HyCache: a User-Level Caching Middleware for Distributed File Systems

Dongfang Zhao, Ioan Raicu

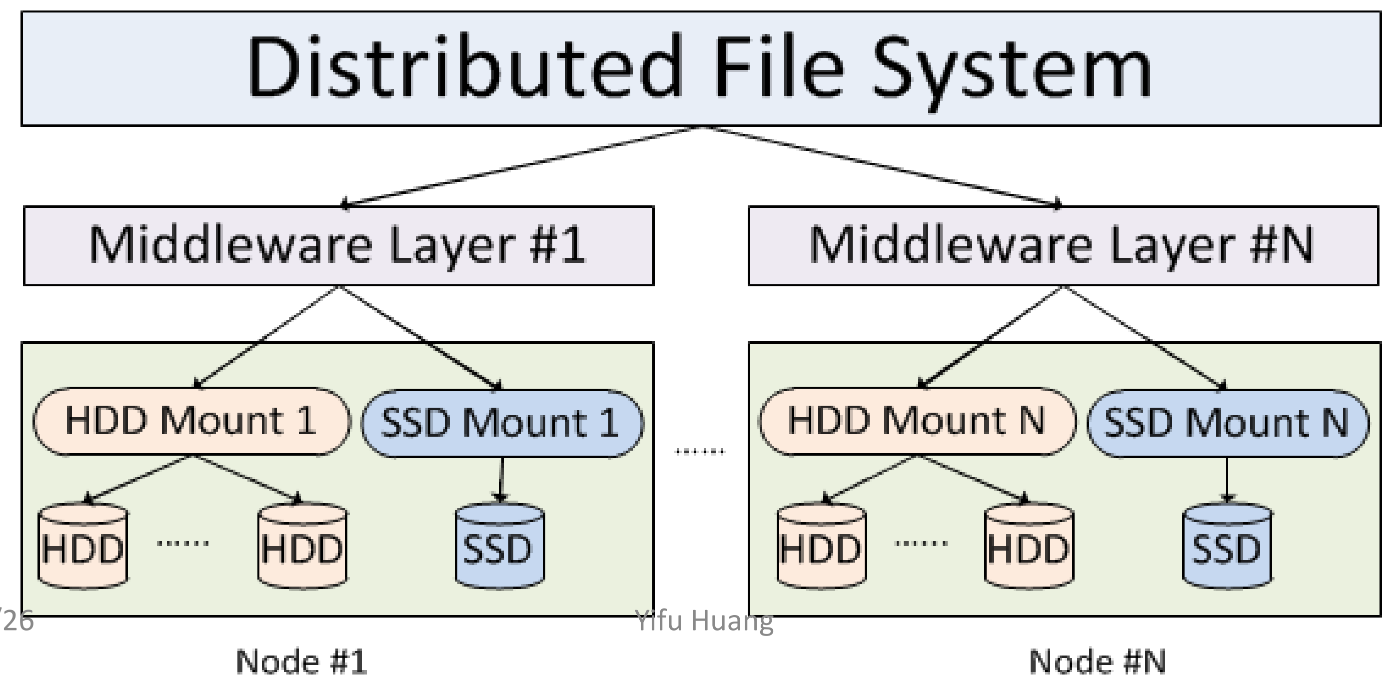
IPDPS workshop 2013

Motivation

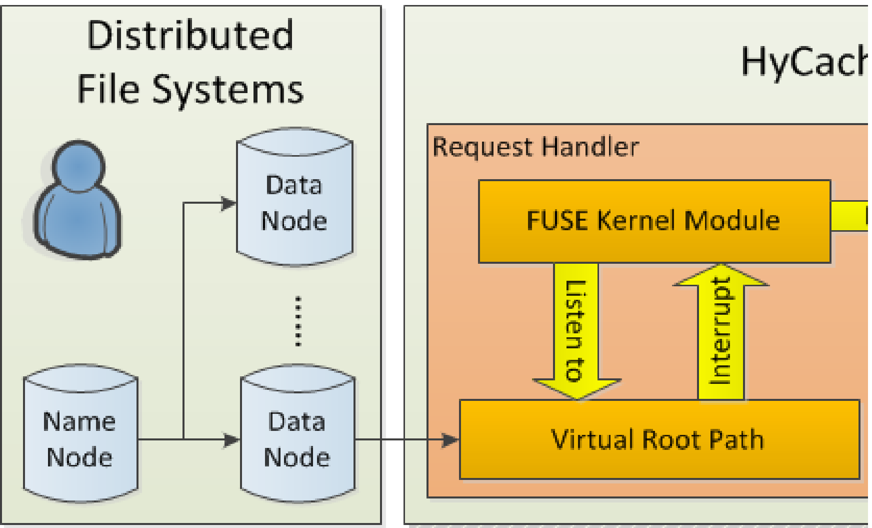
- One of the bottlenecks of distributed file systems is mechanical hard disk drives (**HDD**)
  - **Slow** increase in bandwidth
  - **Slow** decrease in latency
  - **Exponential** increase in capacity
- **Unbalanced!**
- SSD could bridge the gap nicely between RAM and HDD for a distributed file system
- **SSD performance** + **capacity of HDD**

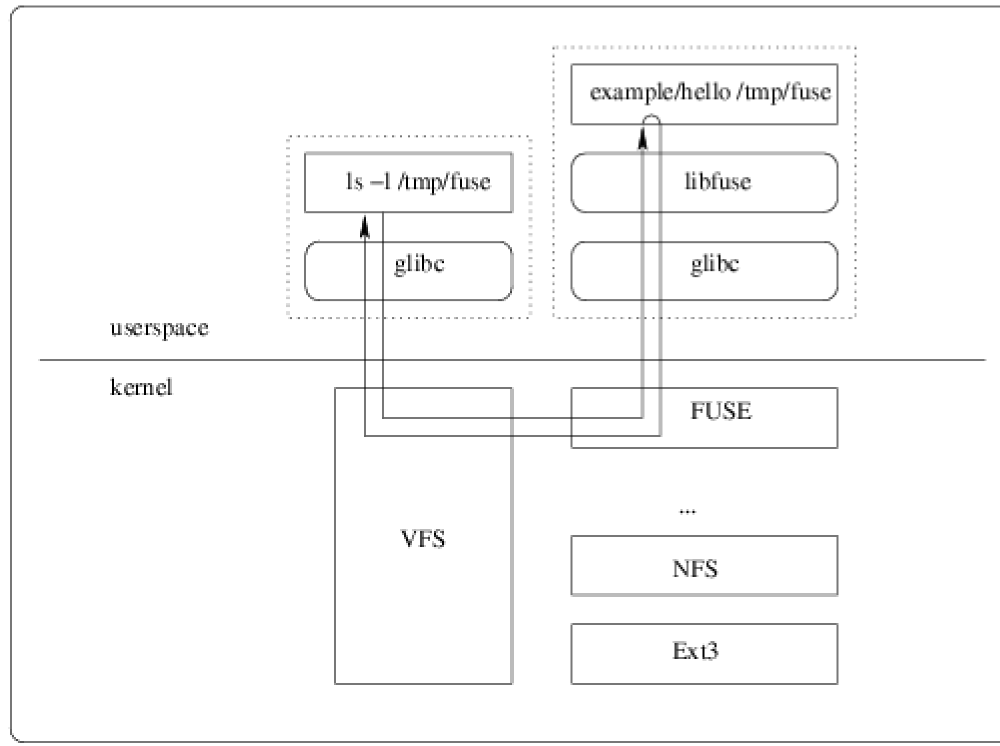
Middleware hierarchy

- The storage hierarchy with a middleware between **distributed file systems** and **local file systems**
- Manage **heterogeneous** storage devices for distributed file systems



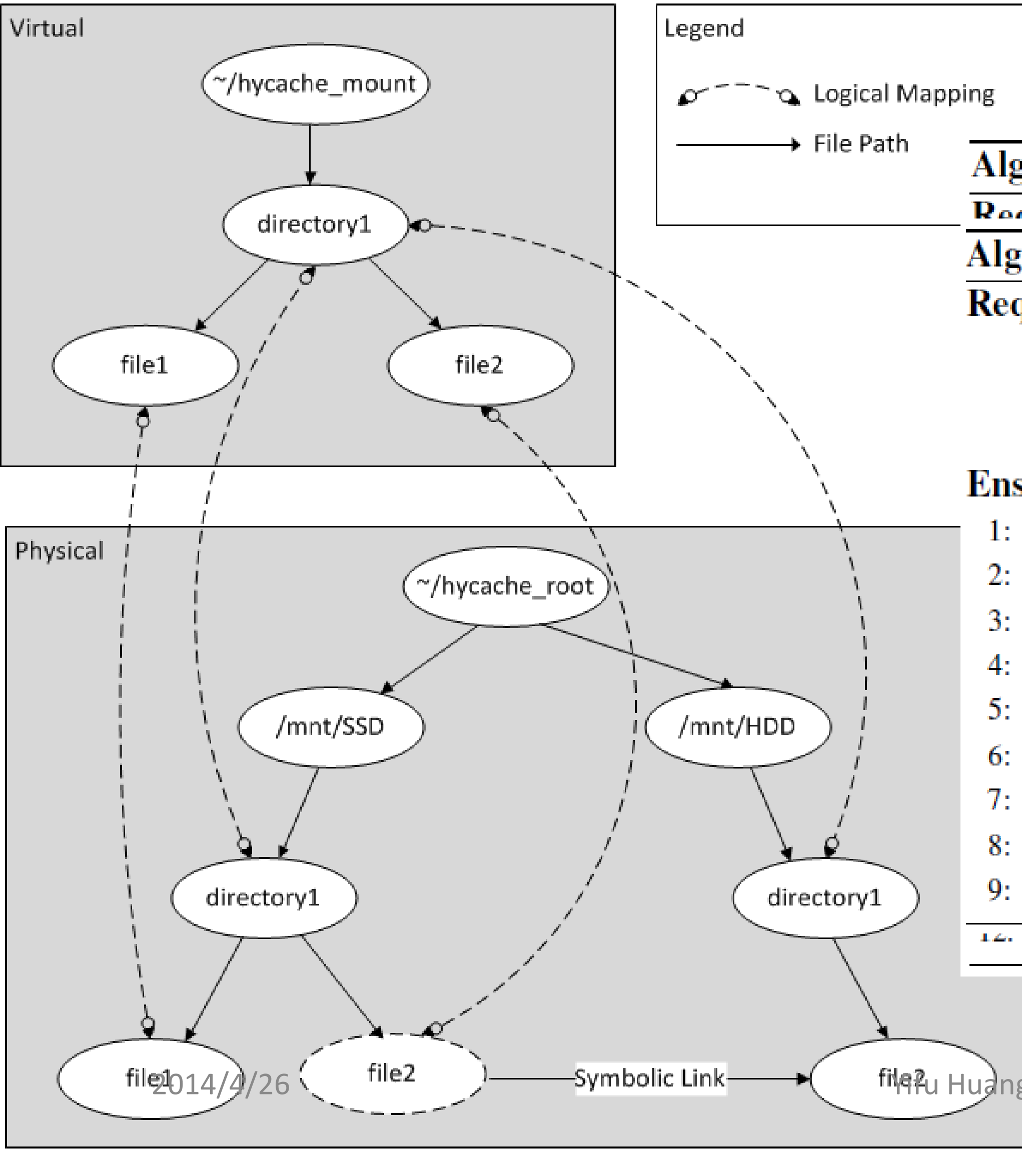
HyCache architecture





- **Request Handler**
  - Mount point is monitored
- **File Dispatcher**
  - File manipulations and cache algorithms
- **Data Manipulator**
  - Manipulates data between two logical access points

HyCache implementation



Algorithm 1 Open a file in HyCache

Require: F is the file requested by the end user; Q is the

Algorithm 3 Rename a file in HyCache

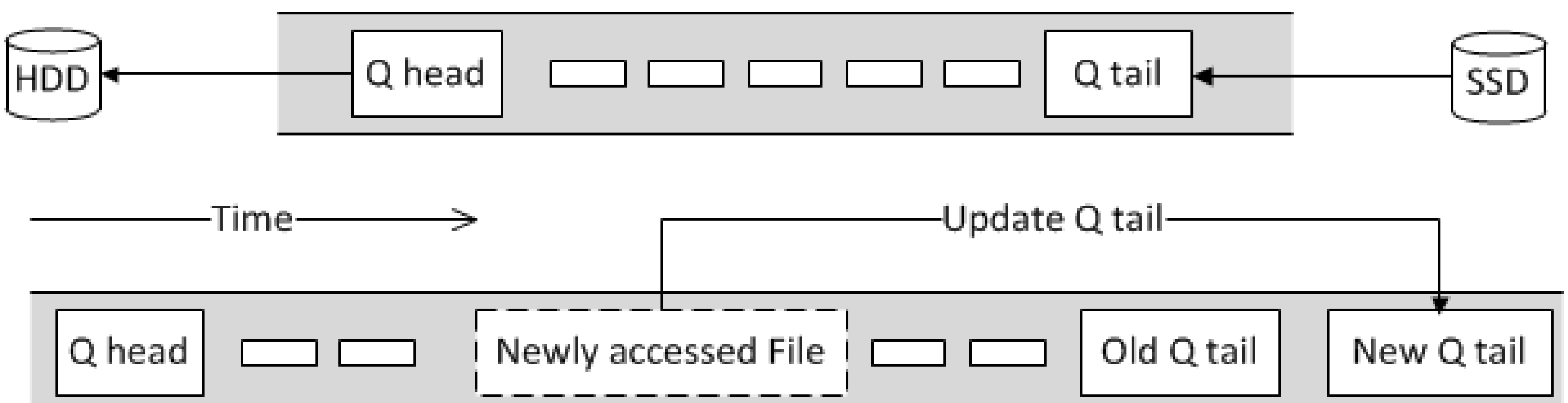
Require: F is the file requested by the end user to rename; F' is the new file name; Q is the queue used for the replacement policy; SSD is the mount point of SSD drive; HDD is the mount point of HDD drive

Ensure: F is renamed to F'

```
if F is a symbolic link in SSD then
    rename F to F' in HDD
    remove F in SSD
    create the symbolic link F' in SSD
else
    rename F to F' in SSD
    rename F to F' in Q
end if
update F' position in Q
```

HyCache implementation (Cont.)

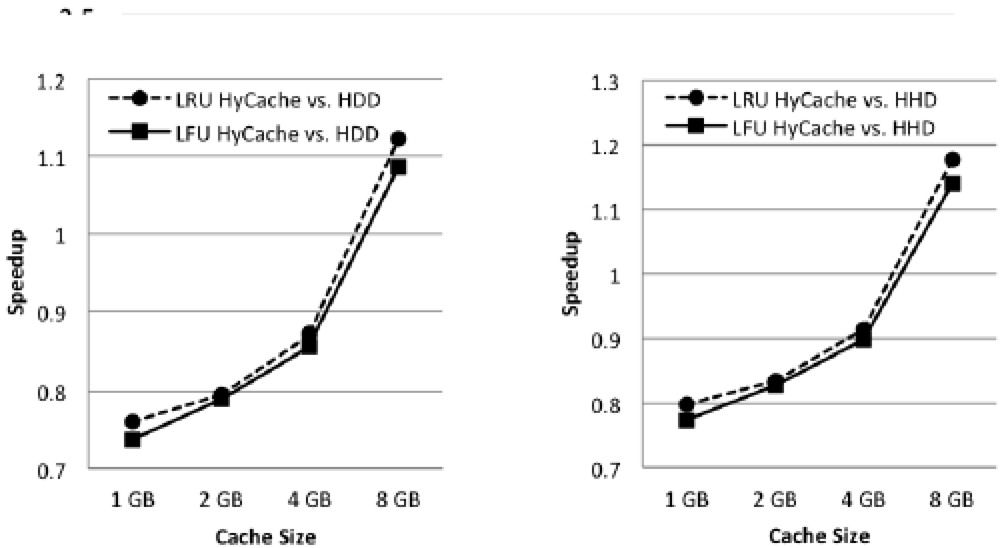
- Two built-in cache algorithms: **LRU** and **LFU**
  - Free to plug in other cache algorithms
  - With the standard C library <search.h>
  - Doubly-linked list
  - Element: filename, access time, number of access, etc
- LRU queue in HyCache



Evaluation

- Tested the **functionality** and **performance** of HyCache in four experiments
  - First two are benchmarks with synthetic data to test the raw bandwidth of HyCache
    - Micro-benchmarks
    - macro-benchmarks
  - The third and fourth experiments are to test the functionality of HyCache with a real application
    - MySQL
    - HDFS

Evaluation (Cont.)



(a) HyCache vs. HDD (b) HyCache vs. HHD

Contribution

- Designed and implemented HyCache
  - High throughput, low latency, strong consistency, single namespace, and multithreaded support
- Developed a middleware layer
  - Delivered 28% improvement in HDFS performance
- Extensive performance evaluation
  - Competitive with kernel-level file systems

HyCache+: Towards Scalable High-Performance Caching Middleware for Parallel File Systems

Dongfang Zhao, Kan Qiao, Ioan Raicu

CCGrid 2014

Motivation

- The ever-growing **gap** between the **computation** and **I/O** is one of the fundamental challenges for today's large scale computing systems
  - The number of compute cores follows Moore's Law
  - The storage systems have been improved at a much slower pace
- Even larger for modern high-performance computing (**HPC**) systems

HyCache+

- Distributed storage middleware HyCache+
  - Allows I/O to effectively leverage the **high bi-section bandwidth** of the **high-speed interconnect** of massively parallel high-end computing systems
  - The primary place for holding hot data for the applications
  - Only asynchronously swaps with cold data on the remote parallel file system
- Opens the door to providing both **high performance** and cost-effective **large capacity**

HyCache+ hierarchy

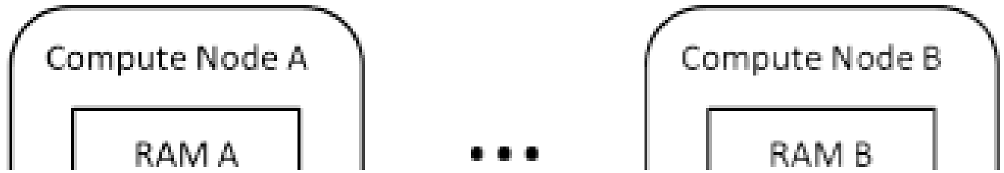
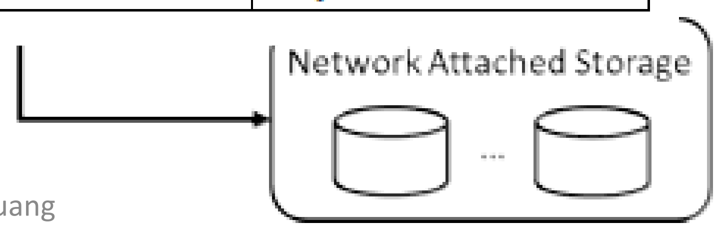


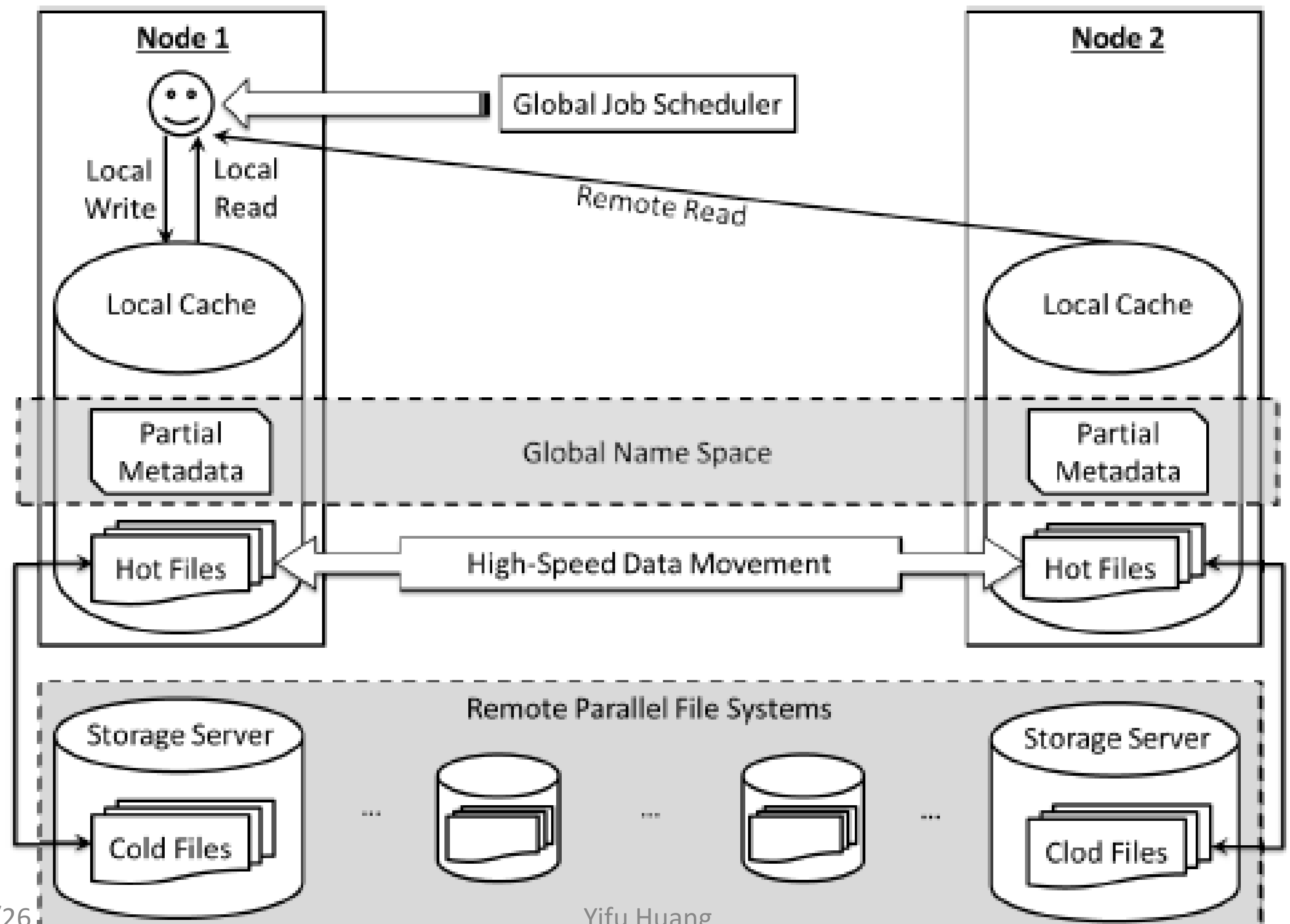
Table I
SOME KEY HYCACHE+ IMPROVEMENTS OVER HYCACHE

Mechanism	HyCache+	HyCache
Network Storage	Yes	No
Data Movement	Local & Remote	Local Only
Replica	Arbitrary (e.g. 3)	1
Scalability	4096-cores	1-node
Metadata	DHT	Symbolic Link

1 – Cache hit
2 – Data swap between cache and GPFS
3 – Data movement across compute nodes



HyCache+ design overview



HyCache+ Metadata

- **DHT** is the translator between local partial metadata and the global namespace

Key	Value
~/	drwxrwxr-x; 4.0K; ~/homedir/subdir
~/homedir/	drwxrwxr-x; 4.0K; ~/homedir/subdir, ~/homedir/homefile
~/homedir/subdir/	drwxrwxr-x; 4.0K; ~/homedir/subdir/subfile
~/homedir/homefile	-rw-rw-r--; 423M; Node 1
~/homedir/subdir/subfile	-rw-rw-r--; 133M; Node 2
⋮	⋮

- Only the primary copy is used for read and write operation, and replicas are only used to avoid data loss in case of node failures

HyCache+ Fault Tolerance

- Synchronous replicas
  - A costly method, and satisfies the strong consistency requirement, if needed
- Asynchronous replicas
  - Would deliver the highest throughput, while being compromised on the possibility of failing to recover before the asynchronous update is completed
- Erasure coding
  - Trades off between performance and consistency

2-Layer Scheduling

- **1. Job Scheduling**

That is to solve the objective function

$$\arg\min_{Q} \sum_{A_k \in A} \sum_{M_l \in M} \sum_{F_i \in F^k} \sum_{M_j \in M} Size(F_i) \cdot P_{i,j} \cdot Q_{k,l},$$

$$\sum_{M_j \in M} P_{i,j} = 1, \forall F_i \in F,$$

$$\sum_{M_j \in M} Q_{i,j} = 1, \forall A_i \in A,$$

$$P_{i,j}, Q_{i,j} \in \{0, 1\}, \forall i, j.$$

by $A_k \in A$
n $M_j \in M$
d on $M_j \in M$

Algorithm 1 Global Schedule

Input: The x^{th} job to be scheduled

Output: The y^{th} machine where the x^{th} job should be scheduled

```
function GLOBALSCHEDULE(x)
    MinCost ← ∞
    y ← NULL
    for M_i ∈ M do
        Cost ← 0
        for F_j ∈ F^x do
            Find M_k such that P_{j,k} = 1
            Cost ← Cost + Size(F_j)
        end for
        if Cost < MinCost then
            MinCost ← Cost
            y ← i
        end if
    end for
    return y
end function
```

2-Layer Scheduling (Cont.)

- **2. Heuristic Caching**
  - The problem of finding optimal caching on multiple-disk is proved to be NP-hard
  - Propose a heuristic algorithm of **O(n lg n)**
  - **Cost:** the to-be-evicted file size multiplied by its access frequency after the current processing position in the reference sequence
  - **Gain:** the to-be-cached file size multiplied by its access frequency after the current fetch position in the reference sequence

2-Layer Scheduling (Cont.)

- **2. Heuristic Caching (Cont.)**
  - **Rule 1.** Every fetch should bring into the cache the very next file in the reference sequence if it is not yet in the cache.
  - **Rule 2.** Never fetch a file to the cache if the total **cost** of the to-be-evicted files is greater than the **gain** of fetching this file.
  - **Rule 3.** Every fetch should discard the files in the increasing order of their **cost** until there is enough space for the newly fetched file. If the cache has enough space for the new file, no eviction is needed.

2-Layer Scheduling (Cont.)

- **2. Heuristic Caching (Cont.)**
  - **Exa**

$R = (r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9)$. Let $File(r_1) = F_1$, $File(r_2) = F_2$, $File(r_3) = F_3$, $File(r_4) = F_4$, $File(r_5) = F_3$, $File(r_6) = F_1$, $File(r_7) = F_2$, $File(r_8) = F_4$, $File(r_9) = F_3$, and $Size(F_1) = 20$, $Size(F_2) = 40$, $Size(F_3) = 9$, $Size(F_4) = 40$. Let the cache capacity be 100. According to *Rule 1*, the first three files to be fetched to cache are (F_1, F_2, F_3). Then we need to decide if we want to fetch F_4. Let $Cost(F_i)$ be the cost of evicting F_i. Then we have $Cost(F_1) = 20 \times 1 = 20$, $Cost(F_2) = 40 \times 1 = 40$, and $Cost(F_3) = 9 \times 2 = 18$. According to *Rule 3*, we sort the costs in the increasing order (F_3, F_1, F_2). Then we evict the files in the sorted list, until there is enough room for the newly fetched file F_4 of size 40. In this case, we only need to evict F_3, so that the free cache space is $100 - 20 - 40 = 40$, just big enough for F_4. Before replacing F_3 by F_4, *Rule 2* is referred to ensure that the cost is smaller than the gain, which is true in this case by observing that the gain of prefetching F_4 is 40, larger than $Cost(F_3) = 18$.

2-Layer Scheduling (Cont.)

- **2. Heuristic Caching (Cont.)**
  - Fetch a file to cache or processor

Algorithm 2 Fetch a file to cache or processor

Input: i is the reference index being processed

procedure FETCH(i)
    if $\{r_j | File(r_j) = File(r_{i+1}) \wedge j > i+1\} \neq \emptyset$ **then**
        $flag, D \leftarrow GetFilesToDiscard(i, i+1)$
        if $flag = successful$ **then**
            Evict D out of the cache
            Fetch $File(r_{i+1})$ to the cache
        end if
    end if
    Access $File(r_{i+1})$ (either from the cache or the disk)
end procedure

2-Laye

- **2. Heuris**
  - Get se

Algorithm 3 Get set of files to be discarded

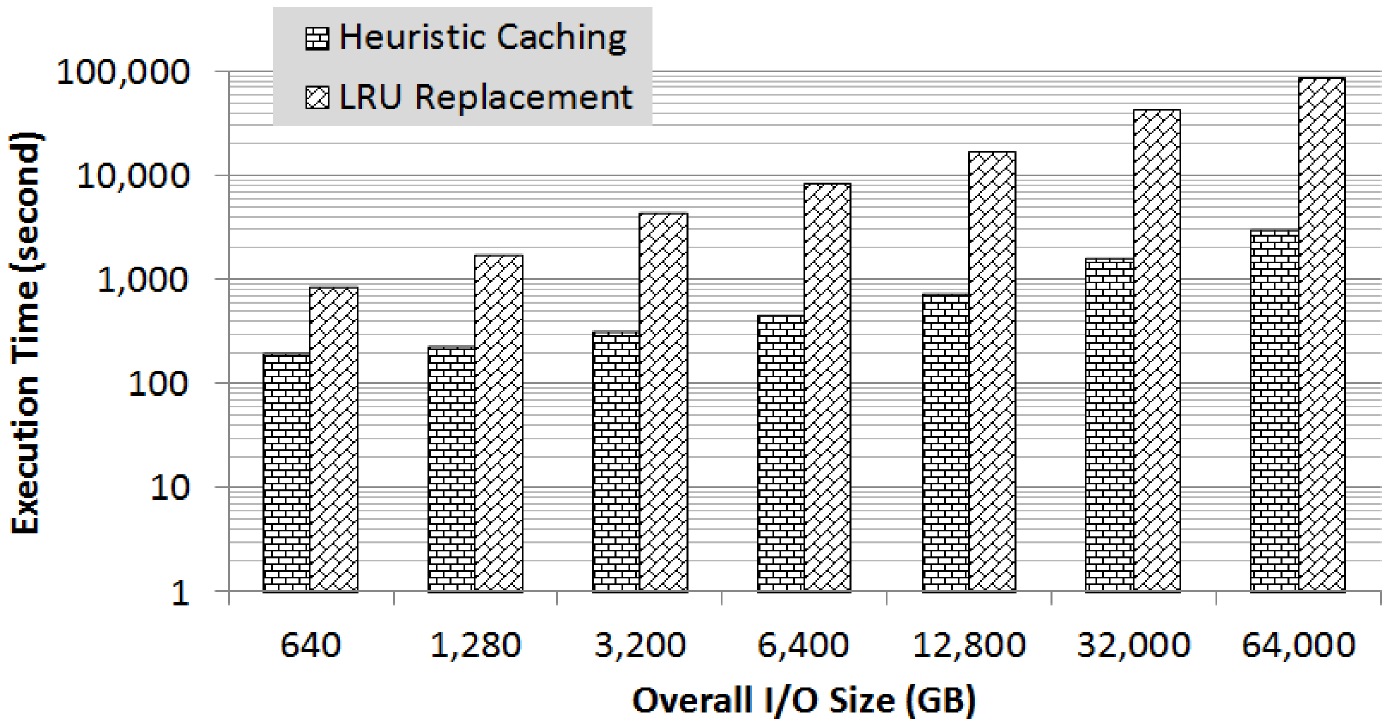
Input: i is the reference index being processed; j is the reference index to be (possibly) fetched to cache

Output: $successful$ – $File(r_j)$ will be fetched to the cache and D will be evicted; $failed$ – $File(r_j)$ will not be fetched to the cache

function GETFILESTODISCARD(i, j)
  if $Size(S) + Size(File(r_j)) \leq C$ **then**
    return $successful$, $\emptyset$
  end if
  $num \leftarrow$ Number of occurrences of $File(r_j)$ from j+1
  $gain \leftarrow num \cdot Size(File(r_j))$
  $cost \leftarrow 0$
  $D \leftarrow \emptyset$
  Sort the files in S in the increasing order of the cost
  for $F \in S$ **do**
    $tot \leftarrow$ Number of references of F from $i + 1$
    $cost \leftarrow cost + tot \cdot Size(F)$
    if $cost < gain$ **then**
      $D \leftarrow D \cup \{F\}$
    else
      $D \leftarrow \emptyset$
      return $failed$,D
    end if
    if $Size(S \setminus D) + Size(File(r_j)) \leq C$ **then**
      break
    end if
  end for
  return $successful$, D
end function

Evaluation

- 1024 nodes (4096 cores)
- E nd
  2



Contribution

- Design and implement a scalable high-performance caching middleware, namely HyCache+
  - Improve the I/O performance
- Propose and analyze a novel caching approach —2-Layer Scheduling (2LS)
  - Optimize the network cost
  - Heuristically reduce the disk I/O cost
- Evaluate at large scale
  - Report their performance on a leadership class supercomputer