End-to-end Data Integrity for File Systems: A ZFS Case Study

Yupu Zhang, Abhishek Rajimwale

Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin - Madison

End-to-end Argument

- Ideally, applications should take care of data integrity

- In reality, file systems are in charge
  - Data is organized by metadata
  - Most applications rely on file systems
  - Applications share data

Data Integrity In Reality

- Preserving data integrity is a challenge
- Imperfect components
  - disk media, firmware, controllers, etc.
- Techniques to maintain data integrity
  - Checksums [Stein01, Bartlett04], RAID [Patternson88]
- Enough about disk. What about memory?

Memory Corruption

- Memory corruptions do exist
  - Old studies: 200 – 5,000 FIT per Mb [O'Gorman92, Ziegler96, Normand96, Tezzaron04]
    - 14 – 359 errors per year per GB
  - A recent work: 25,000 – 70,000 FIT per Mb [Schroeder09]
    - 1794 – 5023 errors per year per GB
  - Reports from various software bug and vulnerability databases

- Isn't ECC enough?
  - Usually correct single-bit error
  - Many commodity systems don't have ECC (for cost)
  - Can't handle software-induced memory corruptions

The Problem

- File systems cache a large amount of data in memory for performance
  - Memory capacity is growing
- File systems may cache data for a long time
  - Susceptible to memory corruptions
- How robust are modern file systems to memory corruptions?

A ZFS Case Study

- Fault injection experiments on ZFS
  - What happens when disk corruption occurs?
  - What happens when memory corruption occurs?
  - How likely a bit flip would cause problems?

- Why ZFS?
  - Many reliability mechanisms
  - "provable end-to-end data integrity" [Bonwick07]

Results

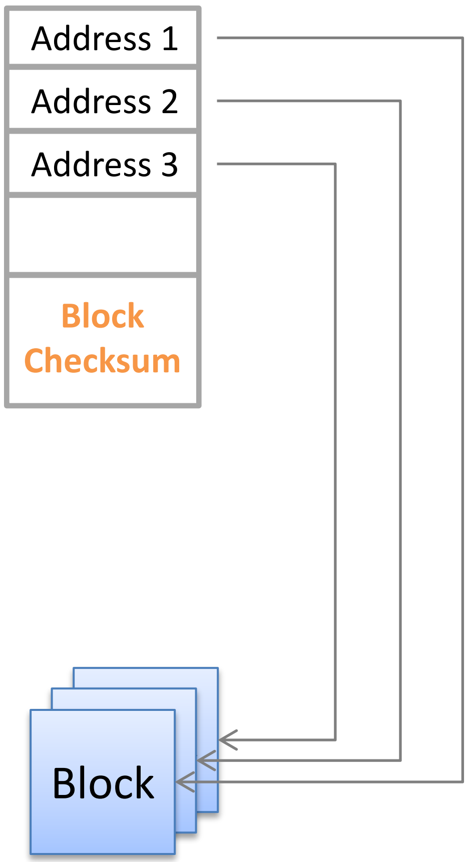
- ZFS is robust to a wide range of disk corruptions
- ZFS fails to maintain data integrity in the presence of memory corruptions
  - reading/writing corrupt data, system crash
  - one bit flip has non-negligible chances of causing failures
- Data integrity at memory level is not preserved

Outline

- Introduction
- **ZFS Background**
- Data Integrity Analysis
  - On-disk Analysis
  - In-mem Analysis
- Conclusion

ZFS Reliability Features

- Checksums
  - Detect silent data corruption
  - Stored in a generic block pointer
- Replication
  - Up to three copies (ditto blocks)
  - Recover from checksum mismatch
- Copy-On-Write transactions
  - Keep disk image always consistent
- Storage pool
  - Mirror, RAID-Z



Outline

- Introduction
- ZFS Background
- Data Integrity Analysis
  - **On-disk Analysis**
  - In-mem Analysis
- Conclusion

Summary of On-disk Analysis

- ZFS detects all corruptions by using checksums
- Redundant on-disk copies and in-mem caching help ZFS recover from disk corruptions
- Data integrity at this level is well preserved

(See our paper for more details)

Outline

- Introduction
- ZFS Background
- Data Integrity Analysis
  - On-disk Analysis
  - In-mem Analysis
    - **Random Test**
    - Controlled Test
- Conclusion

Random Test

- Goal
  - What happens when random bits get flipped?
  - How often do those failures happen?

- Fault injection
  - A trial: each run of a workload
    - Run a workload -> inject bit flips -> observe failures

- Probability calculation
  - For each type of failure
    - P (failure) = # of trials with such failure / total # of trials

Result of Random Test

Workload	Reading Corrupt Data	Writing Corrupt Data	Crash	Page Cache
varmail	0.6%	0.0%	0.3%	31 MB
oltp	1.9%	0.1%	1.1%	129 MB
webserver	0.7%	1.4%	1.3%	441 MB
fileserver	7.1%	3.6%	1.6%	915 MB

- The probability of failures is non-negligible
- The more page cache is consumed, the more likely a failure would occur

Outline

- Introduction
- ZFS Background
- Data Integrity Analysis
  - On-disk Analysis
  - In-mem Analysis
    - Random Test
    - **Controlled Test**
- Conclusion

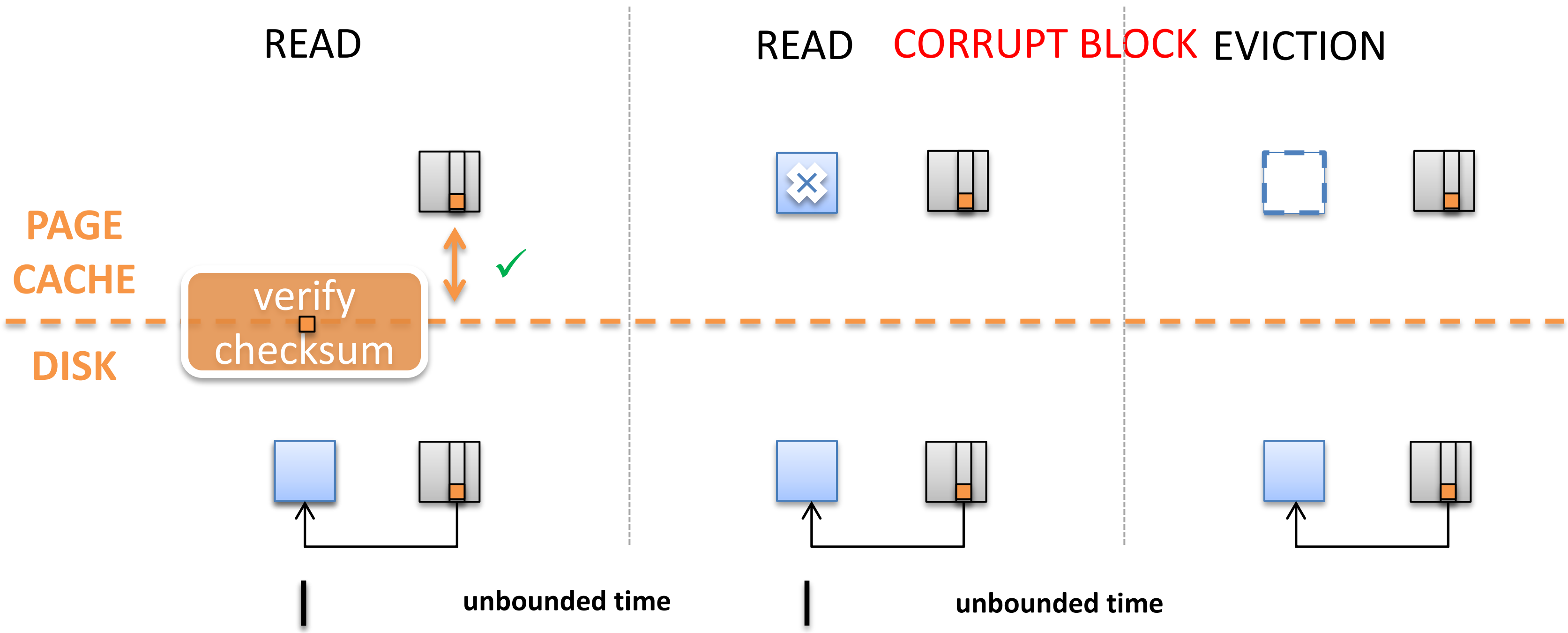
Controlled Test

- Goal
  - Why do those failures happen in ZFS?
  - How does ZFS react to memory corruptions?
- Fault injection
  - Metadata: field by field
  - Data: a random bit in a data block
- Workload
  - For global metadata: the “zfs” command
  - For file system level metadata and data: POSIX API

Result Overview

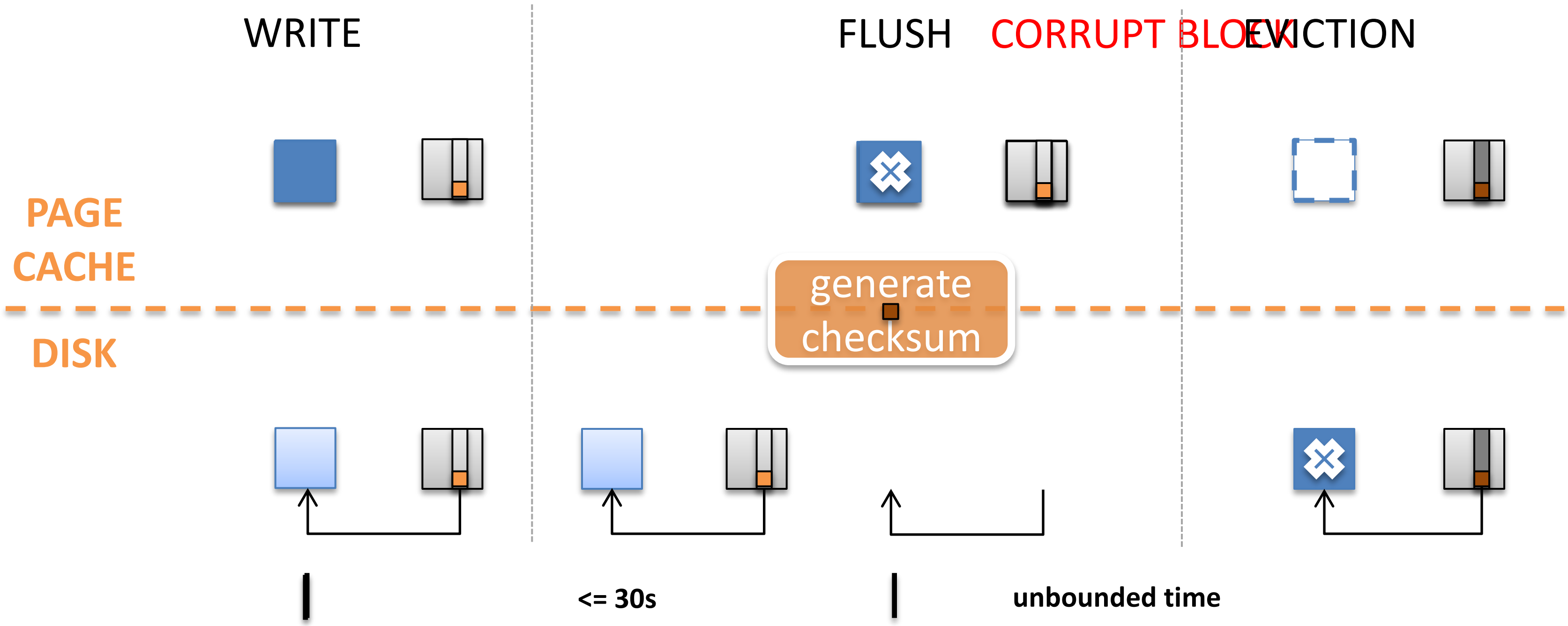
- General observations
  - Life cycle of a block
    - Why does bad data get read or written to disk?
- Specific cases
  - Bad data is returned
  - System crashes
  - Operation fails

Lifecycle of a Block: READ



- Blocks on the disk are protected
- Blocks in memory are not protected
- The window of vulnerability is unbounded

Lifecycle of a Block: WRITE

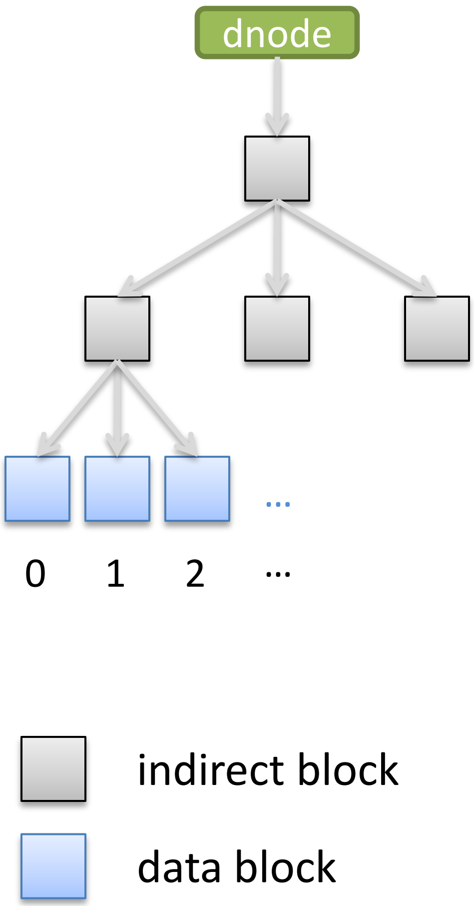


- Corrupt blocks are written to disk permanently
- Corrupt blocks are "protected" by the new checksum

Result Overview

- General observations
  - Life cycle of a block
    - Why does bad data get read or written to disk?
- Specific cases
  - Bad data is returned
  - System crashes
  - Operation fails

Case 1: Bad Data



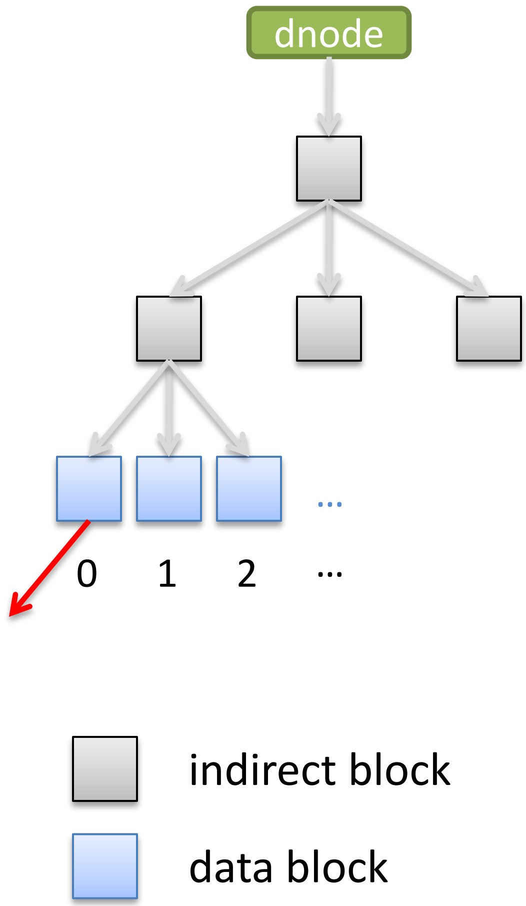
- Read (block 0)

  ✓ dn_nlevels == 3 (011)
  - return data block 0 at the leaf level

  × dn_nlevels == 1 (001)
  - treat an indirect block as data block 0
  - return the indirect block

BAD DATA!!!

Case 2: System Crash



- Read (block 0)

  ✓ dn_nlevels == 3 (011)
  - return data block 0 at the leaf level

  × dn_nlevels == 7 (111)
  - go down to the leaf level
  - treat data block 0 as an indirect block
  - try to follow an invalid block pointer
  - later a NULL-pointer is dereferenced

Case 2: System Crash (cont.)

```
uint64_t size = BP_GET_LSIZE(bp);
...
buf->b_data = zio_buf_alloc(size);
```

a block pointer, now invalid

could be an arbitrarily large value

```
void *zio_buf_alloc(size_t size)
{
    size_t c = (size - 1) >> SPA_MINBLOCKSHIFT;
    ASSERT(c< SPA_MAXBLOCKSIZE
                >>SPA_MINBLOCKSHIFT);

    return (kmem_cache_alloc
            (zio_buf_cache[c],KM_PUSHPAGE));
}
```

ASSERT(c<256)

disabled

```
void * kmem_cache_alloc
(kmem_cache_t *cp, int kmflag)
{
    ...
    ccp = KMEM_CPU_CACHE(cp);
    ...
    mutex_enter(&ccp->cc_ylock);
    ...
}
```

but now c > 256

NULL

NULL-pointer dereference

ccp is also NULL

CRASH!!!

Case 3: Operation Fail

- Open ("file")

  ✓ zp_flags is correct
  - open() succeeds

  × the 41st bit of zp_flags is flipped from 0 to 1
  - EACCES (permission denied)

Case 3: Operation Fail (cont.)

41st bit

.... 0010 ....

```
#define ZFS_AV_QUARANTINED 0x0000020000000000
```

```
...
if (((v4_mode & (ACE_READ_DATA|ACE_EXECUTE)) &&
     (zp->z_phys->zp_flags & ZFS_AV_QUARANTINED)))
{
   *check_privs = B_FALSE;
   return (EACCES);
}
...
```

Summary of Results

- Blocks in memory are not protected
  - Checksum is only used at the disk boundary
- Metadata is critical
  - Bad data is returned, system crashes, or operations fail
- Data integrity at this level is not preserved

Outline

- Introduction
- ZFS Background
- Data Integrity Analysis
  - On-disk Analysis
  - In-mem Analysis
- **Conclusion**

Conclusion

- A lot of effort has been put into dealing with disk failures
  - little into handling memory corruptions
- Memory corruptions do cause problems
  - reading/writing bad data, system crash, operation fail
- Shouldn't we protect data and metadata from memory corruptions?
  - to achieve end-to-end data integrity

Thank you!

Questions?

The ADvanced Systems Laboratory (ADSL)

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