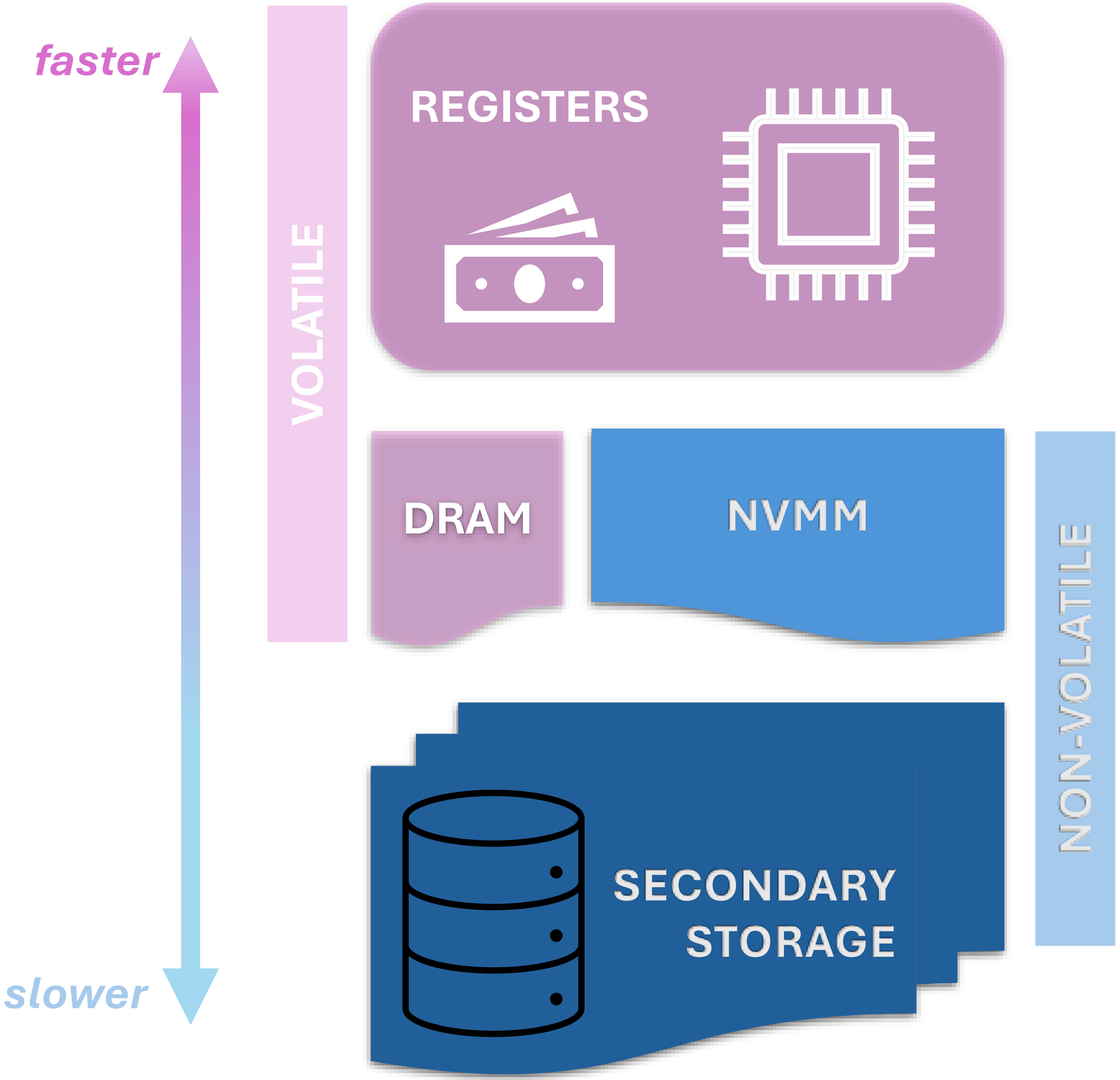
EMERALD
1st Workshop on Distributed Computing with Emerging Hardware Technology

The Power of Software Combining in Persistence

Eleni Kanellou

ICS-FORTH

NVMM → Recoverable Computing



Recoverable Computing Challenges

Non-Volatile Main Memory

- **byte-addressable**
- **large** and **inexpensive**
- **fast** recovery

Persistence Instructions

- pwb, pfence, psync
- **expensive**

Problem: Inefficient recoverable implementations of data structures

Goal: **low** persistence overhead

In this talk

Presented 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

Correctness for Recoverable Objects

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?

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]

Low Synchronization Cost through Software Combining

- 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)*

Principles 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

… applied to Combining Protocols?

A. mechanism for **choosing combiner**

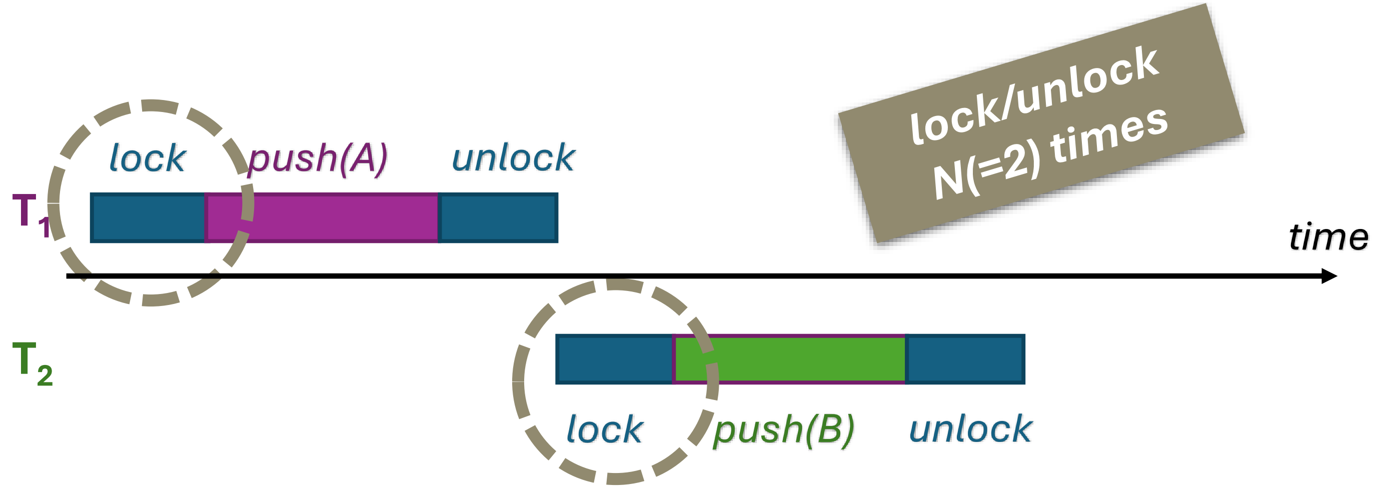
B. data structure to **store** the **active** requests

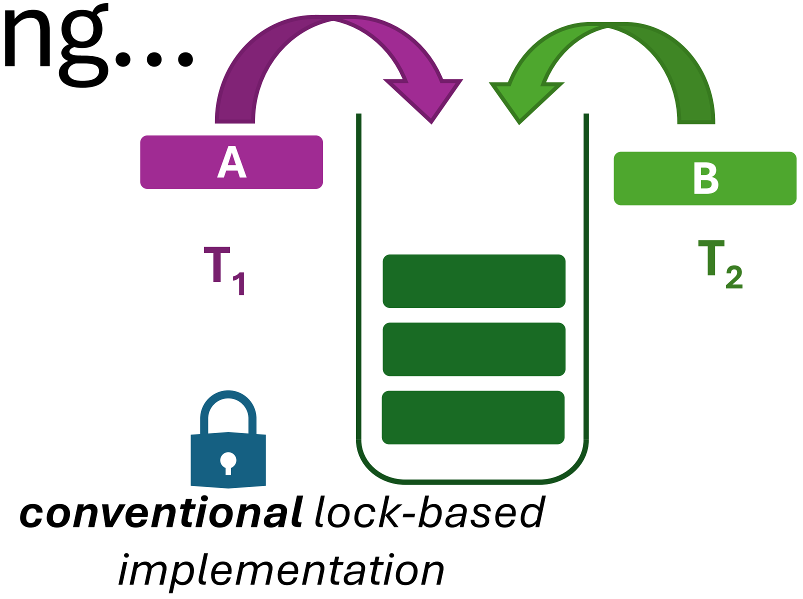
C. mechanism to **apply** the updates

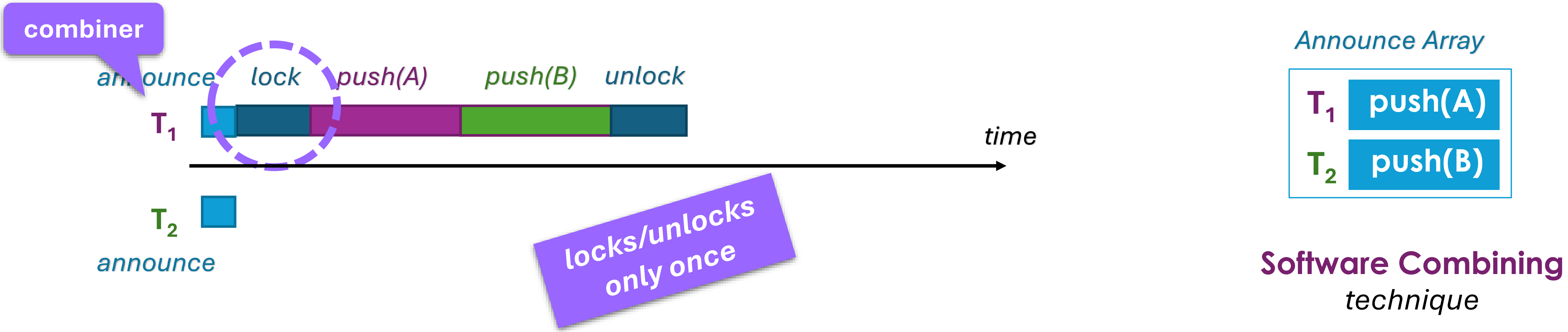
D. mechanism for **collecting** responses

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

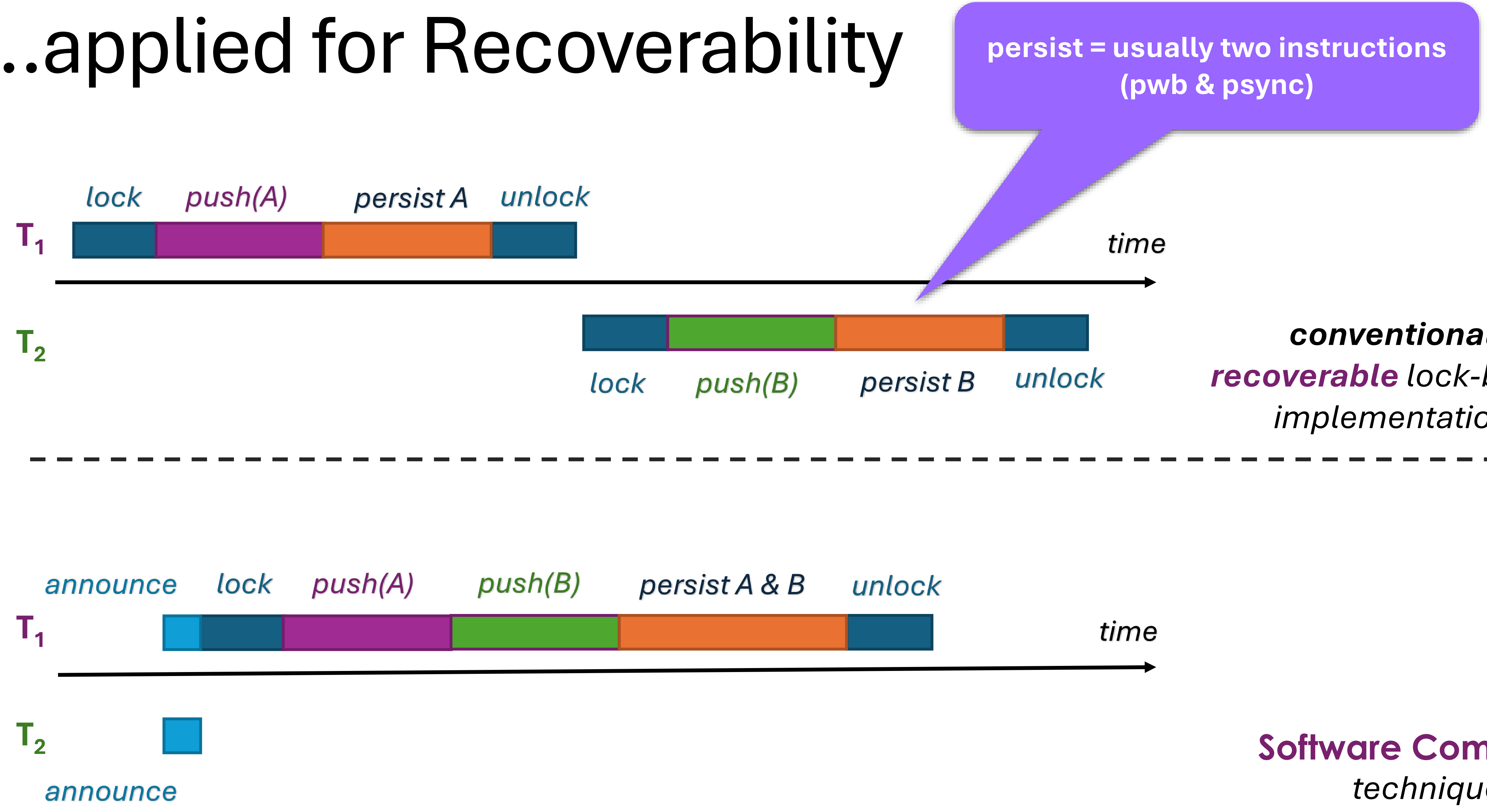
The Power of Software Combining...







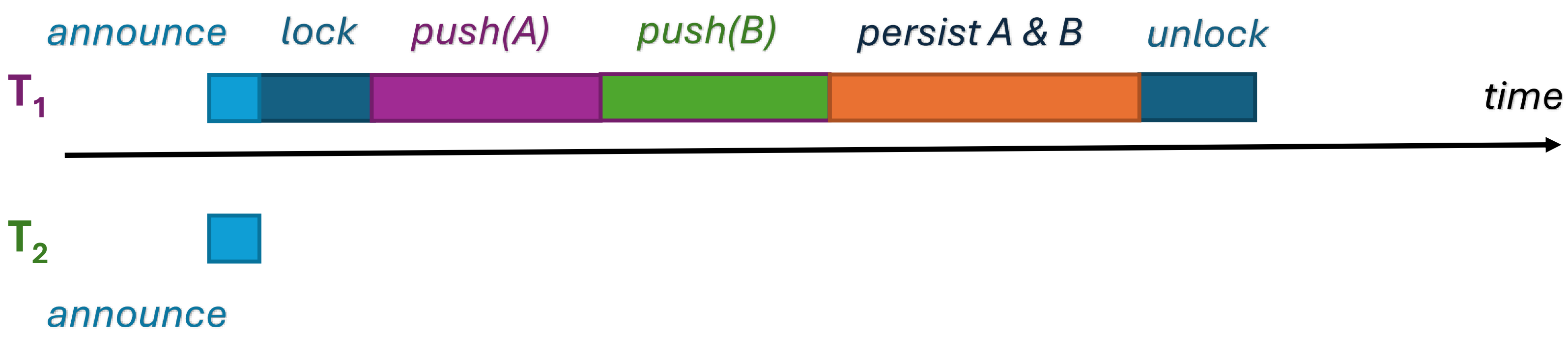
...applied for Recoverability



conventional
recoverable *lock-based implementation*

Software Combining
technique

Why is this a promising approach?



Software Combining →
Efficient Recoverable
Data Structures

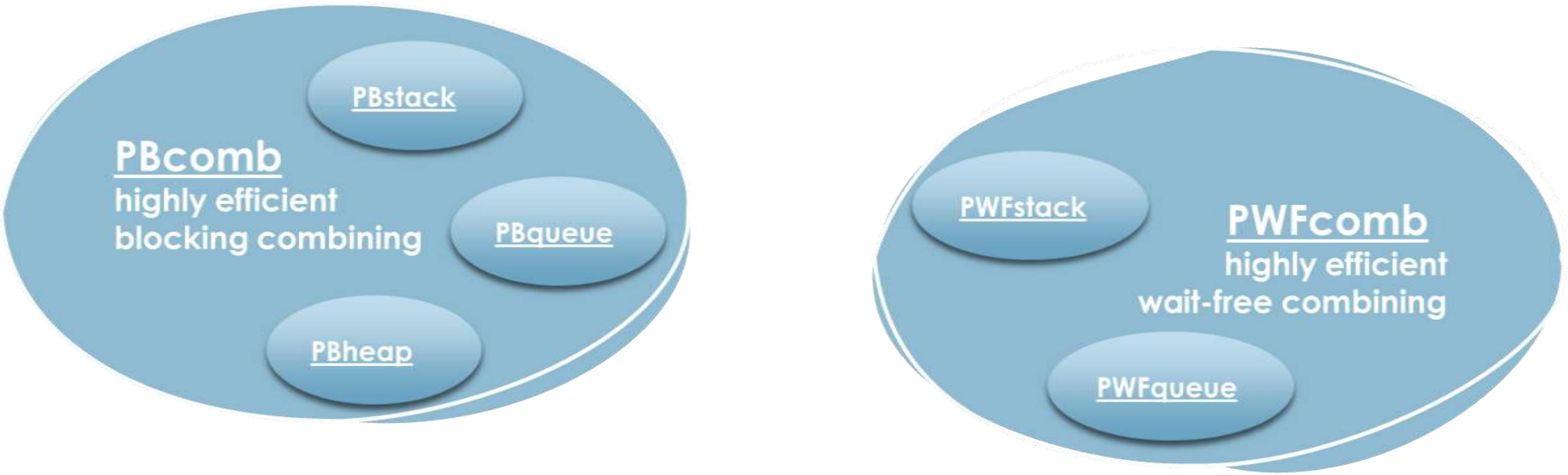
Benefits:

- ✓ **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

Software Combining for low-cost Recoverability



PBcomb: Overview

A. **Announce** array → DRAM

B. `lock` → DRAM

- a thread that fails to acquire the `lock`, waits **at most two** combiners

announced but not applied

C. active request : `activate` bit ≠ `deactivate` bit

↗ DRAM ↗ NVMM

- `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

	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

satisfies **detectability***

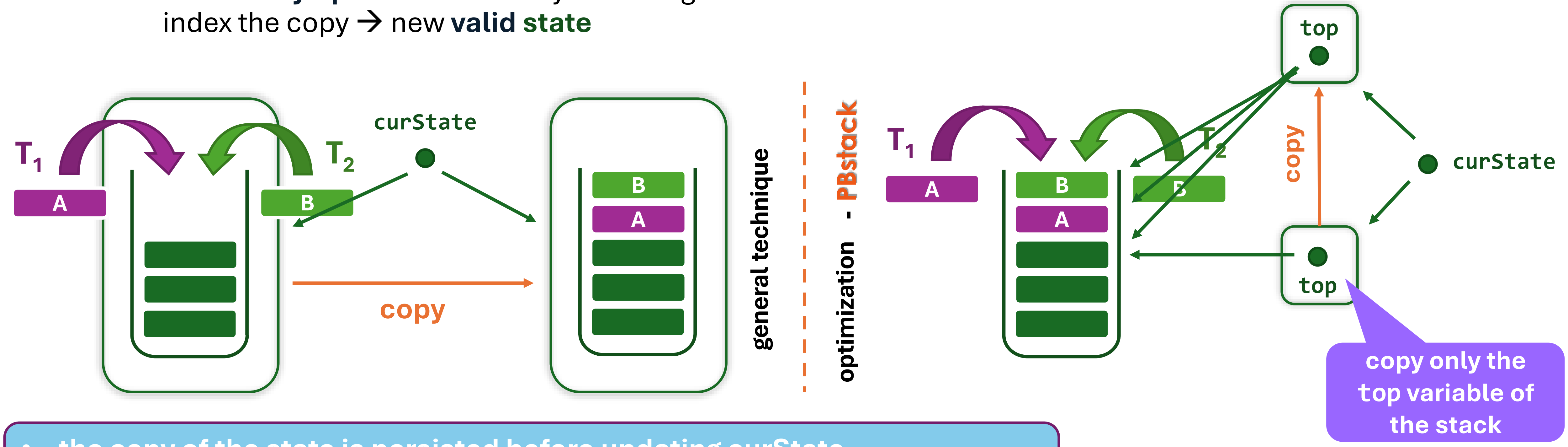
upon recovery a thread is able to determine whether its crashed request took effect and if so, obtain its response

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

PBcomb: How are requests applied?

- **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



- **the copy of the state is persisted before updating curState**
- **the updated value of curState is persisted before releasing the lock**

PBstack: persists only top and the newly allocated nodes

PBcomb: Copying 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

Extending these ideas

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://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**

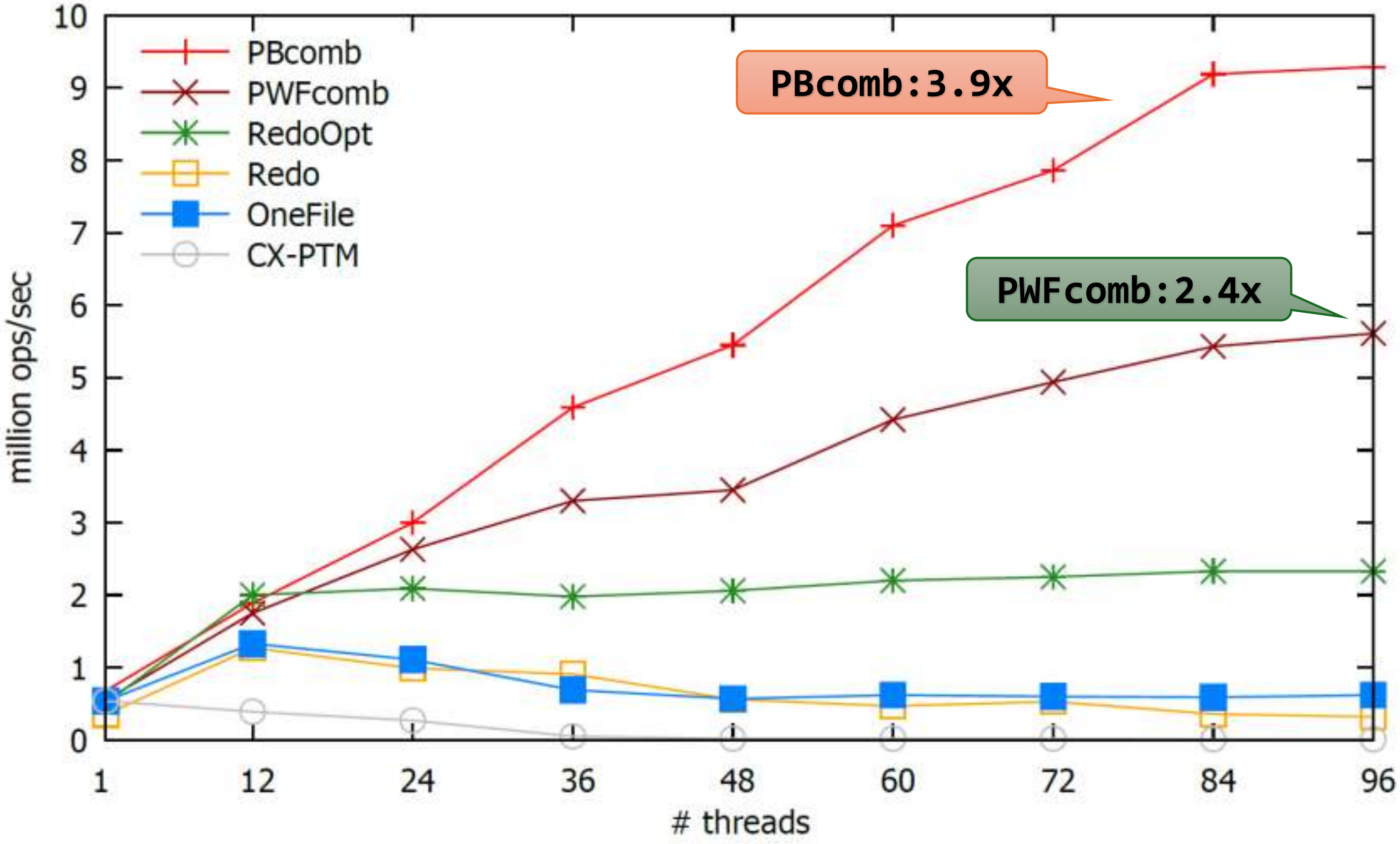
**[Fatourou and Kallimanis. 2011]

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



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

proposed protocols satisfy detectability

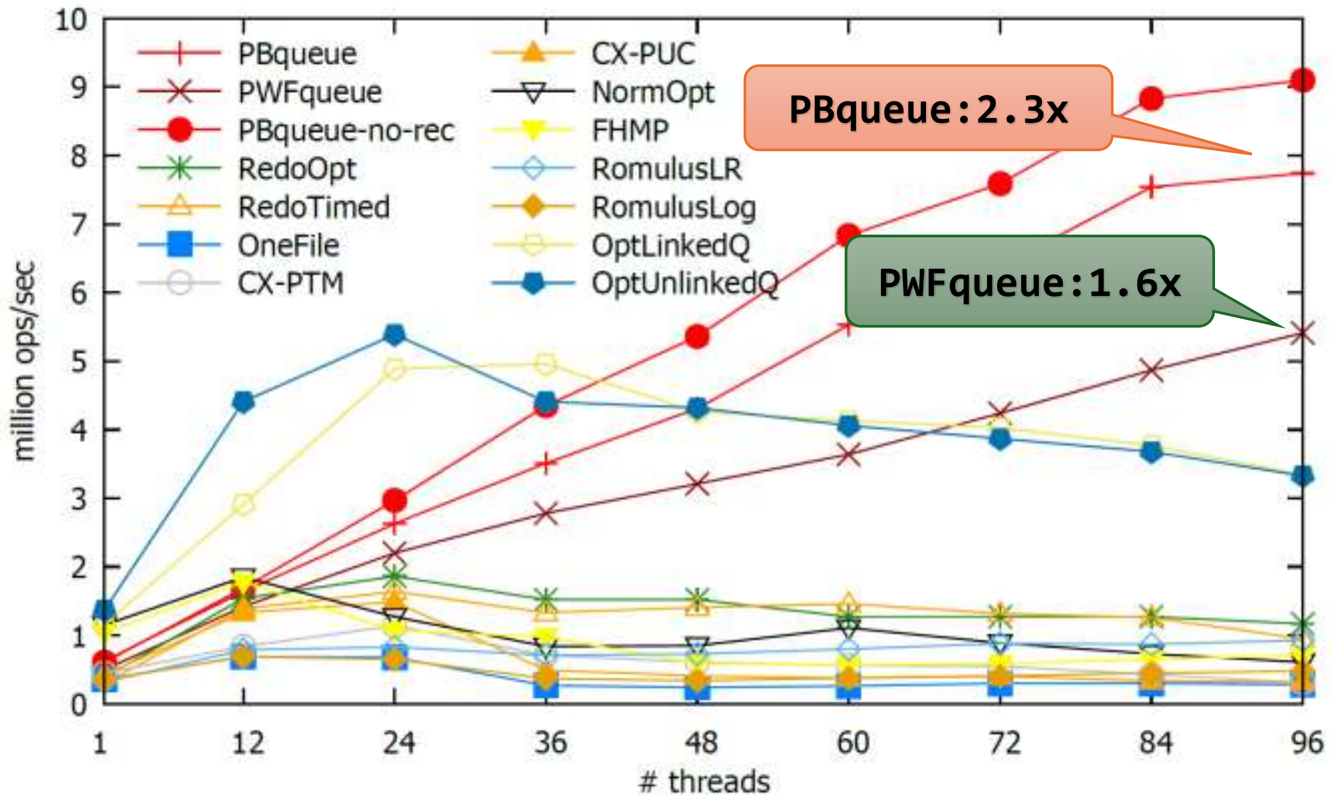
competitors guarantee only weaker consistency (e.g. durable linearizability)

Performance Analysis

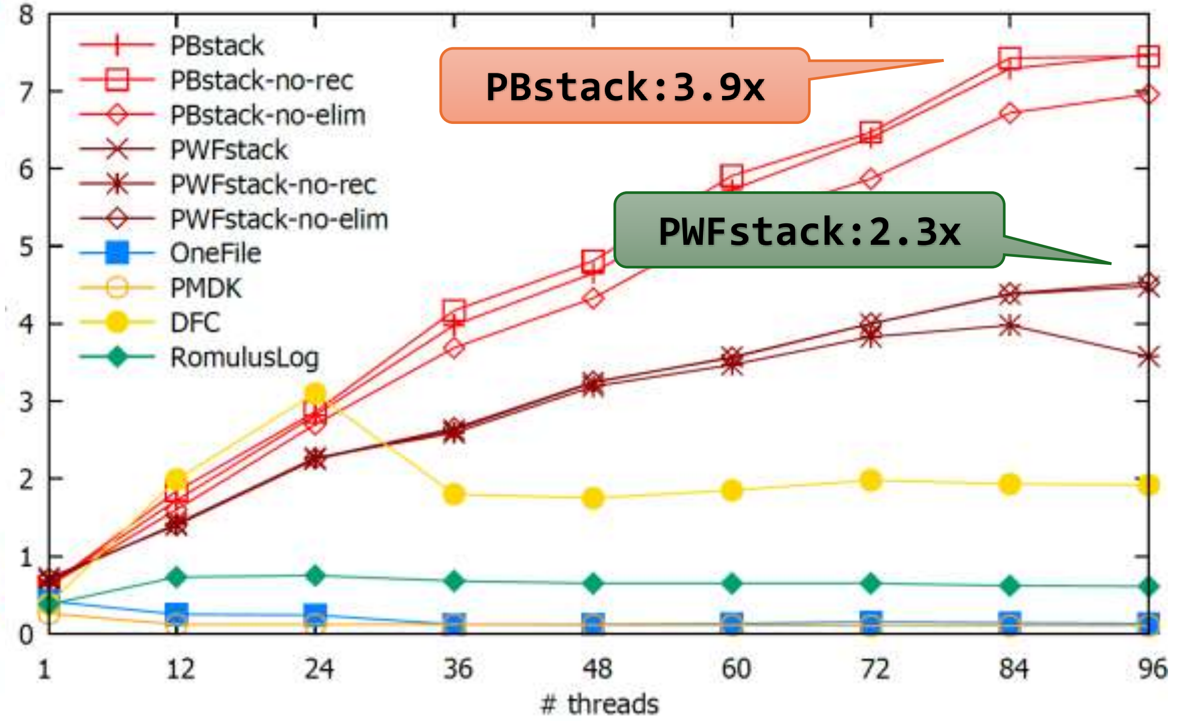
Fundamental Data Structures

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

Recoverable Queue

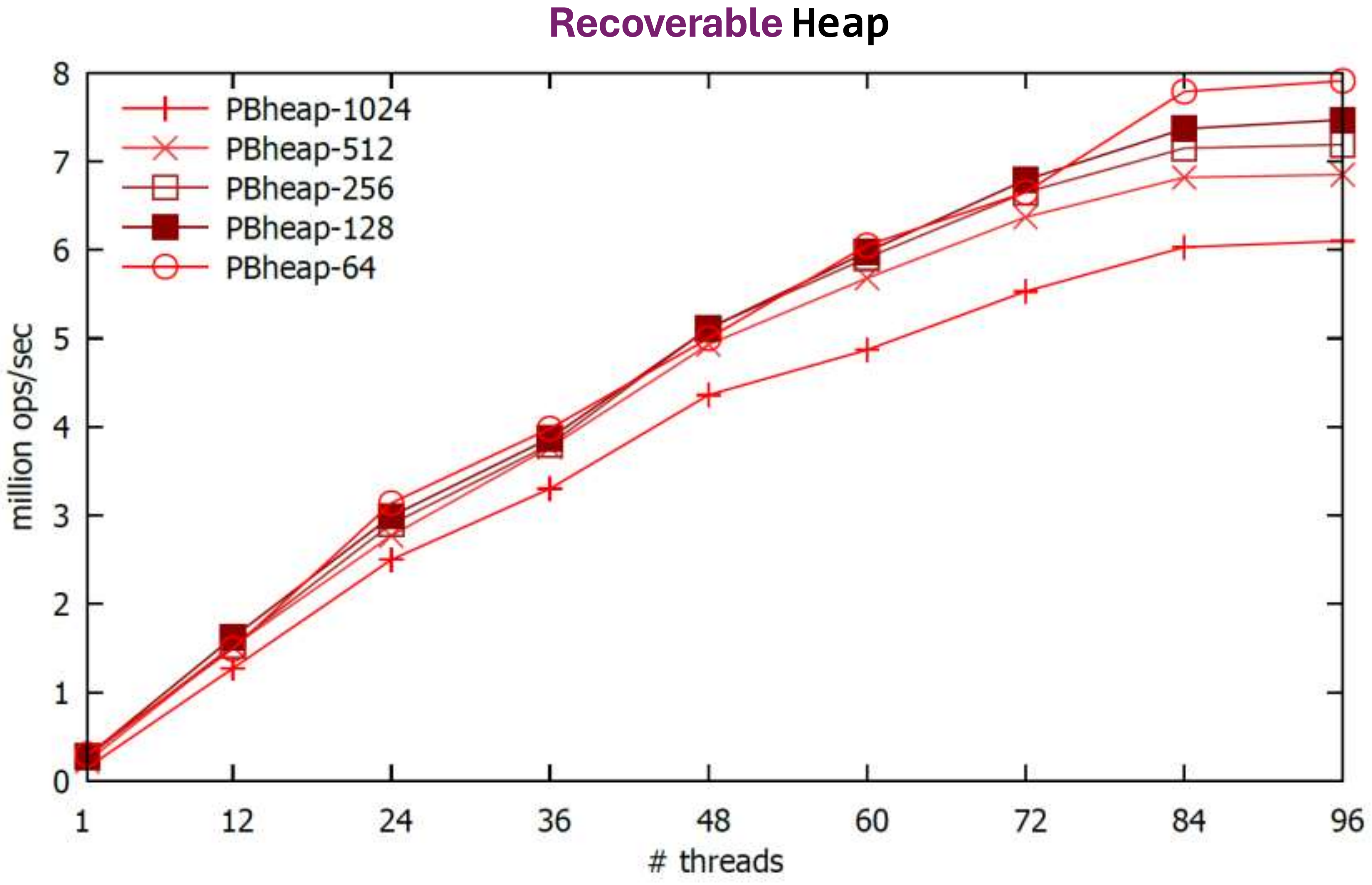


Recoverable Stack



Performance Analysis

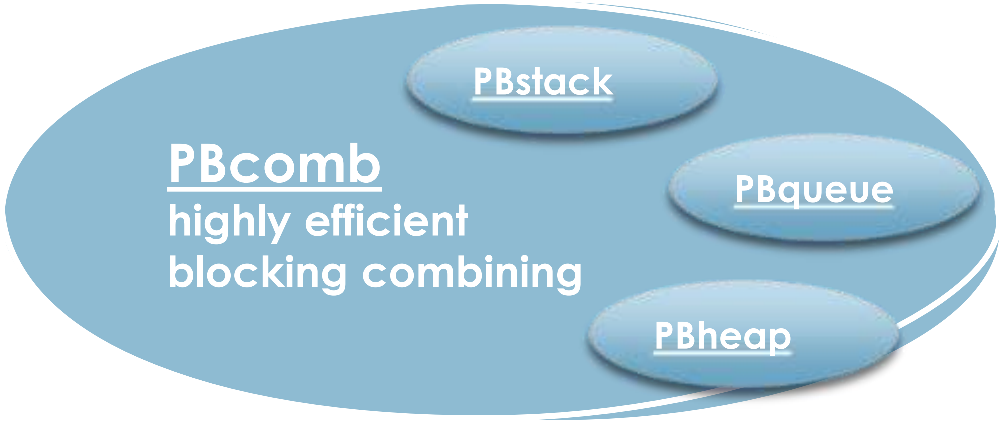
More Complex Data Structures: Heap



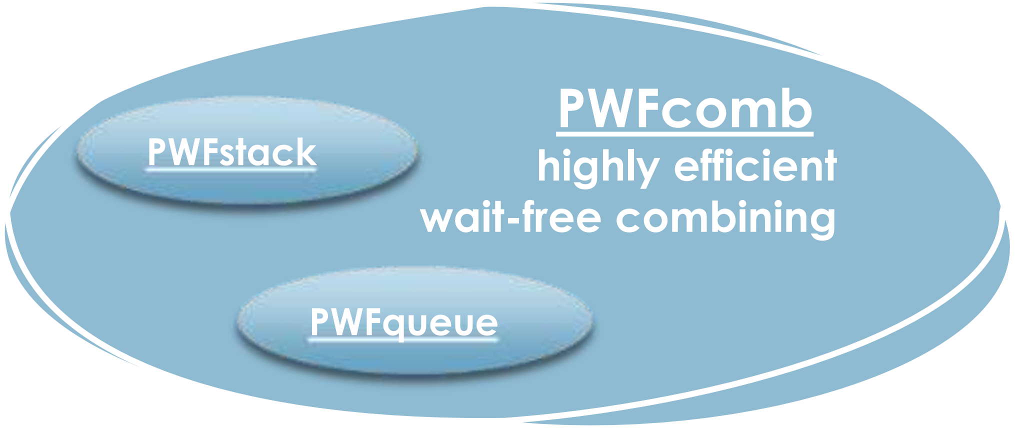
- the **first** recoverable heap implementation
- benchmark performs **equal** number of **`Insert`** and **`DeleteMin`** operations

Conclusion

Software Combining →
low-cost recoverability



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



The Performance Power Of Software Combining In Persistence

Panagiota Fatourou, Nikolaos D. Kallimanis, Eleftherios Kosmas

PPoPP'22

Persistent Software Combining

Panagiota Fatourou, Nikolaos D. Kallimanis, Eleftherios Kosmas

https://arxiv.org/abs/2107.03492