Fast and Failure-Consistent Updates of Application Data in Non-Volatile Main Memory File System

Jiaxin Ou, **Jiwu Shu**

(ojx11@mails.tsinghua.edu.cn)

Storage Research Laboratory
Department of Computer Science and Technology
Tsinghua University

清華大學
Tsinghua University

Outline

- ❑ **Background and Motivation**
- ❑ **FCFS Design**
- ❑ **Evaluation**
- ❑ **Conclusion**

Failure Consistency

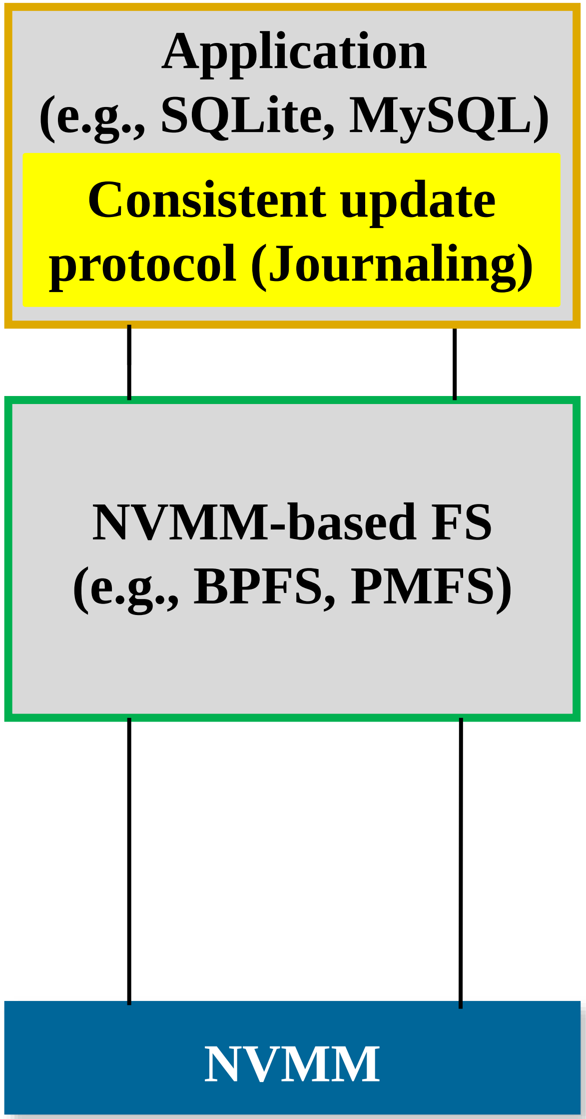
- **Failure Consistency (Failure-Consistent Updates)**
  - Atomicity and durability
  - The system is able to recover to a consistent state from unexpected system failures
- **Application Level Consistency**
  - Update multiple files atomically and selectively

Example:

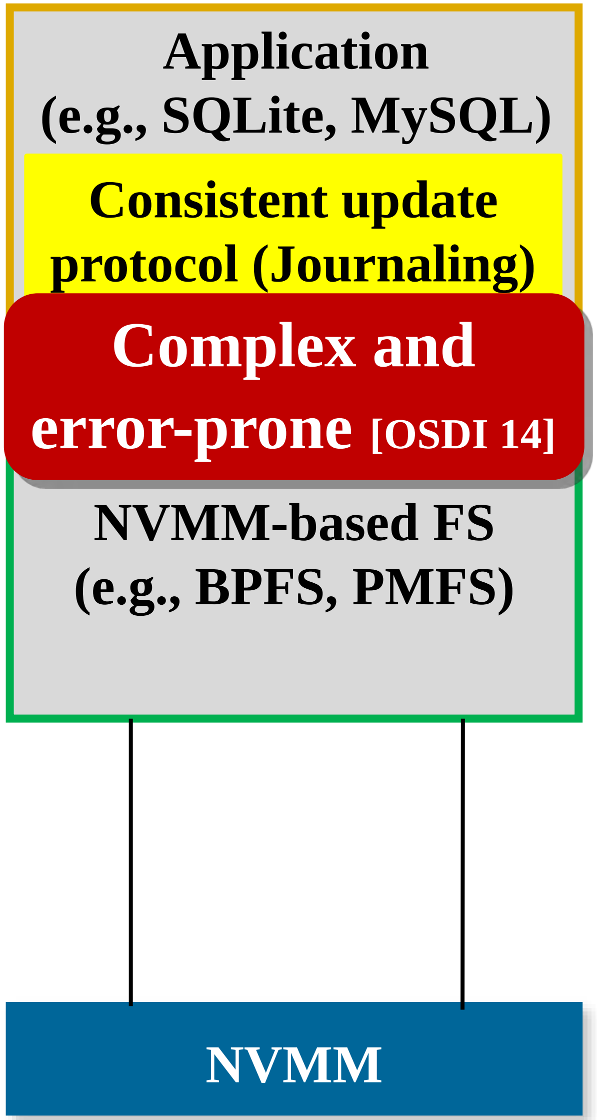
```
Atomic_Group{
write(fd1, “data1”);
write(fd2, “data2”);
}
```

→ Either both writes persist successfully, or neither does

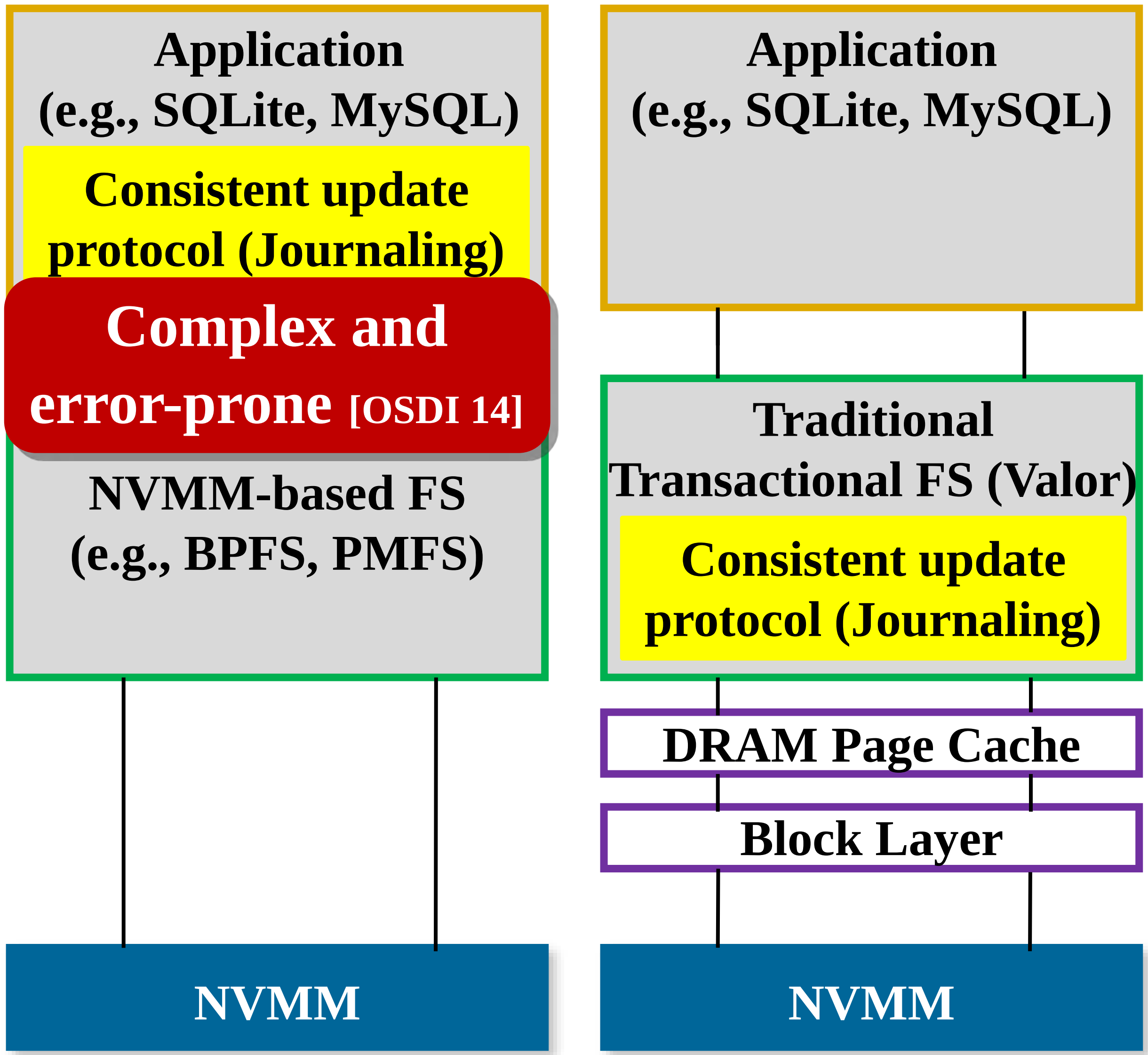
Existing approaches for supporting application level consistency on NVMM



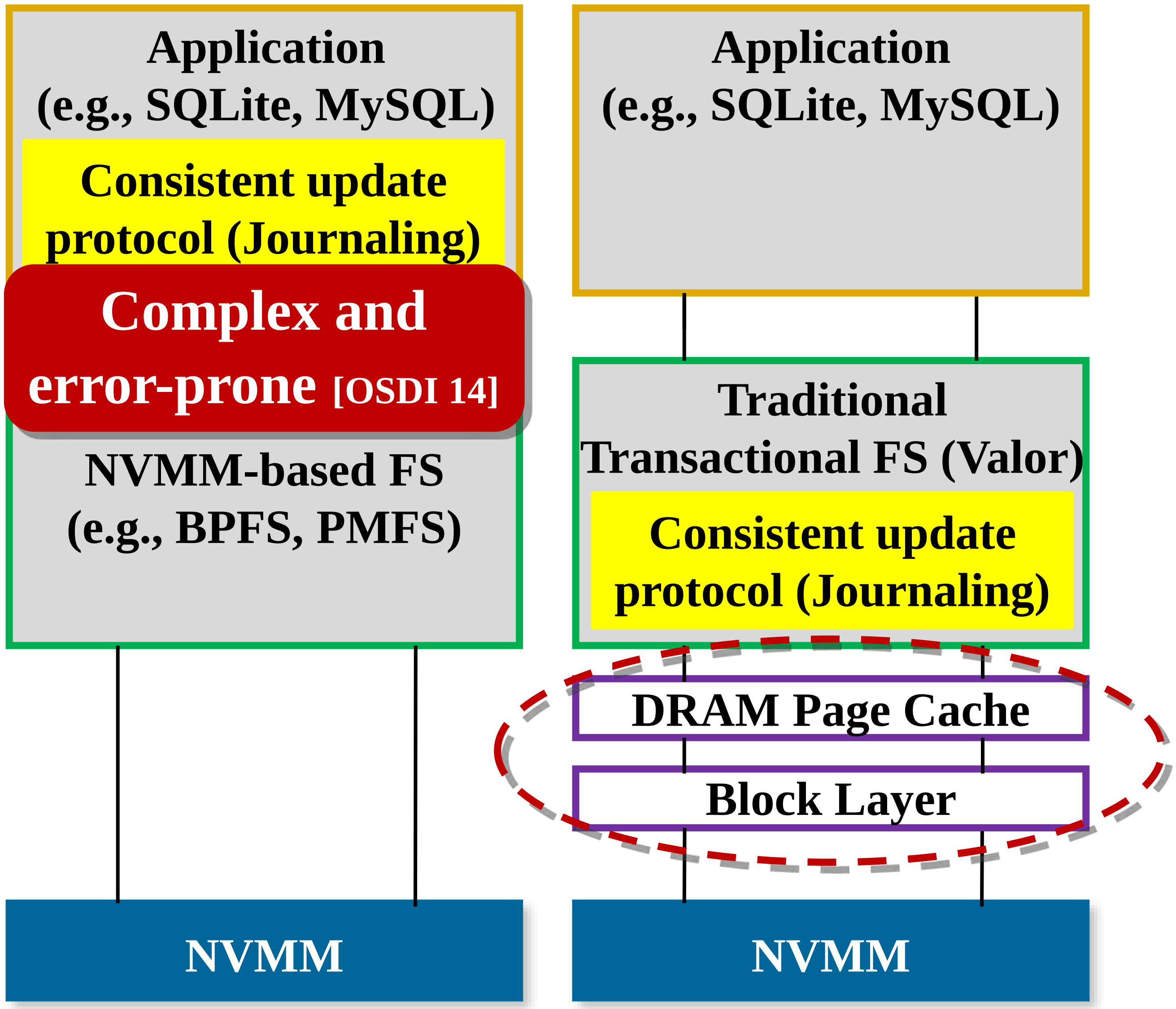
Existing approaches for supporting application level consistency on NVMM



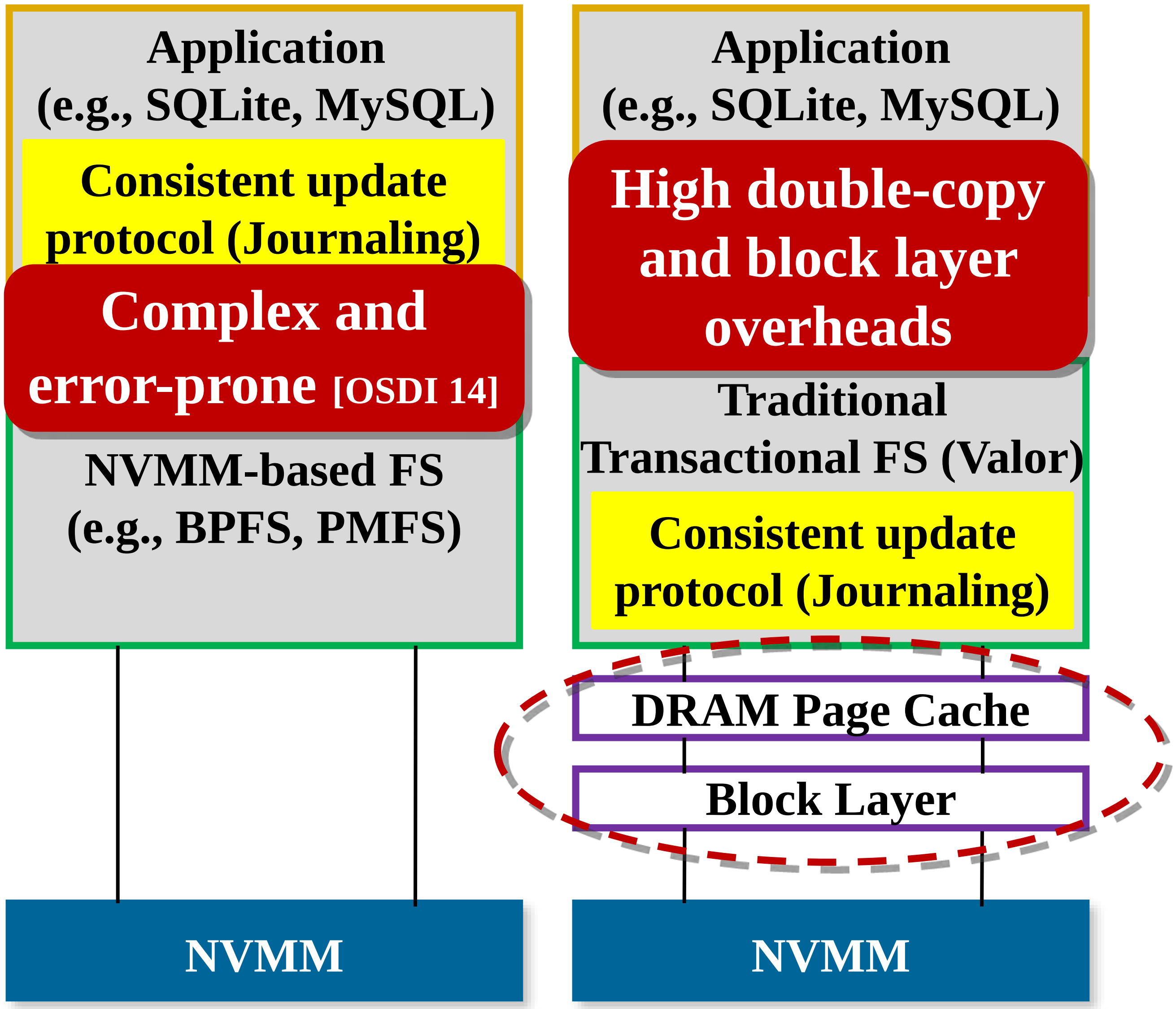
Existing approaches for supporting application level consistency on NVMM



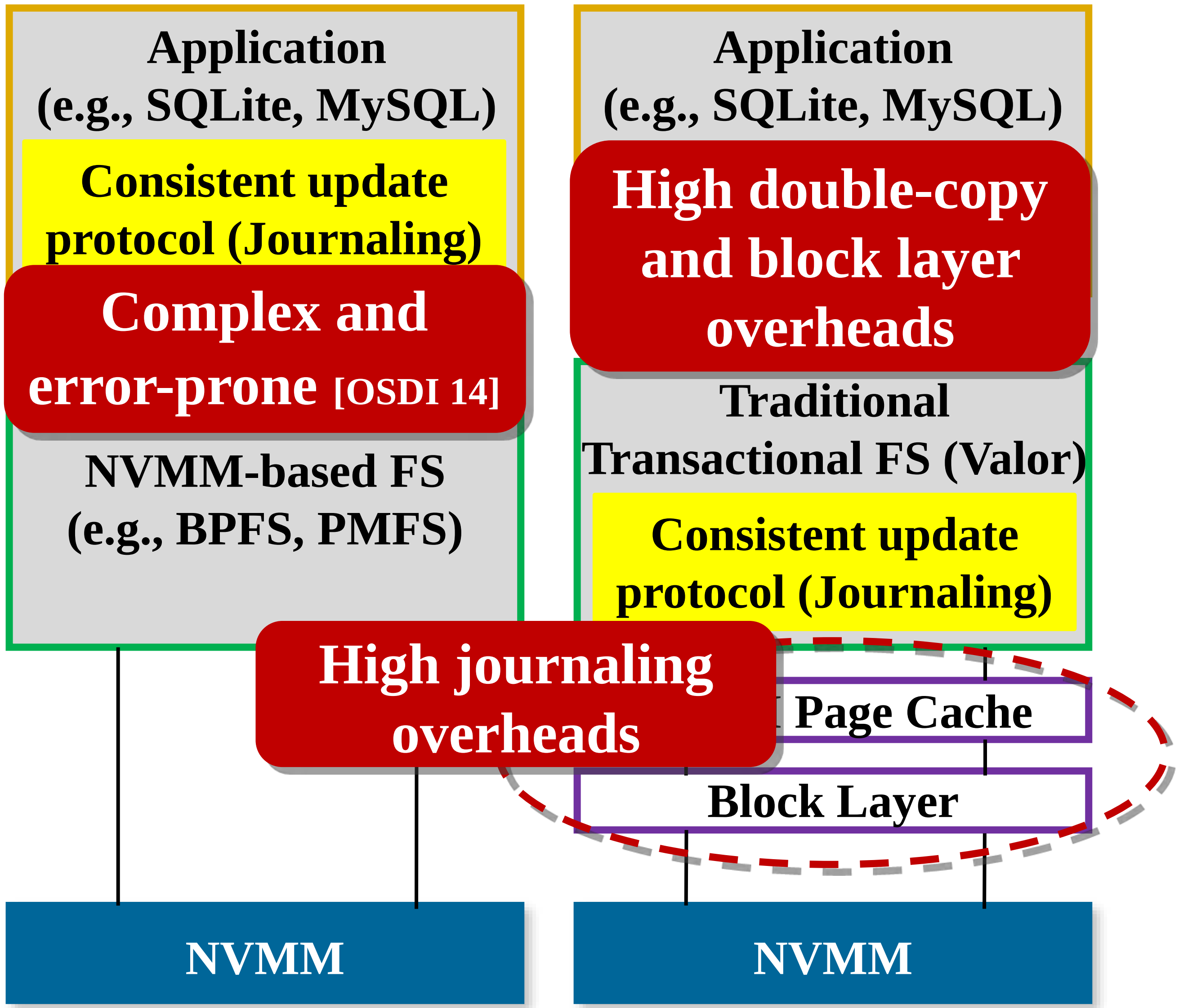
Existing approaches for supporting application level consistency on NVMM



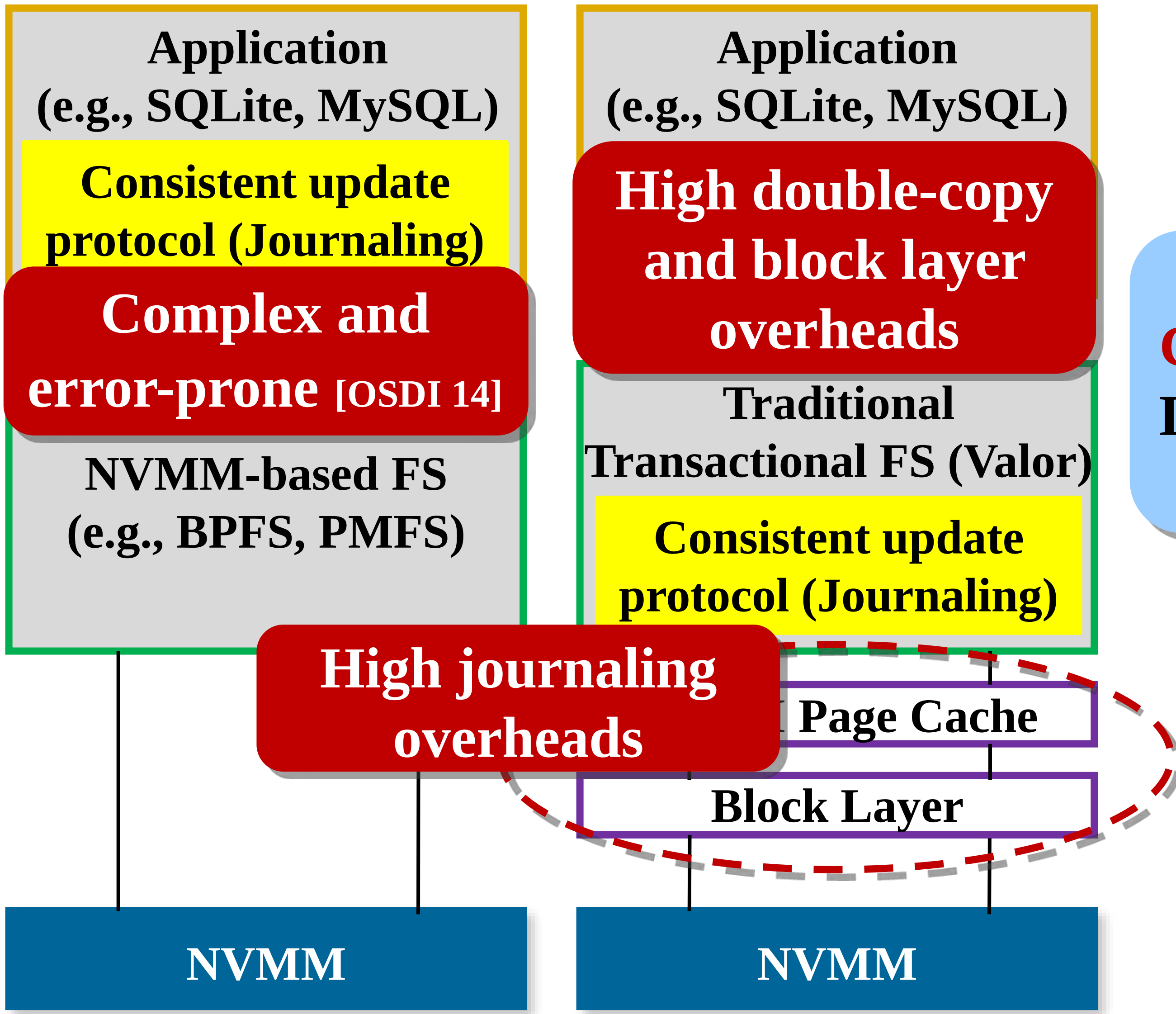
Existing approaches for supporting application level consistency on NVMM



Existing approaches for supporting application level consistency on NVMM

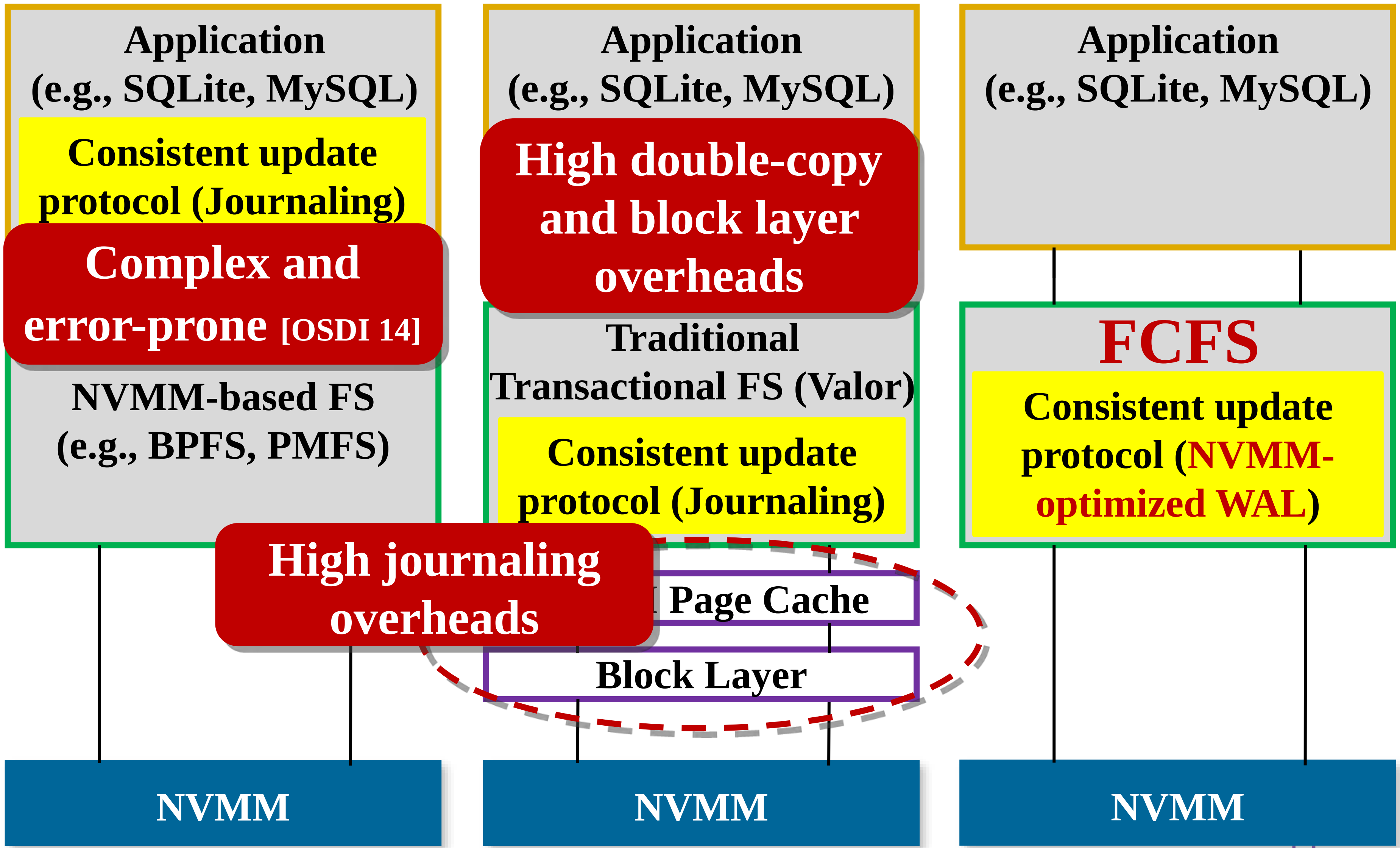


Existing approaches for supporting application level consistency on NVMM

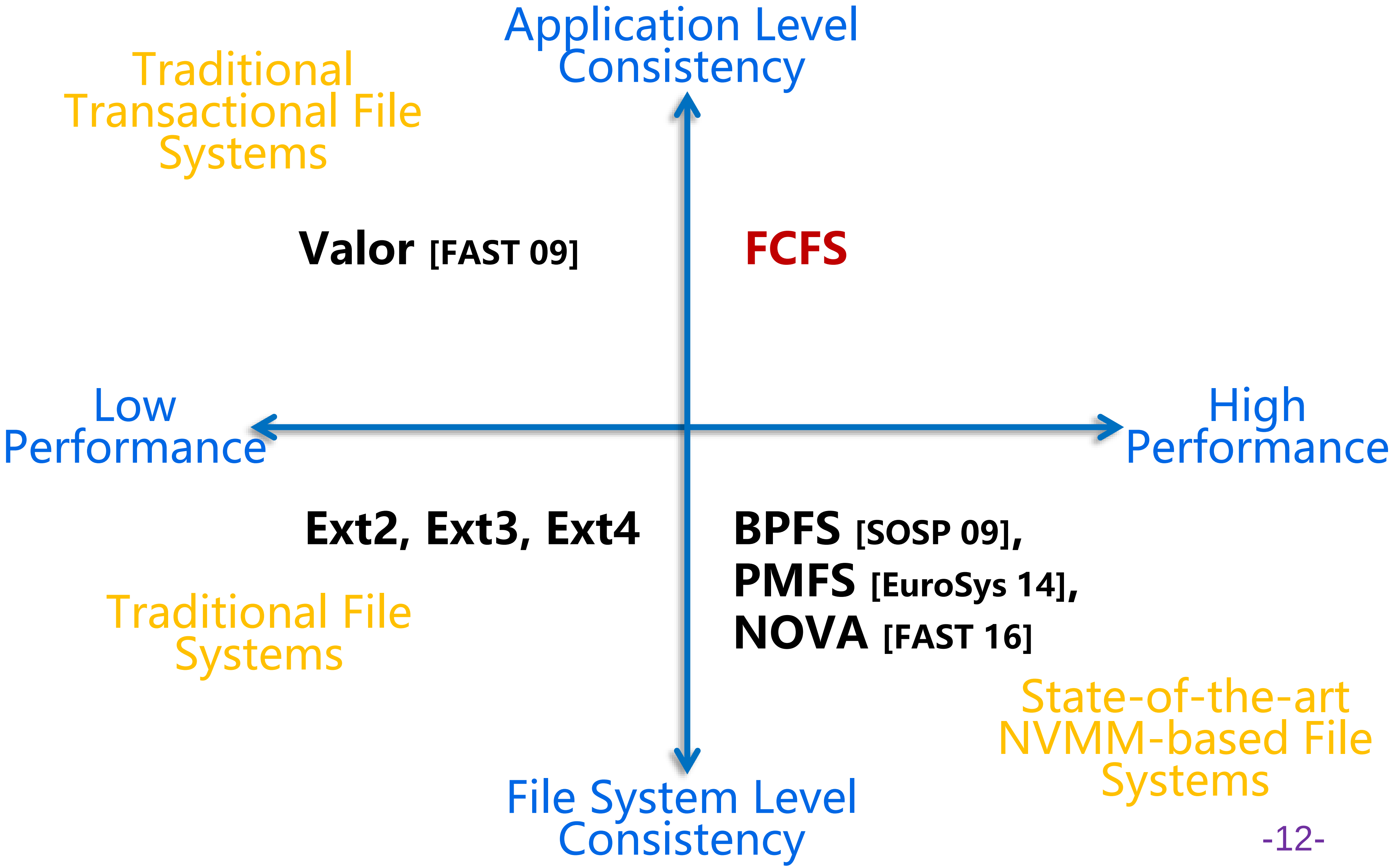


Our Goal:
Correct Application Level Consistency + High Performance

Existing approaches for supporting application level consistency on NVMM



Comparison of Different File Systems on NVMM Storage



Outline

- Background and Motivation
- **FCFS Design**
- Evaluation
- Conclusion

An Example of How to Use FCFS

```
Atomic_Group{
write(fd1, “data1”);
write(fd2, “data2”);
}
```

```
tx_id = tx_begin();
tx_add(tx_id, fd1);
tx_add(tx_id, fd2);
write(fd1, “data1”);
write(fd2, “data2”);
tx_commit(tx_id);
```

Interface	Description
tx_begin(TxInfo)	creates a new transaction
tx_add(TxID, Fd)	relates a file descriptor a designated transaction
tx_commit(TxID)	commits a transaction
tx_abort(TxID)	cancels a transaction entirely

Opportunities and Challenges for Providing Fast Failure-Consistent Update in NVMM FS

❑ Opportunities

- Direct access to NVMM allows fine-grained logging
- Asynchronous checkpointing can move the checkpointing latency off the critical path under low storage load

❑ Challenges

- #1: How to guarantee that a log unit will not be shared by different transactions? (Correctness)
- #2: How to balance the tradeoff between copy cost and log tracking overhead? (Performance)
- #3: How to improve checkpointing performance under high storage load? (Performance)

Key Ideas of FCFS

- **Our Goal:** to propose a novel NVMM-optimized file system (FCFS) providing the application-level consistency but without relying on the OS page cache layer

- **Key Ideas of FCFS (NVMM-optimized WAL):**
  - Hybrid Fine-grained Logging to address Challenge #1 and #2
    - Decouple the logging method of metadata and data updates
    - Using fast *Two-Level Volatile Index* to track uncheckpointed log data
  - Concurrently Selective Checkpointing to address Challenge #3
    - Committed updates to different blocks are checkpointed concurrently
    - Committed updates of the same block are checkpointed using *Selective Checkpointing Algorithm*

1. Hybrid Fine-grained Logging

- **Challenge #1:** Correctness
  - **Logging granularity** (byte vs cacheline)
    - – a log unit should not be shared by different transactions

Metadata	**Data**
• Smallest unshared unit is ***a metadata structure*** • a metadata structure can be of any size (e.g., directory entry)	• Smallest unshared unit is ***a file*** • File is allocated based on block
Byte Granularity **Cacheline Granularity**	**Byte Granularity** **Cacheline Granularity**

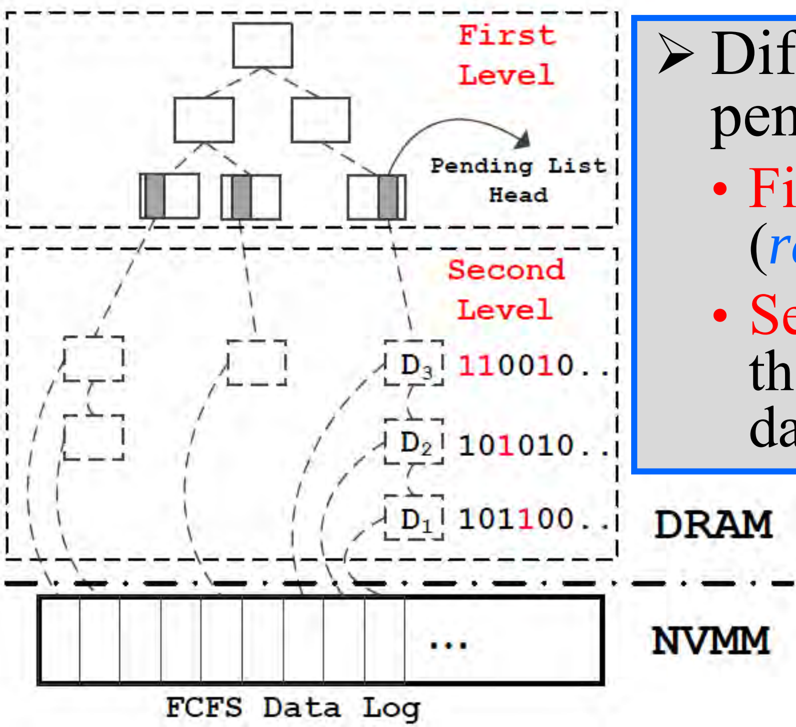
1. Hybrid Fine-grained Logging

- **Challenge #2:** Performance tradeoff **:** log tracking cost vs data copy cost
  - Impacted by *logging granularity* (byte vs cacheline) & *logging mode* **(**undo **vs** redo**)**

Metadata (update size is small)	**Data (update size can be very large)**
• Byte granularity redo logging has ***high*** log tracking cost	• Undo logging has ***high*** data copy cost for *large update* • Byte granularity redo logging has ***high*** log tracking cost
Byte granularity undo logging	**Cacheline granularity redo logging**

1. Hybrid Fine-grained Logging

- **Another Challenge:** How to reduce the log tracking cost of the data log (*cacheline granularity redo logging*) ?
  - Example: each 64B cacheline log unit may need at least 16 bytes of index
- **Solution:** ***Two-Level Volatile Index***



- Different versions' log blocks form a pending list
  - First level: logic block ⟶ pending list head (*radix tree*)
  - Second level: traversing the pending list to get the physical block which contains the latest data of a cacheline using the *cacheline bitmap*

Overheads: Each 4KB log blocks requires at most 16 bytes of index data (first level) and 8 bytes of bitmap (second level)

(Logic block, cacheline id) ⟶ (physical block)

2. Concurrently Selective Checkpointing

Challenge #3: How to improve checkpointing performance under high storage load?

- **Concurrent Checkpointing**
  - Committed updates to different blocks are checkpointed concurrently to enhance the concurrency of checkpointing
- **Selective Checkpointing**
  - Committed updates of the same block are checkpointed using *Selective Checkpointing Algorithm* to reduce the checkpointing copy overhead

2. Concurrently Selective Checkpointing

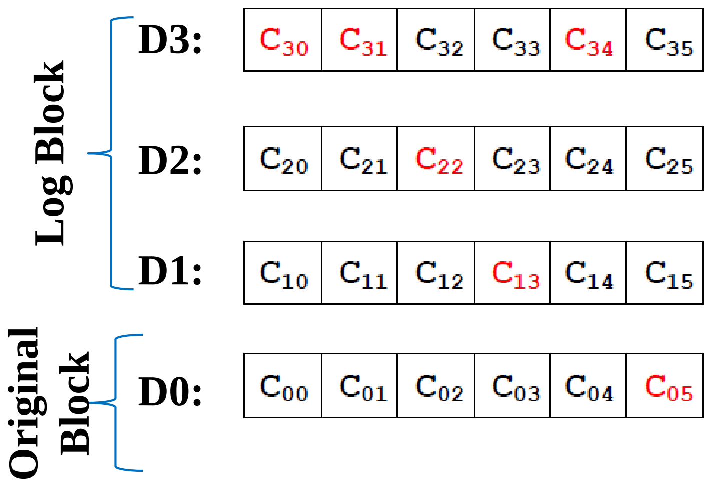
- **Another Challenge:** How to ensure correct failure recovery due to out-of-order checkpointing?
  - What if a newer log entry is deallocated before an older log entry and the system crashes before deallocating the older one?
  - How to guarantee that the commit log entry is deallocated at last?

- **Solution:** Maintaining two *ordering properties* during log deallocation
  - Redo log entries are deallocated following the pending list order
  - Using a global committed list to ensure the deallocation order between the commit log entry and other metadata/data log entries of a transaction?

2. Concurrently Selective Checkpointing

❑ Selective Checkpointing Algorithm

– Leveraging NVMM's byte-addressability to reduce the checkpointing copy overhead

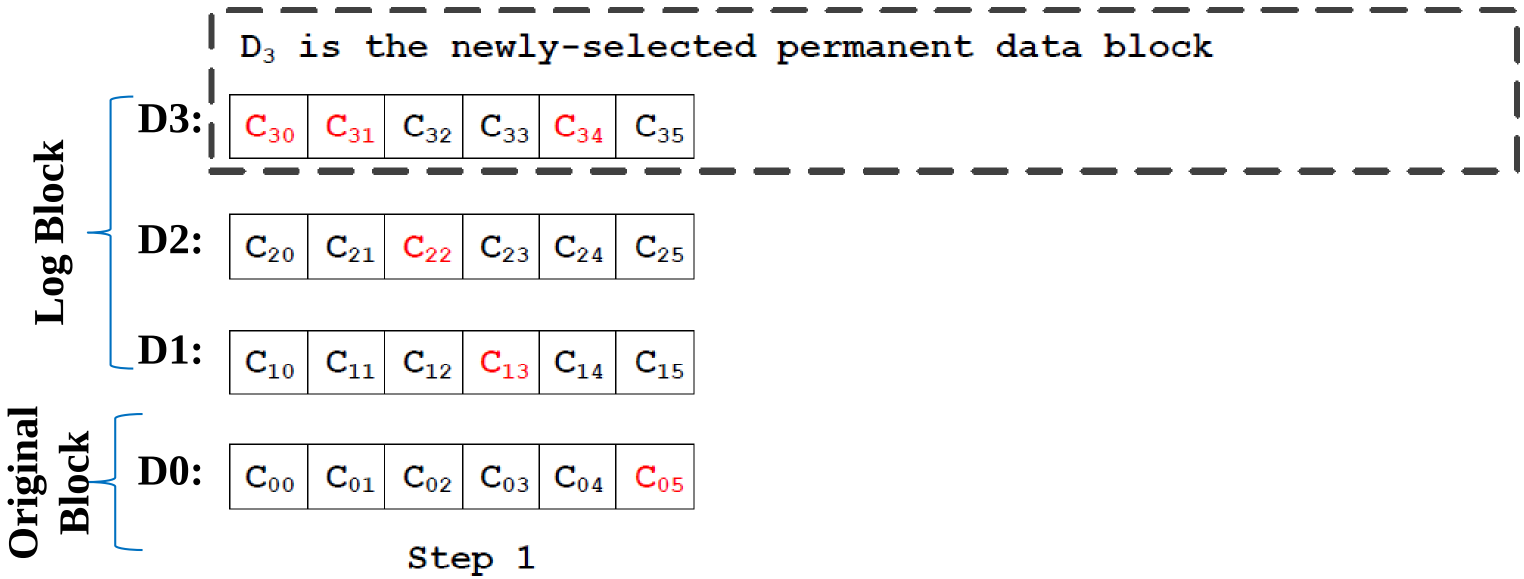


Note: D0~D3 refers to different versions of block D; C_{ij} is the jth cacheline in the ith version of block D

2. Concurrently Selective Checkpointing

❑ Selective Checkpointing Algorithm

– Leveraging NVMM's byte-addressability to reduce the checkpointing copy overhead



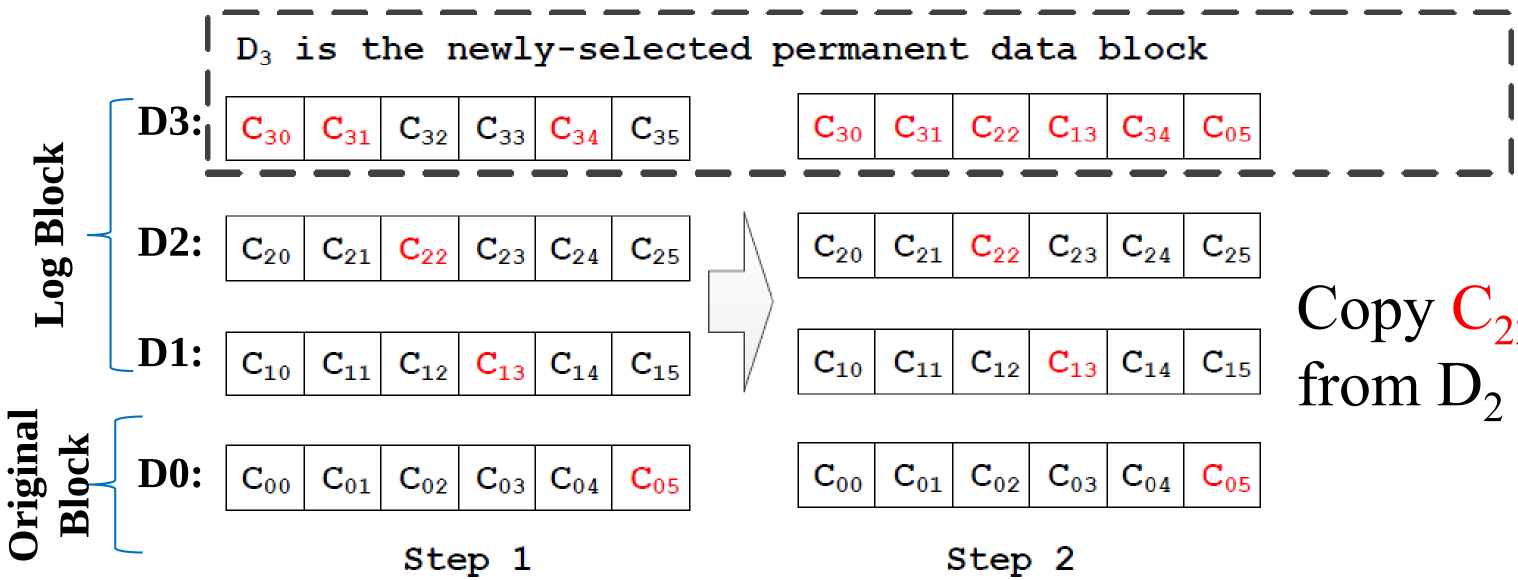
Note: D0~D3 refers to different versions of block D; C_{ij} is the jth cacheline in the ith version of block D

Step1: a new permanent data block, which has the largest number of latest cachelines, is ***carefully selected***

2. Concurrently Selective Checkpointing

❑ Selective Checkpointing Algorithm

– Leveraging NVMM's byte-addressability to reduce the checkpointing copy overhead



Copy C_{22} , C_{13} , C_{05} from D_2 , D_1, D_0 to D_3

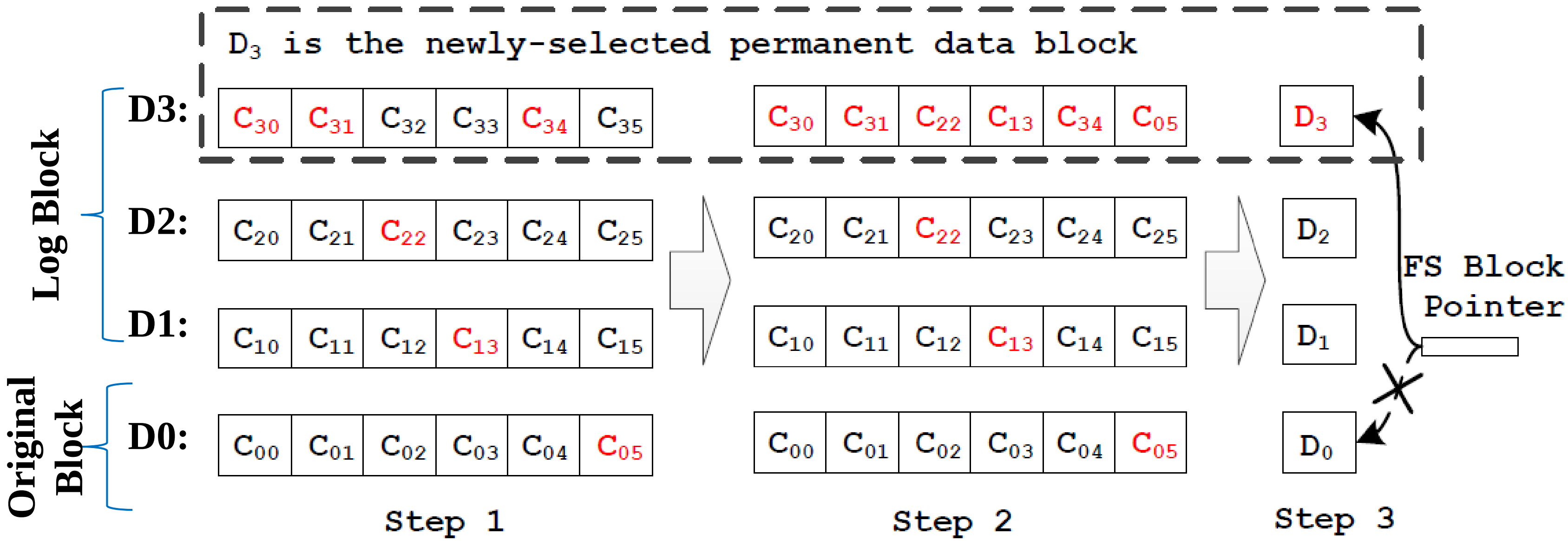
Note: D0~D3 refers to different versions of block D; C_{ij} is the jth cacheline in the ith version of block D

Step2: Copy the latest cacheline data from other blocks to the newly-selected permanent block

2. Concurrently Selective Checkpointing

❑ Selective Checkpointing Algorithm

– Leveraging NVMM's byte-addressability to reduce the checkpointing copy overhead



Note: D0~D3 refers to different versions of block D; C_{ij} is the jth cacheline in the ith version of block D

Step3: Modify the reference to origin original block to refer to newly-selected permanent block atomically

2. Concurrently Selective Checkpointing

❑ Overhead Comparison

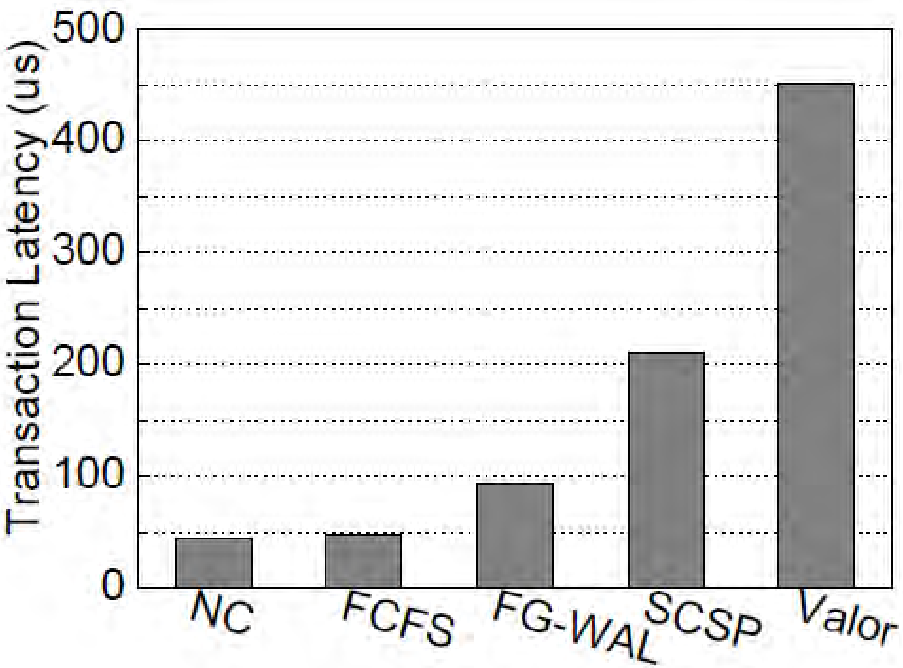
Traditional Constant Checkpointing	Selective Checkpointing
• Copy 3 blocks = 3 * 6 * 64 B = 1152 B	• Copy 3 cacheline and modify one block pointer = 3 * 64 B + 8 B = 200 B

Selective Checkpointing Algorithm significantly reduces the checkpointing copy overhead

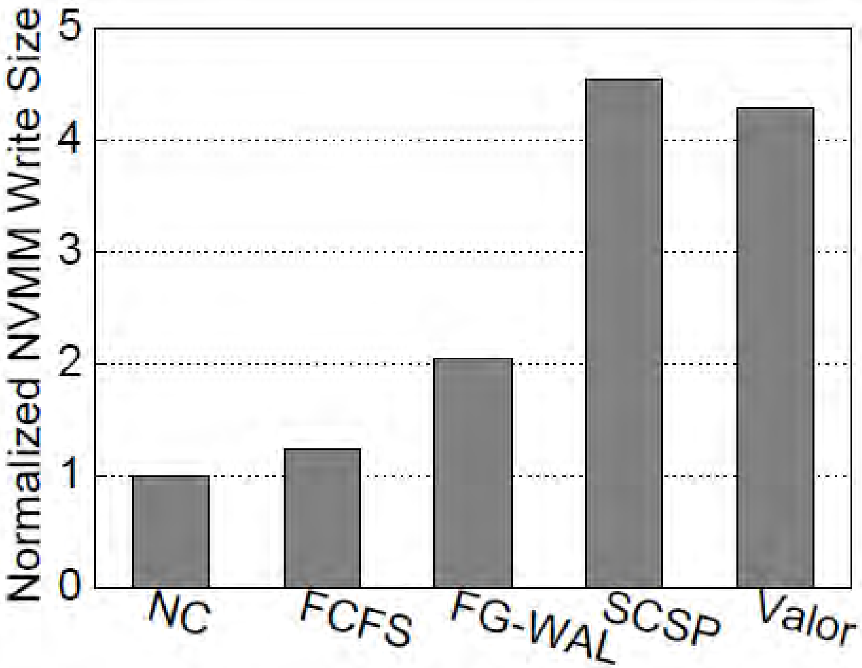
Outline

- ❑ Background and Motivation
- ❑ FCFS Design
- ❑ **Evaluation**
- ❑ Conclusion

Evaluations of Failure-Consistent Updates



(a) Performance in Single-Thread Mode

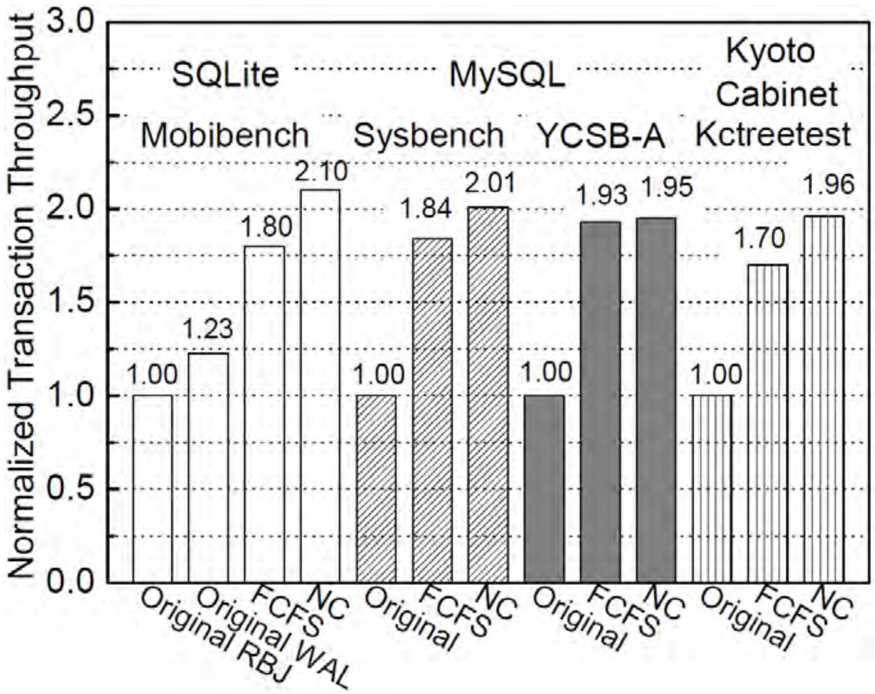


(b) NVMM Write Size

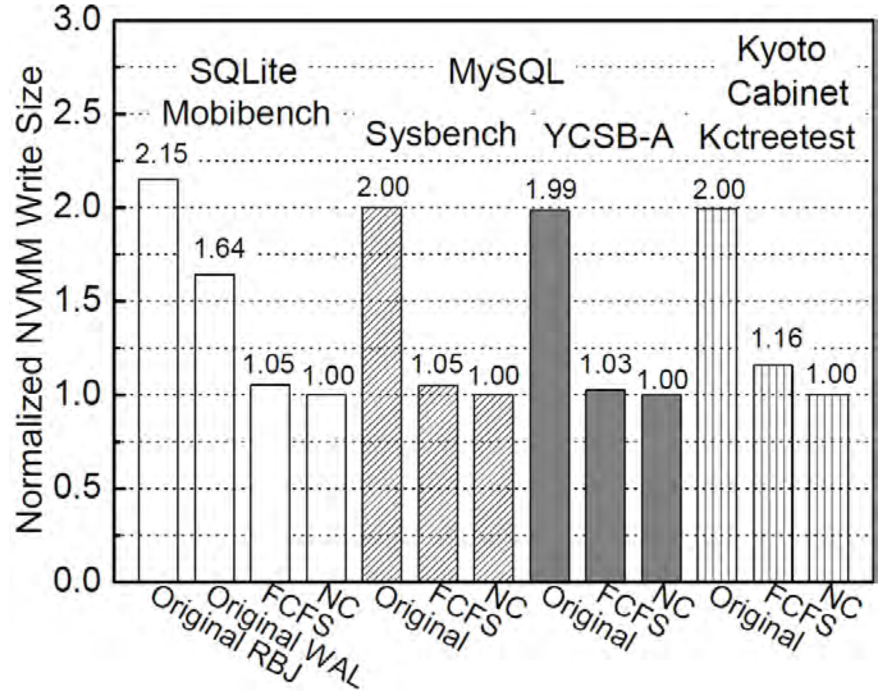
- NC is a no-consistency system
- FG-WAL implements the failure-consistent update protocol using fine-grained write-ahead logging
- SCSP implements the failure-consistent update protocol using short-circuit shadow paging [SOSP 09]
- Valor is a traditional transactional file system

❑ The latency of FCFS-based version is the lowest among all failure-consistent versions (FG-WAL, SCSP, Valor)

Evaluations of Real Applications



Throughput Performance



NVMM Write Size

- NC turns off the transactional part of each application

❑ FCFS-based applications outperform the original ones by up to 93% (MySQL running YSCB workload)

Outline

- Background and Motivation
- FCFS Design
- Evaluation
- **Conclusion**

Conclusion

- Existing NVMM file systems do not guarantee the consistency of application data, while application's own consistency protocols are complex and error-prone
- FCFS is the first NVMM-optimized file system which enables both correctness and high performance for applications to consistently update their data on NVMM storage
- FCFS employs an NVMM-optimized WAL scheme to reduce the overhead towards supporting failure consistency by fully leveraging NVMM's byte addressability and high concurrency but without relying on the page-cache layer
- FCFS's failure-consistent update protocol and FCFS-based applications significantly outperform conventional protocols and original applications respectively

Thank You !

Jiaxin Ou, Jiwu Shu
(ojx11@mails.tsinghua.edu.cn)