SNAPSHOT-FREE, TRANSPARENT, AND ROBUST MEMORY RECLAMATION FOR LOCK-FREE DATA STRUCTURES

Ruslan Nikolaev and **Binoy Ravindran**

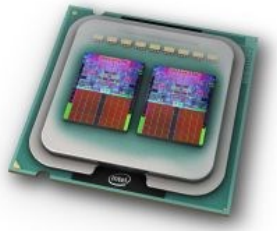
rnikola@vt.edu **binoy@vt.edu**

Systems Software Research Group

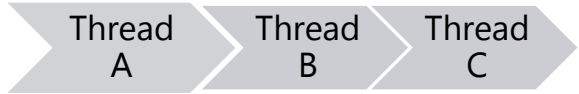
Virginia Tech, USA

CONCURRENT DATA STRUCTURES

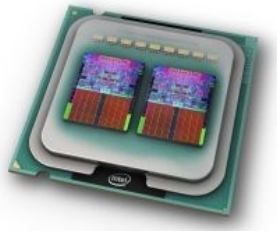
- Many-core systems today require efficient access to da
  - Concurrent data structures



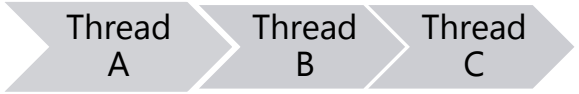
- Multiple threads need to ***safely*** manipulate data structures (similar to sequential data structures)
  - "nothing bad will happen"



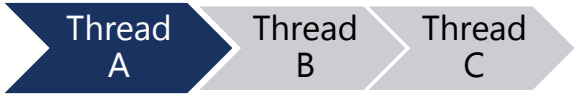
CONCURRENT DATA STRUCTURES



- Many-core systems today require efficient access to da
  - Concurrent data structures
- Multiple threads need to ***safely*** manipulate data structures (similar to sequential data structures)
  - "nothing bad will happen"

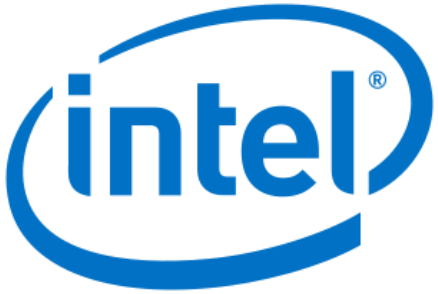


- Concurrency also adds a ***liveness*** property, which stipulates how threads will be able to make progress
  - "something good will happen eventually"

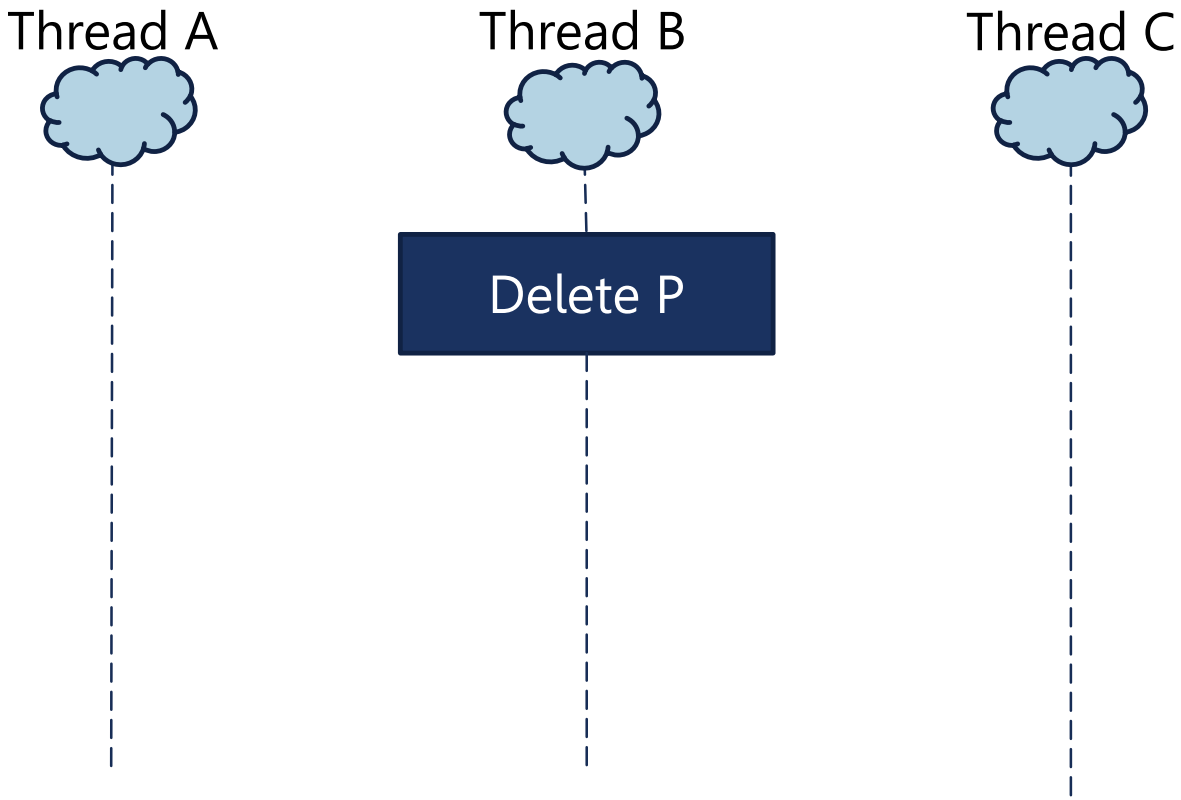


NON-BLOCKING ALGORITHMS

- ***Lock-freedom***: a common type of non-blocking progress
  - At least **one** thread always makes progress in a finite number of steps
- **CAS** (compare-and-swap) is used universally in non-blocking algorithms
  - **Double-width CAS** which modifies two **adjacent** words is available on x86-64 and ARM64
  - **LL/SC** (load-link/store-conditional) which is more versatile is available on PowerPC and ARM64
  - **F&A** (fetch-and-add) is also often available as a specialized instruction to improve performance

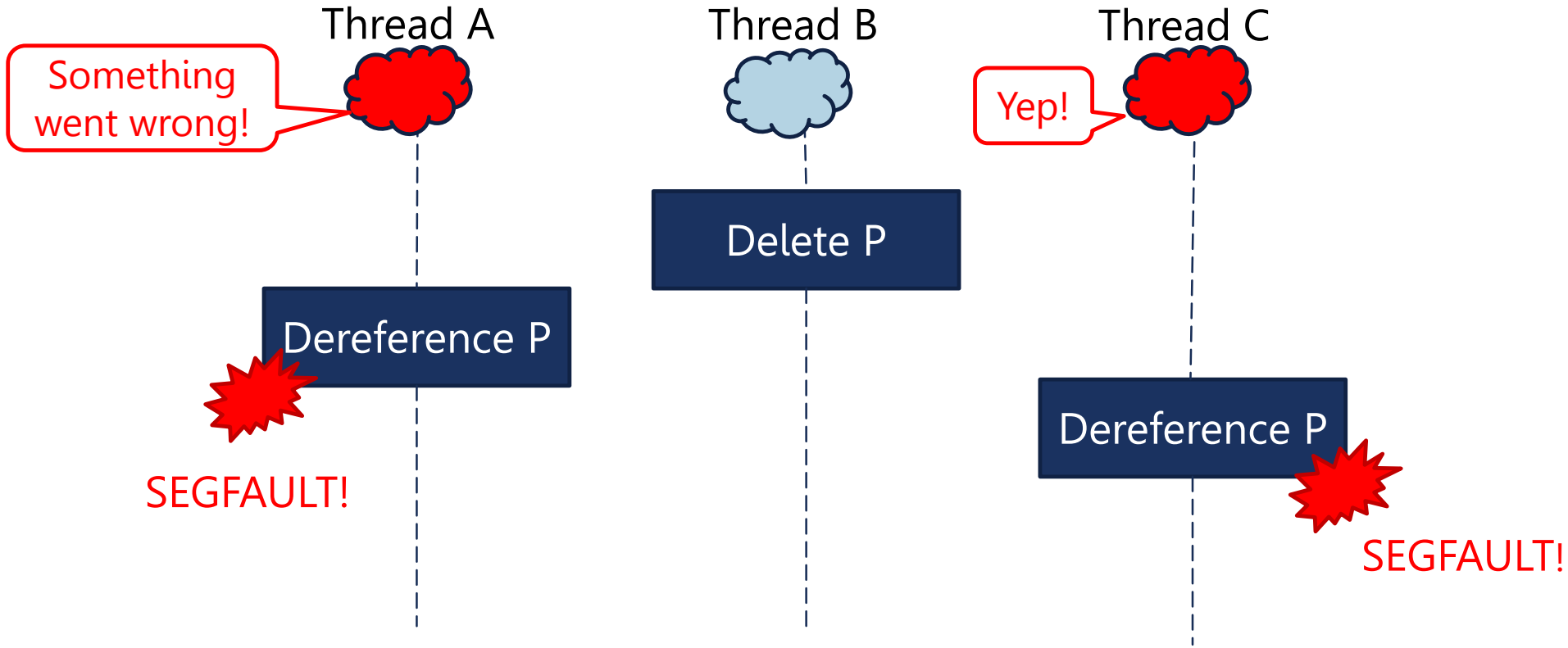


MEMORY RECLAMATION PROBLEM



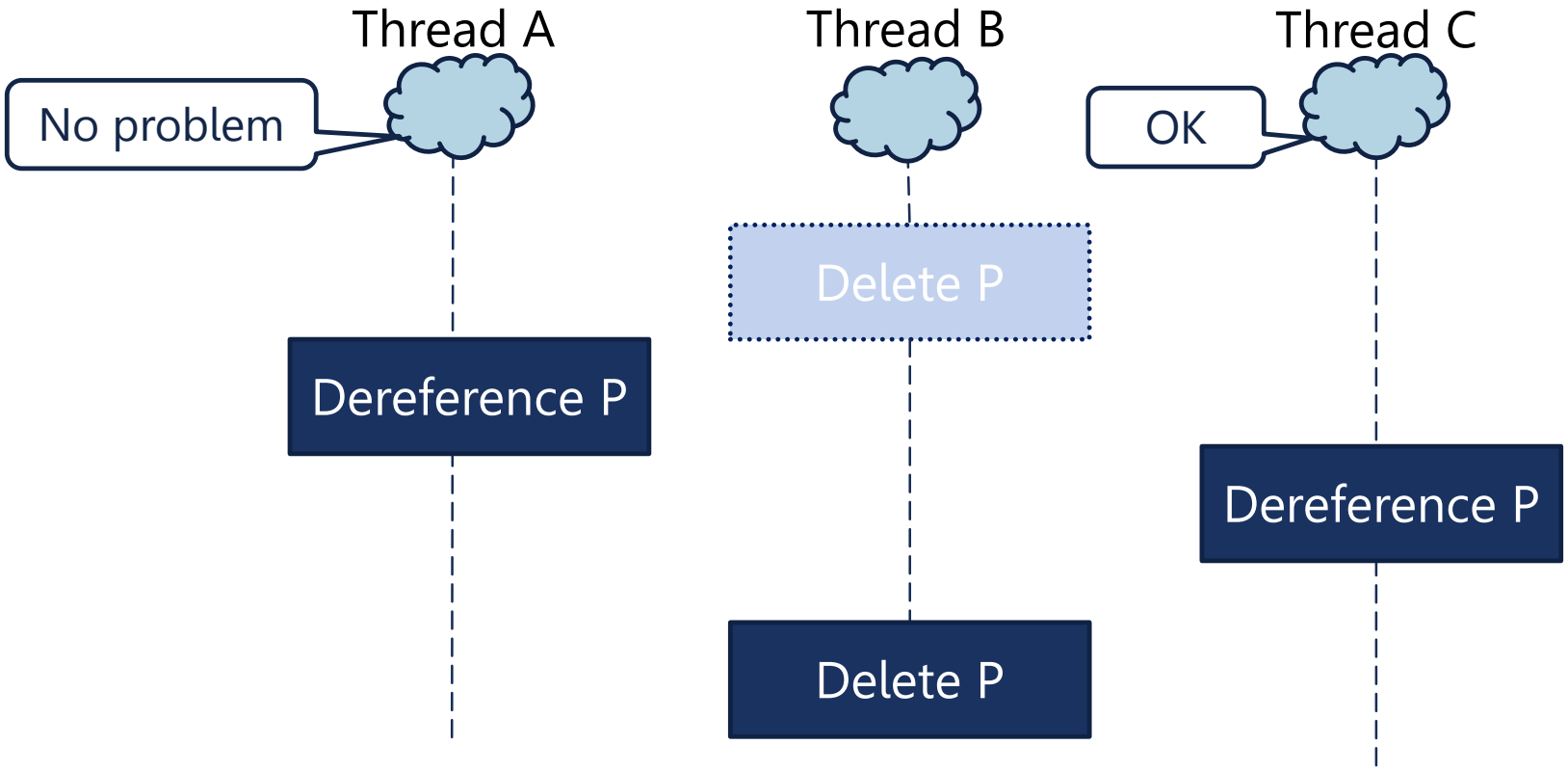
One thread wants to de-allocate a memory object which is still reachable by concurrent threads

MEMORY RECLAMATION PROBLEM



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

MEMORY RECLAMATION PROBLEM



Postpone de-allocation until it is safe to do so

MEMORY RECLAMATION PROBLEM

- Concurrent programming is hard
  - Non-blocking (lock-free) data structures require special treatment of deleted memory objects
  - Garbage collectors are often impractical in C/C++ and lack suitable progress/performance properties

- Desirable properties for memory reclamation
  - ***Non-blocking progress***: avoid using locks
  - ***Robustness***: bounding memory usage even when threads are stalled or preempted
  - ***Transparency***: avoiding implicit assumptions about threads; threads can be created/deleted dynamically
  - ***Snapshot-freedom***: not taking snapshots of the global state to alleviate contention

HYALINE: API

- Memory reclamation must be explicitly embedded into the code

```
handle_t Handle = enter();
// deref is only for robust versions
List = deref(&LinkedList);
Node = deref(&List->Next);
retire(Node); // Mark for deletion
// Do something else...
leave(Handle);
```

HYALINE: API

- Memory reclamation must be explicitly embedded into the code

```
handle_t Handle = enter();
// deref is only for robust versions
List = deref(&LinkedList);
Node = deref(&List->Next);
retire(Node); // Mark for deletion
// Do something else...
leave(Handle);
```

HYALINE: API

- Memory reclamation must be explicitly embedded into the code

```
handle_t Handle = enter();
// deref is only for robust versions
List = deref(&LinkedList);
Node = deref(&List->Next);
retire(Node); // Mark for deletion
// Do something else...
leave(Handle);
```

HYALINE: API

- Memory reclamation must be explicitly embedded into the code

```
handle_t Handle = enter();
// deref is only for robust versions
List = deref(&LinkedList);
Node = deref(&List->Next);
retire(Node); // Mark for deletion
// Do something else...
leave(Handle);
```

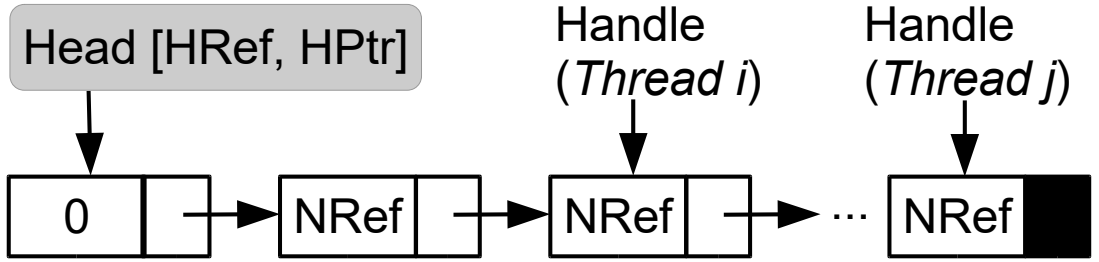
HYALINE: API

- Memory reclamation must be explicitly embedded into the code

```
handle_t Handle = enter();
// deref is only for robust versions
List = deref(&LinkedList);
Node = deref(&List->Next);
retire(Node); // Mark for deletion
// Do something else...
leave(Handle);
```

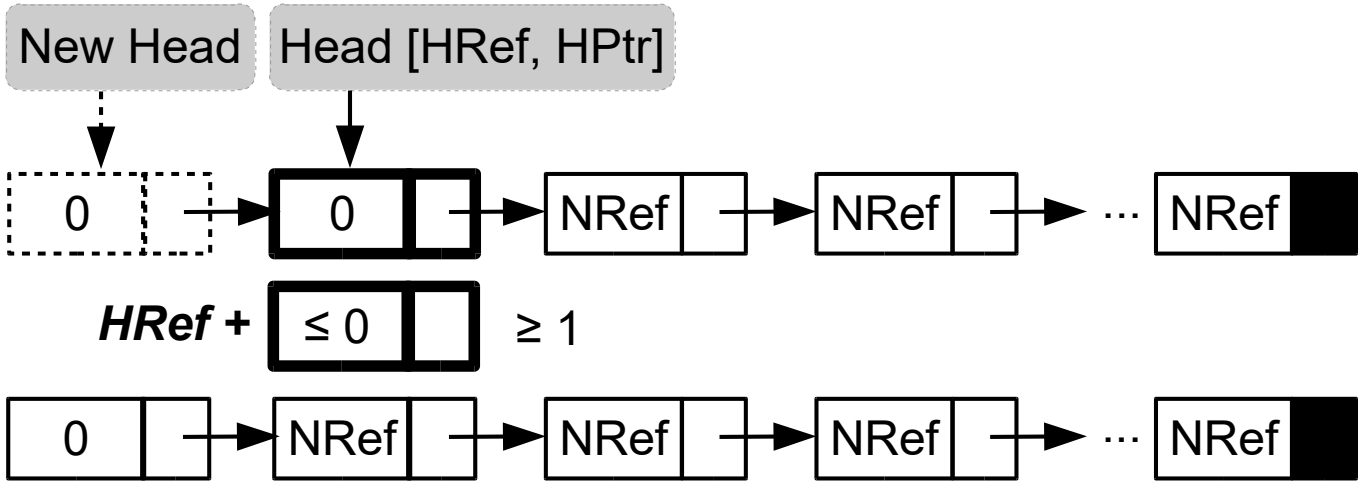
HYALINE: SINGLE LIST

- The main idea
  - Use ***special*** reference counting, which is triggered only when retiring objects
  - Retired objects are appended to a global list
  - The ***handle*** points to the part of the list when the thread ***entered*** its operation



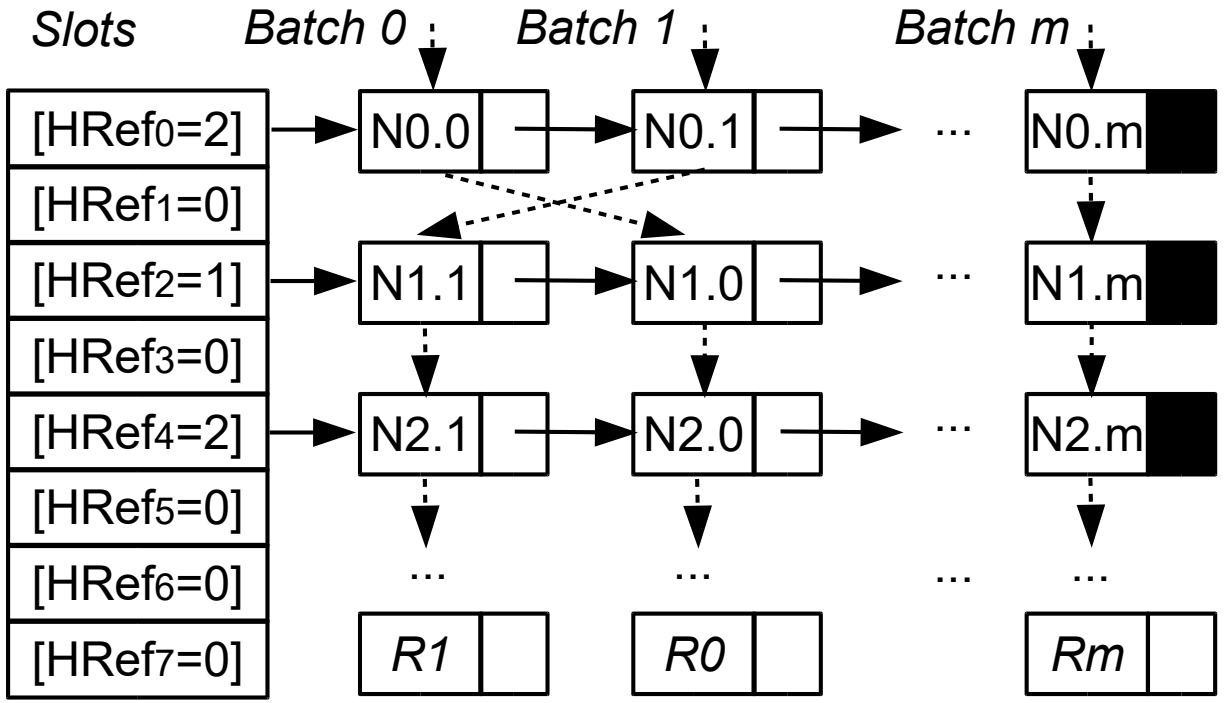
HYALINE: SINGLE LIST

- The main idea
  - Update Head's reference counter (HRef) when ***entering*** and ***leaving*** thread operations
  - When ***leaving***, a thread traverses a sublist from the beginning to the object pointed to by a ***handle***
  - Propagate counters when ***retiring*** objects
    - Treat the very first list element specially: HRef rather than NRef reflects its reference counter
    - When appending to the list, adjust the predecessor's NRef (previously 0) with the HRef value



HYALINE: MULTIPLE LISTS

- The main idea
  - Maintain multiple global lists of retired objects to alleviate contention
    - Each list is used by a subset of threads
    - Retire an entire ***batch*** of objects
    - One reference counter for the entire batch



COMPARISON

Scheme	Performance	Snapshot-Free	Robust	Transparent	Extra Memory	API complexity
Reference Counting	Very Slow	Yes	Yes	Partially (swap)	Double each pointer	Harder, Intrusive
Epoch Based Reclamation	Fast	Yes	No	No (deletion)	1 word	Very Easy
Hazard Pointers	Slow	No	Yes	No (deletion)	1 word	Harder
Hazard Eras	Medium	No	Yes	No (deletion)	3 words	Harder
Interval Based Reclamation	Fast	No	Yes	No (deletion)	3 words	Medium
Hyaline	Fast	Yes	No	Yes	3 words	Very Easy
Hyaline-1	Fast	Yes	No	Almost	3 words	Very Easy
Hyaline-S	Fast	Yes	Yes	Yes	3 words	Medium
Hyaline-1S	Fast	Yes	Yes	Almost	3 words	Medium

COMPARISON

Scheme	Performance	Snapshot-Free	Robust	Transparent	Extra Memory	API complexity
Reference Counting	Very Slow	Yes	Yes	Partially (swap)	Double each pointer	Harder, Intrusive
Epoch Based Reclamation	Fast	Yes	No	No (deletion)	1 word	Very Easy
Hazard Pointers	Slow	No	Yes	No (deletion)	1 word	Harder
Hazard Eras	Medium	No	Yes	No (deletion)	3 words	Harder
Interval Based Reclamation	Fast	No	Yes	No (deletion)	3 words	Medium
Hyaline	Fast	Yes	No	Yes	3 words	Very Easy
Hyaline-1	Fast	Yes	No	Almost	3 words	Very Easy
Hyaline-S	Fast	Yes	Yes	Yes	3 words	Medium
Hyaline-1S	Fast	Yes	Yes	Almost	3 words	Medium

COMPARISON

Scheme	Performance	Snapshot-Free	Robust	Transparent	Extra Memory	API complexity
Reference Counting	Very Slow	Yes	Yes	Partially (swap)	Double each pointer	Harder, Intrusive
Epoch Based Reclamation	Fast	Yes	No	No (deletion)	1 word	Very Easy
Hazard Pointers	Slow	No	Yes	No (deletion)	1 word	Harder
Hazard Eras	Medium	No	Yes	No (deletion)	3 words	Harder
Interval Based Reclamation	Fast	No	Yes	No (deletion)	3 words	Medium
Hyaline	Fast	Yes	No	Yes	3 words	Very Easy
Hyaline-1	Fast	Yes	No	Almost	3 words	Very Easy
Hyaline-S	Fast	Yes	Yes	Yes	3 words	Medium
Hyaline-1S	Fast	Yes	Yes	Almost	3 words	Medium

COMPARISON

Scheme	Performance	Snapshot-Free	Robust	Transparent	Extra Memory	API complexity
Reference Counting	Very Slow	Yes	Yes	Partially (swap)	Double each pointer	Harder, Intrusive
Epoch Based Reclamation	Fast	Yes	No	No (deletion)	1 word	Very Easy
Hazard Pointers	Slow	No	Yes	No (deletion)	1 word	Harder
Hazard Eras	Medium	No	Yes	No (deletion)	3 words	Harder
Interval Based Reclamation	Fast	No	Yes	No (deletion)	3 words	Medium
Hyaline	Fast	Yes	No	Yes	3 words	Very Easy
Hyaline-1	Fast	Yes	No	Almost	3 words	Very Easy
Hyaline-S	Fast	Yes	Yes	Yes	3 words	Medium
Hyaline-1S	Fast	Yes	Yes	Almost	3 words	Medium

COMPARISON

Scheme	Performance	Snapshot-Free	Robust	Transparent	Extra Memory	API complexity
Reference Counting	Very Slow	Yes	Yes	Partially (swap)	Double each pointer	Harder, Intrusive
Epoch Based Reclamation	Fast	Yes	No	No (deletion)	1 word	Very Easy
Hazard Pointers	Slow	No	Yes	No (deletion)	1 word	Harder
Hazard Eras	Medium	No	Yes	No (deletion)	3 words	Harder
Interval Based Reclamation	Fast	No	Yes	No (deletion)	3 words	Medium
Hyaline	Fast	Yes	No	Yes	3 words	Very Easy
Hyaline-1	Fast	Yes	No	Almost	3 words	Very Easy
Hyaline-S	Fast	Yes	Yes	Yes	3 words	Medium
Hyaline-1S	Fast	Yes	Yes	Almost	3 words	Medium

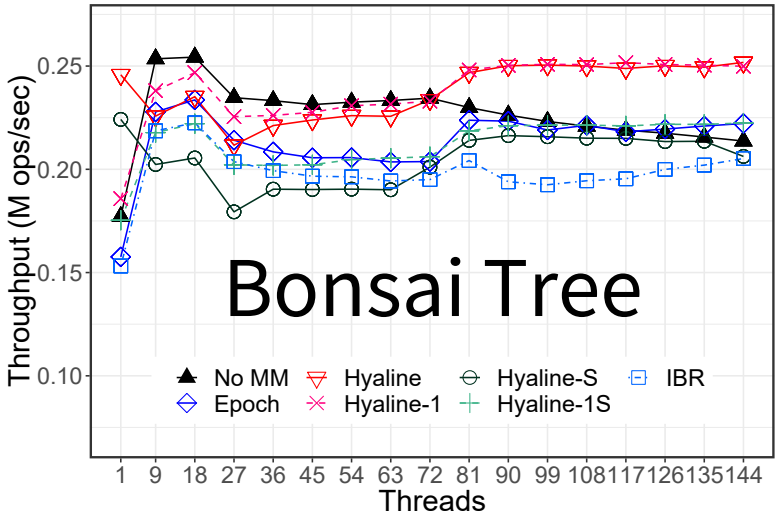
VARIANTS OF HYALINE

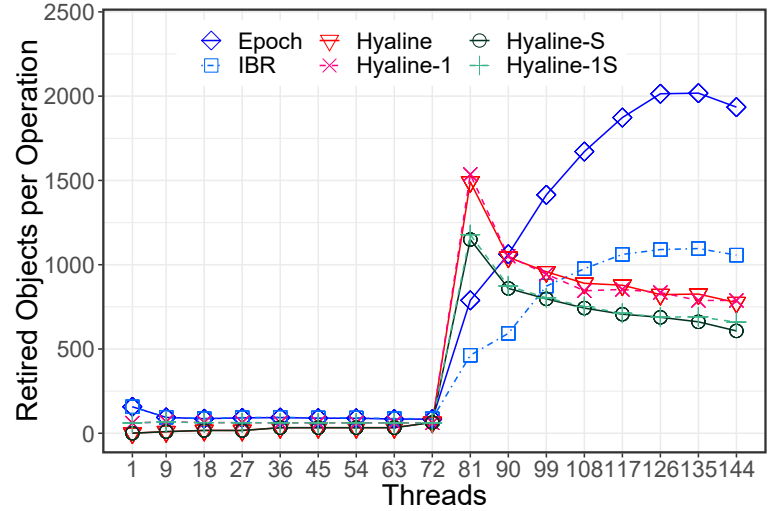
- Hyaline: a non-robust version, uses double-width CAS
- Hyaline-1: a specialized version of Hyaline, each slot is used by just one thread (regular CAS)
- Hyaline-S: a robust version of Hyaline, inspired by “birth eras” from Hazard Eras and Interval Based Reclamation
- Hyaline-1S: a specialized version of Hyaline-1S (regular CAS)

EVALUATION

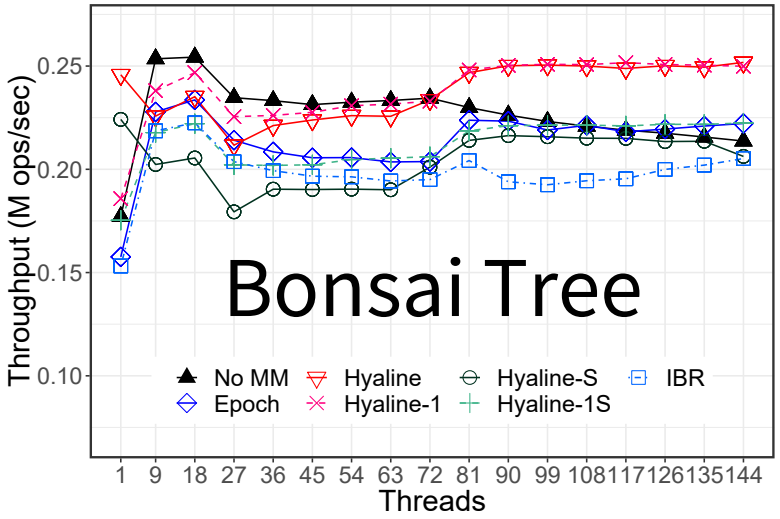
- 4x18 Intel Xeon E7-8880 v3 (2.30GHz) 128GB RAM, Clang 11.0.1 with -O3
- Adopted the benchmark from IBR/PPoPP '18 (by Wen et al.) and compared agai
  - *Epoch-Based Reclamation (**Epoch**)*
  - *Interval-Based Reclamation, 2GEIBR (**IBR**)* [PPoPP '18]
  - *Hazard Eras (**HE**)* [SPAA '17]
  - *Hazard Pointers (**HP**)* [TPDS '04]
  - *No reclamation, i.e., leaking memory (**No MM**)*
- Results are for write-intensive (50% insert, 50% delete) and read-dominated (90% get, 10% put) tests
  - See the paper for more results

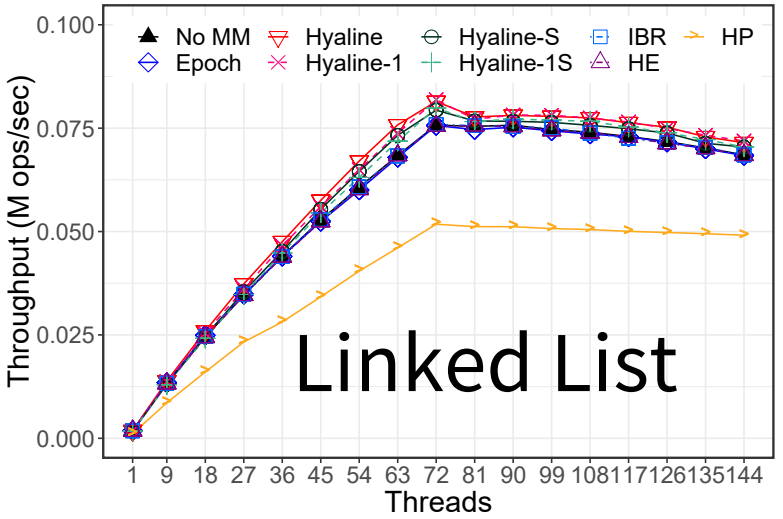
EVALUATION: WRITE-INTENSIVE TESTS

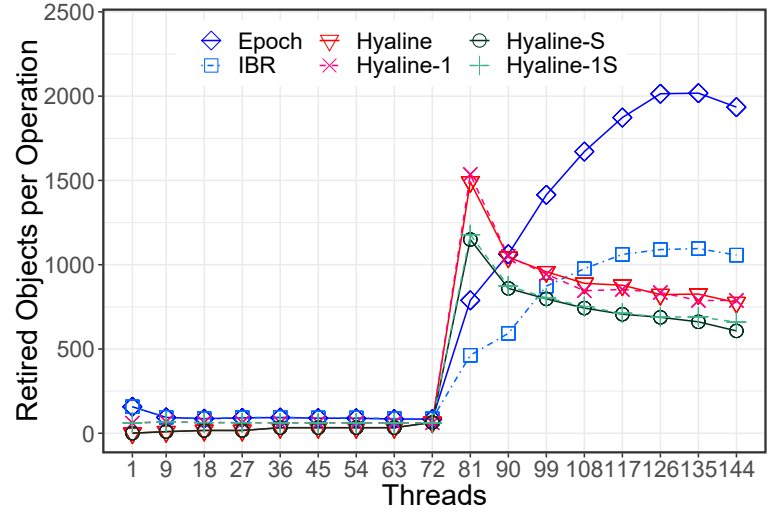


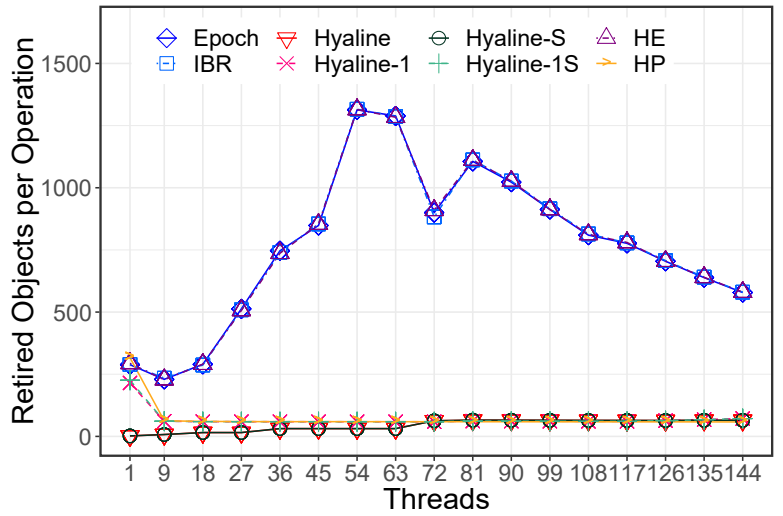


EVALUATION: WRITE-INTENSIVE TESTS

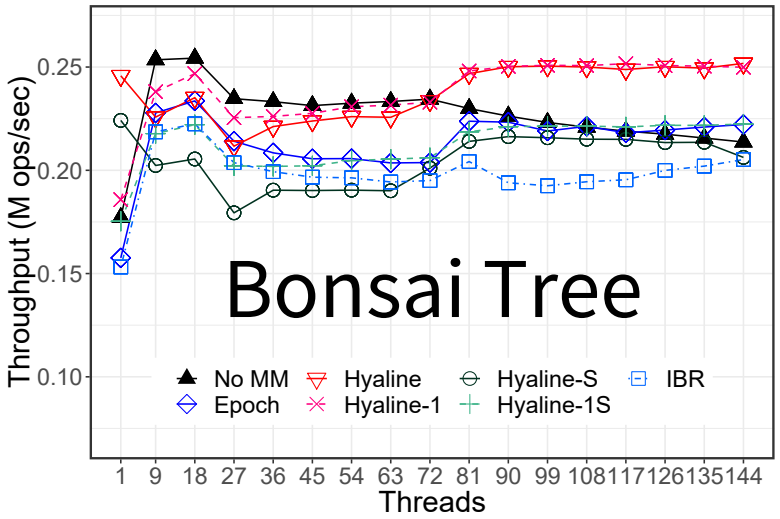


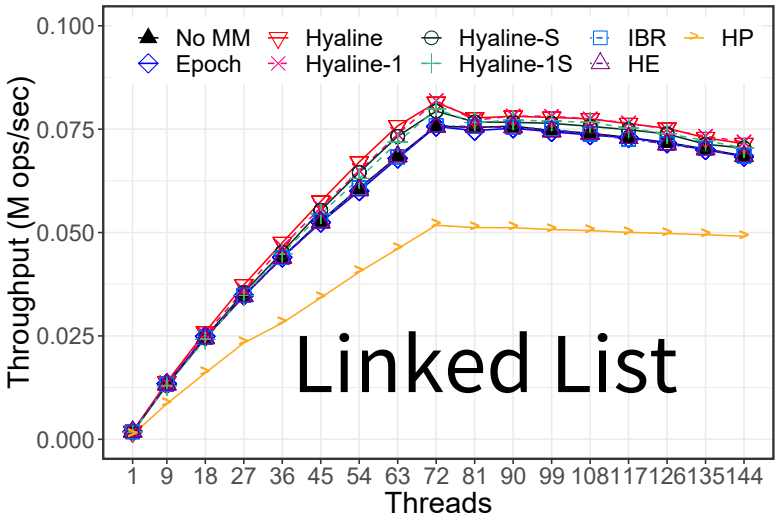


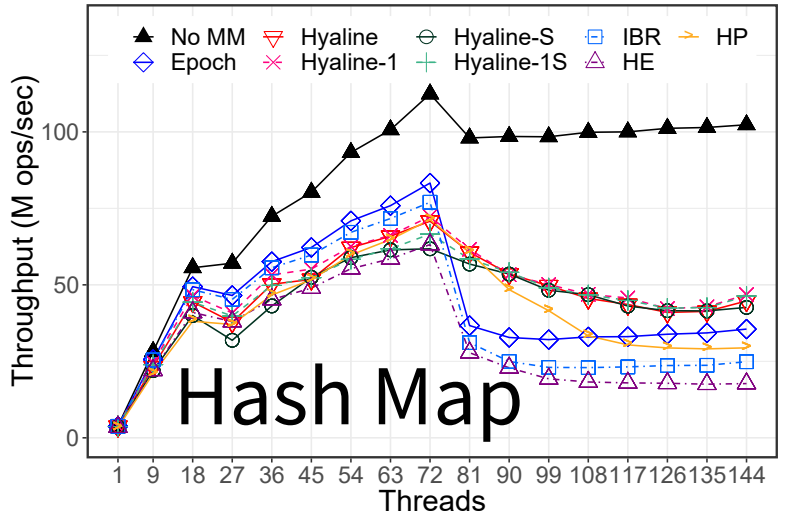


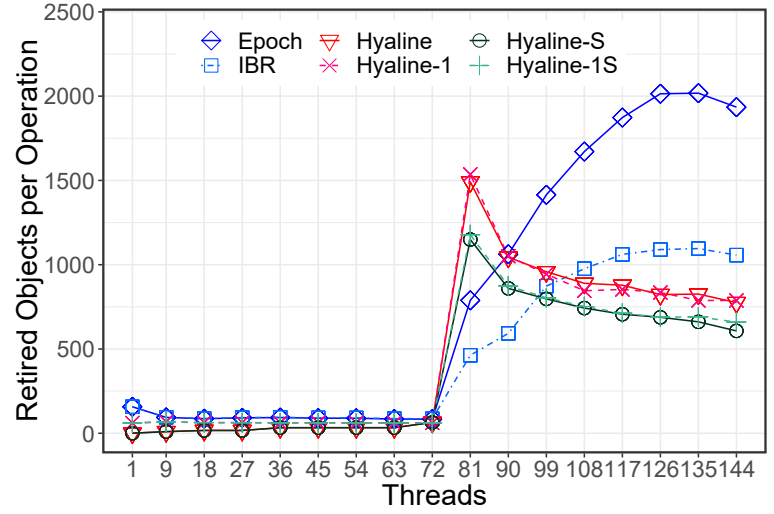


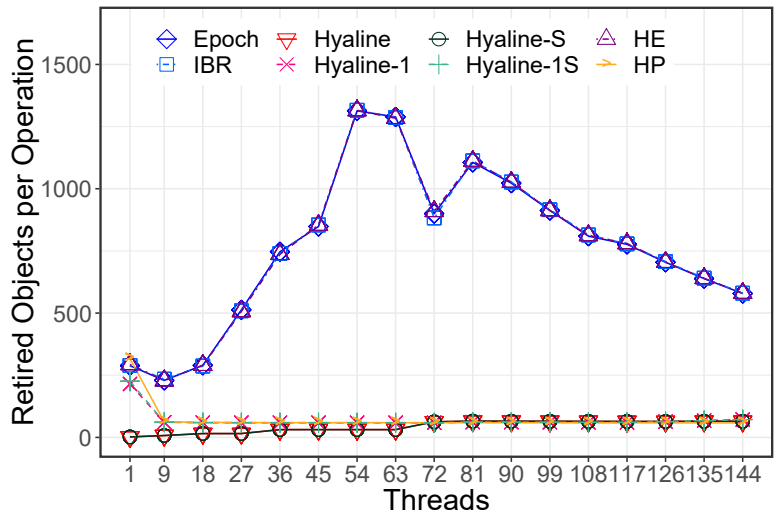
EVALUATION: WRITE-INTENSIVE TESTS

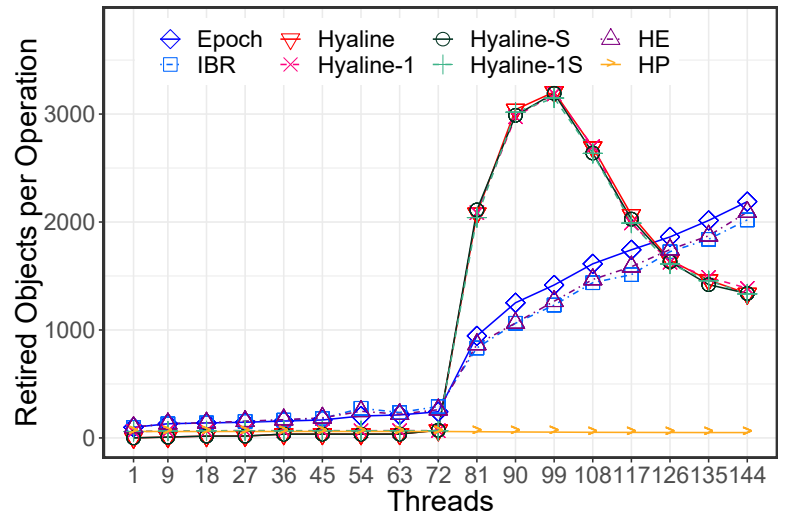




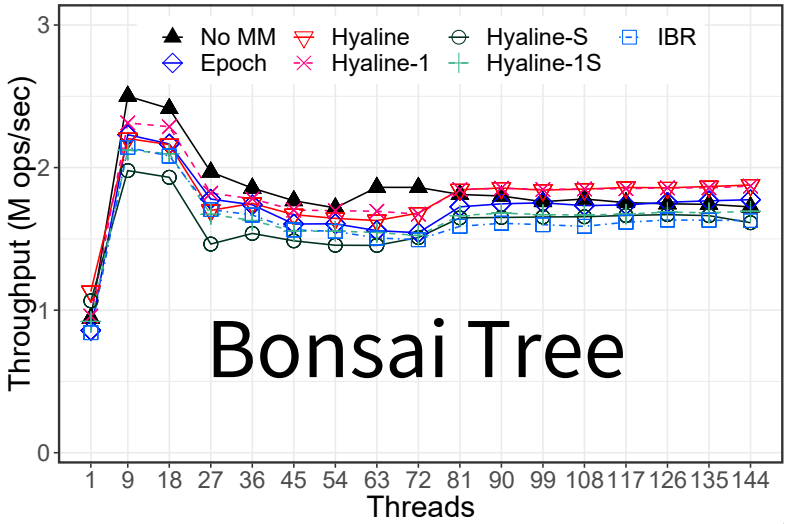


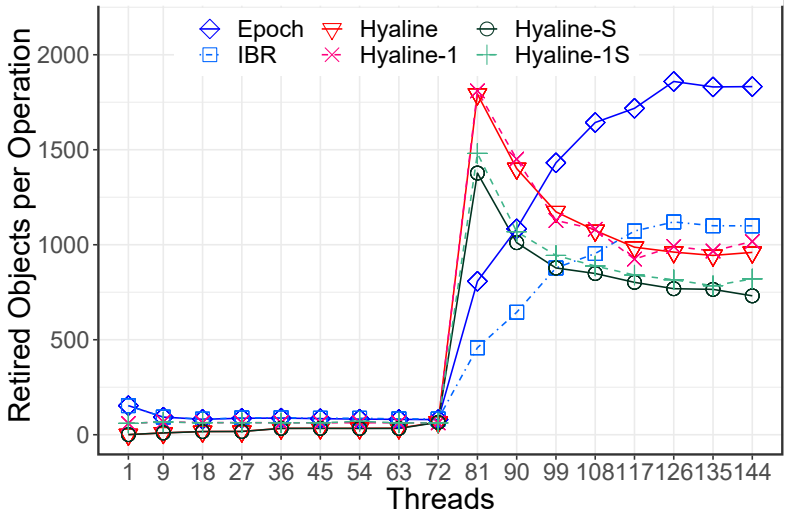




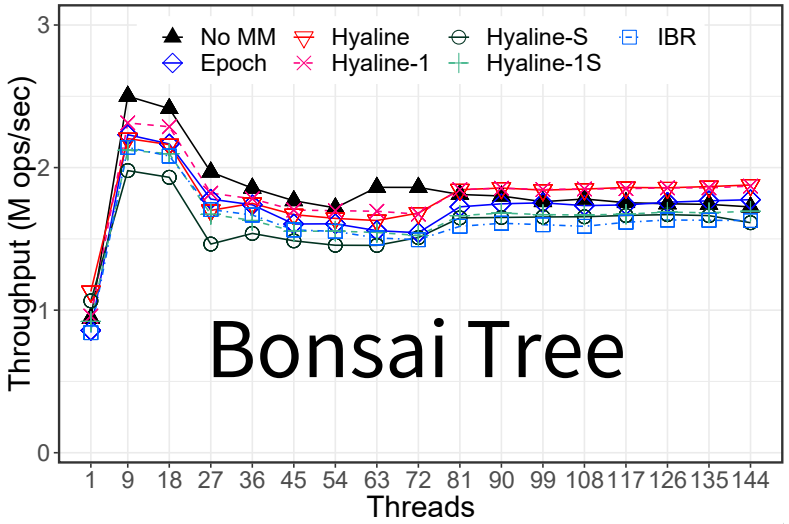


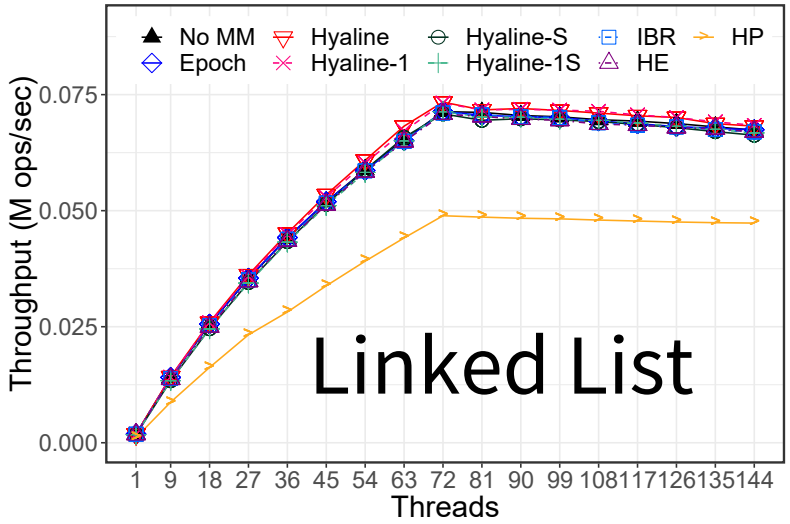
EVALUATION: READ-DOMINATED TESTS

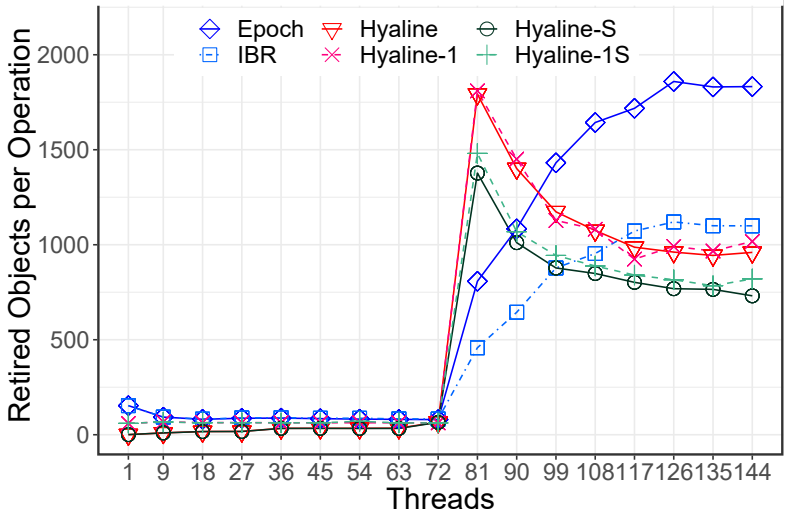


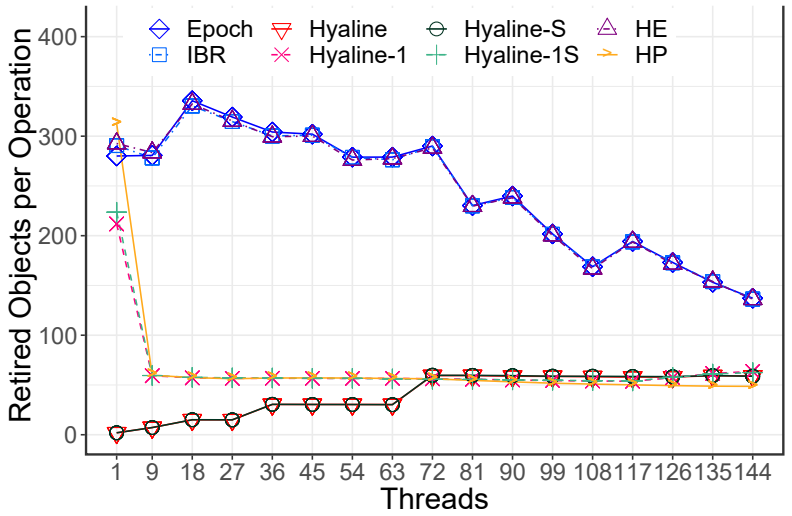


EVALUATION: READ-DOMINATED TESTS

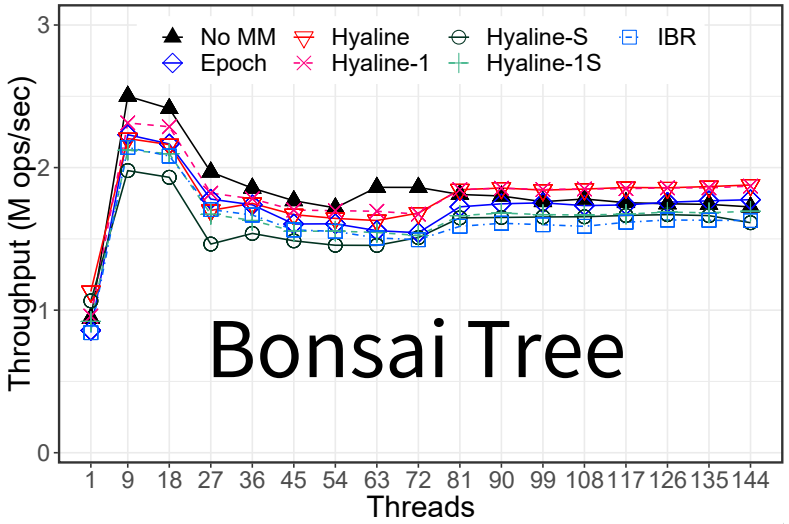


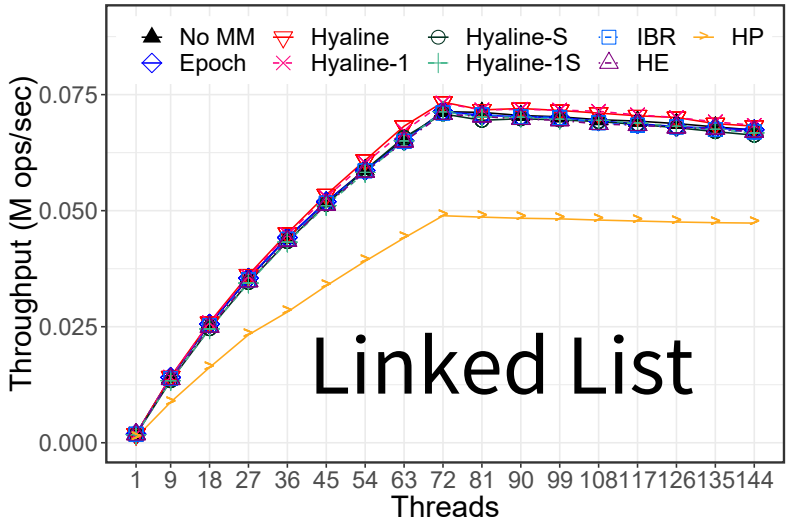


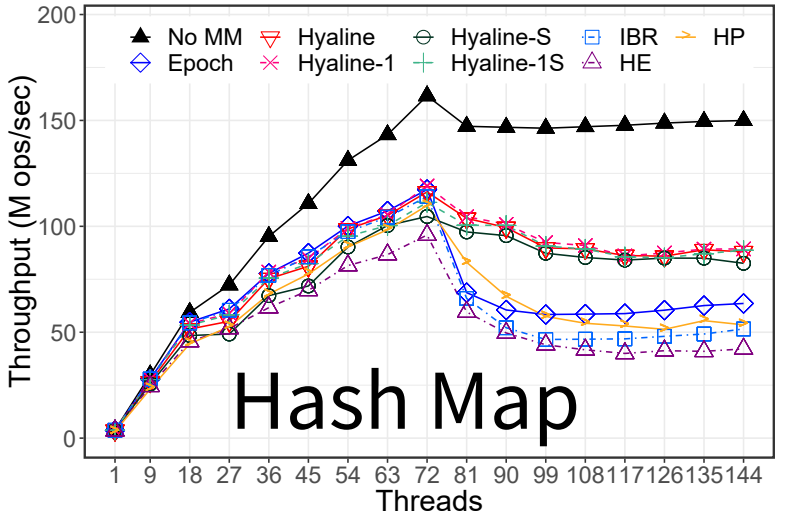


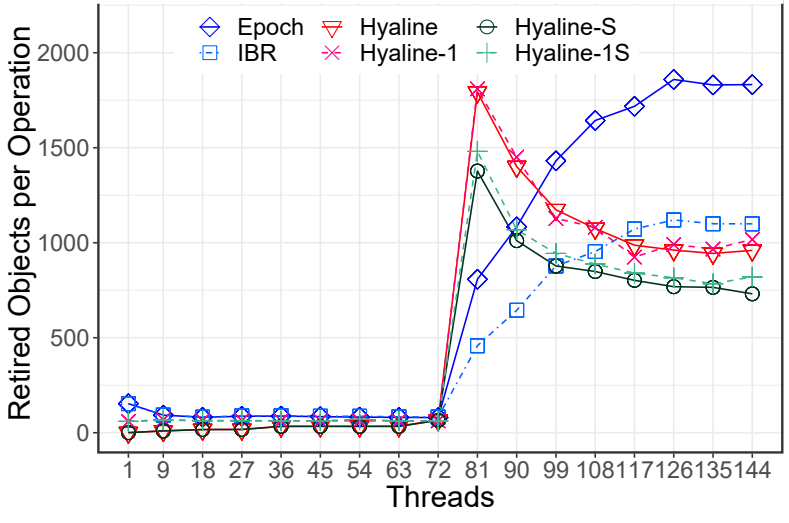


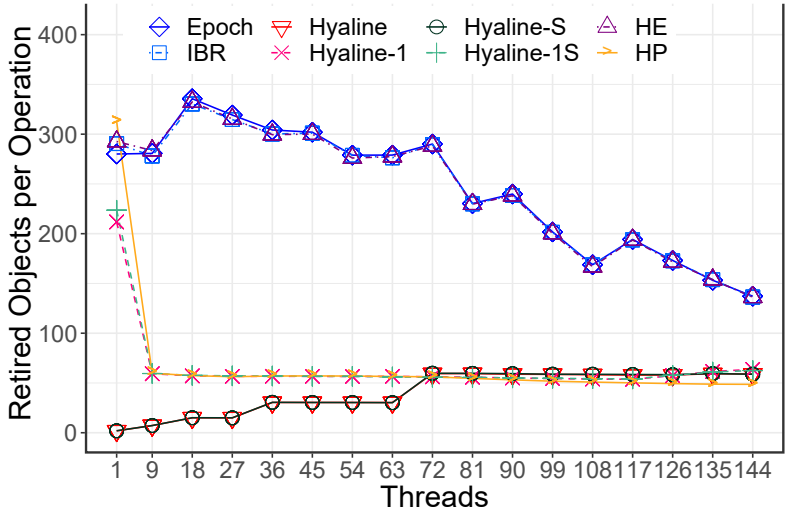
EVALUATION: READ-DOMINATED TESTS

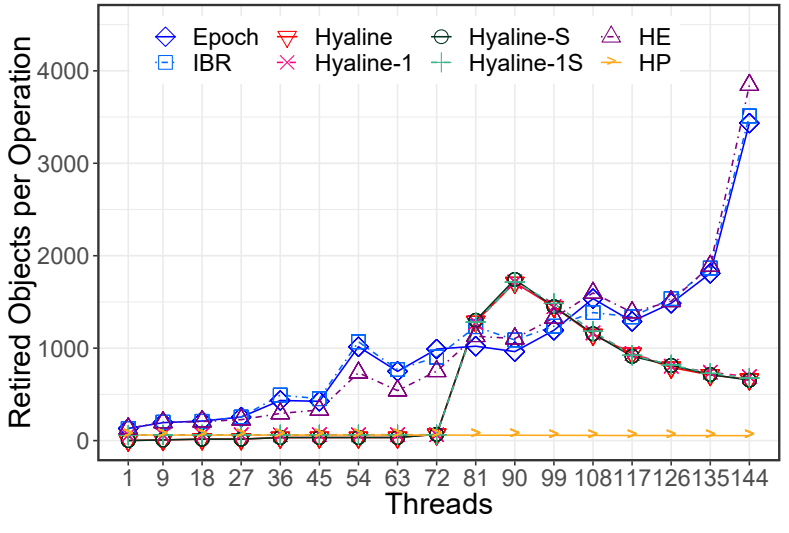












AVAILABILITY

- Hyaline's code and the benchmark are open-source and available at
  - https://github.com/rusnikola/lfsmr
- Additional paper appendices (LL/SC and optimizations) are available at
  - https://arxiv.org/pdf/1905.07903.pdf

The work is supported by AFOSR under grants FA9550-15-1-0098 and FA9550-16-1-0371, and ONR under grants N00014-18-1-2022 and N00014-19-1-2493

AVAILABILITY

- Hyaline's code and the benchmark are open-source and available at
  - https://github.com/rusnikola/lfsmr
- Additional paper appendices (LL/SC and optimizations) are available at
  - https://arxiv.org/pdf/1905.07903.pdf

THANK YOU!

The work is supported by AFOSR under grants FA9550-15-1-0098 and FA9550-16-1-0371, and ONR under grants N00014-18-1-2022 and N00014-19-1-2493

Artwork attribution: *wikipedia.org (Intel, AMD64, ARM, PowerPC logos), intel.com (Xeon logo), techradar.com (multi-core chip), the ONR and AFOSR websites (respective logos)*