An adaptive read/write optimized algorithm for Ceph heterogeneous systems via performance prediction and multi-attribute decision making

Zhike Li, Yong Wang

Cluster Computing' 23

Department of Computer Science, Dankook University

System Software Lab.

Suhwan Shin

02/05/2025

Contents

Introduction

- Ceph is a reliable, self-balancing, self-recovering distributed storage system that eliminates traditional metadata nodes. This is possible because it can map data to storage nodes through a pseudo-random data mapping function called Controlled Replication Under Scalable Hashing (CRUSH)

- The Ceph cloud storage system only **selects data storage nodes based on node storage capacity**.
  This node selection method results in load imbalance and limited storage scenarios in heterogeneous storage systems.
  (It is necessary to manually edit the CRUSH Map to adapt to different storage performance requirements)

- This paper designs a system architecture: Combines the Ceph with Software Defined Network (SDN), based on the Ceph distributed file system
  - The SDN controller **collects information** on heterogeneity, network state, and load for each type of OSD
  - Establish the **OSD read/write performance prediction model** with OSD load state
    → Dynamically adjust the relationship between OSD performance weights and load factors
  - Propose **TOPSIS_PA/TOPSIS_CW/TOPSIS_PACW** algorithms
    (TOPSIS series algorithms adaptively optimize the read/write performance of the cluster through a mathematical model)

Background & Motivation

- When storing data:
  - Client data is cut and numbered according to fixed-size objects

    → Objects mapped evenly to each PG (Placement Group)

    → PG mapped to OSD groups by the CRUSH algorithm

- The most impact on data selection OSD in the system's data mapping path
  - Mapping data objects to PGs
  - Mapping PGs to OSDs
- *HASH(oid) & mask = pgid*
  - The hash function takes *oid* as input to generate a random value
  - The hashed value and "mask" value are processed to get the PG number *pg_id*
- *CRUSH (pgid, CRUSH_Map, ruleno) =* $(OSD_0, OSD_1, \ldots, OSD_i)$
  - *i* : # of replicas
  - CRUSH_Map denotes a cluster map containing information such as cluster topology

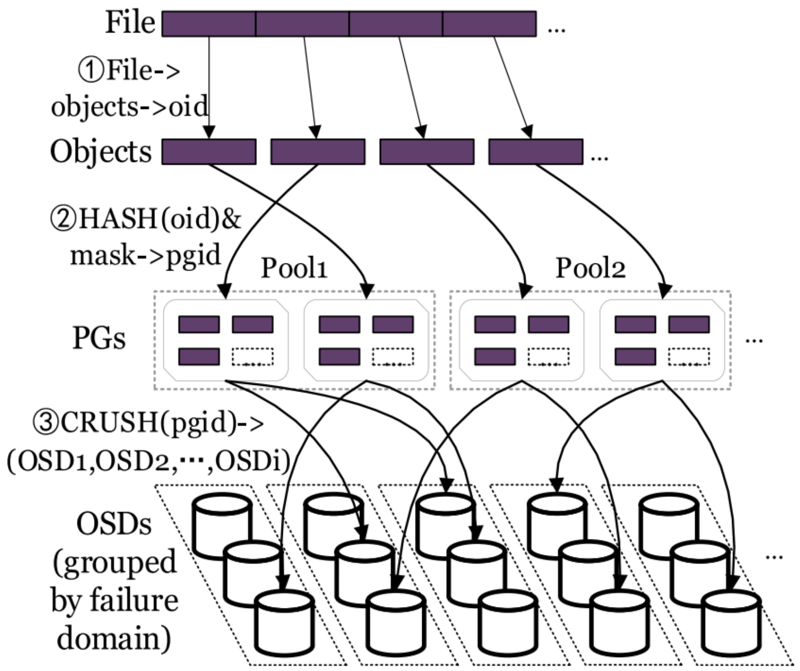


Fig. 1 Ceph cloud storage system data mapping process

Background & Motivation

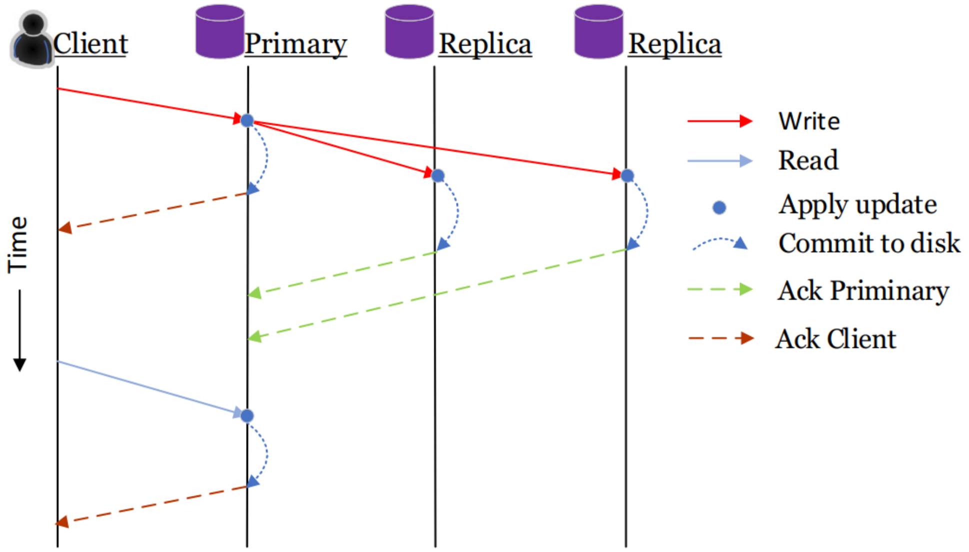
- CRUSH algorithm limitations
- Load balancing
- On-demand allocation of heterogeneous resources

- Ceph's CRUSH algorithm calculates data distribution **using storage capacity as the only determinant to obtain OSD weights**

- This mapping method can satisfy the uniformity of spatial data distribution in the cluster, but ignores the impact of:
  - The underlying **network**
  - **OSD load** on the cluster's read/write performance

- Necessary to establish an adaptive OSD selection strategy to optimize the read/write performance of the system
  - **Node's network state information**
  - **Load information**

Background & Motivation



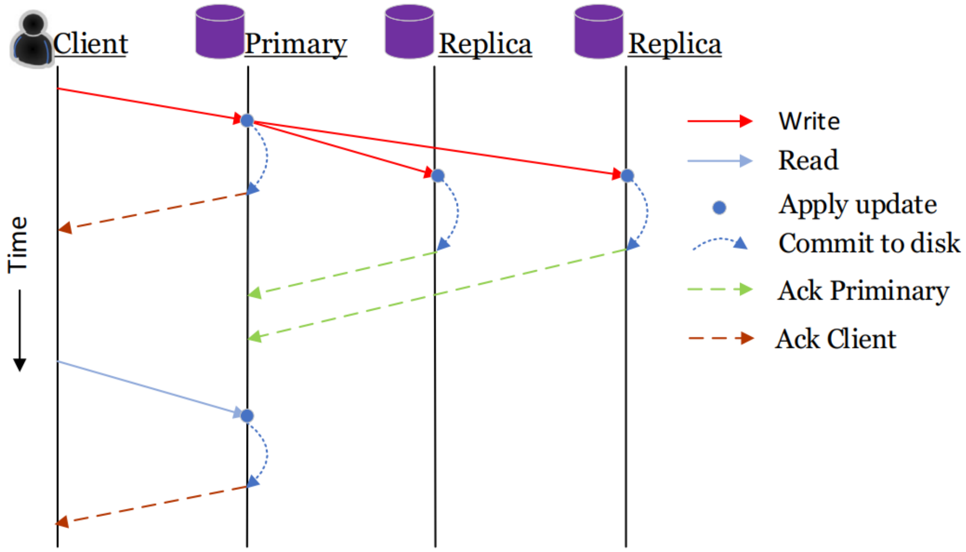
- CRUSH algorithm limitations
- Load balancing
- On-demand allocation of heterogeneous resources

- When a client requests a write operation:
  - The primary OSD first writes the data object → Sends the data object to subordinate OSDs
  - Only sends success feedback to the client after receiving success feedback from subordinate OSDs

- When performing read operations:
  - Only the primary OSD performs read operations
  - Subordinate OSDs are not selected to perform read operations → resulting in high I/O on the primary OSD

- Some OSDs may become overloaded (especially without a strategy for allocating heterogeneous resources)

Background & Motivation



- CRUSH algorithm limitations
- Load balancing
- On-demand allocation of heterogeneous resources

- To address
  - In storage pools with the same class of OSDs
    - Dynamically concentrate read and write requests on lower-loaded OSDs
  - In storage pools with different OSDs
    - Dynamically concentrate client read and write requests on higher-performing OSDs or lower-loaded OSDs

- Additionally, need to reduce the cost of manually rewriting the CRUSH Map

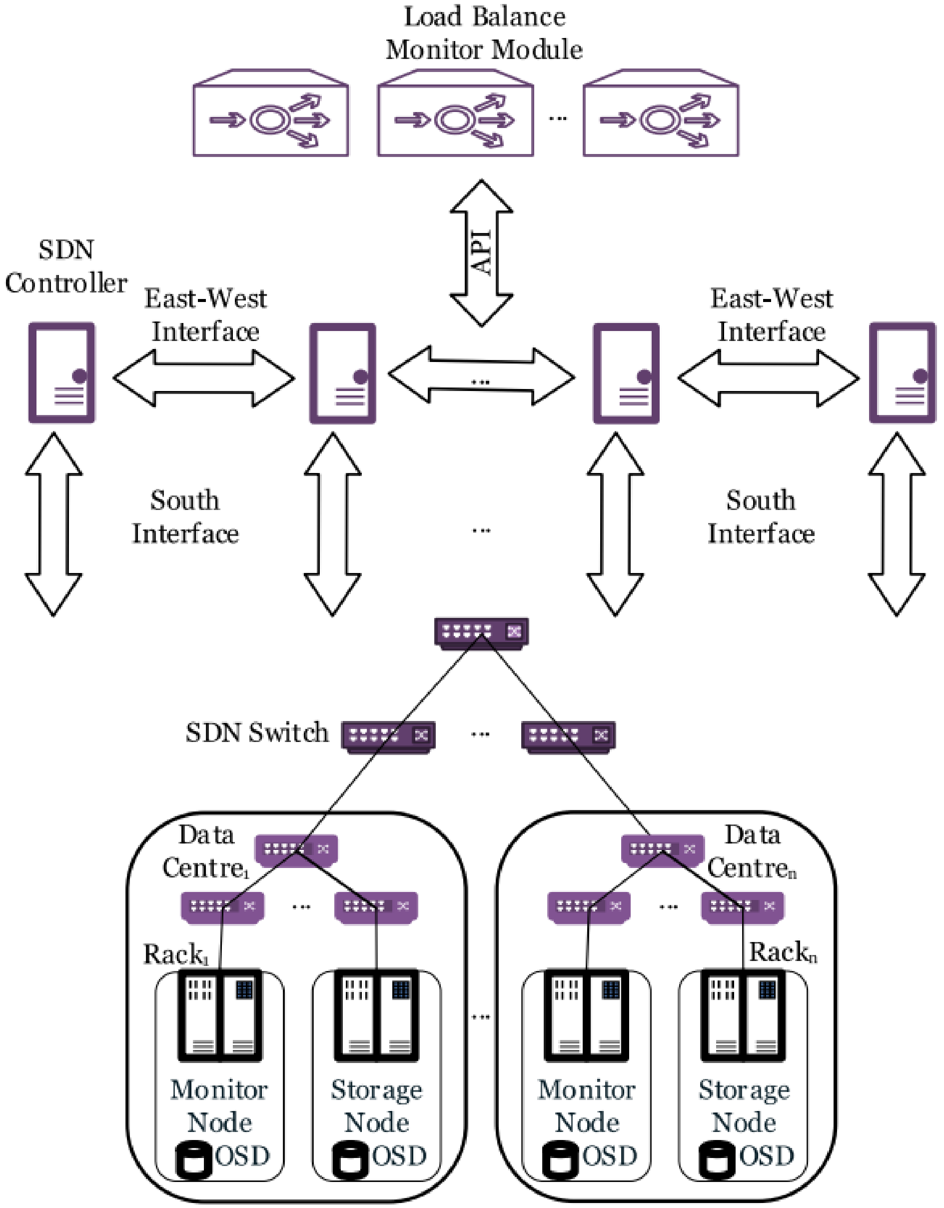
Design

- Present an adaptive read/write optimization model for Ceph heterogeneous storage systems
- 1. Integration of SDN technology architecture
  - Incorporating Software Defined Network technology into the system design
- 2. Combination of OSD host node information
  - Network state information
  - Load information
- 3. Optimize cluster read/write performance from three aspects
  - Limitations of the CRUSH algorithm
  - Load imbalance
  - Storage service scenarios

Design

- System Architecture
- Bottom Layer
  - Consists of monitor nodes and storage nodes
  - Each storage node can contain multiple OSDs
  - Monitor node maintains global configuration information for all nodes in the cluster
  - OpenFlow switch connects all servers and is responsible for data transfer
- Top Layer
  - Contains LBMM (Load Balancing Monitor Module)
  - Monitors required OSD information using the SDN controller
  - Monitor node builds OSD performance prediction model

    → Decide OSD selection on storage nodes based on information collected by SDN controller
- Data Flow
  - Integrates collected OSD load state information and bandwidth information
  - Transmits to the Monitor node of the Ceph system



Design

- OSD Performance Impact Factors
  - Determine read/write performance weights of OSDs
  - Performance Metric 1 (Node Resources)
    - Bandwidth (B)
    - Number of CPUs (C)
    - Memory size (M)
  - Performance Metric 2 (Load Status)
    - OSD's I/O load (L)
    - I/O load of the OSD disk
  - Performance Metric 3 (Node Heterogeneity)
    - Number of OSDs on heterogeneous nodes (H)
    - OSD type on heterogeneous nodes (T)
  - Performance Metric 4 (PG Distribution)
    - Number of PGs occupied by the OSD (P)

Design

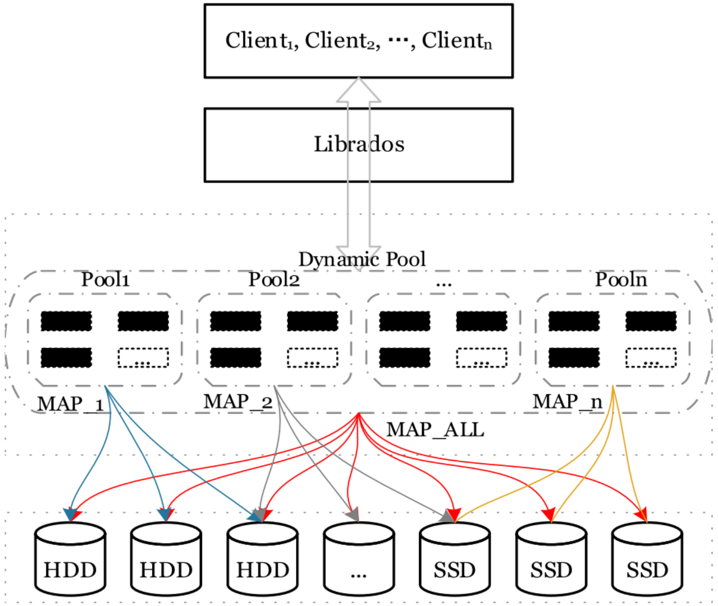
- Node Heterogeneous Resource Division Strategy (Algorithm 1: OSD hetero- Resource Partionning)
  - Line 1:
    - Obtain heterogeneous **OSD information** of nodes in the Ceph system
  - Line 2:
    - Select i ($1 \le i \le 7$) **performance metrics** according to performance requirements
    - Traverse initial heterogeneous performance set $a_j = \{e_1, e_2, ..., e_i\}$
      - e_i: initial value of jth OSD performance metric
      - t is total number of OSDs in Ceph
    - Generate $\alpha = \{\alpha_1, \alpha_2, ..., \alpha_j\}$, $1 \le j \le t$
  - Line 9:
    - Initialize OSD minimal performance set $\beta = \{\alpha_1\}$, Initialize OSD minimal classification set $\chi = \{\}$
    - If $\beta \cup \alpha_j \neq \beta$, then $\beta = \beta \cup \alpha_j$ and $\chi = \chi \cup osd$ (Otherwise β remains unchanged)
  - Final Output:
    - Generates OSD performance set $\beta = \{\alpha_1, \alpha_2, ..., \alpha_l\}$
    - Generates OSD classification set $\chi = \{osd_1, osd_2, ..., osd_l\}$
    - where $1 \le l \le t$, osd_l is number of OSDs corresponding to α_l
  - Storage Pool Performance Hierarchy: $Pool_n > ... > Pool_2 > Pool_1$

Algorithm 1: OSD heterogeneous resource partitionning algorithm

Input: OSD information (OSD_Info)

Output: OSD Minimal Classification Set (χ)

1: $OSD\_Info = get\_osd\_info\ ()$
2: **Procedure** GetOSDPerfFactor (OSD_Info)
3: $\quad \alpha = \{\}$
4: $\quad$ for osd in OSD_Info do
5: $\quad\quad$ for e_i in $OSD\_Info[osd]$ do
6: $\quad\quad\quad$ if e_i is 'expect' then
7: $\quad\quad\quad\quad \alpha[osd][e_i] = OSD\_Info[osd][e_i]$
8: **end procedur**
9: **procedure** GetOSDMiniClassifiSet (α)
10: $\quad \beta = \{\alpha_1\}, \chi = \{\}$
11: $\quad$ for osd in α do
12: $\quad\quad$ if $\beta \cup \alpha[osd] \neq \beta$ then
13: $\quad\quad\quad \beta = \beta \cup \alpha[osd]$
14: $\quad\quad\quad \chi = \chi \cup osd$
15: **end procedure**



Design

- OSD Load Monitoring Strategy (Algorithm 2: OSD load collection)
  - Purpose
    - Responsible for obtaining OSD load status (Runs on cluster's OSD nodes)
    - Passively receives messages forwarded from switch via SDN controller
    - **Collects OSD status by analyzing message packets**
  - Line 1:
    - Input parameter: host IP of all nodes
    - Forms dictionary OSD {osd: host_ip} to record mapping relationships
  - Line 4:
    - Node's CPU usage
    - Memory usage size
    - Number of OSDs
    - OSD type
    - Number of occupied PGs
    - I/O load Records in dictionary OSD_Load_Info{}
  - Line 14:
    - Sends load data to switch via UDP message

Algorithm 2: OSD load collection algorithm

Input: All host IP information in Ceph $(Host\_IP)$

Output: OSD load information (OSD_Load_Info)

1: **Procedure** GetOSD $(Host\_IP)$
2: $OSD <- Host\_IP$
3: **end procedure**
4: **procedure** GetOSDInfo (OSD)
5: $OSD\_Load\_Info = \{\}$
6: for i in OSD do
7: $OSD\_Load\_Info[i] <- \mathrm{cpu}[Host\_IP[i]]$
8: $OSD\_Load\_Info[i] <- mem[Host\_IP[i]]$
9: $OSD\_Load\_Info[i] <- \mathrm{host}[Host\_IP[i]]$
10: $OSD\_Load\_Info[i] <- \mathrm{type}[Host\_IP[i]]$
11: $OSD\_Load\_Info[i] <- \mathrm{pgs}[Host\_IP[i]]$
12: $OSD\_Load\_Info[i] <- \mathrm{io}[Host\_IP[i]]$
13: **end procedure**
14: **procedure** SendData (OSD_Load_Info)
15: $send(IP(src = 'src\_ip', dst = 'dst\_ip') /$
$UDP(dst\_port) / Raw(OSD\_load\_Info))$
16: **end procedure**

Design

- OSD Load Monitoring Strategy (Algorithm 3: Sending OSD load Information)
  - Purpose
    - Process by which Ryu controller packages OSD load information
    - Sends it to Ceph monitor node via Packet-Out packets
  - Line 1:
    - Ryu controller integrates data containing:
      - OSD load information
      - Bandwidth information
    - Combines into Data dictionary
  - Subsequent Steps:
    - Calls add_protocol() function to construct UDP packets
    - Sends out Packet_Out messages based on dictionary ip_to_port {}

Algorithm 3: Sending OSD load information via Packet-Out packets

Input: OSD load information received via Packet-In packets
Output: Packet-Out packets with OSD load information

```
1:  procedure: send_packet_out(msg, Data)
2:      datapath <-msg.datapath
3:      ofproto <-datapath.ofproto
4:      ofp_parser <-datapath.ofproto_parser
5:      pkt <-packet.Packet(msg.data)
6:      eth_header <-pkt.get_protocols
    (ethernet.ethernet)[0]
7:      dst_mac <-eth_header.src
8:      arp_header = pkt.get_protocols(arp.arp)
9:      dst_ip = arp.header.src_ip
10:     out_port = msg.match[in_port]
11:     ether_instance = ethernet.ethernet(dst_mac,
    src = controler_mac,eth_header.ethertype)
12:     ipv4_instance = ipv4.ipv4
    (src = controller_ip, dst = dst_ip, proto = 17)
13:     udp_instance = udp.udp(src_port = 12345,
    dst_port = 10086)
14:     pkt = packet.Packet()
15:     pkt.add_protocol(ether_instance)
16:     pkt.add_protocol(ipv4_instance)
17:     pkt.add_protocol(udp_instance)
18:     pkt.add_protocol(data)
19:     actions = [ofp_parser.OFPActionOutput
    (out_port)]
20      req = ofp_parser.OFPPacketOut(datapath,
    buffer_id = ofproto.OFP_NO_BUFFER,actions,
    in_port = ofproto.OFPP_CONTROLLER,data)
21:     datapath.send_msg(req)
22: end procedure
```

Design

- Adaptive Read/Write Optimization Algorithm Based on Performance Prediction & TOPSIS Model
  - OSD Read/Write Performance Prediction Model (Random Forest)
    - Step 1: Heterogeneous Resource Classification
      - Based on Node Heterogeneous Resource Partitioning Strategy
      - Obtain OSD minimal classification set $\chi = \{osd_1, osd_2, ..., osd_l\}$
      - Where $1 \leq l \leq t$, osdl is number of OSDs

Design

- Adaptive Read/Write Optimization Algorithm Based on Performance Prediction & TOPSIS Model
  - OSD Read/Write Performance Prediction Model (Random Forest)
    - Step 2: OSD Load Information Collection
      1. Initialize uniform distribution:
         - Set Crush Weight (CW) value of all OSDs to 1
         - Reset counter n (range [1, l])
         - Ensures read/write performance loads converge
      2. Data Collection:
         - Set CW step size as s
         - Set acquisition time interval as t
         - For each OSD in set χ:
           - Gradually increase CW value by s until:
             - Cluster IOPS stops growing
             - Expected performance requirements are met
         - Periodically acquire network bandwidth usage using SDN controller
         - Generate parameter sets: $consume_i = \{bw_{i1}, cpu_{i2}, mem_{i3}, pgs_{i4}, r/w\_io_{i5}\} + w/r\_io$
         - Create vector set S for nth OSD: $S = \{\{consume_1, r/w\_io_1, IOPS_1\}, ..., \{consume_p, r/w\_io_p, IOPS_p\}\}$
           (p is number of elements in vector set S)
      3. Data Transmission:
      - SDN controller sends load information in vector set S to Monitor node via Packet-Out
      - Monitor node builds performance prediction model using Random Forest

Design

- Adaptive Read/Write Optimization Algorithm Based on Performance Prediction & TOPSIS Model
  - OSD Read/Write Performance Prediction Model (Random Forest)
    - Step 3: Building OSD Performance Prediction Model
      1. Bootstrap Sampling
         - Use vector S as input
         - Select size B bootstrap samples from entire sample
         - Store in Ti
      2. Feature Selection
         - Set number of sample features to 5
         - Select k features out of 5 feature numbers for B bootstrap samples
         - Build decision tree to obtain best segmentation points
      3. Model Building and Prediction
         - Formulate model rf_reg with best-effect parameters
         - Perform feature importance analysis on OSD performance indicators
         - Obtain corresponding feature weights
         - Aggregate predictions of B bootstrap sample trees
         - Predict new performance pre_ioi (either r_ioi or w_ioi)

Design

- Adaptive Read/Write Optimization Algorithm Based on Performance Prediction and TOPSIS Model
  - Multi-Attribute Decision Model Based on TOPSIS
    - Performance Model Integration
      - When rf_reg prediction model reaches desired accuracy
      - Obtain OSD feature weights through feature importance analysis
      - Model accurately reflects impact of network state and load factors on performance
    - Decision Making Process
      - Obtain integrated performance weights based on feature weights
      - Select optimal OSD or OSD set → multi-attribute decision problem
      - Use Technique for Order Preference by Similarity to Ideal Solution (TOPSIS)
    - Implementation Steps
      1. Classify OSDs in Ceph system
      2. Build and solve TOPSIS model to optimize performance of different storage pools
    - Positive Indicators
      - Bandwidth (B)
      - CPU remaining size (C)
      - Memory remaining size (M)
      - PG ratio (P)
    - Negative Indicator
      - I/O load (L)

Design

- Adaptive Read/Write Optimization Model Based on Performance Prediction and TOPSIS Model
  - Adaptive Read/Write Optimization Model
    - Uniform data distribution and efficient read/write performance
      - Step 1: To meet balanced storage requirements in the cluster, Ceph selects an OSD set for Placement Groups (PG) based on the storage capacity as weight
      - Step 2: After all PGs select their OSDs, the TOPSIS model is used to calculate the relative read proximity of each OSD (Each OSD is scored based on its read performance → stored in a dictionary)
    - PA(Primary Affinity): Probability of an OSD becoming the primary OSD
    - CW(CRUSH Weight): Weight used to distribute PGs to OSDs (0~1)
      - The higher the CW value, the more PGs are assigned to that OSD → handles a higher read/write workload
      - Traditional Ceph only considers storage capacity → Considers: network state, node load, heterogeneity
    - TOPSIS_PACW
      - PA: Optimizes primary OSD selection for read operations
      - CW: Optimizes actual storage distribution of data objects

Algorithm 4: TOPSIS series of algorithms

```
Input: OSD_INFO_Map
Output: OSD_Perf_Set
procedure GetOSDPerf(OSD_INFO_Map)
  OSD_Perf_Set[osd]=
  TOPSIS(OSD_INFO_Map)
  for osd in OSD_Perf_Set do
    if osd is 'down' then
      remove osd from osd_addr
end procedure
procedure SetOSDPaCw(OSD_Perf_Set)
  Select = get_optimize_algorithm()
  if Optimize_Pool is '0' then
    for osd in OSD_Perf_Set do
      If Select is '0' then
        osd_primary_affinity=
        OSD_Perf_Set[osd]
        update {osd: primary affinity}
      elif Select is '1' then
        osd_crush_weight=
        OSD_Perf_Set[osd]
        update {osd: crush weight}
      elif
        osd_primary_affinity=
        OSD_Perf_Set[osd]
        osd_crush_weight=
        OSD_Perf_Set[osd]
        update {osd: primary affinity,
        crush weight }
  else
    OSD_Type = get_osd_type()
    for osd in OSD_Perf_Set do
      if Type[osd] is Optimize_Pool then
        If Select is '0' then
          osd_primary_affinity += s
          update {osd: primary affinity}
        elif Select is '1' then
          osd_crush_weight += s
          update {osd: crush weight}
        elif
          osd_primary_affinity += s
          osd_crush_weight += s
          update {osd: primary affinity,
          crush weight}
end procedure
```

Design

- Adaptive Read/Write Optimization Model Based on Performance Prediction and TOPSIS Model
  - Adaptive Read/Write Optimization Model
    - TOPSIS series algorithms
      - Line 1: The GetOSDPerf() function is called to obtain the relative proximity of each OSD using the TOPSIS model
        - Relative Proximity: An indicator calculated by the TOPSIS model representing the distance of each OSD to the optimal performance. A higher value indicates better performance for that OSD
        - Parameters (metrics): Remaining bandwidth (B), remaining CPU capacity (C), remaining memory capacity (M), PG ratio (P), I/O load (L)
      - Line 7: The SetOSDPaCw() function is called to update PA or CW values of the OSD
        - An OSD performance prediction model is called to predict whether the IOPS value of an OSD is optimized → Make adjustment decisions accordingly

Algorithm 4: TOPSIS series of algorithms

Input: OSD_INFO_Map

Output: OSD_Perf_Set

```
1:  procedure GetOSDPerf(OSD_INFO_Map)
2:    OSD_Perf_Set[osd]=
      TOPSIS(OSD_INFO_Map)
3:    for osd in OSD_Perf_Set do
4:      if osd is 'down' then
5:        remove osd from osd_addr
6:  end procedure
7:  procedure SetOSDPaCw(OSD_Perf_Set)
8:    Select = get_optimize_algorithm()
9:    if Optimize_Pool is '0' then
10:     for osd in OSD_Perf_Set do
11:       If Select is '0' then
12:         osd_primary_affinity=
            OSD_Perf_Set[osd]
13:         update {osd: primary affinity}
14:       elif Select is '1' then
15:         osd_crush_weight=
            OSD_Perf_Set[osd]
16:         update {osd: crush weight}
17:       elif
18:         osd_primary_affinity=
            OSD_Perf_Set[osd]
19:         osd_crush_weight=
            OSD_Perf_Set[osd]
20:         update {osd: primary affinity,
            crush weight }
21:   else
22:     OSD_Type = get_osd_type()
23:     for osd in OSD_Perf_Set do
24:       if Type[osd] is Optimize_Pool then
25:         If Select is '0' then
26:           osd_primary_affinity += s
27:           update {osd: primary affinity}
28:         elif Select is '1' then
29:           osd_crush_weight += s
30:           update {osd: crush weight}
31:         elif
32:           osd_primary_affinity += s
33:           osd_crush_weight += s
34:           update {osd: primary affinity,
              crush weight}
35: end procedure
```

Evaluation

- Experimental Setup
  - Ceph Cluster (6 machines)
    1. Monitor + Storage Nodes (3 machines)
    2. Pure Storage Nodes (3 machines):
       - Node 1: 1 OSD (SSD)
       - Node 2: 2 OSDs (1 SSD + 1 HDD)
       - Node 3: 3 OSDs (all HDD)
  - OSD Configuration
    - Total: 12 OSDs
    - Classified 4 OSD_Types
    - Classification based on Host and Type (HDD/SSD)
  - OSD Distribution by Host
    - Host 1: 2 SSDs
    - Host 2: 2 HDDs + 2 SSDs
    - Host 3: 6 HDDs

OSD	bw	cpu	mem	Host	Type	Set β_i	OSD_Type
{4, 6, 12, 17, 18, 23}	1	8	8	3	1	{1, 8, 8, 3, 1}	1
{8, 9}	1	16	8	1	2	{1, 16, 8, 1, 2}	2
{10, 16}	1	16	8	2	1	{1, 16, 8, 2, 1}	3
{7, 30}	1	16	8	2	2	{1, 16, 8, 2, 2}	4

  - Storage Pool Settings
    - Number of replicas: 2
    - Number of PGs: 512
  - Performance Test Configuration
    - Test data size: 4KB ~ 1024KB
    - Workloads: Random/Sequential, Read/Write
    - FIO settings: iodepth=128, numjobs=8
  - Network Configuration
    1. Public Network
       - For client-cluster communication
    2. Cluster Network
       - For data recovery and migration between OSDs

Evaluation

- Model accuracy

Table 4 OSD performance prediction model accuracy

Vector Set:{S}	Precision(r_io)	Precision(w_io)
$\{S_6\}$	96.31	99.34
$\{S_9\}$	96.10	93.55
$\{S_{10}\}$	94.93	92.19
$\{S_{30}\}$	95.68	96.20

$$precision = \left(1 - \frac{\sum_{i=1}^{n}(\widehat{pre\_io}_i - io_i)^2}{\sum_{i=1}^{n}(io_i - \overline{io})^2}\right) \times 100\%$$

- OSD predictive read/write performance weight

Table 5 OSD predictive read performance weights

Weight set	bw	cpu	mem	pgs	w_io
$\{R_6\}$	0.42	0.17	0.12	0.18	0.11
$\{R_9\}$	0.63	0.16	0.05	0.10	0.06
$\{R_{10}\}$	0.20	0.08	0.15	0.26	0.31
$\{R_{30}\}$	0.07	0.02	0.19	0.44	0.28

HDD-Type OSD

SSD-Type OSD

Table 6 OSD predictive write performance weights

Weight Set	bw	cpu	mem	pgs	r_io
$\{W_6\}$	0.21	0.08	0.22	0.31	0.18
$\{W_9\}$	0.11	0.02	0.19	0.57	0.12
$\{W_{10}\}$	0.28	0.01	0.14	0.37	0.20
$\{W_{30}\}$	0.25	0.07	0.18	0.39	0.11

Evaluation

- Performance Evaluation
  - TOPSIS_PACW
    - TOPSIS_PA (Primary Affinity)
      - PA: Dynamically adjusts primary OSD selection for read operations
        → Places operations on (better reading performance & lower load) OSDs
      - Optimize read performance
    - TOPSIS_CW (CRUSH Weight)
      - CW: Weight used for distributing PGs to OSDs
        → Adaptively migrates PGs carrying data objects to OSDs
      - Optimize write performance and data distribution
  - Limitations of the TOPSIS series algorithms
    - "hot spot" on the best-performing OSDs

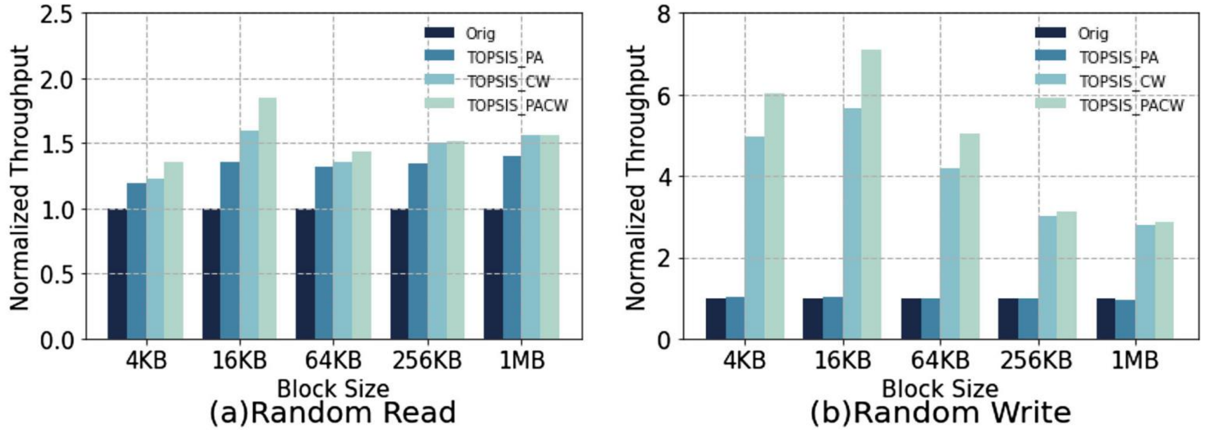


Fig. 8 Comparison of the normalized throughput of the TOPSIS series algorithm at different workloads

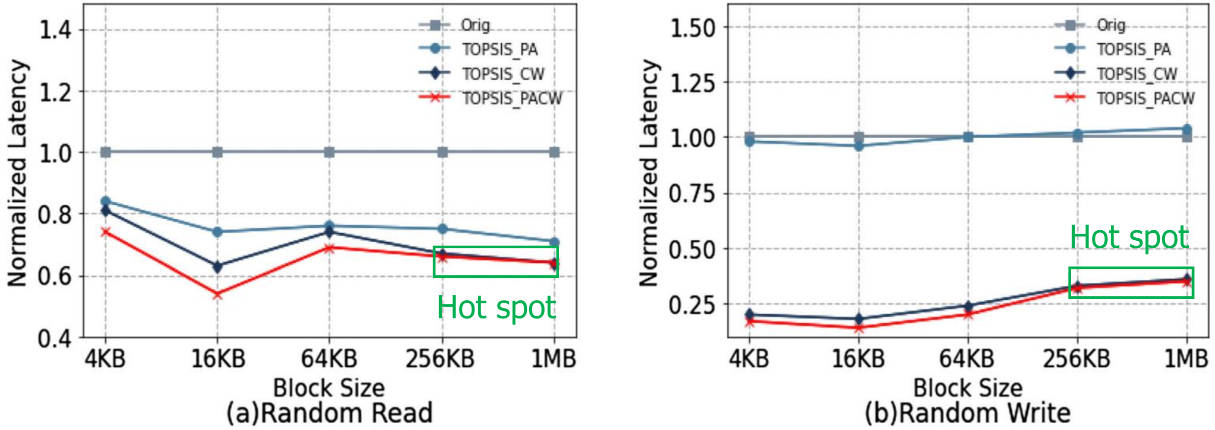


Fig. 9 Comparison of the normalized latency of the TOPSIS series algorithm at different workloads

Conclusion

- To build a large-scale distributed storage system, it is essential to consider heterogeneous distributed storage. However, Ceph's design does not account for factors such as network conditions, node load, and heterogeneity

- To improve the CRUSH algorithm, an adaptive read/write optimization algorithm for Ceph heterogeneous systems is proposed through performance prediction and multi-attribute decision-making
  - OSDs are classified based on a node **heterogeneous resource division strategy**
  - A **prediction model** is established by combining the load states of OSDs
  - An optimal OSD is selected by solving the mathematical model for **multi-attribute decision-making**

- Experimental results show that the TOPSIS_PACW algorithm improves write performance by 180% to 468% and read performance by 23% to 60%, while ensuring system reliability and high availability

Thank you

Suhwan Shin
sshshin@dankook.ac.kr

Design

- SDN Controller Process

1. Delivers UDP packets with OSD load information via Packet-In messages
2. Ryu controller
   - Receives Packet-In message
   - Determines if sent by Ceph Monitor node
   - Records packet information as Packet Out message path
3. Protocol Processing
   - Parses IPv4 and UDP packet header protocols via get_protocols()
   - Verifies specified ADDR and port number
4. Packet Content Processing
   - Decodes packet content
   - Identifies sending host based on message input port and dpid
   - Transcribes host bandwidth and load information into OSD bandwidth and load information

Algorithm 2: OSD load collection algorithm

Input: All host IP information in Ceph ($Host\_IP$)

Output: OSD load information (OSD_Load_Info)

Procedure GetOSD ($Host\_IP$)
  $OSD <- Host\_IP$
end procedure
procedure GetOSDInfo (OSD)
  $OSD\_Load\_Info = \{\}$
  for i in OSD do
    $OSD\_Load\_Info[i] <- cpu[Host\_IP[i]]$
    $OSD\_Load\_Info[i] <- mem[Host\_IP[i]]$
    $OSD\_Load\_Info[i] <- \mathrm{host}[Host\_IP[i]]$
    $OSD\_Load\_Info[i] <- \mathrm{type}[Host\_IP[i]]$
    $OSD\_Load\_Info[i] <- \mathrm{pgs}[Host\_IP[i]]$
    $OSD\_Load\_Info[i] <- \mathrm{io}[Host\_IP[i]]$
end procedure
procedure SendData (OSD_Load_Info)
  $send(IP(src = 'src\_ip', dst = 'dst\_ip')/$
  $UDP(dst\_port)/Raw(OSD\_load\_Info))$
end procedure