SDC
STORAGE DEVELOPER CONFERENCE
SNIA SANTA CLARA, 2015

SSD-friendly Design Changes at Various Software Tiers

Zhenyun Zhuang
LinkedIn Corp.

Introduction

"If I had asked people what they wanted, they would have said faster horses."

Henry Ford

www.QuoteCorner.com

Outline

- Introduction
- Motivation
- SSD internals
- Design changes at different tiers for working with SSD
  - File Systems
  - Data Infrastructure
  - Application designs
  - System configurations
  - Performance measurement and benchmarking
  - Database
- Conclusion

Introduction

SSD is getting popular

- Application performance improves when using SSD (vs. HDD)

SSD merely treated as faster "HDD" by many people

- Naive treatment of SSD results in sub-optimal performance

SSD deserves new designs at many computing tiers

- File System
- Data infrastructure
- Application
- System configuration
- Performance measurement
- Database

What we will discuss in this talk?

Why?	Why does SSD require special designs at various tiers?
How?	How does SSD work internally (and differently from HDD)?
What?	What are the SSD-friendly design changes at various tiers?

Why do we need SSD-friendly design?

Better software performance

- Benefits the particular software/application
- E.g., higher throughput, lower response latency

More efficient storage IO

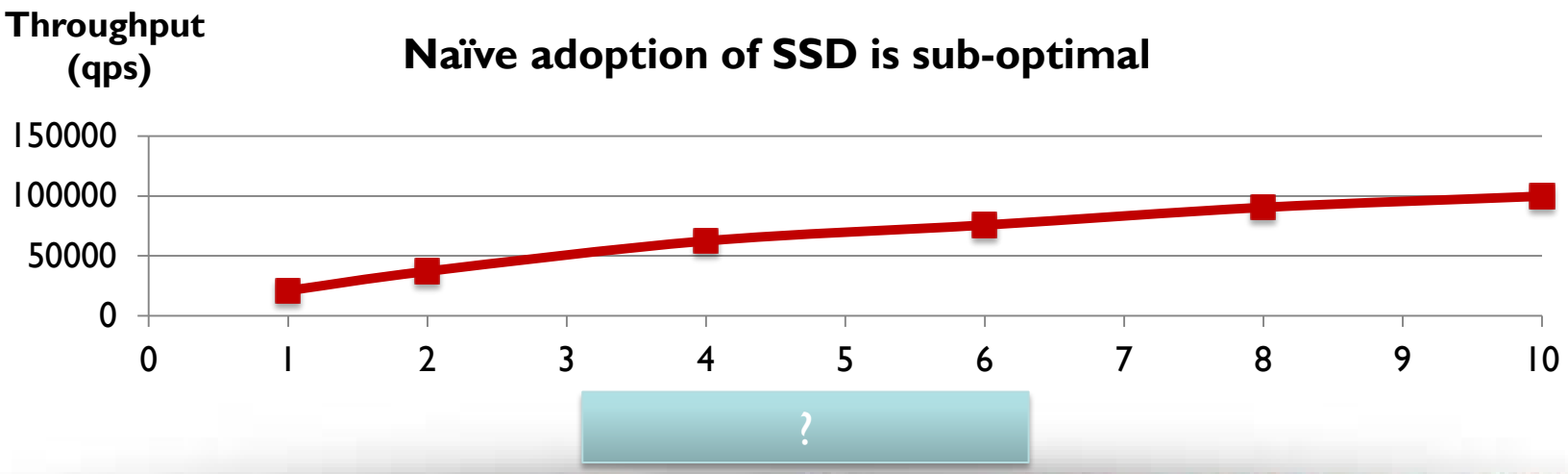
- Allows more applications to share the same storage
- Enables denser deployment

Longer SSD life

- Reduces business cost
- Saves a lot of troubles caused by dead SSD

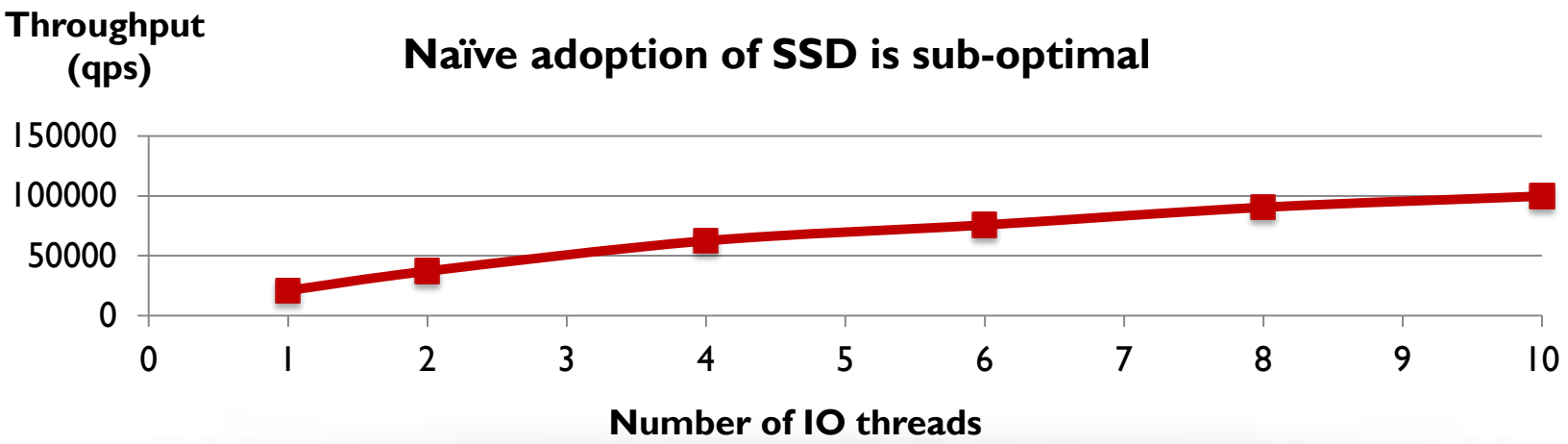
I. Better software performance

Simply replacing HDD with SSD	Redesigning software to make them SSD-friendly	Example application
• Performance improves but sub-optimal	• Could achieve much higher performance gains	• HDD storage: maximum **142** qps • Simply moving to SSD: **20K** qps • Being SSD-friendly: **100K** qps (5X improvement)



I. Better software performance

Simply replacing HDD with SSD	Redesigning software to make them SSD-friendly	Example application
• Performance improves but sub-optimal	• Could achieve much higher performance gains	• HDD storage: maximum **142** qps • Simply moving to SSD: **20K** qps • Being SSD-friendly: **100K** qps (5X improvement)



II. More efficient storage IO

- Read/Write at least page size (4KB)
- One byte write cause at least 4KB written

Inefficient write results in far more bytes written to SSD

- Echo "SSD" > foo.txt // Effective: 3 bytes
- SSD writes: 11 pages or 44KB

File system induced overhead

- 1GB/s SSD IO bandwidth could be saturated by a mere 256KB/s application IO rate (read or write).

SSD also has limited IO bandwidth

- How many applications can be co-located to share the same SSD?
- More efficient usage of SSD allows more applications to co-exist.

Denser deployments

III. Longer SSD life

SSD wears out

- SSD can only be "written" certain number of times before dying
- Costly: Saving SSD life is saving $

How long can a SSD live?

- SSD size: S
- P/E cycles: W
- Write amplification factor: F
- Application writing rate: R
- SSD life: $L = SW/FR$

SSD Type	P/E Cycles	WA Factor	Life (Months)
MLC	10K	4x	10
MLC	10K	10x	4
TLC	3K	10x	1

SSD Size: 1TB
Application Write Rate: 100 MB/s

Being SSD-friendly

- Help lengthening SSD life

Outline

- Introduction
- Motivation
- SSD internals
- Design changes at different tiers for working with SSD
  - File Systems
  - Data Infrastructure
  - Application designs
  - System configurations
  - Performance measurement and benchmarking
  - Database
- Conclusion

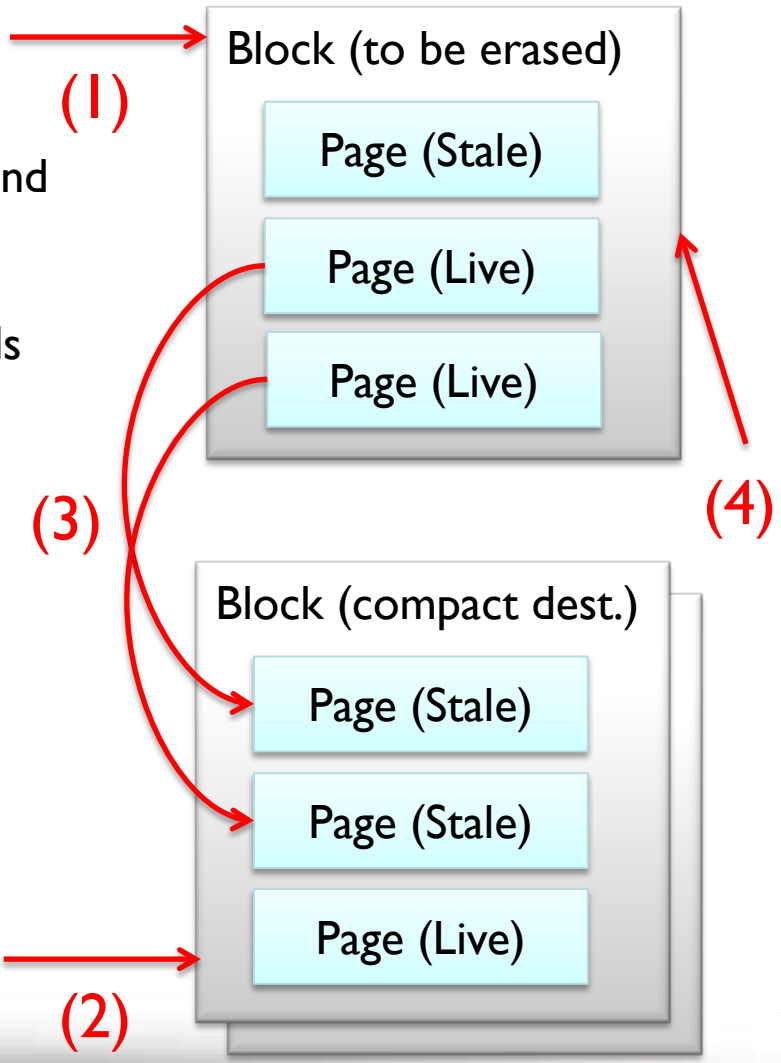
SSD IO Operations and Garbage Collection

IO operations

- Reading: page level; constant time; at microsecond level
- Writing: page level; depend on state; a few hundreds of microseconds or a few milliseconds
- Erasing: block level; a few millisecond

GC (Garbage Collection)

- No "over-writing" in SSD
- Compact blocks/pages to free a block for GC
- Background GC
  - Non-blocking
- Foreground GC
  - Performed online
  - Slow (blocking)



Wear Leveling and Write Amplification

Wear Leveling

- Blocks have limited P/E cycles (Program/Erase, erasure times)
  - SLC: 100K; MLC: 10K; TLC: a few K
- Balancing write actions among blocks

Write Amplification

- Physical write size is larger than logic (application) write size
- WA factor is the ratio; the smaller the better
- Key contributors of WA
  - Page-size write
  - FS-induced operations
  - Garbage Collection (GC)
  - Wear leveling

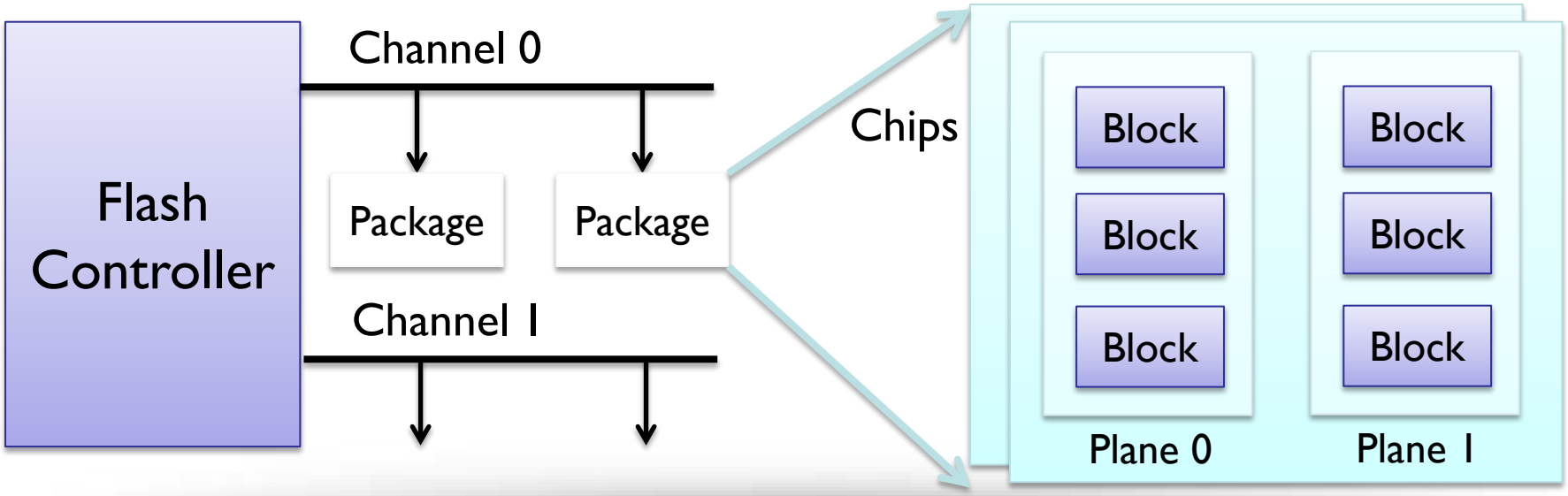
Internal Parallelism

Limitations of non-parallelism

- IO bandwidth: a few GB/s vs. NAND-flash bus only 40MB/s
- IO latency: hundreds of K IOPS vs. MLC Read of 50 us and write of up to 1ms

Multiple levels of parallelism

- Channel-level, Package-level, Chip-level, Plane-level (NOT taco-level!)



Outline

- Introduction
- Motivation
- SSD internals
- Design changes at different tiers for working with SSD
  - File Systems
  - Data Infrastructure
  - Application designs
  - System configurations
  - Performance measurement and benchmarking
  - Database
- Conclusion

What are the design changes at File System tier?

Key SSD characteristics (vs. HDD) that drive FS change

- Pro: Random access vs. sequential access
- Con: Blocks need to be erased for overwriting
- Con: SSD's write amplification caused by internal mechanisms

Two types of SSD-friendly FS

- General FS adapted for SSD
  - Supporting TRIM
  - Examples: Ext4, XFS, JFS, Btrfs
- Specially designed FS for SSD
  - Adopting log-structure
  - Examples: ExtremeFFS, NVFS, JFFS/JFFS2/LogFS, F2FS

“Log structure”— New wine in old bottle

Always sequential writing

- Data and metadata always written to circular buffer (or file tail)

Log structure in HDD world

- Sequential writing vs. random writing
- LFS (log-structured file system)
- HDFS commit log
- Oracle Database redo log

Log structure in SSD world

- “Read-modify-write” to only “write”
- Minimize wear leveling to reduce write amplification factor

Outline

- Introduction
- Motivation
- SSD internals
- Design changes at different tiers for working with SSD
  - File Systems
  - Data Infrastructure
  - Application designs
  - System configurations
  - Performance measurement and benchmarking
  - Database
- Conclusion

What are the design changes at Data Infra tier?
- Revisiting conventional design rationales

Conventional assumptions may not hold!	Local disk vs. remote memory (another node)
	Used to favor remote memory (though with added network hops, deployment complexity, operation cost)

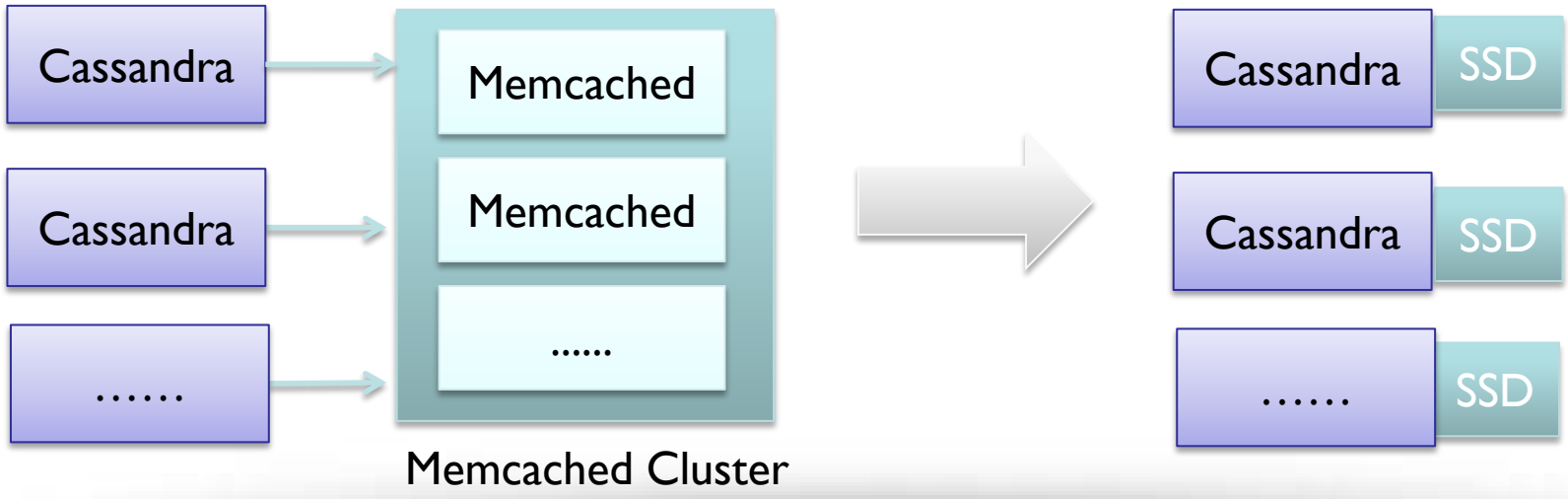
Before (with HDD)	Local Disk	Remote Memory
Latency	A few milliseconds	A few microseconds
Bandwidth	100 MB/s	120 MB/s (Gbit), 1.2 GB/s (10Gbit)

Now (with SSD)	Local Disk	Remote Memory
Latency	A few microseconds	A few microseconds
Bandwidth	Up to a few GB/s	120 MB/s (Gbit), 1.2 GB/s (10Gbit)

What are the design changes at Data Infra Tier?

- An example of removing *memcached* layer

Cassandra + Memcached			
Assumptions	**Before**	**After**	**Cost:**
• 10 Cassandra nodes • Requiring caching 10TB of data	• 100GB RAM per memcached • Needs100 memcached nodes	• Each Cassandra adds 1TB SSD	• Before: 100 nodes • After: 10 SSDs



Outline

- Introduction
- Motivation
- SSD internals
- Design changes at different tiers for working with SSD
  - File Systems
  - Data Infrastructure
  - Application designs
  - System configurations
  - Performance measurement and benchmarking
  - Database
- Conclusion

What are the design changes at application tier?

Data structure

- Avoid in-place update optimizations
- Separate hot data from cold data
- Adopt compact data structure

IO handling

- Avoid long heavy writes
- Prefer not mixing write and read
- Prefer large IO aligned on pages/blocks/more

Threading

- Use multiple threads (vs. few threads) to do small IO
- Use few threads (vs. many threads) to do big IO

Data structure
- Avoid in-place update optimizations

Conventional HDD storage

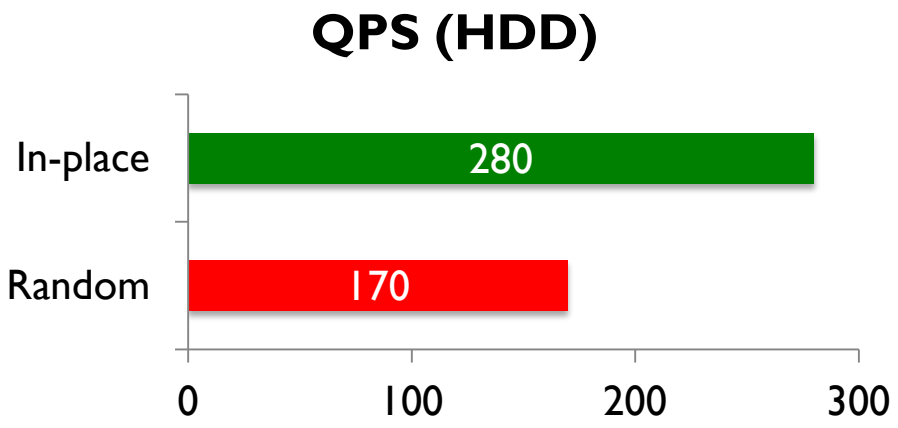
- Optimized for in-place updates (write to the same offset)
- HDD seeking is very costly

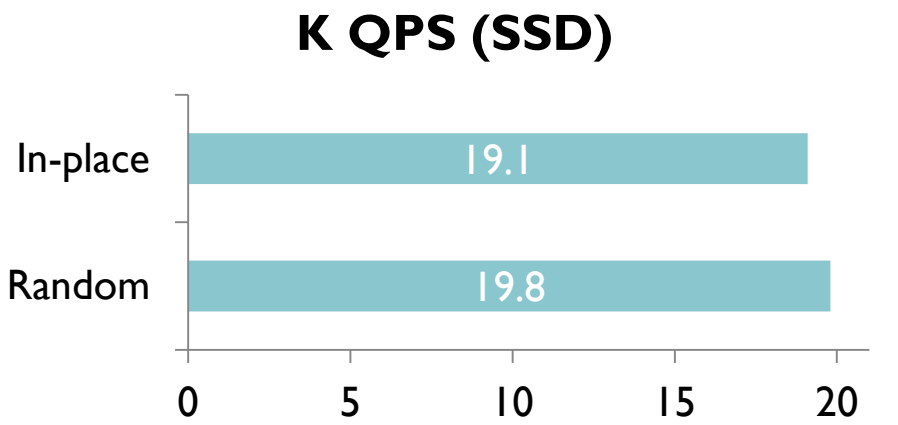
SSD storage

- In-place updates are unnecessary
- IO is slower: "read-modify-write"
- Penalizing SSD: read-disturbance

No in-place update optimizations

- Unless non-in-place updates greatly complicate design
- Consider log-structured updates





Data structure
- Separate hot data from cold data

Data are not equally active

- Page-size IO access and Block-size GC
- Mixing hot/cold data causes useless IO on cold data

Performance penalties of mixing hot/cold

- Reduced application performance (Throughput, response time)
- Decreased IO efficiency (IO bandwidth)
- Increased SSD wear out (life)

Store hot/cold data separately

- Bad example: Store user profiles based on registration time
- Spaced by at least page-size, e.g., different files, different portions in files, different tables

Data structure
- Adopt compact data structure

IO characteristics of SSD

- Page-size (e.g., 4KB) write and read
- Block-size (e.g., 1MB) erase

Store data more compactly

- Increases locality of read/write
- Read/write fewer physical bytes

Example: Storing user profile data

- Use a single file
- Use many files (telephone number, age, address, etc.)

IO handling
- Avoid long heavy writes

Long heavy writes will trigger foreground GC

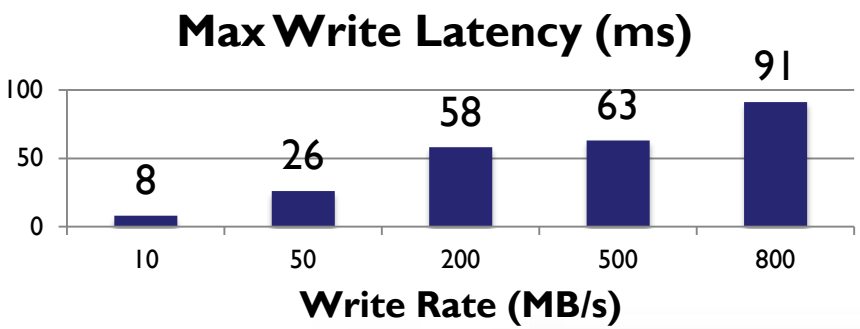
- Background GC can absorb light writes
- If background GC cannot keep up, foreground GC will kick in

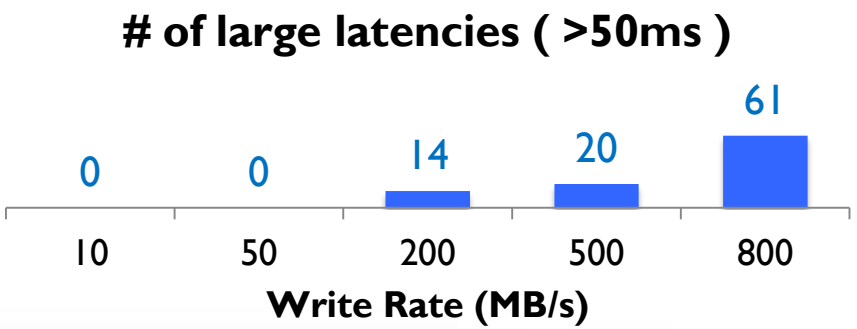
Foreground GC severely degrades write performance

- Every write needs block erasure
- Block erasure takes up to 2ms (Degrades to HDD-like perf)

How to avoid long heavy writes?

- More efficient IO
- Use multiple SSD and/or remote storage
- Consider other persistency methods (e.g., Kafka streaming)





IO handling
- Prefer not mixing write and read

Read and write interfere each other

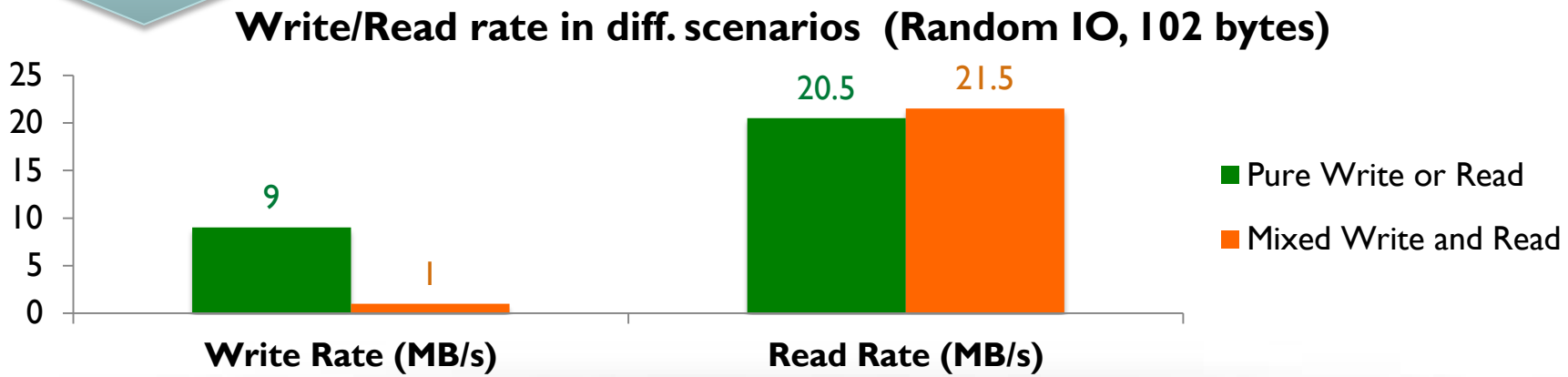
- SSD-Internal shared resources: e.g., Lock-protected mapping table
- SSD pipelining of moving data: e.g., flash, register, controller
- File System: Read-ahead and write-back

Degraded IO rate

- Depends on SSD implementations and workload
- Can hurt either write or read or both

Avoid mixing at the same time

- Phase heavy read and heavy write
- Consider multiple SSD or storage media



IO handling
- Prefer large IO, aligned on page/block/more

Why large IO

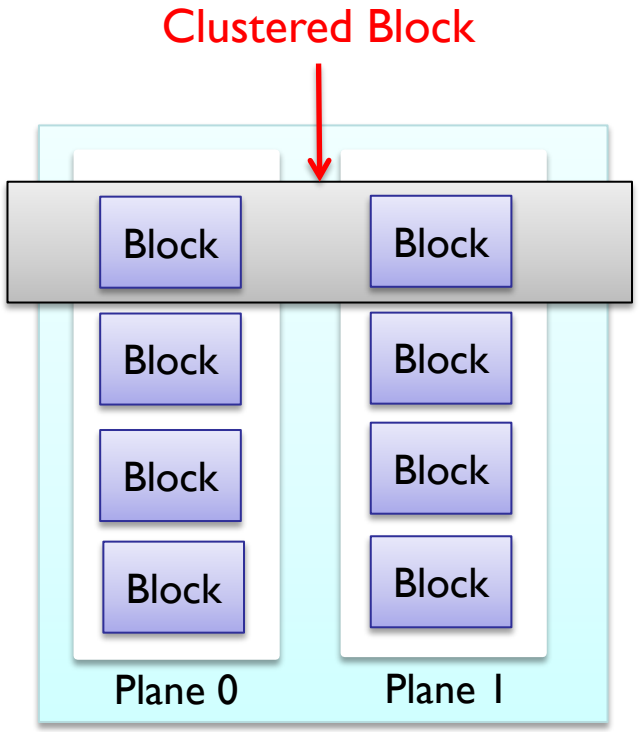
- Writing/reading by page
- Erasing by block
- Using clustered blocks

Benefits of large IO

- Increased IO efficiency
- Reduced Write Amplification factor
- Higher throughput from internal parallelism

Why aligned on pages/blocks

- Reducing by one page/block when data crossing borders
- Faster IO



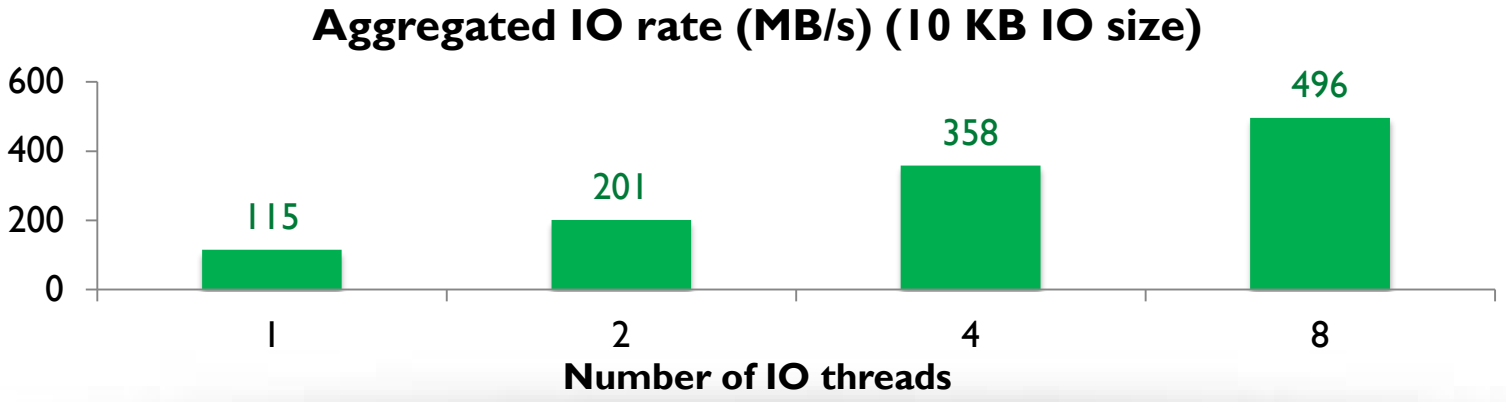
Threading
- Use many threads (vs. few) to do small IO

Why many threads?

- Take advantage of internal parallelism, i.e., channel-level, package-level, chip-level, plane-level

How small is "small"?

- Depends on how compactly the data are stored
- Page-compacted: (page size)*(parallelism level), e.g., 4KB*16=64KB
- Block-compacted: (block size) * (parallelism level), e.g., 0.5MB*16=8MB



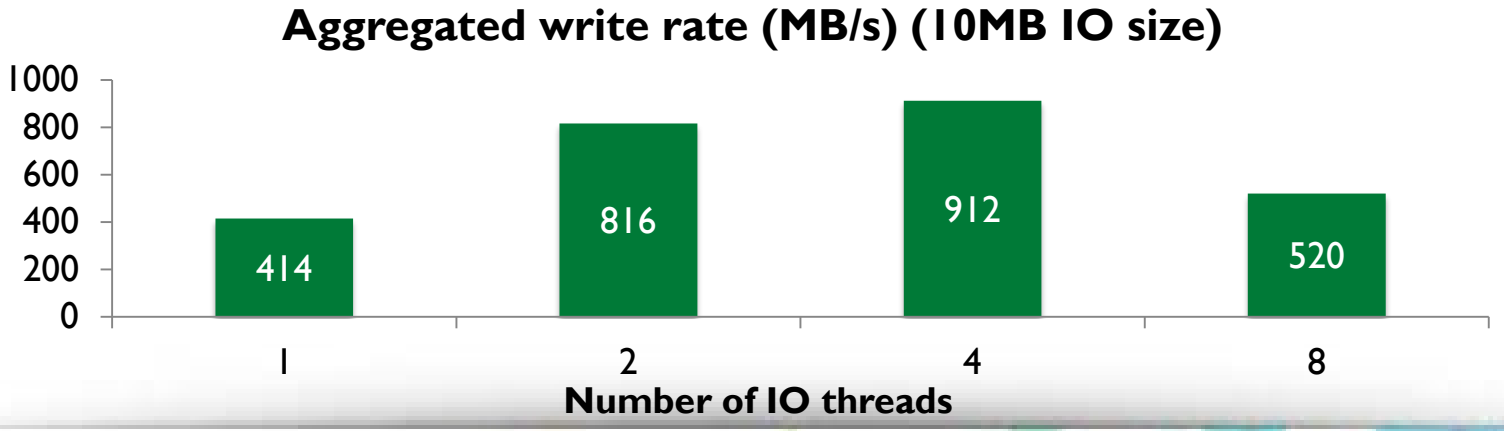
Threading
- Use few threads (vs. many) to do big IO

Why not many threads?

- SSD controller already uses internal parallelism with big IO
- Threads interfere each other (e.g., sharing SSD resources)
- Threads interfere other applications (e.g. pre-fetching)

How big is "big"?

- Depends on data layout
- Larger than (block size)*(parallelism level), e.g., 0.5MB*16=8MB



Outline

- Introduction
- Motivation
- SSD internals
- Design changes at different tiers for working with SSD
  - File Systems
  - Data Infrastructure
  - Application designs
  - System configurations
  - Performance measurement and benchmarking
  - Database
- Conclusion

Avoid full disk usage

Performance impact of disk usage

- Write Amplification factor due to GC
- Write latency during foreground GC

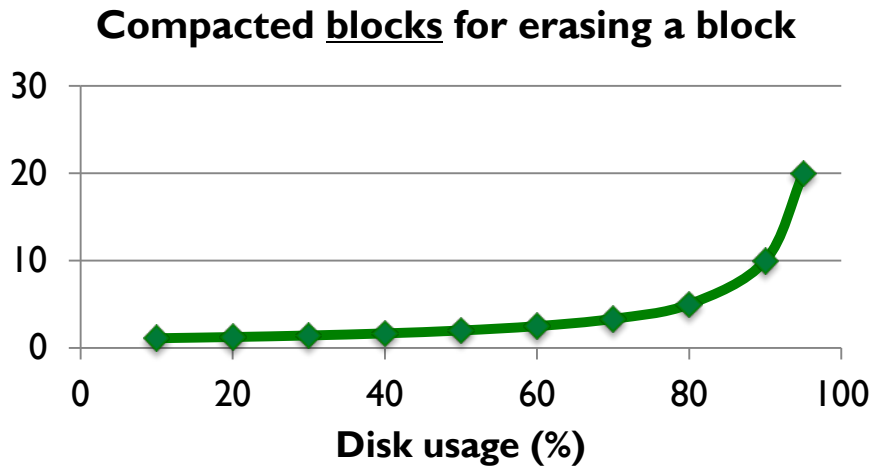
GC needs to compact blocks/pages

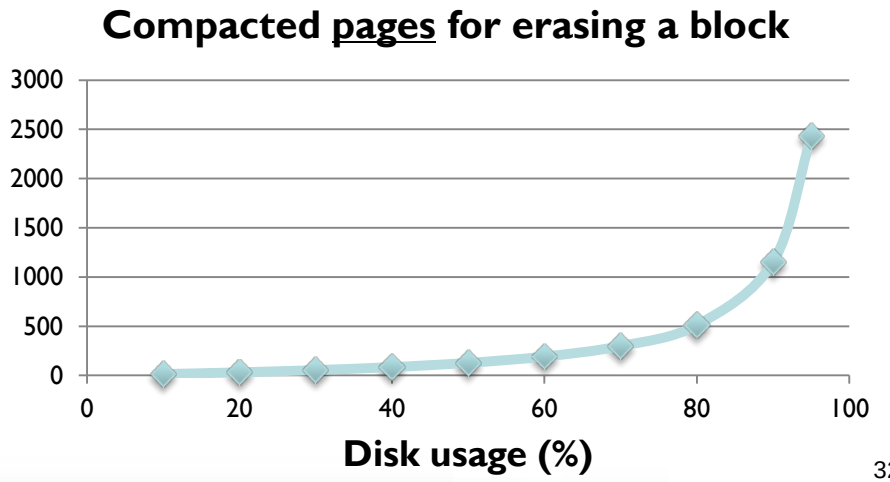
- Number of blocks to be compacted
  - Assuming A% disk usage, a single erasure compacts blocks:
  - A=50: 2 blocks
  - A=80: 5 blocks

$$1/(1-A\%)$$

- Number of pages to be compacted
  - Assuming P pages per block, a single erasure compacts pages:
  - P=128, A=50: 128 pages
  - P=128, A=80: 512 pages

$$P \times A\% / (1-A\%)$$





Be careful when swapping on SSD

Benefits of swapping on SSD

- Faster (100+x faster than HDD)

Problems of swapping/storing on SSD

- Swapping wears out SSD quickly
- A fast storage may hurt performance
  - OS read-ahead fills the cache too fast and encourages swapping out
  - Observed on Voldemort

When to swap on SSD

- Swapping performance is the primary concern
  - Less concerned with SSD life and cost
- Swapping rarely happen
  - Swappiness value set to low to discourage swapping

Outline

- Introduction
- Motivation
- SSD internals
- Design changes at different tiers for working with SSD
  - File Systems
  - Data Infrastructure
  - Application designs
  - System configurations
  - Performance measurement and benchmarking
  - Database
- Conclusion

Performance measurement and benchmarking

Major pitfalls

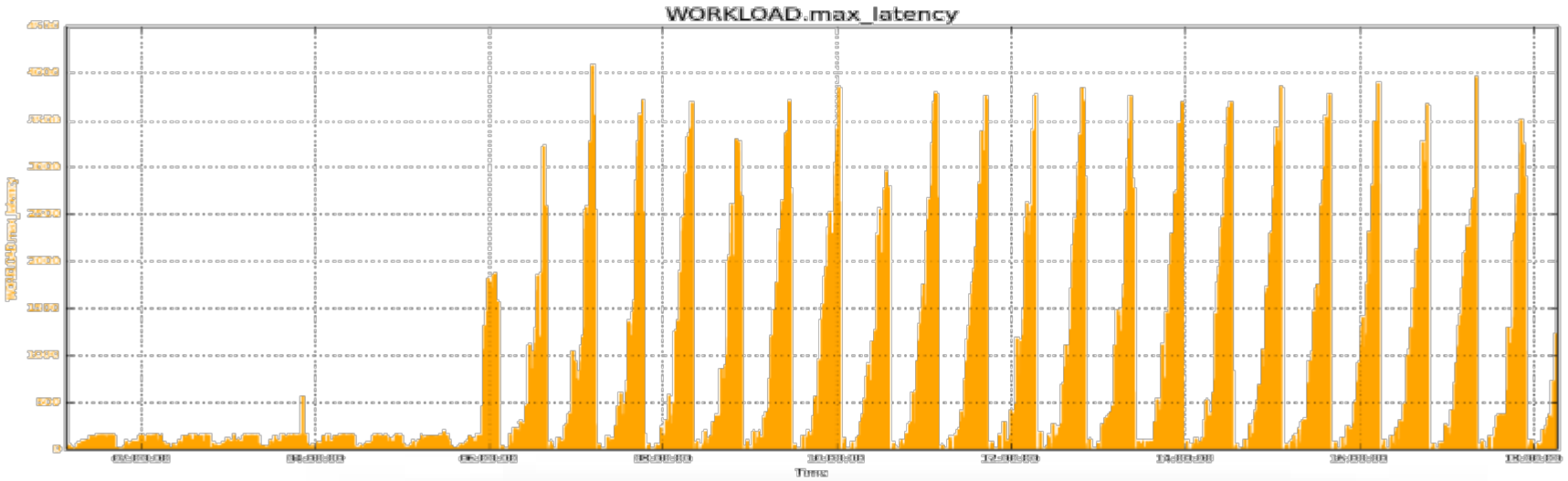
- Performance depends on the previous state
- Foreground GC

Recommendations

- Stress SSD for long time to stabilize
- Use representative workload

Example: SSD IO performance

- Synchronous writing; Write rates iterates between 15 MB/s and 500 MB/s



Outline

- Introduction
- Motivation
- SSD internals
- Design changes at different tiers for working with SSD
  - File Systems
  - Data Infrastructure
  - Application designs
  - System configurations
  - Performance measurement and benchmarking
  - Database
- Conclusion

What are the design changes at Database tier?

Two types of SSD-friendly Database

- Flash only Database (e.g. Aerospike)
- Hybrid flash-HDD systems

Key design changes

- IO Concurrency
  - One thread per database connection is sub-optimal
- Data structure
  - B-tree vs. Log-structured tree
- Data layout
  - Locality matters differently
  - Column-oriented vs. Row-oriented

Outline

- Introduction
- Motivation
- SSD internals
- Design changes at different tiers for working with SSD
  - File Systems
  - Data Infrastructure
  - Application designs
  - System configurations
  - Performance measurement and benchmarking
  - Database
- Conclusion

Key take-away

Don't treat SSD as simply a faster HDD

- SSD has its own unique mechanisms, e.g., no-overwriting, GC

Take full advantage of SSD

- True that SSD has better performance than HDD
- But it may not be fully utilized

Design changes at various tiers

- File Systems
- Data infrastructure
- Application designs
- System configurations
- Performance measurement and benchmarking
- Database

Looking into the future

NAND SSD is much faster than HDD

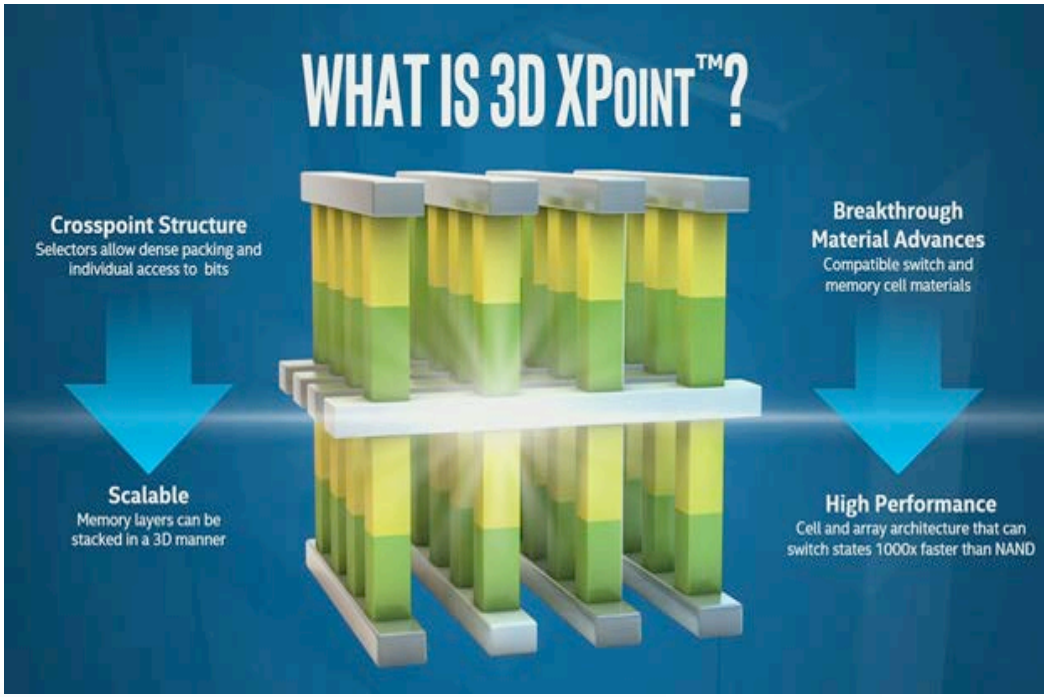
- Two or three orders of improvements on IOPS
- One order of improvement on throughput

Imagine a new storage that is 1000X faster?

- 1000X faster (iops/rate)
- 1000X endurance (life)
- 10X denser (capacity)

What design changes can you imagine?

- Intel/Micron 3D SSD



Source: intel.com

References (Partially)

- Solid state drive: https://en.wikipedia.org/wiki/Solid-state_drive
- Flash file system: https://en.wikipedia.org/wiki/Flash_file_system
- Netflix blog: http://techblog.netflix.com/2012/07/benchmarking-high-performance-io-with.html
- Voldemort on SSD: https://engineering.linkedin.com/voldemort/voldemort-solid-state-drives
- Coding for SSD: http://codecapsule.com/2014/02/12/coding-for-ssds-part-1-introduction-and-table-of-contents/
- Intel and Micron's 3D XPoint Technology http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology
- SSD and Distributed Data Systems: http://blog.empathybox.com/post/24415262152/ssds-and-distributed-data-systems
- Co-design of application software and NAND…, http://radar.oreilly.com/2014/08/how-flash-changes-the-design-of-database-storage-engines.html
- F2FS: a new file system for flash storage, USENIX FAST 2015
- FlashGraph: processing billion-node graphs on an array of commodity SSDs, USENIX FAST 2015
- Wear Unleveling: Improving NAND Flash Lifetime by Balancing Page Endurance, USENIX FAST 2014
- Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems, USENIX FAST 2013
- Consistent and durable data structures for non-volatile byte-addressable memory, USENIX FAST 2011
- Impact of Data Locality on Garbage Collection in SSDs: A General Analytical Study, ACM ICPE 2015
- NAND flash memory-based hybrid file system for high I/O performance, IPDPS 2012
- Parallel I/O aware query optimization, ACM SIGMOD 2012
- Essential Roles of Exploiting Internal Parallelism of Flash Memory based Solid State Drives in High-Speed Data Processing, 2011 IEEE HPCA
- A column-oriented storage query optimization for flash-based database, Future Information Technology and Management Engineering (FITME), 2010
- Revisiting Database Storage Optimizations on Flash, Tech Report of Computer Science at University of Wisconsin Madison
- Query processing techniques for solid state drives, SIGMOD 2009
- Lazy-Update B+- Tree for flash devices, Mobile Data Management, 2009
- Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices, VLDB 2009
- Programming models for emerging non-volatile memory technologies, Andy Rudoff
- Patent (filed): LI-P1648.LNK.US, TRANSPARENT HYBRID DATA STORAGE (Zhenyun Zhuang)
- Paper (in preparation): *Designing SSD-friendly applications*
- Paper (in preparation): *HSL: A Hybrid Storage Layer Harnessing SSD and HDD*