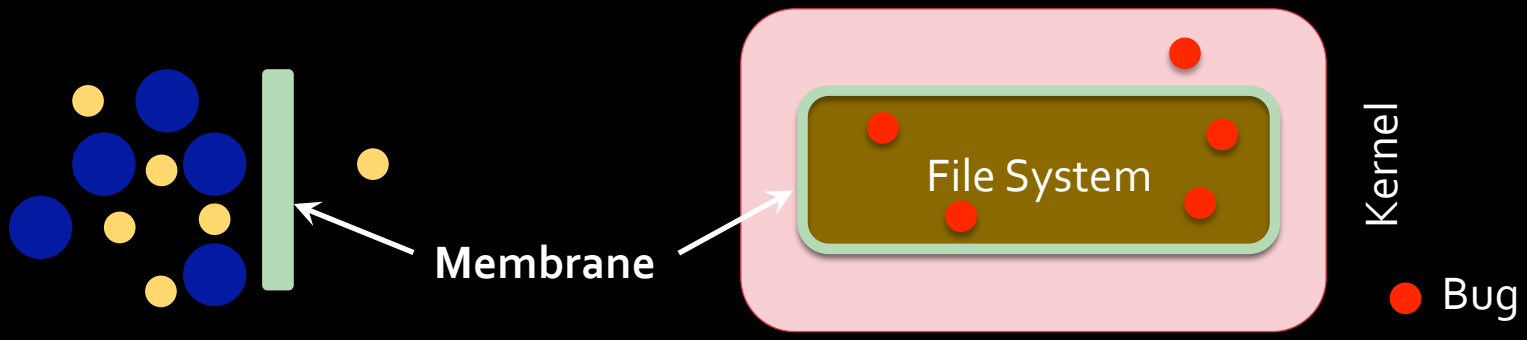
Membrane: Operating System support for Restartable File Systems



Membrane is a layer of material which serves as a selective barrier between two phases and remains impermeable to specific particles, molecules, or substances when exposed to the action of a driving force.

Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift

THE UNIVERSITY of WISCONSIN MADISON

Bugs in File-system Code

- Bugs are common in any large software
  - File systems contain 1,000 – 100,000 loc
- Recent work has uncovered 100s of bugs
  [Engler OSDI '00, Musuvathi OSDI '02, Prabhakaran SOSP '03, Yang OSDI '04, Gunawi FAST '08, Rubio-Gonzales PLDI '09]
  - Error handling code, recovery code, etc.
- File systems are part of core kernel
  - A single bug could make the kernel unusable

Bug Detection in File Systems

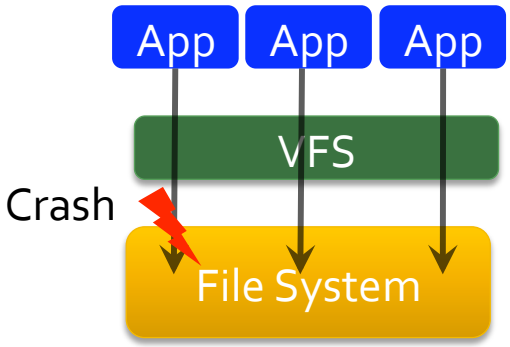
- FS developers are good at detecting bugs
  - "Paranoid" about failures
- Lots of checks all over the file system code!

File System	assert()	BUG()	panic()
xfs	2119	18	43
ubifs	369	36	2
ocfs2	261	531	8
gfs2	156	60	0
afs	106	38	0
ext4	42	182	12
reiserfs	1	109	93
ntfs	0	288	2

Detection is easy but recovery is hard

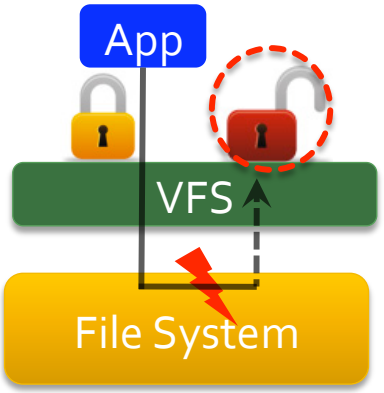
and

Why is Recovery Hard?



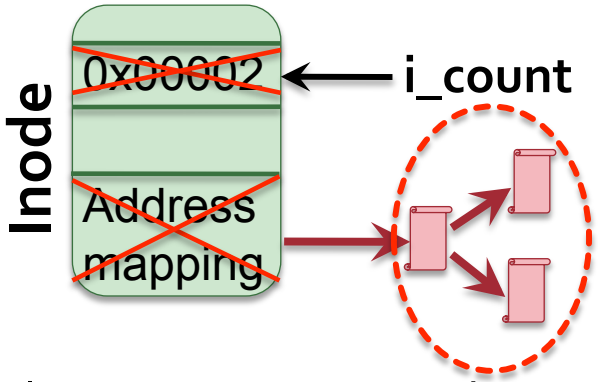
Processes could potentially use corrupt in-memory file-system objects

No fault isolation



Process killed on crash

Inconsistent kernel state



File systems manage their own in-memory objects

Hard to free FS objects

Common solution: crash file system and hope problem goes away after OS reboot

Why not Fix Source Code?

- To develop perfect file systems
  - Tools do not uncover all file system bugs
  - Bugs still are fixed manually
  - Code constantly modified due to new features
- Make file systems handle all error cases
  - Interacts with many external components
    - VFS, memory mgmt., network, page cache, and I/O

Cope with bugs than hope to avoid them

Restartable File Systems

- Membrane: OS framework to support lightweight, stateful recovery from FS crashes
- Upon failure transparently restart FS
  - Restore state and allow pending application requests to be serviced
  - Applications **oblivious** to crashes
- A generic solution to handle all FS crashes
  - Last resort before file systems decide to give up

Results

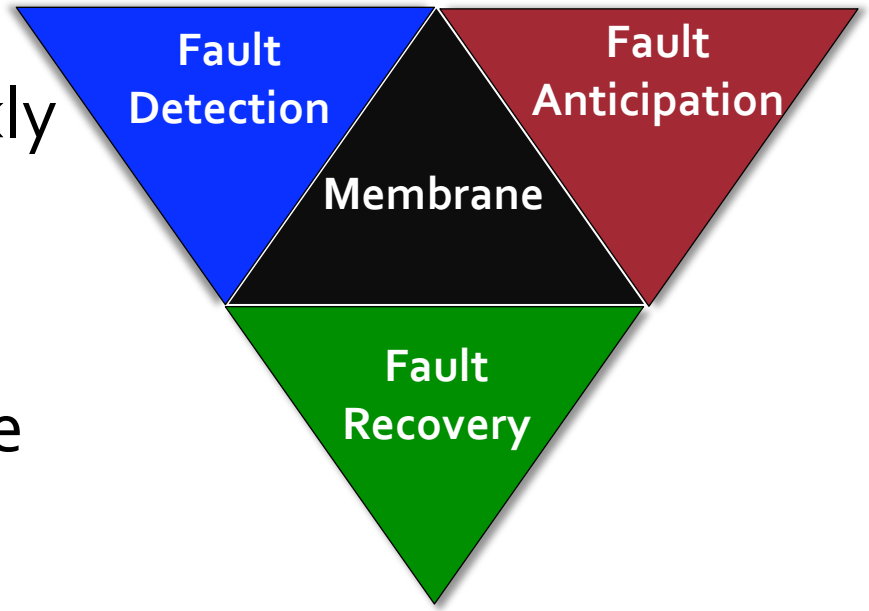
- Implemented Membrane in Linux 2.6.15
  - Evaluated with ext2, VFAT, and ext3
- Evaluation
  - *Transparency*: hide failures (~50 faults) from appl.
  - *Performance*: *< 3%* for micro & macro benchmarks
  - *Recovery time*: *< 30 milliseconds* to restart FS
  - *Generality*: *< 5 lines of code* for each FS

Outline

- Motivation
- **Restartable file systems**
- Evaluation
- Conclusions

Components of Membrane

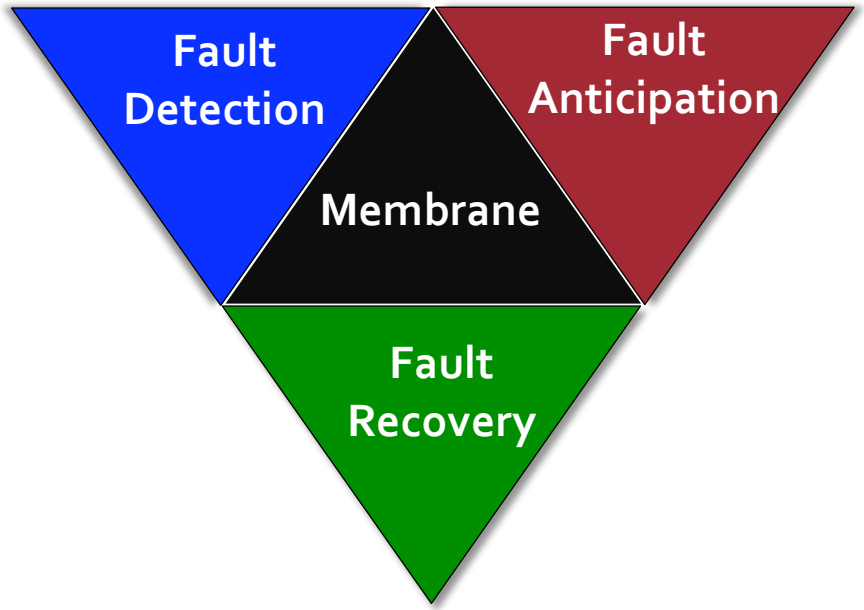
- Fault Detection
  - Helps detect faults quickly
- Fault Anticipation
  - Records file-system state
- Fault Recovery
  - Executes recovery protocol to cleanup and restart the failed file system



Fault Detection

- Correct recovery requires early detection
  - Membrane best handles "*fail-stop*" failures
- Both hardware and software-based detection
  - *H/W*: null pointer, general protection error, …
  - *S/W*: asserts(), BUG(), BUG_ON(), panic()
- Assume transient faults during recovery
  - Non-transient faults: return error to that process

Components of Membrane



Fault Anticipation

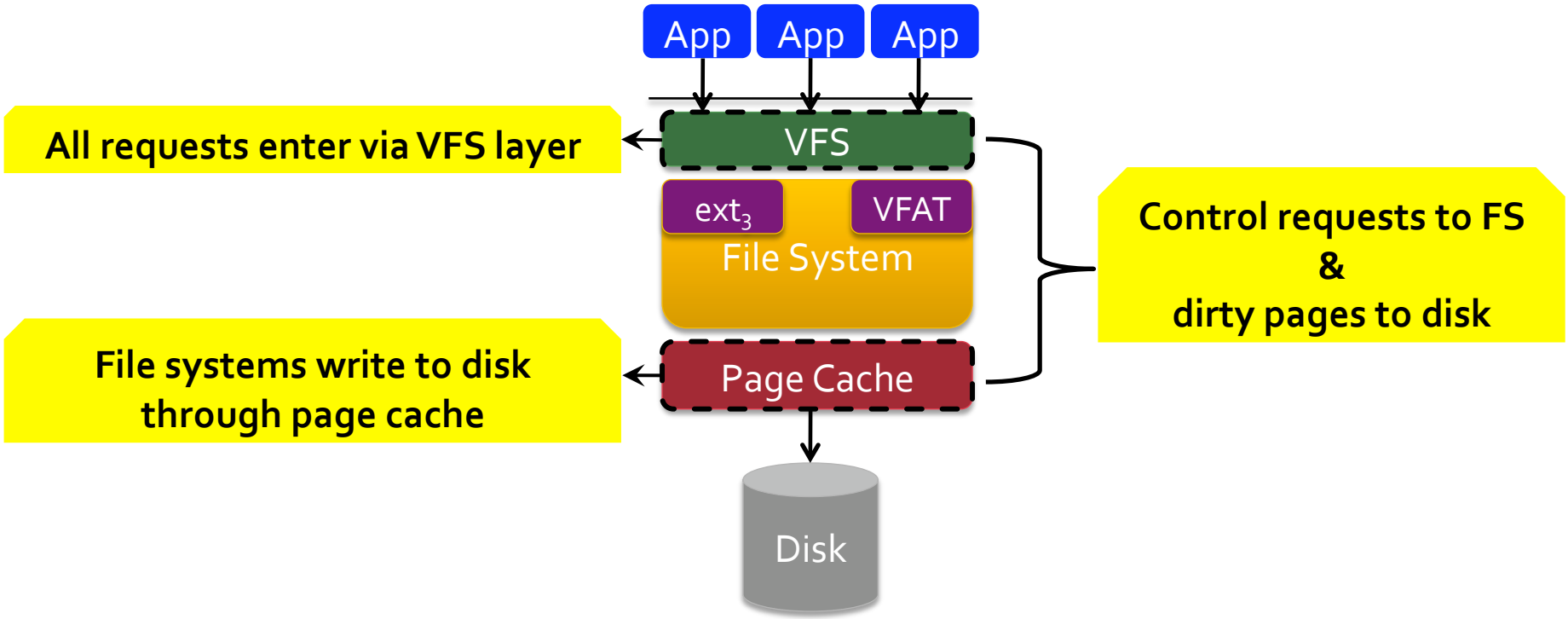
Additional work done in anticipation of a failure

- **Issue**: where to restart the file system from?
  - File systems constantly updated by applications
- Possible solutions:
  - Make each operation atomic
  - Leverage in-built crash consistency mechanism
- Not all FS have crash consistency mechanism

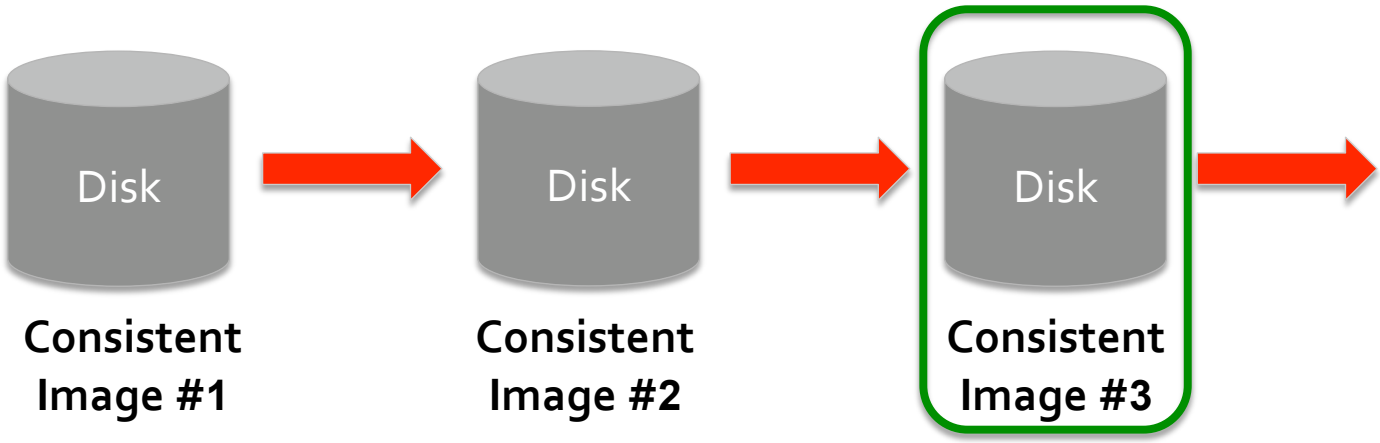
Generic mechanism to checkpoint FS state

Checkpoint File-system State

Checkpoint: *consistent state of the file system that can be safely rolled back to in the event of a crash*

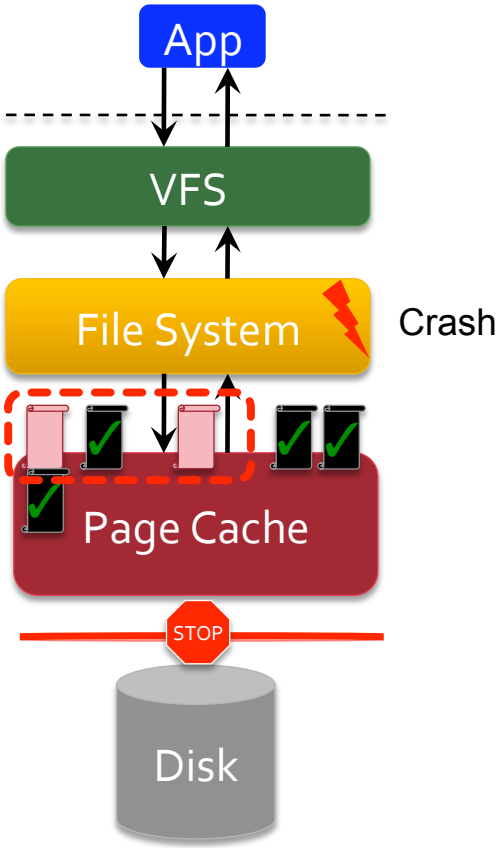


Generic COW based Checkpoint



On crash roll back to last consistent Image

State after checkpoint?



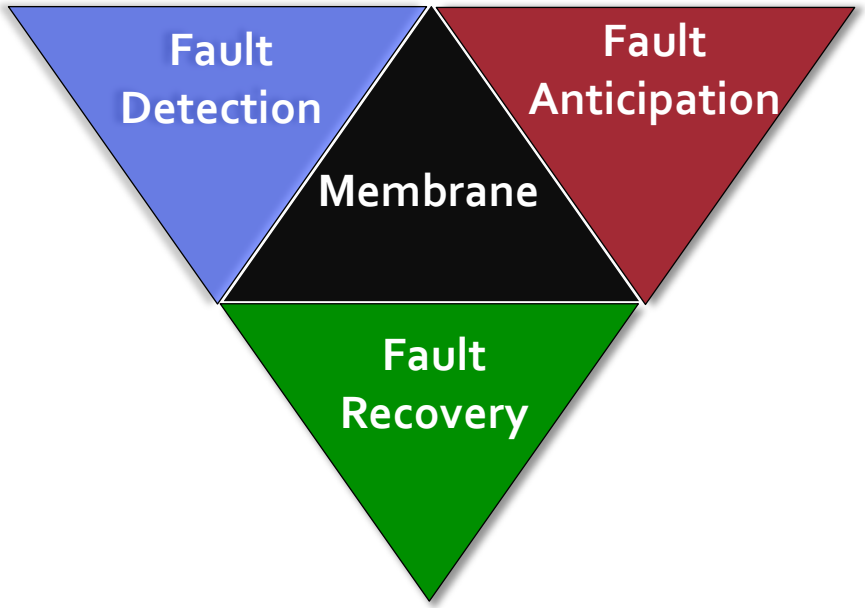
- ***On crash:*** flush dirty pages of last checkpoint
- Throw away the in-memory state
- Remount from the last checkpoint
  - Consistent file-system image on disk
- ***Issue:*** state after checkpoint would be lost
  - Operations completed after checkpoint returned back to applications

Need to recreate state after checkpoint

Operation-level Logging

- Log operations along with their return value
  - Replay completed operations after checkpoint
- Operations are logged at the VFS layer
  - File-system independent approach
- Logs are maintained **in-memory** and **not** on **disk**
- How long should we keep the log records?
  - Log thrown away at checkpoint completion

Components of Membrane



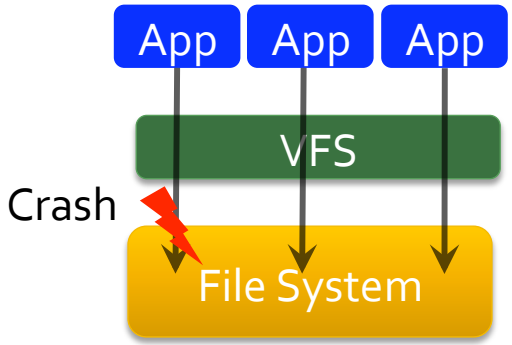
Fault Recovery

Important steps in recovery:

1. Cleanup state of partially-completed operations
2. Cleanup in-memory state of file system
3. Remount file system from last checkpoint
4. Replay completed operations after checkpoint
5. Re-execute partially complete operations

Partially completed Operations

Multiple threads inside file system

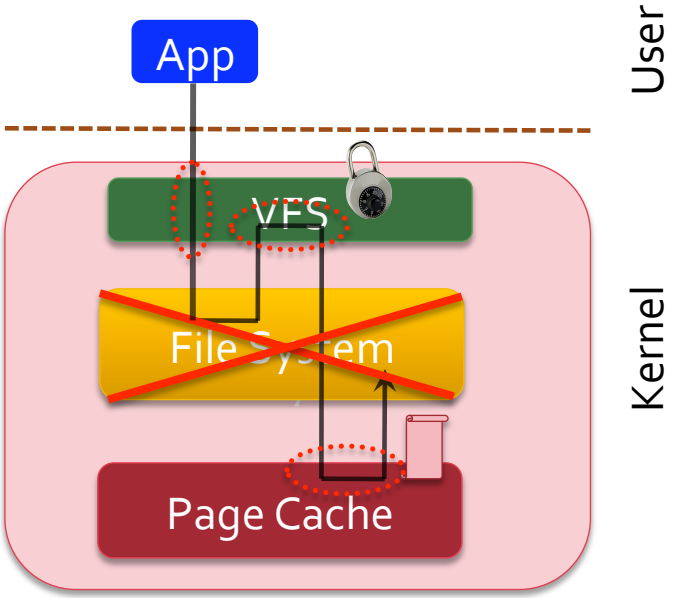


FS code should not be trusted after crash

Application threads killed?

- application state will be lost

Intertwined execution



Processes cannot **be killed** after crash

Clean way to undo incomplete operations

A Skip/Trust Unwind Protocol

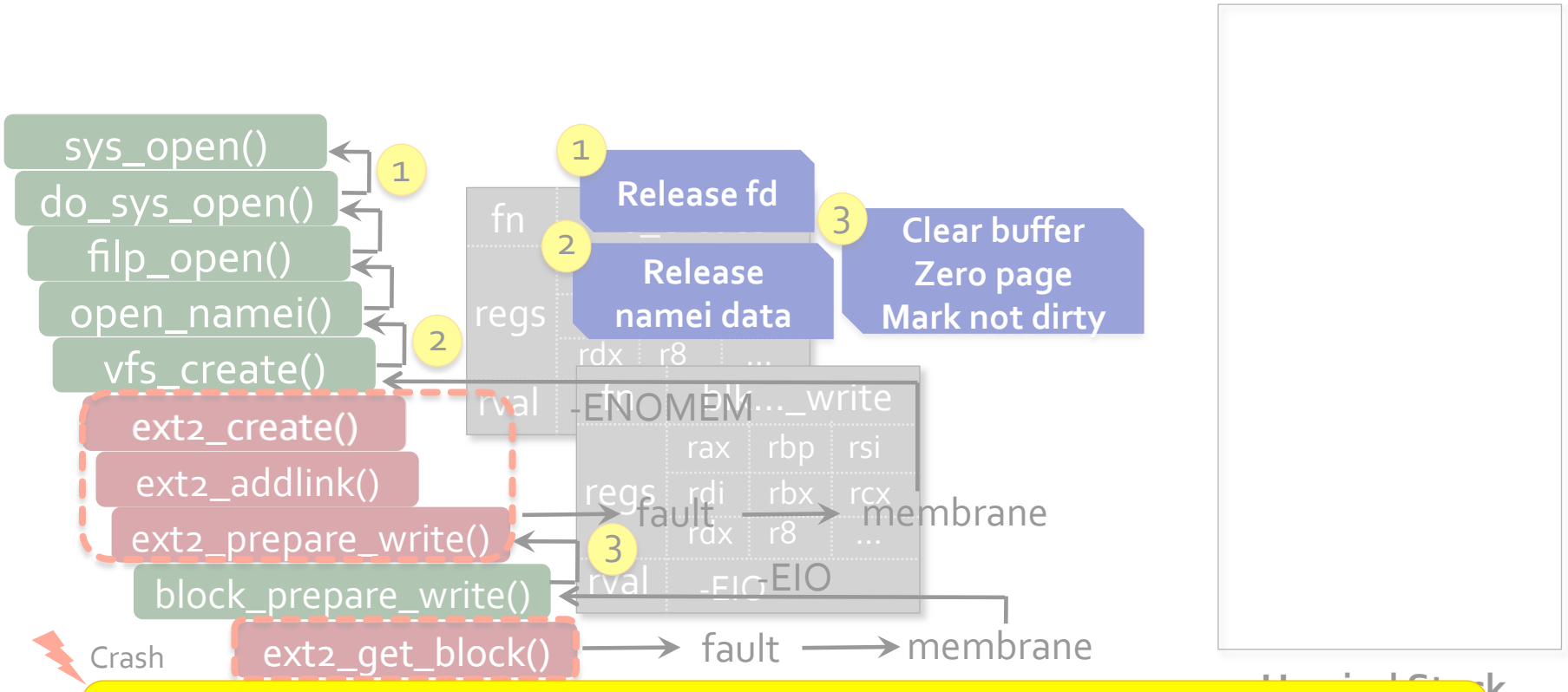
Skip: file-system code
Trust: kernel code (VFS, memory mgmt., …)

- Cleanup state on error from file systems

- How to prevent execution of FS code?
  - ***Control capture mechanism***: marks file-system code pages as non-executable
  - ***Unwind Stack***: stores return address (of last kernel function) along with expected error value

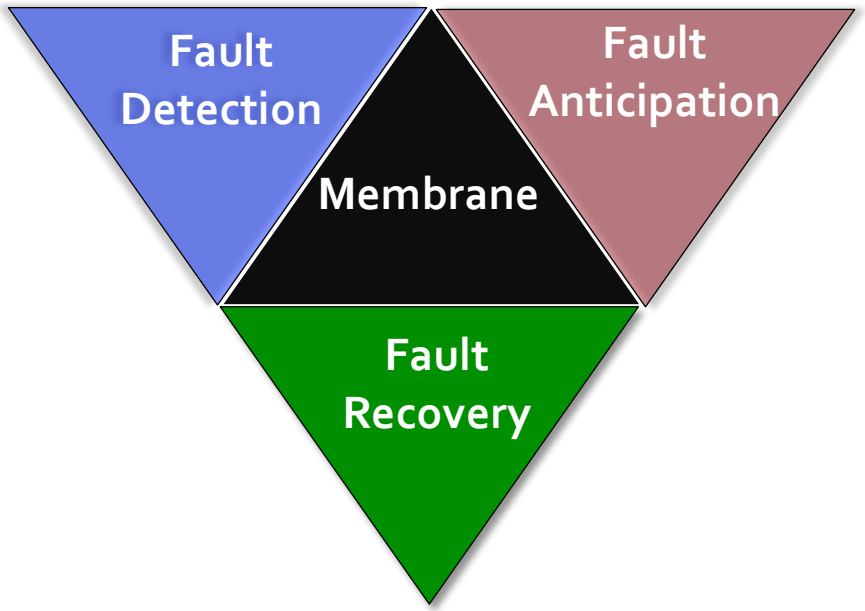
Skip/Trust Unwind Protocol in Action

- E.g., create code path in ext2

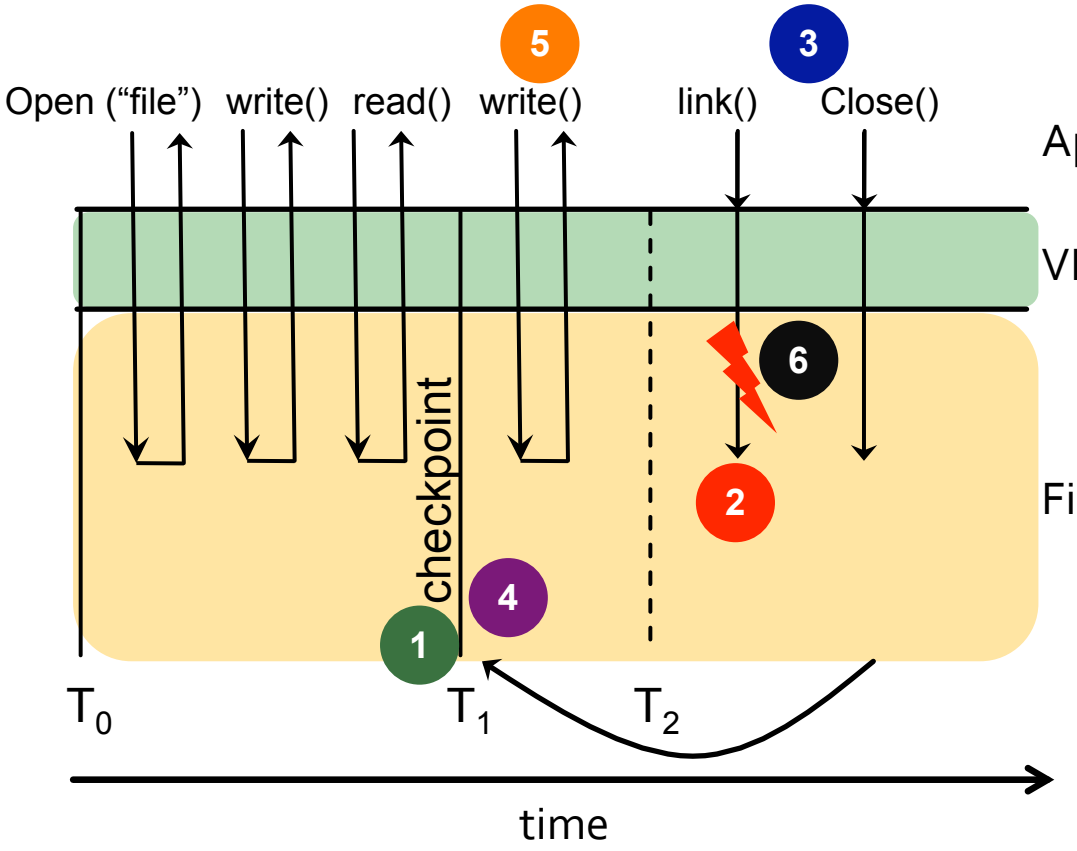


Kernel is restored to a consistent state

Components of Membrane



Putting All Pieces Together



Legend: Completed · In-progress · Crash

1. Periodically create checkpoints
2. File System Crash
3. Unwind in-flight processes
4. Move to recent checkpoint
5. Replay completed operations
6. Re-execute unwound process

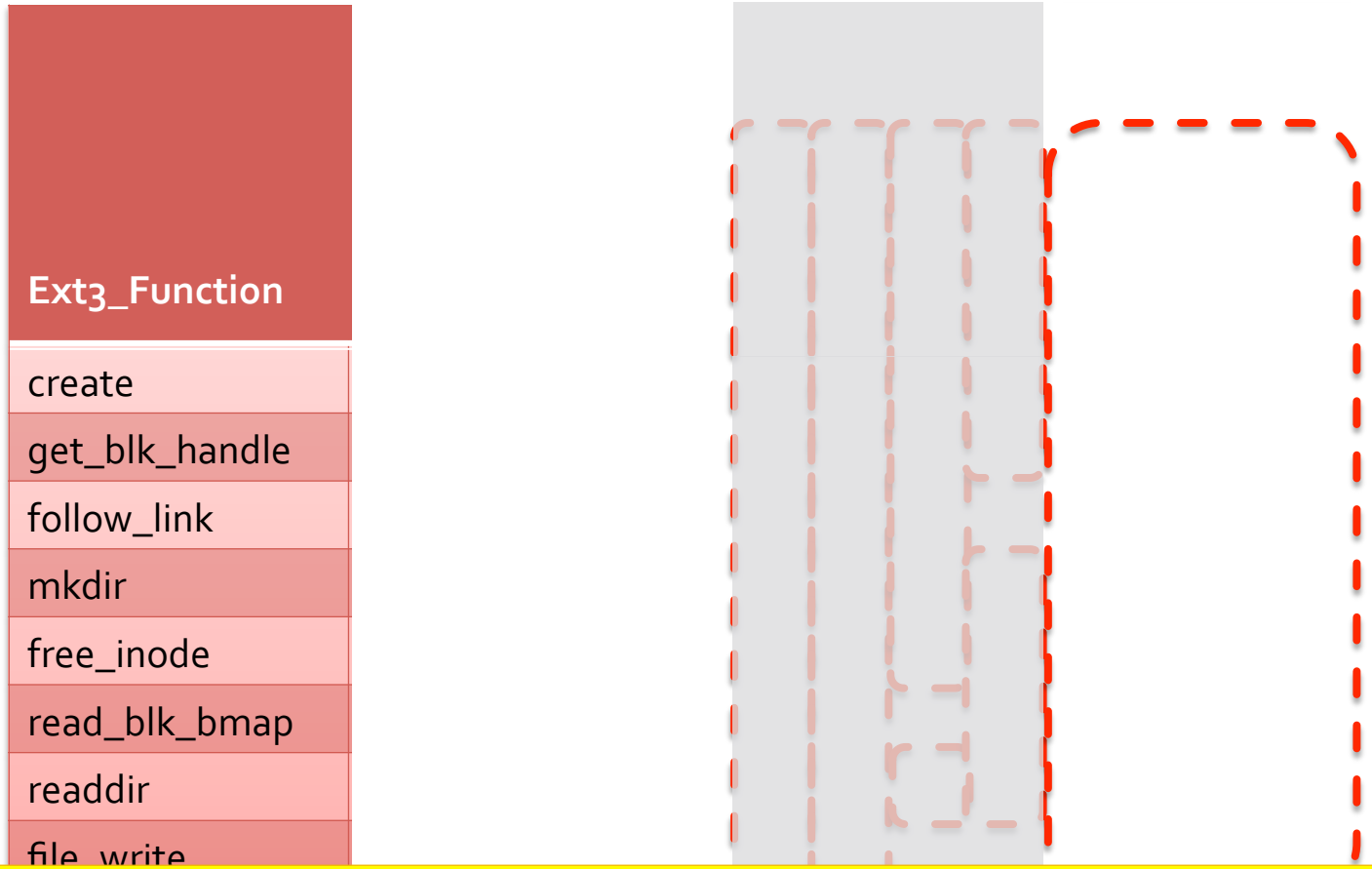
Outline

- Motivation
- Restartable file systems
- **Evaluation**
- Conclusions

Evaluation

- Questions that we want to answer:
  - Can membrane hide failures from applications?
  - What is the overhead during user workloads?
  - Portability of existing FS to work with Membrane?
  - How much time does it take to recover the FS?
- Setup:
  - 2.2 GHz Opteron processor & 2 GB RAM
  - Two 80 GB western digital disk
  - Linux 2.6.15 64bit kernel, 5.5K LOC were added
  - *File systems:* ext2, VFAT, ext3

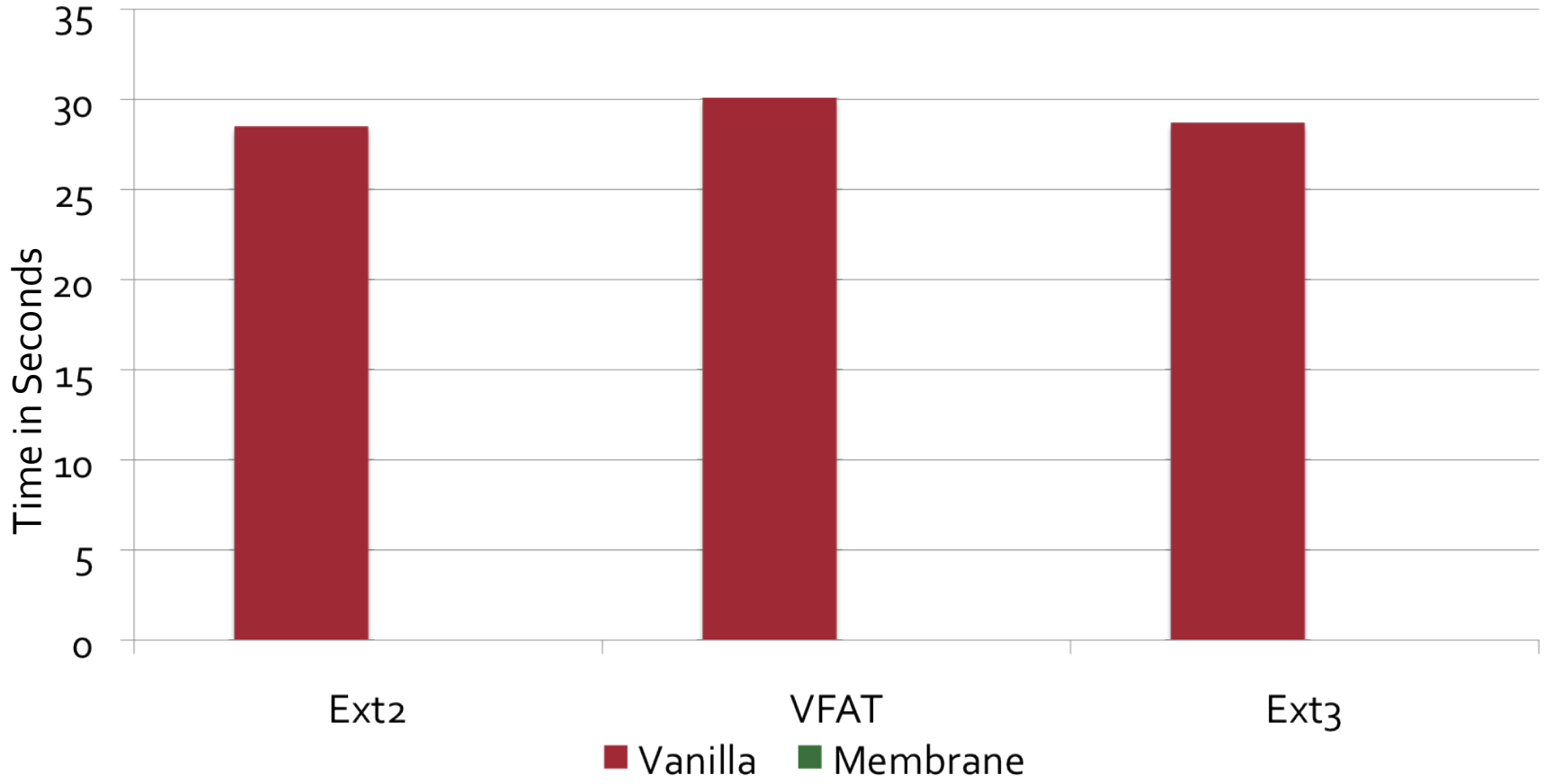
How Transparent are Failures?



Membrane successfully hides faults

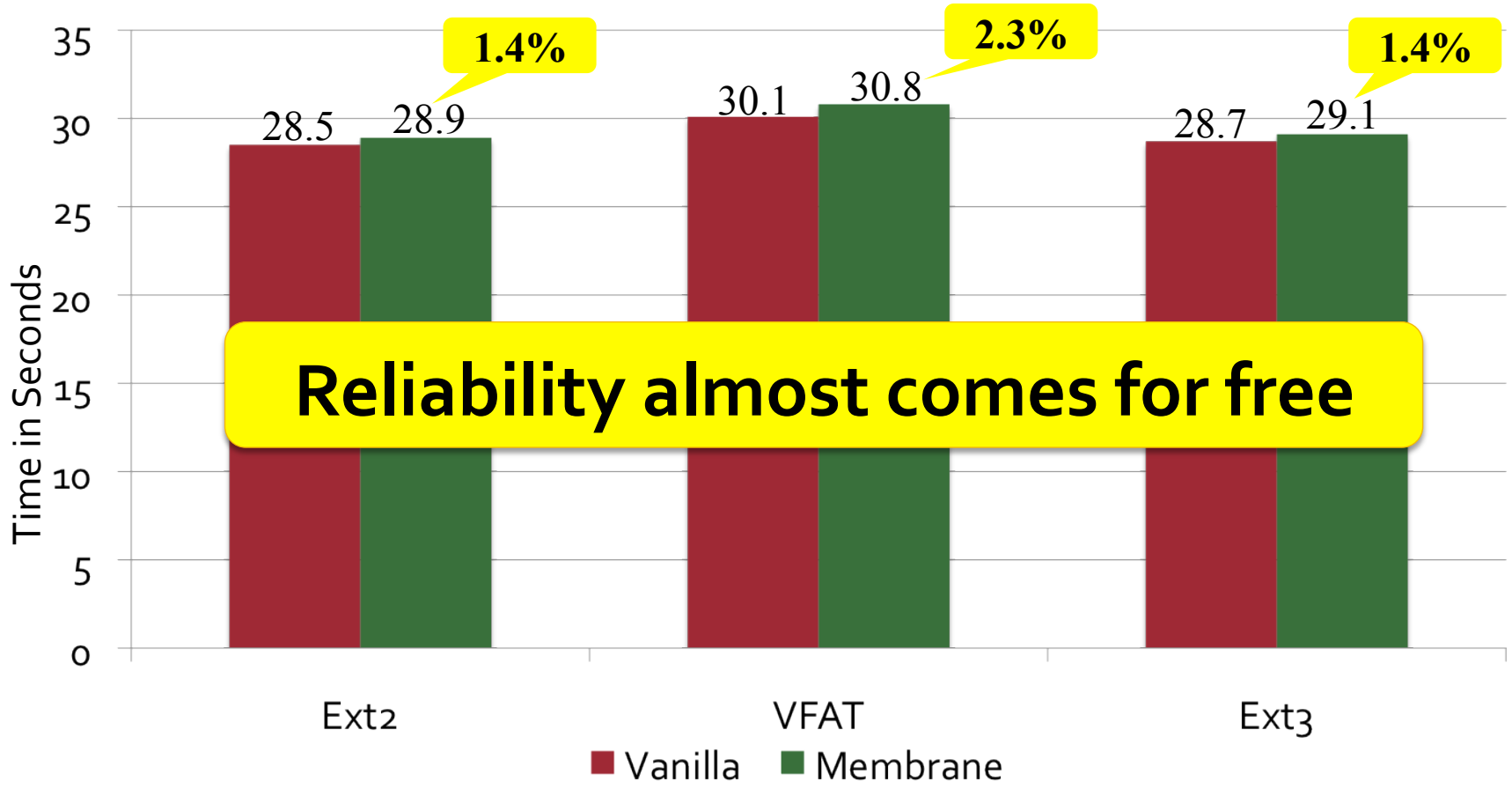
Overheads during User Workloads?

Workload: Copy, untar, make of OpenSSH 4.51

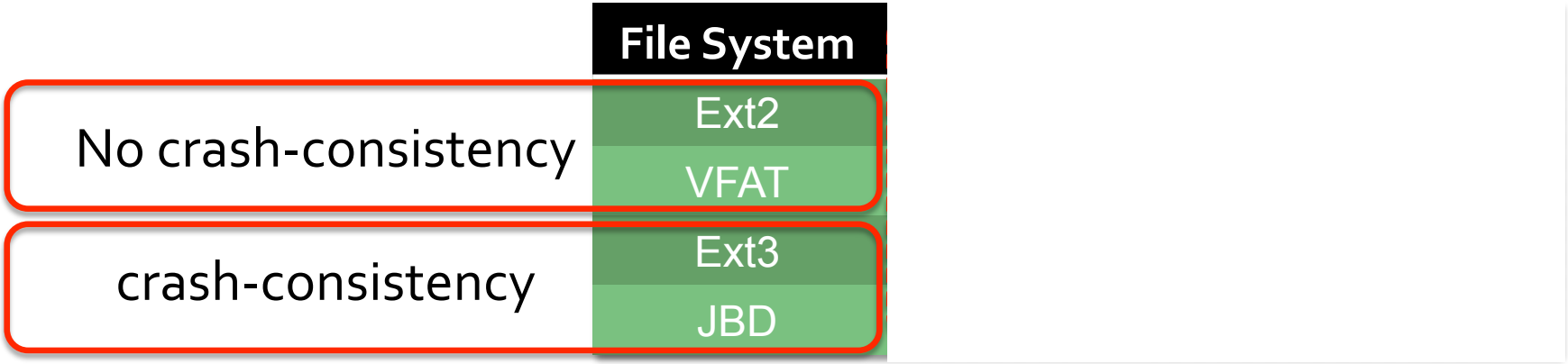


Overheads during User Workloads?

Workload: Copy, untar, make of OpenSSH 4.51



Generality of Membrane?



Individual file system changes

Existing code remains unchanged

Additions: track allocations and write super block

Minimal changes to port existing FS to Membrane

Outline

- Motivation
- Restartable file systems
- Evaluation
- **Conclusions**

Conclusions

- Failures are inevitable in file systems
  - Learn to cope and not hope to avoid them
- ***Membrane***: Generic recovery mechanism
  - Users: Build trust in new file systems (e.g., btrfs)
  - Developers: Quick-fix bug patching
- Encourage more integrity checks in FS code
  - Detection is easy but recovery is hard

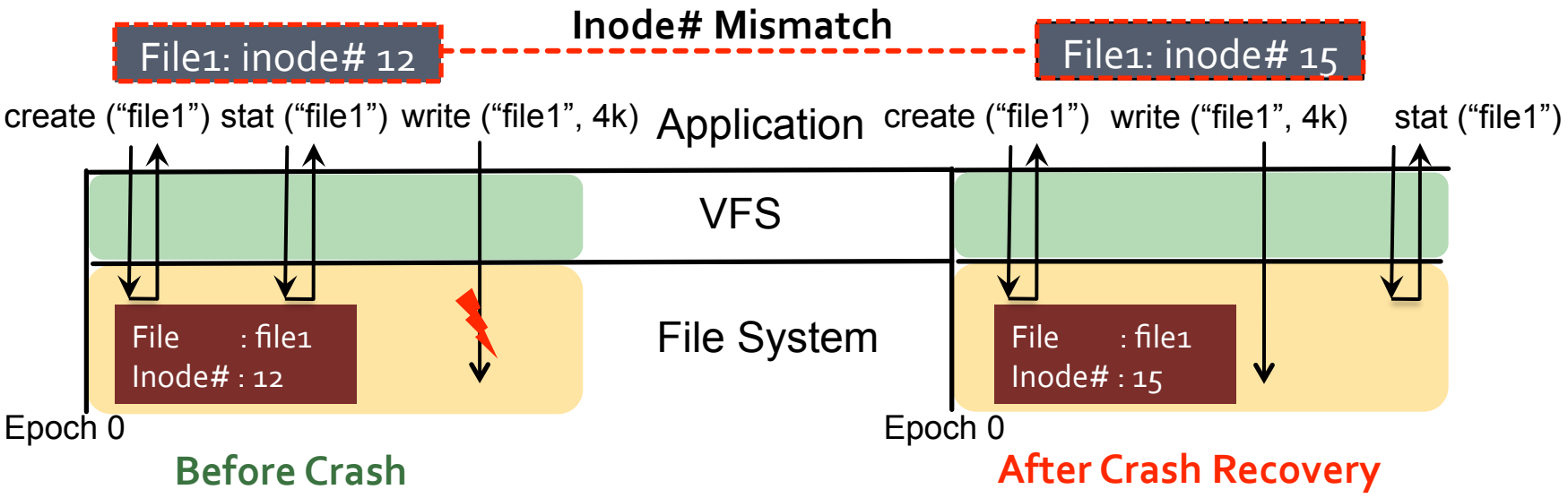
Thank You!

Questions?

Advanced Systems Lab (ADSL)
University of Wisconsin-Madison
http://www.cs.wisc.edu/adsl

Are Failures Always Transparent?

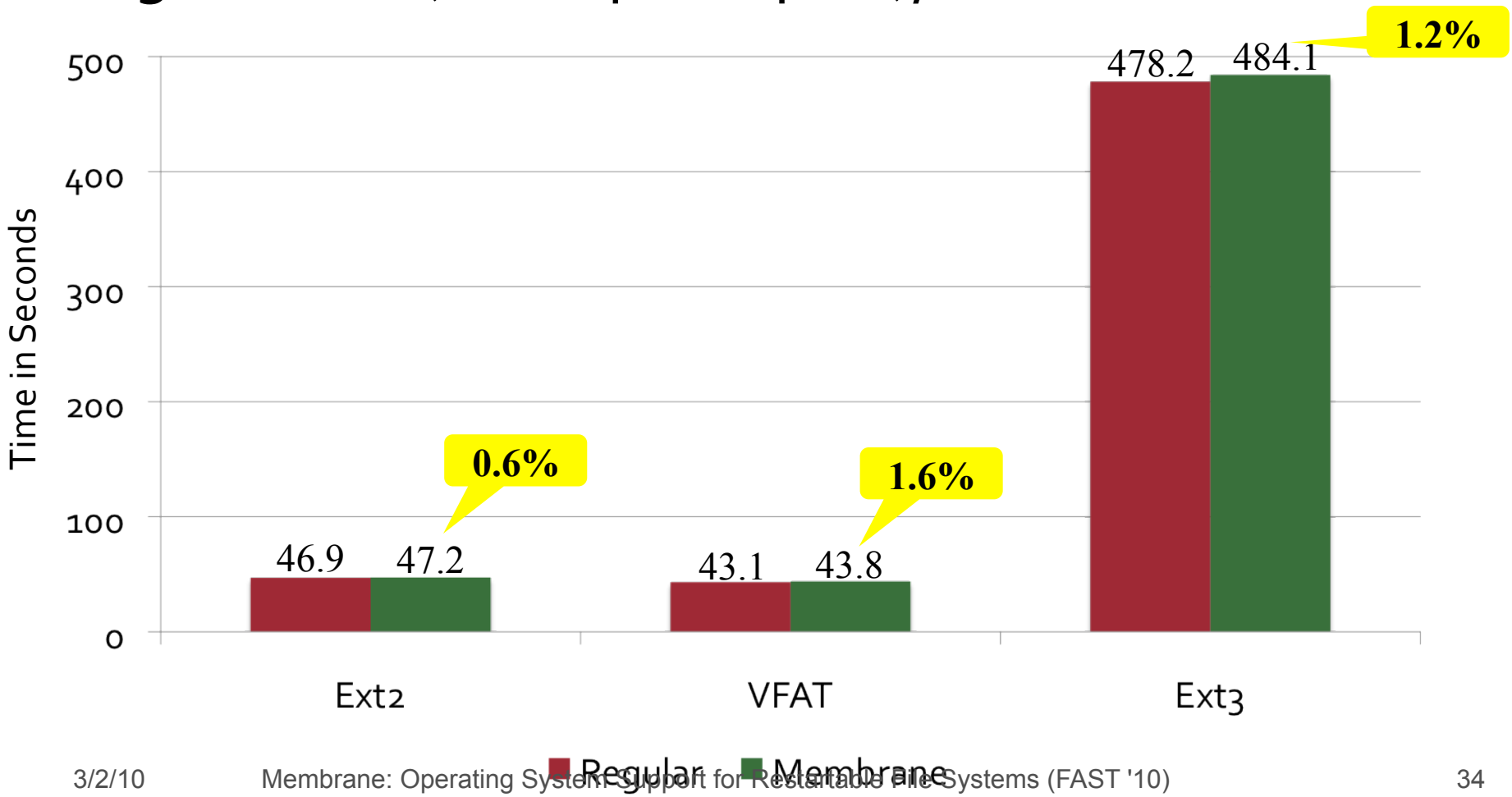
- Files may be recreated during recovery
  - Inode numbers could change after restart



Solution: make create() part of a checkpoint

Postmark Benchmark

- 3000 files (sizes 4K to 4MB), 60K transactions



Recovery Time

- Recovery time is a function of:
  - Dirty blocks, open sessions, and log records
- We varied each of them individually

Data (Mb)	Recovery Time (ms)
10	12.9
20	13.2
40	16.1

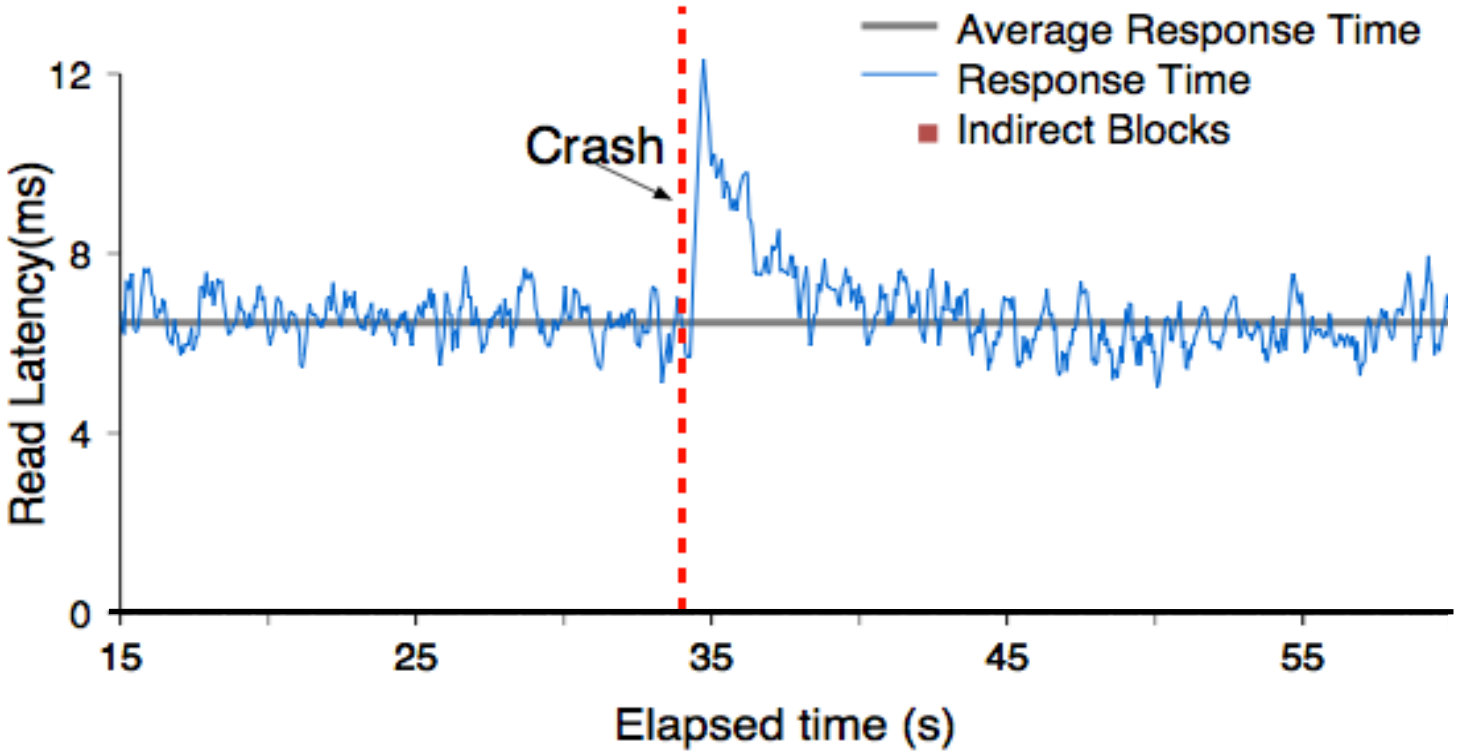
Open Sessions	Recovery Time (ms)
200	11.4
400	14.6
800	22.0

Log Records	Recovery Time (ms)
1K	15.3
10K	16.8
100K	25.2

Recovery time is in the order of a few milliseconds

Recovery Time (Cont.)

Restart ext2 during random-read benchmark



Generality and Code Complexity

Individual file system changes

File System	Added	Modified
Ext2	4	0
VFAT	5	0
Ext3	1	0
JBD	4	0

Kernel changes

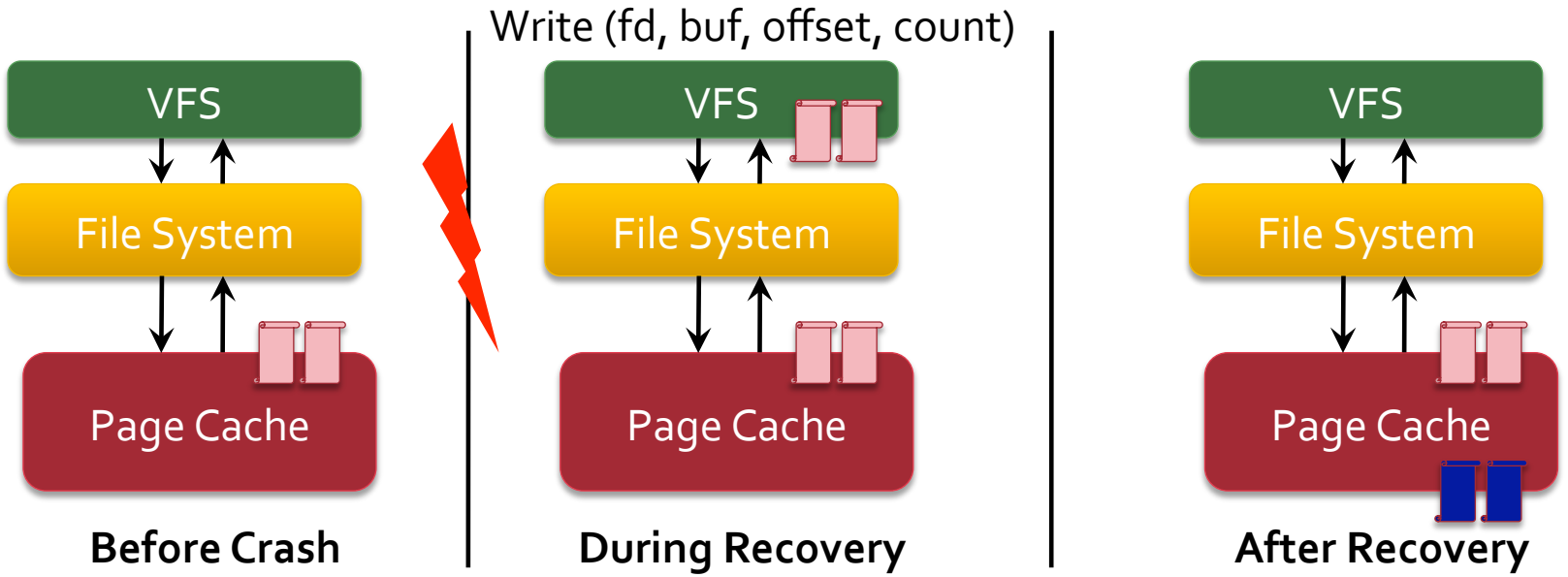
Components	No Checkpoint		With Checkpoint	
	Added	Modified	Added	Modified
FS	1929	30	2979	64
MM	779	5	867	15
Arch	0	0	733	4
Headers	522	6	552	6
Module	238	0	238	0
Total	**3468**	**41**	**5369**	**89**

Interaction with Modern FSes

- Have built-in crash consistency mechanism
  - Journaling or Snapshotting
- Seamlessly integrate with these mechanism
  - Need FSes to indicate beginning and end of an transaction
  - Works for data and ordered journaling mode
  - Need to combine writeback mode with COW

Page Stealing Mechanism

- **Goal:** Reduce the overhead of logging writes
  - Soln: Grab data from page cache during recovery



Handling Non-Determinism

- During log replay could data be written in different order?
  - Log entries need not represent actual order
- Not a problem for meta-data updates
  - Only one of them succeed and is recorded in log
- Deterministic data-block updates with page stealing mechanism
  - Latest version of the page is used during replay

Possible Solutions

1. Code to recover from all failures
   - Not feasible in reality
2. Restart on failure
   - Previous work have taken this approach

FS need: stateful & lightweight recovery