Zettabyte Reliability with Flexible End-to-end Data Integrity

Yupu Zhang, Daniel Myers,
Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau

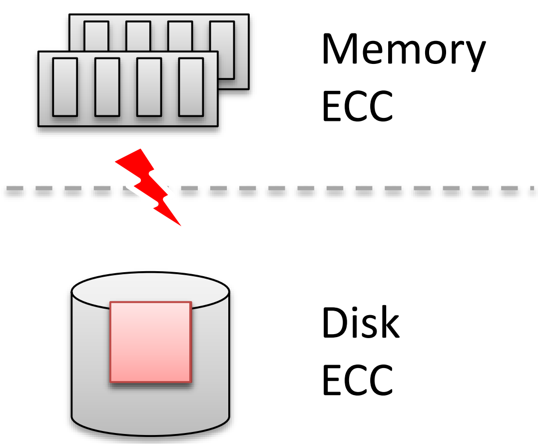
University of Wisconsin - Madison

Data Corruption

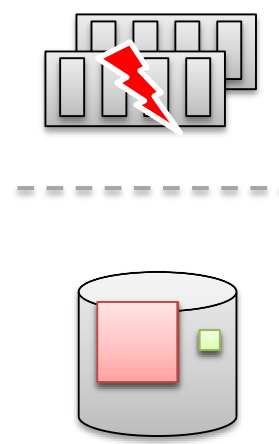
- Imperfect hardware
  - Disk, memory, controllers [Bairavasundaram07, Schroeder09, Anderson03]
- Buggy software
  - Kernel, file system, firmware [Engler01, Yang04, Weinberg04]
- Techniques to maintain data integrity
  - Detection: Checksums [Stein01, Bartlett04]
  - Recovery: RAID [Patterson88, Corbett04]

In Reality

- Corruption still occurs and goes undetected
  - Existing checks are usually isolated
  - High-level checks are limited (e.g, ZFS)
- Comprehensive protection is needed



Isolated Protection



Limited Protection

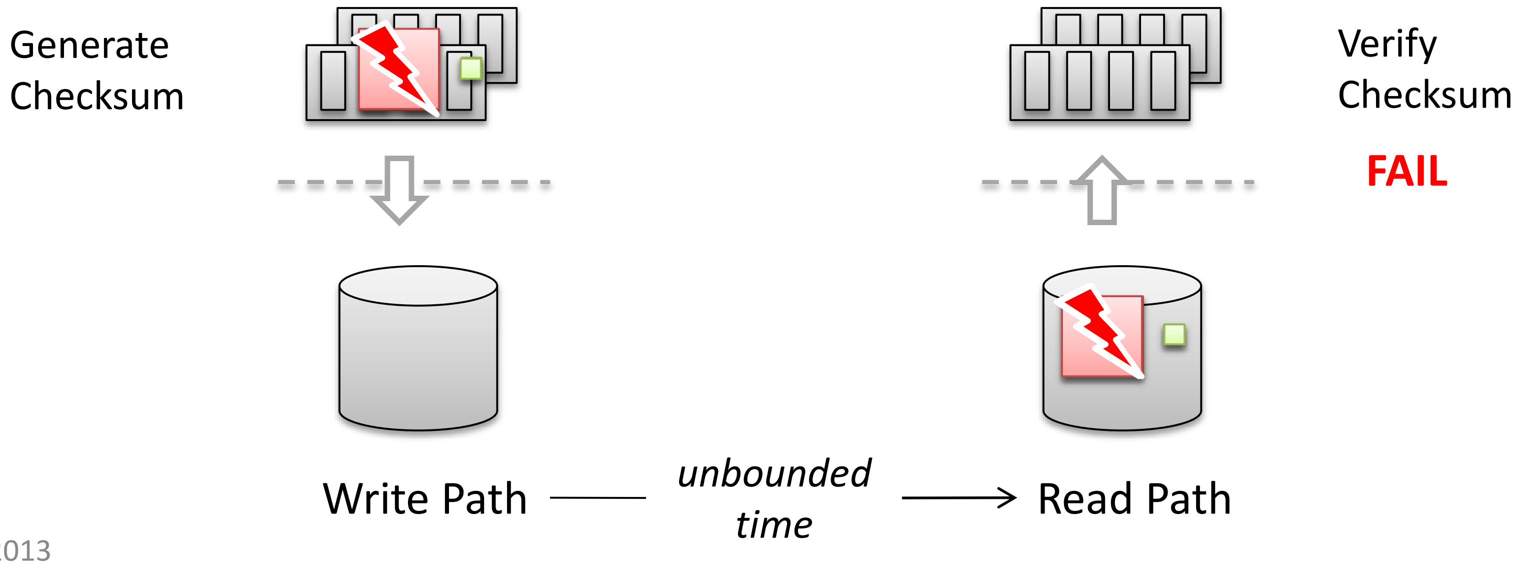
Previous State of the Art

- End-to-end Data Integrity
  - Checksum for each data block is generated and verified by application
  - Same checksum protects data throughout entire stack
  - A strong checksum is usually preferred

Write Path   Read Path

Two Drawbacks

- Performance
  - Repeatedly accessing data from in-memory cache
  - Strong checksum means high overhead
- Timeliness
  - It is too late to recover from the corruption that occurs before a block is written to disk



Flexible End-to-end Data Integrity

- Goal: balance performance and reliability
  - Change checksum across components or over time
- Performance
  - Fast but weaker checksum for in-memory data
  - Slow but stronger checksum for on-disk data
- Timeliness
  - Each component is aware of the checksum
  - Verification can catch corruption in time

Our contribution

- Modeling
  - Framework to reason about reliability of storage systems
  - Reliability goal: Zettabyte Reliability
    - at most one undetected corruption per Zettabyte read
- Design and implementation
  - Zettabyte-Reliable ZFS (Z^2FS)
    - ZFS with flexible end-to-end data integrity

Results

- Reliability
  - Z^2FS is able to provide Zettabyte reliability
    - ZFS: ~ Pettabyte at best
  - Z^2FS detects and recovers from corruption in time

- Performance
  - Comparable to ZFS (less than 10% overhead)
  - Overall faster than the straightforward end-to-end approach (up to 17% in some cases)

Outline

- Introduction
- **Analytical Framework**
  - **Overview**
  - Example
- From ZFS to Z^2FS
- Implementation
- Evaluation
- Conclusion

Overview of the Framework

- Goal
  - Analytically evaluate and compare reliability of storage systems
- Silent Data Corruption
  - Corruption that is undetected by existing checks
- Metric: $P_{undetected}$
  - Probability of undetected data corruption when reading a data block from system (per I/O)
  - Reliability Score = $-log_{10}(P_{undetected})$

Models for the Framework

- Hard disk
  - Undetected Bit Error Rate ($UBER$)
    - Stable, not related to time
  - Disk Reliability Index = $-log_{10}(UBER)$
- Memory
  - Failure in Time (FIT) / Mbit ($Failure\ Rate$)
    - Longer residency time, more likely corrupted
  - Memory Reliability Index = $-log_{10}\ (Failure\ Rate)$
- Checksum
  - Probability of undetected corruption on a device with a checksum

Calculating $P_{undetected}$

- Focus on lifetime of block
  - From it being generated to it being read
  - Across multiple components
  - Find all silent corruption scenarios
- $P_{undetected}$ is sum of probabilities of each silent corruption scenario during lifetime of block in storage system

Reliability Goal

- Ideally, $P_{undetected}$ should be 0
  - It's impossible
- Goal: Zettabyte Reliability
  - At most one SDC when reading one Zettabyte data from a storage system
  - $P_{undetected} = P_{goal} = 3.46 \times 10^{-18}$
    - Assuming a data block is 4KB
  - Reliability Score is 17.5
    - 100MB/s => 2.8×10^{-6} SDC/year
    - $\sim$ 17 nines

Outline

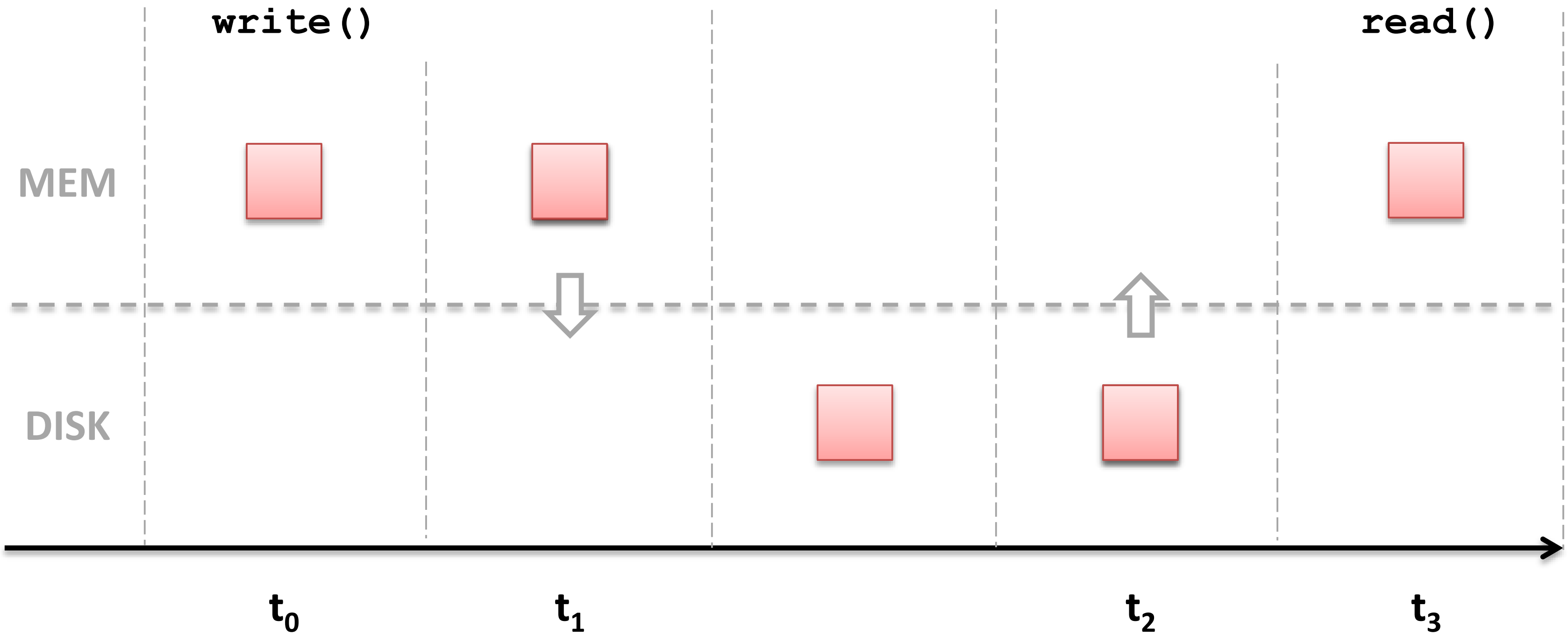
- Introduction
- **Analytical Framework**
  - Overview
  - **Example**
- From ZFS to Z^2FS
- Implementation
- Evaluation
- Conclusion

Sample Systems

- Disk Reliability Index = 10~20
  - Regular disk: 12
- Memory Reliability Index = 13.4~18.8
  - non-ECC memory: 14.2
  - ECC memory: 18.8

Name	Reliability Index		Description
	Memory	Disk	
Worst	13.4	10	Worst memory & worst disk
Consumer	14.2	12	Non-ECC memory & regular disk
Server	18.8	12	ECC memory & regular disk
Best	18.8	20	ECC memory & best disk

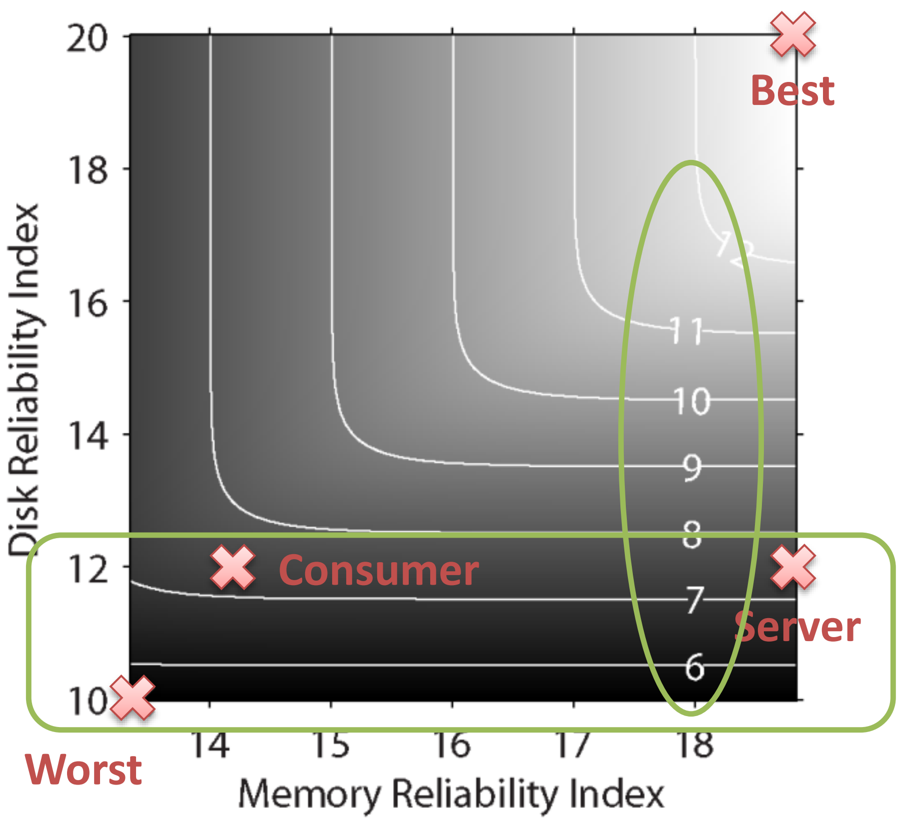
Example



- Assuming there is only one corruption in each scenario
  - Each time period is a scenario
  - $P_{undetected}$ = sum of probabilities of each time period
- Assuming $t_1 - t_0 = 30$ seconds (flushing interval)
- Residency Time: $t_{resident} = t_3 - t_2$

Example (cont.)

- Reliability Score ($t_{resident} = 1$)

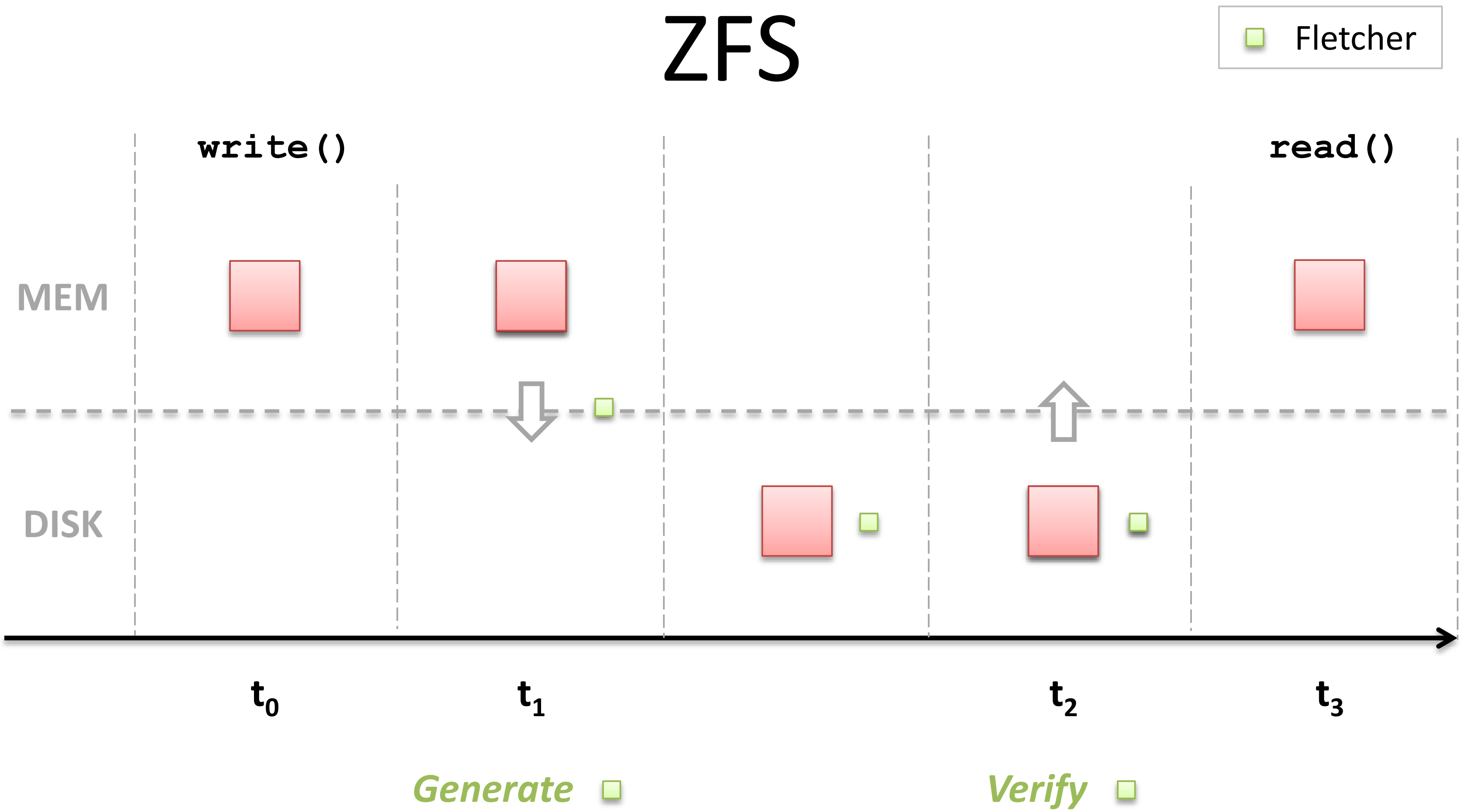


- Goal: Zettabyte Reliability
  - score: 17.5
  - none achieves the goal
- Server & Consumer
  - disk corruption dominates
  - need to protect disk data

Outline

- Introduction
- Analytical Framework
- **From ZFS to Z²FS**
  - **Original ZFS**
  - End-to-end ZFS
  - Z²FS : ZFS with flexible end-to-end data integrity
- Implementation
- Evaluation
- Conclusion

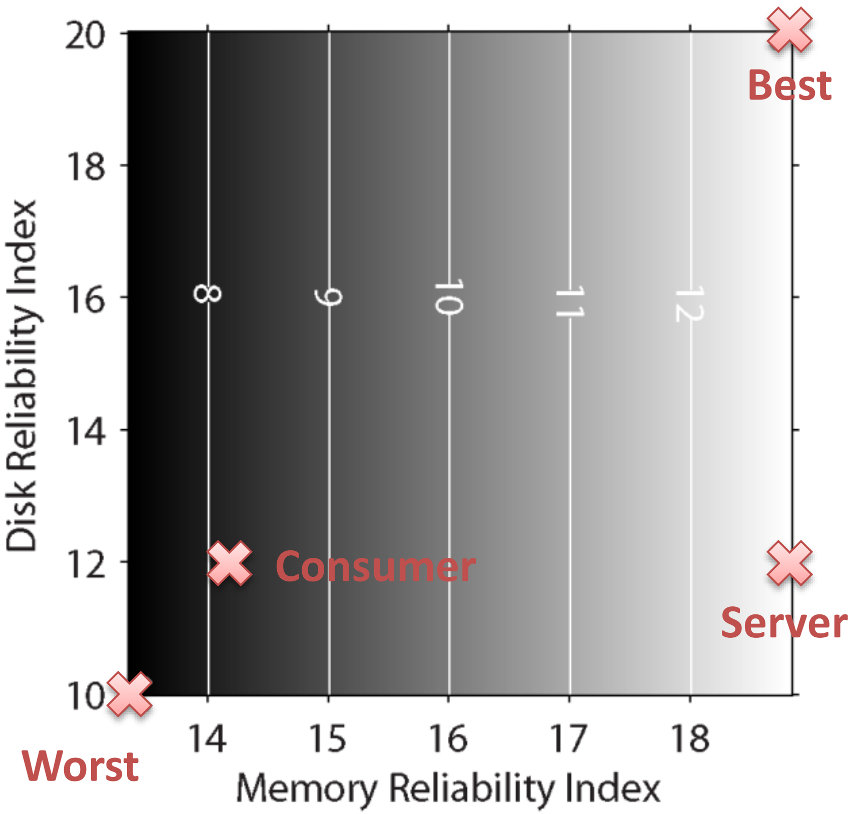
ZFS



Only on-disk blocks are protected

ZFS (cont.)

- Reliability Score ($t_{resident} = 1$)

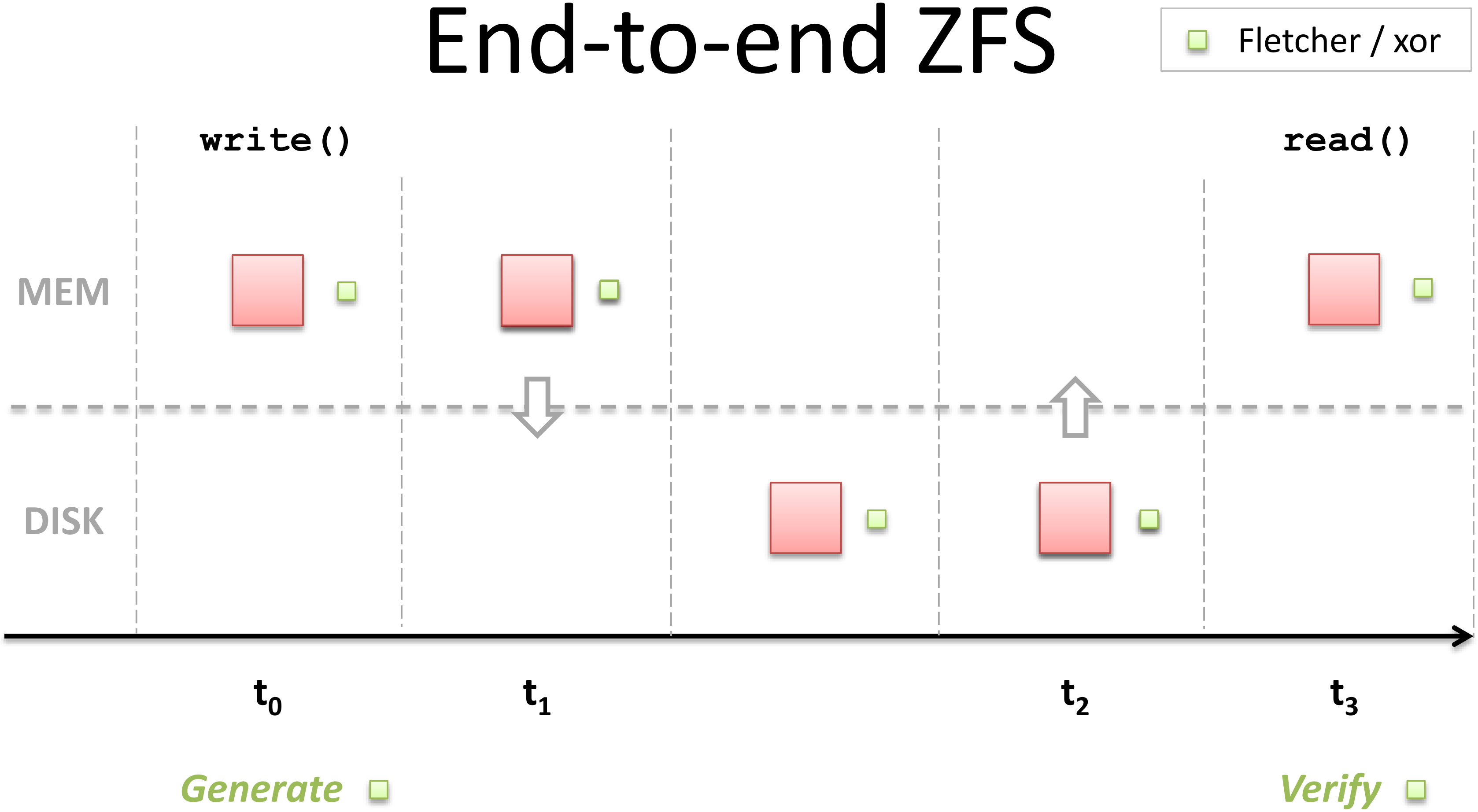


- Goal: Zettabyte Reliability
  - score: 17.5
  - Best: only Petabyte
- Now memory corruption dominates
  - need end-to-end protection

Outline

- Introduction
- Analytical Framework
- **From ZFS to Z^2FS**
  - Original ZFS
  - **End-to-end ZFS**
  - Z^2FS : ZFS with flexible end-to-end data integrity
- Implementation
- Evaluation
- Conclusion

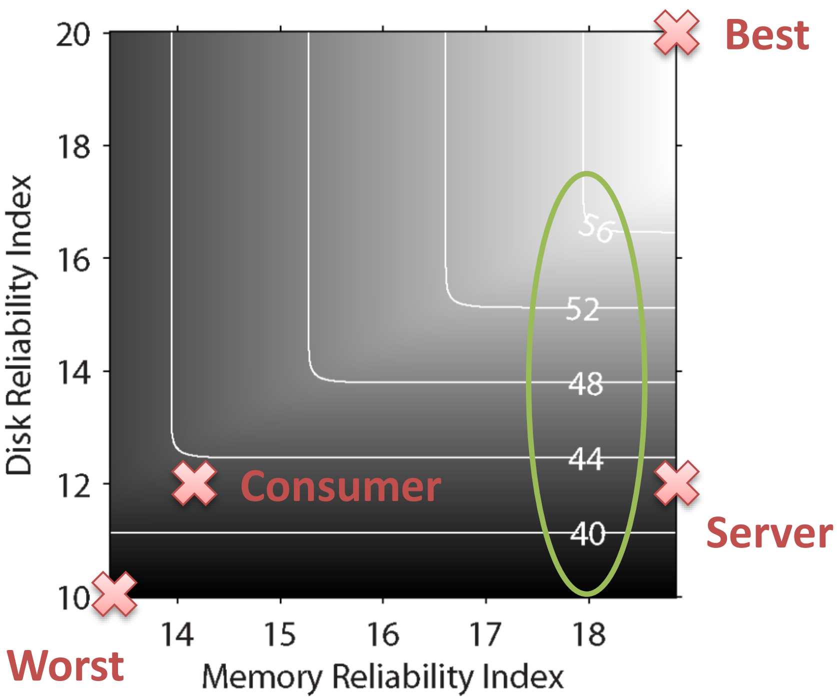
End-to-end ZFS



- Checksum is generated and verified only by application
- Only one type of checksum is used (Fletcher or xor)

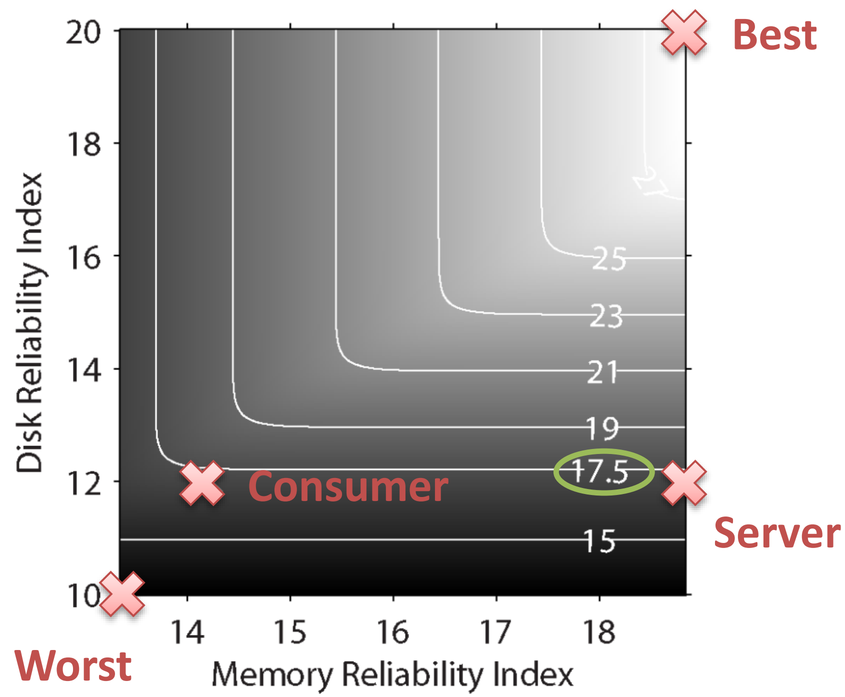
End-to-end ZFS (cont.)

- Reliability Score ($t_{resident} = 1$)



Fletcher

provide best reliability



xor

just fall short of the goal

Performance Issue

System	Throughput (MB/s)	Normalized
Original ZFS	656.67	100%
End-to-end ZFS (Fletcher)	558.22	85%
End-to-end ZFS (xor)	639.89	97%

Read 1GB Data from Page Cache

- End-to-end ZFS (Fletcher) is 15% slower than ZFS
- End-to-end ZFS (xor) has only 3% overhead
  - xor is optimized by the checksum-on-copy technique [Chu96]

Outline

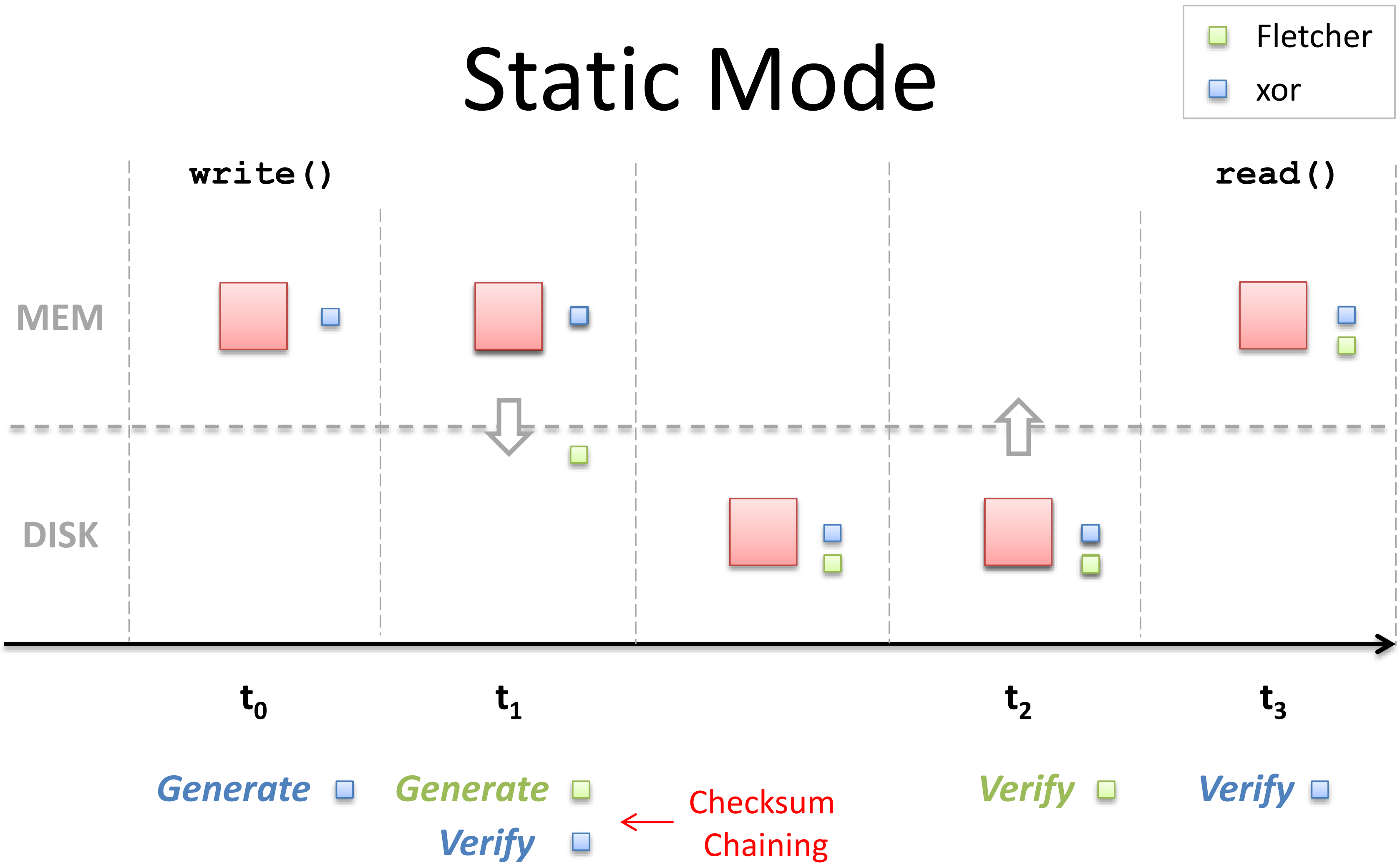
- Introduction
- Analytical Framework
- **From ZFS to Z^2FS**
  - Original ZFS
  - End-to-end ZFS
  - **Z^2FS : ZFS with flexible end-to-end data integrity**
- Implementation
- Evaluation
- Conclusion

Z^2FS Overview

- Goal
  - Reduce performance overhead
  - Still achieve Zettabyte reliability

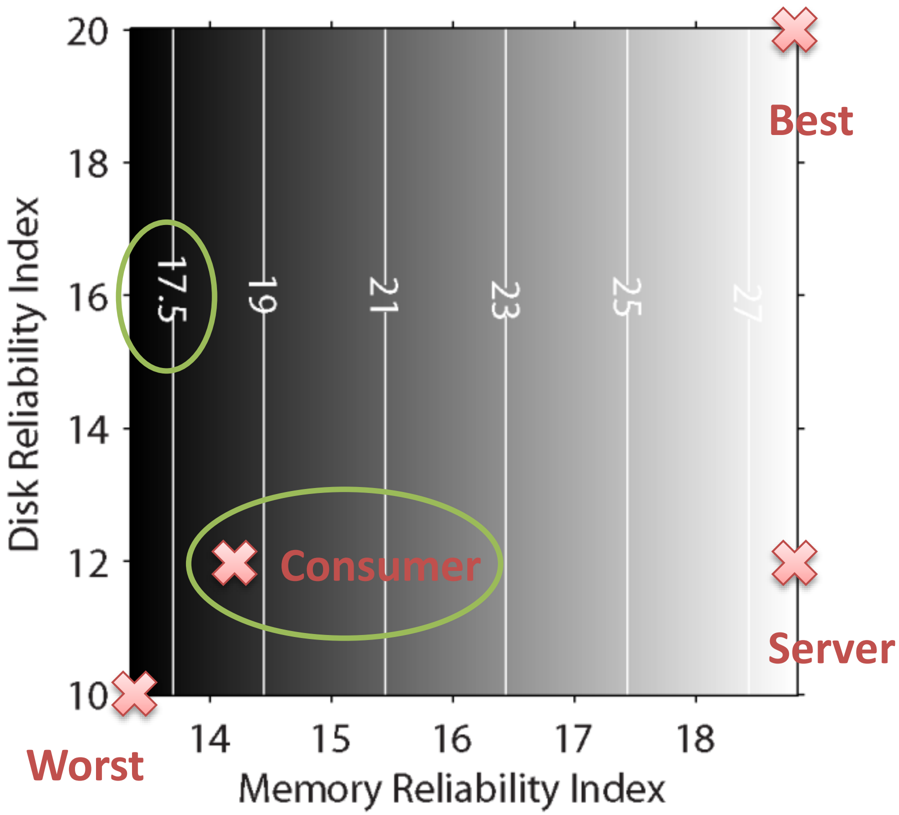
- Implementation of flexible end-to-end
  - Static mode: change checksum across components
    - xor as memory checksum and Fletcher as disk checksum
  - Dynamic mode: change checksum overtime
    - For memory checksum, switch from xor to Fletcher after a certain period of time
    - Longer residency time => data more likely being corrupt

Static Mode


Fletcher
xor
write()
read()
MEM
DISK
t0
t1
t2
t3
Generate
Generate
Verify
Checksum
Chaining
Verify
Verify

Static Mode (cont.)

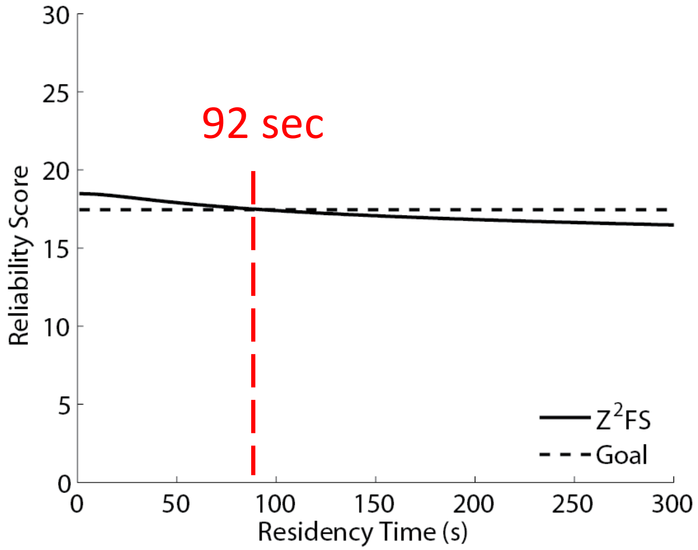
- Reliability Score ($t_{resident} = 1$)



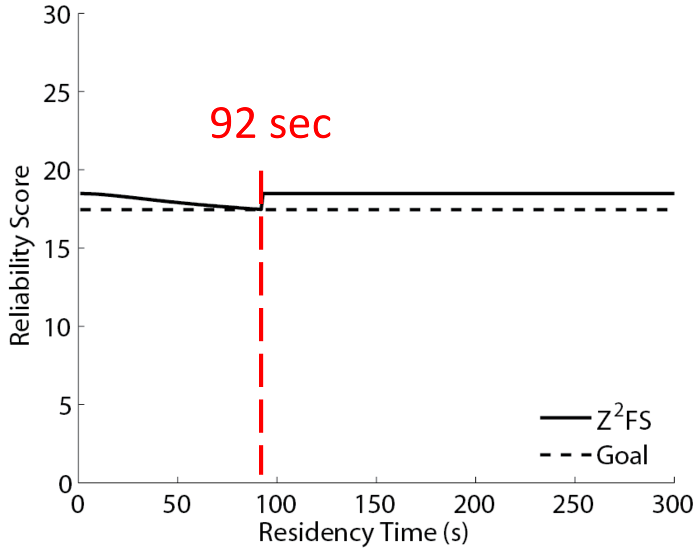
- Worst
  - use Fletcher all the way
- Server & Best
  - xor is good enough as memory checksum
- Consumer
  - may drop below the goal as $t_{resident}$ increases

Evolving to Dynamic Mode

- Reliability Score vs $t_{resident}$ for consumer



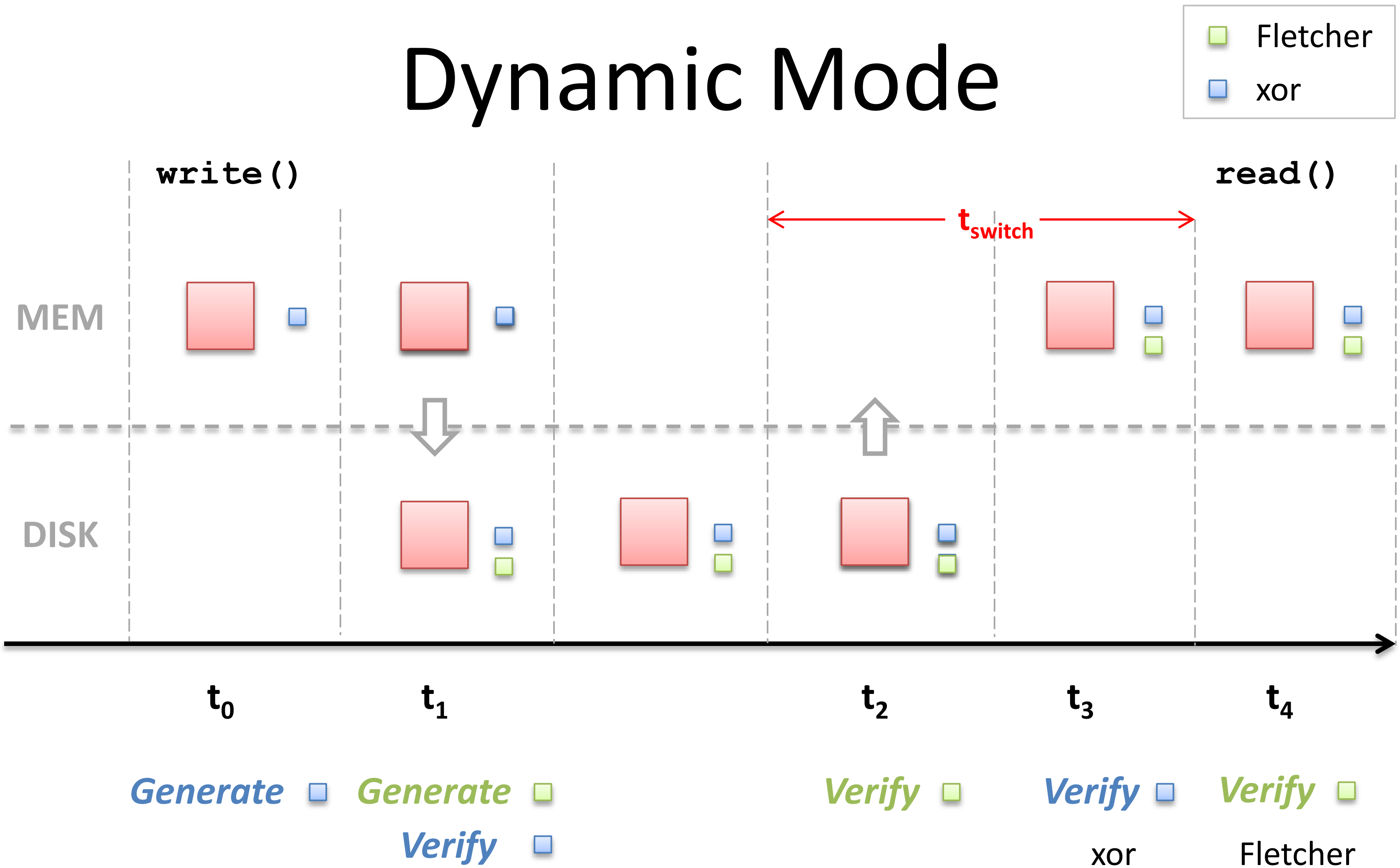
Static



Dynamic

switching the memory checksum from xor to Fletcher after 92 sec

Dynamic Mode



Outline

- Introduction
- Analytical Framework
- From ZFS to Z^2FS
- **Implementation**
- Evaluation
- Conclusion

Implementation

- Attach checksum to all buffers
  - User buffer, data page and disk block
- Checksum handling
  - Checksum chaining & checksum switching
- Interfaces
  - Checksum-aware system calls (for better protection)
  - Checksum-oblivious APIs (for compatibility)
- LOC : ~6500

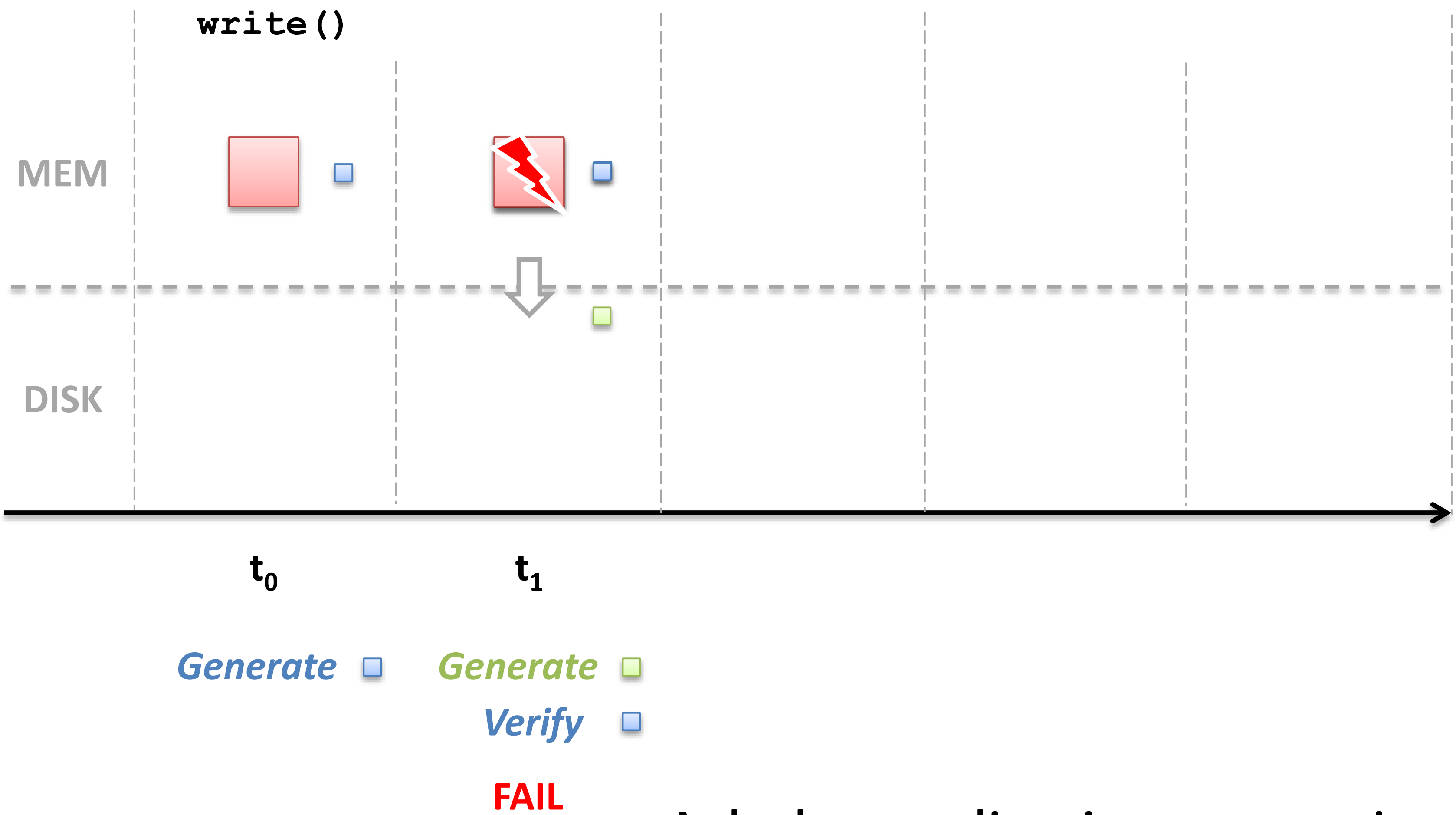
Outline

- Introduction
- Analytical Framework
- From ZFS to Z^2FS
- **Evaluation**
- Conclusion

Evaluation

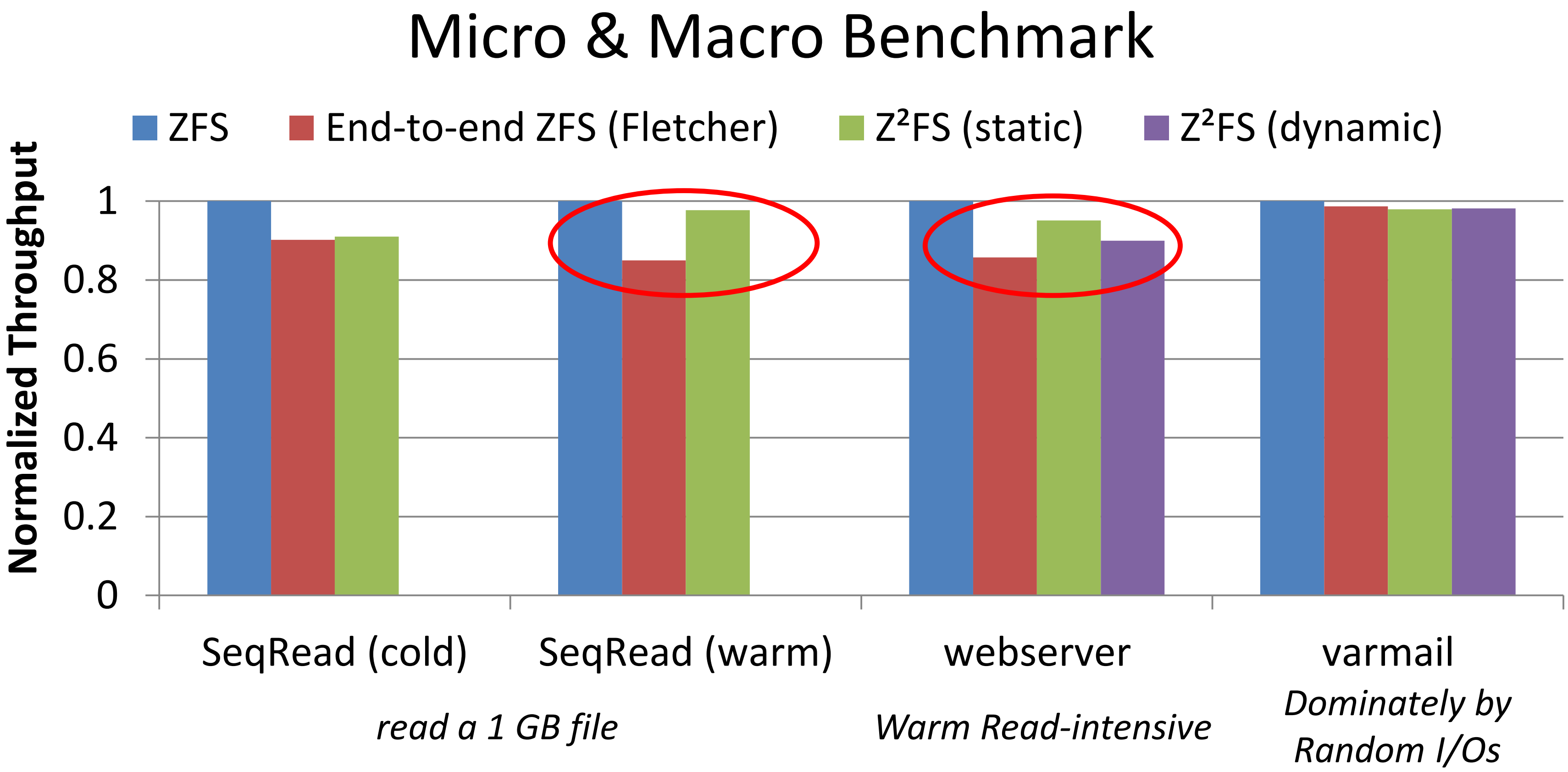
- Q1: How does Z^2FS handle data corruption?
  - Fault injection experiment
- Q2: What's the overall performance of Z^2FS?
  - Micro and macro benchmarks

Fault Injection: Z^2FS



Ask the application to rewrite

Overall Performance



- Better protection usually means higher overhead
- Z^2FS helps to reduce the overhead, especially for warm reads

Outline

- Introduction
- Analytical Framework
- From ZFS to Z^2FS
- Evaluation
- **Conclusion**

Summary

- Problem of straightforward end-to-end data integrity
  - Slow performance
  - Untimely detection and recovery
- Solution: Flexible end-to-end data integrity
  - Change checksums across component or overtime
- Analytical Framework
  - Provide insight about reliability of storage systems
- Implementation of Z^2FS
  - Reduce overhead while still achieve Zettabyte reliability
  - Offer early detection and recovery

Conclusion

- End-to-end data integrity provides comprehensive data protection

- One "checksum" may not always fit all
  - e.g. strong checksum => high overhead

- Flexibility balances reliability and performance
  - Every device is different
  - Choose the best checksum based on device reliability

Thank you!

Questions?

Advanced Systems Lab (ADSL)

University of Wisconsin-Madison

http://www.cs.wisc.edu/adsl

Wisconsin Institute on Software-defined Datacenters in Madison

http://wisdom.cs.wisc.edu/