ZBD: Using Transparent Compression at the Block Level to Increase Storage Space Efficiency

Thanos Makatos, Yannis Klonatos, Manolis Marazakis,
Michail D. Flouris, and Angelos Bilas
`{mcatos,klonatos,maraz,flouris,bilas}@ics.forth.gr`

FORTH-ICS Institute of Computer Science
Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)

Motivation

- Disk storage cost per GB declining
  - Capacity demands surpass cost improvements
- Techniques for improving effective capacity
  - Compression, de-duplication
- Benefits
  - Less disks for same capacity → lower cost
    - Simpler packaging → easier management
    - Less components → less HW failures/human failures
    - Less spindles → less power
  - RAID-1, RAID-10 → reduce capacity penalty
  - Versioning → more versions
- Compression to reduce capacity requirements *online*

Who manages compressed volumes?

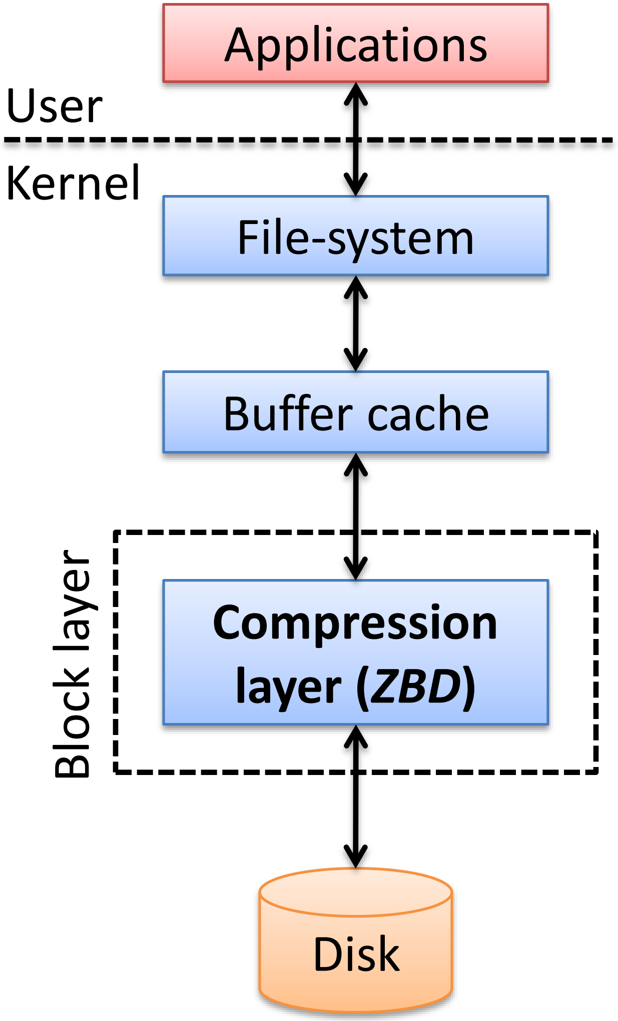
- File-system
  - Restricts FS choice
    - What about ext3, ext4, XFS, reiser3, JFS?
  - Doesn't support raw I/O databases
  - Restricts where compression is applied in the I/O path
    - Storage controllers?
    - Storage virtualization layers?
- Our approach: move compression at the **block level**
  - Addresses above concerns

Related Work

- FS compression
  - Sprite LFS, NTFS, ZFS, BTRFS
- Block-level compression
  - CBD, cloop: read-only block devices (avoid most complexity)
- Reduce DRAM requirements by compressing memory pages
- Improve I/O performance by compression
  - Compression increases effective disk bandwidth:
    - Mostly used in DBMS (Oracle, IBM's IMS)
    - Implemented at the DBMS level: specifically targets DB
  - Compress SSD caches → improve effective cache capacity

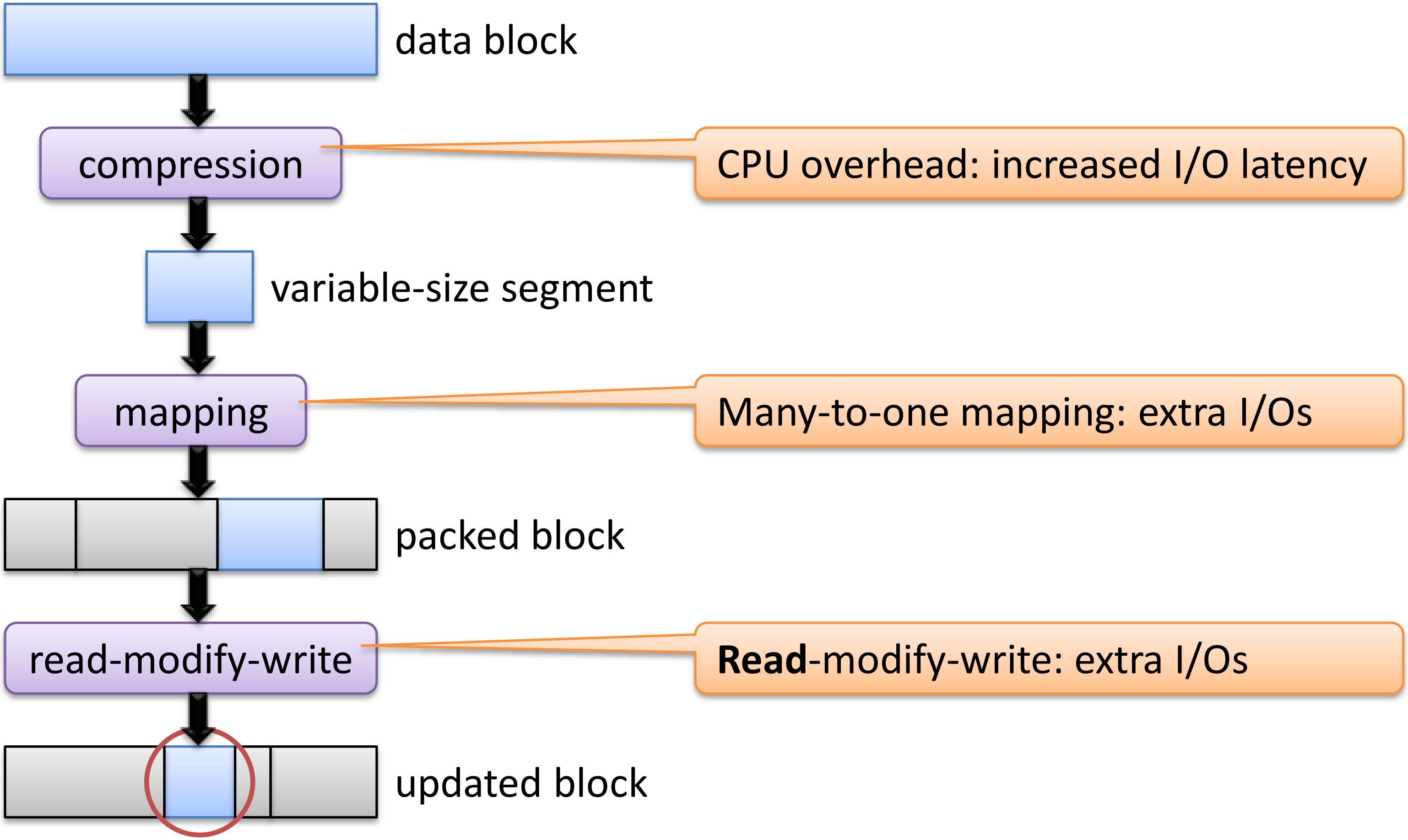
T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas,
"Using Transparent Compression to Improve SSD-based I/O Caches", EuroSys 2010

Compression in the I/O path



- All I/Os affected
  - Writes compressed
  - Reads decompressed
- We build "ZBD"
  - A Linux virtual block device (/dev/zbd)
  - Intercepts and compresses I/Os
  - Can be placed **anywhere** between the FS and the disk
- Trades multicore CPU cycles for disk capacity

Challenges



CPU overhead: increased I/O latency

Many-to-one mapping: extra I/Os

Read-modify-write: extra I/Os

Outline

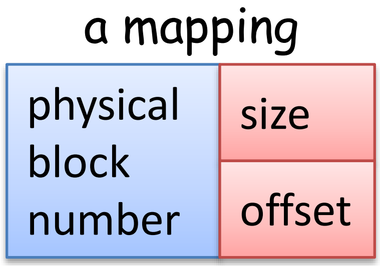
- Motivation & Challenges
- Design
  - CPU overhead & I/O Latency
  - Increased Number of I/Os
    - Metadata
    - Read-modify-write
    - Cleaner
- Evaluation
  - Overall impact on performance and CPU utilization
  - System and workload parameters
- Conclusions

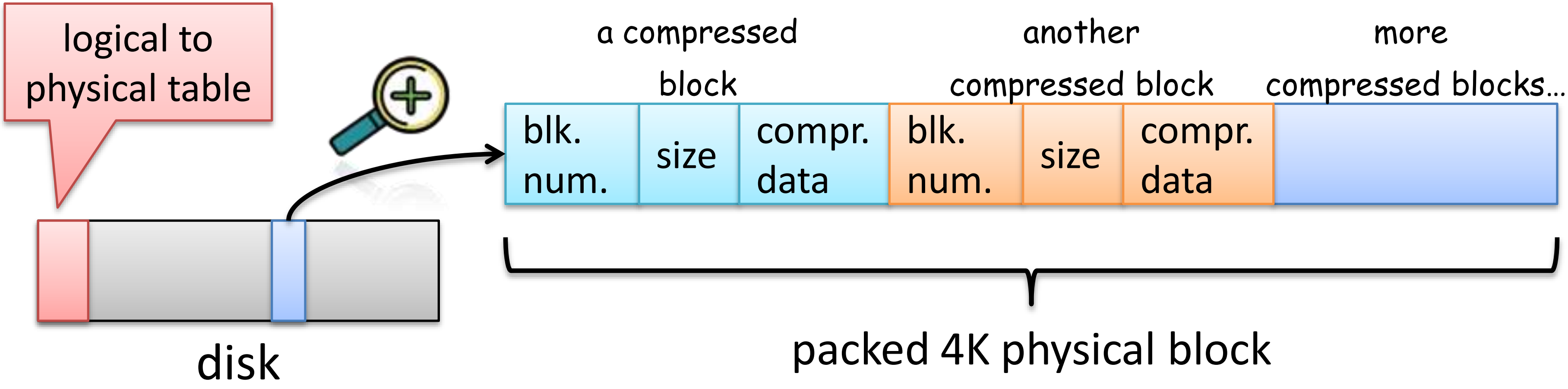
Hiding compression overhead

- Compression requires a lot of CPU (+2.4 ms for 64K of data)
  - Decompression 3x faster
  - Our design **agnostic** to **compression method**
- High I/O concurrency (many independent I/Os)
  - Need to **load balance** requests across cores with low overhead
  - Use two global work-queues
    - One for reads (high priority)
    - One for writes
- Low I/O concurrency
  - Small I/Os *doomed*: can’t hide decompression overhead
  - Large I/Os more interesting:
    - Large I/Os split to 4K blocks
    - Processed *in parallel* by multiple cores

Reducing I/Os

- Need metadata to locate segment within physical block
  - Conceptually a *logical-to-physical* translation table (**L2P**)
- Translation metadata split to *two* levels
  - 1^{st} level stored in beginning of disk
    - Too big to fit in DRAM (2GB per TB), use a *cache*
  - 2^{nd} level stored in physical block
    - Remove size & offset fields (decrease memory footprint)





Reducing I/Os

- Dirty metadata blocks placed on NVRAM
  - Avoid sync. metadata writes
  - Used only for *pending* metadata writes
  - Only need few tens of MB’s
- 4K blocks too small
  - Free space fragments
  - Combine multiple physical blocks into *extents* (e.g. 32K)
  - **Unit of I/O** → affects I/O volume & fragmentation
- **Read**-modify-write: +1 read for every write!
  - Choose any suitable extent in DRAM (*remap-on-write*)
  - Avoids complexity of compressed footprint mismatch

Reducing I/Os

- Extents managed by *extent pool*
  - Full extents flushed to disk sequentially
- Pool design tradeoff
  - Fragmentation
  - Preserve temporal locality
    - Blocks of *same* request placed on *same* extent
    - Blocks of requests close in time to *same* extent
    - Otherwise we introduce *seeks*…
- Pool replenished with empty extents
  - Empty/non-empty → “bitmap” for free extents
  - Less metadata

Reducing I/Os

- **Allocator** replenishes extent pool
  - Free list in memory
  - Allocator returns *any* extent when called (*fast*)
  - List requires replenishing
- **Garbage collector** (cleaner) reclaims space/replenishes list
  - Triggered when few free extents left (low/high watermarks)
  - Scans & compacts old extents
  - Places empty extents in free list
  - Read-modify-write *deferred* to garbage collection time
  - Expected to take place during **idle** I/O

Outline

- Motivation & Challenges
- Design
  - CPU overhead & I/O Latency
  - Increased Number of I/Os
    - Metadata
    - Read-modify-write
    - Cleaner
- Evaluation
  - Overall impact on performance and CPU utilization
  - System and workload parameters
- Conclusions

Experimental evaluation

- Platform
  - Dual-socket, quad-core Intel XEON, 2 GHz, 64 bit (**8 cores** total)
  - 8 SATA-II disks, 500 GB (WD-5001AALS)
  - Areca SAS storage controller (ARC-1680D-IX-12)
  - RAID0 configuration, 64KB chunks
  - Linux kernel 2.6.18.8 (x86_64), CentOS 5.3
- Benchmarks
  - PostMark (mail server)
  - SPECsfs2008 (file server)
  - TPC-C (OLTP)
  - TPC-H (data-warehouse)
- zlib, lzo compression libraries
  - Compression ratio between 11%-54% (depending on method and data)

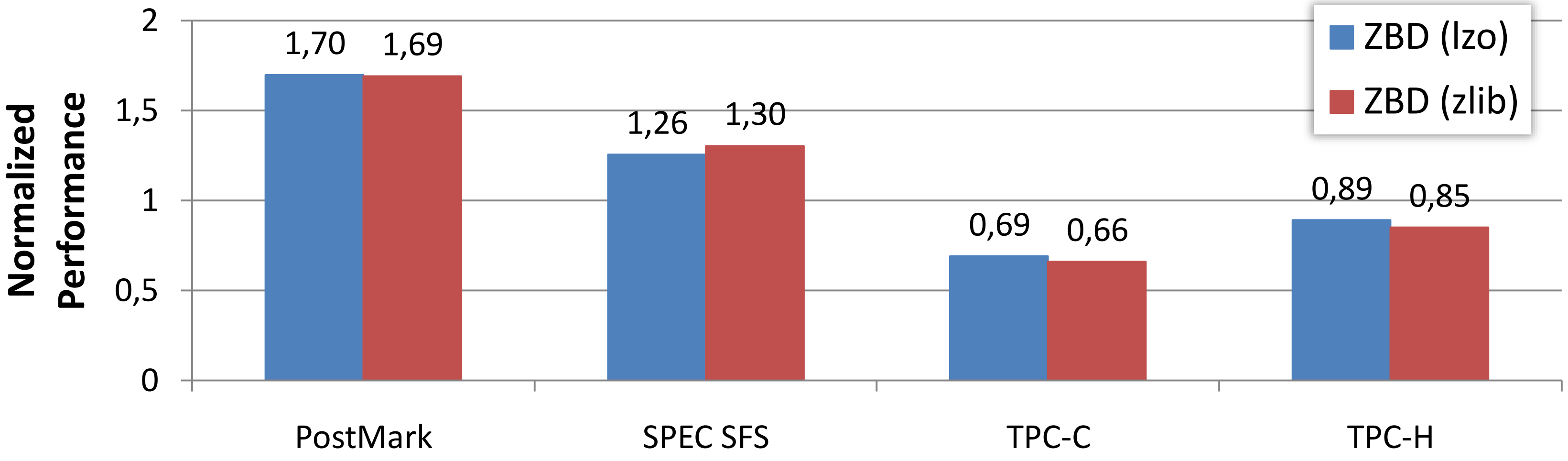
	Read (Decompression)	Write (Compression)	Resp. Time (4K block)
Disk (1 spindle)	100 MB/s	90 MB/s	12.6 ms
zlib (1 core)	65 MB/s	26 MB/s	N/A
lzo (1 core)	279 MB/s	85 MB/s	N/A

Compression Efficiency

Files	Orig. MB	gzip -r	gzip .tar	NTFS	ZFS	ZBD (zlib)	ZBD (lzo)
mbox 1	125	N/A	29%	7%	4%	17%	11%
mbox 2	63	N/A	68%	39%	31%	54%	34%
MS word	1100	50%	51%	37%	35%	44%	33%
MS excel	756	67%	67%	47%	41%	55%	47%
PDF	1400	22%	22%	14%	15%	15%	12%
Linux source	277	55%	76%	27%	33%	69%	46%
compiled	1400	63%	71%	47%	52%	67%	58%

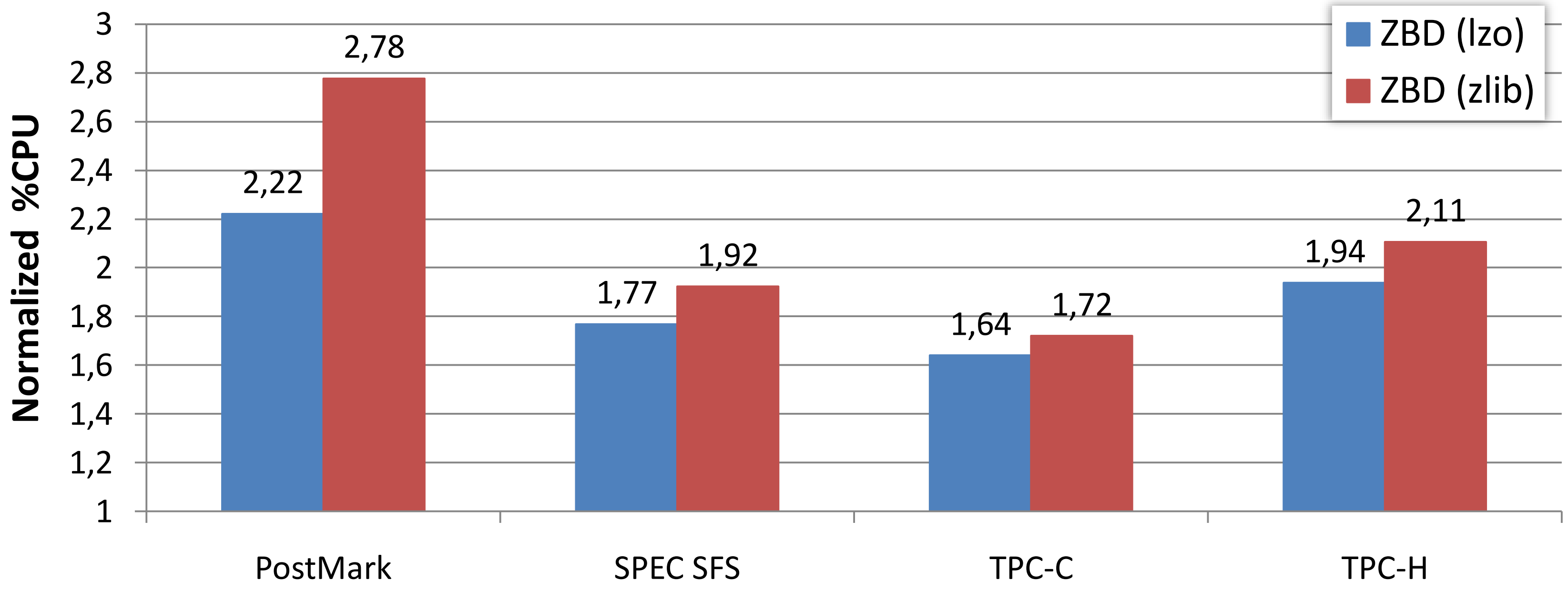
- Higher is best (percentage of data saved)
- Comparable space savings to NTFS, ZFS
  - zlib slightly *better*
  - lzo slightly *worse*

Overall Impact on Performance



- Performance **improves** by 70% for PostMark, 30% for SPEC SFS
  - Mainly due to log-structured writes
  - Compression reduces write I/O volume → performance further improves
- Performance **degrades** by 34% for TPC-C, 15% for TPC-H
  - TPC-C: (a) read-intensive **and** (b) poor spatial locality → excessive read I/O volume
  - TPC-H: (a) read-only **and** (b) low I/O concurrency **and** (b) small I/Os → decompression cost exposed

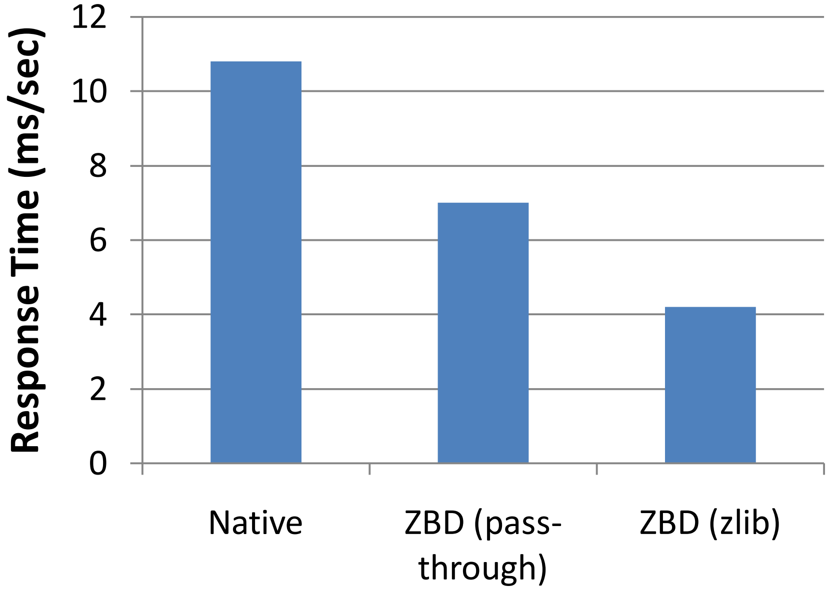
Impact on CPU Utilization

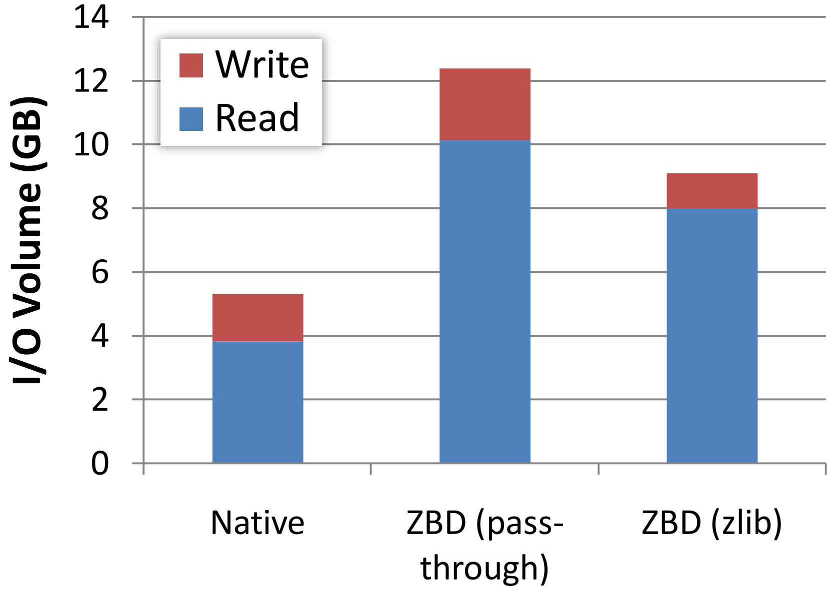


Increased capacity isn’t for free (1-2 additional cores consumed)

- PostMark: 122%-178%
- SPEC SFS: 77%-92%
- TPC-C: 64%-72%
- TPC-H: 94%-111%

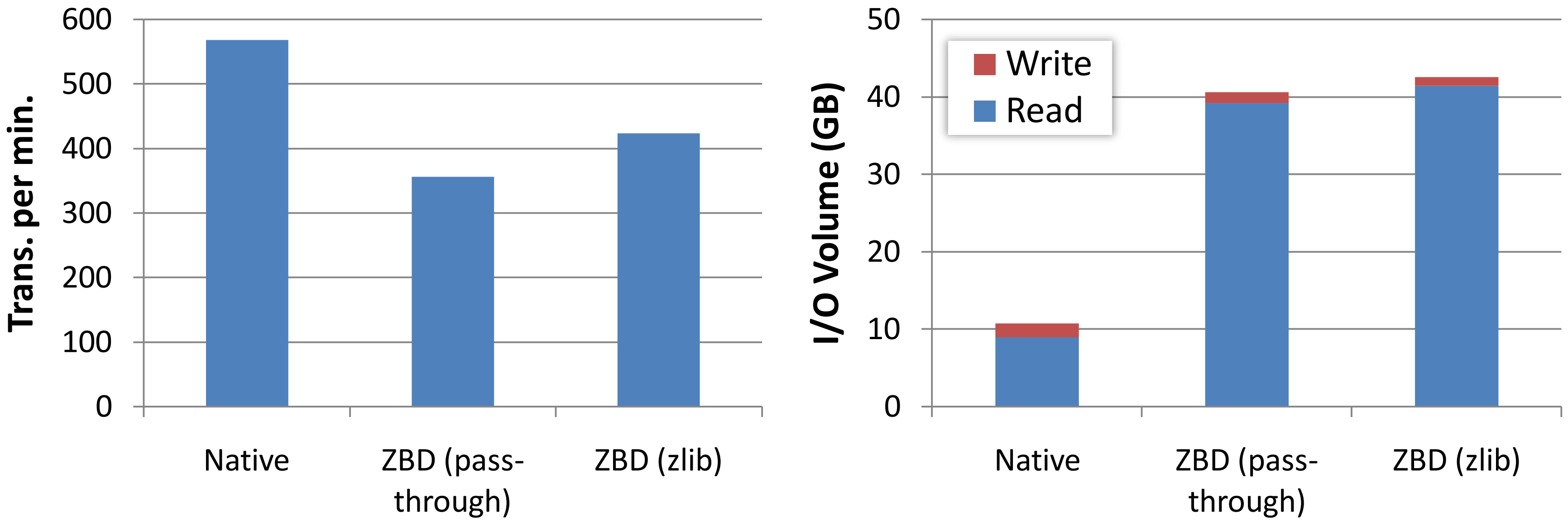
Effect of Log-structured Writes on Performance – SPEC SFS





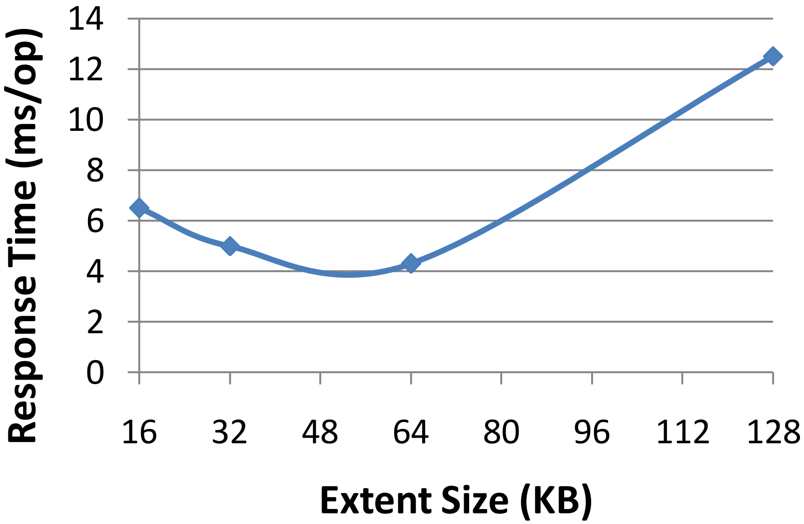
- ZBD in pass-through mode → compression omitted
- Log-structured writes
  - Higher performance due to write I/O volume reduction
  - Additional read I/O volume is *offset*

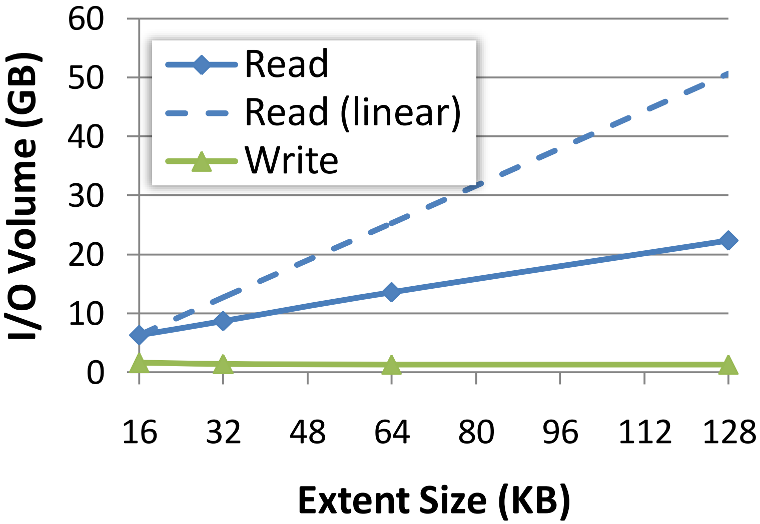
Effect of Log-structured Writes on Performance – TPC-C



- ZBD in pass-through mode → compression omitted
- Higher R:W ratio than SPEC SFS → less writes to optimize
- Each app. read (4K) fetches *entire* extent (32K) → 4x read I/Os
- Compression to reduce read I/Os?
  - 4K blocks change application locality
  - Need *extents*

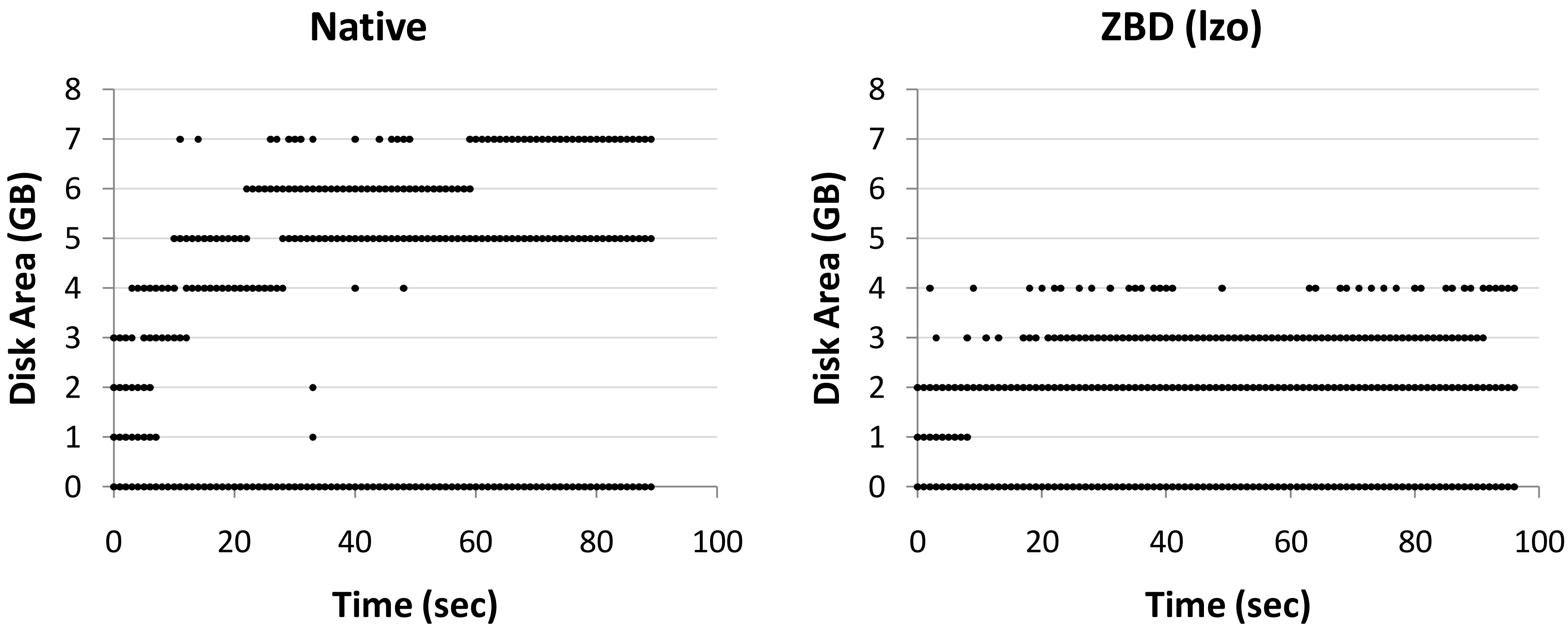
Extent Size - SPEC SFS





- Read I/Os increase with larger extents
  - Sequential I/Os, no seeks introduced
  - Medium extents provide *pre-fetching*, offset extra read I/O volume
- Write I/Os marginally decrease → less free space
- Extent size depends on workload (32K extents used so far)
  - 16K-64K good for most workloads

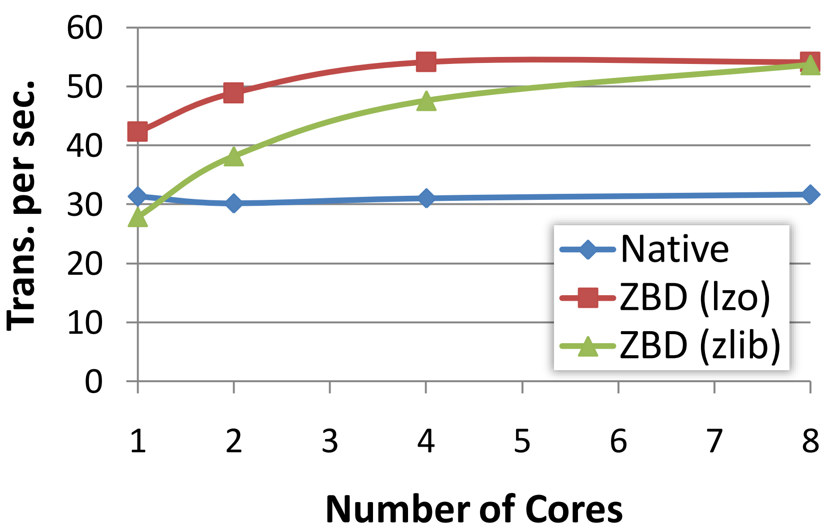
Impact on Access Pattern – TPC-H (Q3)

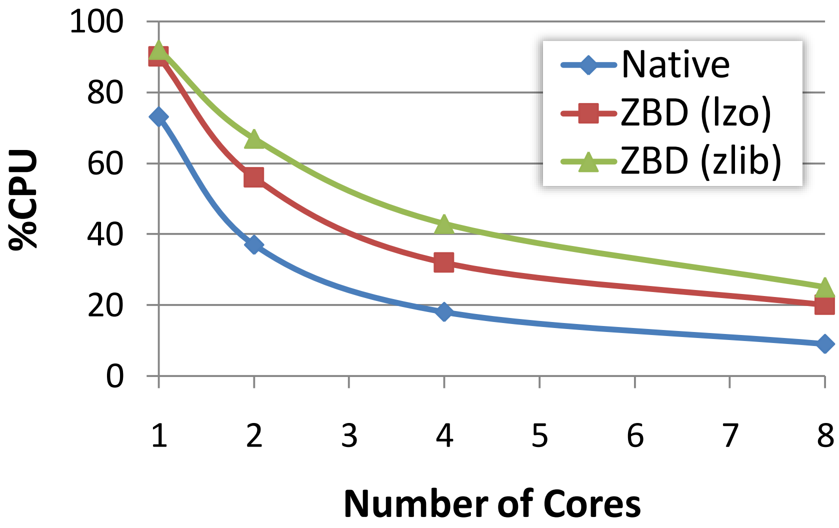


- Data *compacted* on smaller disk zone (4GB vs. 7GB)
  - Smaller seek distance
- Smaller read I/O volume
  - Lower transfer time

Not enough to offset decompression overhead!

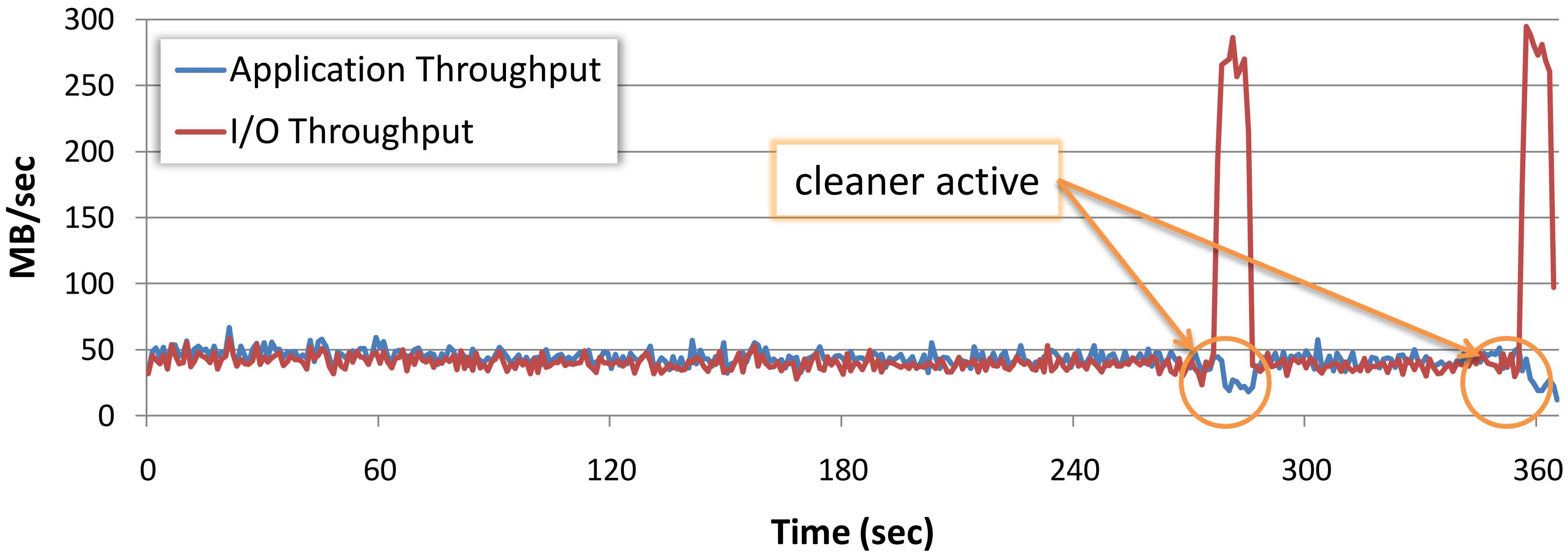
Exploiting Multicores - PostMark





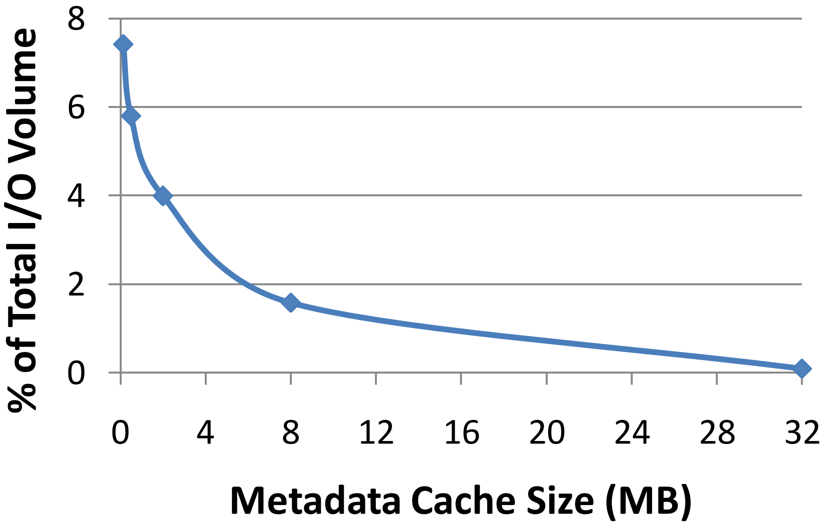
- 1 core → compression **CPU bound**, lzo more light-weight
- 2 cores → performance better than native
- 4 cores → lzo doesn't scale beyond → **disk bound**
- 8 cores → both lzo and zlib disk bound
- PostMark low concur. & response-time bound → no linear scaling

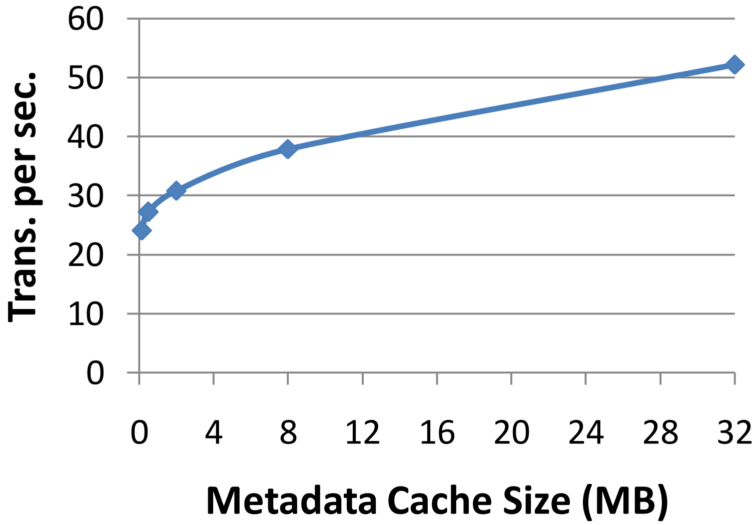
Effect of Cleanup on Performance - PostMark



- Free extents depleted, cleaner on the rescue
  - Cleaner ”steals” IOPS → PostMark throughput **decreases** by 50%
  - Large cleaner I/Os → device I/O activity **increases** by 450%
- Two time periods (“valleys”): 280-290, 355-370 sec
- 15% of capacity reclaimed (1.4 GB) in 1 min.

Metadata I/Os - PostMark





- No metadata I/Os for previous experiments (except for SPEC SFS)
- 2x improvement when no metadata I/Os
- 32MB of DRAM for a 24GB file-set
- Practically *random, blocking* I/Os interfering with app. I/Os
- Similar observations for rest of benchmarks (64KB – 256MB)

Conclusions

- Compress data at the block level for increased space efficiency
  - Transparent to FS and raw I/O apps.
  - Trade CPU cycles for storage capacity
- Transparent compression challenges:
  - Increased I/O response time (compression cost)
  - Increased number of I/Os (metadata & read-modify-write sequence)
- Performance **improves** by 70% in PostMark, 30% in SPECsfs2008
  - Log-structured writes & reduced write I/O volume
- Performance **degrades** by 34% in TPC-C, 15% in TPC-H
  - Small **and** random I/Os → excessive read I/O volume
  - Poor I/O concurrency **and** small I/Os → decompression cost exposed
- Potential in *increasing* I/O performance on disks
  - Reduced transfer time
  - Reduced seek distance

Thank You!

Questions?

"*ZBD*: Using Transparent Compression at the Block Level to Increase Storage Space Efficiency"

Thanos Makatos, Yannis Klonatos, Manolis Marazakis,
Michail D. Flouris, and Angelos Bilas
`{mcatos,klonatos,maraz,flouris,bilas}@ics.forth.gr`

FORTH-ICS
Institute of Computer Science

Foundation for Research & Technology - Hellas
http://www.ics.forth.gr/carv/scalable

Benchmark Parameters

- PostMark (mail server)
  - 50K transactions, 35%:65% RW ratio, 16K accesses
  - Record 5 min. of execution
  - 100 mboxes, 500 msgs/mbox, 4K-1M msg size, 24 GB file-set
- SPECsfs2008 (file server)
  - 3,400-4,600 ops/sec, 300 step value, 540 GB file-set
- TPC-C (OLTP)
  - 300 warehouses (28 GB database), 3,000 connections, 10 terminals per warehouse, execution time set to 30 min.
- TPC-H (data-warehouse)
  - 4 GB database (+2.5 for indices)
  - Queries executed back-to-back (1, 3, 4, 6, 7, 10, 12, 14, 15, 19, and 22)