Checksumming Software Raid

Brian Kroth, Suli Yang

2010-12-11

Outline

Outline

Who's that?

Brian Kroth

- Graduated with a Bachelors of Science in Math and CS from UW-Madison in 2007.
- Currently a Unix Systems Administrator for College of Engineering.
- Pursuing a Masters degree in Computer Science from UW-Madison.

Suli Yang

- Graduate student at UW-Madison
- Working on Master's degree in Computer Science and Physics
- Bachelors of Science in Physics from Peking University

The Problem

Disks Fail

- Disk failures are not stop-fail
  - Bit rot ($1/10^{14}$ bits according to ZFS paper)
  - Misdirected writes
  - Phantom writes
  - IO subsystem failures
- Partial failures can cause the loss of subtrees of data, or for files to become useless.
- Backups are expensive. Not a complete solution.

Solutions?

Available Solutions?

- RAID
  - Parity can recover from errors, but can't detect them.
  - *i.e.*: Doesn't handle any partial failures.
  - Expensive for home users.
- SCSI Data Integrity Extensions (DIF/DIX)

  (extends sector size by 8 bytes for integrity data)
  - Not widely available in consumer products.
  - Can't handle phantom writes or misdirected writes.
- FS Layer?
  - Hard to do without full integration ...
  - ZFS? Not available for Linux (ignoring FUSE port).

Outline

Our Solution

Checksumming RAID

- Standard RAID provides parity to recover a single block failure from a stripe.
- Extend RAID levels to include a checksum block in each stripe to determine when to recover.
- Write checksums when writing a block.
- Read them back and verify them for a given data/parity block upon read.
- If mismatch detected, issue a recovery from the remaining good data/parity blocks.

Checksumming RAID Layout

Design Analysis

Integrity Analysis

- Checksums spread over multiple disks/blocks.
- Bit rot caught and repaired through checksum verifications during read.
- Misdirected writes caught through checksum block number and data block offsets.
- Phantom writes of data blocks caught through checksums.
- Phantom writes of checksum blocks caught indirectly through multiple checksum mismatches during rebuild.
- DIX/DIF still useful for detecting IO subsystem problems at failure time.

Outline

Implementation

Software

- Altered the Multi-Device (MD) Software RAID layer in Linux 2.6.32.25 to make RAID4C and RAID5C.
- For calculating checksums we use the kernel's built-in CRC32 libraries. Fast, reliable, but some wasted space.
- All the parity and memory operations are done asynchronously but checksum calculations are currently synchronous.

Typical Processes

Typical Write

1. When writing to a data block, also calculate its checksum and new parity.
   Might need to read in the checksum block and possibly some other blocks during this process (eg: RMW).
2. Then issue writes for the data block, parity block and the checksum block.

Typical Processes continued ...

Typical Read

1. When issuing a read to a data block, also issue read to its corresponding checksum block.
2. Upon completion of reading the data block, wait for the checksum block read to complete.
3. Calculate and verify the checksums of the checksum block and the data block.

Typical Processes continued ...

Data Block Recovery

1. Checksum mismatch detected (during a read).
2. Read all other blocks in that stripe.
3. Restore the corrupted from parity calculation.

Checksum Block Recovery

1. Checksum block corruption detected (during a read to a checksum block).
2. Read all other blocks in that stripe.
3. Recalculate all the checksums of the blocks in that stripe and restore checksum block content based on the recalculation.

Implementation continued ...

Cache Policy

- A fixed size stripe cache pool is used to speed up read. So that if we read stuff from the same stripe later, the checksum and parity block don't need to be re-read from disk.
- Partial writes are buffered for a while (amount of time depend on memory pressure) in the hope that later write requests would turn them into full stripe writes.

Outline

Test Setup

Test Setup

- Debian VM with 2G RAM, 2CPUs, 1 system disk and 10 8G Virtual Disks
- ESX storage backed by a 14 disk 15K RAID50, which is otherwise bored
- Single disk tests run on a Dell Optiplex 755 with 2GB RAM, 3.0GHz Core2 Duo, and an extra 80GB Seagate.
- Compared original RAID 4/5 levels with our checksumming RAID 4C/5C levels.

Correctness

Correctness Test Description

1. Assembled a minimal 4 disk array for both RAID4C and RAID5C.
2. Used `dd` to corrupt the first 750 pages of a device (eg: `sdb1`) in the array.

   For RAID4C it corrupted only data blocks.

   For RAID5C it corrupted both data blocks and checksum blocks.
3. Read the first part of the array (eg: `md0`) to induce checksum mismatch detection and correction.
4. Count the messages reported in `dmesg`.

   [172.543364] raid5c: md0: checksum_page checksum mismatch detected (sector 728 on sdb2).

   [172.546539] raid5c: md0: checksum_page checksum mismatch corrected (8 sectors at 728 on sdb2) .

Correctness continued ...

Correctness Results

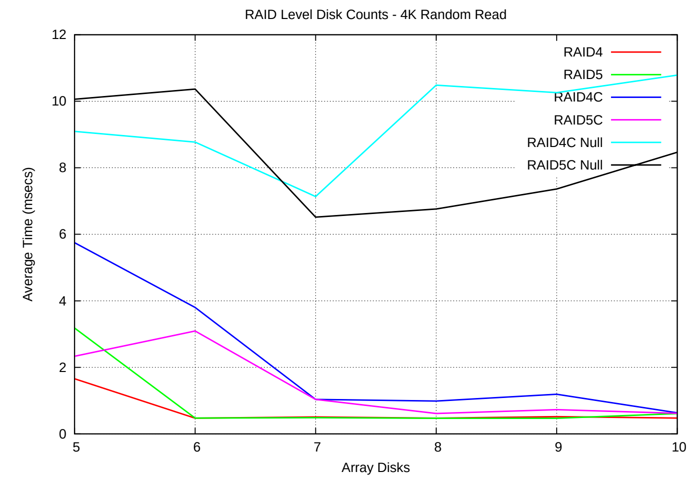
- RAID4C: We detected 750 corrupted data pages.
- RAID5C: We detected 494 corrupted data pages and 128 corrupted checksum pages.
  The remaining 128 are the parity blocks that we won't have read in normal operation.
- Verified that the file we read back was properly corrected.

Disk Count Performance

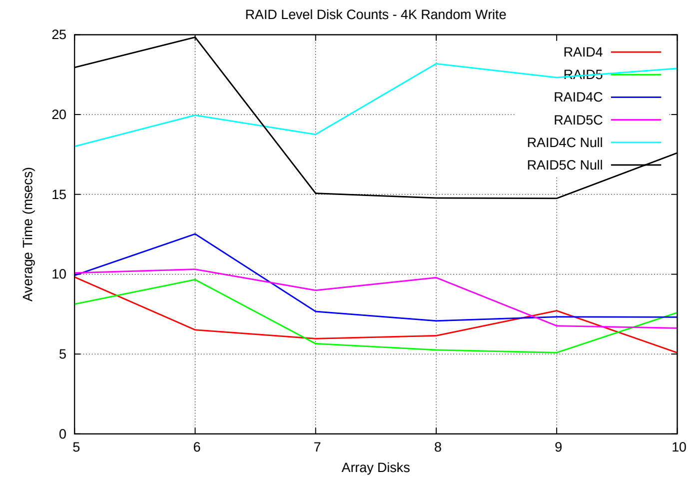
Disk Count Test Description

1. Assembled arrays of various numbers of disks using software RAID levels 4, 5, 4C, 5C.
2. Ran two tests with RAID levels 4C and 5C with an entire disk fully corrupted (eg: `dd if=/dev/urandom of=/dev/sdb1`)
3. Performed 100 100MB sequential reads/writes on the array.
4. Performed 50000 random 4K reads/writes on the array.
5. Averaged the results for each run into the following graphs.

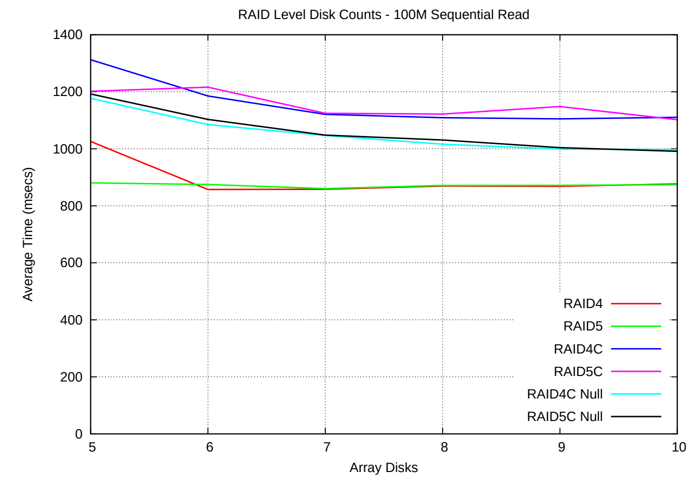
Disk Count Random Read Performance



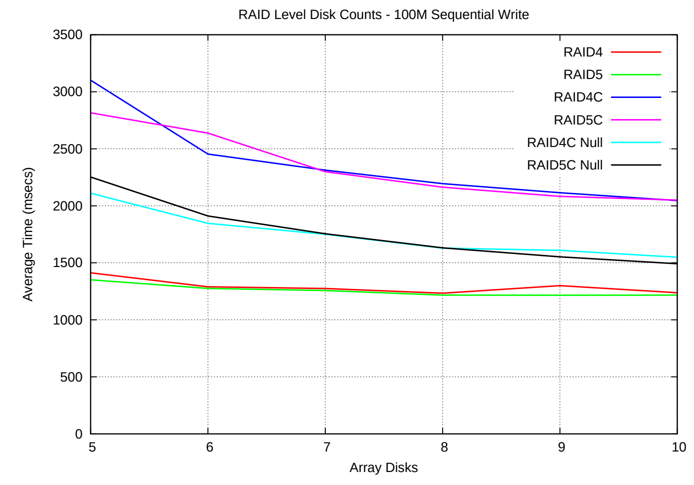
Disk Count Random Write Performance



Disk Count Sequential Read Performance



Disk Count Sequential Write Performance



Disk Count Conclusions

Disk Count Conclusions

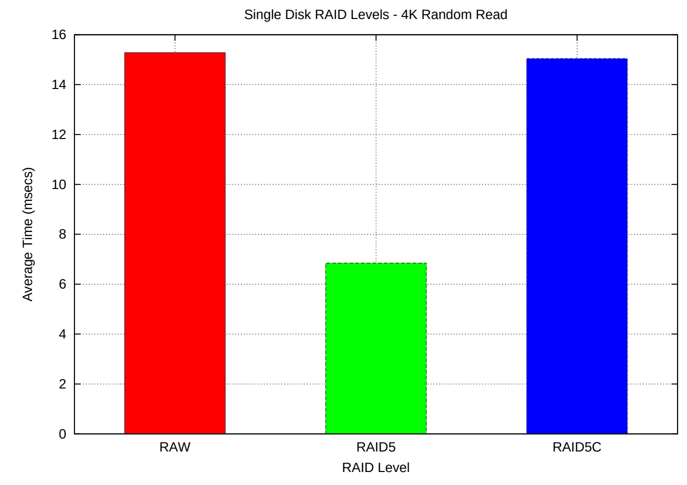
- Degraded arrays vary wildly and are much worse than healthy ones, as expected.
- Read performance of non-degraded arrays converges as the number of disks in the array increases.
  The cost of checksums are amortized over increased stripe size.
- Sequential read performance exhibits 50% overhead compared to original RAID levels.
- Sequential write performance exhibits 100% overhead.
  We think this is due to an extra read in our implementation.

Single Disk Performance

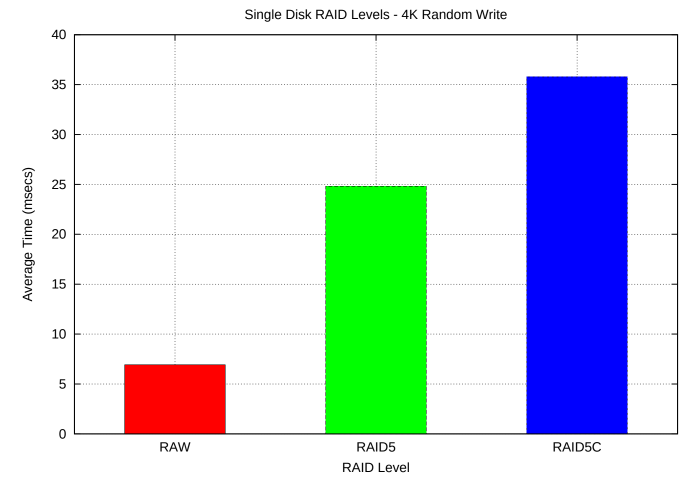
Single Disk Test Description

1. Split a single 80GB physical disk into 4 20G partitions and assembled arrays out of them.
2. Ran tests on RAW disk, RAID5, and RAID5C.
3. Performed 100 100MB sequential reads/writes on the array.
4. Performed 50000 random 4K reads/writes on the array.
5. Averaged the results for each run into the following graphs.

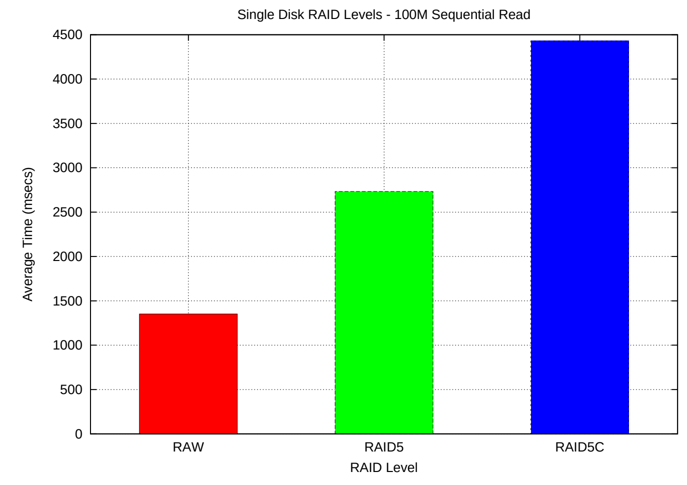
Single Disk Random Read Performance



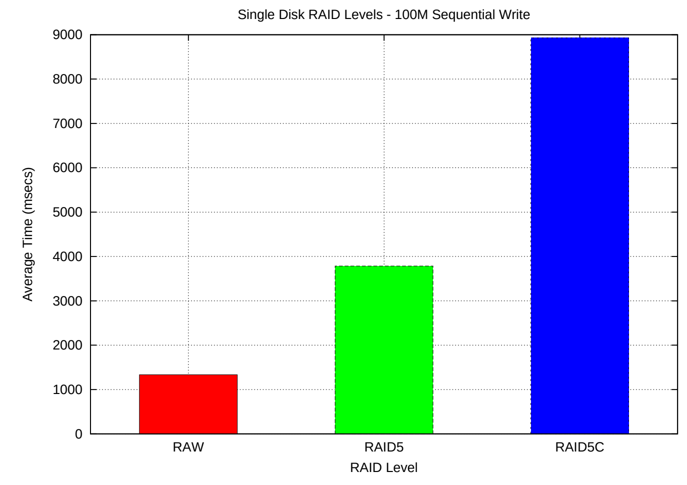
Single Disk Random Write Performance



Single Disk Sequential Read Performance



Single Disk Sequential Write Performance



Single Disk Conclusions

Single Disk Conclusions

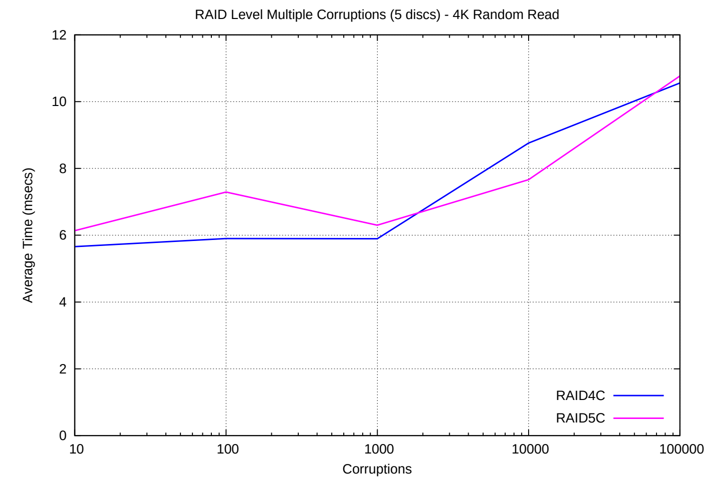
- As expected, this naive approach to single disk RAID results in excessive seeks which seriously degrades performance.

Corruptions Performance

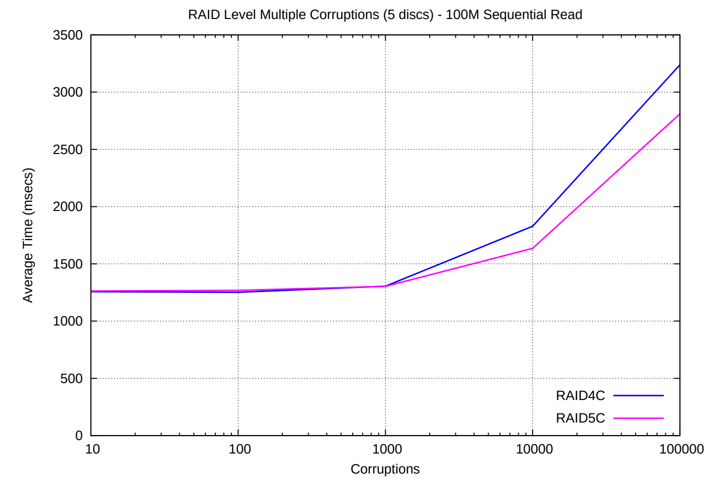
Corruptions Test Description

1. Assembled a 5 disk array for both RAID4C and RAID5C.
2. Used `dd` to randomly corrupt increasing amounts of sectors from a device (eg: `sdb1`) in the array.
3. Performed 100 100MB sequential reads/writes on the array.
4. Performed 50000 random 4K reads/writes on the array.
5. Averaged the results for each run into the following graphs.

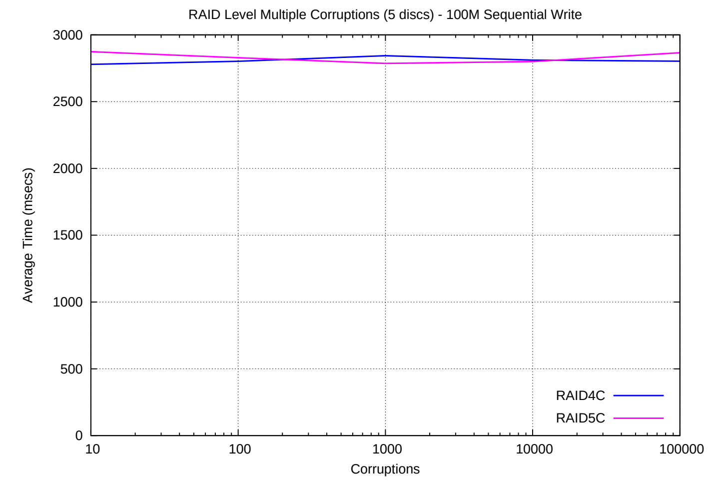
Corruptions Random Read Performance



Corruptions Sequential Read Performance



Corruptions Sequential Write Performance



Corruptions Conclusions

Corruptions Conclusions

- Sequential writes are largely unchanged due to the fact that we can skip checksum verification entirely for full stripe operations.
- In all other tests times predictably increase as the number of corruptions increase since there's a higher probability of recovery work to do.

Outline

Conclusions

Conclusions

- Corruptions in both data and checksum blocks are caught and corrected.
- Performance overhead is within 50-100% in our naive implementation.

Conclusions continued ...

Future work

- Room for improvements
  - Asynchronous checksum calculations.
  - Skip checksum block reads during full stripe writes.
  - More optimized checksum calculation (kernel loops over array one byte at a time).
  - More space efficient layout.
  - Better single disk layout.
  - Incomplete implementation support for growing, reshaping, raid6, initialization, etc.
  - Journal guided resync through LVM layers ...

Conclusions continued ...

Crash Recovery

- Partial write crash recovery poses a problem. Checksums/parity/data blocks may not be consistent.
- Really the only solution (short of COW) is to rebuild the checksums/parity.
  - We can reuse prior work on *Journal Guided RAID Resynchronization* to have the journalled filesystem(s) on top of the RAID to inform it which stripes should be rebuilt.
  - MD has also added support for an intent log which can do the same thing, at worse performance.

Questions?

Questions?