Improving Restore Speed for Backup Systems that Use Inline Chunk-Based Deduplication

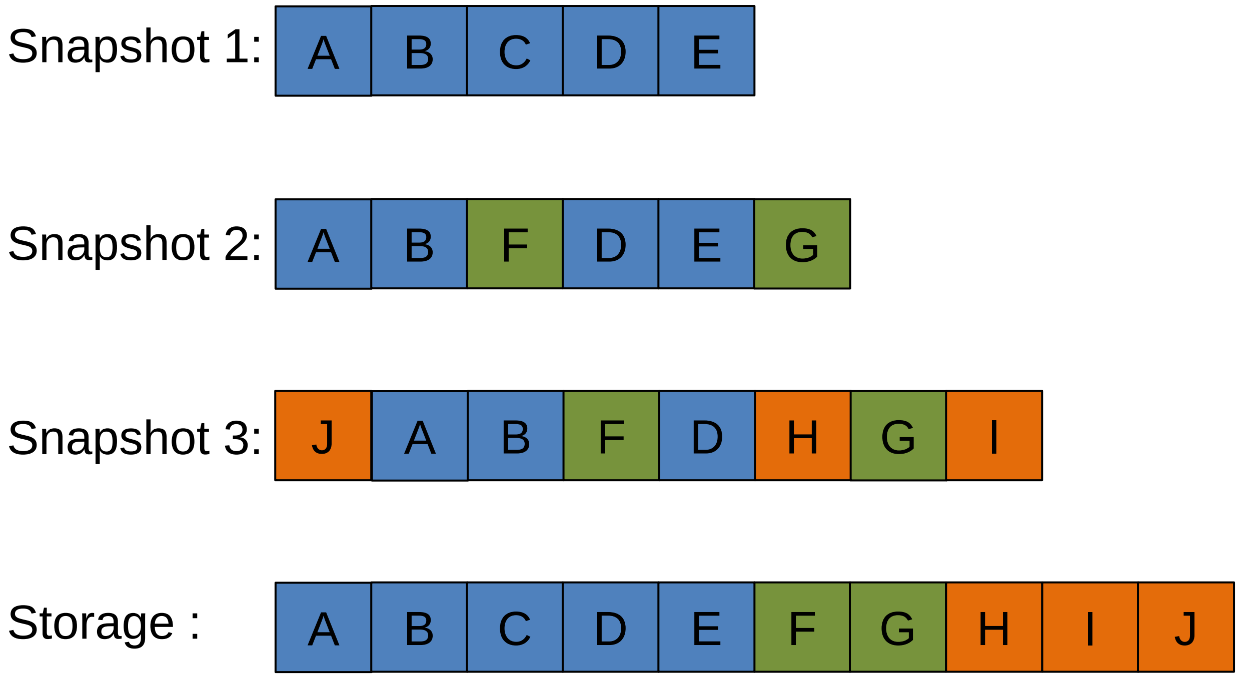
Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat

HP, Inc

FAST 13

The Problem

- Slow restore performance due to chunk fragmentation
  - Store chunks according to the order of appearance of unique chunks
  - Restore the newer snapshot, the more random reads, the lower the performance



The Problem

- Solve the problem via defragment data periodically?
- No layout makes most backups restore fast
  - Backups disagree about optimal order of chunks (MFDedup[FAST'21])
  - Could focus on last week's backups (RevDedup[APSYS'13])
- Rearranging chunks is expensive
  - Example: keep last snapshot's chunks in order
- This work: improve speed without rearranging data

Measuring Fragmentation & Restore Speed

- Deduplication storage unit: containers
  - Standard size: 4 MB
  - Observation: reading containers is dominant restore cost (over 80%)
- Measure method: Container read per MB restored
  - Total # of container reads that occur / restored backup size
  - Depends on caching used
  - Uncaptured restore-time variance:
    - Container size is difference
    - File system fragmentation, seek time variance, etc.

Simulating Restoration

- Baseline algorithm:

  For chunk c in backup B:

  Read in c's container C

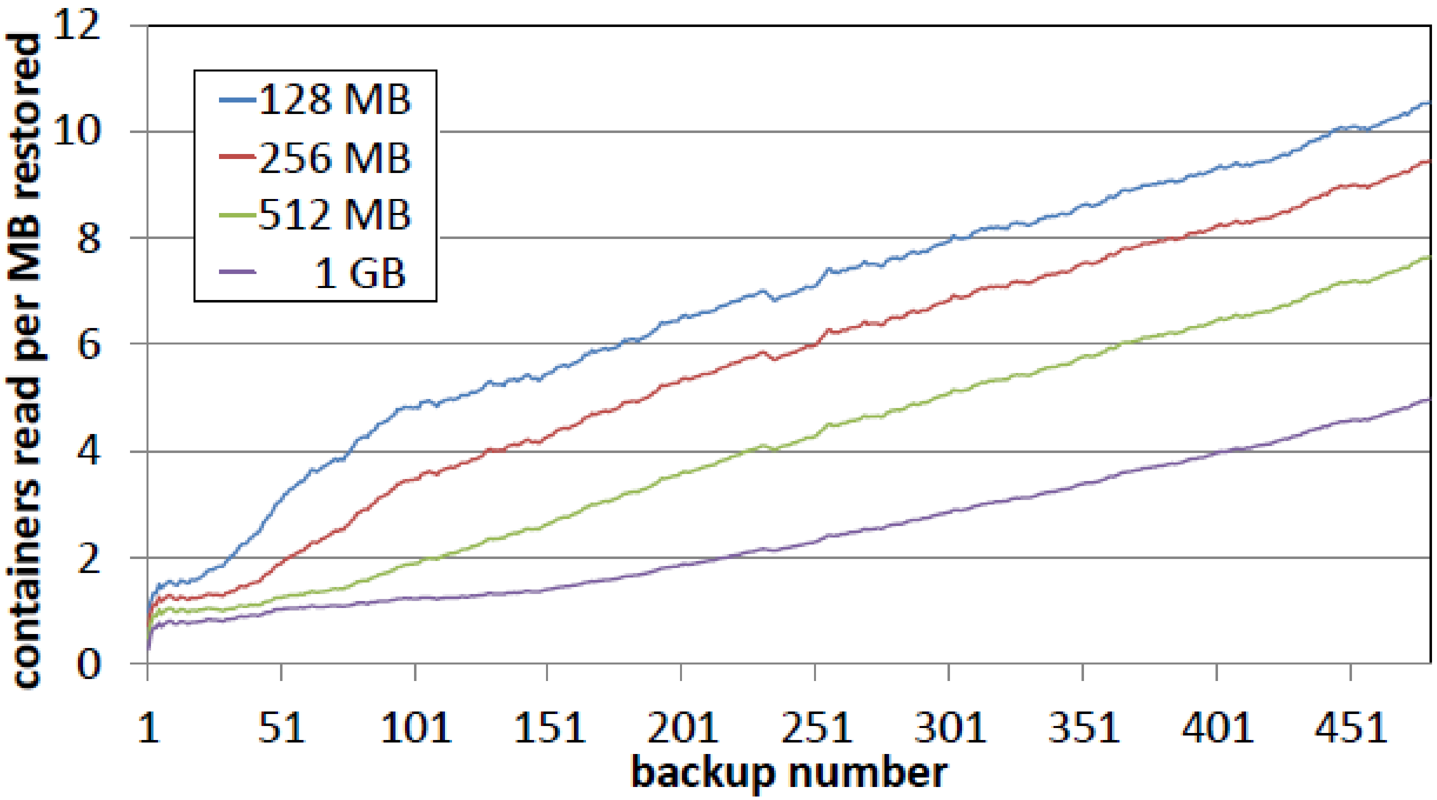
  Extract c's content from C via offset and size

  Send out c's data

- Subtleties:
  - LRU caching (Impact of cache size)
  - Read entire containers: Read speed >> Seek time

Fragmentation Over Time

- The curves are different LRU cache sizes

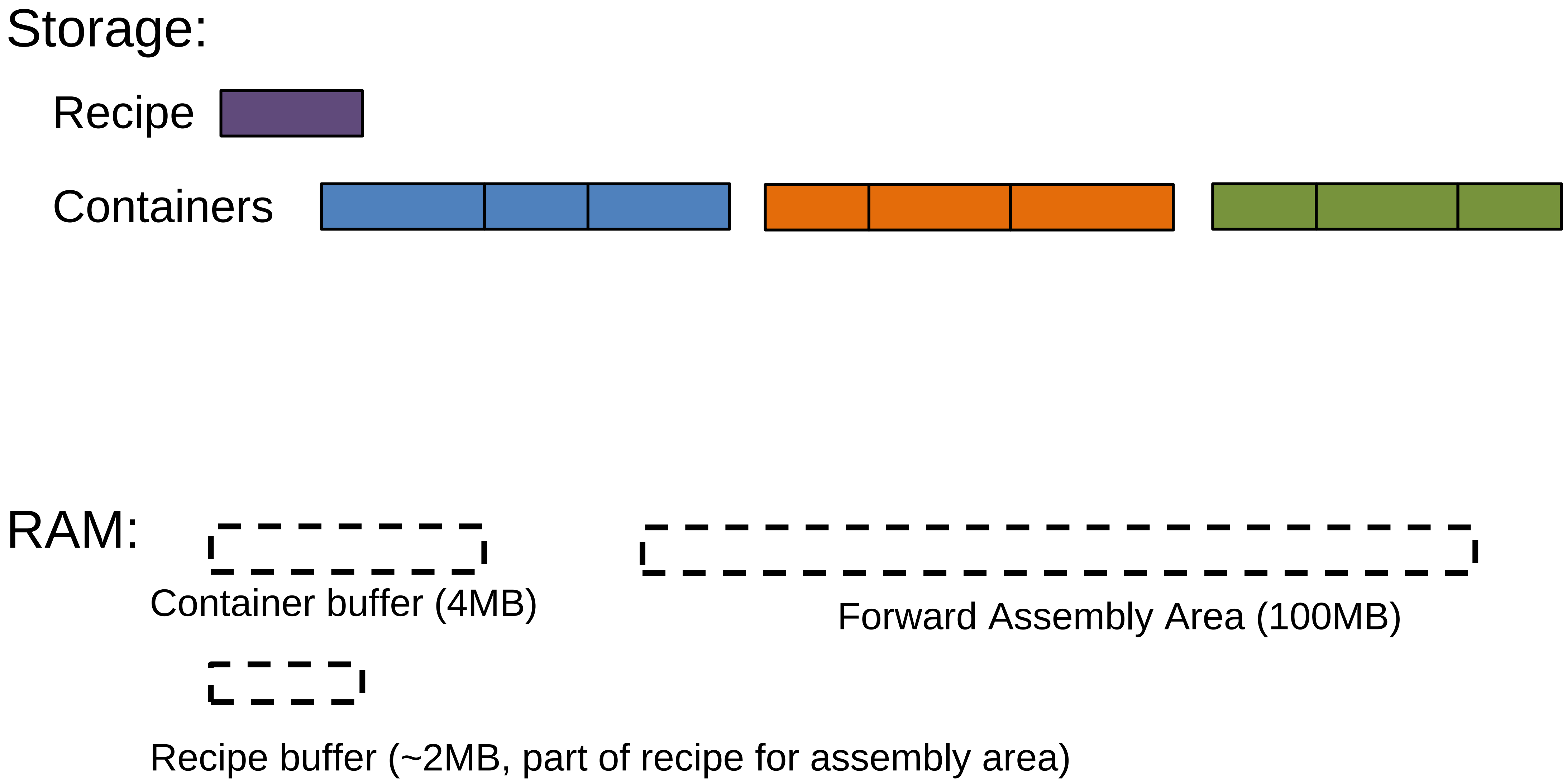


Large caches smooth out small-scale fragmentation

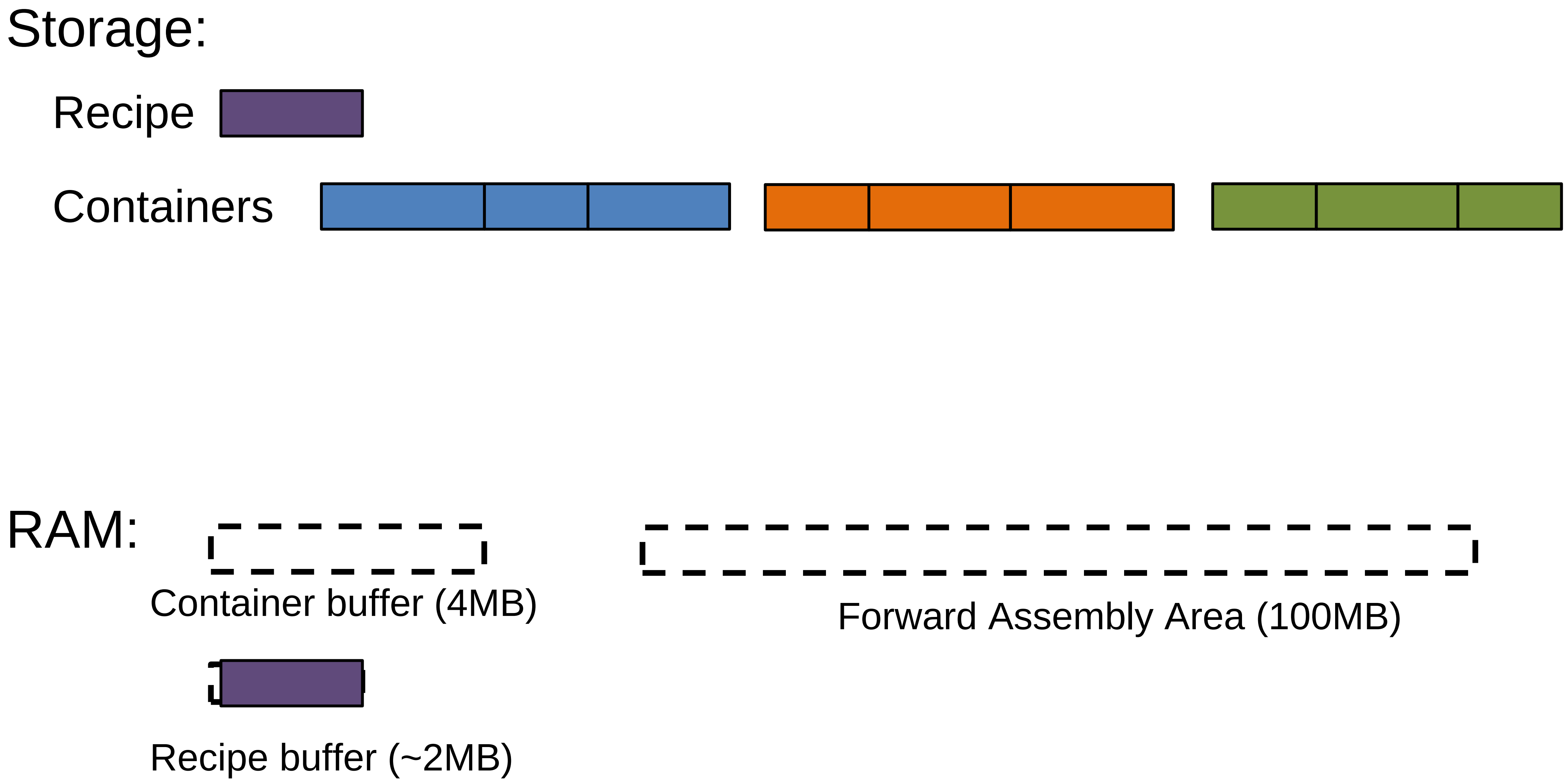
Improving Restore-time Caching

- A new caching method:
  - The forward assembly area method
- Exploits:
  - Perfect knowledge of future accesses
    - Courtesy of the recipe
  - Keeps only needed chunks
- Designed to minimize memory overhead
  - Lots of small variable-sized objects
  - Reduce memory copies

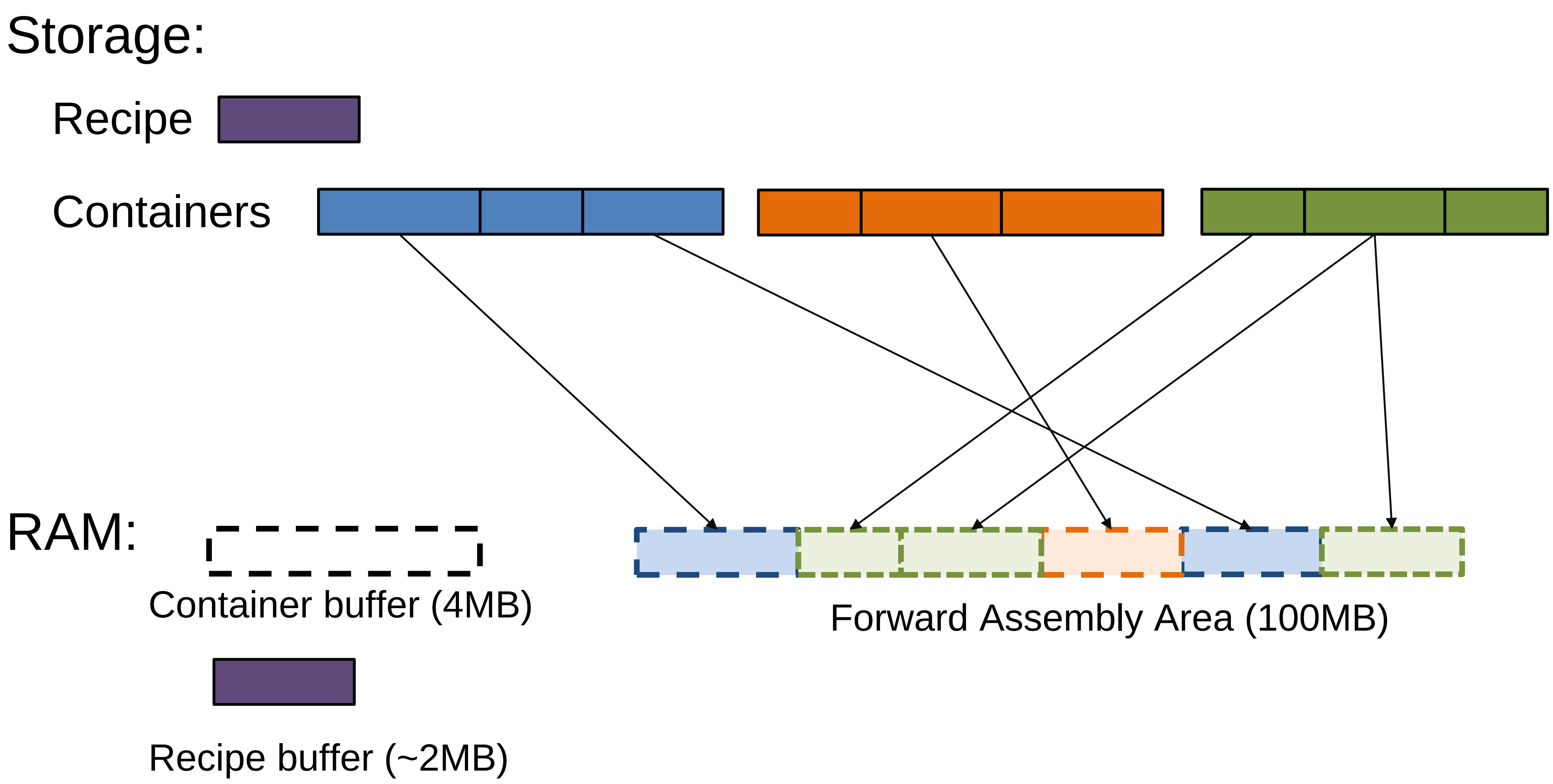
Forward Assembly Area Method



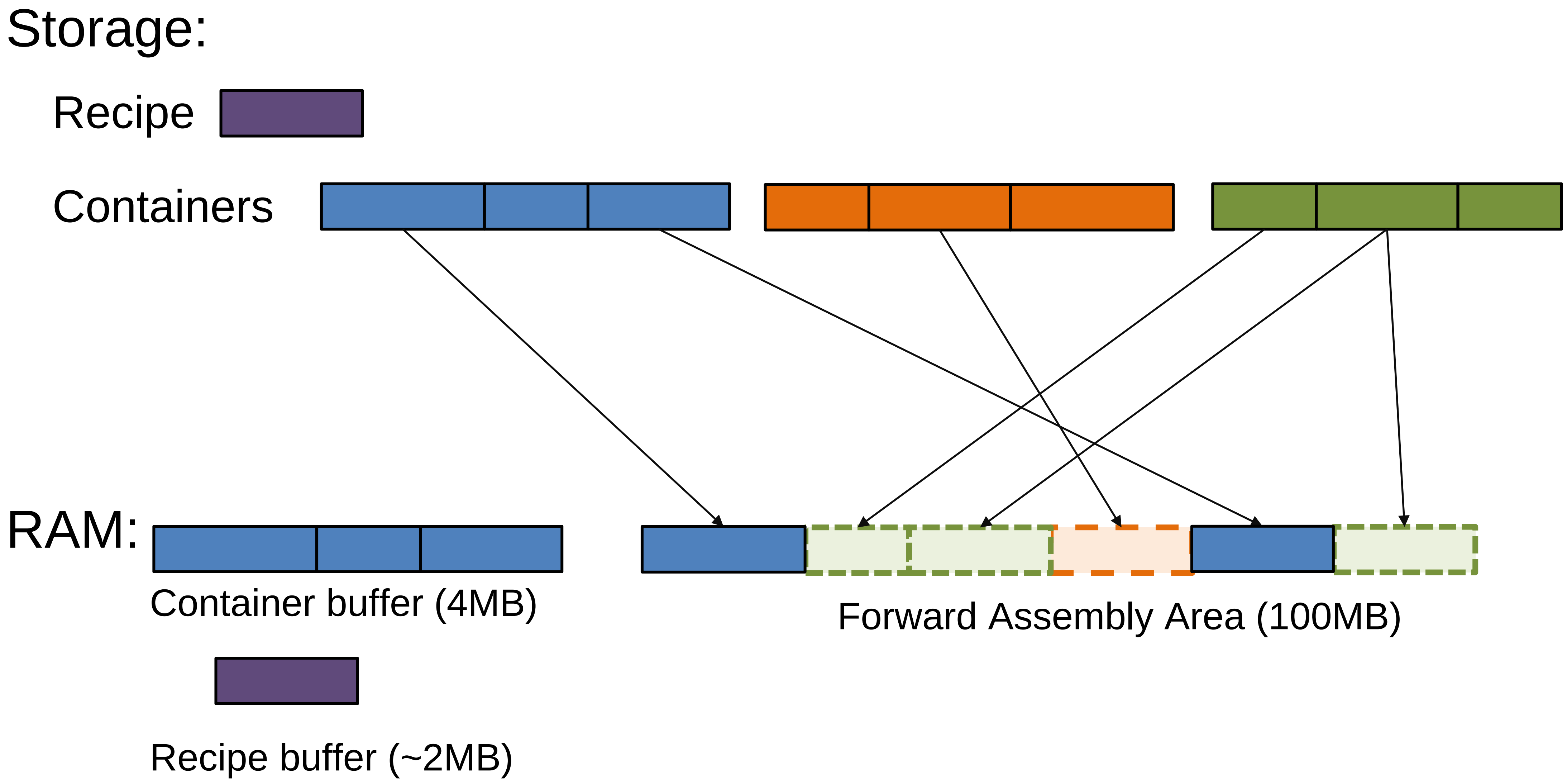
Forward Assembly Area Method



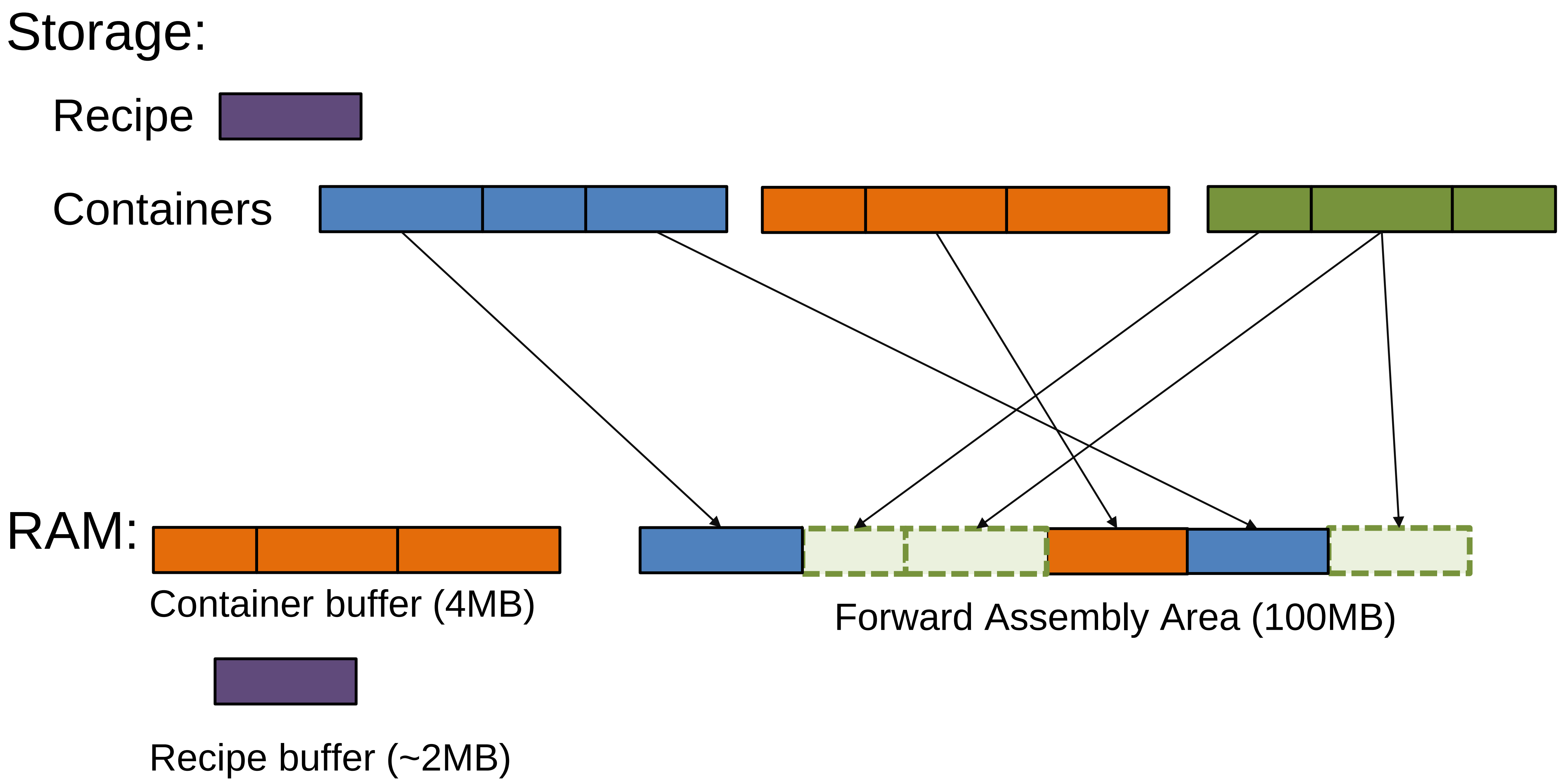
Forward Assembly Area Method



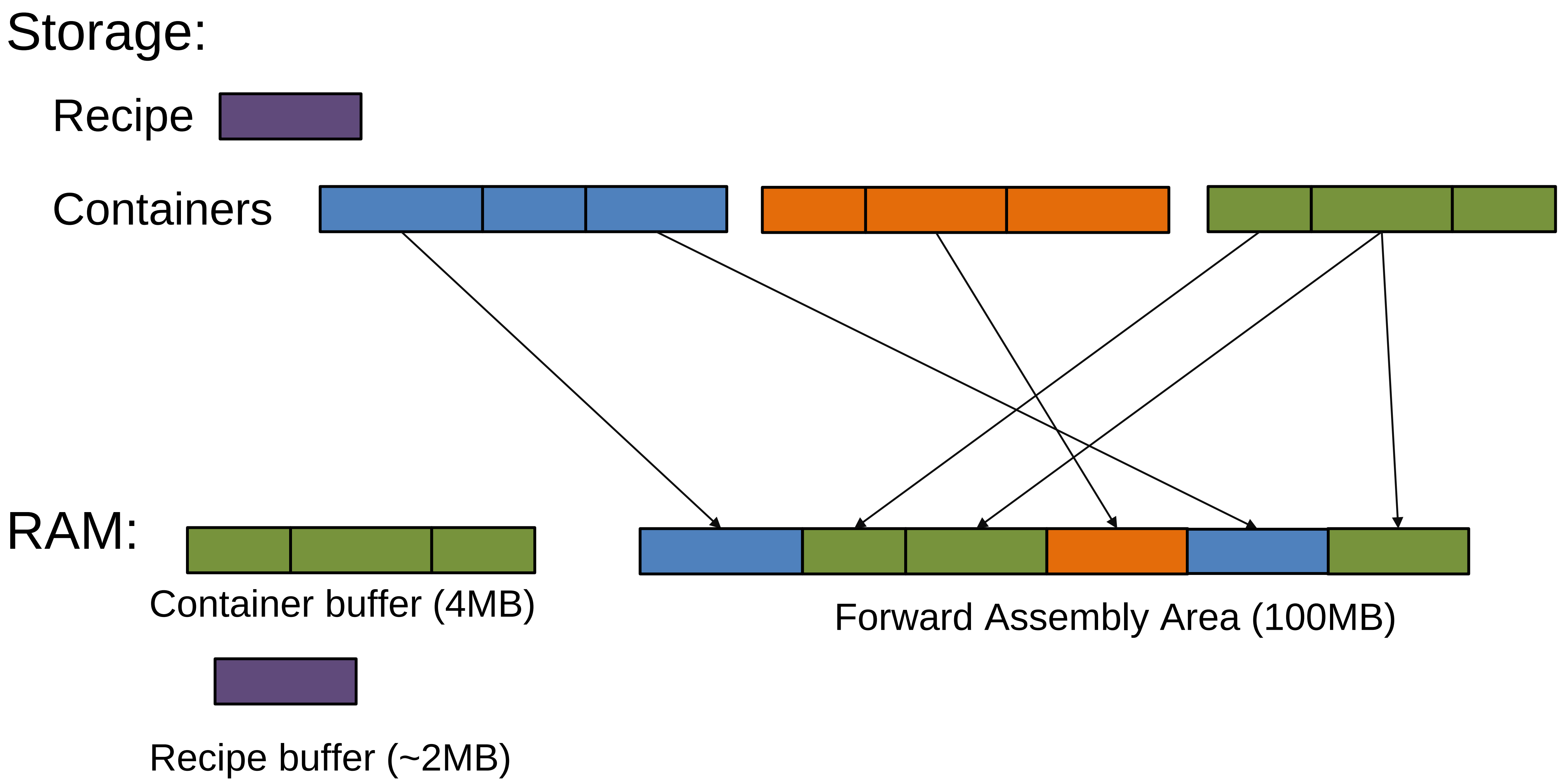
Forward Assembly Area Method



Forward Assembly Area Method



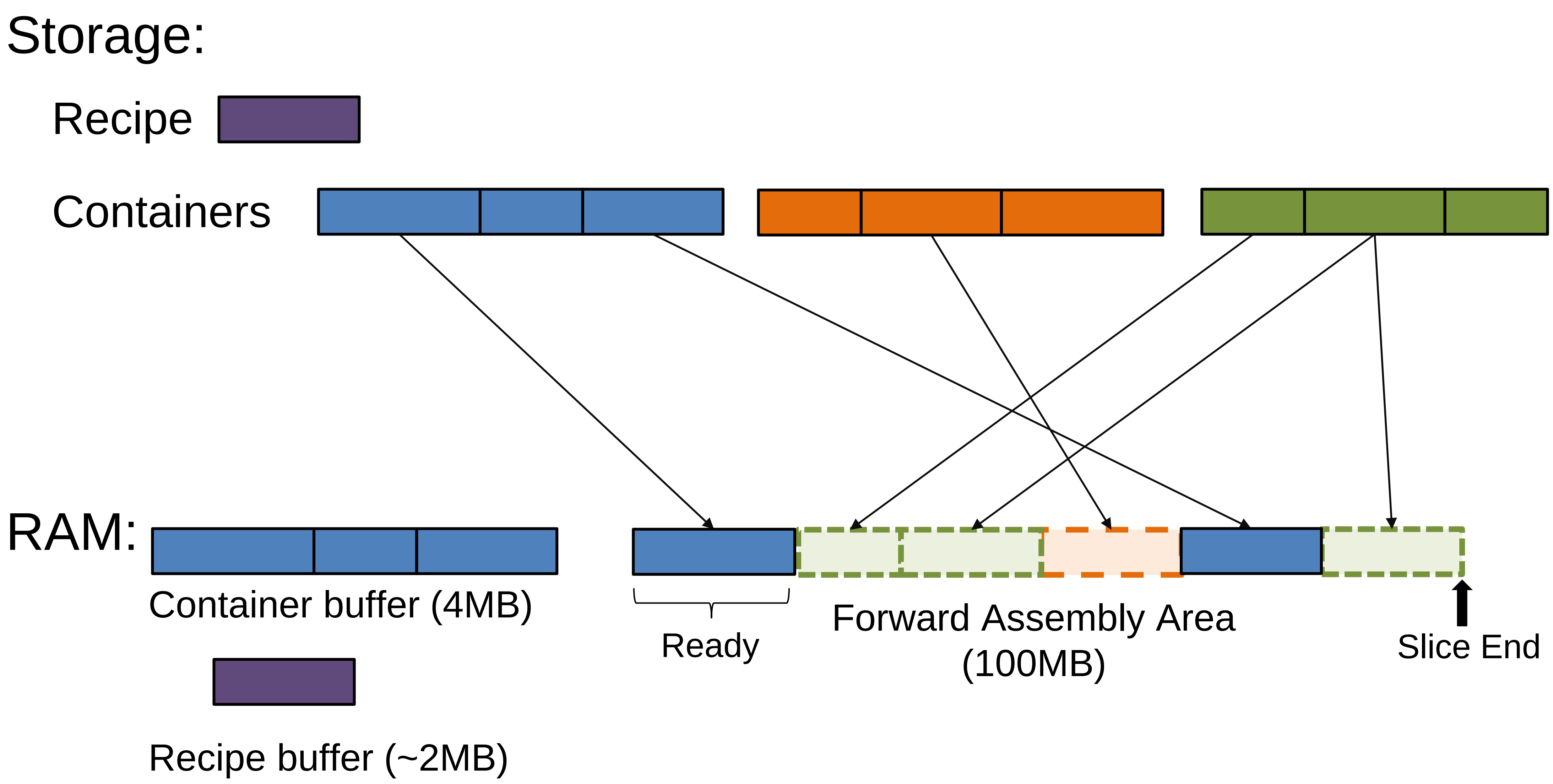
Forward Assembly Area Method



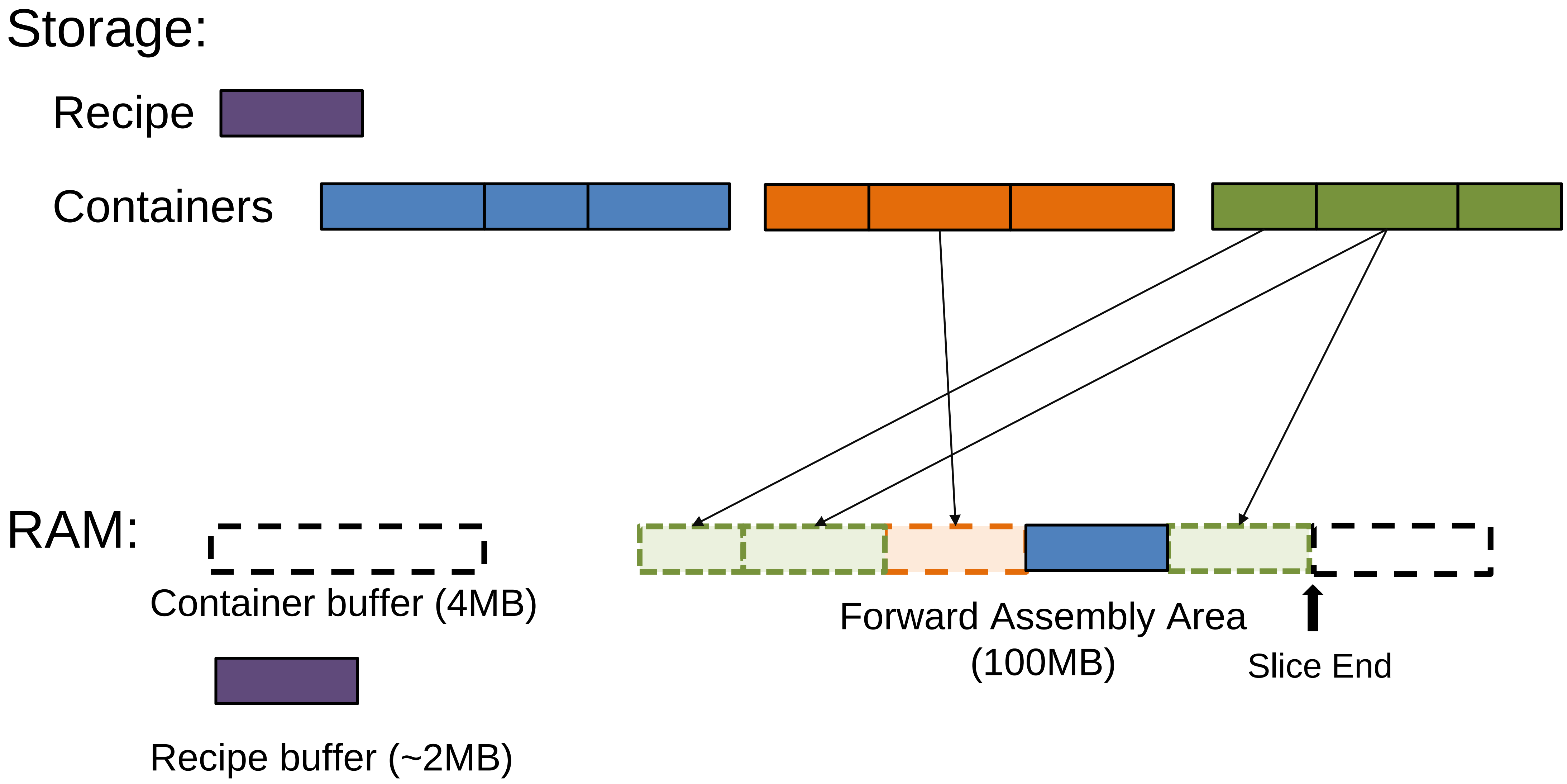
Forward Assembly Area Method

- Cache only needed chunks
  - Load each container once per slice (100 MB)
  - Uses memory more efficiently
- Moves chunks directly into position
  - No intermediate storage or extra copies
- Better rolling approach
  - A ring-buffer based approach,
  - Send out the continuous filled-in part at the (logical) start of the forward assembly area and then rotate the ring buffer.
  - Not need to reloads around slice boundaries

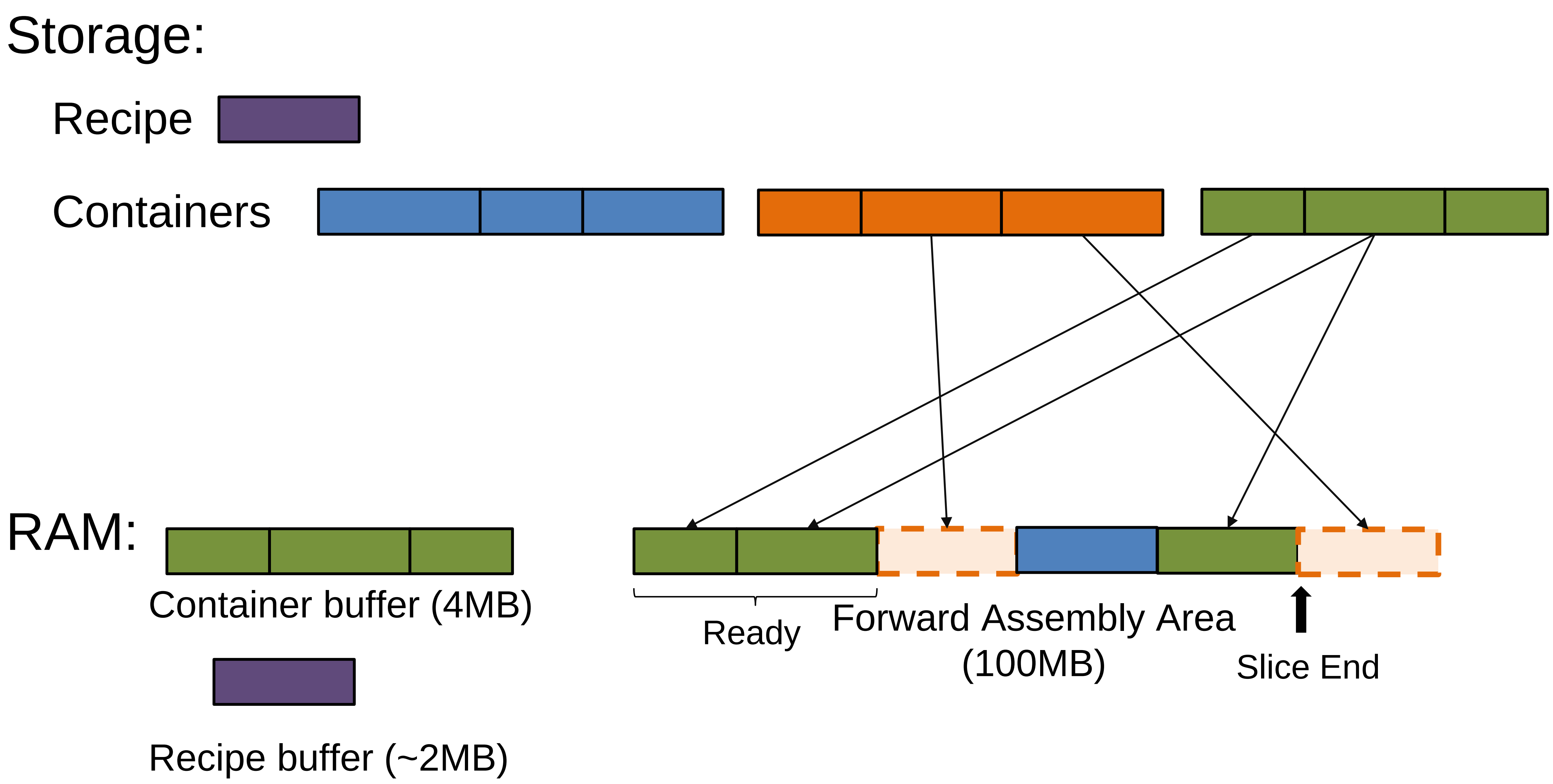
Forward Assembly Area Rolling Method



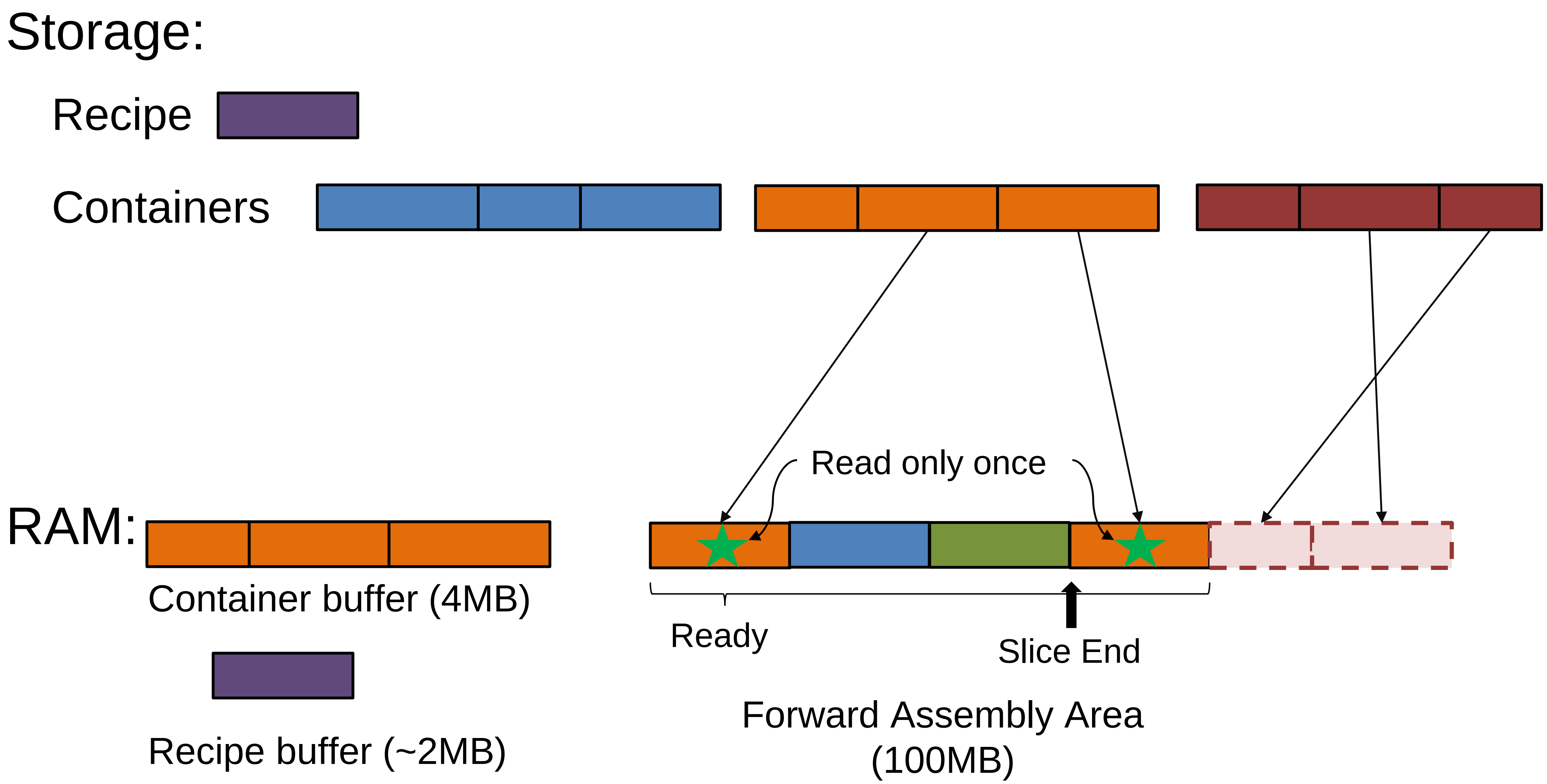
Forward Assembly Area Rolling Method



Forward Assembly Area Rolling Method



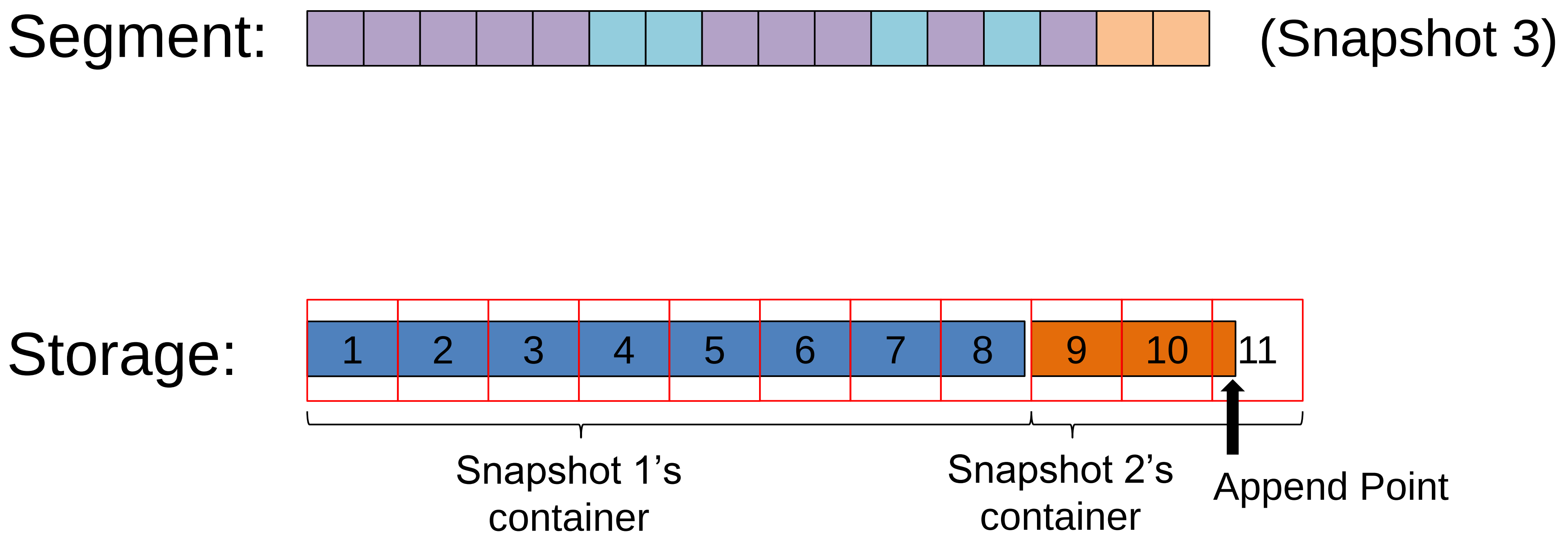
Forward Assembly Area Rolling Method



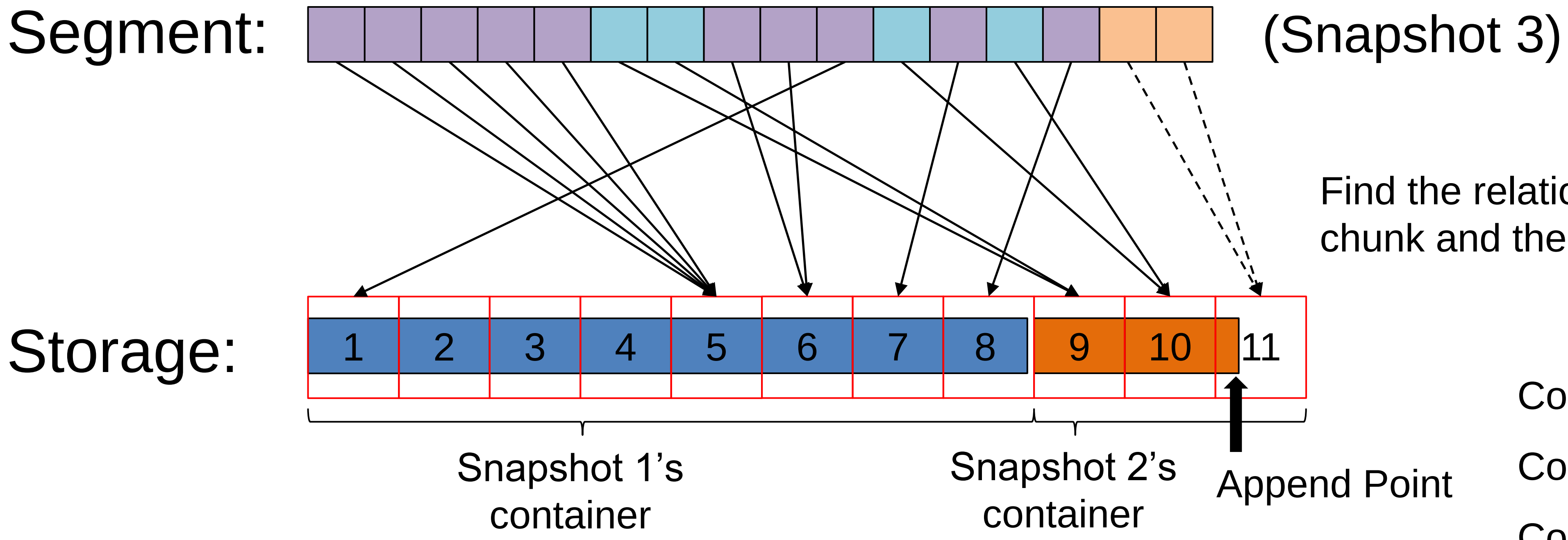
Limiting Fragmentation by Reducing Deduplication

- Goal: bound containers read per MB ingested
  - Guarantees minimal restore speed
- The capping method
  - Break input into 20 MB segments
  - Deduplicate segment against at most T containers
  - Limits fragmentation to (T+5)/20MB
    - Here we use 4 MB container, 5 = 20 MB/4MB (That is, the effective T=0 situation)
  - Minimize chunks duplicating by using best T containers
    - Select T according to the number of duplicate chunks contained in each container

Container Capping



Container Capping



Find the relationship between each chunk and the stored containers

Container ID : Number

Container 5: 5

Container 6: 2

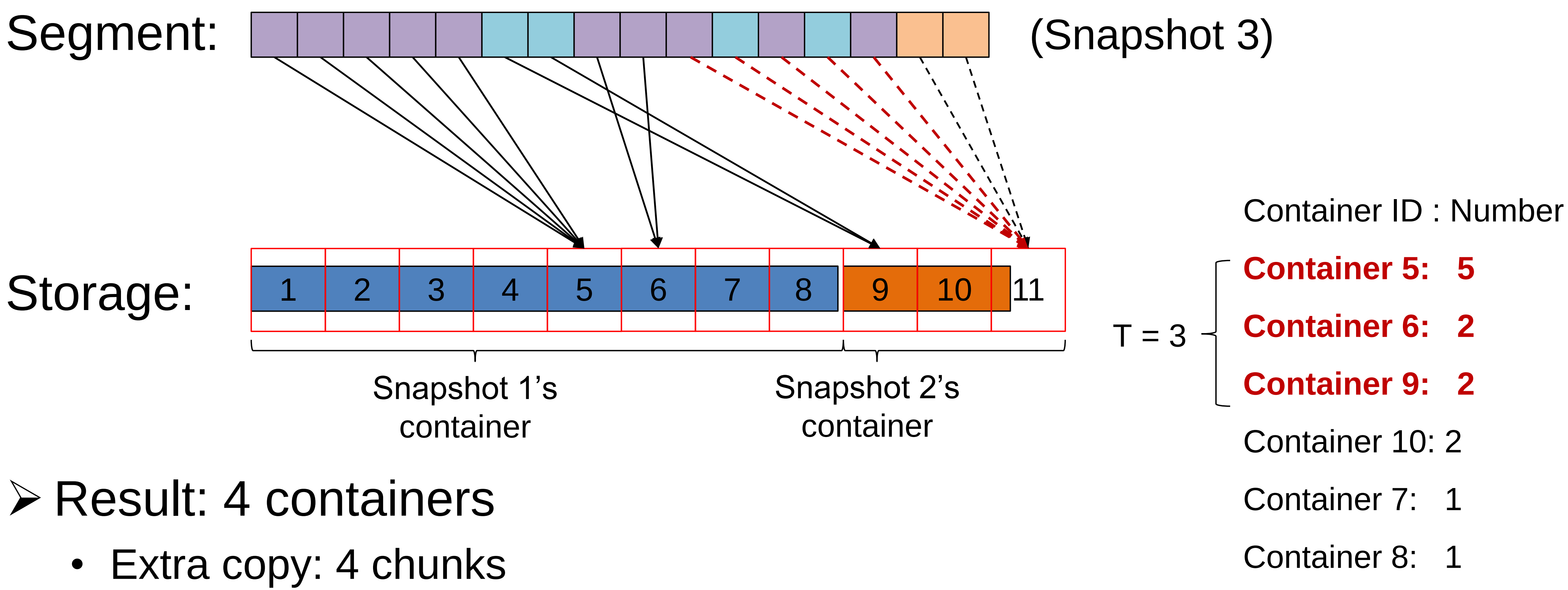
Container 9: 2

Container 10: 2

Container 7: 1

Container 8: 1

Container Capping



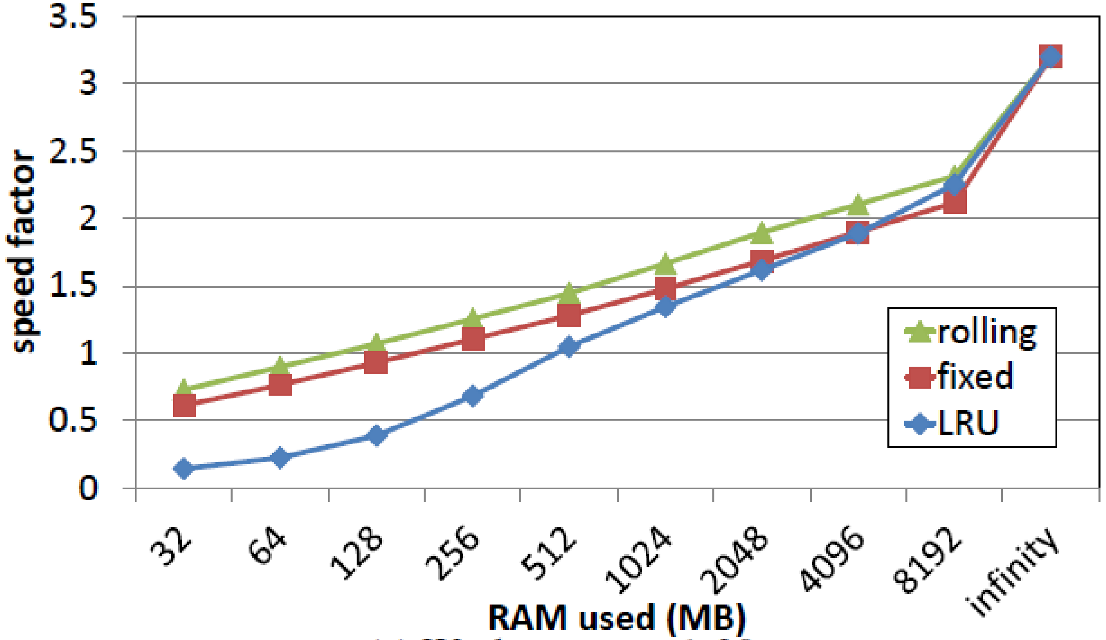
- Result: 4 containers
  - Extra copy: 4 chunks
  - No capping: restore need to access 8 containers

Evaluation

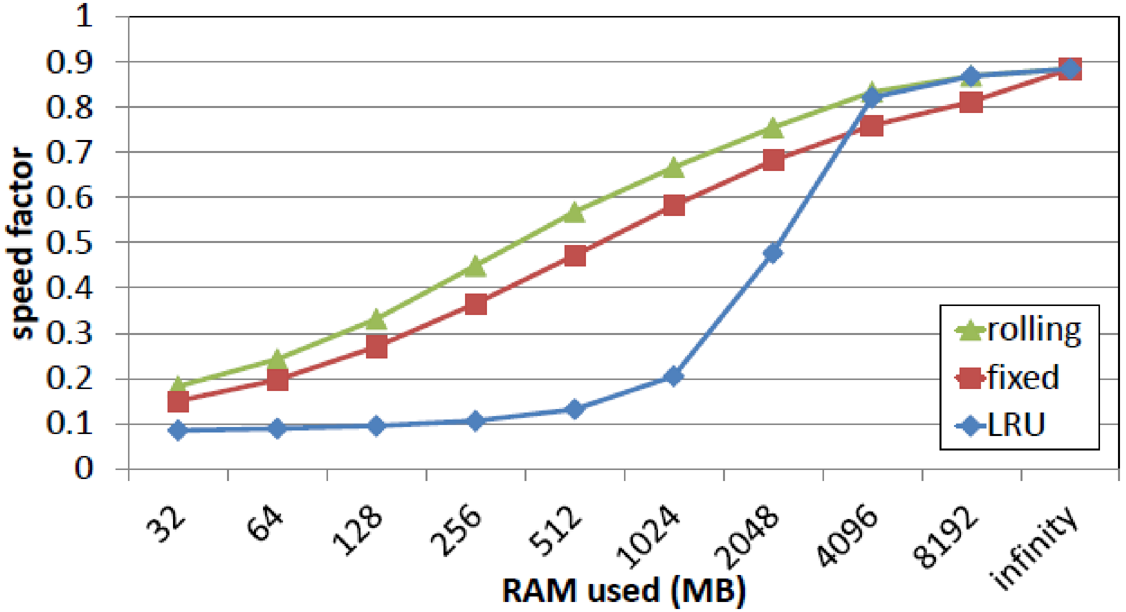
- Glossary:
  - Speed factor: 1/ container read per MB
  - Deduplication factor: logic size / stored size
- Dataset:
  - 2year 212 TB
    - Created based on HP customer data (10 GB snapshot), modify 2% of files by overwriting 10%, and add 200 MB data for each day.
    - 1 full + 4 incrementals per week for 2 years (480 backups)
    - ~4KB mean-size chunks
  - Workgroup 3.8 TB
    - Backups of 20 desktop PCs for 91 days.

Evaluation

- Assembly vs. LRU
  - The bigger the speed factor, the better



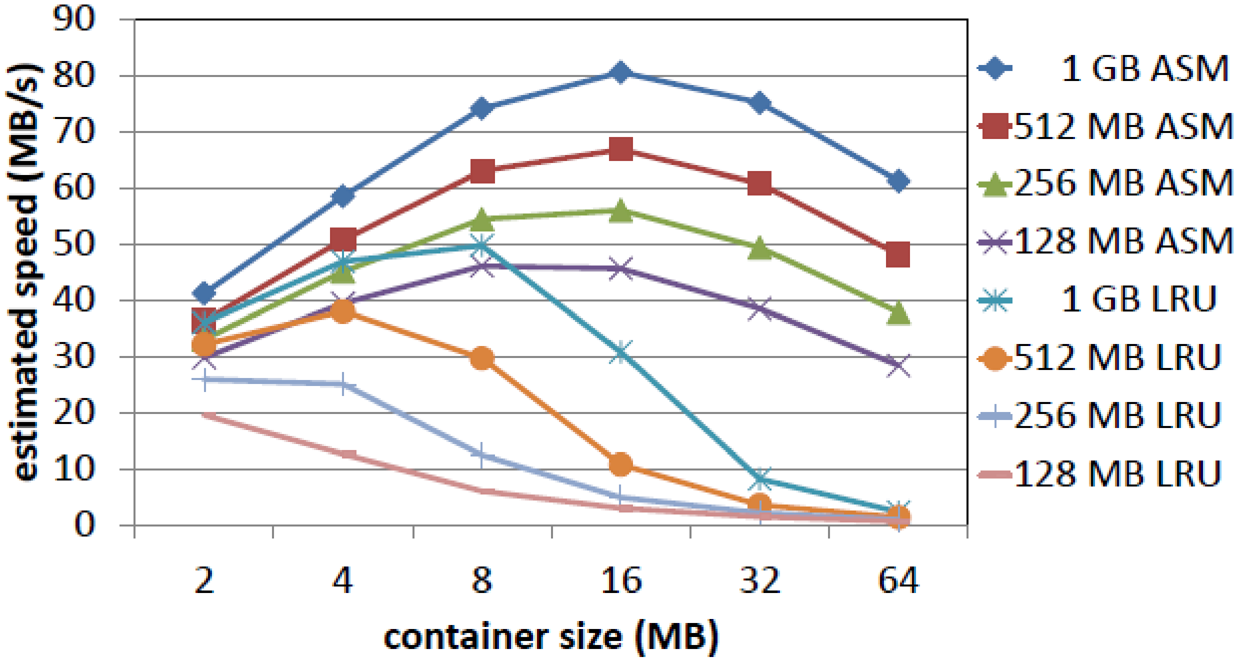
(a) Workgroup-cycle30



(b) 2year-cycle30

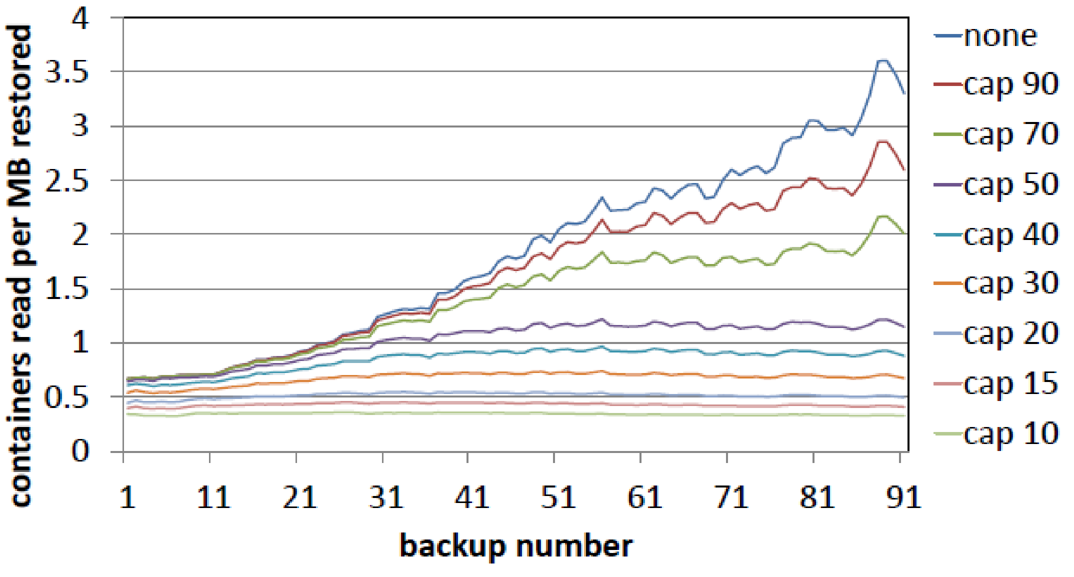
Evaluation

- Estimated speed for a sample system as container size is varied
  - Workgroup-cycle30
  - 4 cache sizes and assembly (rolling) and caching
  - Assume system reads at 1000 MB/s and opens a container in 20 ms
  - All containers were assumed to be full.
  - LRU collapses as container size grows

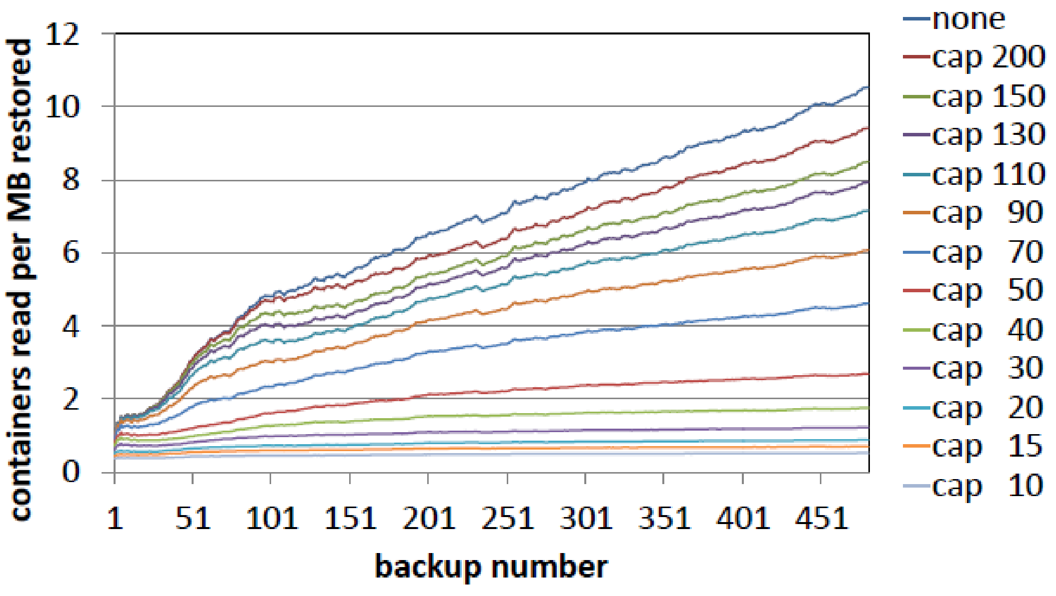


Evaluation

- Effect of varying capping level on fragmentation



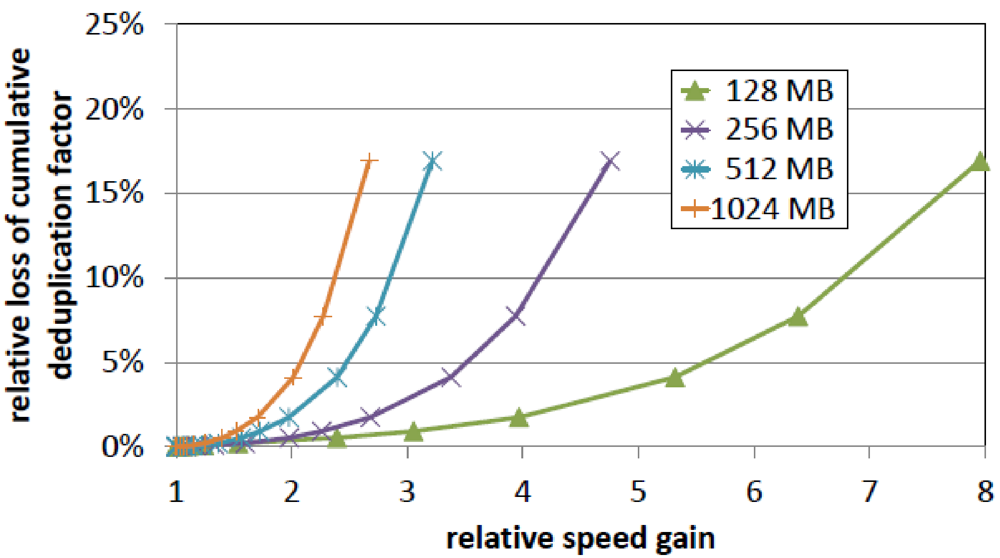
(a) Workgroup-cycle30



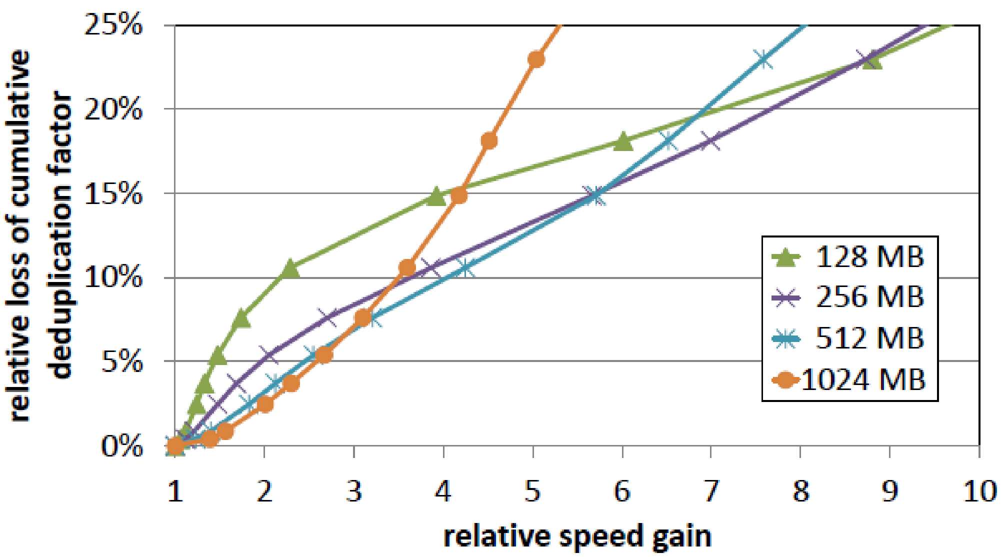
(b) 2year-cycle30

Evaluation

- Relative deduplication loss versus relative speed gain as a result of capping
  - T = 10, 15, 20, 30, 40, 50, 70, 90, 110, 130, 150, 200, 250, none



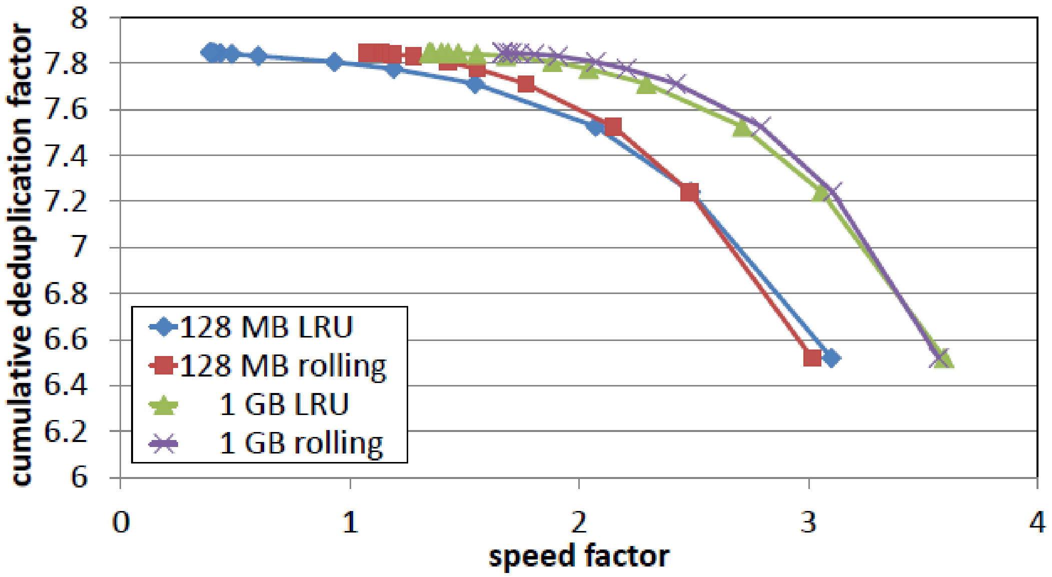
(a) Workgroup-cycle30



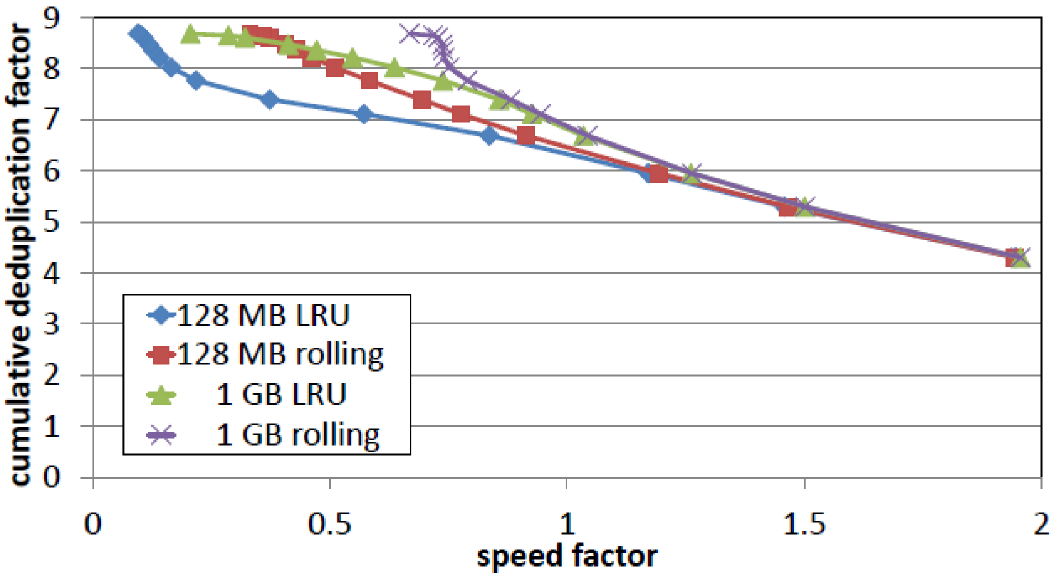
(b) 2year-cycle30

Evaluation

- Comparing deduplication and speed as capping level varies between LRU and assembly
  - T = 10, 15, 20, 30, 40, 50, 70, 90, 110, 130, 150, 200, 250, none



(a) Workgroup-cycle30



(b) 2year-cycle30

Conclusion

- Fragment analysis:
  - Reading containers is dominant restore cost
  - Measure fragment by container read per MB
- Caching: Reduce seeks through better caching
  - Forward assembly area
  - Load each container once per slice (100 MB)
- Capping: Limiting fragmentation by reducing deduplication
  - Deduplicate segment against at most T containers
  - Bound containers read per MB ingested