Vectorizing RAIDZ calculations in ZFS for speeding up data reconstruction

Gvozden Nešković
neskovic@compeng.uni-frankfurt.de

LSDMA Spring Meeting
GSI

March 2016

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

GSI

FIAS Frankfurt Institute for Advanced Studies

Outline

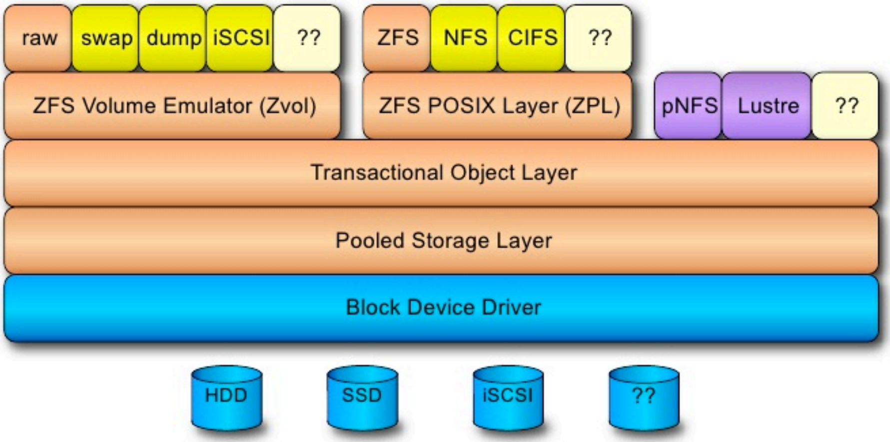
ZFS

File system

- ZFS features:
  - Data and metadata checksums
  - Copy on Write (COW) filesystem
  - Online `fsck` (scrub)
  - Snapshots
  - Compression
  - Volume Management
- Supported volumes:
  - Striped (RAID 0)
  - Replication (RAID 1)
  - RAIDZ1 (RAID 5)
  - RAIDZ2 (RAID 6)
  - RAIDZ3

ZFS

Layers


raw
swap
dump
iSCSI
??
ZFS
NFS
CIFS
??
ZFS Volume Emulator (Zvol)
ZFS POSIX Layer (ZPL)
pNFS
Lustre
??
Transactional Object Layer
Pooled Storage Layer
Block Device Driver
HDD
SSD
iSCSI
??

ZFS & Lustre

Motivation

- Lustre on `ldisk`:
  - Version of `ext3/4` (`ldisk`)
  - Random writes limited by disk IOPs
  - Limited OST size
  - Long `fsck` time
  - Hardware RAID controllers required
- Lustre on ZFS:
  - Lustre is decoupled from `ldisk` by creating Object Storage Device (OSD) layer
  - OSD interface is coupled with the ZFS Transactional Object Layer, bypassing POSIX layer
  - Random writes bound by disk bandwidth, not IOPs
  - Most of ZFS features used by Lustre (ZFS Intent Log is missing)

ZFS RAIDZ Pools

Overview

- RAIDZ1/2/3 Levels:
  - Provide Error Correction Erasure scheme
  - RAIDZ2/3 use specialized Reed-Solomon Codes on GF$[2^8]$ elements (byte size)

$$\mathbf{P} = D_0 \oplus D_1 \oplus \ldots \oplus D_n \quad (1)$$

$$\mathbf{Q} = 2^0 * D_0 \oplus 2^1 * D_1 \oplus \ldots \oplus 2^n * D_n \quad (2)$$

$$\mathbf{R} = 4^0 * D_0 \oplus 4^1 * D_1 \oplus \ldots \oplus 4^n * D_n \quad (3)$$

where $\mathbf{2} = X^1$, and $\mathbf{4} = X^2$ in GF$[2^8]$ (generated with $X^8 + X^4 + X^3 + X^2 + 1$ polynomial)

- Goals:
  - Implement efficient multiplication by 2 and 4 (parity generation)
  - Implement efficient multiplication by any element/constant (data reconstruction)

ZFS RAIDZ Calculation

Parity generation

- To simplify multiplication Q and R are calculated as:
  - $Q = D_0 \oplus 2 * (D_1 \oplus ... \oplus 2 * (D_{n-1} \oplus 2 * D_n))$
  - $R = D_0 \oplus 4 * (D_1 \oplus ... \oplus 4 * (D_{n-1} \oplus 4 * D_n))$
- Originally, these multiplications are performed as follows:

```
#define  MUL_2(x)   (((x) << 1) ^ (((x) & 0x80) ? 0x1d : 0))
#define  MUL_4(x)   (MUL_2(MUL_2(x)))
```

- In reconstruction path all multiplications are performed with table lookups:

```
l = raidz_log2[a] + raidz_log2[b];
if (l > 255) l -= 255;
return (raidz_pow2[l]);
```

- Q and R codes are easily translated to vectorized code[1]

[1] H. P. Anvin. The mathematics of RAID-6, 2011

ZFS RAIDZ Calculation

Data reconstructions

- Multiplication with a constant[1]:
  - Uses 2 precomputed 16 element lookup tables (left-right table)

$$a = (a_l << 4) \oplus a_r \quad (4)$$

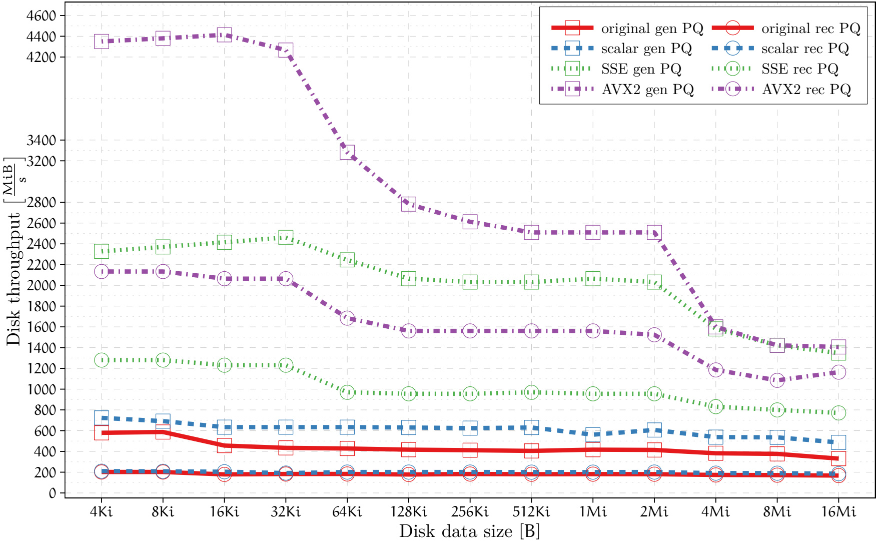
$$C * a = C * (a_l << 4) \oplus C * a_r \quad (5)$$

  - Since a_l and a_r are 4-bit wide $C * a_{l,r}$ have 16 possible solutions
  - Two 16-byte LT are precomputed for the constant
  - Vector shuffle instruction can perform 16 simultaneous table lookups (`mm_shuffle_epi8()`, `mm256_shuffle_epi8()`)
- Vectorized implementation:
  - New **scalar**, **SSE**, and **AVX2** implementations
  - Multiplication by scalar in 2 vector shuffle operation with precomputed LT
  - Streaming loads/stores
  - Selection of the fastest & supported algorithm in runtime

[1] K.Greenan, E.Miller, T.J.Schwartz. Optimizing Galois Field arithmetic for diverse processor architectures and applications. 2008

ZFS RAIDZ2 Results

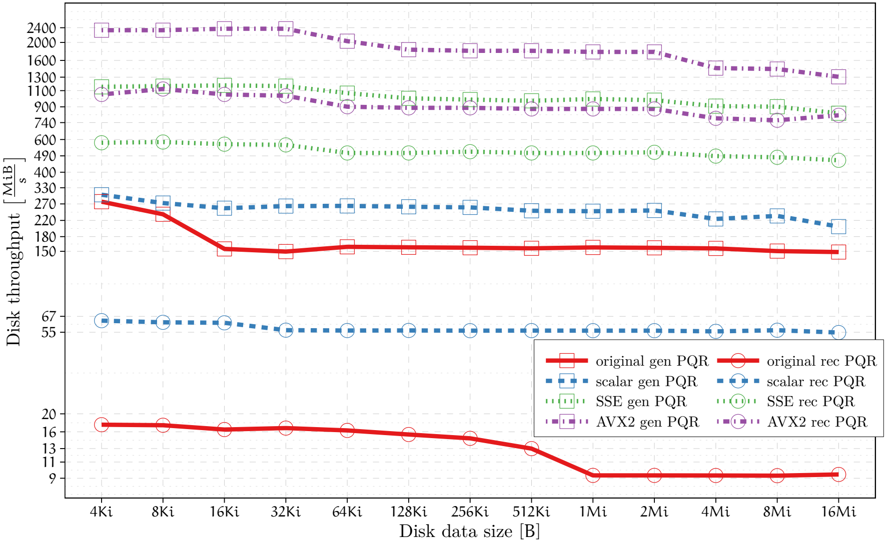
Original, Scalar, SSE, AVX2, 8 Data + 2 Parity disks



Intel(R) Xeon(R) CPU E5-2660v3 @ 2.60GHz, single core

ZFS RAIDZ3 Results

Original, Scalar, SSE, AVX2, 8 Data + 3 Parity disks



Intel(R) Xeon(R) CPU E5-2660v3 @ 2.60GHz, single core

ZFS RAIDZ Results

Speed-up

RAIDZ operation	**scalar**	**SSE**	**AVX2**
Generate P	2.2	2.4	2.6
Reconstruct using P	1.4	2.0	2.2
Generate PQ	1.5	4.1	4.3
Reconstruct using Q	1.5	7.2	8.8
Reconstruct using PQ	1.2	4.7	7.1
Generate PQR	1.4	5.6	8.8
Reconstruct using R	4.8	20.7	32.3
Reconstruct using PR	8.5	43.0	69.1
Reconstruct using QR	5.0	35.5	60.2
Reconstruct using PQR	5.9	50.1	85.8

Table: **Speed-up** relative to the original RAIDZ methods

Conclusion & Future work

- Summary:
  - Faster parity generation
  - Faster missing data recalculation
  - Shorter scrub and resilvering times
  - Increased reliability
  - Decreased system costs
- Future work:
  - Test and verify the implementation[1]
  - Upstream to *ZFS on Linux*

[1] "A program can be made arbitrarily fast if you relax the requirement of correctness." - D. Knuth

The End

Thank you!

Questions?