Linux and H/W optimizations for MySQL

Yoshinori Matsunobu

MySQL Geek, Oracle ACE Director, DeNA
Former APAC Lead MySQL Consultant at MySQL/Sun/Oracle
Yoshinori.Matsunobu@gmail.com
http://yoshinorimatsunobu.blogspot.com

Table of contents

- MySQL performance overview and improvement history
- SSD and MySQL performance
- Memory and swap space management
- File I/O
- Network
- Useful command line tools and tips

History of MySQL performance improvements

- **H/W improvements**
  - HDD RAID, Write Cache
  - Large RAM
  - SATA SSD、PCI-Express SSD
  - More number of CPU cores
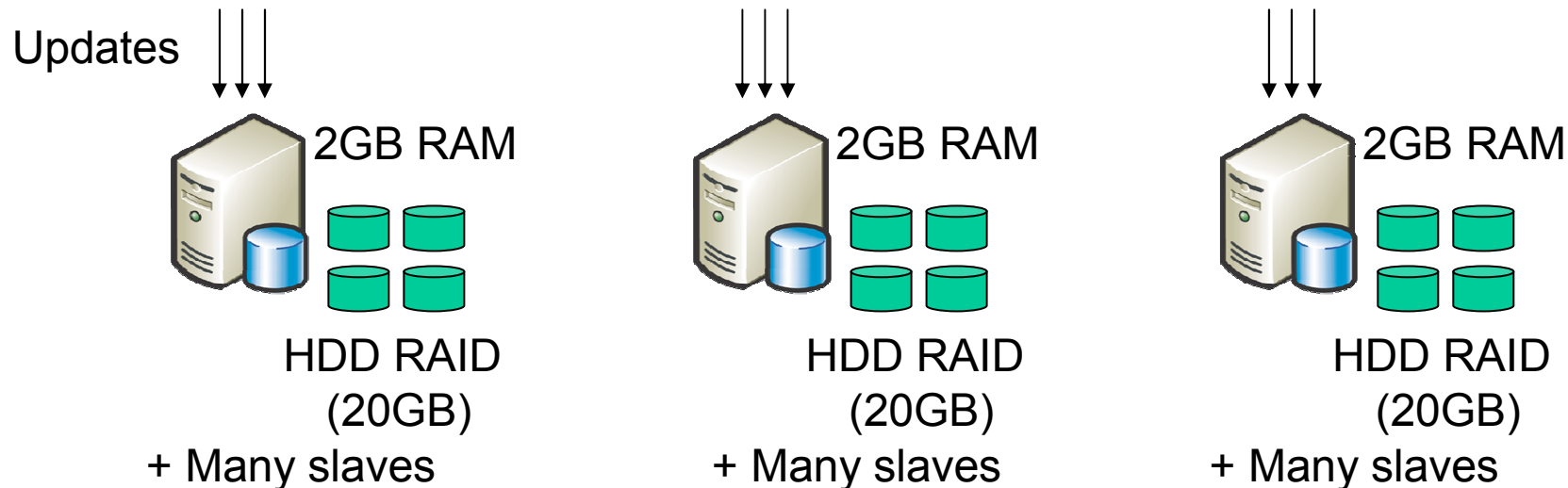 - Faster Network

- **S/W improvements**
  - Improved algorithm (i/o scheduling, swap control, etc)
  - Much better concurrency
  - Avoiding stalls
  - Improved space efficiency (compression, etc)

Per-server performance is important

- **To handle 1 million queries per second..**
  - 1000 queries/sec per server : 1000 servers in total
  - 10000 queries/sec per server : 100 servers in total

- **Additional 900 servers will cost 10M$ initially, 1M$ every year**

- **If you can increase per server throughput, you can reduce the total number of servers, which will decrease TCO**

- **Sharding is not everything**

32bit Linux

Updates

2GB RAM

2GB RAM

2GB RAM

HDD RAID
(20GB)
+ Many slaves

HDD RAID
(20GB)
+ Many slaves

HDD RAID
(20GB)
+ Many slaves

- **Random disk i/o speed (IOPS) on HDD is very slow**
  - 100-200/sec per drive
- **Database easily became disk i/o bound, regardless of disk size**
- **Applications could not handle large data (i.e. 30GB+ per server)**
- **Lots of database servers were needed**
- **MyISAM was widely used because it's very space efficient and fast**

64bit Linux + large RAM + BBWC

16GB RAM

+ Many slaves

HDD RAID
(120GB)
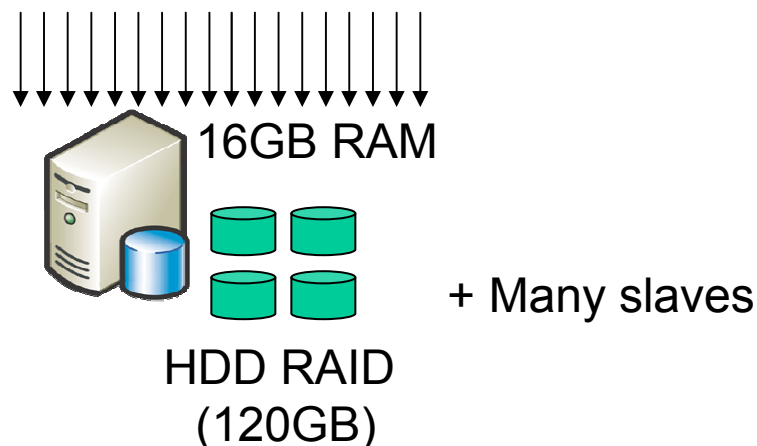
- ■ Memory pricing went down, and 64bit Linux went mature
- ■ It became common to deploy 16GB or more RAM on a single linux machine
- ■ Memory hit ratio increased, much larger data could be stored
- ■ The number of database servers decreased (consolidated)
- ■ Per server traffic increased (the number of users per server increased)
- ■ "Transaction commit" overheads were extremely reduced thanks to battery backed up write cache
- ■ From database point of view,
  - ● InnoDB became faster than MyISAM (row level locks, etc)
  - ● Direct I/O became common

Side effect caused by fast server

Master

HDD RAID

Slave
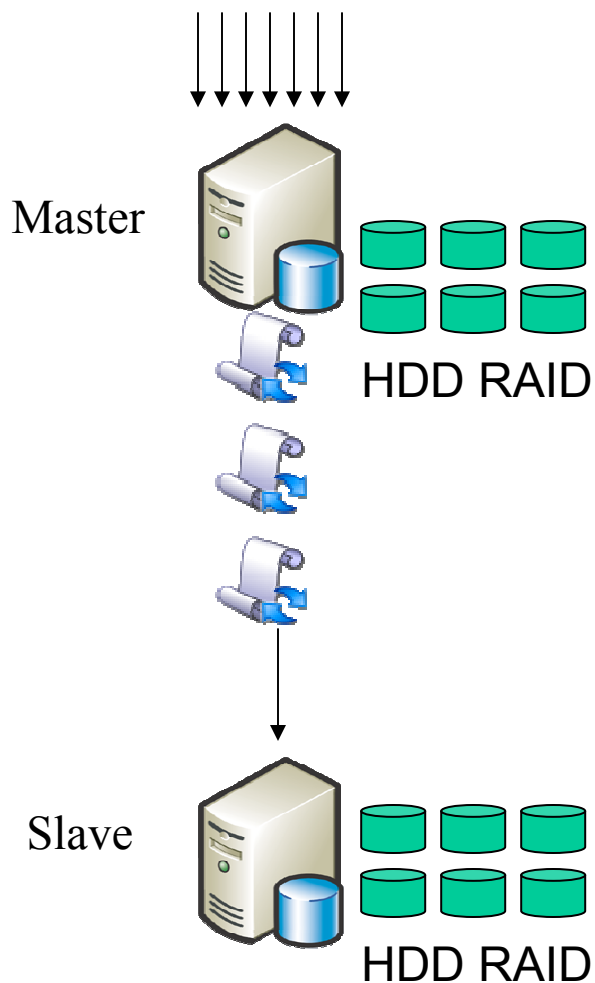
HDD RAID

- After 16-32GB RAM became common, we could run many more users and data per server
  - Write traffic per server also increased
- 4-8 RAID 5/10 also became common, which improved concurrency a lot
- On 6 HDD RAID 10, single thread IOPS is around 200, 100 threads IOPS is around 1000-2000
- Good parallelism on both reads and writes on master

- On slaves, there is only one writer thread (SQL thread). No parallelism on writes
  - 6 HDD RAID10 is as slow as single HDD for writes

- Slaves became performance bottleneck earlier than master

- Serious replication delay happened (10+ minutes at peak time)

Using SATA SSD on slaves

Master

HDD RAID
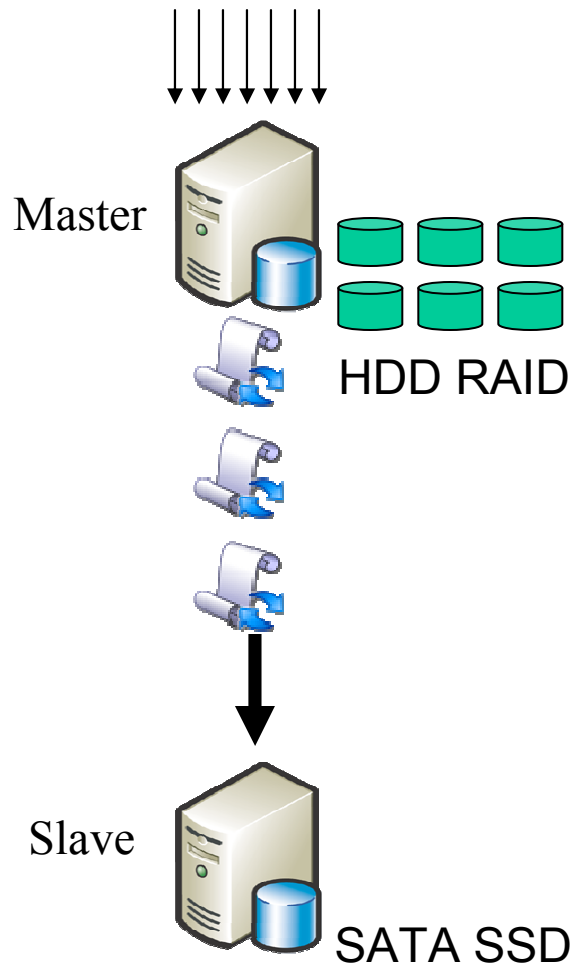
Slave

SATA SSD

- IOPS differences between master (1000+) and slave (100+) have caused serious replication delay
- Is there any way to gain high enough IOPS from single thread?

- Read IOPS on SATA SSD is 3000+, which should be enough (15 times better than HDD)
- Just replacing HDD with SSD solved replication delay
- Overall read throughput became much better

- Using SSD on slaves (IOPS: 100+ -> 3000+) was more effective than using on master (IOPS: 1000+ -> 3000+)
- We have mainly deployed SSD on slaves
- The number of slaves could be reduced

- From MySQL point of view:
  - Good concurrency on HDD RAID has been required : InnoDB Plugin

Concurrency improvements around MySQL

- **Row level locks, not table/page level locks**
  - InnoDB: Row level locks, MyISAM: Table level locks

- **No Hot Giant Lock(Global Mutex)**
  - Critical sections (spots that only single thread can be accessible) reduce concurrency a lot
  - InnoDB Buffer Pool, Data Dictionary, etc
  - InnoDB now scales pretty well with many CPU cores (5.1 Plugin, 5.5)

- **Parallel I/O threads**
  - Increasing the number of disk i/o threads
    - Front-end operations (i.e SELECT) throughput improved
  - Introducing asynchronous I/O

- **Parallel background tasks**
  - Dividing checkpoint thread (main thread) and purge thread (physically removing delete-marked records)

- **Single-threaded replication channel (SQL Thread) is currently hot spot**
  - Can be mitigated in MySQL 5.6 (per-database SQL thread)

Avoiding sudden performance drops

Perf

Product A

Product B

Time
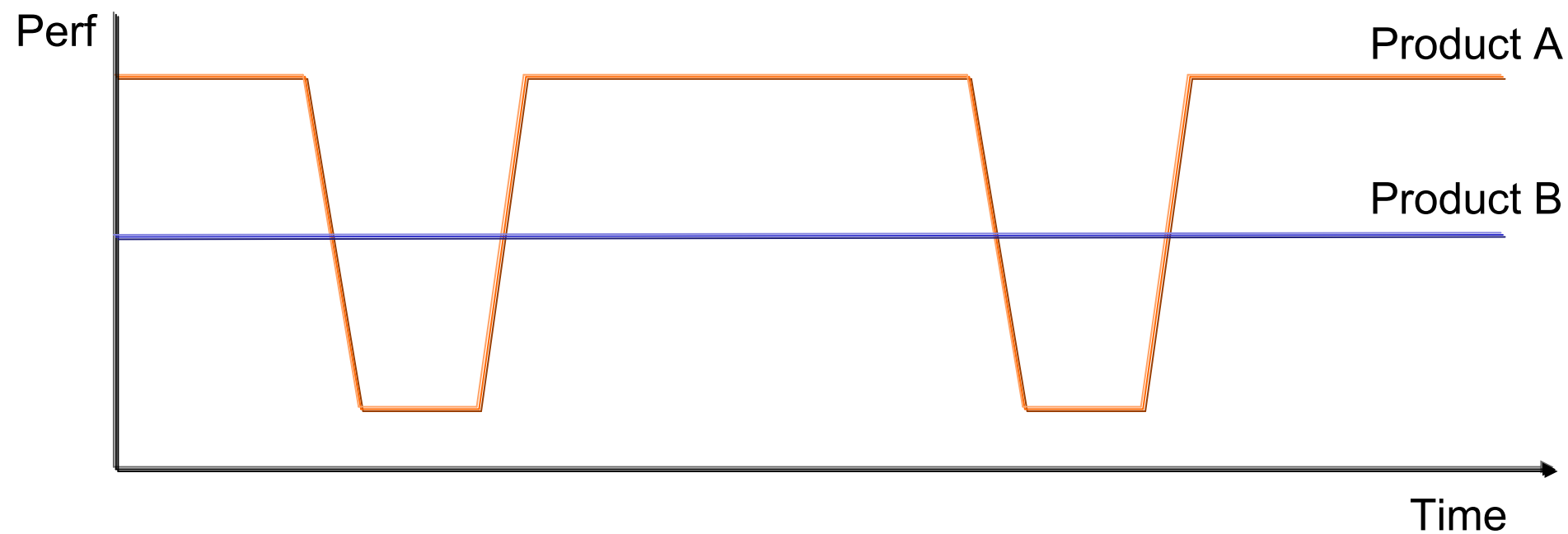
- Some unstable database servers suddenly drop performance in some situations
- Low performance is a problem because we can't meet customers' demands
- Through product A is better on average, product B is much more stable
- Don't trust benchmarks. Vendors' benchmarks show the best score but don't show worse numbers

Avoiding stalls

- All clients are blocked for a short period of time (less than one second – a few seconds)
  - The number of connections grow significantly (10-100 on average, but suddenly grows to 1000+ and TOO MANY CONNECTIONS errors are thrown)
  - Increased response time
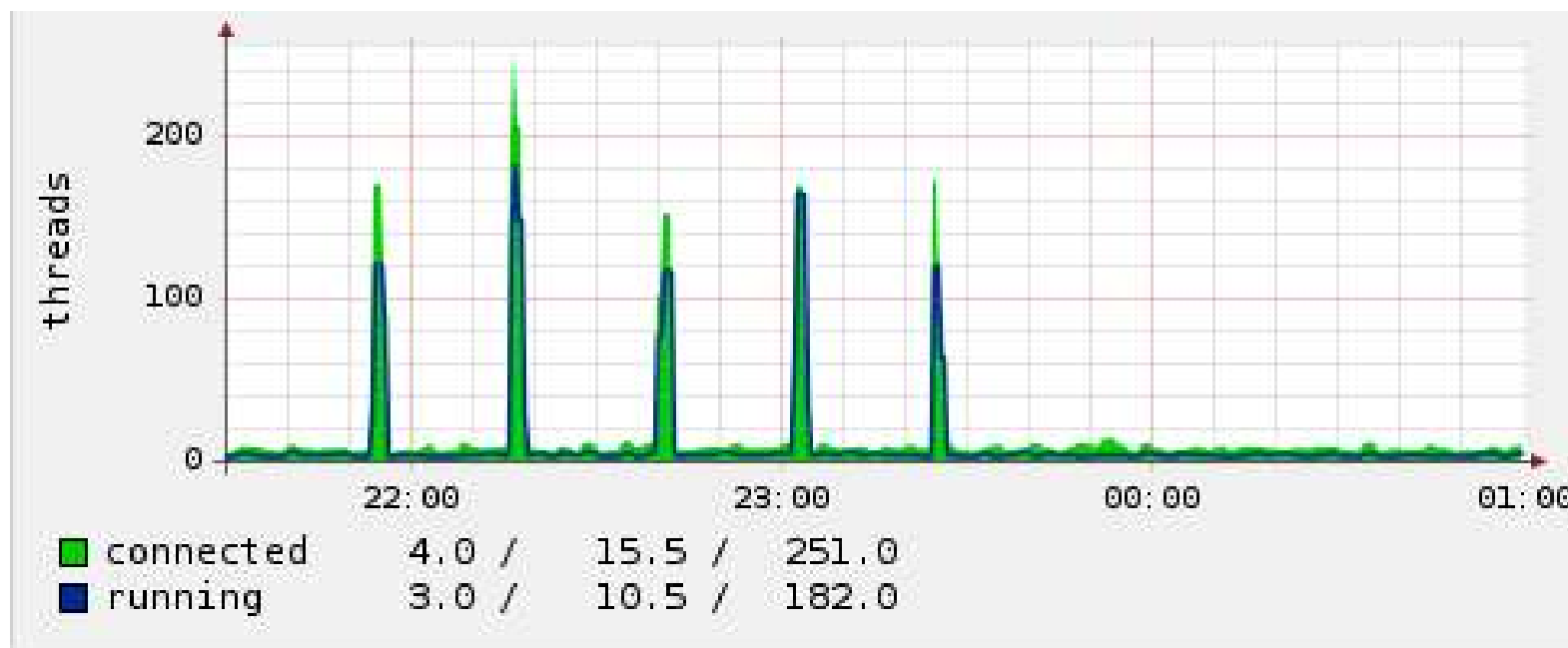
Avoiding stalls(2)

- **Mainly caused by database internal problems**
  - Some of them are known issues. Follow best practices
  - Improved every day
  - Don't use too old MySQL versions (4.0/4.1, 5.0.45, etc..)

- **Typical stalls**
  - Dropping a huge table (LOCK_open mutex)
  - Burst write (at checkpoint, at the time when redo log file free space becomes not enough)
  - pthread_create()/clone() (called at connection establishment)
  - etc

Handling Real−World workloads..

DeNA

■ Company Introduction: DeNA and Mobage Platform

- ● One of the largest social game providers in Japan
  - – Both social game platform and social games themselves
  - – Subsidiary ngmoco:) in SF
- ● Japan localized phone, Smart Phone, and PC games
- ● 2-3 billion page views per day
- ● 25+ million users
- ● 1.3B$ revenue in 2010

RDBMS or NoSQL ?

- **Is MySQL good for social games ?**
  - It's good!
  - DeNA uses MySQL for data stores, memcached for caching summary/objects
  - H/W is getting fast enough for RDBMS to handle lots of workloads (Nehalem CPU, Large RAM, PCI-Express SSD)
  - Complex query, multiple columns, transaction, secondary index, online backup, monitoring, utility tools are still very helpful for social games

- **Is RDBMS slow (Can't MySQL handle 1000 updates/second) ?**
  - Some of our MySQL servers handle 10,000+ UPDATE statements per second
  - Highly depending on random disk i/o speed and indexing
  - Any NoSQL database can't be fast if it becomes disk i/o bound

- **Schema, transaction and indexes should be more taken care**

Social Game workloads

DeNA

- **Easily increasing millions of users in a few days**
  - Database size grows rapidly
    - Especially if PK is "user_id + xxx_id"
    - Increasing GB/day is typical

- **Load balancing reads is not difficult**
  - Adding slaves or caching servers

- **Load balancing writes is not trivial**
  - Sharding

- **Solutions depend on what kinds of tables we're using, INSERT/UPDATE/DELETE workloads, etc**

INSERT-mostly tables

- **History tables such as access logs, diary, battle history**
  - INSERT and SELECT mostly
  - Table size becomes huge (easily exceeding 1TB)
  - Locality (Most of SELECT go to recent data)

- **INSERT performance in general**
  - Fast in InnoDB (Thanks to Insert Buffering. Much faster than MyISAM)
  - To modify index leaf blocks, they have to be in buffer pool
  - When index size becomes too large to fit in the buffer pool, disk reads happen
  - In-memory workloads -> disk-bound workloads
    - Suddenly suffering from serious performance slowdown
    - UPDATE/DELETE/SELECT also getting much slower
 - Any faster storage devices can not compete with in-memory workloads

InnoDB Feature: Insert Buffering
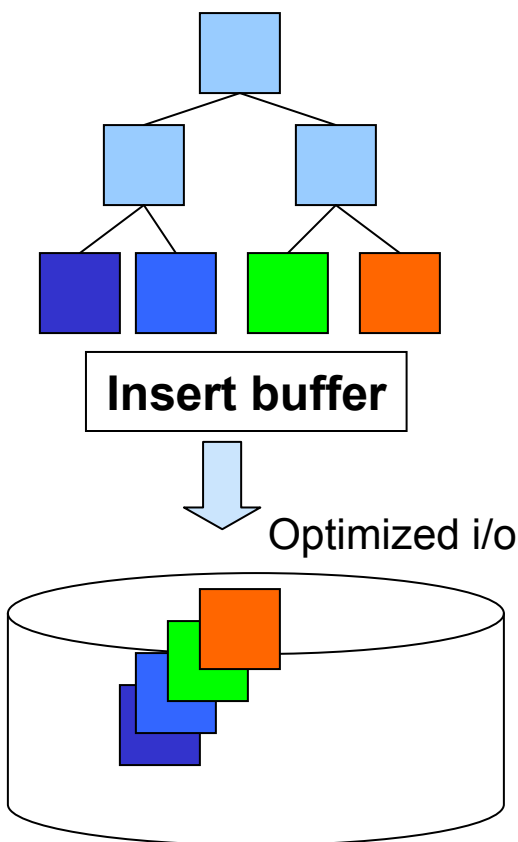
- If non-unique, secondary index blocks are not in memory, InnoDB inserts entries to a special buffer("insert buffer") to avoid random disk i/o operations
  - Insert buffer is allocated on both memory and innodb SYSTEM tablespace

- Periodically, the insert buffer is merged into the secondary index trees in the database ("merge")

- Pros: Reducing I/O overhead
  - Reducing the number of disk i/o operations by merging i/o requests to the same block
  - Some random i/o operations can be sequential

- Cons:
  Additional operations are added
  Merging might take a very long time
  - when many secondary indexes must be updated and many rows have been inserted.
  - it may continue to happen after a server shutdown and restart

Insert buffer

Optimized i/o

INSERT gets slower



Time to insert 1 million records (InnoDB, HDD)
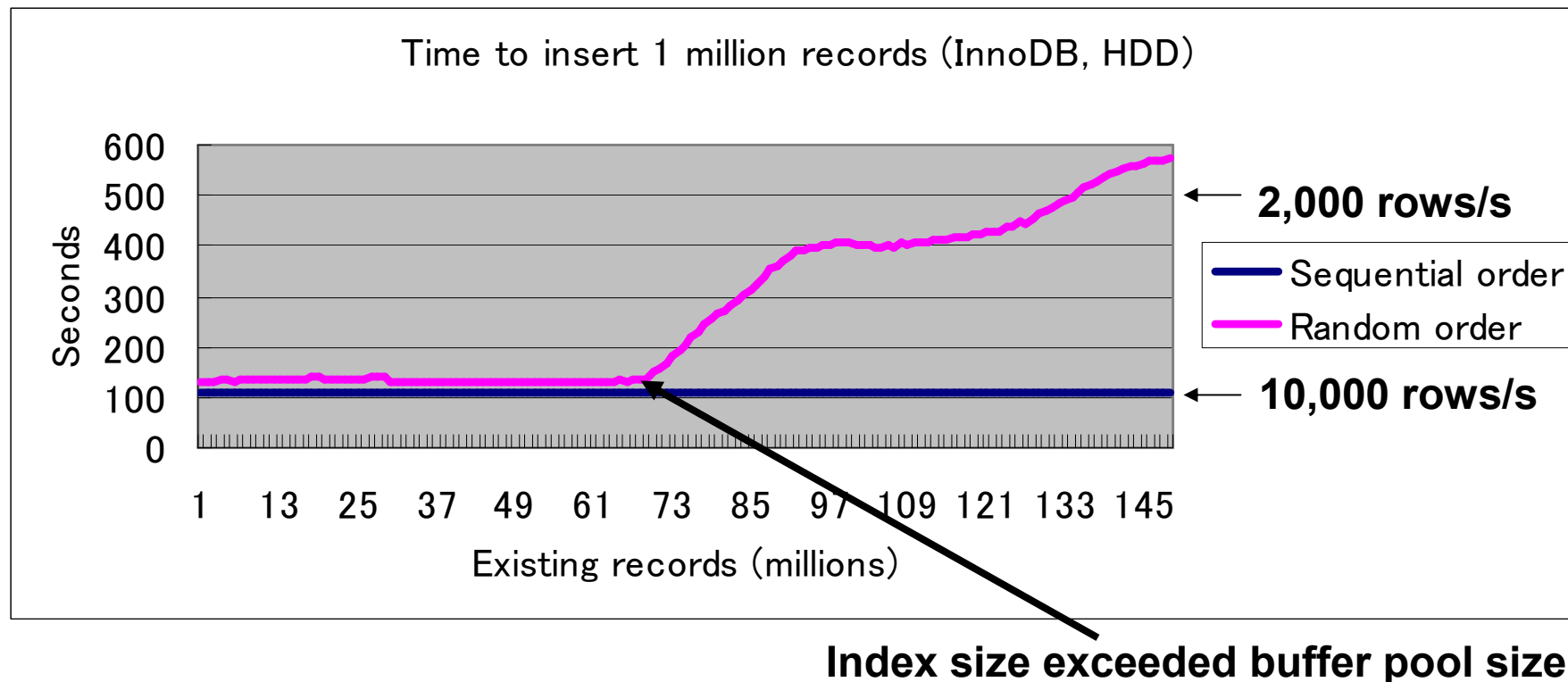
- Index size exceeded innodb buffer pool size at 73 million records for random order test
- Gradually taking more time because buffer pool hit ratio is getting worse (more random disk reads are needed)
- For sequential order inserts, insertion time did not change. No random reads/writes

INSERT performance difference

- **In-memory INSERT throughput**
  - 15000+ insert/s from single thread on recent H/W

- **Exceeding buffer pool, starting disk reads**
  - Degrading to 2000-4000 insert/s on HDD, single thread
  - 6000-8000 insert/s on multi-threaded workloads

- **Serious replication delay often happens**

- **Faster storage does not solve everything**
  - At most 5000 insert/s on fastest SSDs such as tachIOn/FusionIO
    - InnoDB actually uses CPU resources quite a lot for disk i/o bound inserts (i.e. calculating checksum, malloc/free)

- **It is important to minimize index size so that INSERT can complete in memory**

Approach to complete INSERT in memory

Single big physical table(index)

Partition 1 Partition 2

Partition 3 Partition 4
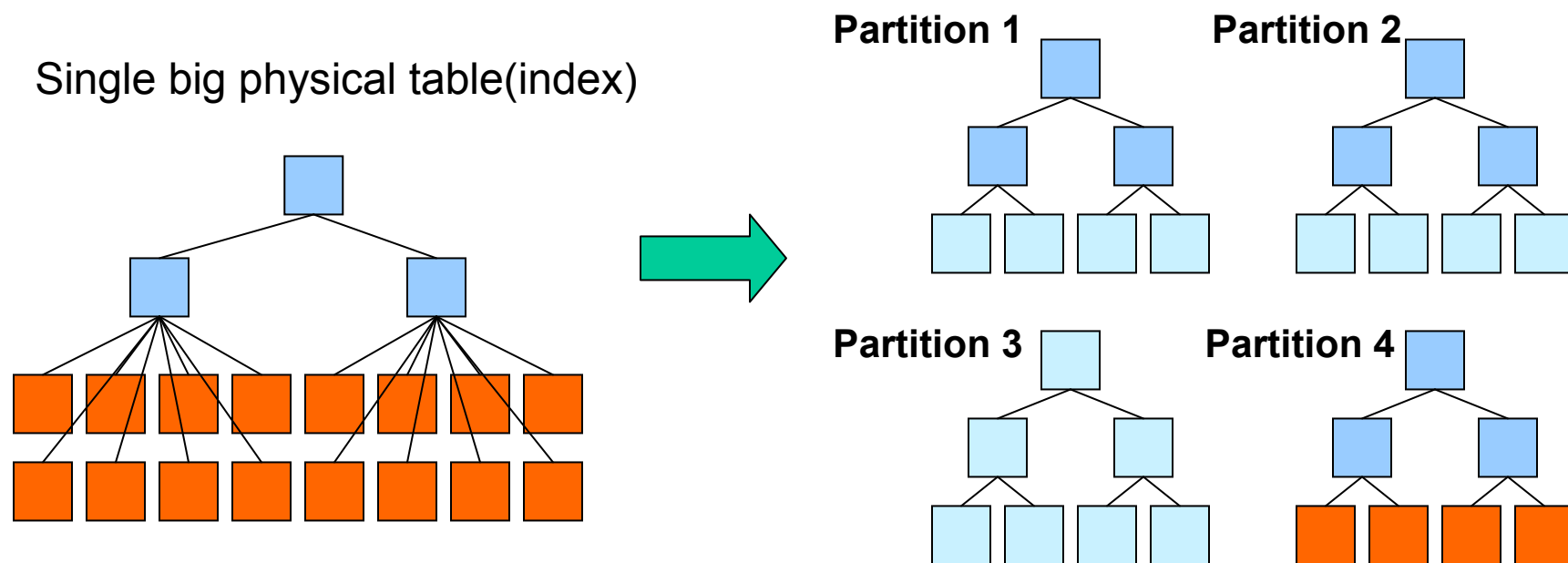
- ■ Range partition by datetime
  - ● Started from MySQL 5.1
  - ● Index size per partition becomes total_index_size / number_of_partitions
  - ● INT or TIMESTAMP enables hourly based partitions
    - – TIMESTAMP does not support partition pruning
  - ● Old partitions can be dropped by ALTER TABLE .. DROP PARTITION

Approaches to complete INSERT in memory

- Dividing tables by datetime (up to 5.0, or if your tables are too big to store on single server's disks)
- Purging or archiving older data
- Dropping indexes
- Dropping columns
- Using space efficient data types, compression
- Using large RAM
- Sharding

UPDATE-mostly tables

- ■ Typical usage
  - ● Updating users' status
    - – Current HP, experiences, money, number of items, friends' status, battle status, etc
    - – UPDATE and SELECT are most frequently executed for handling user status

- ■ UPDATE performance
  - ● Need to read target record blocks and index blocks
    - – Fast if completed in buffer pool, otherwise massive foreground disk reads happen
    - – Data size does not grow significantly, depending on INSERT/DELETE
  - ● Huge performance difference between storage devices
    - – In-memory UPDATE: 12,000/s
    - – HDD UPDATE: 300/s
    - – SATA SSD UPDATE: 1,800/s
    - – PCI-E SSD UPDATE: 4,000/s
    - – * Single Thread
    - – Random reads happen in foreground, so random read speed matters a lot

■ SSD performance and deployment strategies for MySQL

What do you need to consider? (H/W layer)

- **SSD or HDD?**
- **Interface**
  - SATA/SAS or PCI-Express?, How many drives?
- **RAID**
  - H/W RAID, S/W RAID or JBOD?
- **Network**
  - Is 100Mbps or 1Gbps enough?
- **Memory**
  - Is 2GB RAM + PCI-E SSD faster than 64GB RAM + 8HDDs?
- **CPU**
  - Nehalem, Opteron or older Xeon?

What do you need to consider?

- **Redundancy**
  - RAID
  - DRBD (network mirroring)
  - Semi-Sync MySQL Replication
  - Async MySQL Replication

- **Filesystem**
  - ext3, xfs, raw device ?

- **File location**
  - Data file, Redo log file, etc

- **SSD specific issues**
  - Write performance deterioration
  - Write endurance

Why SSD? IOPS!

- IOPS: Number of (random) disk i/o operations per second

- Almost all database operations require random access
  - Selecting records by index scan
  - Updating records
  - Deleting records
  - Modifying indexes

- Regular SAS HDD : 200 iops per drive (disk seek & rotation is slow)

- SSD : 2,000+ (writes) / 5,000+ (reads) per drive
  - highly depending on SSDs and device drivers
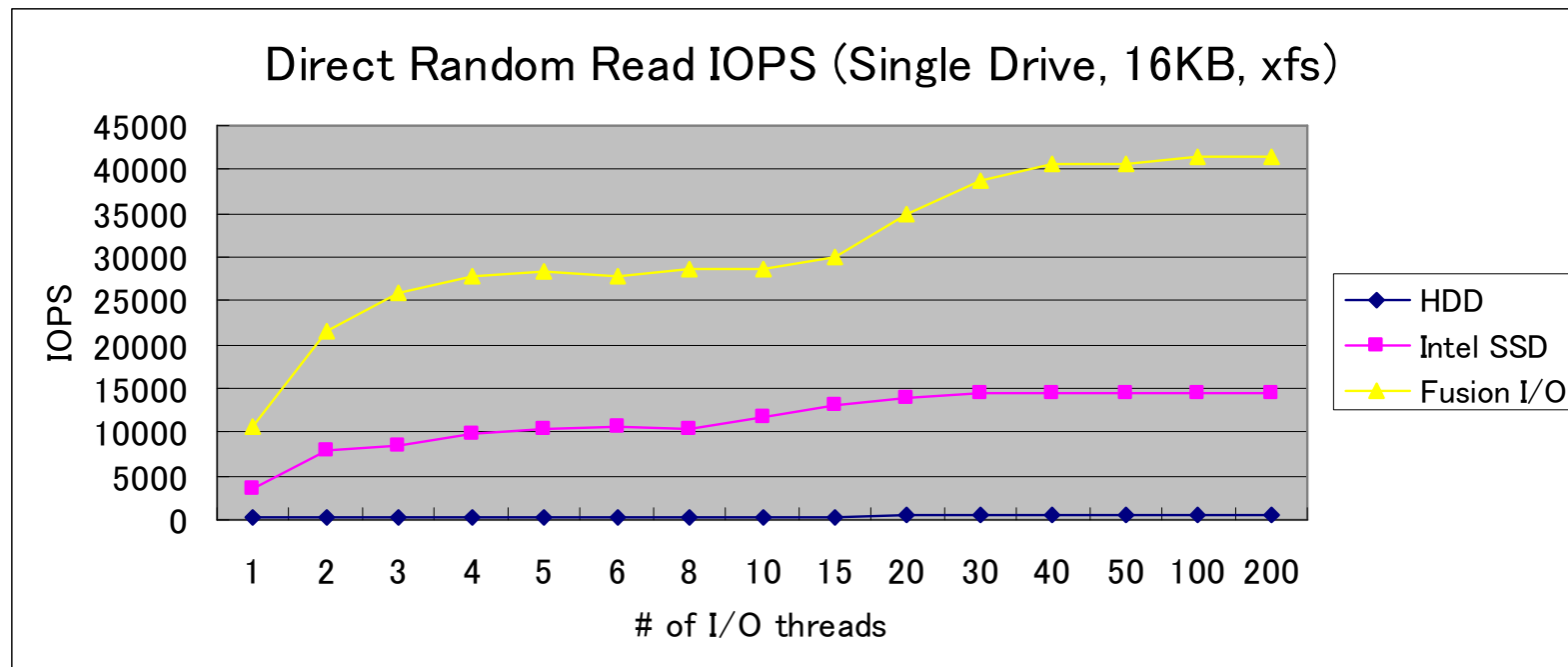
Table of contents

- **Basic Performance on SSD/HDD**
  - Random Reads
  - Random Writes
  - Sequential Reads
  - Sequential Writes
  - fsync() speed
  - Filesystem difference
  - IOPS and I/O unit size
  - SLC vs MLC
  - # of interfaces/drives
  - Opteron vs Nehalem

- **MySQL Deployments for benchmarks**
- **MySQL Deployments for real workloads**

Random Read benchmark

Direct Random Read IOPS (Single Drive, 16KB, xfs)

- HDD: 196 reads/s at 1 i/o thread, 443 reads/s at 100 i/o threads
- Intel : 3508 reads/s at 1 i/o thread, 14538 reads/s at 100 i/o threads
- Fusion I/O : 10526 reads/s at 1 i/o thread, 41379 reads/s at 100 i/o threads
- Single thread throughput on Intel is 16x better than on HDD, Fusion is 25x better
- SSD's concurrency (4x) is much better than HDD's (2.2x)
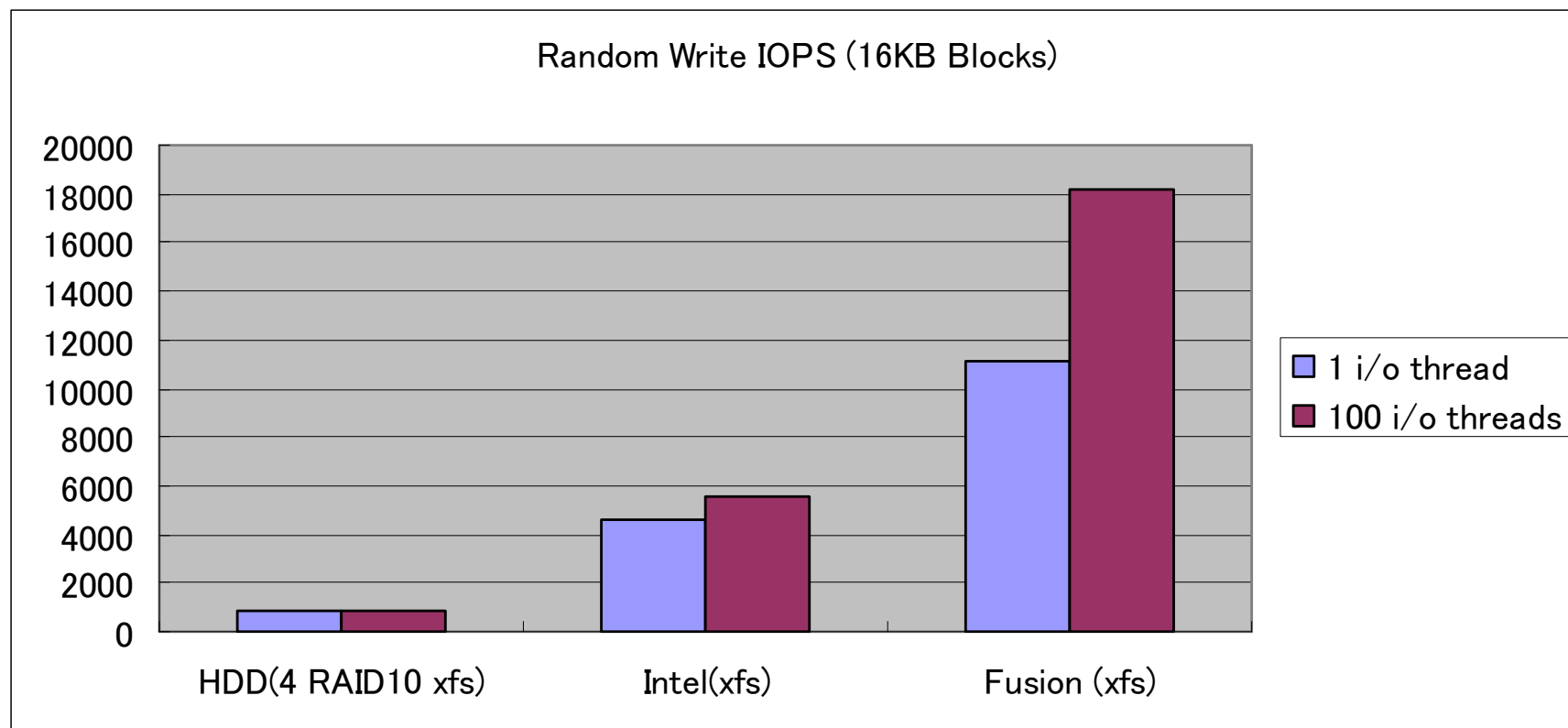- Very strong reason to use SSD

PCI-Express SSD

```
           ┌─────────────────┐
           │       CPU       │
           └────────┬────────┘
                    │
   ┌────────────────┴──┐          ┌───────────────────┐
   │   North Bridge    │──────────│   South Bridge    │
   │ PCI-Express Controller │     │  SAS/SATA Controller │
   └─────────┬─────────┘          └─────────┬─────────┘
             ↕                              ↕
   2GB/s (PCI-Express x 8)                300MB/s
   ┌─────────────────┐          ┌─────────────────┐
   │ SSD I/O Controller │       │ SSD I/O Controller │
   └────────┬────────┘          └────────┬────────┘
         ┌──┴──┐                      ┌──┴──┐
         │Flash│                      │Flash│
         └─────┘                      └─────┘
```
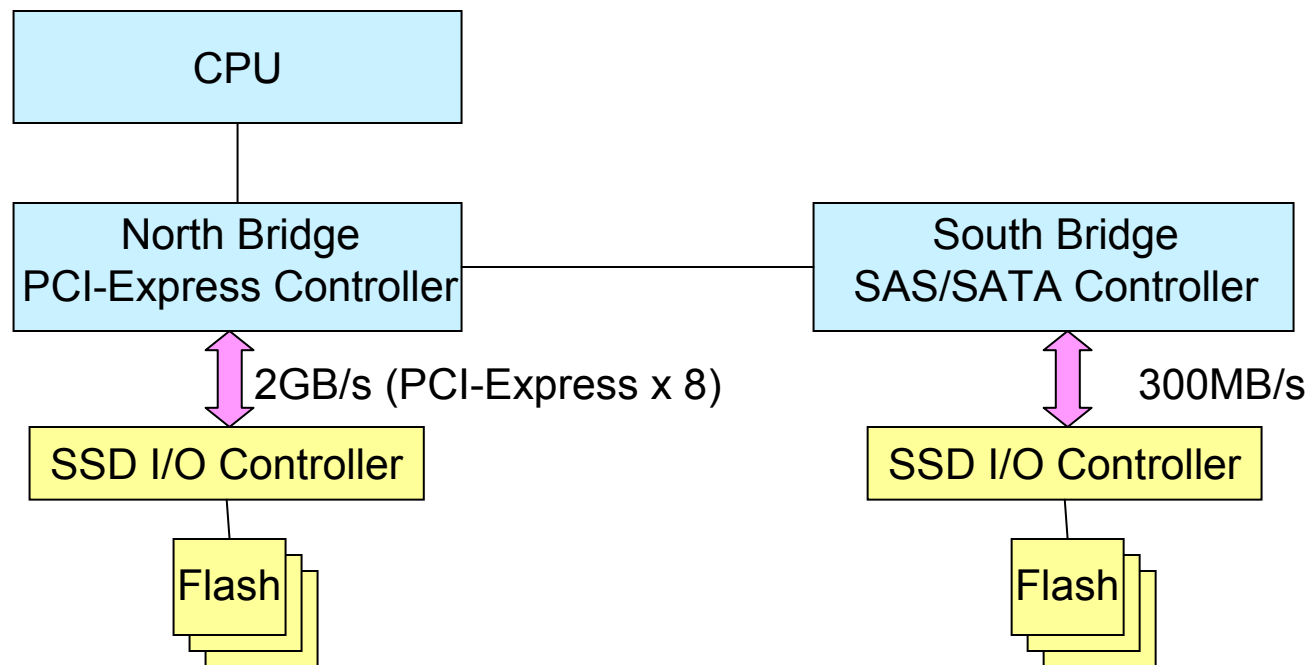
- ■ Advantage
  - ● PCI-Express is much faster interface than SAS/SATA

- ■ (current) Disadvantages
  - ● Most motherboards have limited # of PCI-E slots
  - ● No hot swap mechanism

Write performance on SSD

Random Write IOPS (16KB Blocks)

- ■ Very strong reason to use SSD
- ■ But wait.. Can we get a high write throughput *everytime*?
  - ● Not always.. Let's check how data is written to Flash Memory

Write performance deterioration

Write IOPS deterioration (16KB random write)

Continuous write-intensive workloads

Stopping writing for a while

Legend: Fastest, Slowest

Y-axis: IOPS (0, 5000, 10000, 15000, 20000, 25000, 30000)

X-axis: Intel, Fusion(150G), Fusion(120G), Fusion(96G), Fusion(80G)
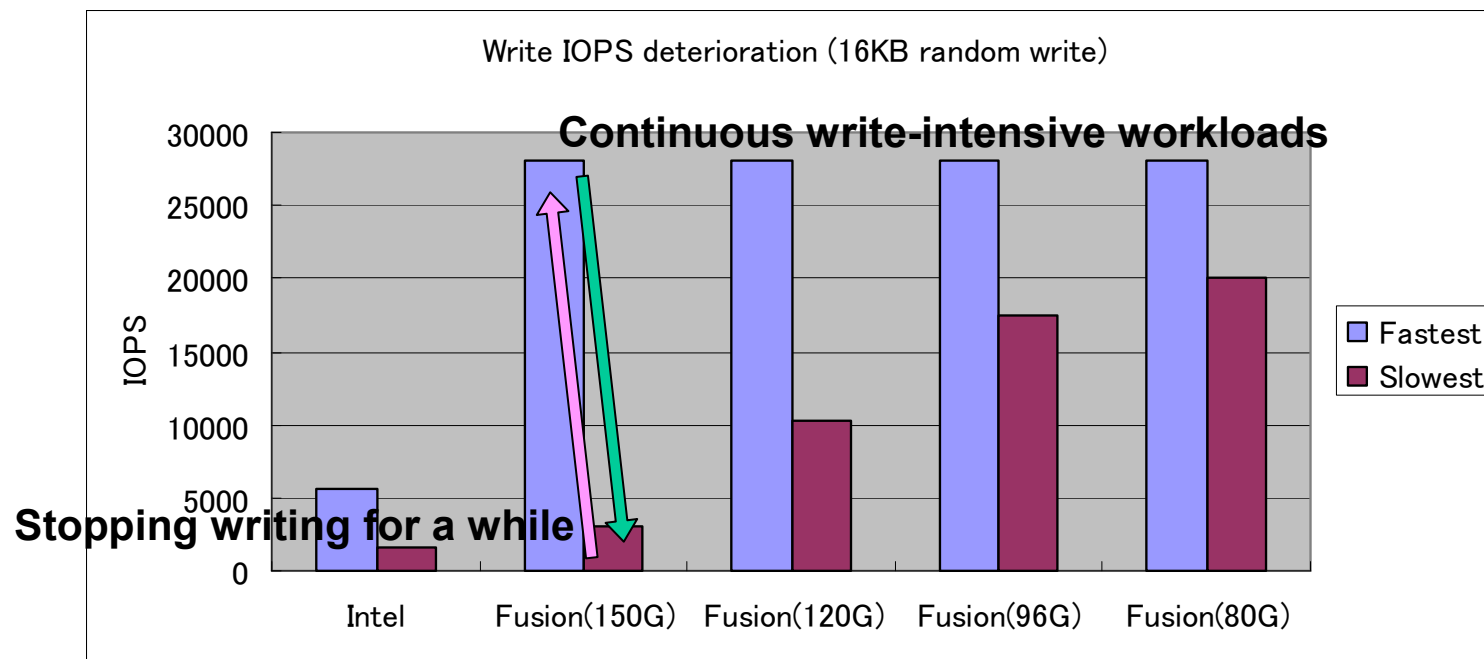
- At the beginning, write IOPS was close to "Fastest" line
- When massive writes happened, write IOPS gradually deteriorated toward "Slowest" line (because massive ERASE happened)
- Increasing reserved space improves steady-state write throughput
- Write IOPS recovered to "Fastest" when stopping writes for a long time (Many blocks were ERASEd by background job)
- Highly depending on Flash memory and I/O controller (TRIM support, ERASE scheduling, etc)

Mitigating write performance deterioration

■ **Allocating more reserved space**
- Per space pricing increases. This is not always acceptable

■ **Using tachIOn**
- tachIOn is highly optimized for keeping write performance higher

■ **Using Software RAID0**
- Write traffic is distributed to multiple drives. You can gain high enough write throughput even though additional reserved space is not allocated
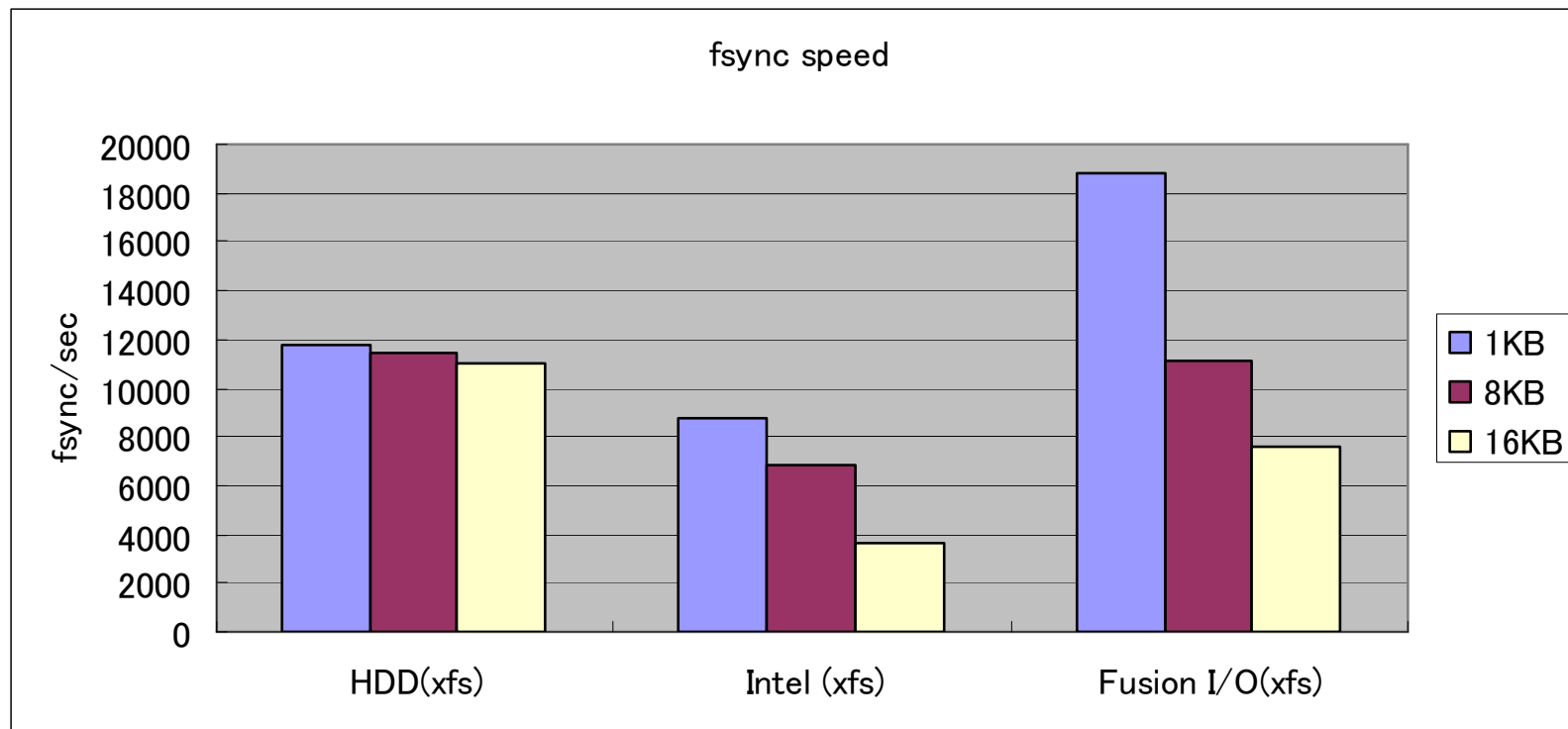
Sequential I/O

Sequential Read/Write throughput (1MB consecutive reads/writes)
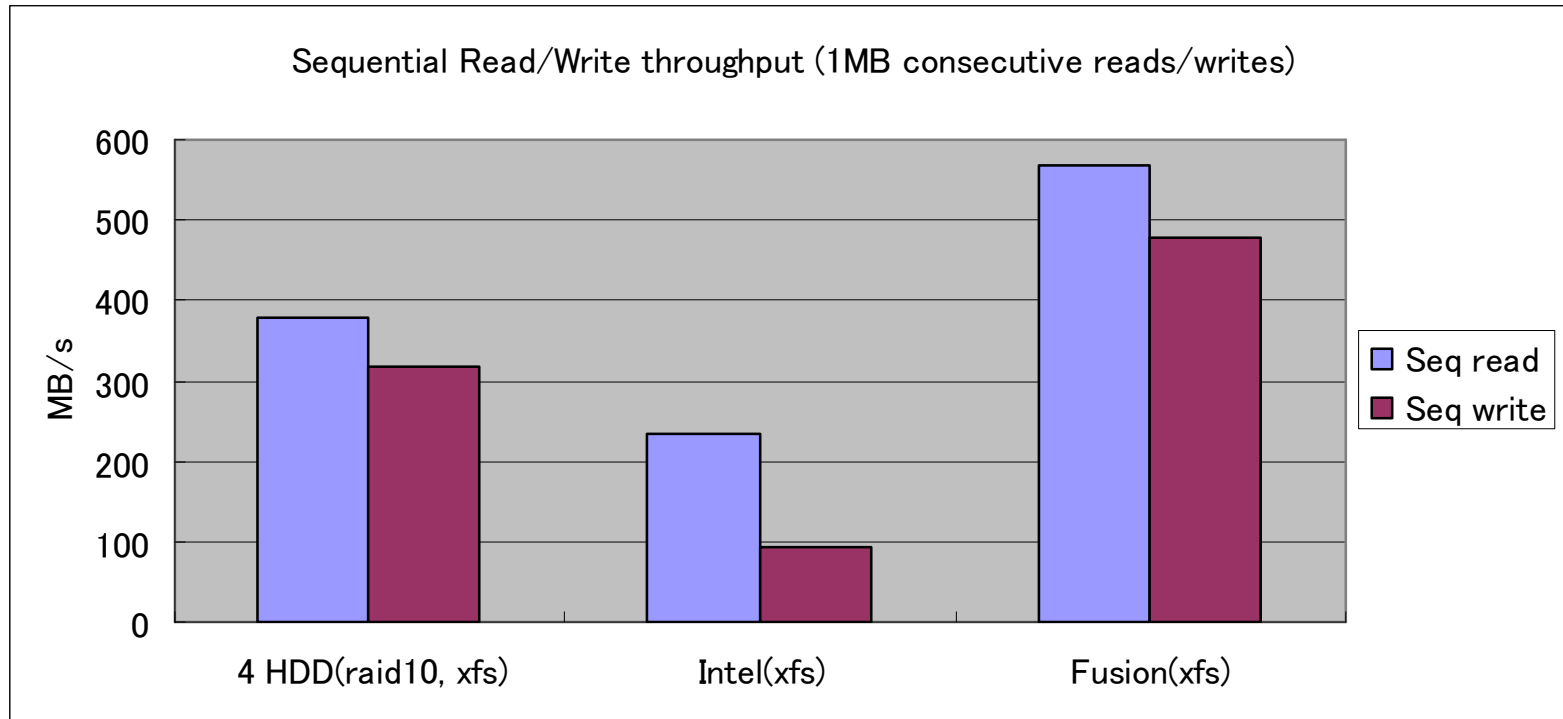
- Typical scenario: Full table scan (read), logging/journaling (write)
- SSD outperforms HDD for sequential reads, but less significant
- HDD (4 RAID10) is fast enough for sequential i/o
- Data transfer size by sequential writes tends to be huge, so you need to care about write deterioration on SSD
- No strong reason to use SSD for sequential writes

fsync() speed

fsync speed

- 10,000+ fsync/sec is fine in most cases
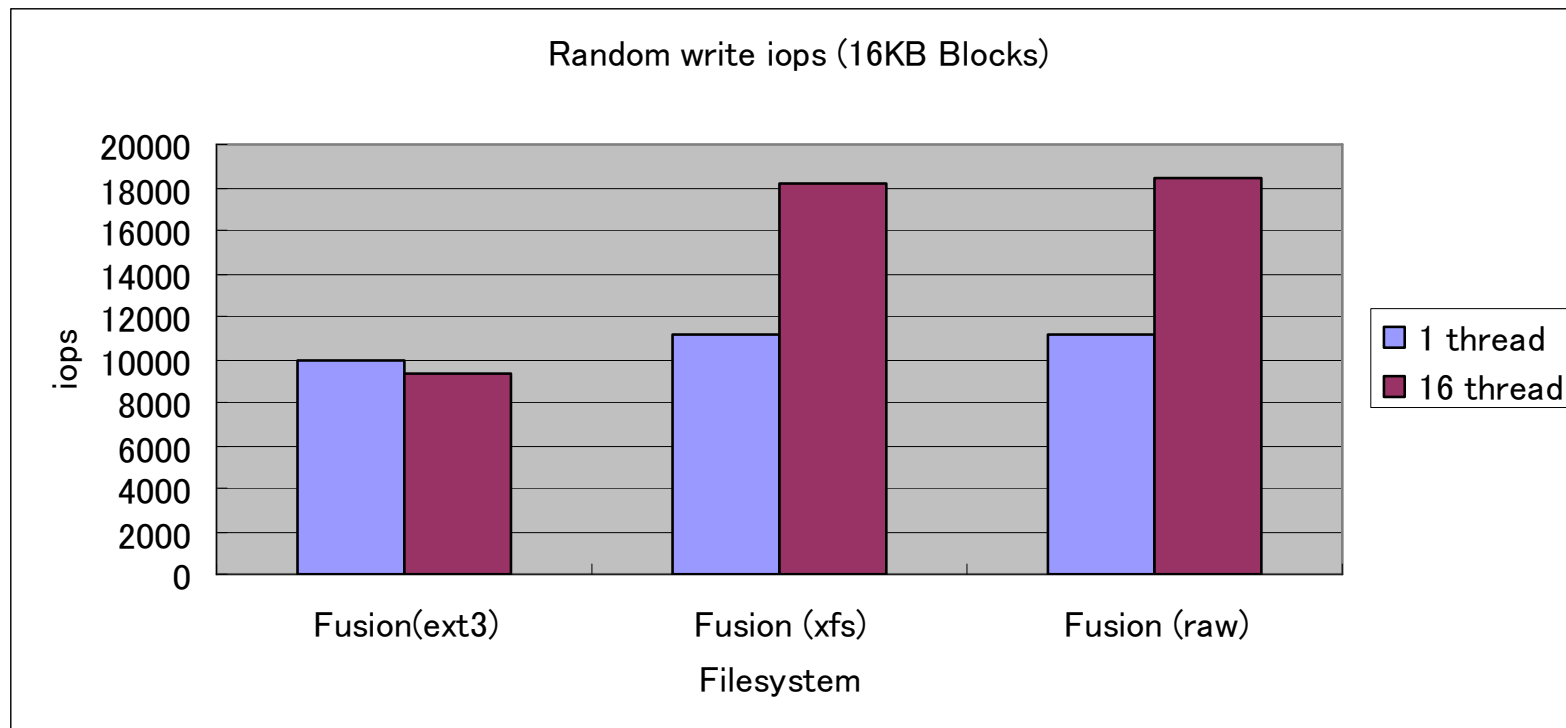- Fusion I/O was CPU bound (%system), not I/O bound (%iowait).

HDD is fast for sequential writes / fsync

- Best Practice: Writes can be boosted by using BBWC/FBWC (Battery Backed up Write Cache), especially for REDO Logs (because it's sequentially written)
- No strong reason to use SSDs here

seek & rotation time

disk

Write cache

disk

seek & rotation time

Filesystem matters
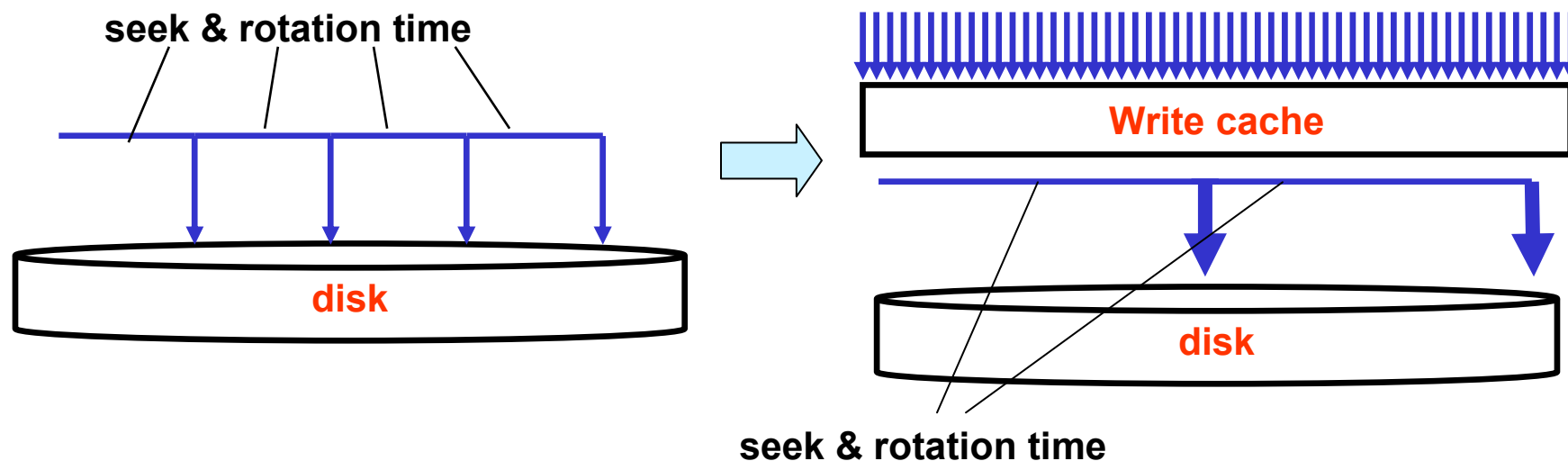
Random write iops (16KB Blocks)
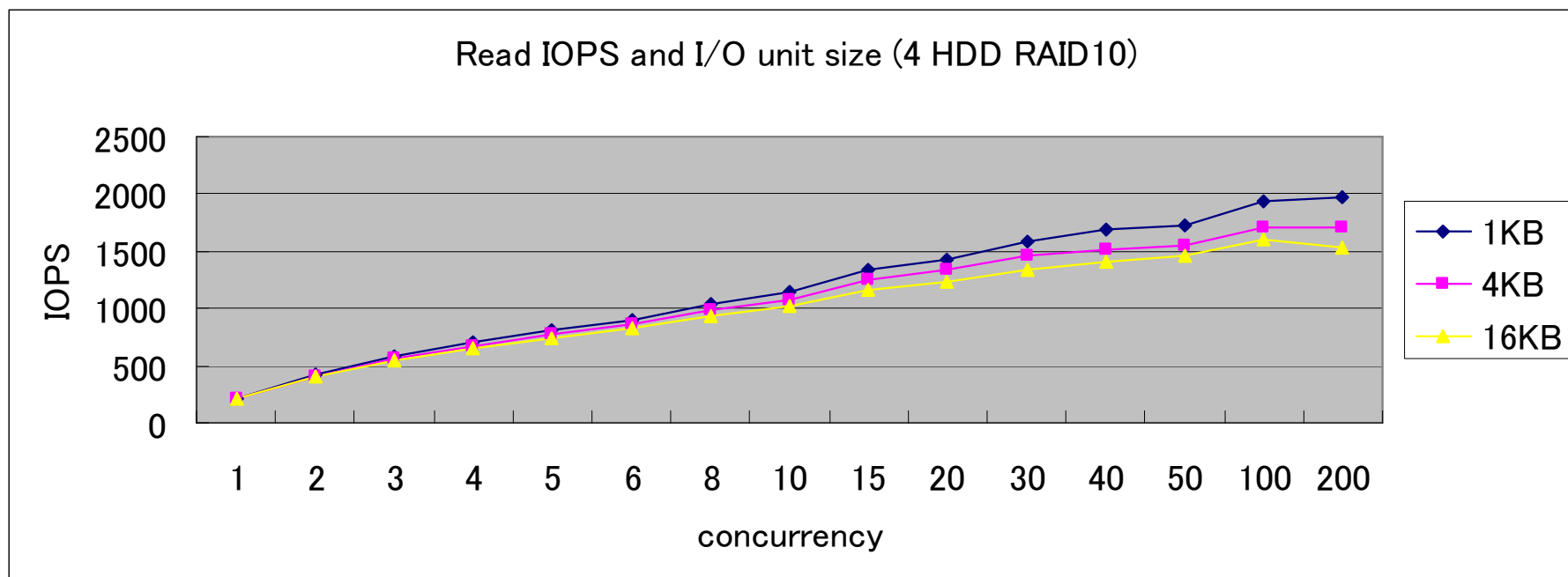

- On xfs, multiple threads can write to the same file if opened with O_DIRECT, but can not on ext*
- Good concurrency on xfs, close to raw device
- ext3 is less optimized for Fusion I/O

Changing I/O unit size

Read IOPS and I/O unit size (4 HDD RAID10)

- On HDD, maximum 22% performance difference was found between 1KB and 16KB
- No big difference when concurrency < 10

Changing I/O unit size on FusionIO

Random Read IOPS (FusionIO, SLC)
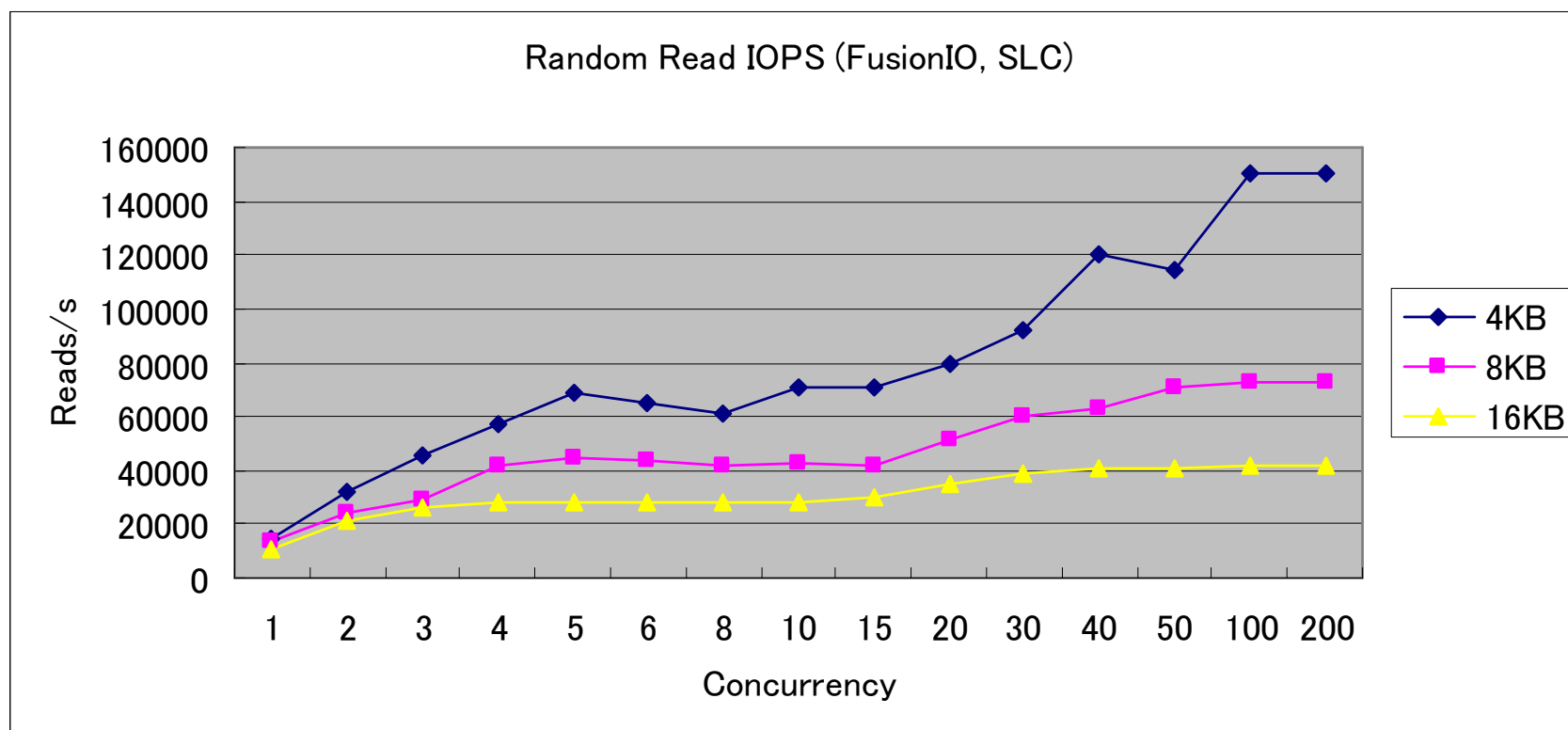
- Huge difference
- On SSDs, not only IOPS, but also I/O transfer size matters
- It's worth considering that Storage Engines support "configurable block size" functionality
  - Check InnoDB lab release

SLC vs MLC（16KB）

Random Read IOPS, FusionIO（16KB）
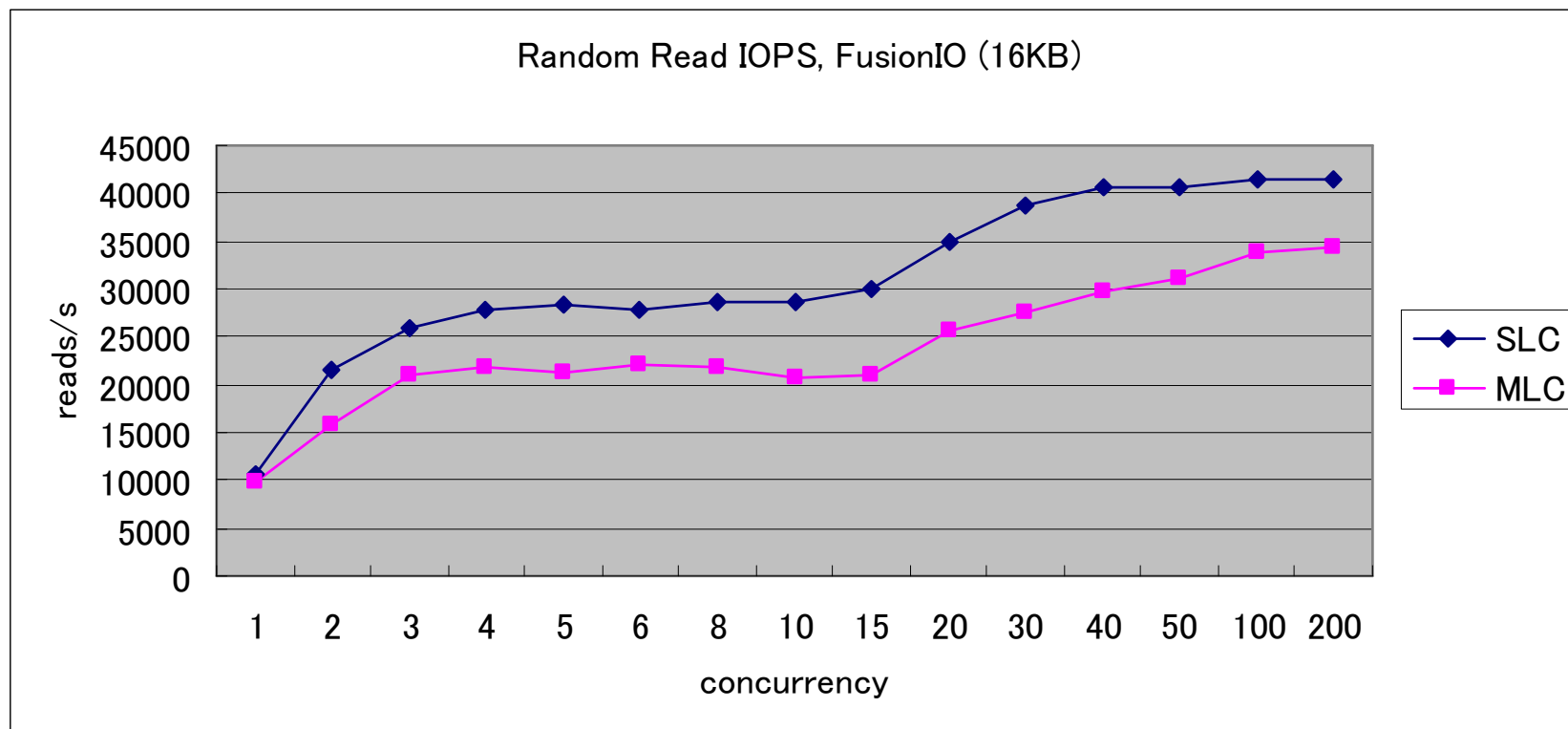
- 8-40% better throughput on SLC

SLC vs MLC （8KB）

Random Read IOPS, FusionIO （8KB）
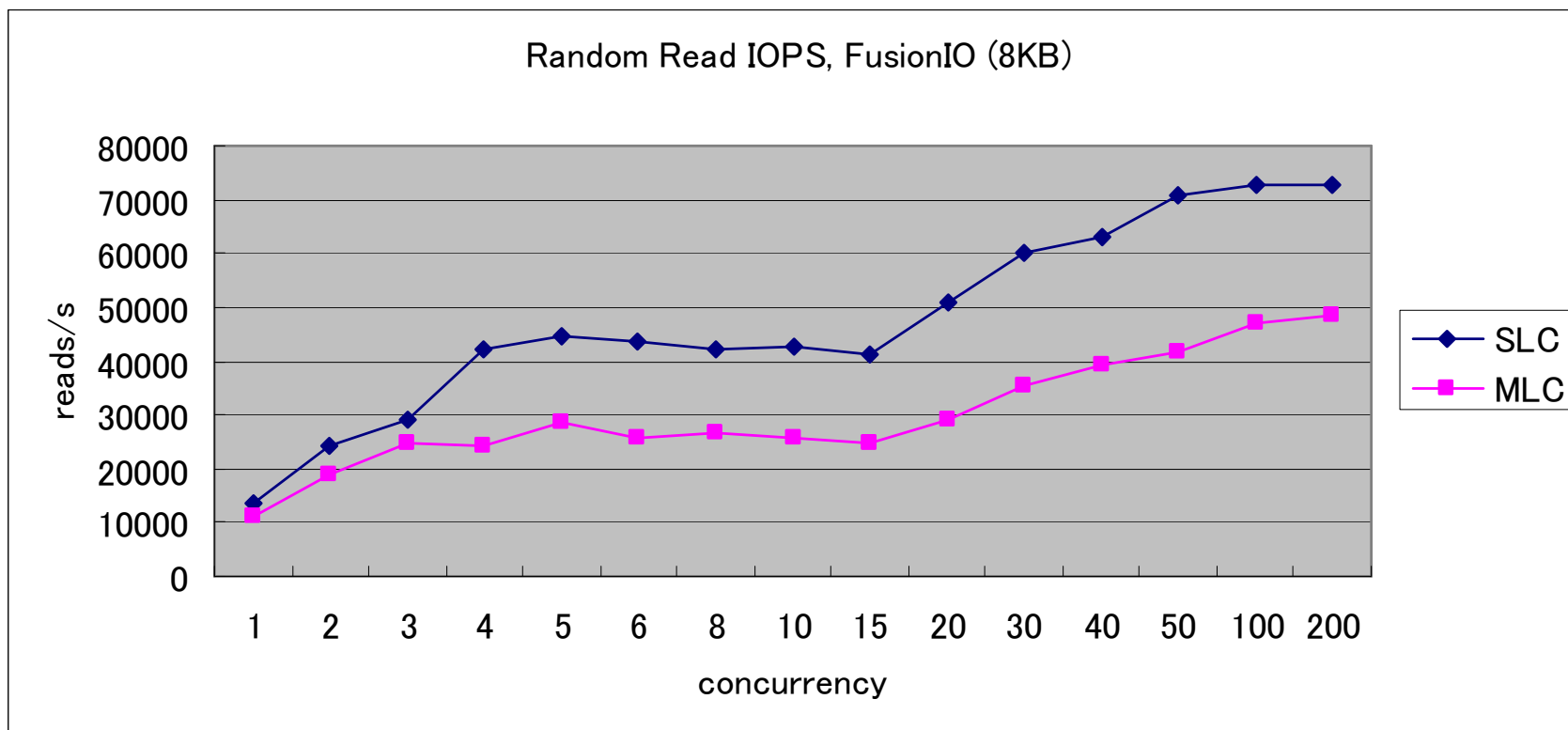
- 25-75% better throughput on SLC
- Random I/O benchmarks should be done carefully (SLC or MLC, I/O unit size, etc..)

tachIOn vs FusionIO (SLC)

Random Read IOPS (16KB)
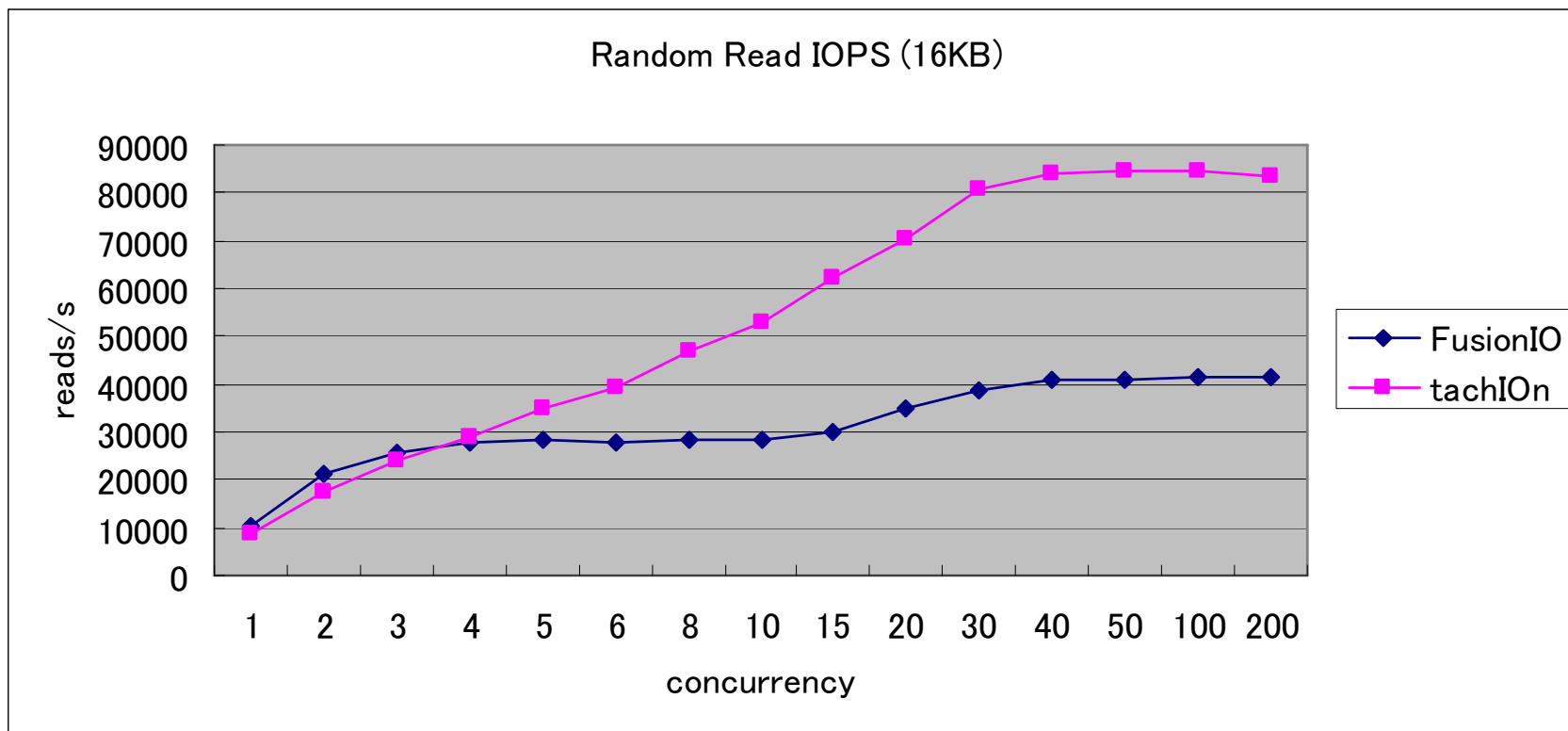
■ tachIOn performs very well, especially when concurrency is high enough

PCI-Express interface and CPU util

```
# cat /proc/interrupts | grep PCI
 83:          ... PCI-MSI     vgcinit
202:          ... PCI-MSI-X   eth2-0
210:          ... PCI-MSI-X   eth2-1
218:          ... PCI-MSI-X   eth2-2
226:          ... PCI-MSI-X   eth2-3
234:          ... PCI-MSI-X   eth2-4
```

```
# mpstat -P ALL 1
CPU    %user    %nice    %sys %iowait     %irq    %soft    %idle      intr/s
all     0.45     0.00    7.75   26.69     1.65     0.00    63.45    40046.40
  0     1.00     0.00   12.60   86.40     0.00     0.00     0.00     1000.20
  1     1.00     0.00   13.63   85.37     0.00     0.00     0.00        0.00
  2     0.40     0.00    4.80   26.80     0.00     0.00    68.00        0.00
  3     0.00     0.00    0.00    0.00    79.20     0.00    20.80    39033.20 ...
```

- ■ %irq (Hardware interrupts) was bottleneck
- ■ Only one IRQ port was allocated per single PCI-E SSD drive
- ■ By using multiple drives the number of IRQ ports increases

of interfaces (tachIOn SLC)

Random Read IOPS (16KB, tachIOn)
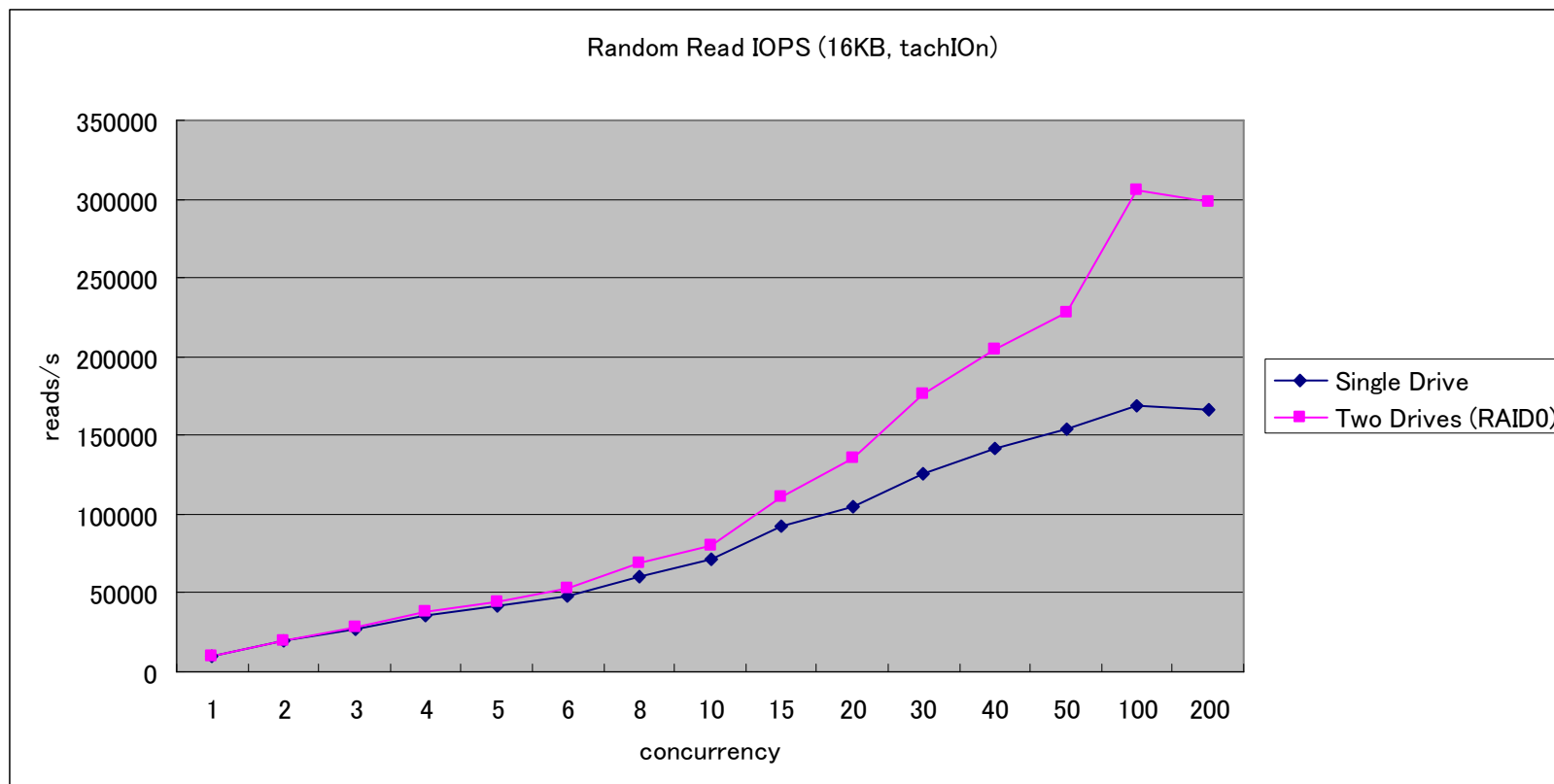
- Two drives result in nearly two times better throughput, if enough read i/o is coming
- When the number of clients was small (not enough i/o requests were coming), %irq was not so high, so using two drives was not so much helpful
- The number of slots are limited on most motherboards

of interfaces （FusionIO MLC）

Random Read IOPS （16KB）
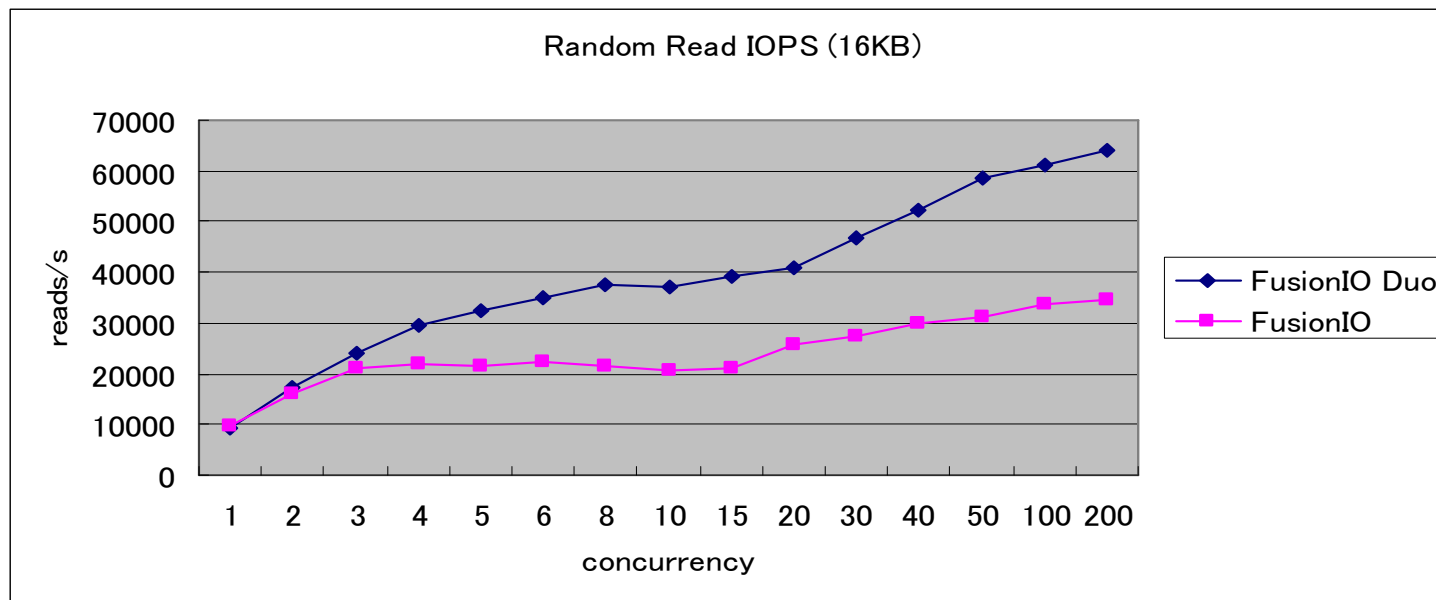
- FusionIO Duo internally has two drives per single PCI-Express connector
  - Two IRQ ports can be used, which greatly increases throughput
- A couple of restrictions
  - FusionIO Duo has two device files (/dev/fioa, /dev/fiob). Single large native filesystem can not be created
  - FusionIO Duo has physical(height/length) and electrical restrictions
    - Only one Duo drive can be installed on HP DL360 (two PCI-E ports) physically
    - On some servers, without an optional power cable, maximum performance can not be gained

tachIOn(SLC) vs FusionIO Duo(MLC)

Random Read IOPS (16KB)
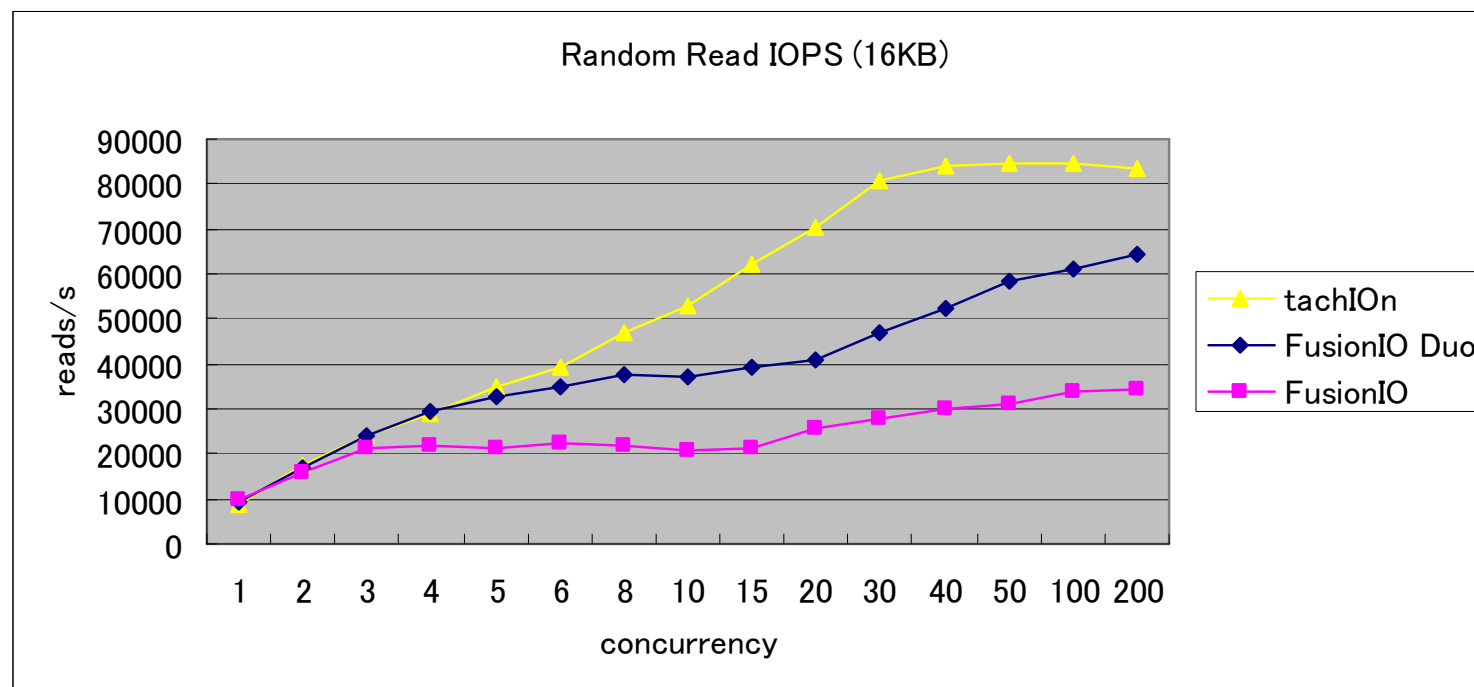
- Fusion IO Duo performs good enough, considering it's MLC

Opteron vs Nehalem(tachIOn)

tachIOn Random Read IOPS (16KB)
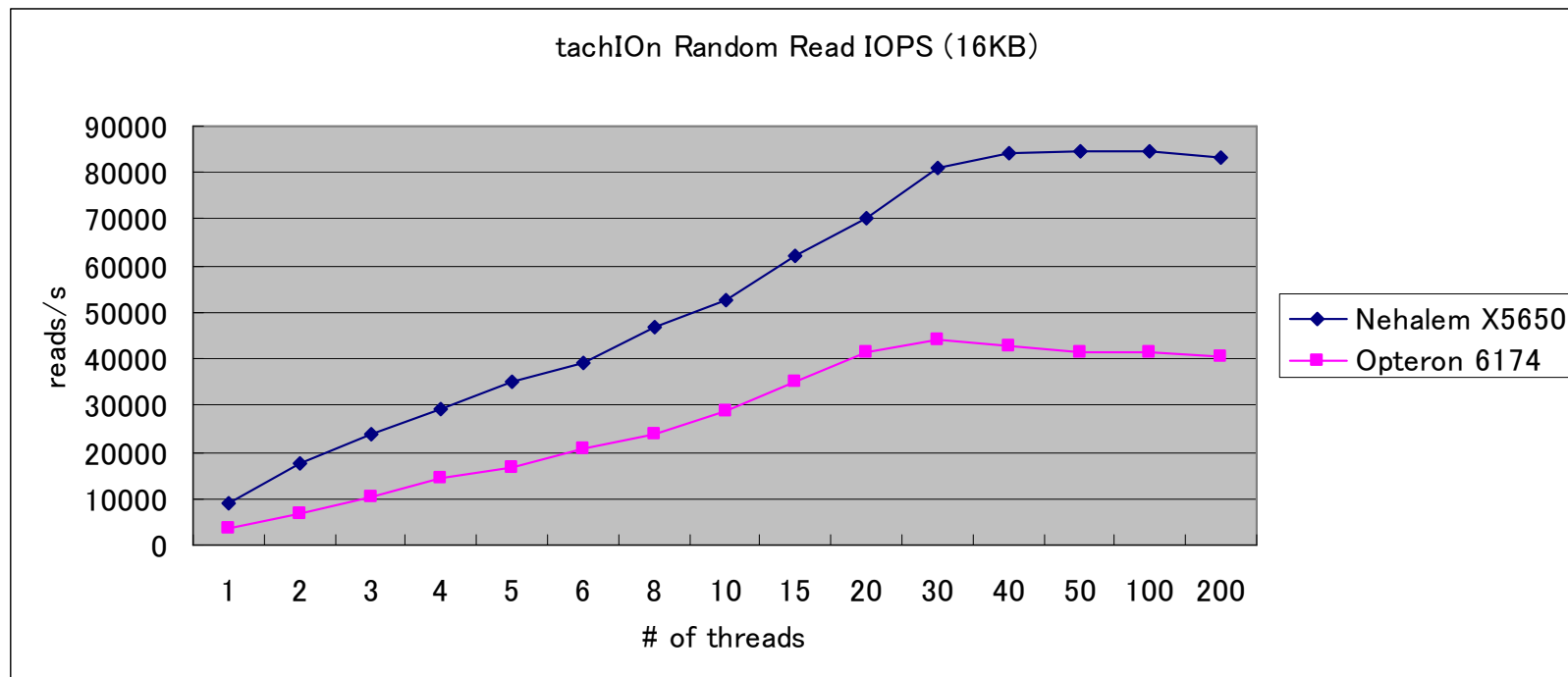
- ■ 2 times performance difference
  - ● Single thread IOPS is bad
  - ● Latency difference between PCI-E and CPU seems quite high

Opteron vs Nehalem(tachIOn)

tachIOn Random Read IOPS(4KB)

- Nehalem X5650
- Opteron 6174

(x-axis: # of threads — 1, 2, 3, 4, 5, 6, 8, 10, 15, 20, 30, 40, 50, 100, 200)
(y-axis: reads/s — 0, 50000, 100000, 150000, 200000, 250000, 300000)
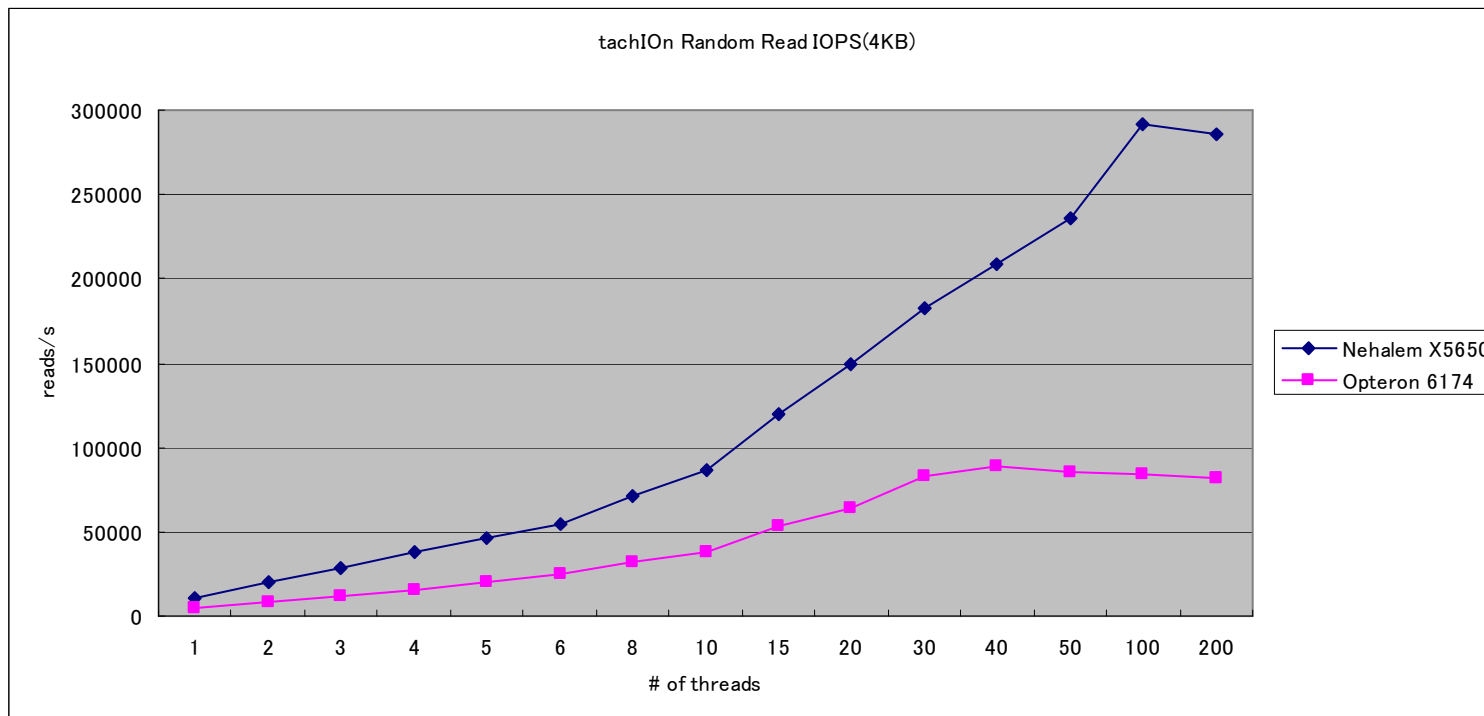
■ 2-4 times performance difference

● In both cases %irq was 100% used

● Handling Hardware Interrupts requires high CPU clock speed

Don't use old firmware

Random Read IOPS, 16KB
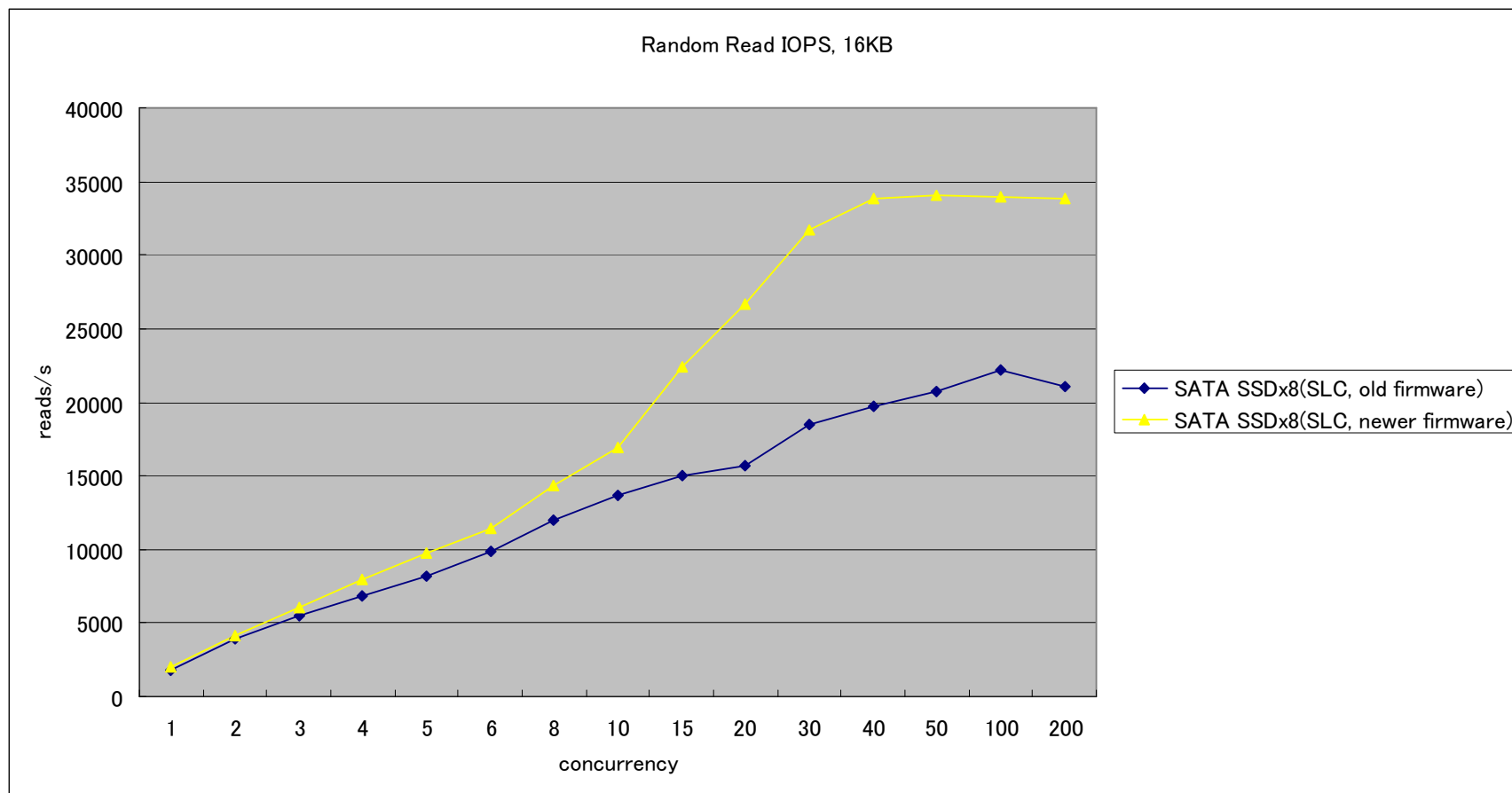
- Historically SATA/SAS SSD RAID performance was poor
- Recently it gets much better with newer firmware

SAS SSD

Random Read IOPS, 16KB
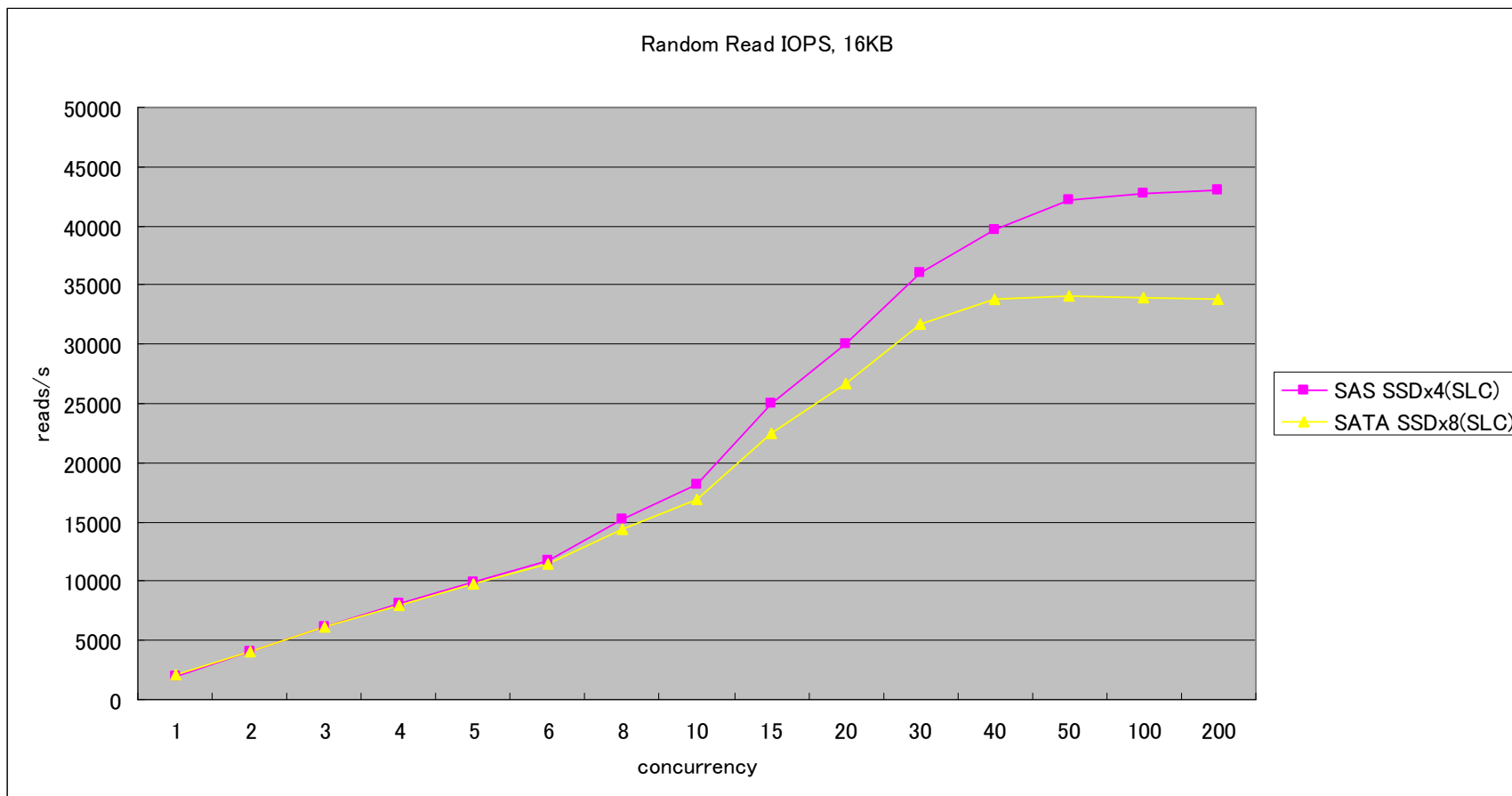
- SAS SSD: HP 300GB SAS drives x 8
  - Needs newer version of RAID firmware
- Good concurrency: IOPS on 30+ threads is competitive against FusionIO

SAS SSD vs PCIe SSD

Random Read IOPS (16KB)
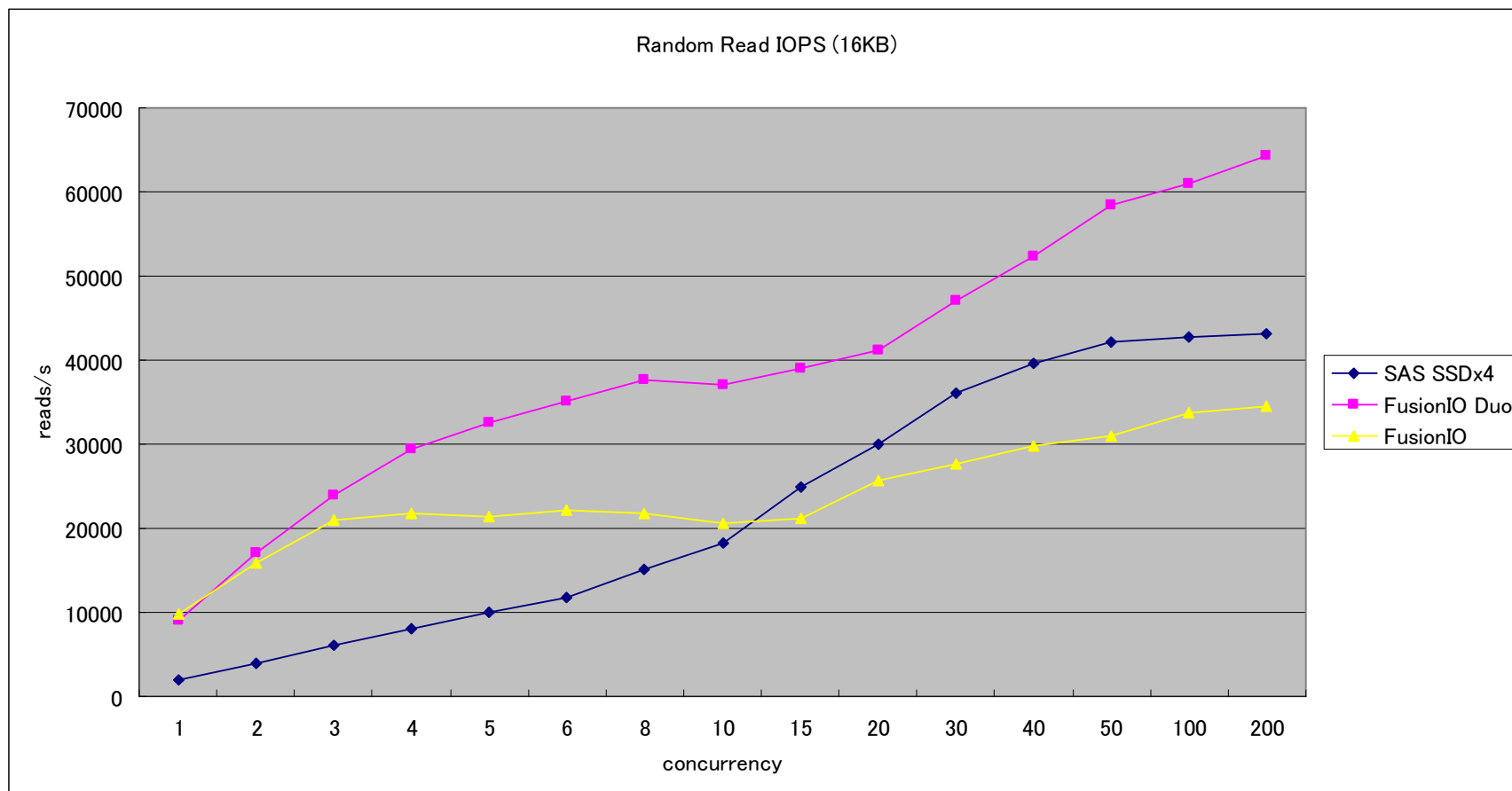
- Per GB pricing on SAS SSD can be less than half of PCIe SSD
- It can be practical to use SSD-only environment with SATA or SAS SSD, instead of using PCIe SSD as secondary cache

Single Thread read IOPS

	South Bridge		PCI-Express	
HDD	SATA SSD	SAS SSD	FusionIO	tachIOn
250	1,800	2,000	10,000	10,000

- There are still many single threaded components on MySQL, Linux and applications
  - Replication SQL Thread
  - Traditional batch jobs

- Even though SAS SSD shows competitive IOPS against PCI-E SSD on many threads, single threaded performance is still poor

- Slow SQL Thread results in replication delay. This is really serious

Boosting single threaded slave performance

- **Making it multi-threaded**
  - Parallel SQL threads in MySQL 5.6

- **Reducing foreground disk i/o**
  - Pre-fetching by other threads, warming up buffer pool
  - Then SQL Thread does not need disk reads, which can improve throughput a lot

- **Asynchronous I/O**
  - Change buffering in MySQL 5.5 is very helpful to improve delete performance because client(SQL Thread) does not have to wait for deleting secondary index entries
    - It's also helpful for UPDATE when it updates secondary indexes

New Project: Replication Booster

DeNA

MySQL Slave

Relay Log Files

relay-log.info
(file or table)

SQL Thread can be much faster
by higher cache hit rate

Application Tables

1. Reading raw relay log events
from current SQL thread pos
(via Binlog API)

Getting current SQL thread's pos
(every N milliseconds),
Checking SQL/IO threads are
Running, etc

c. Executing SELECT if the event pos
is newer than current SQL thread's pos

Main Thread
(Loop 1..3)

SQL Thread's Position
Tracking Thread

Prefetching Worker
Threads
(Loop a..c)

2. Dropping bogus events
(Other than query events,
"BEGIN", etc)

b. Converting UPDATE to SELECT

3. Pushing to queue

Query Event Queue

a. Popping from queue

•When reading and pushing enough relay log events (to EOF
or timestamp is N seconds ahead of SQL Thread's timestamp),
clearing the queue and go back to 1

Replication Booster for MySQL

Performance improvements of Replication Booster

Executing 4,000 update/s on master

Replication Delay

Normal
1,779 update/s
(Slave delays)

Using Repl Booster
5,418 update/s
(Slave catches up)

Using Repl Booster
4,000 update/s
(No delay)
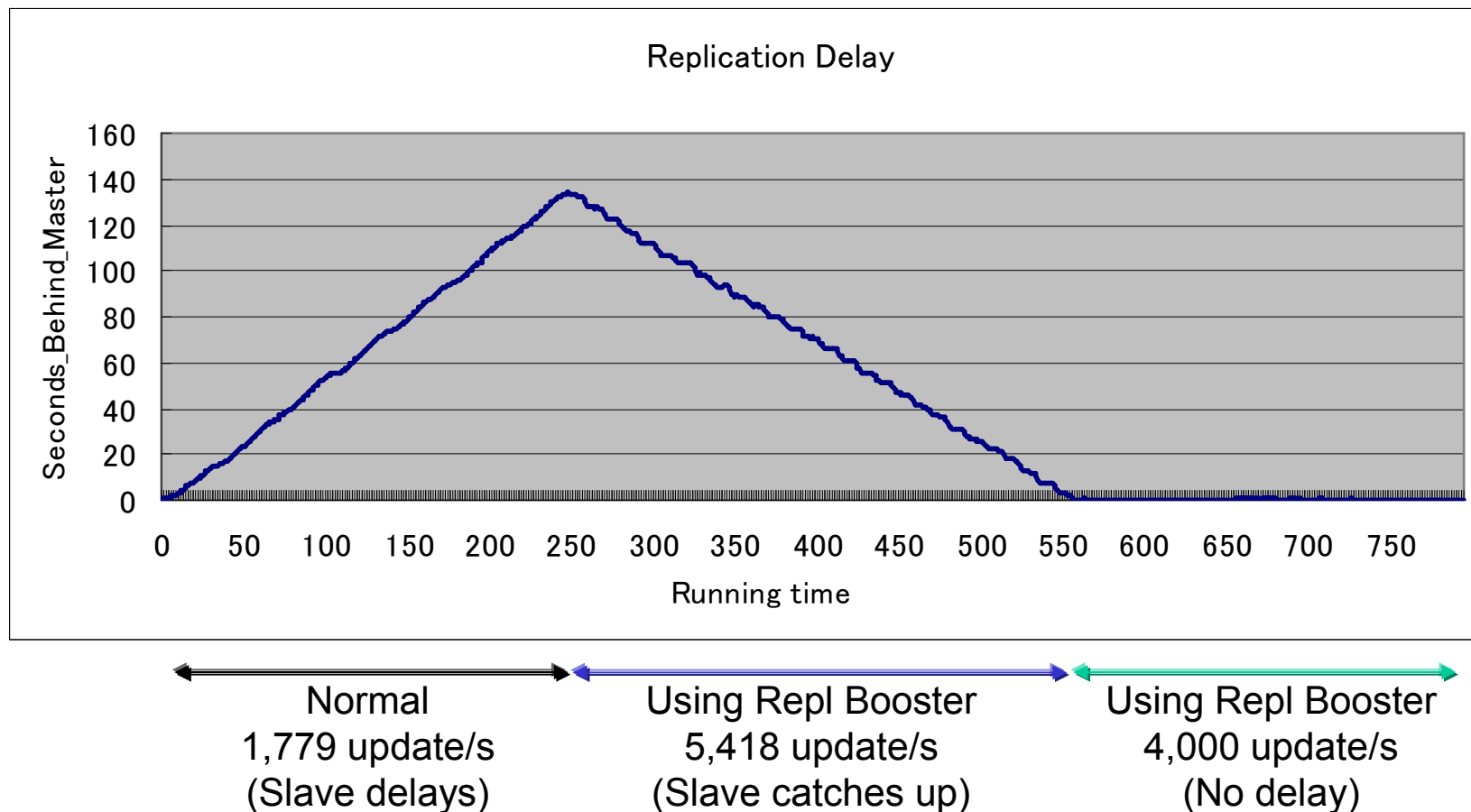
■ Currently alpha (I just started development a few weeks ago).
Will release very soon (maybe this week)

Random Write IOPS on SAS drives

DeNA

Random Write IOPS (16KB)
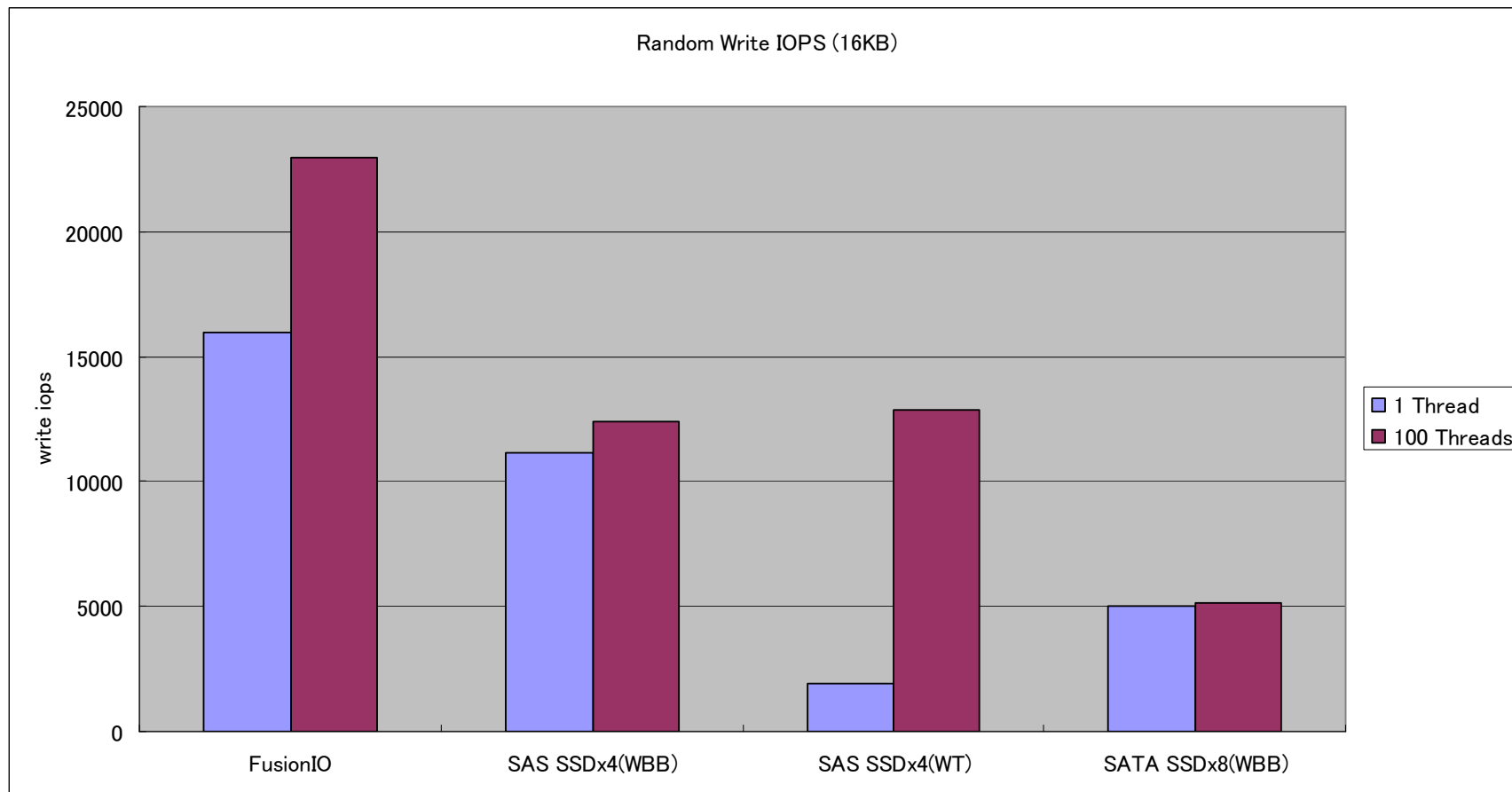
- SAS SSD is good enough (10,000+ random write iops should be enough for most of workloads)
- 100T WBB (Write back cache protected by battery) should be faster than 100T WT, maybe internal RAID controller / firmware issue

How about using SSD as L2 Cache? DeNA

- Using SSD as L2Cache
  - Interesting approach, as long as SSD is costly and capacity is limited
  - Transparent from applications

- Recent lineups
  - ZFS L2ARC
    - Part of ZFS filesystem
  - Facebook FlashCache
    - Working as Linux Kernel module
  - FusionIO DirectCache
    - Working between OS and FusionIO. Depending on FusionIO drives
  - Oracle Smart Flash Cache
    - L2 cache of Oracle Database. Depending on Oracle database

- Issues
  - Performance is not good (FlashCache)
    - Overheads on Linux Kernel seem huge
    - Even though data is 100% on SSD, random read iops dropped 40%, random write iops dropped 75% on FusionIO (tested with CentOS5.5)
    - Less performance drops on Intel X25-E
  - In practice it's Single Point of Failure
    - It's not just a cache. We expect SSD-level performance. If it's broken, the system will go down.
  - Total write volume grows significantly. L2Cache needs to be written when reading from HDD

Virtualization?

- **Currently performance drops are serious (on faster drives)**
  - Got only 1/30 throughput when tested with Ubuntu 10.4 + KVM + FusionIO

- **When tested with HDD, performance drops were not so serious**

- **Not many case studies about running FusionIO/tachIOn on virtualization environments**

- **It's better to run multiple MySQL instances on single OS**

Virtualization benchmarks (HDD)

Random Read IOPS (Dell could HDD, 16KB unit)
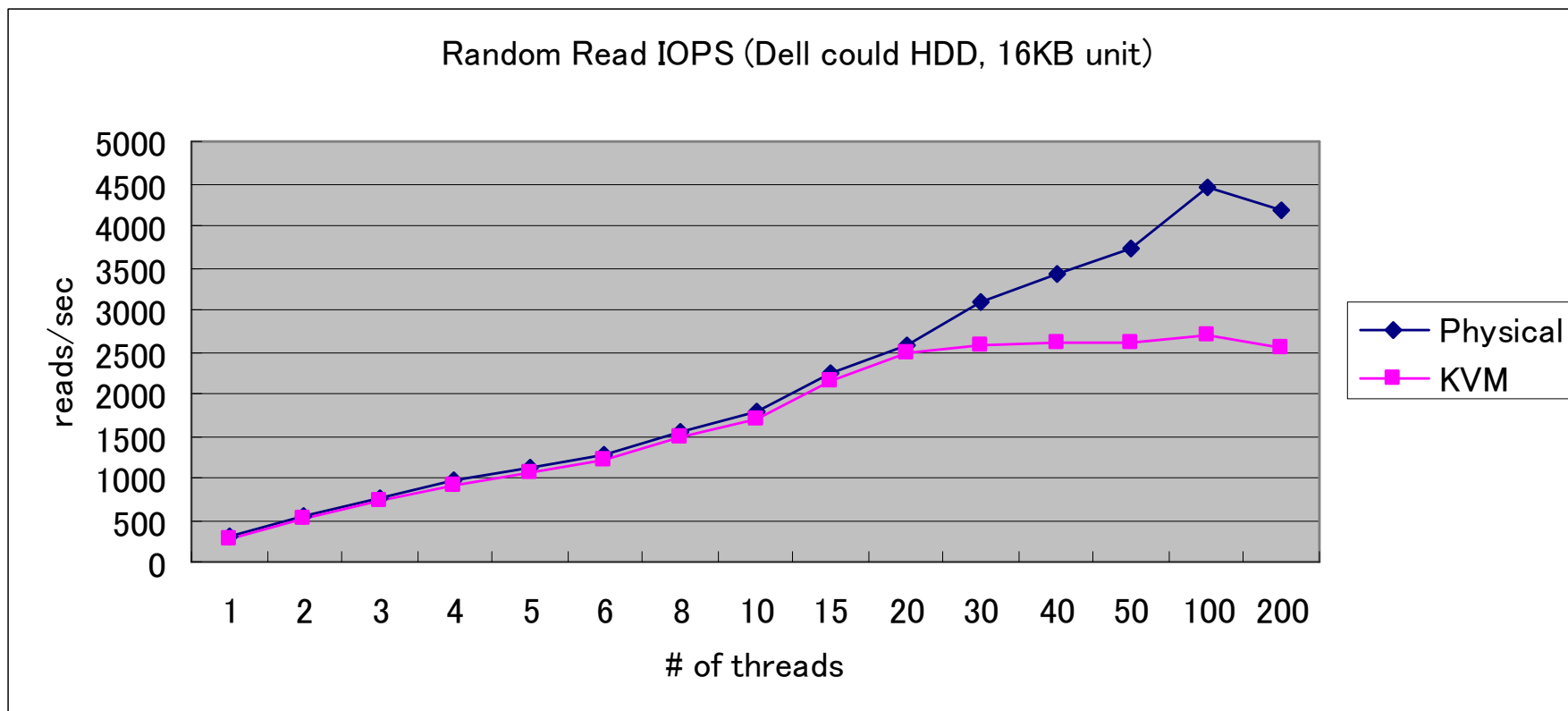
■ Not so bad on HDD

Virtualization benchmarks (SATA SSD)

DeNA

Random Read IOPS (Dell cloud SSD, 16KB unit)
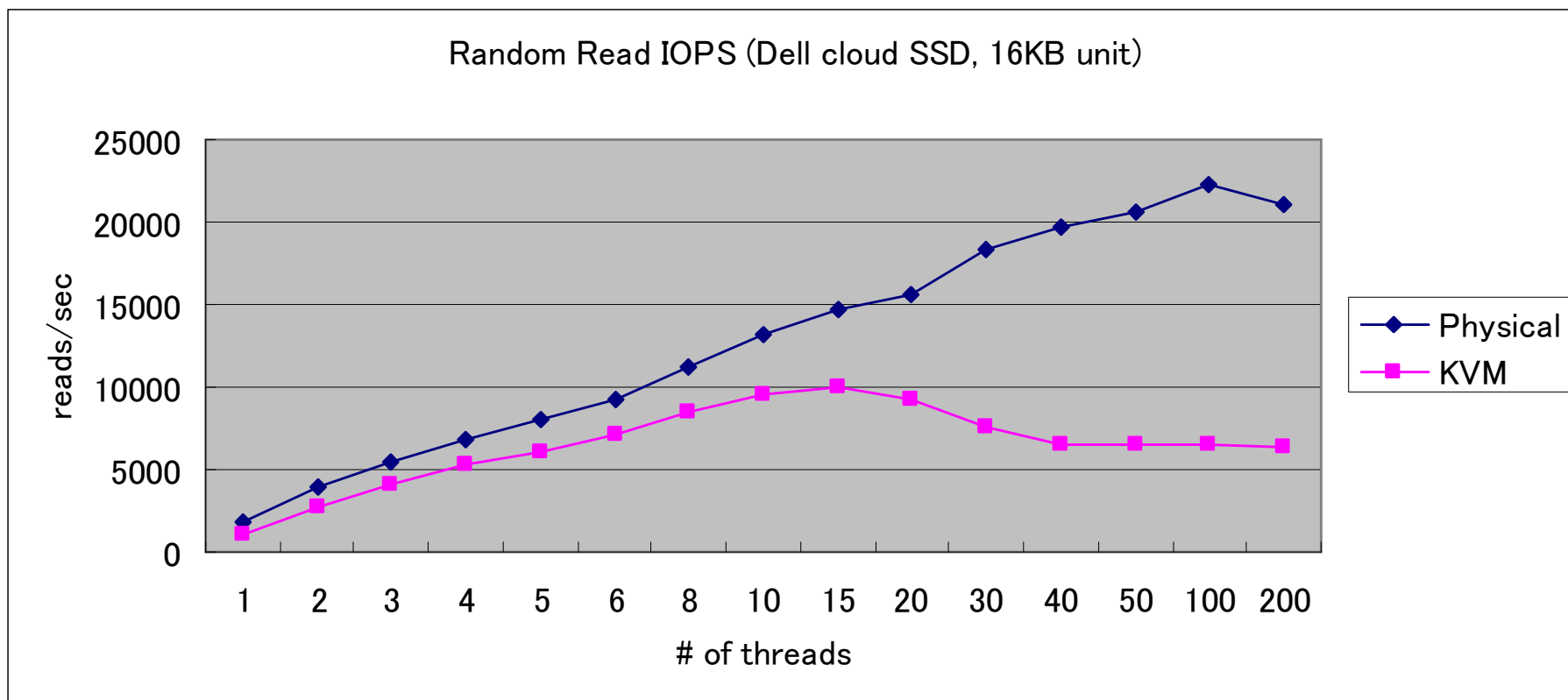
■ Difference becomes huge when using SSD

MySQL Deployment Practices on SSD

DeNA

- **What will happen if …**
  - Replacing HDD with Intel SSD (SATA)
  - Replacing HDD with Fusion I/O (PCI-E)
  - Moving log files and ibdata to HDD
  - Not using Nehalem
  - MySQL 5.5 or 5.1

- **DBT-2 benchmarks (write intensive)**
  - 200 Warehouses (20GB – 25GB hot data)
  - Fusion I/O: SLC, 96GB data space, 64GB reserved space

HDD vs Intel SSD vs Fusion I/O

	HDD	Intel	Fusion I/O
Buffer pool 1G	1125.44	5709.06	15122.75

NOTPM: Number of Transactions per minute

- Fusion I/O is a PCI-E based SSD
- PCI-E is much faster than SAS/SATA
- 14x improvement compared to 4HDDs

Which should we spend money, RAM or SSD?

	HDD	Intel	Fusion I/O
Buffer pool 1G	1125.44	5709.06	15122.75
Buffer pool 2G	1863.19		
Buffer pool 5G	4385.18		
Buffer pool 30G (Caching all hot data)	36784.76		

- Increasing RAM (buffer pool size) reduces random disk reads
  - Because more data are cached in the buffer pool
- If all data are cached, only disk writes (both random and sequential) happen
- Disk writes happen asynchronously, so application queries can be much faster
- Large enough RAM + HDD outperforms too small RAM + SSD

Which should we spend money, RAM or SSD?

	HDD	Intel	Fusion I/O
Buffer pool 1G	1125.44	5709.06	15122.75
Buffer pool 2G	1863.19	7536.55	20096.33
Buffer pool 5G	4385.18	12892.56	30846.34
Buffer pool 30G (Caching all hot data)	36784.76	-	57441.64

■ It is not always possible to cache all hot data

■ Fusion I/O + good amount of memory (5GB) was pretty good

■ Basic rule can be:
  ● If you can cache all active data, large enough RAM + HDD
  ● If you can't, or if you need extremely high throughput, spend on both RAM and SSD

MySQL file location

- SSD is extremely good at random reads
- SSD is very good at random writes
- HDD is good enough at sequential reads/writes
- No strong reason to use SSD for sequential writes

- Random I/O oriented:
  - Data Files (*.ibd)
    - Sequential reads if doing full table scan
  - Undo Log, Insert Buffer (ibdata)
    - UNDO tablespace (small in most cases, except for running long-running batch)
    - On-disk insert buffer space (small in most cases, except that InnoDB can not catch up with updating indexes)

- Sequential Write oriented:
  - Doublewrite Buffer (ibdata0)
    - Write volume is equal to *ibd files. Huge
  - Binary log (mysql-bin.XXXXXX)
  - Redo log (ib_logfile)
  - Backup files

Moving sequentially written files into HDD

	Fusion I/O	Fusion I/O + HDD	Up
Buffer pool 1G	15122.75 (us=25%, wa=15%)	19295.94 (us=32%, wa=10%)	+28%
Buffer pool 2G	20096.33 (us=30%, wa=12.5%)	25627.49 (us=36%, wa=8%)	+28%
Buffer pool 5G	30846.34 (us=39%, wa=10%)	39435.25 (us=49%, wa=6%)	+28%
Buffer pool 30G	57441.64 (us=70%, wa=3.5%)	66053.68 (us=77%, wa=1%)	+15%

- Moving ibdata, ib_logfile, (+binary logs) into HDD
- High impact on performance
  - Write volume to SSD becomes half because doublewrite area is allocated in HDD
  - %iowait was significantly reduced
  - Write volume on SSD is less than half

Does CPU matter?

Nehalem

Older Xeon

```
┌──────────┐ ┌──────────┐              ┌──────────┐
│   CPUs   │ │  Memory  │              │   CPUs   │
└────┬─────┘ └──────────┘              └────┬─────┘
     │                                      │
   QPI: 25.6GB/s                          FSB: 10.6GB/s
     │                                      │
┌──────────┐              ┌──────────┐ ┌──────────┐
│North Bridge│            │  Memory  │─│North Bridge│
│  (IOH)   │              └──────────┘ │  (MCH)   │
└────┬─────┘                           └────┬─────┘
     │                                      │
┌──────────┐                           ┌──────────┐
│PCI-Express│                          │PCI-Express│
└──────────┘                           └──────────┘
```
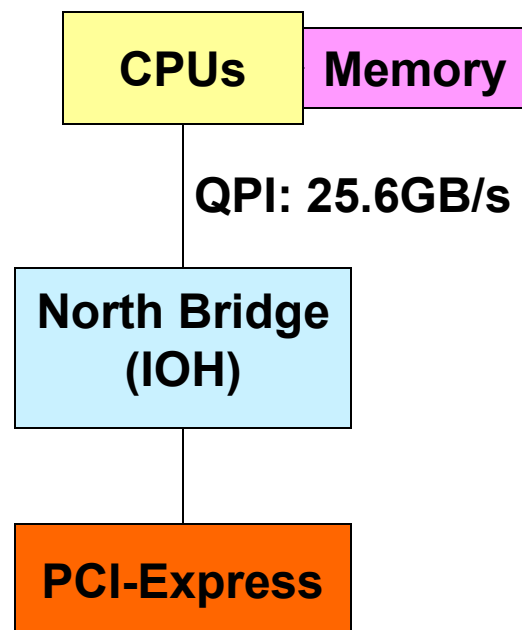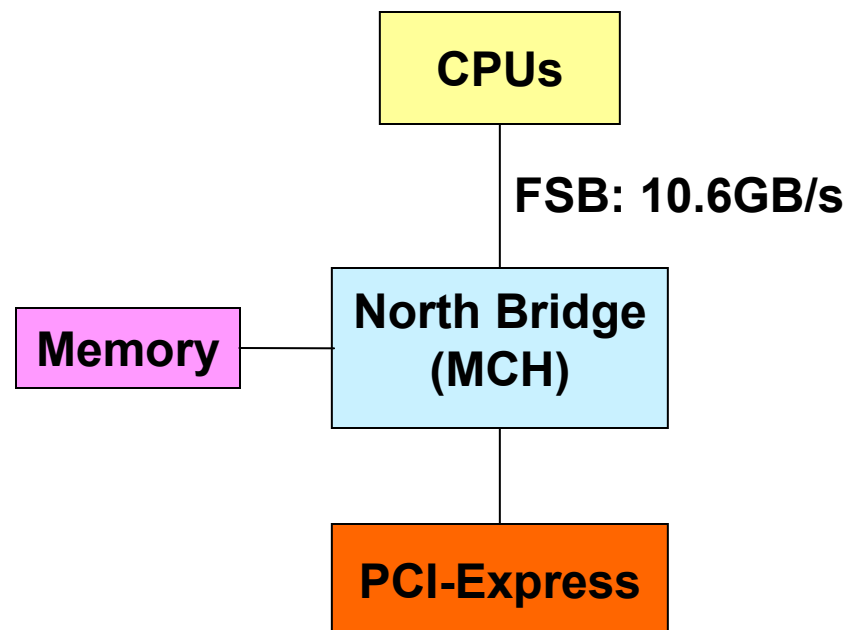
- **Nehalem has two big advantages**
  1. Memory is directly attached to CPU : Faster for in-memory workloads
  2. Interface speed between CPU and North Bridge is 2.5x higher, and interface traffics do not conflict with CPU<->Memory workloads
     Faster for disk i/o workloads when using PCI-Express SSDs

HDD	Harpertown X5470, 3.33GHz	Nehalem(X5570, 2.93GHz)	Up
Buffer pool 1G	1135.37 (us=1%)	1125.44 (us=1%)	-1%
Buffer pool 2G	1922.23 (us=2%)	1863.19 (us=2%)	-3%
Buffer pool 5G	4176.51 (us=7%)	4385.18(us=7%)	+5%
Buffer pool 30G	30903.4 (us=40%)	36784.76 (us=40%)	+19%

us: userland CPU utilization

- CPU difference matters on CPU bound workloads

Harpertown X5470 vs Nehalem X5570 (Fusion)

Fusion I/O+HDD	Harportown X5470, 3.33GHz	Nehalem(X5570, 2.93GHz)	Up
Buffer pool 1G	13534.06 (user=35%)	19295.94 (user=32%)	+43%
Buffer pool 2G	19026.64 (user=40%)	25627.49 (user=37%)	+35%
Buffer pool 5G	30058.48 (user=50%)	39435.25 (user=50%)	+31%
Buffer pool 30G	52582.71 (user=76%)	66053.68 (user=76%)	+26%

- TPM difference was much higher than HDD
- For disk i/o bound workloads (buffer pool 1G/2G), CPU utilizations on Nehalem were smaller, but TPM were much higher
  - Verified that Nehalem is much more efficient for PCI-E workloads
- Benefit from high interface speed between CPU and PCI-Express
- Fusion I/O fits with Nehalem much better than with traditional CPUs

Enabling HyperThreading

Fusion I/O + HDD	HT OFF (8)	HT ON (16)	Up
Buffer pool 1G	19295.94	20785.42	+7.7%
Buffer pool 2G	25627.49	28438.00	+11%
Buffer pool 5G	39435.25	45785.12	+16%
Buffer pool 30G	66053.68	81412.23	+23%

- InnoDB Plugin and 5.1 scales well with 16-24 CPU cores
- HT is more effective on SSD environments because loads are more CPU bound

MySQL 5.5

Fusion I/O + HDD	MySQL5.1	MySQL5.5	Up
Buffer pool 1G	19295.94	24019.32	+24%
Buffer pool 2G	25627.49	32325.76	+26%
Buffer pool 5G	39435.25	47296.12	+20
Buffer pool 30G	66053.68	67253.45	+1.8%

- Got 20-26% improvements for disk i/o bound workloads on Fusion I/O
  - Both CPU %user and %iowait were improved
    - %user: 36% (5.5.2) to 44% (5.5.4) when buf pool = 2g
    - %iowait: 8% (5.5.2) to 5.5% (5.5.4) when buf pool = 2g, but iops was 20% higher
  - Could handle a lot more concurrent i/o requests in 5.5 !
  - No big difference was found on 4 HDDs
    - Works very well on faster storages such as Fusion I/O, lots of disks

Where can PCI-Express SSD be used?

- **Deploying on master?**
  - If PCI-E SSD is used on master, replication delay will happen again
    - 10,000IOPS from single thread, 40,000+ IOPS from 100 threads
  - 10,000IOPS from 100 threads can be achieved from SATA SSD
 - Parallel SQL threads should be implemented in MySQL

- **Deploying on slave?**
  - If using HDD on master, SATA SSD should be enough to handle workloads
    - PCI-Express SSD is much more expensive than SATA SSD
  - How about running multiple MySQL instances on single server?
    - Virtualization is not fast
    - Running multiple MySQL instances on single OS is more reasonable

- **Does PCI-E SSD have enough storage capacity to run multiple instances?**
  - On HDD environments, typically only 100-200GB of database data can be stored because of slow random IOPS on HDD
  - FusionIO SLC: 320GB Duo + 160GB = 480GB
  - FusionIO MLC: 1280GB Duo + 640GB = 1920GB (or using ioDrive Octal)
  - tachIOn SLC: 800GB x 2 = 1600GB

Running multiple slaves on single box

Before

After
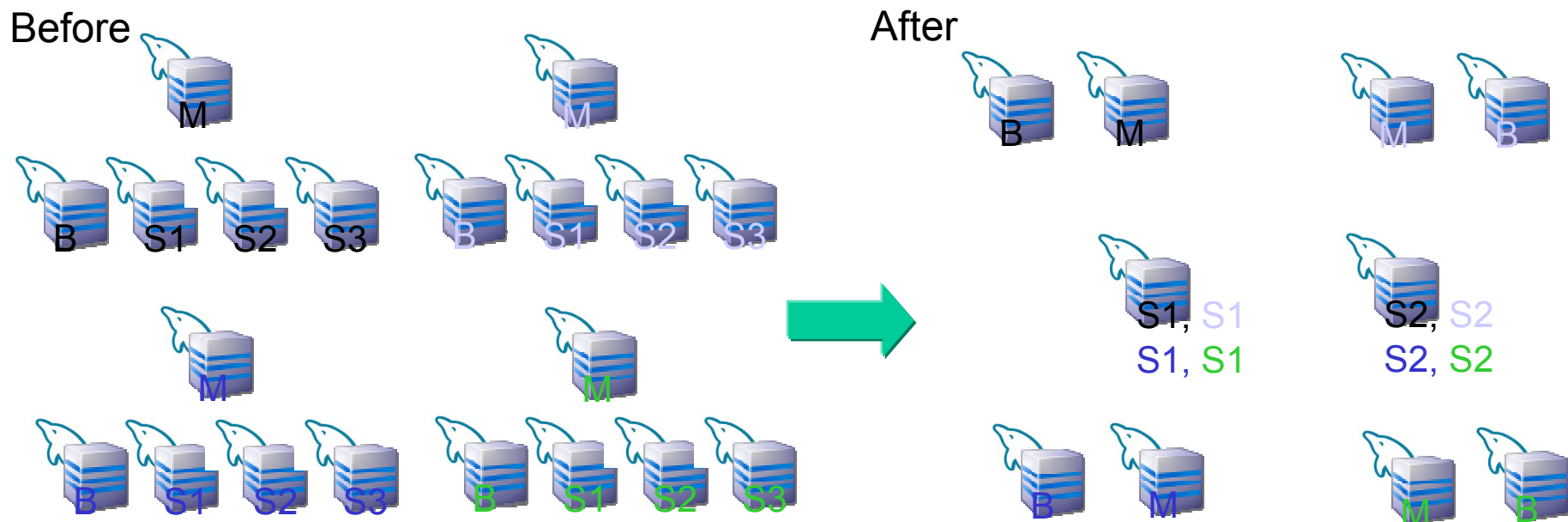
- ■ Running multiple slaves on a single PCI-E slave
  - ● Master and Backup Server are still HDD based
  - ● Consolidating multiple slaves
  - ● Since slave's SQL thread is single threaded, you can gain better concurrency by running multiple instances
  - ● The number of instances is mainly restricted by capacity

Our environment

- Machine
  - HP DL360G7 (1U), or Dell R610
- PCI-E SSD
  - FusionIO MLC (640-1280GB Duo + 320-640GB)
  - tachIOn SLC (800GB x 2) -> MLC
- CPU
  - Two sockets, Nehalem 6-core per socket, HT enabled
    - 24 logical CPU cores are visible
    - Four socket machine is too expensive
- RAM
  - 60GB or more
- Network
  - Broadcom BCM5709, Four ports
  - Bonding x 2
  - Two physical IP addresses, 6-12 (=instances) virtual IP addresses
- HDD
  - 4-8 SAS RAID1+0
  - For backups, redo logs, relay logs, (optionally) doublewrite buffer

Statistics

DeNA

- ■ Consolidating 7 instances on FusionIO (640GB MLC Duo + 320GB MLC)
  - ● Let half of SELECT queries go to these slaves
  - ● 6GB innodb_buffer_pool_size

- ■ Peak QPS (total of 7 instances)
  - ● 61683.7 query/s
  - ● 37939.1 select/s
  - ● 7861.1 update/s
  - ● 1105 insert/s
  - ● 1843 delete/s
  - ● 3143.5 begin/s

- ■ CPU Utilization
  - ● %user 27.3%, %sys 11%(%soft 4%), %iowait 4%
  - ● C.f. SATA SSD：%user 4%, %sys 1%, %iowait 1%

- ■ Buffer pool hit ratio
  - ● 99.4%
  - ● SATA SSD : 99.8%

- ■ No replication delay
- ■ No significant (100+ms) response time delay caused by SSD

CPU loads

CPU usage 6 hours

user	current: 23.0	average: 25.7	max: 28.0	
sys	current: 9.0	average: 9.6	max: 11.0	
iow	current: 3.0	average: 3.3	max: 4.0	
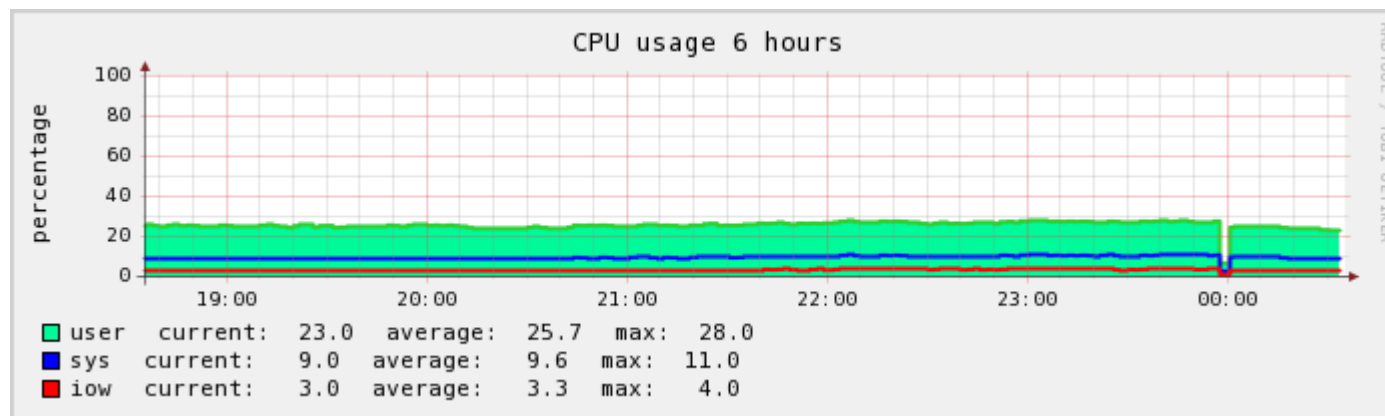
	CPU	%user	%nice	%sys	%iowait	%irq	%soft	%steal	%idle	intr/s
22:10:57										
22:11:57	all	27.13	0.00	6.58	4.06	0.14	3.70	0.00	58.40	56589.95
...										
22:11:57	23	30.85	0.00	7.43	0.90	1.65	49.78	0.00	9.38	44031.82

- CPU utilization was high, but should be able to handle more
  - %user 27.3%, %sys 11%(%soft 4%), %iowait 4%
  - Reached storage capacity limit (960GB). Using 1920GB MLC should be fine to handle more instances

- Network will be the first bottleneck
  - Recv: 14.6MB/s, Send: 28.7MB/s
  - CentOS5 + bonding is not good for network requests handling (only single CPU core can handle requests)
  - Using four or more network cables, then building two or more bond interfaces should scale

Things to consider

- **Allocating different IP addresses or port numbers**
  - Administration tools are also affected
  - We allocated different IP addresses because some of existing tools depend on "port=3306"
  - ip addr
  - bind-address="virtual ip address" in my.cnf

- **Creating separated directories and files**
  - Socket files, data directories, InnoDB files, binary log files etc should be stored on different location each other

- **Storing some files on HDD, others on SSD**
  - Binary logs, Relay logs, Redo logs, error/slow logs, ibdata0 (files where doublewrite buffer is written), backup files on HDD
  - Others on SSD

taskset

```
# taskset -pc 0, 12, 2, 14 `cat /var/lib/mysql/mysqld1.pid`
# taskset -pc 1, 13, 3, 15 `cat /var/lib/mysql/mysqld2.pid`
# taskset -pc 5, 17, 7, 19  `cat /var/lib/mysql/mysqld3.pid`
# taskset -pc 8, 20, 10, 22  `cat /var/lib/mysql/mysqld4.pid`
# taskset -pc 9, 21, 11, 23  `cat /var/lib/mysql/mysqld5.pid`
# taskset -pc 4, 16, 6, 18  `cat /var/lib/mysql/mysqld6.pid`
# taskset -pc 8, 20, 10, 22  `cat /var/lib/mysql/mysqld7.pid`
```

- MySQL server currently does not scale well with 24 logical CPU cores
- When running 6+ instances on 24 cores, the number of utilized CPU cores should be limited per instance
- Check /proc/cpuinfo output
- Use same physical id (socket id) for the same instance
- Within the same physical id, use same core id (physical core) for the same instance

Application Design

- **Bottleneck shifts from storage to CPU/Network**
  - Massively executed SQL statements should be migrated to NoSQL / HandlerSocket /etc

- **Separating tables**
  - History tables (only recent data is accessed) fit with HDD because most of active data fit in memory
  - Rarely accessed tables can also be stored on HDD
  - Other tables on SSD

Making MySQL better

- **Parallel SQL threads**

- **Pool of threads**

- **8KB/4KB InnoDB Blocks**

- **Minimized performance stalls**

- **No hot global/large mutex**
  - LINEAR HASH partitions on large tables helps
    - Index mutex is allocated per partition

Future improvements from DBA perspective

- ■ One master, one backup/slave and one DR slave
  - ● Single slave should be enough from performance perspective

- ■ Differential recovery (no full restore) on crash
  - ● Restoring 800GB data file takes hours in total

- ■ MySQL should be better for crashes
  - ● Master should flush transactional data to disks at commit
    - – sync-binlog=1, innodb_flush_log_at_trx_commit=1
    - – Right now performance is poor
  - ● Semi-sync replication (GA in 5.5)
  - ● Crash-safe binary logs (binlog checksum)
  - ● Relay log execution status should be stored to InnoDB tables, not to file(relay-log.info) (5.6)
    - – Relay log status can be consistent

Conclusion for choosing H/W

- **Disks**
  - PCI-E SSDs (i.e. FusionIO, tachIOn) perform very well
  - Recent SAS/SATA drives with RAID performs much better than before
    - 8-10 drives per 1U box shows great concurrency improvements
    - Single threaded performance is still poor. Slave prefetching tool (i.e. Replication Booster) may mitigate this issue
  - Keep enough reserved space, use tachIOn, or use RAID0 if you need to handle massive write traffics
  - HDD is good at sequential writes

- **Use Nehalem CPU**
  - Especially when using PCI-Express SSDs

Conclusion for database deployments

- **Put sequentially written files on HDD**
  - ibdata, ib_logfile, binary log files
  - HDD is fast enough for sequential writes
  - Write performance deterioration can be mitigated
  - Life expectancy of SSD will be longer

- **Put randomly accessed files on SSD**
  - *ibd files, index files(MYI), data files(MYD)
  - SSD is 10x -100x faster for random reads than HDD

- **Archive less active tables/records to HDD**
  - SSD is still much expensive than HDD

- **Use 5.1 InnoDB Plugin or 5.5**
  - Higher scalability & concurrency matters on faster storage

What will happen in the real database world?

- ■ These are just my thoughts..

- ■ Less demand for NoSQL
  - Isn't it enough for many applications just to replace HDD with Fusion I/O?
  - Importance on functionality will be relatively stronger

- ■ Stronger demand for Virtualization
  - Single server will have enough capacity to run two or more mysqld instances
  - Right now performance is poor. Running multiple instances is a good workaround

- ■ I/O volume matters
  - Not just IOPS
  - Configurable block size, disabling doublewrite, etc

- ■ Concurrency matters
  - Single SSD scales as well as 8-16 HDDs
  - Concurrent ALTER TABLE, parallel query

■ Memory and Swap Space Management

Random Access Memory

- RAM access speed is much faster than HDD/SSD
  - RAM: -60ns
    - 100,000 queries per second is not impossible
  - HDD: -5ms
  - SSD: 100-500us
- 16-100+GB RAM is now pretty common

- *hot application data* should be cached in memory
- Minimizing hot application data size is important
  - Use compact data types (SMALLINT instead of VARCHAR/BIGINT, TIMESTAMP instead of DATETIME, etc)
  - Do not create unnecessary indexes
 - Delete records or move to archived tables, to keep hot tables smaller

Cache hot application data in memory

| DBT-2 (W200)	Transactions per Minute	%user	%iowait
Buffer pool 1G	1125.44	2%	30%
Buffer pool 2G	1863.19	3%	28%
Buffer pool 5G	4385.18	5.5%	33%
Buffer pool 30G (All data in cache)	36784.76	36%	8%

- DBT-2 benchmark (write intensive)
- 20-25GB hot data (200 warehouses, running 1 hour)
- RAM size affects everything. Not only for SELECT, but also for INSERT/UPDATE/DELETE
  - INSERT: Random reads/writes happen when inserting into indexes in random order
  - UPDATE/DELETE: Random reads/writes happen when modifying records

Use Direct I/O

Buffered I/O
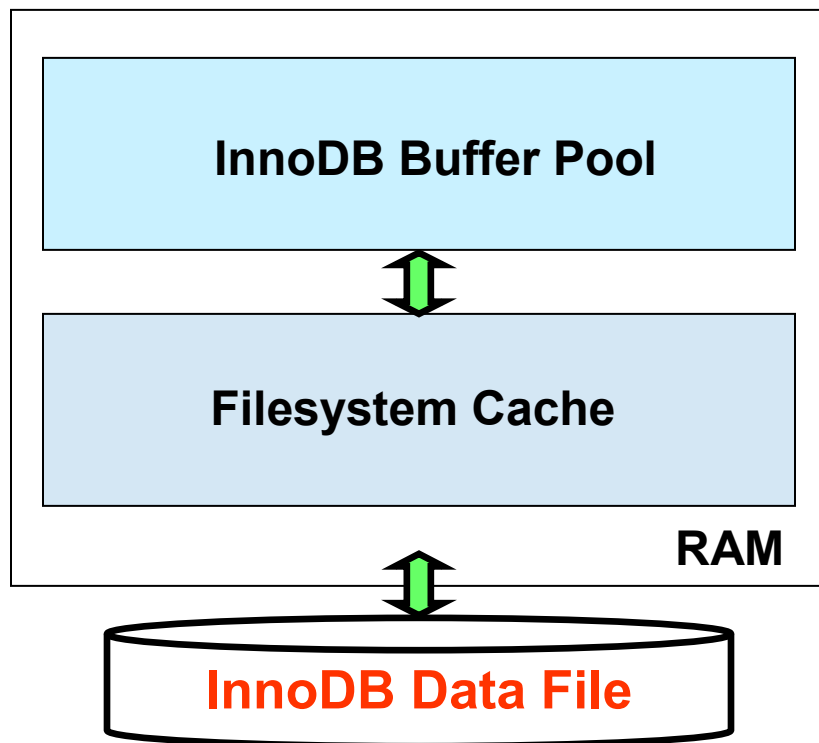
Direct I/O
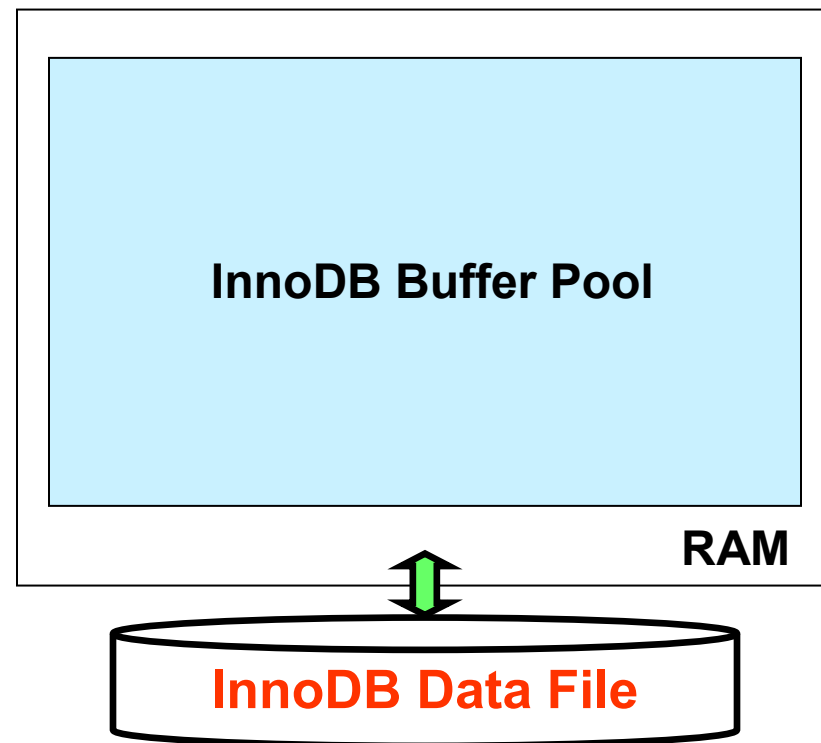
- Direct I/O is important to fully utilize Memory
- innodb_flush_method=O_DIRECT
- Alignment: File i/o unit must be a factor of 512 bytes
  - Can't use O_DIRECT for InnoDB Log File, Binary Log File, MyISAM, PostgreSQL data files, etc

Do not allocate too much memory

```
user$ top
Mem:  32967008k total, 32808696k used,   158312k free,    10240k buffers
Swap: 35650896k total,  4749460k used, 30901436k free,   819840k cached
 PID USER     PR  NI  VIRT  RES  SHR S %CPU %MEM   TIME+  COMMAND
5231 mysql     25   0 35.0g  30g  324 S  0.0 71.8  7:46.50 mysqld
```

■ What happens if no free memory space is available?
  ● Reducing filesystem cache to allocate memory space
  ● Swapping process(es) to allocate memory space

■ Swap is bad
  ● Process spaces are written to disk (swap out)
  ● Disk reads happen when accessing on-disk process spaces (swap in)
  ● Massive random disk reads and writes will happen

What if setting swap size to zero?

- By setting swap size to zero, swap doesn't happen anymore. But..
  - Very dangerous

- When neither RAM nor swap space is available, OOM killer is invoked. OOM Killer may kill any process to allocate memory space

- The most memory-consuming process (mysqld) will be killed at first
  - It's abort shutdown. Crash recovery takes place at restart
  - Priority is determined by ORDER BY /proc/<PID>/oom_score DESC
  - Normally mysqld has the highest score
    - Depending on VMsize, CPU time, running time, etc

- It often takes very long time (minutes to hours) for OOM Killer to kill processes
  - We can't do anything until enough memory space is available

Do not set swap=zero

```
top - 01:01:29 up  5:53,  3 users,  load average: 0.66, 0.17, 0.06
Tasks: 170 total,   3 running, 167 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us, 24.9%sy,   0.0%ni,75.0%id,0.2%wa,0.0%hi,  0.0%si,0.0%st
Mem: 32967008k total, 32815800k used,    151208k free,      8448k buffers
Swap:        0k total,         0k used,         0k free,    376880k cached

  PID USER       PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
26988 mysql      25   0   30g  30g 1452 R 98.5 97.7   0:42.18 mysqld
```

- If no memory space is available, OOM killer will be invoked
- Some CPU cores consume 100% system resources
  - 24.9% (average) = 1 / 4 core use 100% cpu resource in this case
  - Terminal freezed (SSH connections can't be established)
- Swap is bad, but OOM killer is much worse than swap

What if stopping OOM Killer?

■ If /proc/<PID>/oom_adj is set to -17, OOM Killer won't kill the process

- Setting -17 to sshd is a good practice so that we can continue remote login
- # echo -17 > /proc/<pid of sshd>/oom_adj

■ But don't set -17 to mysqld

- If over-memory-consuming process is not killed, Linux can't have any available memory space
- We can't do anything for a long long time.. -> Long downtime

Swap space management

- **Swap space is needed to stabilize systems**
  - But we don't want mysqld swapped out

- **What consumes memory?**
  - RDBMS
    - Mainly process space is used (innodb_buffer_pool, key_buffer, sort_buffer, etc)
    - Sometimes filesystem cache is used (InnoDB Log Files, binary/relay log files, MyISAM files, etc)
  - Administration (backup, etc)
    - Mainly filesystem cache is used

- **We want to keep mysqld in RAM, rather than allocating large filesystem cache**

Be careful about backup operations

```
Mem:  32967008k total, 28947472k used,  4019536k free,   152520k buffers

Swap: 35650896k total,        0k used, 35650896k free,   197824k cached

PID USER     PR  NI  VIRT  RES  SHR S %CPU %MEM   TIME+  COMMAND

 5231 mysql    25   0 27.0g  27g  288 S  0.0 92.6   7:40.88 mysqld
```
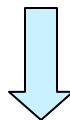
Copying 8GB datafile

```
Mem:  32967008k total, 32808696k used,   158312k free,    10240k buffers

Swap: 35650896k total, 4749460k used, 30901436k free,  8819840k cached

 PID USER     PR  NI  VIRT  RES  SHR S %CPU %MEM   TIME+  COMMAND

5231 mysql     25   0 27.0g  22g  324 S  0.0 71.8   7:46.50 mysqld
```

■ Copying large files often causes swap

vm.swappiness = 0

```
Mem:  32967008k total, 28947472k used,  4019536k free,   152520k buffers

Swap: 35650896k total,        0k used, 35650896k free,   197824k cached

PID USER     PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND

 5231 mysql    25   0 27.0g  27g  288 S  0.0 91.3   7:55.88 mysqld
```

Copying 8GB of datafile

```
Mem:  32967008k total, 32783668k used,   183340k free,    3940k buffers

Swap: 35650896k total,    216k used, 35650680k free,  4117432k cached

PID USER     PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND

 5231 mysql    25   0 27.0g  27g  288 S  0.0 80.6   8:01.44 mysqld
```

- Set vm.swappiness=0 in /etc/sysctl.conf
  - Default is 60
- When physical RAM was fully consumed, Linux kernel reduces filesystem cache with high priority (lower swappiness increases priority)
- After no file system cache is available, swapping starts
  - OOM killer won't be invoked if large enough swap space is allocated. It's safer

But swap happens even though swappiness==0

- **32GB RAM box, CentOS5 (2.6.18.128)**
  - innodb_buffer_pool_size = 26GB
  - mysqld other than innnodb_buf_pool_size: approx 3GB
- **Kernel space : approx 1GB**

```
top - 11:54:51 up 7 days, 15:17,  1 user,  load average: 0.21, 0.14, 0.10
Tasks: 251 total,   1 running, 250 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.5%us,  0.2%sy,  0.0%ni, 98.9%id,  0.3%wa,  0.0%hi,  0.1%si,  0.0%st
Mem:  32818368k total, 31154696k used,  1663672k free,   125048k buffers
Swap:  4184924k total, 1292756k used,  2892168k free,  2716380k cached

 PID USER     PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
6999 mysql     15   0 28.4g  26g 5444 S  9.3 85.2 210:52.67 mysqld
```

Swap caused by log files

InnoDB Buffer Pool

Filesystem Cache

InnoDB Data File
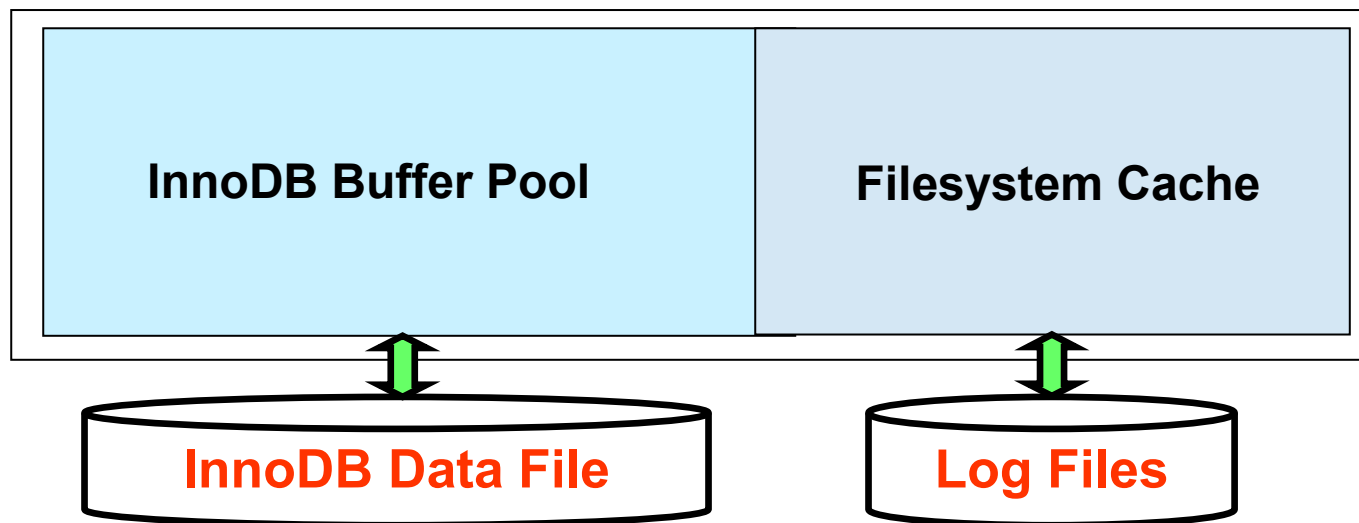
Log Files

- Swap happened even though filesystem cache was not wiped out
- InnoDB log file size : 3GB in total
    - It's common in InnoDB Plugin because crash recovery becomes much faster
- Binary logs (or relay logs)

Swap bug on Linux

- **A known issue of Linux Kernel**
  - Fixed in 2.6.28: Merged on RHEL6
  - https://bugzilla.redhat.com/show_bug.cgi?id=160033
  - When will CentOS 6 be released?

- **Before the bug was fixed:**
  - swap_tendency = mapped_ratio / 2 + distress + sc->swappiness;
  - If swap_tendency > 100, swap happens regardless of filesystem cache
    - This might happen if distress is high enough, even though swappiness == 0

- **When distress becomes high?**
  - Writing to InnoDB log files, binary log files, relay log files
  - On recent MySQL this issue becomes even more serious because innodb_log_file_size can be bigger
    - Slow crash recovery bug has been fixed in InnoDB Plugin, so using 3-4GB inodb log file becomes common

- **What workarounds can be available in RHEL5?**
  - Decrease innodb_buffer_pool_size or innodb_log_file_size?
    - I don't like the solution
  - Unmap InnoDB log files and binary/relay log files from file system cache

Tool: unmap_mysql_logs

- https://github.com/yoshinorim/unmap_mysql_logs
  - Perl based command line tool
    - Depending on Inline::C CPAN module
- Unmapping the below files from filesystem cache
  - InnoDB Log Files: All files
  - Binary log files, Relay log files:
    - Current (hot) binlog/relay log: All except the latest (tail) 10% of the file
    - Because the tail of the current files are read by Binlog Dump thread or SQL thread: needs to be cached
    - Others: all
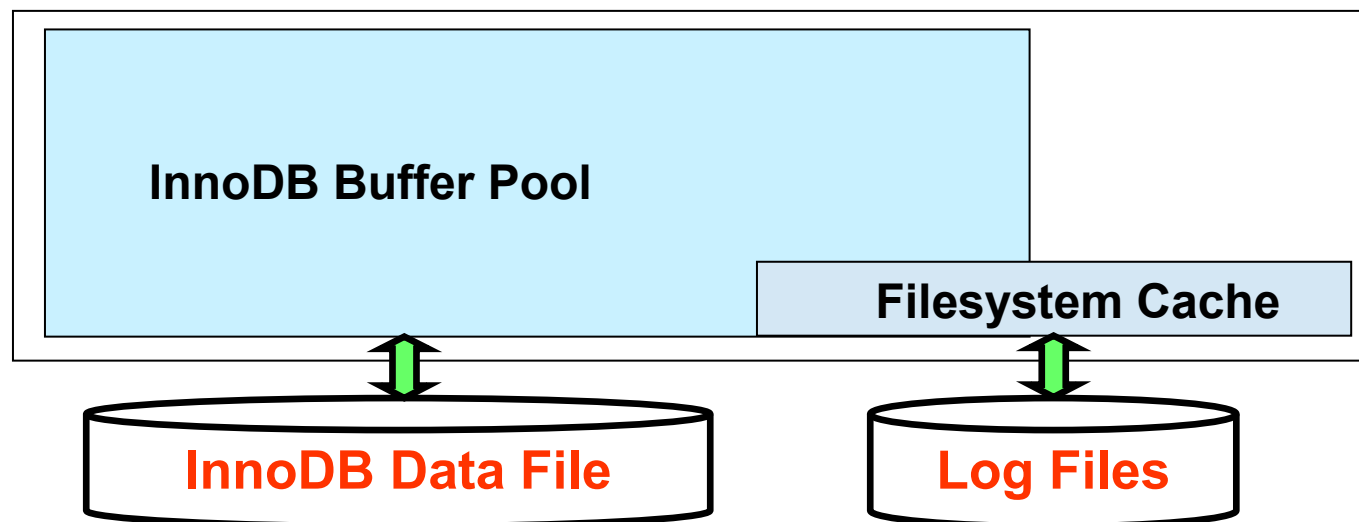
Performance effect

```
Mem:  32825116k total, 32728924k used,   96192k free,   246360k buffers
Swap:  4186072k total,   118764k used,  4067308k free,  2803088k cached

 PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
30814 mysql     15   0 28.8g  27g 5644 S 81.1 89.4 158198:37 mysqld
```

```
Mem:  32825116k total, 30061632k used, 2763484k free,   247420k buffers
Swap:  4186072k total,   118764k used,  4067308k free,   143372k cached

 PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
30814 mysql     15   0 28.8g  27g 5644 S 76.1 89.4 158200:59 mysqld
```

- Filesystem cache was reduced, free memory space was increased
- No swapping has happened anymore
- Regularly invoke the script
  - Every 10 minutes from cron

Memory allocator

- **mysqld uses malloc()/mmap() for memory allocation**

- **Faster and more concurrent memory allocator such as tcmalloc can be used**
  - Install Google Perftools (tcmalloc is included)
    - \# yum install libunwind
    - \# cd google-perftools-x.y ; ./configure --enable-frame-pointers; make; make install
  - export LD_PRELOAD=/usr/local/lib/tcmalloc_minimal.so; mysqld_safe &

- **InnoDB internally uses its own memory allocator**
  - Can be changed in InnoDB Plugin
    - If Innodb_use_sys_malloc = 1(default 1), InnoDB uses OS memory allocator
    - tcmalloc can be used by setting LD_PRELOAD

Memory allocator would matter for CPU bound workloads

	Default allocator	tcmalloc_minimal	%user	up
Buffer pool 1G	1125.44	1131.04	2%	+0.50%
Buffer pool 2G	1863.19	1881.47	3%	+0.98%
Buffer pool 5G	4385.18	4460.50	5.5%	+1.2%
Buffer pool 30G	36784.76	38400.30	36%	+4.4%

- DBT-2 benchmark (write intensive)
- 20-25GB hot data (200 warehouses, running 1 hour)

Be careful about per-session memory

- Do not allocate much more memory than needed (especially for per-session memory)

- Allocating 2MB takes much longer time than allocating 128KB
  - Linux malloc() internally calls brk() if size <= 512KB, else calling mmap()

- In some cases too high per-session memory allocation causes negative performance impacts
  - SELECT * FROM huge_myisam_table LIMIT 1;
  - SET read_buffer_size = 256*1024; (256KB)
    - -> 0.68 second to run 10,000 times
  - SET read_buffer_size = 2048*1024; (2MB)
    - -> 18.81 seconds to run 10,000 times

- In many cases MySQL does not allocate per-session memory than needed. But be careful about some extreme cases (like above: MyISAM+LIMIT+FullScan)

■ File I/O

File I/O and synchronous writes

- RDBMS calls fsync() many times (per transaction commit, checkpoints, etc)
- Make sure to use Battery Backed up Write Cache (BBWC) on raid cards
  - 10,000+ fsync() per second, without BBWC less than 200 on HDD
  - Disable write cache on disks for safety reasons
- Do not set "write barrier" on filesystems (enabled by default in some cases)
  - Write-through to disks even though BBWC is enabled (very slow)
  - ext3: mount -o barrier=0 (needed for some distros such as SuSe Linux)
  - xfs: mount -o nobarrier
  - drbd: no-disk-barrier in drbd.conf

seek & rotation time

disk

Write cache with battery

disk

seek & rotation time
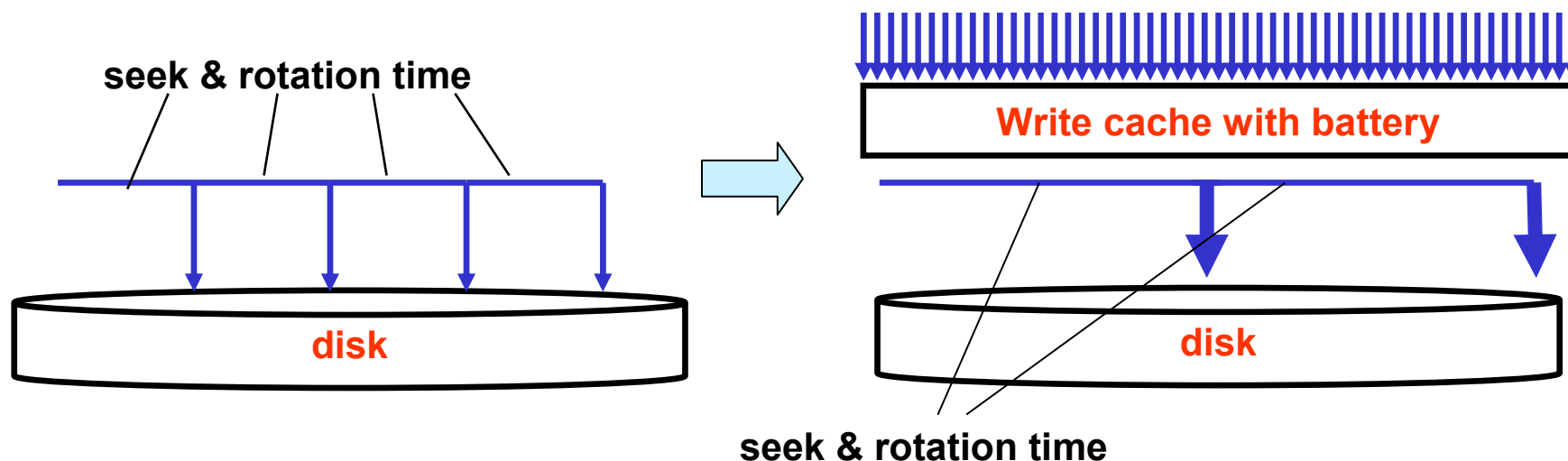
BBWC and "Auto Learn" Issues

- Battery needs to be discharged and recharged per every specific period
  - "Auto Learn"
  - Disabling write cache (entering write through), recharging battery
    - Takes 1 Hour or even more
  - Sudden performance drops happen
    - iowait suddenly increases a lot
  - Automated auto learn
    - Default behavior
    - It is serious if happened at peak time
  - Manual auto learn (in many cases officially not supported)
    - Manually discharge and recharge on off-peak loads
    - If you forget to execute for a long time (i.e. 8 months), battery will be broken

- Expected behavior on Dell servers (not a bug)
  - LSI RAID Controllers with BBWC
  - Approximately every 90 days
    - "Event Description: BBU disabled; changing WB virtual disks to WT"

FBWC

- **Flash Back Write Cache**
  - Using NOR flash memory and capacitor instead of battery
- **No "Auto Learn" issue**

- **HP adopts FBWC on recent lineups (DL360G7)**

Overwriting or Appending?

- Some files are overwritten (fixed file size), others are appended (increasing file size)
  - Overwritten: InnoDB Logfile
  - Appended: Binary Logfile

- Appending + fsync() is much slower than overwriting + fsync()
  - Additional file space needs to be allocated & file metadata needs to be flushed per fsync()
  - 10,000+ fsync/sec for overwriting, 3,000 or less fsync/sec for appending
    - Appending speed highly depends on filesystems
    - Copy-on-write filesystems such as Solaris ZFS is fast enough for appending (7,000+)
  - Be careful when using sync-binlog=1 for binary logs
    - Consider using ZFS
    - Check "preallocating binlog" worklog: WL#4925
  - Do not extend files too frequently
    - innodb-autoextend-increment = 20 (default 8)

Quick file i/o health check

■ Checking FBWC/BBWC is enabled, and write barrier is disabled

  ● Overwriting + fsync() test

    – Run mysqlslap insert(InnoDB, single-threaded, innodb_flush_log_at_trx_commit=1), check qps is over 1,000

      $ mysqlslap --concurrency=1 --iterations=1 --engine=innodb ¥

        --auto-generate-sql --auto-generate-sql-load-type=write ¥

        --number-of-queries=100000

Buffered and asynchronous writes

- Some file i/o operations are not direct i/o, not synchronous
  - file copy, binlog, MyISAM, mysqldump, innodb_flush_log_at_trx_commit=2, etc

- Dirty pages in filesystem cache needs to be flushed to disks in the end
  - pdflush takes care of it, maximum 8 threads

- When? -> highly depending on vm.dirty_background_ratio and vm.dirty_ratio
  - Flushing dirty pages starts in background after reaching dirty_background_ratio * RAM (Default is 10%, 10% of 64GB is 6.4GB)
  - Forced flush starts after reaching dirty_ratio * RAM (Default is 40%)

- Forced, and burst dirty page flushing is problematic
  - All buffered write operations become synchronous, which hugely increase latency

- Do flush dirty pages aggressively
  - Execute sync; while doing massive write operations
  - Reduce vm.dirty_background_ratio
  - Upgrade to 2.6.32 or higher
    - pdflush threads are allocated per device. Flushing to slow devices won't block other pdflush threads

Filesystem – ext3

- By far the most widely used filesystem

- But not always the best

- Deleting large files takes long time
  - Internally has to do a lot of random disk i/o (slow on HDD)
  - In MySQL, if it takes long time to DROP table,
    all client threads will be blocked to open/close tables (by LOCK_open mutex)
  - Be careful when using MyISAM, InnoDB with innodb_file_per_table, PBXT, etc

- Writing to a file is serialized
  - Serialized by "i-mutex", allocated per i-node
  - Sometimes it is faster to allocate many files instead of single huge file
  - Less optimized for faster storage (like PCI-Express SSD)

- Use "dir_index" to speed up searching files
- Use barrier=0 to disable write-through

Tip: fast large file remove

- **Creating hard links**
  - ln /data/mysql/db1/huge_table.ibd /var/tmp/huge_table.ibd.link
  - DROP TABLE huge_table;
  - rm –f /var/tmp/huge_table.ibd.link

  - Physical file remove doesn't happen because ref count is not zero
  - When removing the hard link, physical file remove happens because at that time ref count is zero
  - Physical remove operation causes massive random disk i/o, but at this stage this doesn't take MySQL mutex so it won't block MySQL operations
    - But iowait increases so don't run rm command when load is high

Filesystem – xfs/ext2

- **xfs**
  - Fast for dropping files
  - Concurrent writes to a file is possible when using O_DIRECT
  - Not officially supported in current RHEL (Supported in SuSE)
  - Disable write barrier by setting "nobarrier"
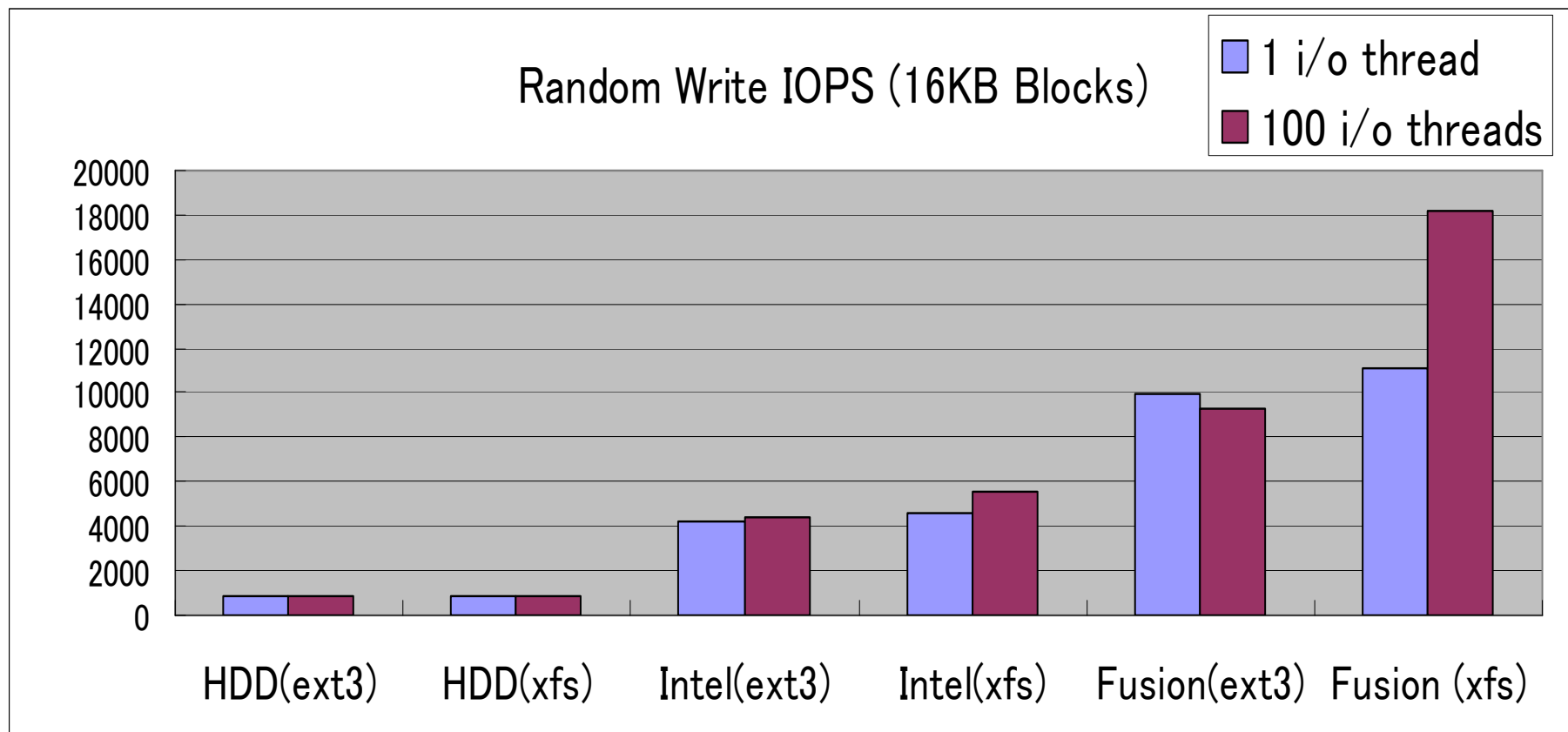
- **ext2**
  - Faster for writes because ext2 doesn't support journaling
  - It takes very long time for fsck
  - On active-active redundancy environment (i.e. MySQL Replication),
    in some cases ext2 is used to gain performance

- **Btrfs (under development)**
  - Copy-on-write filesystem
  - Supporting transactions (no half-block updates)
  - Snapshot backup with no overhead

Concurrent write matters on fast storage

Random Write IOPS (16KB Blocks)

Legend:
- 1 i/o thread
- 100 i/o threads

Categories: HDD(ext3), HDD(xfs), Intel(ext3), Intel(xfs), Fusion(ext3), Fusion (xfs)

Y-axis: 0 to 20000

■ Negligible on HDD (4 SAS RAID1)

■ 1.8 times difference on Fusion I/O

I/O scheduler

- Note: RDBMS (especially InnoDB) also schedules I/O requests so theoretically Linux I/O scheduler is not needed

- Linux has I/O schedulers
  - to efficiently handle lots of I/O requests
  - "I/O scheduler type" and "Queue Size" matters

- Types of I/O schedulers (introduced in 2.6.10: RHEL5)
  - noop: Sorting incoming i/o requests by logical block address, that's all
  - deadlilne: Prioritize read (sync) requests rather than write requests (async) to some extent (to avoid "write-starving-reads" problem)
  - cfq(default): Fairly scheduling i/o requests per i/o thread
  - anticipatory: Removed in 2.6.33 (bad scheduler. Don't use it)

- Default is cfq, but noop / deadline is better in many cases
 - # echo noop > /sys/block/sdX/queue/scheduler

cfq madness

Running two benchmark programs concurrently
1. Multi-threaded random disk reads (Simulating RDBMS reads)
2. Single-threaded overwriting + fsync() (Simulating redo log writes)

| Random Read i/o threads	write+fsync() running	Scheduler	reads/sec from iostat	writes/sec from iostat
1	No	noop/deadline	260	0
		cfq	260	0
100	No	noop/deadline	2100	0
		cfq	2100	0
1	Yes	noop/deadline	212	14480
		cfq	248	246
100	Yes	noop/deadline	1915	12084
		cfq	2084	0

■ In RDBMS, write IOPS is often very high because HDD + write cache can handle thousands of transaction commits per second (write+fsync)

■ Write iops was adjusted to per-thread read iops in cfq, which reduced total iops significantly

Changing I/O scheduler (InnoDB)

DBT-2 (MySQL5.1)
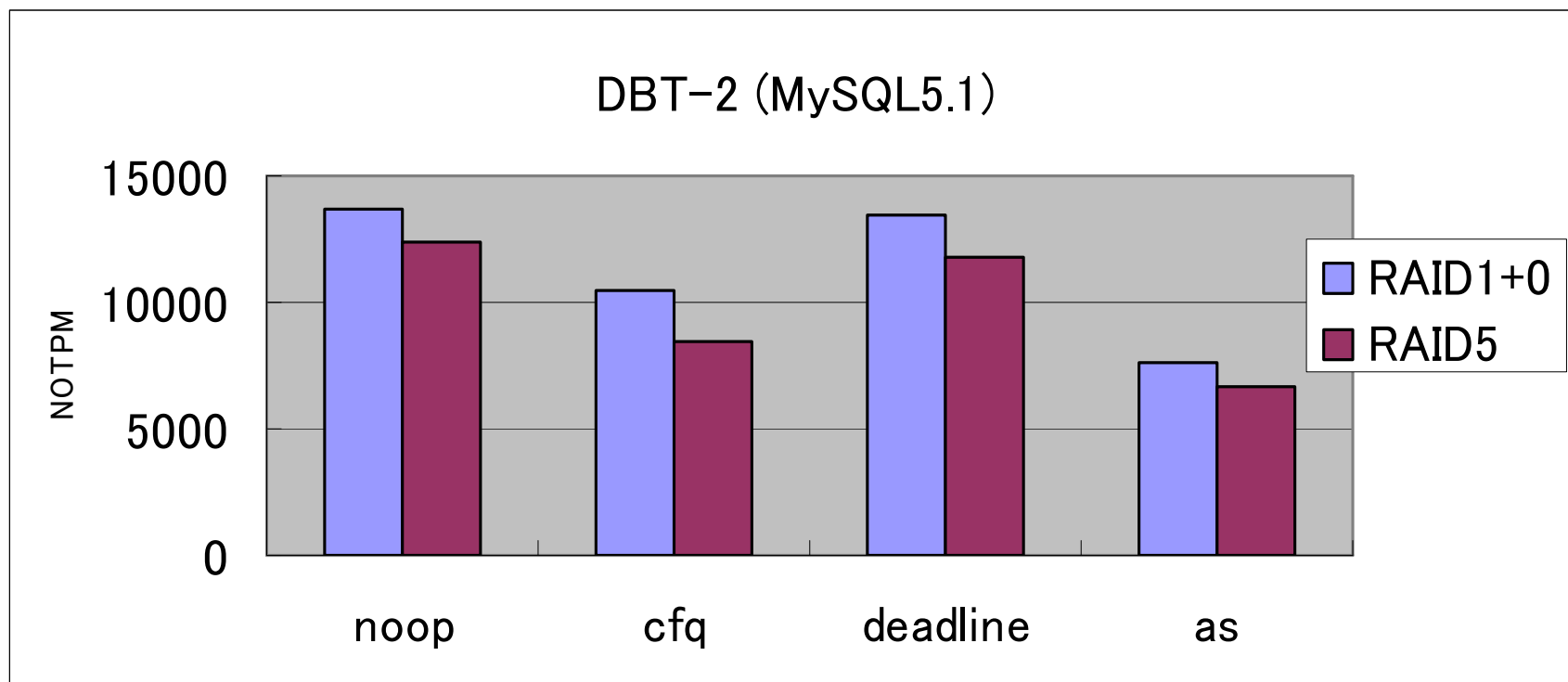
- Sun Fire X4150 (4 HDDs, H/W RAID controller+BBWC)
- RHEL5.3 (2.6.18-128)
- Built-in InnoDB 5.1

Changing I/O scheduler queue size (MyISAM)

Time to insert 1 million records (HDD)
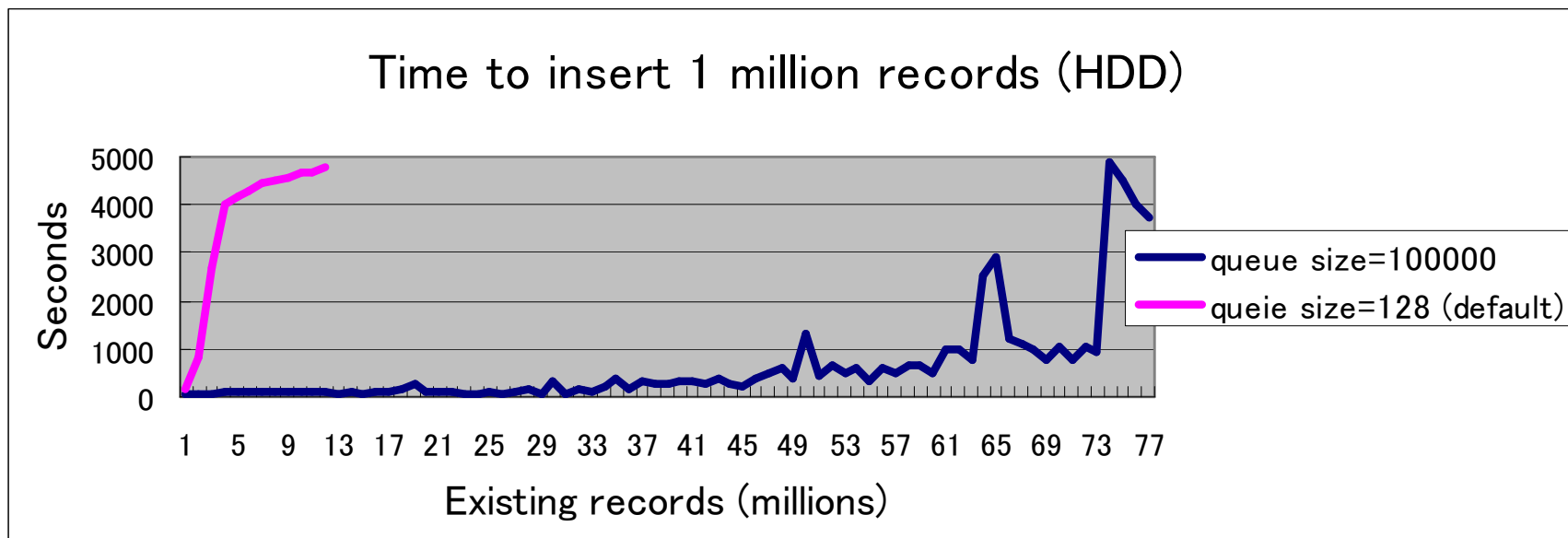
- Queue size = N
  - Sorting N outstanding I/O requests to optimize disk seeks
- MyISAM does not optimize I/O requests internally
  - Highly depending on OS and storage
  - When inserting into indexes, massive random disk writes/reads happen
- Increasing I/O queue size reduces disk seek overheads
  - # echo 100000 > /sys/block/sdX/queue/nr_requests
- No impact in InnoDB
  - Many RDBMS including InnoDB internally sort I/O requests

■ Network

Fighting against network bottlenecks

- **Latency**
  - 100Mbit Ethernet: 3000 us
    - Around 35,000qps from 100 memcached clients
    - Not good if you use SSD
    - Easily reaches 100Mbps bandwidth when copying large files
  - 1Gbit Ethernet: 400us
    - Around 250,000qps from 100 memcached clients
    - 100,000qps is not impossible in MySQL
    - On 100,000qps environments, network bandwidth may matter
  - Latency is not so amazing at 10Gbit Ethernet
    - Check Dolphin Supersockets, etc

- **Single CPU core might be bottleneck**
  - Older Linux kernels do not scale with CPU cores for network requests handling
    - Use CentOS5.7+ when this becomes really issue

Tx/Rx multiqueue

```
# cat /proc/interrupts | grep PCI
202:            … PCI-MSI-X  eth2-0
210:            … PCI-MSI-X  eth2-1
218:            … PCI-MSI-X  eth2-2
226:            … PCI-MSI-X  eth2-3
234:            … PCI-MSI-X  eth2-4
```

- Network requests can be distributed between CPU cores
- Some NICs and drivers work multiqueue on older kernels, but don't work when bonding is enabled
- Check /proc/interrupts

- Check %irq and %soft from mpstat
  - Single CPU core might occupy 100% %soft
- Use latest kernels CentOS5.7+ when this becomes really issue

When multiqueue is helpful?

	memcahced	MySQL (5.6)
w/o multiqueue	309,000 req/s	218,000 select/s
w/ multiqueue	405,000 req/s	218,000 select/s

■ Memcached
  ● w/o multiqueue: 440Mbps in total (Recv 200Mbps, Send 240Mbps), 310,000 packets/s
    – 1 CPU core was fully utilized for software interrupts
    – CPU %user %nice   %sys %iowait   %irq %soft %steal %idle   intr/s
    – 23   0.00   0.00   0.00   0.00   0.00 100.00   0.00   0.00   15.00
  ● W/ multiqueue: 580Mbps in total (Recv 260Mbps, Send 320 Mbps), 420,000 packets/s

■ MySQL: 880Mbps in total (Recv 160Mbps, Send 720Mbps), 220,000 packets/s in both cases
  ● CPU usertime became bottlenecks than software interrupts on 24 core box

Be careful about bandwidth

- Between servers, racks, switches

- Be careful when sending large data
  - Setting up OS from PXE server, or cloning (copying) entire OS image from other machine
  - Sending large backup file
  - Restoring large data and initializing replication

- Do not use 100Mbps networks
  - Network switch, Network Cards, Cables

- When setting up new replication slave, the master sends large binlog events to slaves, which easily occupies 100Mbps traffics (or even 1Gbps network)
  - START/STOP SLAVE IO_THREADS; repeatedly
  - Manually send binlogs by scp with upper limits, then apply events by mysqlbinlog and mysql
  - Scripting may help

Remote datacenter

- **RTT (round trip time) from Tokyo to West coast easily exceeds 200 ms**

- **MySQL Replication for Disaster Recovery purpose**
  - I/O thread delays RTT/2
    - Time to send binary log from master to slave
    - Not bad (ping delays RTT)
    - Lots of binary log events can be sent at once
  - Do not use semi-synchronous replication here
    - Every commit takes RTT for send/recv. Only few transactions can be handled per second

Waiting three seconds for SYN

- **Establishing network connections is expensive**
  - Client: Sending TCP SYN
  - Server: Sending TCP SYN+ACK
  - Client: Sending TCP ACK

- **If client fails to receive SYN+ACK packet, it waits for three seconds, then sending SYN again**
  - This happens quite a lot of times on high traffic environments
  - Not limited to MySQL
  - Increasing backlog helps, but problems still exist
    - echo 8192 > /proc/sys/net/ipv4/tcp_max_syn_backlog
    - back_log in my.cnf

- **Using persistent connections / connection pooling is the fundamental solution**

Flows to establish N/W connections

Client

Server

1. Sending SYN,
Changing state to SYN_SENT

2. Receiving SYN

3. Checking conditions (i.e. back_log)
If it doesn't meet criteria, dropping it

4. Generating SYN+ACK,
Changing state to SYN_RECV

5. Sending SYN+ACK

6. Receiving SYN+ACK

7. Generating ACK

8. Sending ACK

9. Receiving ACK,
Changing state to ESTABLISHED

- If a packet is lost or dropped between 1 and 6, the client re-sends a packet again
  - But the client waits three seconds to resend, which sacrifices response times

Database connections
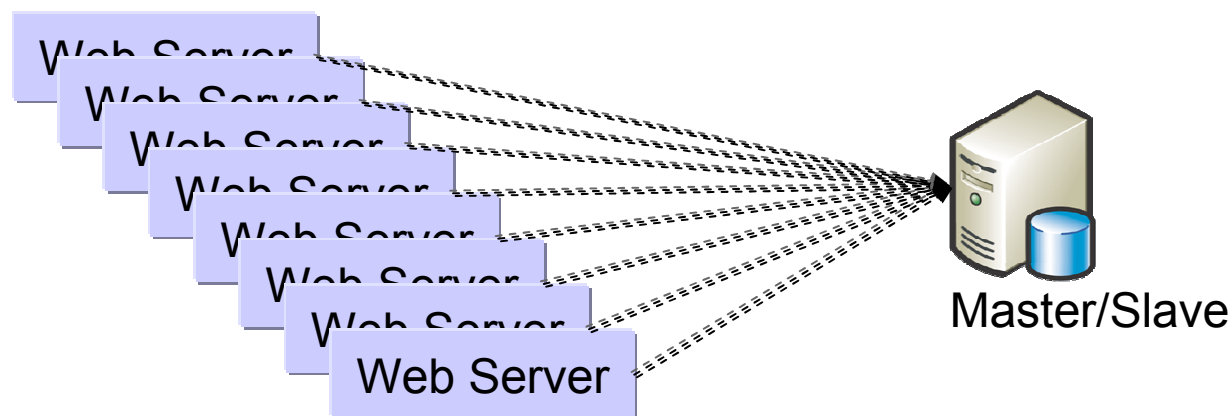
Web Server
Web Server
Web Server
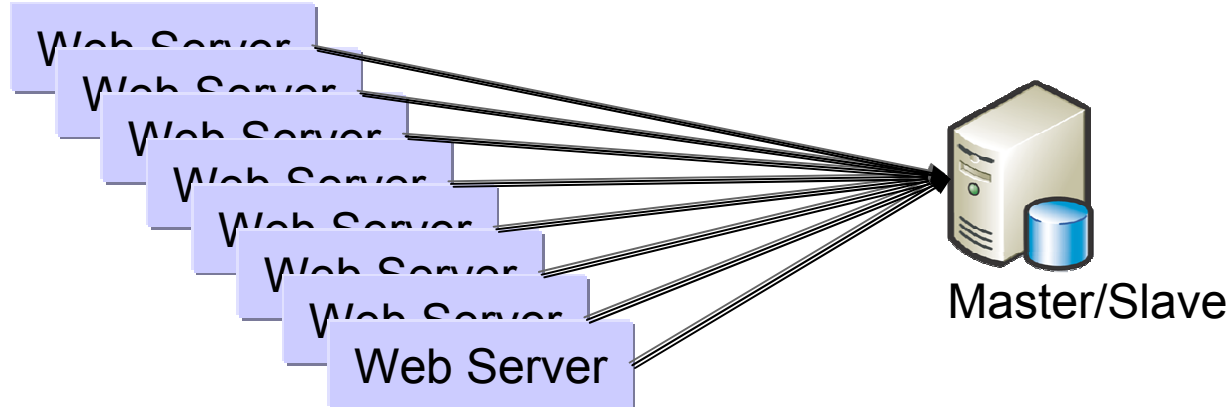Web Server
Web Server
Web Server
Web Server
Web Server

Master/Slave

■ **Non-Persistent connections are majority**
- ● Right now it's not so much issue
  - – Establishing / terminating network connections is not relatively costly compared to disk access overheads
  - – Waiting 3 seconds for TCP SYN retry is frequent problem
- ● Overheads will not be negligible when bottleneck shifts to CPU/Network
  - – Expensive network accesses (SYN-> SYN+ACK -> ACK, initialization SQL statements (SET xxx), etc)

Persistent Connections

Master/Slave

- Keeping connections established permanently
- No connection/disconnection overheads
- The number of connections equals to
  "The number of Web Servers" x "Connections per process"
  - Easily reaches 10000 connections
  - max-connections my.cnf has to be increased
- It's effective as long as the number of web servers is small…
- It doesn't work when the number of web servers becomes large
  - Performance doesn't drop even though there are thousands of idle connections
  - But RAM space is consumed
  - If mysqld stalls for a short period of time (0.5-1 second), suddenly 1000+ clients execute queries at the same time
 - Very serious problems such as OOM might happen

Use persistent connections properly
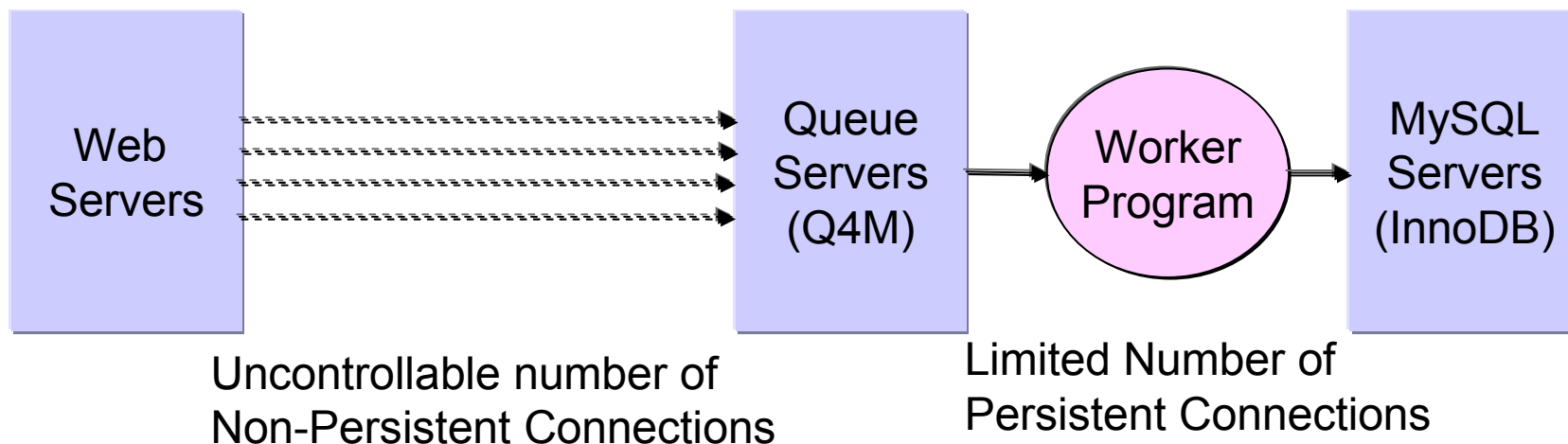
- If the number of connections can be controlled, persistent connection helps a lot
- Before using persistent connections between queue servers and MySQL servers, very often the number of connections was suddenly increased, which caused some errors (1023 error, deadlock error, etc)

Proxy-based connection pooling
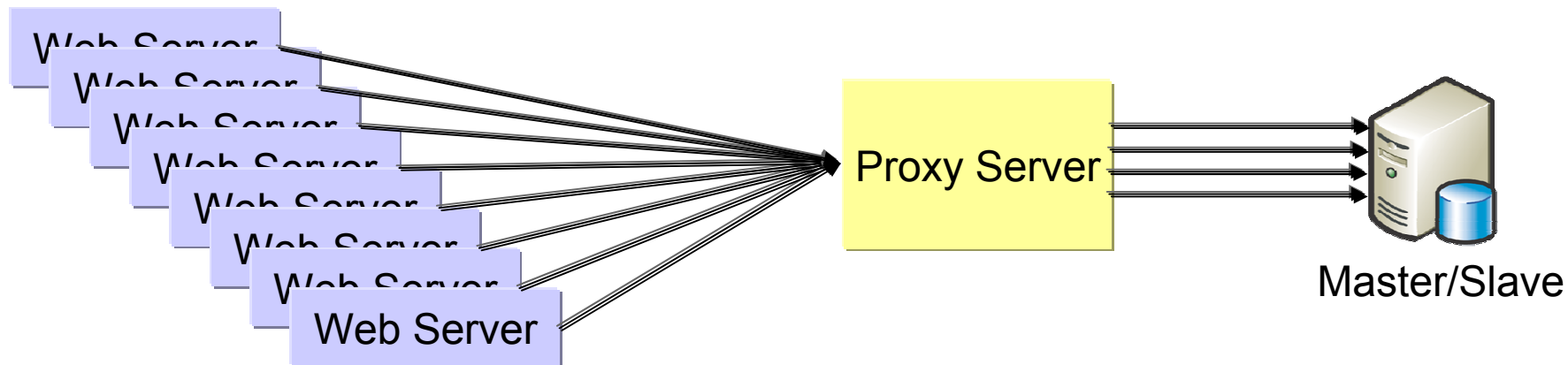
- Web servers establish persistent connections to a proxy server, and Proxy server establishes limited number of persistent connections to MySQL
- Similar to Java EE
- Proxy server is a lightweight network server
    - It's not serious even though the number of connections reach 10,000
- The number of connections at MySQL is limited, so memory usage is ok, but..
    - Proxy server has to receive/send massive network requests
    - Response time increases
    - Need High Availability architecture for Proxy

Thread pooling within MySQL

Master/Slave
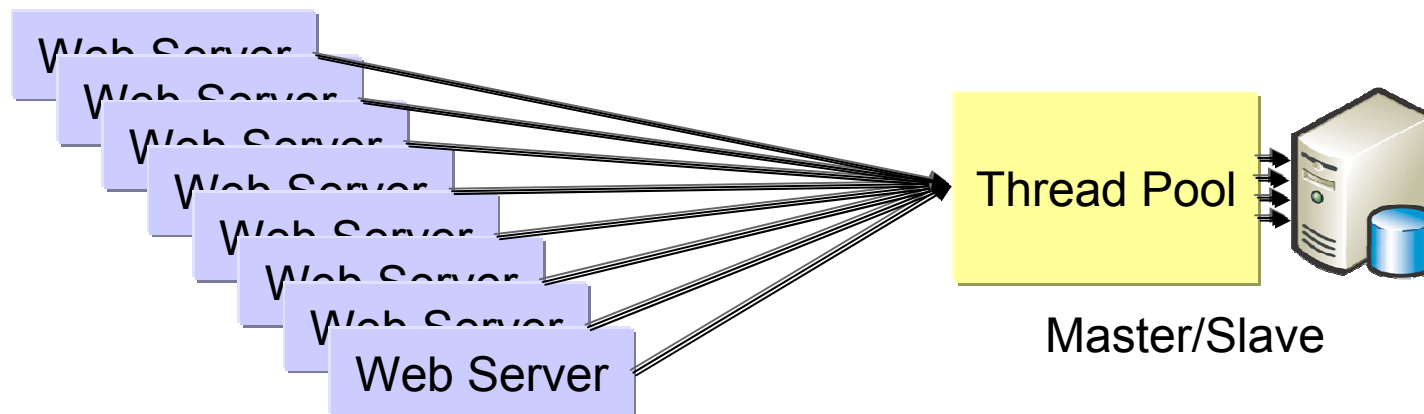
- For stability reasons, the number of threads within mysqld should be limited
- Adding proxy servers costs more, especially when database servers become faster
- It is better that mysqld accepts thousands of connections but running limited number of threads internally: Pool of threads
- MySQL has thread pool plugin interface
- MySQL 5.5 Enterprise

fcntl() serialization

qps (24core, network, in-memory, read only)
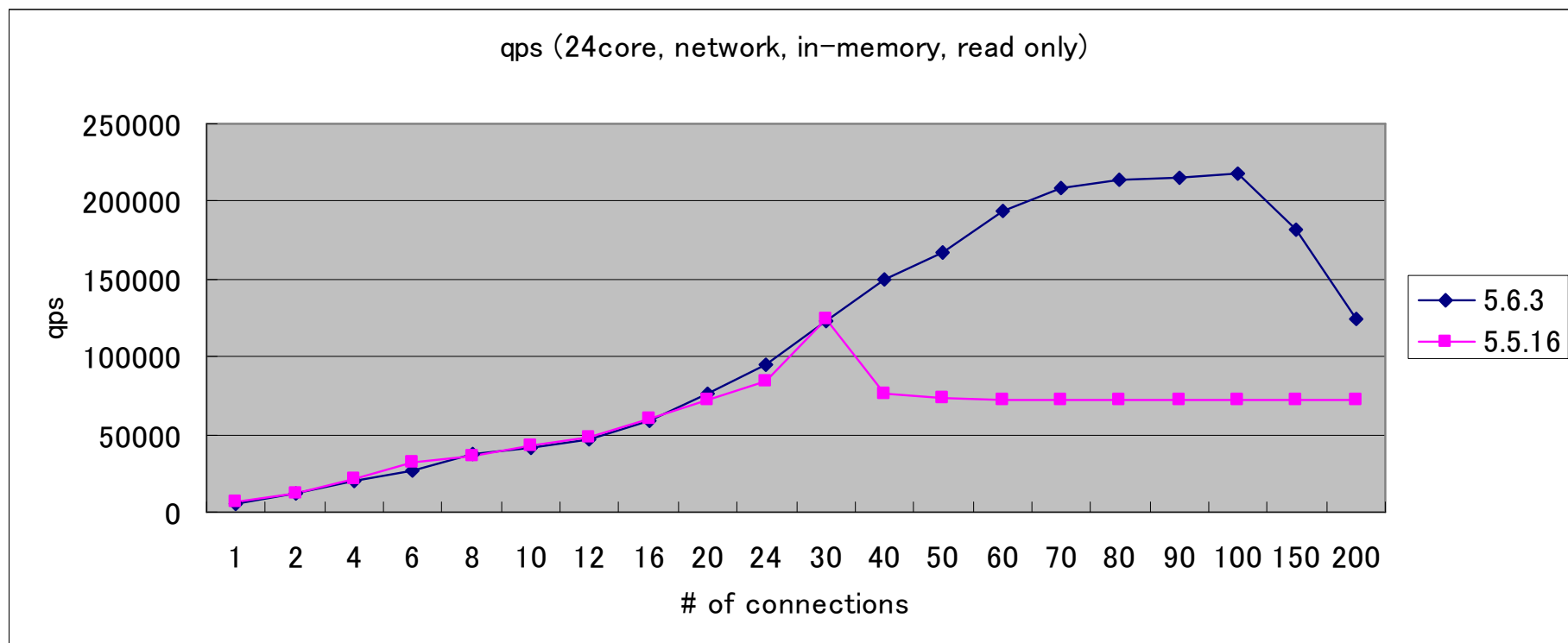
- Prior to 5.6.3, under high loads MySQL internally had to call fcntl() lots of times to change non-blocking sockets to blocking (with timeout and alarm) and vice versa
- fcntl() is serialized on Linux, so it doesn't scale with many CPU cores

Useful commands and tools

- iostat
- mpstat
- dstat
- oprofile
- gdb
- pmp (Poor Man's Profiler)
- gcore

iostat

- Showing detailed I/O statistics per device
- Very important tool because in most cases RDBMS becomes I/O bound
- iostat -x
- Check r/s, w/s, svctm, %util
  - IOPS is much more important than transfer size
- Always %util = (r/s + w/s) * svctm (hard coded in the iostat source file)

```
# iostat –xm 10
avg-cpu: %user %nice %system %iowait %steal %idle
          21.16  0.00    6.14    29.77   0.00 42.93
Device: rqm/s wrqm/s    r/s    w/s  rMB/s  wMB/s avgrq-sz avgqu-sz await svctm %util
   sdb    2.60 389.01 283.12  47.35  4.86   2.19    43.67      4.89 14.76  3.02 99.83
```

(283.12+47.35) * 3.02(ms)/1000 = 0.9980 = 100% util

iostat example (DBT-2)

```
# iostat -xm 10
avg-cpu:  %user %nice %system %iowait %steal %idle
          21.16  0.00    6.14    29.77   0.00  42.93
Device: rqm/s wrqm/s    r/s     w/s  rMB/s wMB/s avgrq-sz avgqu-sz await svctm %util
   sdb    2.60 389.01 283.12 47.35  4.86  2.19     43.67       4.89 14.76  3.02 99.83
```

(283.12+47.35) * 3.02(ms)/1000 = 0.9980 = 100% util

```
# iostat -xm 10
avg-cpu:  %user %nice %system %iowait %steal %idle
          40.03  0.00   16.51    16.52   0.00  26.94
Device: rrqm/s wrqm/s    r/s     w/s rMB/s wMB/s avgrq-sz avgqu-sz await svctm %util
   sdb    6.39 368.53 543.06 490.41  6.71  3.90    21.02       3.29  3.20  0.90 92.66
```

(543.06+490.41) * 0.90(ms)/1000 = 0.9301 = 93% util

- **Sometimes throughput gets higher even though %util reaches 100%**
  - Write cache, Command Queuing, etc
- **In both cases %util is almost 100%, but r/s and w/s are far different**
- **Do not trust %util too much**
- **Check svctm rather than %util**
  - If your storage can handle 1000 IOPS, svctm should be less than 1.00 (ms) so you can send alerts if svctm is higher than 1.00 for a couple of minutes

- Per CPU core statistics

- vmstat displays average statistics

- It's very commmom that only one of CPU cores consumes 100% CPU resources
  - The rest CPU cores are idle
  - Especially applies to batch jobs

- If you check only vmstat/top/iostat/sar you will not notice single threaded bottleneck

- You can also check network bottlenecks (%irq, %soft) from mpstat
  - vmstat counts them as %idle

vmstat and mpstat

```
# vmstat 1
...
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
 r  b    swpd    free    buff   cache    si    so    bi    bo    in    cs us sy id wa st
 0  1 2096472 1645132   18648   19292     0     0  4848     0  1223   517  0  0 88 12  0
 0  1 2096472 1645132   18648   19292     0     0  4176     0  1287   623  0  0 87 12  0
 0  1 2096472 1645132   18648   19292     0     0  4320     0  1202   470  0  0 88 12  0
 0  1 2096472 1645132   18648   19292     0     0  3872     0  1289   627  0  0 87 12  0
```

```
# mpstat -P ALL 1
...
11:04:37 AM  CPU    %user    %nice    %sys %iowait    %irq   %soft   %steal    %idle   intr/s
11:04:38 AM  all     0.00     0.00    0.12   12.33    0.00    0.00     0.00    87.55  1201.98
11:04:38 AM    0     0.00     0.00    0.00    0.00    0.00    0.00     0.00   100.00   990.10
11:04:38 AM    1     0.00     0.00    0.00    0.00    0.00    0.00     0.00   100.00     0.00
11:04:38 AM    2     0.00     0.00    0.00    0.00    0.00    0.00     0.00   100.00     0.00
11:04:38 AM    3     0.00     0.00    0.00    0.00    0.00    0.00     0.00   100.00     0.00
11:04:38 AM    4     0.99     0.00    0.99   98.02    0.00    0.00     0.00     0.00   206.93
11:04:38 AM    5     0.00     0.00    0.00    0.00    0.00    0.00     0.00   100.00     0.00
11:04:38 AM    6     0.00     0.00    0.00    0.00    0.00    0.00     0.00   100.00     4.95
11:04:38 AM    7     0.00     0.00    0.00    0.00    0.00    0.00     0.00   100.00     0.00
```

- vmstat displays average statistics. 12% * 8 (average) = 100% * 1 + 0% * 7

Typical mpstat output

- **Replication Slave**
  - %idle on a single CPU core becomes 0-10%
    - Mostly used for %user, %system, %iowait, %soft

- **Bulk loading/inserting on InnoDB**
  - %idle on a single CPU core becomes 0-10%
    - Mostly used for %user, and %iowait

dstat

```
# dstat
----total-cpu-usage---- -dsk/total- -net/total- ---paging-- ---system--
usr sys idl wai hiq siq| read  writ| recv  send|  in   out | int   csw
  3   1  95   0   0   0|2096k   13M|1437k 2096k|   0     0 |5644   11k
  3   2  95   0   0   0|1752k 4096k|1555k 2372k|   0     0 |6015   11k
  3   1  95   0   0   0|4640k   12M|1689k 2659k|   0     0 |6092   12k
  2   1  96   0   0   0|1616k 5352k|1590k 2493k|   0     0 |5593   10k
  3   1  96   0   0   0|1968k 6960k|1558k 2400k|   0     0 |5663   10k

# dstat -N bond0
----total-cpu-usage---- -dsk/total- -net/bond0- ---paging-- ---system--
usr sys idl wai hiq siq| read  writ| recv  send|  in   out | int   csw
  2   1  95   1   0   0|2416k 5112k| 782k 1196k|   0     0 |5815   11k
  3   1  95   0   0   0|3376k   15M| 792k 1213k|   0     0 |5632   11k
  3   2  95   0   0   0|2120k 3264k| 793k 1229k|   0     0 |5707   11k
  2   1  96   0   0   0|1920k 3560k| 788k 1193k|   0     0 |5856   11k
  2   1  96   0   0   0|1872k   13M| 770k 1112k|   0     0 |5463   10k
```

■ yum install dstat
■ Similar UI as vmstat, but it also shows network statistics
■ Disk and Net total is incorrect, if you use RAID/bonding (double counted)
   ● Filter by disk/interface name that you want to trace
■ "mtstat" additionally supports mysql status outputs
   ● https://launchpad.net/mtstat

Oprofile

- Profiling CPU usage from running processes
- You can easily identify which functions consume CPU resources
- Supporting both user space and system space profiling
- Mainly used by database-internal developers
- If specific functions consume most of recourses, applications might be re-designed to skip calling them
- Not useful to check low-CPU activities
  - I/O bound, mutex waits, etc

- How to use
  - opcontrol --start (--no-vmlinux)
  - benchmarking
  - opcontrol --dump
  - opcontrol --shutdown
  - opreport -l /usr/local/bin/mysqld

Oprofile example

```
# opreport -l /usr/local/bin/mysqld
samples  %         symbol name
83003       8.8858  String::copy(char const*, unsigned int, charset_info_st*,
charset_info_st*, unsigned int*)
79125       8.4706  MYSQLparse(void*)
68253       7.3067  my_wc_mb_latin1
55410       5.9318  my_pthread_fastmutex_lock
34677       3.7123  my_utf8_uni
18359       1.9654  MYSQLlex(void*, void*)
12044       1.2894  _ZL15get_hash_symbolPKcjb
11425       1.2231  _ZL20make_join_statisticsP4JOINP10TABLE_LISTP4ItemP16st_dynamic_array
```

- You can see quite a lot of CPU resources were spent for character conversions (latin1 <-> utf8)
- Disabling character code conversions on application side will improve performance

```
samples  %         symbol name
83107      10.6202  MYSQLparse(void*)
68680       8.7765  my_pthread_fastmutex_lock
20469       2.6157  MYSQLlex(void*, void*)
13083       1.6719  _ZL15get_hash_symbolPKcjb
12148       1.5524  JOIN::optimize()
11529       1.4733  _ZL20make_join_statisticsP4JOINP10TABLE_LISTP4ItemP16st_dynamic_array
```

Checking stalls

- **Database servers sometimes suffer from performance stalls**
  - Not responding anything for 1-2 seconds

- **In many cases it's database internal issue**
  - Holding global mutex and doing expensive i/o
  - Buffer pool, redo log preflush, rollback segments (InnoDB mutex)
  - pthread_create()/clone() (LOCK_thread_count mutex)
  - Opening/closing tables (LOCK_open mutex)

- **Collecting statistics**
  - Per-minute statistics (i.e. vmstat 60) is not helpful. Collect per-second statistics

Checking stalls by gdb or pmp

- ■ Debugging tool

- ■ gdb has a functionality to take thread stack dumps from a running process

- ■ Useful to identify where and why mysqld hangs up, slows down, etc
  - ● But you have to read MySQL source code

- ■ During taking stack dumps, all threads are stopped

- ■ Debugging symbol is required on the target program

gdb case study

```
mysql> SELECT query_time, start_time, sql_text
    -> FROM mysql.slow_log WHERE start_time
    -> BETWEEN '2010-02-05 23:00:00' AND '2010-02-05 01:00:00'
    -> ORDER BY query_time DESC LIMIT 10;
+------------+---------------------+----------+
| query_time | start_time          | sql_text |
+------------+---------------------+----------+
| 00:00:11   | 2010-02-05 23:09:55 | begin    |
| 00:00:09   | 2010-02-05 23:09:55 | Prepare  |
| 00:00:08   | 2010-02-05 23:09:55 | Prepare  |
| 00:00:08   | 2010-02-05 23:09:55 | Init DB  |
| 00:00:08   | 2010-02-05 23:09:55 | Init DB  |
| 00:00:07   | 2010-02-05 23:09:55 | Prepare  |
| 00:00:07   | 2010-02-05 23:09:55 | Init DB  |
| 00:00:07   | 2010-02-05 23:09:55 | Init DB  |
| 00:00:07   | 2010-02-05 23:09:55 | Init DB  |
| 00:00:06   | 2010-02-05 23:09:55 | Prepare  |
+------------+---------------------+----------+
10 rows in set (0.02 sec)
```

- **Suddenly all queries were not responding for 1-10 seconds**
- **Checking slow query log**
- **All queries are simple enough, it's strange to take 10 seconds**
- **CPU util (%us, %sy) were almost zero**
- **SHOW GLOBAL STATUS, SHOW FULL PROCESSLIST were not helpful**

Taking thread dumps with gdb

```
gdbtrace() {
…

  PID=`cat /var/lib/mysql/mysql.pid`
  STACKDUMP=/tmp/stackdump.$$
  echo 'thread apply all bt' >
$STACKDUMP
  echo 'detach' >> $STACKDUMP
  echo 'quit' >> $STACKDUMP
  gdb --batch --pid=$PID -x $STACKDUMP
}

while loop
do
  CONN=`netstat -an | grep 3306 | grep
ESTABLISHED | wc | awk '{print $1}'`
  if [ $CONN -gt 100 ]; then
    gdbtrace()
  done
  sleep 3
done
```

- Attaching running mysqld, then taking a thread dump
- Taking dumps every 3 seconds
- Attaching & Dumping with gdb is expensive so invoke only when exceptional scenario (i.e. conn > threshold) happens

- Check if the same LWPs are waiting at the same place

Stack Trace

```
.....
Thread 73 (Thread 0x46c1d950 (LWP 28494)):
#0  0x00007ffda5474384 in __lll_lock_wait () from /lib/libpthread.so.0
#1  0x00007ffda546fc5c in _L_lock_1054 () from /lib/libpthread.so.0
#2  0x00007ffda546fb30 in pthread_mutex_lock () from /lib/libpthread.so.0
#3  0x0000000000a0f67d in my_pthread_fastmutex_lock (mp=0xf46d30) at
thr_mutex.c:487
#4  0x000000000060cbe4 in dispatch_command (command=16018736, thd=0x80,
    packet=0x65 <Address 0x65 out of bounds>, packet_length=4294967295)
    at sql_parse.cc:969
#5  0x000000000060cb56 in do_command (thd=0xf46d30) at sql_parse.cc:854
#6  0x0000000000607f0c in handle_one_connection (arg=0xf46d30) at
sql_connect.cc:1127
#7  0x00007ffda546dfc7 in start_thread () from /lib/libpthread.so.0
#8  0x00007ffda46305ad in clone () from /lib/libc.so.6
#9  0x0000000000000000 in ?? ()
```

- Many threads were waiting at pthread_mutex_lock(), called from sql_parse.cc:969

```
953  bool dispatch_command(enum enum_server_command command, THD *thd,
954                        char* packet, uint packet_length)
955  {
956    NET *net= &thd->net;
957    bool error= 0;
958    DBUG_ENTER("dispatch_command");
959    DBUG_PRINT("info",("packet: '%*.s'; command: %d", packet_length,
packet, command));
960
961    thd->command=command;
962    /*
963      Commands which always take a long time are logged into
964      the slow log only if opt_log_slow_admin_statements is set.
965    */
966    thd->enable_slow_log= TRUE;
967    thd->lex->sql_command= SQLCOM_END; /* to avoid confusing VIEW
detectors */
968    thd->set_time();
969    VOID(pthread_mutex_lock(&LOCK_thread_count));
```

Who locked LOCK_thread_count for seconds?

```
Thread 1 (Thread 0x7ffda58936e0 (LWP 15380)):
#0  0x00007ffda4630571 in clone () from /lib/libc.so.6
#1  0x00007ffda546d396 in do_clone () from /lib/libpthread.so.0
#2  0x00007ffda546db48 in pthread_create@@GLIBC_2.2.5 () from
/lib/libpthread.so.0
#3  0x0000000000600a66 in create_thread_to_handle_connection (thd=0x3d0f00)
    at mysqld.cc:4811
#4  0x00000000005ff65a in handle_connections_sockets (arg=0x3d0f00) at
mysqld.cc:5134
#5  0x00000000005fe6fd in main (argc=4001536, argv=0x4578c260) at
mysqld.cc:4471
#0  0x00007ffda4630571 in clone () from /lib/libc.so.6
```

- gdb stack dumps were taken every 3 seconds
- In all cases, Thread 1 (LWP 15380) was stopped at the same point
- clone() (called by pthread_create()) seemed to take a long time

```
4795 void create_thread_to_handle_connection(THD *thd)
4796 {
4797   if (cached_thread_count > wake_thread)
4798   {
4799     /* Get thread from cache */
4800     thread_cache.append(thd);
4801     wake_thread++;
4802     pthread_cond_signal(&COND_thread_cache);
4803   }
4804   else
4805   {
4811     if ((error=pthread_create(&thd->real_id, &connection_attrib,
4812                               handle_one_connection,
4813                               (void*) thd)))
4839   }
4840   (void) pthread_mutex_unlock(&LOCK_thread_count);
```

- pthread_create is called under critical section (LOCK_thread_count is released after that)
- If cached_thread_count > wake_thread, pthread_create is not called
- Increasing thread_cache_size will fix the problem!

Poor Man's Profiler

- A simple shell script (wrapper script) to summarize thread call stacks
    - Using gdb thread dump functionality
- Useful for identifying why MySQL stalls
- http://poormansprofiler.org/

```
#!/bin/bash
nsamples=1
sleeptime=0
pid=$(pidof mysqld)
for x in $(seq 1 $nsamples)
  do
    gdb -ex "set pagination 0" -ex "thread apply all bt" -batch -p $pid
    sleep $sleeptime
  done | ¥
awk '
  BEGIN { s = ""; }
  /Thread/ { print s; s = ""; }
  /^¥#/ { if (s != "" ) { s = s "," $4} else { s = $4 } }
  END { print s }' | ¥
sort | uniq -c | sort -r -n -k 1,1
```

pmp output example

291 pthread_cond_wait@@GLIBC_2.3.2,one_thread_per_connection_end,handle_one_connection
 57 read,my_real_read,my_net_read,do_command,handle_one_connection,start_thread
 26
pthread_cond_wait@@GLIBC_2.3.2,os_event_wait_low,os_aio_simulated_handle,fil_aio_wait,io_handler_
thread,start_thread
  3 pthread_cond_wait@@GLIBC_2.3.2,os_event_wait_low,srv_purge_worker_thread
  1 select,os_thread_sleep,srv_purge_thread
  1 select,os_thread_sleep,srv_master_thread
  1 select,os_thread_sleep,srv_lock_timeout_and_monitor_thread
  1 select,os_thread_sleep,srv_error_monitor_thread
  1 select,handle_connections_sockets,main,select
  1 read,vio_read_buff,my_real_read,my_net_read,cli_safe_read,handle_slave_io
  1
pthread_cond_wait@@GLIBC_2.3.2,os_event_wait_low,sync_array_wait_event,rw_lock_s_lock_spin,buf_pa
ge_get_gen,btr_cur_search_to_nth_level,row_search_for_mysql,ha_innodb::index_read,handler::index_
read_idx_map,join_read_const,join_read_const_table,make_join_statistics,JOIN::optimize,mysql_sele
ct,handle_select,execute_sqlcom_select,mysql_execute_command,mysql_parse,dispatch_command,do_comm
and,handle_one_connection

Disastvantages of gdb/pmp

■ All threads are stopped while taking thread dumps

■ The more the number of threads, the more time it takes

- It might take 5 seconds if 1000 threads are running
- 5 seconds stall is not acceptable in many cases
- It is not recommended taking thread dumps on master when lots of threads are running
- Be careful if you set high thread_cache_size

gcore

- **Dumping core file**
- **Dumping core file takes time, and blocks all threads during dump**
  - Might take 1 hour when VIRT is 50GB+
  - Not useful on running systems, but can be used on a non-serving slave
- **If you have a program with DWARF symbols, target (running) process does not need symbols**
  - You can run stripped mysqld on production, as long as you have debug-built mysqld

Case study

Problem:
- mysqld on one of slaves used much more CPU resources than other slaves, and frequently replication delay happened
- the mysqld used 50GB+ VIRT on 24G box. (other slaves used 20G VIRT)

```
top - 20:39:14 up 360 days, 17:56,  1 user,  load average: 1.26, 1.29, 1.32
Tasks: 125 total,   2 running, 123 sleeping,   0 stopped,   0 zombie
Cpu(s): 10.8% us,  0.8% sy,  0.0% ni, 87.2% id,  0.7% wa,  0.0% hi,  0.5% si
Mem:  24680588k total, 24609132k used,   71456k free,   99260k buffers
Swap:  4192956k total,   160744k used,  4032212k free,  4026256k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
11706 mysql     16   0 50.5g  19g 5168 R 90.9 81.4 247117:59 mysqld
19253 root      16   0  6172 1164  852 R  0.3  0.0   0:00.03 top
```

From mysql status

Slave_open_temp_tables was extremely high.

| Slave_open_temp_tables | 15927 |

But we don't use tmp tables right now.

Previously developers created tmp tables, but in theory SQL
thread should have closed them.

SQL thread internally checks all its tmp tables
whether it can be closed or not, per each SQL statement.

Parsing 15927 tmp tables per statement was very expensive
and it should be the reason that caused replication delay.

I wanted to know what tables were opened, but
MySQL command line tools do not provide ways to identify them.

Time to analyze core file !

Core file analysis example

```
# Install debuginfo mysqld if not installed (don't need to run debugged mysqld)
# ulimit -c unlimited
# gdb -p $(pidof mysqld)
(gdb) gcore core  (Takes long time..)
(gdb) detach

# gdb /usr/lib/debug/usr/sbin/mysqld.debug /data/mysql/core.11706

(gdb) p active_mi->rli->save_temporary_tables->s->table_name
$1 = {str = 0x31da400c3e "claimed_guids", length = 13}
(gdb) p $a->file->stats
$16 = {data_file_length = 1044496, max_data_file_length = 5592400,
 index_file_length = 1044496, max_index_file_length = 0, delete_length = 0,
 auto_increment_value = 0, records = 1, deleted = 0, mean_rec_length = 8,
 create_time = 0, check_time = 0, update_time = 0, block_size = 0}
```

- Tmp tables are MEMORY engine so 1044496 + 1044496 bytes (2MB) were used per table.
- Table name was "claimed_guids"

Dumping slave's tmp tables info

```
define print_all_tmp_tables
 set $a= active_mi->rli->save_temporary_tables
 set $b= slave_open_temp_tables
 while ($i < $b)
  p $a->alias
  p $a->file->stats
  set $a= $a->next
  set $i=$i+1
 end
end

set pagination 0
set $i=0
print_all_tmp_tables
detach
quit
```

gdb /usr/lib/debug/usr/sbin/mysqld.debug /data/mysql/core.11706 -x above_script

List length was 15927 (same as slave_open_tmp_tables), all tables were the
same name, used same size (2MB)
15927 * 2MB = 31GB

This should be the reason why VIRT was 30+ GB larger than other slaves.
Rebooting the slave should be fine because they are tmp tables.

Capturing MySQL packets

■ Capturing MySQL packets is useful to trace slow queries / transactions / connection requests.
  ● Modifying applications MySQL settings is not needed
    – Starting from 5.1, you can dynamically enable/disable slow/general query logs

■ libpcap
  ● library to capture network packets
  ● Most of network capturing tools including tcpdump rely on libpcap
  ● Packet loss often happens

tcpdump + mk-query-digest

1) [root#] tcpdump -i bond0 port 3306 -s 65535 -x -n -q -tttt -c 10000 > tcpdump.out

2) $ mk-query-digest --type=tcpdump --report-format=query_report tcpdump.out
```
# Query 1: 951.34 QPS, 1.56x concurrency, ID 0x6D23B63A5DA67F5D at byte …
# Attribute    pct   total    min    max    avg    95% stddev  median
# ========= ====== ======= ======= ======= ======= ======= =======…
# Count        22   1837
# Exec time    40    3s      0  207ms   2ms    8ms    6ms  236us
…
SELECT * FROM user WHERE user_id = 100;
# Query 2: 1.06k QPS, 0.69x concurrency, ID 0x2EB286C018F2DE27 at byte …
…
```

- mk-query-digest has a functionality to parse tcpdump outputs
    - It can trace how long each query takes, and it can summarize and sort by total execution time
- No need to modify my.cnf settings
- Queries that take less than 1 second can be captured
    - Very helpful in MySQL 5.0 or eariler

Monitoring lock contention

- ■ Lock wait timeout / Long running transactions
  - ● Many clients may be blocked, resulting in too many connections / slow response time
  - ● Lock wait timeout error can be traced on application side (logging errors)
  - ● Set innodb_lock_wait_timeout to smaller size (we set 4, default 50)

- ■ Deadlocks between multiple MySQL servers
  - client A
  1. Exclusive lock on id=100, master1
  - client B
  2. Exclusive lock on id=200, master2
  - client A
  3. Exclusive lock on id=200, master2 (lock wait)
  - client B
  4. Exclusive lock on id=100, master1 (lock wait)

  - ● InnoDB can't detect cross-instance deadlock, so either client A or B has to wait until innodb_lock_wait_timout
  - ● Follow best practices for getting locks (Ordered locking)

MySlowTranCapture to detect long trans

```
[mysql-server]# myslowtrancapture -i eth0 –t 4000
Monitoring eth0 interface..
Listening port 3306..
Transactions that take more than 4000
milliseconds..

From 192.168.0.1:24441
2011/01/07 09:12:17.258307 ->
begin
2011/01/07 09:12:17.258354 <-
GOT_OK
2011/01/07 09:12:17.264797 ->
select * from diary where diary_id=100 for update
2011/01/07 09:12:17.265087 <-
GOT_RES
2011/01/07 09:13:01.232620 ->
update diary set diary_date=now() where
diary_id=100
2011/01/07 09:13:01.232960 <-
GOT_OK
2011/01/07 09:13:17.360993 ->
commit
```

```
From 192.168.0.2:24442
2011/01/07 09:12:20.969288 ->
begin
2011/01/07 09:12:20.969483 <-
GOT_OK
2011/01/07 09:12:20.977699 ->
update diary set diary_date=now() where
diary_id=100
2011/01/07 09:13:11.300935 <-
GOT_ERR:Lock wait timeout exceeded; try
restarting transaction
2011/01/07 09:13:13.136967 ->
rollback
```

- A tool to print "transactions" that take more than N milliseconds
- https://github.com/yoshinorim/MySlowTranCapture

■ Summary

SSD

- **SATA SSD**
  - Much more cost effective than spending on RAM only
  - Recent RAID controller works much better than before
  - Be careful about single threaded read iops

- **PCI-Express SSD**
  - Check FusionIO and tachIOn
  - Expensive, but MLC might be affordable
  - Master handles traffics greatly, but slave servers can not catch up due to single threaded replication channel
  - Consider running multiple instances on a slave server

RAM

- Allocate swap space (approx half of RAM size)
- Set vm.swappiness = 0 and use O_DIRECT
  - Restart linux if kernel panic happens
    - kernel.panic_on_oops = 1
    - kernel.panic = 1
- Use linux kernel 2.6.28+, or frequently unmap logs from filesystem cache

- Do not allocate too much memory per session

- Do not use persistent connections if you can't control maximum number of MySQL internal threads

File I/O

- Set /sys/block/sdX/queue/scheduler = deadline or noop
- Filesystem Tuning
  - relatime (noatime)
  - ext3: tune2fs −O dir_index -c −l −i 0
  - xfs: nobarrier
  - Make sure write cache with battery/flash is enabled
- Others
  - Make sure to allocate separate database partitions (/var/lib/mysql, /tmp) from root partition (/)
    - When database size becomes full, it should not affect Linux kernels
  - /etc/security/limits.conf
    - soft nofile 8192
    - hard nofile 8192

CPU, Network, Virtualization

- These need to be considered more, especially when you deploy SSD

- HyperThreading should be enabled. Now MySQL scales well with many logical CPU cores

- Utilize Tx/Rx multiqueue. Latest kernel (including RHEL6) handles better

- I prefer running multiple MySQL instances on a single OS than running multiple VMs