Sharded Elimination and Combining for Highly-Efficient Concurrent Stacks

Ajay Singh, Nikos Metaxakis, Panagiota Fatourou

FORTH INSTITUTE OF COMPUTER SCIENCE

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
UNIVERSITY OF CRETE

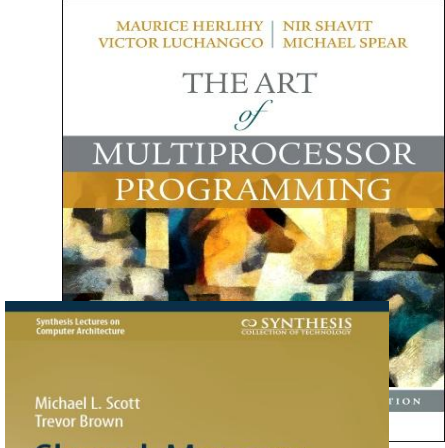
HAR.S.H.
HARdware-aware Similarity searH

Greece 2.0
NATIONAL RECOVERY AND RESILIENCE PLAN

Funded by the European Union
NextGenerationEU

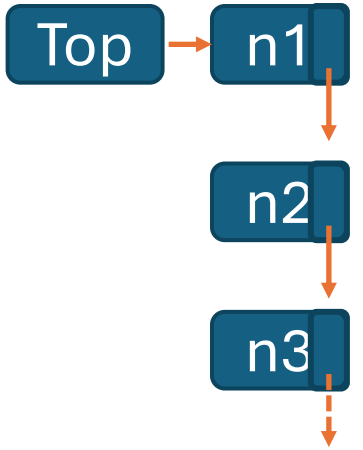
* Research funded by project HAR.S.H. (project no. ΥΠ3ΤΑ-0560901), within the framework of the National Recovery and Resilience Plan “Greece 2.0” with funding from the European Union – NextGenerationEU.

Concurrent Stacks



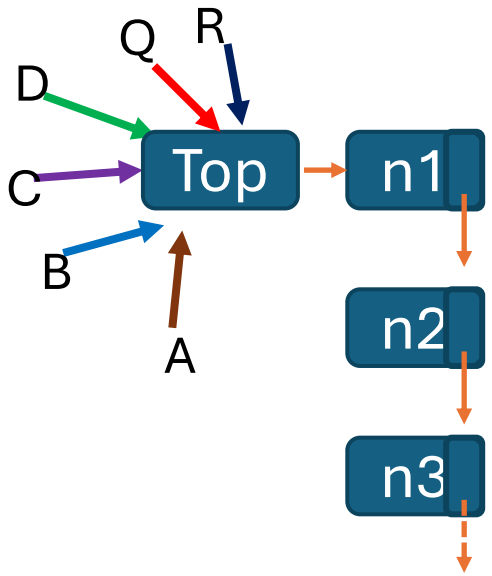
- Fundamental to many concurrent applications.
- Used in inter-process communication structures, scheduling techniques, memory pools, graph algorithms and in garbage collection freelists.

- Provide Last In First Out (LIFO) semantics where threads modify a shared top pointer using concurrent Push or Pop operations.



Problem

Top pointer becomes a contention hotspot, causing synchronization overhead and forming a sequential bottleneck, limiting throughput and scalability.



Approaches to Reduce the Bottleneck

- **Combining:** threads delegate their operations to a designated combiner thread, avoiding repeated accesses to the shared top pointer.

  → Reduces synchronization overhead

- **Elimination:** A push can cancel a concurrent pop, allowing both to complete while entirely skipping the need to access the top pointer.

  → More concurrency

Reduce synchronization overhead and ease sequential bottleneck (operations can complete in parallel), lowering overall cache invalidation traffic at the top pointer of the shared stack.

Some Specific Implementations

- **EBF:** Uses elimination as a backoff. [Hendler, Shavit & Yerushalmi, SPAA '04]
  - High overhead per eliminated push/pop pair.
  - Push and pop operations may fail to eliminate despite being concurrent.
- **CC:** Uses a FIFO list for combining. [Fatourou & Kallimanis, PPoPP '12]
  - Limits achievable concurrency: while the combiner executes, other threads wait & cannot eliminate/combine.
- **TSI:** Stack as a pool of timestamped nodes. [Dodd, Haas & Kirsch, POPL '15]
  - Fast push operations at the cost of slower pop operations.

Contribution

SEC: Sharded Elimination and Combining

A lightweight method that unifies elimination and combining to build a fast & scalable concurrent stack

- Aggregator and Batch structures to divide and localize contention [Roh, Wei, Ruppert, Fatourou, Jayanti, Shun, PPoPP '25].
- Per batch two counter-based approach significantly lowers the overhead.
- Elimination and combining is performed in parallel across batches.
- Eliminates all push-pop pairs within a batch; determining the return values of combined operations is almost free.

High-Level Description & Key Components

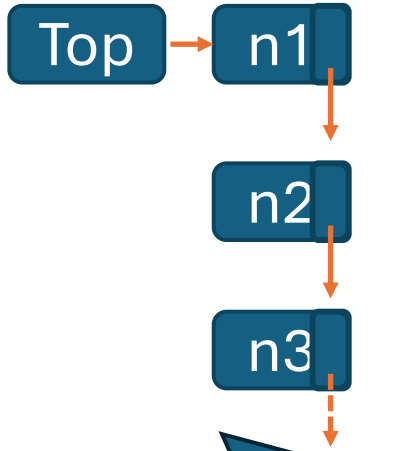
Within each batch threads cooperatively perform elimination and combining

Different batches execute in parallel with a subset of threads.

Multiple batches per aggregators

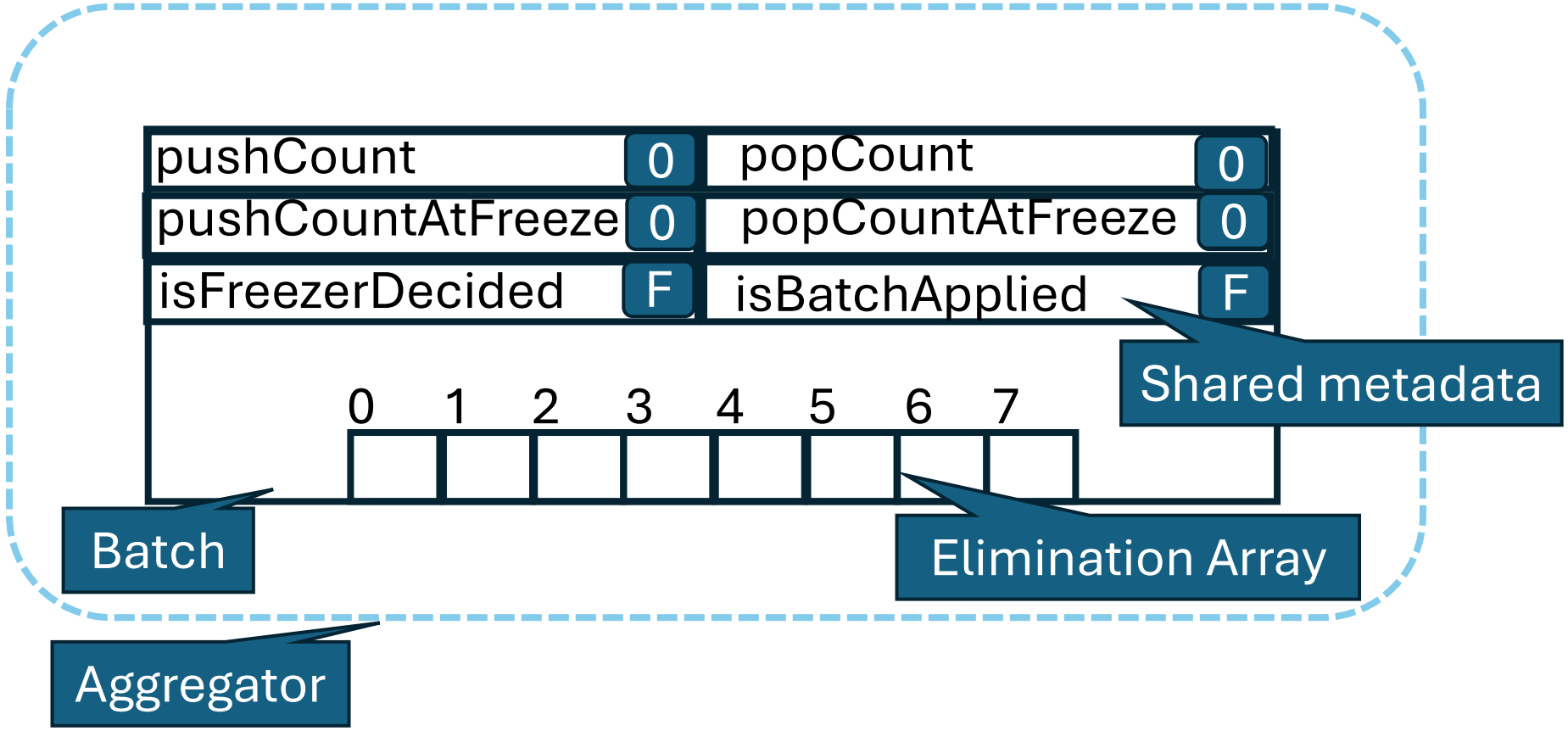
Each aggregator has one **active batch**

Multiple aggregator structures: distribute contention between threads



Shared Stack

A Batch Zoomed-In

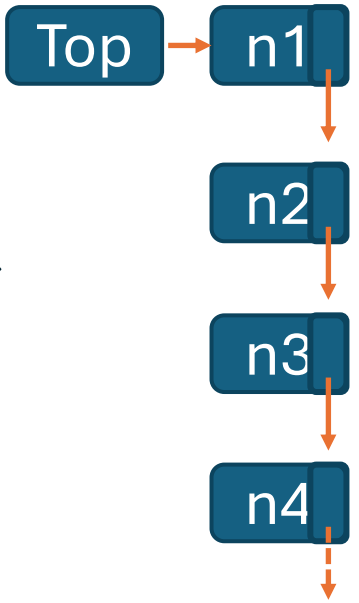


Threads Execute in Three Phases

- Determines which push and pop operations in the active batch will participate in the next phase.
- Two fetch-and-add counters (one for pushes and one for pops).

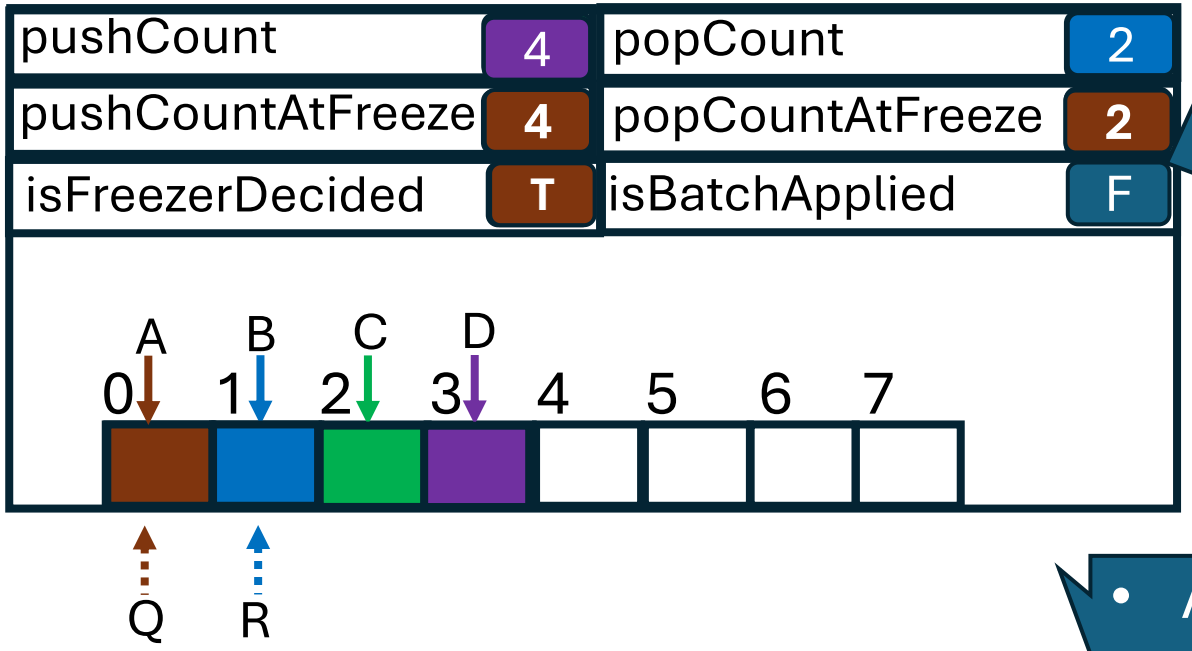
Freezing → Elimination → Combining

- If all are pushes, create a substack and apply to the shared stack.
- If all are pops, remove a substack from the shared stack. Threads scan this substack to find their return values.



Freeze Phase

Active Batch



threads A, B, C and D execute push
threads Q and R execute pop

Threads **announce** operations in the active batch
- Threads get a slot in the eliminationArray.
- Fetch&Add on `pushCount` or `popCount`.

Decide operations that are part of the batch.
- One of the slot-0 threads is elected as freezer using Test&set on `isFreezerDecided`.
- Other threads wait for freezer to finish.

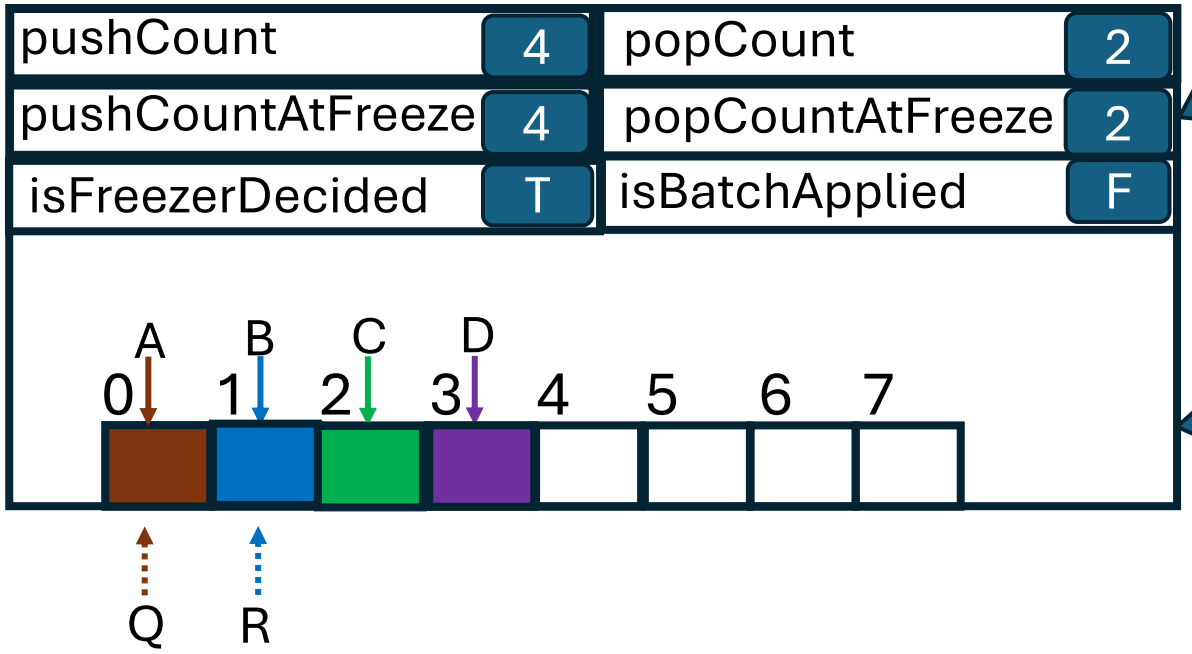
Freezer actions:

```
1. popCountAtFreeze ← popCount;
2. pushCountAtFreeze ← pushCount;
3. active batch ← new Batch()
```

- Am I part of the batch? Compare slot values with `popCountAtFreeze` or `pushCountAtFreeze`.
- If not, retry at a subsequent active batch.

Elimination Phase

Frozen Batch



Threads check for matching operations to eliminate:

- push: slot value < popCountAtFreeze
- pop : slot value < pushCountAtFreeze

To eliminated perform the exchange:

- push returns true.
- pop returns the value of the matching push.

Threads with non-eliminated operations move to combining phase.

→ thread with push operation

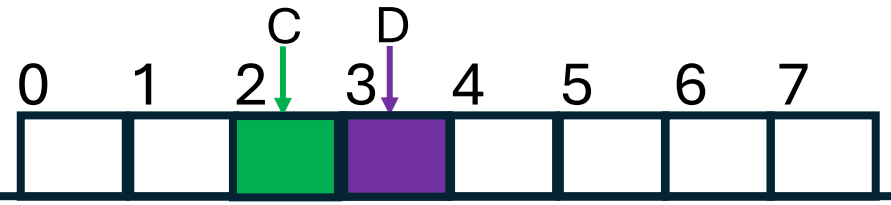
⇢ thread with pop operation

Combining Phase

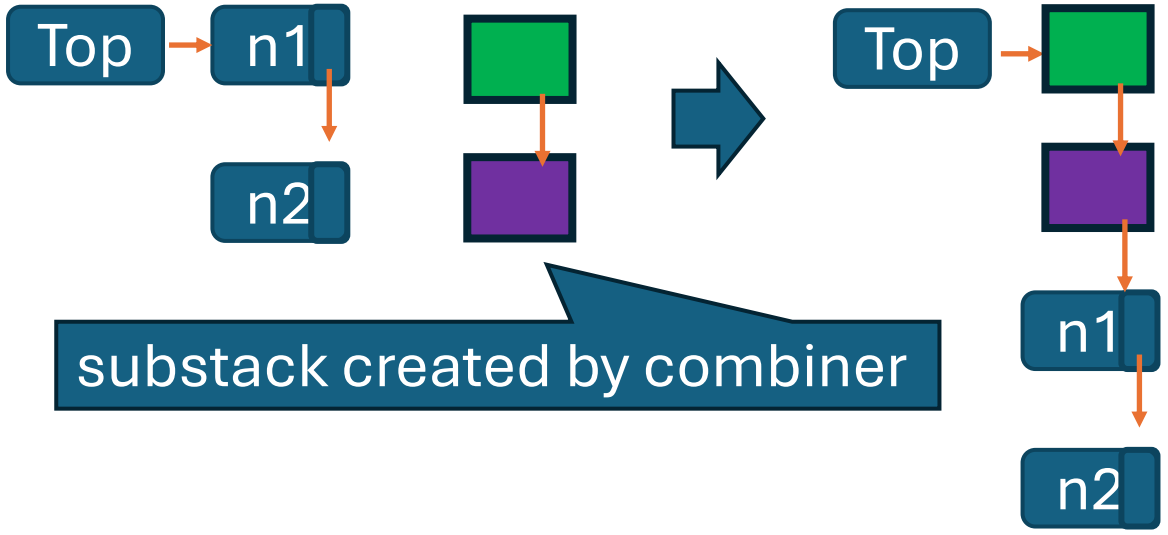
- First non-eliminated thread becomes the combiner.
- Combiner applies the operations to the shared stack.
- Other threads wait for combiner to finish to return responses.

Frozen Batch

pushCount	4	popCount	2
pushCountAtFreeze	4	popCountAtFreeze	2
isFreezerDecided	T	isBatchApplied	T



Case1: all operations pushes.



→ thread with push operation

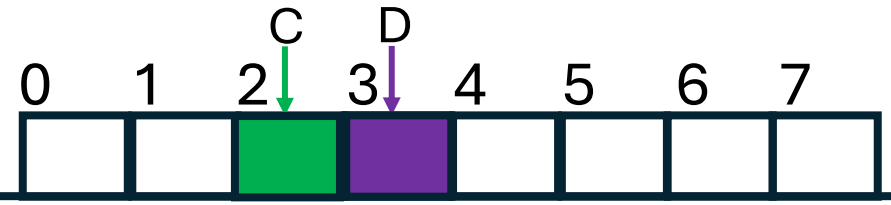
⇢ thread with pop operation

Combining Phase

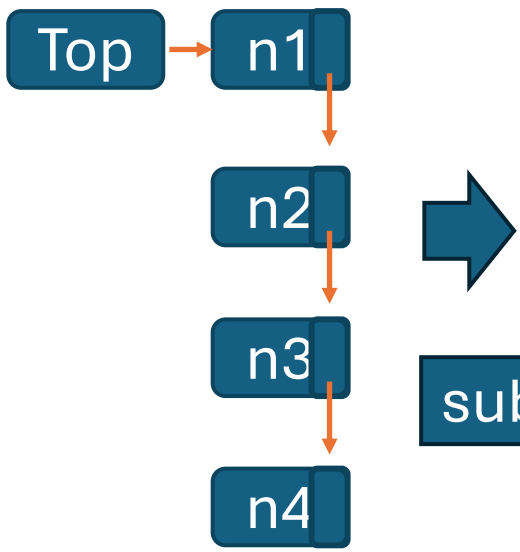
- First non-eliminated thread becomes the combiner.
- Combiner applies the operations to the shared stack.
- Other threads wait for combiner to finish to return responses.

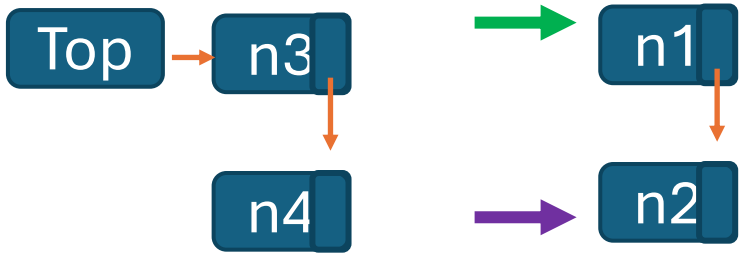
Frozen Batch

pushCount	4	popCount	2
pushCountAtFreeze	4	popCountAtFreeze	2
isFreezerDecided	T	isBatchApplied	T



Case2: all operations are pops.



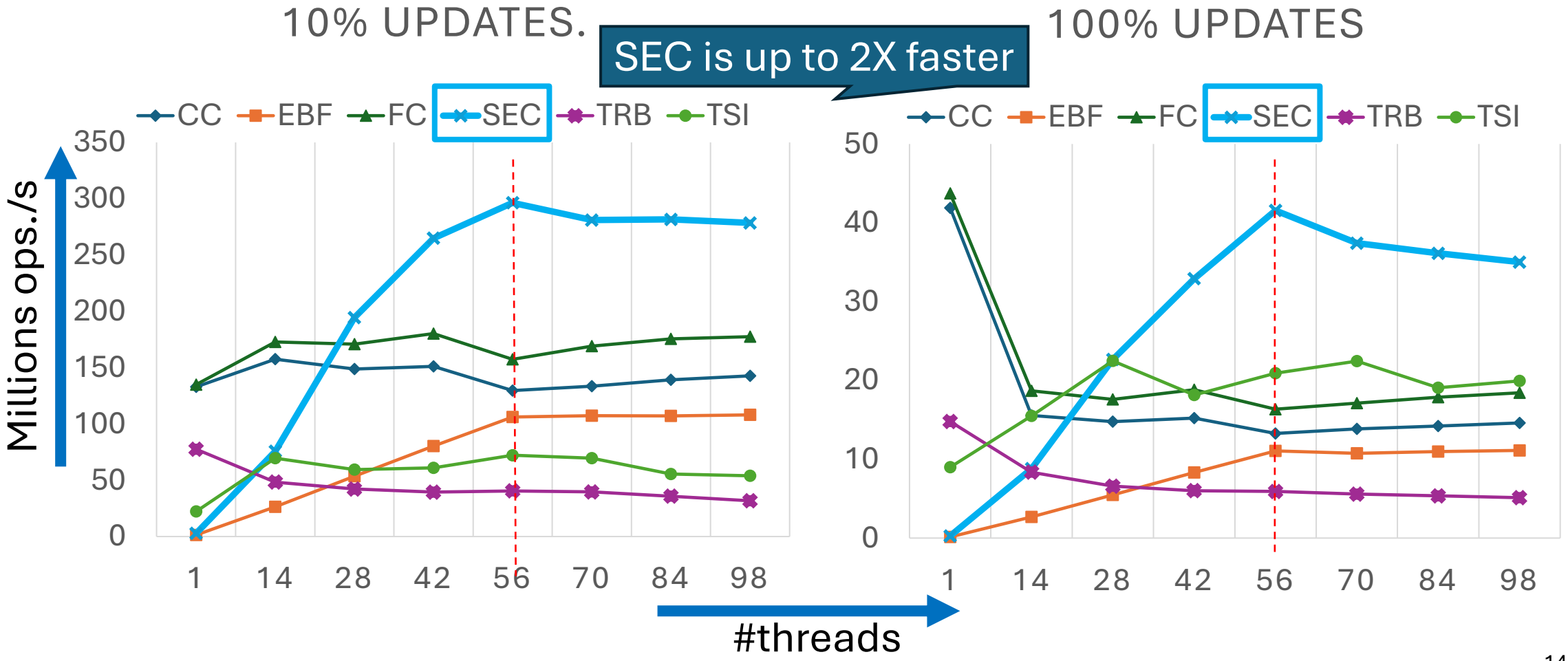


substack removed by combiner

→ thread with push operation

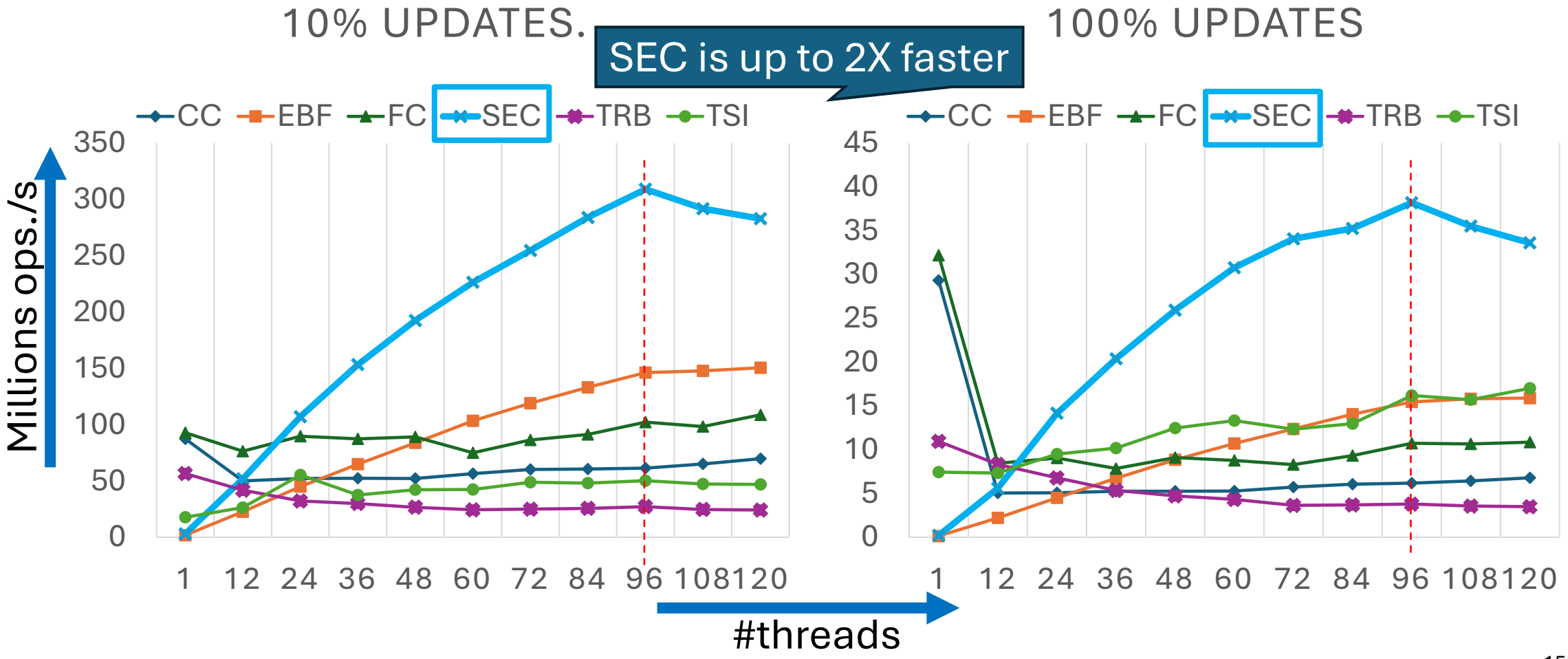
⇢ thread with pop operation

Stack Throughput on a 56-thread Machine



UoC & ICS FORTH, Greece 2.0 and NextGeneration EU

Stack Throughput on a 96-thread Machine



Conclusion

- A fast linearizable but blocking concurrent stack that uses a lightweight method that unifies elimination and combining.
  - Aggregator and Batch structures to divide and localize contention.
  - Per batch two counter-based approach significantly lowers the overhead.
  - Elimination and combining is performed in parallel across batches.
  - Eliminates all push-pop pairs within a batch; determining the return values of combined operations is almost free.

In Paper:

- Reclamation of stack nodes and batches, more experiments with different workloads, and correctness.

paper

HAR.S.H. HARdware-aware Similarity searH

Reclamation

- reclaiming batch objects.

- reclaiming stack nodes.

Conclusion

- A fast linearizable but blocking concurrent stack that uses a unified lightweight elimination and combining mechanism.
  - Efficient batch level elimination using two counters.
  - A combiner applies a batch of nodes to the shared stack with a single CAS.
  - All push-pop pairs within a batch are eliminated.
  - Determining the return values of combined operations is simple.
  - Elimination and combining occurs in parallel across batches.

In Paper:

- Reclamation of stack nodes and batches, more experiments with different workloads, and correctness.

Ajay's Webpage

HAR.S.H. HARdware-aware Similarity searH

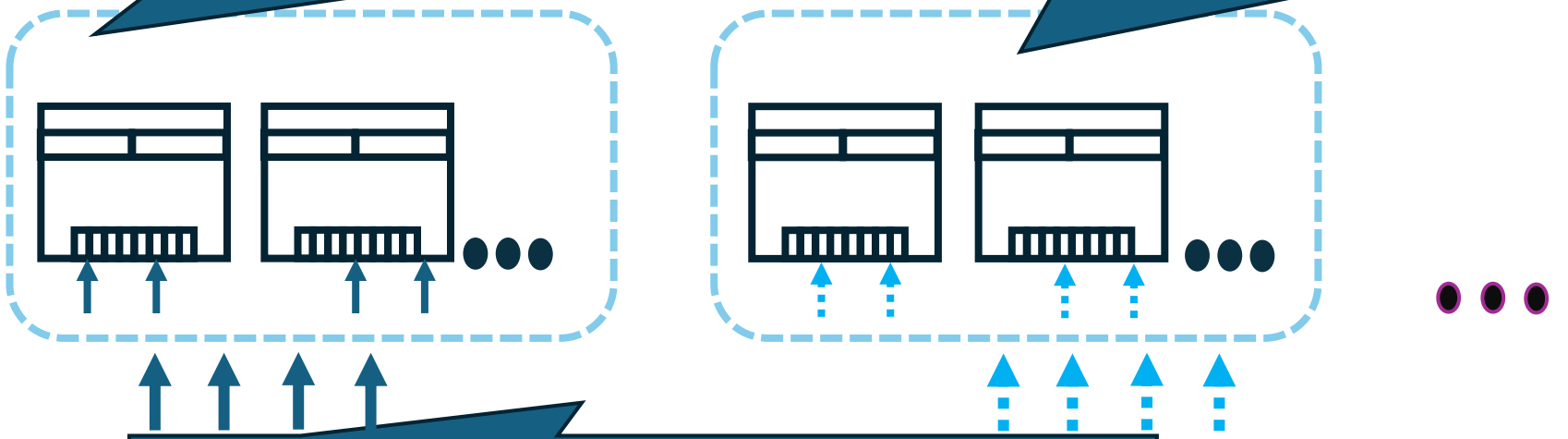
High-Level Description & Key Components

Within each batch threads cooperatively perform elimination and combining

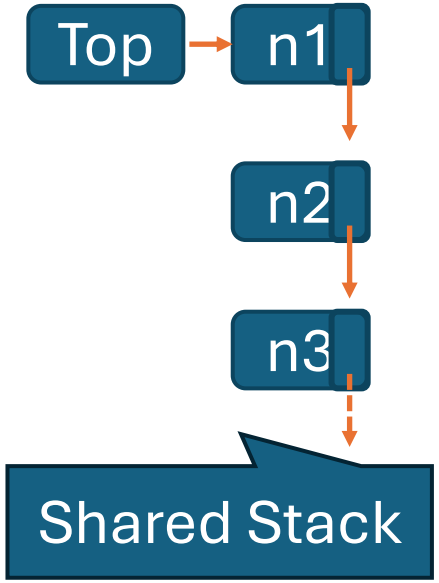
Different batches execute in parallel with a subset of threads.

Multiple batches per aggregators

Each aggregator has one active batch



Multiple aggregator structures: distribute contention between threads



Use of aggregator and batch structures is inspired from Aggregating Funnels [Wei et. al, PPoPP '25]

Our Contribution

SEC: Sharded Elimination and Combining

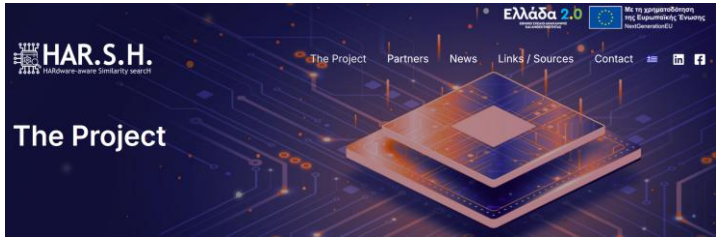
A lightweight design that unifies elimination and combining to build a fast & scalable concurrent stack

- Efficient batched elimination:
  - A simple method using two fetch-and-add counters to match and eliminate operations.
- Batched combining for shared stack access:
  - Threads add or remove a batch of nodes to or from the shared stack using a single compare&swap operation.
- All push-pop pairs within a batch are eliminated, and determining the return values of combined operations is simple.

Conclusion

- A fast linearizable but blocking concurrent stack that uses a unified lightweight elimination and combining mechanism.
  - batch level meta data.
  - division between aggregators
  - c

HAR.S.H.
HARdware-aware Similarity searH



In Paper:

- Reclamation of stack nodes and batches, more experiments with different workloads, and correctness.

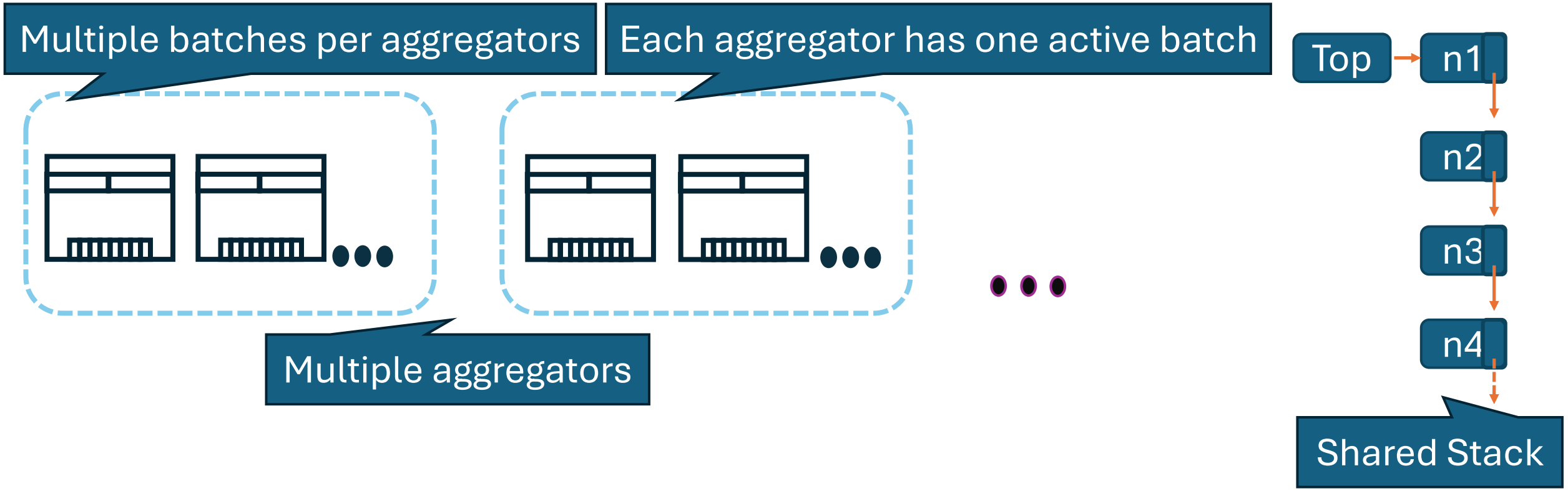
Greece 2.0
NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union
NextGenerationEU

SEC: **S**harded **E**limination and **C**ombining Stack

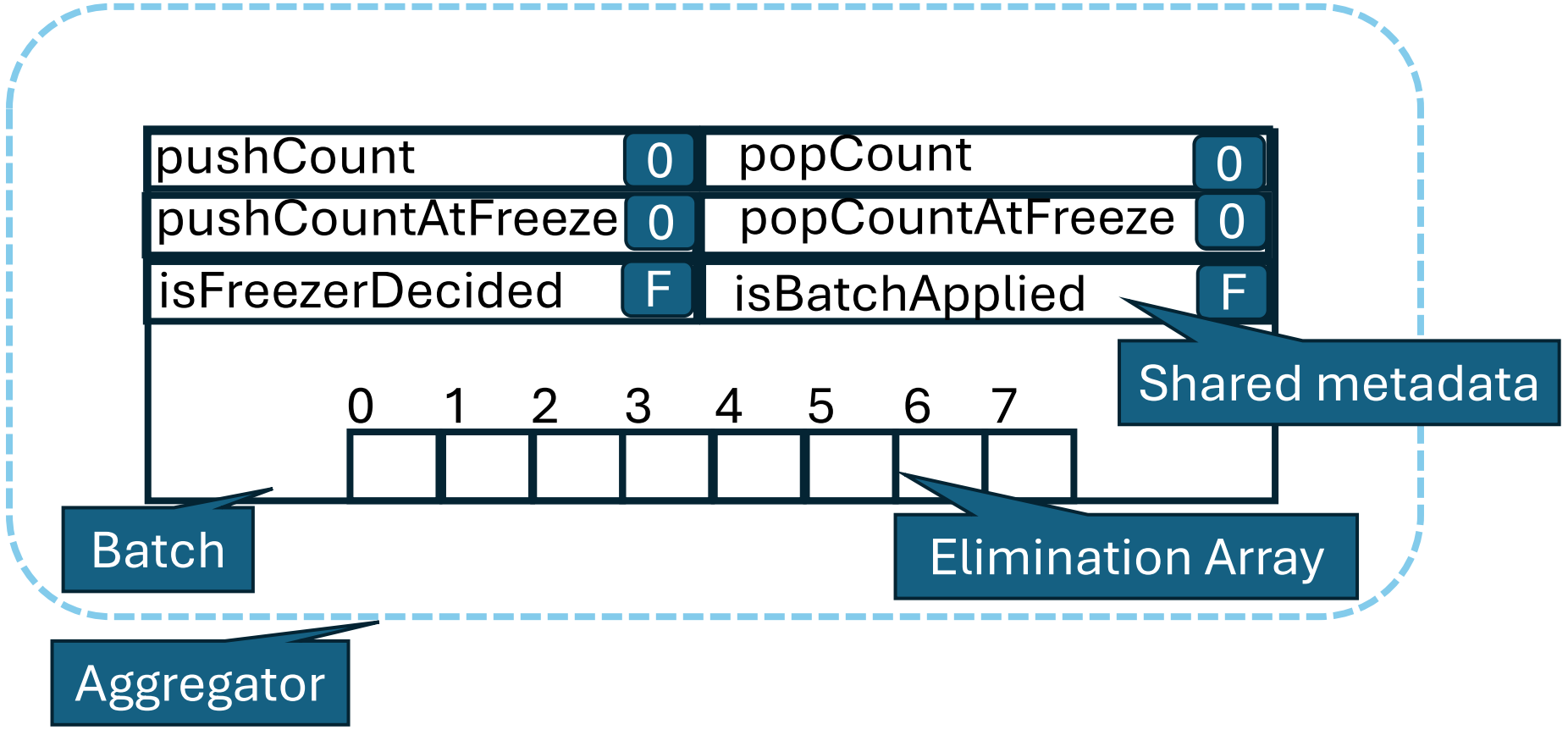
A light-weight design that unifies elimination and combining for fast & scalable concurrent stack

- Divide threads across multiple **aggregators**, which helps to distribute contention. Inside each aggregator, threads form **batches** of push/pop operations.
- Within a **batch**, threads cooperatively **perform elimination and combining**. Different batches execute in parallel.
- Each batch eventually reduces to only pushes, only pops, or becomes empty, and the remaining operations are applied to the **shared stack**.

Key Components



Key Components: Zoomed-In

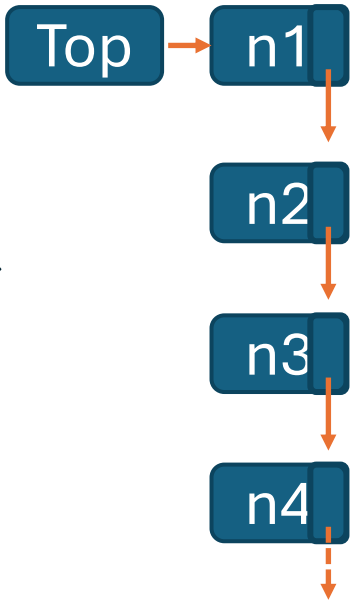


Threads Execute in Three Phases

- Determines which push and pop operations in the active batch will participate in the next phase.
- Two fetch-and-add counters (one for pushes and one for pops).

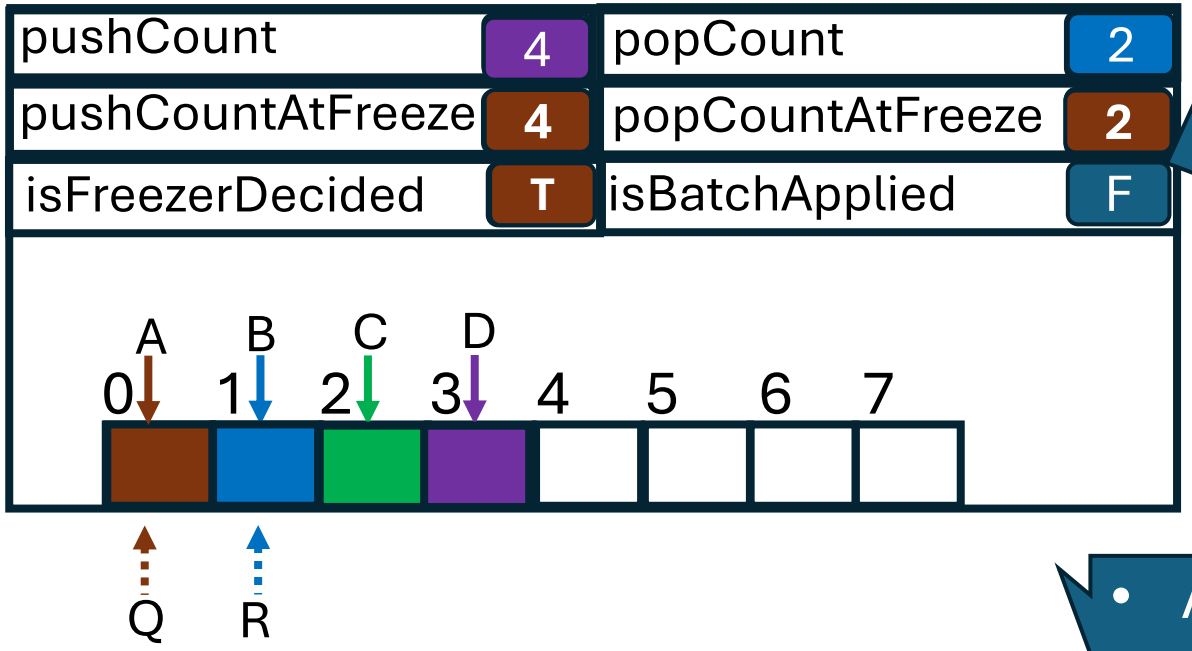
Freezing → Elimination → Combining

- If all are pushes, create a substack and apply to the shared stack.
- If all are pops, remove a substack from the shared stack. Threads scan this substack to find their return values.



Freeze Phase

Active Batch



threads A, B, C and D execute push
threads Q and R execute pop

Threads **announce** operations in the active batch
- Threads get a slot in the eliminationArray.
- Fetch&Add on `pushCount` or `popCount`.

Decide operations that are part of the batch.
- One of the slot-0 threads is elected as freezer using Test&set on `isFreezerDecided`.
- Other threads wait for freezer to finish.

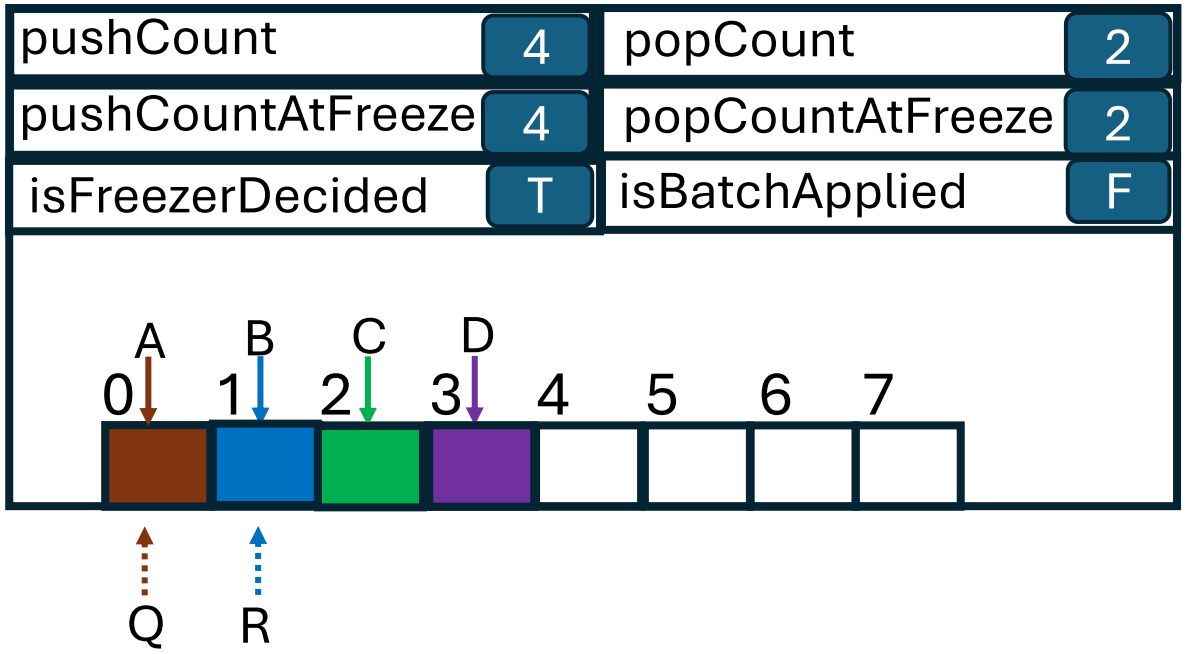
Freeze mechanism:

```
1. popCountAtFreeze ← popCount;
2. pushCountAtFreeze ← pushCount;
3. active batch ← new Batch()
```

- Am I part of the batch? Compare slot values with `popCountAtFreeze` or `pushCountAtFreeze`.
- If not, retry at a subsequent active batch.

Elimination Phase

Frozen Batch



Check if operations can be eliminated:

- push: slot value < popCountAtFreeze
- pop : slot value < pushCountAtFreeze

If can be eliminated:
Exchange: push returns true and pop returns the value of the matching push.

Threads with non-eliminated operations move to combining phase.

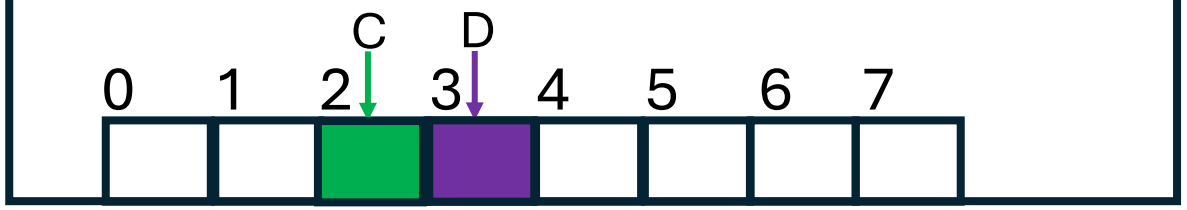
→ thread with push operation
⇢ thread with pop operation

Combining Phase

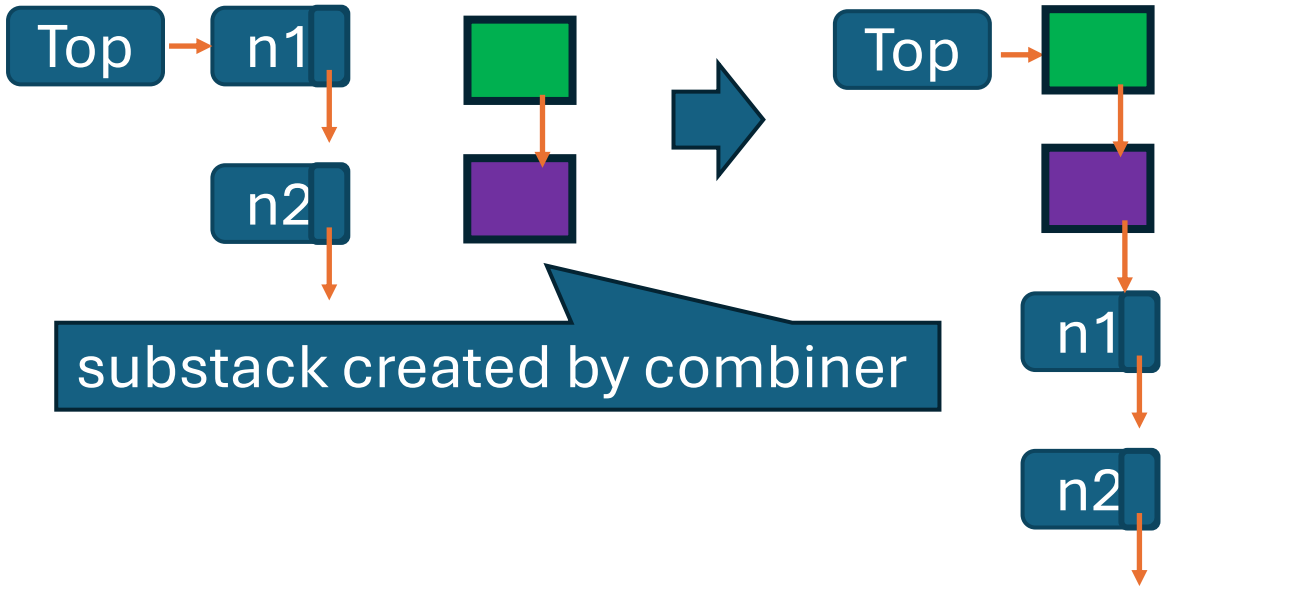
- First non-eliminated thread becomes the combiner.
- Combiner applies the operations to the shared stack.
- Other threads wait for combiner to finish to return responses.

Frozen Batch

pushCount	4	popCount	2
pushCountAtFreeze	4	popCountAtFreeze	2
isFreezerDecided	T	isBatchApplied	T



Case1: all operations pushes.



→ thread with push operation

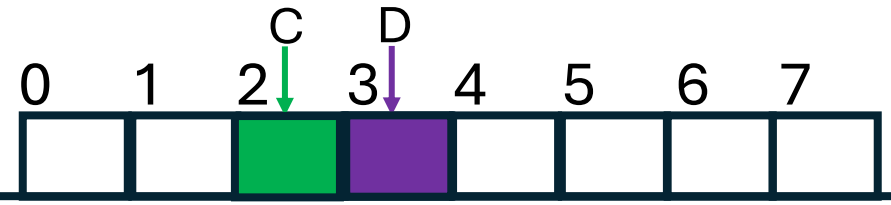
⇢ thread with pop operation

Combining Phase

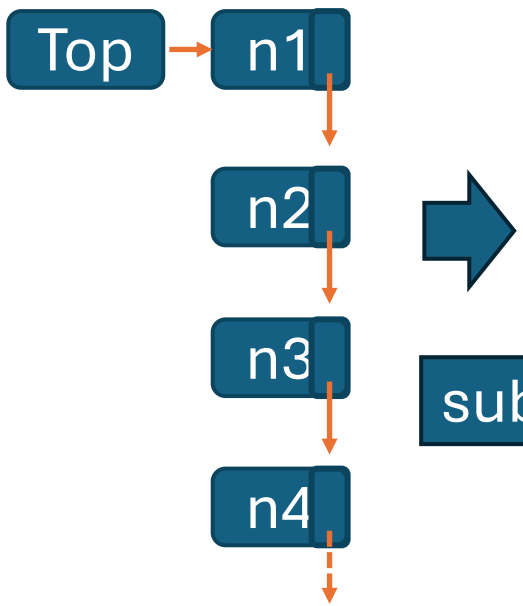
- First non-eliminated thread becomes the combiner.
- Combiner applies the operations to the shared stack.
- Other threads wait for combiner to finish to return responses.

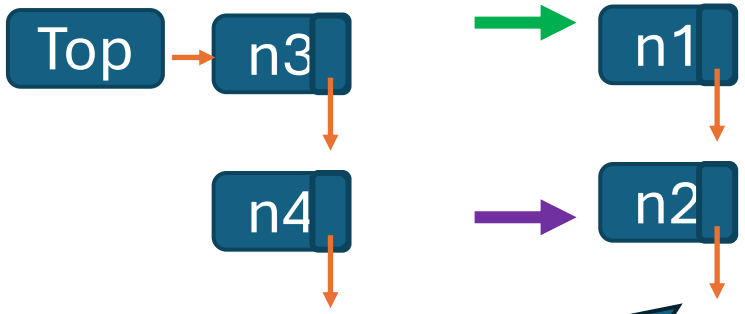
Frozen Batch

pushCount	4	popCount	2
pushCountAtFreeze	4	popCountAtFreeze	2
isFreezerDecided	T	isBatchApplied	T



Case2: all operations are pops.





substack removed by combiner

→ thread with push operation

⇢ thread with pop operation