Recoverable Software Combining

PANAGIOTA FATOUROU	FORTH ICS &UNIVERSITY OF CRETE
NIKOLAOS D. KALLIMANIS	FORTH ICS
ELEFTHERIOS KOSMAS	UNIVERSITY OF CRETE

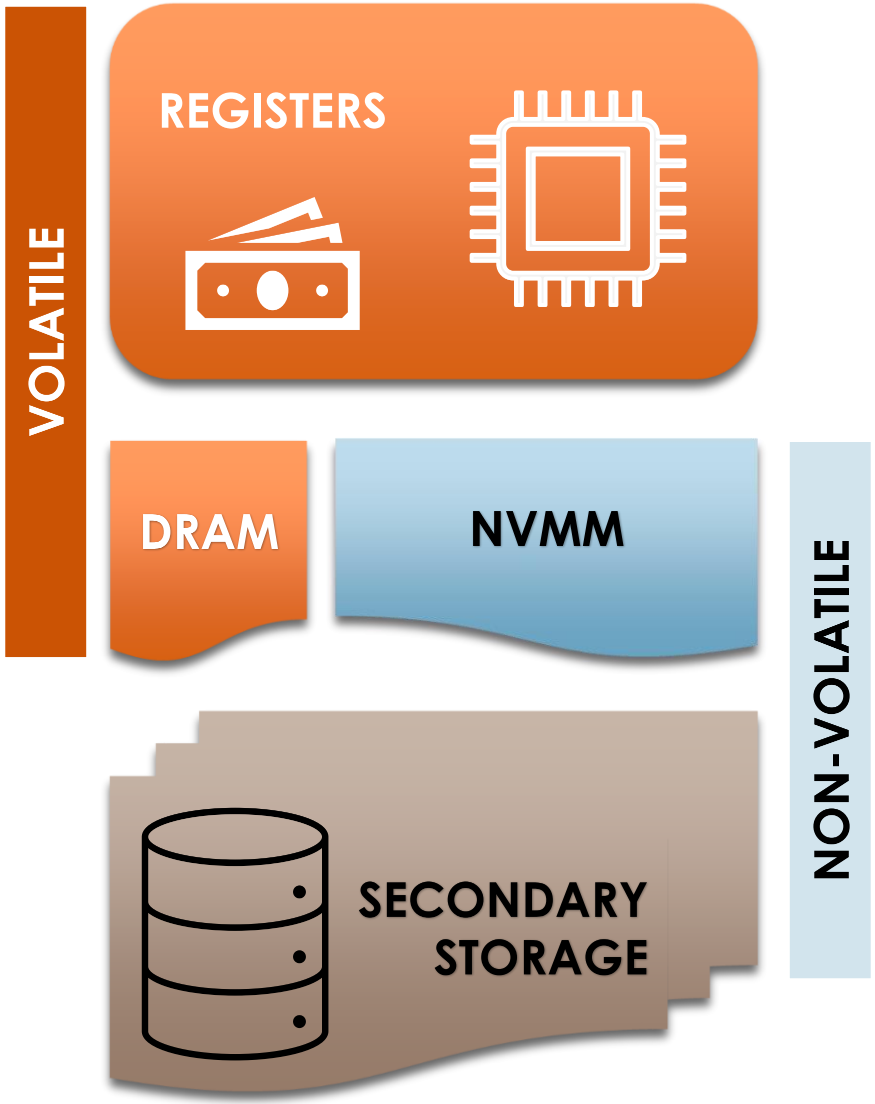
NVMW 2022

Recoverable Computing

Challenge

- **Non-Volatile Main Memory (NVMM)**
  - **byte-addressable**
  - **large** and **inexpensive**
  - **fast** recovery
- **persistence** instructions
  - `pwb`, `pfence`, `psync`
  - **expensive**
- **inefficient** recoverable implementations of data structures

☞ **Goal: low** persistence overhead



Our Contribution

Our Algorithms		Faster than best competitor
Blocking	`Sync Prot.`	`3.9x`
	`Stack`	`3.9x`
	`Queue`	`2.3x`
Wait-Free	`Sync Prot.`	`2.4x`
	`Stack`	`2.3x`
	`Queue`	`1.6x`

Highly efficient **recoverable**
blocking and wait-free

- **synchronization protocols**
  - **outperform** by far (up to **3.9x**) many recently proposed recoverable UCs [RedoOpt]$_{\text{EuroSys'20}}$ and STMs [CX-PTM]$_{\text{EuroSys'20}}$, [OneFile]$_{\text{DSN'19}}$
- **stacks** and **queues**
  - **outperform** by far previous implementations (including specialized)
    - queues (up to **2.3x**): [OptLinkedQ, OptUnLinkedQ]$_{\text{(SPAA'21)}}$, [CX-PUC, CX-PTM, RedoOpt]$_{\text{EuroSys'20}}$, [OneFile]$_{\text{DSN'19}}$, [Capsules]$_{\text{SPPA'19}}$, [Friedman et al]$_{\text{PPoPP'18}}$, [Romulus]$_{\text{(SPAA'18)}}$
    - stacks (up to **3.9x**): DFC$_{\text{arXiv'20}}$, OneFile$_{\text{DSN'19}}$, RomulusLog$_{\text{SPAA'18}}$, PMDK

✍ often **guarantee stronger** consistency properties

Recoverable Objects

Correctness

Durable Linearizability

- **all completed operations** before the **crash**, are **reflected** in the object's state upon **recovery**

[Izraelevitz, Mendes and Scott. 2016]

  - operation **responses**?
  - **re-execute** operation upon recovery? → **not** always an **acceptable** option

Detectability

- **recovery code** infers if the **failed** operation was linearized or not
- if it is linearized, obtains its **response**

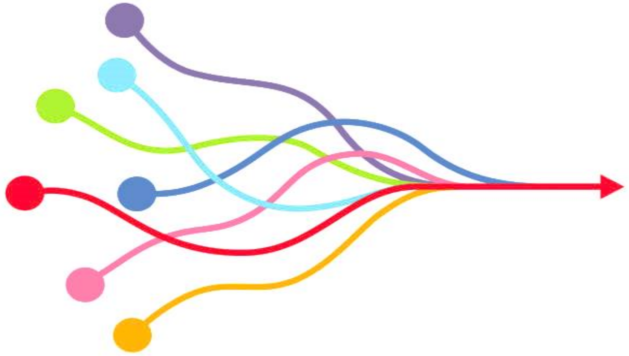
[Friedman, Herlihy, Marathe and Petrank. 2018]

Software Combining

Low synchronization cost

- state-of-the-art **synchronization** technique
- goal: execute synchronization **requests** at **low** cost
    - access the **same** data → must be executed in mutual exclusion
    - **ideally**,
        - ✓ **zero** synchronization cost
        - ✓ time required to execute them **sequentially**
- **announce** requests
- **combiner** serves active requests from **all** other threads
- **other** threads
    - (in a blocking setting) **local spin** until request is served
    - (otherwise) **pretend*** to be the combiner, e.g., using **local copy** of the state

      **(eventually, just one will indeed become the combiner)*



Recoverable Computing

Crucial for low persistence overhead

Persistence Principles

1. **low number** of persistence instructions
   - store in NVMM only those variables (and persist only those from their values) that are necessary for recoverability
2. **low-cost** persistence instructions
   - e.g., avoid persisting highly-contented variables
3. persist **consecutive** data
   - ✍ pwbs are applied on **cache-line** granularity

Design Decisions of Combining Protocols

A. mechanism for **choosing combiner**

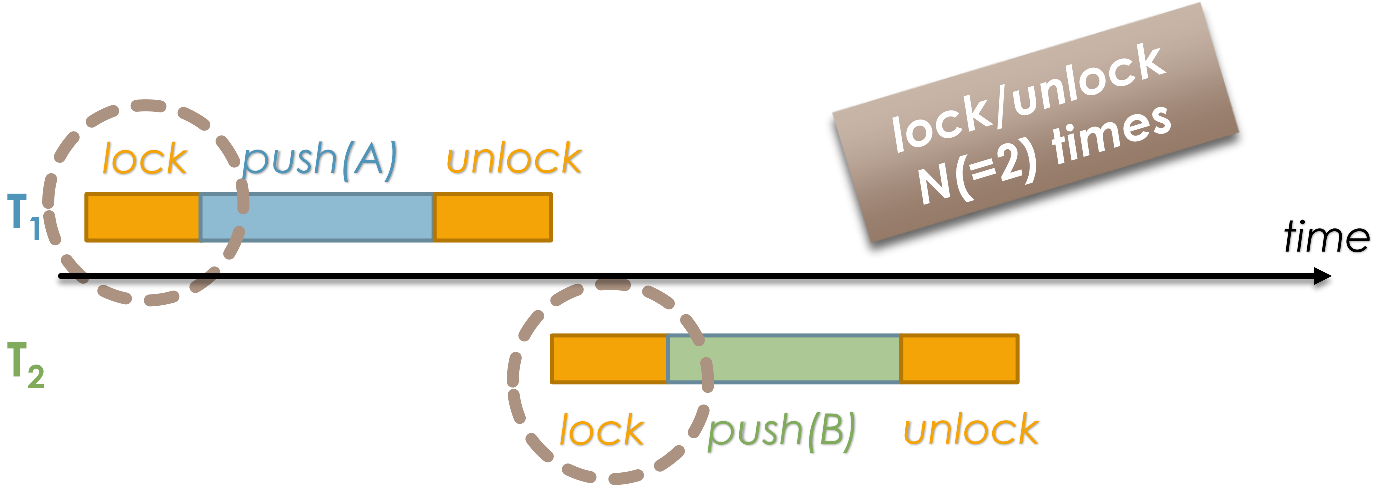
B. data structure to **store** the **active** requests

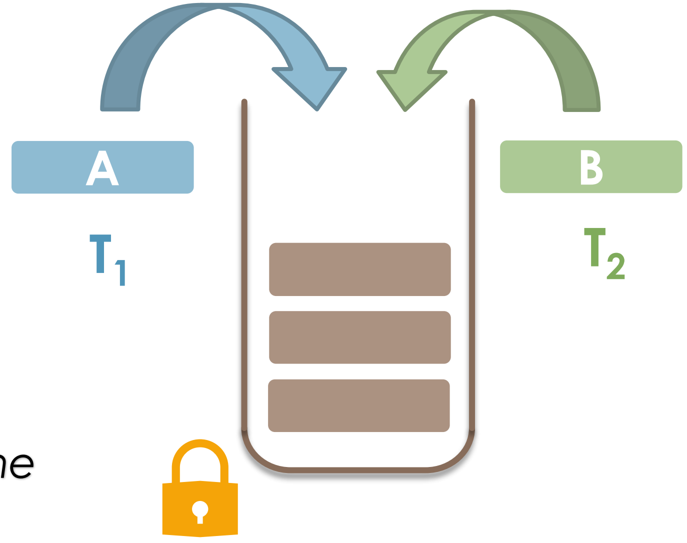
C. mechanism to **apply** the updates

D. mechanism for **collecting** responses

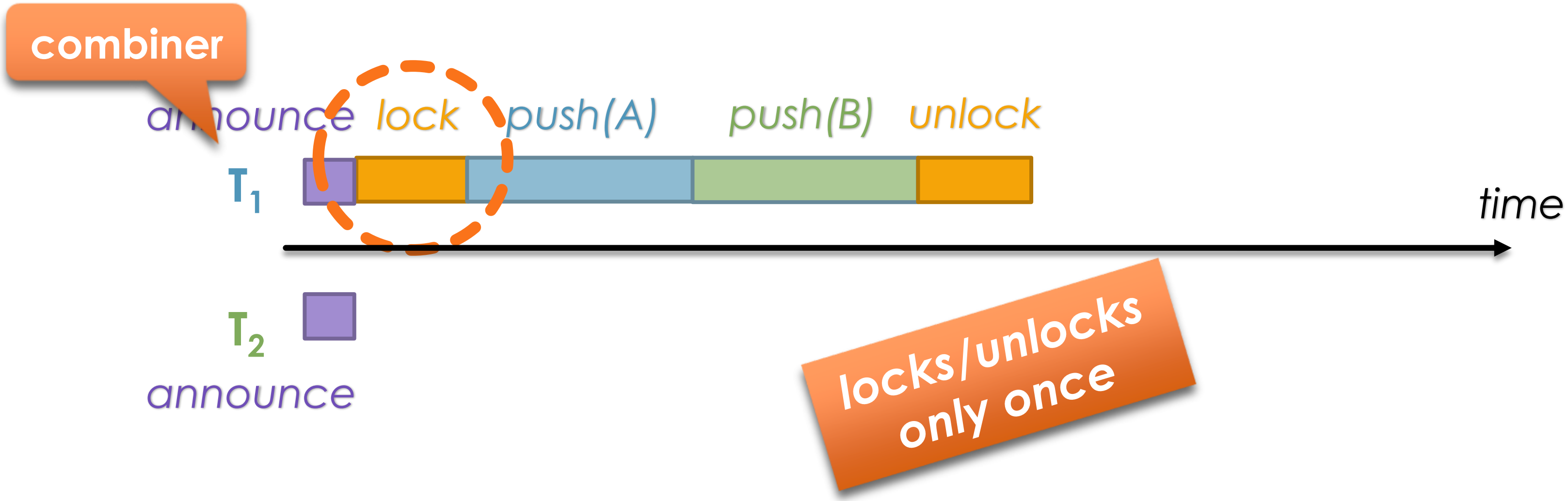
E. mechanism to **discover** (not) applied requests

Software Combining



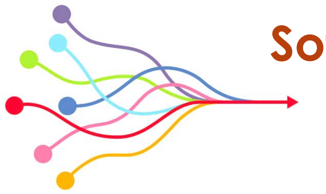


conventional *lock-based implementation*



Announce Array

T_1	push(A)
T_2	push(B)

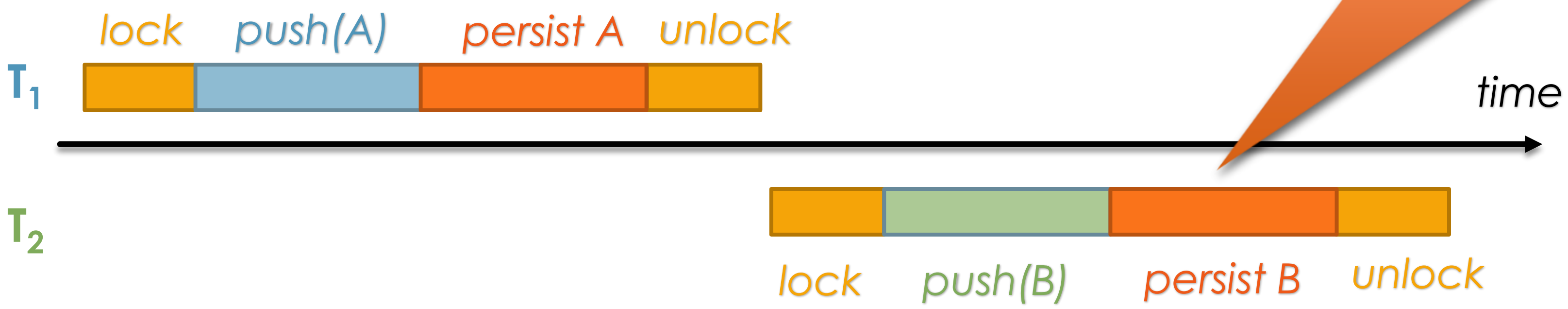


Software Combining
technique

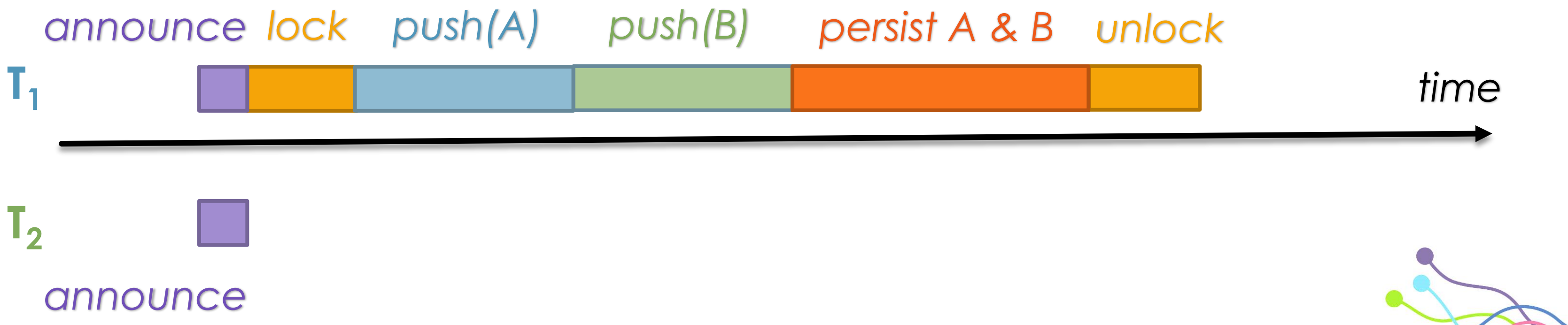
Key Idea

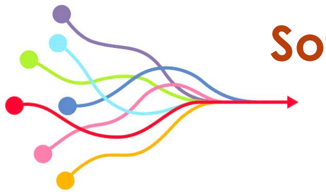
Why is this a promising approach?

persist = usually two instructions (pwb & psync)



conventional **recoverable** *lock-based implementation*

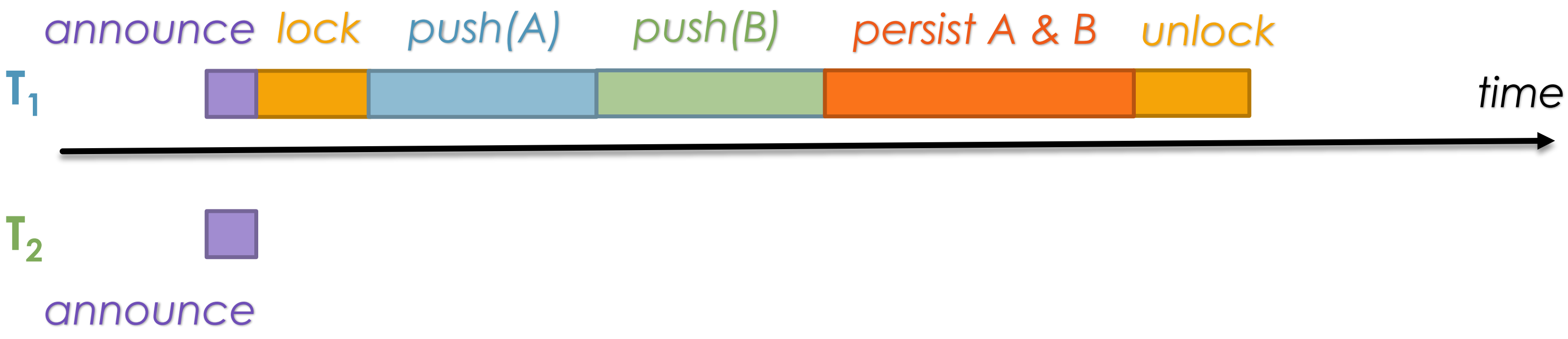


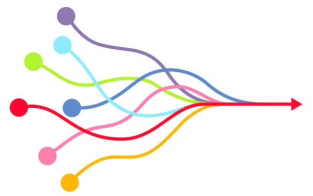


Software Combining *technique*

Key Idea

Why is this a promising approach?





Software Combining →
Efficient Recoverable
Data Structures

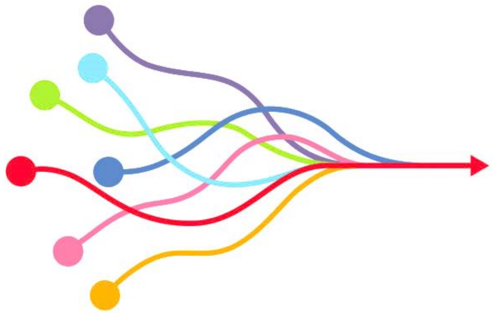
Benefits:

- ✓ **reduced** number of **fence** instructions
  - combiner executes only **one fence**
- ✓ store **multiple** nodes into a single cache line
- ✓ allocate/persist **consecutive** memory addresses
- ✓ **elimination** is applicable

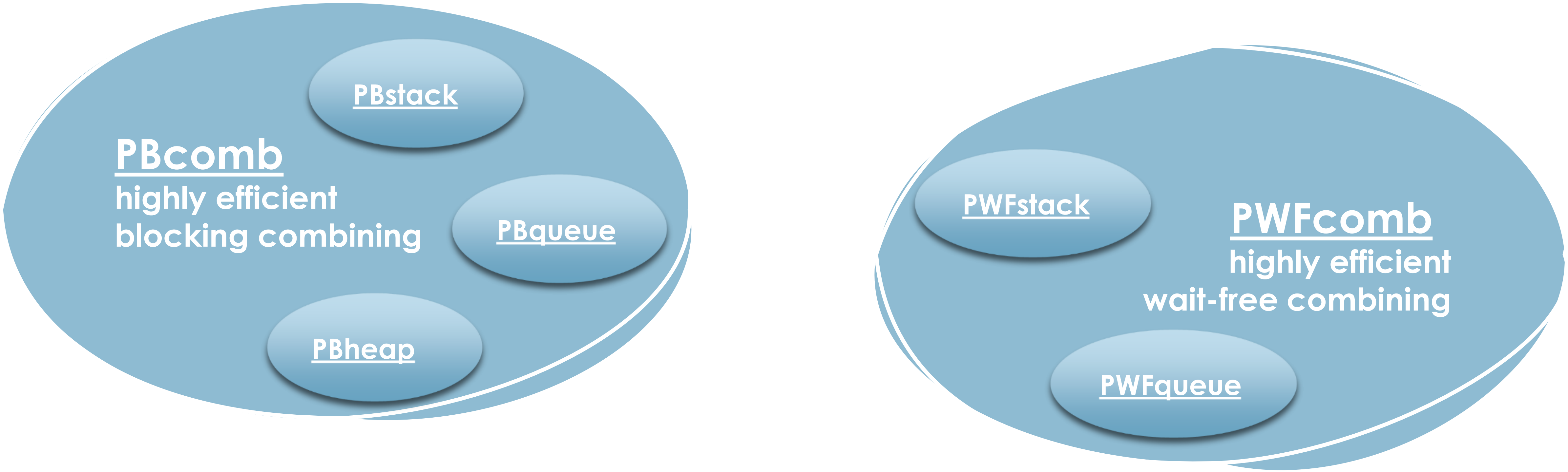
- ✓ **efficient** solution for **highly contended** data structures
  - e.g., stacks and queues

fundamental data structures

Our results



reveal the power of **Software Combining** →
low-cost recoverability



PBcomb
highly efficient
blocking combining

PBstack

PBqueue

PBheap

PWFcomb
highly efficient
wait-free combining

PWFstack

PWFqueue

PBcomb

Design decisions

- A. **Announce** array → **DRAM**
- B. **lock** → **DRAM**
  - a thread that fails to acquire the **lock**, waits **at most two** combiners

announced but not applied

↗ **DRAM** ↗ **NVMM**

- C. **active** request: `activate` bit ≠ `deactivate` bit
  - `activate` **flipped** upon request announce
  - `deactivate` **flipped** after serving request

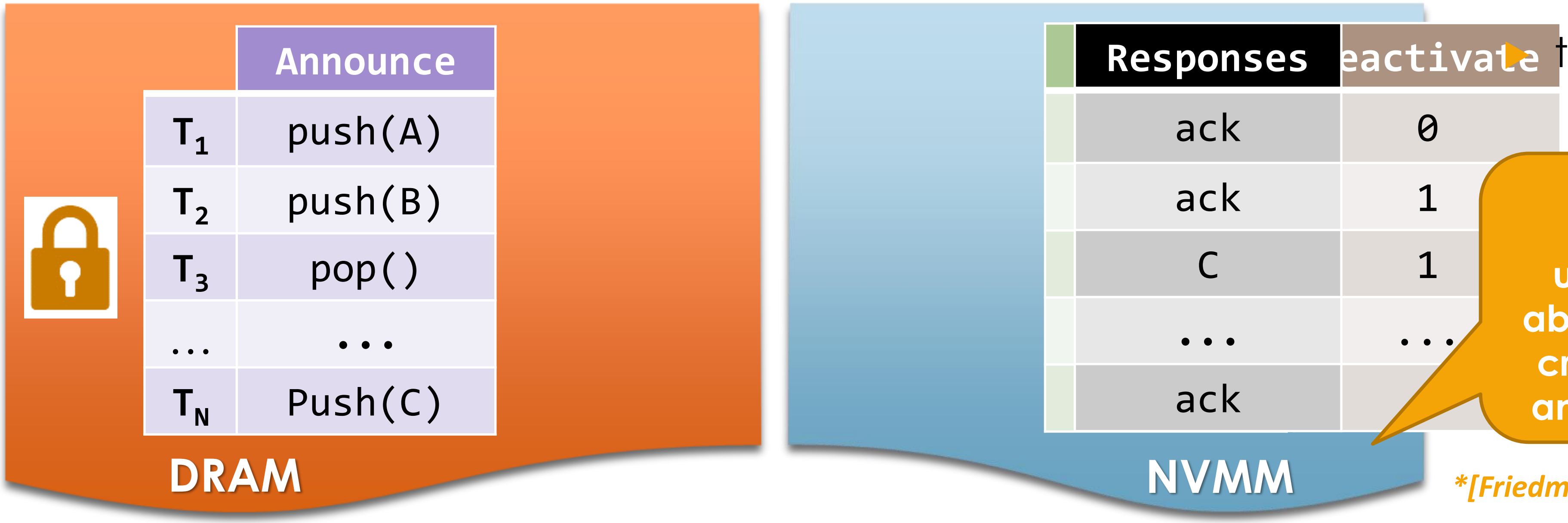
- D. **Responses** → **NVMM**
  - combiner **stores** responses of served requests
  - threads **retrieve** them

    request of **T_1** is active

satisfies **detectability***

upon recovery a thread is able to determine whether its crashed request took effect and if so, obtain its response

[Friedman, Herlihy, Marathe, and Petrank. 2018]



	Announce
T_1	push(A)
T_2	push(B)
T_3	pop()
…	…
T_N	Push(C)

DRAM

	Responses	**eactivate**
	ack	0
	ack	1
	C	1
	…	…
	ack	

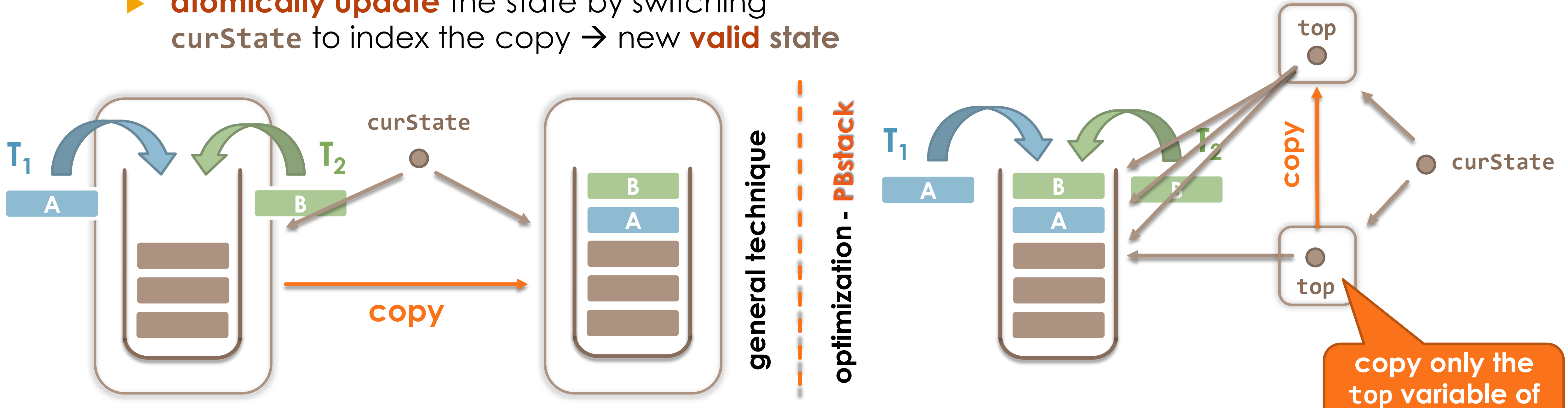
NVMM

PBcomb

Design decisions – Apply Requests

- **copy** the **state** of the data structure
- **apply** requests on this copy
- **atomically update** the state by switching `curState` to index the copy → new **valid state**

- **optimization**: copy only the **state** of the **synchronization points** of the data structure



copy only the top variable of the stack

- the copy of the state is persisted before updating cu
- the updated value of curState is persisted before rel

PBstack: persists only top and the newly allocated nodes

PBcomb

Design decisions – Copy of the state

Benefits of copying:

- ✓ enables allocation and **persistence** of **consecutive** memory locations
  - ▶ private copy
  - ✍ **enhancement**: stores together with the state **all** other **persistent metadata** of PBcomb
    - ▶ **responses** and **`deactivate`** bits
- ✓ allows atomic update of the simulated state with a **single** instruction
  - ▶ **crash-resistant**: retains the data structure in **consistent** state
- ✓ **fast** recovery
  - ▶ already supports durable linearizability → **null-recovery**
  - ▶ to support detectability → a **single** check to determine if a request has been served and retrieve its response

durable linearizability*

the effects of all requests that have completed before a crash, are reflected in the state of the data structure, upon recovery

[Izraelevitz, Mendes, and Scott. 2016]

Additional results*

Key points

Blocking Recoverable Software Combining

- **PBqueue**
  - uses **two instances** of **`PBcomb`**
    - the **first** coordinates accesses on **head**
    - the **second** coordinates accesses on **`tail`**
  - copies only the state of the synchronization points (**head** and **`tail`**)of the queue
- **PBheap**
  - **state**: heap elements and heap bounds

****Full Version*:**
https://doi.org/10.1145/3503221.3508426
https://arxiv.org/abs/2107.03492

Wait-free Recoverable Software Combining

- **PWFcomb**
  - extends ideas from PBcomb and Psim**
  - **several** threads may concurrently attempt to become the combiner → **increased** persistence overhead
  - **additional** techniques used to **reduce** persistence overhead
- **PWFstack**: copies only **`top`**
- **PWFqueue**
  - uses **two instances** of **`PWFcomb`**
  - copies only **head** or **`tail`**

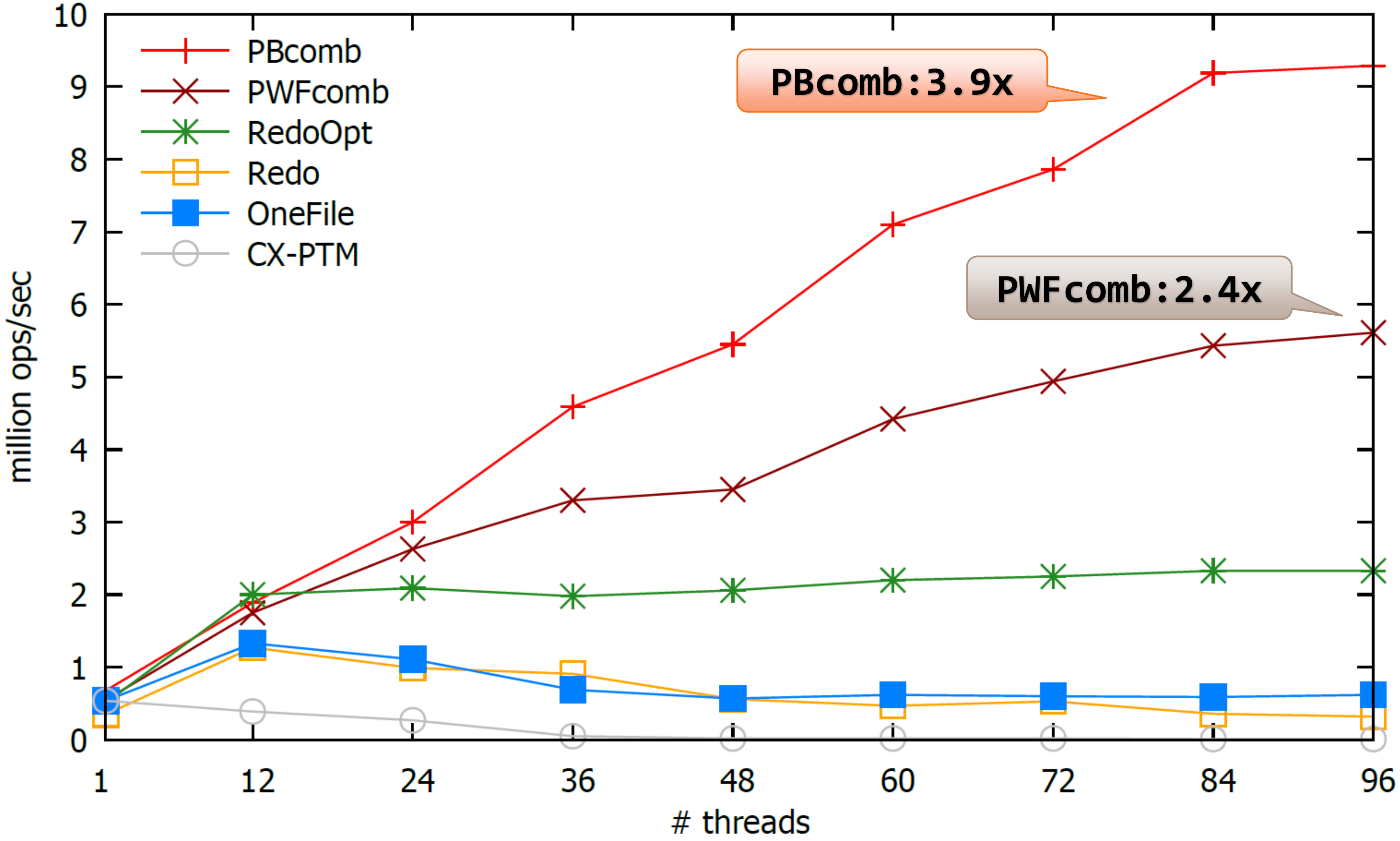
*****[Fatourou and Kallimanis. 2011]***

Performance Analysis

Testbed and Synthetic-Benchmark

`2-processor Intel Xeon Platinum 8260M (96 logical cores) with 1TB Intel Optane DC persistent memory (DCPMM) in AppDirect mode`

Recoverable `Fetch&Multiply`



PBcomb:3.9x

PWFcomb:2.4x

- a thread adds a randomly produced **workload** between **consecutive** `Fetch&Multiply` ops

our protocols satisfy <u>detectability</u>

competitors guarantee only <u>weaker</u> consistency (e.g. durable linearizability)

Performance Analysis

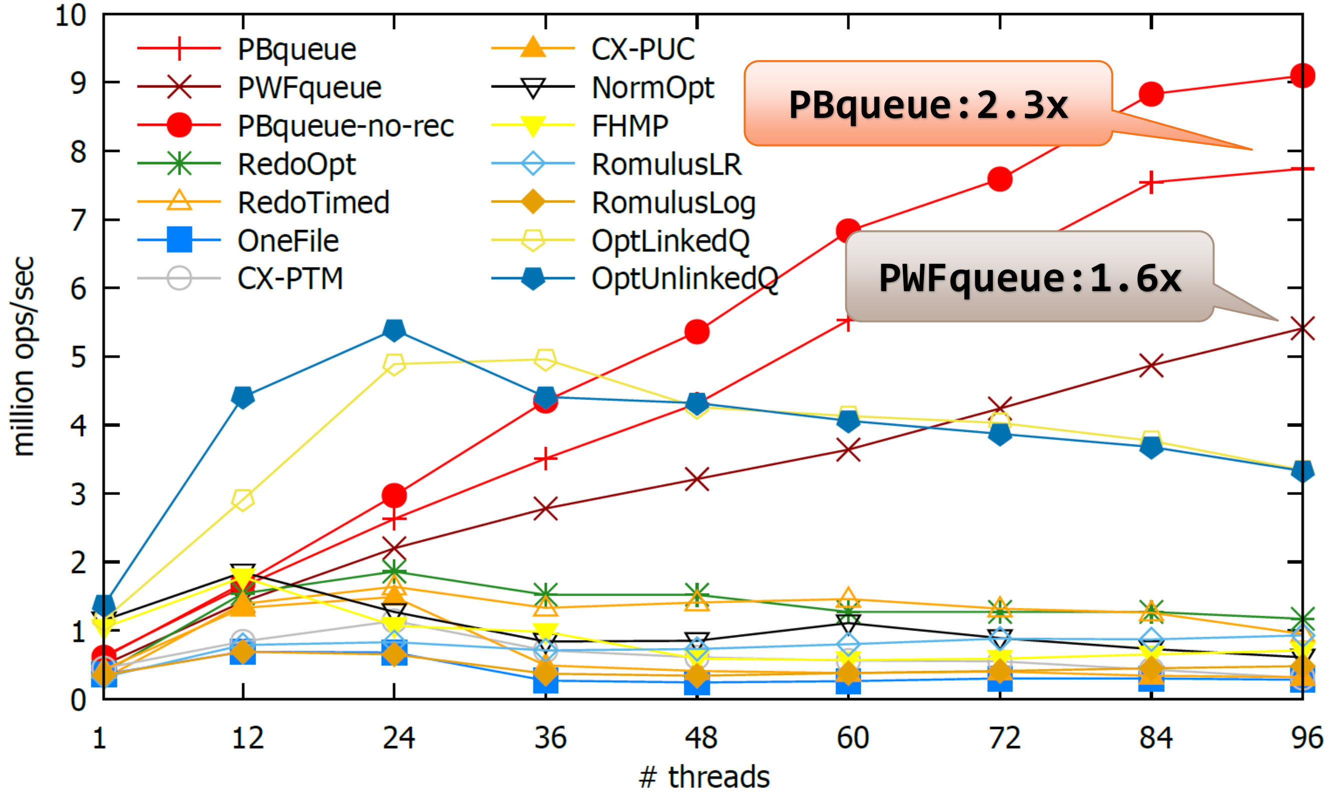
Fundamental Data Structures

- benchmarks perform **pairs** of `enqueues-dequeues & push-pops`

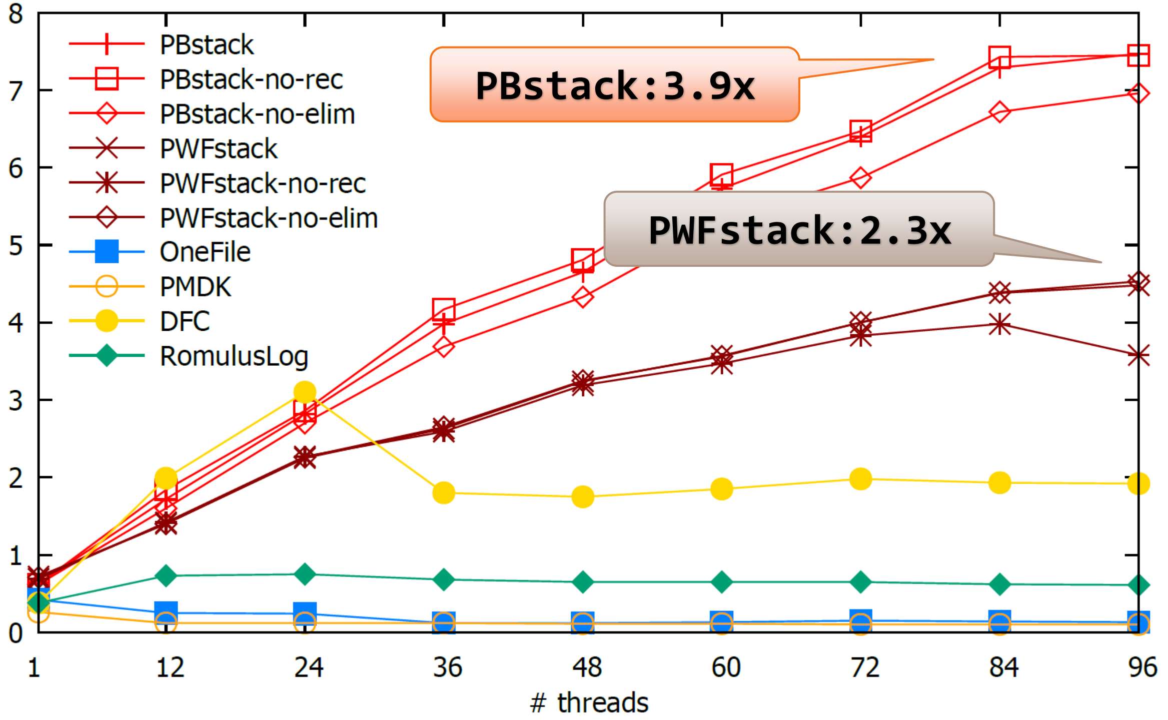
Why our implementations perform so well?

low synchronization & persistence cost

Recoverable **Queue**

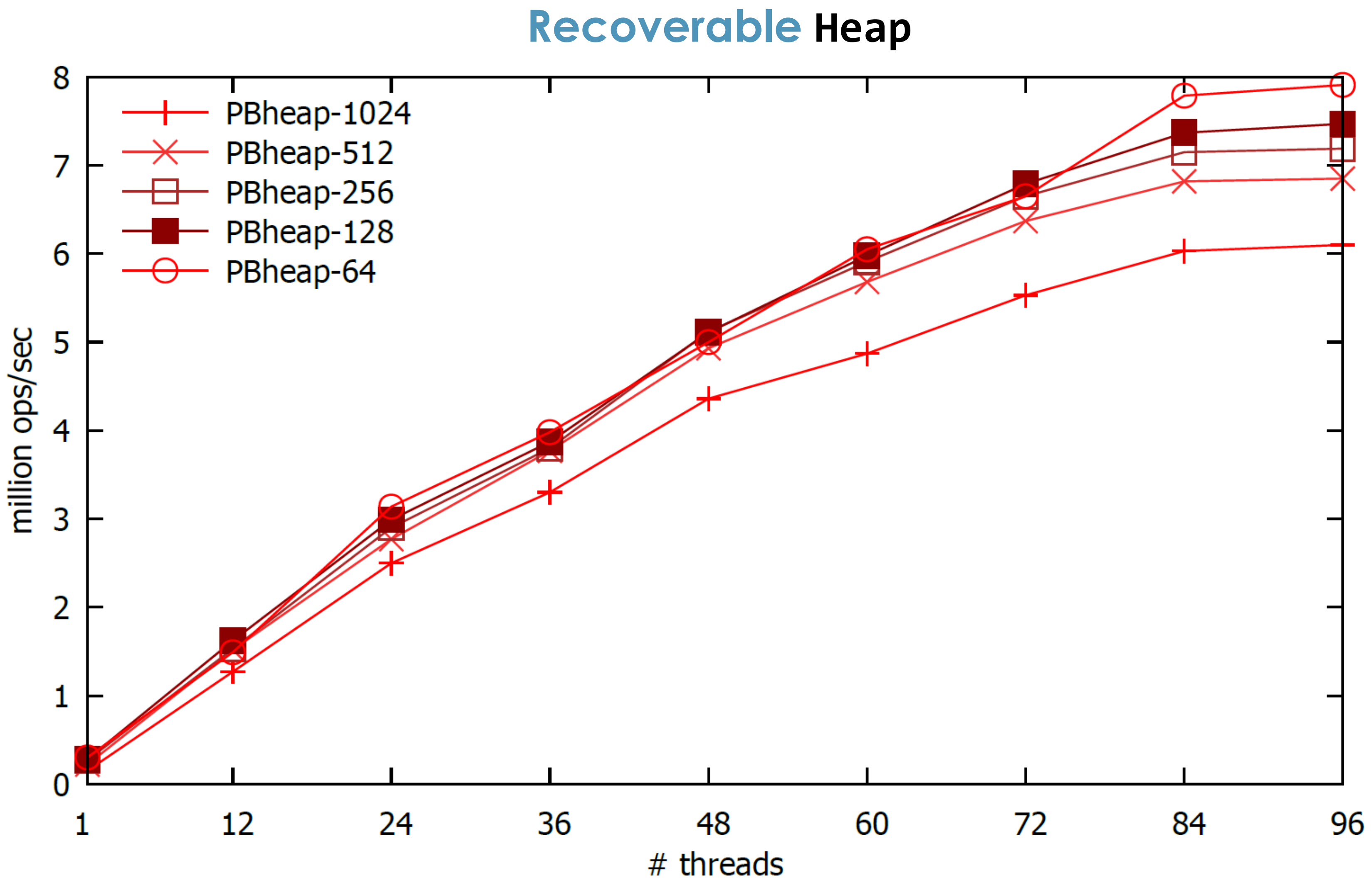


Recoverable **Stack**



Performance Analysis

More Complex Data Structures – Heap

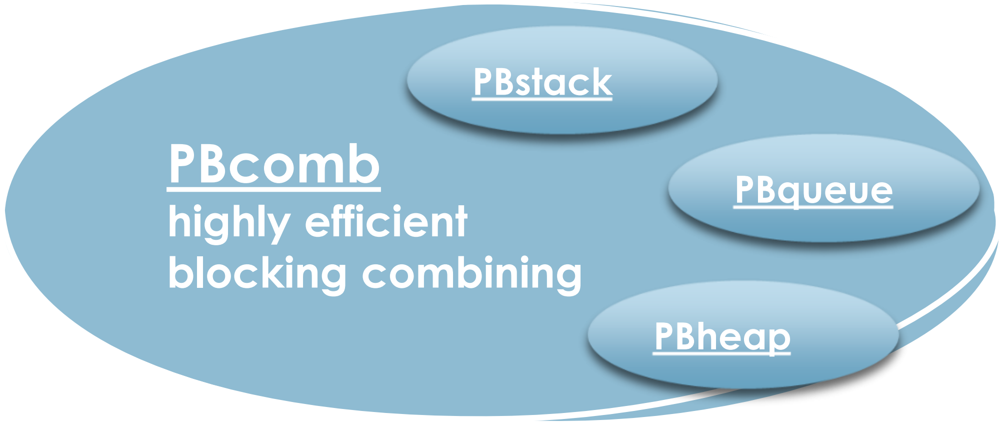


- the **first** recoverable heap implementation
- benchmark performs **equal** number of **`Insert`** and **`DeleteMin`** operations

Conclusion

reveal the power of
Software Combining →
low-cost recoverability

- **persistence principles**
  - follow to achieve **good** performance
- **many** times faster than competitors
- we are **detectably** recoverable
  - most competitors are **only** durably linearizable

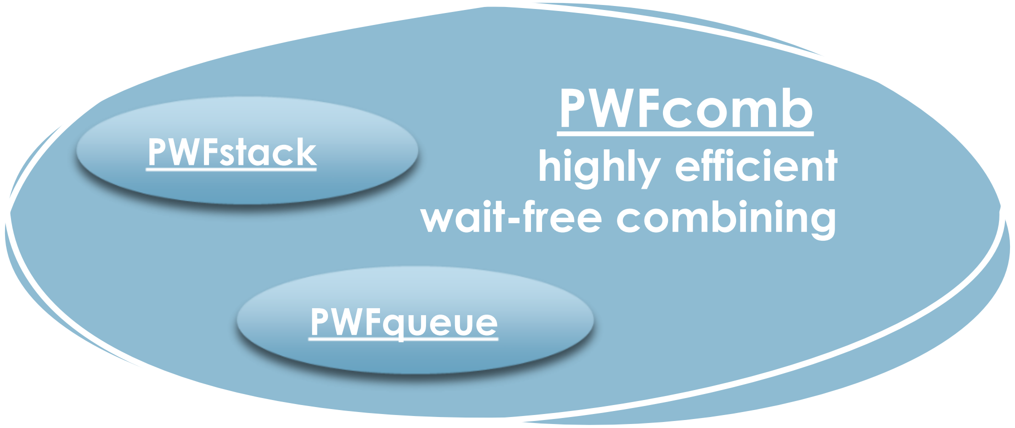


PBcomb
highly efficient
blocking combining

PBstack

PBqueue

PBheap



PWFcomb
highly efficient
wait-free combining

PWFstack

PWFqueue