

NV-Tree

Reducing Consistency Cost for NVM-based Single Level Systems

Jun Yang¹, Qingsong Wei¹, Cheng Chen¹, Chundong Wang¹,
Khai Leong Yong¹ and Bingsheng He²

¹ Data Storage Institute, A-STAR, Singapore

² Nanyang Technological University

Overview

- 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
- Results:
 - Reduce CPU cache line flush by 82-96%
 - 12X faster than existing approaches under write-intensive workloads
 - NV-Store, a KV-store prototype
 - Up to 4.8X faster than Redis under YCSB workloads

Motivation

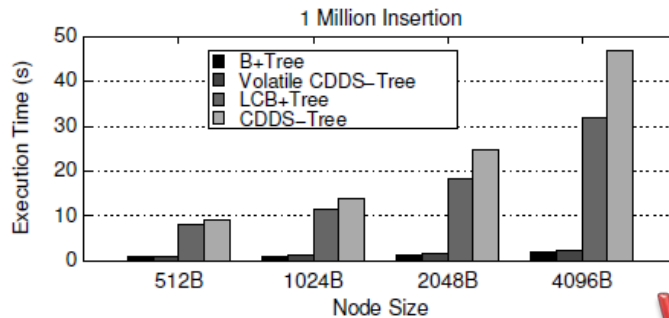
- Next generation of non-volatile memory (NVM)
 - Provides DRAM-like performance and disk-like persistency
 - Can replace both DRAM and disk to build a single level system

| | CPU | | CPU | |
|-------------|----------|-------------------------------|--------------------------------|------------------------------------|
| | Category | Read Latency (<i>ns</i>) | Write Latency (<i>ns</i>) | Endurance (# of writes per bit) |
| | SRAM | 2-3 | 2-3 | ∞ |
| | DRAM | 15 | 15 | 10^{18} |
| | STT-RAM | 5-30 | 10-100 | 10^{15} |
| Persistency | PCM | 50-70 | 150-220 | 10^8 - 10^{12} |
| | Flash | 25,000 | 200,000-500,000 | 10^5 |
| | HDD | 3,000,000 | 3,000,000 | ∞ |

- **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

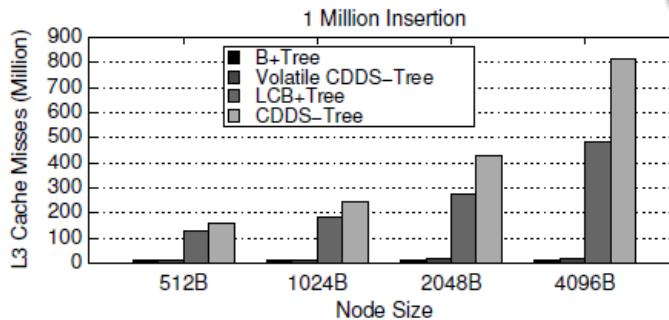
- Making B+tree or its variants consistent is expensive



B+tree : 16X slower

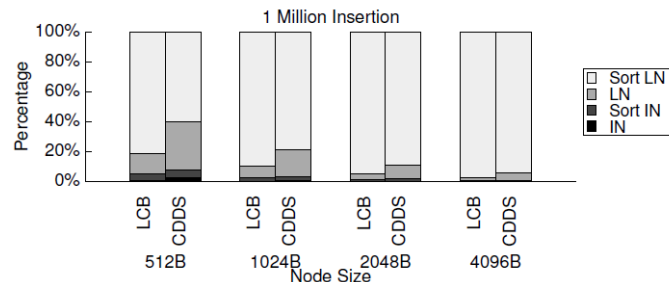
CDDS-tree : 20X slower

WHY?






CPU cache line invalidation
is **amplified** due to CLFLUSH

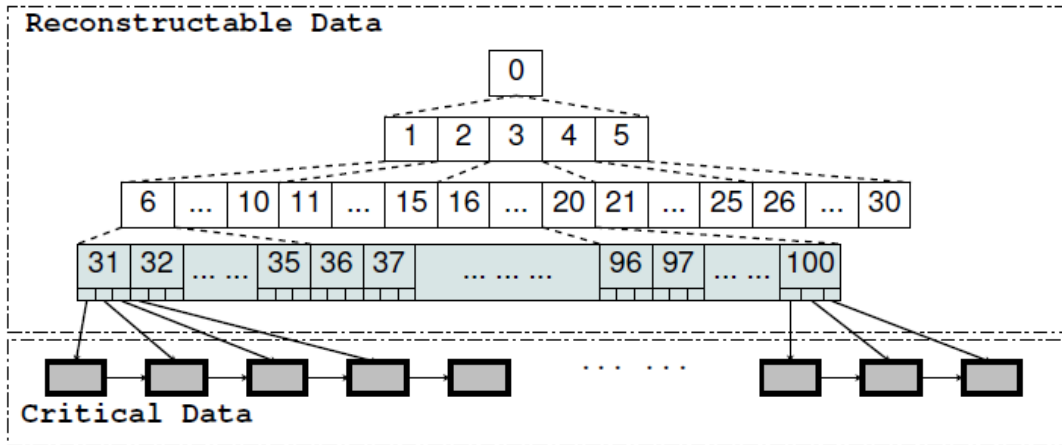
CLFLUSH FOR WHAT?



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

NV-Tree Overview

IN  PLN  LN 

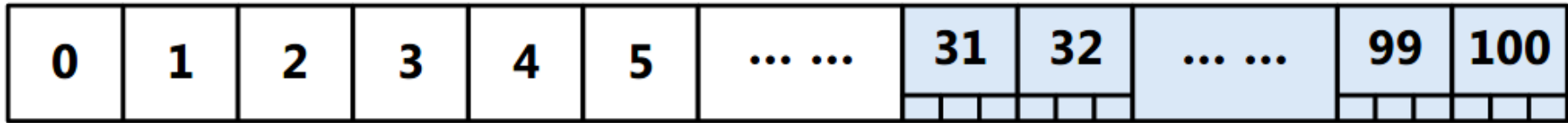


Node Layout

| | | | | | | | | |
|-----|-----------|---------------|--------|-------|---------------|---------|-------|-----|
| IN | nKeys | key[0] | key[1] | ... | key[2m] | | | |
| PLN | nKeys | key[0] | key[1] | ... | key[m] | | | |
| | | LN[0] | LN[1] | ... | LN[m] | LN[m+1] | | |
| LN | nElements | flag | key | value | flag | key | value | ... |
| | | LN Element[0] | | | LN Element[1] | | | |

- Decouple LN and IN
 - Critical data – LN
 - Reconstructable data - IN
- Unsorted LN
 - entries are encapsulated and **appended** in LN
- Cache-optimized IN

IN Design



- IN layout
 - All INs are stored in a *continuous* memory space
 - Memory address of node *id*

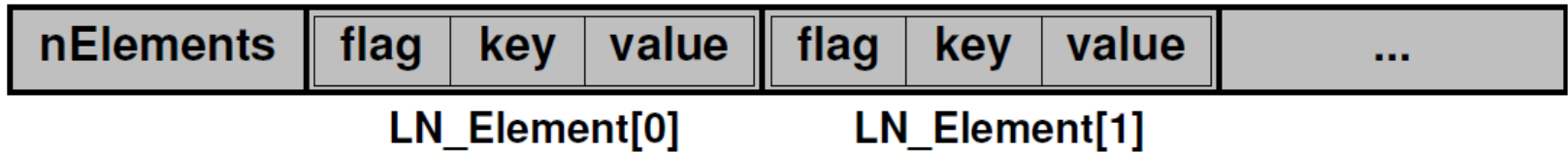
$$= \text{addr} + \text{id} * \text{size_IN}$$

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*

$$= \text{addr} + (\text{id} * m + k) * \text{size_IN}$$

LN Design

LN

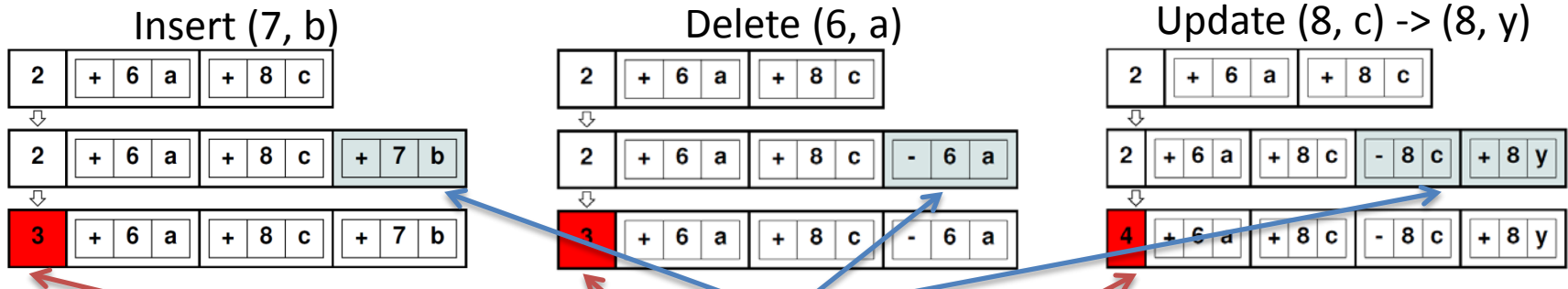


- LN layout
 - Dynamically allocated and aligned to CPU cache line
 - Every LN has a pointer from PLN
 - Data is encapsulated in **LN_Elements**
- **LN_Elements** are **unsorted** and **append-only**
- In-node search is bounded by **nElements**
- **No partial modification**
 1. Append **LN_Element**, (MFENCE, CLFLUSH, MFENCE)
 2. Atomically increase **nElements** by 1 (8-byte), (MFENCE, CLFLUSH, MFENCE)
- Reads are never blocked by writes

Insert/Delete/Update

LN

| nElements | flag | key | value | flag | key | value | ... |
|---------------|------|-----|-------|---------------|-----|-------|-----|
| LN_Element[0] | | | | LN_Element[1] | | | |



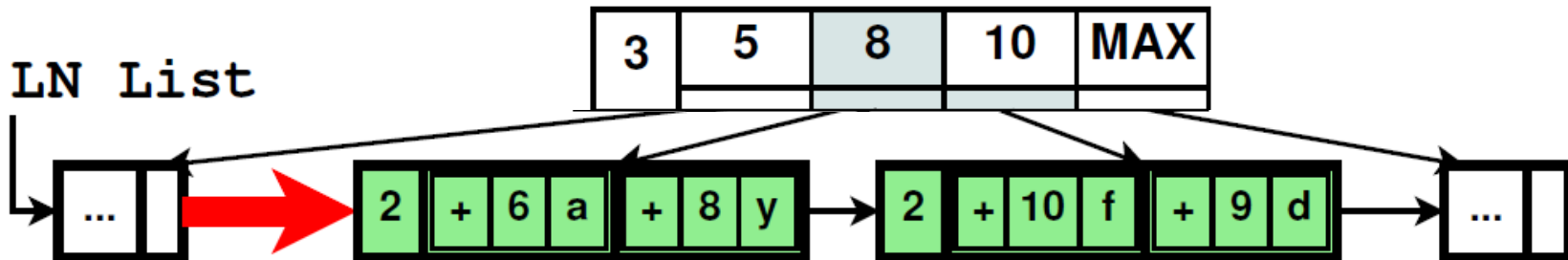
Step 1: Append LN_Elements

Step 2: Atomically increase nElement

Split

- **No partial split**

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



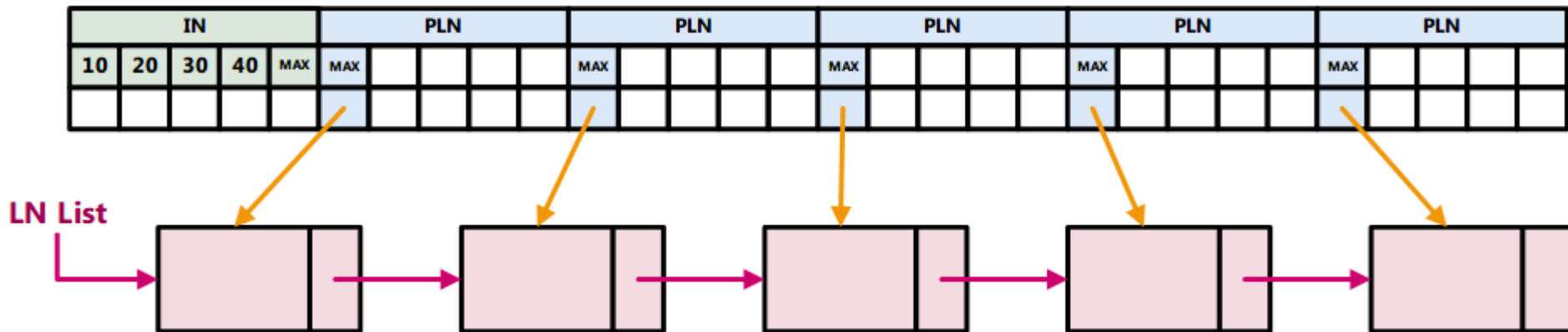
- Split / Replace / Merge

- **Minimal Fill Factor (MFF)**

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

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

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 shutdown and NVM has enough space
 - Keep all INs in NVM
- Otherwise
 - Rebuilding-from-LN

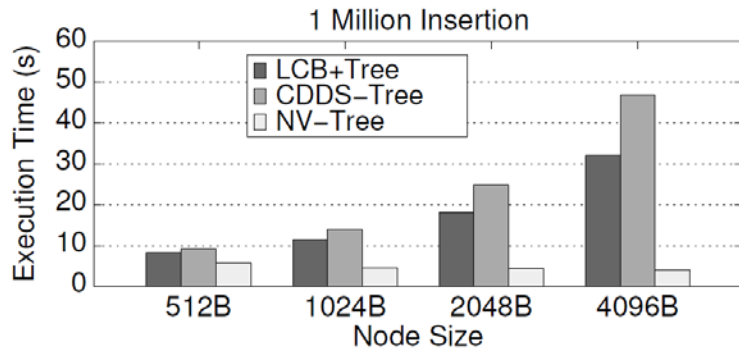
Experiment Setup



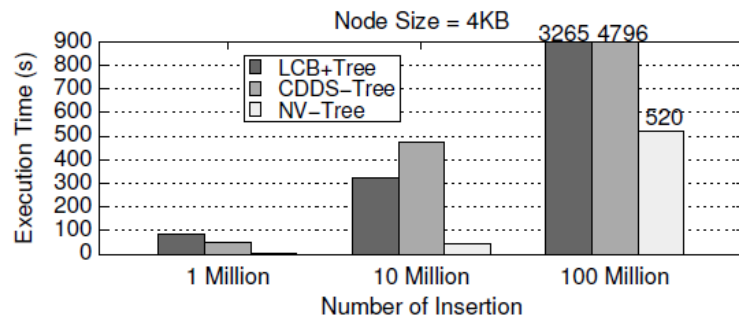
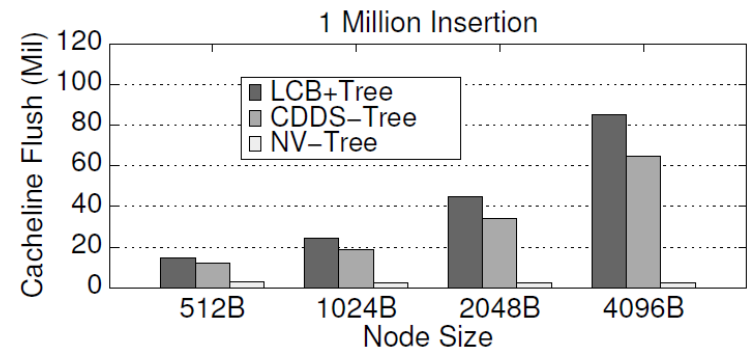
- NVDIMM server
 - Intel Xeon E5-2650
 - 2.4GHz, 512KB/2MB/20MB L1/L2/L3 Cache
 - 16GB DRAM, 16GB NVDIMM
 - NVDIMM has the same performance as DRAM

Insertion Performance

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



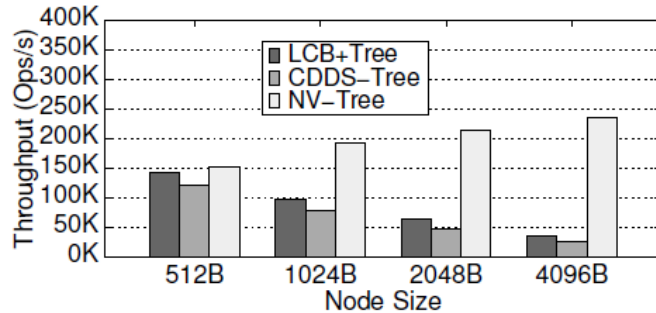
| LCB+Tree | CDDS-Tree |
|----------|-----------|
| 8X | 12X |



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

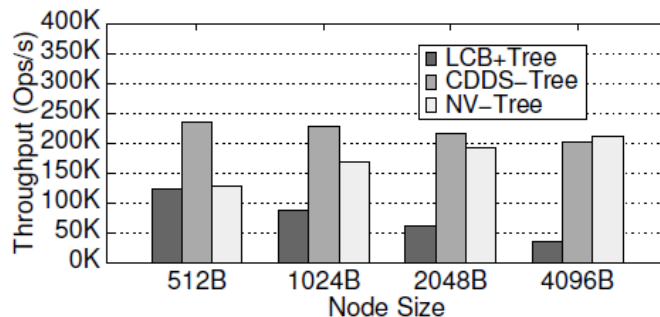
Update/Delete/Search Throughput

- Update



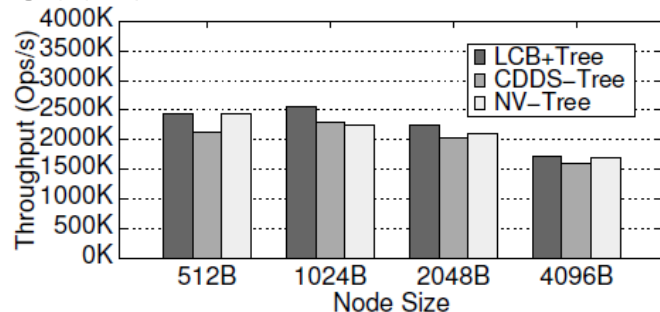
| LCB+Tree | CDDS-Tree |
|----------|-----------|
| 5.6X | 8.5X |

- Delete



Comparable to CDDS-Tree with larger nodes

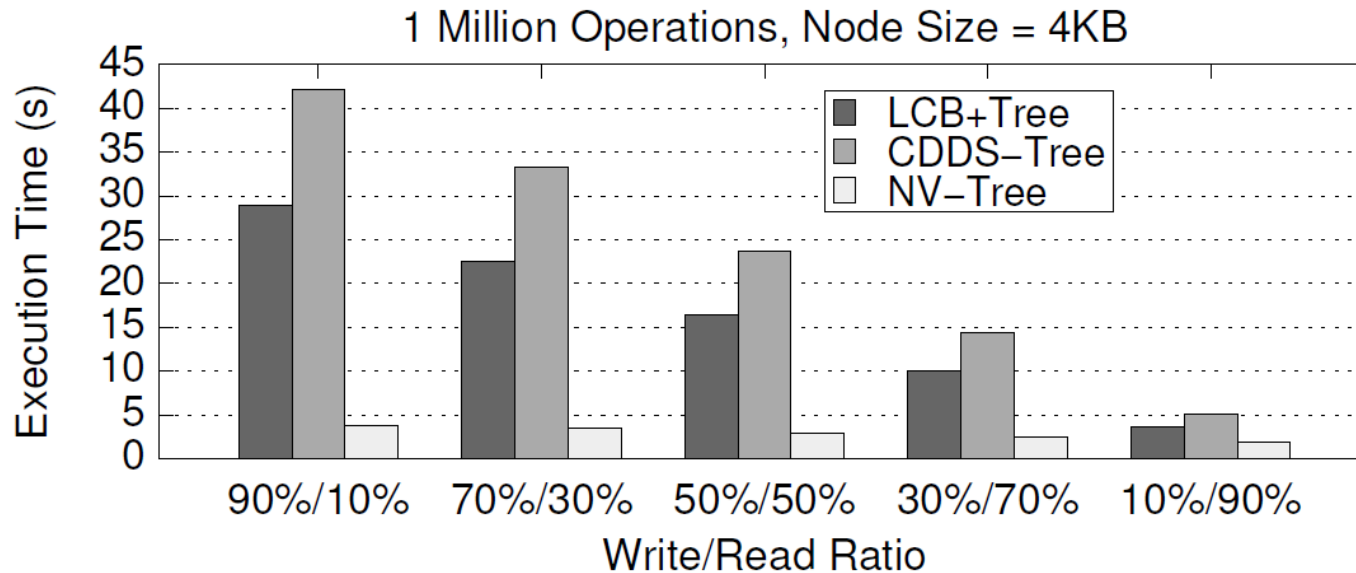
- Search



Comparable to both competitors

Mixed Workloads

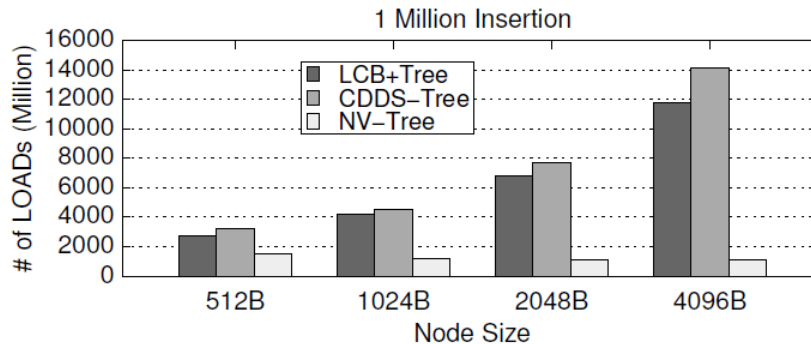
- 1 million operations (insertion/search)
 - On an existing NV-Tree with 1 million entries



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

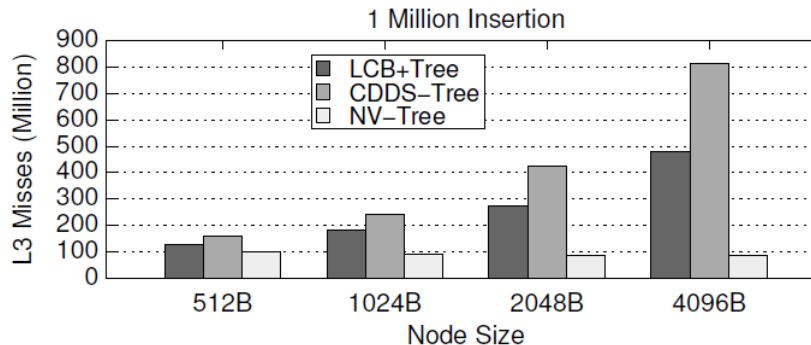
CPU Cache Efficiency

- Intel vTune Amplifier
 - Number of LOADs



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

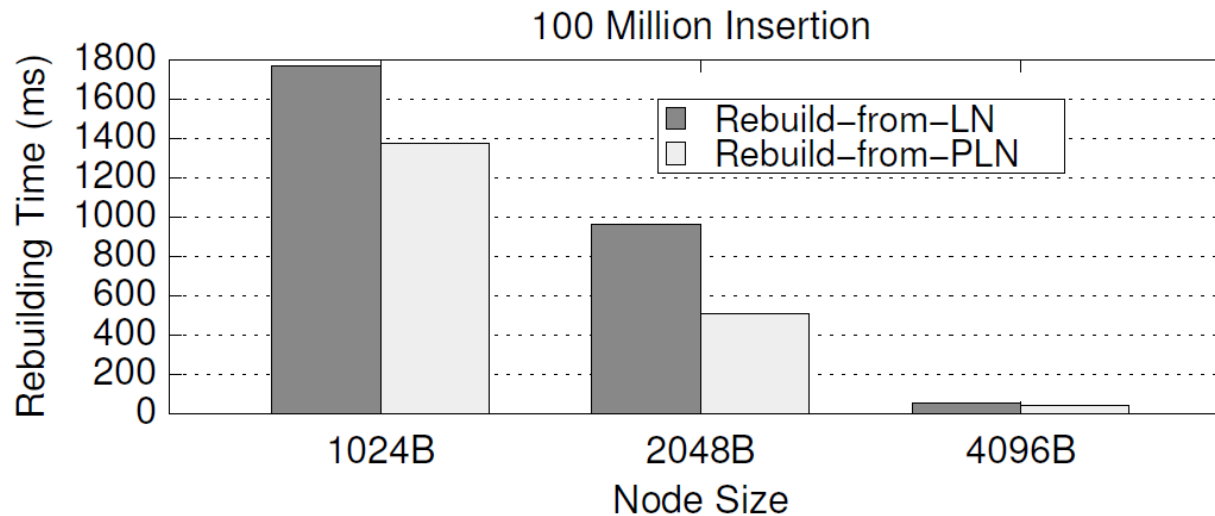
- Number of L3 Misses



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

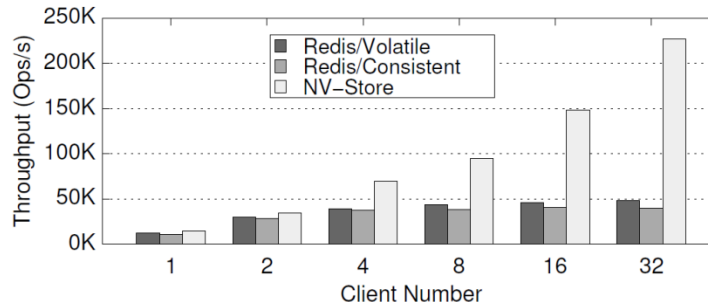
Rebuilding

- 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

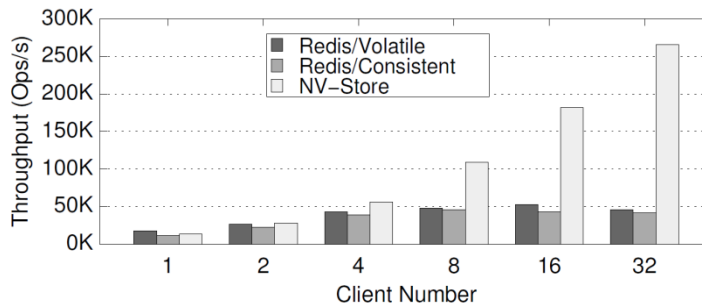


End-to-End Performance

- 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

Additional Materials in The Paper

- NV-Tree performance simulation on different types of NVM
 - Different read/write performance
 - STT-MRAM / PCM
- Future hardware support
 - Epoch
 - CLWB/CLFLUSHOPT/PCOMMIT



THANK YOU!

Q & A

Email: yangju@dsi.a-star.edu.sg

CREATING AN INNOVATION ECONOMY