RAID sucks

(and what we can do about it)

Christoph Hellwig

RAID: Redundant Array of ~~Inexpensive~~*ndependent* Disks

RAID 0 Striping (no actual redundancy)
RAID 1 Mirroring
RAID 2 Bit-level striping with dedicated parity
RAID 3 Byte-level striping with dedicated parity
RAID 4 Block-level striping with dedicated parity
RAID 5 Block-level striping with distributed parity
RAID 6 Block-level striping with double distributed parity

Parity RAID: erasure coding over disks

Generalized parity RAID Layout:

$C = D + P$

$C :=$ *number of disks*

$D :=$ *number of data units*

$P :=$ *number of parity units*

Allows up to P failures

Reconstructs based on parity and the remaining data units

Parity RAID encoding

Single Parity uses simple XOR encoding

Additional Parities need to use more complex encodings

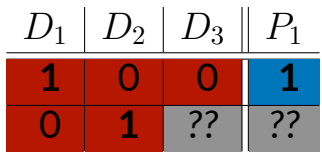
- Linux RAID 6 uses Reed-Solomon for second parity
- Polynomial chosen for minimizing computational overhead (in 2003)

D_1	D_2	D_3	P_1
0	0	0	0
1	0	0	1
0	1	1	0
1	1	1	1

Bit layout for single XOR parity

Terminology

Stripe Unit	Individual element of data
Parity Unit	Element of parity for D stripe units
Chunk	Another name for *Stripe Unit*
Stripe	Set of stripe and parity units



Full Stripe	A stripe that contains a complete $C = D + P$
Partial Stripe	A stripe where not all data units are updated: $D_{updated} < D$

Basic RAID implementation

- Presents a block device abstraction
  - Stack block device
  - Fixed layout
  - Each Stripe spans all disks
  - Adding an extra disk completely changes geometry
- Asymmetric I/O pattern
  - Each write of a Stripe must update all $D + P$ devices
  - Normal reads access $1 <= N <= D$ devices
  - Degraded reads access all $D + P$ devices

Write Hole

All $D + P$ disks must store coherent data

Easy enough when first writing data

Updates can be lost for only some of the disks

-> **Data corruption** due to mix of old and new data

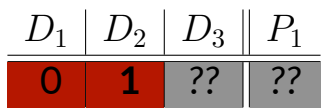
	D_1	D_2	D_3	P_1
Old data	0	1	1	0
New data	0	1	0	1
Inconsistent data after power fail	0	1	1	1

Read-modify write on Partial Stripe writes

All stripe units are required to generate parity

Must read all stripe units that are not updated

- Turns a write into a read, arithmetic operation and write
- Bad performance
- And also subject to the *Write Hole* problem



Single Disk bottleneck for rebuilds

When a disk fails, rebuild is bottlenecked by writes to the replacement

Solutions are well known

Write Hole avoidances

- Log Structured RAID
  - Remap logical block to different physical location on update
  - Garbage collection frees up full stripes
- Stage changes outside main layout
  - Batch up enough data to always write back full Stripes
  - Can use a journal or non-volatile memory

Eliminate RMW for partial stripe writes

- Pad writes to full stripes
  - Fill the rest of a stripe with dummy data
  - (Semi)-Permanently loses capacity
- Variable stripe geometry
  - Reduce D so that it fits the length of the write
  - Permanently loses capacity (but less so than padding)
- Partial parity logging
  - Only write parity after all data blocks are updated
  - Store it in a log/journal in the meantime

Declustering

Declustering means spreading $D + P$ stripes over more than $D + P$ disk

- Keeps spare capacity on each disk
- Allows for (relatively) easy capacity adjustment
- Supports devices with different sizes

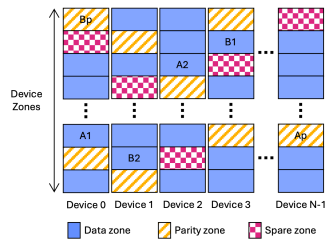
Placement needs to be smarts so that most disks participate in rebuilds

XRAID - declustered RAID for Zoned XFS

- Zoned XFS always writes data sequentially out of place:
  - **No overwrites, no Write hole!**
- XFS has an efficient transaction log at its core
  - Can log partial parity in the existing file system log
- Add support for RAID with 0 to N parities to XFS:
  - RAID sits below file system
  - Declustered layout for fast rebuilds and easy capacity adjustment

XRAID - Layout

- Only covers Zoned RT device data
  - Uses MDRAID mirroring for metadata
- Parity and striping applied at the real-time group level
  - Each RTG maps to $D + P$ physical zones
  - Mapping stored in rmap inode xattrs
  - Hot spare capacity can be on all devices



XRAID - Partial parity logging

- Only logs parity, not data
  - Because overwrites are impossible
- Log forces can require padding of stripe
  - Happens rarely outside of unmount/freeze
  - Space loss recoverable when garbage collecting
- Partial parity is logged together with file system metadata
  - And only in-memory except for synchronous writes

Rebuild

- Rebuild only needs to rebuild actually written data
  - Real-time groups track valid blocks and locations for each zone
  - Skip unused zoned, or larger enough chunks of deleted data
- Rebuild does not conflict with foreground I/O
  - Zones are immutable once written
- Rebuild order is dynamic
  - Fix multiply degraded zones first
  - Rebuild data before parity to reduce read impact

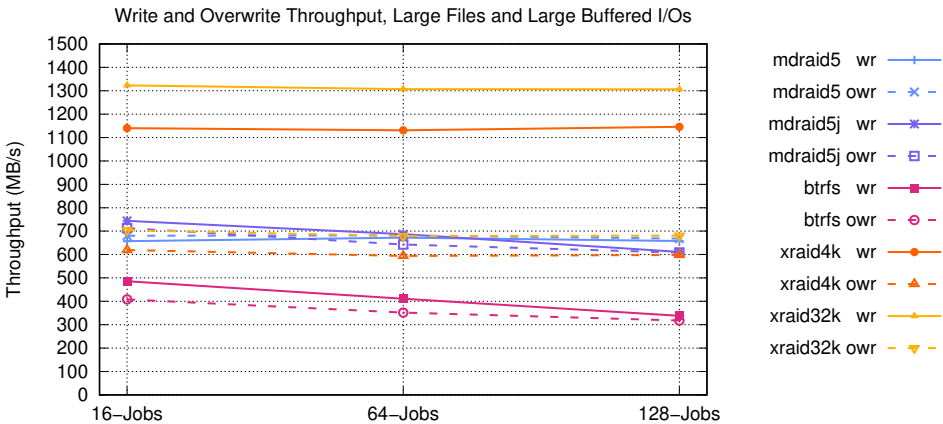
HDD Benchmarks

HDD setup

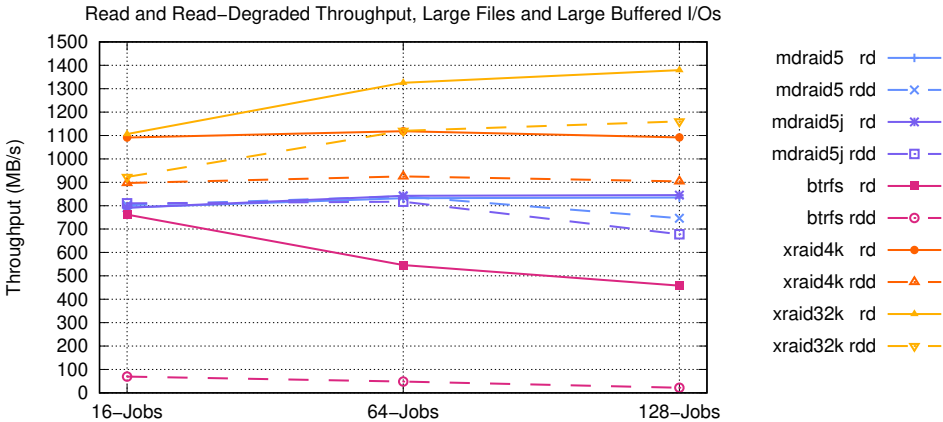
- 48-core dual-socket Xeon 8260, 256GiB RAM
- 2x WD DC-SN650 for metadata (mdraid mirror)
- 6x WD DC-HC590 26 TB SATA HDD (conventional)
- 6-21x WD DC-HC690 30 TB SATA HDD (SMR)

Thanks to Damien for benchmarking and plots!

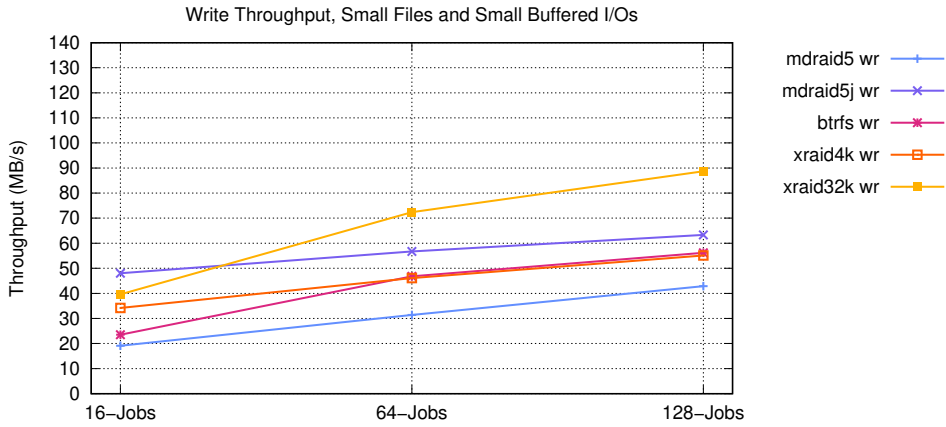
CMR HDD performance - large writes



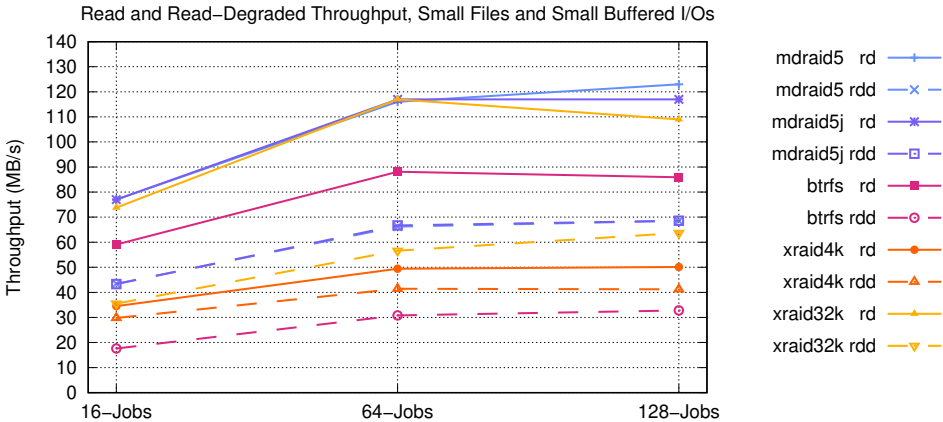
CMR HDD performance - large reads



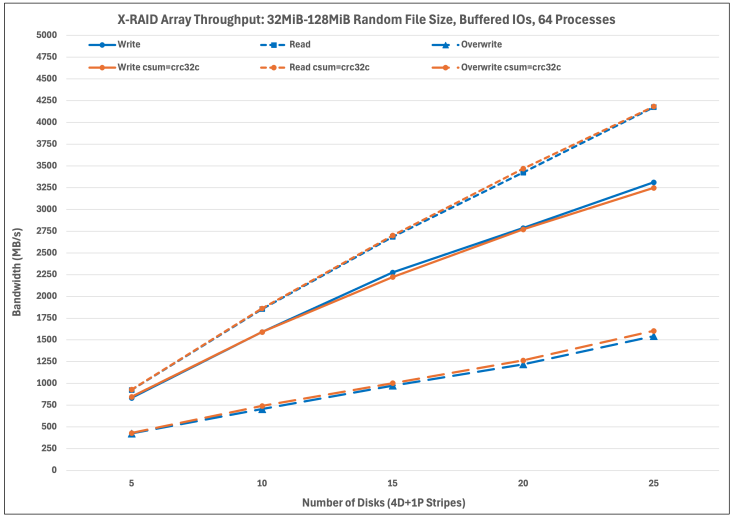
CMR HDD performance - small synchronous writes



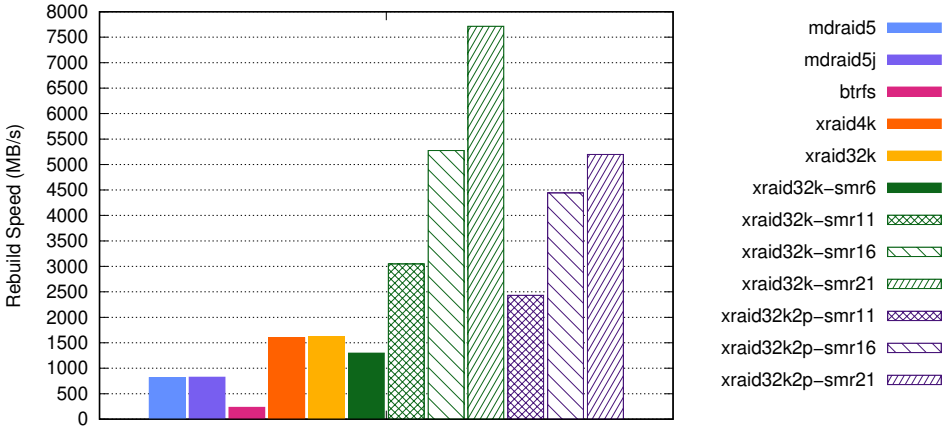
CMR HDD performance - small reads



SMR HDD performance scaling



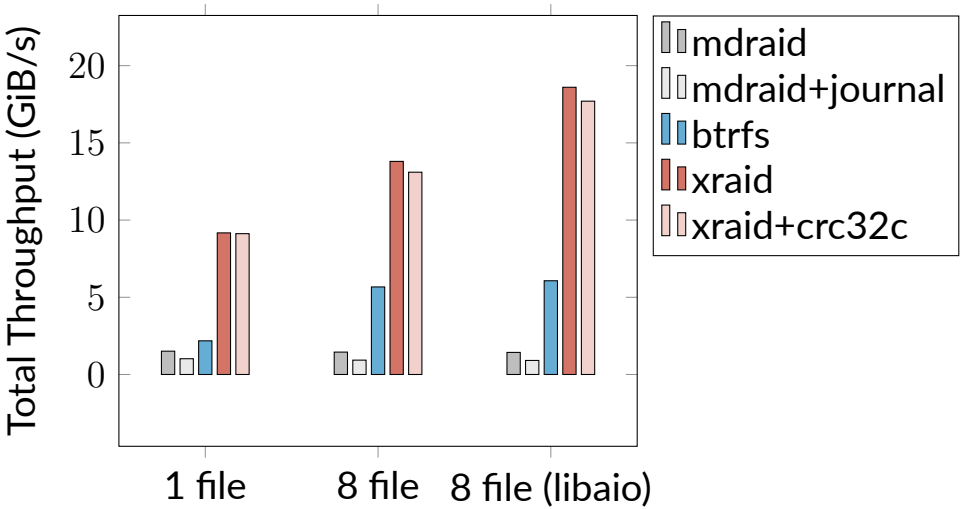
HDD rebuild speed



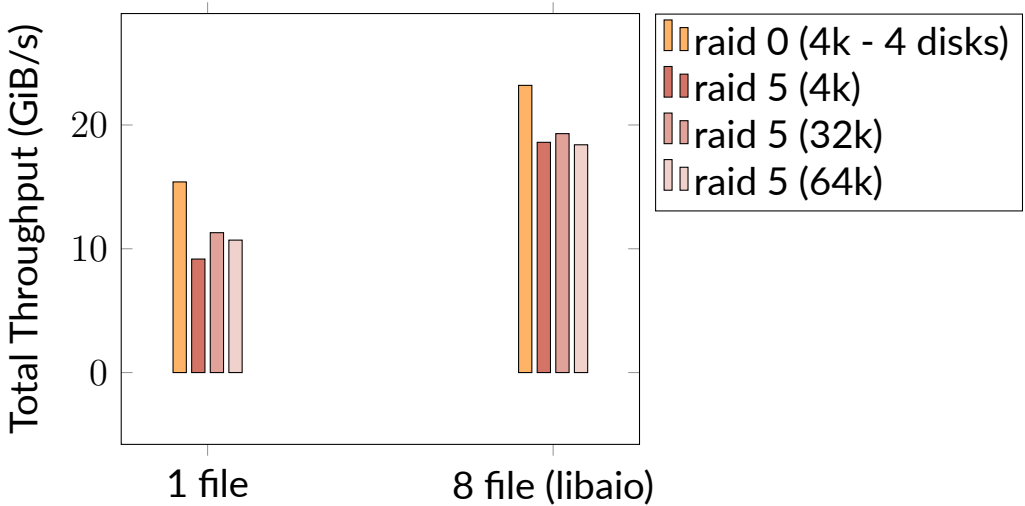
SSD Benchmarks

- SSD setup
  - 16-core / 32-thread AMD EPYC 8124P
  - 1x 4 TiB Samsung PM9D3a for metadata / mdraid journal
  - 5x 4 Tib Samsung PM9D3a for RAID (4 + 1)
- Very simple throughput oriented tests
  - 1MiB direct Read/Write using fio
  - Using io_uring or libaio when noted
  - Single file or 8 files, queue depth 16

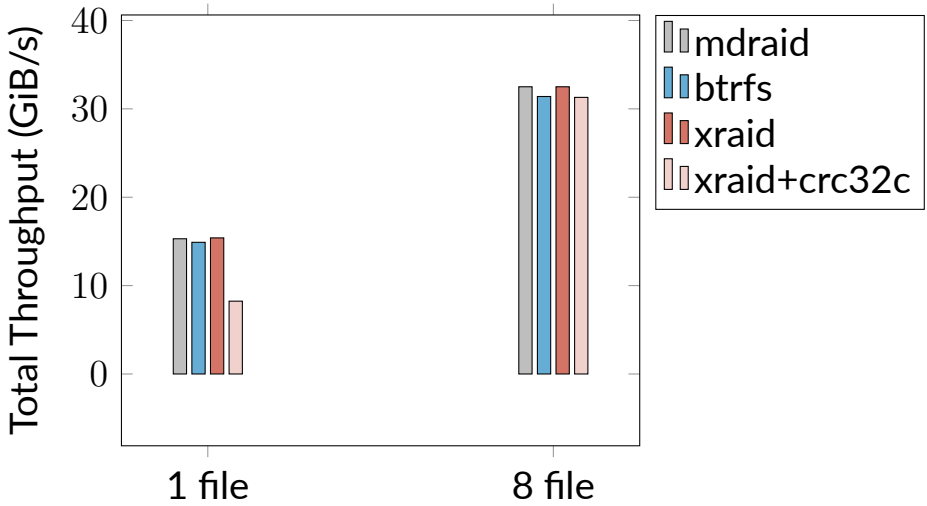
SSD Write Performance



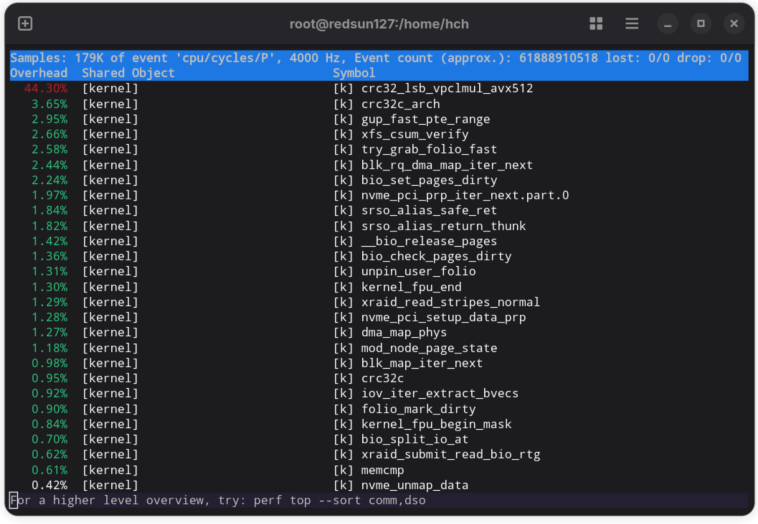
SSD Write Performance - fifty shades of XRAID



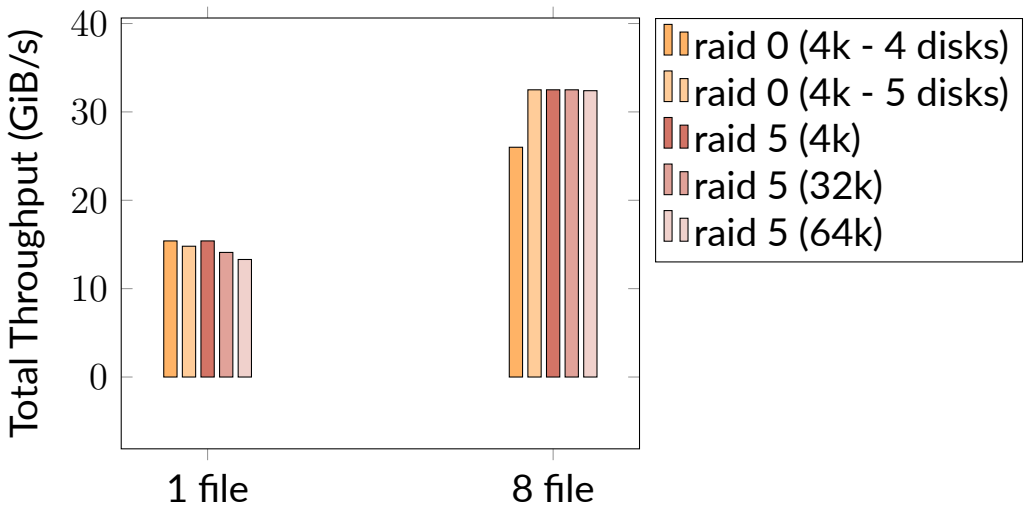
SSD Read Performance



SSD Read Performance - perf top



SSD Read Performance - fifty shades of XRAID



Triple-Parity and beyond

- Multi-TB devices need more than 2 parities
  - Existing Linux RAID libraries only support up to 2
- SnapRAID has a library to support up to 6 parities
  - Kernel port submitted in 2013, but not merged
  - Evolved since then with additional optimized implementations
- Re-ported to Linux 7.x
  - relative low effort the initial port
  - additional work for parity updates and kernel optimizations

The Polynomial matters

Parity computation operates in $GF(2^8)$

- Can use any 8th degree Polynomial
- No difference in data integrity, chosen for performance.

A tale of two polynomials

Linux RAID6	$(x^8 + x^4 + x^3 + x^2 + 1)$
AES	$(x^8 + x^4 + x^3 + x + 1)$

vgf2p8mulb

The *vgf2p8mulb* x86 instruction multiplies in $GF(2^8)$ in a single cycle

- Uses the AES polynomial
- Part of the GFNI (Galois Field New Instructions) extension

Triple-Parity in 8 single-cycle instructions

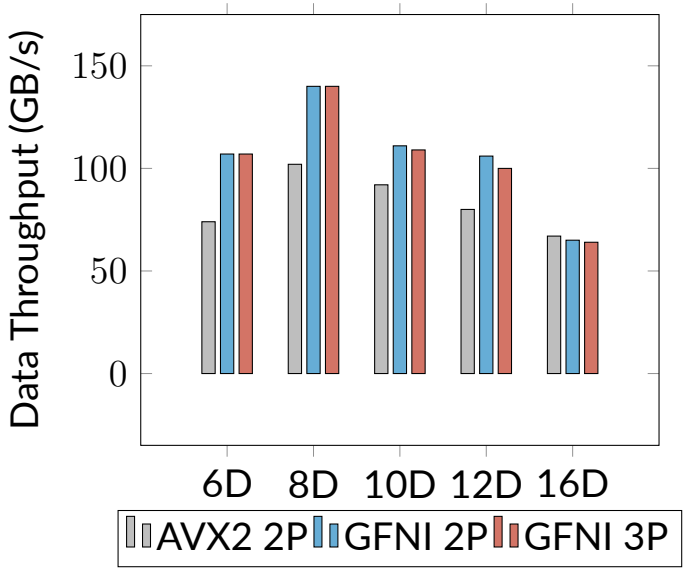
```
for ( ; d < nd; d++) {
        /* load data */
        asm volatile ("vmovdqa64 %0,%%zmm6" : : "m" (v[d][i]));

        /* 1st parity: plain XOR */
        asm volatile ("vpxorq %zmm6,%zmm0,%zmm0");

        /* 2nd parity */
        asm volatile ("vpbroadcastb %0,%%zmm7" : : "m" (gfcauchy[1][d]));
        asm volatile ("vgf2p8mulb %zmm6,%zmm7,%zmm12");
        asm volatile ("vpxorq %zmm12,%zmm1,%zmm1");

        /* 3rd parity */
        asm volatile ("vpbroadcastb %0,%%zmm8" : : "m" (gfcauchy[2][d]));
        asm volatile ("vgf2p8mulb %zmm6,%zmm8,%zmm13");
        asm volatile ("vpxorq %zmm12,%zmm1,%zmm1");
}
```

Parity generation performance



The XRAID team

Christoph Hellwig	Instigator and chief trouble maker
Damien Le Moal	GC performance, benchmarks, block layer infrastructure
Wilfred Mallawa	Observability in kernel, CLI and GUI, bug fixes
Johannes Thumshirn	Device management kernel and tools
Shin'ichiro Kawasaki	CI setup and testing
Hans Holmberg	Benchmarking, Zoned XFS improvements
Naohiro Aota	Competitive benchmarking

Additional shoutouts

Andrea Mazzoleni	Snapraid RAID code
Darrick J. Wong	XFS RT device modernisation and online repair

References

- XRAID source code
  - kernel
  - xfsprogs
- SnapRAID
  - Website
  - Source Code
- Articles and Papers
  - Retrofitting Zoned Storage into a Scalable Enterprise File System
  - The mathematics of RAID-6
  - Triple-Parity RAID and Beyond

Questions?