FarReach:

Write-back Caching in Programmable Switches

USENIX ATC'23

2022.08.09

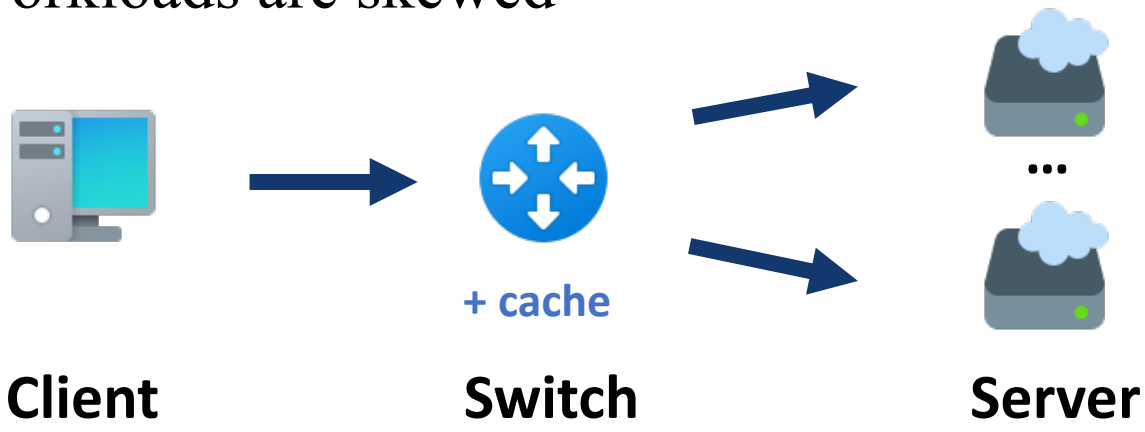
Background

Key-Value Store

Widely deployed in modern data centers to manage structured data (in units of ***records***) for data-intensive applications.

Current Features:

- Writes dominate in production key-value storage workloads
- Write-intensive workloads are skewed



Background

Programmable Switches

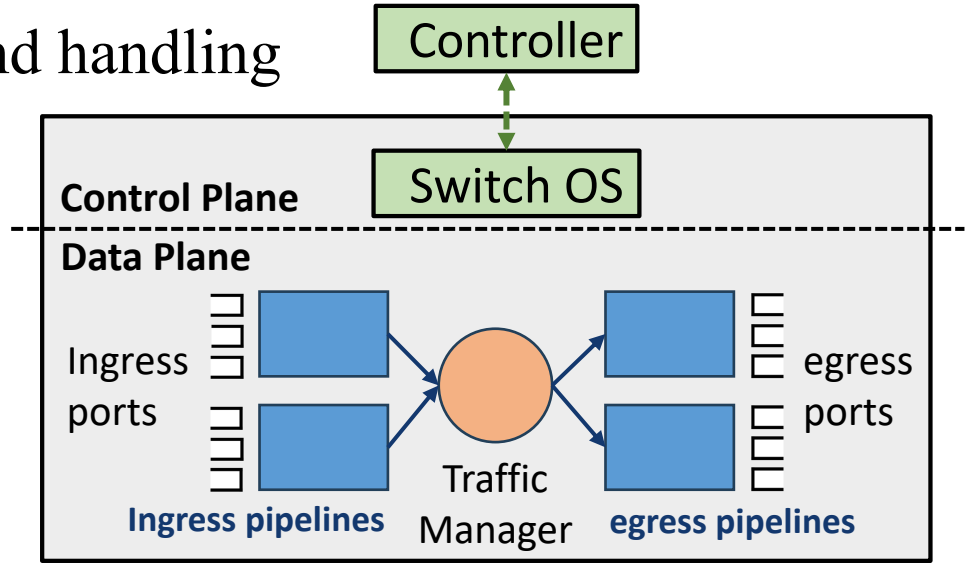
Allow network operators to dictate packet routing and handling

Data Plane

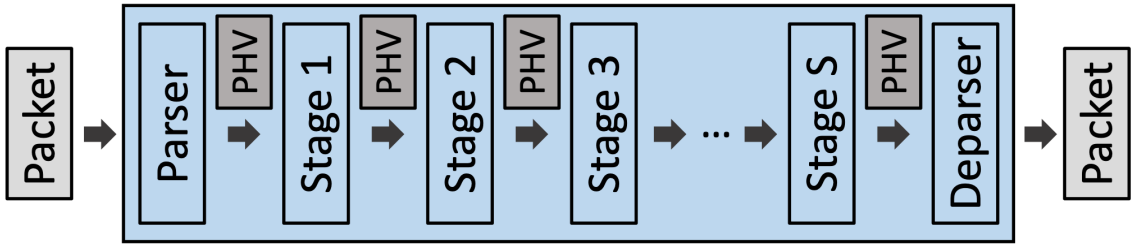
- Process packets according to predefined rules and procedures and forward them to their next destination

Control Plane

- Sets forwarding rules for the data plane
- Provide high-level control and manage overall network policies, routing algorithms, and other administrative tasks



Programmable Switch



Ingress/Egress Pipeline

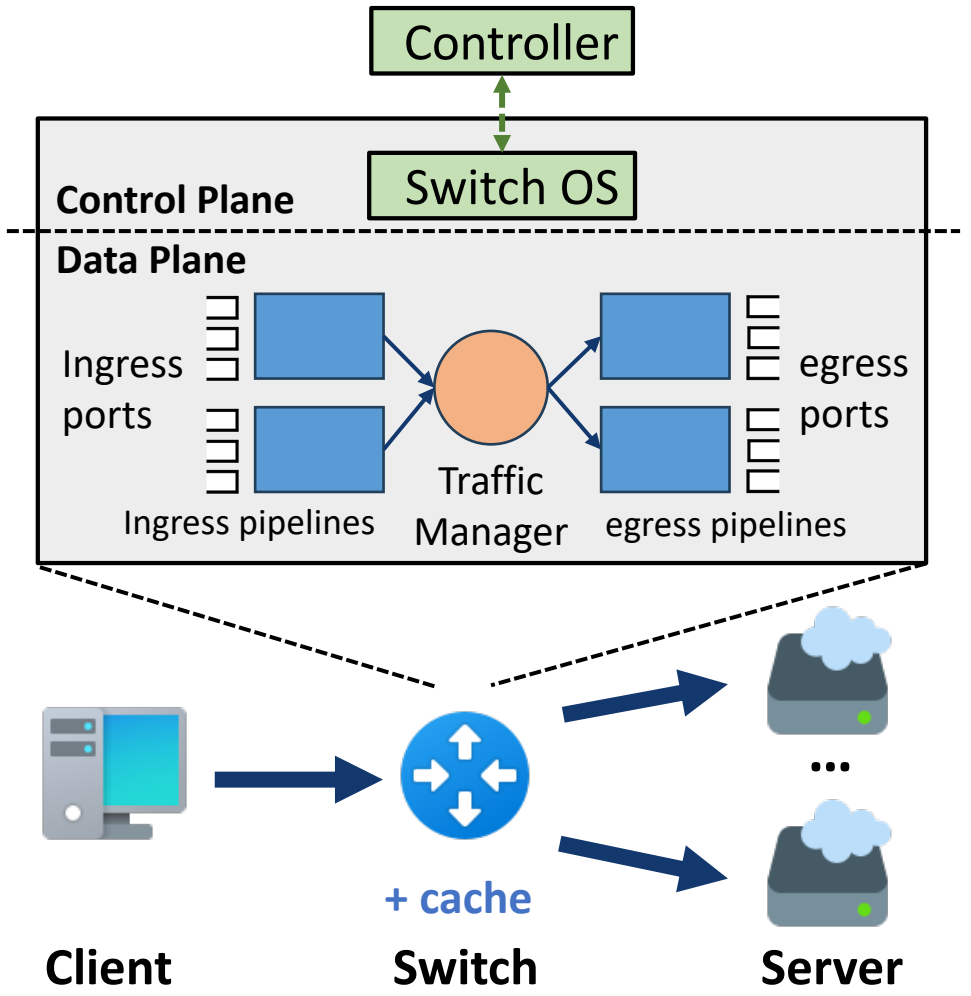
Background

Programmable Switches

Feasible to intercept I/O requests for some servers and provide stateful memory for caching records

However, existing in-switch caching approaches mainly implement **write-through** caching

- Target read-intensive workloads
- Provide low write performance gains under skewed write-intensive workloads



Background

Goal

To explore in-switch **write-back** caching, as it offers the potential to improve write performance over write-through caching by delaying server-side updates

Challenges

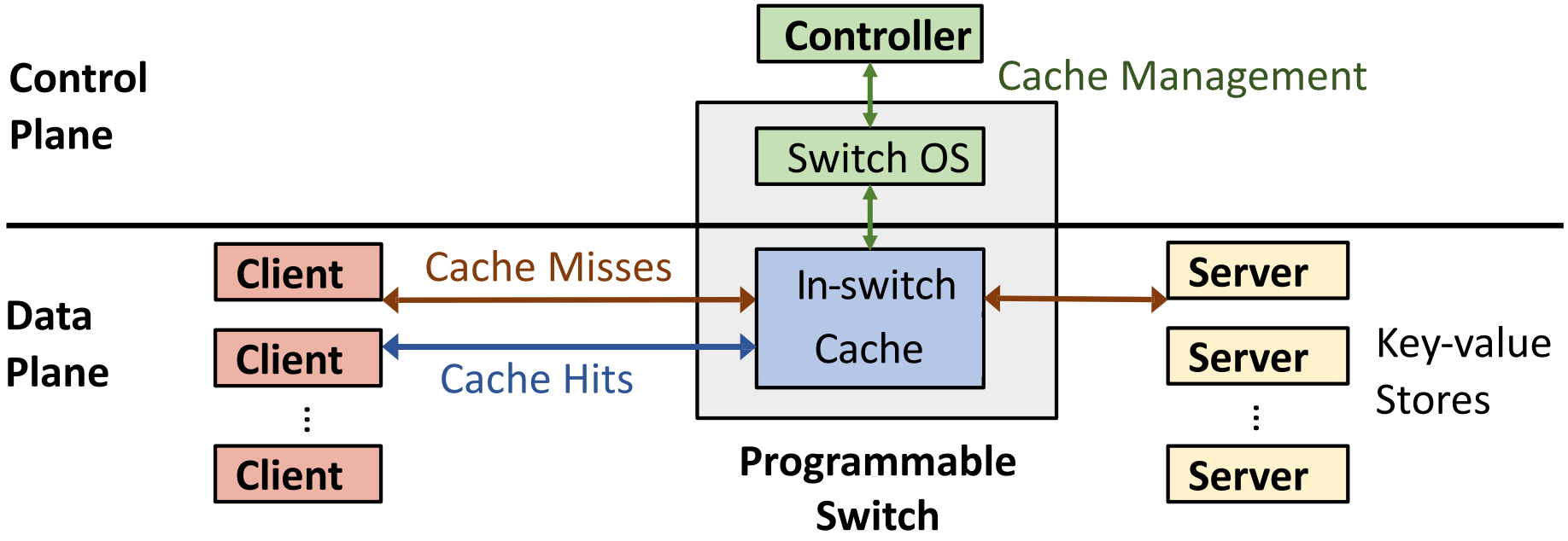
- **Performance challenge**: Limit switch resources require offloading management to a controller, which may cause high controller-to-switch latency.
- **Availability challenge**: Without proper synchronization between in-switch cache and server-side storage, the latest records may become unavailable to clients.
- **Reliability challenge**: Fault tolerance mechanism is required to avoid in-cache data loss during switch failures.

Architecture Overview

FarReach

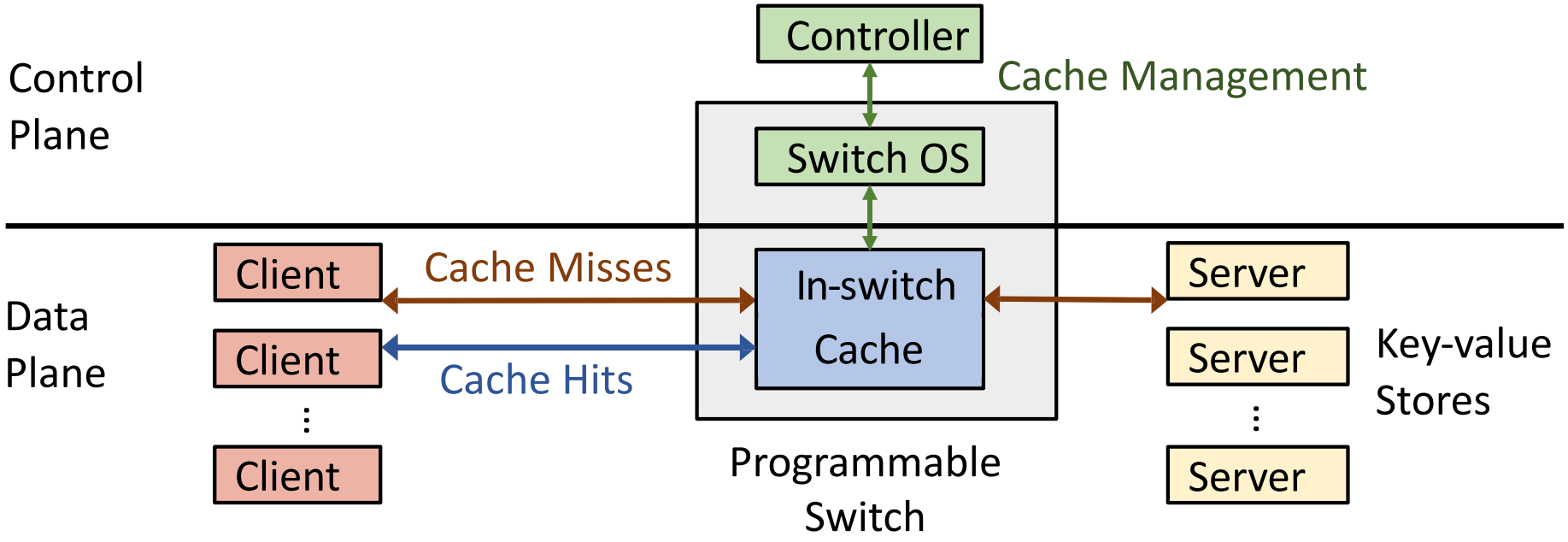
A fast, available and reliable in-switch write-back cache architecture

- Clients are connected via the in-switch cache to multiple servers for key-value storage
- Controller performs cache management through switch OS



Main Idea

- **Fast access:** Non-blocking cache admission
- **Availability:** Available cache eviction
- **Reliability:** Crash-consistent snapshot generation

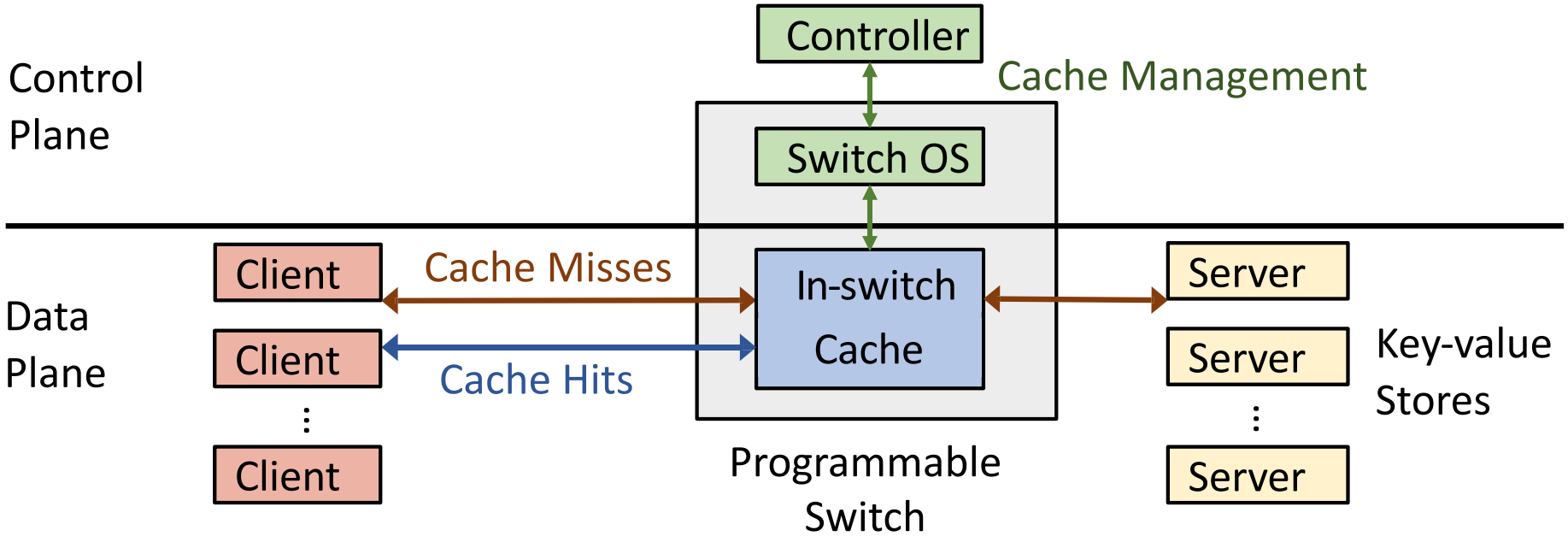


Design 1 : Non-blocking cache admission

Problems

Subsequent writes arrive at switch before cache admission

- Blocking subsequent writes undermines I/O performance
- Absorbing subsequent writes in switch undermines availability



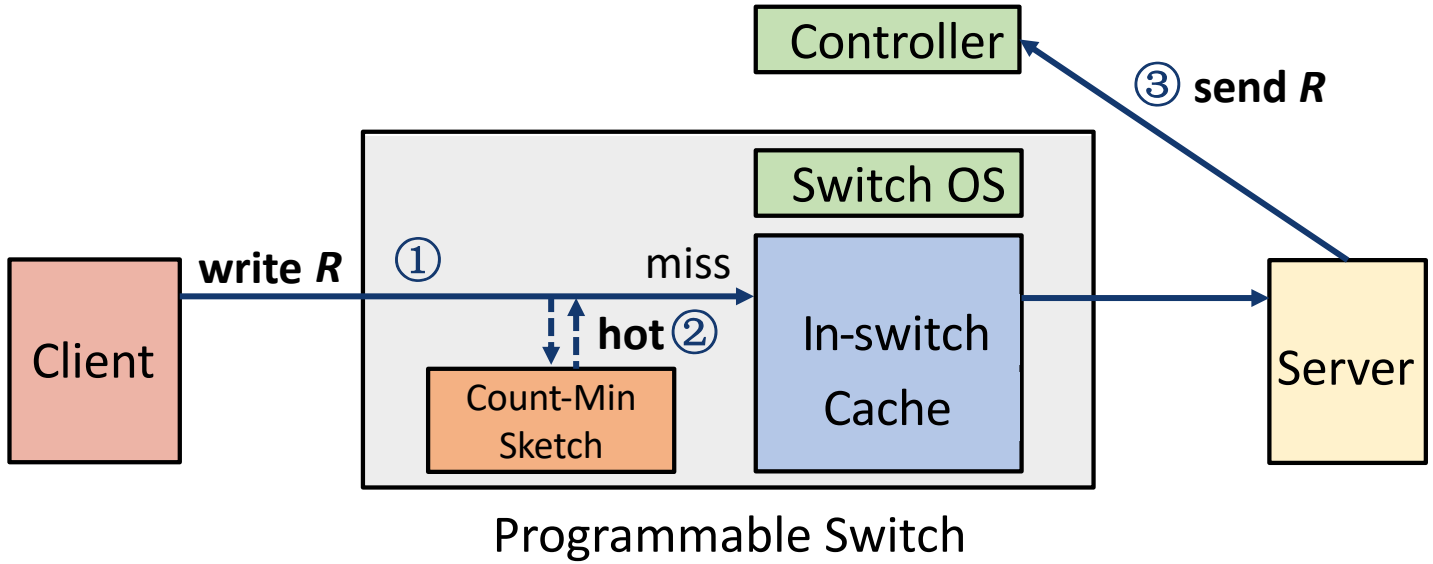
Design 1 : Non-blocking cache admission

Before cache admission

① Switch tracks record key frequencies with CM Sketch, identifying hot keys

② If record *R* is hot and uncached, switch forwards it to server

③ Server updates storage, forwards record to controller for cache admission



Design 1 : Non-blocking cache admission

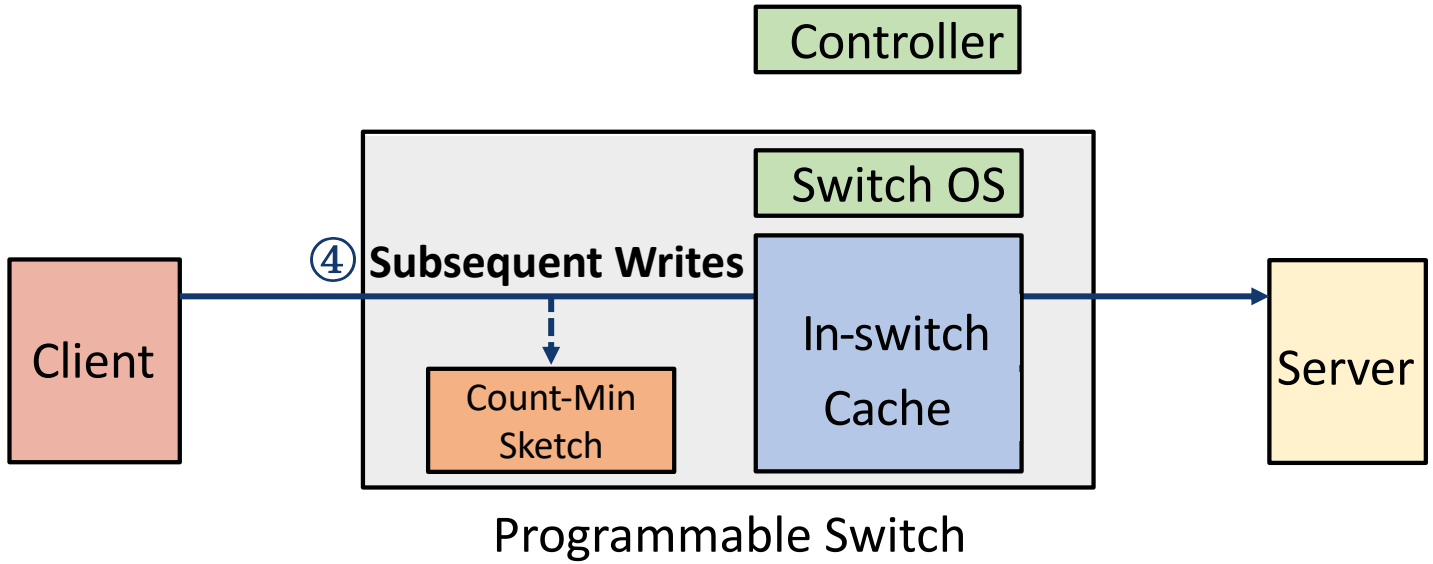
Before cache admission

① Switch tracks record key frequencies with CM Sketch, identifying hot keys

② If record *R* is hot and uncached, switch forwards it to server

③ Server updates storage, forwards record to controller for cache admission

④ Switch forwards subsequent same *R*'s key writes to server during cache admission wait



Design 1 : Non-blocking cache admission

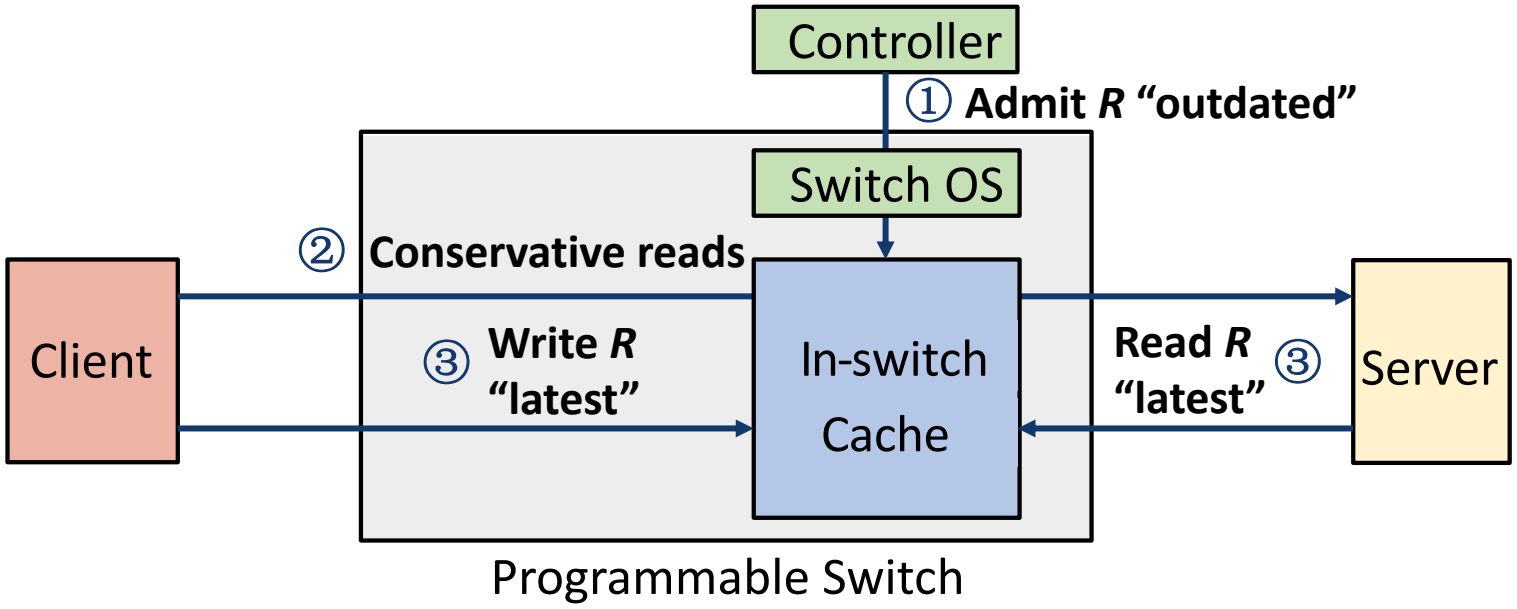
After cache admission

① Controller marks admitted *R* as "outdated"

② For any read requests to "outdated" key, switch conservatively forwards to server

③ Switch marks "outdated" *R* as "latest" if : (a) server read response for same key

(b) client write request for same key

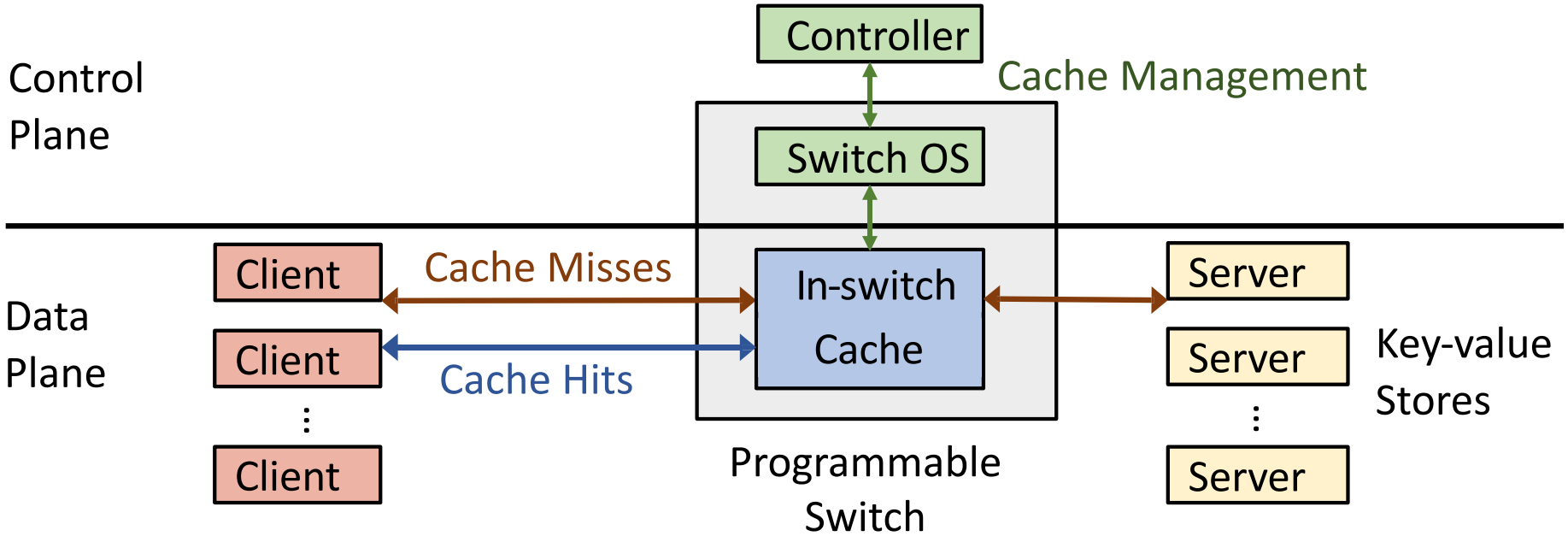


Design 2 : Available cache eviction

Problems

Subsequent writes arrive at switch during cache eviction

- Processing without synchronization undermines availability
- Synchronization by controller incurs large overhead



Design 2 : Available cache eviction

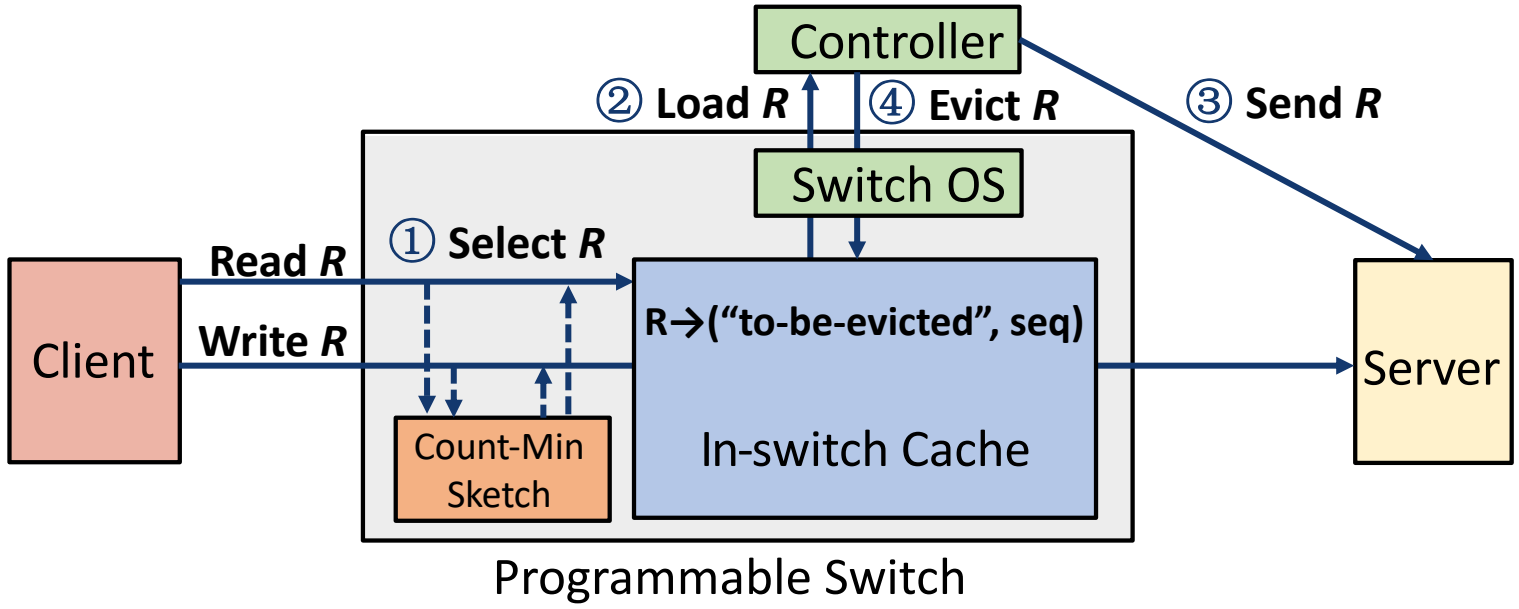
Cache eviction main workflow

① If in-switch cache is full, switch selects least accessed record *R* to evict

② Controller loads *R*, which is marked "to-be-evicted" in in-switch cache

③ Controller sends *R* to server for storage

④ After server has stored latest "to-be-evicted" *R*, controller acknowledge cache to evict *R*



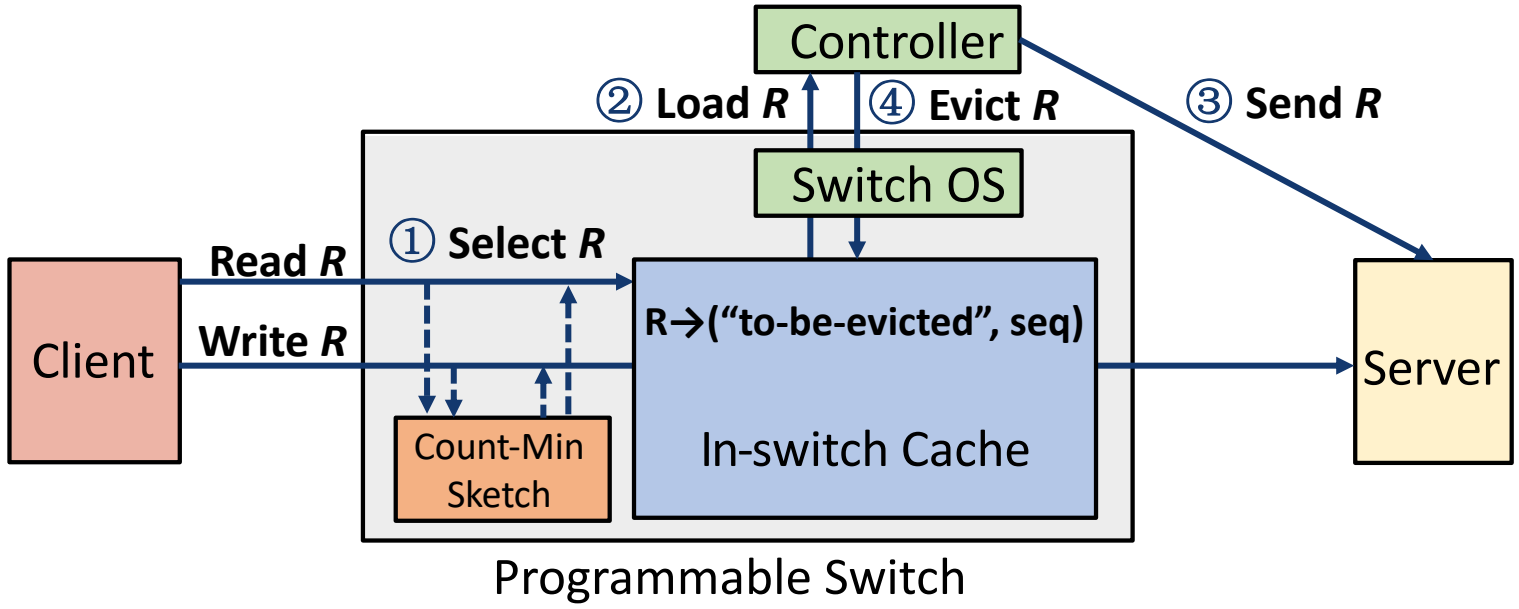
Design 2 : Available cache eviction

For any write request to *R* after ③ before ④

Switch : forwards request to server, marks *R* as "outdated"

Server :

- If sequence number of received *R* > existing *R*: overwrites existing *R*
- If sequence number of received *R* < existing *R*: discards received *R*

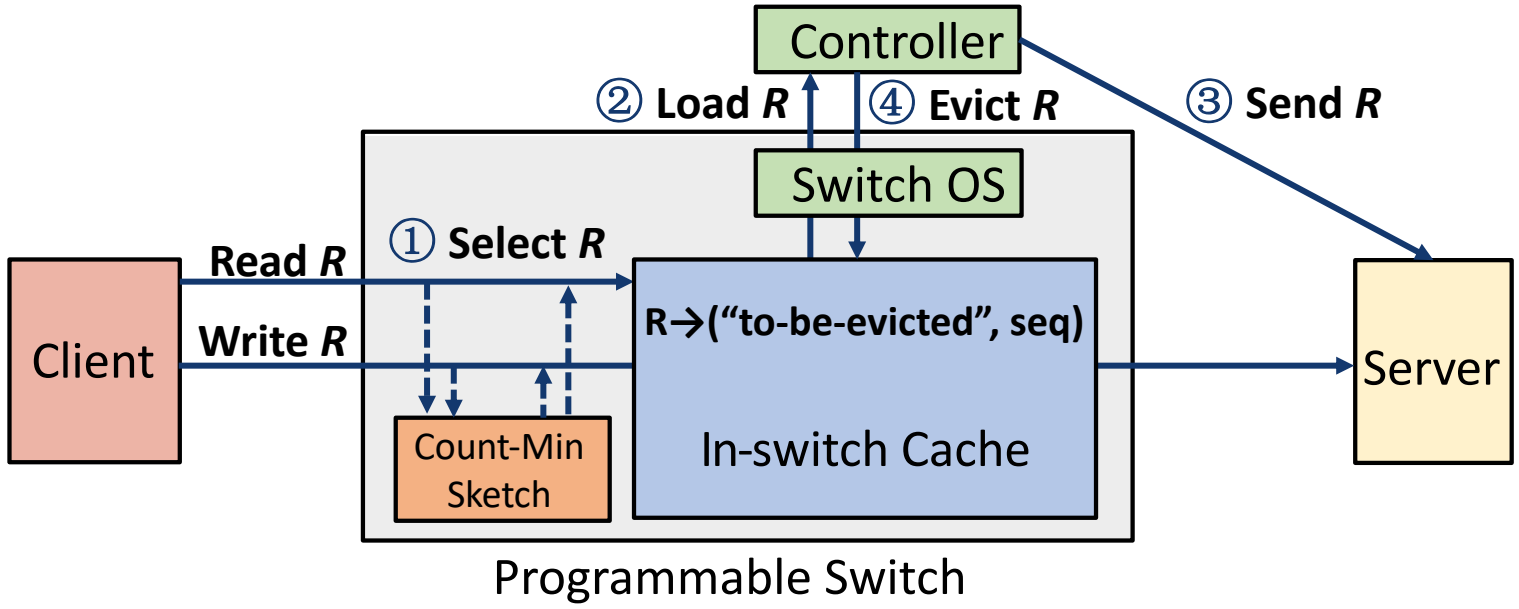


Design 2 : Available cache eviction

For any read request to *R* after ③ before ④

Switch :
- If R is “latest”: returns *R* to client
- If R is “outdated”: embeds "outdated" *R*, forwards to server

Server :
- If sequence number of embed *R* > existing *R*: overwrites with embed *R* and returns it
- If sequence number of embed *R* < existing *R*: returns existing *R* to client

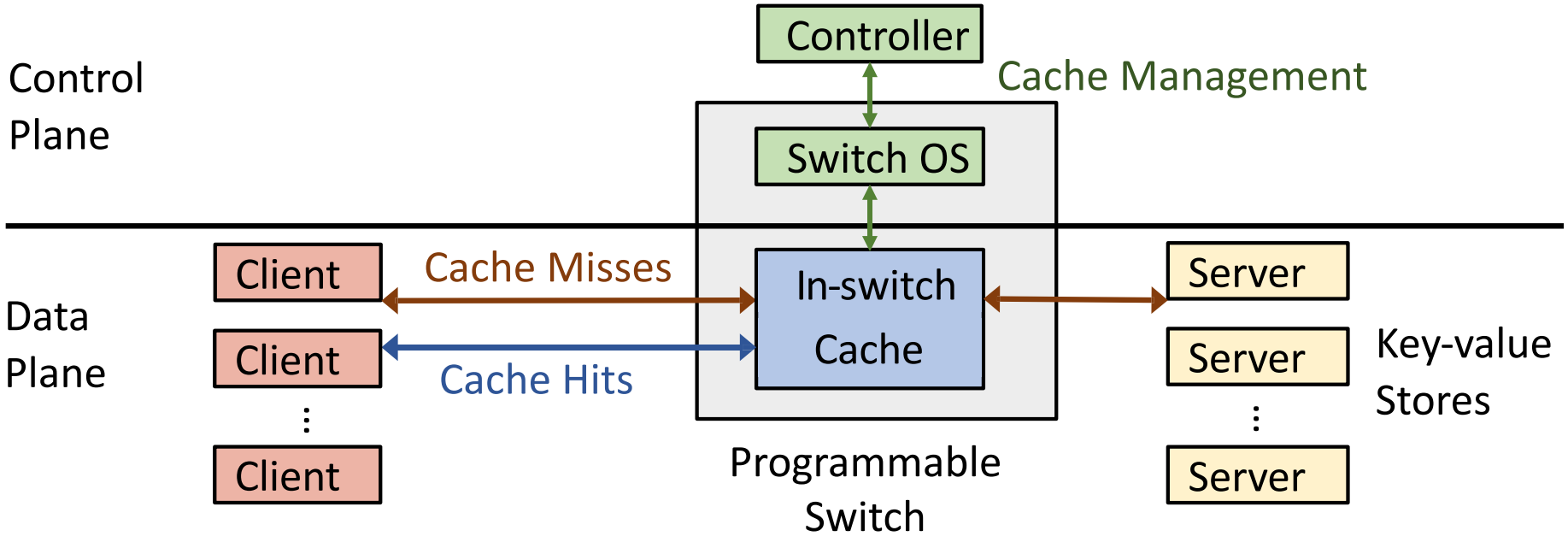


Design 3 : Crash-consistent snapshot generation

Problems

Subsequent writes arrive at switch during snapshot generation

- Updating cache records incurs inconsistent snapshots



Design 3 : Crash-consistent snapshot generation

Triggering Snapshot Generation

① At regular time points, controller notifies switch to trigger snapshot generation.

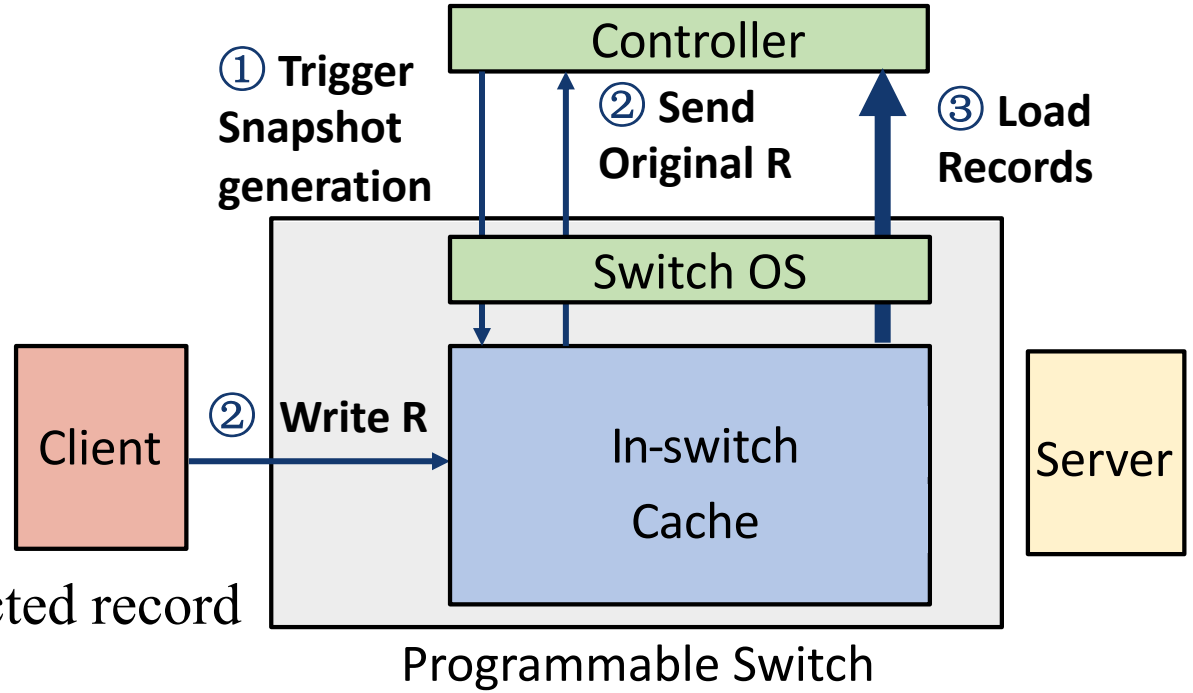
② On first write to cached *R* during snapshot, switch sends original *R* to controller.

Making a consistent snapshot

③ Controller loads all cached records for snapshot generation

④ Controller updates snapshot :

- Overwritten record: reverts with original one
- New post-snapshot admission: not included
- Post-snapshot eviction: replaces with saved evicted record



Design 3 : Crash-consistent snapshot generation

Zero-loss crash recovery

Client-side record preservation – after latest snapshot point

- After sending write request and receiving response, client keeps value and seq locally
- After completion of snapshot generation, clients release preserved records with seq no larger than those notified

Replay-based approach – after a switch failure

① Server collects latest snapshot and client-side records, selects record with largest seq per key.

② If seq of selected record is larger, server replays write request for persistent storage.

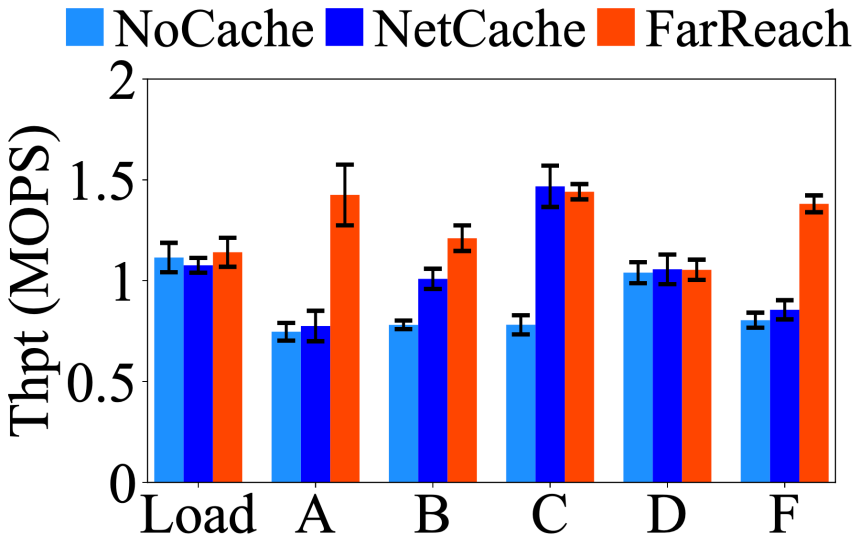
③ After persisting latest records, clients release preserved records.

④ In-switch cache recovered by replaying cache admission from latest snapshot, marks each record as “outdated”.

Evaluation

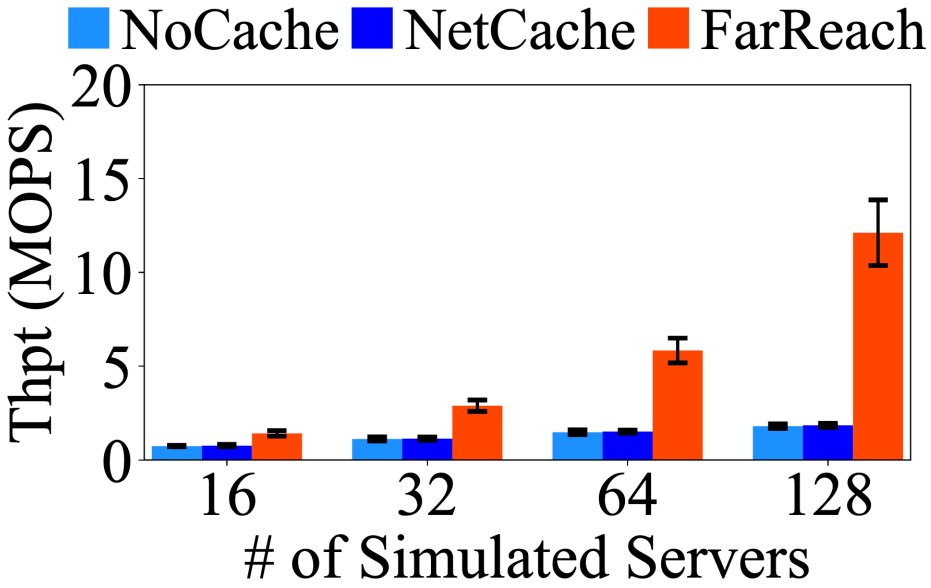
Performance under YCSB Workloads

Throughput analysis



FarReach achieves higher throughput gains over NoCache and NetCache for most workloads

Scalability analysis

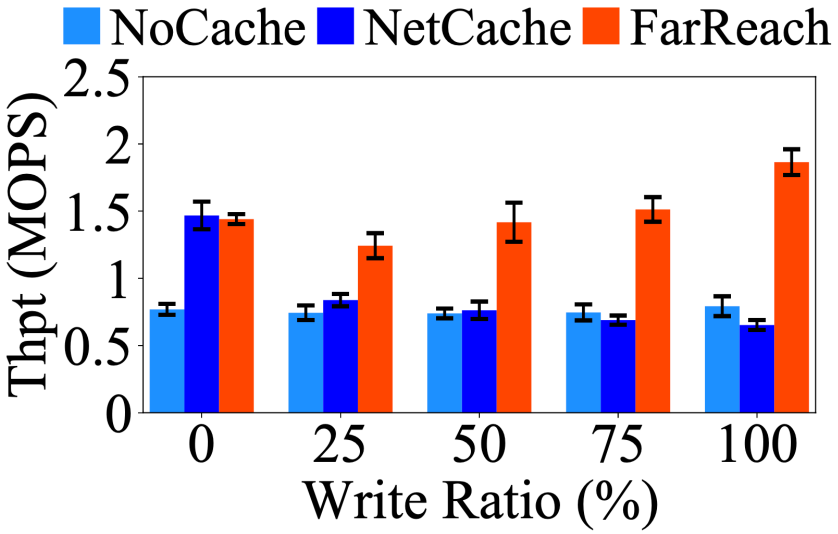


FarReach scales to a large number of servers under skewed write-intensive workloads.

Evaluation

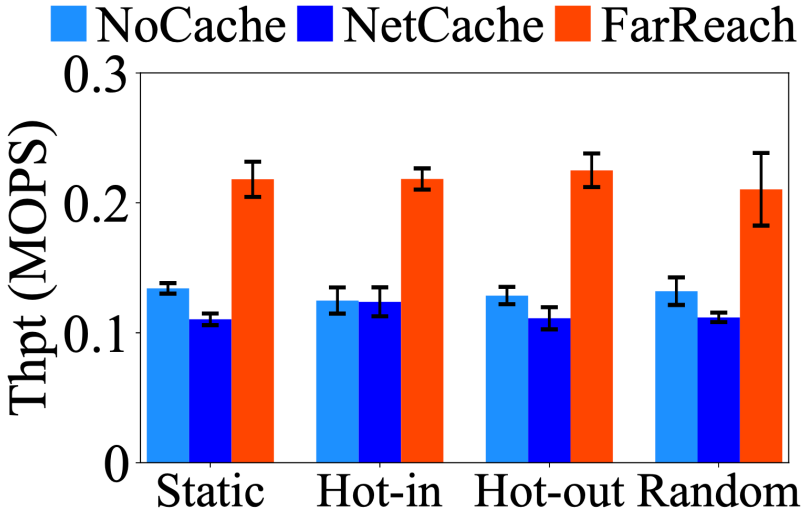
Performance under Synthetic Workloads

Impact of write ratio



FarReach achieves higher throughput gains over NoCache and NetCache for more write-intensive workloads

Impact of key popularity changes

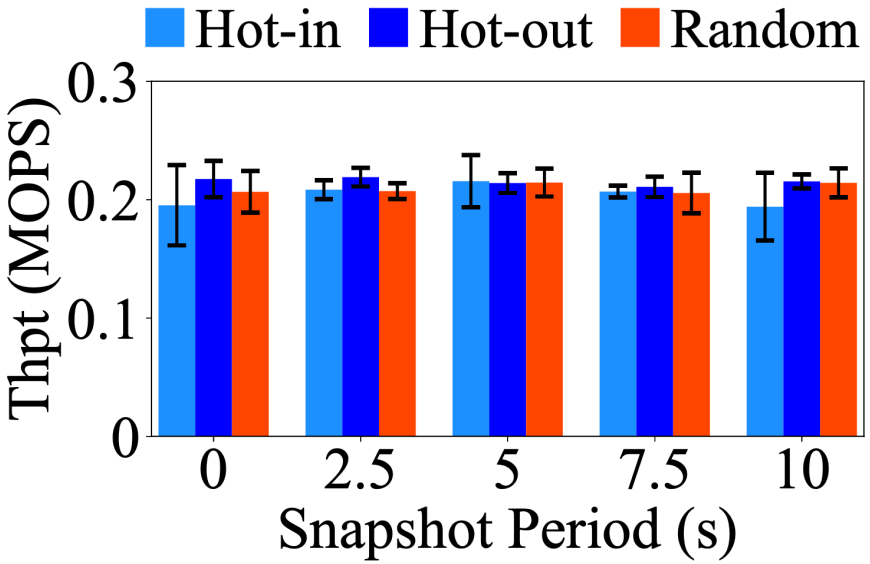


FarReach quickly reacts to the key popularity changes, so it maintains the cache hit rate and hence the average throughput.

Evaluation

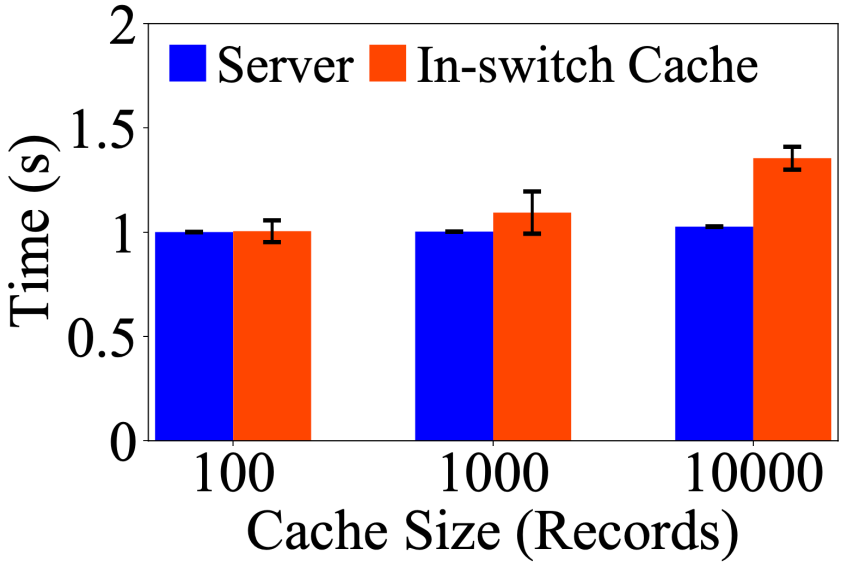
Snapshot Generation and Crash Recovery

Performance of snapshot generation



Snapshot generation has a limited impact on throughput for various snapshot periods under different dynamic patterns

Crash recovery time



Crash recovery time is within 2.35 s for various in-switch cache sizes

Paper Summary