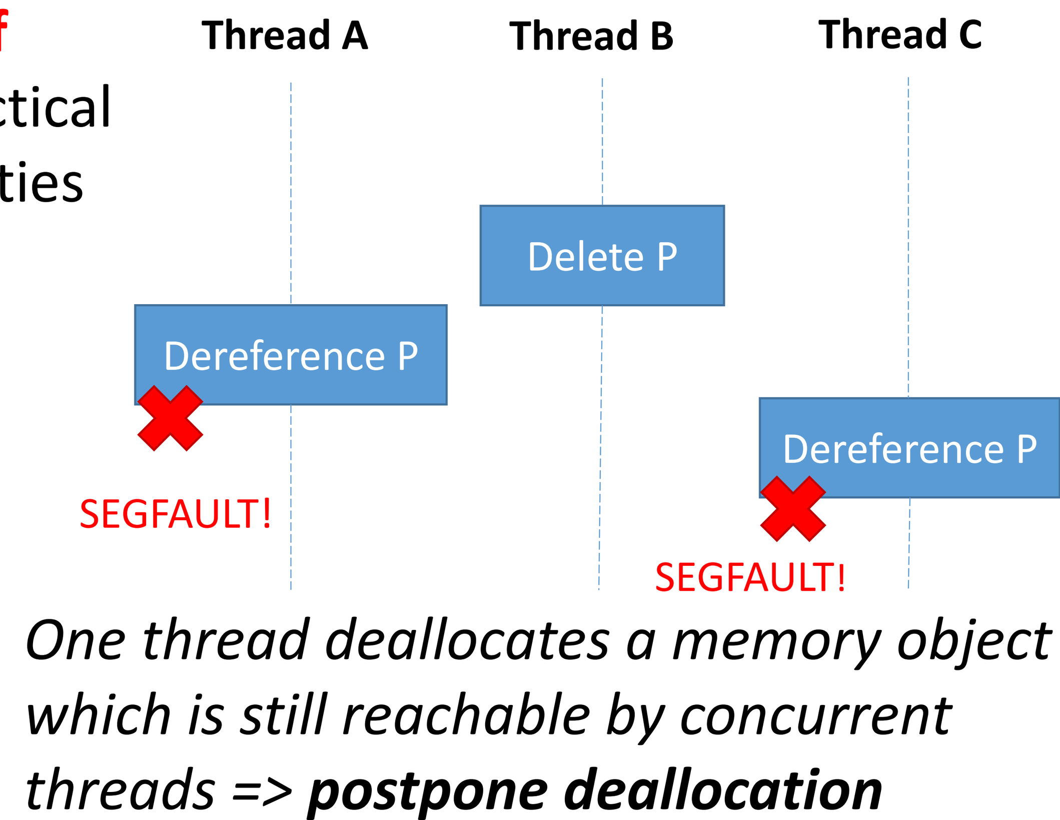
Snapshot-Free, Transparent, and Robust Memory Reclamation for Lock-Free Data Structures

Ruslan Nikolaev and Binoy Ravindran {rnikola, binoy}@vt.edu

Virginia Tech — Systems Software Research Group, ssrg.ece.vt.edu

Introduction

- **Concurrent data structures require special treatment of deleted memory objects** – garbage collectors are impractical in C/C++ and lack suitable progress/performance properties
- **Desirable properties for memory reclamation:**
  - ***Non-blocking progress***: not 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



*One thread deallocates a memory object which is still reachable by concurrent threads => **postpone deallocation***

Hyaline vs. Existing Schemes

- **Reference counting** is very slow
- **Robust** schemes typically lack snapshot-freedom
- **Hyaline-S** has all the desirable properties while retaining good performance and reasonable memory overhead

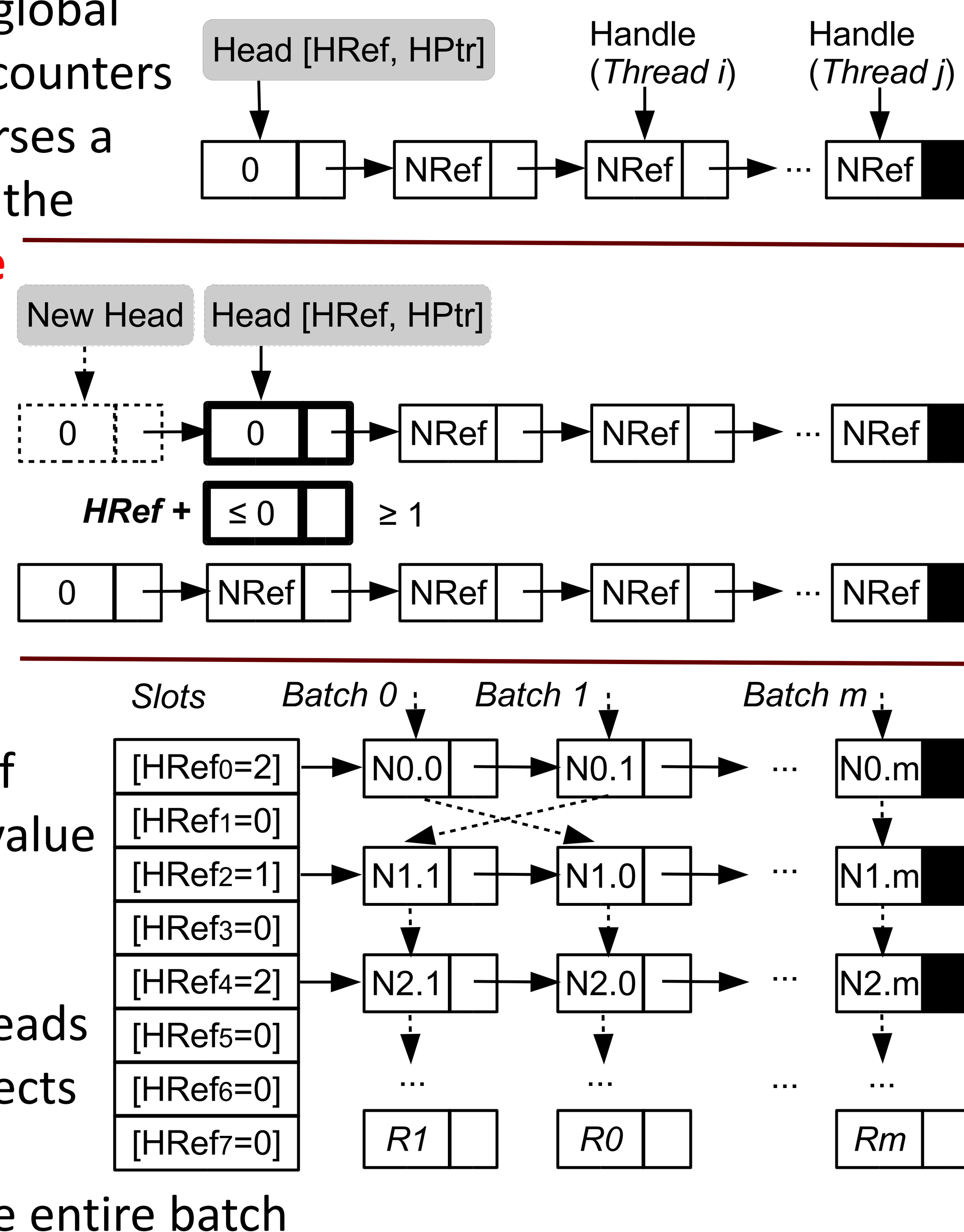
Scheme	Performance	Snapshot-Free	Robust	Transparent	Extra Memory	API complexity
Reference Counting	Very Slow	Yes	Yes	Partially (swap)	Double each pointer	Harder, Intrusive
Hazard Pointers	Slow	No	Yes	No (deletion)	1 word	Harder
Epoch Based Reclamation	Fast	Yes	No	No (deletion)	1 word	Very Easy
Hazard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

Problem: how do we create a safe memory reclamation scheme which satisfies all these properties?

Hyaline's Main Idea

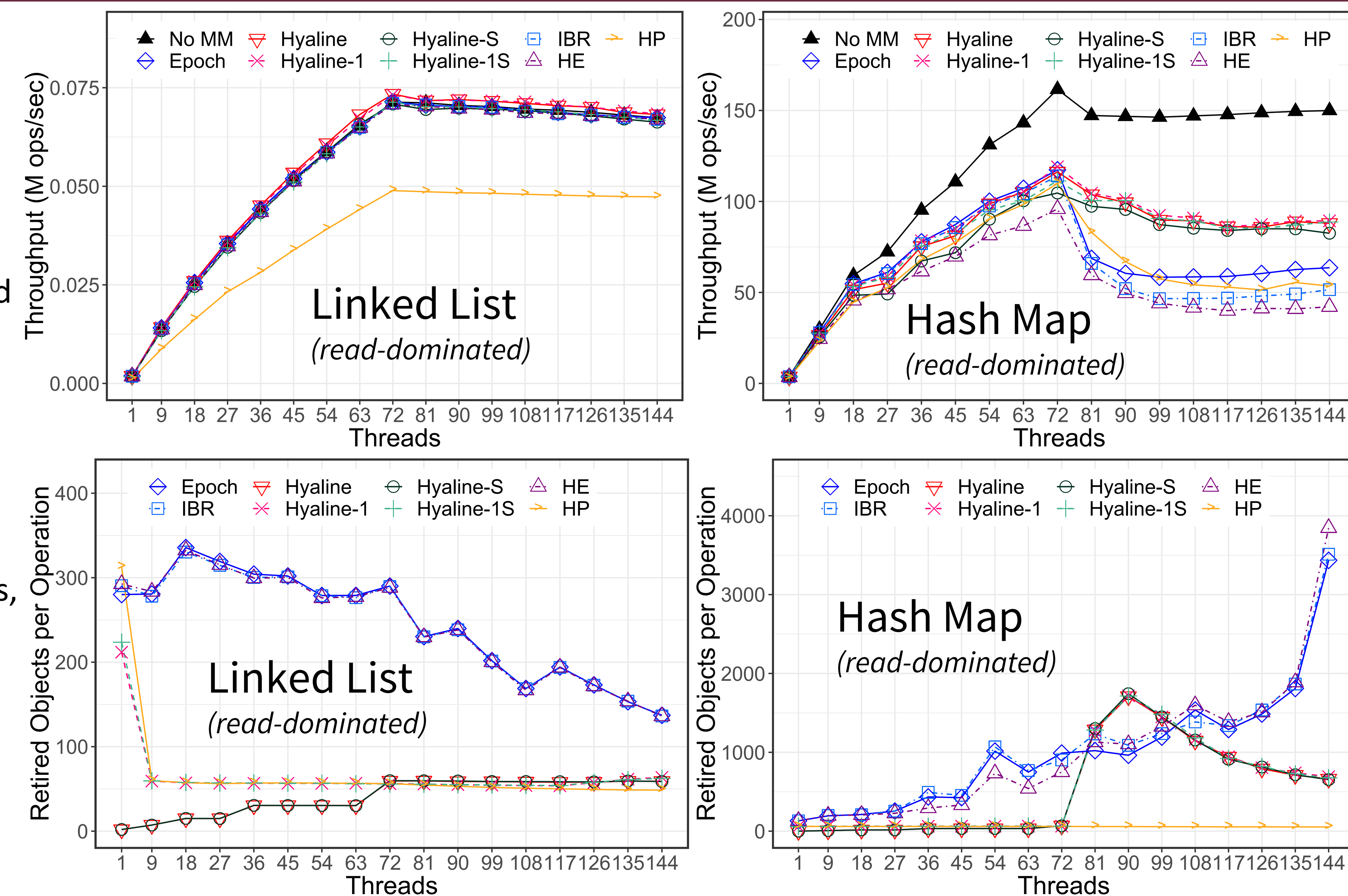
- Use **special** reference counting, which is triggered only when deleting objects
  - Update Head's reference counter (HRef) when **entering** and **leaving** thread operations
  - Append deleted objects to a global list and propagate reference counters
  - When leaving, a thread traverses a sublist from the beginning to the object pointed to by a **handle**
  - The handle points to the part of the list when the thread entered its operation
- 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
- Maintain **multiple** global lists to alleviate contention
  - Each list is for a subset of threads
  - Delete an entire **batch** of objects rather than just one object
  - One reference counter for the entire batch

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



Evaluation

- We tested Hyaline variants on x86(-64), ARM32/64, PowerPC, and MIPS
- All Hyaline variants exhibit very high throughput on various data structures, and ensure that the number of retired, but not-yet-reclaimed objects is small
- Hyaline's advantages are especially visible in certain read-dominated workloads
- In oversubscribed scenarios, Hyaline obtains up to **2x** throughput boost
- We present results for x86-64, read-dominated tests (90% get, 10% put)



Conclusions

- With Hyaline, threads can be created and deleted dynamically: threads are "off the hook" as soon as they leave operations
- Hyaline and Hyaline-S are fully transparent: they need not explicitly (un)register threads

Availability and Acknowledgments

- Hyaline's code and the benchmark are open-source and available at https://github.com/rusnikola/lfsmr

- 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