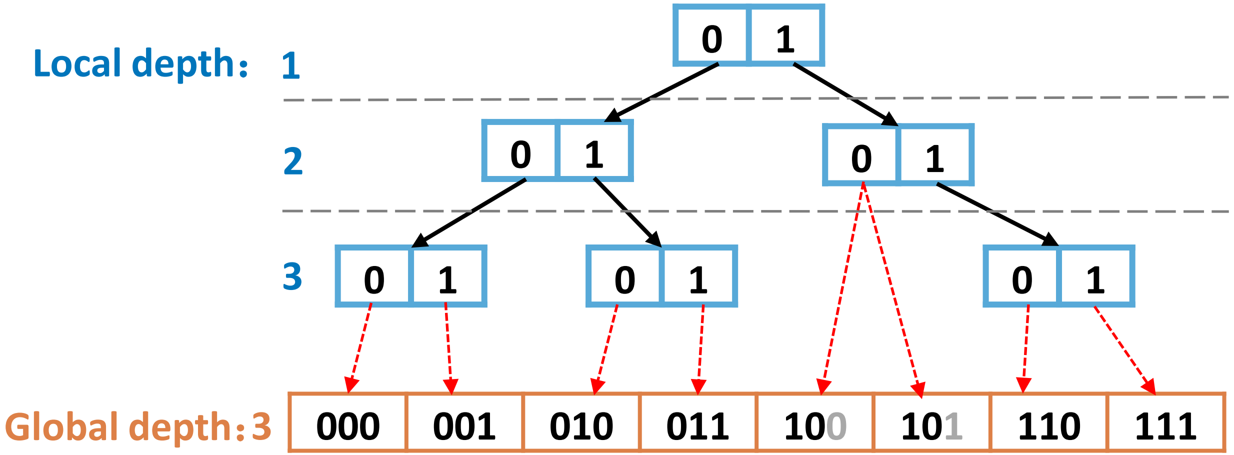
Our Scheme: HMEH

- HMEH: Extendible Hashing for Hybrid DRAM-NVM Memory
  - ✓ Flat-structured Directory for fast access and radix-tree Directory for recovery
  - ✓ Directory → segment → cacheline-sized bucket



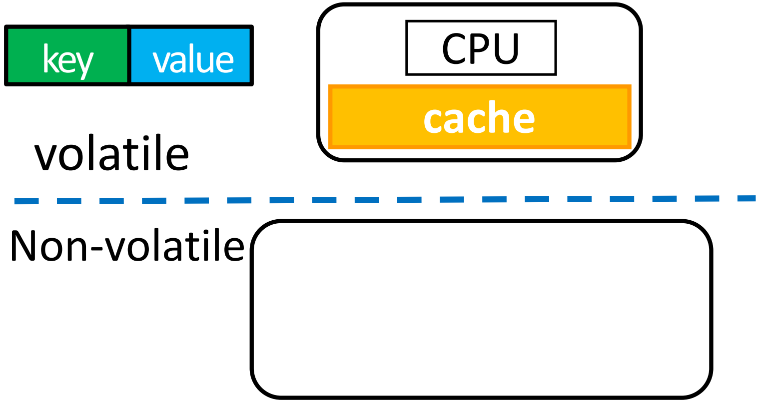
HMEH : Two directories

- **Flat-structured Directory VS Radix-tree Directory**
  - ✓ Radix tree is friendly to NVM
  - ✓ exploit RT-directory to rebuild FS-directory upon recovery
  - ✓ every segment is pointed by 2^{G-L} directory entries



HMEH : Low data consistency overhead

- **Cross-KV mechanism**
  - ✓ **Split kv item into several pieces and alternately store key and value as several 8-byte atomic blocks**
  - ✓ **Avoid lots of Flush and Fence instructions**

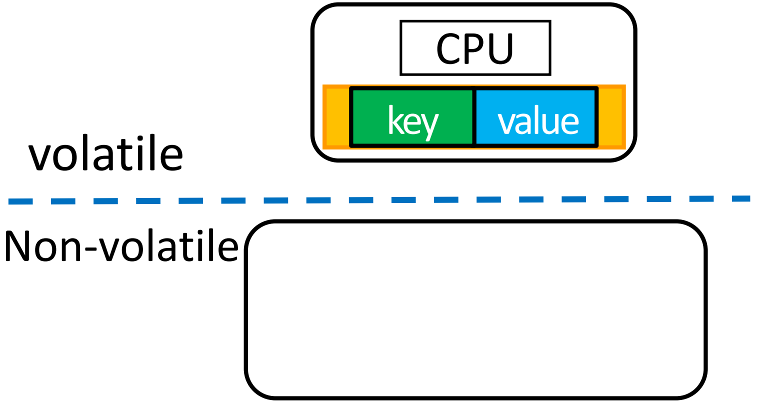


Program reordering

St value;
Fence();
St key;
Flush()

HMEH : Low data consistency overhead

- **Cross-KV mechanism**
  - ✓ **Split kv item into several pieces and alternately store key and value as several 8-byte atomic blocks**
  - ✓ **Avoid lots of Flush and Fence instructions**

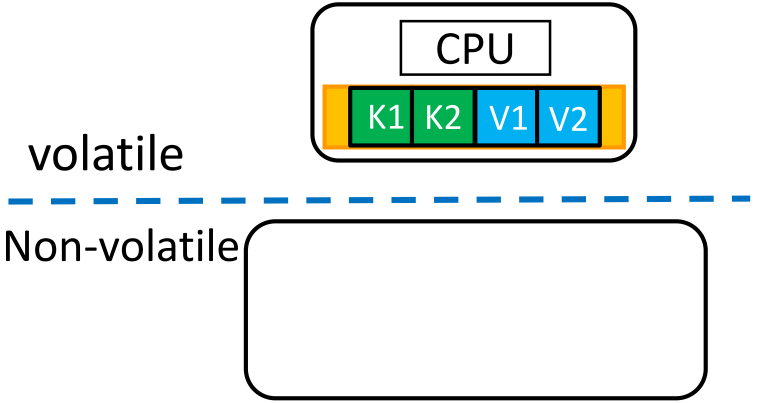


Program reordering

St value;
Fence();
St key;
Flush()

HMEH : Low data consistency overhead

- **Cross-KV mechanism**
  - ✓ **Split kv item into several pieces and alternately store key and value as several 8-byte atomic blocks**
  - ✓ **Avoid lots of Flush and Fence instructions**

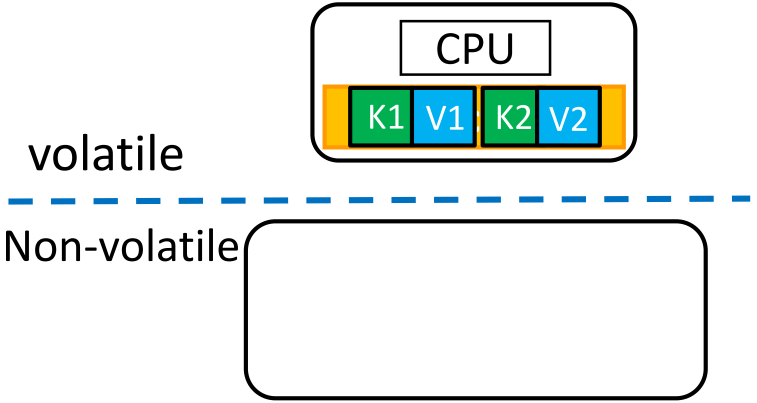


Program reordering

St value;
Fence();
St key;
Flush()

HMEH : Low data consistency overhead

- **Cross-KV mechanism**
  - ✓ **Split kv item into several pieces and alternately store key and value as several 8-byte atomic blocks**
  - ✓ **Avoid lots of Flush and Fence instructions**

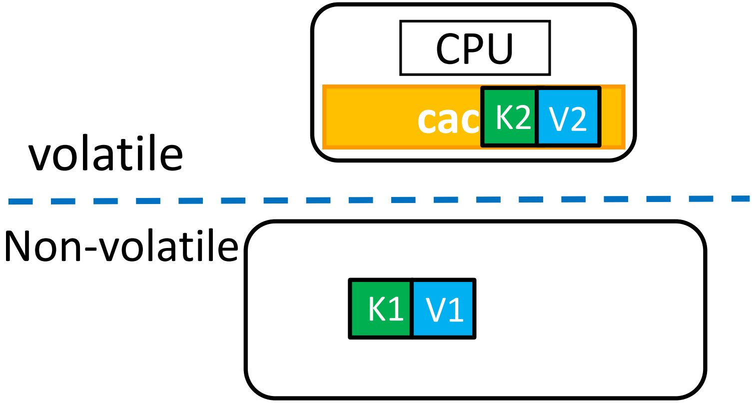


Program reordering

St value;
Fence();
St key;
Flush()

HMEH : Low data consistency overhead

- **Cross-KV mechanism**
  - ✓ **Split kv item into several pieces and alternately store key and value as several 8-byte atomic blocks**
  - ✓ **Avoid lots of Flush and Fence instructions**

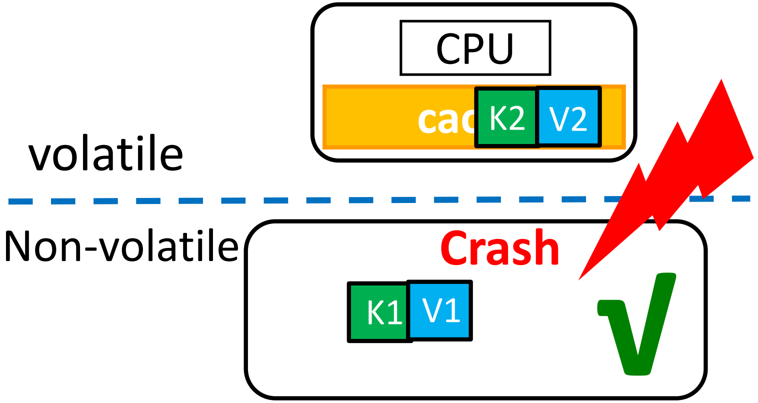


Program reordering

St value;
Fence();
St key;
Flush()

HMEH : Low data consistency overhead

- **Cross-KV mechanism**
  - ✓ **Split kv item into several pieces and alternately store key and value as several 8-byte atomic blocks**
  - ✓ **Avoid lots of Flush and Fence instructions**

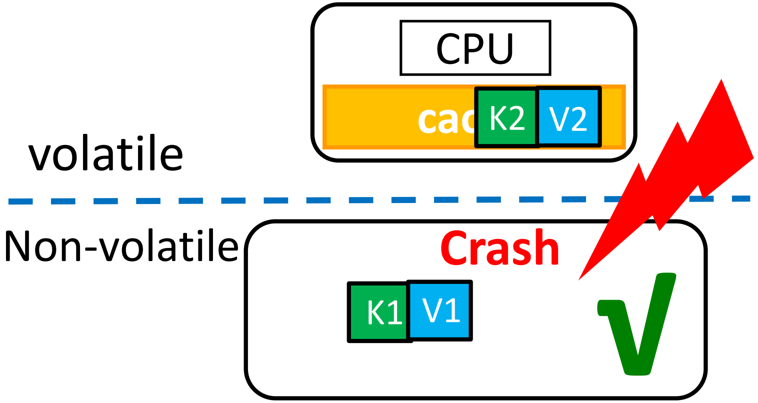


Program reordering

St value;
Fence();
St key;
Flush()

HMEH : Low data consistency overhead

- **Cross-KV mechanism**
  - ✓ **Split kv item into several pieces and alternately store key and value as several 8-byte atomic blocks**
  - ✓ **Avoid lots of Flush and Fence instructions**



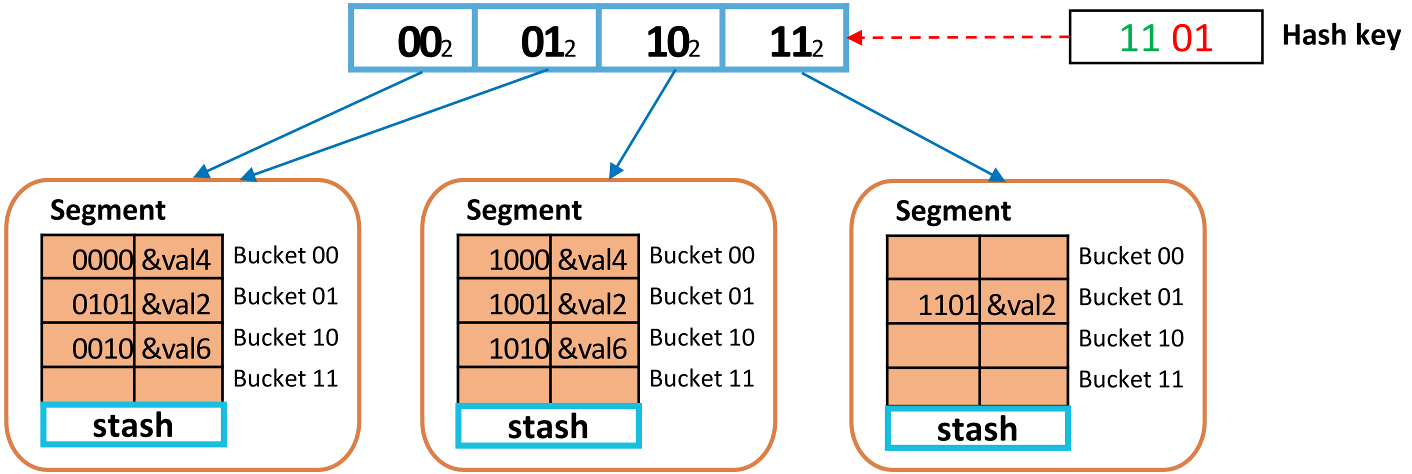
Program reordering

St value;
Fence();
St key;
Flush()

→ **St cross-KVs**

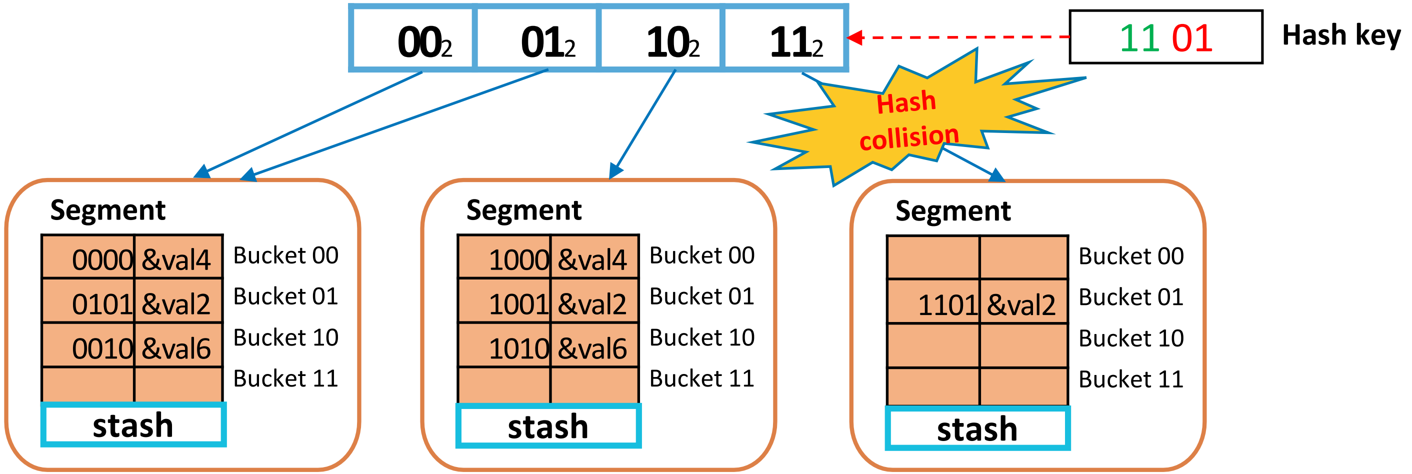
HMEH : Improve load factor

- Resolve hash collisions
  - ✓ linear probing： allow probe 4 buckets (256bytes, the access granularity of intel optane DCPMM)
  - ✓ stash: non-addressable and used to store colliding items



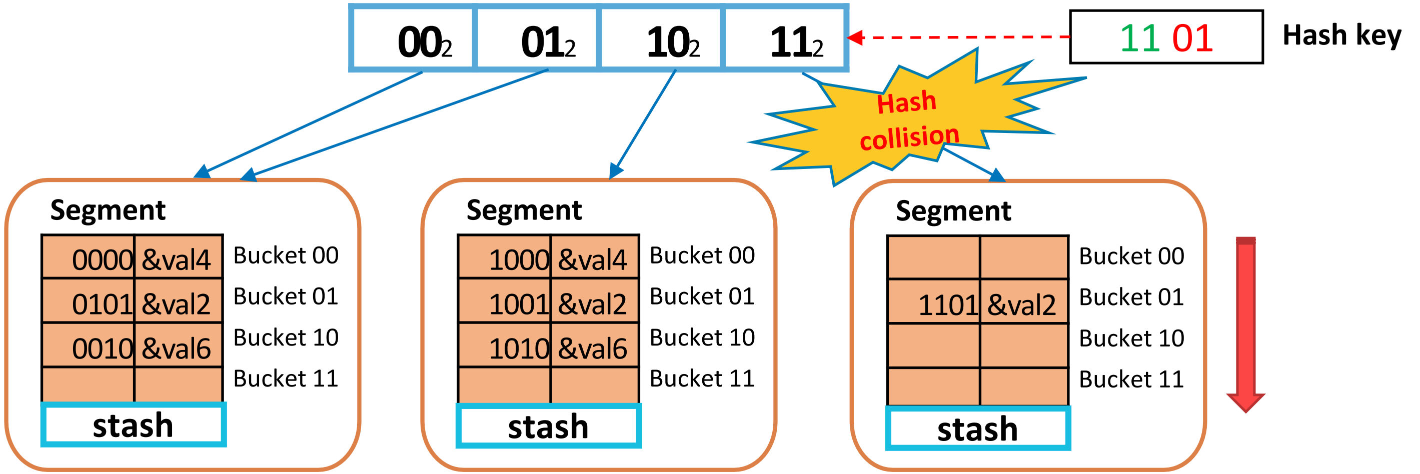
HMEH : Improve load factor

- Resolve hash collisions
  - ✓ linear probing ： allow probe 4 buckets (256bytes, the access granularity of intel optane DCPMM)
  - ✓ stash: non-addressable and used to store colliding items



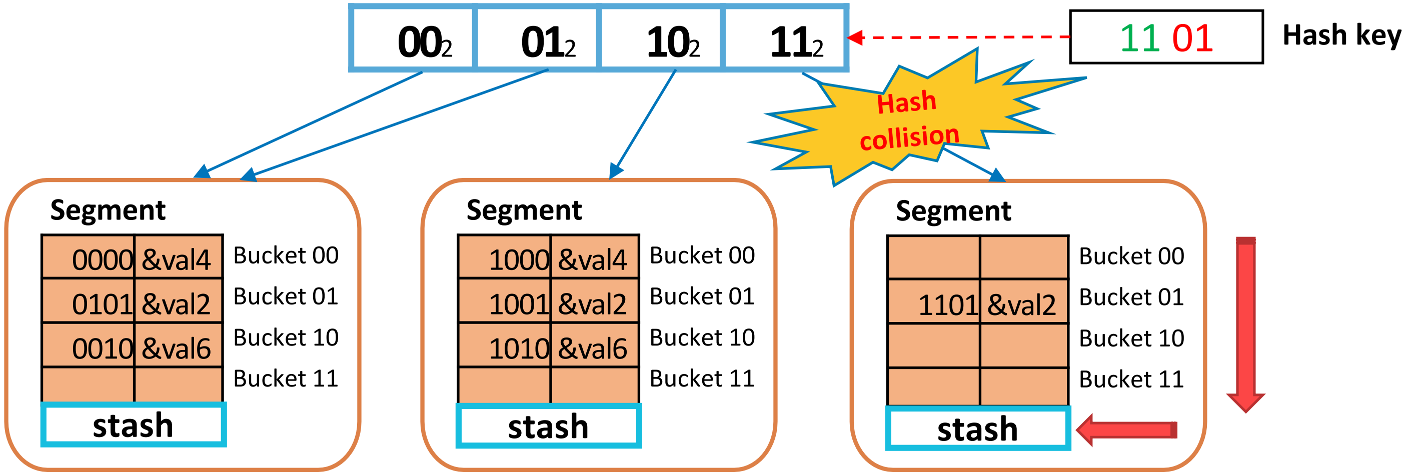
HMEH : Improve load factor

- **Resolve hash collisions**
  - ✓ linear probing ： allow probe 4 buckets (256bytes, the access granularity of intel optane DCPMM)
  - ✓ stash: non-addressable and used to store colliding items



HMEH : Improve load factor

- Resolve hash collisions
  - ✓ linear probing : allow probe 4 buckets (256bytes, the access granularity of intel optane DCPMM)
  - ✓ stash: non-addressable and used to store colliding items

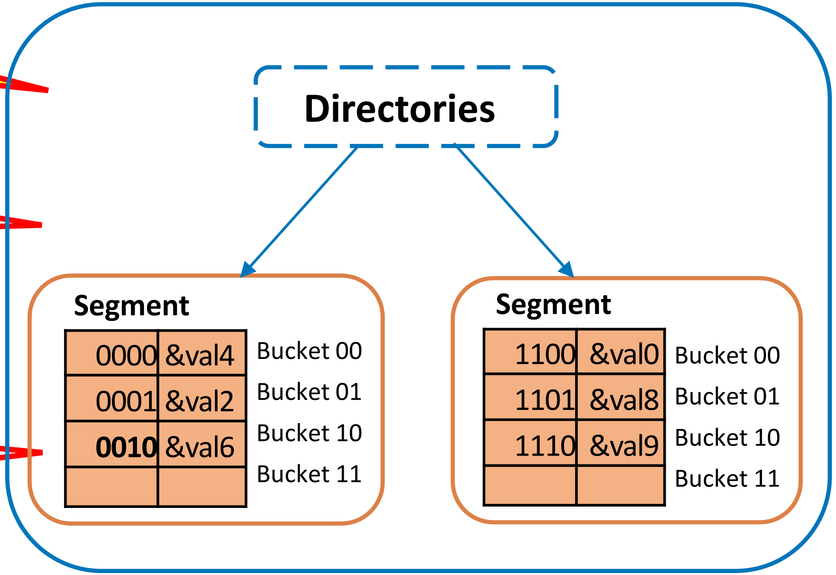


HMEH : Optimistic Concurrency

Mutex and **version number** for directories

Fine-grained lock for segment split
lock-free read

Compare-and-swap Instructions for Slots

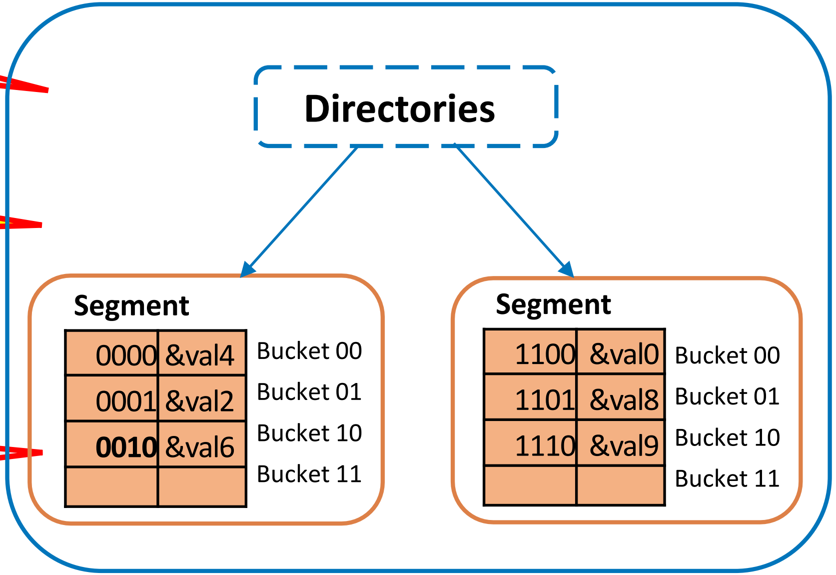


HMEH : Optimistic Concurrency

Mutex and **version number** for directories

Fine-grained lock for segment split
lock-free read

Compare-and-swap Instructions for Slots

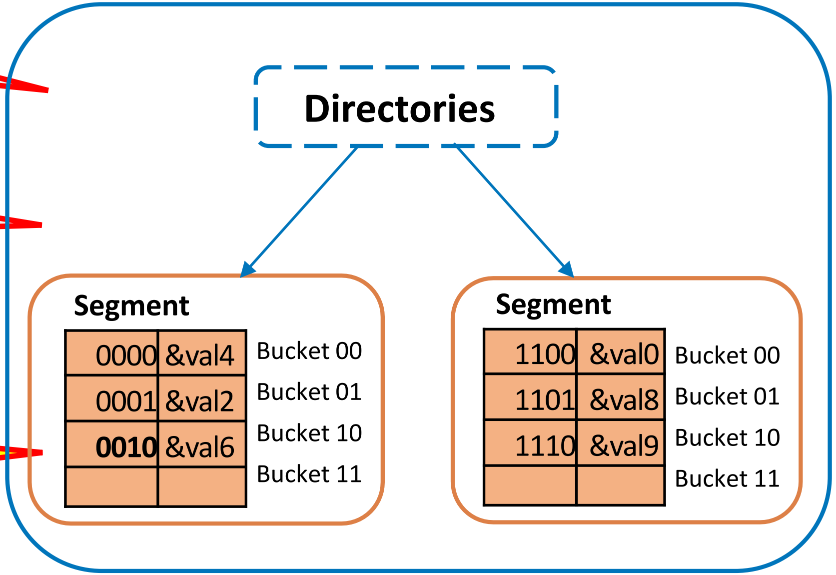


HMEH : Optimistic Concurrency

Mutex and **version number** for directories

Fine-grained lock for segment split
lock-free read

Compare-and-swap Instructions for Slots

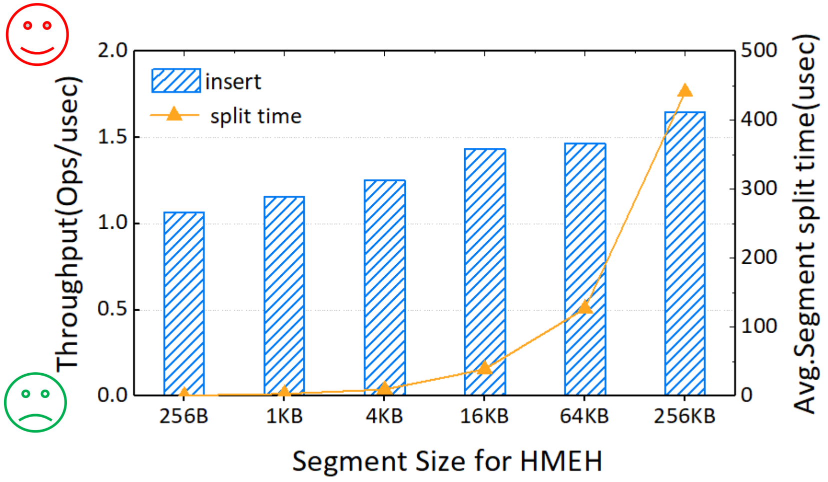


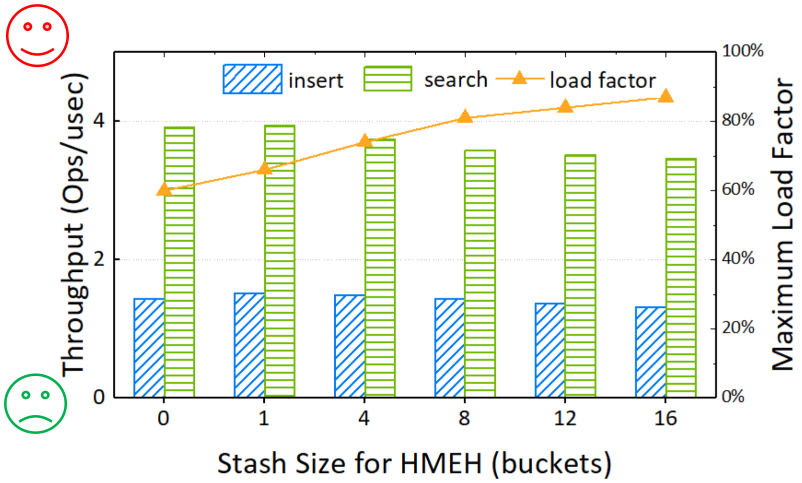
Performance Evaluation

➢Experimental setup

CPU	2-socket 36-core machine with 32MB LLC
Memory	1.5 TB DCPMM, 192GB DRAM
workload	160 Million random number dataset YCSB
Comparisons	CCEH [FAST 2019] LEVL [OSDI 2018] P-CUCK: persistent cuckoo hashing P-LINP: persistent linear probing

Experiment - Sensitivity Analysis

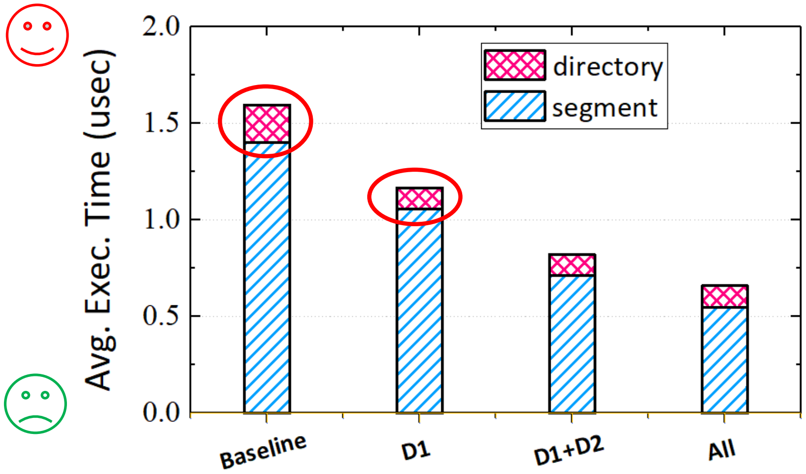


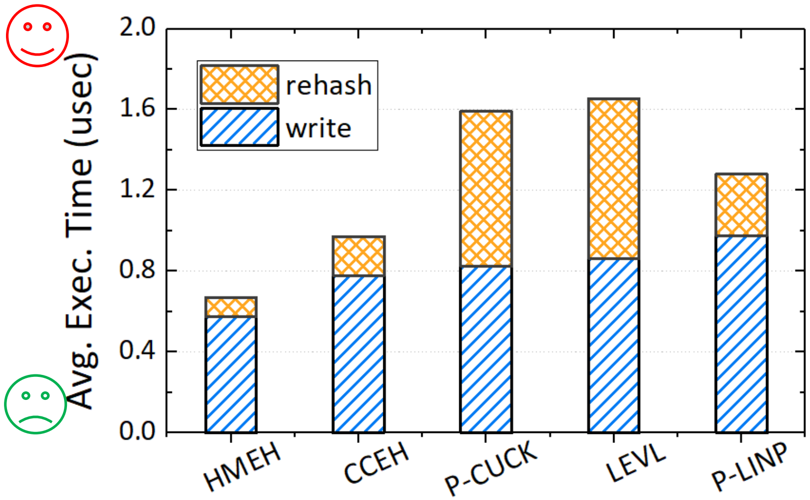


- Segment size
  - ✓ The reasonable segment size is in the range of 4KB to 16KB.
- Stash size
  - ✓ The optimal stash size is between 1 bucket and 8 buckets

✓ **we set the segment size as 16KB with a stash whose size is 4 buckets for the rest of the experiments**

Experiment - Comparative Performance

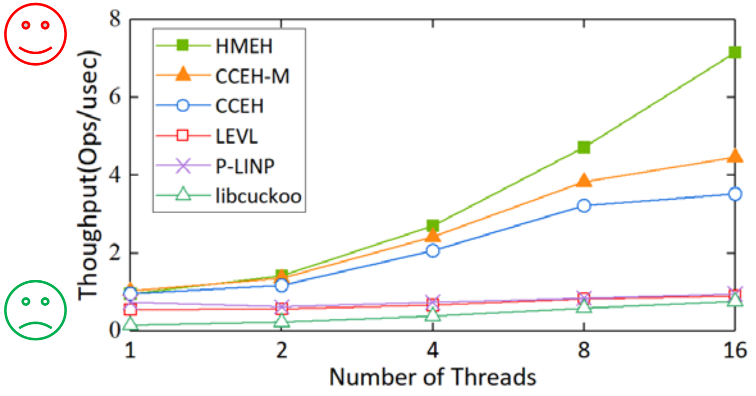




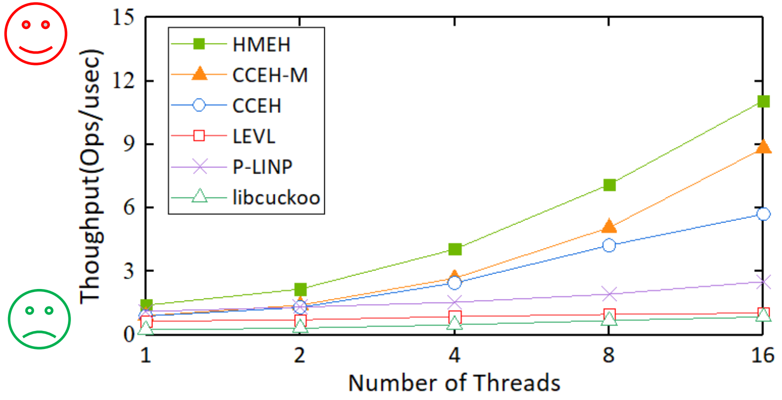
- Design gain
  - ✓ Baseline: EH with persist barriers
  - ✓ D1: the changes of structure
  - ✓ D2: Cross-KV
  - ✓ All: entire HMEH

- Insertion latency of different researches
  - ✓ Compared with CCEH, P-CUCK, LEVL, and P-LINP, HMEH speeds up the insertions by over 1.49×, 2.37×, 2.47×, and 1.91×

Experiment - Concurrent performance

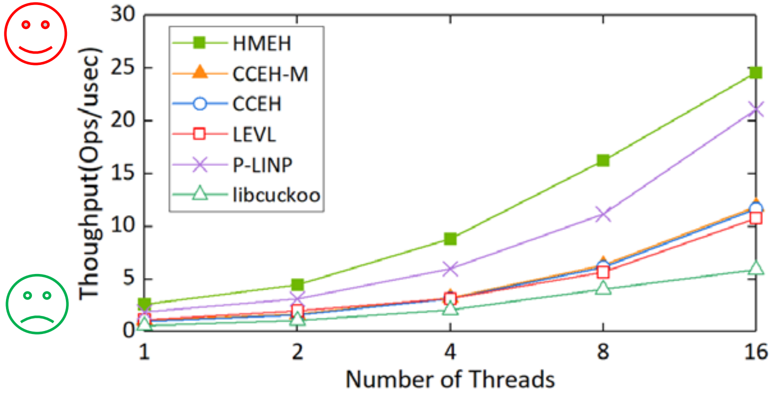


(a) Scalability on YCSB: 100% insert



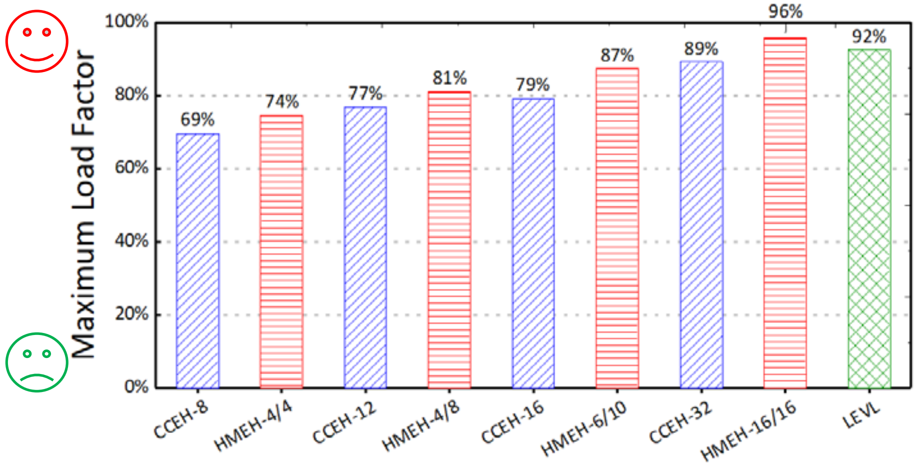
(b) Scalability on YCSB: 50% insert-50% search

- Three YCSB workloads test
  - ✓ Concurrent HMEH also delivers superior performance and high scalability under YCSB workloads with different search/insertion ratios



(c) Scalability on YCSB: 100% search

Experiment – Other evaluations



Number of Indexed Records	1.6 million	16 million	160 million
RT-directory Recovery Time(ms)	0.47	6.3	50.1
FS-directory Rebuild Time(ms)	2.5	21.8	172.2

- Maximum Load Factor
  - ✓ As linear probing distance and stash size grow, the max load factors of HMEH increase stably and all exceed 74%
- Recovery Time of directories
  - ✓ directories of HMEH can achieve an instantaneous recovery

Conclusion

- **Problem**
  - ✓ the structures of previous work have shortcomings
  - ✓ Existing data consistency mechanisms incur high overhead
- **A write-optimal extendible hashing for hybrid memory**
  - ✓ Flat-structured Directory in DRAM for fast access
  - ✓ Radix-tree-structured Directory in NVM for recovery
  - ✓ Cross-KV mechanism
  - ✓ linear probing+stash
  - ✓ Optimistic Concurrency
- **Results**
  - ✓ Outperforms the state-of-the-art work by up to 2.47×
  - ✓ High scalability and fast recovery

Thanks!

Q&A