FLORIDA STATE UNIVERSITY

UNIVERSITY OF TORONTO

Silhouette: Leveraging Consistency Mechanisms to Detect Bugs in Persistent Memory-Based File Systems

Bing Jiao
Florida State University

Ashvin Goel
University of Toronto

Andy Wang
Florida State University

Detecting Bugs in Persistent Memory FSes

Challenges:

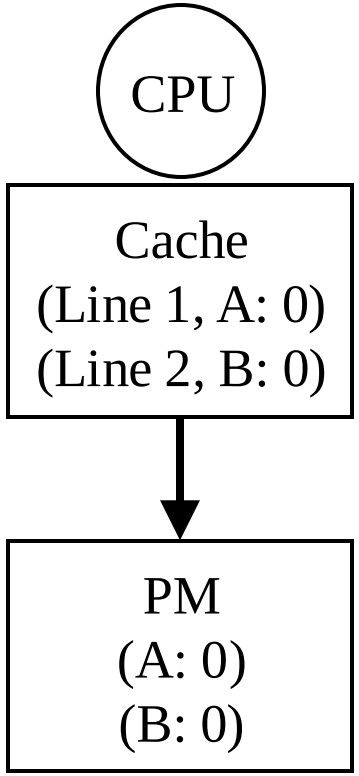
- Persistent Memory (PM) programs are prone to bugs due to out-of-order persistence
- Exponential search space in PM FSes
  - N **in-flight** (not guaranteed to be persisted) stores at a **fence point** → 2^N **crash scenarios (crash plans)**

Proposed **Silhouette,** which:

- First checks whether a PM file system implements its consistency mechanisms correctly
- If so, then all stores associated with the consistency mechanism are not reordered during testing, which dramatically reduces the search space

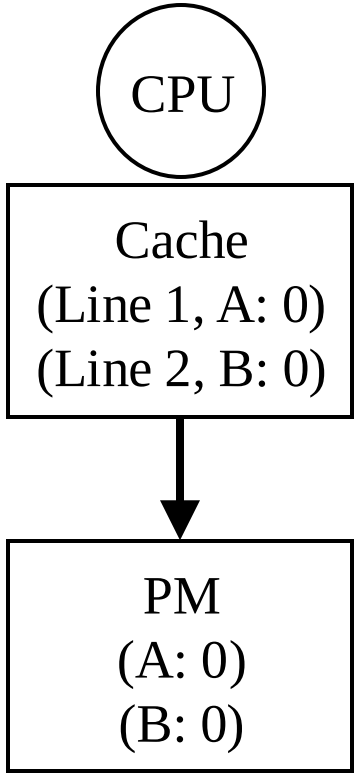
Out-of-order Persistence

1. Store A, 2
2. Flush A
3. Store B, 3
4. Fence



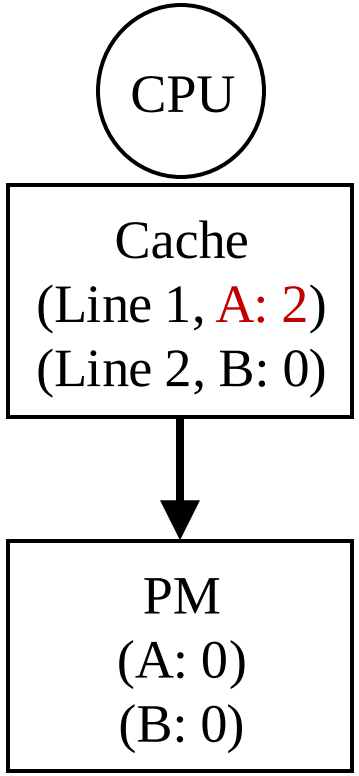
Out-of-order Persistence

1. Store A, 2
2. Flush A
3. Store B, 3
4. Fence



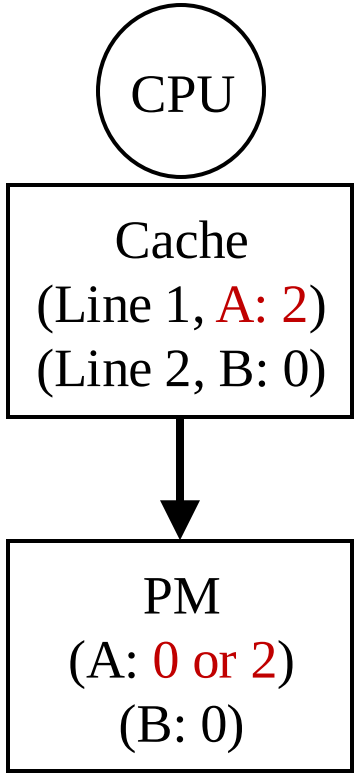
Out-of-order Persistence

1. Store A, 2
2. Flush A
3. Store B, 3
4. Fence



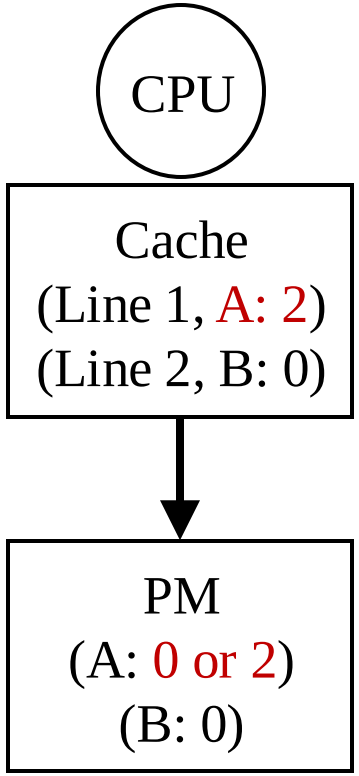
Out-of-order Persistence

1. Store A, 2
2. Flush A
3. Store B, 3
4. Fence



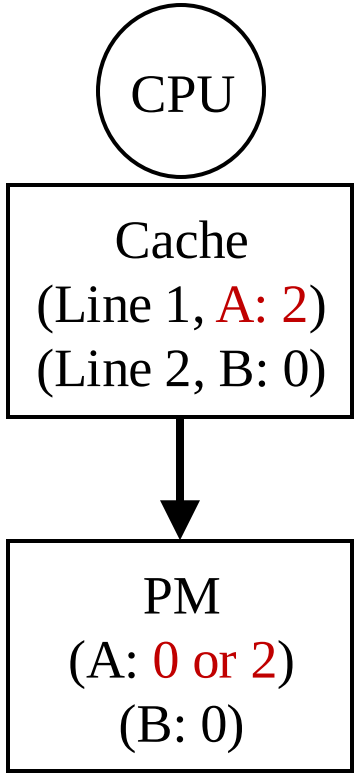
Out-of-order Persistence

1. Store A, 2
2. Flush A
3. Store B, 3
4. Fence



Out-of-order Persistence

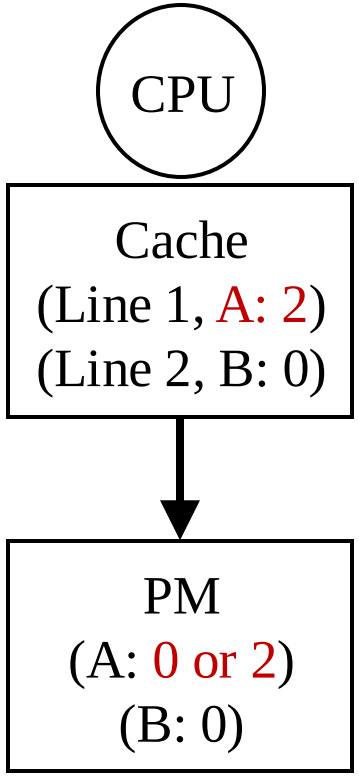
1. Store A, 2
2. Flush A
3. Store B, 3
4. Fence



The execution of an instruction might be reordered by either compiler or CPU.

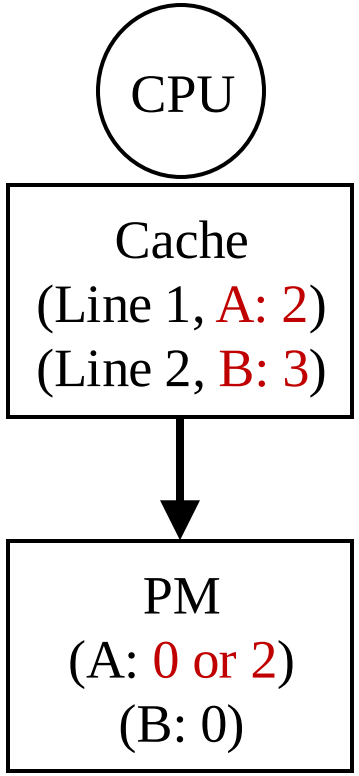
Out-of-order Persistence

1. Store A, 2
2. Flush A
3. Store B, 3
4. Fence



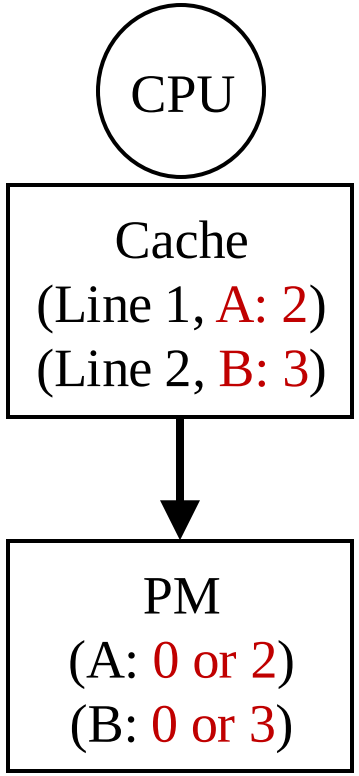
Out-of-order Persistence

1. Store A, 2
2. Flush A
3. Store B, 3
4. Fence



Out-of-order Persistence

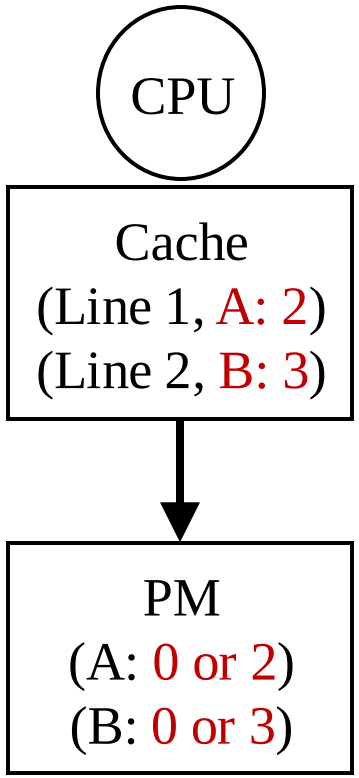
1. Store A, 2
2. Flush A
3. Store B, 3
4. Fence



Out-of-order Persistence

1. Store A, 2
2. Flush A
3. Store B, 3

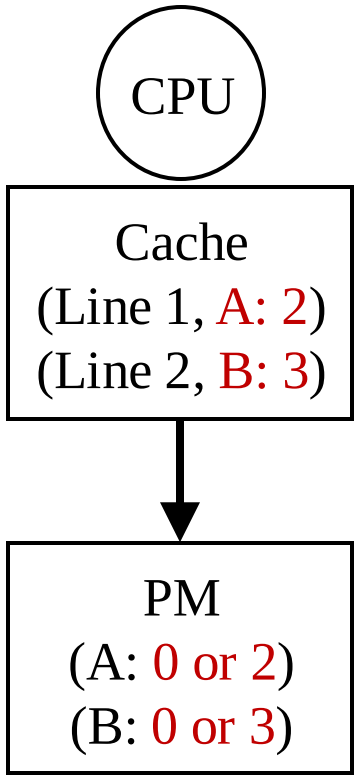
4. Fence



Out-of-order Persistence

1. Store A, 2
2. Flush A
3. Store B, 3

4. Fence



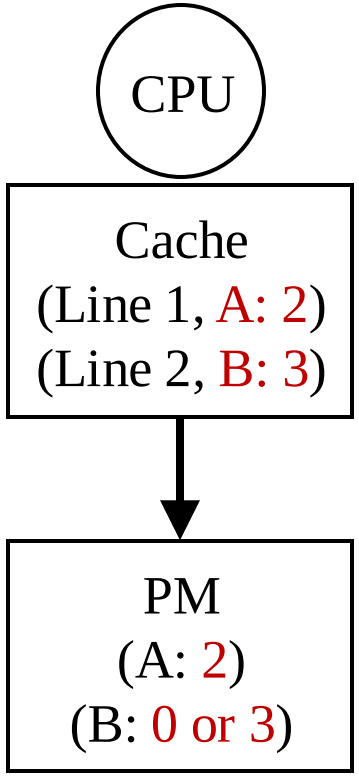
4 (i.e., 2^2) crash plans:

- A and B are persisted
- A is persisted
- B is persisted
- A and B are not persisted

N in-flight stores → 2^N crash plans

Out-of-order Persistence

1. Store A, 2
2. Flush A
3. Store B, 3
4. Fence

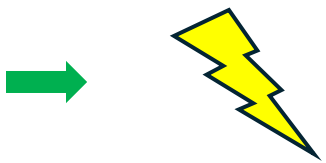


Out-of-order Persistence

1. Store A, 2
2. Flush A
3. Store B, 3
4. Fence
5. Store C, 4
6. Store D, 5
7. Flush C, D
8. Fence

Out-of-order Persistence

1. Store A, 2
2. Flush A
3. Store B, 3
4. Fence
5. Store C, 4
6. Store D, 5
7. Flush C, D



8. Fence

3 in-flight stores → 8 (i.e., 2^3) crash plans

N in-flight stores → 2^N crash plans

How to avoid search space explosion?

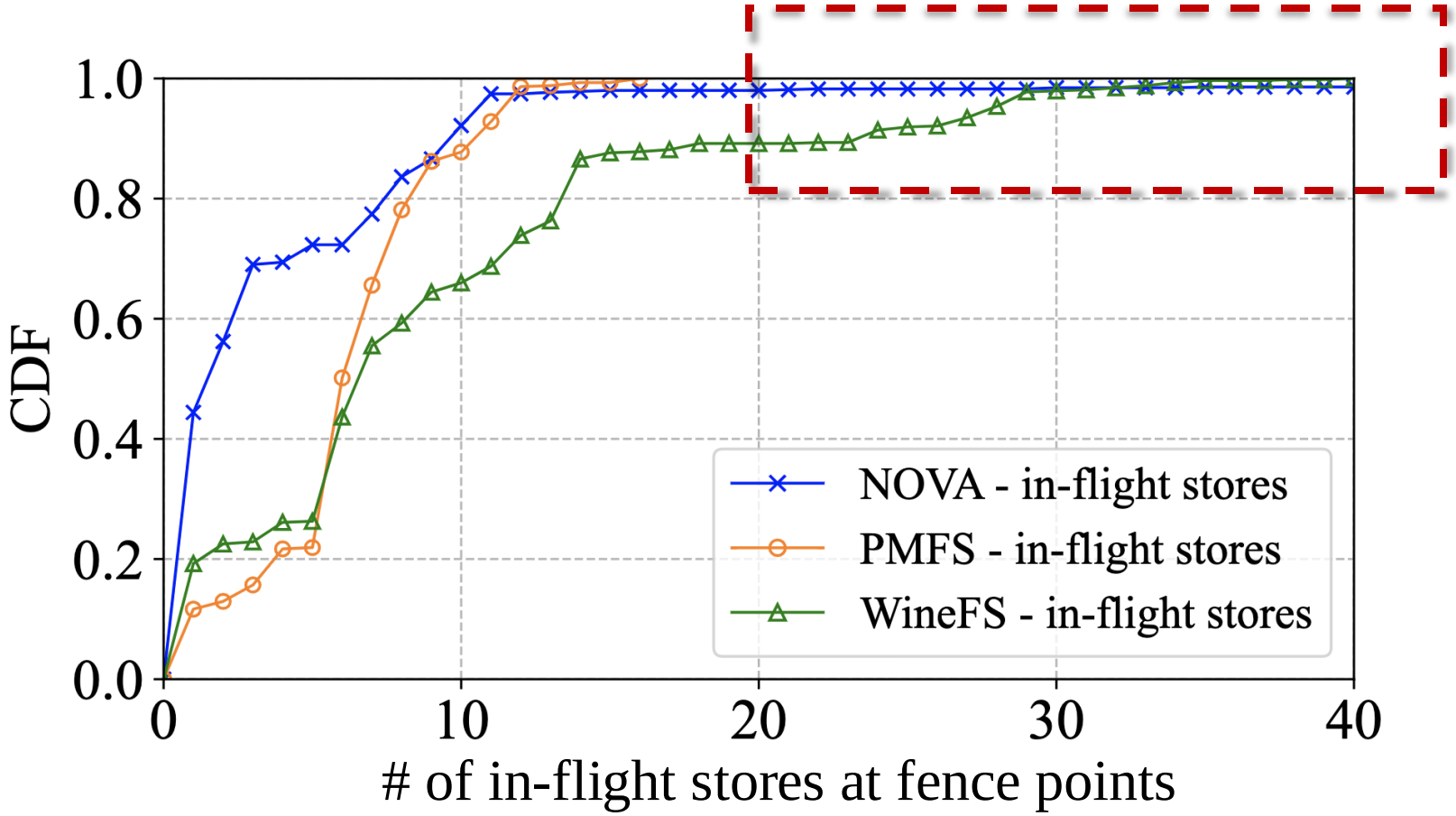
[1] Assume B, C, and D are in different cache lines.

Observations

- PM FSes can have 20-40 in-flight stores at fence points

  → millions and trillions of crash plans

In-flight Stores in FSes



- **Long tails**

Fig 1. Cumulative Distribution Function (CDF) of in-flight stores in PMFS, NOVA, and WineFS under the ACE workloads.

Observations

- PM FSes can have 20-40 in-flight stores at fence points

  → millions and trillions of crash plans

- PM FSes use well-known crash-consistency mechanisms (e.g., journaling) to provide atomicity and durability guarantees

  → < ~10 in-flight stores that are not associated with any crash-consistency mechanisms (denote unprotected store) at fence points

  → < ~1000 crash plans

In-flight vs. Unprotected In-flight Stores

- Focusing on in-flight stores not associated with consistency mechanisms (unprotected stores) eliminates the long tail.

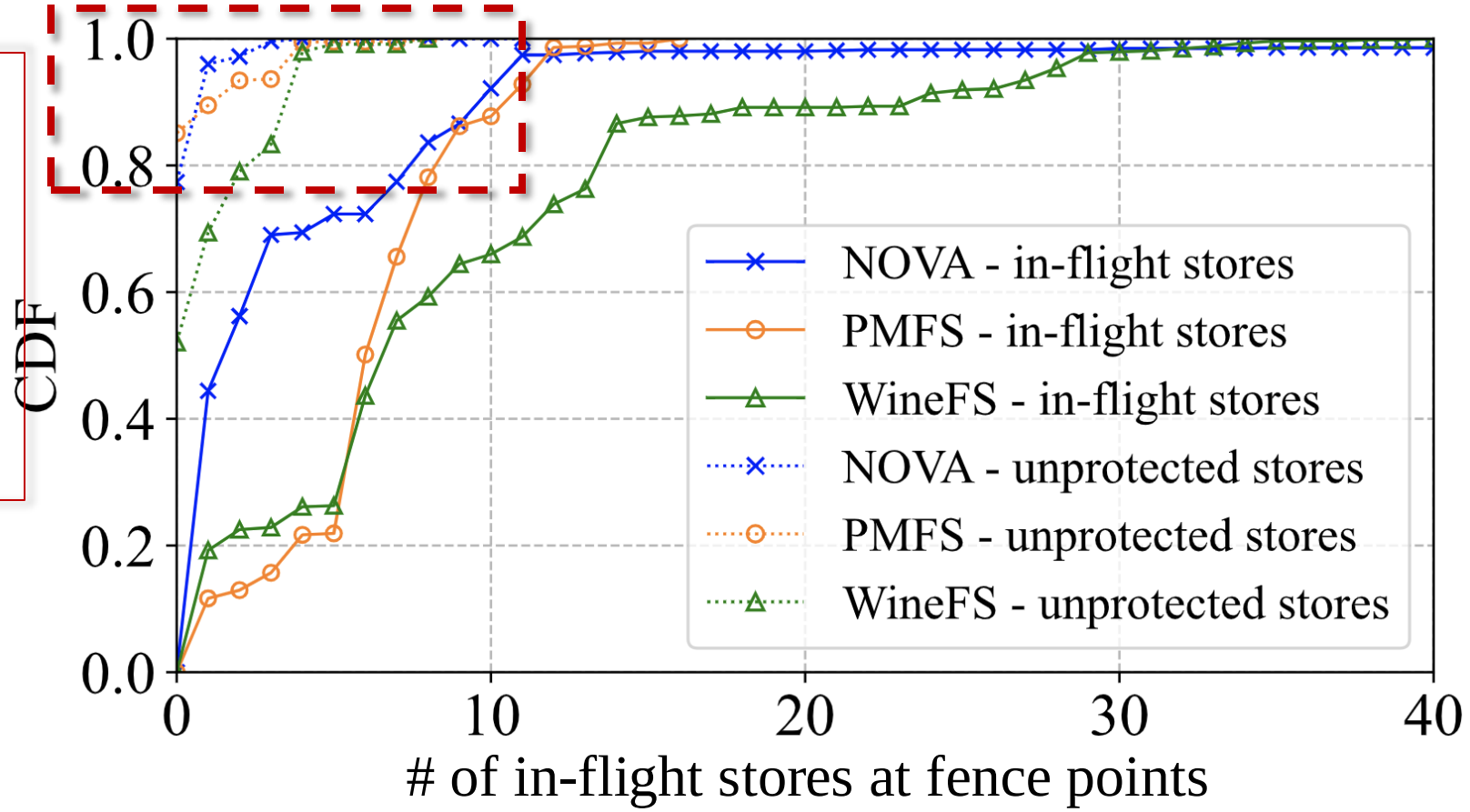


Fig 1. CDF of in-flight and unprotected stores in PMFS, NOVA, and WineFS under the ACE workloads.

In-flight vs. Unprotected In-flight Stores

- Focusing on in-flight stores not associated with consistency mechanisms (unprotected stores) eliminates the long tail.

$$2^{10} = 1024$$

Can we further reduce the number of crash plans?

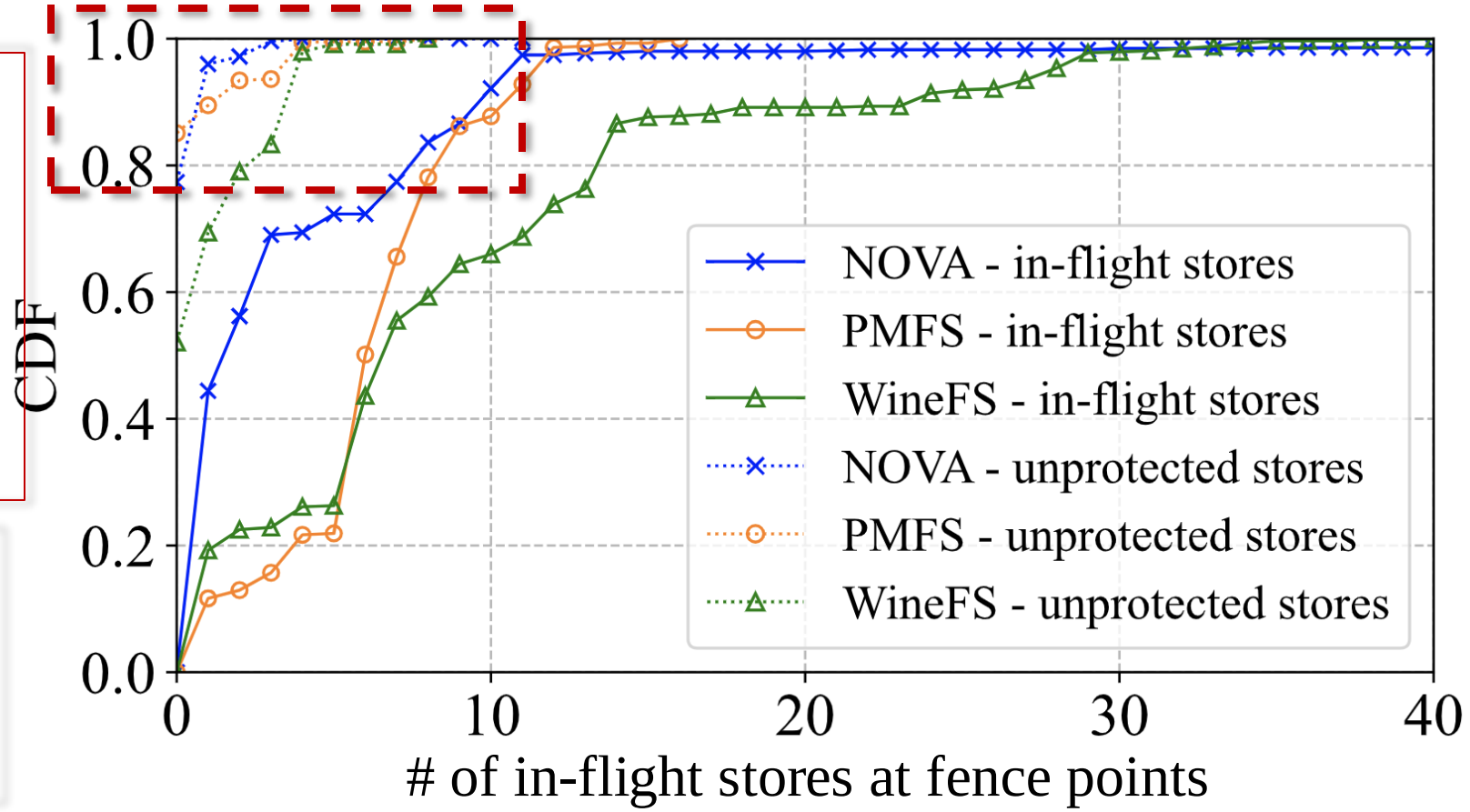


Fig 1. CDF of in-flight and unprotected stores in PMFS, NOVA, and WineFS under the ACE workloads.

Heuristic Reordering

Observation

- PM programs often use one critical store to indicate consistency of n in-flight stores
  - Flag, atomic variable, epoch, etc.
  - If the critical store is not ordered with respect to other in-flight stores → bug

However, we do not know whether a store is a critical store.

Heuristic: for each unprotected in-flight store S, we assume S is a critical store and test only 2 crash plans (2CP):

- Persists only S, leave other stores unpersisted
- Persists all the other stores, leave S unpersisted

Heuristic Reordering

Observation

- PM programs often use one critical store to indicate consistency of n in-flight stores
  - Flag, atomic variable, epoch
  - If th… bug

However, we c…

With N in-flight stores*:*

Traditionally combinatorial approach: 2^N *crash plans*

Our 2CP approach*:* $2N$ *crash plans*

Heuristic: for each unprotected in-flight store S, we assume S is a critical store and test only 2 crash plans (2CP):

- Persists only S, leave other stores unpersisted
- Persists all the other stores, leave S unpersisted

Silhouette

Key Ideas:

- Detect crash-consistency mechanisms and check their invariants to ensure the associated stores are crash-consistent w.r.t. persistence reordering.
- Use the heuristic (2CP) approach to test unprotected stores.

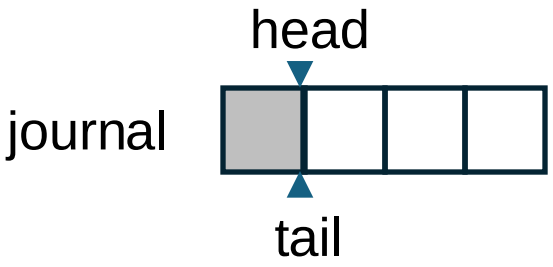
Major Challenges:

- How to define invariants for crash-consistency mechanisms?
- How to check these invariants?
- …

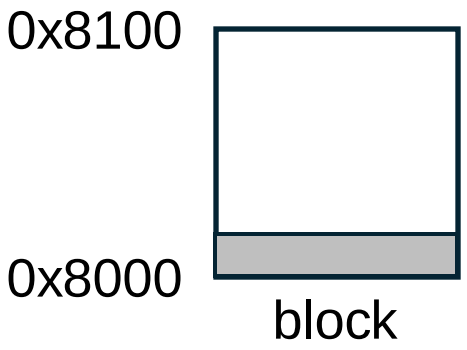
More details in the paper

Consistency Invariants

Crash-consistency mechanisms order writes in multiple phases.



Take an undo-journal as an example:

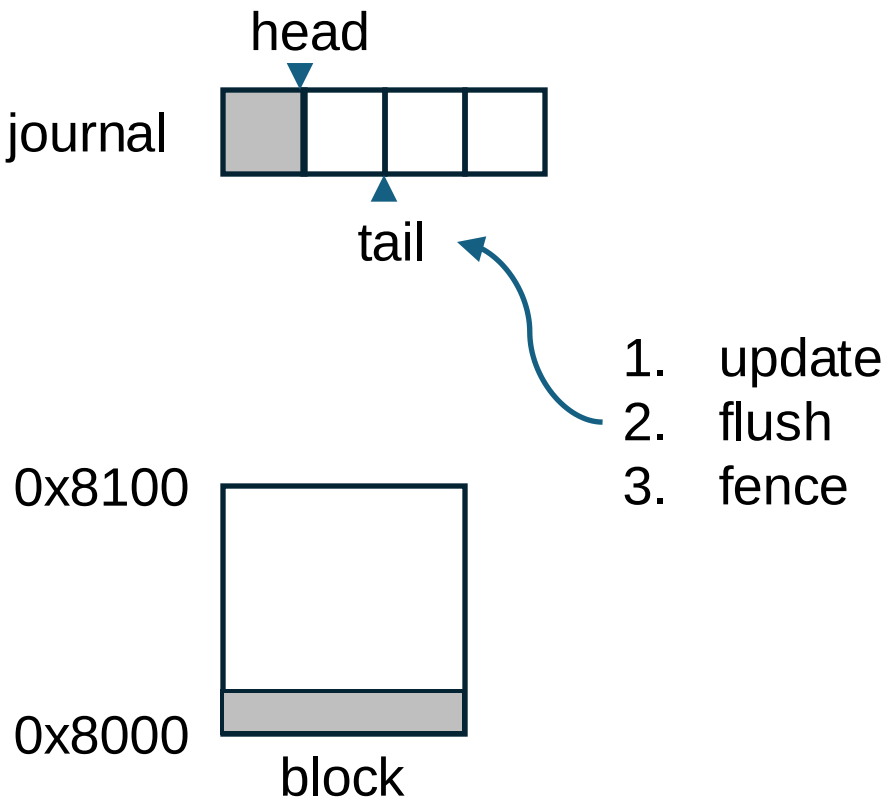


Consistency Invariants

Crash-consistency mechanisms order writes in multiple phases.

Take an undo-journal as an example:

1. Allocate logs

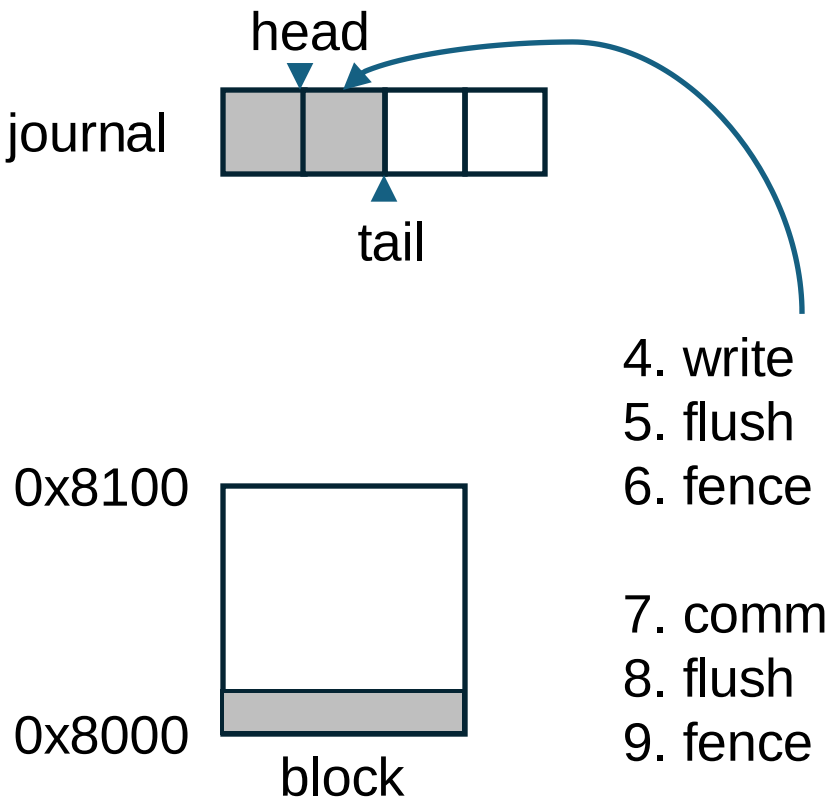


Consistency Invariants

Crash-consistency mechanisms order writes in multiple phases.

Take an undo-journal as an example:

1. Allocate logs
2. Log writes

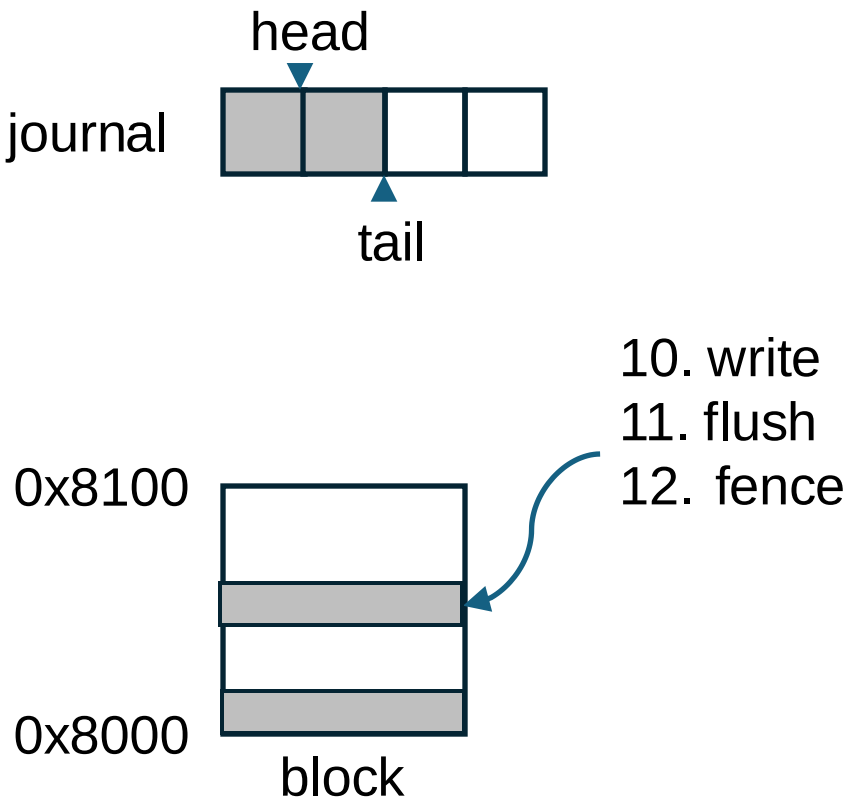


Consistency Invariants

Crash-consistency mechanisms order writes in multiple phases.

Take an undo-journal as an example:

1. Allocate logs
2. Log writes
3. In-place updates

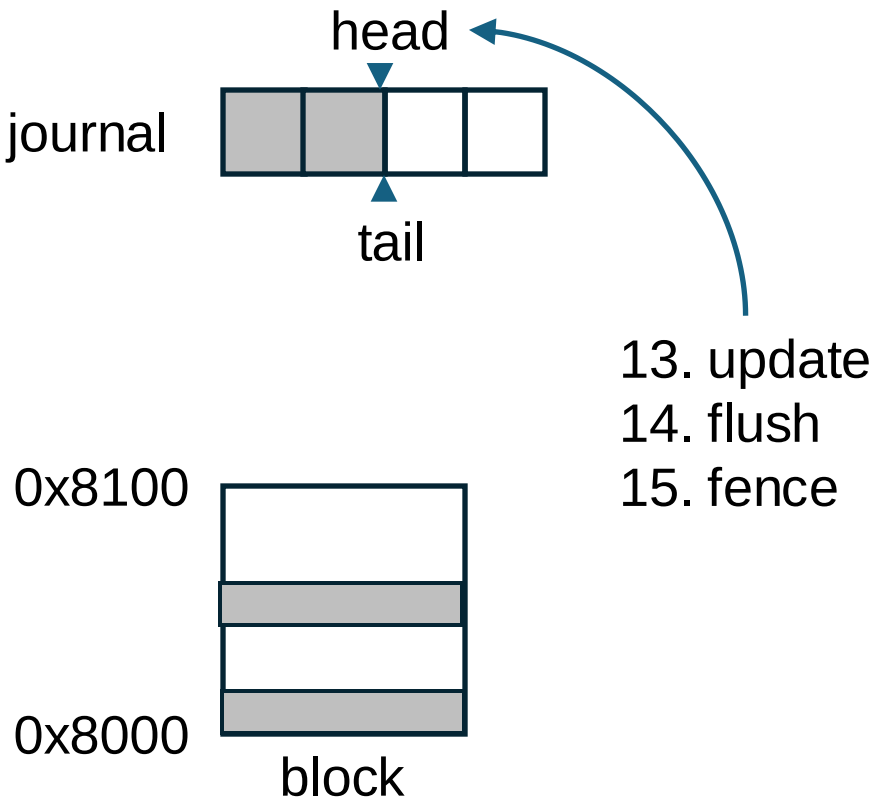


Consistency Invariants

Crash-consistency mechanisms order writes in multiple phases.

Take an undo-journal as an example:

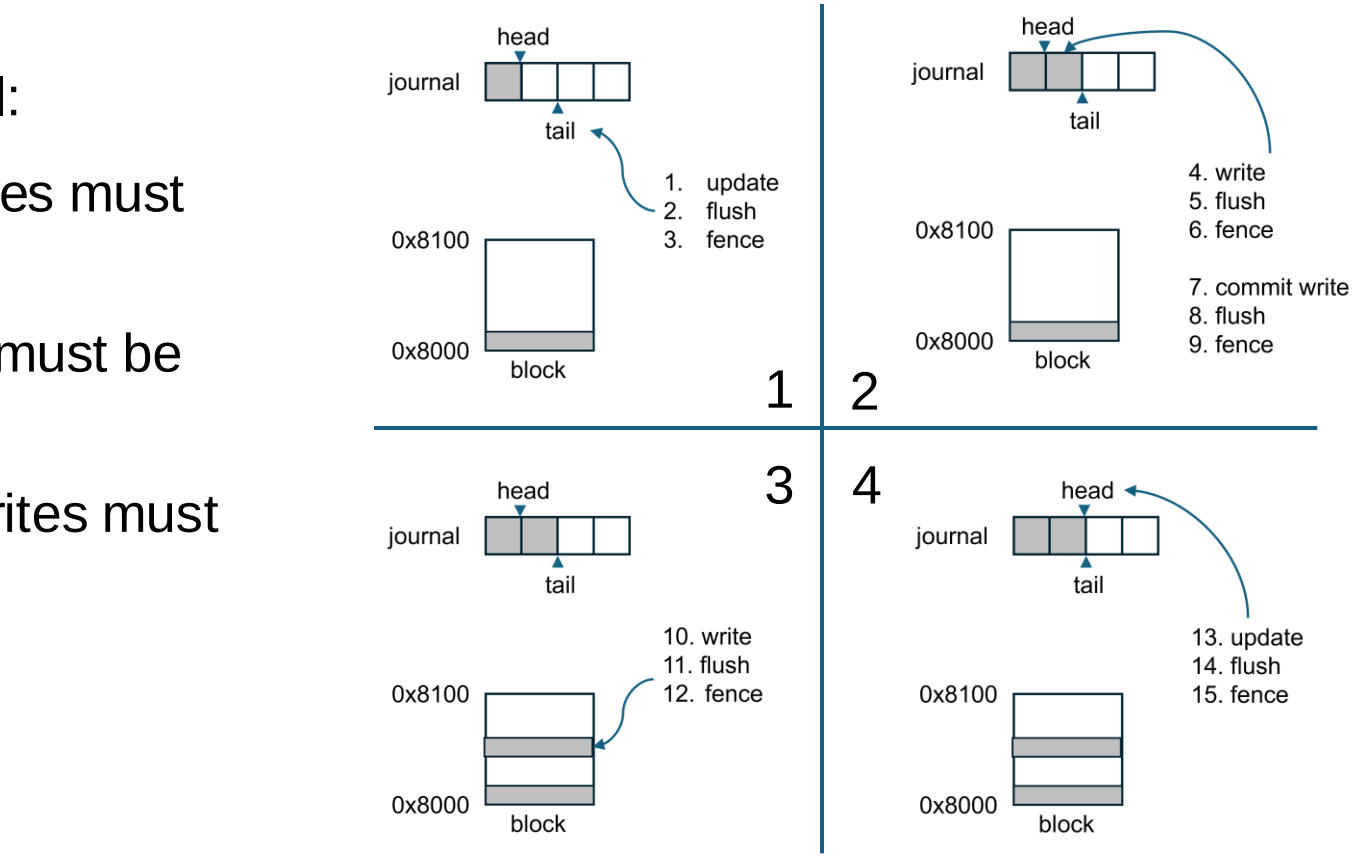
1. Allocate logs
2. Log writes
3. In-place updates
4. Reclaim logs



Consistency Invariants

Invariants for this undo-journal:

- **Ordering invariants**: phases must persist in sequence.
- **Location invariants**: logs must be written to valid regions.
- **Data invariants**: logged writes must match in-place updates.



Checking Consistency Invariants

- Use **LLVM Pass** to instrument FS source code to dynamically trace instructions.
- Tag stores (e.g., stores, CAS, memcpy) in the trace with data types (e.g., structure names, fields).
- Use a lightweight annotation to determine the stores associated with the consistency mechanism.
- Apply invariant checking on these associated stores.
- If invariants pass, mark these stores as protected.

Lightweight Annotation Example

Metadata	Annotation
Journal head	pmfs_journal.base + pmfs_journal.head
Journal tail	pmfs_journal.base + pmfs_journal.tail
Dest addr	pmfs_logentry_t.addr_offset
Dest size	pmfs_logentry_t.size
…	…

An annotation example for PMFS's undo journal.

Lightweight Annotation Example

Metadata	Annotation
Journal head	pmfs_journal.base + pmfs_journal.head
Journal tail	pmfs_journal.base + pmfs_journal.tail
Dest addr	pmfs_logentry_t.addr_offset
Dest size	pmfs_logentry_t.size
…	…

An annotation example for PMFS's undo journal.

- No source code level modifications.
- Only need the struct names and fields.
- Support simple arithmetic operations
- <10 lines per structure

Evaluation

We implemented Silhouette in C++ and Python.

- Tested FSes: PMFS, NOVA, WineFS
- Platform: QEMU VMs ran on a Dell 7820 host machine
- PM device: Emulated via the kernel cmd line
- Workload: ACE-seq1/2/3 and custom test cases
- Comparison:
  - Vinter [ATC ‘22]: reorders stores that are read during recovery
  - Chipmunk [EuroSys ‘23]: kProbe-based trace and comb. reordering
- Open-sourced: https://github.com/iaoing/Silhouette

New Bugs Found by Silhouette

We tested Silhouette on NOVA, PMFS, and WineFS and found 15 new bugs:

- **Segfault** due to incorrect pointer persistence in NOVA
- **Data leak** since truncate is not atomic in NOVA
- **Data loss** due to reusing *inodes* in orphan list in PMFS and WineFS
- …

The full bug reports and reproduction guide can be found in our repository.

Scalability – Bug Finding Time

Silhouette found more bugs in less time than Chipmunk and Vinter.

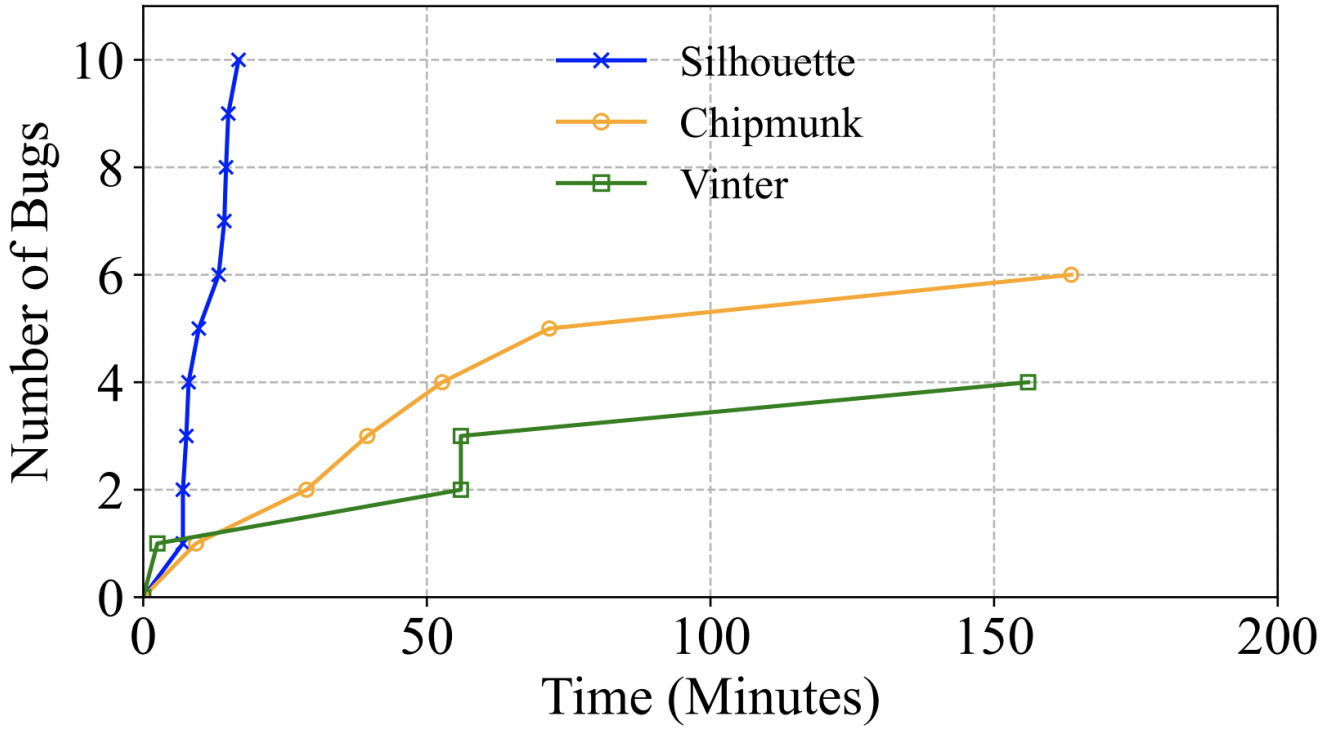


Fig 3. Bugs found in Nova with the ACE Seq3 workload. Time was truncated since no bugs were found beyond this time.

Scalability – Number of Crash Plans

Scheme	NOVA	PMFS	WineFS
Vinter	1.9M	2.3M	3.3M
Chipmunk	3.3M	1.6M	1M
Silhouette	14K	2.3K	1K

Conclusions

- Silhouette is a novel tool designed to detect bugs in PM file systems by leveraging consistency mechanisms.
  - Checks the implementation of the consistency mechanism.
  - Only reorders the stores not associated with the consistency mechanisms.
  - Further reduces the bug search space with the 2CP heuristic
- Silhouette is efficient, scalable, and outperforms existing approaches
  - Has identified 15 new bugs with 10x speedup
- Silhouette is open-sourced and available on GitHub: https://github.com/iaoing/Silhouette

Thank you!

Q&A