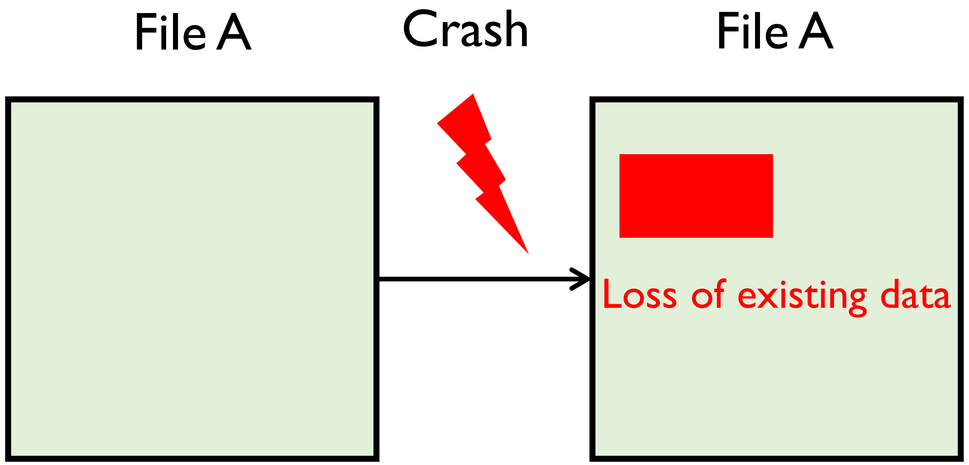
An Efficient Order-Preserving Recovery for F2FS with ZNS

HotStorage'23

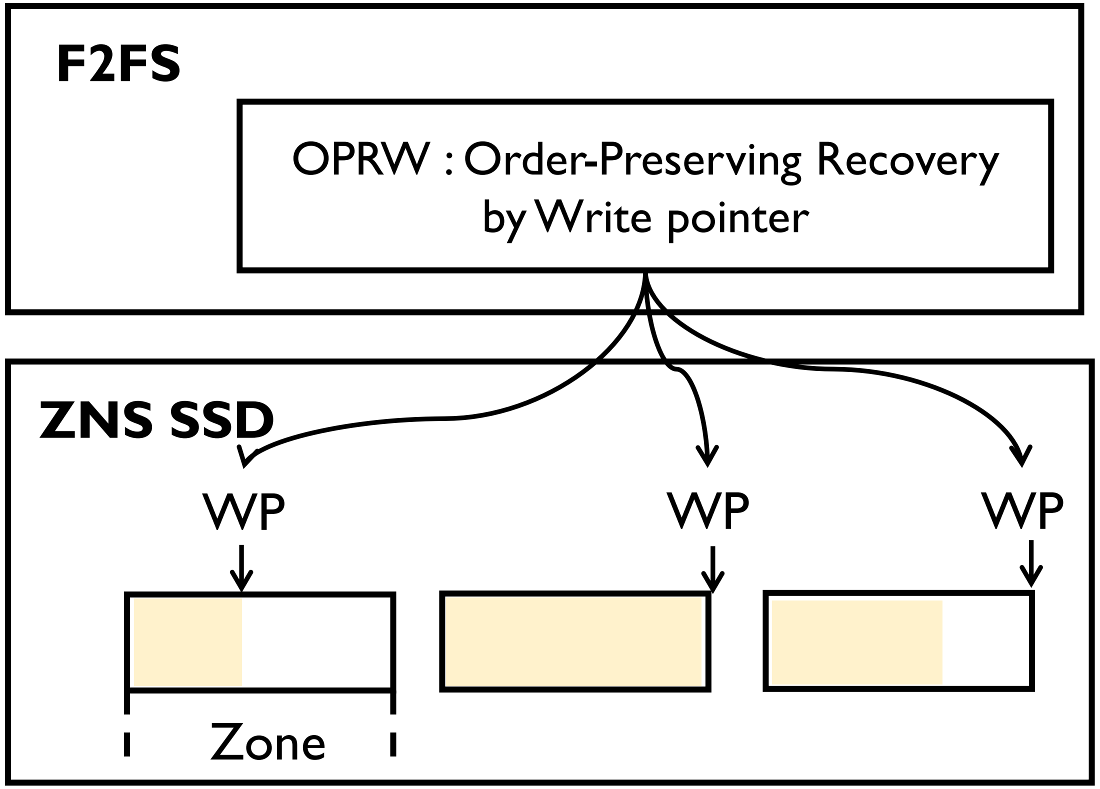
Euidong Lee, Ikjoon Son, Jin-Soo Kim
Seoul National University

Overview

Problem : Data loss in F2FS

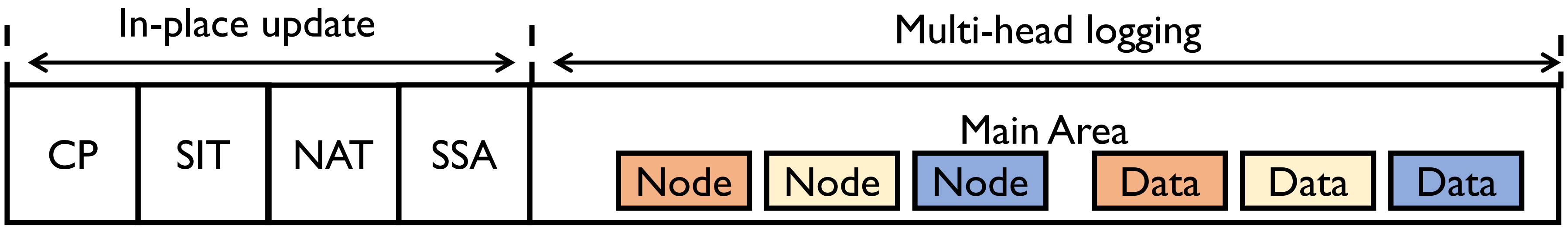


Solution : Recovery technique which uses the write pointer of ZNS



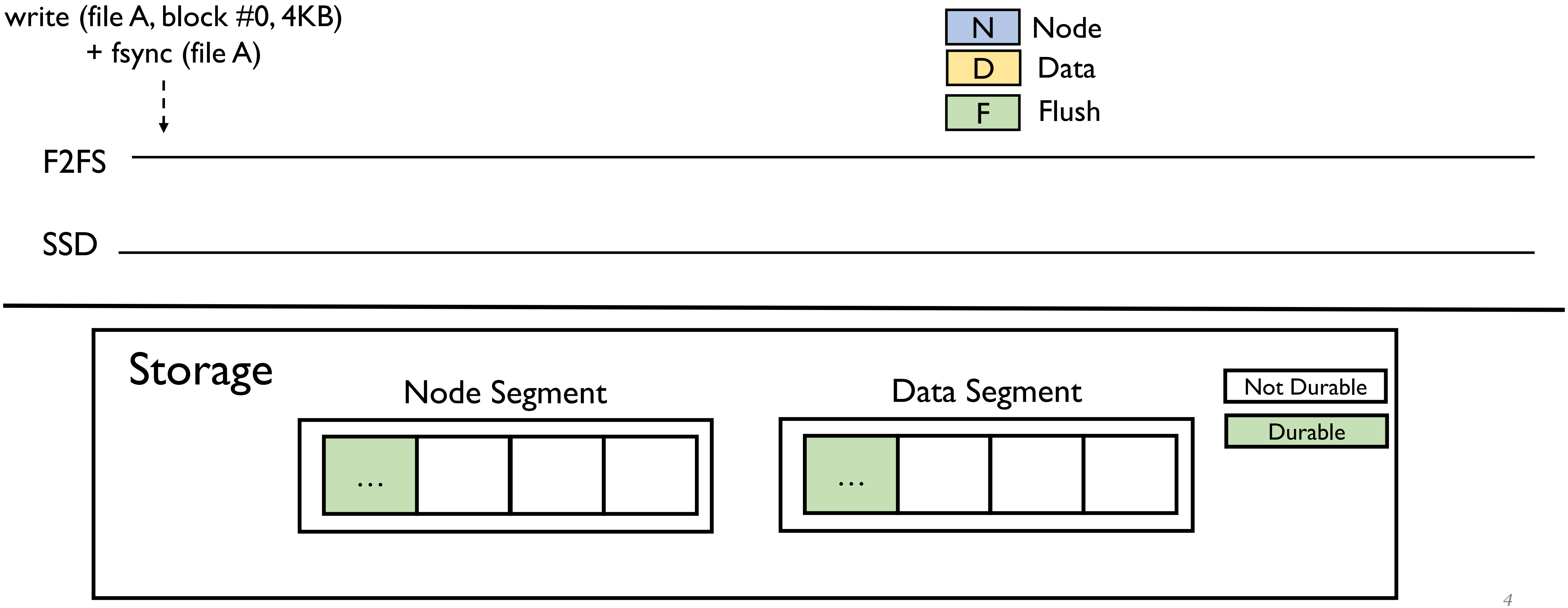
F2FS Filesystem

- Based on the log-structured filesystem
- Metadata : Node & CP, SIT, NAT, SSA, …
- fsync() : Log only the direct nodes of a file
- Recovery : Roll-back & Roll-forward



fsync() in F2FS

- Make all dirty data and nodes in a file persistent



write (file A, block #0, 4KB)
+ fsync (file A)

N Node

D Data

F Flush

F2FS

SSD

Storage

Node Segment

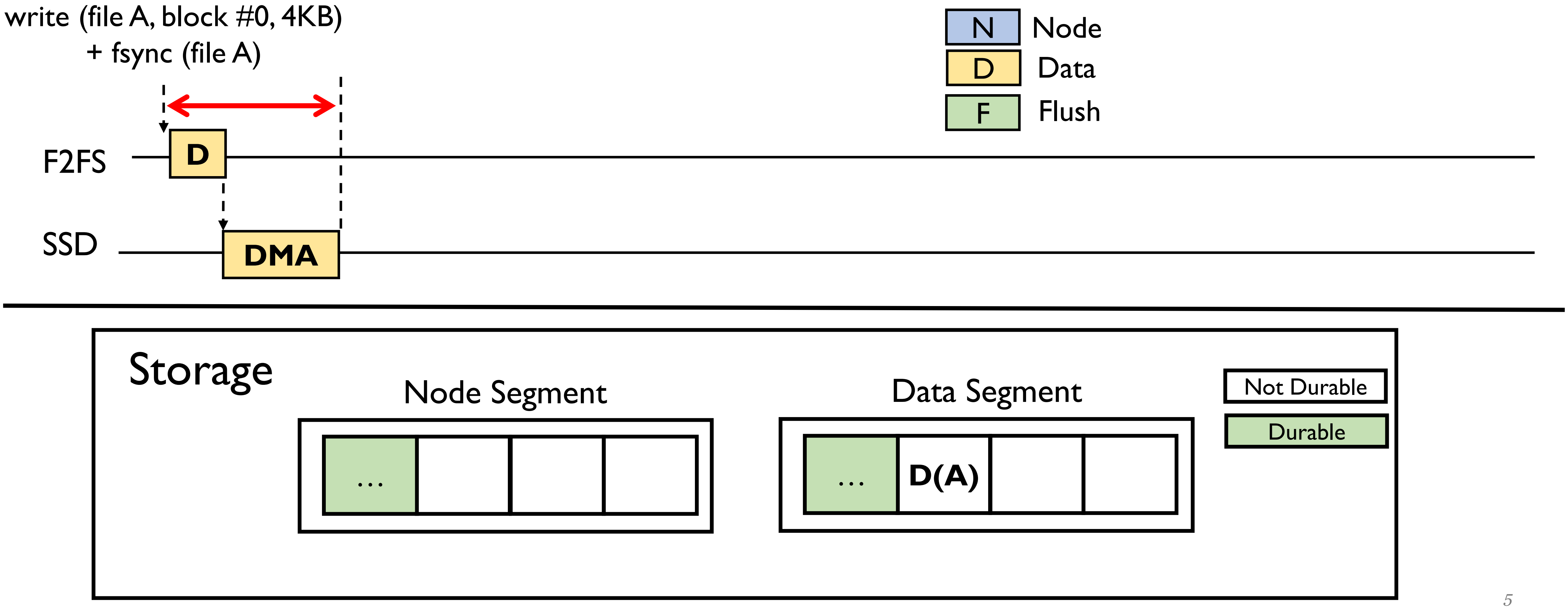
Data Segment

Not Durable

Durable

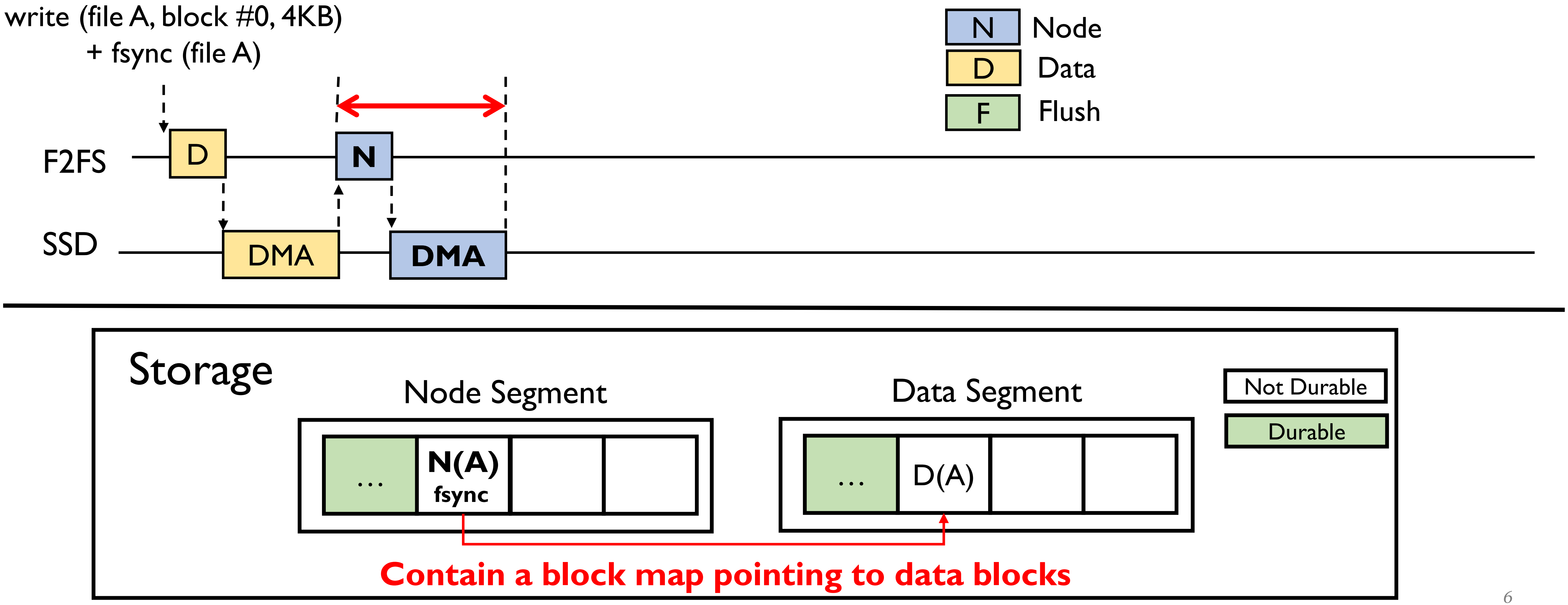
fsync() in F2FS

- Make all dirty data and nodes in a file persistent



fsync() in F2FS

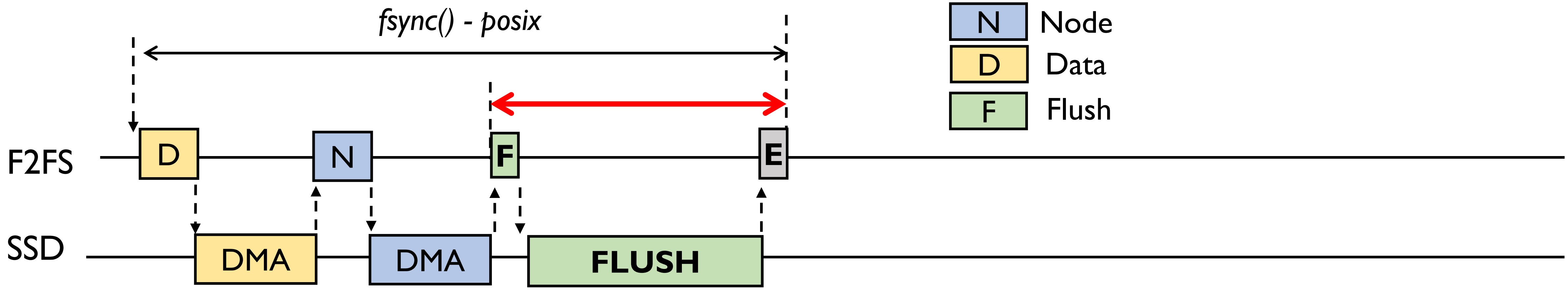
- Make all dirty data and nodes in a file persistent

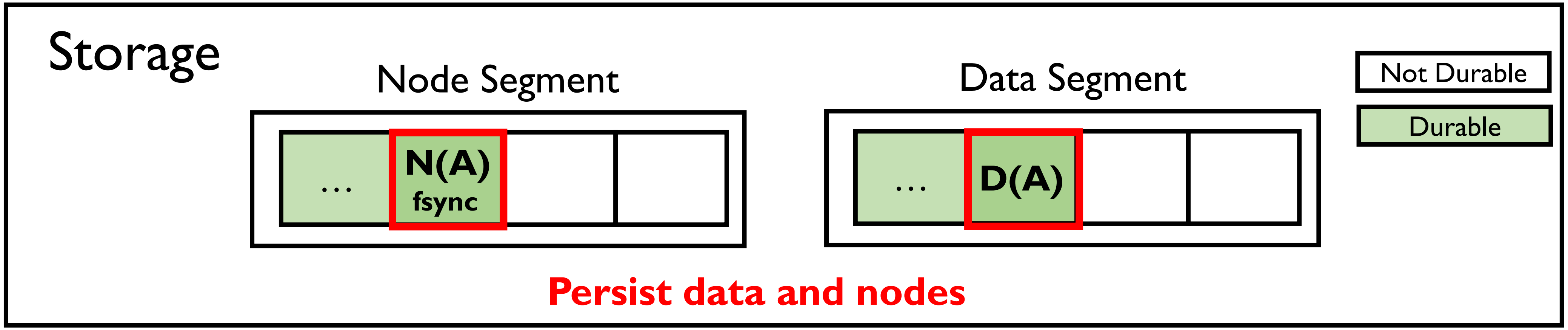


Contain a block map pointing to data blocks

fsync() in F2FS

- Make all dirty data and nodes in a file persistent

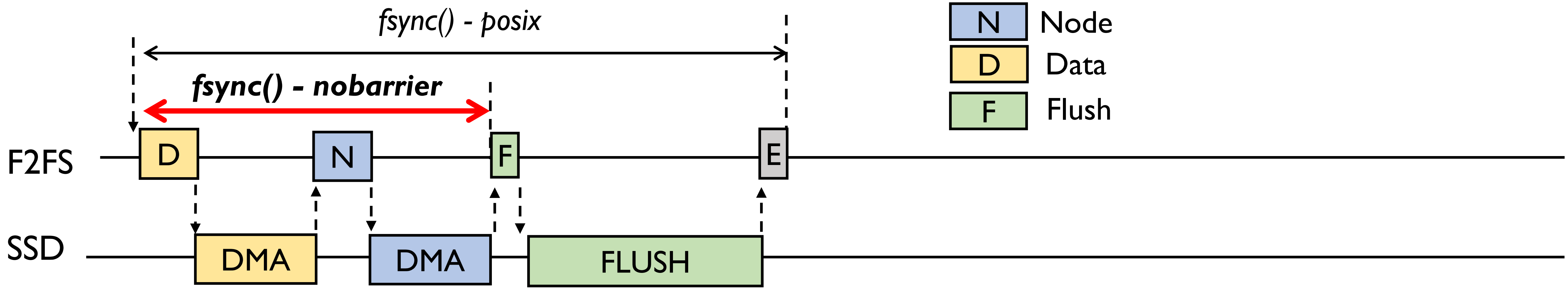


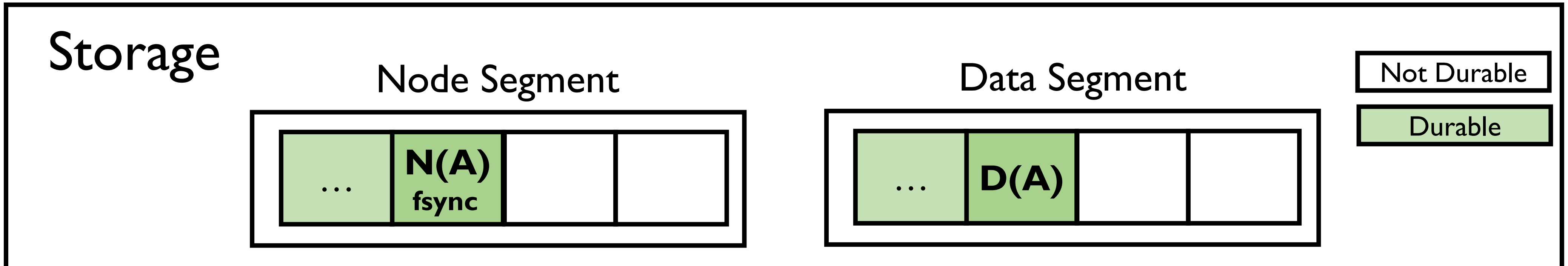


Persist data and nodes

fsync() in F2FS

- Make all dirty data and nodes in a file persistent

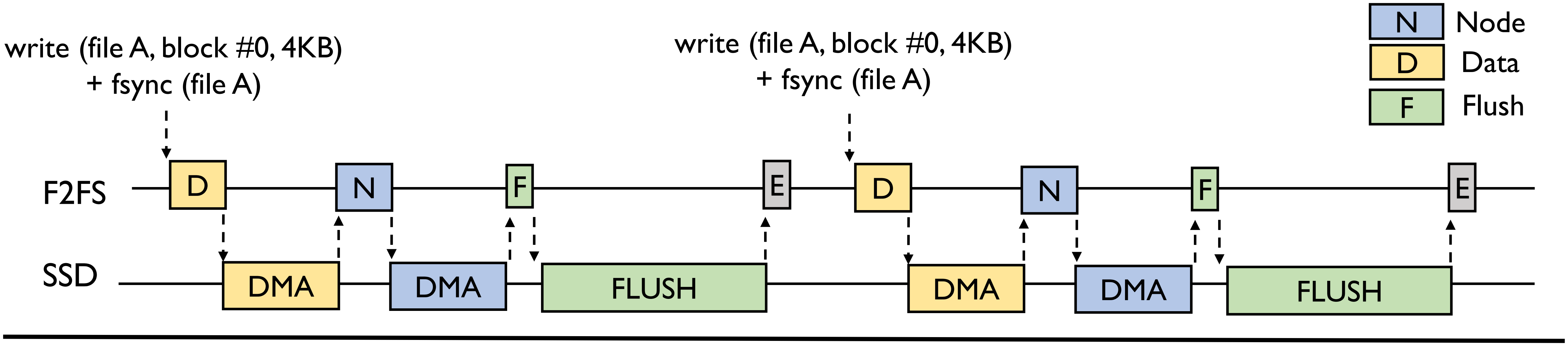


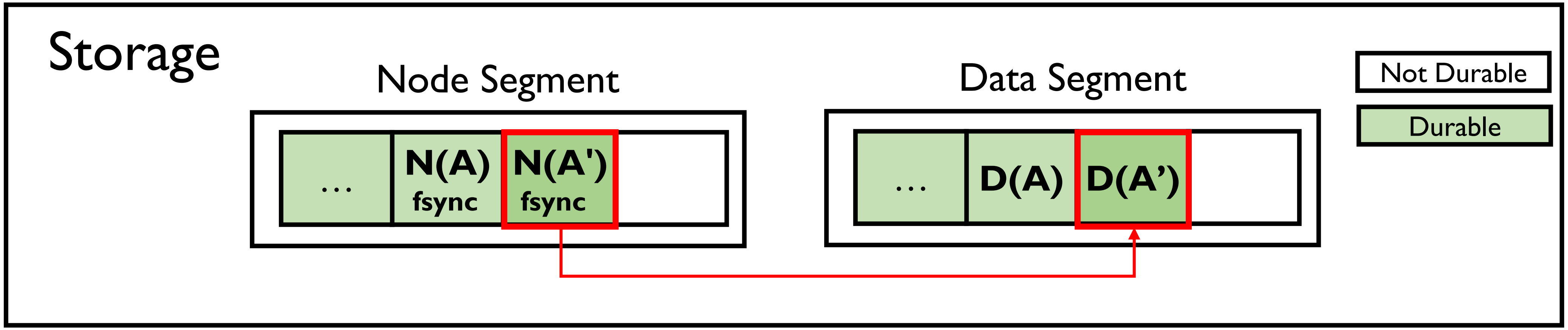


PLP devices guarantee data persistence without a flush command

fsync() in F2FS

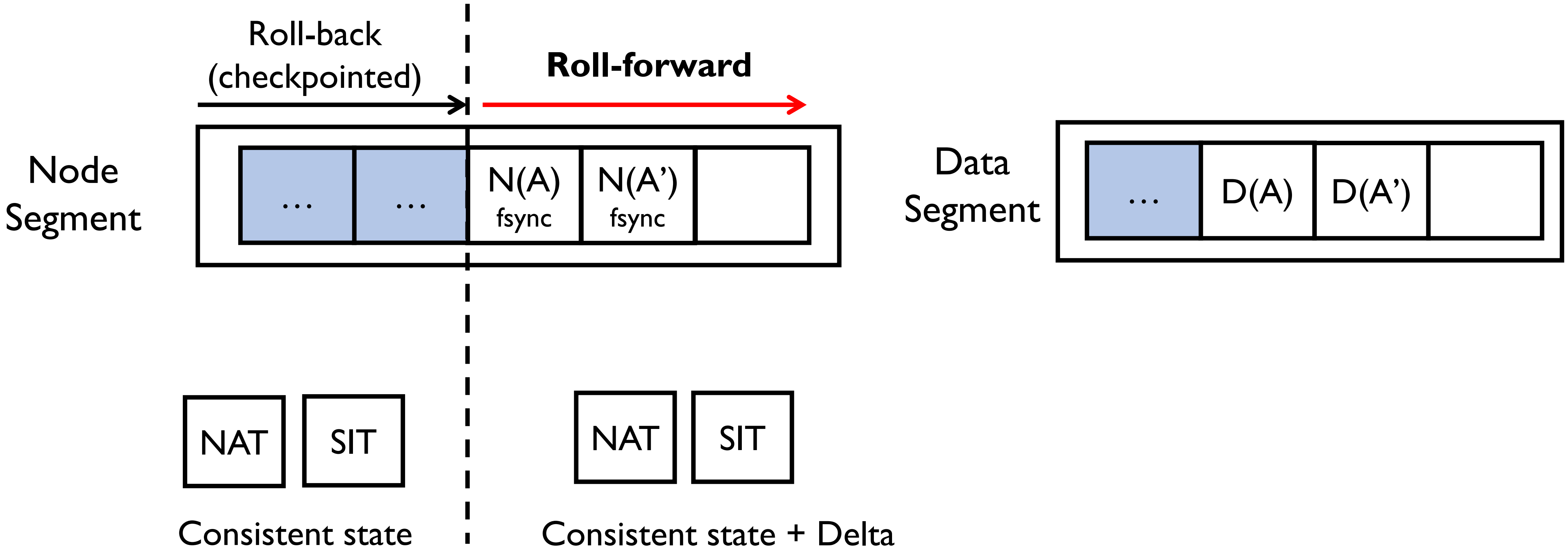
- Make all dirty data and nodes in a file persistent





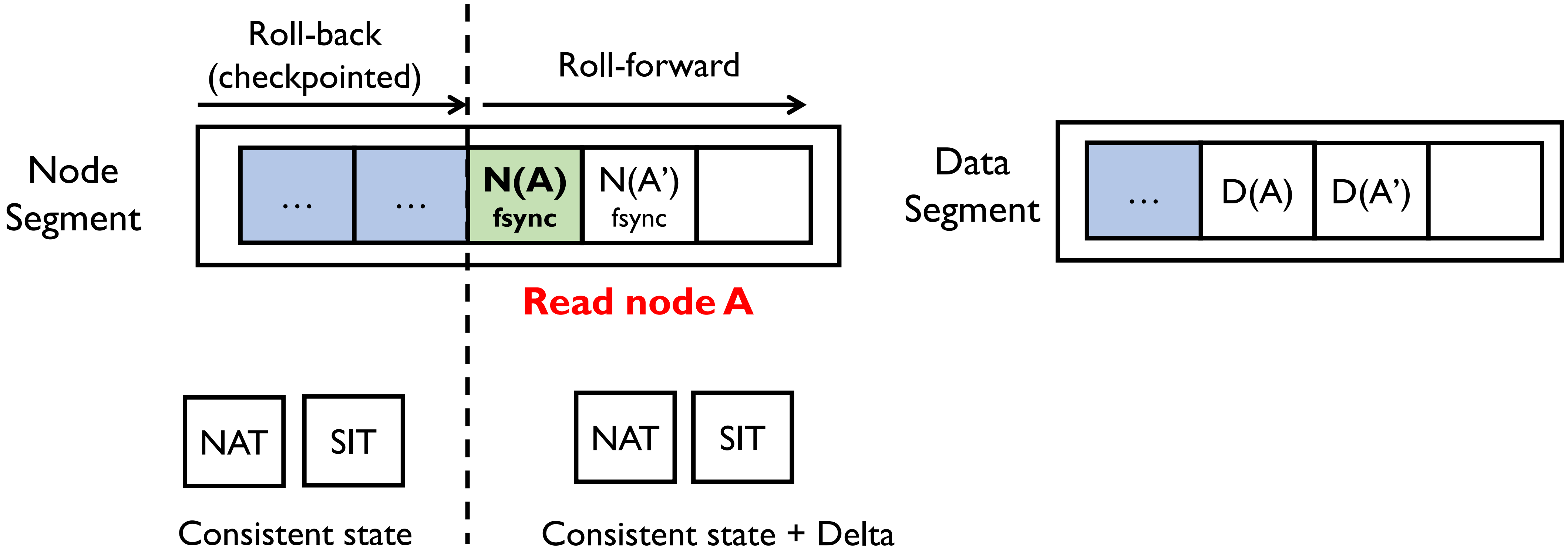
Roll-Forward in F2FS

- Recover node blocks written after the last checkpoint
  - Mark old blocks as invalid and new blocks as valid to SIT
  - Update node address in NAT



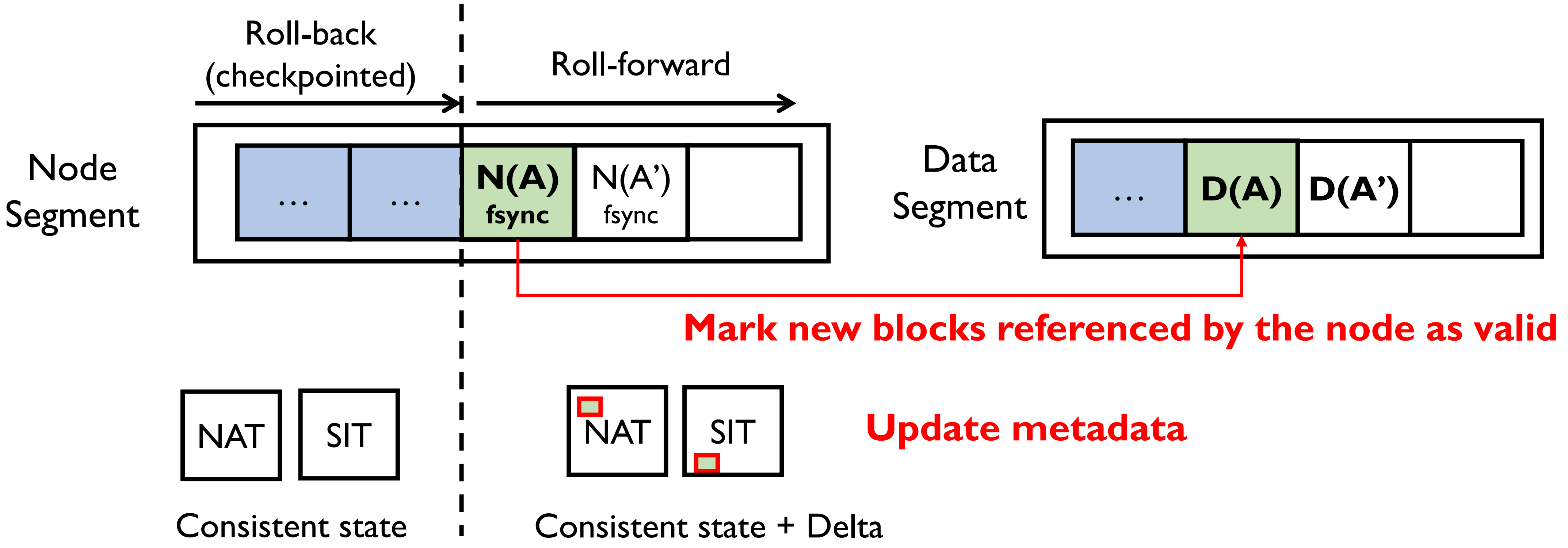
Roll-Forward in F2FS

- Recover node blocks written after the last checkpoint
  - Mark old blocks as invalid and new blocks as valid to SIT
  - Update node address in NAT



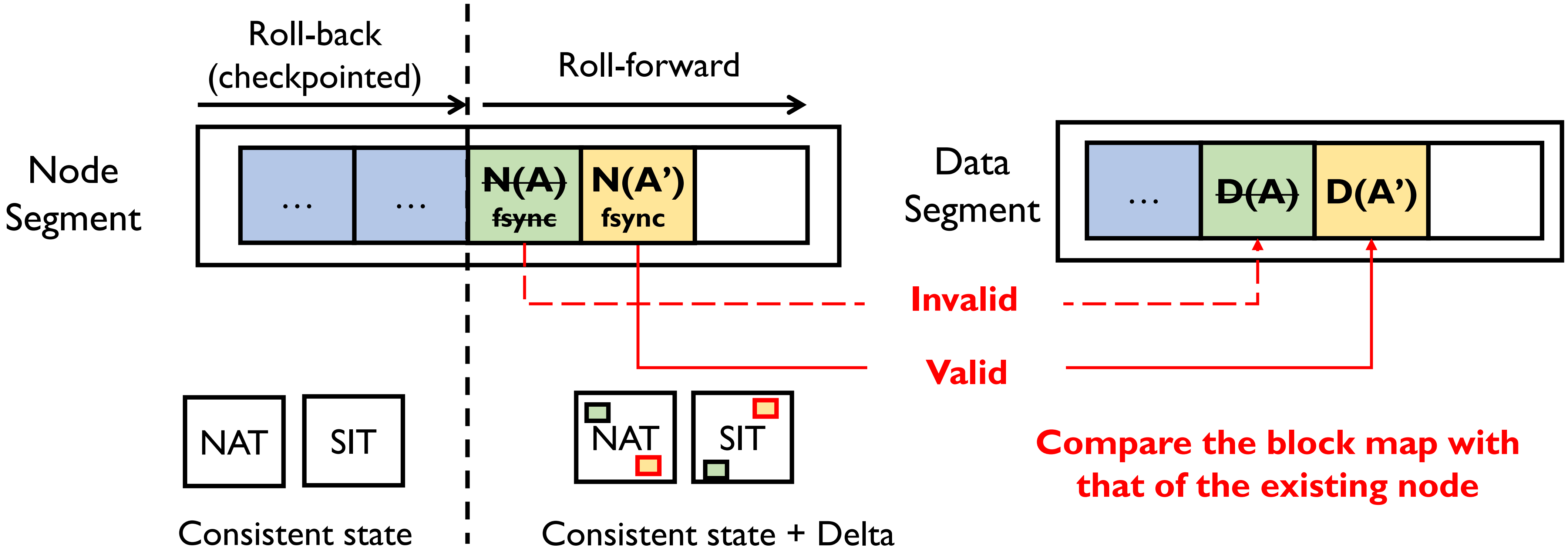
Roll-Forward in F2FS

- Recover node blocks written after the last checkpoint
  - Mark old blocks as invalid and new blocks as valid to SIT
  - Update node address in NAT



Roll-Forward in F2FS

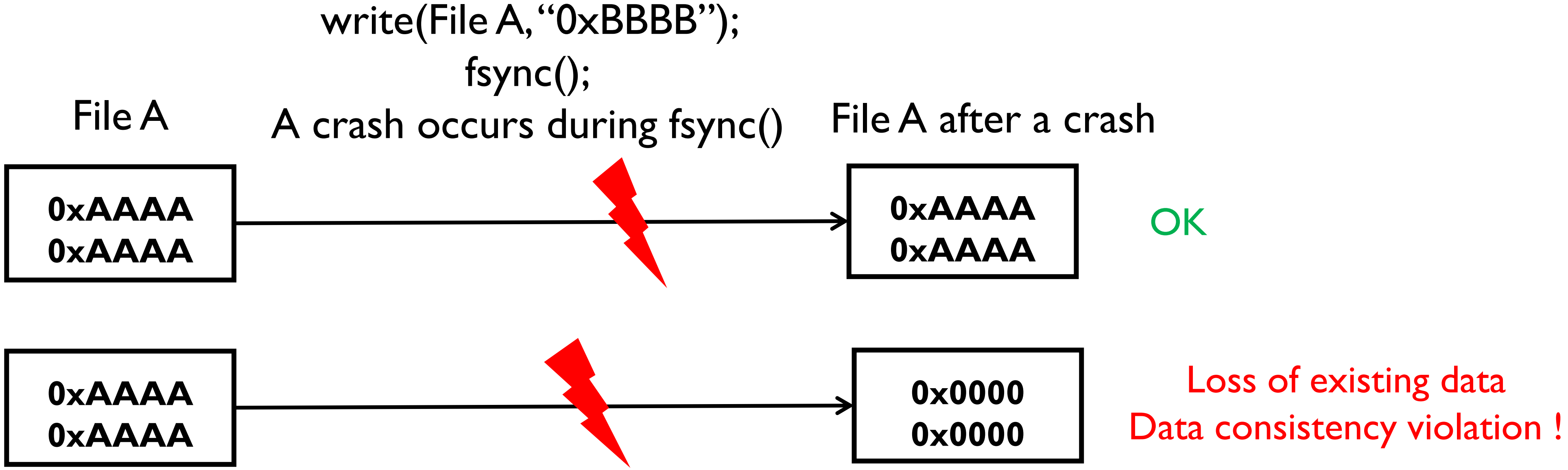
- Recover node blocks written after the last checkpoint
  - Mark old blocks as invalid and new blocks as valid to SIT
  - Update node address in NAT



Compare the block map with that of the existing node

Data Corruption in F2FS (1/2)

- F2FS has a risk of data corruption, when a crash occurs during fsync()



write(File A, “0xBBBB”);
fsync();
A crash occurs during fsync()

File A

0xAAAA
0xAAAA

File A after a crash

0xAAAA
0xAAAA

OK

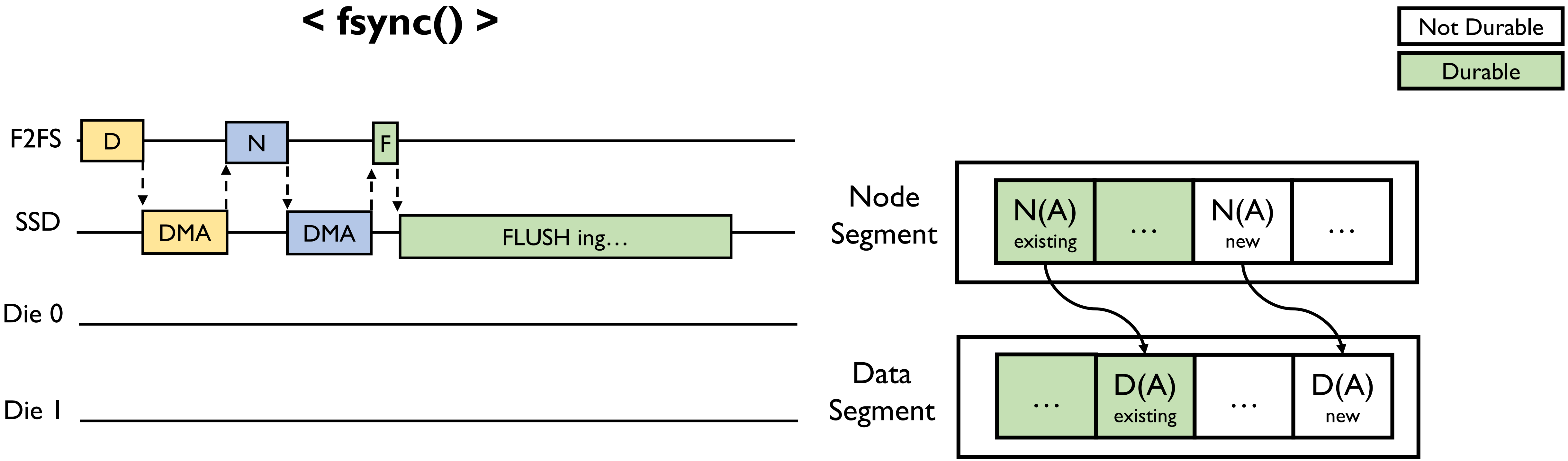
0xAAAA
0xAAAA

0x0000
0x0000

Loss of existing data
Data consistency violation !

Data Corruption in F2FS (2/2)

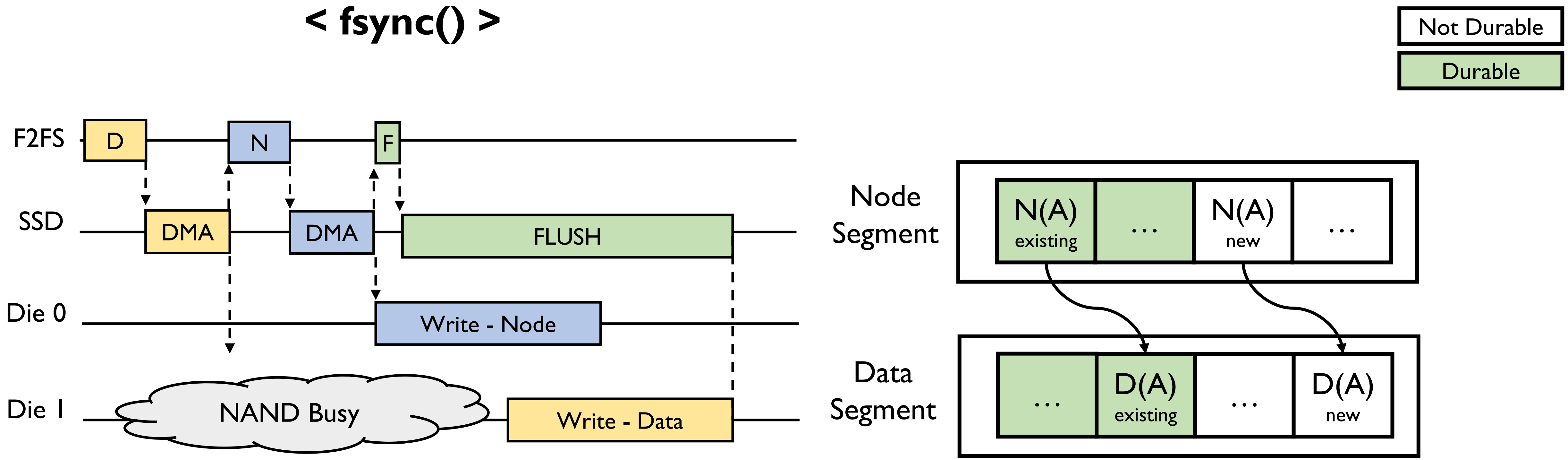
- Node blocks can be persisted before data blocks become durable
- Node may point to garbage data, resulting in loss of existing data



< fsync() >

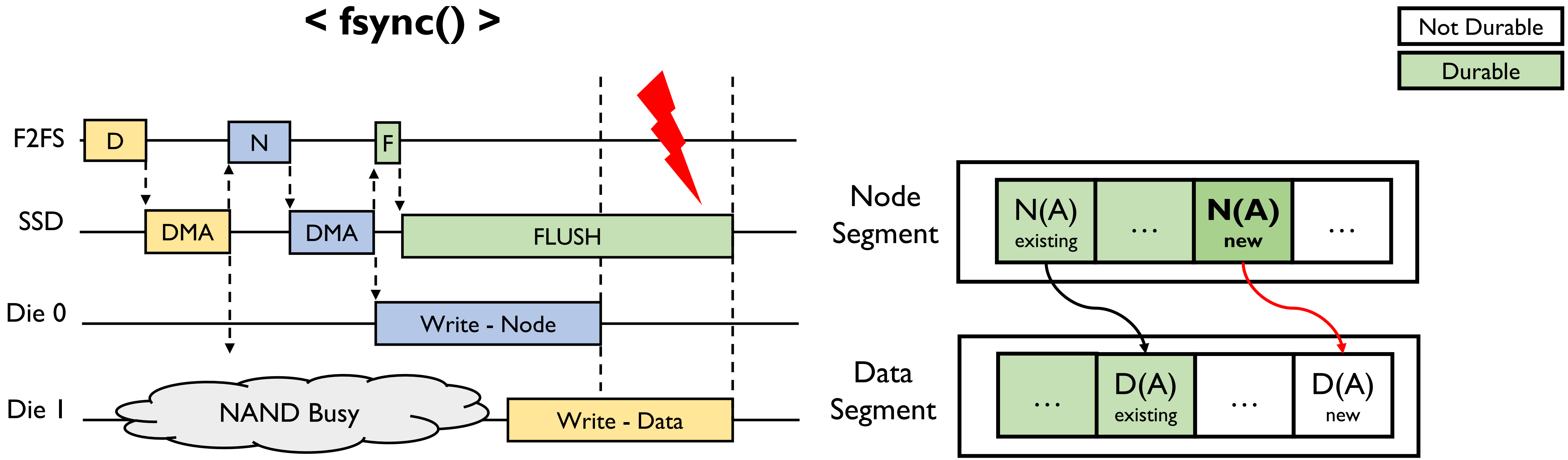
Data Corruption in F2FS (2/2)

- Node blocks can be persisted before data blocks become durable
- Node may point to garbage data, resulting in loss of existing data



Data Corruption in F2FS (2/2)

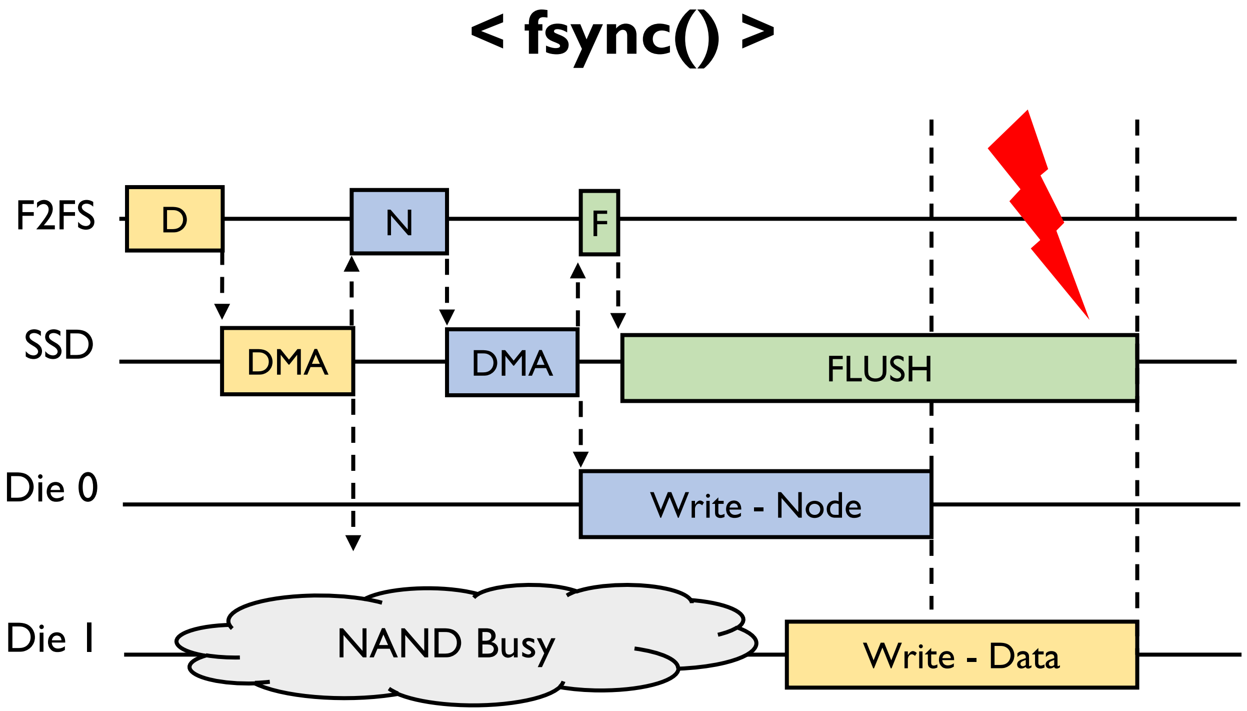
- Node blocks can be persisted before data blocks become durable
- Node may point to garbage data, resulting in loss of existing data

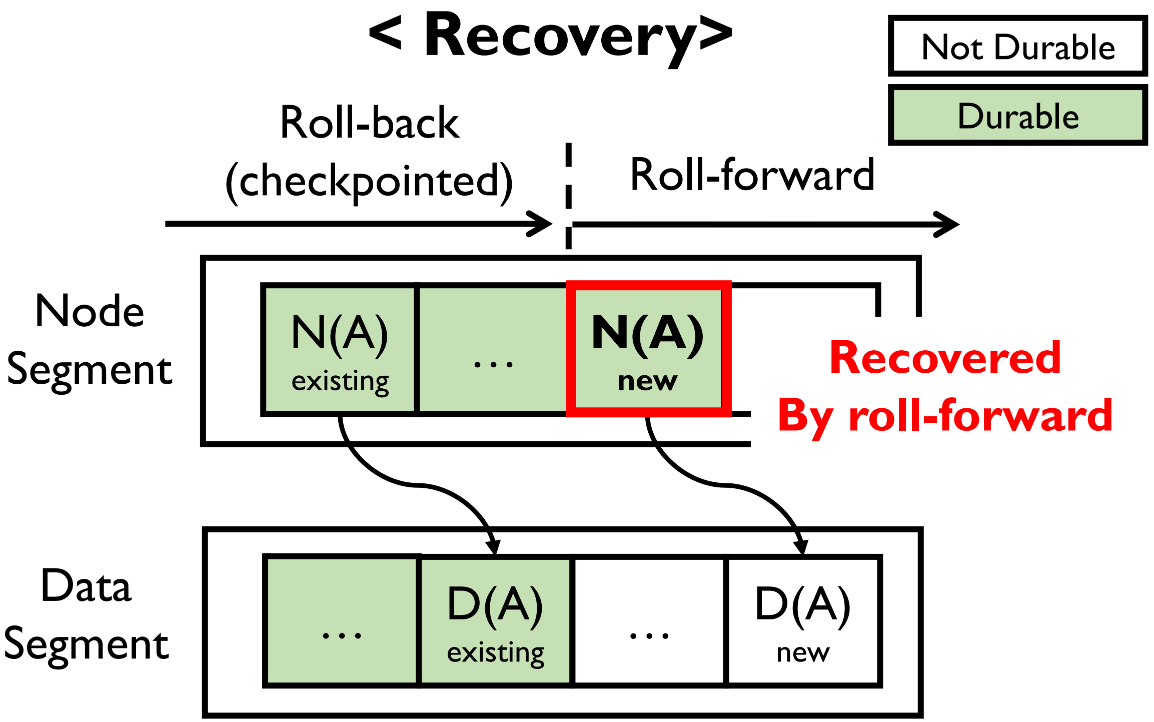


point to garbage data!

Data Corruption in F2FS (2/2)

- Node blocks can be persisted before data blocks become durable
- Node may point to garbage data, resulting in loss of existing data

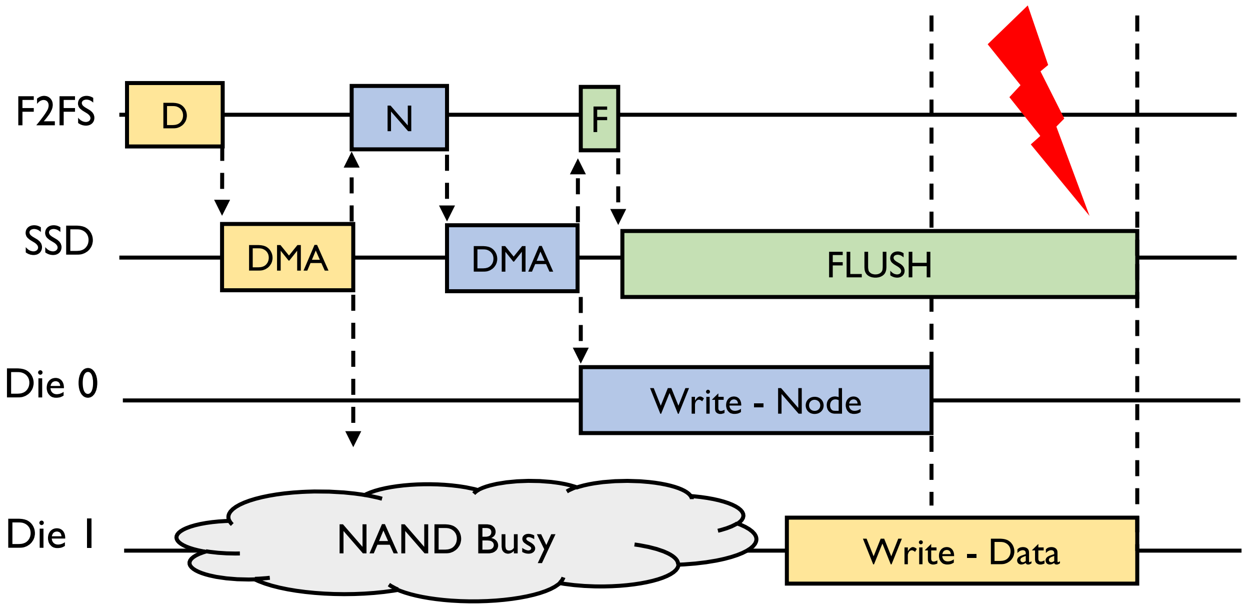




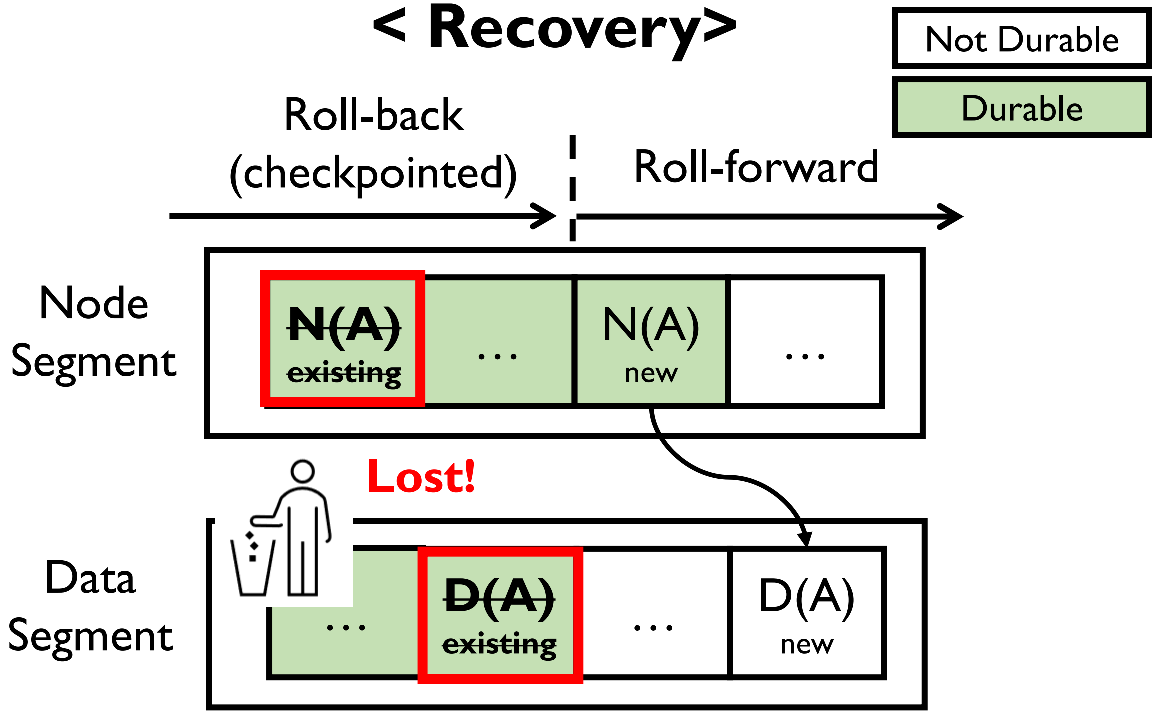
Data Corruption in F2FS (2/2)

- Node blocks can be persisted before data blocks become durable
- Node may point to garbage data, resulting in loss of existing data

< fsync() >

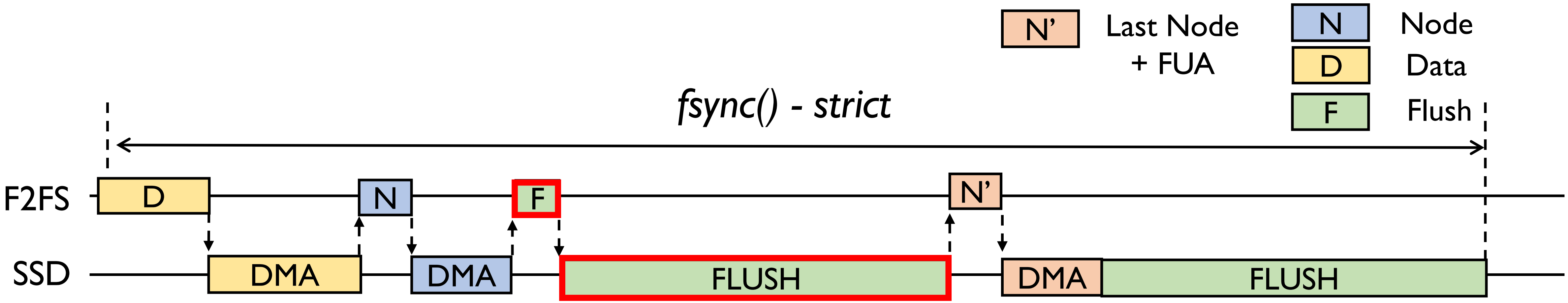


< Recovery>



Naïve Solution - Pessimistic Approach

- Enforce write order between data and nodes during fsync()
  - Pros : fast recovery
  - Cons : **performance degradation** due to a flush operation
- e.g. *strict* mode in F2FS
  - Use atomic writes, inserting a flush command before the last node block



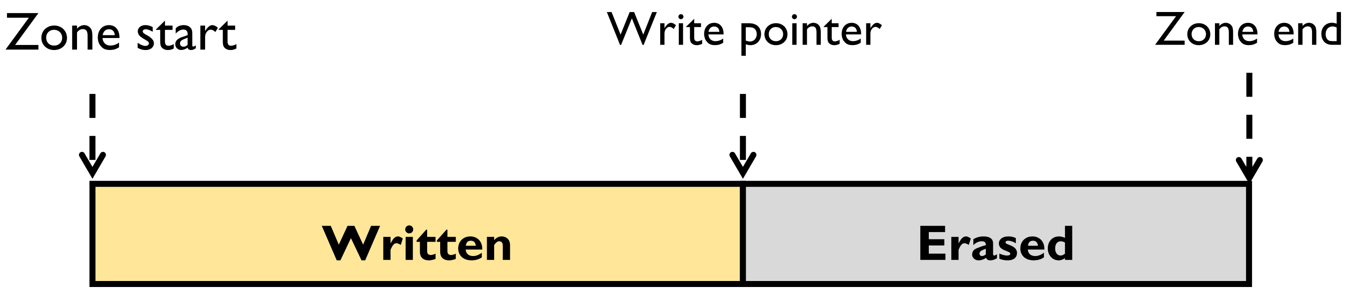
Naïve Solution - Optimistic Approach

- Detect write order reversal during filesystem recovery
  - Pros : high performance in fsync()
  - Cons : **long recovery** time
- Difficult to determine persistence of data blocks in the block interface
  - Requires additional metadata and I/O operations

Is there a way to efficiently identify the persistence of data blocks ?

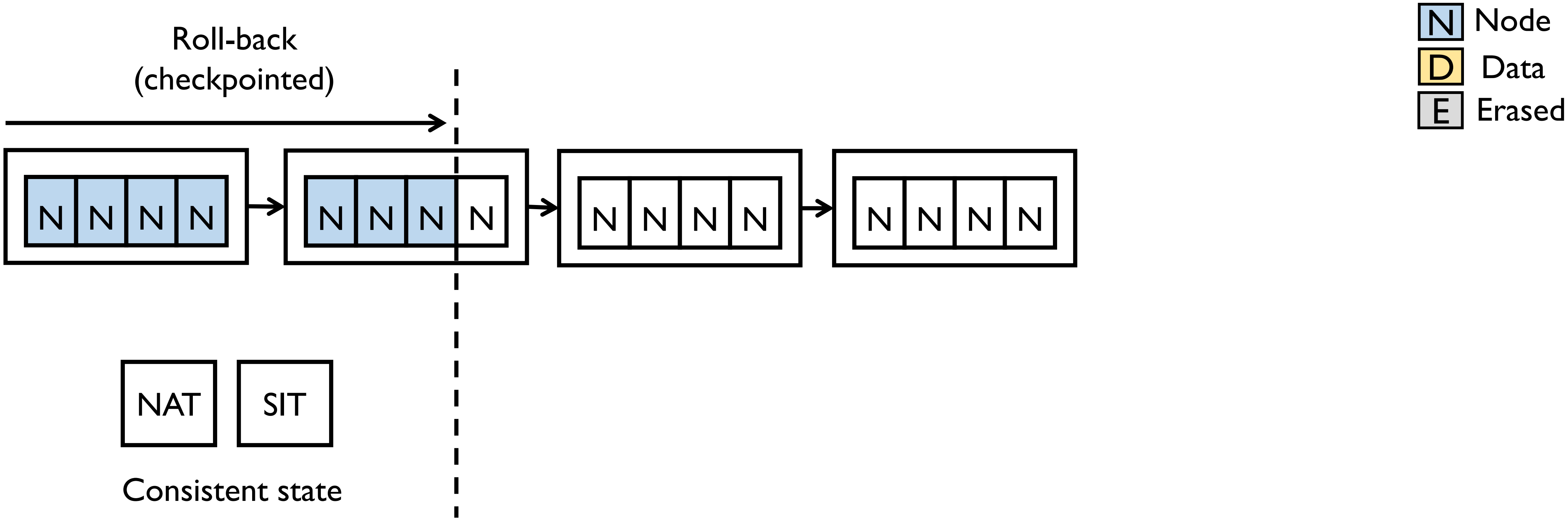
Observation - Write Pointer of ZNS

- Can be utilized to efficiently determine the validity of blocks
  - (LBAs < write pointer) → written, valid
  - (LBAs >= write pointer) → erased, invalid
- No I/O operations required



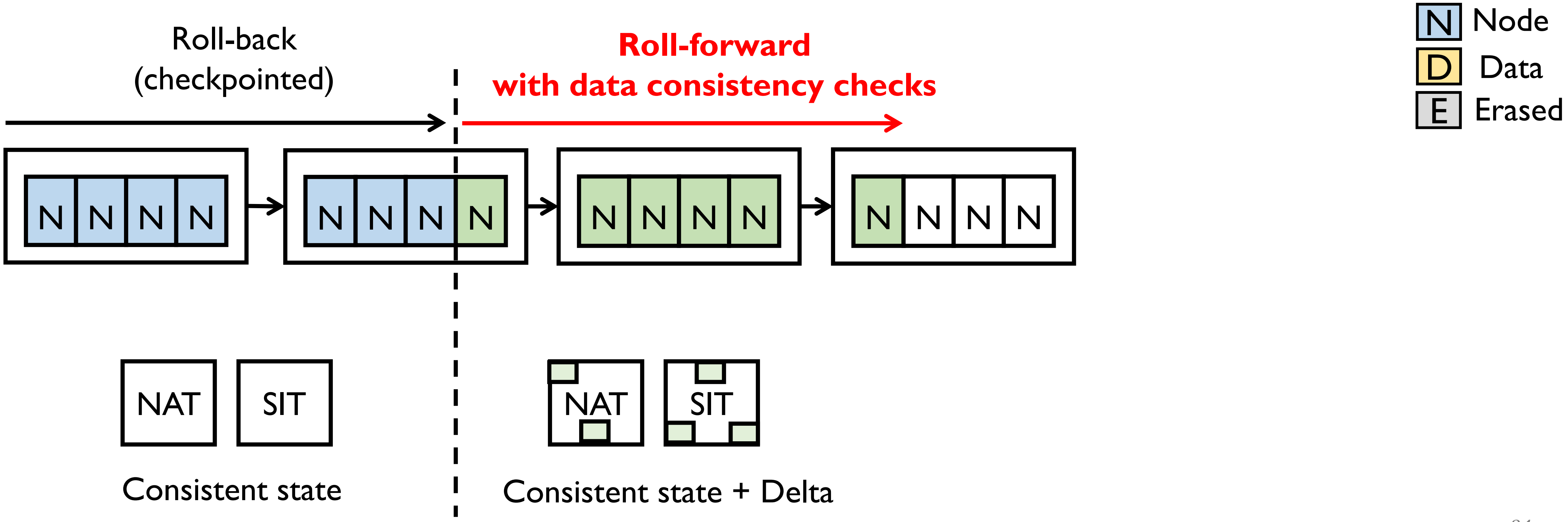
Order-Preserving Recovery by Write Pointer

- Ensure data consistency during recovery with low overhead
- Check all nodes for data consistency during the roll-forward process
  - Exclude nodes with data loss from the recovery target



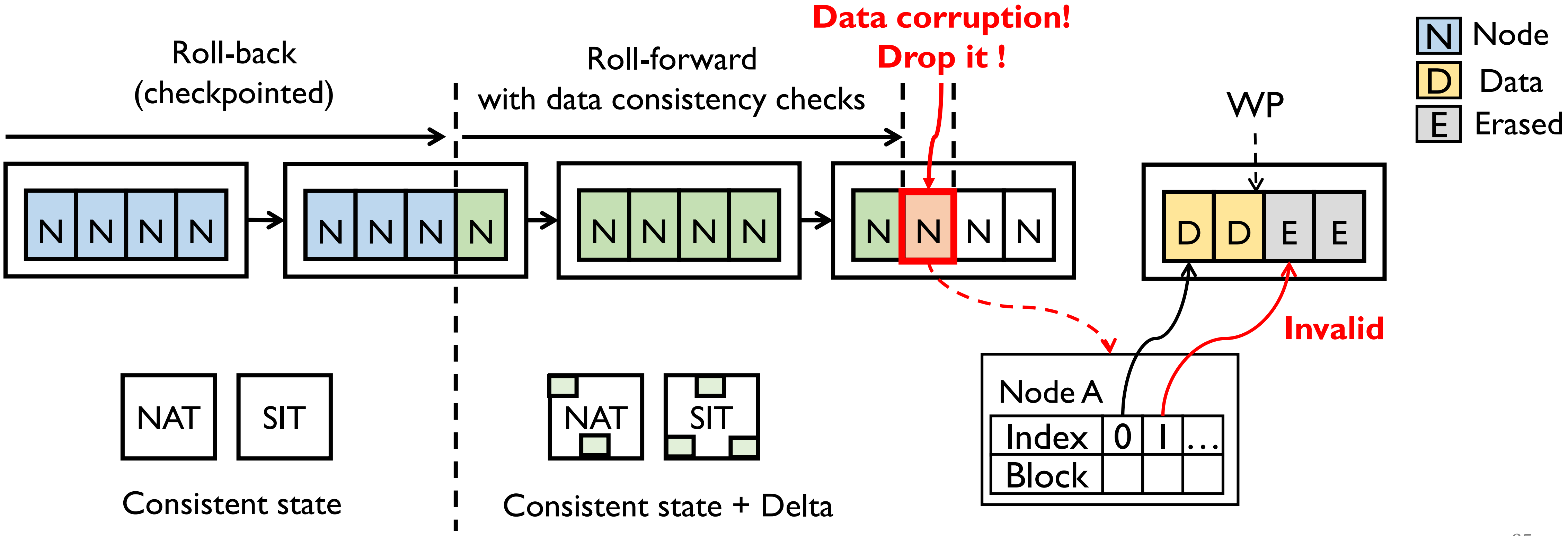
Order-Preserving Recovery by Write Pointer

- Ensure data consistency during recovery with low overhead
- Check all nodes for data consistency during the roll-forward process
  - Exclude nodes with data loss from the recovery target



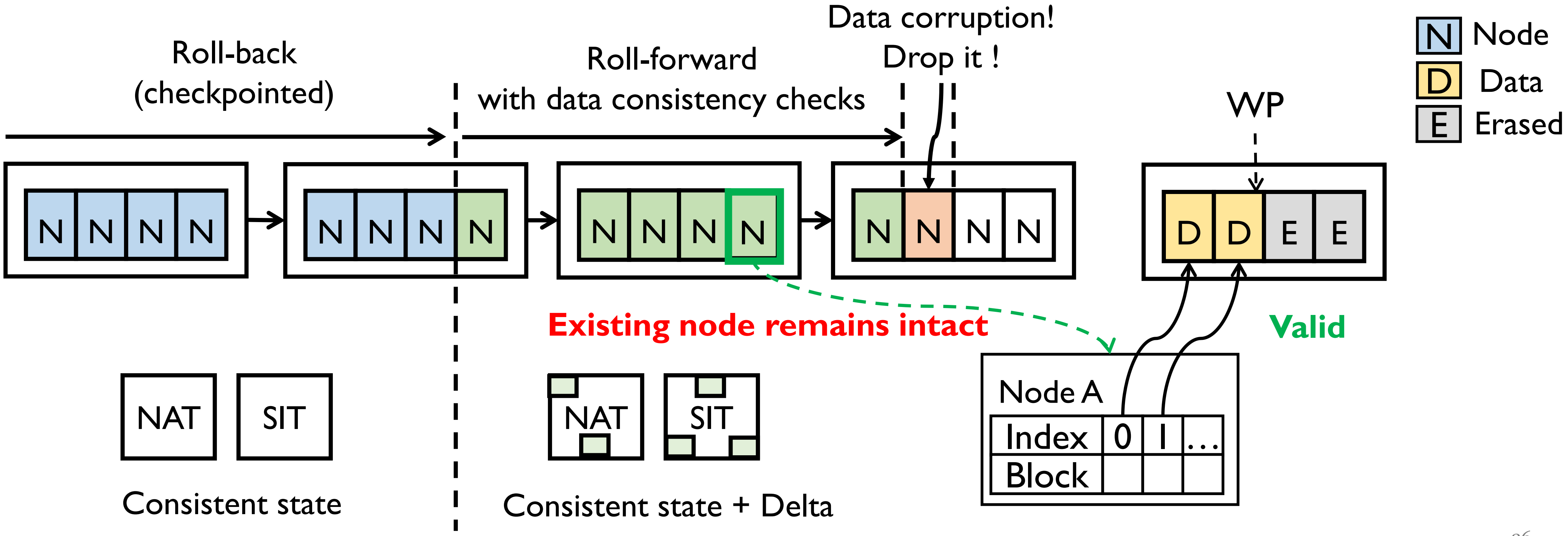
Order-Preserving Recovery by Write Pointer

- Ensure data consistency during recovery with low overhead
- Check all nodes for data consistency during the roll-forward process
  - Exclude nodes with data loss from the recovery target



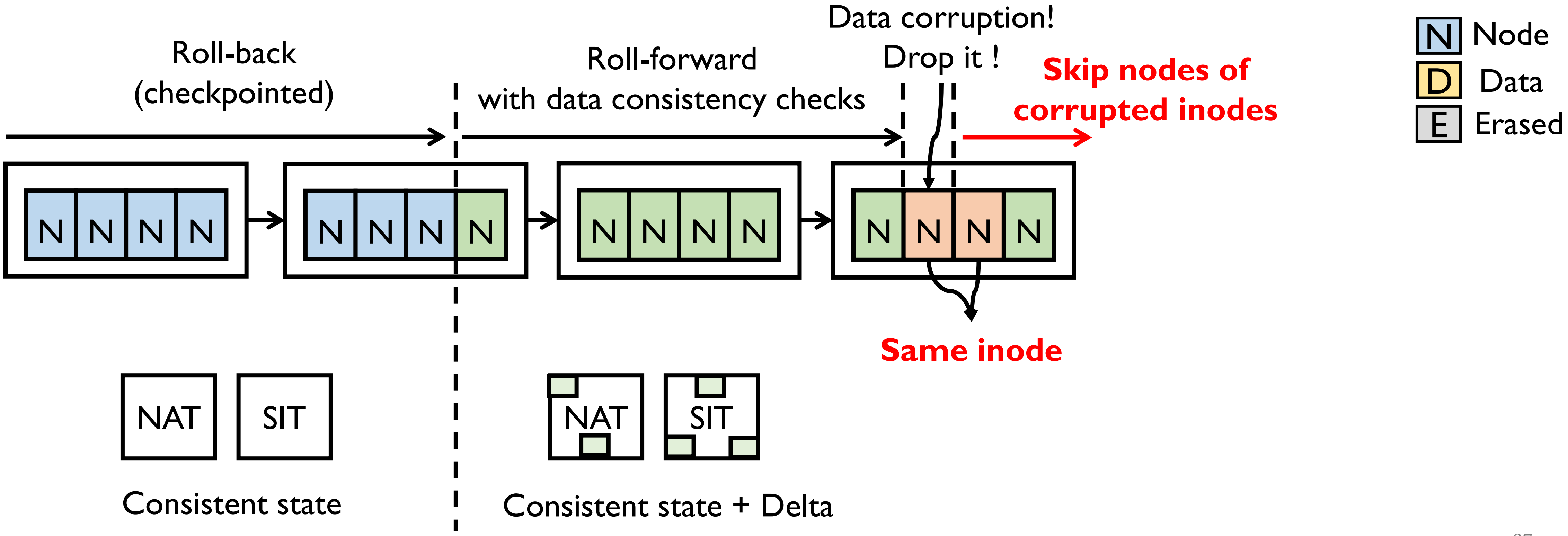
Order-Preserving Recovery by Write Pointer

- Ensure data consistency during recovery with low overhead
- Check all nodes for data consistency during the roll-forward process
  - Exclude nodes with data loss from the recovery target



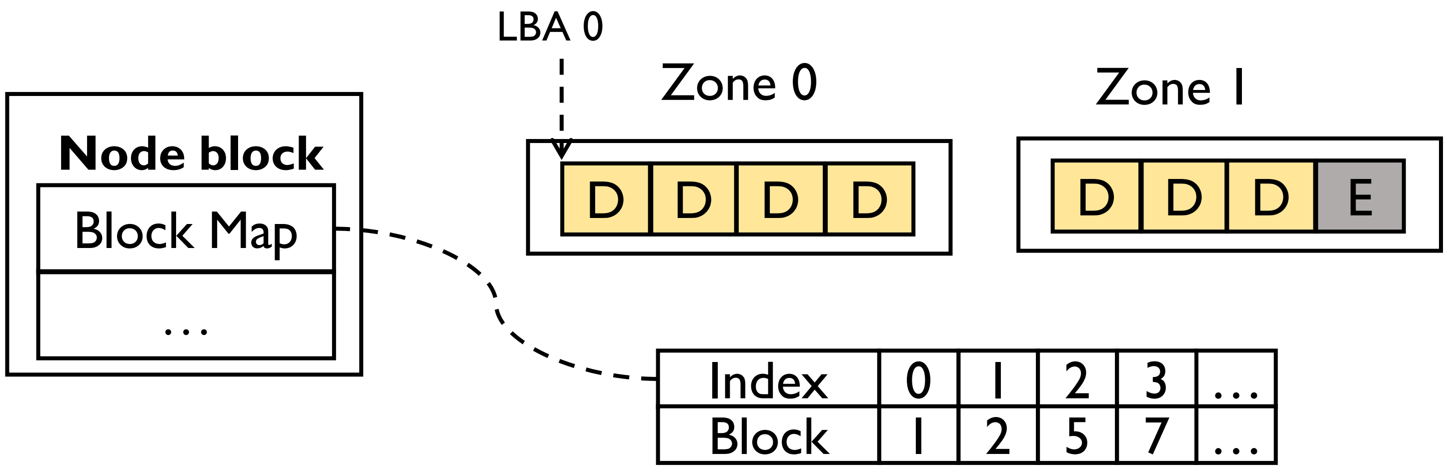
Order-Preserving Recovery by Write Pointer

- Ensure data consistency during recovery with low overhead
- Check all nodes for data consistency during the roll-forward process
  - Exclude nodes with data loss from the recovery target



Data Consistency Check for a Node

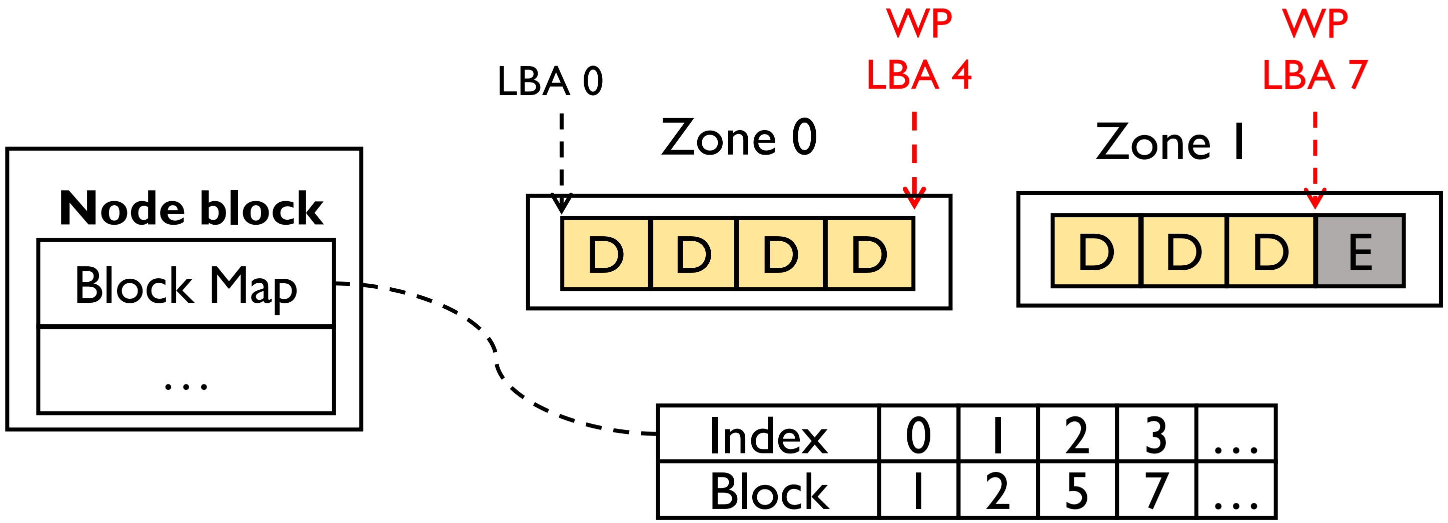
- Load the entire write pointer table into memory once at boot time
- Check each entry to see if it is less than the write pointer
  - (address >= write pointer) → corrupted node



Index	0	1	2	3	…
Block	1	2	5	7	…

Data Consistency Check for a Node

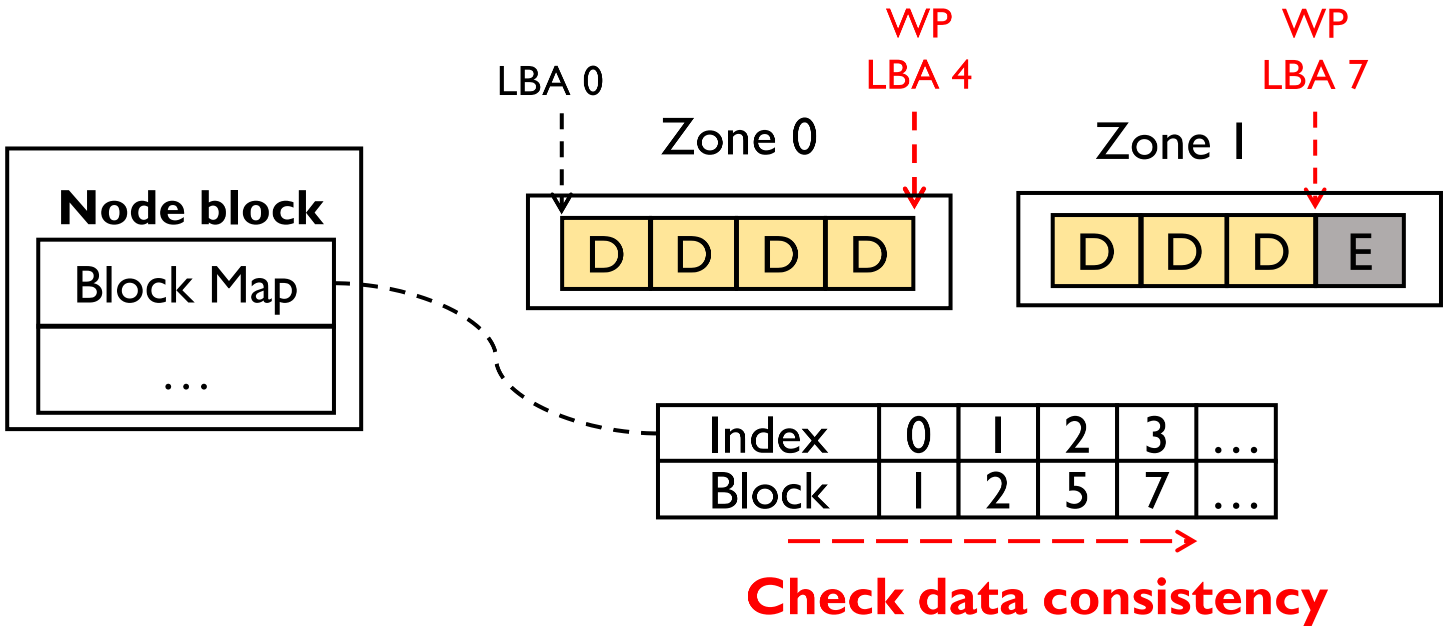
- Load the entire write pointer table into memory once at boot time
- Check each entry to see if it is less than the write pointer
  - (address >= write pointer) → corrupted node



Index	0	1	2	3	…
Block	1	2	5	7	…

Data Consistency Check for a Node

- Load the entire write pointer table into memory once at boot time
- Check each entry to see if it is less than the write pointer
  - (address >= write pointer) → corrupted node

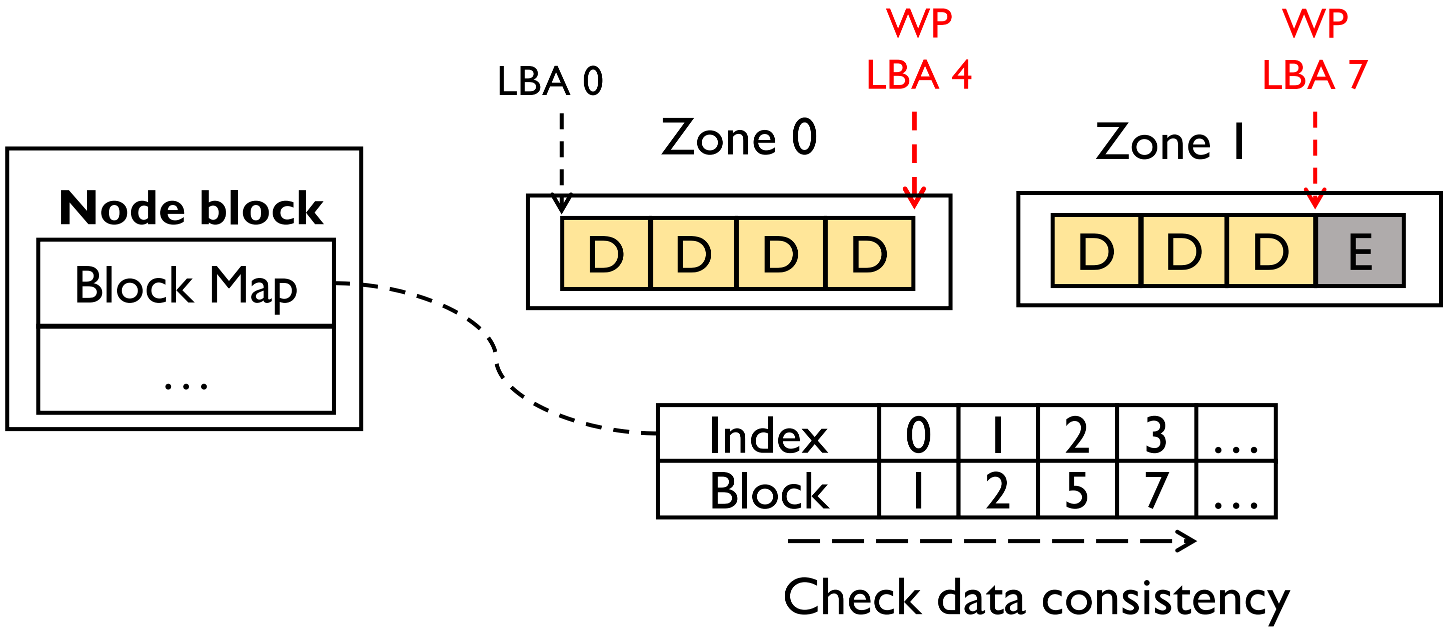


Index	0	1	2	3	…
Block	1	2	5	7	…

Check data consistency

Data Consistency Check for a Node

- Load the entire write pointer table into memory once at boot time
- Check each entry to see if it is less than the write pointer
  - (address >= write pointer) → corrupted node

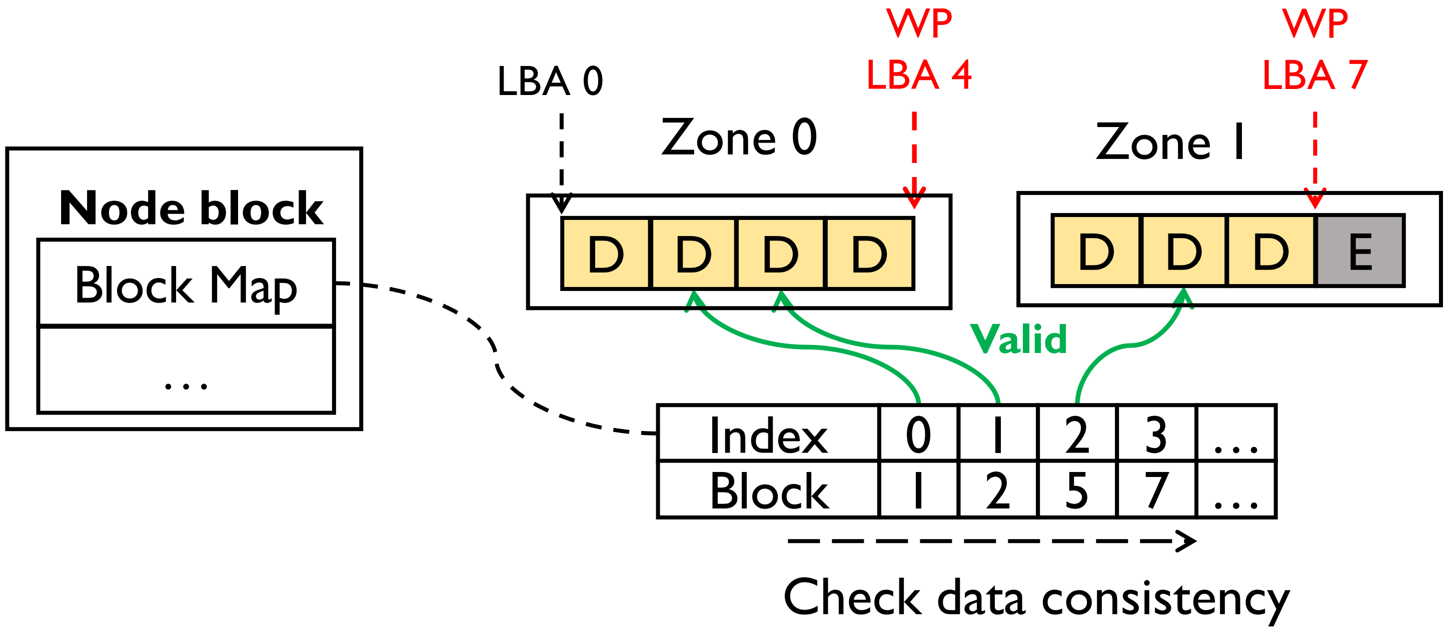


Index	0	1	2	3	…
Block	1	2	5	7	…

```
for (i = 0; i < # index; i++) {
  If (block_map[i] < write pointer)
      continue;   /* written, valid */
  else
      break;      /* erased, invald */
}
```

Data Consistency Check for a Node

- Load the entire write pointer table into memory once at boot time
- Check each entry to see if it is less than the write pointer
  - (address >= write pointer) → corrupted node

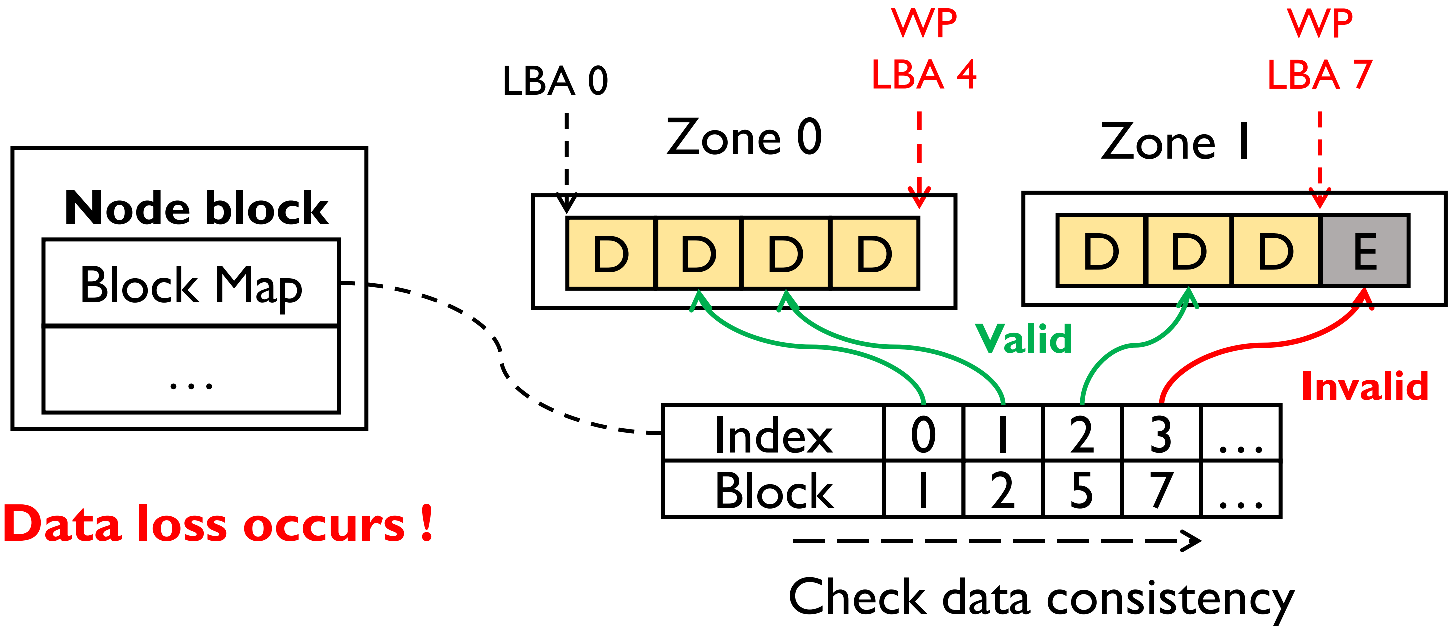


Index	0	1	2	3	...
Block	1	2	5	7	...

```
for (i = 0; i < # index; i++) {
   If (block_map[i] < write pointer)
      continue;   /* written, valid */
   else
      break;      /* erased, invald */
}
```

Data Consistency Check for a Node

- Load the entire write pointer table into memory once at boot time
- Check each entry to see if it is less than the write pointer
  - (address >= write pointer) → corrupted node



Index	0	1	2	3	…
Block	1	2	5	7	…

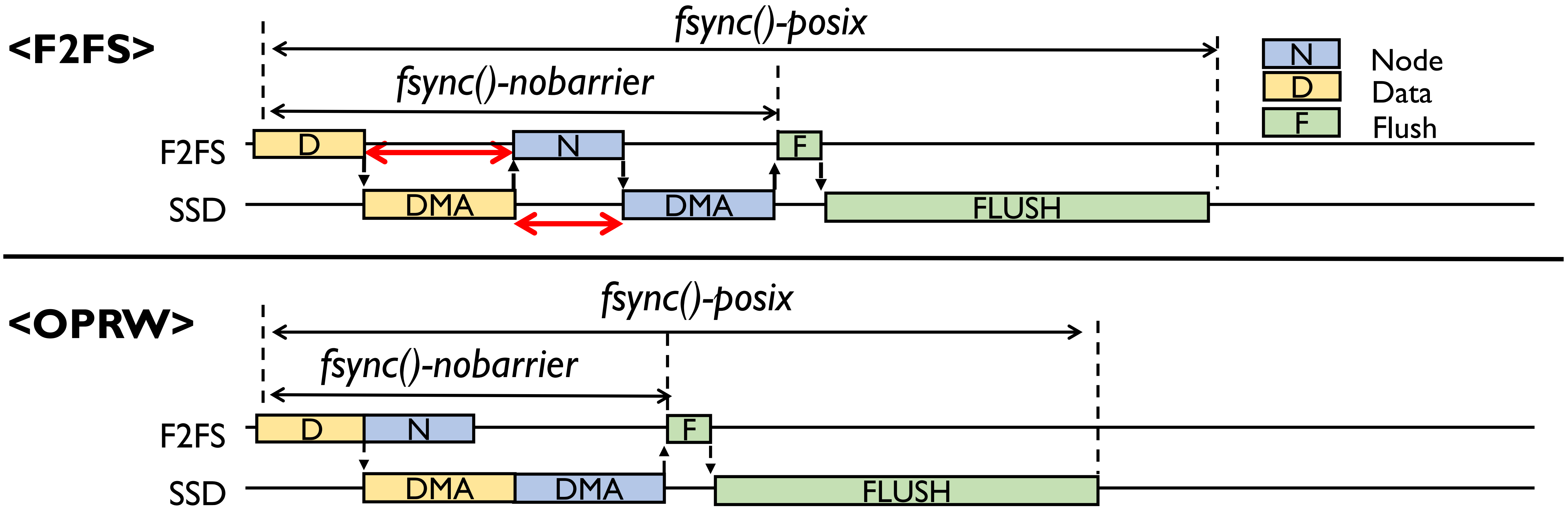
Check data consistency

Data loss occurs !

```
for (i = 0; i < # index; i++) {
   If (block_map[i] < write pointer)
      continue;   /* written, valid */
   else
      break;      /* erased, invald */
}
```

Performance Improvement

- No need to enforce the write order during fsync()
  - OPRW ensures data consistency during the recovery process



Evaluation Setup

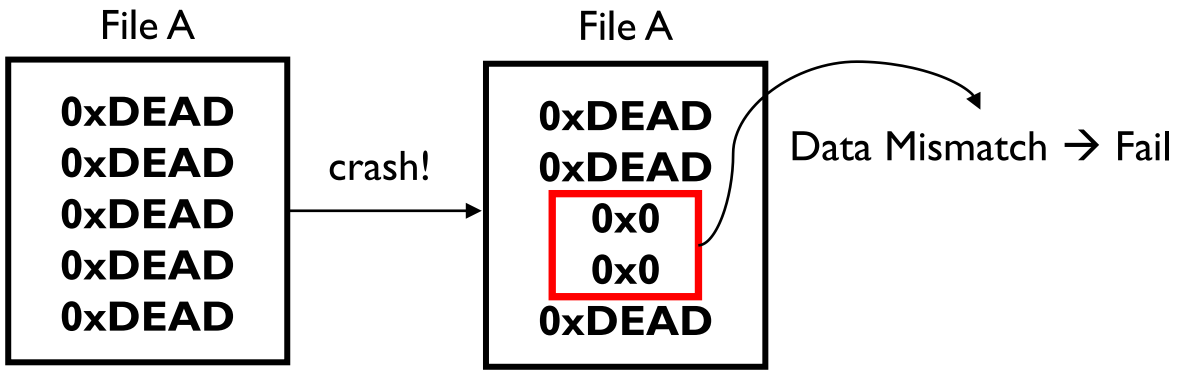
- Platform
  - Intel Xeon Silver 4116 2.10GHz
  - 5.14.4 Kernel

- Storage
  - Western Digital ZN540 : Supporting Power-Loss Protection (PLP)
  - NVMeVirt (FAST 23') : Emulating ZN540 without PLP

- Workload
  - FIO, Varmail, OLTP-Insert

Data Consistency Test

- Sequence
  1. Fill the file with the specific pattern
  2. Write the same pattern in the file
  3. Inject a crash at a random time
  4. Check to see if the pattern matches, after the file system is restored.



File A

0xDEAD
0xDEAD
0xDEAD
0xDEAD
0xDEAD

crash!

File A

0xDEAD
0xDEAD
0x0
0x0
0xDEAD

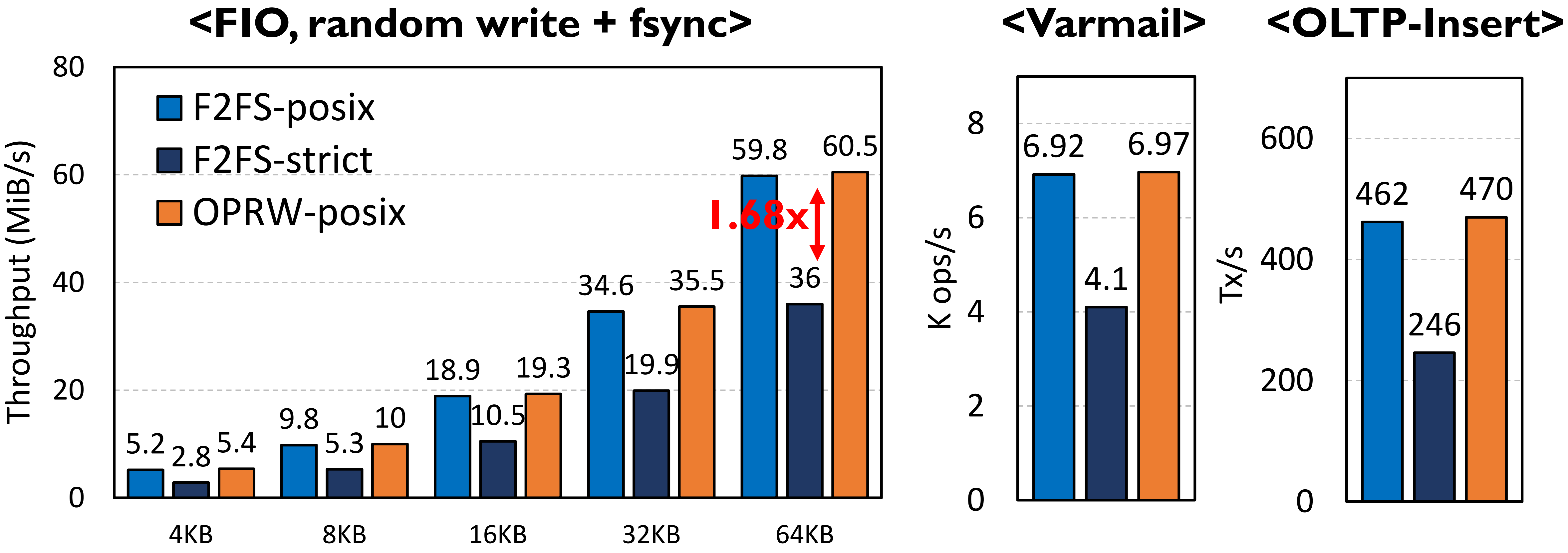
Data Mismatch → Fail

<Result>

	Failure Rate
F2FS-posix	3%
F2FS-strict	0%
OPRW-posix	0%

Performance of fsync() – Non-PLP Device

- OPRW can ensure data consistency, without sacrificing performance



Recovery Time

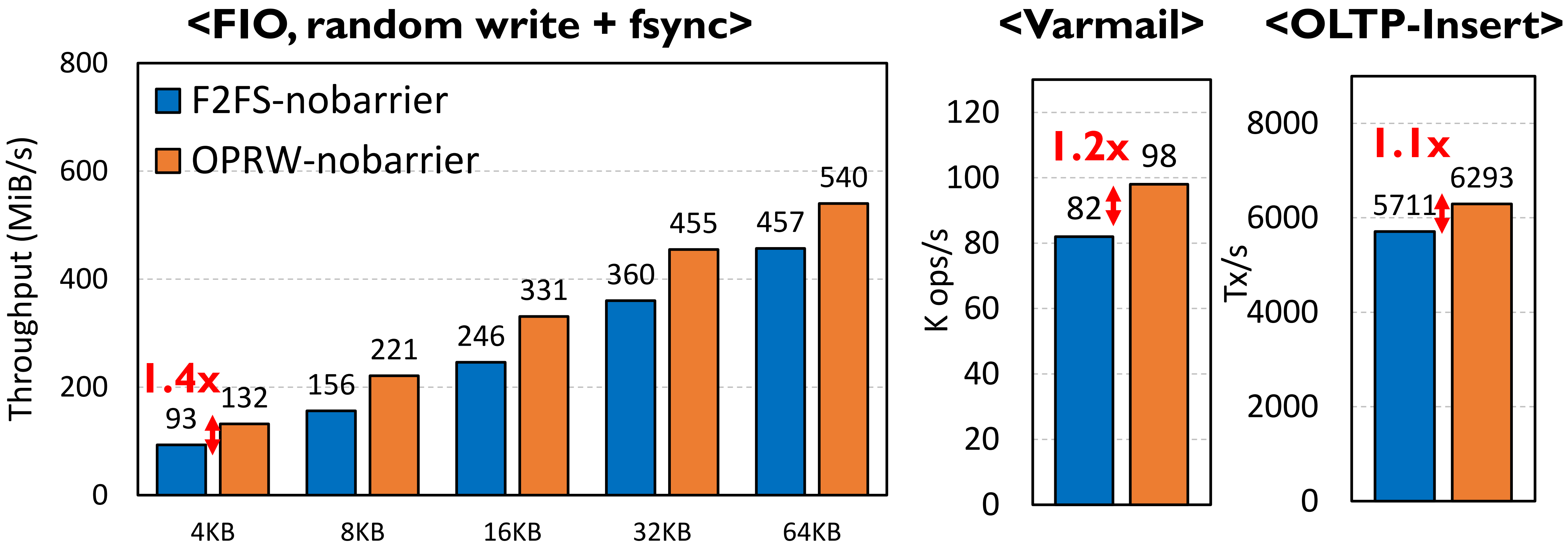
- OPRW has a negligible impact on recovery time

<Roll-forward execution time>

# scanned node	40791	50962	57500
F2FS (ms)	4177	7307	9751
OPRW (ms)	4380	7584	10093
Difference (ms)	**+203**	**+277**	**+342**

Performance of fsync() – PLP Device

- F2FS does not suffer from data consistency issues on PLP devices
- But OPRW can still provide performance gains



<FIO, random write + fsync>

<Varmail>

<OLTP-Insert>

Conclusion

- Point out the data corruption problem in F2FS

- Observe that write pointers can be used to determine the validity of data

- Propose OPRW technique using the write pointer provided by ZNS
  - Ensure data consistency with minimal overhead
  - Improve fsync() performance by removing synchronization actions

Thanks! Any Questions?