Using Transparent Compression to Improve SSD-based I/O Caches

- 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

- I/O performance an important problem today
- NAND-Flash SSDs emerge as mainstream storage component
  - Low read response time (no seeks), high throughput, low power
  - Compared to disk low density, high cost per GB
  - No indication of changing trends
- Disks not going away any time soon [Narayanan09]
  - Best medium for large capacities
- I/O hierarchies will contain mix of SSDs & disks
- <u>SSDs have potential as I/O caches</u> [Kgil08]

[Narayanan09] D. Narayanan et al., “Migrating server storage to SSDs:Analysis of tradeoffs”, EuroSys 2009

[Kgil08] T. Kgil et al., "Improving NAND Flash Based Disk Caches“, ISCA 2008

Impact of SSD cache size

- (1) … on cost
  - For given I/O performance, smaller cache reduces system cost
  - System with 4x SSDs, 8x disks → removing two SSDs saves 33% of I/O devices cost
- (2) … on I/O performance
  - For given system cost, larger cache improves I/O performance
- Can we increase effective SSD-cache size?

Increasing effective SSD cache size

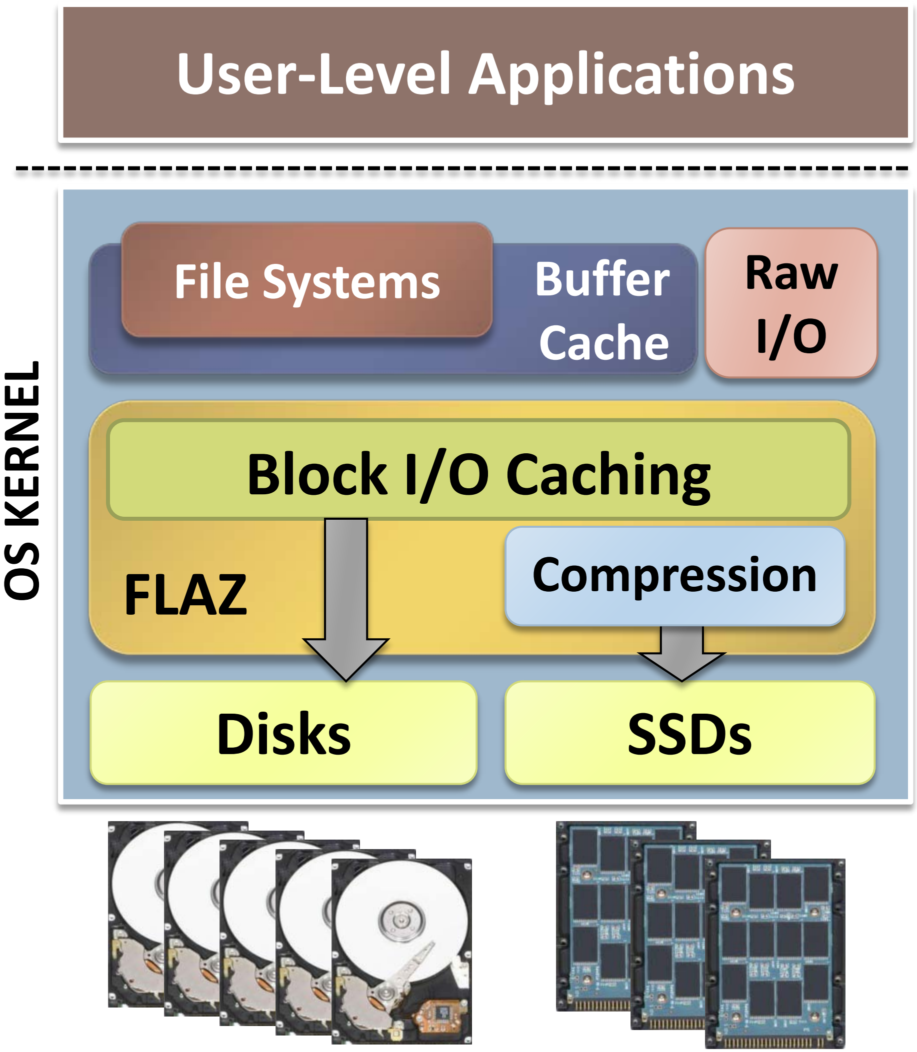
1. Use MLC (multi-layer cell) SSDs
   - Stores two bits per NAND cell, doubles SSD-cache capacity
   - Reduces write performance (higher miss penalty)
   - Increases failure rate
   - Device-level approach
2. Our approach: compress SSD cache online
   - System-level solution
   - Orthogonal to cell density

Who manages the compressed SSD cache?

- Filesystem
  - Requires FS ➔ does not support raw I/O databases
  - Restricts choice of FS
  - Cannot offload to storage controller
- Our approach: move **management at block level**
  - Addresses above concerns
  - Similar observations for SSDs by others [Rajimwale09]

[Rajimwale09] A.Rajimwale et al., "Block Management in Solid-State Devices", Usenix ATC 2009

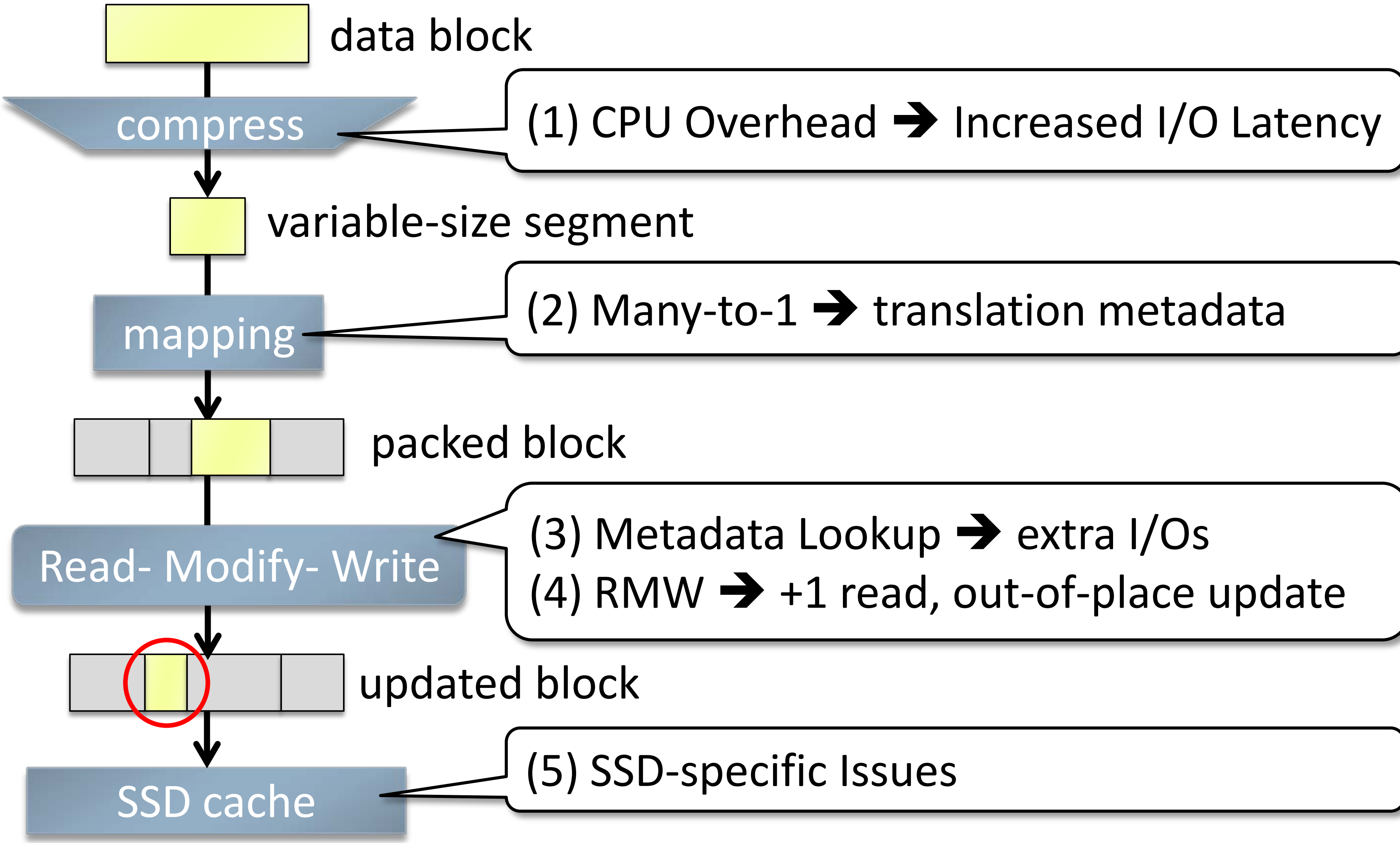
Compression in common I/O path!



- Most I/Os affected
- Read hits require decompression
- All misses and write hits require compression

- We design **"FlaZ"**
- Trades (cheap) multi-core CPU cycles for (expensive) I/O performance…
- <u>**…after we address all related challenges!**</u>

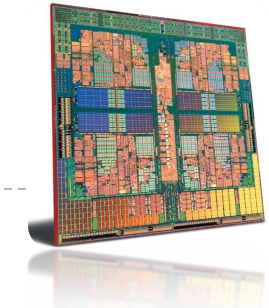
Challenges


data block
compress
(1) CPU Overhead ➔ Increased I/O Latency
variable-size segment
mapping
(2) Many-to-1 ➔ translation metadata
packed block
Read- Modify- Write
(3) Metadata Lookup ➔ extra I/Os
(4) RMW ➔ +1 read, out-of-place update
updated block
SSD cache
(5) SSD-specific Issues

Outline

- Motivation
- Design - Addressing Challenges
  1. CPU overhead & I/O latency
  2. Many-to-one translation metadata
  3. Metadata lookup
  4. Read-modify-write
     - Fragmentation & garbage collection
  5. SSD-specific cache design
- Evaluation
- Related work
- Conclusions

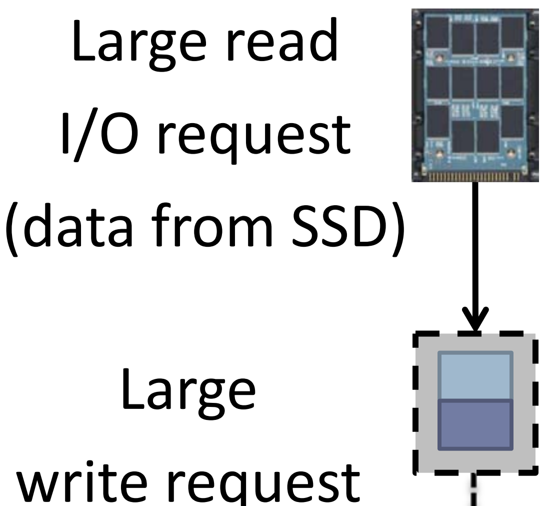
(1) CPU Overhead & I/O Latency

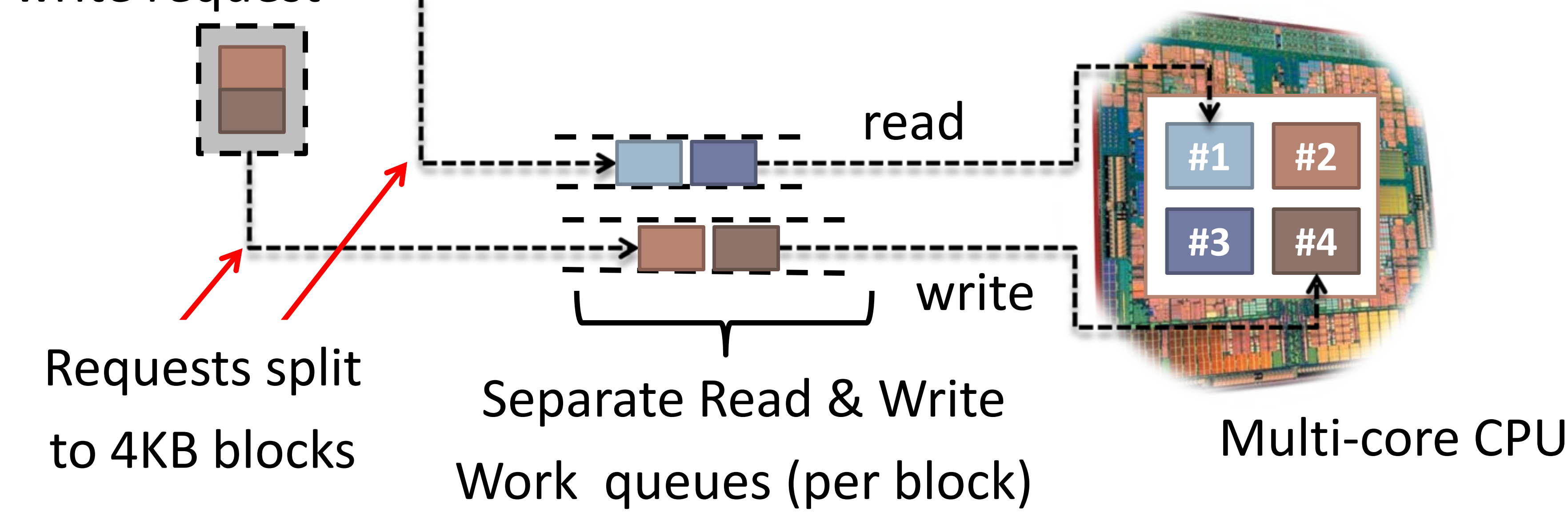


- Compression requires a lot of CPU cycles
  - zlib compress = 2.4 ms for 64KB data, decompress 3x faster
  - CPU overhead varies with workload, compression method
  - Our design is **agnostic to compression method**
- At high I/O concurrency ➔ many independent I/O requests
  - Need to **load balance requests** across cores with low overhead
  - We use global work-queues
  - Scheme scales with number of cores
- Low I/O concurrency, small I/Os problematic
  - May suffer from increased response time due to compression overhead when they hit in SSD cache
- Low I/O concurrency, but with large I/Os more interesting

Load-balancing & I/O Request Splitting

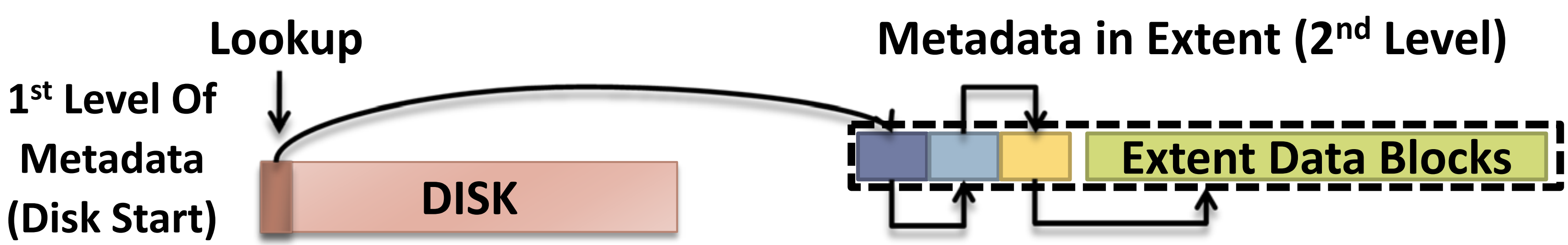
- Blocks of same large I/O request processed in parallel on all CPUs
- All blocks placed on two global queues: (1) read, (2) writes
- Reads have priority over writes (blocking operations)





(2) Many-to-one Translation Metadata

- Block devices operate with fixed-size blocks
- We use a fixed-size extent as the physical container for compressed segments
  - Extent is unit of I/O to SSD, equals cache-line size, typically a few blocks (e.g. 64KB)
  - Extent size affects fragmentation, I/O volume, and is related to SSD erase block size
- Multiple segments packed in single extent in append-only manner
- Need metadata to locate block within extent
  - Conceptually logical to physical translation table
- Translation metadata split to two levels
  - First level stored in beginning of disk ➔ 2.5 MB per GB of SSD
  - Second level stored in extent as list ➔ overhead mitigated by compression
- Additional I/Os only from access to logical-to-physical map
- Placement of L2P map addressed by metadata cache



(3) Metadata Lookup

- Every read/write requires metadata lookup
  - If metadata fits in memory, lookup is cheap
  - However, we need 600MB metadata for 100GB SSD, too large to fit in RAM
- Metadata lookup requires additional read I/O
- To reduce metadata I/Os we use a **metadata cache**
  - Fully-set-associative, LRU, write-back, cache-line size 4KB
- Required cache size
  - Two-level scheme minimizes size of metadata that require caching
  - 10s of MB of cache adequate for 100s of GB of SSD (depends on workload)
  - Metadata size scales with SSD capacity (small), not disk (huge)
- Write-back avoids synchronous writes for updates to metadata
  - But, after failure cannot tell if latest version of block in cache or disk
  - *Needs* **write-through SSD cache**, data always written on disk
  - After failure, start with cold SSD cache
- Design optimizes failure-free case (after clean shutdown)

(4) **Read**-Modify-**Write** Overhead

- Write of R-M-W cannot always be performed in place
  - Perform **out-of-place updates** in any extent with enough space
  - We use **remap-on-write**
- Read of R-M-W requires extra read for every update
  - Remap-on-write allows selecting any suitable extent in RAM
- We maintain a pool of extents in RAM
  - Pool contains small number of extents, e.g. 128
  - Full extents are flushed to SSD sequentially
  - Pool design addresses tradeoff between maintaining temporal locality of I/Os and reducing fragmentation
- Extent pool replenished only with empty extents (**allocator**)
- Part of old extent becomes garbage (**garbage collector**)

Allocator & Garbage Collector

- **Allocator** called frequently to replenish the extent pool
  - Maintains small free list in memory, flushed at system shutdown
  - Free list contains only completely empty extents
  - Allocator returns any of these extents when called → fast
  - Free list requires replenishing
- **Garbage collector** (cleaner) reclaims space and replenishes list
  - Triggered by low, high watermarks for allocator free list
  - Starts from any point on SSD
  - Scans & compacts partially-full extents → generates **many sequential** I/Os
  - Places completely empty extents in free list
- Free space reclaimed mostly **during idle I/O periods**
  - Most systems exhibit idle I/O periods
- Both remap-on-write and compaction change **data layout** on SSD
  - Less of an issue for SSDs vs. disks

(5) SSD-specific Cache Design

- SSD cache vs. memory cache
  - Larger capacity
  - Behave well for reads and *large* writes only
  - Expected benefit from many reads after write for same block…
  - … vs. any combination of reads/writes
  - Persistent vs. volatile
- Our design
  - Large capacity ➔ direct-mapped (smaller metadata footprint)
  - Large writes ➔ large cache-line (extent size)
  - Desirable many reads after write ➔ we do not optimize for this
    - We always write to both disk and SSD (many SSD writes)
    - Alternatively, we could selectively write to SSD by predicting access-pattern
  - Persistence ➔ use persistent cache metadata (tags)
    - Could avoid metadata persistence, if cache cold after clean shutdown
  - Write-through, cache cold after failure

Outline

- Motivation
- Design - Addressing Challenges
  1. CPU overhead & I/O latency
  2. Many-to-one translation metadata
  3. Metadata lookup
  4. Read-modify-write
     - Fragmentation & garbage collection
  5. SSD-specific cache design
- Evaluation
- Related work
- Conclusions

Evaluation

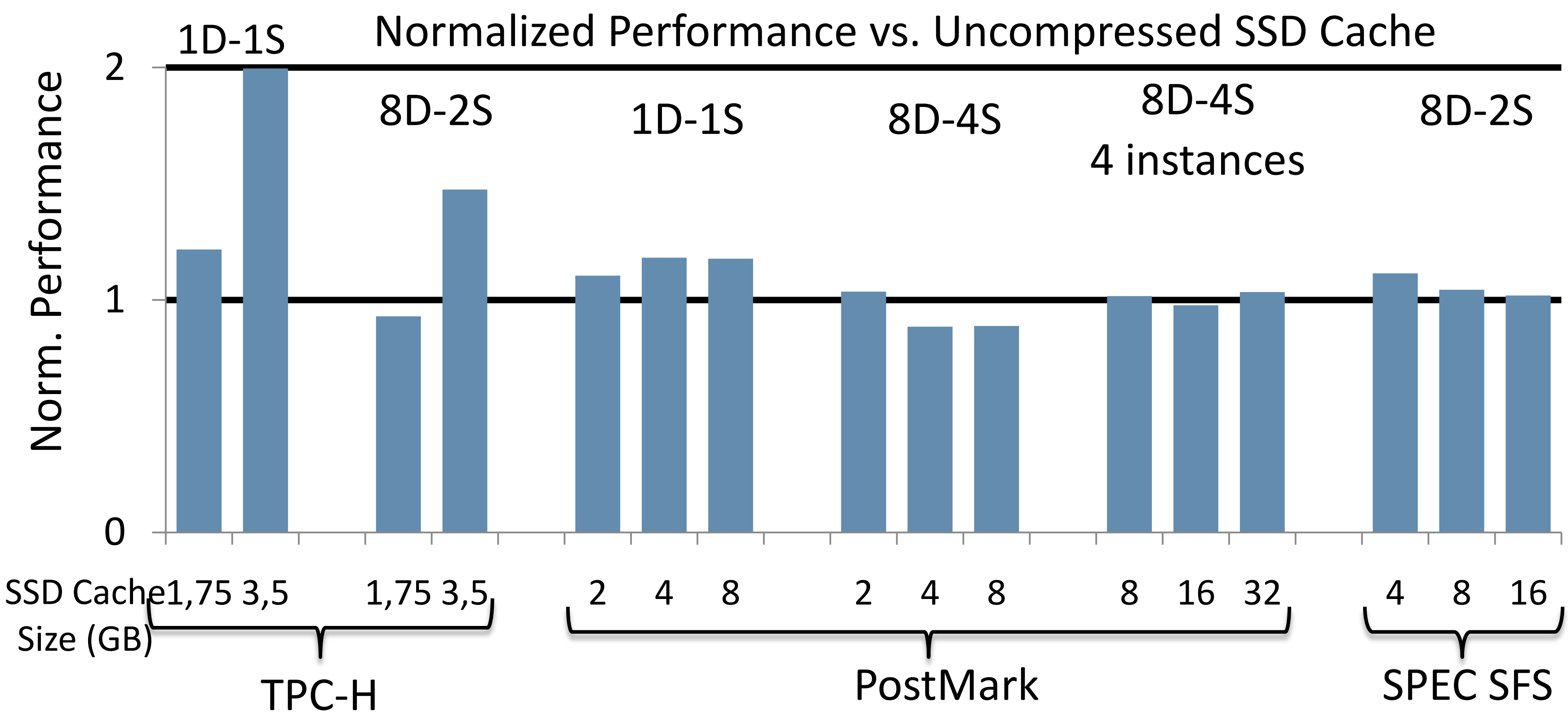
- Platform
  - Dual-socket, Quad-core Intel XEON, 2 GHz, 64 bit (8 cores total)
  - 8 SATA-II disks, 500 GB (WD-5001AALS)
  - 4 SLC SSDs, 32 GB (Intel X25-E)
  - Areca SAS storage controller (ARC-1680D-IX-12)
  - Linux kernel 2.6.18.8 (x86_64), CentOS 5.3
- Benchmarks
  - PostMark (mail server)
  - TPC-H (data-warehouse): Q3,11,14
  - SPECsfs2008 (file server)
  - Compressible between 11%-54% (depending on method and data)
- System configurations
  - 1D1S, 8D4S, 8D2S
  - Both LZO and zlib compression
- We scale down workloads and system to limit execution time

	Read MB/s	Write MB/s	Resp (ms)
HDD	100	90	12.6
SSD	277	202	0.17

We examine

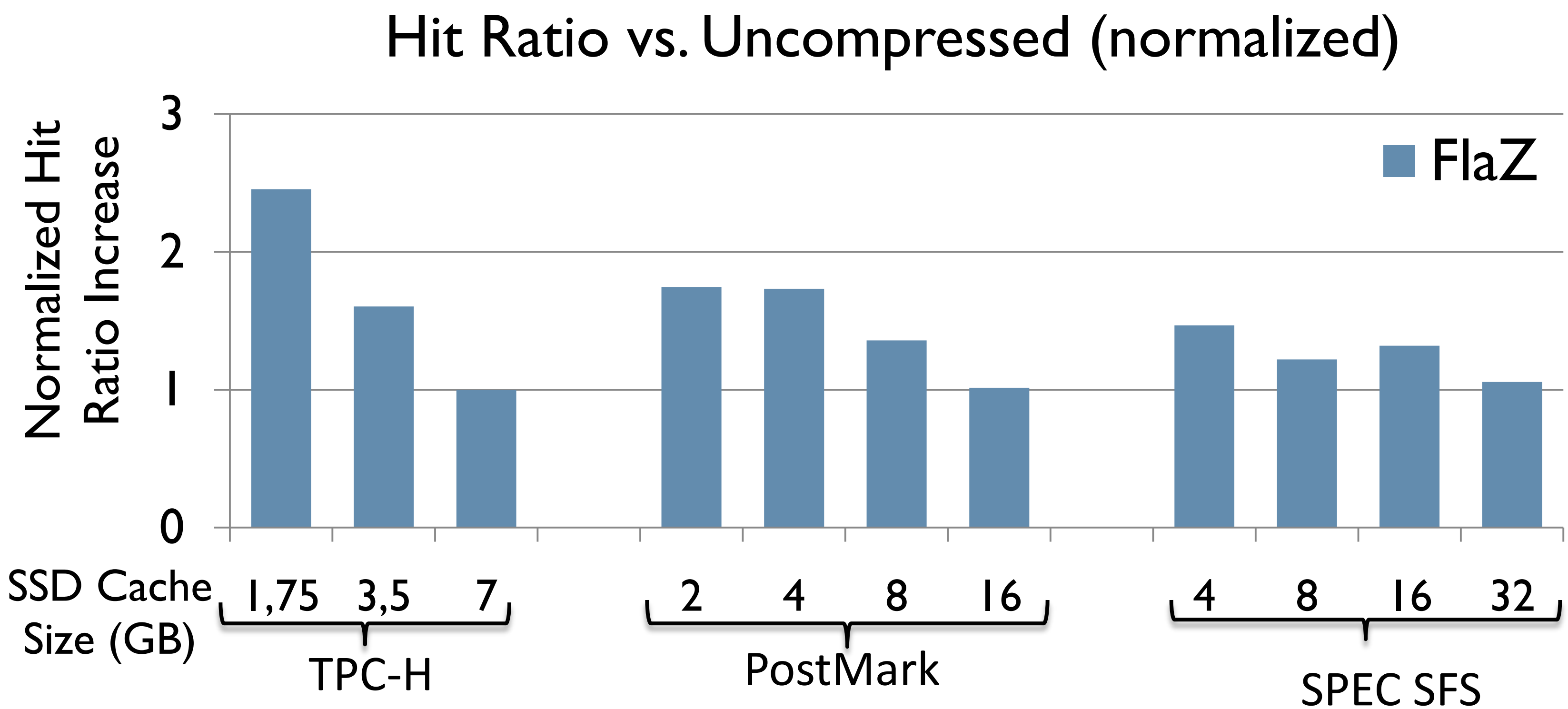
- Overall impact on application I/O performance
  - Cache hit ratio
  - CPU utilization
- Impact of system parameters
  - I/O request splitting
  - Extent size
- Garbage collection overhead

Overall impact on application I/O performance



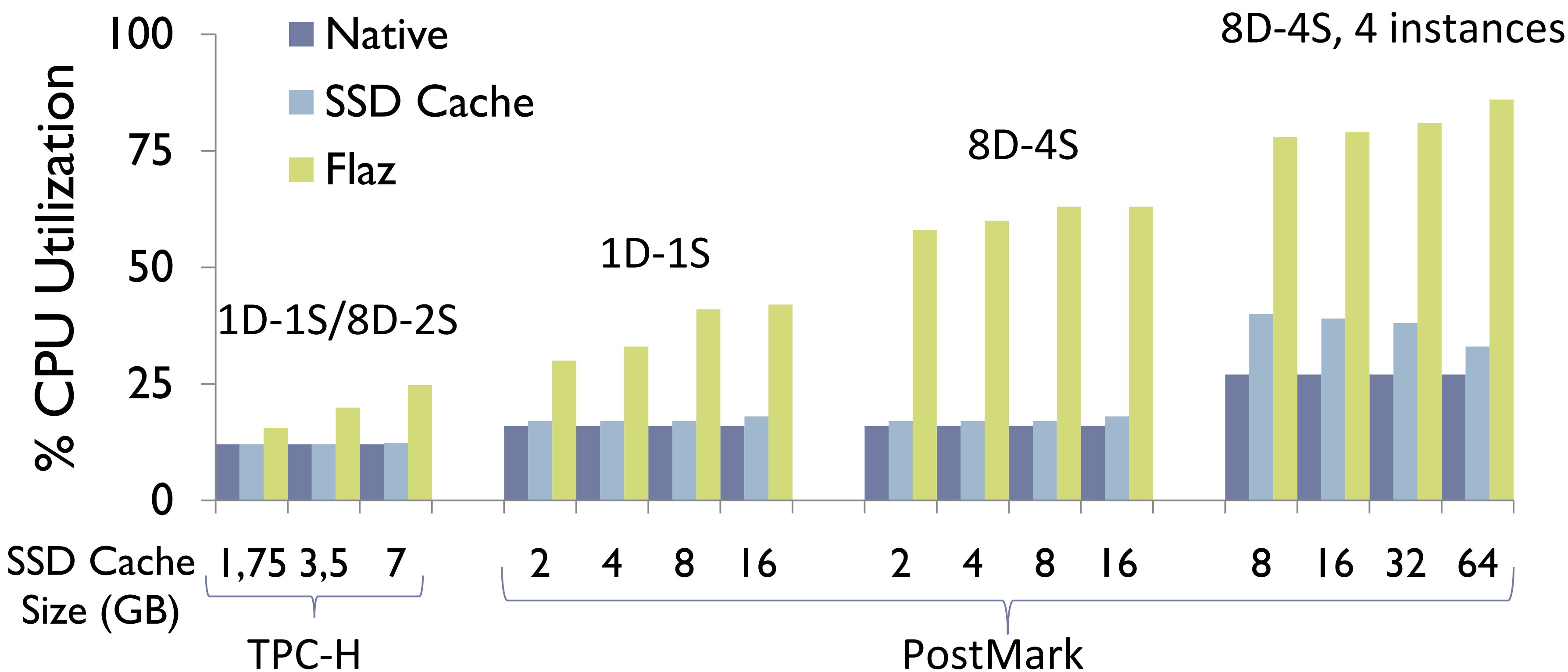
- All configurations between 0%-99% improvement, except for degradation in
  - Single-instance Postmark: 6%-15%, due to (a) low concurrency **and** (b) small I/Os
  - 4-instance Postmark: 2% at 16 GB cache
  - TPC-H 7% in 8D-2S/small cache

Impact on cache hit ratio



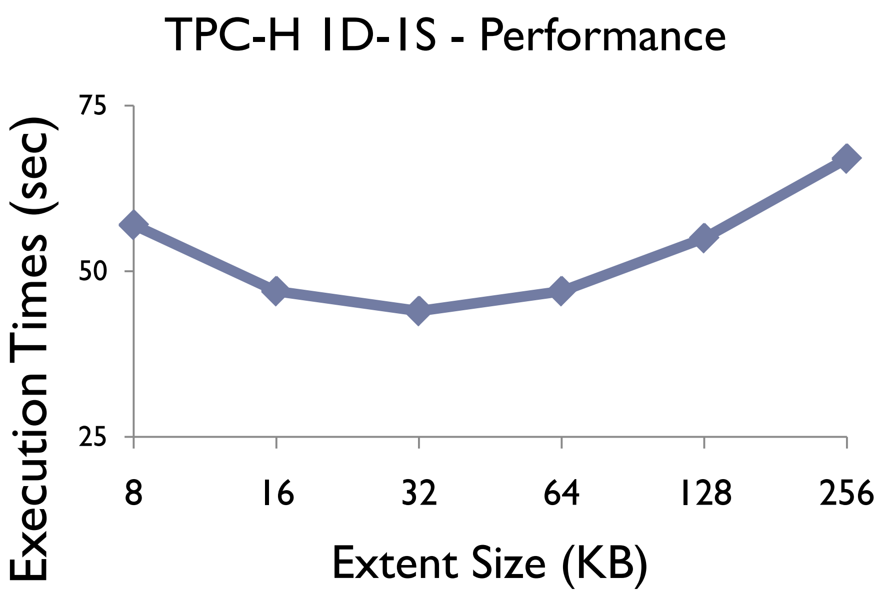
- Normalized increase of SSD Cache hit ratio vs. uncompressed
- TPC-H: Up to 2.5x increase in hit ratio
- Postmark: Up to 70% increase, SPEC SFS: Up to 45%

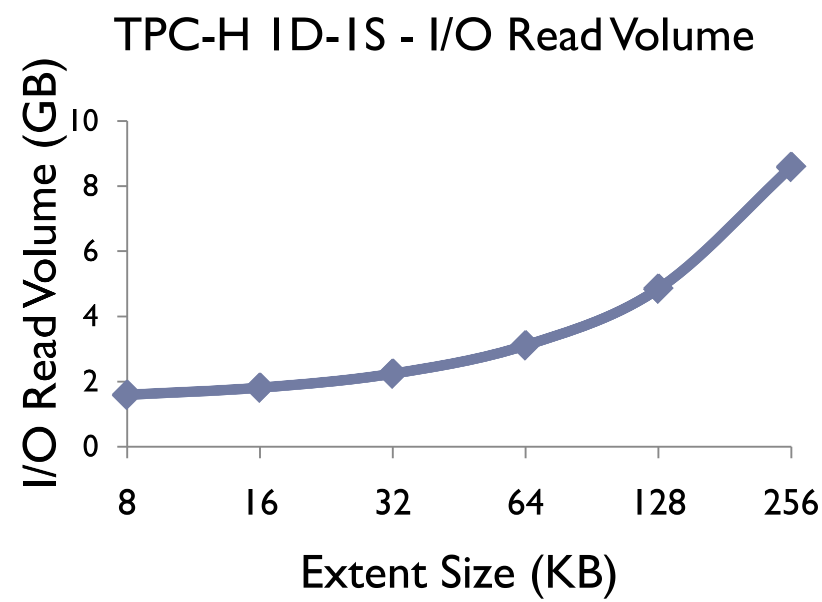
Impact on CPU utilization



- TPC-H: Up to 2x CPU utilization
- Postmark: Up to 4.5x CPU utilization
- SPEC SFS CPU utilization up to 25% higher

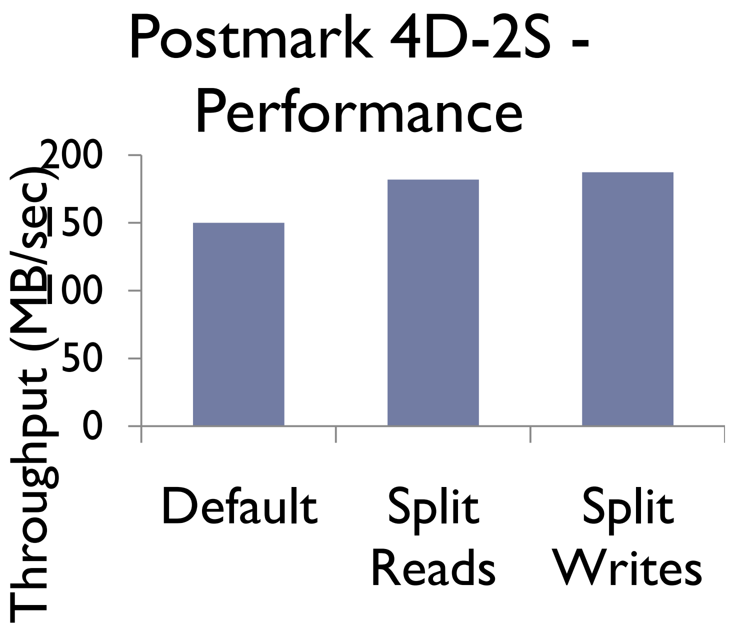
Impact of extent size

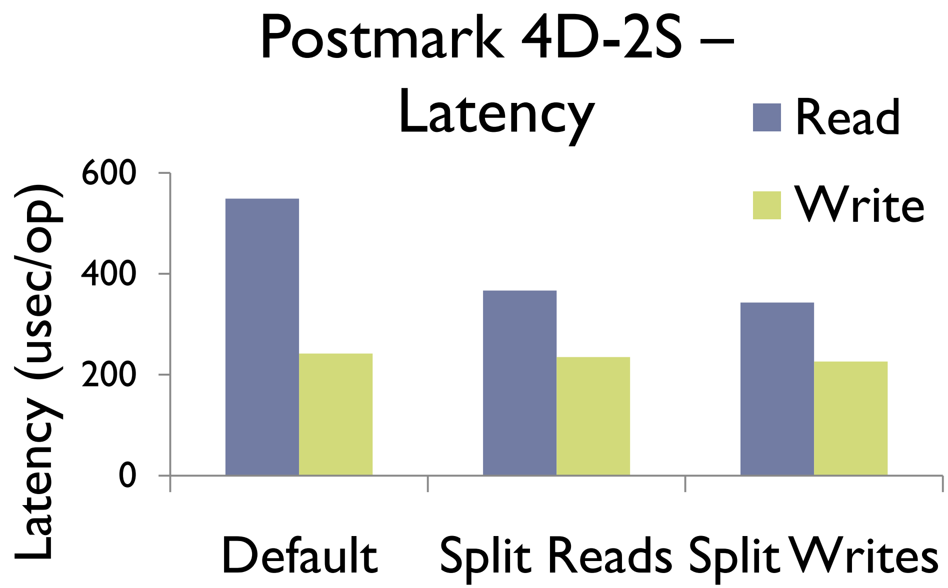




- Good choice for extent size 32-64KB
- Large extent size ➔ higher I/O volume
- Smaller extent size ➔ higher fragmentation , lower cache efficiency

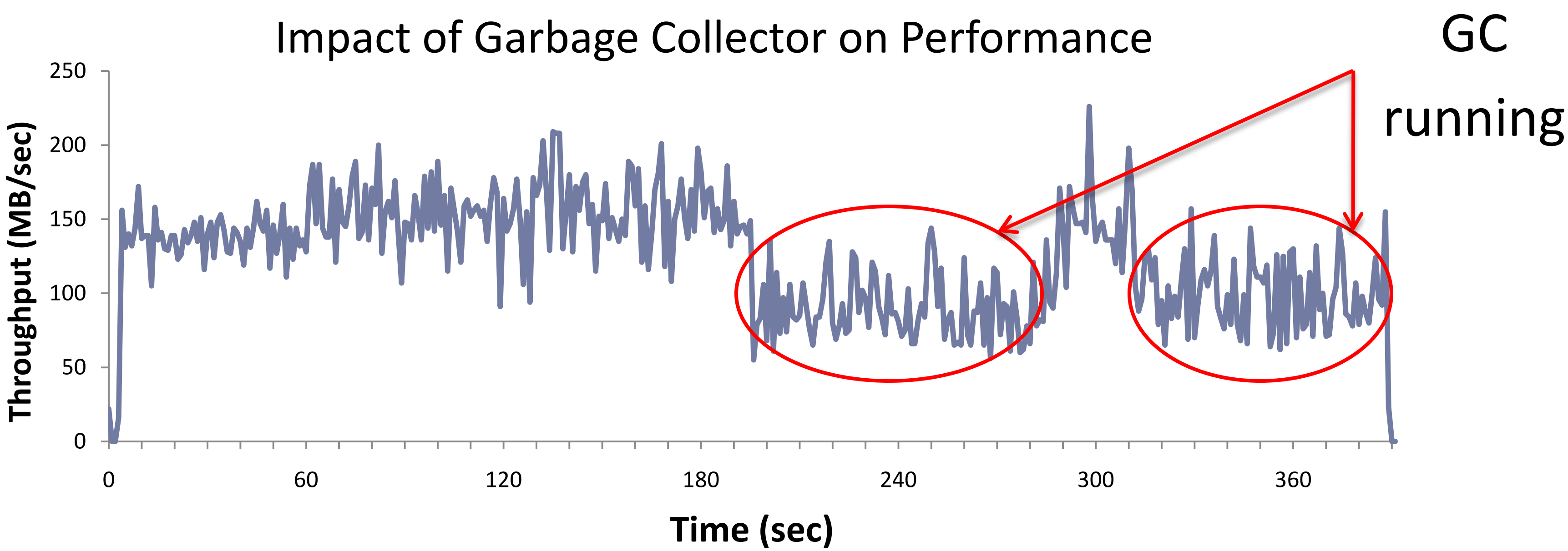
Impact of I/O request splitting





- Single-instance Postmark is bound by I/O response time due to blocking reads
- Read splitting improves overall throughput by 25%
- Adding write splitting small impact
  - Write concurrency due to write-back kernel buffer cache
- Response time of reads improves by 62% (35-65 read/write ratio)

Garbage collection overhead



- Workload: PostMark 2HDD-1SSD for cache
- Write volume exceeds SSD cache capacity
- GC is triggered to reclaim free space
  - In 90 seconds it reclaims 20% of capacity (6,3 GB)
  - GC activity seen as two “valleys”, 50% performance hit
- GC typically runs during idle I/O periods

Related Work

- Improve I/O performance with SSDs
  - 2nd level cache for web servers [CASES '06]
  - Transaction logs, rollback & TPC workloads [SIGMOD '08, EuroSys '09]
- FusionIO, Adaptec MaxIQ, ZFS's L2ARC, HotZone
  - Use SSDs as general-purpose uncompressed I/O caches
- ReadyBoost [Microsoft]
- Improve I/O performance by compression
  - Increased effective bandwidth [ACM SIGOPS '92]
  - DBMS performance optimizations [Oracle, IBM's IMS, TKDE '97]
- Reduce DRAM requirements by compressing memory pages
- Improve space efficiency (not performance) by FS compression
  - Sprite LFS, NTFS, ZFS, BTRFS, SquashFS, CramFS, etc.
- Other block-level compression: CBD, cloop: read-only devices

Conclusions

- Improve SSD caching efficiency using online compression
  - Trade (cheap) CPU cycles for (expensive) I/O performance
- Address challenges in online block-level compression for SSDs
  - Our techniques mitigate CPU and additional I/O overheads
- Results in increased performance with realistic workloads
  - TPC-H up to 99%, PostMark up to 20%, SPECsfs2008 up to 11%
  - Cache hit ratio improves between 22%-145%
  - Increased CPU utilization by up to 4.5x
  - Low concurrency, small I/O workloads problematic
- Overall our approach worthwhile, but adds complexity…
- Future work
  - Power-performance implications interesting, hardware off-loading
  - Improving compression efficiency by grouping *similar* blocks

Thank You!

Questions?

"Using Transparent Compression to Improve SSD-based I/O Caches"

Thanos Makatos, Yannis Klonatos, Manolis Marazakis, Michail Flouris, and Angelos Bilas

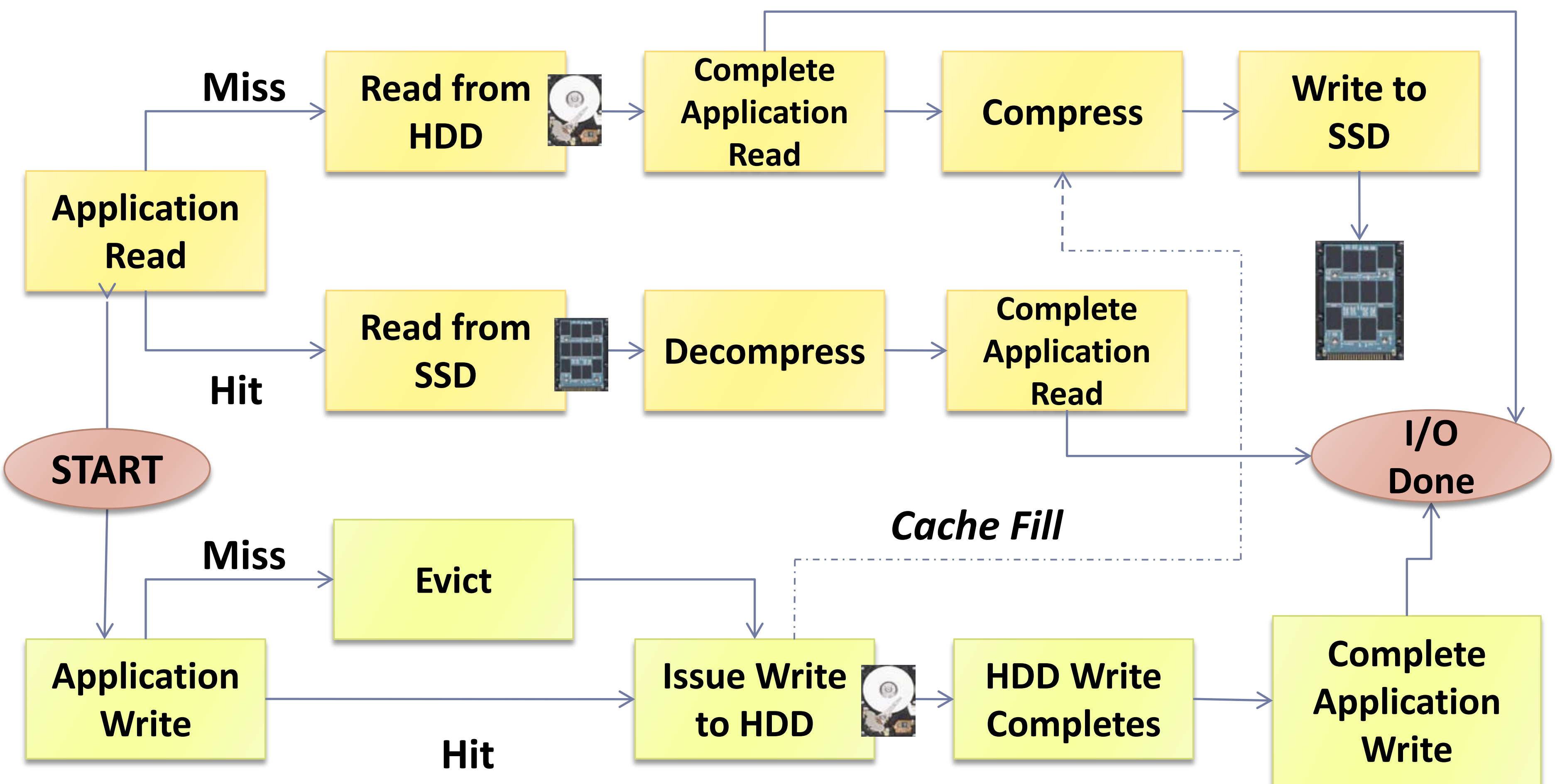
{mcatos,klonatos,maraz,flouris,bilas}@ics.forth.gr

FORTH-ICS
Institute of Computer Science

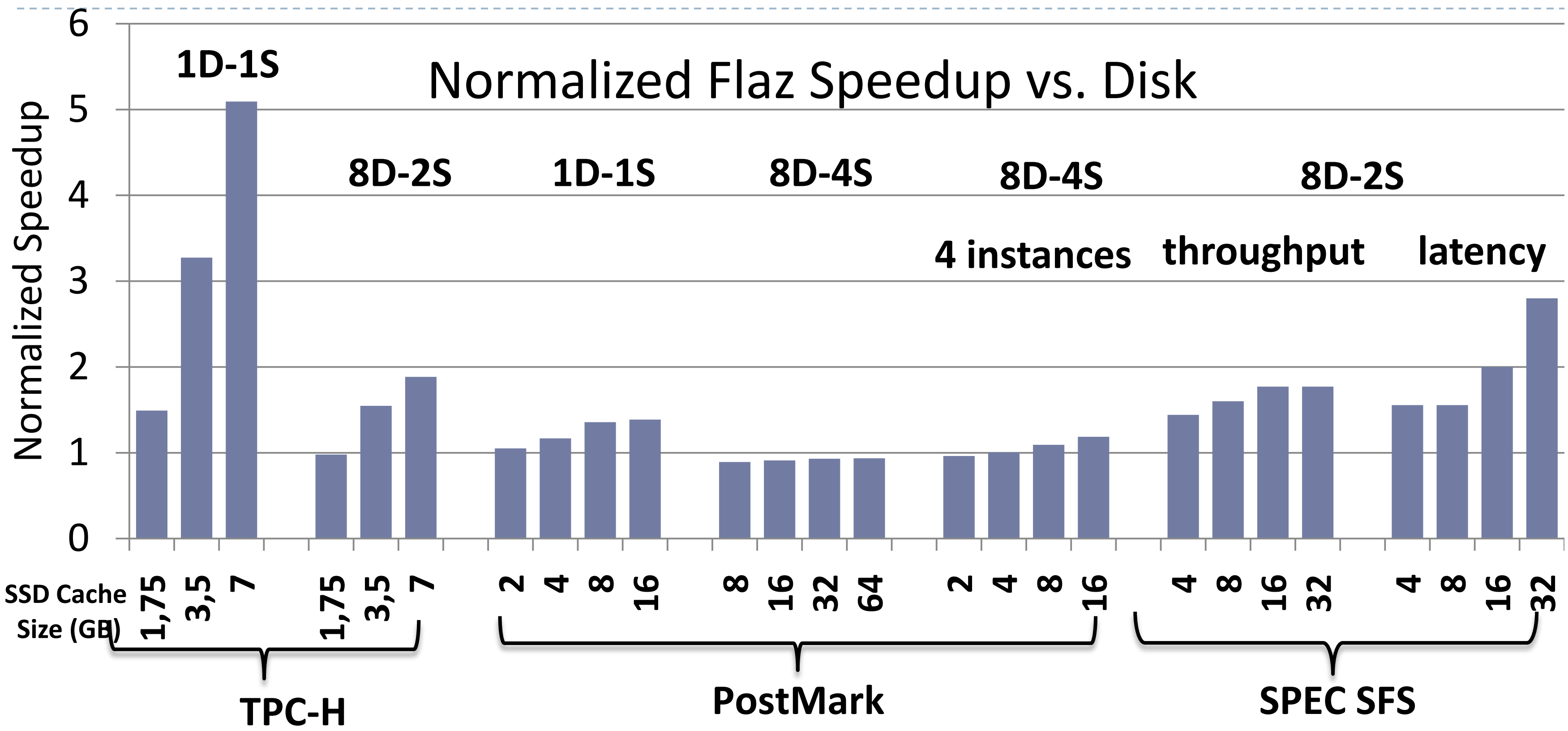
Foundation for Research & Technology - Hellas

http://www.ics.forth.gr/carv/scalable

I/O Request Logic



Overall impact on application I/O performance



- Normalized Flaz performance vs. **Disk**
- Improvement up to 1.5x-5x for TPC-H