Network-Accelerated Multiget Coordination for Distributed Key-Value Stores

Jiyoon Bang and Gyuyeong Kim
Department of Computer Engineering
Sungshin Women's University
Seoul, South Korea
Email: {220234008,gykim}@sungshin.ac.kr

***Abstract*—Modern key-value stores support multiget requests, which allow clients to retrieve the values of multiple specified keys in a single request. However, the multiget operation causes extra coordination overhead when requested keys are distributed across multiple servers. Unfortunately, existing coordination architectures suffer from high client overhead or limited scalability. To this end, we propose NetMC, a network-accelerated multiget coordination architecture that achieves high throughput, low latency, and scalability simultaneously. Our key idea is to distribute the labor of multiget coordination between the network switch and the client. Specifically, we offload the stateless request splitting to the I/O-optimized network switch, while the stateful reply aggregation is performed on the client side. By leveraging the strengths of each system component for coordination functionality, NetMC reduces the coordination overhead significantly. We implement a NetMC prototype on a cluster of commodity servers connected via an Intel Tofino switch. Our experimental results show that NetMC outperforms existing architectures and is robust to various system conditions.**

I. Introduction

Today's online services like web search and social networking often require hundreds of thousands of storage accesses to handle concurrent user requests. Key-value stores like Redis [1] and Memcached [2] have been fundamental building blocks for online services thanks to their ability to access key-value items quickly. Service applications often generate transactional read requests to multiple keys. For example, the client should fetch a number of data objects from key-value storage to render a single web page.

Many key-value stores support the `multiget` operation to reduce the overhead to handle multi-key access [1], [2], [3], [4], [5]. The multiget operation allows the client to get the values of multiple specified keys in a single read request. This significantly reduces latency to read multiple keys and packet processing overhead compared to accessing the keys with individual `get` requests. Thanks to the performance benefits, the multiget operation has been widely employed. For example, Meta leverages the multiget to improve the performance of Memcached clusters of Facebook [6].

While promising, the multiget operation requires extra coordination when requested keys are distributed over multiple storage servers. Specifically, we should split a single request into multiple sub-requests for different servers since not all requested keys may be stored in the same storage server. In a similar vein, we should aggregate sub-replies into a single reply. Performing the coordination at the client is a straightforward solution [1], [2], [7], but it suffers from excessive client overhead, harming throughput and latency. We may use a coordinator node to reduce the client overhead [8], [3], [5], [4]. However, the coordinator-based architecture is not scalable since it can easily become a bottleneck as request rates grow, degrading overall performance. Building a coordination layer with multiple nodes between clients and storage servers is not cost-effective. In this context, we ask the following question: *Can we coordinate the multiget operation while achieving high throughput, low latency, and scalability simultaneously?*

We answer the above question affirmatively by presenting NetMC, a Network-accelerated Multiget Coordination architecture. NetMC divides the work of multiget coordination (i.e., request splitting and reply aggregation) between the network switch and the client. Specifically, we offload the request splitting function to the network switch while performing the reply aggregation at the client. Our design rationale is that each coordination function is best performed at its vantage point. For example, the I/O-optimized network switch is well-suited for *stateless* and I/O-intensive request splitting. Furthermore, programmable switch ASICs like Intel Tofino [9] allow us to customize the switch data plane while maintaining high packet processing throughput. The client is a better vantage point for reply aggregation than the switch. This is because the operation is *stateful* and requires complex logic for variable-length data, which is non-trivial to implement in the switch data plane.

The idea of offloading request splitting to the switch imposes several technical challenges due to the strict hardware constraints. We address the challenges by simplifying the switch mechanism as much as possible through client-side assistance. This allows us to avoid potential limits on application semantics. For example, we let the client group the keys for the same server in advance and embed hints in the packet header so that the switch can identify key groups destined for the same storage server without loops. Furthermore, the client puts the server ID of key groups into the header. This allows us to support an arbitrary number of storage servers, although the switch only supports modulo operations for a power of two. With the assistance, the task of the switch is greatly simplified. It splits a request into multiple sub-requests using built-in features of the programmable switch, which are

2993-2114/25/$31.00 ©2025 IEEE
DOI 10.1109/CCGRID64434.2025.00012

packet cloning and recirculation.

We implement a NetMC prototype on a testbed consisting of 8 commodity servers and a programmable switch with Intel Tofino [9]. We conduct extensive experiments to demonstrate the efficiency and robustness of NetMC. We compare NetMC against the client-based and coordinator-based multiget coordination architectures. Experimental results show that NetMC outperforms the other architectures in throughput, latency, and scalability. NetMC is robust to various system conditions like write ratio, key size, value size, and multiget size. We show that NetMC can also deal with switch failures with rapid throughput recovery. We demonstrate the practicality through experiments with Redis [1], an in-memory key-value store widely deployed in production environments.

In summary, we make the following contributions.

- We propose NetMC, a new multiget coordination architecture that divides the labor of multiget coordination functions between the switch and the client for high performance and scalability.
- We design a custom switch data plane that implements switch-based request splitting by addressing technical challenges through client-side assistance.
- We implement a NetMC prototype with an Intel Tofino switch and demonstrate the efficiency and robustness of NetMC through extensive experiments.

The remainder of this paper is organized as follows. In Section II, we provide an overview of multiget operations and discuss the limitations of existing architectures. We describe the design of NetMC and the implementation of the system in Section III and Section IV, respectively. Section V presents the experimental results. We review related works in Section VI. Finally, we conclude our work in Section VII.

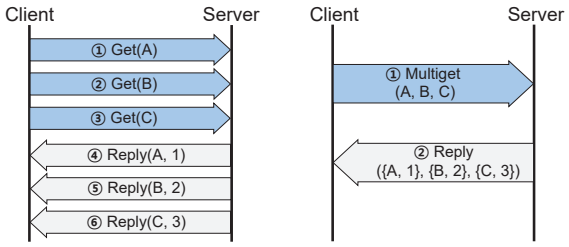
II. Background and Motivation

In this section, we first provide background on the multiget operation and its coordination overhead. We also motivate the necessity of a new multiget coordination architecture.

A. Multiget Operations in Key-Value Stores

Today's large-scale online services often need to access multiple specified keys to handle user requests. To access data efficiently, many key-value stores support the `multiget` operation, which batches read operations for multiple keys to get the values within a single request [1], [3], [5], [2]. Compared to using multiple `get` requests, the multiget operation offers better latency by reducing the number of messages to read many specified keys. `Multiget` is different from the `SCAN`, which returns key-value pairs that match a given pattern or range by searching a bulk of keys sequentially.

In Fig. 1, we illustrate an example to show the difference between get and multiget operations in a situation where a client reads 3 keys from a server. As shown in Fig. 1 (a), when we use get requests, the client should send 3 individual requests for each key. However, if the client uses the multiget operation as in Fig. 1 (b), it is enough to send a single request that contains all the requested keys. The server returns the values of the requested keys in a single reply as well.



(a) Get operation (b) Multiget operation

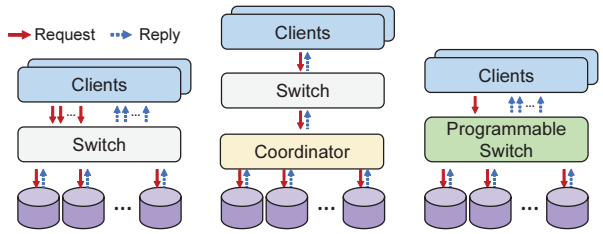
Fig. 1: An example of get and multiget operations. *With the multiget operation, the client can read the values of 3 keys using a single request, reducing client-side overhead and latency.*

B. Multiget Coordination Overhead

Multiget coordination. While the multiget is an efficient technique, it requires extra coordination when requested keys are distributed across multiple servers. The multiget coordination consists of *request splitting* and *reply aggregation*. Specifically, we should split the request into multiple sub-requests for different servers unless all requested keys are stored in the same storage server. Since each server returns the values of the requested keys in the same sub-request, we also have multiple sub-replies. Therefore, we should merge the sub-replies into a single reply. There are two typical multiget coordination architectures based on the location where the coordination is performed.

Client-based architecture (CliMC). In this architecture, the client application is responsible for both request splitting and reply aggregation. Some key-value stores including Redis [1], Memcached [2], and RocksDB [7] employ this architecture. While this approach is straightforward, it increases the complexity of the client-side logic and may cause excessive overhead because the client should generate multiple sub-request packets sequentially for a single multiget request. The operation is committed only if all sub-replies are aggregated into a single reply (i.e., all or nothing). Therefore, the time to coordinate multiget requests in the client highly impacts the multiget latency.

Coordinator-based architecture (CoordiMC). Several key-value stores like Cassandra [3], DynamoDB [4], and MongoDB [5] use a dedicated coordinator node to coordinate multiget operations. The client only needs to communicate with the coordinator, greatly reducing the client-side complexity. However, the coordinator-based architecture is not scalable. It easily becomes a bottleneck as the system load increases. This is because the coordinator relies on the CPU, which has limited performance and can handle only a few storage servers. Therefore, if the number of servers is larger than the capability of the coordinator, the overall performance is degraded. Building a coordination layer with multiple nodes can be an option, but this is not a cost-effective manner. Even under low load, latency increases since packets have to pass through the additional node.



(a) CliMC (b) CoordiMC (c) NetMC

Fig. 2: Different approaches for multiget coordination. *CliMC [1], [2], [7]: Client-based architecture; CoordiMC [3], [4], [5]: Coordinator-based architecture.*

C. Network-Accelerated Multiget Coordination

Key idea. We aim to build a scalable and high-performance multiget coordination architecture. The idea to achieve the goal is to divide the work of the multiget coordination between the network switch and the client. We offload request splitting to the switch while the client handles reply aggregation. The design rationale behind the idea is to consider the characteristics of coordination functions and their vantage place where each function can be performed best.

Why offloading request splitting? The network switch is a well-suited location for request splitting, a *stateless and I/O-intensive* function that groups keys and generates sub-request packets for different storage servers. The switch can process billions of packets per second with a deterministic processing delay of only a few hundred nanoseconds. This performance is significantly better than that of a CPU in a coordinator node, suggesting that scalable multiget coordination can be realized in a centralized manner. Furthermore, offloading the function to the switch reduces the client overhead and improves latency by calling the packet transmit function only once regardless of the multiget size.

Meanwhile, recent programmable switch ASICs like Intel Tofino [9], [10] enable us to realize our idea in practice. We can customize the packet processing logic in the switch data plane because the switch ASIC employs the Reconfigurable Match Table (RMT) switch architecture [11]. We can parse the packet header up to the application level and manipulate the header fields. A packet is processed by passing through several match-action stages where custom computation and memory access occur.

Why not offloading reply aggregation? We let clients handle reply aggregation as usual instead of offloading it to the switch due to the limitations of programmable switch ASICs. Aggregating replies requires maintaining soft states with variable-length key-value pairs for each multiget request until the reply is committed. However, implementing variable-length reply aggregation on the switch is non-trivial because of strict constraints on computational capability and memory access: the switch can process only a few bytes of data per stage, and each stage has only a few megabytes of static memory. Therefore, we place the reply aggregation function on the client, where *stateful* operations can be performed more efficiently than in the switch data plane.

	CliMC [1]	CoordiMC [3]	NetMC
Coordination point	Client	Coordinator	Switch/Client
Client overhead	High	Low	Medium
High throughput	√	×	√
Low latency	×	×	√
Scalability	√	×	√

TABLE I: Comparison to existing architectures.

Technical challenges. Realizing switch-based request splitting is challenging due to the strict hardware constraints. For example, the switch does not support loops and modulo operations for a random number. Instead of proposing approximate in-switch techniques that might limit application semantics [12], [13], [14], [15], we make the switch logic as simple as possible through the client-side assistance that puts metadata as hints into the custom packet header. With the hints and built-in features like recirculation and cloning, we realize the stateless in-switch request splitting.

What are the differences? Fig. 2 and Table I compare NetMC with the existing architectures. CliMC achieves high throughput and scalability but suffers from high latency due to high client overhead. CoordiMC makes the client overhead low but offers limited performance because the coordinator node has limited scalability. It also increases latency since requests should go through the additional node. NetMC balances the existing approaches by placing the request splitting function in the switch. Thanks to reduced client overhead, NetMC can achieve low latency. Furthermore, we can achieve high throughput and scalability since the switch has high packet processing throughput.

III. NetMC Design

In this section, we describe the design of NetMC. We first describe the NetMC architecture and the packet format. After that, we explain how the switch processes NetMC packets in detail. We also discuss several issues related to our design.

A. NetMC Architecture

In Fig. 3, we illustrate the NetMC architecture comprising the switch data plane, clients, and storage servers.

Switch data plane. The switch data plane is responsible for request splitting, which is the core of the multiget coordination. The programmable switch provides standard L2/L3 routing functions like a traditional fixed-function switch. Our custom modules are as follows.

- *Key group identification*: This module identifies the current key group and checks if there are more key groups to process.
- *Request addressing*: This module assigns the IP address of the target storage server of the current key group.
- *Sub-request generation*: This module generates a sub-request using packet cloning and recirculation. The original packet is forwarded to the target server as a sub-request. The cloned packet visits the switch data plane

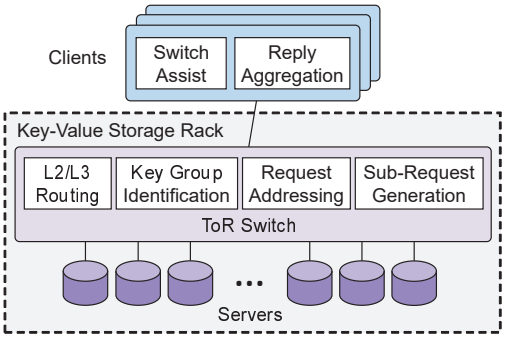


Fig. 3: NetMC architecture. *The switch data plane performs requests splitting with custom modules. Clients handle reply aggregation while supporting the switch with metadata generation. Note that clients do not necessarily have to be in the same rack as the storage servers.*

again via the recirculation port so that the switch can generate other sub-requests.

Clients and servers. In the NetMC architecture, the client basically generates multiget requests and receives replies. We have two custom modules as follows.

- *Switch assist*: This module puts extra metadata into the packet header to help the request splitting process in the switch data plane.
- *Reply aggregation*: This module aggregates sub-replies from storage servers and merges the sub-replies into a single multiget reply.

Meanwhile, storage servers run a lightweight application that acts as a shim layer. When the server application receives a message, it translates the message to necessary API calls for the underlying key-value store and vice versa.

B. Packet Format

Fig. 4 shows the packet header format of NetMC. We basically handle messages using UDP for better latency similar to existing works [16], [17], [18], [19]. The NetMC header is encapsulated as the L4 payload of the packet. This allows NetMC to integrate with existing network protocols while implementing our custom protocol. To distinguish NetMC packets from normal packets, NetMC reserves an L4 port. The switch parses the NetMC header and applies our custom processing logic only if the L4 port number is matched. The NetMC header consists of the following fields.

- `OP` (1 B): the request operation type, which can be `MGET` (multiget request), `GET` (get request), `PUT` (write request), `MGET_REPLY` (multiget reply), `GET_REPLY` (get reply), and `PUT_REPLY` (write reply).
- `ID` (4 B): the multiget request ID shared by sub-requests and sub-replies, used for reply aggregation.
- `OID` (1 B): the order of key groups in a multiget request. The value of the field increases with each sub-request generation.

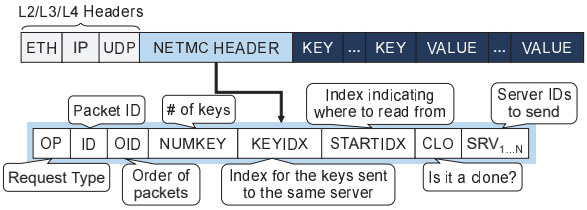


Fig. 4: The NetMC packet format.

Algorithm 1 Multiget Request Generation at Clients

– pkt: Packet to be processed
– seq: Sequence number for multigets
– n: Number of servers
– K: Set of requested keys
– k_i: ith requested key in K
– S: Array of server IDs for each key

1: $pkt.op \leftarrow$ `MGET` ▷ Set request type to multiget
2: $pkt.id \leftarrow seq$++ ▷ Assign a unique request ID
3: $pkt.numkey \leftarrow |K|$ ▷ Set the total number of requested keys
4: $S[i] \leftarrow \text{HASH}(k_i)\%n, \forall i$ ▷ Set dest. server for all keys
5: Sort(S) ▷ Sort server IDs in ascending order
6: Set ith bit to 1 in $pkt.keyidx$ if $S_i \neq S_{i+1}$
7: Sort(K) by S ▷ Sort keys by target servers in S
8: $pkt.key_i \leftarrow k_i, \forall i$ ▷ Put requested keys to packet
9: $pkt.srv_k \leftarrow S_{\text{unique},j}, \forall j, j \leq |S|$ ▷ Put server ID of key groups
10: FORWARD(pkt) ▷ Send the packet to the switch

- `NUMKEY` (1 B): the number of requested keys in a multiget request. The value of this field is updated every sub-request generation.
- `KEYIDX` (2 B): the bitmap index indicating which keys belong to the same group. This acts as a separator for the key group identification process. The value changes through bit shift operations with each sub-request generation.
- `STARTIDX` (1 B): the index of the first key in the header from which to start reading for storage servers.
- `CLO` (1 B): the field indicating whether the packet was cloned: 0 means not cloned, and 1 means cloned.
- `SRV` (n B): the array of destination storage server ID for key groups, which indicates where the sub-request should be forwarded. There are n 1-B `SRV` fields where n is the number of storage servers in a rack.

C. Request Generation

In NetMC, generating requests is similar to existing architectures. However, for custom packet processing in the switch, clients put the required metadata into the NetMC header fields, which include `OP`, `ID`, `NUMKEY`, and `KEYIDX`.

Algorithm 1 presents the pseudocode of multiget request generation. The client sets the operation type to `MGET` (line 1). Next, the client puts the linearly-increasing sequence number to `ID` as a unique multiget request ID (line 2). This ID is used for reply aggregation in the client. The `NUMKEY` field is also updated with the number of requested keys (line 3). The client then calculates the server ID for the requested keys using a hash function and modulo operations (line 4). Next, the client sorts the server ID array in ascending order (line 5). Using

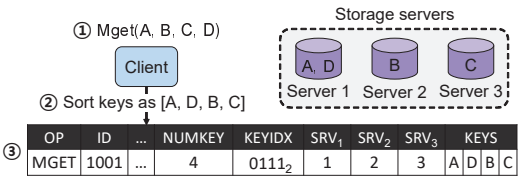


Fig. 5: Example of multiget request generation. ① *The client generates a multiget request for keys {A,B,C,D}.* ② *Server 1 stores keys A and D, server 2 stores key B, and server 3 has key C.* ③ *The client then sorts the keys into {A,D,B,C} and puts the metadata in the message header.*

this, the client constructs a bitmap called KEYIDX by setting ith bit to 1 if S_i and S_{i+1} are different (line 6). The 1 in the bitmap acts as a separator between key groups when the switch splits the request. After that, the client sorts the requested key array (line 7), which clusters keys belonging to the same storage server together. This clustering allows the switch to generate sub-requests more efficiently. The requested keys are then placed into the packet header (line 8). The client puts the server ID of key groups into the SRV fields by extracting unique values from S (line 9). For example, assume that we have 32 servers, but the requested keys are for server 17 and server 8 only. Then, the client puts 8 and 17 for the first and second fields in 32 SRV fields. Note that we set the number of SRV fields statically to simplify the switch logic.

Example. Fig. 5 shows an example of request generation at clients. In this example, we consider a client who wants to retrieve the values of keys {A,B,C,D} that are distributed across 3 storage servers. Server 1 stores keys A and D, server 2 stores key B, and server 3 stores key C. The client generates a multiget request with $pkt.op$ = MGET and $pkt.numkey = 4$ since we have 4 keys to get. We assume that this is the 1001st multiget request by setting $pkt.id = 1001$. This ID will be used for reply aggregation. The client sorts the keys in the order the servers store them, resulting in {A,D,B,C}. The client puts $pkt.keyidx = 0111_2$ based on the sorted key list. Here, each bit corresponds to keys A, D, B, and C, respectively. Next, the client puts the destination server ID of key group i into the $pkt.srv_i$ field using the hash function, resulting in 1, 2, and 3, respectively.

D. Request Splitting in the Switch

The request splitting process consists of three steps: key group identification, request addressing, and sub-request generation.

Key group identification. The first step in performing request splitting is to group keys that are stored in the same storage server. This is because each sub-request should contain the keys for the corresponding key group. This typically requires repetitive operations on multiple keys in a multiget request. However, the current programmable switch ASIC does not support loops to guarantee a deterministic packet processing delay. We may leverage packet recirculation to group keys, but this may significantly reduce throughput as the number of recirculations depends on the number of requested keys.

Algorithm 2 Request Splitting in Switch Data Plane

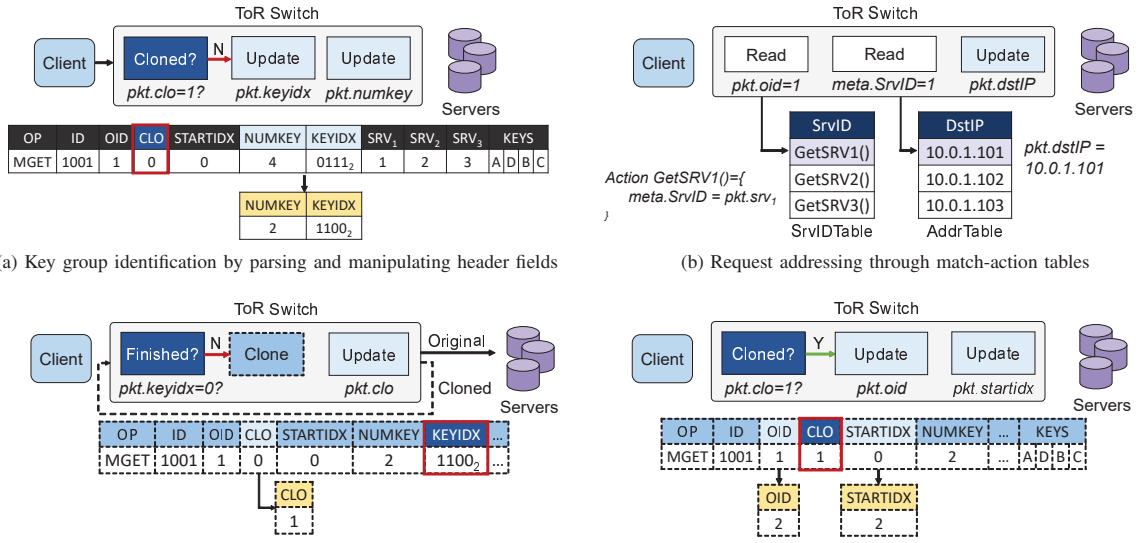
```
      – pkt: Packet to be processed
      – AddrTable: Server IP address table
      – SrvIDTable: Server ID table
      – n: Number of keys in the current key group
 1: if pkt.clo == 1 then                    ▷ Cloned packet (continued splitting)
 2:     pkt.oid += 1                        ▷ Update the order of key group
 3:     pkt.startidx += pkt.numkey          ▷ Point to next key group
 4: end if
 5: n ← COMPARE(pkt.keyidx)                 ▷ Get # of keys in current group
 6: pkt.keyidx ← pkt.keyidx ≪ n             ▷ Mark the cur. group as done
 7: pkt.numkey ← n                          ▷ Put the number of keys in the cur. group
 8: pkt.dstIP ← AddrTable[pkt.srv[SrvIDTable[pkt.oid]]]
 9: if pkt.keyidx > 0 then                  ▷ More key groups to split?
10:     CLONE(pkt)                         ▷ Forward sub-request and continue splitting
11: else                                    ▷ No more key groups to process
12:     FORWARD(pkt)                       ▷ Forward the last sub-request packet
13: end if
```

We address these challenges by making the client perform preprocessing to simplify the switch logic. Specifically, the client groups keys by sorting the keys based on the target storage server and puts the clustered keys into the message header. In addition, the client puts the KEYIDX field into the header to provide a hint to the switch. This field encodes the grouping information as a bitmap, where each bit represents a specific key in a multiget request. The client sets a bit to 1, a separator between key groups. With the grouped keys and the bitmap, the task of the switch is simplified to identify key groups. For example, suppose a multiget request for keys $\{A, B, C, D\}$ where keys (A, D) are stored on server 1, and keys (B, C) are stored on server 2. The client groups keys as $\{(A, D), (B, C)\}$ and constructs the KEYIDX field as 01010000_2 with assumption of 8-bit KEYIDX.

The switch identifies the number of keys in the current key group by comparing the KEYIDX field with predefined constants representing the number of keys in a key group in order. We have n constants where n is the length of the KEYIDX field in bits. For example, the switch knows the number of keys for the first key group of keys (A, D) as 2 by comparing $01010000_2 = 80$ and $01000000_2 = 64$. The switch puts the number of keys into the NUMKEY field to notify the server of the number of requested keys. To prepare for processing the next key group, the switch performs a bitwise left shift operation on the KEYIDX field by NUMKEY. In the example, 01010000_2 becomes to 01000000_2.

Request addressing. The second step of in-switch request splitting is to get the IP address of the storage server to which the current sub-request should be forwarded. A challenge arises because we consider hash-partitioned storage where we need to perform a modulo operation using the number of storage servers. Unfortunately, the switch can efficiently handle modulo operations for a power of two but cannot perform operations with arbitrary numbers. This is problematic since the number of storage servers in practice may not be a power of two.

To address this challenge, we let the client precompute



(a) Key group identification by parsing and manipulating header fields

(b) Request addressing through match-action tables

(c) Sub-request generation using cloning and recirculation

(d) Updating header fields of the cloned packet for the next key group

Fig. 6: Example of request splitting in switch data plane. *The switch data plane sequentially performs (a) key group identification, (b) request addressing, and (c) sub-request generation. If there are remaining key groups to split, the switch clones the packet so that the original packet is forwarded to the storage server as a sub-request and the cloned packet is recirculated back to the ingress pipeline for further processing. (d) For the recirculated clone packet, the header fields are updated to prepare the splitting process for the next key group.*

the target server ID and put it in the message header, while the switch only maintains the mapping between the server ID and the IP address. Specifically, the client gets the target server ID for key groups using a hash function with a modulo operation when generating a multiget request. The client stores the precomputed IDs in the `SRV` fields of the message header. We have $\texttt{SRV}_1, \texttt{SRV}_2, ..., \texttt{SRV}_n$ fields in the header where n is the number of storage servers in a storage rack, which can be configured statically. The switch has two match-action tables to translate the server ID to the IP address. The first match-action table extracts the server ID as metadata from the corresponding `SRV` field. The `OID` field is a match key for the table. Recall that the `OID` denotes the order of the current sub-request (i.e., key group) in the multiget request. For example, if `OID` = 1, the switch refers to $\texttt{SRV}_1$ and the value in $\texttt{SRV}_1$ is extracted as metadata. The switch then looks up the extracted server ID using the second match-action table, which maps the server ID to the corresponding IP address. With the table, the packet finally gets the appropriate destination IP address.

Sub-request generation. After request addressing, the switch now generates sub-requests for each key group. A challenge here is that the switch ASIC does not support a function that splits a single packet into multiple packets. To implement a request splitting function in the switch data plane, we leverage two built-in features of the switch ASIC: packet recirculation and packet cloning. Packet recirculation allows the packet to revisit the ingress pipeline by passing through an internal recirculation port. The switch can generate a copy of the current packet using packet cloning. The cost of cloning is small because the switch has the Packet Replication Engine (PRE), a special hardware module that copies the packet descriptor only, not the whole packet. There are two packet cloning options in the programmable switch ASIC: multicast and packet mirroring. In NetMC, we use the multicast since it is easier to configure.

To generate sub-requests, the switch clones the packet at the end of the ingress pipeline. Specifically, to clone the packet, the switch specifies the multicast group ID as metadata when the cloning function is called. The multicast group ID is predefined by the switch control plane along with the two output port numbers to which the original and cloned packets should be forwarded. The original packet (i.e., the sub-request) is forwarded to the output port directed to the target storage server, and the cloned packet goes through the recirculation port to revisit the ingress pipeline. As the recirculated packet re-enters the ingress pipeline, the switch updates the `OID` and `STARTIDX` fields as explained in Algorithm 2. The packet then undergoes key group identification, addressing, and cloning again. This process is continued until no key groups are remaining in the current packet to split (i.e., `KEYIDX` = 0).

Ideally, each sub-request packet should contain only the keys of the corresponding key group. However, the current design does not pop the requested keys of the current key group when generating a sub-request. Therefore, the storage server receives the request packet with all requested keys. This is because the switch data plane should parse the entire packet header, including the key fields, but the switch has limited byte depth to parse. If we parse the key fields, this may limit the number of keys to request. We let the sub-request packet contain all keys to avoid such limits on application semantics. Instead, we manipulate the `NUMKEY` and `STARTIDX` fields to

make the storage server parse the key fields correctly. Note that the request packet does not have value fields, and the replies for sub-requests contain only key and value fields for the corresponding key group.

Pseudocode. Algorithm 2 is the pseudocode for request splitting in the switch data plane. Upon receiving a packet with $pkt.op$ = MGET, the switch checks the CLO field to determine whether the packet has been previously cloned (line 1). The cloned packet means that the current packet has been recirculated because we have remaining sub-requests to generate. If it is, the switch increases the OID field by 1 (line 2) and the STARTIDX field by the value of NUMKEY field (line 3). OID is used for indexing the SRV fields to get the server ID of the current key group. STARTIDX refers to the index of the first key of the matched key group between requested keys by storage servers. Regardless of cloning, the switch gets the number of keys in the current key group n by comparing KEYIDEX and n (line 5). For example, if KEYIDX is 4 bits and greater than 8 (indicating the most significant bit is set), the current key group contains only one key to request. The switch identifies the key group destined for the same server by using a bitwise left shift operation on the KEYIDX field to locate the position of the first 1 bit, which marks the end of the current key group (line 6). The switch then sets the NUMKEY field to the number of keys in this group (line 7). Using two match-action tables to look up the server ID and the IP address, the switch sets the destination IP address (line 8). Next, the switch checks whether KEYIDX is positive (line 9). If it is, the switch clones the packet because we have remaining key groups to process (line 10). The original packet (i.e., the sub-request of the current key group) is forwarded to the storage server. The cloned packet (i.e., the continued request for remaining key groups) passes through the recirculation port to revisit the ingress pipeline. The switch sets CLO to 1 for the cloned packet. Otherwise, the switch simply forwards the packet because the current packet is the last sub-request of the multiget request. (lines 11-12).

Example. Fig. 6 illustrates the example of the request splitting process in the switch data plane. We consider a multiget request in Fig. 5. Upon receiving the request packet, the switch checks the OP field and identifies it as a multiget request. Fig. 6 (a) shows the key group identification process. If $pkt.op$ = MGET, the switch examines the CLO field, which is initially 0, indicating that the packet is the multiget request never been split yet. Here, $pkt.clo$ = 0, hence the switch updates the KEYIDX and NUMKEY fields as follows. Using the bitmap in KEYIDX, the switch gets the number of keys in the first key group by identifying that the second bit is 1, indicating that the key group consists of two keys, A and D. The switch then performs a bitwise left shift operation by 2 on KEYIDX, resulting in 1100_2. Next, it updates NUMKEY from 4 to 2.

Fig. 6 (b) shows how the switch assigns the destination IP address to the current sub-request. The switch first refers to a match-action table SrvIDTable using the OID field as the match key. SrvIDTable returns the server ID metadata SrvID by referring to the SRV_{OID} field. Specifically, the switch executes an action that assigns $pkt.srv_{oid}$ to metadata

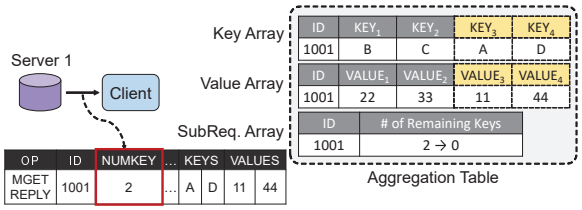


Fig. 7: Example of reply aggregation. *The client maintains the aggregation table. Upon receiving a reply, the client appends key-value pairs to the corresponding slot and decreases the number of remaining keys by NUMKEY. When it becomes zero, the client commits the request.*

$meta.SrvID$. In this example, we get $meta.SrvID$ = 1 since $pkt.srv_{oid}$ = 1. The switch then looks up another match-action table AddrTable using $meta.SrvID$ as the match key. As $meta.SrvID$ = 1, the switch updates $pkt.dstIP$ = 10.0.1.101.

Fig. 6 (c) shows how the switch generates sub-requests. In this example, since $pkt.keyidx = 1100_2$, the switch clones the packet to generate the sub-request for the current key group and to continue sub-request generation for the next key group. By assigning the multicast group ID, the packet is cloned. The original packet is sent to server 1 with 10.0.1.101 as the sub-request. The cloned packet is recirculated back into the ingress pipeline for further processing. Meanwhile, the switch sets the CLO field to 1 before cloning. For the recirculated clone, the switch updates related header fields as shown in Fig. 6 (d). The switch increments the STARTIDX field by the number of keys already processed, updating it to 2, and increments the OID to 2. The switch then continues processing from key group identification.

E. Request Processing in the Storage Server

Upon receiving a sub-request, the server application extracts the requested keys of the corresponding key group by reading NUMKEY key fields starting from the index denoted in the STARTIDX field. After that, the server returns the values of the requested keys by translating the NetMC message to the multiget API call of the underlying key-value stores and vice versa (e.g., MGET in Redis [1] and getMulti in Memcached [2]). The returned sub-reply message with $pkt.op$ = MGET_REPLY only contains the key-value pairs of the corresponding key group in the message header.

F. Reply Aggregation at Clients

The client maintains the number of remaining keys and aggregated key-value pairs for each multiget request in the aggregation table indexed by the ID field (i.e., multiget request ID). The aggregation table consists of multiple arrays for keys and values, as well as the remaining number of keys. Upon receiving a sub-reply, the client appends the contained key-value pairs to the aggregation table and decreases the number of remaining keys by NUMKEY. If the remaining keys are zero, the client commits the reply by finishing the aggregation process.

Example. Fig. 7 depicts an example of reply aggregation at clients. We share the scenario of Fig. 6. Here, we assume that the client already keeps key-value pairs for keys B and C. Now, the reply with key-value pairs for keys A and D arrives. The client appends them alongside the key-value pairs for keys B and C. Next, the client updates the number of remaining keys from 2 to 0. Since every key-value pair is aggregated, the client finally commits the request.

G. Discussion

Multi-packet requests. We consider a single-packet multiget request, but the request may span multiple packets if the sum of key size exceeds the MTU size. To implement request splitting in the switch for multi-packet requests directly, the switch logic should be stateful and complex because we need to maintain the required metadata like `OID`, `STARTIDX`, and `KEYIDX` in the switch memory for coordination between multiple packets. Therefore, to avoid such complexity, we let clients generate multiple single-packet multiget requests. This can handle many cases easily without extensively changing the switch logic.

Multi-rack deployment. NetMC can scale out to multiple racks as follows. In multi-rack environments, we consider that only ToR switches perform request splitting, not spine switches. We assume that the ToR switch connected to storage servers performs request splitting, not the client-side ToR switch. This is because if we split requests in the client-side ToR switch, the overhead of managing storage server information in the packet header may increase excessively. We should put the switch ID into the packet header to prevent requests from splitting by the ToR switch connected to the client. The ToR switch maintains its ID in a register and only applies the request splitting when its ID and that in the request packet are matched. One assumption is that each storage rack should contain the servers that store requested keys. To handle cases where requested keys are located in different racks, we should let clients generate multiple requests for different racks by referring to the rack ID of keys, which can be obtained using hash functions.

IV. Implementation

Switch data plane and controller. The switch data plane is written in $P4_{16}$ [20]. The data plane is compiled to Intel Tofino [9] using Intel P4 Studio SDE 9.7.0. Our prototype is lightweight because the switch only performs stateless request splitting and does not maintain any state. Specifically, the prototype uses 5 match-action stages, 1.75% SRAM, 2.50% Match Input Crossbar, and 2.88% Hash Bits of the ASIC resources. We do not consume any stateful ALU. Meanwhile, our controller in the switch control plane is written in Python 3.9.12. The controller configures the switch and manages the rules of the match-action tables, which include tables related to request splitting and the traditional L2/L3 packet forward table.

Client-server applications. We implement an open-loop client-server application in C. We use the NVIDIA Messaging Accelerator (VMA) library [21] to reduce the packet processing delay of clients, the coordinator, and storage servers. VMA provides low latency by intercepting the socket function calls and translating them into native RDMA verbs, allowing host packet processing in user space while bypassing the kernel network stack. The client application is basically responsible for generating requests and throughput/latency measurement. For each architecture, the client application performs extra work. The time gap between consecutive requests follows an exponential distribution. As a shim layer, the server application processes incoming requests based on the request type and returns replies to the client by retrieving the values of requested keys. We use multiple worker threads where each worker is pinned to a disjoint CPU core. We implement an in-memory key-value store using TommyDS [22], a high-performance hash table library used in an existing work [12]. We also use Redis [1] to demonstrate the efficiency of NetMC with real-world applications.

V. Evaluation

A. Methodology

Testbed setup. We build a testbed that consists of 8 nodes. The nodes are interconnected via an APS Networks BF6064X-T switch with Intel Tofino ASIC. Each node has a 10-core (Intel i5-12600K CPU @ 3.7 GHz), 32 GB of DDR5 memory, and a 100GbE NVIDIA CX-5 NIC. The nodes run Ubuntu 22.04 LTS with Linux kernel 6.8.0. The two nodes act as clients. Another node is for the coordinator node The remaining 5 nodes are designated as storage servers.

Compared schemes. We compare NetMC with CliMC [1], [2], [7] and CoordiMC [3], [4], [5]. CliMC indicates a client-based multiget coordination architecture where the client handles both request splitting and reply aggregation. CoordiMC refers to a coordinator-based architecture that leverages a CPU-based coordinator node to perform request splitting and reply aggregation.

Workloads. We consider a default workload that follows the features observed in production workloads [8] and well-known YCSB benchmark workloads [23]. The key distribution follows Zipfian distributions with skewness parameters $\alpha = 0.9, 0.95, 0.99$. For most experiments, we use Zipf-0.99, which is commonly used to model a highly-skewed key access pattern in many production workloads [24], [25]. The number of keys for each multiget request (i.e., the multiget size) follows a heavy-tailed distribution of a production workload where the average size is 8.6 keys [8]. The workload has a write ratio of 5% that reflects a common workload where read requests dominate [14], [13]. We set the key size to 4 bytes and the value size to 256 bytes, considering that most keys are tiny and values are small [26], [27]. Each key-value pair is hash-partitioned across storage servers.

B. Main Results

In this subsection, we present experimental results in the order of throughput, latency, and scalability to show that NetMC is the only solution that offers high throughput, low latency, and scalability simultaneously.

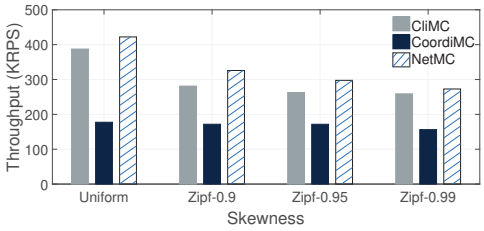
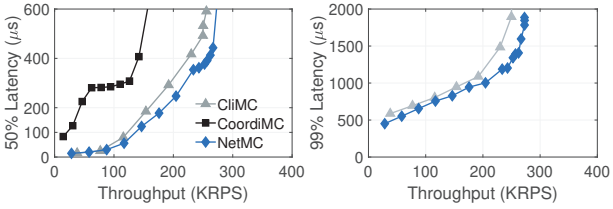
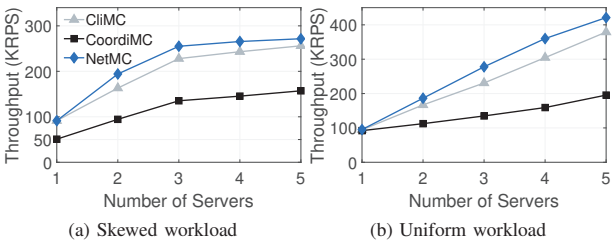


Fig. 8: Throughput vs. skewness.



(a) Median latency (b) 99th percentile latency

Fig. 9: Latency vs. throughput.



(a) Skewed workload (b) Uniform workload

Fig. 10: Scalability results with the various workloads.

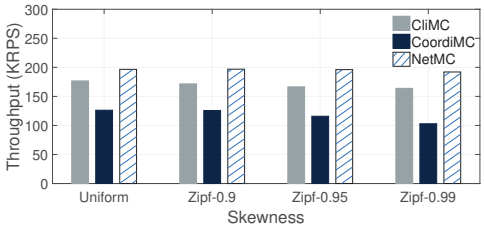


Fig. 11: [Redis] Throughput vs. skewness.

Throughput. We evaluate whether NetMC can achieve the best throughput. We measure the throughput of each scheme under a uniform workload and skewed workloads with $\alpha = 0.9, 0.95$, and 0.99. Fig. 8 shows the results. Across all solutions, performance generally degrades as skewness increases. However, NetMC consistently delivers the highest throughput for all distributions by reducing client-side overhead through request splitting in the switch. NetMC beats CliMC and CoordiMC by 1.11× and 1.94× on average, respectively. CliMC outperforms CoordiMC, but it still falls short of NetMC. Meanwhile, CoordiMC shows the worst throughput. This is because the coordinator node does not have enough capability to handle multiple storage servers, resulting in limited performance. These results demonstrate that NetMC is resilient to skewness in key distributions.

Latency. We measure the median and tail latencies by varying the Tx throughput to demonstrate that NetMC provides low latency while supporting high throughput.

Fig. 9 shows the median and 99th percentile latencies as a function of Rx throughput. Note that Fig. 9 (b) omits the result for CoordiMC, as its latency exceeds the Y-axis limit. NetMC consistently achieves the lowest latency across most throughput levels, with the performance gap between NetMC and the other schemes widening as throughput increases. For example, when the throughput is around 256 KRPS, NetMC offers lower tail latency than CliMC by 0.60×. This gap stems from that NetMC always sends a single multiget request regardless of system load, thanks to switch-based request splitting. In contrast, CliMC suffers from client-side overhead as throughput increases, due to the need for splitting and aggregating a large number of requests/replies. CoordiMC shows the highest latency even at low system load, as requests and replies must pass through a coordinator node.

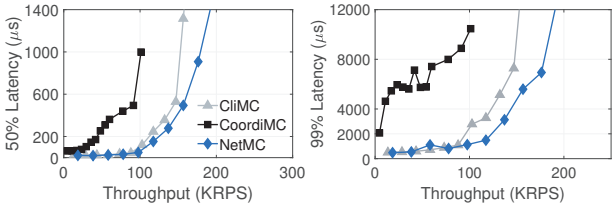
Scalability. We now evaluate the scalability of NetMC against CliMC and CoordiMC to see whether NetMC can scale to multiple storage servers. In this experiment, we measure the saturated throughput achieved by each solution by varying the number of storage servers. We also perform experiments for both skewed (Zipf-0.99) and uniform workloads to clarify the impact of key distributions on the scalability results.

Fig. 10 (a) plots the throughput and the 99th percentile latency results with the skewed workload. We can see that NetMC provides the best throughput regardless of the number of servers. CliMC shows lower performance than NetMC because of the client-side overhead for request splitting. The maximum gap between NetMC and CliMC is 1.19× when we use two storage servers. NetMC achieves better throughput than CoordiMC by 1.86× on average since we leverage the high-performance switch as a multiget coordinator, which performs better than the coordinator node of CoordiMC. Meanwhile, we see that every solution does not provide linearly increasing throughput. This is because of the skewness in key distributions.

Fig. 10 (b) shows the scalability results with the uniform workload. We can observe that the throughput of NetMC increases linearly. CliMC also has scalable performance but is worse than NetMC due to the client-side overhead in request splitting. These results demonstrate that NetMC provides high throughput and scalability simultaneously.

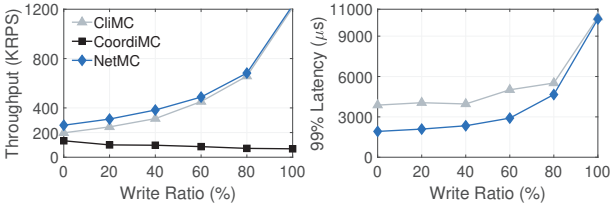
C. Performance with Redis

We now evaluate the performance with Redis [1] to demonstrate that NetMC can work with real-world applications. Redis is a widely adopted in-memory key-value store for many production services. In this experiment, upon receiving a request, the server gets the values of the specified keys using `MGET`, a multiget command in Redis. We first measure the throughput using different workload skewnesses. Next, we measure the median and the 99th percentile latencies.



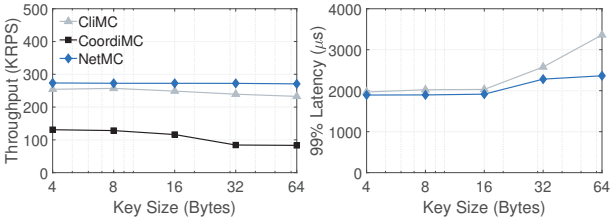
(a) Median latency (b) 99th percentile latency

Fig. 12: [Redis] Latency vs. throughput



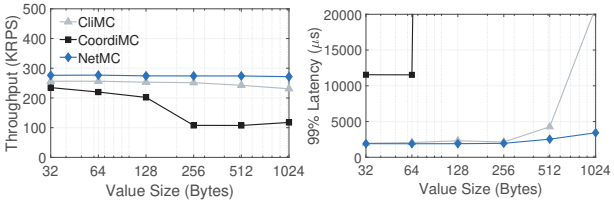
(a) Throughput (b) 99th percentile latency

Fig. 13: Impact of write ratio.



(a) Throughput (b) 99th percentile latency

Fig. 14: Impact of key size.



(a) Throughput (b) 99th percentile latency

Fig. 15: Impact of value size.

Throughput. Fig. 11 shows the throughput of different solutions with different workloads. Similar to Fig. 8, we can see that NetMC achieves the best performance for all the distributions. NetMC performs better than CliMC by 1.15× on average. Compared to CoordiMC, NetMC achieves better throughput by 1.67× on average.

Latency. Fig. 12 depicts how the median and 99th percentile latencies change as throughput grows. We can clearly see that NetMC achieves lower latency than the other schemes, especially from moderate system loads. The latency gap between NetMC and CliMC becomes larger as throughput increases. For example, CliMC shows higher tail latency than NetMC by 30.06× for the throughput of around 176 KRPS. These results demonstrate that NetMC is effective with real-world key-value store applications.

D. Deep Dive

Impact of write ratio. We examine the impact of the write ratio on the throughput and latency since real-world workloads often involve a mix of read and write requests. In this experiment, clients measure the saturated throughput by generating requests at different write ratios, ranging from 0% to 100% in 20% increments. We also measure the 99th percentile latency at the near-saturated throughput of NetMC.

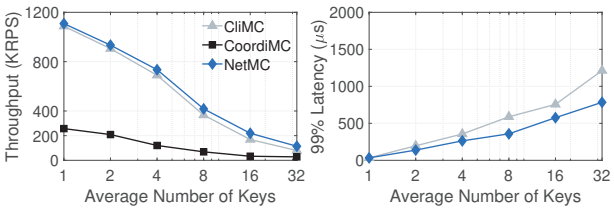
Fig. 13 (a) shows the throughput with various write ratios. Our observations are as follows. First, as the write ratio increases, the throughput of NetMC and CliMC increases. This is because each write request contains only a single key-value pair, unlike a read request (i.e., multiget request) that contains multiple key-value pairs. Therefore, writes do not require request splitting and reply aggregation, causing no extra coordination overhead. Second, the performance gap between NetMC and CliMC becomes narrow as the write ratio increases. For example, NetMC is better than CliMC by 1.31× for the read-only workload. However, for the workload with 60% writes, the gap is only 1.08×. This is because the portion of requests that experience switch-based request splitting decreases as the write ratio increases. Meanwhile, CoordiMC still performs the worst due to the limited performance of the coordinator node.

Fig. 13 (b) shows the tail latency result. We omit the result of CoordiMC since its result is beyond the Y-axis limit. Latency gaps exist between NetMC and CliMC at low write ratios since the client of CliMC causes more overhead to handle requests. However, similar to the throughput result, the difference becomes small as the write ratio increases.

Impact of key size. We evaluate the impact of key size on the performance of NetMC. To do this, we measure the throughput by varying the key size. We also measure the 99th percentile latency. We consider the key sizes ranging from 4 bytes to 64 bytes. Note that most keys are generally tens of bytes in production workloads [26].

Fig. 14 (a) shows throughput as the key size increases. NetMC consistently achieves the highest throughput across all the key sizes. Unlike NetMC, CoordiMC and CliMC result in slightly degraded performance when the key size increases. This is because the multiget coordination overhead increases as the key size increases. Fig. 14 (b) presents the tail latency result. CoordiMC is omitted since its result is beyond the Y-axis limit. We can see that the latency gap increases as the key size becomes larger. This is because, similar to Fig. 14 (a), one factor of the coordination overhead is the key size.

Impact of value size. We conduct experiments with different value sizes similar to the key size experiment. We consider the value size up to 1024 bytes as most values are hundreds of bytes [26], [27]. Furthermore, 1024-byte is the typical value



(a) Throughput (b) 99th percentile latency

Fig. 16: Impact of multiget size.

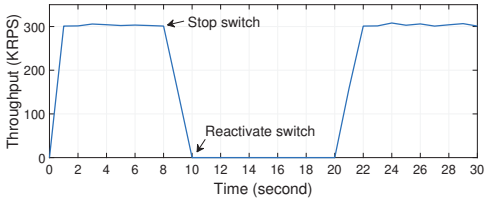


Fig. 17: Throughput under switch failures.

size used in many existing works [19], [24], [28], [29], [30], [31].

In Fig. 15 (a) and (b), the trend is similar to what is observed in Fig. 14 (a) and (b). NetMC consistently delivers the best performance across all value sizes. In contrast, both CliMC and CoordiMC show a decline in throughput as the value size increases, which can be attributed to the added client-side coordination overhead. Additionally, the latency gap between NetMC and CliMC widens with increasing value size. This is because larger values require the client to process more data per request and response, resulting in longer processing times and, consequently, higher latency and degraded performance. These findings indicate that NetMC is robust to changes in value size.

Impact of multiget size. In this experiment, we examine how the performance changes as the number of keys in a single multiget request grows. We measure the throughput and the 99th percentile latency by varying the average number of keys in a multiget request. Like other experiments, the key skewness in the workload follows that of the production workload [8].

Fig. 16 (a) shows the throughput result. We see that throughput decreases for all three schemes as the number of keys increases. This is because the overhead of request splitting and reply aggregation increases as the number of keys per request increases. Despite this drop, NetMC maintains higher throughput than CliMC and CoordiMC. The average gaps between NetMC against CliMC and CoordiMC are 1.17× and 5.28×, respectively. The gap between NetMC and CliMC increases as the multiget size grows. When the average number of keys is 2, the gap is only 1.03×. However, with the 32 keys on average, NetMC is better than CliMC by 1.44×.

Fig. 16 (b) shows how the tail latency changes. The result of CoordiMC is omitted since it is beyond the Y-axis limit. We can see latency gaps between NetMC and CliMC grow as the multiget size increases. This is because NetMC offloads request splitting to the network switch. When the average multiget size is 32, CliMC is slower than NetMC by 1.55×. The experiments demonstrate that NetMC is robust to workload dynamics in terms of the multiget size.

Performance under switch failures. We evaluate the performance of NetMC during a switch failure scenario. In this experiment, we intentionally stop and then reactivate the switch to emulate a switch failure.

Fig. 17 shows the change in throughput under the switch failure. We can see that the NetMC throughput drops sharply when the switch is stopped at 8 seconds. The switch was reactivated at 10 seconds, and throughput gradually returned to its original level after 10 seconds of the switch being turned back on. The downtime relies on switch hardware, not the NetMC mechanism. Since the switch data plane in NetMC is stateless, we have no issues with consistency. This demonstrates that NetMC is robust to switch failures and can quickly recover throughput.

VI. Related Work

We briefly review existing works related to NetMC.

Multiget requests. Several works have been proposed for high-performance multiget operations. ELFJ [32] is an approximation algorithm that optimizes the distribution of multiget requests when some requested keys are replicated over multiple storage servers. This can be leveraged as a theoretical foundation for the key grouping of our work when we consider replicated storage. Rein [8] is a multiget scheduler that prioritizes requests with smaller execution times. TailX [33] improves Rein by considering the load of storage servers and the expected execution times at the same time. These scheduling works are orthogonal to NetMC since the problem space differs. NetMC aims at reducing the multiget coordination overhead.

In-network acceleration for key-value stores. Recent works have explored the potential of programmable switches to accelerate key-value stores [12], [13], [34], [35], [14], [36]. NetMC is in line with existing works since we also try to accelerate key-value stores using programmable switches. However, NetMC is different from them by considering multiget operations, which no existing work has addressed.

VII. Conclusion

This paper presented NetMC, a network-accelerated multiget coordination architecture that provides high throughput, low latency, and scalability simultaneously. The key idea of NetMC is to offload stateless and I/O-intensive request splitting to the programmable switch while handling stateful reply aggregation at the client. We addressed technical challenges caused by strict hardware constraints in designing a custom switch data plane through client-side assistance. Our experimental results demonstrated that NetMC outperforms existing client- and coordinator-based architectures.

Acknowledgement

This research was sponsored by the National Research Foundation of Korea (NRF) grants funded by the Ministry of Science and ICT (No. RS-2025-00522990). Gyuyeong Kim is the corresponding author.

References

[1] "Redis key-value store." https://redis.io/, 2023.

[2] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, pp. 5–, Aug. 2004.

[3] "Apache cassandra." https://cassandra.apache.org/, Last accessed date: March 4, 2025, 2024.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proc. of ACM SOSP*, (New York, NY, USA), p. 205–220, 2007.

[5] S. Zhou and S. Mu, "Fault-Tolerant replication with Pull-Based consensus in MongoDB," in *Proc. of USENIX NSDI*, pp. 687–703, Apr. 2021.

[6] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Proc. of USENIX NSDI*, (Berkeley, CA, USA), pp. 385–398, 2013.

[7] "Rocksdb: A persistent key-value store for flash and ram storage." https://rocksdb.org/, 2024.

[8] W. Reda, M. Canini, L. Suresh, D. Kostić, and S. Braithwaite, "Rein: Taming tail latency in key-value stores via multiget scheduling," in *Proc. of ACM EuroSys*, pp. 95–110, 2017.

[9] "Tofino switch." https://github.com/barefootnetworks/Open-Tofino, 2023.

[10] "Advanced congestion & flow control with programmable switches." https://opennetworking.org/wp-content/uploads/2020/04/JK-Lee-Slide-Deck.pdf, April 2020.

[11] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proc. of ACM SIGCOMM*, pp. 99–110, 2013.

[12] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proc. of ACM SOSP*, pp. 121–136, 2017.

[13] G. Kim and W. Lee, "In-network leaderless replication for distributed data stores," *Proc. VLDB Endow.*, vol. 15, pp. 1337–1349, Mar. 2022.

[14] H. Zhu, Z. Bai, J. Li, E. Michael, D. R. K. Ports, I. Stoica, and X. Jin, "Harmonia: Near-linear scalability for replicated storage with in-network conflict detection," *Proc. VLDB Endow.*, vol. 13, p. 376–389, Nov. 2019.

[15] H. Zhu, T. Wang, Y. Hong, D. R. K. Ports, A. Sivaraman, and X. Jin, "NetVRM: Virtual register memory for programmable networks," in *Proc. of USENIX NSDI*, (Renton, WA), pp. 155–170, USENIX Association, Apr. 2022.

[16] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. K. Ports, "Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories," in *Proc. of USENIX OSDI*, pp. 387–406, Nov. 2020.

[17] S. Sheng, H. Puyang, Q. Huang, L. Tang, and P. P. C. Lee, "FarReach: Write-back caching in programmable switches," in *Proc. of USENIX ATC*, (Boston, MA), pp. 571–584, USENIX Association, July 2023.

[18] Z. Zhu, Y. Zhao, and Z. Liu, "In-memory key-value store live migration with netmigrate," in *Proc. of USENIX FAST*, (USA), 2024.

[19] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with switchkv," in *Proc. of USENIX NSDI*, (USA), pp. 31–44, 2016.

[20] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, July 2014.

[21] "Nvidia messaging accelerator (vma)." https://docs.nvidia.com/networking/display/vmav9840, 2024.

[22] "Tommyds c library." https://www.tommyds.it/, 2018.

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. of ACM SoCC*, (New York, NY, USA), p. 143–154, Association for Computing Machinery, 2010.

[24] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," in *Proc, of USENIX NSDI*, (USA), p. 429–444, 2014.

[25] A. Katsarakis, V. Gavrielatos, M. S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. Nagarajan, "Hermes: A fast, fault-tolerant and linearizable replication protocol," in *Proc. of ACM ASPLOS*, (New York, NY, USA), p. 201–217, 2020.

[26] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at twitter," in *Proc. of USENIX OSDI*, pp. 191–208, USENIX Association, Nov. 2020.

[27] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *Proc. of USENIX FAST*, (Santa Clara, CA), Feb. 2020.

[28] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "Incbricks: Toward in-network computation with an in-network cache," in *Proc. of ACM ASPLOS*, (New York, NY, USA), p. 795–809, 2017.

[29] Z. Guo, H. Zhang, C. Zhao, Y. Bai, M. Swift, and M. Liu, "Leed: A low-power, fast persistent key-value store on smartnic jbofs," in *Proc. of ACM SIGCOMM*, (New York, NY, USA), p. 1012–1027, Association for Computing Machinery, 2023.

[30] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: a fast array of wimpy nodes," in *Proc. of ACM SOSP*, (New York, NY, USA), p. 1–14, Association for Computing Machinery, 2009.

[31] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "Kvell: the design and implementation of a fast persistent key-value store," in *Proc. of ACM SOSP*, (New York, NY, USA), p. 447–461, Association for Computing Machinery, 2019.

[32] L.-C. Canon, A. Dugois, and L. Marchal, "Solving the restricted assignment problem to schedule multi-get requests in key-value stores," in *Proc. of Euro-Par*, (Cham), pp. 195–209, 2024.

[33] V. Jaiman, S. Ben Mokhtar, and E. Rivière, "Tailx: Scheduling heterogeneous multiget queries to improve tail latencies in key-value stores," in *Proc. of IFIP DAIS*, (Berlin, Heidelberg), p. 73–92, 2020.

[34] G. Kim, "Holistic in-network acceleration for heavy-tailed storage workloads," *IEEE Access*, vol. 11, pp. 77416–77428, 2023.

[35] G. Kim, "Netclone: Fast, scalable, and dynamic request cloning for microsecond-scale rpcs," in *Proc. of ACM SIGCOMM*, p. 195–207, Sept. 2023.

[36] G. Kim, "Pushing the limits of in-network caching for key-value stores," in *Proc. of USENIX NSDI*, (Philadelphia, PA), USENIX Association, Apr. 2025.