CSDI Review

Rong Chen
Institute of Parallel and Distributed Systems
Shanghai Jiao Tong University
http://ipads.se.sjtu.edu.cn

Course Topics

File Systems/Flash FS

Non-volatile Memory

Crash Recovery

Multiprocessor: Concurrency and Locks

Transactional Memory/DB

NoSQL

Distributed Transactions

RDMA-enable Transaction Processing

RDMA-friendly Key/value Store

Low (Tail) Latency

Network Function Virtualization

Graph Query on RDF

Bugs

Grading

General (Tentative)

- Final Exam：60%
  - Papers will be reflected in exam
- Paper reading, Q&A: 10%
  - Help you understand systems
- Paper presentation: 25%
- Course performance 5%
  - Q&A in classroom

Flash Memory Review

Non-volatile solid state memory

- Individual cells are comparable in size to a transistor
- Not sensitive to mechanical shock
- Re-write requires prior bulk erase
- Limited number of erase/write cycles

Two categories of flash:

- NOR flash: random access, used for firmware
- NAND flash: block access, used for mass storage

Two types of memory cells:

- SLC: single level cell that encodes a single bit per cell
- MLC: multi-level cell that encodes multiple bits per cell

F2FS: Motivation

NAND flash memory has been widely used in various devices.
- Server system started utilizing flash devices as their primary storage.

BUT, there are several limitations on flash memory.
- erase-before-write, write on erased block sequentially and limited write cycles per erase block.

Random writes are not good for flash storage devices.
- Free space fragmentation.
  - Sustained random write performance degrades.
  - Lifetime reduction.

Sequential write oriented file system
- Log structured file system, copy-on-write file system.

Summary

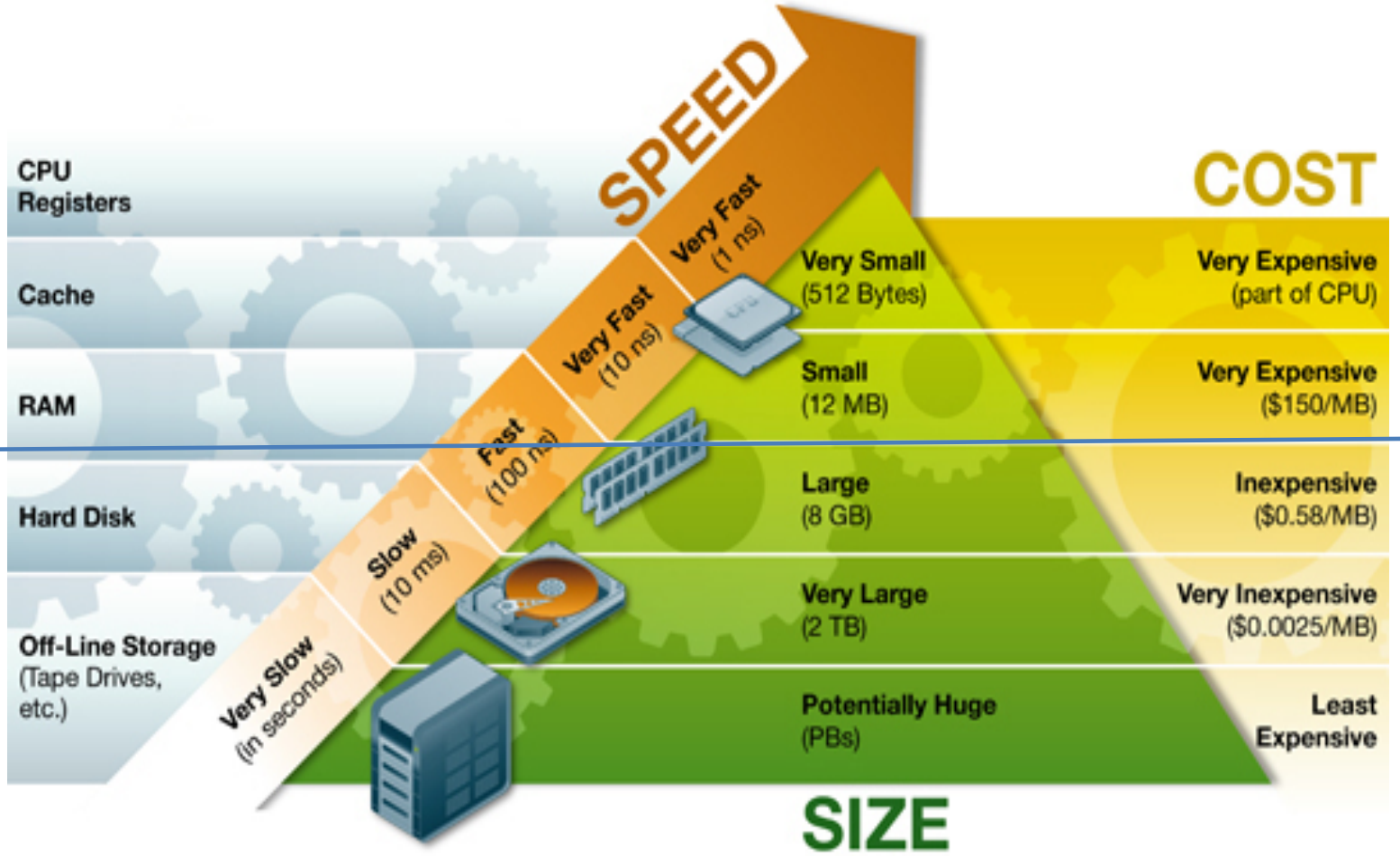
F2FS features

- Flash friendly on-disk layout → align FS GC unit with FTL GC unit,
- Cost-effective index structure → restrain write propagation,
- Multi-head logging → cleaning cost reduction,
- Adaptive logging → graceful performance degradation in aged condition,
- Roll-forward recovery → fsync acceleration.

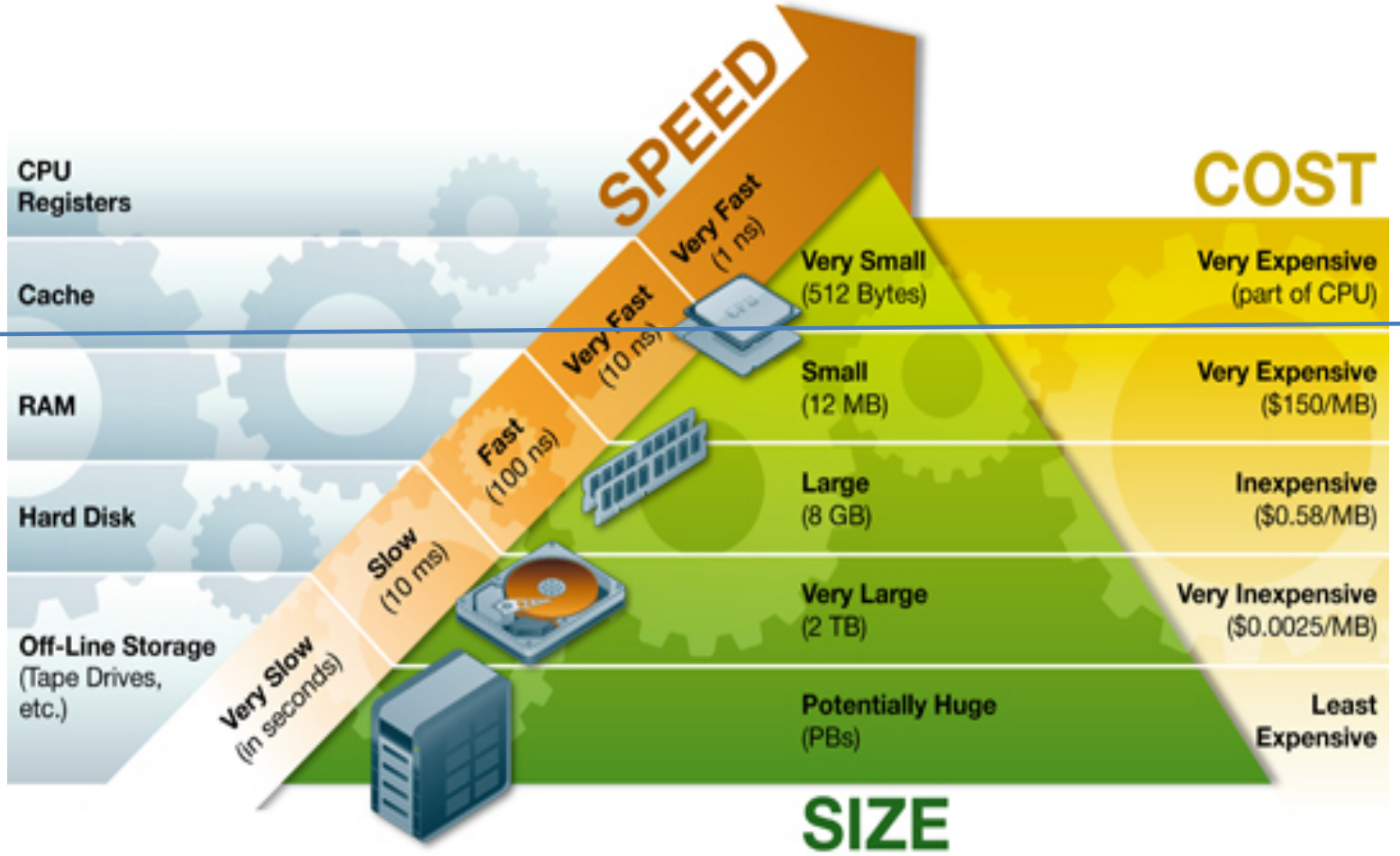
F2FS shows performance gain over other Linux file systems.

- 3.1x(iozone) and 2x (SQLite) speedup over ext4,
- 2.5x(SATA SSD) and 1.8x(PCIe SSD) speedup over ext4(varmail)

Memory Hierarchy



NVM Memory Hierarchy



Tape is Dead

Disk is Tape

Flash is Disk

RAM is Flash?

Cache Locality/Pallelism is King?

PMFS Overview

Introduction of *pm_wbarrier*

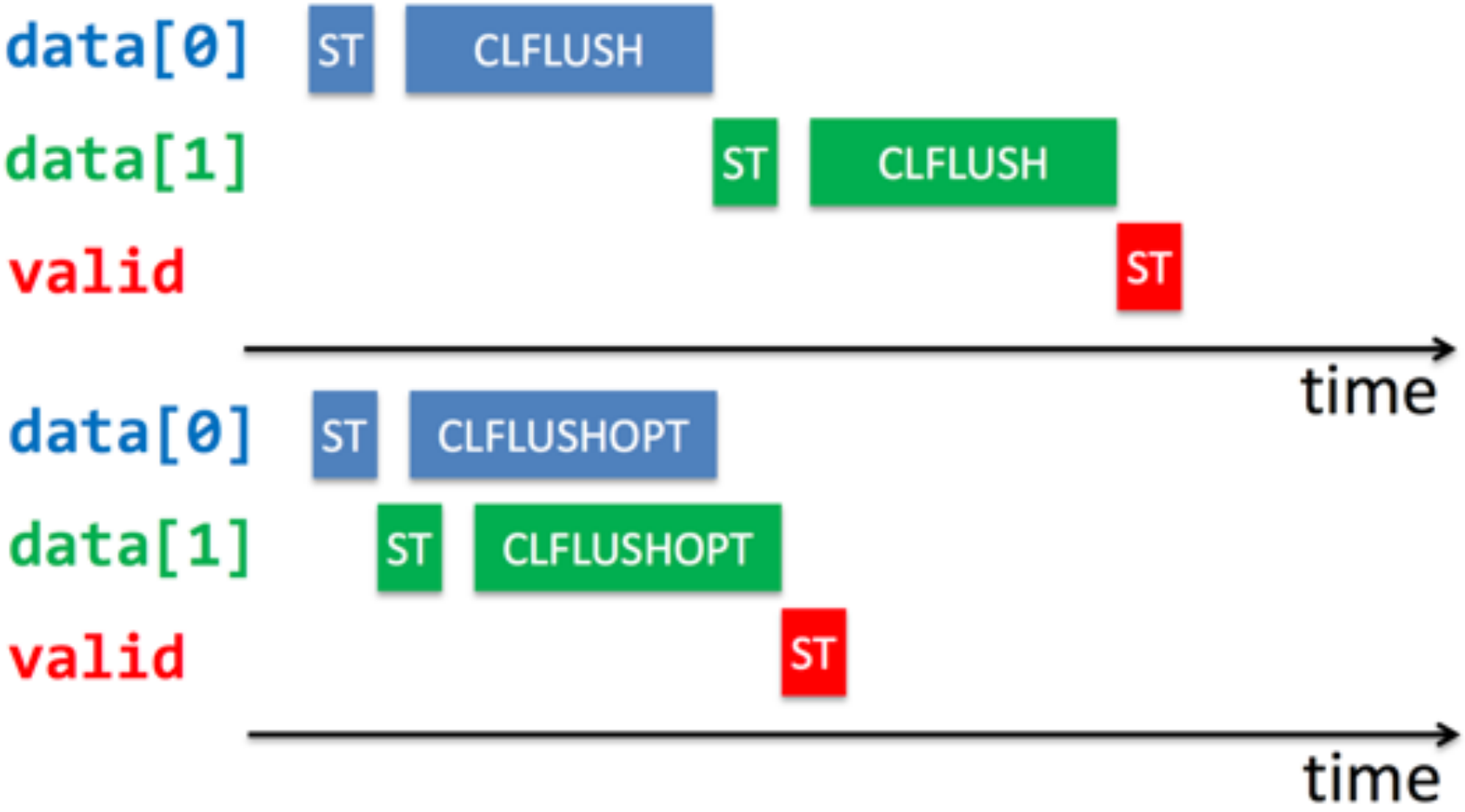
- *Now: Intel's pcommit*

File system architecture optimized for PM

- light-weight and consistent POSIX file system
- memory-mapped I/O
- protecting stray writes

Performance evaluation with PM emulator

CLFLUSHOPT


data[0]
ST
CLFLUSH
data[1]
ST
CLFLUSH
valid
ST
time
data[0]
ST
CLFLUSHOPT
data[1]
ST
CLFLUSHOPT
valid
ST
time

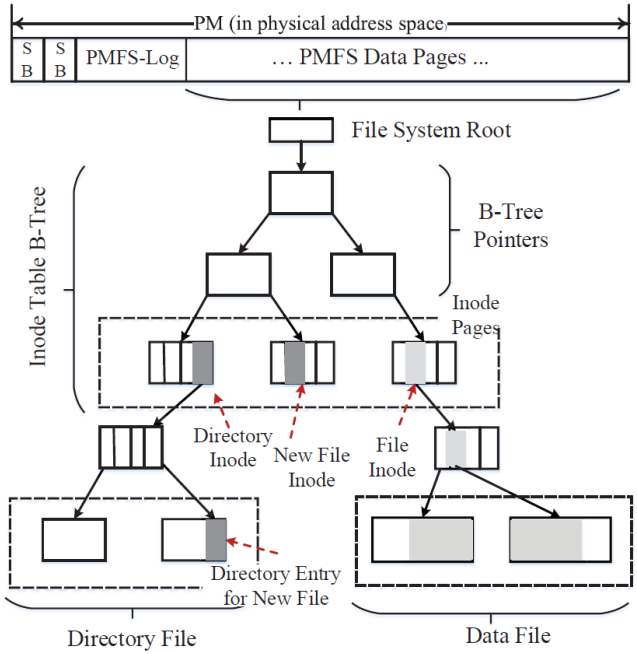
Hybrid Approach in PMFS

Metadata

- Updated by *fine-grained logging*

Data

- Use Copy on Write method

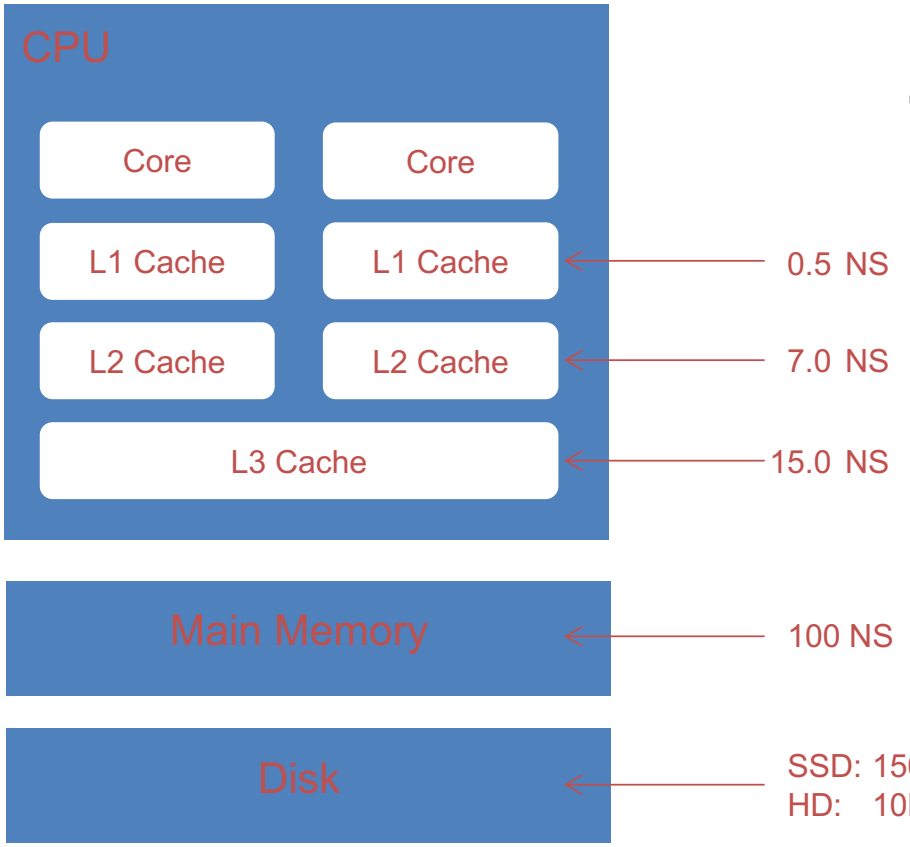


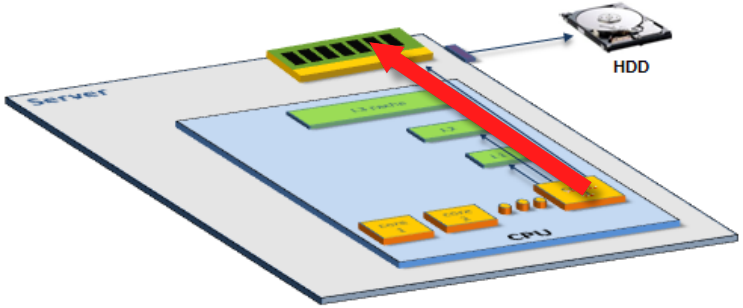
	Distributed small modification	Centralized large modification
Copy on Write	✕ (Write Amplification)	◯ (Freely after copy)
Journaling	◯ (Just append logs)	✕ (Double writes)

Summary

- Non-volatile memory to the market soon
- Changes the landscape of systems software
- PMFS
  - PM_barrier interface
  - Well-considered consistency protocol
  - Real evaluation with PM emulator

Review: Latency Numbers





Yes, DRAM is 100,000 times faster than disk, but DRAM access is still 6-200 times slower than on-chip caches

Source: SAP

RAMCloud Overview

Storage for datacenters

1000-10000 commodity servers

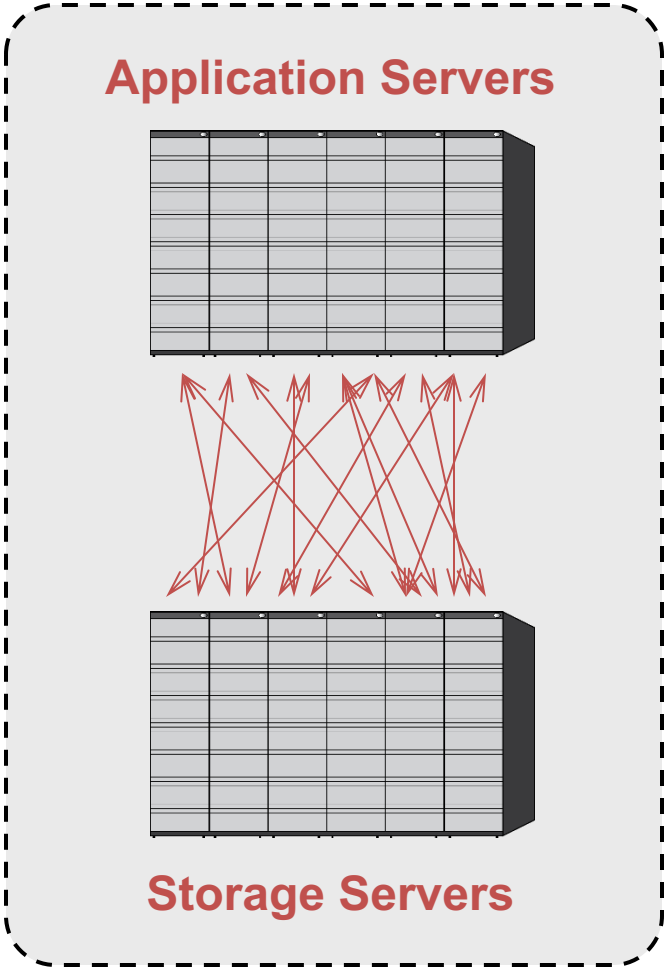
64 GB DRAM/server

All data always in RAM

Durable and available

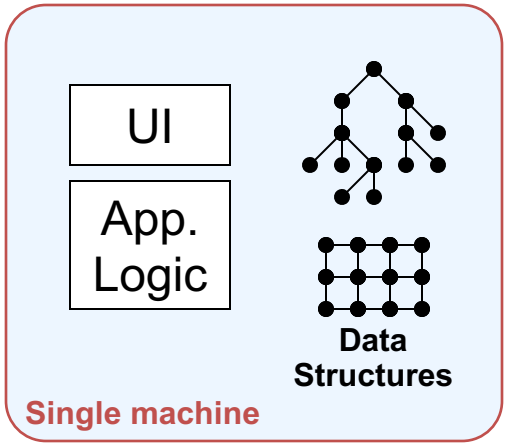
Performance goals:

- High throughput: 1M ops/sec/server
- Low-latency access: 5-10µs RPC

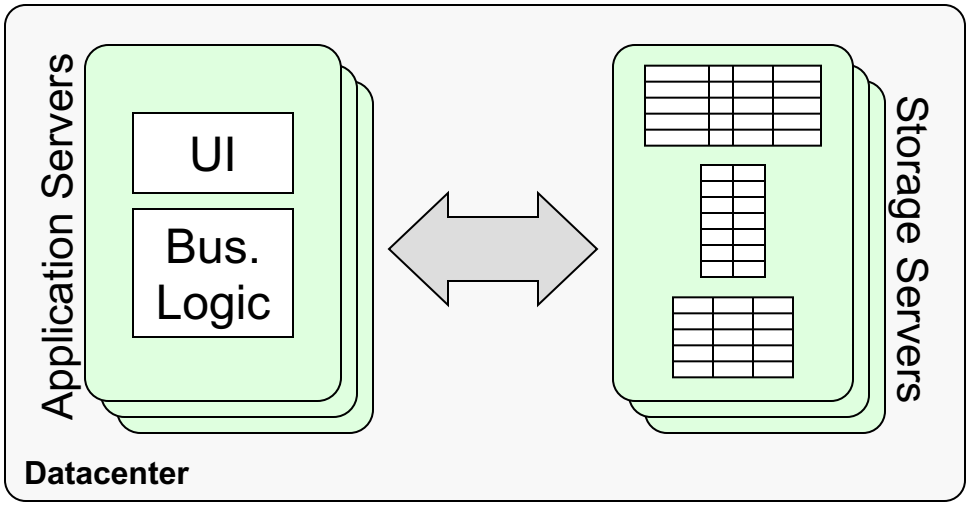


Motivation: Latency

Traditional Application



Web Application



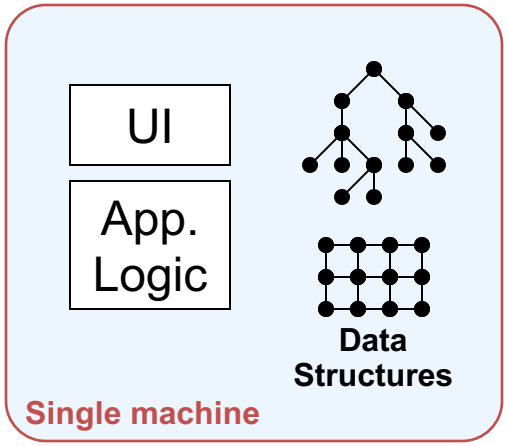
<< 1µs latency

0.5-10ms latency

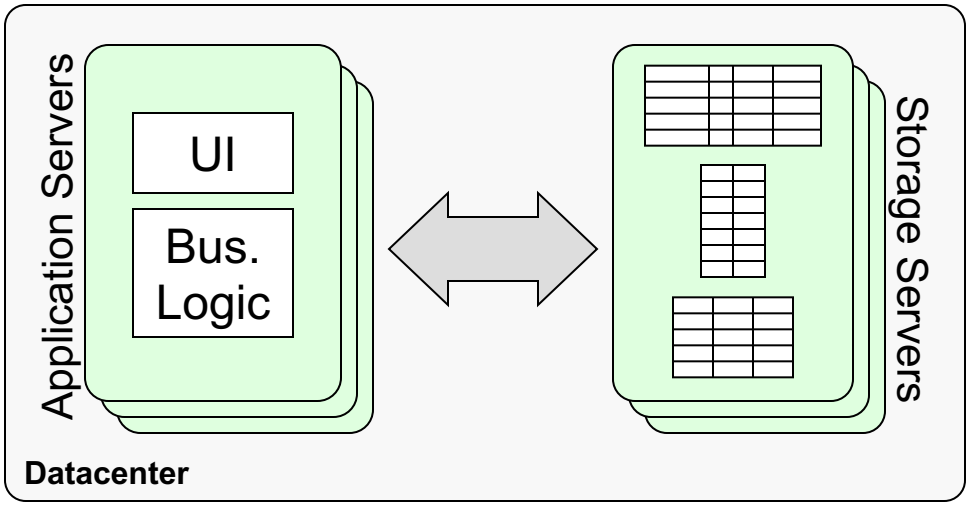
- Large-scale apps struggle with high latency
  - Facebook: can only make 100-150 internal requests per page

Motivation: Latency

Traditional Application



Web Application



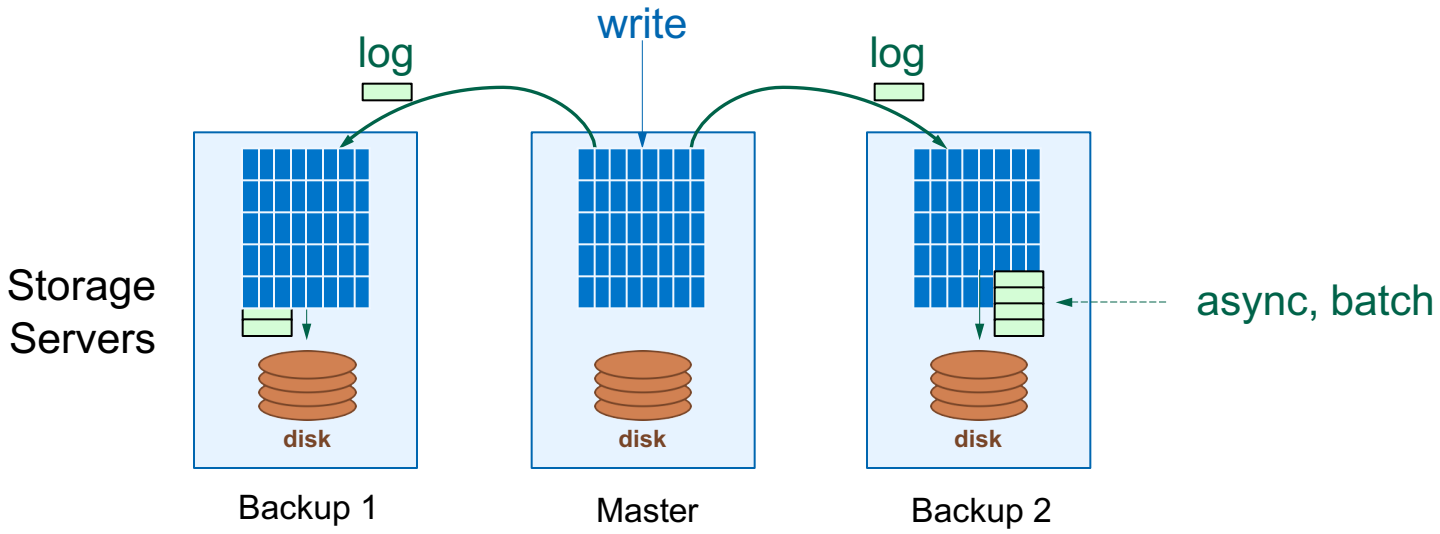
<< 1µs latency

~~0.5-10ms~~ latency
5-10µs

- RAMCloud goal: large scale and low latency
- Enable a new breed of information-intensive applications

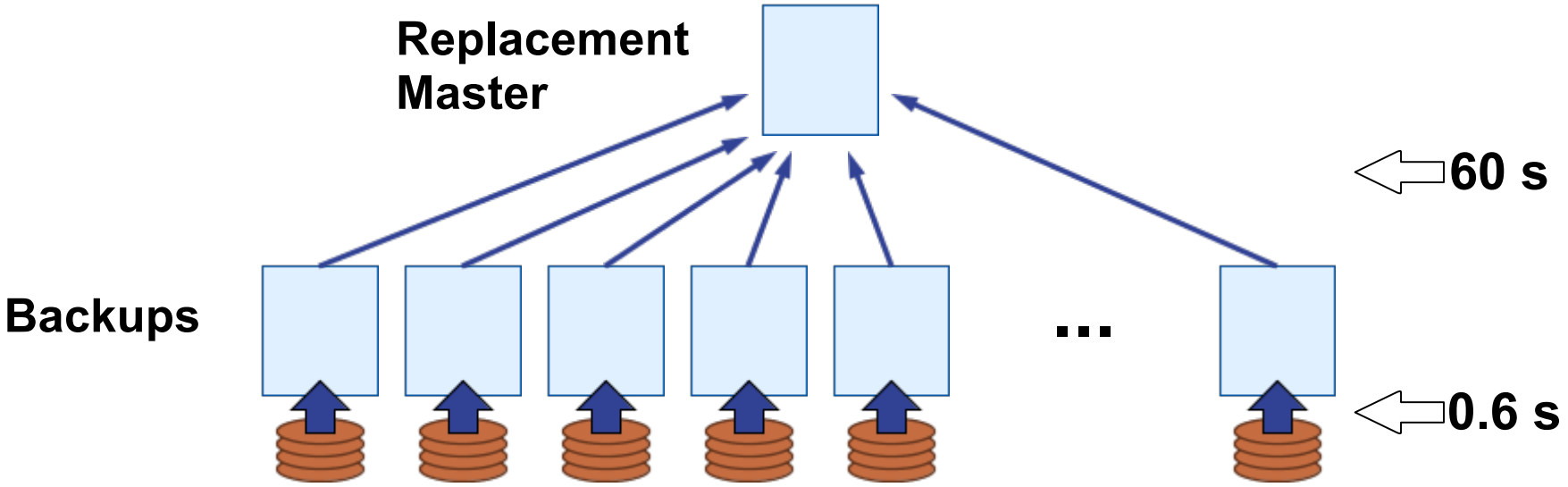
Buffered Logging

- **Each server maintains a log of updates to its objects**
  - We call the owner server a "*master*"
- **Masters' log updates are sent to *R* backups**
  - RPCs return when backups have updates buffered in RAM
  - Backups batch and write to disk asynchronously
  - Assume for now each master always uses same *R* backups
- **Each master is also a backup for other servers**



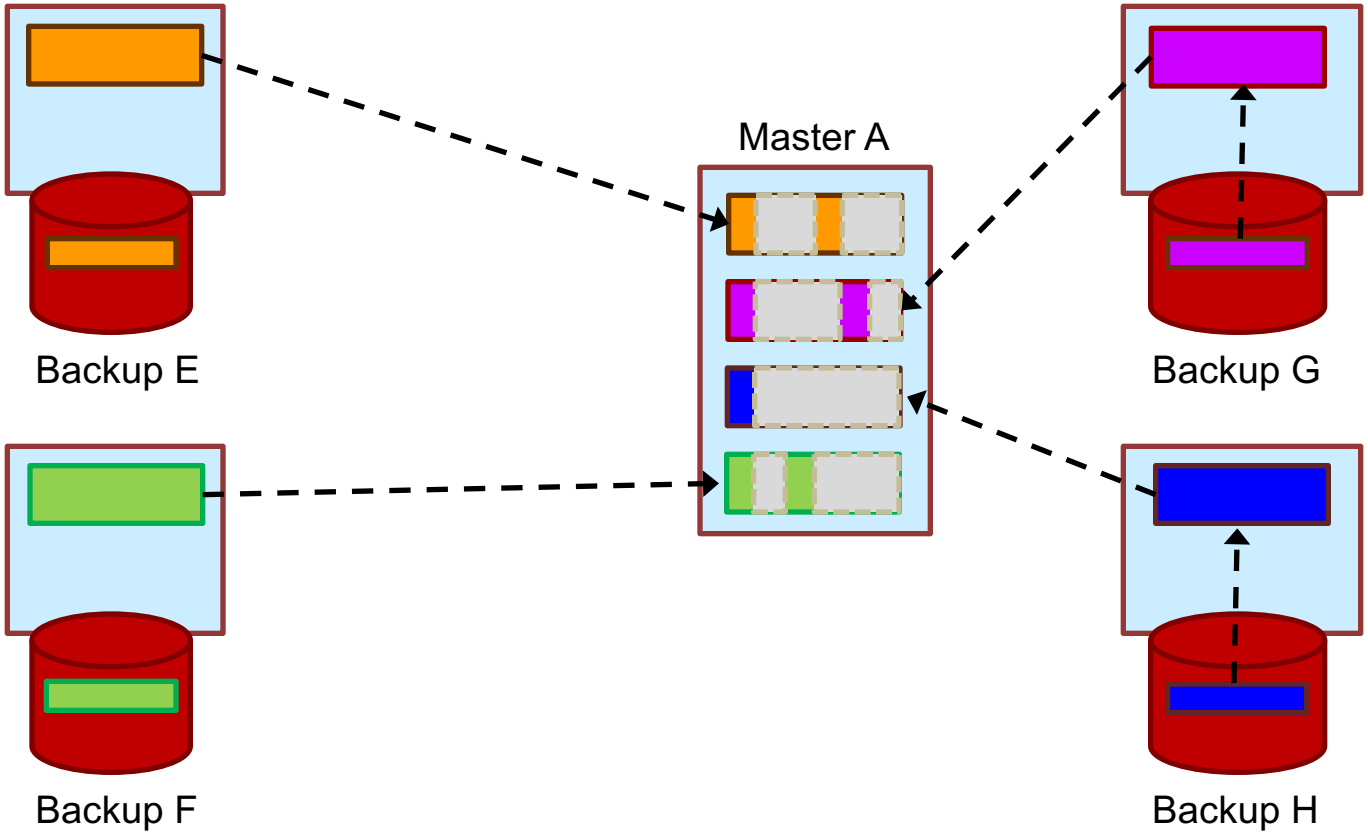
Fast Recovery: The Problem

- **After crash, all backups read disks in parallel**
  (64 GB/1000 backups @ 100 MB/sec = **0.6 sec, great!**)
- **Collect all backup data on replacement master**
  (64 GB/10Gbit/sec ~ **60 sec: too slow!**)
  Problem: Network is now the bottleneck!



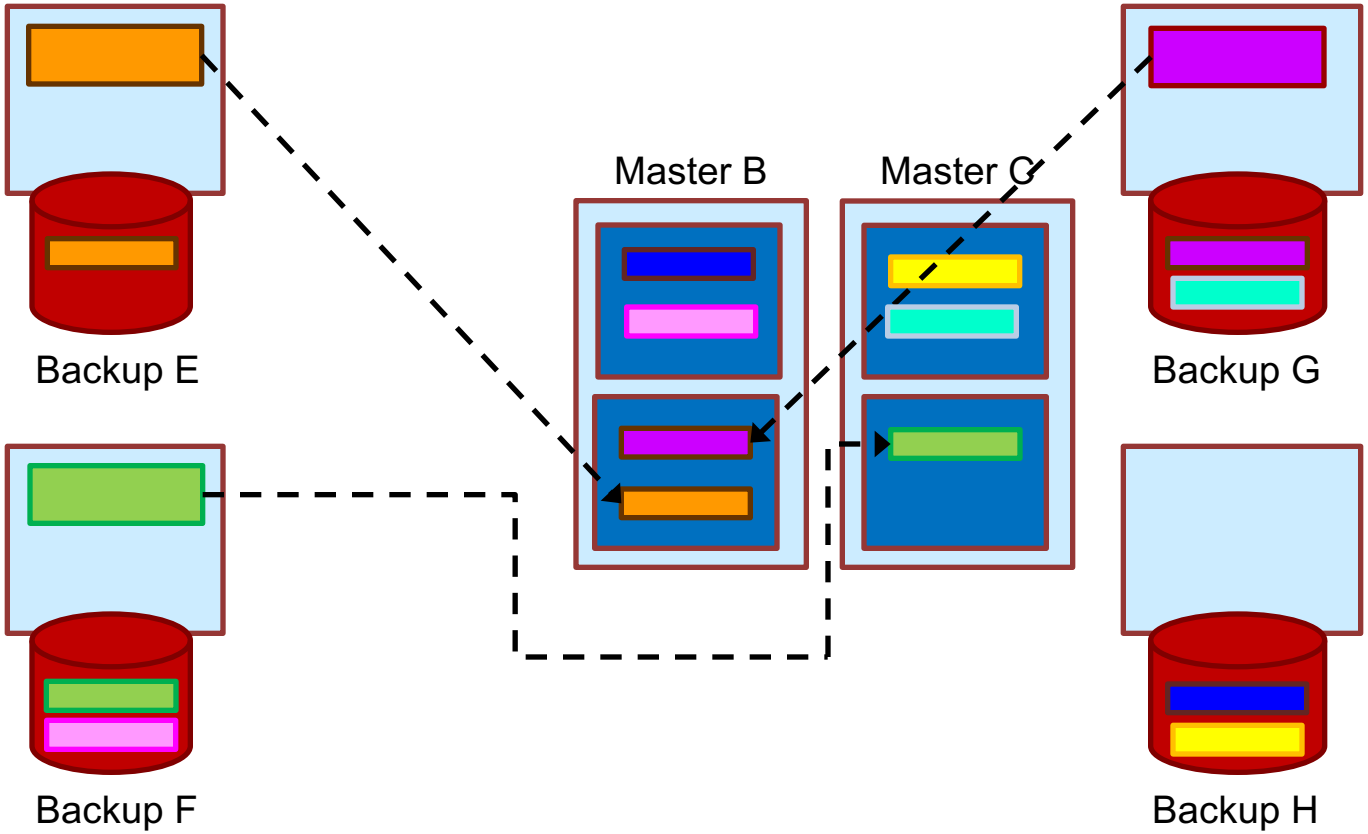
2-Phase Recovery

- **Phase #1: Recover location info (< 1s)**
  - Read all data into memories of backups
  - Send only location info to replacement master



Partitioned Recovery

- Reconstitute partitions on many hosts
- 64 GB / 100 partitions = 640 MB
- 640 MB / 10 GBit/s = 0.6s for full recovery



What is scalability?

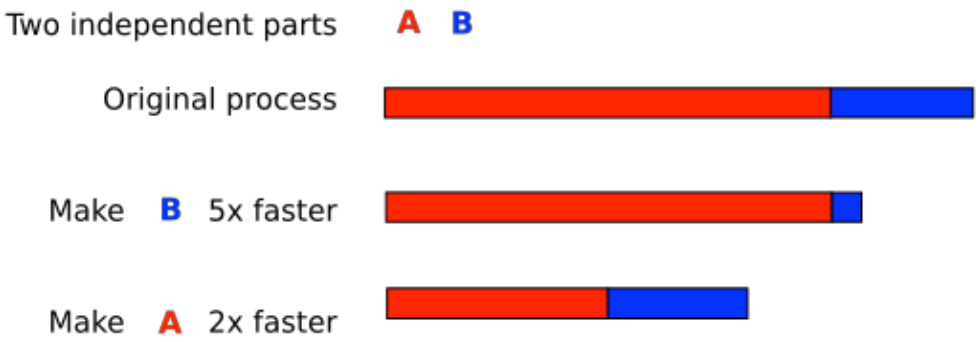
Application does N times as much work on N cores as it could on 1 core

Scalability may be limited by Amdahl's Law:

- Locks, shared data structures, ... Shared hardware (DRAM, NIC, ...)

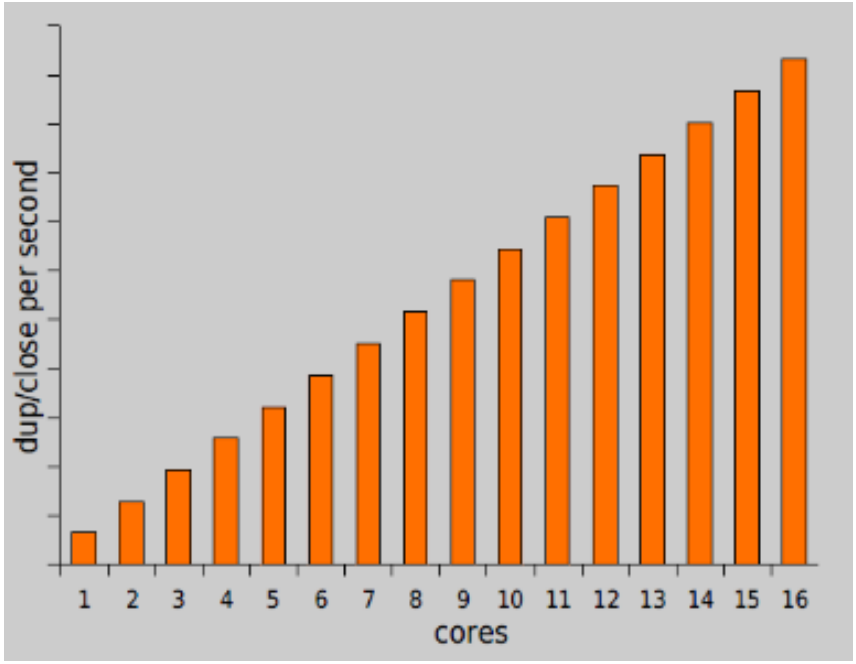
Amdahl's Law

$$\frac{1}{(1-P) + \frac{P}{S}}$$

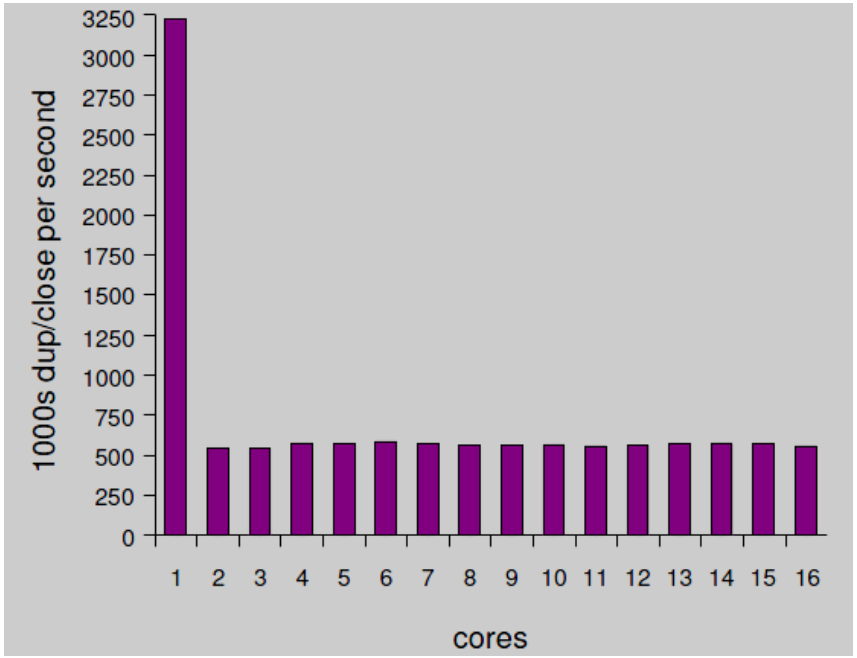


Motivating example: file descriptors

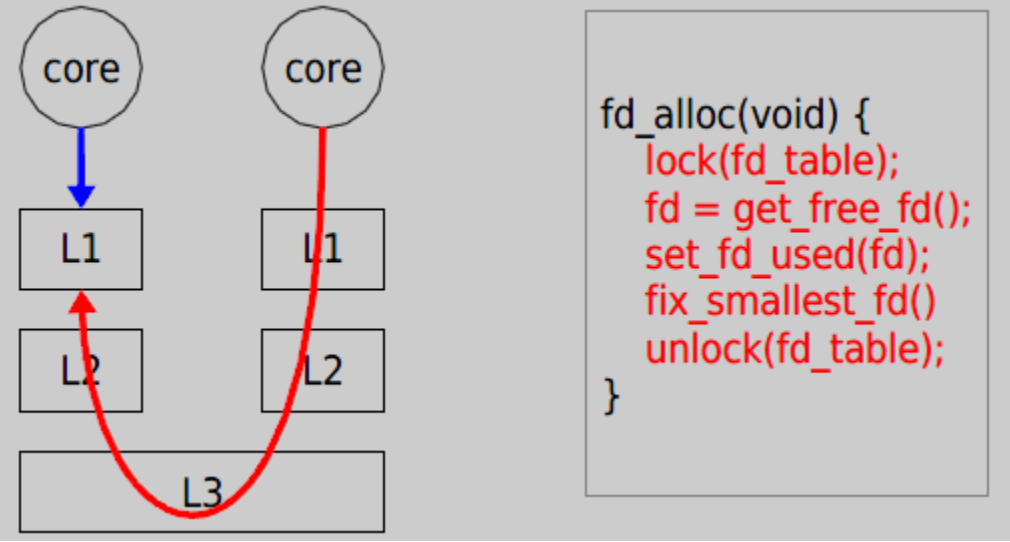
Ideal FD performance graph



Actual FD performance



Recap: Why throughput drops?



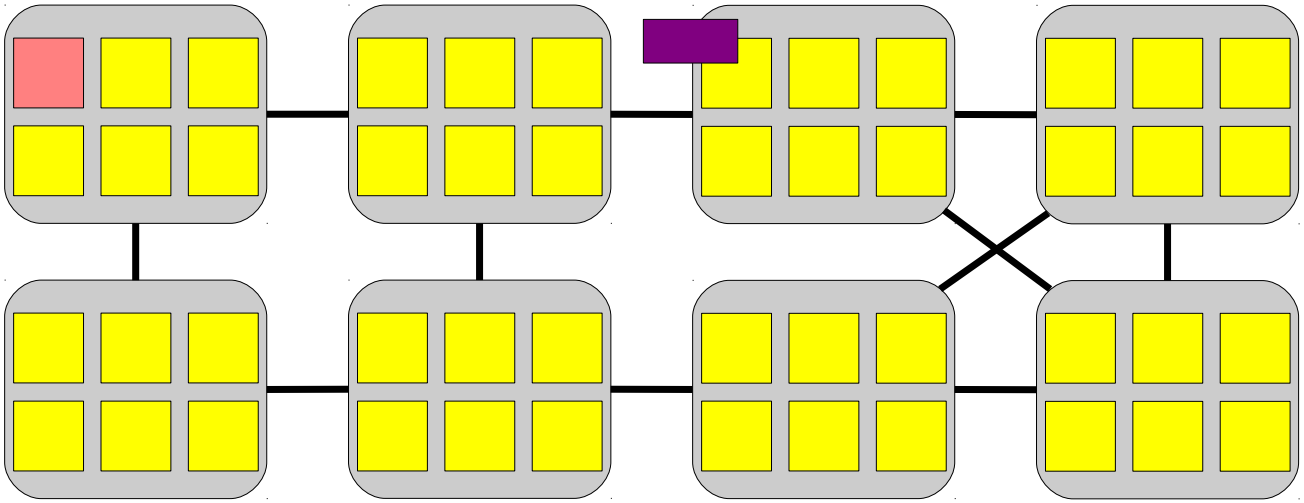
Now it takes 121 cycles!

Allocate a ticket
read current ticket and spin

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



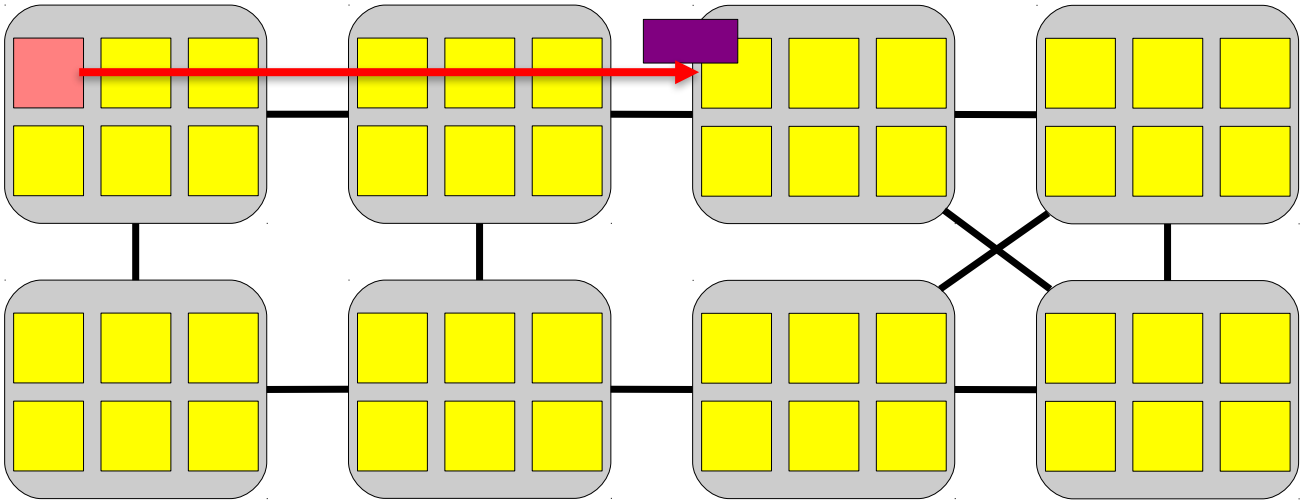
Allocate a ticket
read current ticket and spin

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

cache coherence message



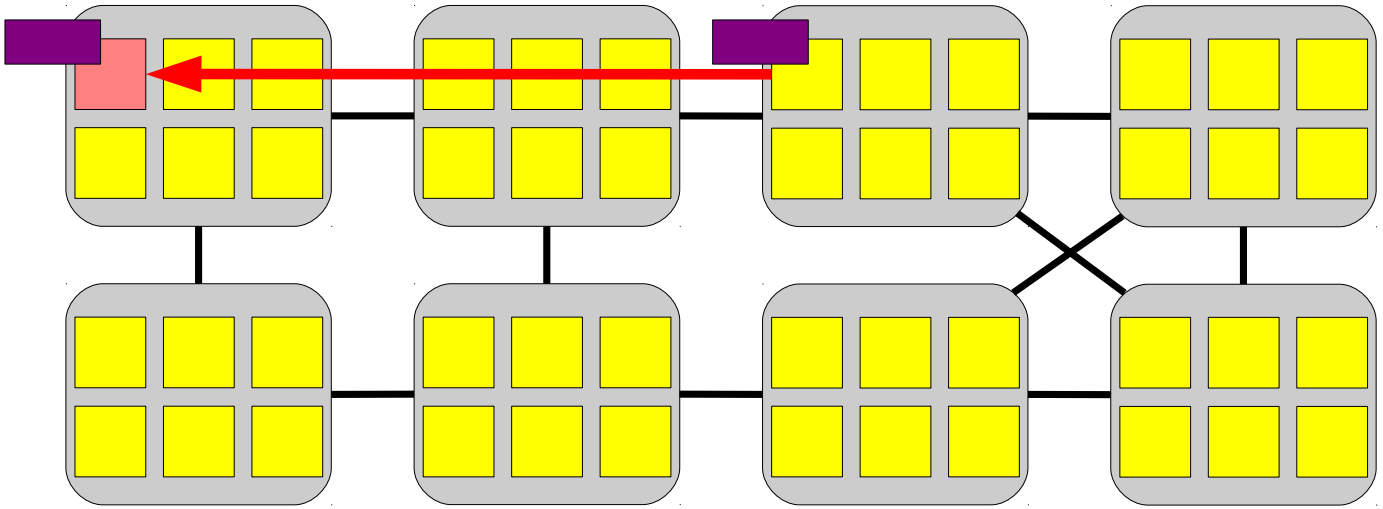
Allocate a ticket
read current ticket and spin

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

cache coherence message



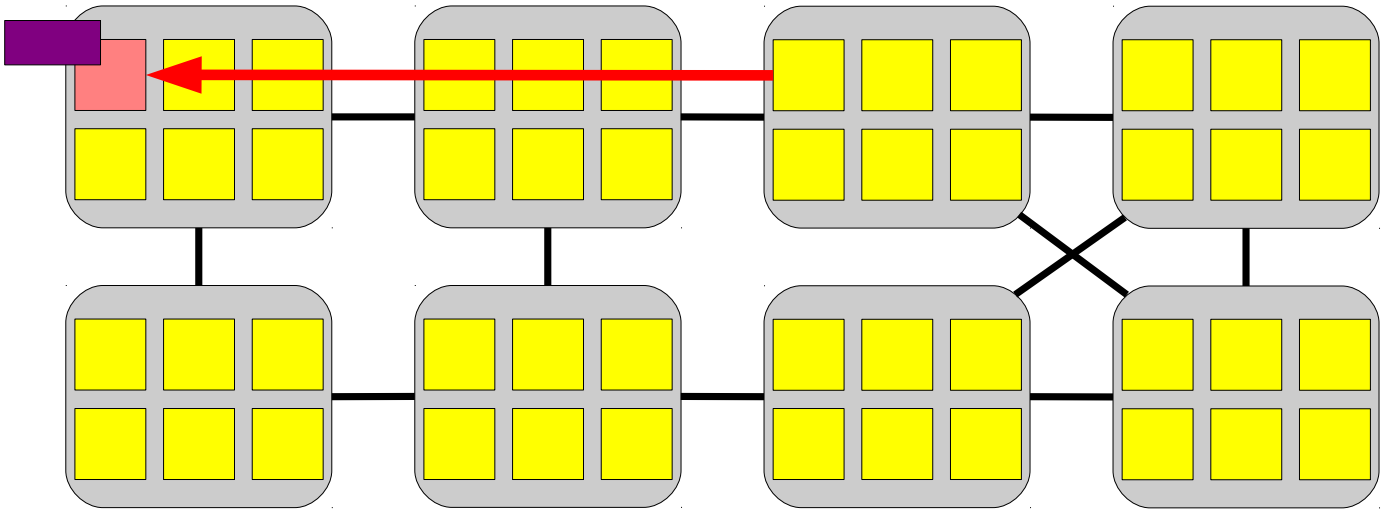
Allocate a ticket
read current ticket and spin

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

120 ~ 420 cycles

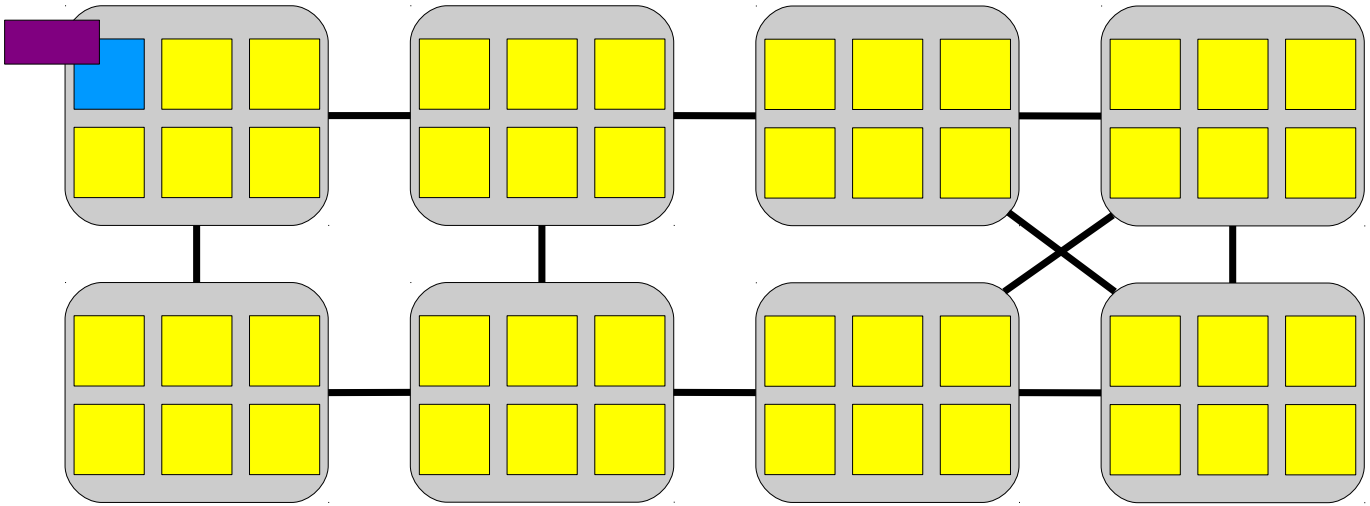


update the ticket value

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

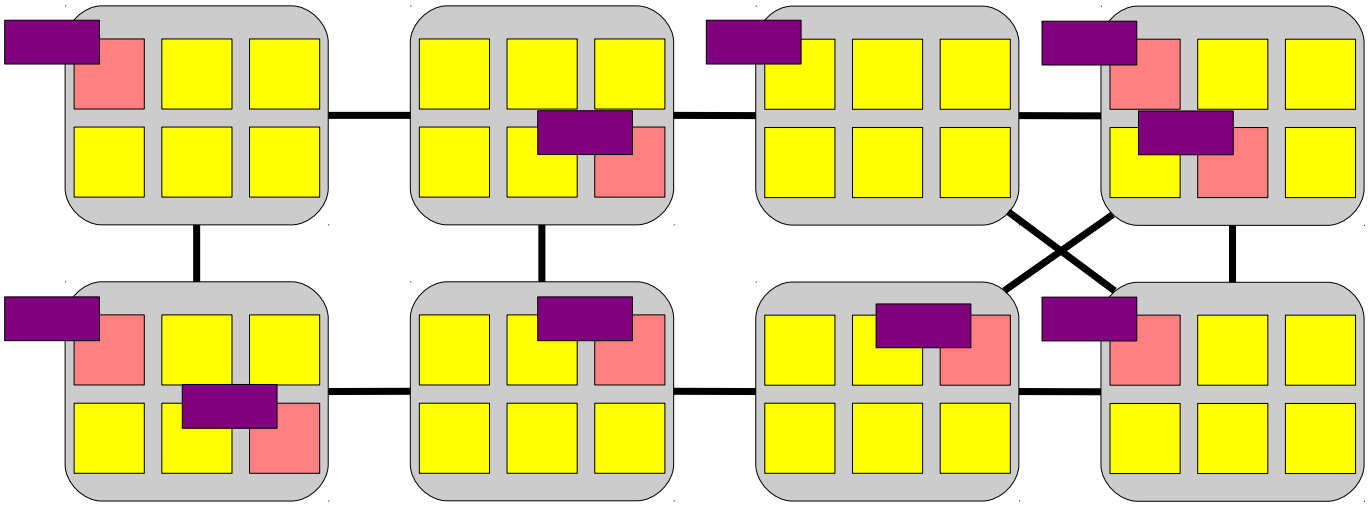
```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

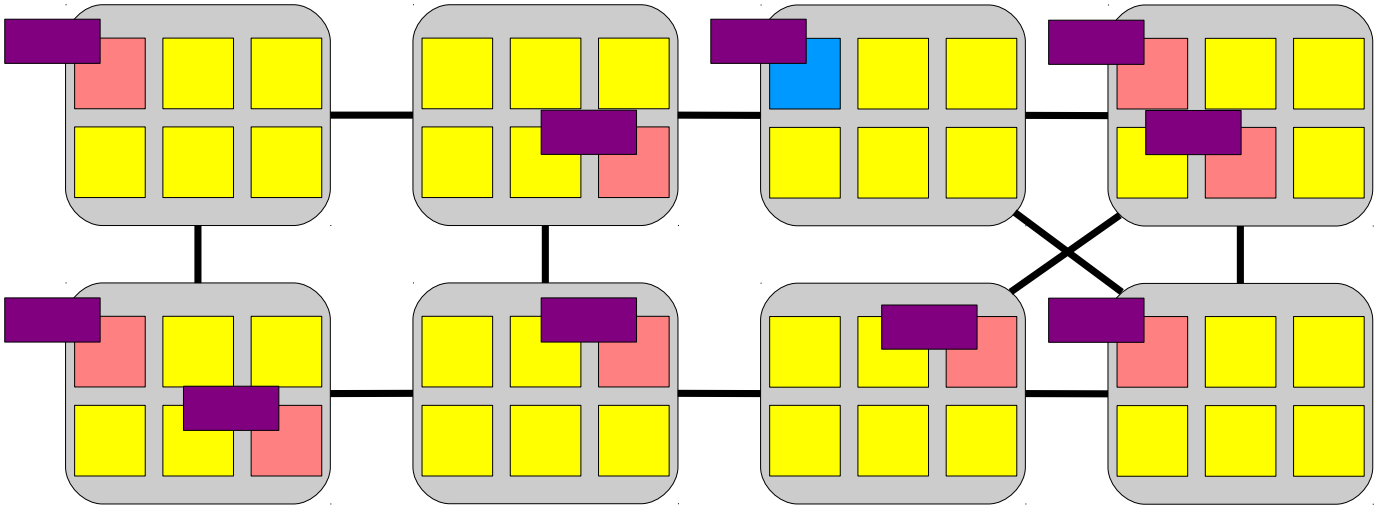


Lock shared by all core

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



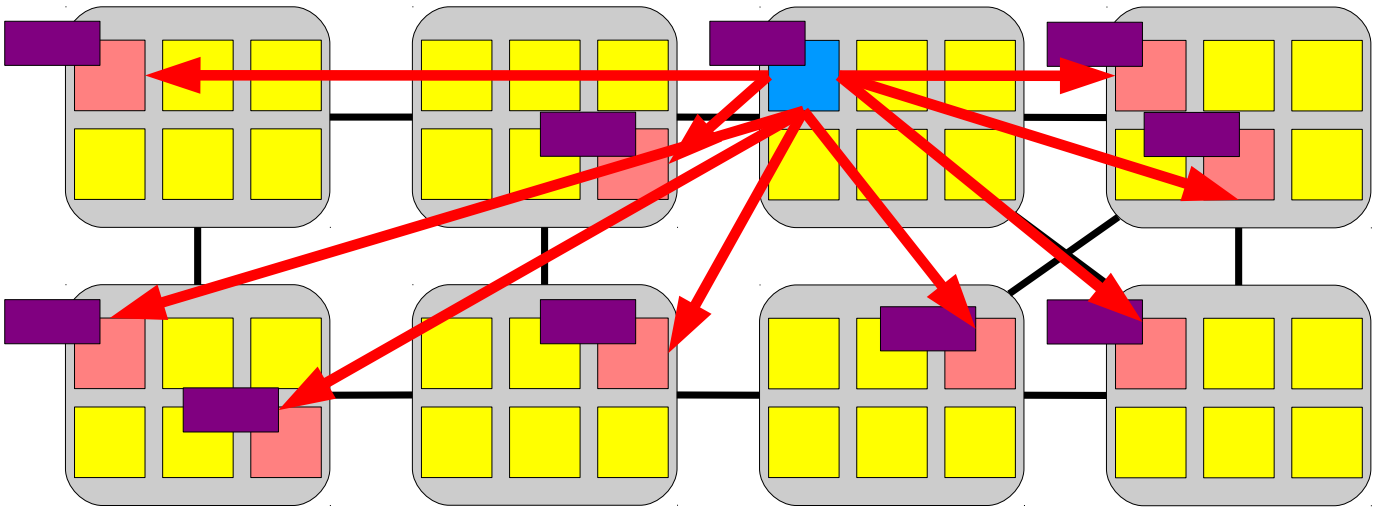
Lock holder update ticket

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

Invalidate message



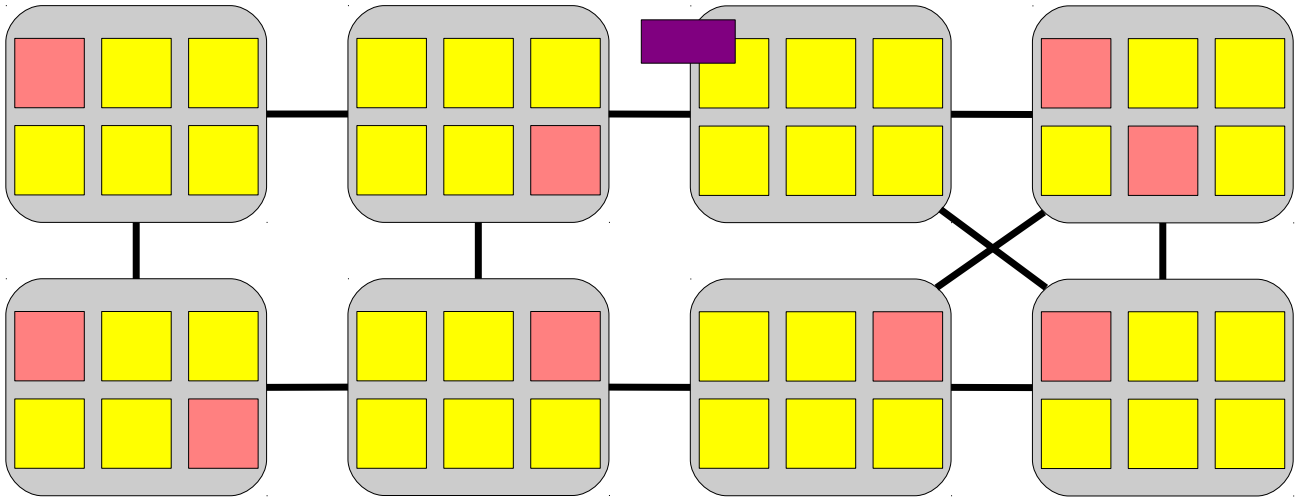
Lock holder update ticket

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

Only lock holder has lock in cache

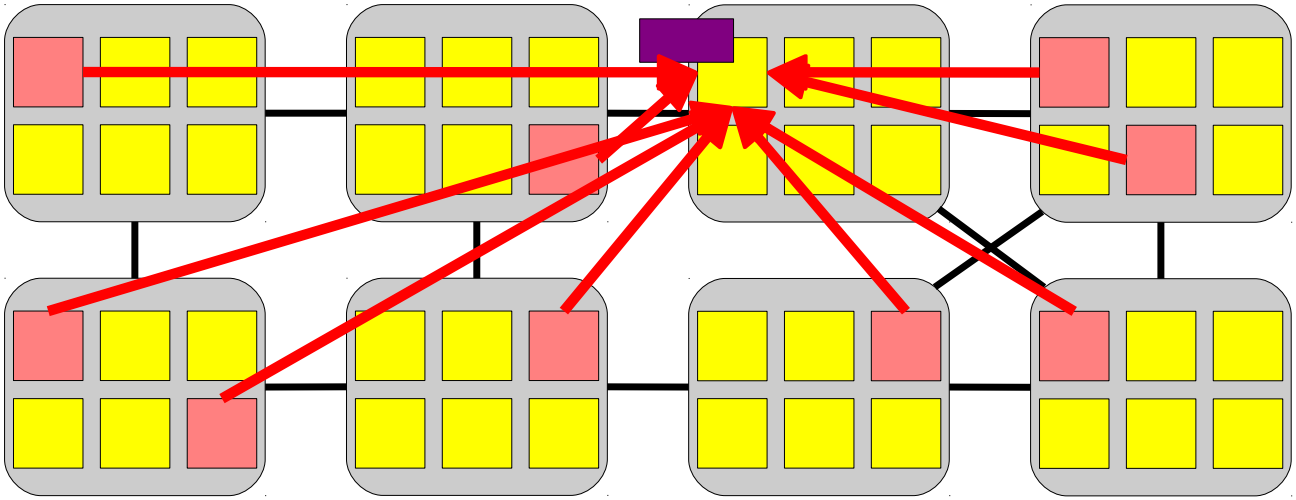


Lock holder update ticket

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



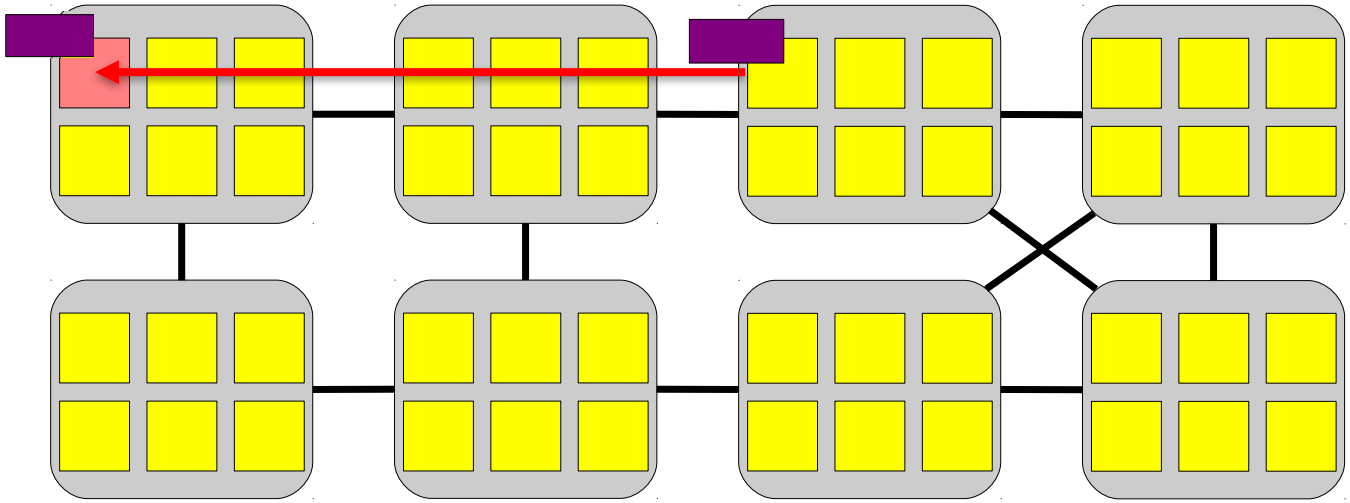
All waiters read the lock

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

500 ~ 4000 cycles!



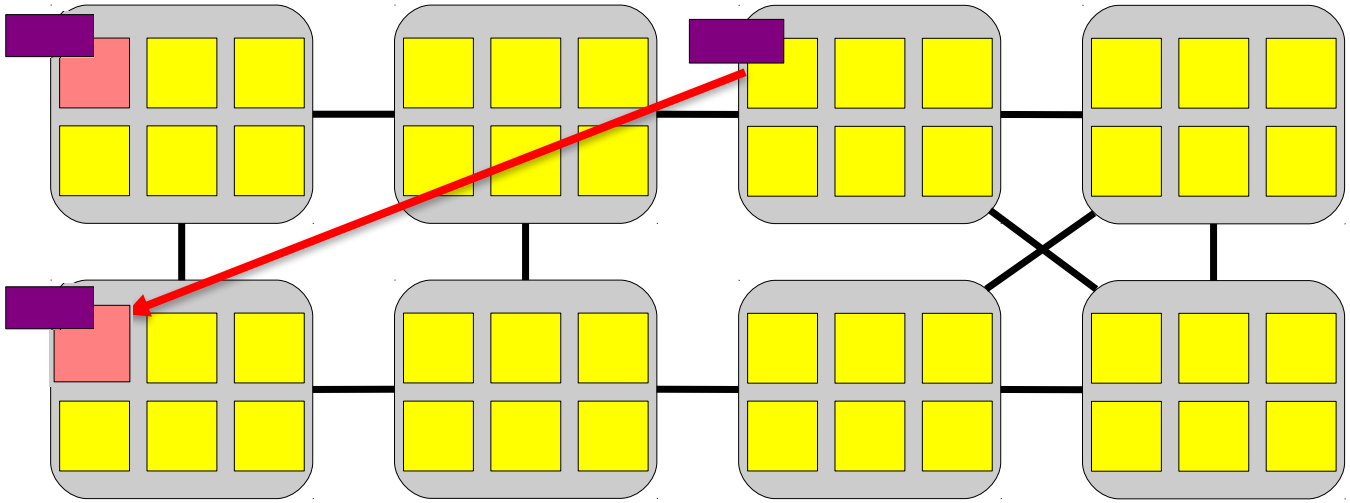
All waiters read the lock

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

Reply read request one by one



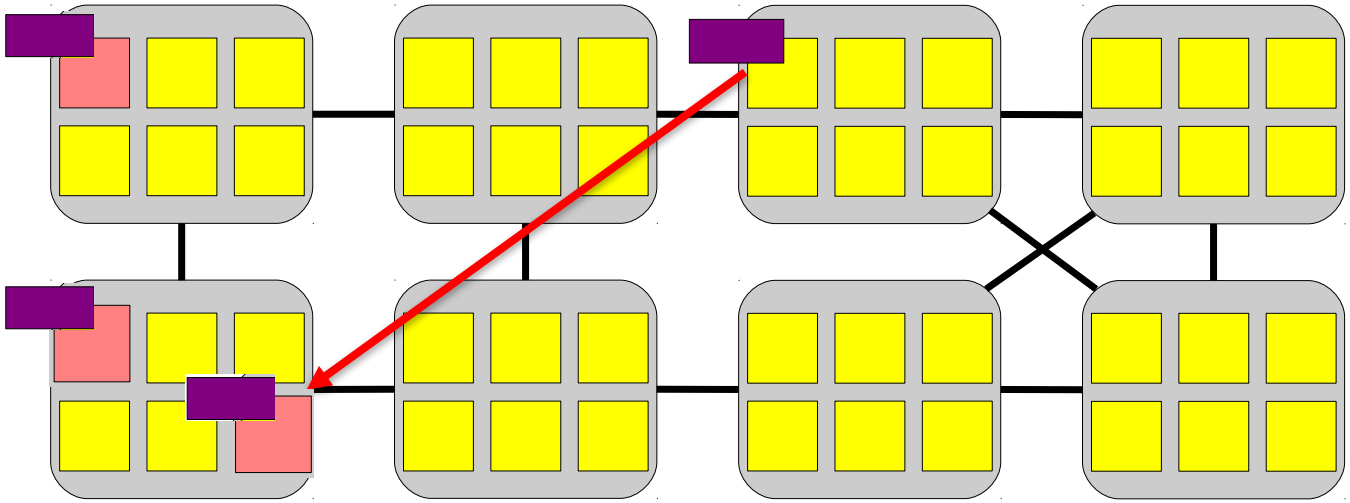
All waiters read the lock

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

Reply read request one by one



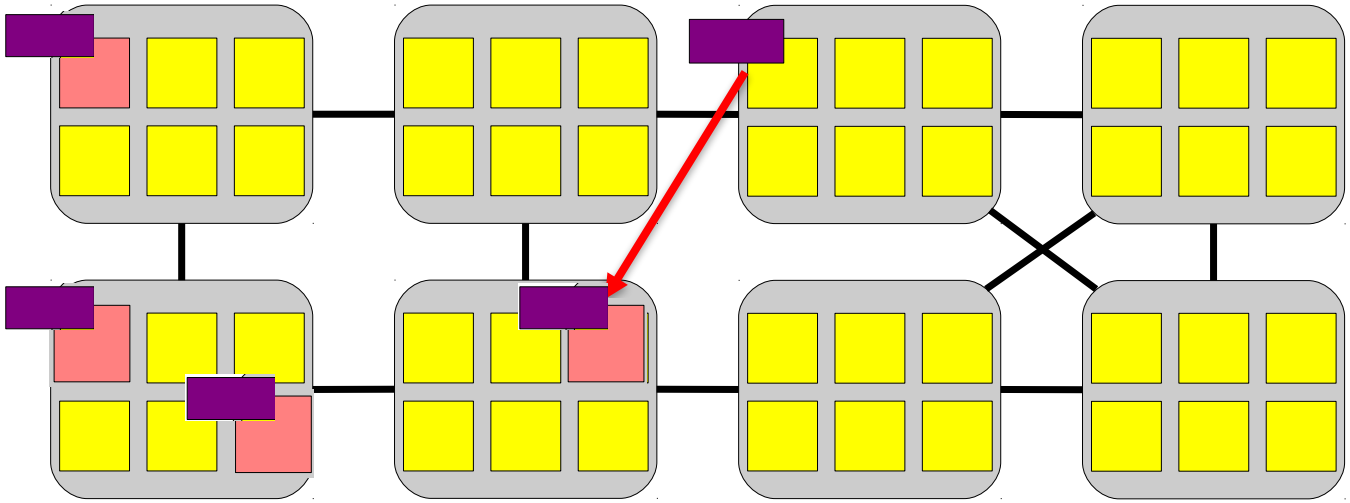
All waiters read the lock

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

Previous lock holder notifies next lock holder after sending out N/2 replies



All waiters read the lock

Using scalable locks

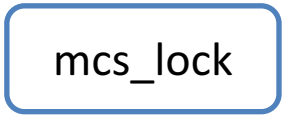
Many existing scalable locks

Main idea is to avoid contending on a single cache line

Example

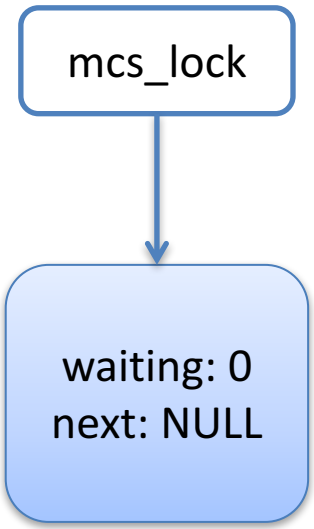
- MCS (John M. Mellor-Crummey and Michael L. Scott)
- K42

General idea of MCS lock



NULL

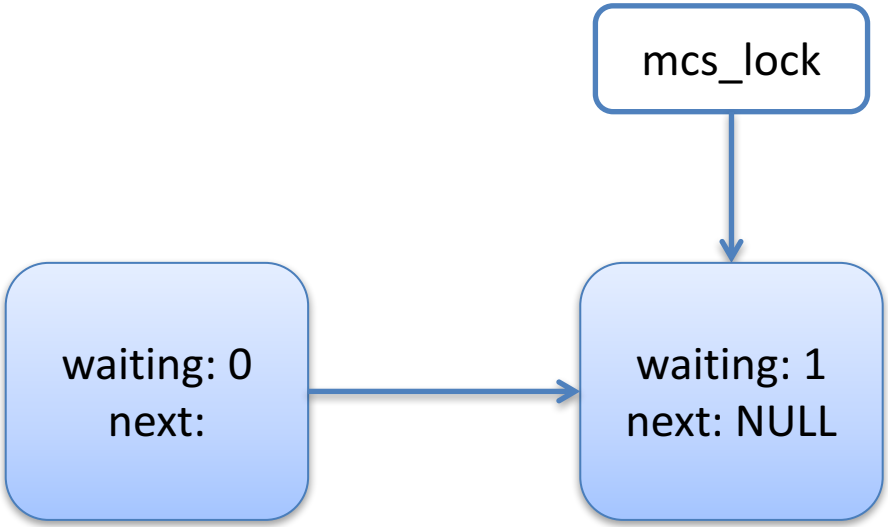
General idea of MCS lock



Use compare and swap to change mcs_lock point to self node

Check previous node
NULL in this case, no need to wait

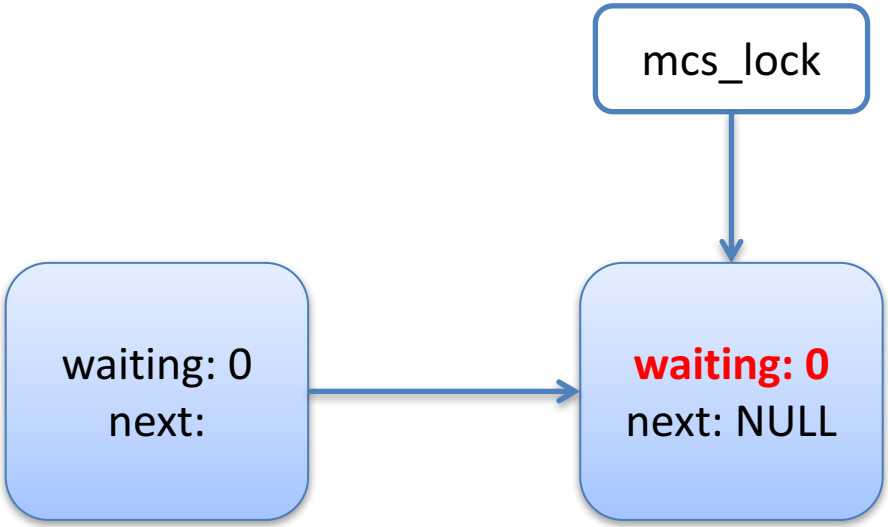
General idea of MCS lock



Previous node is not NULL
Current waiting is 1

Wait until lock holder
set waiting to 0

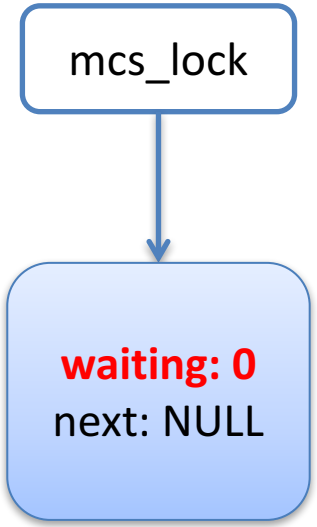
General idea of MCS lock



Previous node is not NULL
Current waiting is 1

Wait until lock holder
set waiting to 0

General idea of MCS lock



Previous node is not NULL
Current waiting is 1

Wait until lock holder
set waiting to 0

Conclusion

Non-scalable locks are dangerous

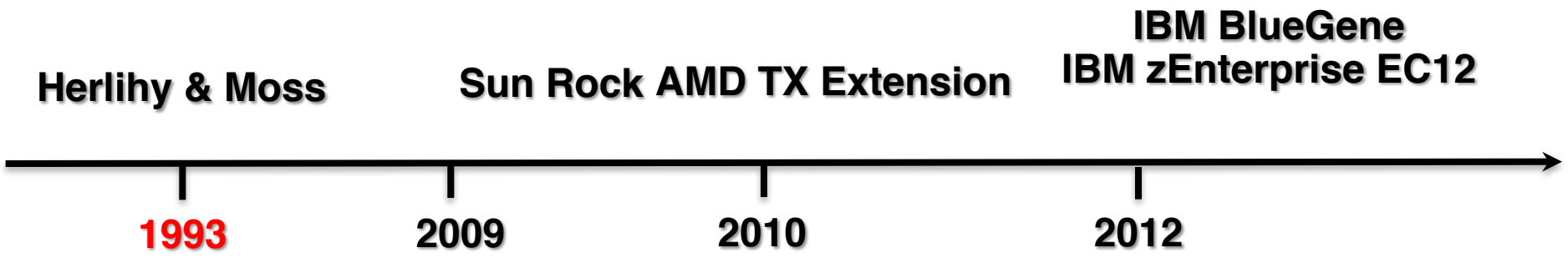
- Short critical section may lead to performance collapse
- Caused by contention on lock cache line

Scalable locks is a way to relax the time-criticality of applying more fundamental scaling improvements to the kernel

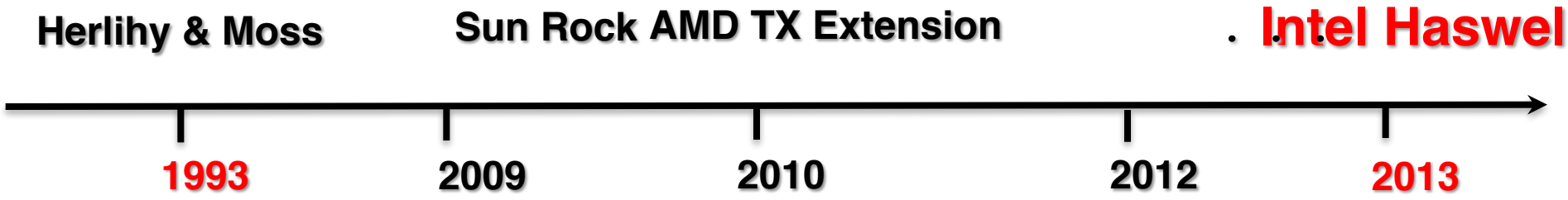
Basic Idea of Transactional Memory

- Leverage existing ideas
- HW - LL/SC
  - Serves as atomic RMW
  - MIPS II and Digital Alpha
    - Restricted to single-word
- SW - Database
    - Transactions
      - How about expand LL&SC to multiple words
    - Apply ATOMICITY
      - COMMIT or ABORT

Hardware Transactional Memory



Hardware Transactional Memory



Massively available

Intel Haswell

Restricted Transactional Memory (RTM)

- Hardware transactional memory with limitations

Major limitations

- Working set is limited
- Some system events abort the TX

New instruction set

- Xbegin, Xend, Xabort

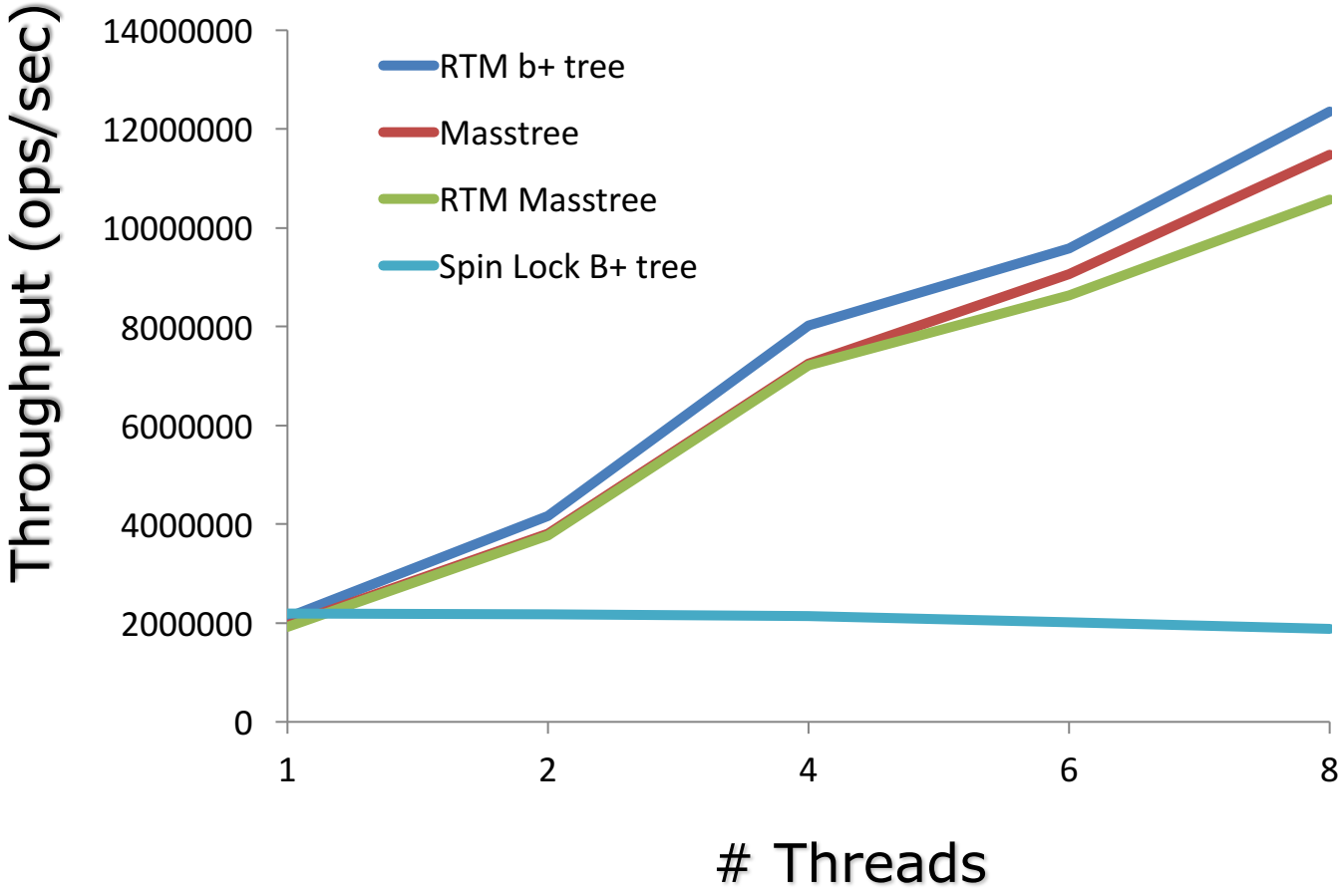
Programming With RTM

RTM Usage

```
if _xbegin() == _XBEGIN_STARTED
        do some critical work
    _xend()
else
    fallback routine
```

Handle the abort event

RTM B+ Tree vs. Masstree – YCSB



RTM B+ tree outperforms masstree due to its simple

NoSQL: Not only SQL

SQL

- Internet-Scale data ✗
- High Concurrency ✗
- Highly Scalable ✗

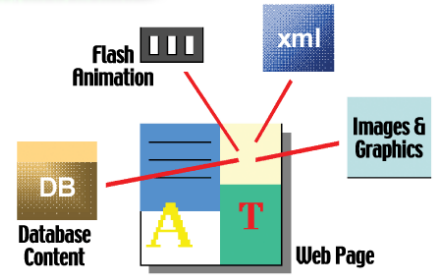
Optimization to RDBMS

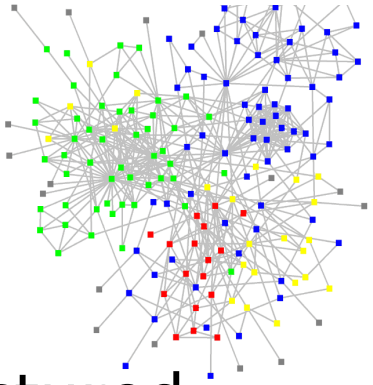
- Master-Slave
- Partition/Sharding
- Multi-Master Replication
- INSERT only, not UPDATE/DELETE
- No JOINs
- In-Memory Database

Not only SQL

Not only SQL

- High Volume
- Semi/UN-Structured
- High Concurrency
- High Scalability

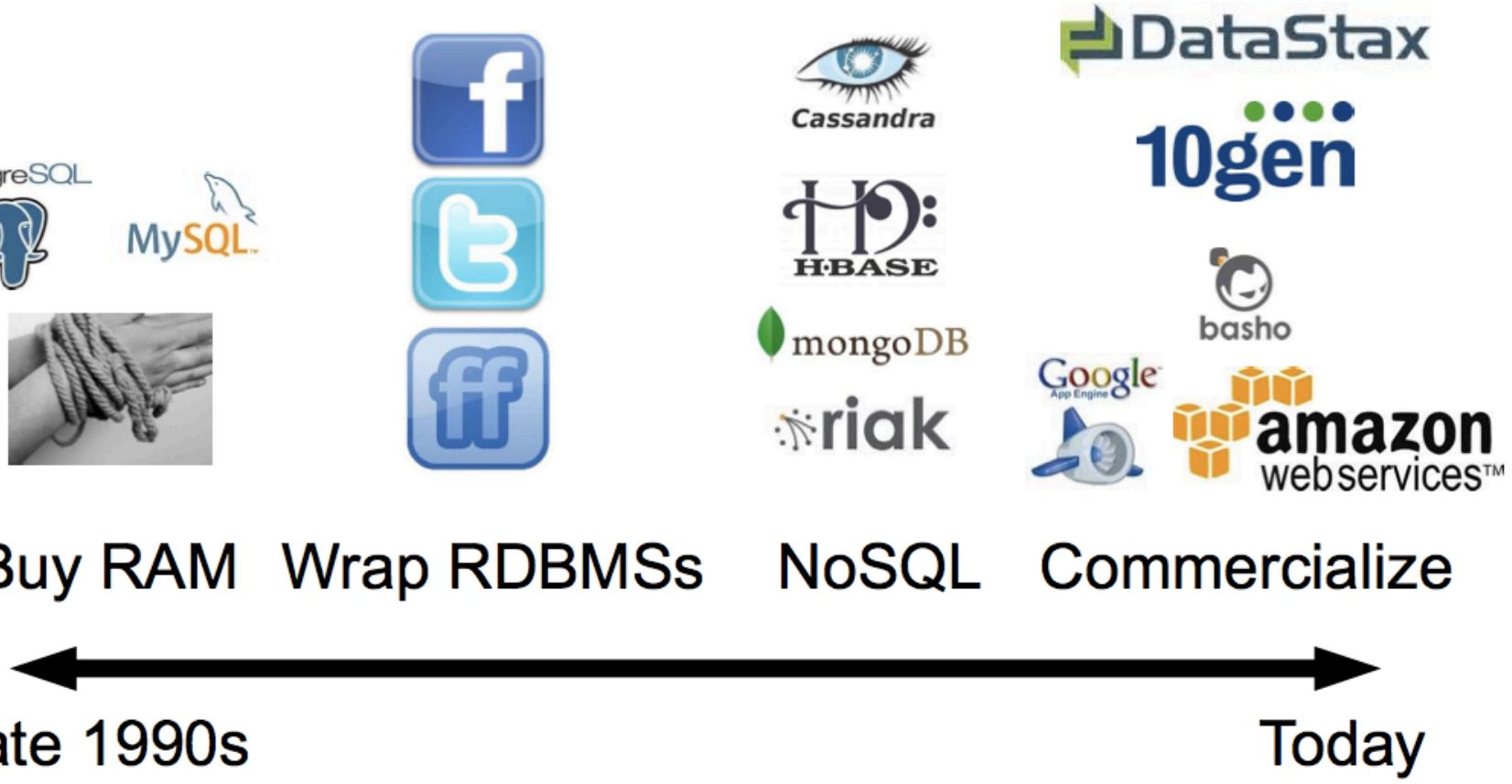




Over 90% data in the world in Unstructured
Such : Pictures、Video、Audio ……

(From IDC)

(Compressed) Histories of NoSQL



Standards-compliant SQL Systems

Relational model

Powerful query language

Transactional semantics

Predefined schemas

Strong consistency between replicas

NoSQL: Maybe you don't need all of these?

NoSQL Systems are a Buffet (of progressively larger grenades)

Data model

Query model

Durability

Transactional consistency

Partitioning

Replica consistency

NoSQL

"NoSQL DBMSs cause their developers to spend too much time writing code to handle inconsistent data and that transactions provide a useful abstraction that is easier for humans to reason about."

Google

New requirements

- Large scale systems, with huge and growing data sets
- Information is frequently generated by
- devices High concurrency requirements
- Usually, data model with some relations
- Often, transactional integrity

NewSQL: definition

"A DBMS that provide the same scalable performance of NoSQL for online transaction processing (OLTP) read- write workloads while still maintaining ACID guarantees for transactions."

Andrew Pavlo & Matthew Aslett

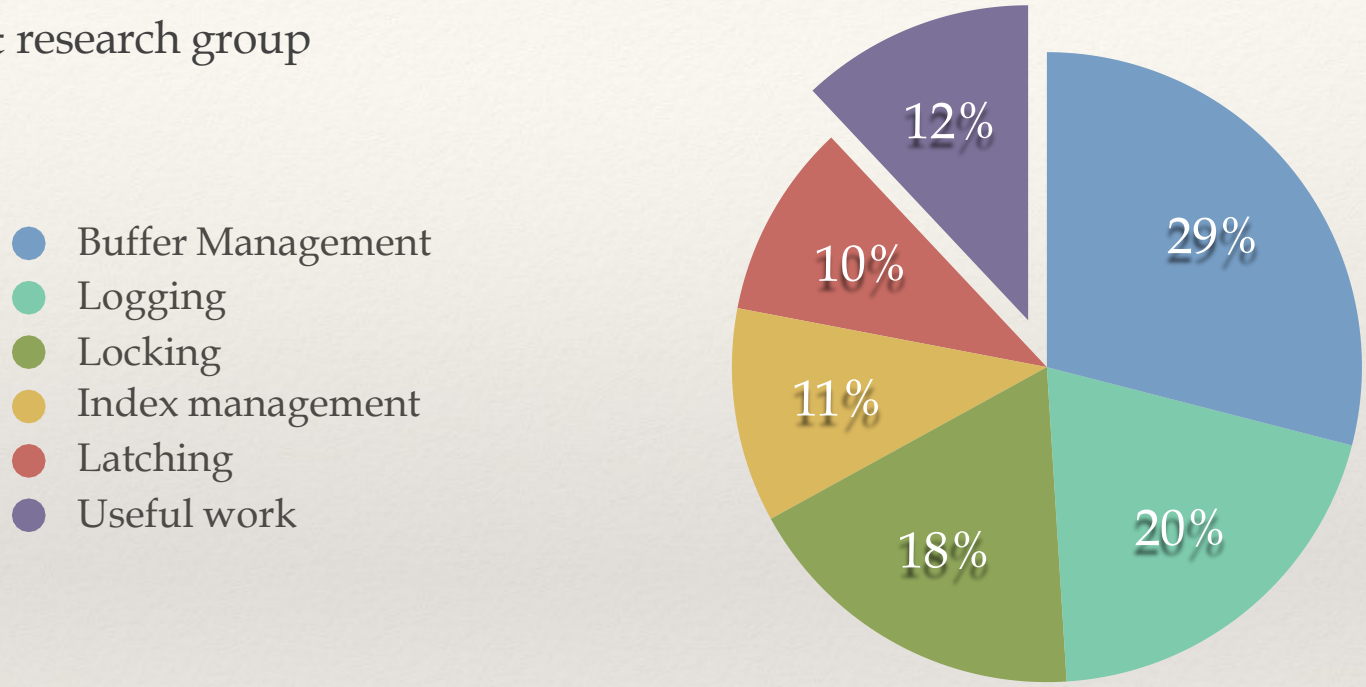
NewSQL: definition

- *SQL as the primary interface*
- *ACID support for transactions*
- *Non-locking concurrency control*
- *High per-node performance*
- *Scalable, shared nothing architecture*

Michael Stonebraker

Traditional DBMS overheads

by Stonebraker & research group



"Removing those overheads and running the database in main memory would yield orders of magnitude improvements in database performance"

Conclusions

- NewSQL is an established trend with a number of options
- Hard to pick one because they're not on a common scale
- No silver bullet
- Growing data volume requires ever more efficient ways to store and process it

High COST for Distributed TX

Many scalable systems have low performance

- Usually 10s~100s of thousands of TX/second
- High COST[1] (config. that outperform single thread)
- e.g., HStore, Calvin[SIGMOD'12]

Emerging speedy TX systems not scale-out

- Achieve over 100s of thousands TX/second
- e.g., Silo[SOSP'13], DBX[EuroSys'14]

Dilemma:

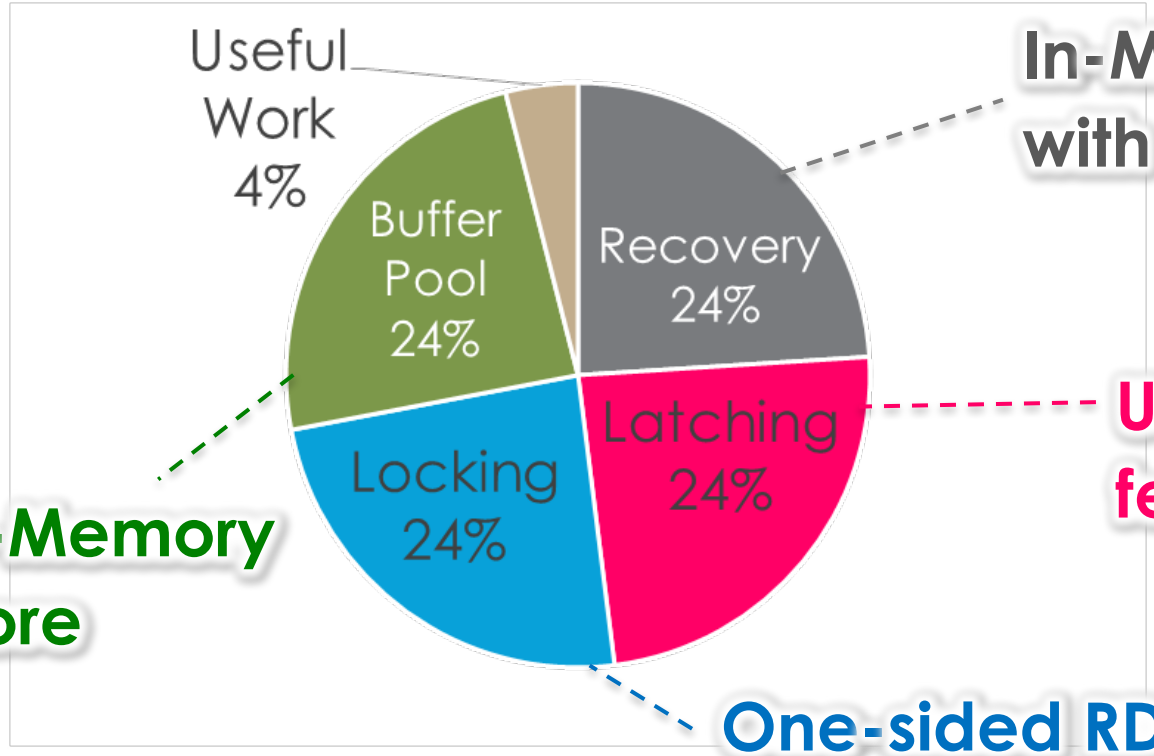
single-node perf. *vs.* **scale-out**

[1] Salability! But at what Cost? HotOS 2015

DrTM: Fast In-memory Transaction Processing using RDMA and HTM

Use HTM's ACI properties for local TX execution

Use one-sided RDMA to glue multiple HTM TXs



In-Memory Logging with NVM

Use HTM's ACI features

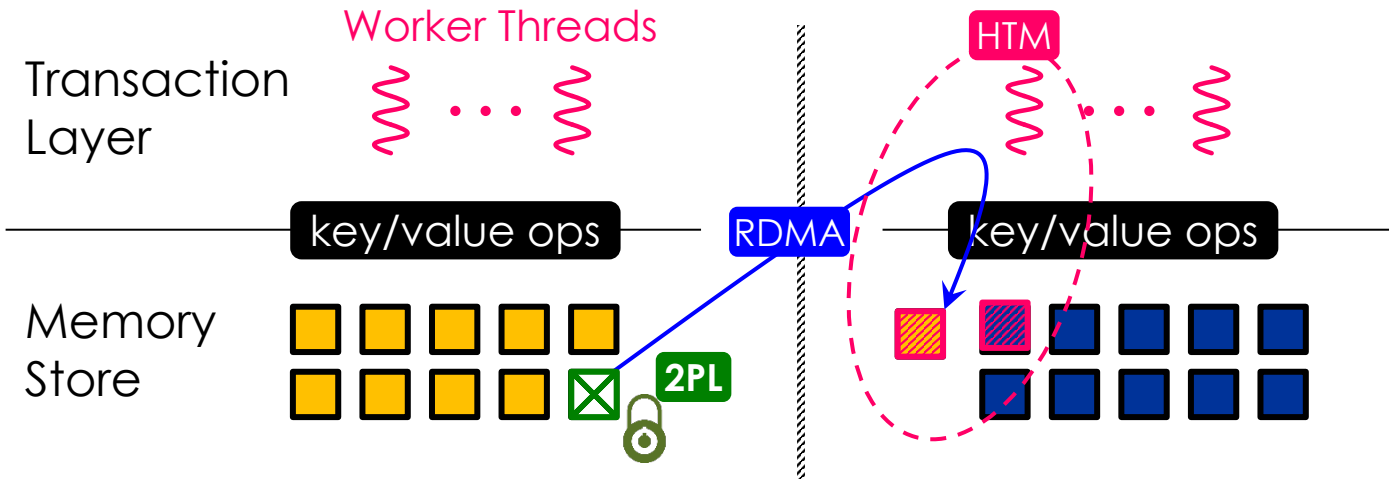
In-Memory Store

One-sided RDMA Ops

Combining HTM with 2PL

Using **2PL** to accumulate all remote records prior to accesses in an HTM transaction

- Transform a distributed TX to a local one
- Limitation: require advanced knowledge of read/write sets of transactions[1]



[1] This is similar with prior work (e.g. Sinfonia & Calvin) and the case for typical OLTP workloads

DrTM's Concurrency Control

Local TX vs. Local TX: HTM

Distributed TX vs. Distributed TX: 2PL

Local TX vs. Distributed TX: abort local TX

DrTM's Concurrency Control

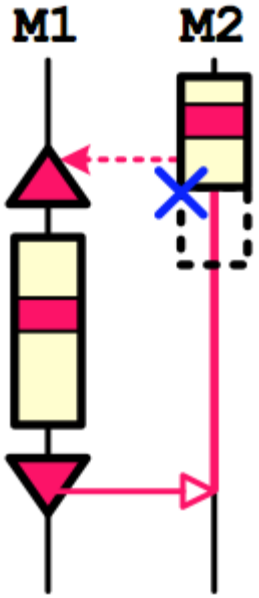
Local TX vs. Local TX: HTM

Distributed TX vs. Distributed TX: 2PL

Local TX vs. Distributed TX: abort local TX

Local TX prior to
Distributed TX

→ RDMA (strong consistency)
+ HTM (strong atomicity)



DrTM's Concurrency Control

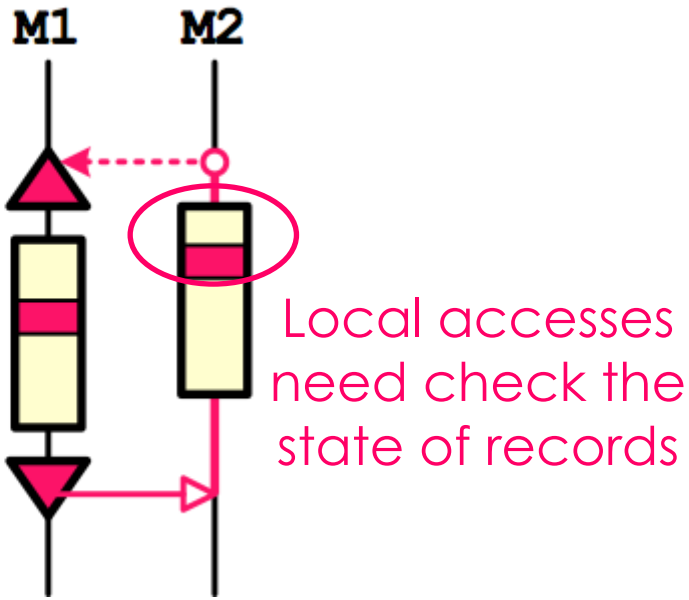
Local TX vs. Local TX: HTM

Distributed TX vs. Distributed TX: 2PL

Local TX vs. Distributed TX: abort local TX

Distributed TX
prior to Local TX

→ Local TX checks (but not locks) the records



Conclusion

High COST of concurrency control in distributed transactions calls for new designs

New hardware technologies open opportunities

DrTM : The first design and impl. of combining HTM and RDMA to boost in-memory transaction system

Achieving orders-of-magnitude higher throughput and lower latency than prior general designs

In-memory KV-Stores

Interface: **GET**, **PUT**, etc.

A key pillar for many systems

- Data cache (e.g., Memcached in FB)
- In-memory database (e.g., OLTP, OLAP)
- Graph query processing

Key requirements

- Low latency
- High request rate (Throughput)

Key Questions

Communication mode btw. machines

- Symmetric vs. asymmetric

Data structure in KVS for RDMA

- Hash-based vs. tree-based
- How to store keys and values

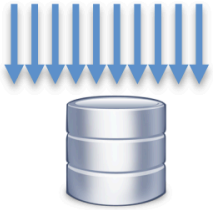
Interference between primitives

- Read vs. Write
- Local vs. Remote

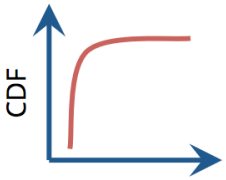
Where is Latency?

- CPU and its cache
- Client and server over a network
- Application and disk
- Anywhere a system does work

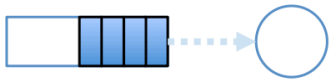
Why latency exists?



Resource contention



Skewed access patterns



Queueing delays

Background activities

Living with Latency Variability

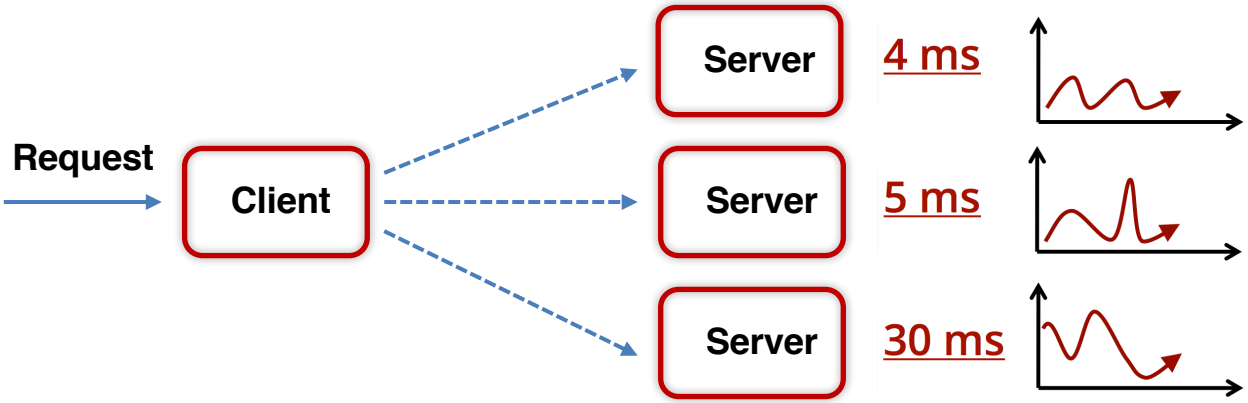
Latency variability is inevitable

Tail-tolerant techniques are necessary

- Within request short-term adaptations
- Cross request long-term adaptations

Hedged Requests

- Issue the same request to multiple replicas
- Use the results from whichever replica responds *first*



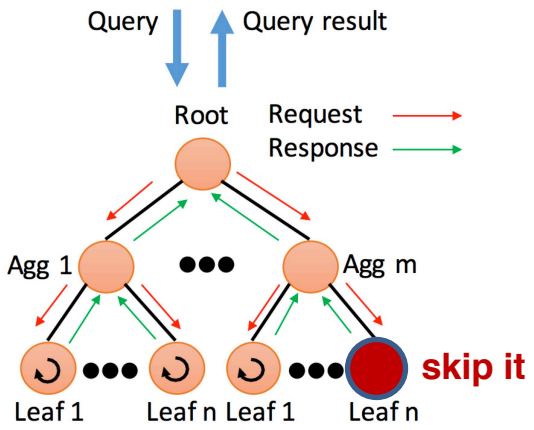
Selective Replication

- Enhancement of Micro-partition
- Predict possible load imbalance and create additional replicas
  - e.g., Google's web search system will make additional copies of popular and important documents in multiple micro-partitions

Good-enough Schemes

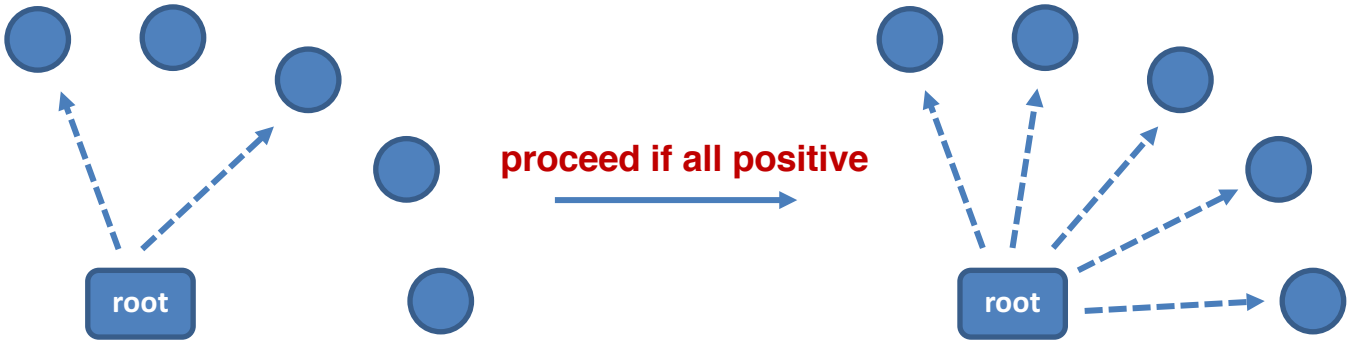
- Return once *a sufficient fraction* of all the leaf servers has responded

- Skip nonessential subsystems to improve responsiveness
  - e.g. Results from ads or spelling-correction systems are easily skipped for Web searches if they do not respond in time



Canary Requests

- Request incurs unexpected crash or long delays



Is this request OK?

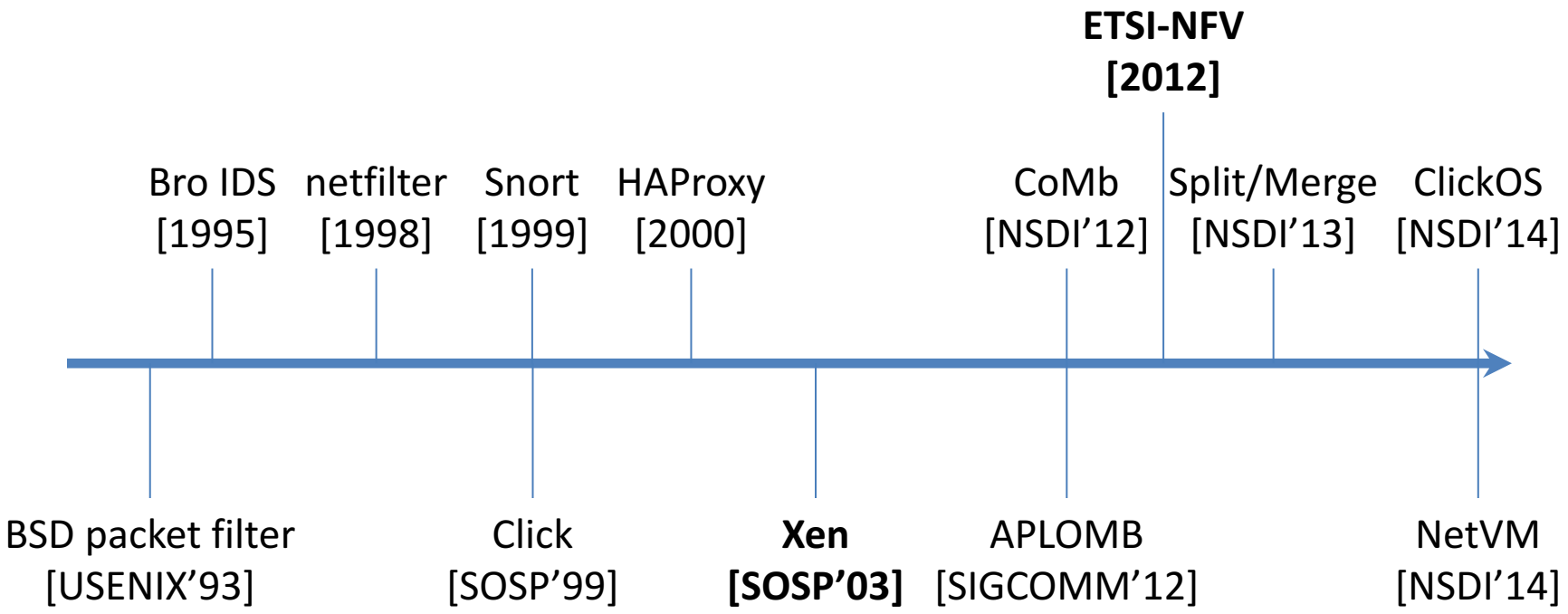
Goal of NFV

- Save money
  - Commodity server hardware
  - Workload consolidation
  - Lower power consumption
  - Simplified network maintenance
- Make money
  - Accelerated service deployment
  - Network infrastructure as a service

What is NFV?

- Network Function
  - Middleboxes (firewall, IDS, WAN optimizer)
- Virtualization
  - Commodity hardware
  - Consolidation
- **Network Services** running on **Cloud Infrastructure**

NFV: History



Software Packet Processing

Network Function Virtualization

NFV: Summary

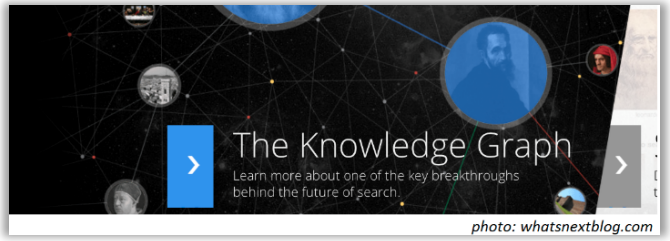
- Virtualization of network appliances
- Performance and flexibility
- System design
  - Multicore and NUMA hardware
  - Distributed processing
  - I/O optimization

Summary

- NFV: everything old is new again
- Cloud networks need
  - Flexible operational model
  - State-of-the-art infrastructure services
  - Programmable network functions
- NFV is changing the networking industry

Graphs are Everywhere

Online ***graph query*** plays a vital role for searching, mining and reasoning linked data



Jena

TAO

Unicorn

Graph Analytics vs. Graph Query

	Graph Analytics	Graph Query
Graph Model	Property Graph	Semantic (RDF) Graph
Working Set	A whole Graph	A small frac. of Graph
Processing	Batched & Iterative	Concurrent
Metrics	Latency	Latency & Throughput

System Overview

Wukong : A distributed in-memory RDF store

- *RDMA-friendly* graph model
- RDMA-based join-free graph exploration
- Concurrent query processing
- Results vs. state-of-the-art (TriAD/Trinity.RDF)
  - Latency: 11.9X – 28.1X reduction
  - Throughput: 269K queries/sec (up to 740X improvement)

What is a Bug?

a bug is -

- a *contradiction* in beliefs
  - MUST - implied by the code

- a deviation from *common behavior*
  - MAY - inferred from the code
  - probability of coincidence

Bugs Cost??

Patriot missile defense system

- 28 dead soldiers, 98 wounded

Therac-25 medical device

- Several people dead, others wounded

General Electric XA/21

- 50 million people left without water, electricity.

How to avoid unstable code

- Programmers
  - Fix bugs
  - Work around: disable certain optimizations
- Compilers & checkers
  - Many bug-finding tools fail to model C spec correctly
  - Use our ideas to generate better warnings
- Language designers: revise the spec
  - Eliminate undefined behavior? Perf impact?