Space and Time Bounded Multiversion Garbage Collection

Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, Yihan Sun, and Yuanhao Wei

Introduction

- Multiversioning widely used:
  - Database systems
  - Software Transactional Memory **[FC'11] [LS'13]**
  - Concurrent data structures **[WBBFRS'21] [NHP'20]**

- High space usage ⇒ obsolete versions must be reclaimed
  - Multiversion garbage collection problem (MVGC)
  - Observed to be a bottleneck in modern database systems **[LSPKNCSH'16] [BLNK'19]**

Research Question

How do you garbage collect efficiently for multiversioning?

Main results

A **general** MVGC scheme with:

- **Time**: **O(1)** per reclaimed version, on average
- **Space**: roughly **constant factor** more versions than needed
- **Progress: wait-free**

Previous solutions either use:

- **unbounded space** [WBBFRS'21] [FC'11], or
- **O(P)** time per reclaimed version [BLNK'19] [LSPKNCSH'16] [LS'13]
  - **P**: number of processes

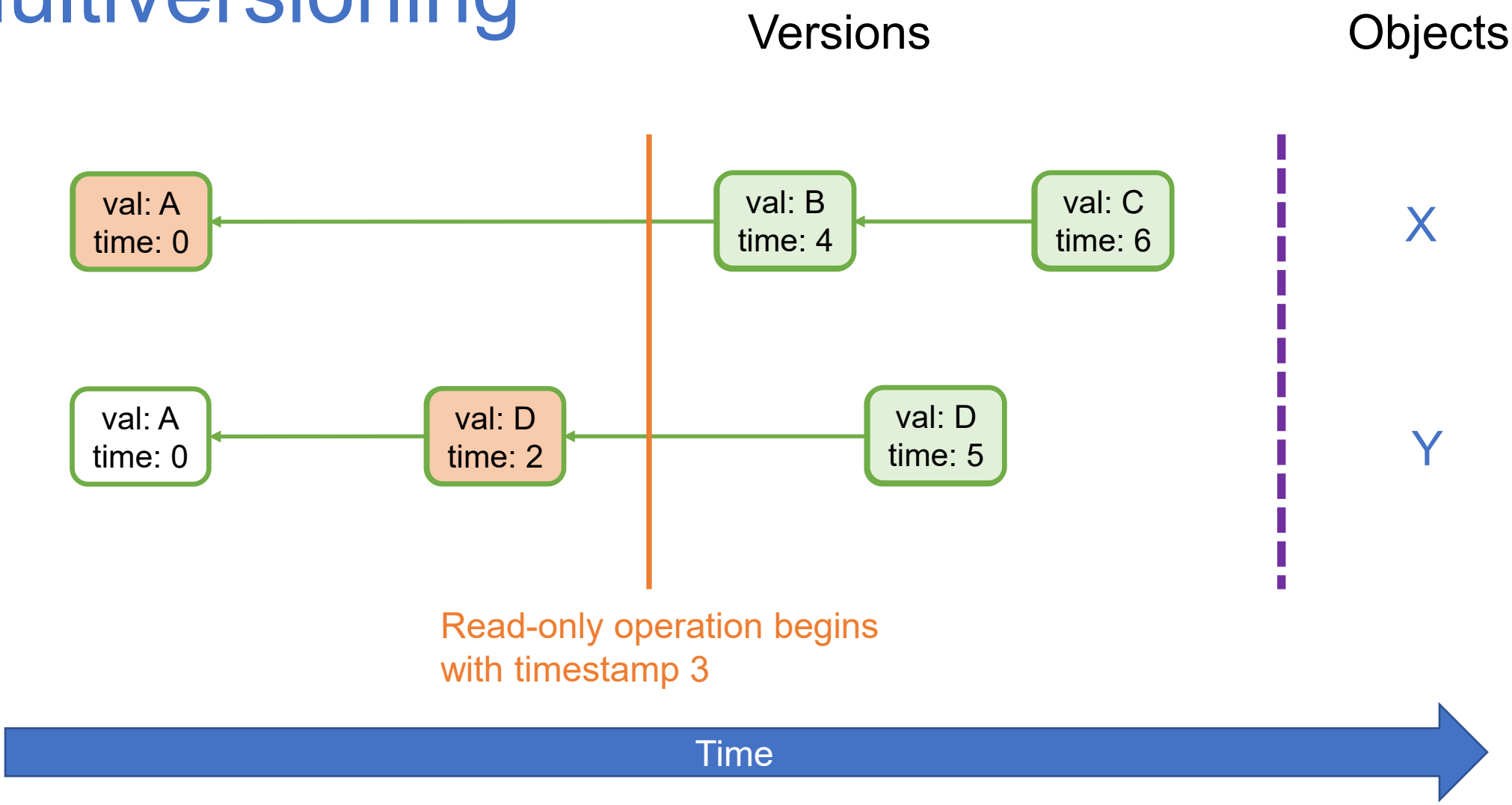
Main results

A **general** MVGC scheme with:

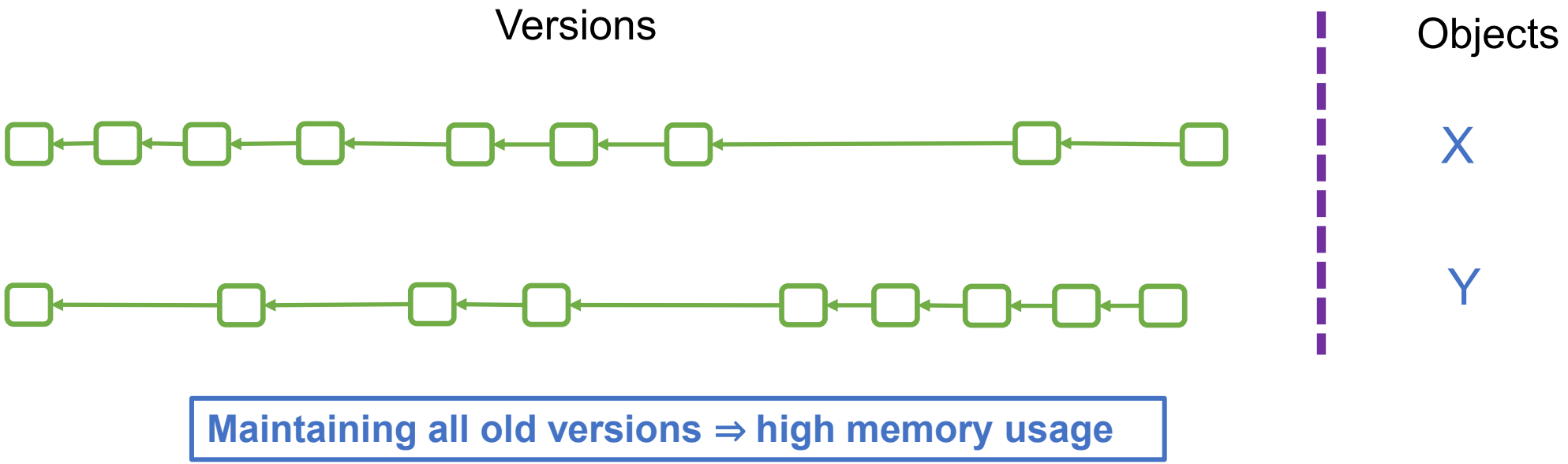
- **Time**: **O(1)** per reclaimed version, on average
- **Space**: roughly **constant factor** more versions than needed
- **Progress: wait-free**

- Components of independent interest:
  - Range tracking data structure
  - Concurrent doubly-linked-list with amortized O(1) time remove()

Multiversioning

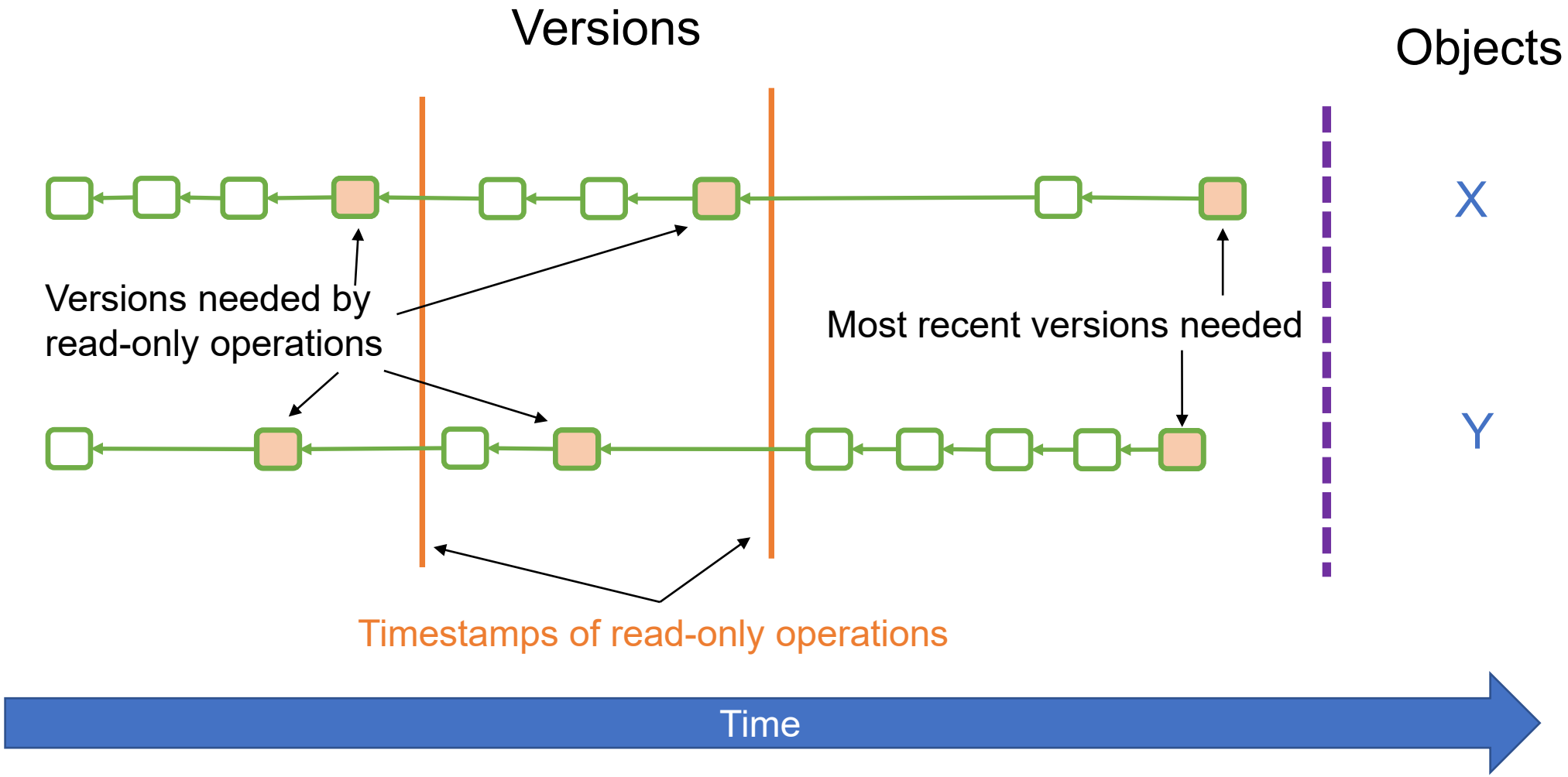


Multiversion Garbage Collection (MVGC)

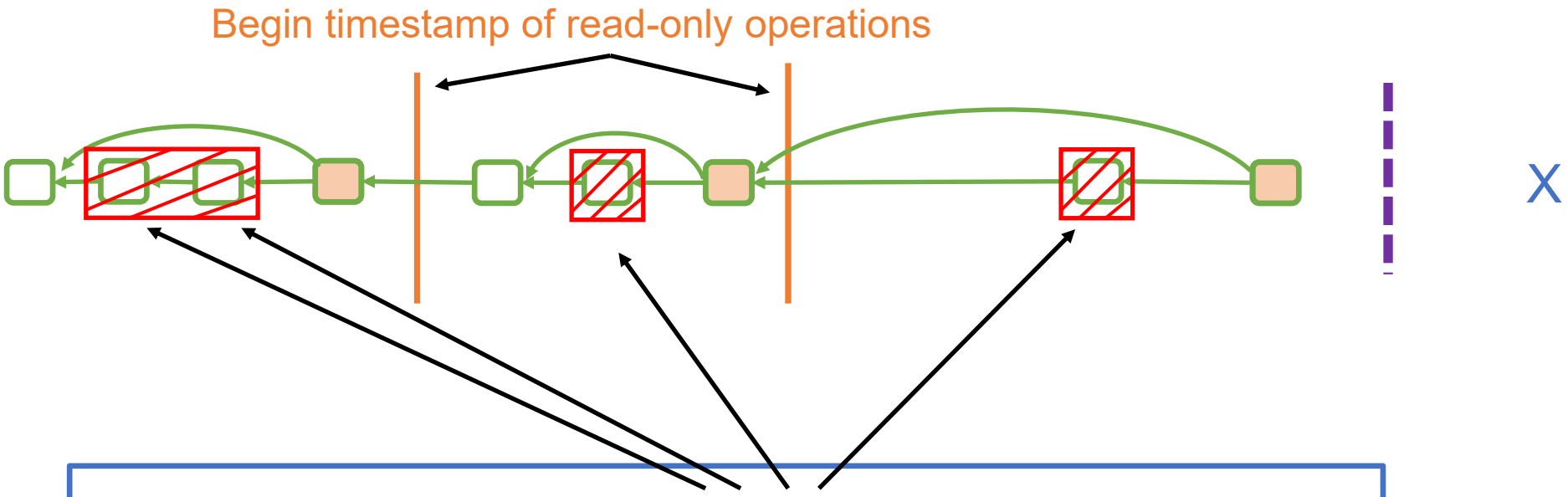


- How do we know which versions obsolete?
- How do we safely reclaim them?

Which Versions are Needed?



Multiversion Garbage Collection (MVGC)



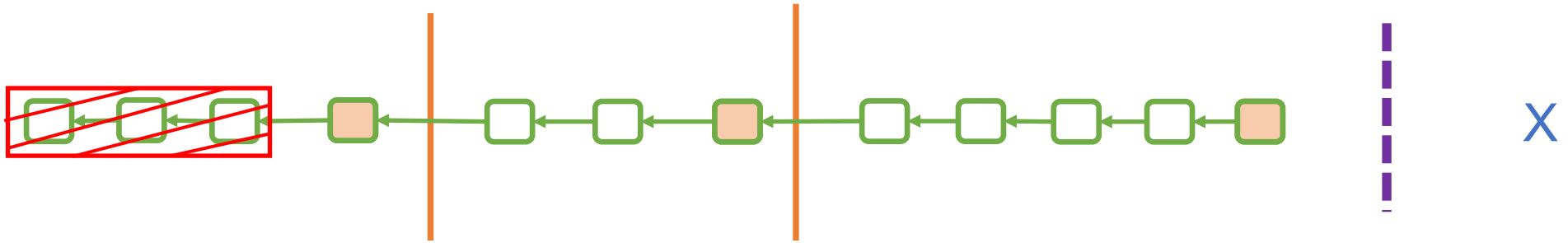
Step 1: Identify obsolete versions

Step 2: Unlink from version list

Step 3: Reclaim memory of unlinked versions

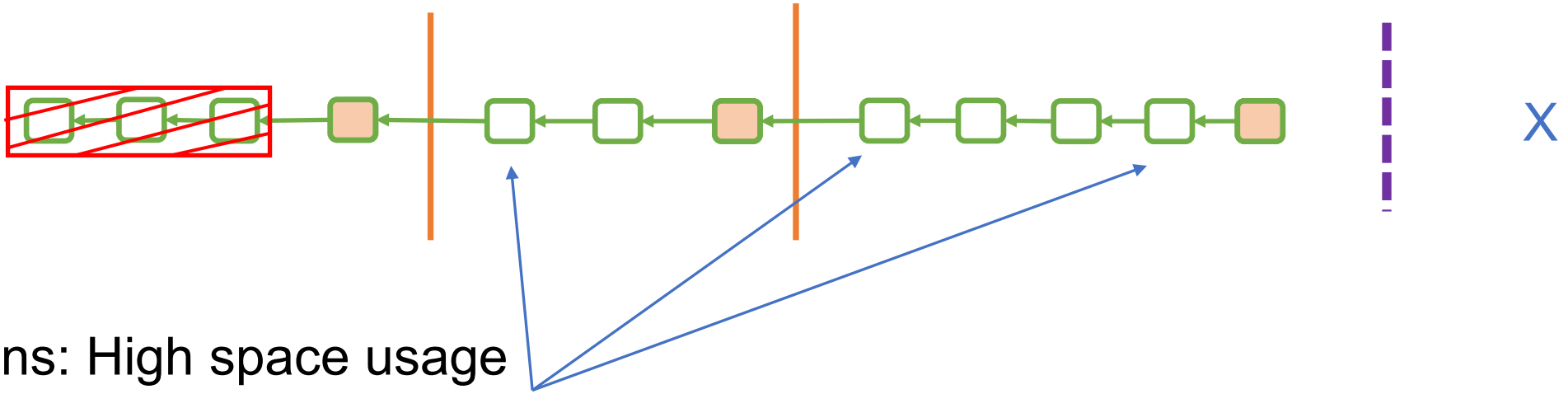
Related Work – Epoch-Based Solutions

- Track the oldest active read-only operation and reclaim any version overwritten before the start of this operation
- Most commonly used



- Pros: Fast, easy to implement

Related Work – Epoch-Based Solutions



- Cons: High space usage
  - Unable to collect newer obsolete versions
  - Particularly bad with long read-only operations, which is one of the main motivations for multiversioning
    - E.g. database scans, range queries, etc
  - Paused process can lead to unbounded space usage

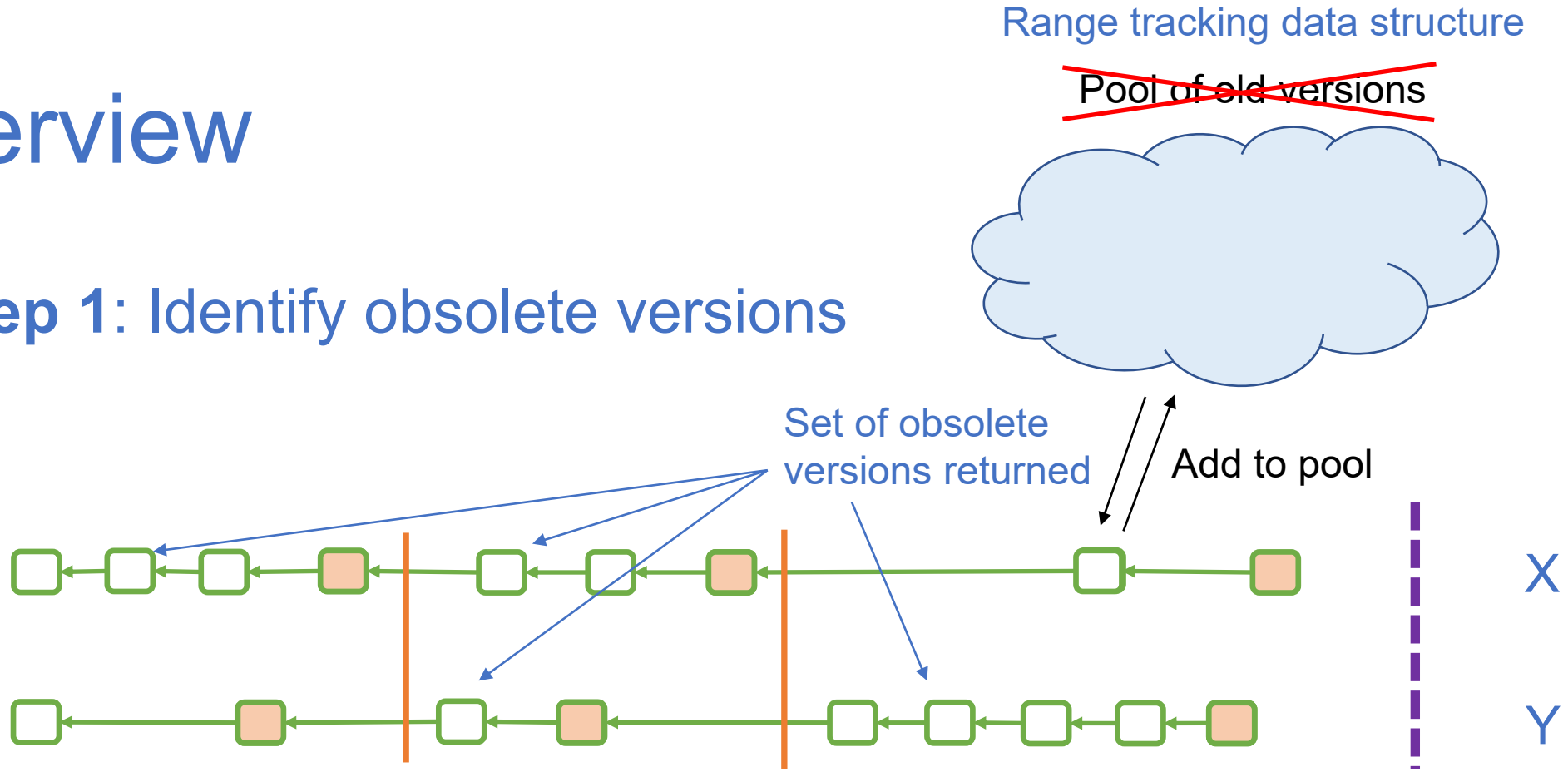
Related Work – Other Solutions

- Techniques have been developed to address shortcomings of epoch-based solutions
  - GMV **[LS'13]**, Hana **[LSPKNCSH'16]**, Steam **[BLNK'19]**
  - Requires Ω(P) time, on average, to collect each version in worst case executions,
  - Keeps up to P times more versions than necessary

Overview

Step 1: Identify obsolete versions
Step 2: Unlink from version list
Step 3: Reclaim memory of unlinked versions

Overview



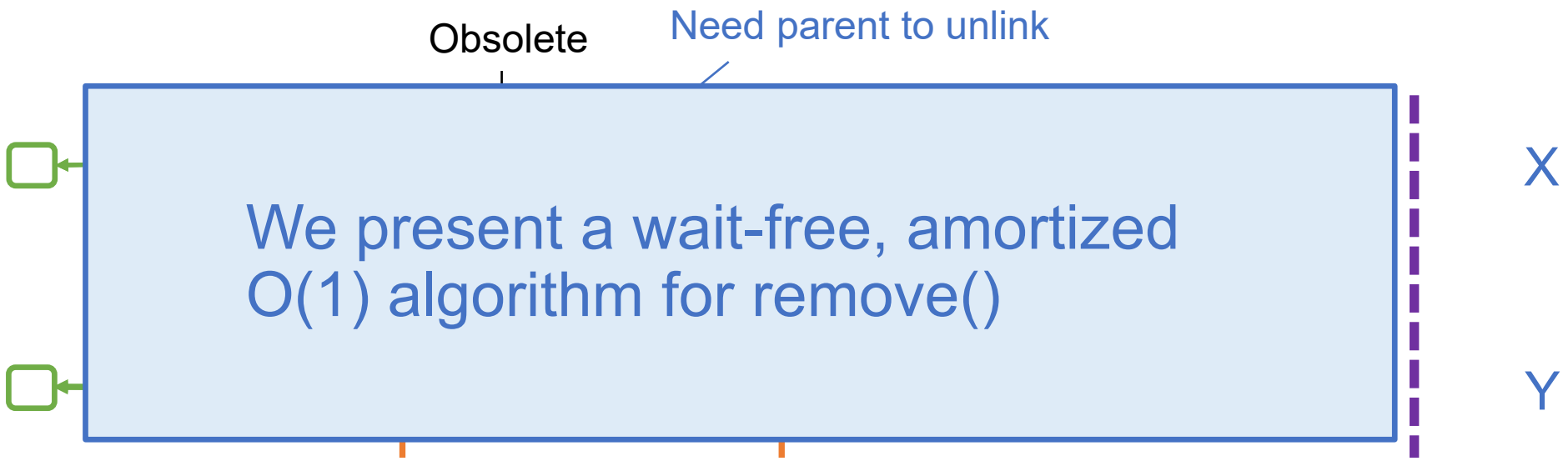
Step 1: Identify obsolete versions

Step 2: Unlink from version list

Step 3: Reclaim memory of unlinked versions

Overview

Step 1: Identify obsolete versions
Step 2: Unlink from version list



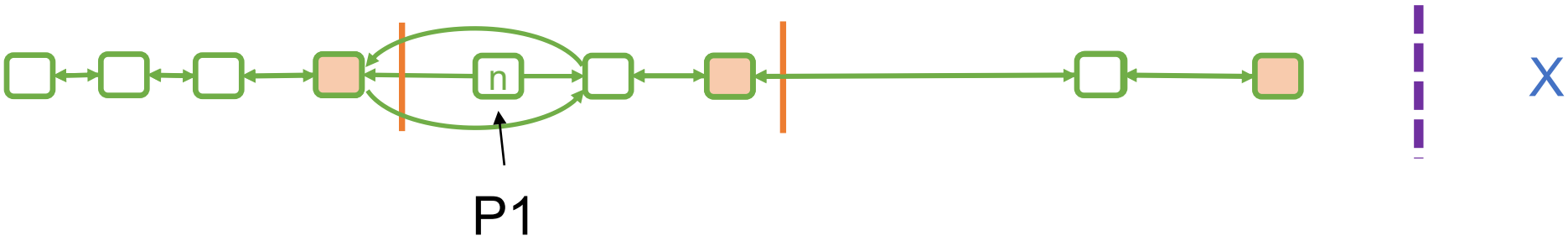
Step 3: Reclaim memory of unlinked versions

Overview

Step 1: Identify obsolete versions

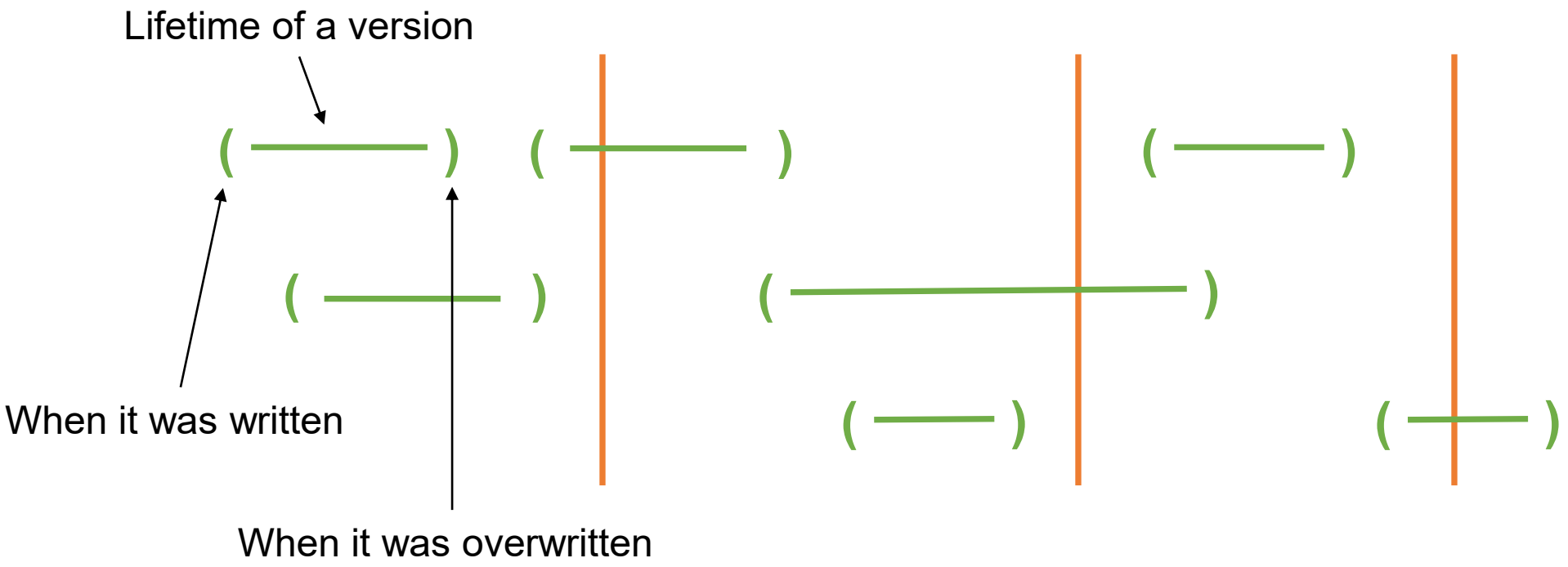
Step 2: Unlink from version list

Step 3: Reclaim memory of unlinked versions



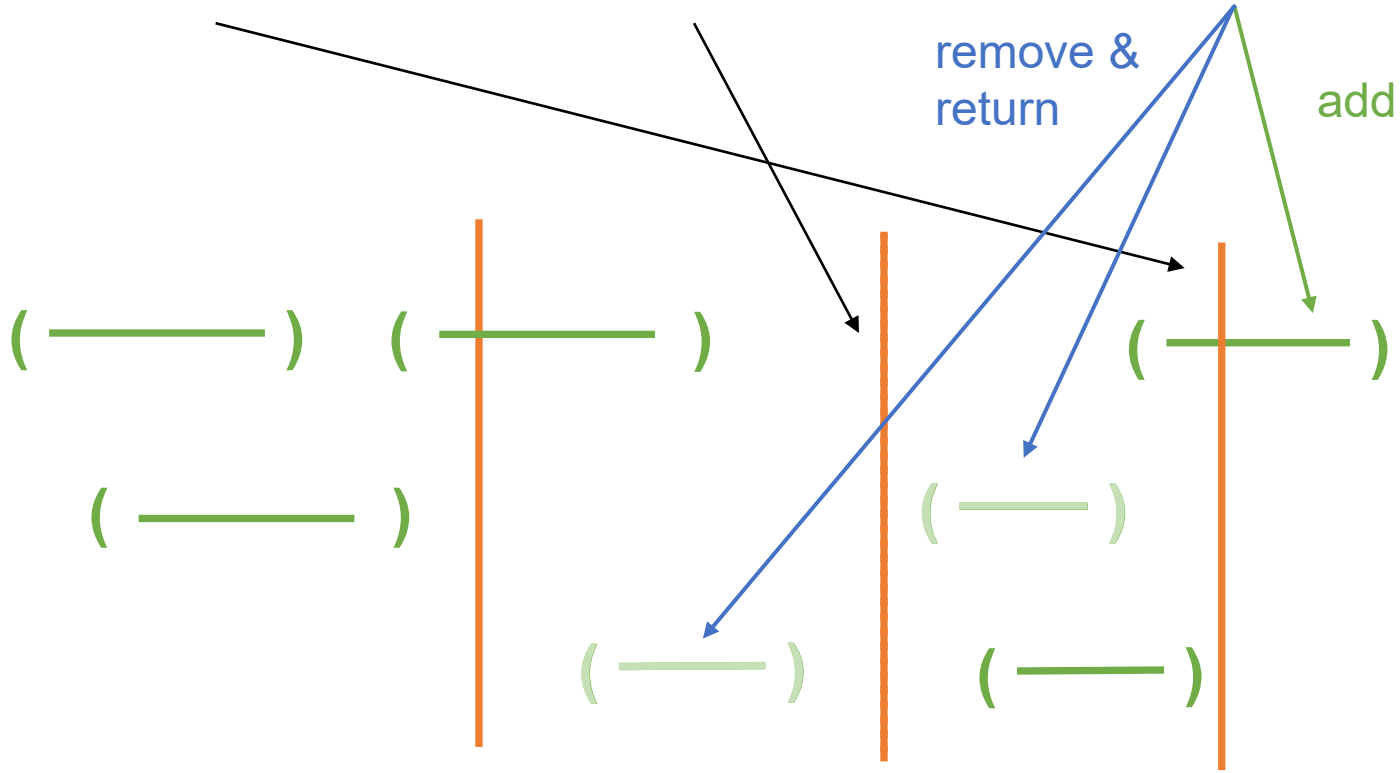
- n is not safe to reclaim right away because a process (P1) could be paused on it
- Using Hazard Pointers (HP) or Concurrent Reference Counting (CRC) would solve this problem, but
  - HP sacrifices wait-freedom
  - CRC has bad worst case space bounds
- We design a new safe reclamation scheme specifically for our doubly linked version list

Step 1: Identifying Obsolete Versions

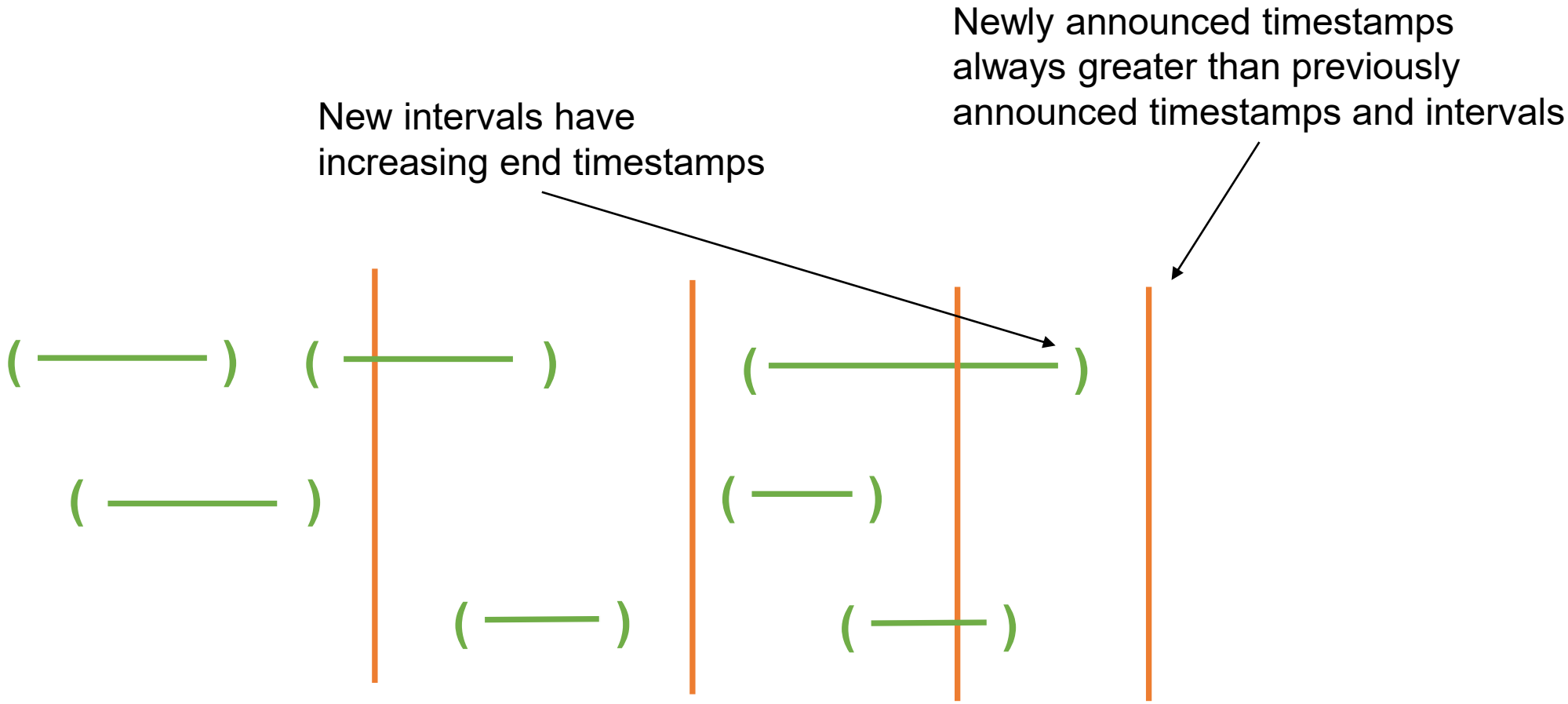


Range Tracker: Definition

- Supports announce(), unannounce(), and deprecate()



Observations



Range Tracker: Implementation

- Announce() & unannounced() write to an announcement array
- Deprecate(Range r):
  - Append r to process local list (sorted by end timestamp)
  - If local list reaches size O(P log P),
    - Push local list onto a shared queue
    - Pop two lists from the shared queue
    - Scan announcement array
    - Separate the two popped lists into intersected (A) and non-intersected intervals (B)
      - O(P log P) time
    - Push A back to the shared queue and return B

Range Tracker: Time Bounds

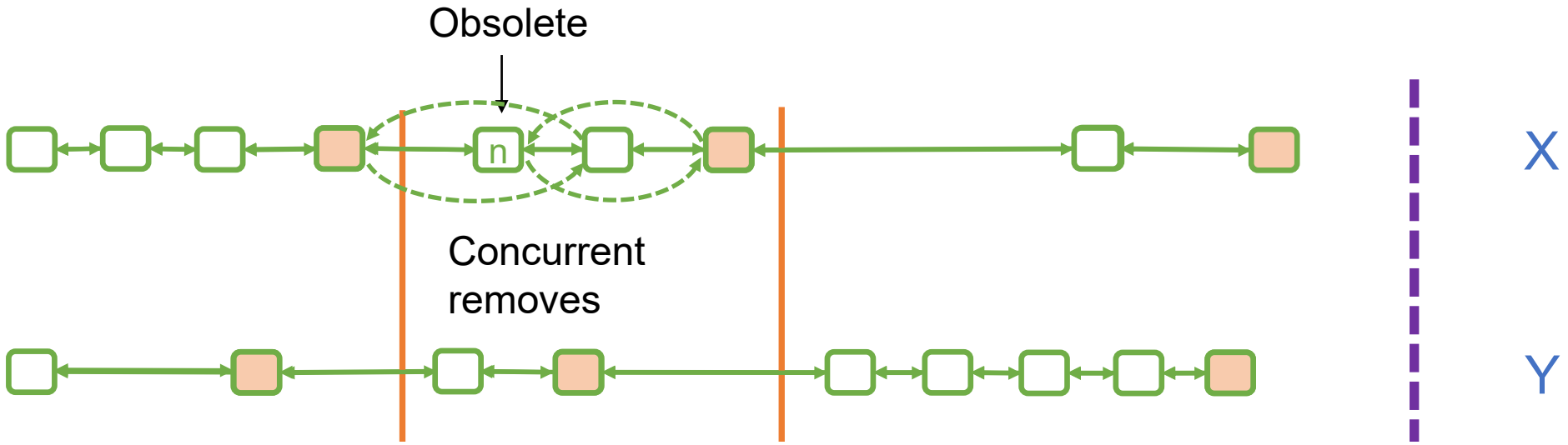
- O(1) time announce() & unannounced()
- O(P) time push/pop from shared queue (P-SIM, wait-free)
- Every O(P log P) calls to deprecate() we perform:
  - 2 pop, 1-3 push to shared queue
  - O(P log P) algorithm for finding intersected intervals
- Deprecate() takes amortized O(1) time

Range Tracker: Space Bounds

- $O(P^2 \log P)$ intervals in local lists, in total
- O(M) intervals in the shared queue
  - M: maximum number of needed intervals at any point in history
- Overall, the range tracker stores a constant factor more intervals than needed plus an additive term

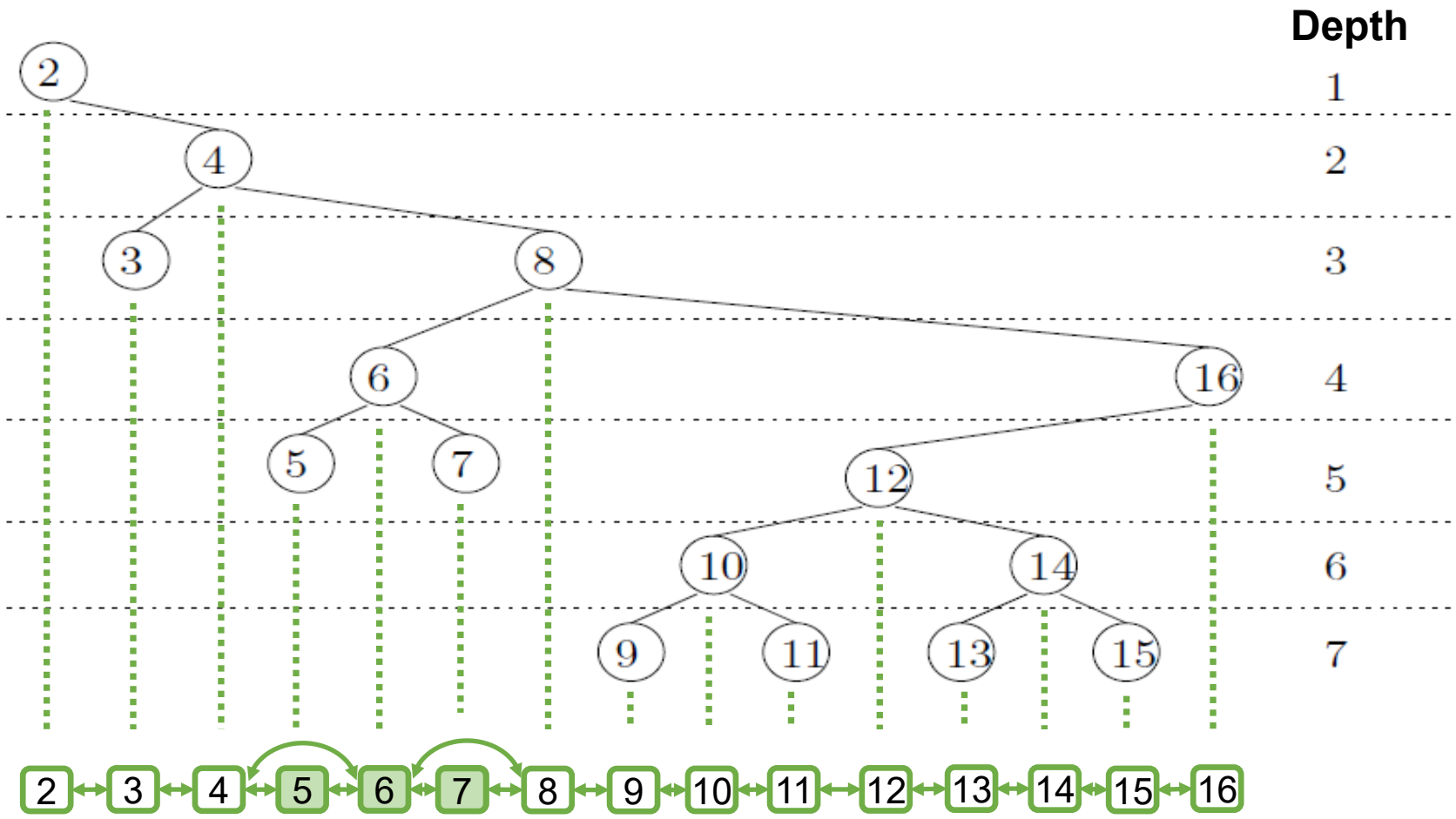
Overview

Step 1: Identify obsolete versions
Step 2: Unlink from version list



Step 3: Reclaim memory of unlinked versions

Step 2: Unlinking from version lists

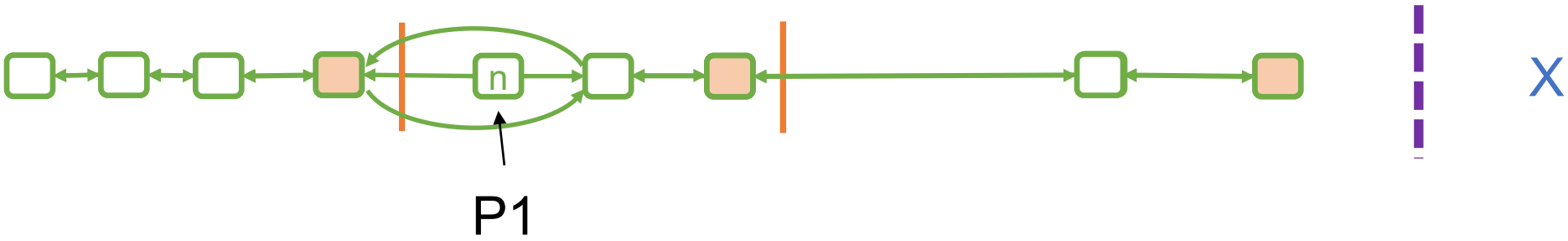


Overview

Step 1: Identify obsolete versions

Step 2: Unlink from version list

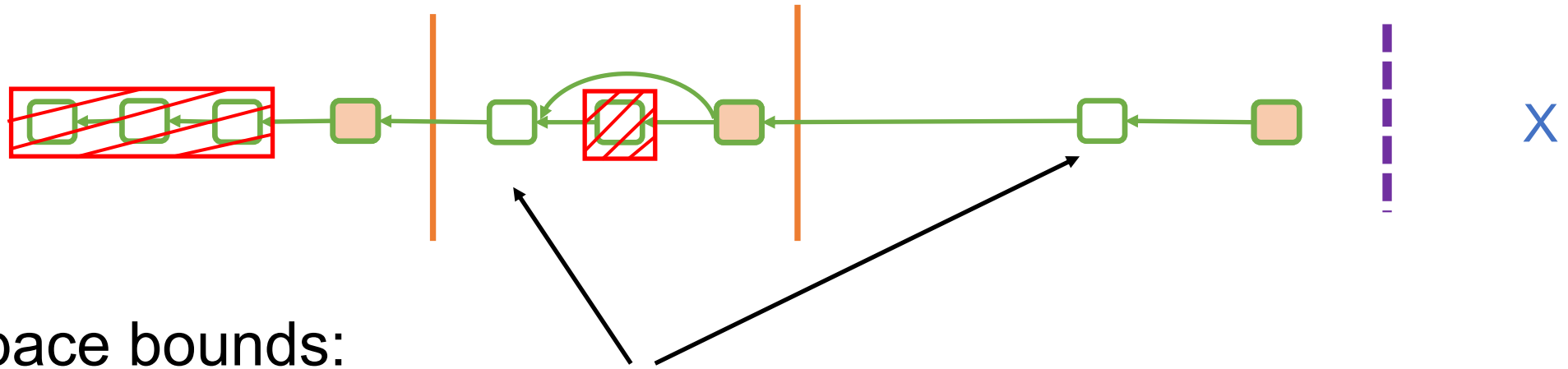
Step 3: Reclaim memory of unlinked versions



- n is not safe to reclaim right away because a process (P1) could be paused on it
- Using Hazard Pointers (HP) or Concurrent Reference Counting (CRC) would solve this problem, but
  - HP sacrifices wait-freedom
  - CRC has bad worst case space bounds
- We design a new safe reclamation scheme specifically for our doubly linked version list

Our results

Diagram might be good for title page



- Space bounds:
  - Number of unreclaimed versions ∈ ~O(# required versions)
- Time bounds:
  - O(1) time, on average, to identify, remove, and reclaim a version
  - Wait-free

Space Bounds

Our MVGC technique (counting all three steps) achieves:

- Amortized O(1) time, in expectation, for each reclaimed version
- A maximum of $O(N + P^2 \log P + P \log L)$ unreclaimed versions
  - N: high watermark number of needed versions throughout execution
  - P: number of processes
  - L: maximum number of versions added to a single version list
- In large data structures, $N >> (P^2 \log P + P \log L)$
  - Rough notes:
    - Emphasize this somehow, Log L is small, P is small compared to N
    - Size of databases vs how many processes they run with

Conclusion

- Multiversion Garbage Collection is an important problem
- Our paper presents a theoretically efficient solution
- Currently working on a practical version, preliminary results look promising

Thank you for listening!

References

- J. Lee, H. Shin, C. G. Park, S. Ko, J. Noh, Y. Chuh, W. Stephan, and W.-S. Han. Hybrid garbage collection for multi-version concurrency control in SAP HANA. In SIGMOD. ACM, 2016.
- S. M. Fernandes and J. Cachopo. Lock-free and scalable multi-version software transactional memory. In ACM Symposium on Principles and Practice of Parallel Programming (PPOPP), page 179–188, 2011.
- L. Lu and M. L. Scott. Generic multiversion STM. In *Proc. International Symposium on Distributed Computing*, pages 134–148. Springer, 2013.
- J. Böttcher, V. Leis, T. Neumann, and A. Kemper. Scalable garbage collection for in-memory MVCC systems. *Proceedings of the VLDB Endowment*, 13(2):128–141, 2019.