

HART: A Concurrent Hash-Assisted Radix Tree for DRAM-PM Hybrid Memory Systems

Wen Pan, Tao Xie, Xiaojia Song

San Diego State University, California, USA

Agenda

1. Background & Motivation

2. Design

3. Algorithms

4. Evaluation

5. Conclusions



Background & Motivation

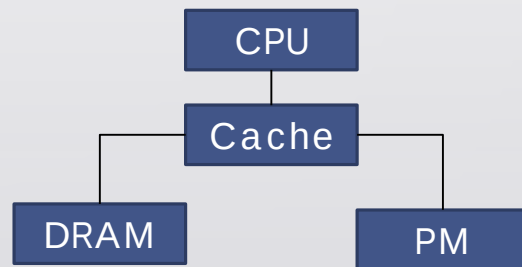
Persistent Memory

- ❑ Persistent memory is driving a rethink of storage systems towards a single-level architecture

	HDD	NAND Flash	STT-MRAM	PCM	DRAM
Read Latency (ns)	5×10^6	2.5×10^4	5 - 30	20 - 70	10
Write Latency (ns)	5×10^6	2×10^5	10 - 100	150 - 220	10
Byte-addressable?	X	X	O	O	O

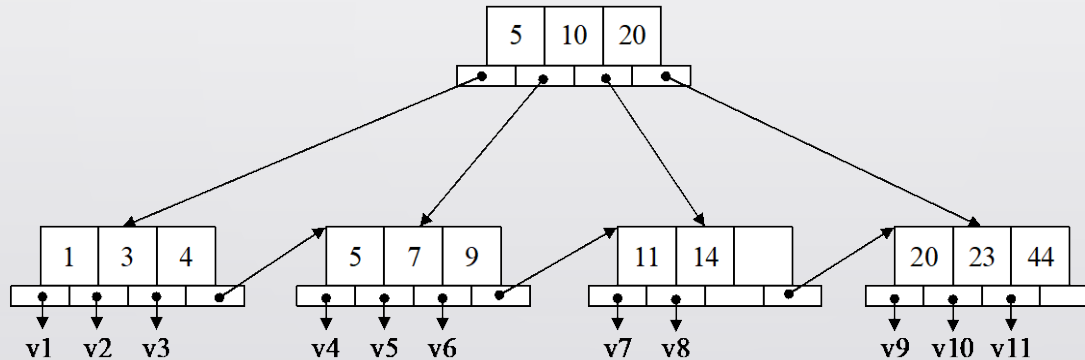
- ❑ Persistent indexing data structures

- Consistency
- Performance
- Preventing persistent memory leak



B+ Tree

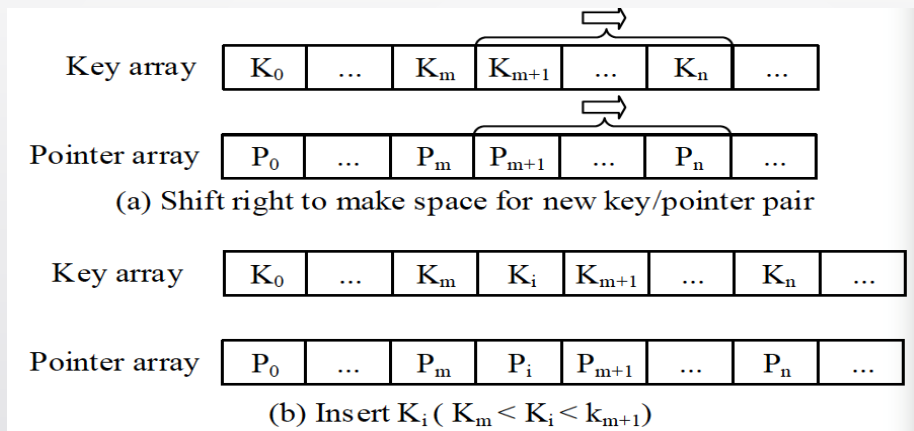
- ❑ Leaf nodes are linked
- ❑ Internal & leaf nodes both have multiple children
- ❑ At least half of a node capacity is used



Shift Operations in a B+ Tree



Keys & pointers need to be shifted to keep the node sorted



Consistent shift on PM can be extremely expensive

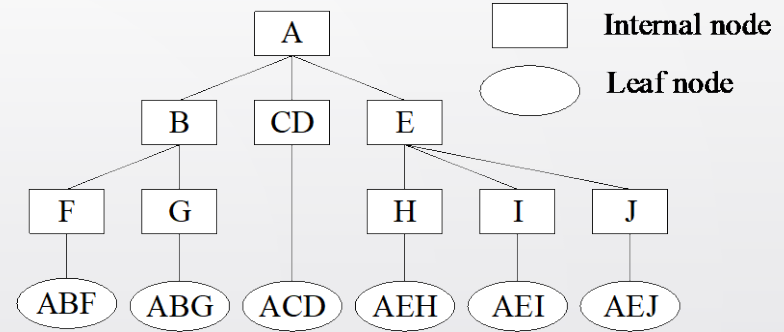
Radix Tree & ART (Adaptive Radix Tree)

Radix Tree

- One-size-fits-all inner nodes

ART (Adaptive Radix Tree)

- Use 4 different kinds of internal nodes (NODE4, NODE16, NODE48, NODE256) depending on the number of children
- Path compression: an internal node is merged with its parent if its parent only has one child



Motivation of HART

- ❑ Compared with B+-trees or radix trees, hash table has better search performance for sparse keys. However, its range query performance is much worse.
- ❑ Without hash collisions, the time complexity of a search/insertion operation is $O(1)$.
- ❑ The scalability of a hash table is not as good as that of a tree, its insertion performance is worse than that of a radix tree
- ❑ To exploit the complementary merits of a radix tree and a hash table, we propose a novel concurrent and persistent tree called HART (Hash-assisted Adaptive Radix Tree), which utilizes a hash table to manage multiple adaptive radix trees (ARTs)

Indexing Trees for PM

	B/B+ Tree variants	Radix Tree variant
Pure PM	CDDS B Tree ² NV-Tree ³	WORT ¹ WOART ¹
PM-DRAM Hybrid	FPTree ⁴	HART

- ❑ Radix tree has been proven to be more efficient than B/B+ trees in both DRAM and persistent memory
- ❑ Persistent B/B+ Tree dilemma:
 - Unsorted keys in a node>Search performance degradation
 - Sorted keys in a node: higher consistency cost

- ❑ Hybrid architecture takes advantage of fast DRAM speed and reduces memory fence/memory flush cost

1. S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh. Wort: Write optimal radix tree for persistent memory storage systems. In FAST, pages 257–270, 2017.
2. S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In FAST, volume 11, pages 61–75, 2011.
3. J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In FAST, volume 15, pages 167–181, 2015.
4. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In Proceedings of the 2016 International Conference on Management of Data, pages 371–386. ACM, 2016.



Design

Design Assumptions

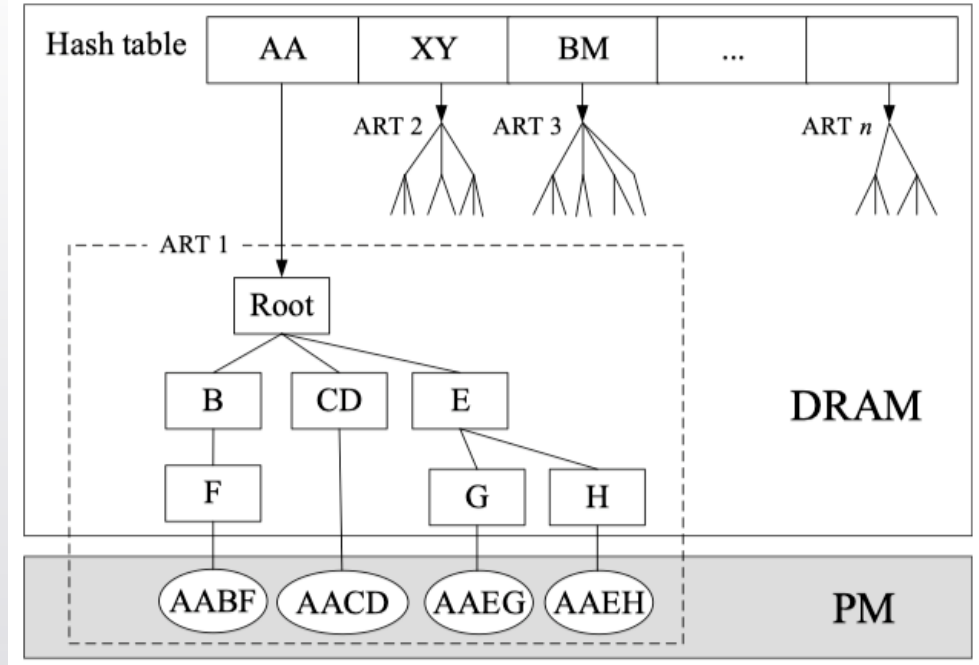
- ❑ PM next to DRAM: PM is connected directly to CPU
- ❑ PM can be accessed by LOAD/STORE semantics
- ❑ 8-byte atomic write: supported by modern CPUs
- ❑ A durable function *persistent()*: *mfence* + *clflush* + *mfence*
- ❑ A *malloc()/free()*-like interface to allocate/free space from persistent memory

Design Principles

- ❑ Hash-assisted ARTs
- ❑ Selective persistence
- ❑ Concurrent access
- ❑ An enhanced persistent memory allocator
- ❑ Variable-size values support
- ❑ Memory leak prevention

Hash-assisted ARTs

- ❑ A hash table manage many ARTs
- ❑ A key is divided into 2 parts: a hash key and an ART key



Selective Persistence

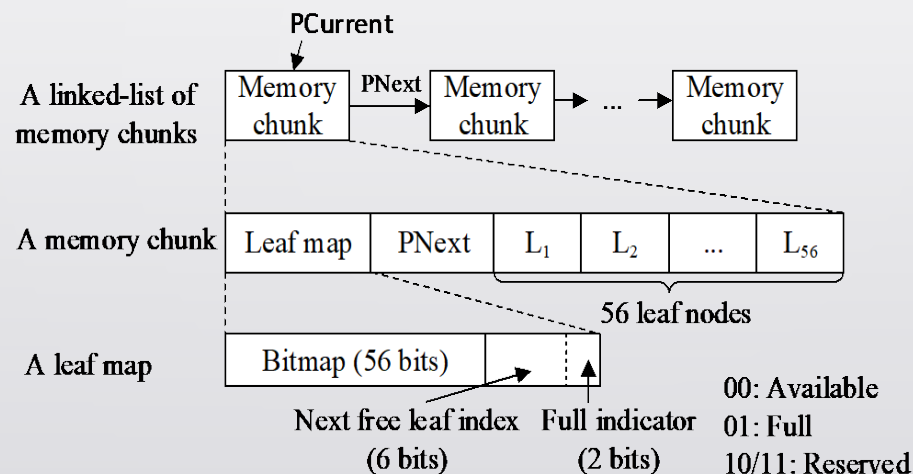
- ❑ Hash table & ART inner nodes stored in DRAM: performance (DRAM speed + sorted keys + no consistency cost)
- ❑ Leaf nodes in PM: the key is also stored in a leaf node

Concurrent access

- ❑ A read/ Write lock on each ART (each bucket of the hash table)
- ❑ Support up to k concurrent writes, where k is the number of ARTs
- ❑ Multiple reads can share a read lock

An Enhanced Persistent Memory Allocator (1)

- ❑ Persistent memory allocation is expensive than DRAM allocation
- ❑ Our strategy: allocates a memory chunk contains multiple leaves
- ❑ Both value space and leaf space are allocated by EPAllocator

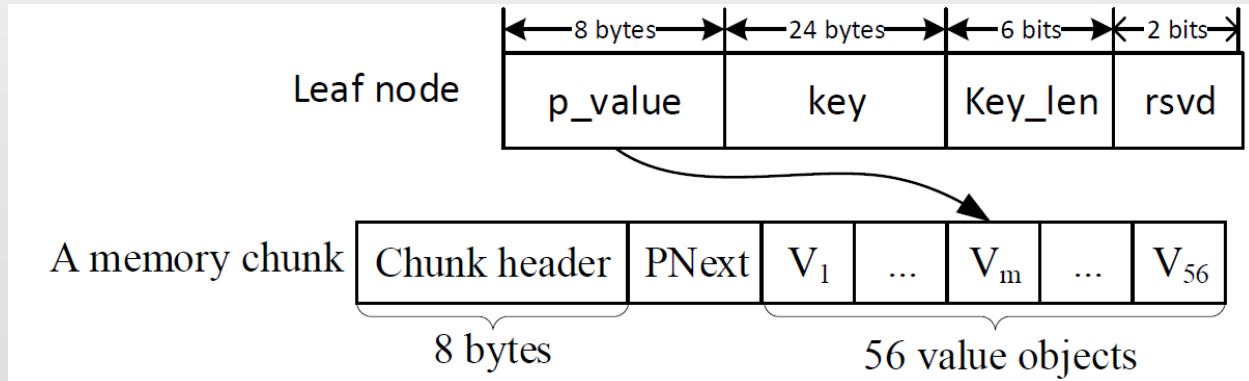


An Enhanced Persistent Memory Allocator (2)

- ❑ 2 functions: `EPMalloc()` & `EPRecycle()`
- ❑ `P_Next` is also used for leaf nodes traversal, which is critical in failure recovery
- ❑ `P_Next` in each memory chunk instead of in each leaf nodes (B+ Tree)
- ❑ Bitmap is used as a commit flag. Only after a leaf node has been successfully inserted into HART, the related bit is set
- ❑ It can prevent persistent memory leak

Variable-size values support

- ❑ HART stores a 8-byte pointer (i.e., p value) to the value in the leaf node
- ❑ HART currently only supports two sizes of value objects: 8-byte values and 16-byte values





Algorithms

Operations: Insertion

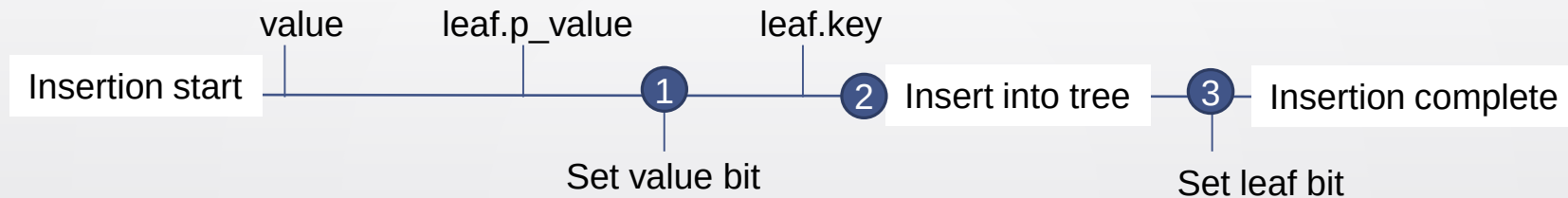
- ❑ 1. Key split to a hash key and an ART key. Find corresponding ART based on the hash key
- ❑ Allocate PM space for a leaf node & value using EPAllocator
- ❑ 2. Update value; `persistent(value)`
- ❑ 3. `leaf.p_value = &value; persistent(leaf.p_value )`
- ❑ 4. Set corresponding value bit in bitmap of enhance PM allocator
- ❑ 5. Update `leaf.key`; `persistent(leaf.key)`
- ❑ 6. Insert into tree with conventional ART algorithm
- ❑ 7. Set and persistent the leaf bit

Insertion Algorithm

Algorithm 3 Insertion(Key K, Value V, HART HT)

```
1: HashKey, ARTKey = SplitKey(K)
2: T = HashFind(HashKey, HT)
3: if !T then
4:   T = NewART()
5:   HashInsert(HashKey, T)
6: leaf = SearchNode(ARTKey, HT)
7: if leaf then
8:   Update(ARTKey, V, leaf)
9: else
10:  leaf = EPMalloc(LEAF)
11:  value = EPMalloc(VALUE)
12:  value = V; persistent(value)
13:  leaf.p_value = &value; persistent(leaf.p_value)
14:  Set and persistent() the corresponding value bit
15:  leaf.key = K; persistent(leaf.key);
16:  leaf.key_len = len(K); persistent(leaf.key_len)
17:  Insert2Tree(T, leaf)
18:  Set and persistent() the corresponding leaf bit
```

Failure Recovery for Insertion



- ❑ Crash happens before 1: no action is needed
- ❑ Crash happens 1~2: This inconsistency can be detected & fixed next time `EPMalloc()` is called
- ❑ Crash happens 2~3: This inconsistency can be detected & fixed by `EPMalloc()` and a check in search function

Operations: Deletion

- ❑ 1. Split a key into a hash key and an ART key. Find the ART based on the hash key
- ❑ On the ART, search for leaf, return NOT_FOUND if not exists
- ❑ 2. Delete leaf from tree using conventional ART algorithm
- ❑ 3. Reset corresponding leaf bit in the bitmap
- ❑ 3. Reset corresponding value bit in the bitmap
- ❑ 4. Call EPRecycle() to check if the related chunks can be recycle

Operations: Deletion

- ❑ Leaf bit is 1 if value bit is 1:
 - Deletion: reset leaf bit -> reset value bit
 - Insertion: Set value bit -> set leaf bit

❑ EPCycle() only works when

the whole chunk is free

Algorithm 5 Deletion(Key K, HART HT)

```
1: HashKey, ARTKey = SplitKey(K)
2: T = HashFind(HashKey, HT)
3: if !T then
4:   return NOT_FOUND
5: leaf = SearchNode(ARTKey, T)
6: if !leaf then
7:   return NOT_FOUND
8: else
9:   DeleteFromTree(T, leaf)
10:  value = leaf.p_value
11:  Reset and persistent() the leaf bit
12:  Reset and persistent() the value bit
13:  EPCycle(MemChunkOf(value))
14:  EPCycle(MemChunkOf(leaf))
15:  if is_empty(T) then
16:    free(T)
```

Operations: Search

- ❑ Search operation: First find the ART, then it is similar to conventional ART search, only add a leaf bit check (value bit check is not necessary)

```
HashKey, ArtKey = SplitKey(key)  
t = HashFind(HashKey)  
leaf = search(t, ArtKey)  
if bitmapGet(leaf)  
    return leaf->value  
else  
    return NOT_FOUND
```

Recovery

- Traverse through all valid leaf nodes in memory chunks and insert into a new HART

```
t = new_hart()
```

```
Get PHead of the memory chunk linked-list: p = P_head
```

```
While (P != NULL)
```

```
    for ( i=0; i < LEAF_NUM_PER_CHUNK; i++)
```

```
        If bitmapGet(p->bitmap, i)
```

```
            Insert(t, p->leaf[i])
```



Evaluation

Persistent Memory Emulation

- ❑ Why: no hardware platform available

- ❑ Challenge part: performance influence by CPU cache need to be considered

- Write: Performance influence can be ignored since *persistent()* evicts data from cache to PM

- Read:

- cache hit: PM latency is hidden by cache

- Cache miss: PM access happens

Emulators

❑ PMEP (Persistent Memory Emulator Platform) by Intel: No longer available

- Use PMFS to manage PM space
- No integrated memory allocator

❑ Quartz by HP: Not accurate in PM-DRAM hybrid mode

- Calls *numa_alloc_onnode()* to mimic PM allocation, which wastes memory and causes severe performance degradation

PM Latency Emulation

□ Our solution:

- Write: Add extra write latencies in every *persistent()* call
- Add extra read latencies offline
 - ▷ Pros: Accurate
 - ▷ Cons: each experiment has to run two times:
 - ▷ Under a pure DRAM environment : get runtime t_{base} on DRAM + extra write latency
 - ▷ Under a emulated DRAM-PM hybrid environment: calculate extra read latency $\delta_{extra_latency}$ caused by accessing PM ($Latency_{rPM} - Latency_{rDRAM}$)

□ Runtime on PM: $t = t_{base} + \delta_{extra_latency}$

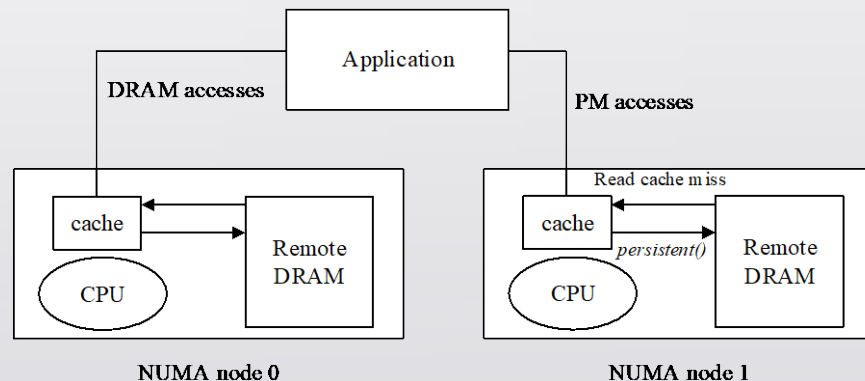
PM Latency Model

Write latency emulation: add extra write latency ($Latency_{wPM} - Latency_{wDRAM}$) in each *persistent()* call

Read latency emulation (considering cache): utilizing CPU counters to get stall cycles S when serving LOAD requests

$$\delta_{stall_cycles} = S * \frac{Latency_{rPM} - Latency_{rDRAM}}{Latency_{rDRAM}}, \quad (1)$$

$$\delta_{extra_latency} = \delta_{stall_cycles} / CPU_frequency, \quad (2)$$



Experimental Setup

- Three PM configurations: Write latency/ Read Latency (ns)

300/100, 300/300, 600/300

- Six workloads:

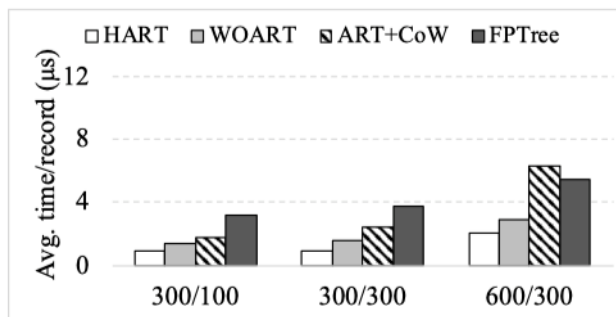
Dictionary, Sequential, Random, and 3 mixed workloads from YCSB

- Compared with WOART[1], ART+COW[1](copy-on-write), FPTree[4]

Insertion performance



(a) Dictionary

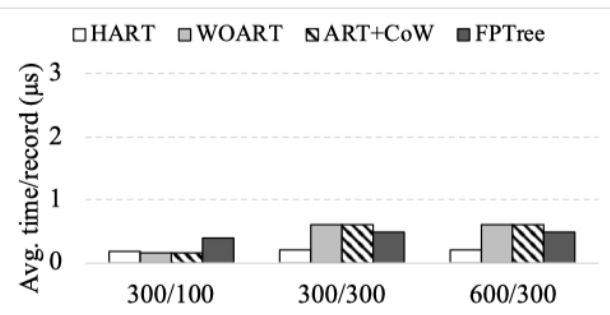


(b) Sequential

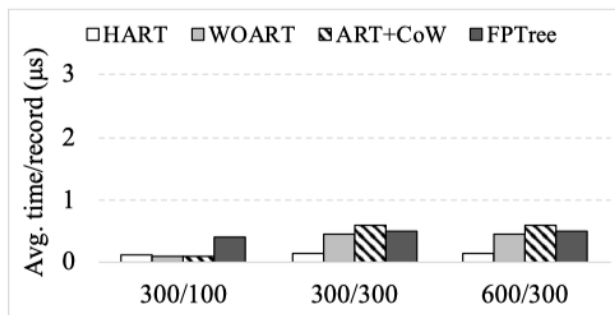


(c) Random

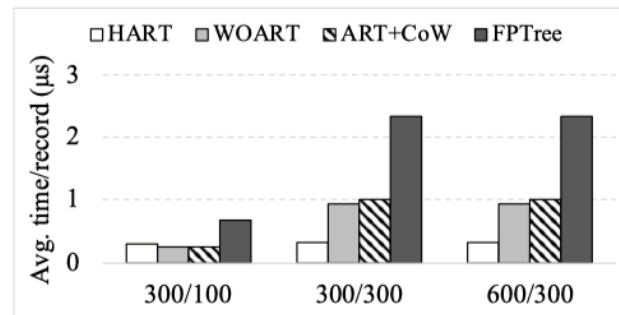
Search Performance



(a) Dictionary



(b) Sequential



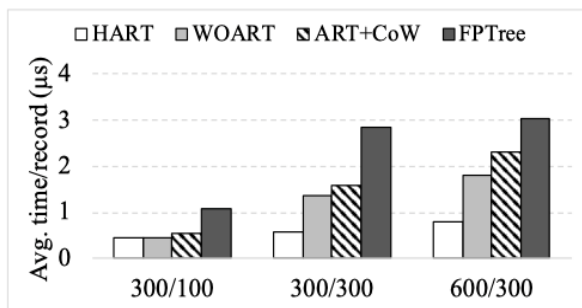
(c) Random

Mixed Workloads

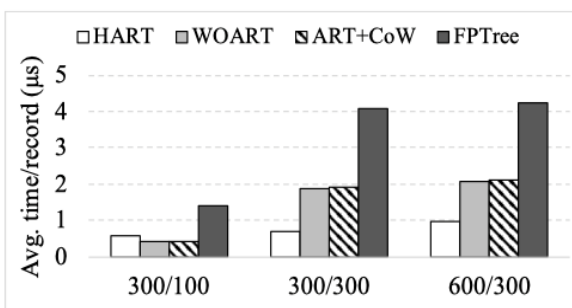
Read-Intensive: 10% insertion, 70% Search, 10% update, 10% deletion

Read-Modified-Write: 50% search, 50% update,

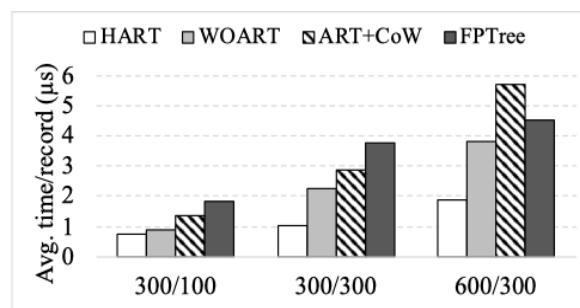
Write Intensive: 40% Insertion, 20% Search, 40% update



(a) Read-Intensive

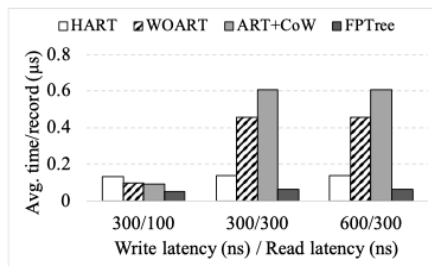


(b) Read-Modified-Write

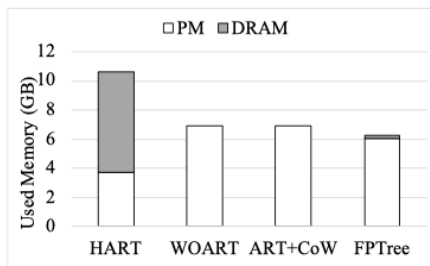


(c) Write-Intensive

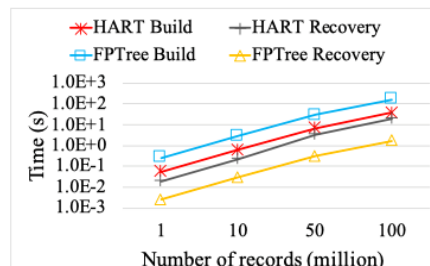
Miscellaneous Results



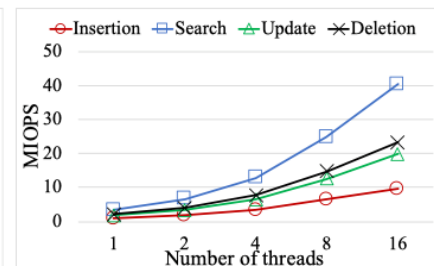
(a) Range query



(b) Memory consumption



(c) Recovery



(d) HART scalability



Conclusions

Conclusions

- ❑ We proposed a new hybrid PM-DRAM persistent tree
 - Selective persistence/consistency
 - Enhanced persistent memory allocator
 - Concurrent access optimization
- ❑ HART shows significant performance improvements
- ❑ HART can be downloaded at <https://github.com/CASL-SDSU/HART>

Acknowledgements

□ This work is sponsored by the U.S. National Science Foundation under grant

CNS-1813485

□ We thank Ismail Oukid for his help in FPTree implementation

□ We thank Bo-Wen Shen for providing us with the Mercury RM102 1U

Rackmount Server

Questions?