Easy Lock-Free Programming in Non-Volatile Memory

Tianzheng Wang
Justin Levandoski
Paul Larson

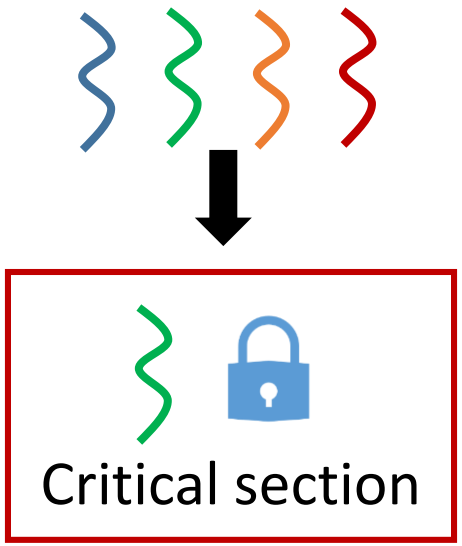
SFU
SIMON FRASER UNIVERSITY

aws

UNIVERSITY OF WATERLOO

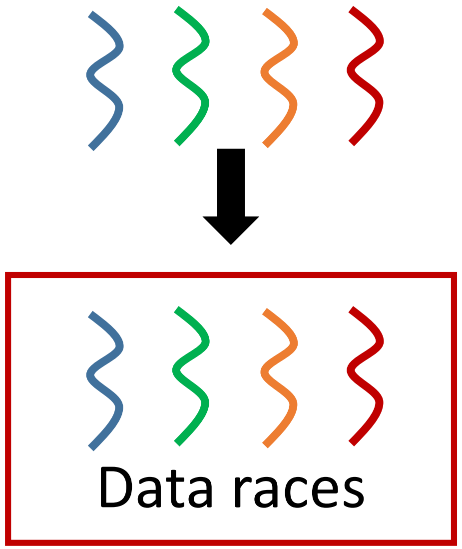
The making of concurrent data structures

- With locks: one thread at a time



- Limited concurrency
- Deadlocks
- Relatively easy

- Lock-free: use atomic instructions directly



- More concurrency, faster
- Higher CPU utilization
- Extremely difficult

Lock-free data structures

- Queues
- Hash tables
- Trees
- Linked lists and skip lists

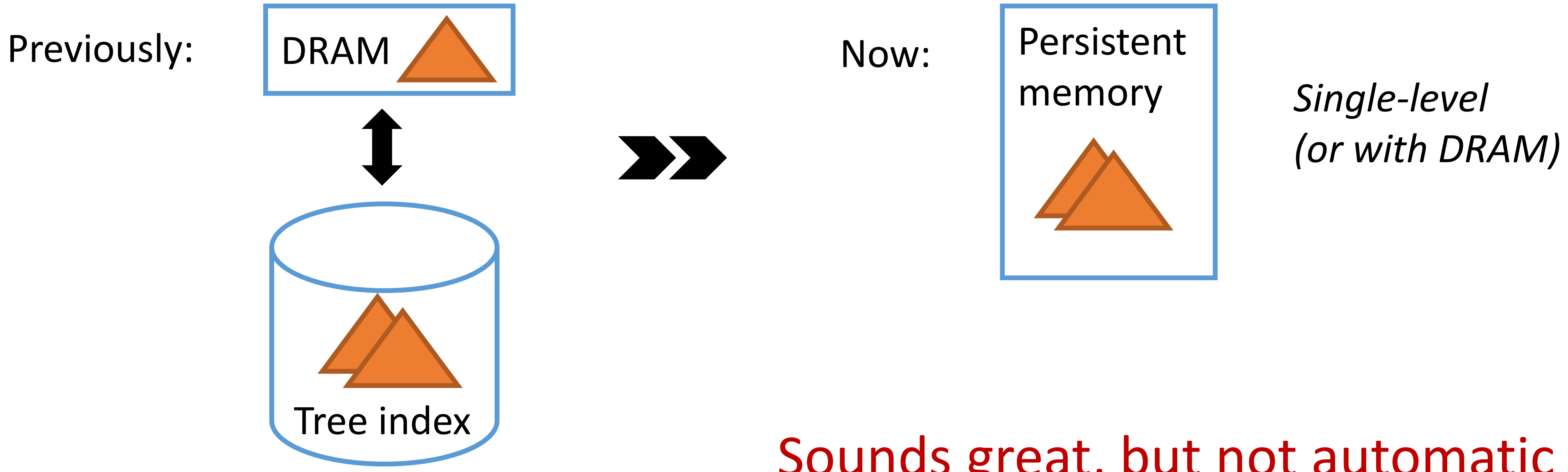
. . .

+ many more . . .

Widely used in performance-critical systems

Lock-free in persistent memory: more potential

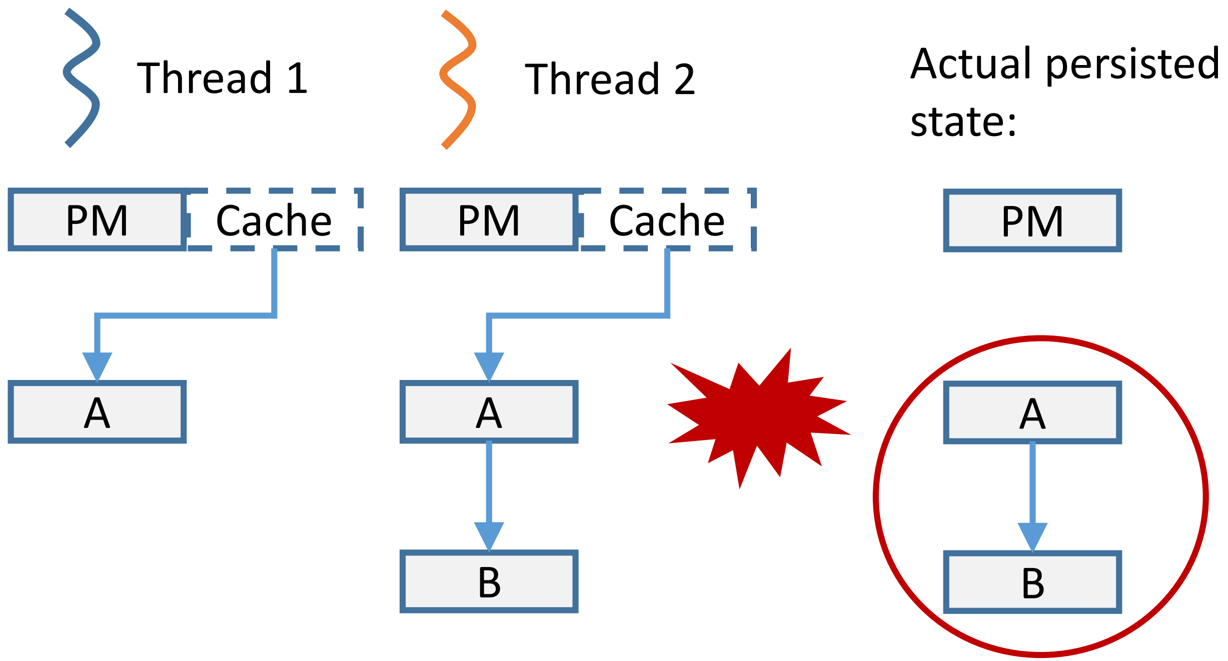
- Fast performance, high CPU utilization
- Instant recovery
- Fewer layers: simplified persistence model/architecture



Sounds great, but not automatic

Lock-free programming: even **harder in PM**

- Inherits all the existing challenges in DRAM
  - Race conditions
  - Memory reclamation issues



Unreachable

- New challenges
  - Volatile CPU caches **(new)**
  - Recovery **(new)**
  - Permanent memory leaks **(new)**

Difficult and error-prone to deal with using hardware instructions

Compare-and-swap (CAS)

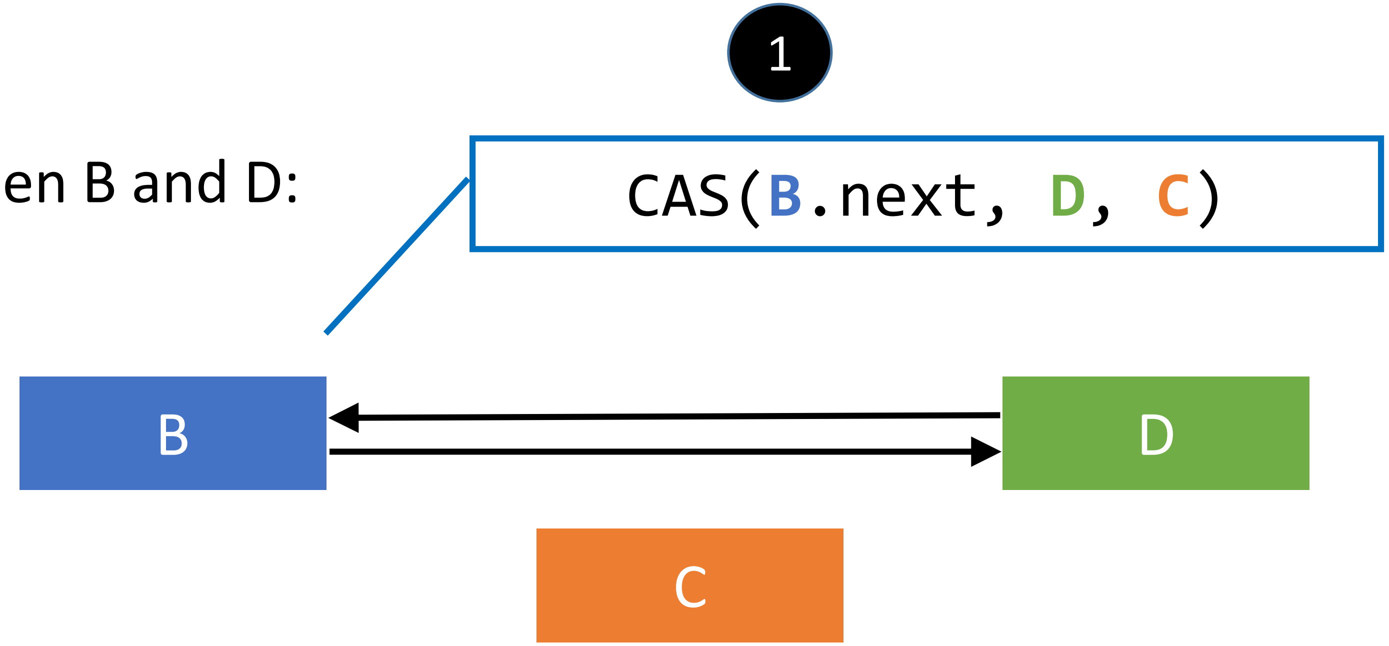
Conceptually:

```
CAS(*address, expected, desired)
   v = *address
   if v == expected then
      *address = desired
   return v
```

Powerful, but limited to single 8-byte words

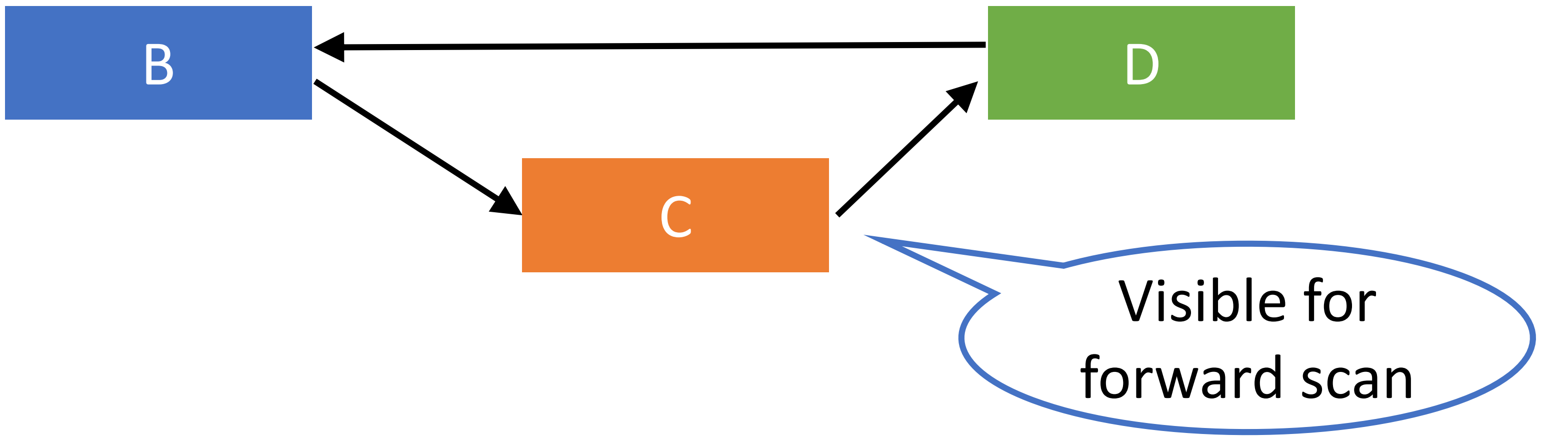
Example: doubly-linked list

Insert C between B and D:



Example: doubly-linked list

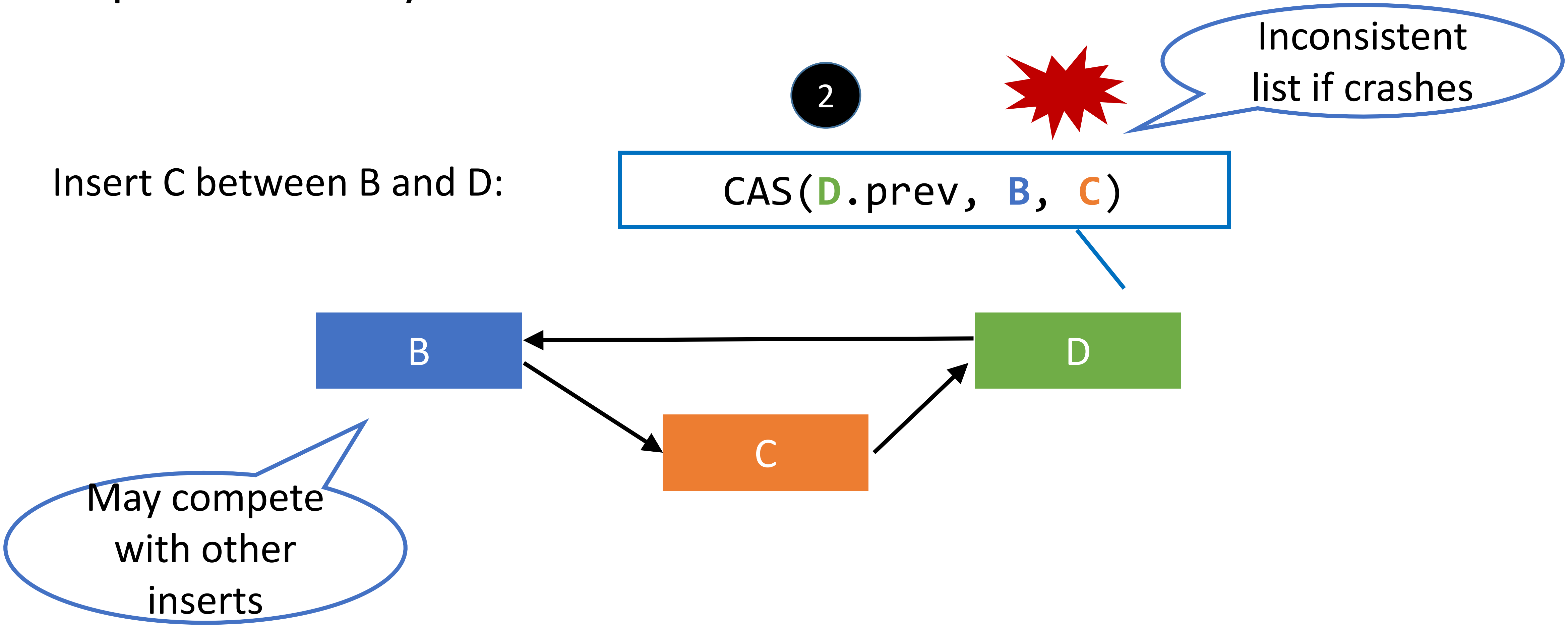
Insert C between B and D:



Intermediate state exposed to concurrent threads

Example: doubly-linked list

Insert C between B and D:



Many papers on devising lock-free doubly-linked lists

Persistent multi-word CAS (PMwCAS)*

- Atomically changing *multiple* 8-byte words *with persistence guarantee*
  - Either all specified updates succeed, or none of them
- Software-only
- Lock-free

- Based on a volatile MwCAS design [Harris+Fraser+Pratt 2002]
  - We made it work on persistent memory
  - With new necessary features on
    - Guaranteeing persistence
    - Recovery
    - Persistent memory management

** Easy Lock-Free Indexing in Non-Volatile Memory, ICDE 2018*

The PMwCAS operation

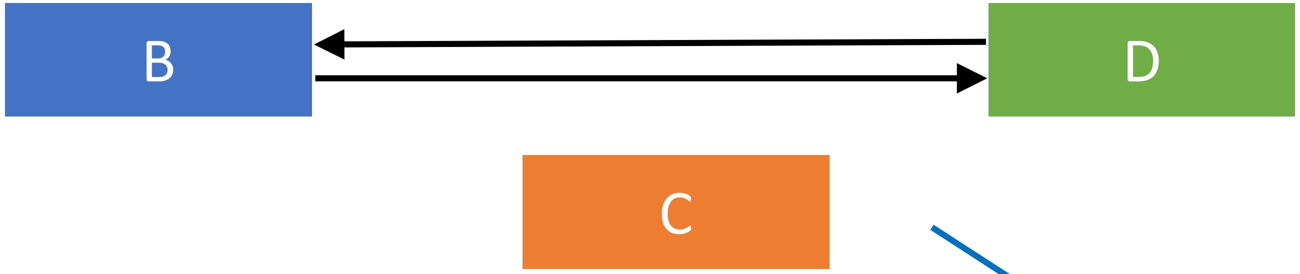
- Application specifies words to change atomically, in a descriptor
  - Following CAS interface for each word
  - Issue (launch) the operation after adding all words
  - Final result: either all words changed, or none of them

PMwCAS descriptor

Address 1	Expected 1	Desired 1
Address 2	Expected 2	Desired 2
Address 3	Expected 3	Desired 3
. . .		Status

Doubly-linked list with PMwCAS

Insert C between B and D:



PMwCAS descriptor

&B.next	D	C
&D.prev	B	C

`PMwCAS(desc)`

One step, C becomes atomically visible in both directions

So how does it work exactly?

- PMwCAS algorithm
- Guaranteeing persistence
  - Flush-upon-read – no logging needed
  - Recovery
- Memory Management
  - Preventing persistent memory leaks
  - Integration with persistent memory allocator
  - Epoch-based memory reclamation

So how does it work exactly?

- PMwCAS algorithm
- Guaranteeing persistence
  - Flush-upon-read – no logging needed
  - Recovery
- Memory Management
  - Preventing persistent memory leaks
  - Integration with persistent memory allocator
  - Epoch-based memory reclamation

See paper for more details

PMwCAS algorithm

1. Persist entire descriptor

Phase 1
Install a pointer to descriptor on each word (using CAS)
Change to 'failed' status if any CAS failed
Otherwise change to 'succeed' status.

Conflicting threads will "help" each other

2. Persist all modified words

Phase 2
If Phase 1 succeeded, install new values
Otherwise roll back

3. Persist all modified words + set status to 'finished' + flush status

Recovery

- Fixed-size descriptor pool
  - Doesn’t need to be large, 1000s-10k is good
- Recovery = scan descriptor pool
  - Roll forward ‘succeeded’ PMwCAS operations
  - Roll back failed ones

- Application-transparent recovery
  - Application transforms data structure from one consistent state to another
  - No application-specific code for recovery needed!
  - Volatile and persistent versions use the same code (turn persistence on/off)

Case studies and adoptions

- Two non-trivial data structures, focusing on database index structures
- Bw-Tree
  - Lock-free B+-tree in Microsoft SQL Server Hekaton
  - See details in paper
- Doubly-linked skip list

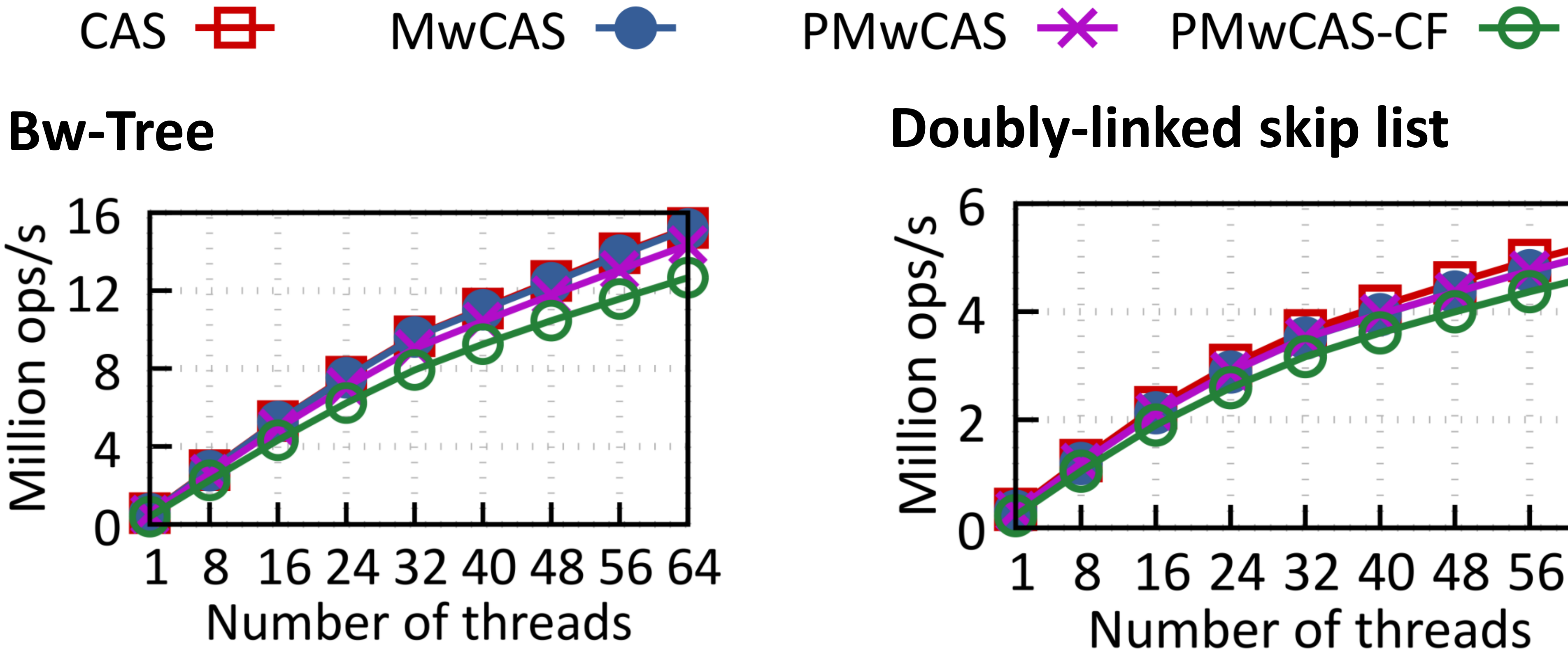
- Bz-Tree [Arulraj et al. VLDB 2018]
  - A new B+-tree for persistent memory
  - By Microsoft Research
- Other institutions using PMwCAS now for their own research

Evaluation

- Quad-socket, 8-core Xeon E5-4620 clocked at 2.2GHz
  - 32 physical cores, 64 hyperthreads in total
  - 256KB/2MB/16MBL1/L2/L3 caches
- Persistent memory emulation
  - 512GB DRAM – assuming NVDIMM-N
  - CLFLUSH (SFENCE + CLFLUSHOPT)
    - Upper bound overhead
  - SFENCE + CLWB emulation with injected delays
    - Calibrated using non-temporal writes
- Synthetic workloads
  - Insert/delete/search/scan on index structures (Bw-tree and doubly-linked skip list)
  - 20% write + 80% read (80% search + 20% range scan)

PMwCAS: easy implementation + fast

- Code almost as mechanical as lock-based (check out repo)
- < 10% overhead under realistic workloads (80% read + 20% write)



Summary

- Lock-free programming is already very hard in volatile memory
- Even harder in persistent memory
  - Performance
  - Persistence and recovery
  - Race conditions
- PMwCAS: primitive for easy lock-free programming in persistent memory
  - Code almost as simple as lock based – everything covered by PMwCAS
  - Transparent recovery – no application-specific code needed

  ➔ Use the same code for both persistent and volatile versions

Now open source at:

https://github.com/Microsoft/pmwcas

Thank you!