A Family of Fast and Memory Efficient Lock- and Wait-Free Reclamation

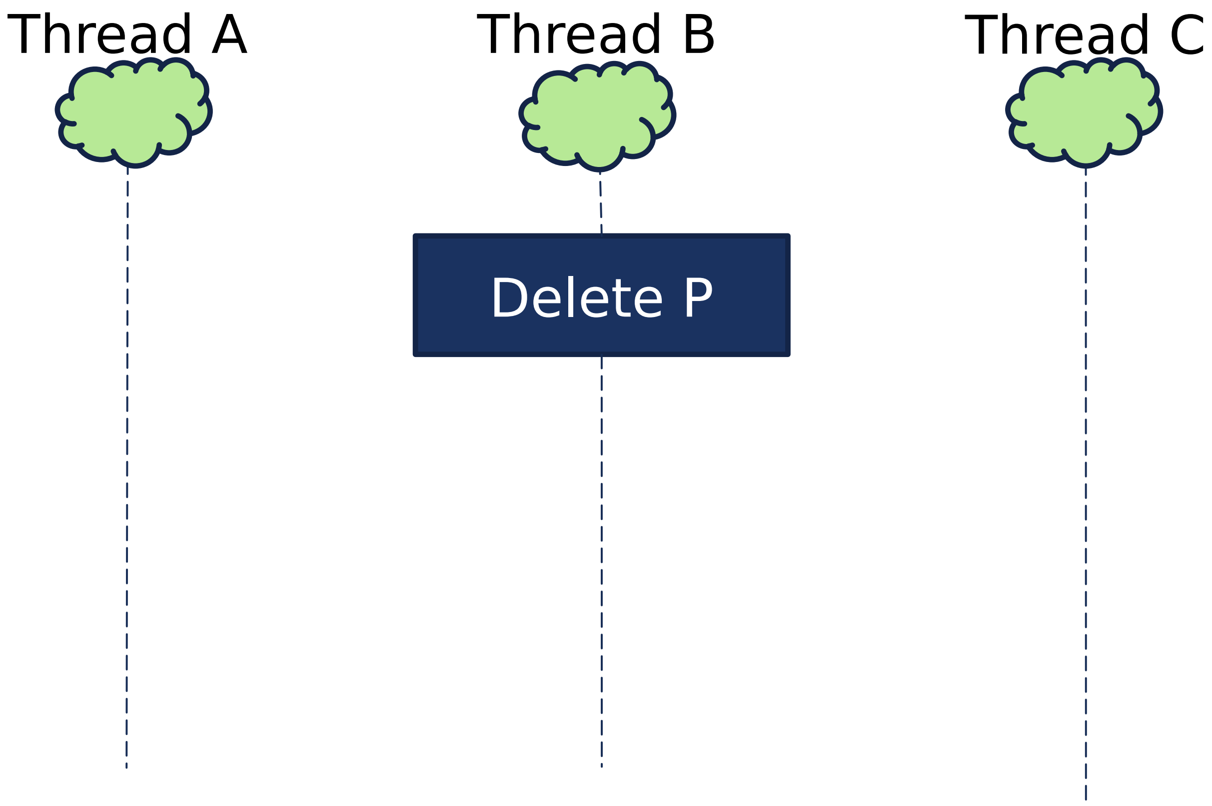
Ruslan Nikolaev, rnikola@psu.edu, Penn State University, USA

Binoy Ravindran, binoy@vt.edu, Virginia Tech, USA

Memory Reclamation

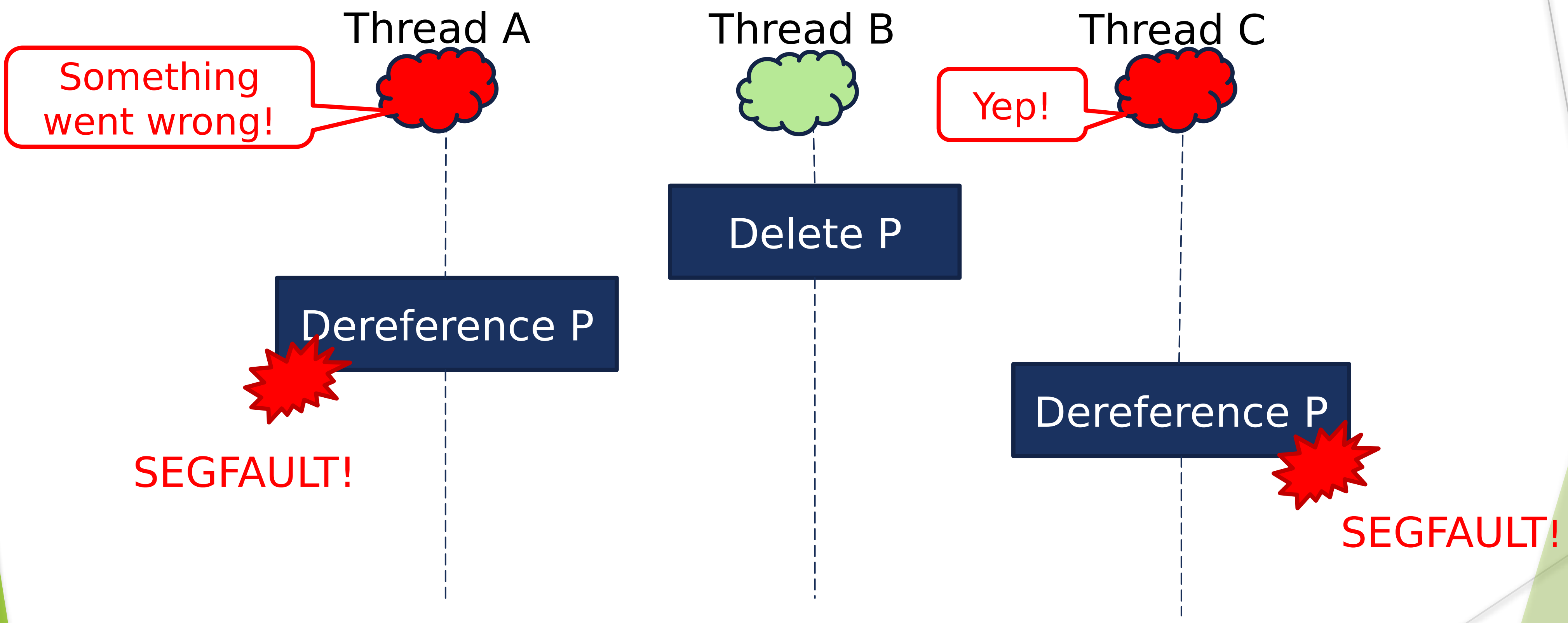
- **Non-blocking** data structures do not use simple mutual exclusion
  - A *concurrent* thread may hold an **obsolete** pointer to an object which is about to be freed by *another* thread
  - *Safe memory reclamation* (SMR) schemes are typically used for unmanaged code (C/C++)

Memory Reclamation



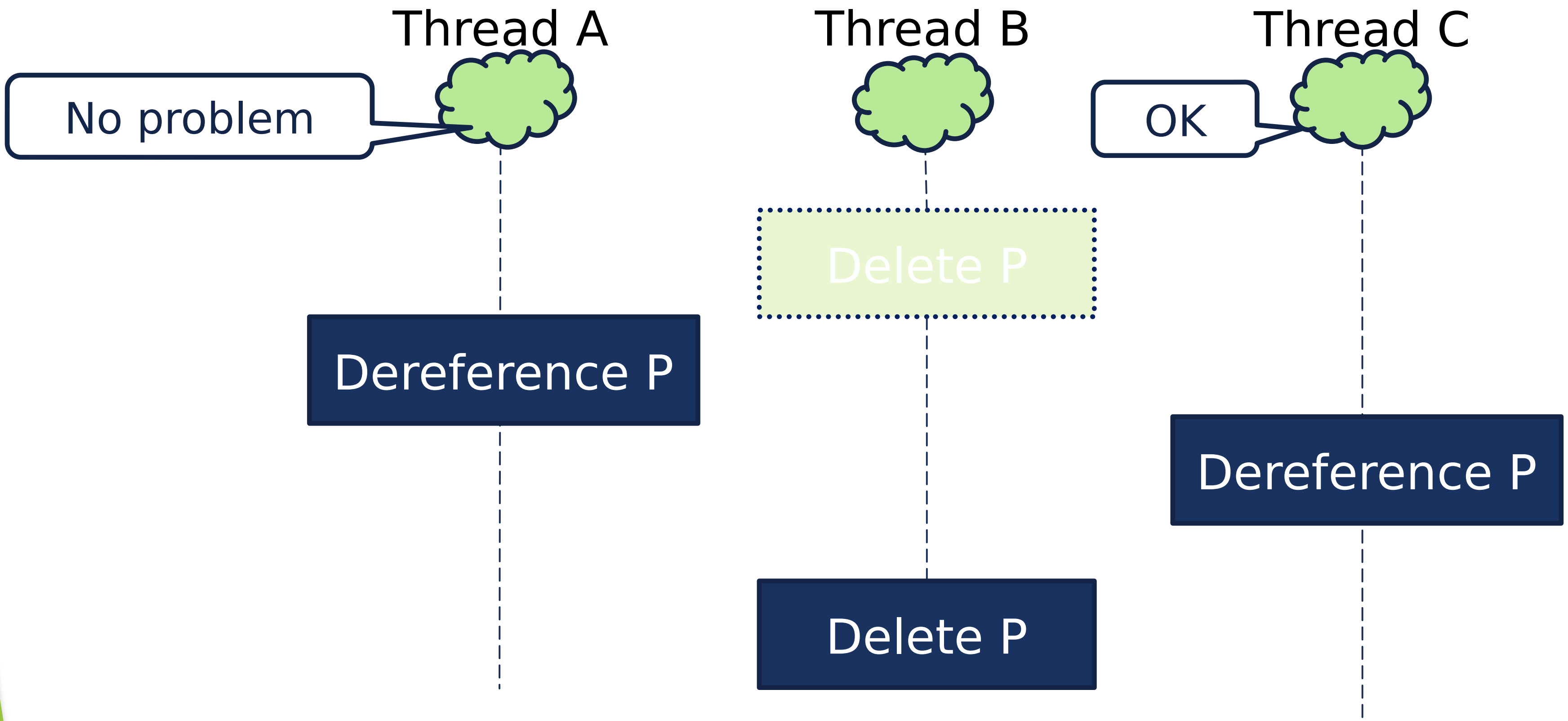
One thread wants to de-allocate a memory object which is still reachable by concurrent threads

Memory Reclamation



One thread wants to de-allocate a memory object which is still reachable by concurrent threads

Memory Reclamation



Postpone de-allocation until it is safe to do so

Non-blocking algorithms

- **Obstruction-free** algorithms guarantee that a thread makes progress if it runs in isolation from everything else
- **Lock-free** algorithms guarantee that at least **one** thread always makes progress in a finite number of steps
- **Wait-free** algorithms guarantee that **all** threads always make progress in a finite number of steps

Memory Reclamation

- Reclamation workload **balancing**
  - Read operations dominate, but data is still modified
    - In typical SMR schemes, most threads are not actively reclaiming memory
  - The problem have not received adequate attention in the literature
- **Synchronous** vs. **asynchronous** reclamation
  - In typical SMR schemes, threads *periodically* examine which objects marked for deletion can be safely freed
  - Reference counting: an *arbitrary* thread with the last reference frees an object

Memory Reclamation

- **Reference counting**
  - Impractical due to very high overheads when accessing objects
  - Hyaline [PLDI '21] is an approach where reference counters are only used when objects are retired
    - **Pros**: asynchronous and exhibits high performance, protects against **stalled** threads
    - **Cons**: can still use unbounded memory (i.e., blocking) when threads starve
- We present **Crystalline**
  - **Crystalline-L** is based on Hyaline-1S but is lock-free even when threads starve
  - **Crystalline-LW** extends Crystalline-L to make it wait-free under some circumstances
  - **Crystalline-W** further extends Crystalline-LW to make it fully wait-free

Memory Reclamation

Scheme	Balanced	Fast	[With Restart] 1 DS	[With Restart] 2+ DS	[W/o Restart] 1 DS	[W/o Restart] 2+ DS	S-Free	Header
EBR [TechReport '04]	✘	✔	BLK	BLK	BLK	BLK	✔	1 word
IBR [PPoPP '18]	✘	✓	WF	BLK	BLK	BLK	✘	3 words
HP [TPDS '04]	✘	✘	WF	**WF**	LF	**LF**	✘	1 word
HE [SPAA '17]	✘	✓	WF	BLK	LF	BLK	✘	3 words
WFE [PPoPP '20]	✘	✓	WF	**WF**	WF	**WF**	✘	3 words
Hyaline-1 [PLDI '21]	✔	✓	BLK	BLK	BLK	BLK	✔	3 words
Hyaline-1S [PLDI '21]	✔	✓	LF	BLK	BLK	BLK	✔	3 words
Crystalline-L	✔	✔	LF	BLK	LF	BLK	✔	3 words
Crystalline-LW	✔	✔	WF	BLK	LF	BLK	✔	3 words
Crystalline-W	✔	✔	WF	**WF**	WF	**WF**	✔	3 words

Crystalline-L

- **Background (Hyaline)**
  - Threads explicitly annotate each operation
  - When objects are detached from a data structure, they are first **retired** and then **freed** when it is safe to do so
  - Hyaline-1S is a variant that bounds memory usage for **stalled** threads by explicitly tracking *local pointers* via a special **protect** method using the global era clock
    - Each allocated object is assigned a “birth era”
    - Not lock-free unless operations are periodically restarted for starving threads
    - ***Example***: one “unlucky” thread is stuck traversing a list because it keeps growing
- **Crystalline-L** adopts a different API
  - Hyaline-1S’s API enables retrieving an unbounded number of local pointers
  - Alternative APIs used in Hazard Pointers [TPDS’04] or Hazard Eras [SPAA’17] explicitly differentiate each local pointer reservation in **protect**

Crystalline-L: API

- **protect():** safely retrieve a pointer to the protected object by creating a reservation, each local pointer should be reserved **separately** and identified by an index
- **retire:** mark an object for deletion; the retired object must be deleted from the data structure first, i.e., only in-flight threads can still access it
- **clear:** reset all prior reservations made by the current thread in protect
- **alloc_node:** allocate a memory block and initialize its alloc era to the global era clock value

Crystalline-L: Challenges

- Hyaline-1S aggregates objects in a **batch**
  - Can only **retire** the entire batch
  - Each thread has its own **retirement list,** and each object from the batch is inserted to the corresponding list
  - One of the objects keeps a per-batch reference counter
  - Needs at least MAX_THREADS+1 objects per a batch
- Crystalline-L handles MAX_IDX local pointers
  - The above problem is further aggravated
  - Needs at least MAX_THREADS×MAX_IDX+1 objects per a batch

Crystalline-L: Solution

- The required number of objects is **much lower** in practice
  - Each object is appended to the respective list only if the list’s era overlaps with the batch’s minimum birth era
- Crystalline-L uses **dynamic batches**
  - retire first checks how many lists are to be changed for the batch to be fully retired and records the location of the corresponding (per-thread) lists
  - If the number of objects in the batch suffices, retire completes by appending the objects to their corresponding lists
  - Otherwise, retire is repeated later when more objects are available
    - But the number of iterations is still **bounded** by the worst-case number of objects

Crystalline’s Wait-Freedom Challenges

- Crystalline-L is only lock-free because
  - **retire** has an unbounded loop: **protect** or another **retire** contends on the same list
    - Does not let a CAS loop in retire to converge
  - **protect** has an unbounded loop which must converge on the era value
    - The era clock unconditionally increments when a new object is allocated

Hardware Support

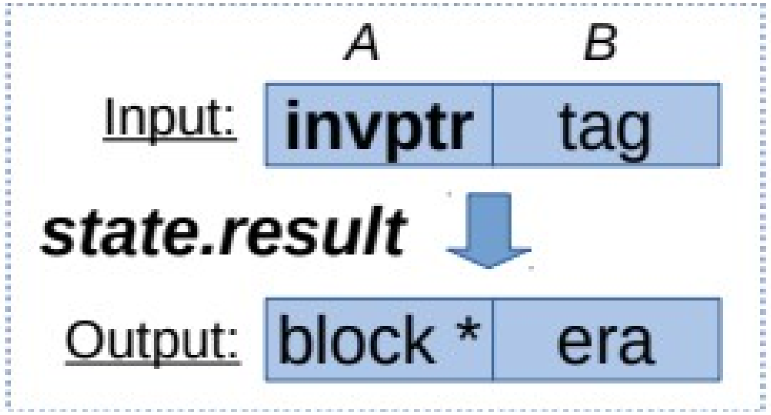
- F&A (fetch-and-add) and SWAP: available on x86-64 and AArch64 as of v8.1 and suitable for wait-free algorithms due to bounded execution time
- WCAS (wide CAS): also available on x86-64 and AArch64
- Both instructions help to solve wait-free consensus

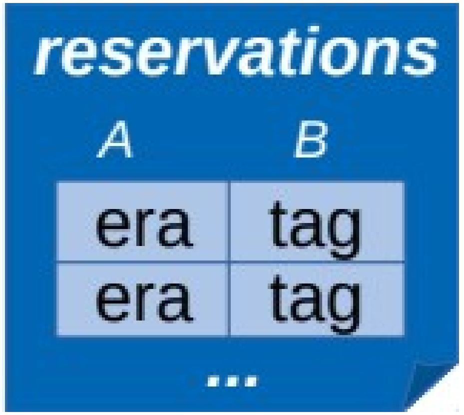
Crystalline-LW

- The first problem with **retire**
  - When “traversing” retirement lists, i.e., dereferencing a thread from each batch that appears in its retirement list, **next** pointers in the corresponding list are ***tainted*** with SWAP
  - **retire** attaches new objects with SWAP rather than a CAS loop
    - If the **next** field of the new object is intact, the old list is attached as a tail (using CAS)
    - If the **next** field of the new object is tainted, **retire** traverses the “docked tail” (i.e., the old list) on behalf of the thread that tainted **next**
    - Some corner cases exist but are handled in wait-free fashion

Crystalline-W

- The second problem with **protect**
  - Adopts a mechanism similar to that of **Wait-Free Eras** [PPoPP'20]
    - The fast-path-slow-path approach to coordinate global era clock increments
    - Helping other threads before incrementing the era clock
  - Tags identify slow-path cycles
  - Per-thread state: result is used for both input and output
    - Use pairs for result { .A, .B }
  - Reservations also use pairs { .A, .B }





Crystalline-W

- Despite similarities, **Crystalline-W** diverges from Wait-Free Eras substantially
  - Cannot rescan retirement lists multiple times due to asynchronous reclamation
  - Uses *special* tricks: odd and even tags, an array of parent objects, “terminal” nodes in the retirement lists, etc.

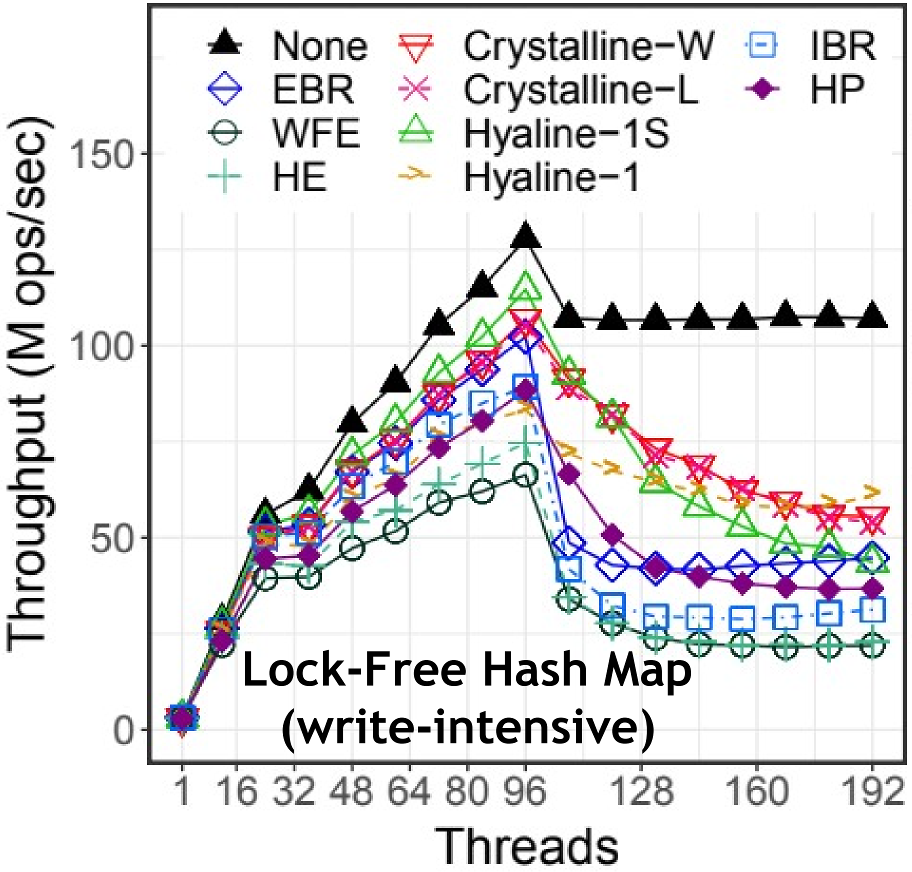
Evaluation

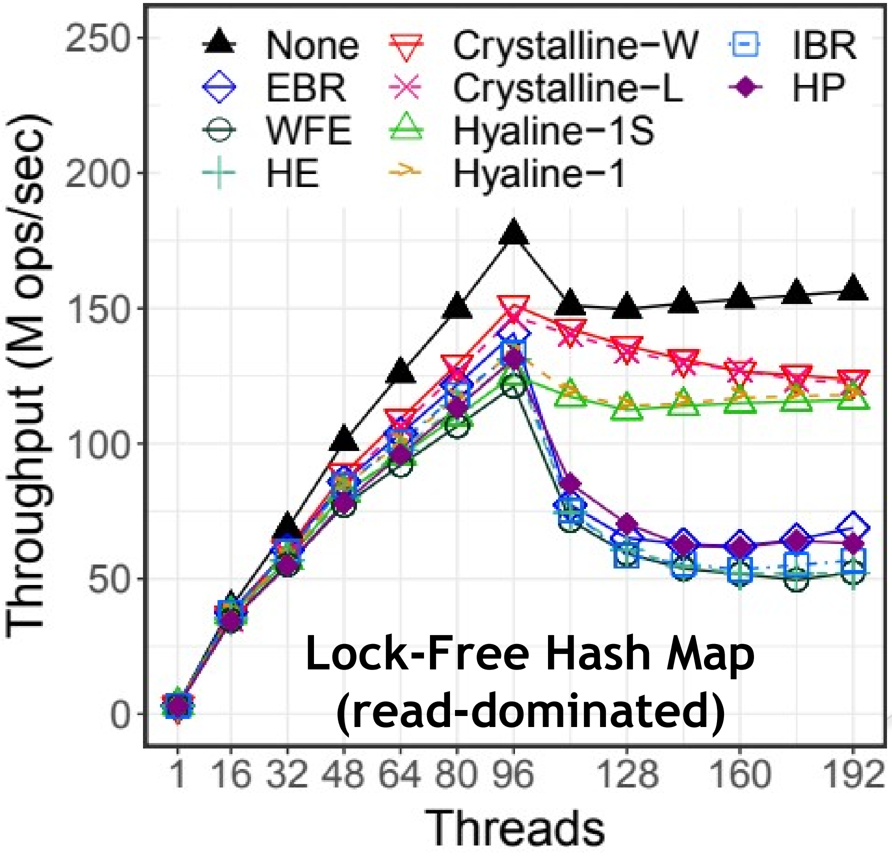
- 4 x Intel Xeon E7-8890 v4 2.20 GHz CPUs (96 cores), 256GB of RAM
- Results are for write-intensive (50% insert, 50% delete) and read-dominated (90% get, 10% put) tests
- More data structures (wait-free queue, lock-free linked list) are in the paper

None: no reclamation (leak memory)
HP: Hazard Pointers [TPDS’04]
HE: Hazard Eras [SPAA’17]
IBR: 2GE Interval-Based Reclamation [PPoPP’18]
WFE: Wait-Free Eras [PPoPP’20]
Hyaline: Hyaline-1 and Hyaline-1S [PLDI’21]
EBR: Epoch-Based Reclamation

Evaluation

None: no reclamation (leak memory)
HP: Hazard Pointers [TPDS'04]
HE: Hazard Eras [SPAA'17]
IBR: 2GE Interval-Based Reclamation [PPoPP'18]
WFE: Wait-Free Eras [PPoPP'20]
Hyaline: Hyaline-1 and Hyaline-1S [PLDI'21]
EBR: Epoch-Based Reclamation

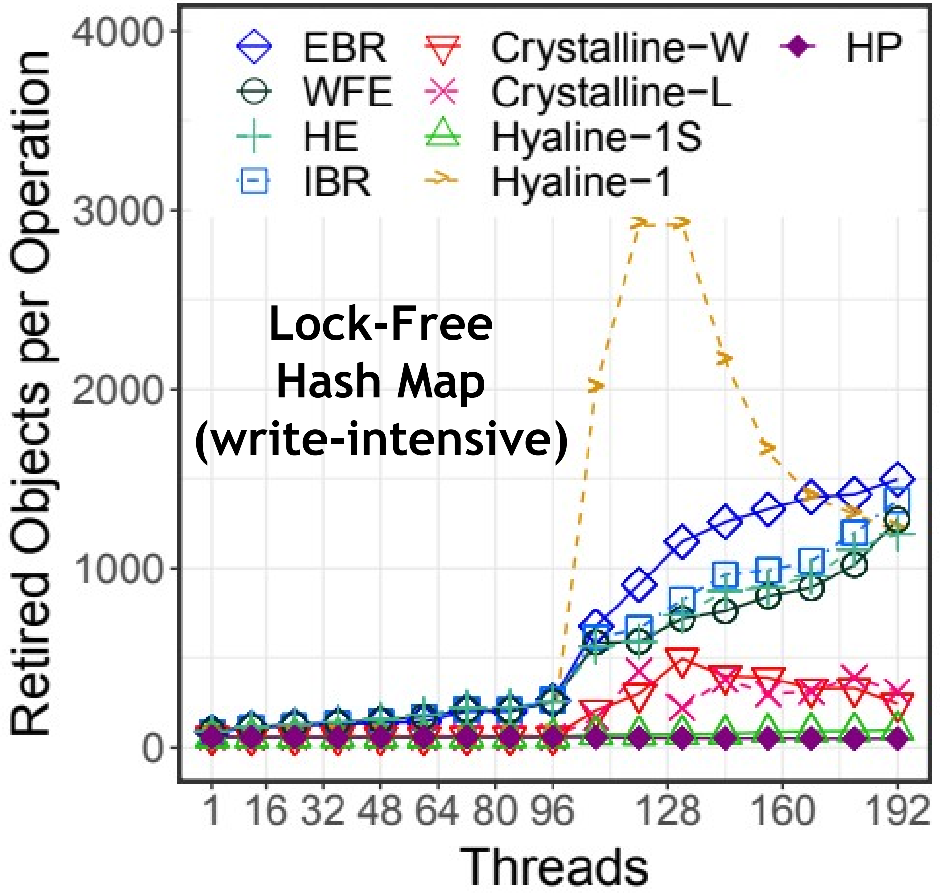


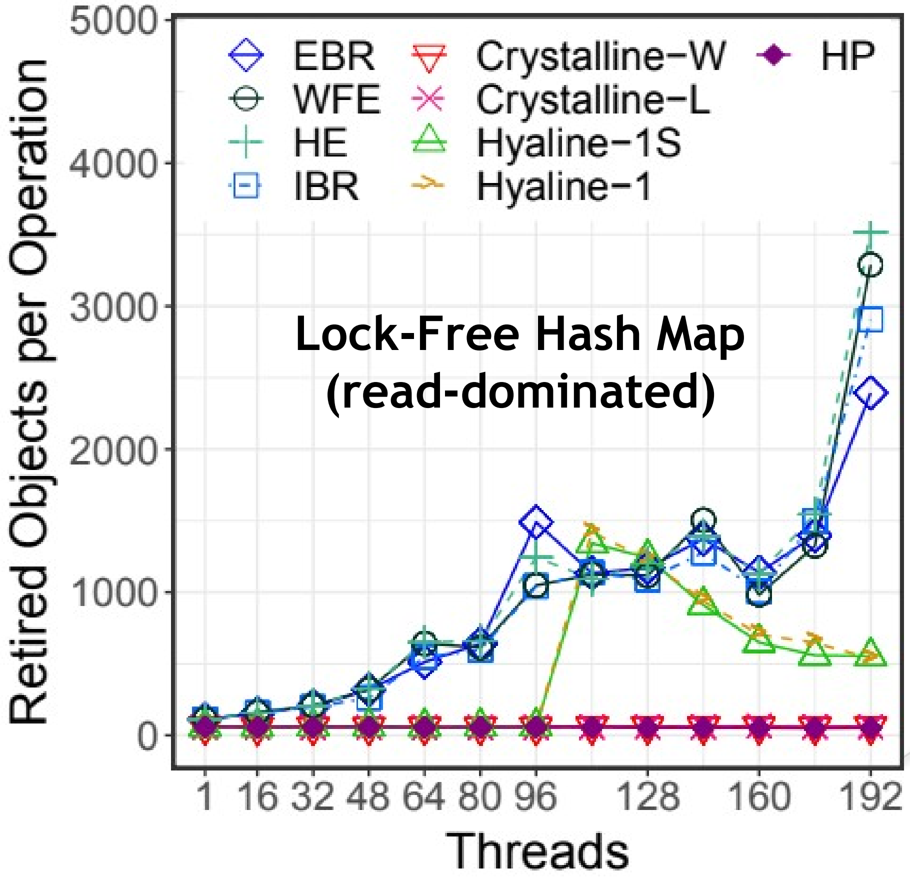


4 x Intel Xeon E7-8890 v4 2.20 GHz CPUs (96 cores), 256GB of RAM

Evaluation

None: no reclamation (leak memory)
HP: Hazard Pointers [TPDS'04]
HE: Hazard Eras [SPAA'17]
IBR: 2GE Interval-Based Reclamation [PPoPP'18]
WFE: Wait-Free Eras [PPoPP'20]
Hyaline: Hyaline-1 and Hyaline-1S [PLDI'21]
EBR: Epoch-Based Reclamation

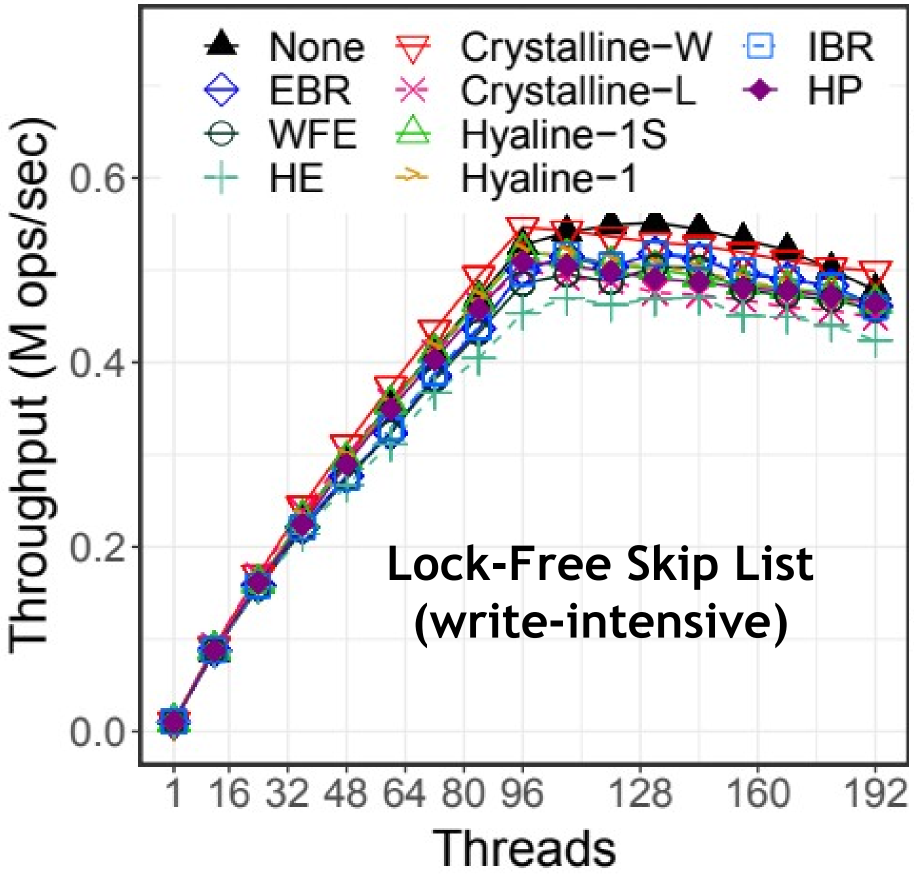


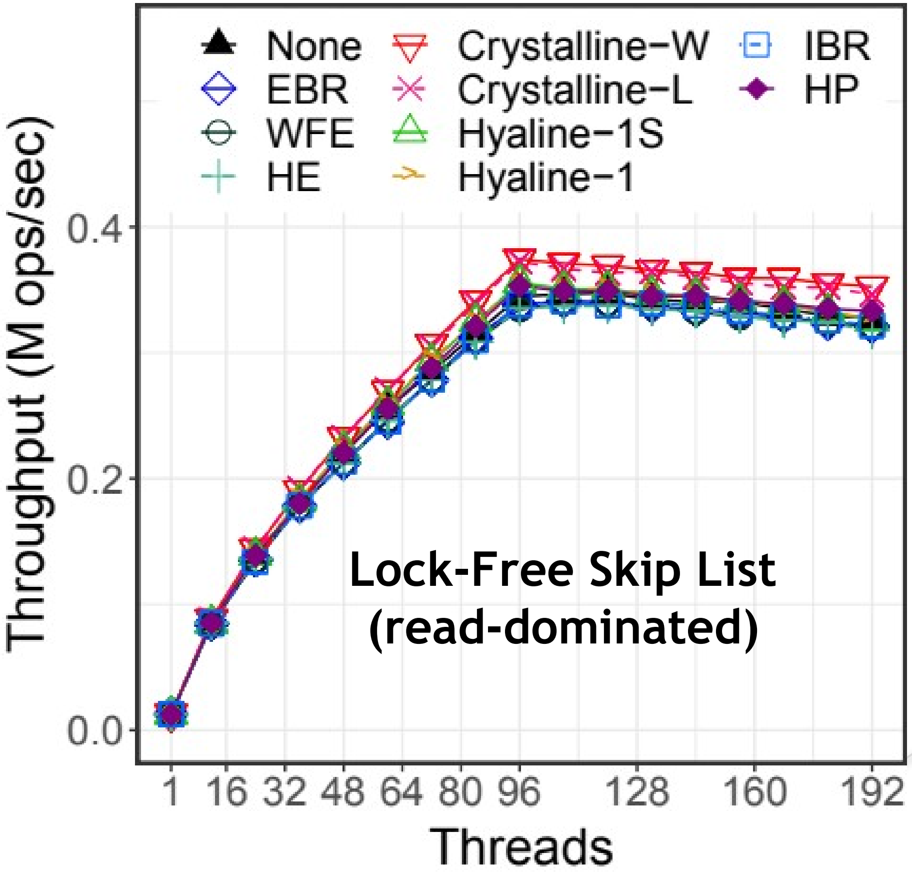


4 x Intel Xeon E7-8890 v4 2.20 GHz CPUs (96 cores), 256GB of RAM

Evaluation

None: no reclamation (leak memory)
HP: Hazard Pointers [TPDS’04]
HE: Hazard Eras [SPAA’17]
IBR: 2GE Interval-Based Reclamation [PPoPP’18]
WFE: Wait-Free Eras [PPoPP’20]
Hyaline: Hyaline-1 and Hyaline-1S [PLDI’21]
EBR: Epoch-Based Reclamation

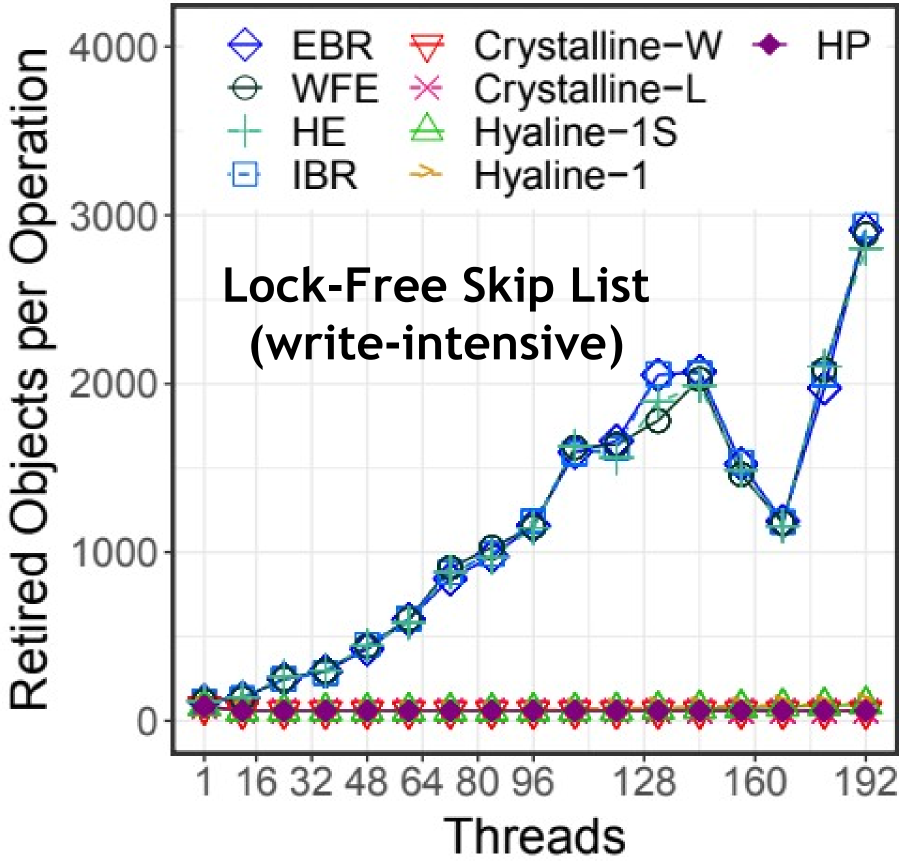


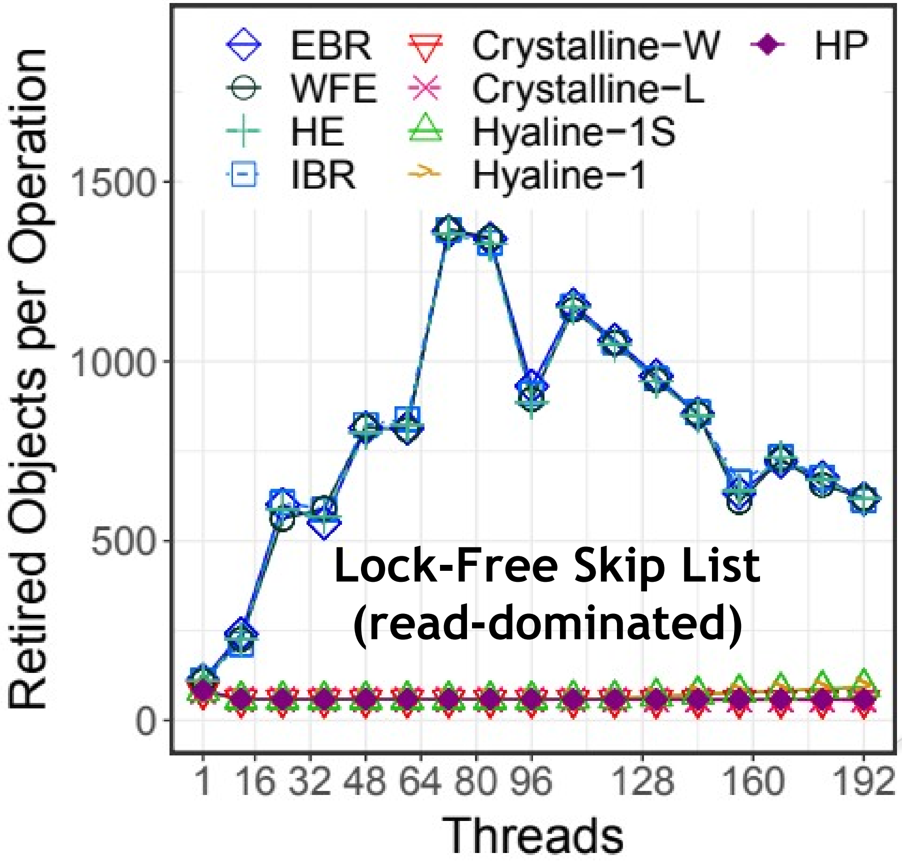


4 x Intel Xeon E7-8890 v4 2.20 GHz CPUs (96 cores), 256GB of RAM

Evaluation

None: no reclamation (leak memory)
HP: Hazard Pointers [TPDS'04]
HE: Hazard Eras [SPAA'17]
IBR: 2GE Interval-Based Reclamation [PPoPP'18]
WFE: Wait-Free Eras [PPoPP'20]
Hyaline: Hyaline-1 and Hyaline-1S [PLDI'21]
EBR: Epoch-Based Reclamation





4 x Intel Xeon E7-8890 v4 2.20 GHz CPUs (96 cores), 256GB of RAM

Availability

- Code is open-source and available at:
  - https://github.com/rusnikola/wfsmr

THANK YOU!

This work is supported in part by the startup fund (Pennsylvania State University) as well as ONR under grants N00014-18-1-2022, N00014-19-1-2493, and N00014-21-1-2523, and AFOSR under grant FA9550-16-1-0371 (Virginia Tech)

Artwork attribution:
wikipedia.org (Intel, AMD64, ARM), intel.com (Xeon logo), the ONR and AFOSR websites (respective logos)