The True Value of Storage Drives with Built-in Transparent Compression: Far Beyond Lower Storage Cost

Tong Zhang
ScaleFlux Inc.
San Jose, CA
The Rise of Computational Storage

Homogeneous Computing → Heterogenous Computing

Computing
- FPGA/GPU/TPU
- Domain Specific Compute
- End of Moore’s Law

Networking
- SmartNICs
- 10 → 100-400Gb/s

Storage
- Computational
- Fast & Big Data Growth

2020 Storage Developer Conference. © ScaleFlux. All Rights Reserved.
Computational Storage: A Very Simple Idea

- End of Moore’s Law → heterogeneous computing

Low-hanging fruits

Computational Storage Drive (CSD) with Data Path Transparent Compression
ScaleFlux Computational Storage Drive: CSD 2000

- Complete, validated solution
  - Pre-Programmed FPGA
  - Hardware
  - Software
  - Firmware
- No FPGA knowledge or coding
- Field upgradeable
- Standard U.2 & AIC form factors

Multiple, discrete components for Compute and SSD Functions

Single FPGA combines Compute and SSD Functions
CSD 2000: Data Path Transparent Compression

FIO: 4K Random R/W IOPS

- 170% improvement
- 70/30 R/W
- NVMe SSD
- 2.5:1 Compressible Data, 8 jobs, 32 QD, steady state after preconditioning

FIO: 16K Random R/W IOPS

- 220% improvement
- 70/30 R/W
- NVMe SSD
- 100% Writes

2020 Storage Developer Conference. © ScaleFlux. All Rights Reserved.
## Comparing Compression Options

<table>
<thead>
<tr>
<th></th>
<th>No Compression</th>
<th>Host-Based</th>
<th>Offload Card</th>
<th>CSD 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>No CPU Overhead</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reduced $/User GB</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Performance scales with capacity</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Transparent App Integration</td>
<td></td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Zero App Latency</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>No incremental power usage</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>No incremental physical footprint</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Scalable CSD-based compression reduces Cost/GB without choking the CPU**

2020 Storage Developer Conference. © ScaleFlux. All Rights Reserved.
In-Storage Transparent Compression: Why is It Hard to Build?

Flash Translation Layer (w/o compression)

- Relatively simple FTL implementation
- Relatively easy to achieve high speed
- Relatively easy to ensure storage stability

Logical Block Address (LBA) ➔ Flash Memory

4KB LBA ➔ 4KB flash block mapping

Regularity & uniformity
In-Storage Transparent Compression: Why is It Hard to Build?

Flash Translation Layer (w. compression)

4KB LBA $\rightarrow$ variable-length flash block mapping

Irregularity & randomness

- Much more complicated FTL implementation
- Much harder to achieve high speed
- Much harder to ensure storage stability
CSD 2000: Highest OLTP TPS, Lowest $/User GB

- Sysbench (MySQL 5.7.25, InnoDB)
- 50M records, 64 Threads
- 1hr Test run
- Intel(R) Xeon(R) CPU E5-2667 v4 @ 3.20GHz, 256GB DRAM

2.4TB Dataset Physical Flash consumed on NVMe A; 0.9TB on CSD 2000
4.8TB Dataset Physical Flash consumed on NVMe A; 1.6TB on CSD 2000

CSD 2000 delivers 30% higher Read-Write TPS in this cost comparison

Flexible Drive Capacity Enables the Best Performance ↔ Cost
Open a Door for System Innovation

**Logical** storage space utilization efficiency

**Physical** storage space utilization efficiency

Exposed LBA space (e.g., 32TB)

FTL with transparent compression

NAND Flash (e.g., 4TB)

Valid user data

0’s

Compressed data

Unnecessary to use all the LBAs

Unnecessary to fill each 4KB sector with user data

OS/Applications can **purposely waste** logical storage space to gain benefits

2020 Storage Developer Conference. © ScaleFlux. All Rights Reserved.
Case Study 1: PostgreSQL

8KB/page

Data

Fillfactor (FF)

Reserved for future update

Performance

Storage space

Normalized Performance

Commodity SSD

SFX CSD 2000

Transparent compression

Compressed data

8KB/page

Data

0’s

8KB/page

Data

Reserved for future update

Performance

Storage space

Normalized Performance

Commodity SSD

SFX CSD 2000

Transparent compression

Compressed data
Case Study 1: PostgreSQL

<table>
<thead>
<tr>
<th>Fillfactor</th>
<th>Drive</th>
<th>Logical size (GB)</th>
<th>Physical size (GB)</th>
<th>Comp Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Vendor-A CSD 2000</td>
<td>740</td>
<td>740</td>
<td>1.00</td>
</tr>
<tr>
<td>75</td>
<td>Vendor-A CSD 2000</td>
<td>905</td>
<td>905</td>
<td>1.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fillfactor</th>
<th>Drive</th>
<th>Logical size (GB)</th>
<th>Physical size (GB)</th>
<th>Comp Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Vendor-A CSD 2000</td>
<td>1,433</td>
<td>1,433</td>
<td>1.00</td>
</tr>
<tr>
<td>75</td>
<td>Vendor-A CSD 2000</td>
<td>1,762</td>
<td>1,762</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Case Study 2: Sparse Write-Ahead Logging

- Write-ahead logging (WAL)
  - Universally used by data management systems to achieve **atomicity** and **durability**

In-memory WAL buffer
- Commit at t1
- Synthetic write to TRX-1
- Fsinc at t1

On-storage WAL
- Commit at t2
- Synthetic write to TRX-1
- Fsinc at t2

NAND Flash memory
- Transparent compression
Case Study 2: Sparse Write-Ahead Logging

- **In-memory WAL buffer**
  - TRX-1: 0's
  - fsync @ t1
  - Commit @ t1

- **On-storage WAL**
  - LBA x0001
  - TRX-1: 0's
  - fsync @ t1
  - fsync @ t2
  - Commit @ t2

  - LBA x0001
  - TRX-1: 0's
  - Commit @ t3

- **NAND Flash memory**
  - More interference with other IOs 😞
  - Shorter NAND flash memory lifetime 😞

- **Transparent compression**

2020 Storage Developer Conference. © ScaleFlux. All Rights Reserved.
Case Study 2: Sparse Write-Ahead Logging

- Sparse WAL: Allocate a new 4KB sector per transaction commit
  - Waste logical storage space → reduce WAL-induced write amplification

```
In-memory WAL buffer

commit @ t1

TRX-1: 0's

fsync @ t1

On-storage WAL

LBA x0001

TRX-1: 0's

commit @ t2

TRX-2: 0's

LBA x0002

fsync @ t2

commit @ t3

TRX-3: 0's

LBA x0003

NAND Flash memory

... ...

Transparent compression
```

2020 Storage Developer Conference. © ScaleFlux. All Rights Reserved.
Case Study 2: Sparse Write-Ahead Logging

- Sparse WAL: Allocate a new 4KB sector per transaction commit
  - Waste logical storage space \(\Rightarrow\) reduce WAL-induced write amplification

![](chart)

- 94% reduction
Case Study 3: Table-less Hash-based KV Store

- Very simple idea
  - Hash key space directly onto logical storage space \(\rightarrow\) eliminate the in-memory hash table
  - Transparent compression eliminates the “unoccupied space” from physical storage space

- Hash function \(f_K \rightarrow T\)
- In-memory hash table
- Key space \(K\)
- LBA space \(L\)
- KV pairs are tightly packed in \(L\)

- Hash function \(f_K \rightarrow L\)
- Transparent compression
- Unoccupied space
- KV pairs are loosely packed in \(L\)

- NAND Flash
Case Study 3: Table-less Hash-based KV Store

- Eliminate in-memory hash table
  - Very small memory footprint
  - High operational parallelism
  - Short data access data path
  - Very simple code base

- Under-utilize logical storage space
  - Obviate frequent background operations (e.g., GC and compaction)

→ High performance, low memory cost, and low CPU usage
Case Study 3: Table-less Hash-based KV Store

- **Experimental Setup**
  - 24-core 2.6GHz Intel CPU, 32GB DDR4 DRAM, and a 3.2TB SFX CSD2000
  - RocksDB 6.10 (12 compaction threads and 4 flush threads)
  - 400-byte KV pair size, 1 billion KVs → 400GB raw data
  - Memory usage: RocksDB (5GB), KallaxDB (600MB)

<table>
<thead>
<tr>
<th>YCSB</th>
<th>Description</th>
<th>Storage Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>50% reads, 50% updates</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>95% reads, 5% updates</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>100% reads</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>95% reads, 5% inserts</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>50% reads, 50% read-modify-writes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RocksDB (no compression)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RocksDB (LZ4-only)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RocksDB (LZ4+ZSTD)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>KallaxDB</td>
</tr>
</tbody>
</table>
Case Study 3: Experimental Results (24 clients)

Average ops/s:
- RocksDB (no compression)
- RocksDB (LZ4-only)
- RocksDB (LZ4+ZSTD)
- KallaxDB

Average Read Latency (us):
- RocksDB (no compression)
- RocksDB (LZ4-only)
- RocksDB (LZ4+ZSTD)
- KallaxDB

99.9% Read Tail Latency (us):
- RocksDB (no compression)
- RocksDB (LZ4-only)
- RocksDB (LZ4+ZSTD)
- KallaxDB

Ccycle/Op (K):
- RocksDB (no compression)
- RocksDB (LZ4-only)
- RocksDB (LZ4+ZSTD)
- KallaxDB

Better
Open a Door for System Innovation

**Logical** storage space utilization efficiency

- Exposed LBA space (e.g., 32TB)
- FTL with transparent compression
- NAND Flash (e.g., 4TB)

**Physical** storage space utilization efficiency

- Valid user data
- 0’s
- Transparent compression
- Compressed data

**Unnecessary** to use all the LBAs

**Unnecessary** to fill each 4KB sector with user data

OS/Applications can **purposely waste** logical storage space to gain benefits
Open a Door for System Innovation

Reserve more space for future update to improve performance @ zero storage overhead

Sparse WAL  Reduce WAL-induced write amplification @ zero storage overhead

Table-less hash-based KV store

High performance, low memory/CPU usage @ zero storage overhead
Thank You

www.scaleflux.com
info@scaleflux.com
tong.zhang@scaleflux.com