NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems

Jun Yang, Qingsong Wei, Cheng Chen,
Chundong Wang, Khai Leong Yong and Bingsheng He

Data Storage Institute, A-STAR, Singapore
Nanyang Technological University
Fast 15

OUTLINE

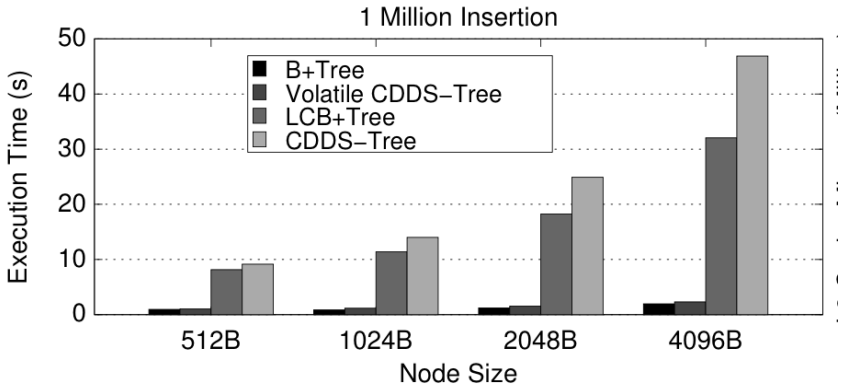
- Motivation
- NV-Tree
- Evaluation
- Related Work
- Conclusion

Motivation

- Next generation of non-volatile memory (NVM)
  - Provides DRAM-like performance and disk-like persistency
  - Can replace both DRAM and disk to build a single level system
- In-NVM data consistency is required
  - Ordering memory writes
  - Fundamental for keeping data consistency
  - Non-trivial in NVM due to CPU design
    - E.g, w1, (MFENCE,CLFLUSH,MFENCE), w2, (MFENCE,CLFLUSH,MFENCE)

Motivation

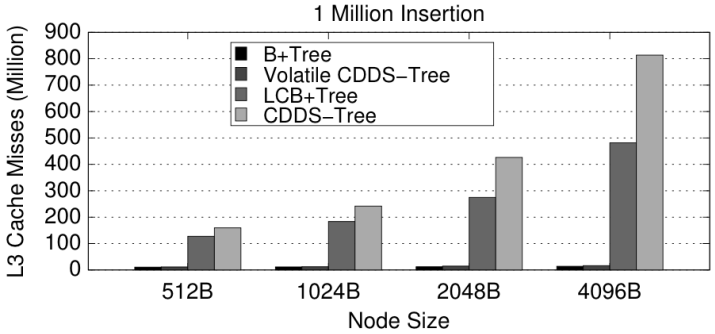
- Making B+tree or its variants consistent is expensive



B+ tree: 16X slower

CDDS-tree: 20X slower

- Reason: CPU cache line invalidation is amplified due to CLFLUSH



Motivation

CFLUSH What?

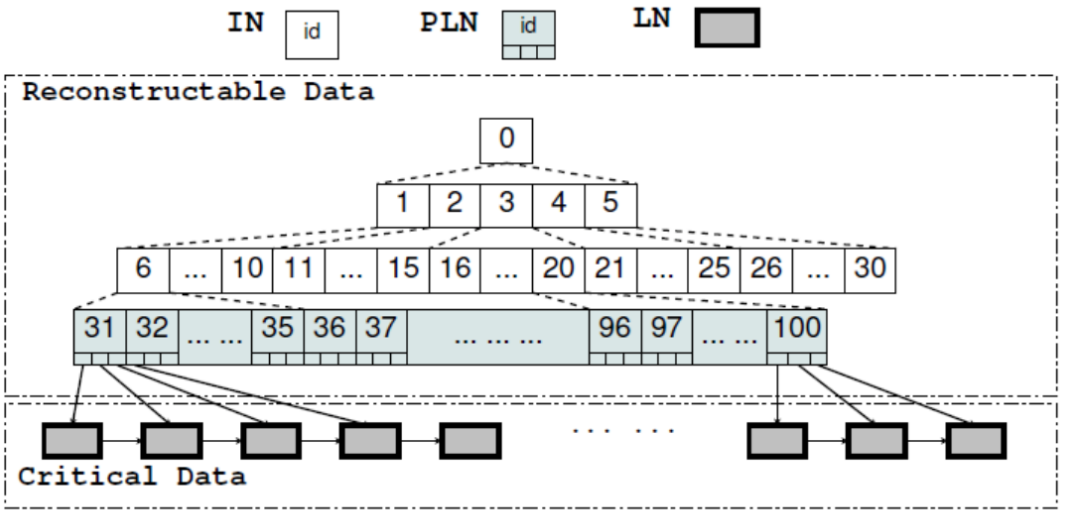
Sorting entries in LN produces up to **>90%** of total CLFLUSH

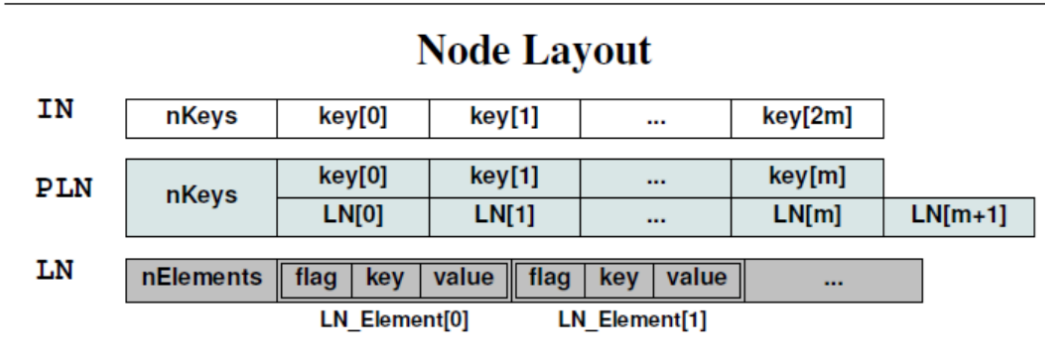
NV-Tree

- Design decisions
  - Selectively Enforce Data Consistency
    - Only enforces consistency on LNs(Leaf Nodes)
  - Keep Entries in LN Unsorted
  - Organizing IN in Cache-optimized Format
    - All INs are stored in a consecutive memory space and located by offset instead of pointers
    - All nodes are aligned to CPU cacheline

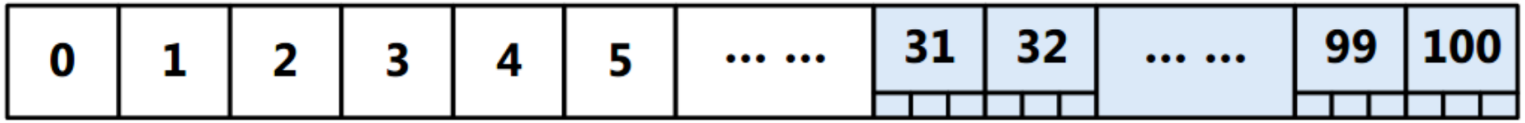
NV-Tree

- Overview



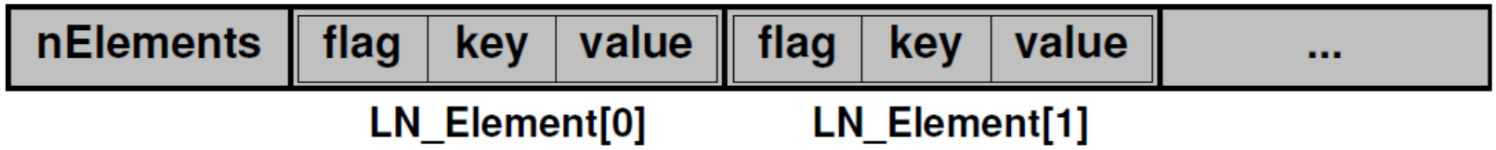


NV-Tree



- IN Design
  - All INs are stored in a continuous memory space
    - Memory address of node id
      = addr + id * size_IN
    - addr : memory address of node 0
    - size_IN : size of a IN
  - Can be located without pointers
  - No consistency required
  - Locating the next IN during tree traverse
    - E.g. one IN have m children
      Memory address of the k-th (k = 1 .. m) child of node id =
      addr + (id * m + k) * size_IN

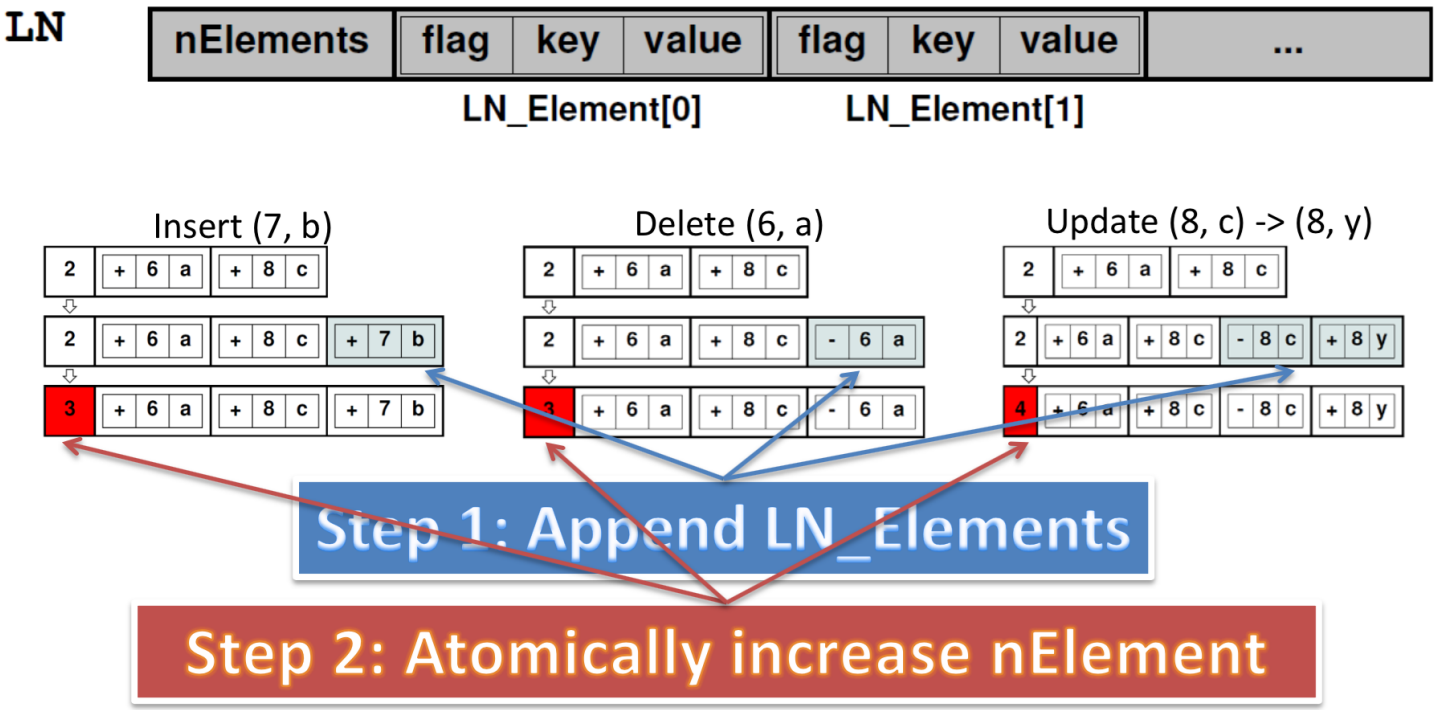
NV-Tree



- LN Design
  - Dynamically allocated and aligned to CPU cacheline
    - Every LN has a pointer from PLN
  - Data is encapsulated in ***LN_Elements***
  - ***LN_Elements*** are unsorted and append-only
  - In-node search is bounded by ***nElements***
  - ***No partial modification***
    - Append ***LN_Element***,(MFENCE,CLFLUSH,MFENCE)
    - Atomically increase ***nElements*** by 1 (8-byte), (MFENCE,CLFLUSH, MFENCE)

NV-Tree

- Insert/Update/Delete



NV-Tree

- Insert Algorithm

Algorithm 2: NV-Tree Insertion

Input: k: key, v: value, r: root
Output: SUCCESS/FAILURE

```
1  begin
2      if r = NULL then /* Create new tree
       with the given KV-pair.                */
3          r ← create_new_tree(k, v);
4          return SUCCESS
5      leaf ← find_leaf(k, r);
6      if LN has space for new KV-pair then
7          newElement ← CreateElement(k, v);
8          flush(newElement);
9          AtomicInc(leaf.number);
10         flush(leaf.number);
11     else
12         leaf_split_and_insert(leaf, k, v)
13     return SUCCESS
```

NV-Tree

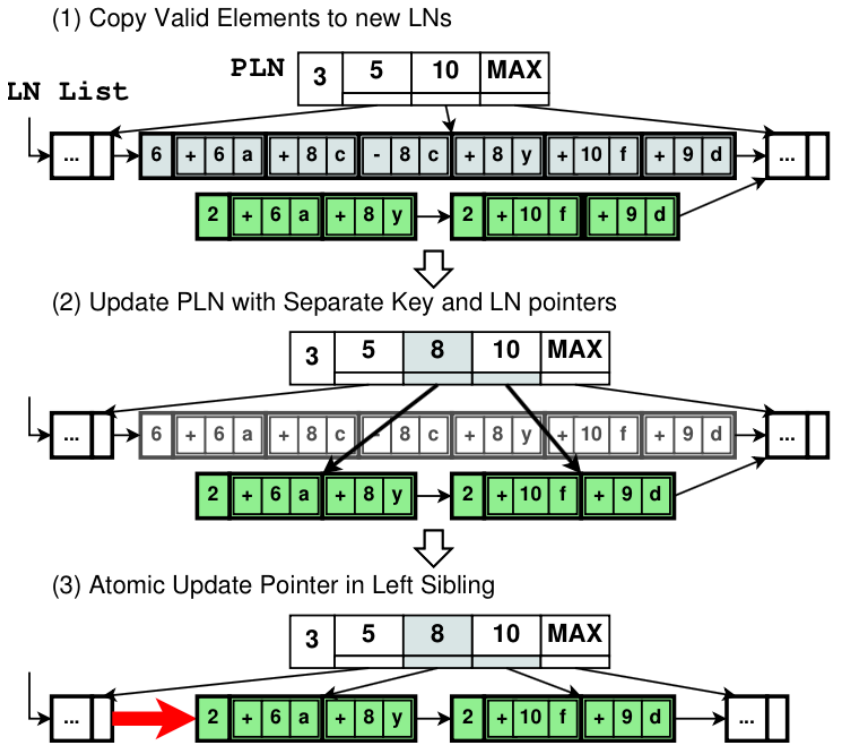


Figure 4: Example of LN Split

- Split
  - All data modified by unfinished split is *invisible* upon system failure
  - Those data become *visible after* a 8-byte atomic update

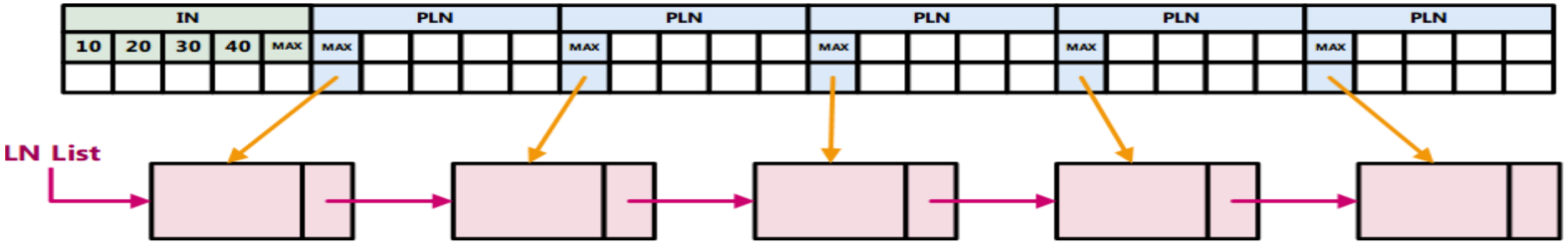
NV-Tree

- Split / Replace / Merge
  - ***Minimal Fill Factor (MFF)***

Percentage of *Valid* Elements in Full Node	Percentage of *Total* Elements in Right Sibling	Action
> MFF	-	**Split**
< MFF	> MFF	**Replace**
< MFF	< MFF	**Merge**

NV-Tree

- Rebuilding
  - Triggered when a PLN is full
    – Due to the fixed position of each IN



- Strategy
  - Rebuild-from-PLN
    - Reuse the existing <key, LN_pointer> array in PLNs
  - Rebuild-from-LN

NV-Tree

- Recovery

Shutdown Type	Shutdown Action	Recovery Action
Normal	Store all INs to NVM	Retrieve the root
System Failure	N/A	Rebuild-from-LN

- Instant recovery
  - Normal shut down and NVM has enough space
    - Keep all INs in NVM
  - Otherwise
    - Rebuilding-from-LN

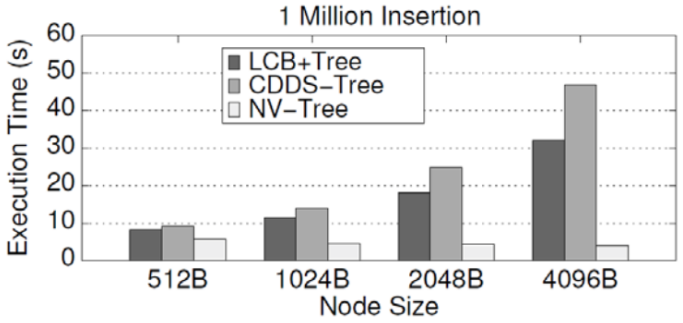
Evaluation

- Experiment Setup

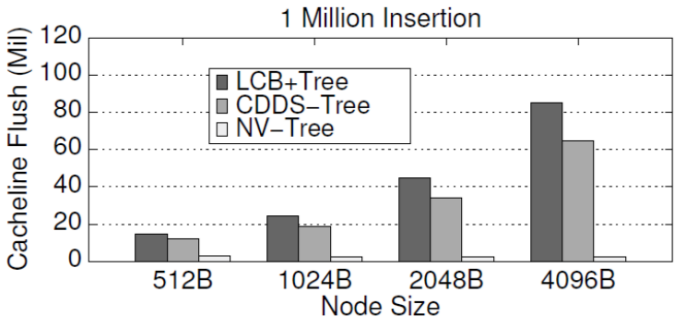
- NVDIMM server
  - IntelXeonE5-2650
  - 2.4GHz, 512KB/2MB/20MB L1/L2/L3 Cache
  - 16GBDRAM,16GBNVDIMM
  - NVDIMM has the same performance as DRAM

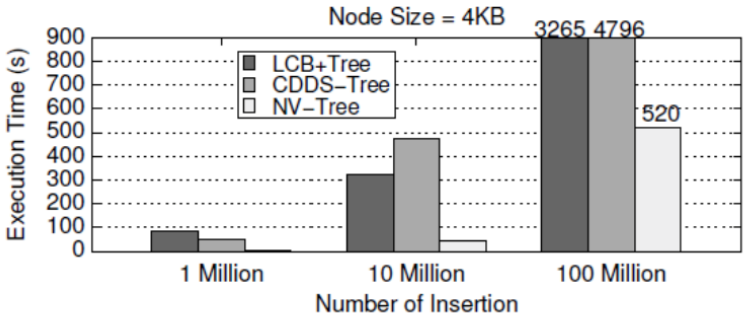
Evaluation-Insertion Performance

- LCB+Tree (Log-based Consistent B⁺Tree)
- CDDS-Tree
- NV-Tree



LCB+Tree	CDDS-Tree
8X	12X

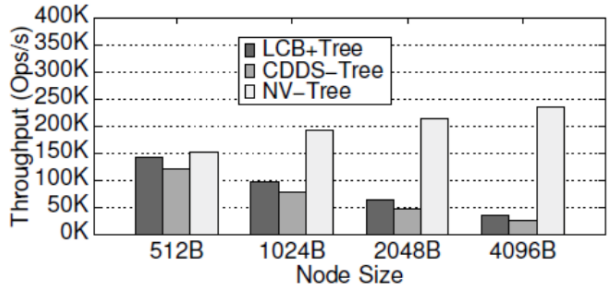




	LCB+Tree	CDDS-Tree
1M	15.2X	8X
10M	6.3X	9.7X
100M	5.3X	8.2X

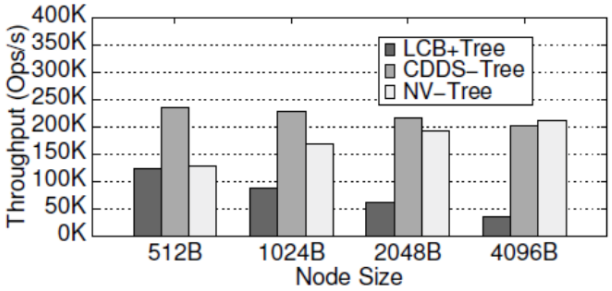
Evaluation-Update/Delete/ Search

Update



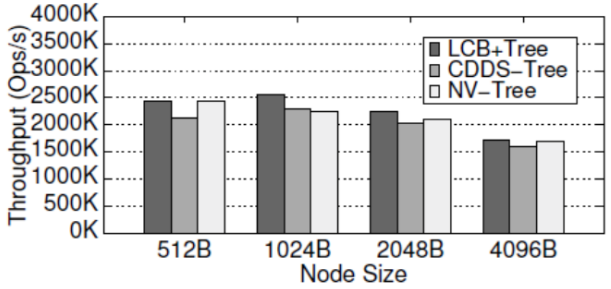
LCB+Tree	CDDS-Tree
5.6X	8.5X

Delete



Comparable to CDDS-Tree with larger nodes

Search

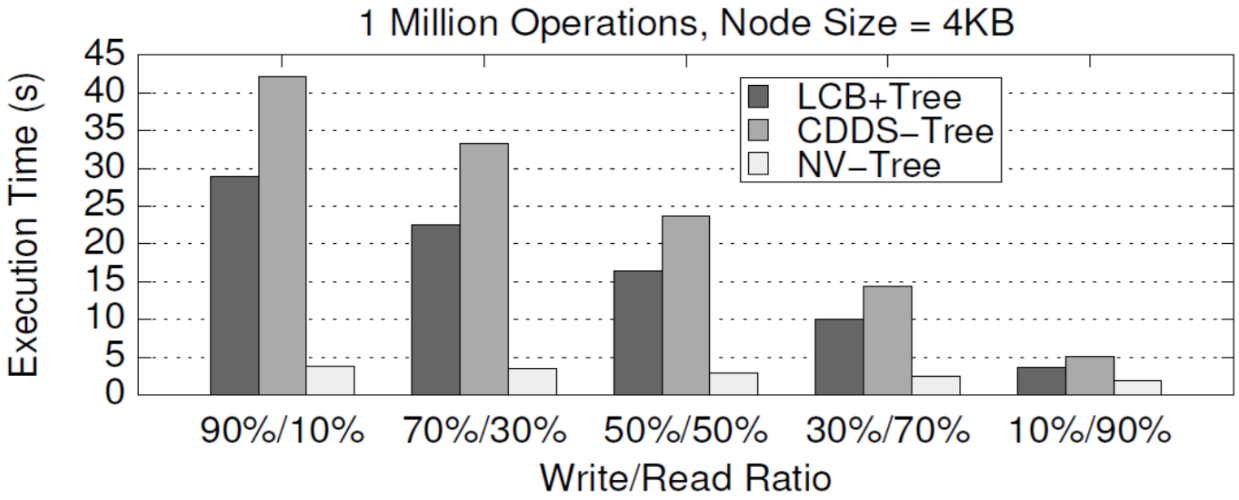


Comparable to both competitors

Evaluation-Mixed Workloads

1 million operations (insertion/search)

– On an existing NV-Tree with 1 million entries

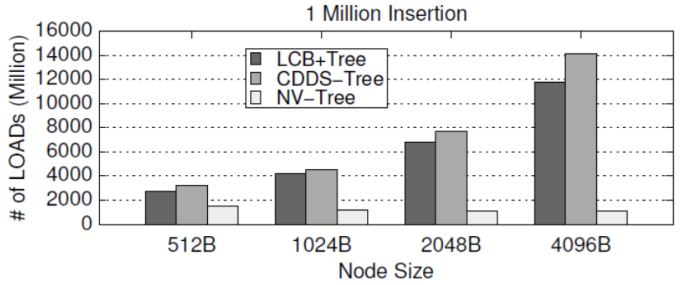


w/r	LCB+Tree	CDDS-Tree
90%/10%	6.6X	10X
10%/90%	2X	2.8X

Evaluation-CPU Cache Efficiency

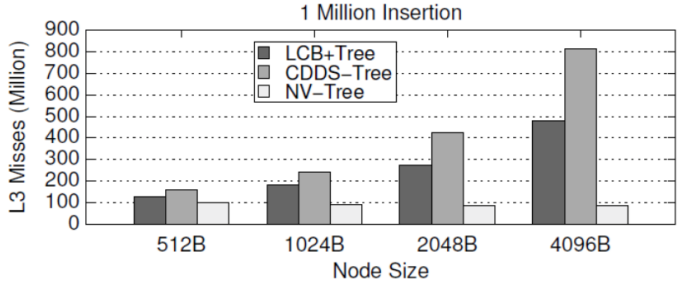
Intel vTune Amplifer

– Number of LOADs



LCB+Tree	CDDS-Tree
Up to **90%** reduced	Up to **92%** reduced

– Number of L3 Misses



LCB+Tree	CDDS-Tree
Up to **83%** reduced	Up to **90%** reduced

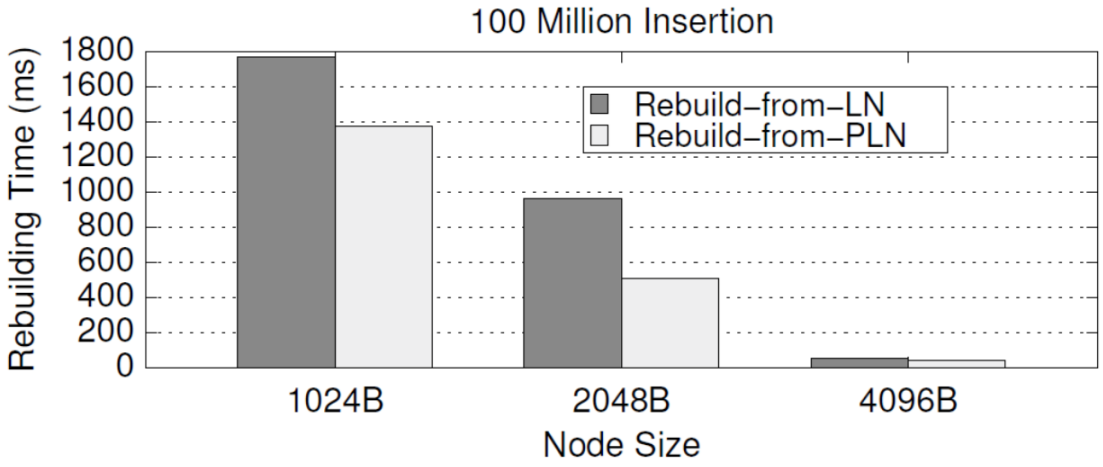
Evaluation-Rebuilding

1/10/100 Million Insertion, 512B/1KB/2KB/4KB Node Size

- Rebuilding time is neglectable
  - 0.01% - 2.77%

Rebuilding strategy

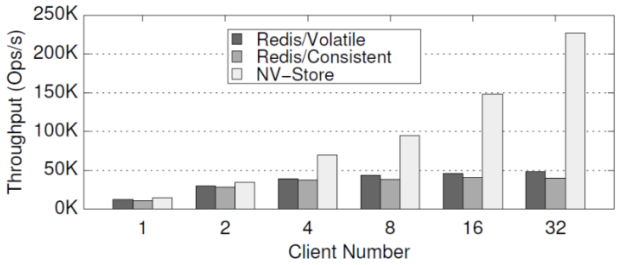
- Rebuild-from-PLN is 22% - 47% faster



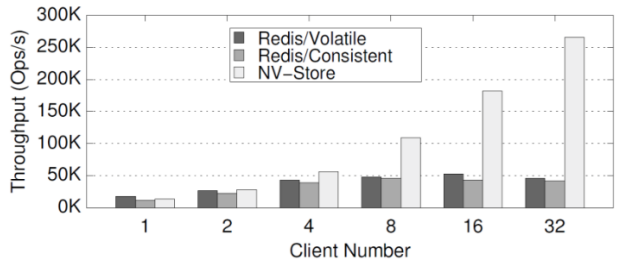
Evaluation-End To End Performance

KV-Stores
- NV-Store
- Redis
  - Volatile / Consistent

Workloads
- YCSB
  - StatusUpdate (read-latest)



  - SessionStore (update-heavy)



5% Insertion
Up to **3.2X** speedup

Scalability

50%/50% Search/Update
Up to **4.8X** speedup

Related Work

- CDDS-Tree
  - Uses flush to enforce consistency on all the tree nodes.
  - In order to keep entries sorted, when an entry is inserted to a node, all the entries on the right side of the insertion position need to be shifted.
- Others
  - MFENCE and CLFLUSH
    - E.g. Mnemosyne
  - Epoch
    - E.g BPFS NV-Heaps

Conclusion

Providing data consistency for B$^+$tree or its variants in Non-volatile Memory is **costly**

- *Ordering memory writes* is non-trivial and expensive in NVM
- Logs are needed because the size of atomic writes is limited
- Keeping in-node data *sorted* produces *unnecessary* ordered writes

NV-Tree

- Consistent, log-free and cache-optimized
- Decouple leaf nodes (LNs) and internal nodes (INs)
  - LN
    - Enforce consistency
    - Unsorted keys with append-only scheme
  - IN
    - Reconstructable, no consistency guaranteed
    - Sorted keys and cache-optimized layout