LightTx: A Lightweight Transactional Design in Flash-based SSDs to Support Flexible Transactions

Youyou Lu[1], Jiwu Shu[1], Jia Guo[1],
Shuai Li[1], Onur Mutlu[2]

[1]Tsinghua University
[2]Carnegie Mellon University

Executive Summary

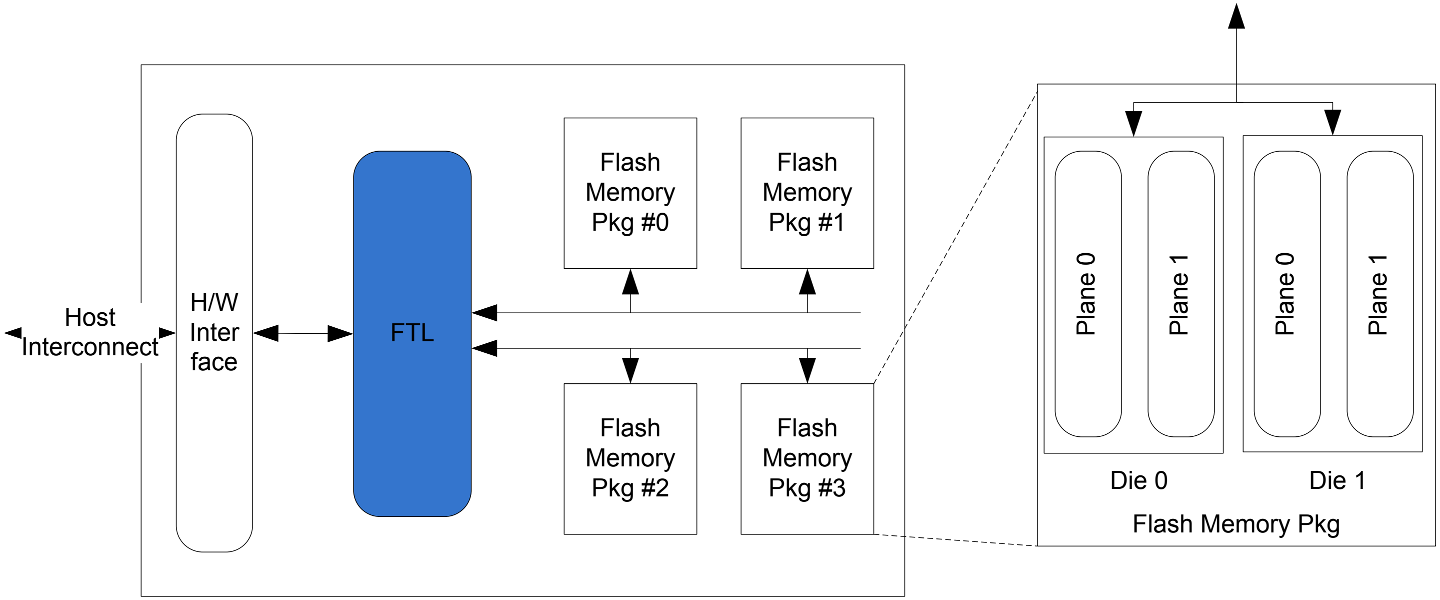
- Problem: Flash-based SSDs support transactions naturally (with out-of-place updates) but inefficiently:
  - Only a limited set of isolation levels are supported (inflexible)
  - Identifying transaction status is costly (heavyweight)
- Goal: a lightweight design to support flexible transactions
- Observations and Key Ideas:
  - Simultaneous updates can be written to different physical pages, and the FTL mapping table determines the ordering

    => (Flexibility) make commit protocol page-independent
  - Transactions have birth and death, and the near-logged update way enables efficient tracking

    => (Lightweight) track recently updated flash blocks, and retire the dead transactions
- Results: up to 20.6% performance improvement, stable GC overhead, fast recovery with negligible persistence overhead

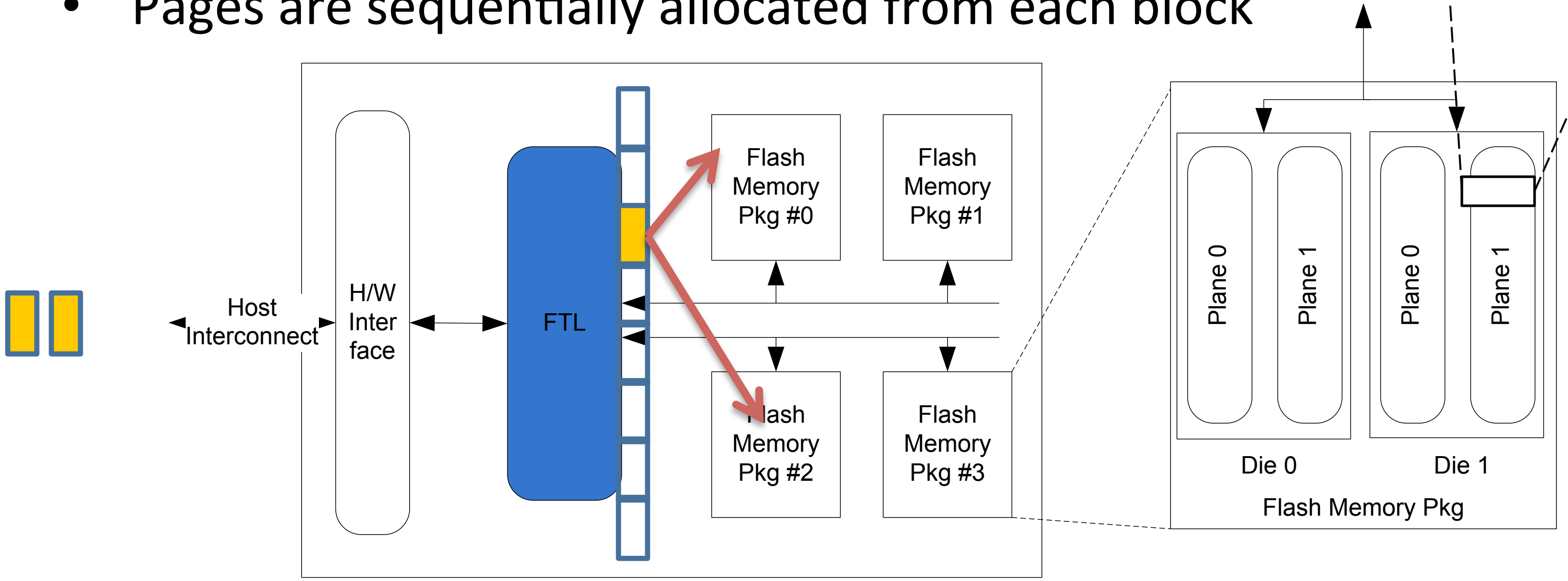
SSD Basics

- FTL (Flash Translation Layer)
  - Address mapping, garbage collection, wear leveling
- Out-of-place Update (address mapping)
  - Pages are updated to new physical pages instead of overwriting original pages
- Internal Parallelism
  - New pages are allocated from different pkgs/planes
- Page metadata (OOB): (4096 + 224)Bytes



Two Observations

- Simultaneous updates and FTL ordering
  - (Out-of-place update) pages for the same LBA can be updated simultaneously
  - (Ordering in mapping table) Only when the mapping table is updated, the write is visible to the external
- Near-logged update way
  - Pages are allocated from blocks over different parallel units
  - Pages are sequentially allocated from each block

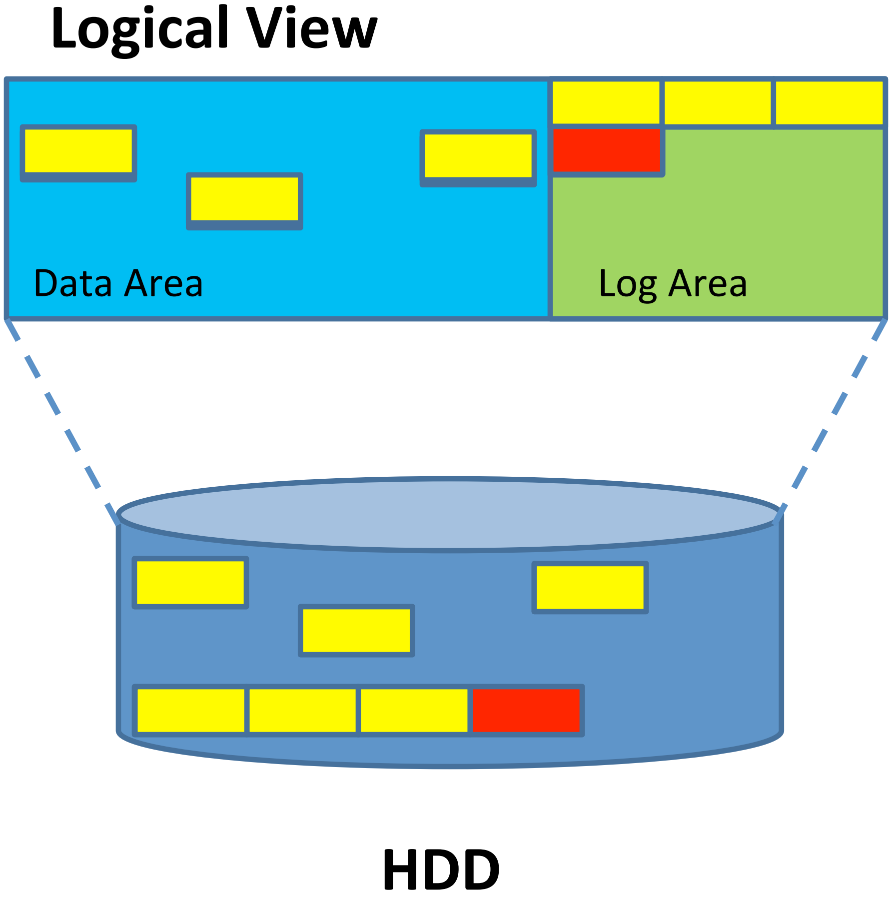


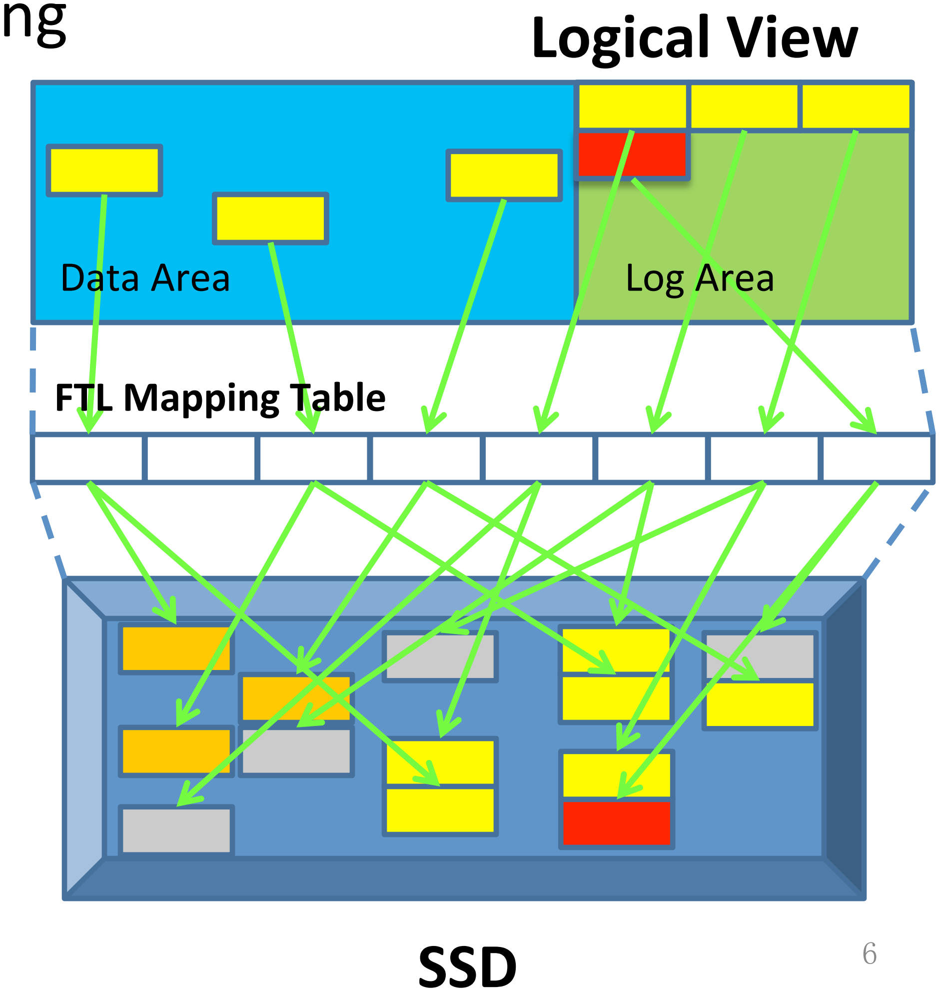
Outline

- Executive Summary
- Background
  - Traditional Software Transactions
  - Existing Hardware Transactions
- LightTx Design
- Evaluation
- Conclusions

Traditional S/W Transaction

- Transaction: Atomicity and Durability
- Software Transaction
  - Duplicate writes
  - Synchronization for ordering





We have both old and new versions in the SSD (out-of-place update).

Why shall we write the log?

Why not support transactions inside the SSD?

Existing H/W Approaches

- Atomic-Write [HPCA'11]
  - Log-structured FTL
  - Commit protocol: Tag the last page "1", while the others "0"
  - Limited Parallelism: one tx at a time
  - High mapping persistence overhead: persistence on each commit
- SCC/BPCC (Cyclic commit protocols) [OSDI'08]
  - Commit Protocol: Link all flash pages in a cyclic list by keeping pointers in page metadata
  - High overhead in differentiate broken cyclic lists for *partial erased committed txs* and *aborted txs*
    - SCC forces aborted pages erased before writing the new one
    - BPCC delays the erase of pages to its previous aborted pages are erased
  - Limited Parallelism: txs without overlapped accesses are allowed

[HPCA'11] Beyond block i/o: Rethinking traditional storage primitives
[OSDI'08] Transactional flash

Problems:

- Tx support is inflexible (limited parallelism)
  - Cannot meet the flexible demands from software
  - Cannot fully exploit the internal parallelism of SSDs
- Tx state tracking causes high overhead in the device

Our Goal:

A lightweight design
to support flexible transactions

Outline

- Executive Summary
- Background
- LightTx Design
  - Design Overview
  - Page Independent Commit Protocol
  - Zone-based Transaction State Tracking
- Evaluation
- Conclusions

Goal

A lightweight design
to support flexible transactions

Flexible

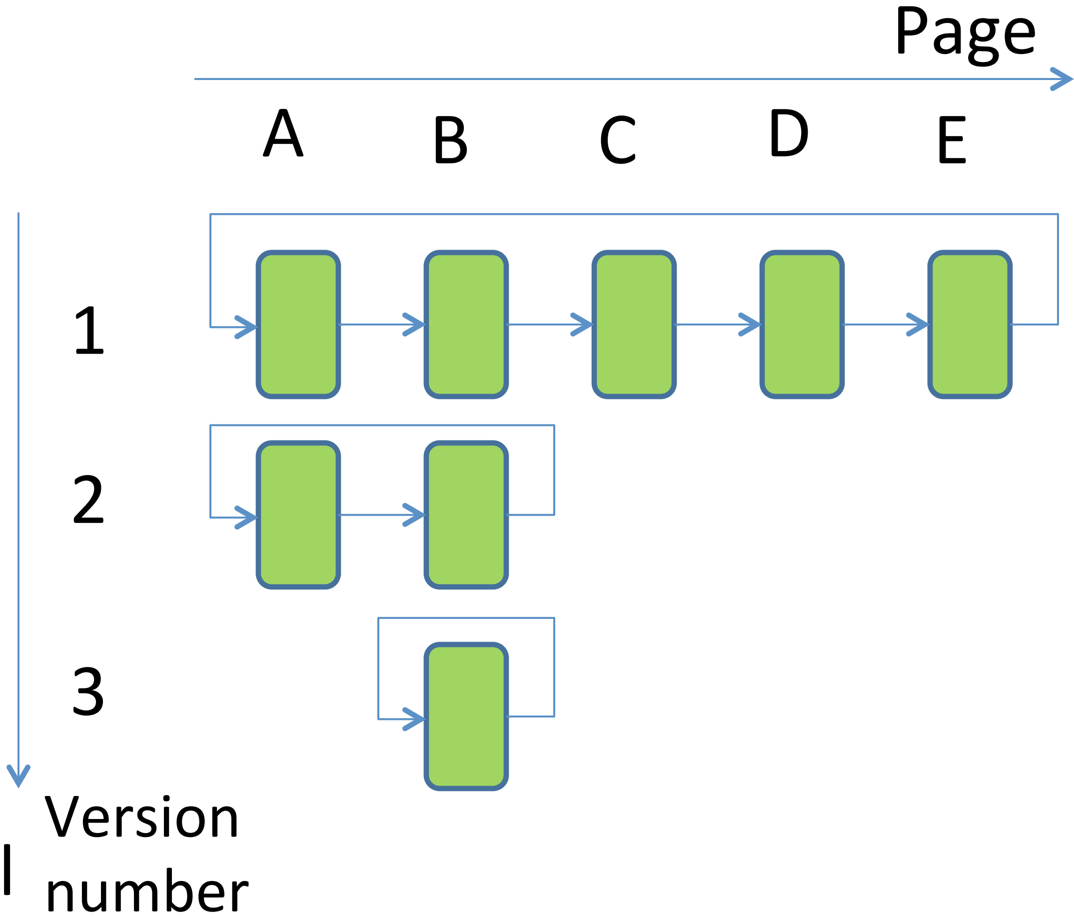
- Page-independent commit protocol: support simultaneous updates, to enable flexible isolation level choices in the system

Lightweight

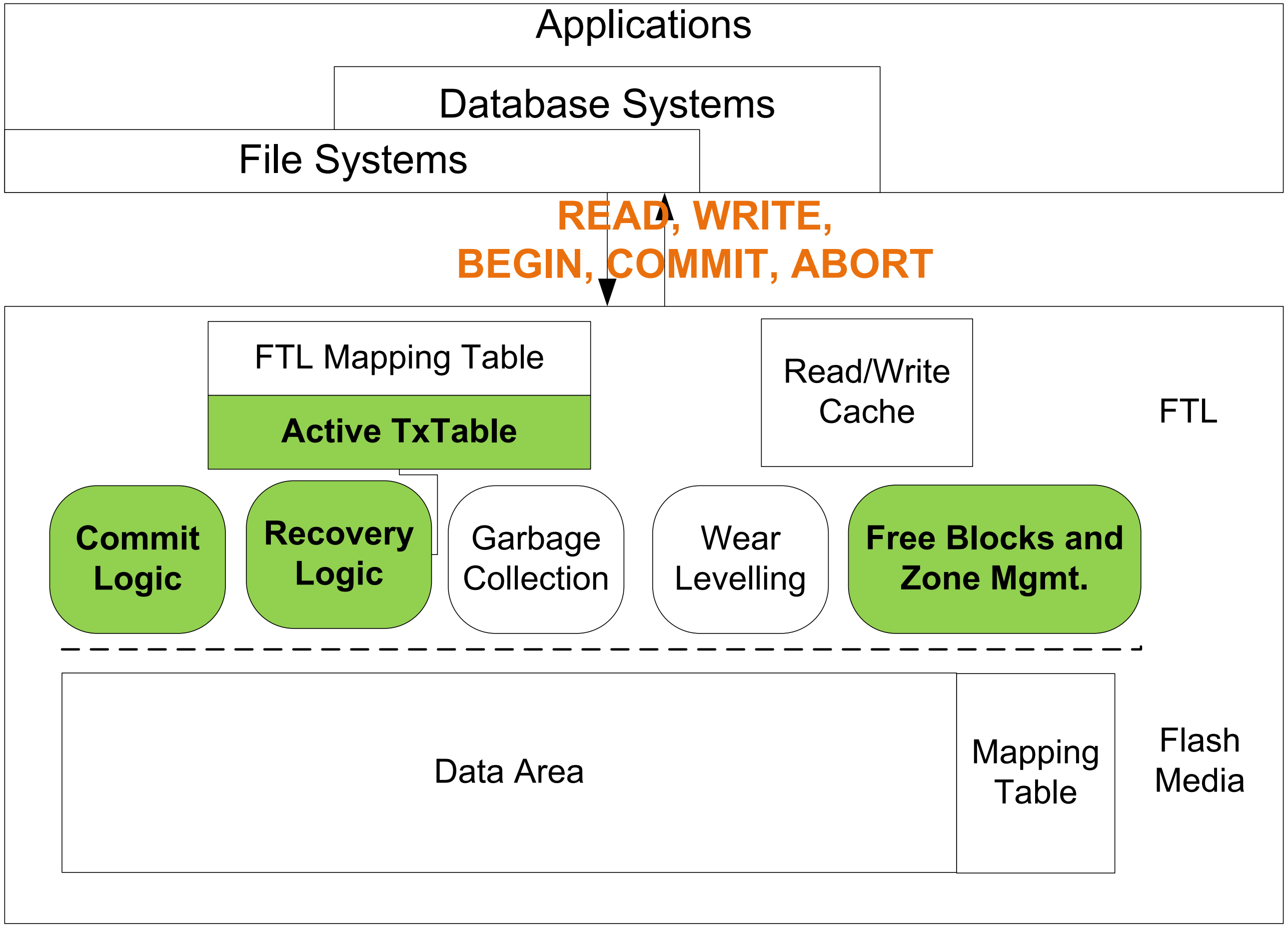
- Zone-based transaction state tracking scheme: track only blocks that have live txs and retire the dead ones, to reduce lower the cost

Page-independent Commit Protocol

- Observations:
  - Simultaneous Updates (Out-of-place update)
  - Version order (FTL mapping table)
- How to support this?
  - Extend the storage interface
  - Make commit protocol page-independent



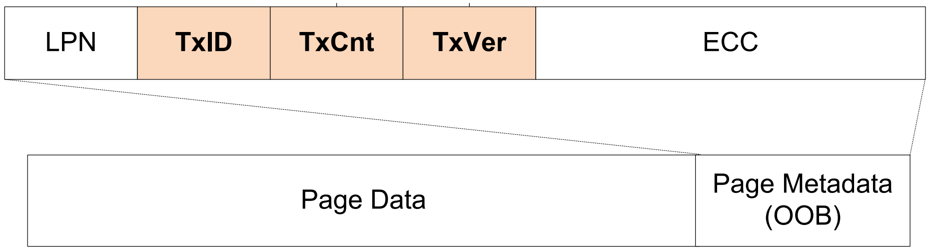
Design Overview

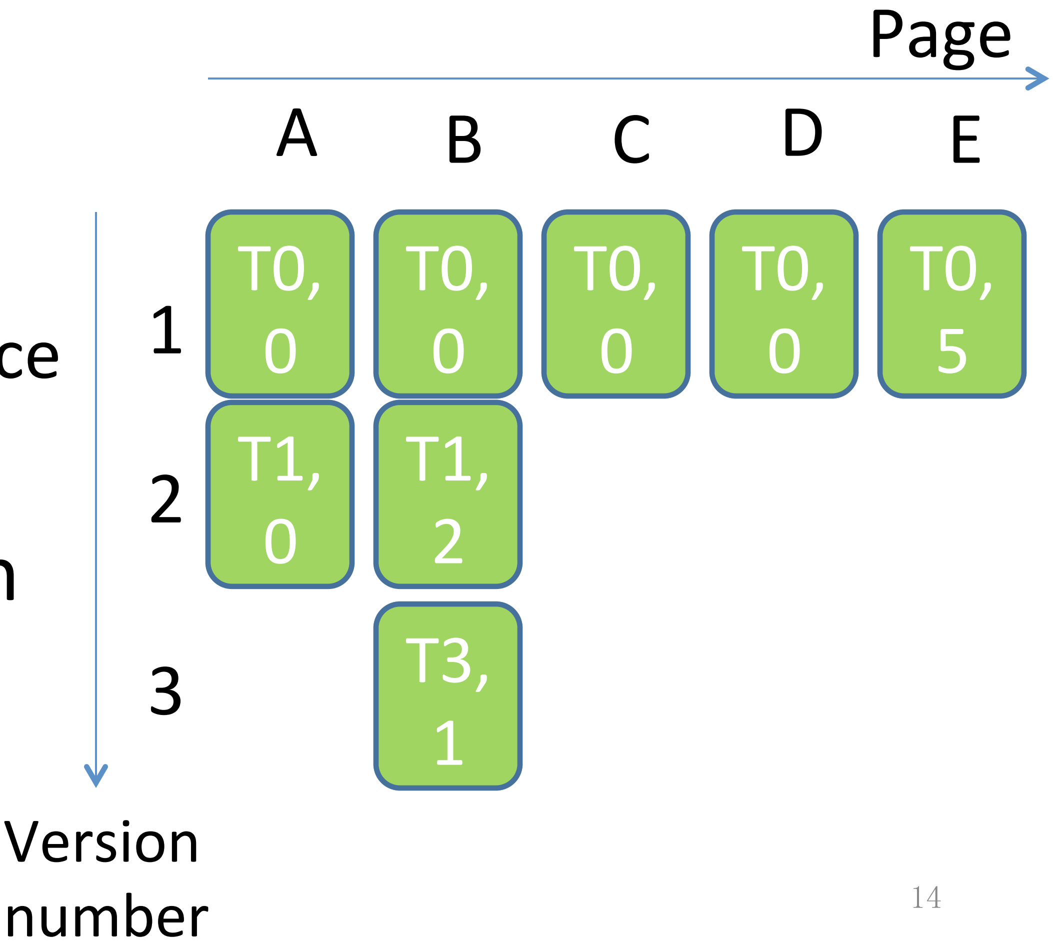


- Transaction Primitive
  - BEGIN(*TxID*)
  - COMMIT(*TxID*)
  - ABORT(*TxID*)
  - WRITE(*TxID, LBA, len …*)

Page-independent Commit Protocol

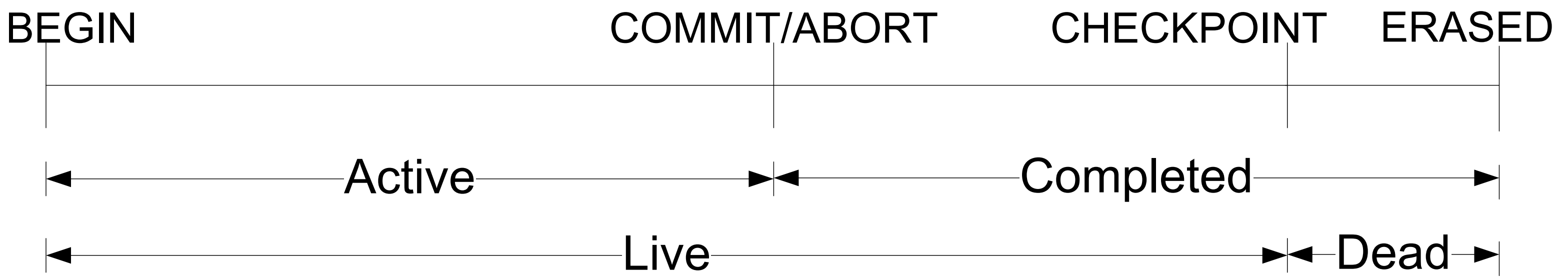
- Transactional metadata: <TxID, TxCnt, TxVer>
  - TxID
  - TxCnt: (00…0N)
  - TxVer: commit sequence
- Keep it in the page metadata of each flash page


LPN
TxID
TxCnt
TxVer
ECC
Page Data
Page Metadata (OOB)


Page
A
B
C
D
E
1
T0, 0
T0, 0
T0, 0
T0, 0
T0, 5
2
T1, 0
T1, 2
3
T3, 1
Version number

Zone-based Transaction State Tracking

- Transaction Lifetime



BEGIN
COMMIT/ABORT
CHECKPOINT
ERASED

Active
Completed
Live
Dead

  - Retire the dead: write back the mapping table, and remove the dead from tx state maintenance

Can we write back the mapping back for each commit?

- Ordering cost (waiting for mapping table persistence)
- Mapping persistent is not atomic

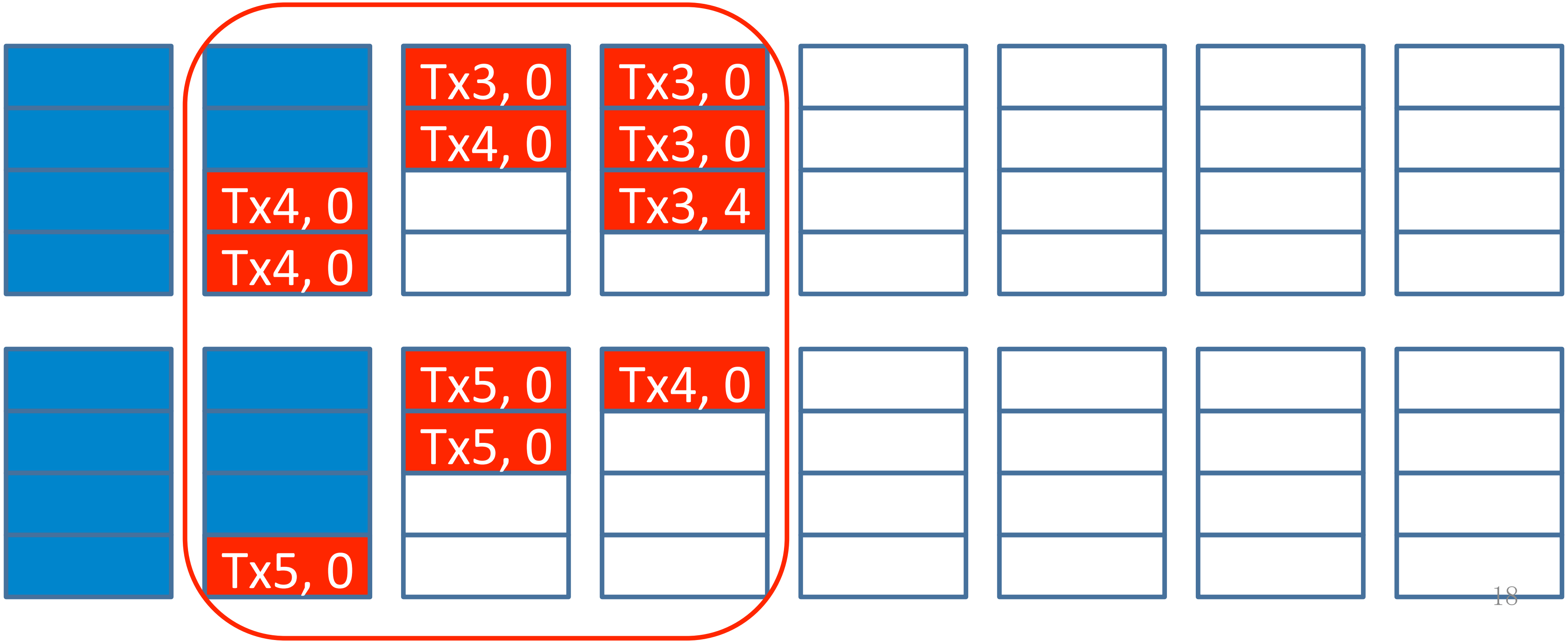
- Writes appended in the free flash blocks
  - Track the recently updated flash blocks

- Block Zones
  - Free block: all pages are free
  - Available block: pages are available for allocation
  - Unavailable block: all pages have been written to but some pages belong to (1) a live tx, or (2) a dead tx but has at least one page in some available block
  - Checkpointed block: all pages have been written to and all pages belong to dead txs
- Respectively, we have Free, Available, Unavailable and Checkpointed Zones.

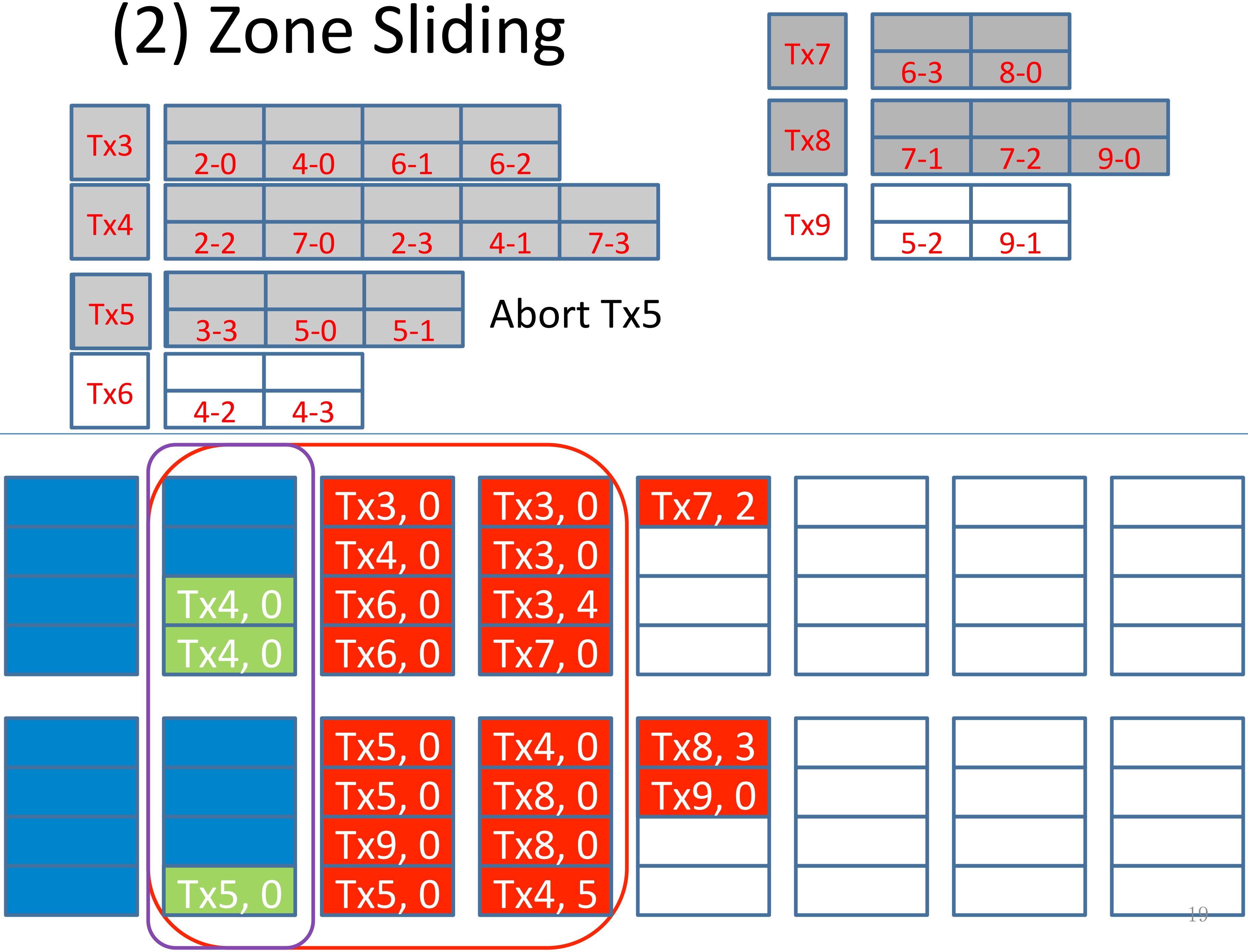
- Checkpoint
  - Periodically write back the mapping table (making the txs dead)
  - And, sliding the zones (available + unavailable)
- Zone Sliding
  - Check all blocks in available and unavailable zones
    - Move the block to the checkpointed zone if the block is checkpointed
    - Move the block to the unavailable zone if the block is unavailable
  - Pre-allocate free blocks to the available zone
  - Garbage collection is only performed on the checkpointed zone

(1) Available Zone Updating

Tx3	2-0	4-0	6-1	6-2
Tx4	2-2	7-0	2-3	4-1
Tx5	3-3	5-0	5-1	

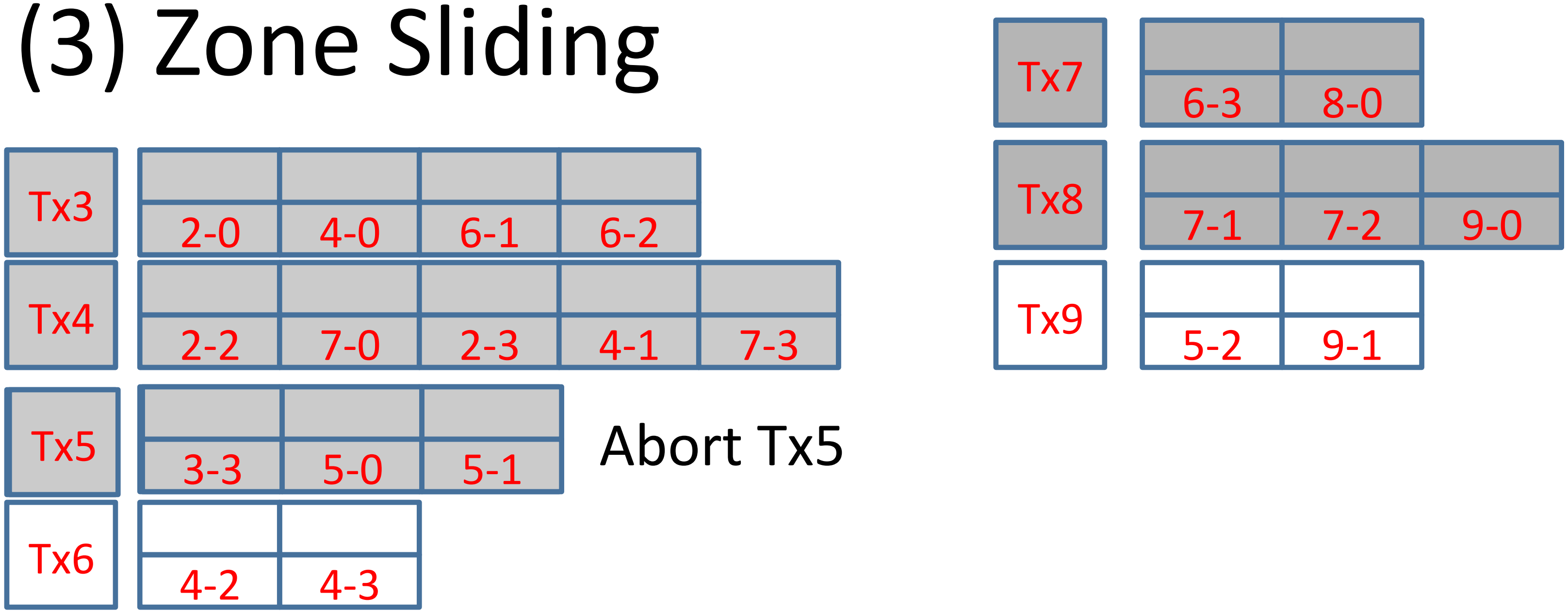


(2) Zone Sliding

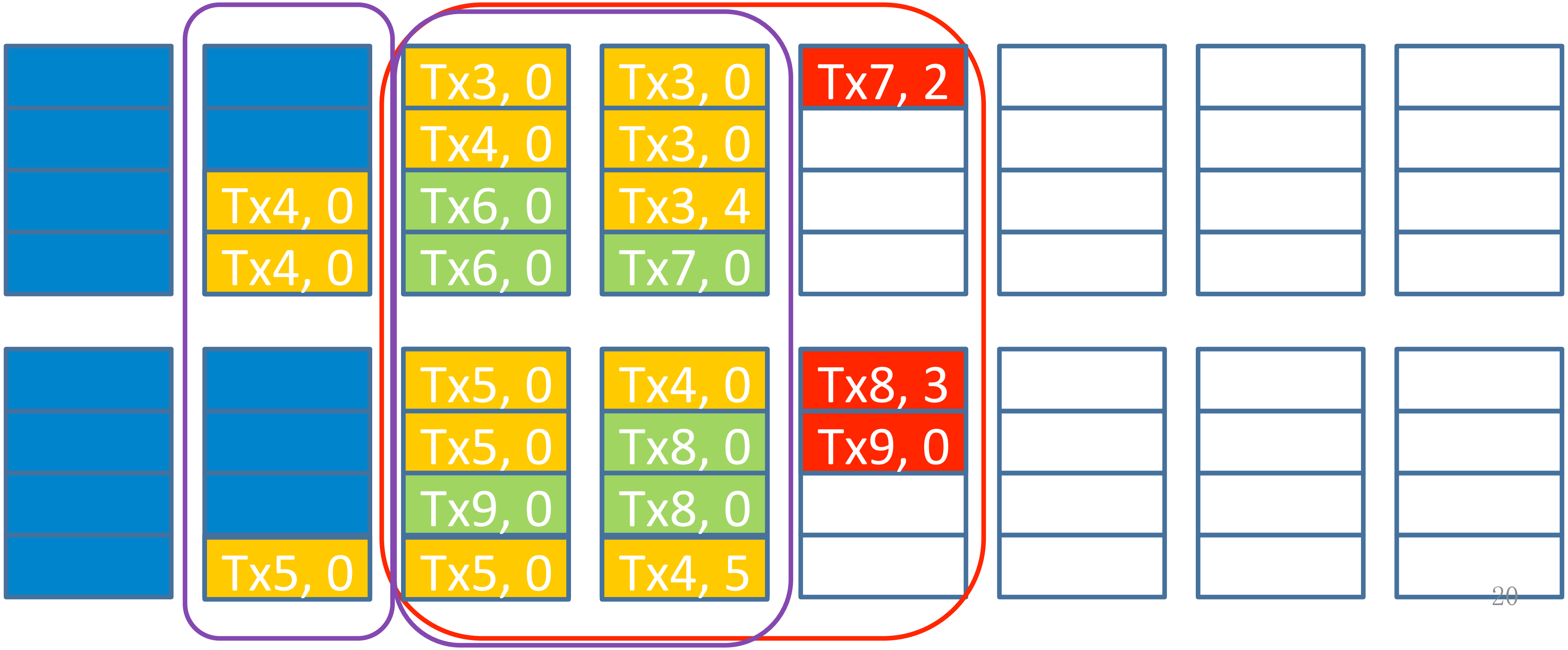


Abort Tx5

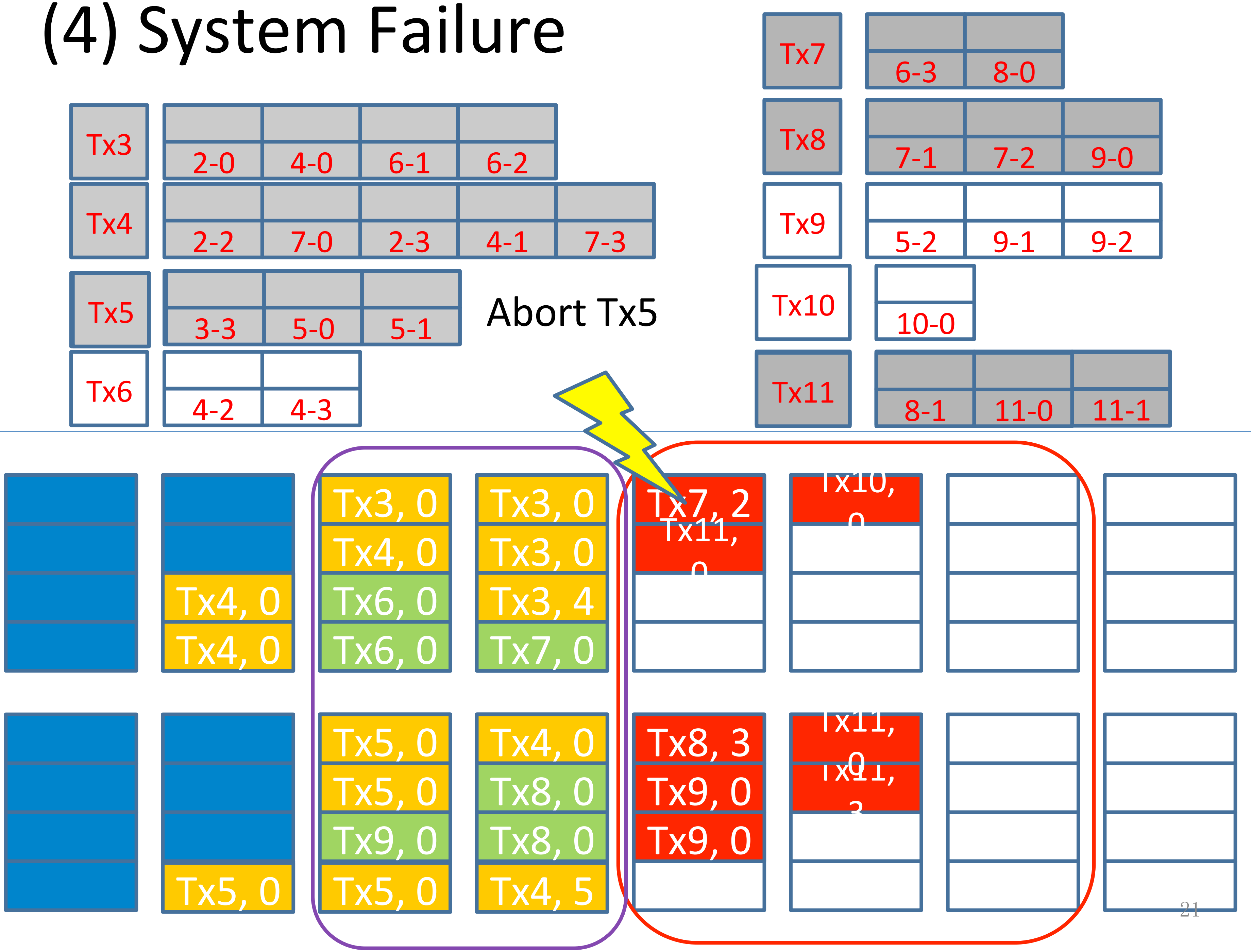
(3) Zone Sliding



Abort Tx5



(4) System Failure



Tx3	2-0	4-0	6-1	6-2	
Tx4	2-2	7-0	2-3	4-1	7-3
Tx5	3-3	5-0	5-1		
Tx6	4-2	4-3			
Tx7	6-3	8-0			
Tx8	7-1	7-2	9-0		
Tx9	5-2	9-1	9-2		
Tx10	10-0				
Tx11	8-1	11-0	11-1		

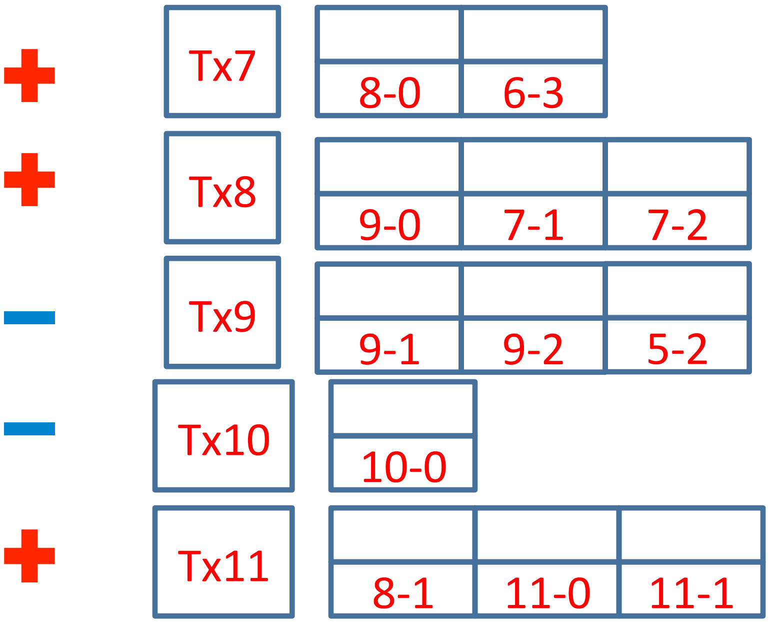
Abort Tx5

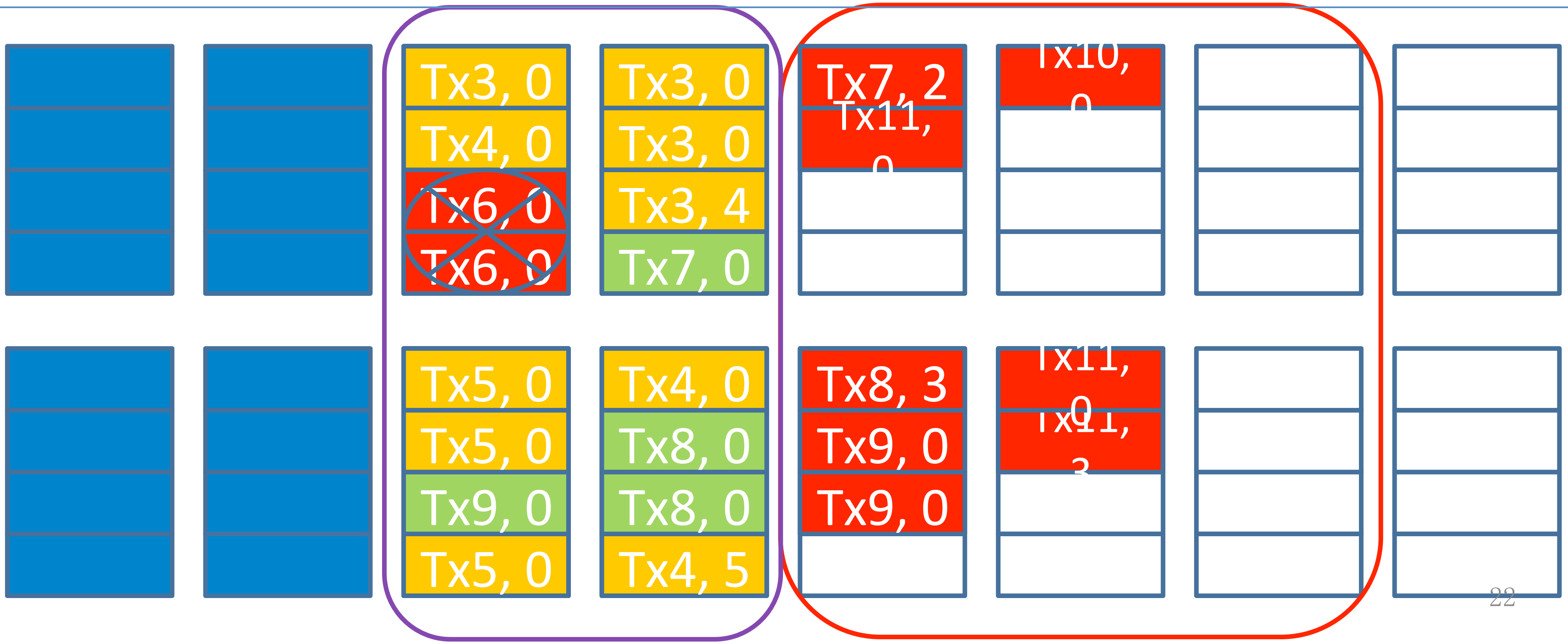
		Tx3, 0	Tx3, 0	Tx7, 2	Tx10, 0		
		Tx4, 0	Tx3, 0	Tx11, 0			
	Tx4, 0	Tx6, 0	Tx3, 4				
	Tx4, 0	Tx6, 0	Tx7, 0				

		Tx5, 0	Tx4, 0	Tx8, 3	Tx11, 0		
		Tx5, 0	Tx8, 0	Tx9, 0	Tx11, 3		
		Tx9, 0	Tx8, 0	Tx9, 0			
	Tx5, 0	Tx5, 0	Tx4, 5				

Recovery

- Scan the available zone
- Scan the unavailable zone





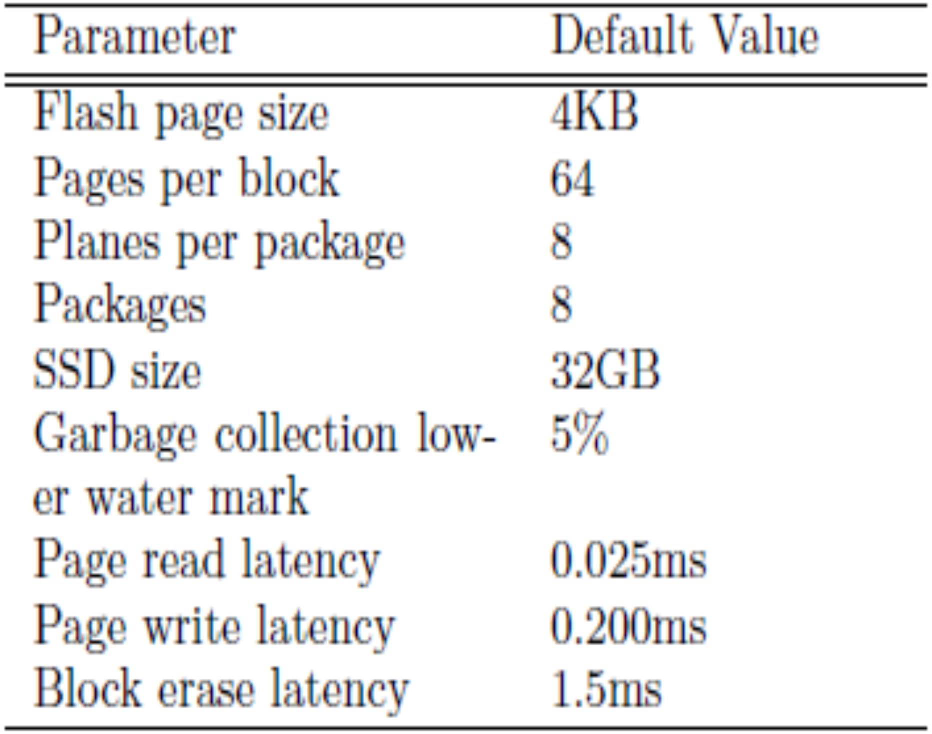
- Recovery
  - Scan the available zone
    - If TxCnt matches, completed tx
    - If not, add the tx to the pending list
  - Scan the unavailable zone
    - If TxID in the pending list, check TxCnt again. If TxCnt matches, completed tx
    - If txID not in the pending list, discard it
    - If TxCnt still doesn’t match, uncompleted tx
  - Replay with the sequence of TxVer

Outline

- Executive Summary
- Background
- LightTx Design
- Evaluation
- Conclusions

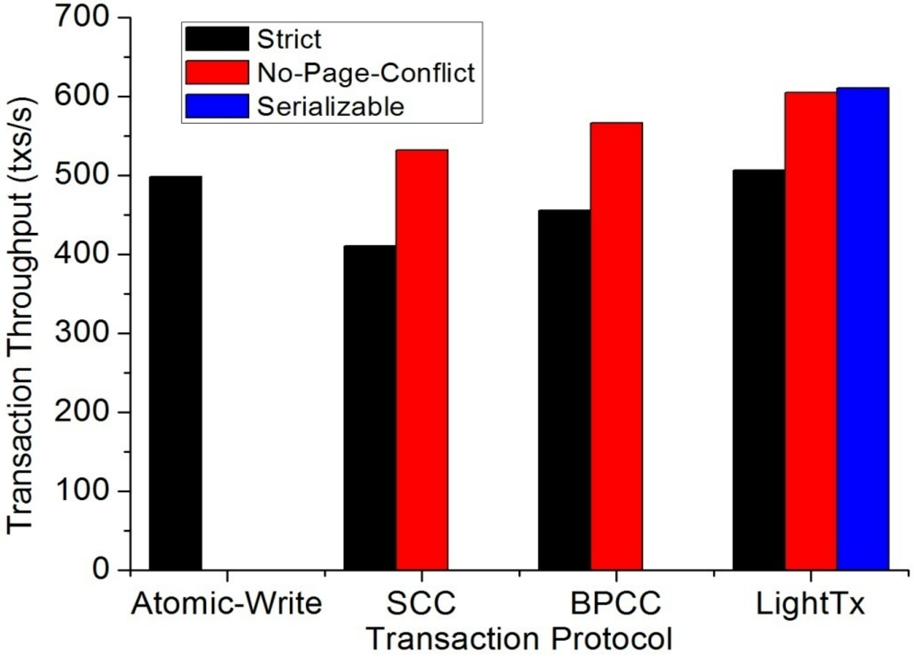
Experimental Setup

- SSD simulator
  - SSD add-on from Microsoft on DiskSim
  - Parameters from Samsung K9F8G08UXM NAND flash
- Trace
  - TPC-C benchmark: DBT2 on PostgreSQL 8.4.10



Parameter	Default Value
Flash page size	4KB
Pages per block	64
Planes per package	8
Packages	8
SSD size	32GB
Garbage collection lower water mark	5%
Page read latency	0.025ms
Page write latency	0.200ms
Block erase latency	1.5ms

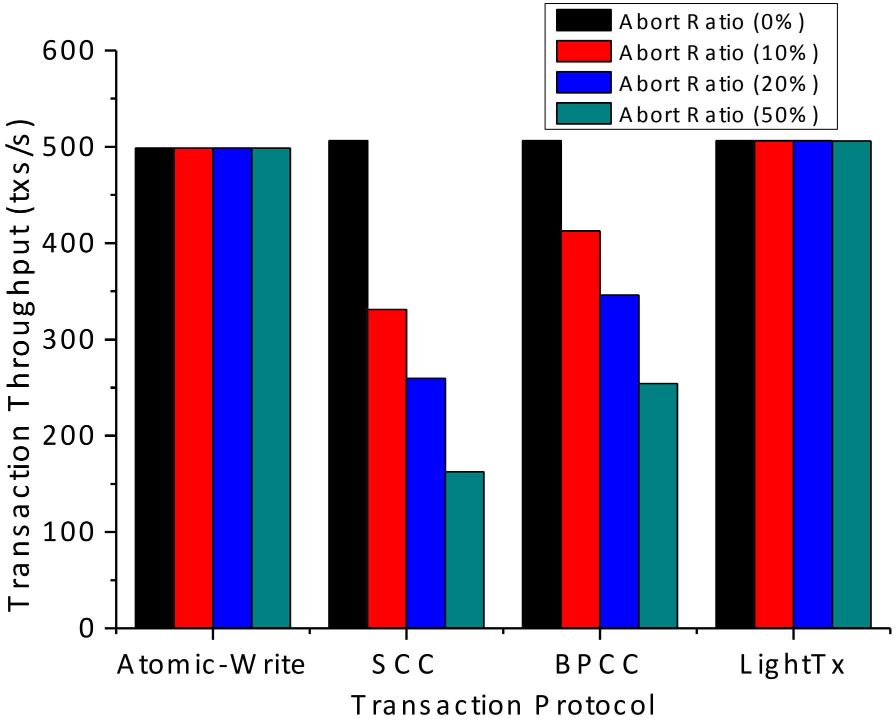
Flexibility

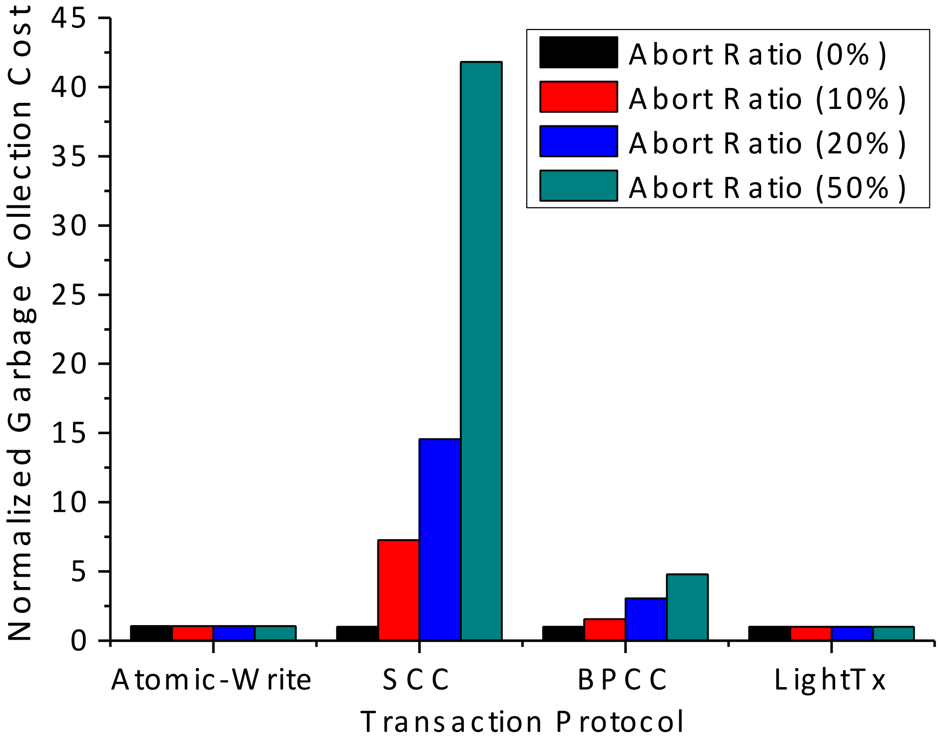


(1) For a given isolation level, LightTx provides as good or better tx throughput than other protocols.

(2) In LightTx, no-page-conflict and serialization isolation improve throughput by 19.6% and 20.6% over strict isolation.

Garbage Collection Cost

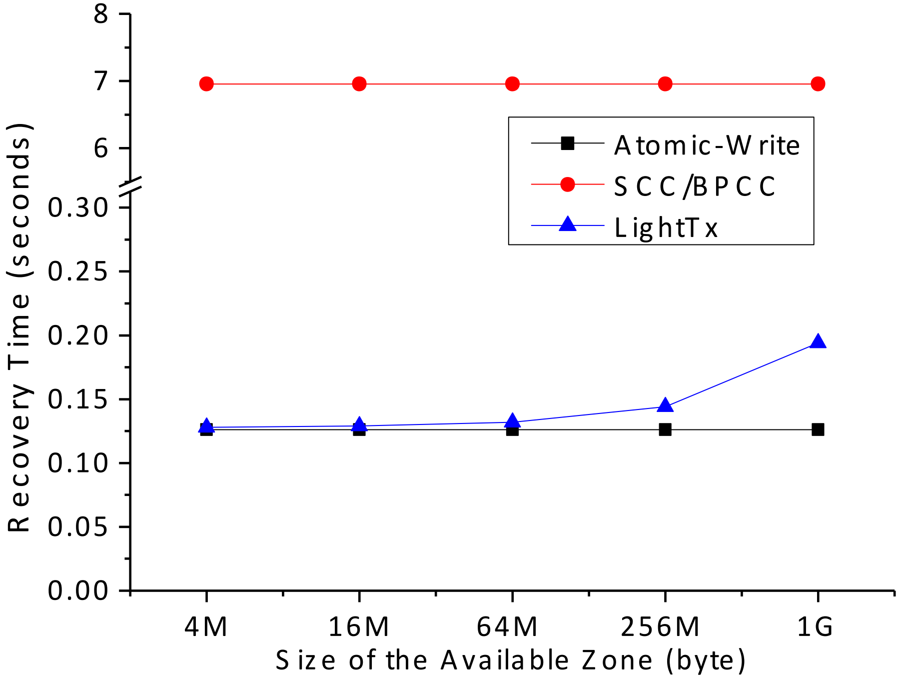


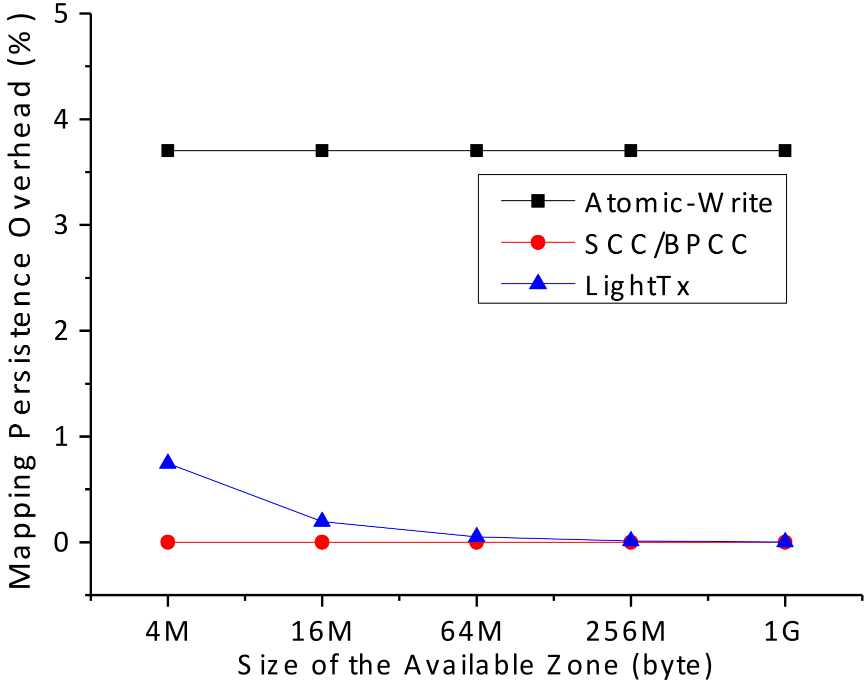


(1) LightTx significantly outperforms SCC/BPCC when abort ratio is not zero.

(2)Garbage collection overhead in SCC/BPCC goes extremely high when abort ratio goes up.

Recovery Time
and Persistence Overhead





LightTx achieves fast recovery with low mapping persistence overhead.

Outline

- Executive Summary
- Background
- LightTx Design
- Evaluation
- Conclusions

Conclusion

- Problem: Flash-based SSDs support transactions naturally (with out-of-place updates) but inefficiently:
  - Only a limited set of isolation levels are supported (inflexible)
  - Identifying transaction status is costly (heavyweight)
- Goal: a lightweight design to support flexible transactions
- Observations and Key Ideas:
  - Simultaneous updates can be written to different physical pages, and the FTL mapping table determines the ordering

    => (Flexibility) make commit protocol page-independent
  - Transactions have birth and death, and the near-logged update way enables efficient tracking

    => (Lightweight) track recently updated flash blocks, and retire the dead transactions
- Results: up to 20.6% performance improvement, stable GC overhead, fast recovery with negligible persistence overhead

Thanks

LightTx: A Lightweight Transactional Design in Flash-based SSDs to Support Flexible Transactions

Youyou Lu[1], Jiwu Shu[1], Jia Guo[1],
Shuai Li[1], Onur Mutlu[2]

[1]Tsinghua University
[2]Carnegie Mellon University

Backup Slides

Existing Approaches (1)

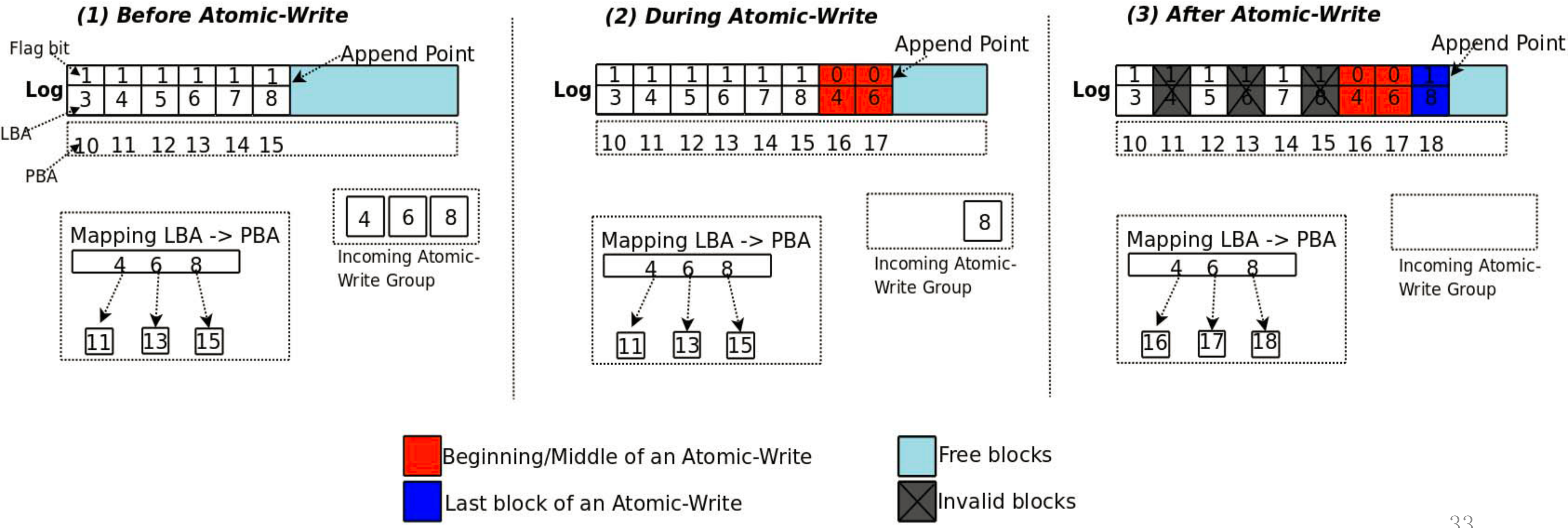
- Atomic Writes
  - Log-structured FTL
  - Transaction state: <00…1>

+ No logging, no commit record

+ No tx state maintenance cost

- Poor parallelism

- Mapping persistence overhead

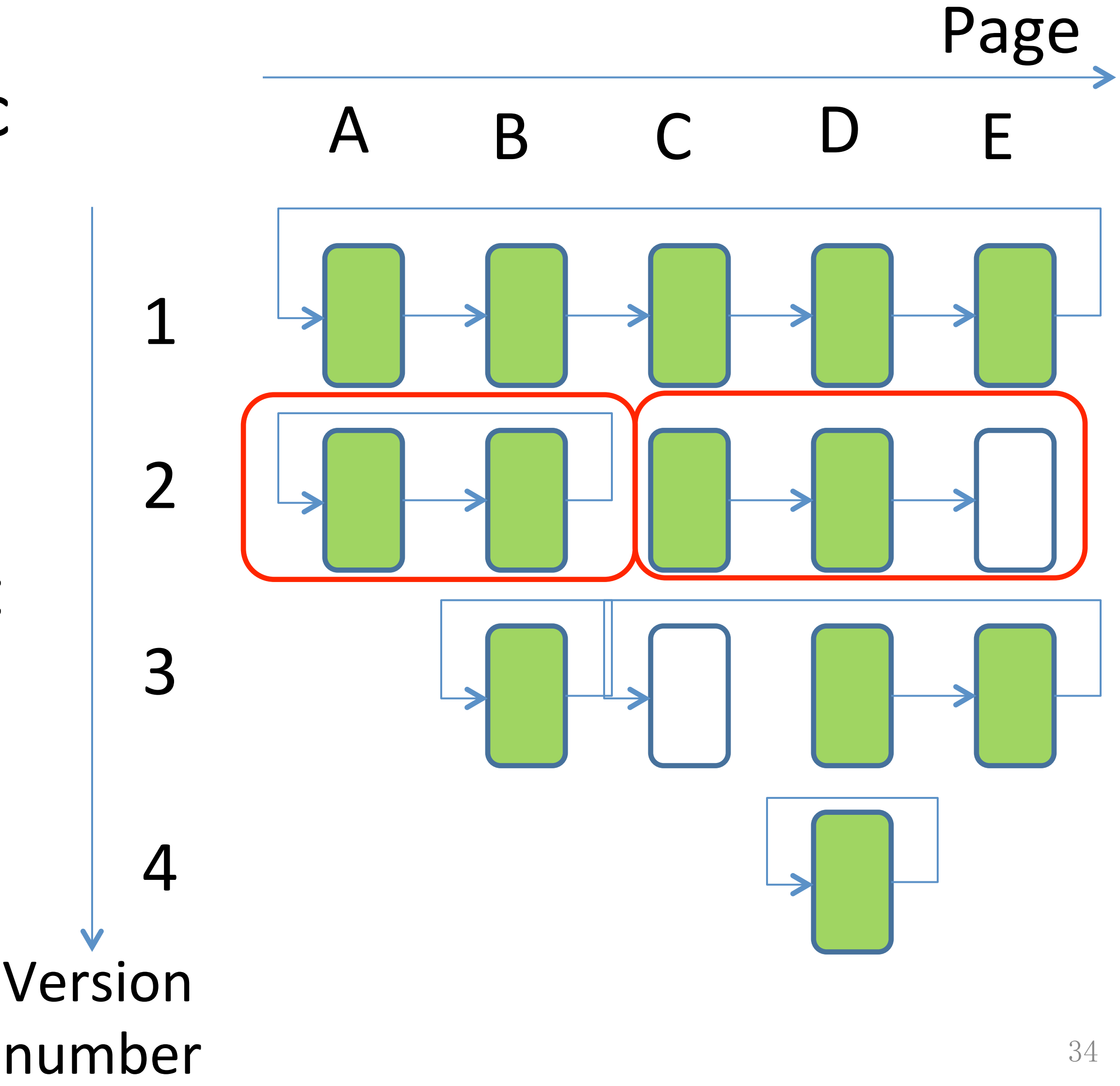


[HPCA'11] Beyond block i/o: Rethinking traditional storage primitives

Existing Approaches (2)

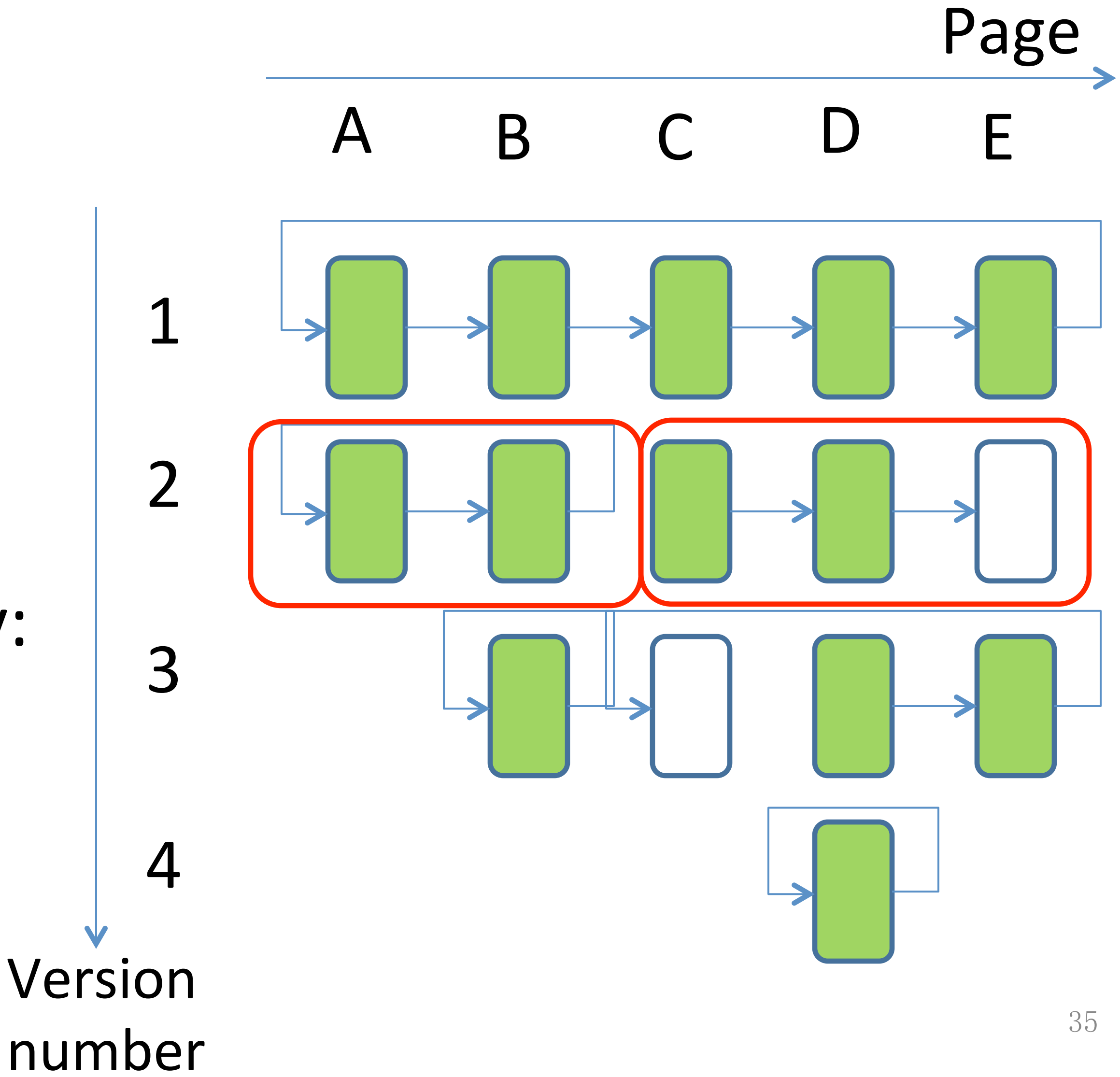
- SCC/BPCC (Cyclic commit protocol)
  - Use pointers in the page metadata to put all pages in a cycle for each tx

[OSDI'08] Transactional flash



- SCC
  - Block erase forced for aborted pages

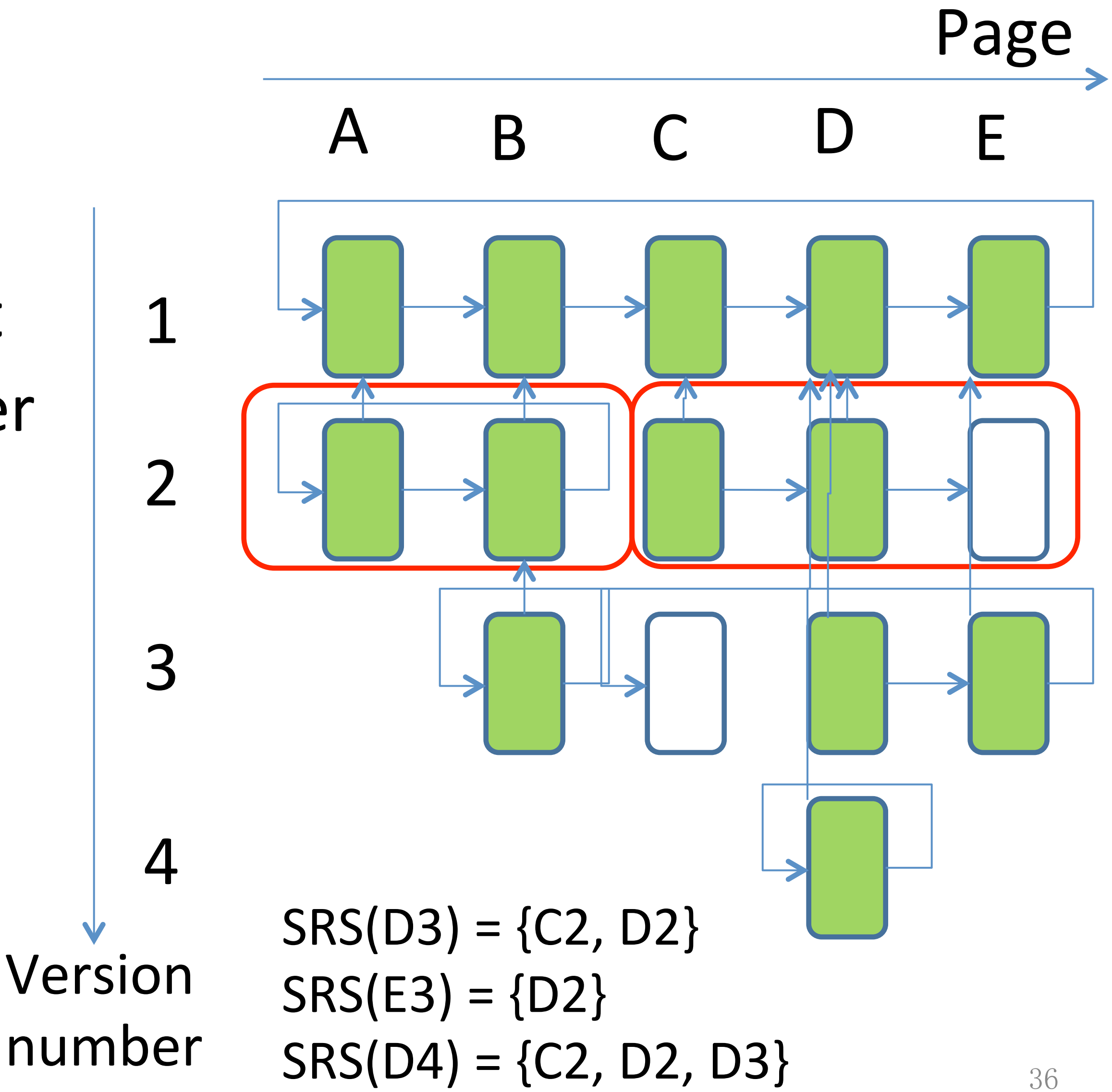
- Low garbage collection efficiency: lots of data moves due to forced block erase



- BPCC
  - SRS: Straddle Responsibility Set
  - Erasable only after SRS is empty

\- Complex and costly SRS updates

\- Low garbage collection efficiency: wait until SRS is empty



SRS(D3) = {C2, D2}

SRS(E3) = {D2}

SRS(D4) = {C2, D2, D3}

Atomic Writes

+ No logging, no commit record

+ No tx state maintenance overhead

- Poor parallelism

- Mapping persistence overhead

SCC/BPCC

+ No logging, no commit record

+ Improved parallelism

- Limited parallelism

- High tx state maintenance overhead

Problems:

- Tx support is inflexible (limited parallelism)
  - Cannot meet the flexible demands from software
  - Cannot fully exploit the internal parallelism of SSDs
- Tx state tracking causes high overhead in the device

A lightweight design

to support flexible transactions