Utilize P4 Switch as the Cache of Key-Value Store System

Tse-Jui Huang (Raymond Huang)

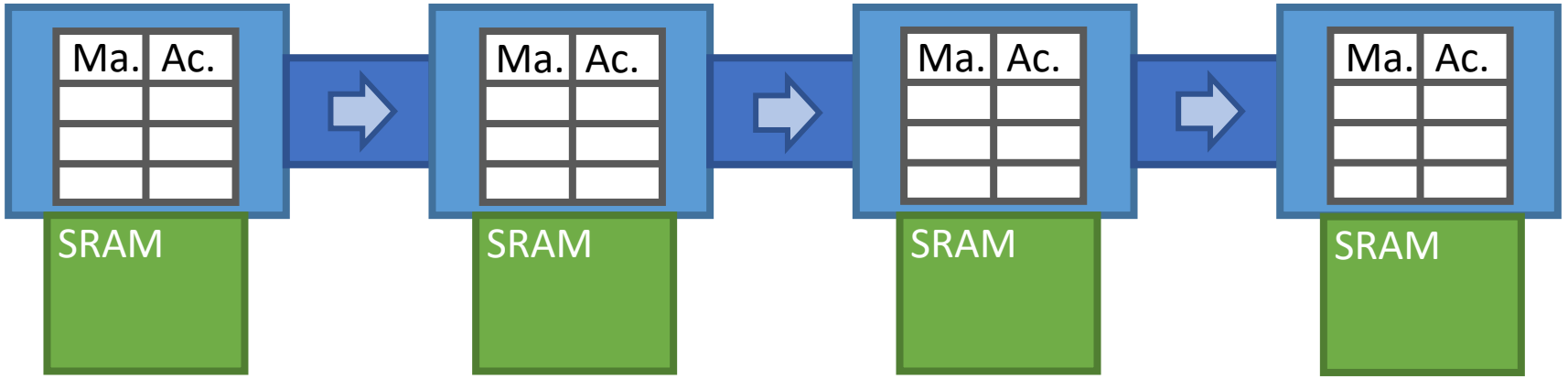
MSCS ’21

Git Repo:

https://github.com/RaymondHuang210129/P4KVCache

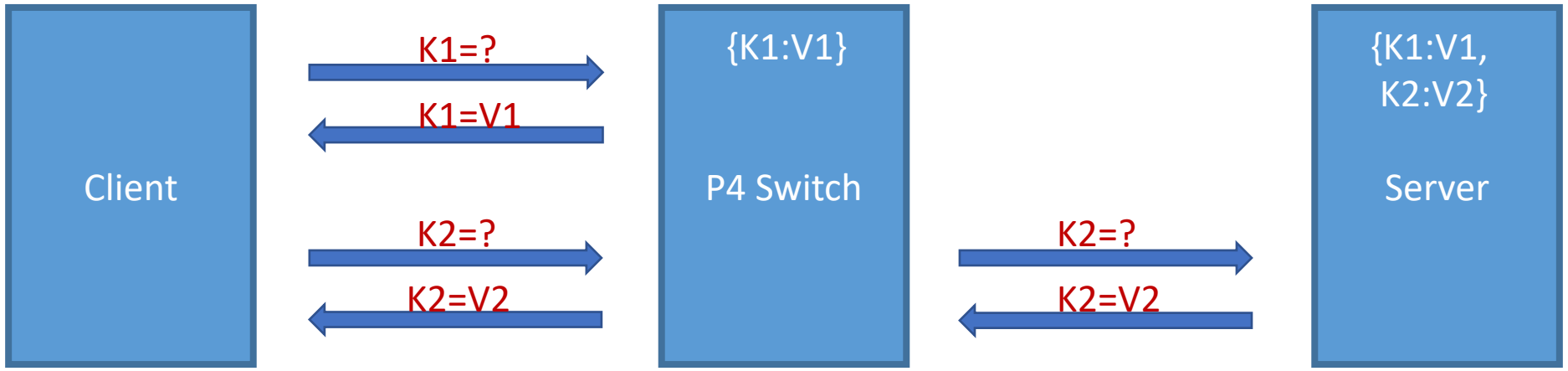
Recall

- P4 switch is a packet processing pipeline with programmability
- Can process packets statelessly using Match-Action Units
- Can process packets statefully with SRAMs

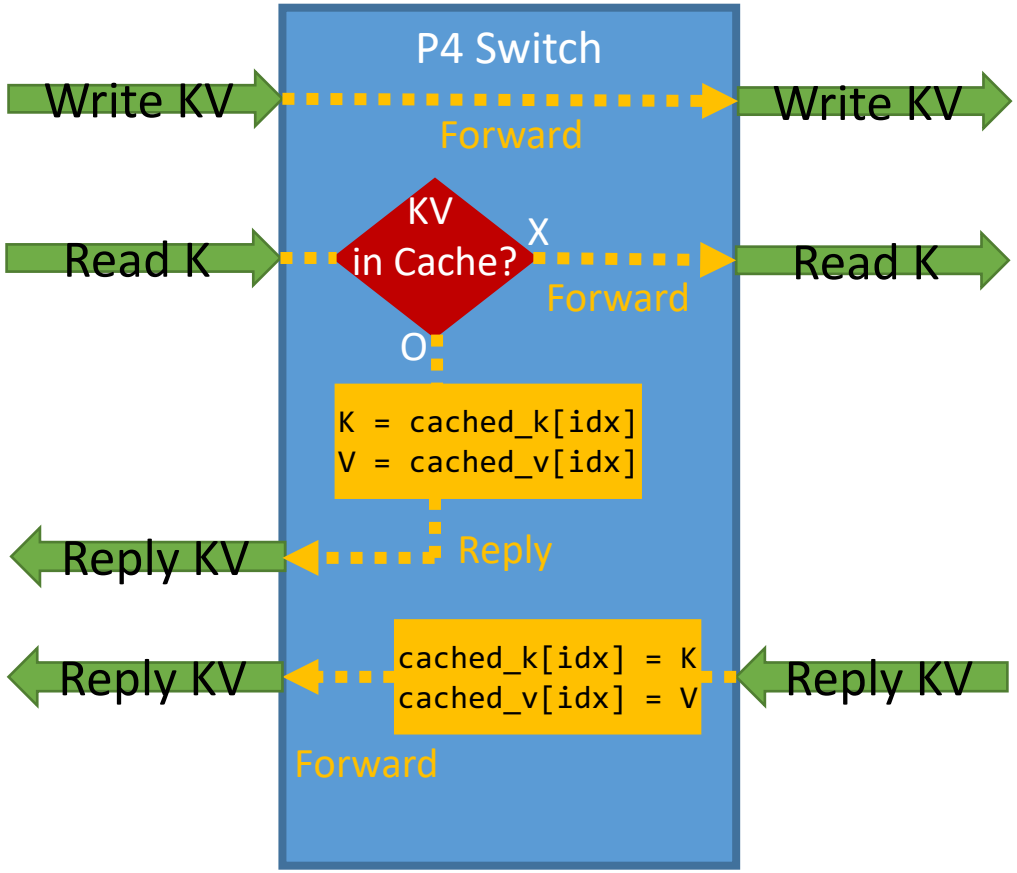


Goal of the Project:

- Apply P4 switch as a KVStore's cache
- Implementation & Optimization



Attempt 1: Simple Approach

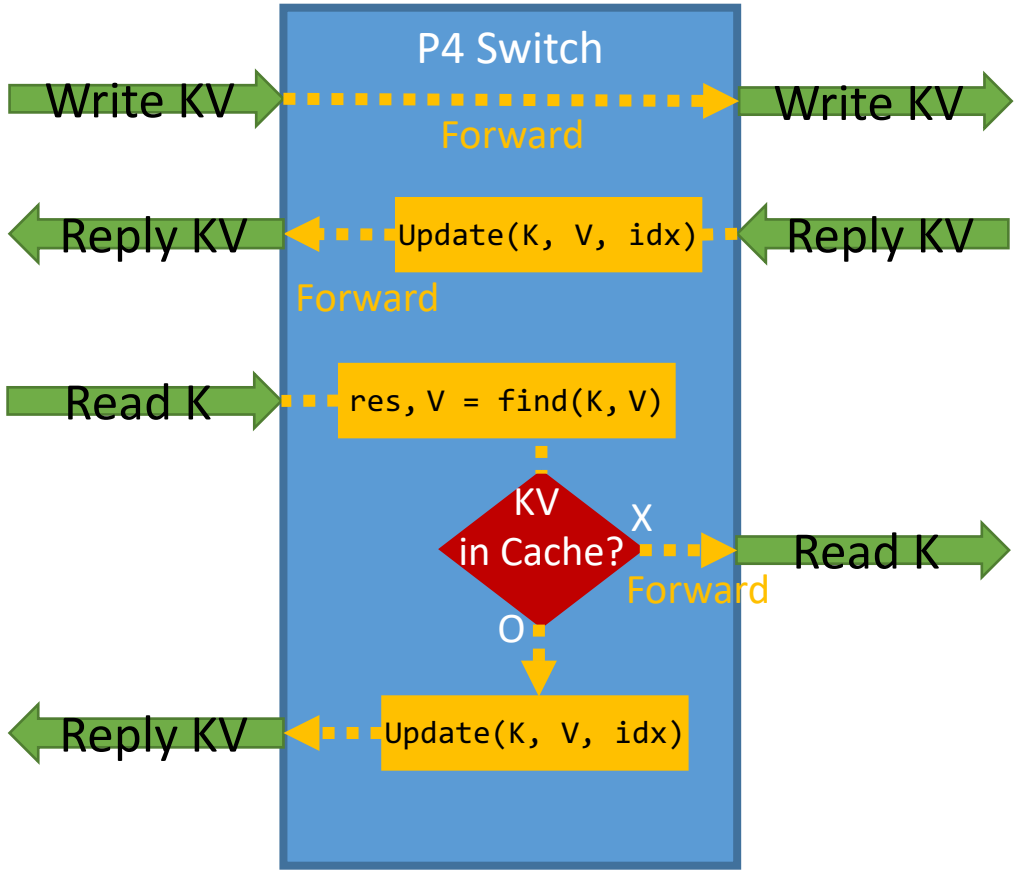



```
global var cached_key[NUM_ENTRIES]
global var cached_value[NUM_ENTRIES]
for each pkt(K, V, Type):
   idx = Hash(K)
   switch Type:
      case Write:
         forward(K, _, Write)
      case Read:
         if cached_key[idx] == K:
            reply(K, cached_value[idx])
         else:
            forward(K, _, Read)
      case Reply:
         cached_key[idx] = K
         cached_value[idx] = V
         forward(K, V, Reply)
```

Attempt 1: Observation

- Only use one cache table
  - Only use one pipeline stage in P4 switch
  - SRAM blocks at other pipeline stages are wasted!
  - Possible to use SRAMs in other pipeline stages? Yes.
- When hash collision happens, the old KV should be evicted
  - If two popular keys collide with their hash value, cache thrashing will happen
  - Possible to implement set-associative cache in a P4 switch? Yes.
- How? Implement LRU's linked list on pipeline stages!

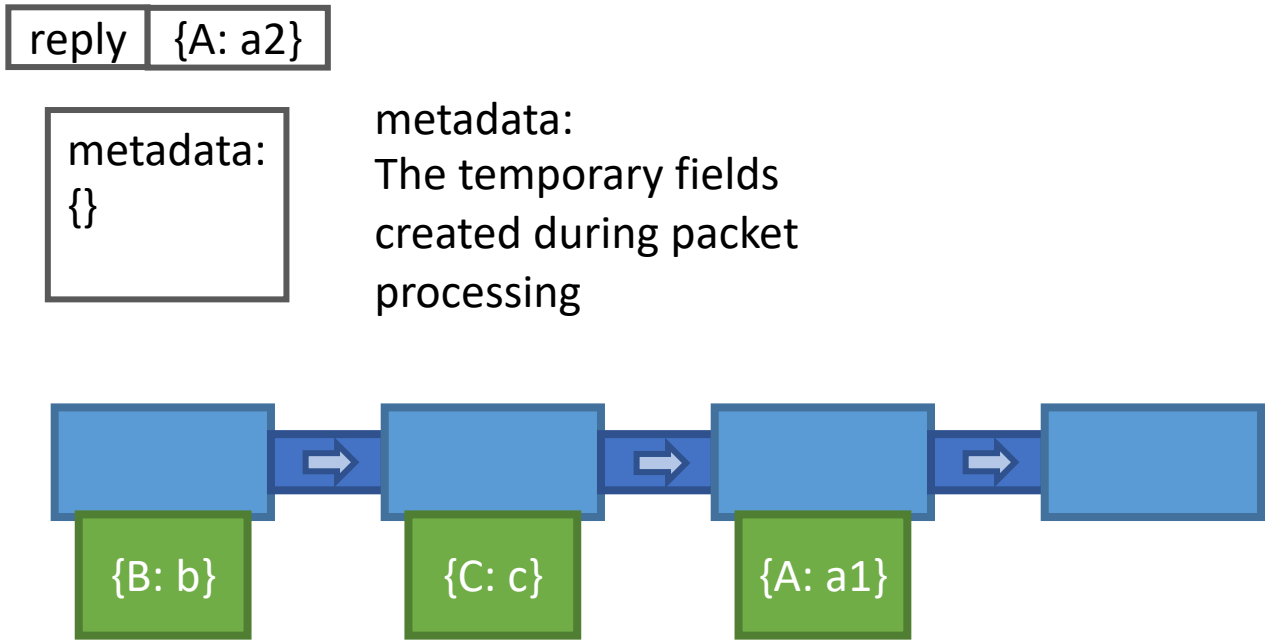
Attempt 2: Set-Associative Cache w/ LRU policy



This program cannot run on P4 switches!

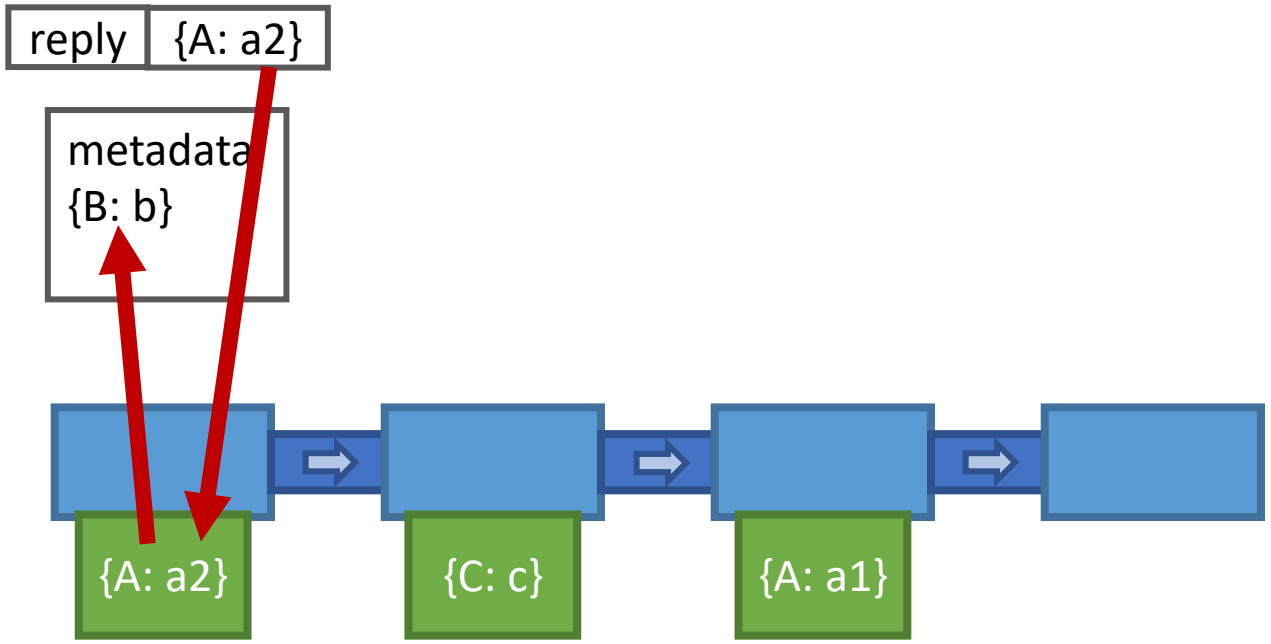
```
global var cache_K[NUM_STAGES][NUM_ENTRIES]
global var cache_V[NUM_STAGES][NUM_ENTRIES]
for each pkt(K, V, Type):
    idx = Hash(K)
    switch Type:
        case Write:
            forward(K, V, Write)
        case Reply:
            update(K, V, idx)
            forward(K, V, Reply)
        case Read:
            res, V = find(K, idx)
            if res == FOUND:
                update(K, V, idx)
                reply(K, V)
            else:
                forward(K, _, Read)
```

Attempt 2: Set-Associative Cache w/ LRU policy



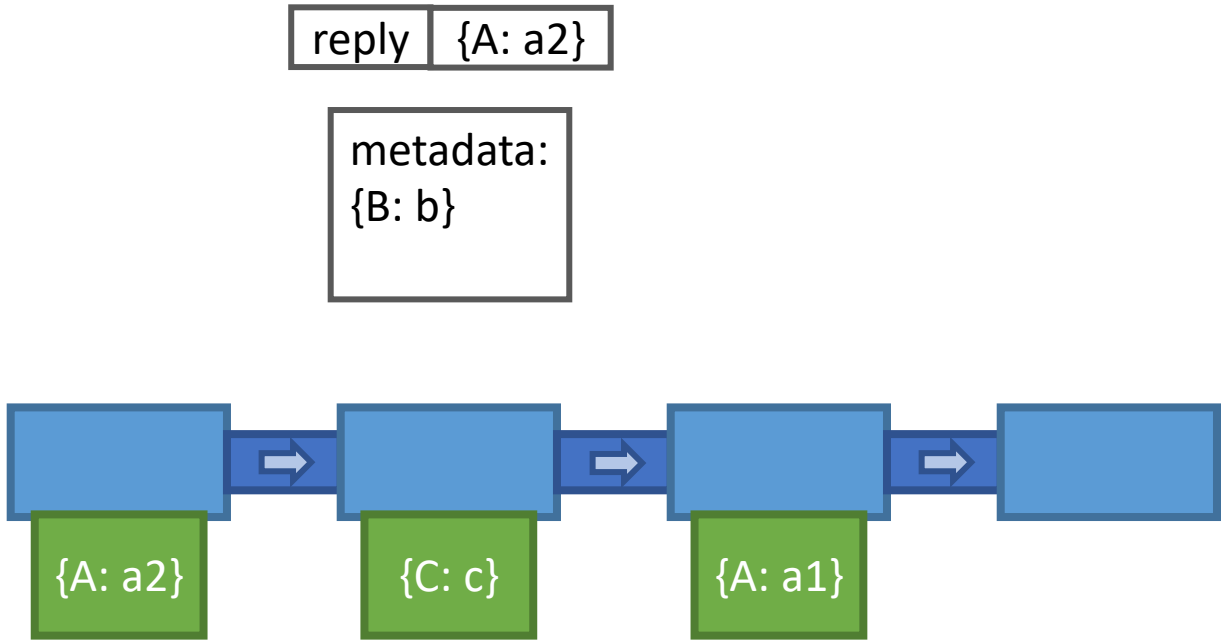

```
func update(K, V, idx):
    var meta_K, meta_V
    meta_K = cached_K[1][idx]
    meta_V = cached_V[i][idx]
    cached_K[1][idx] = K
    cached_V[1][idx] = V
    for i = 2 to num_of_cache:
        if K != evict_K:
            swap(meta_K, cached_K[i][idx])
            swap(meta_V, cached_V[i][idx])
```

Attempt 2: Set-Associative Cache w/ LRU policy



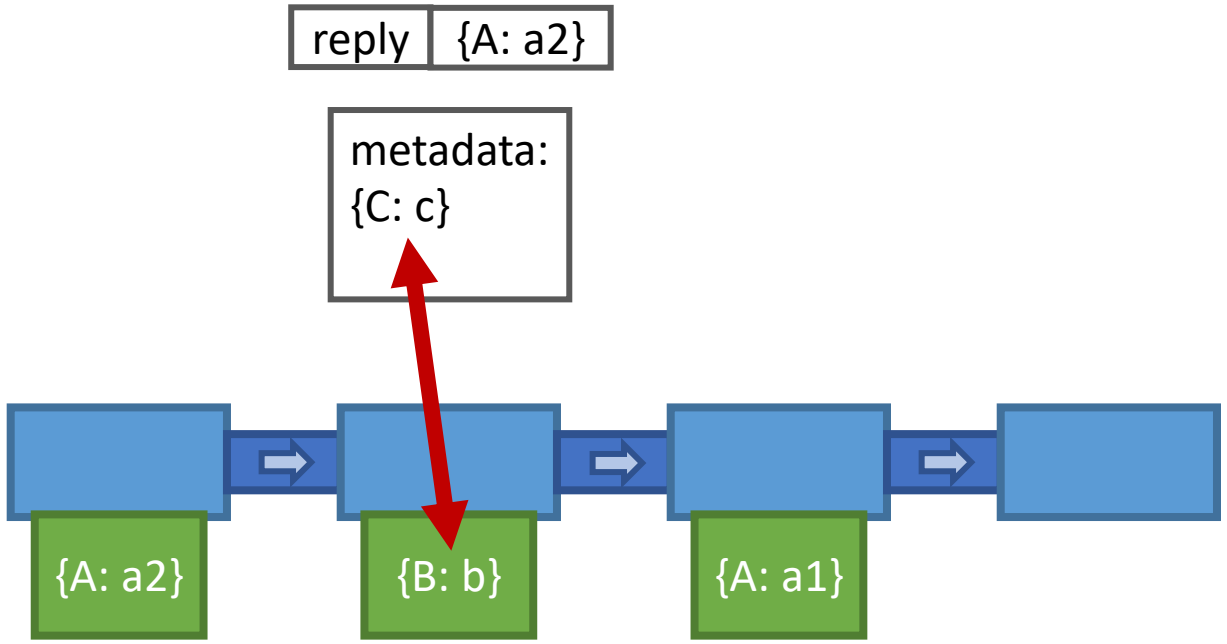

```
func update(K, V, idx):
    var meta_K, meta_V
    meta_K = cached_K[1][idx]
    meta_V = cached_V[1][idx]
    cached_K[1][idx] = K
    cached_V[1][idx] = V
    for i = 2 to num_of_cache:
        if K != evict_K:
            swap(meta_K, cached_K[i][idx])
            swap(meta_V, cached_V[i][idx])
```

Attempt 2: Set-Associative Cache w/ LRU policy



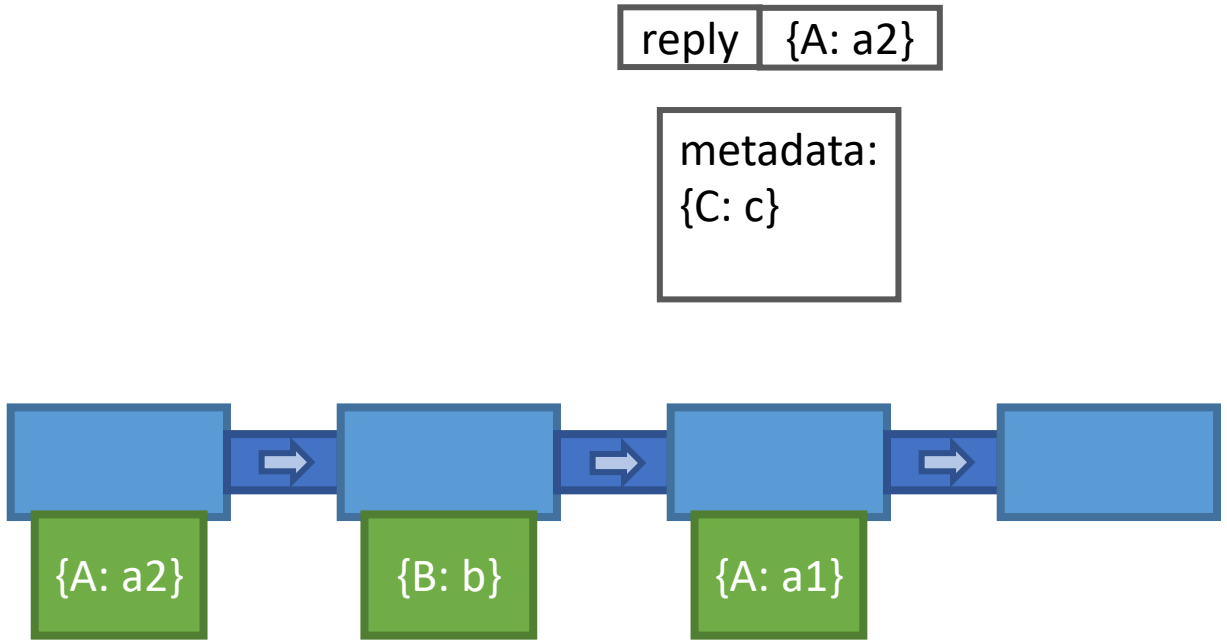

```
func update(K, V, idx):
    var meta_K, meta_V
    meta_K = cached_K[1][idx]
    meta_V = cached_V[1][idx]
    cached_K[1][idx] = K
    cached_V[1][idx] = V
    for i = 2 to num_of_cache:
        if K != evict_K:
            swap(meta_K, cached_K[i][idx])
            swap(meta_V, cached_V[i][idx])
```

Attempt 2: Set-Associative Cache w/ LRU policy



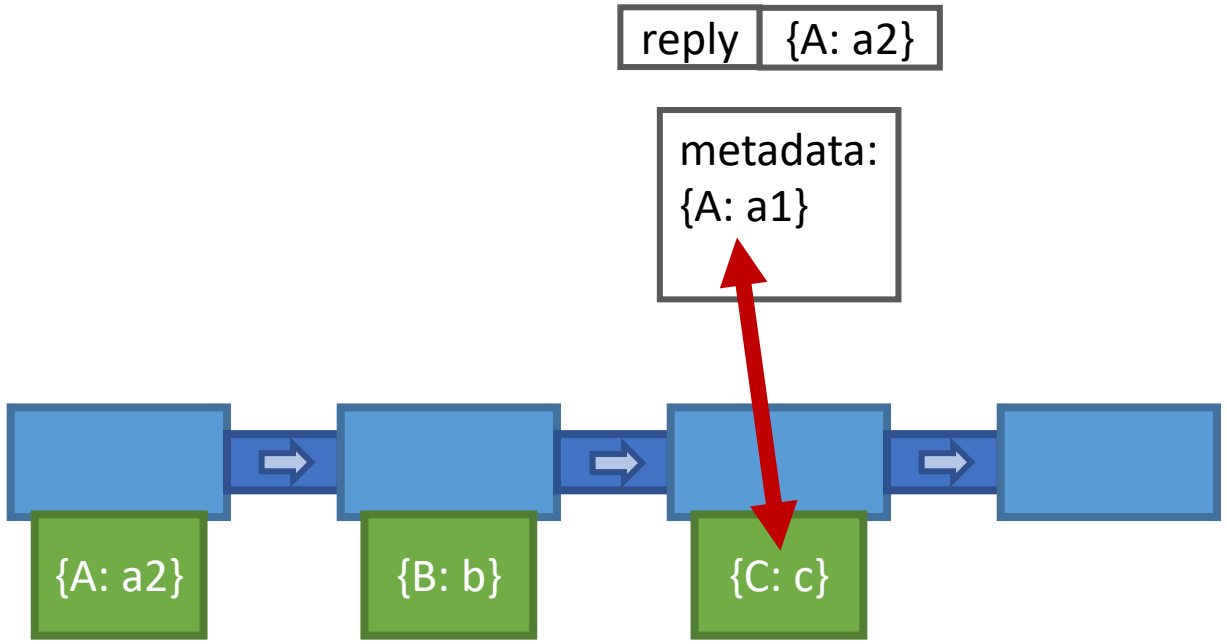

```
func update(K, V, idx):
    var meta_K, meta_V
    meta_K = cached_K[1][idx]
    meta_V = cached_V[1][idx]
    cached_K[1][idx] = K
    cached_V[1][idx] = V
    for i = 2 to num_of_cache:
        if K != evict_K:
            swap(meta_K, cached_K[i][idx])
            swap(meta_V, cached_V[i][idx])
```

Attempt 2: Set-Associative Cache w/ LRU policy



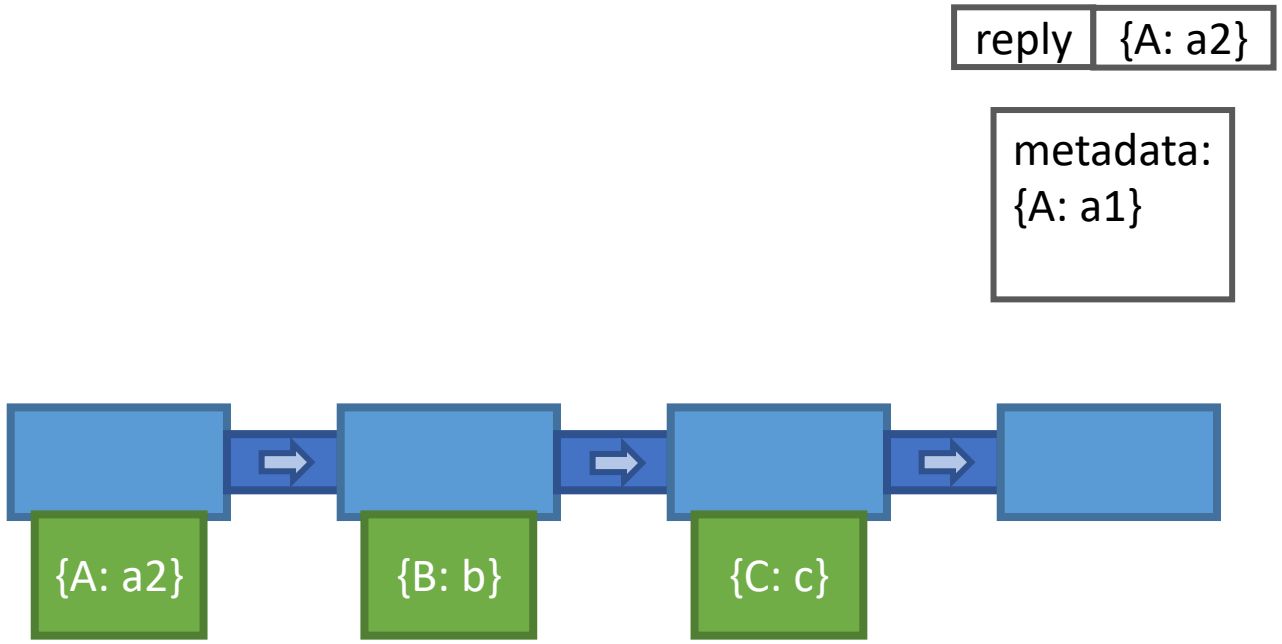

```
func update(K, V, idx):
    var meta_K, meta_V
    meta_K = cached_K[1][idx]
    meta_V = cached_V[1][idx]
    cached_K[1][idx] = K
    cached_V[1][idx] = V
    for i = 2 to num_of_cache:
        if K != evict_K:
            swap(meta_K, cached_K[i][idx])
            swap(meta_V, cached_V[i][idx])
```

Attempt 2: Set-Associative Cache w/ LRU policy



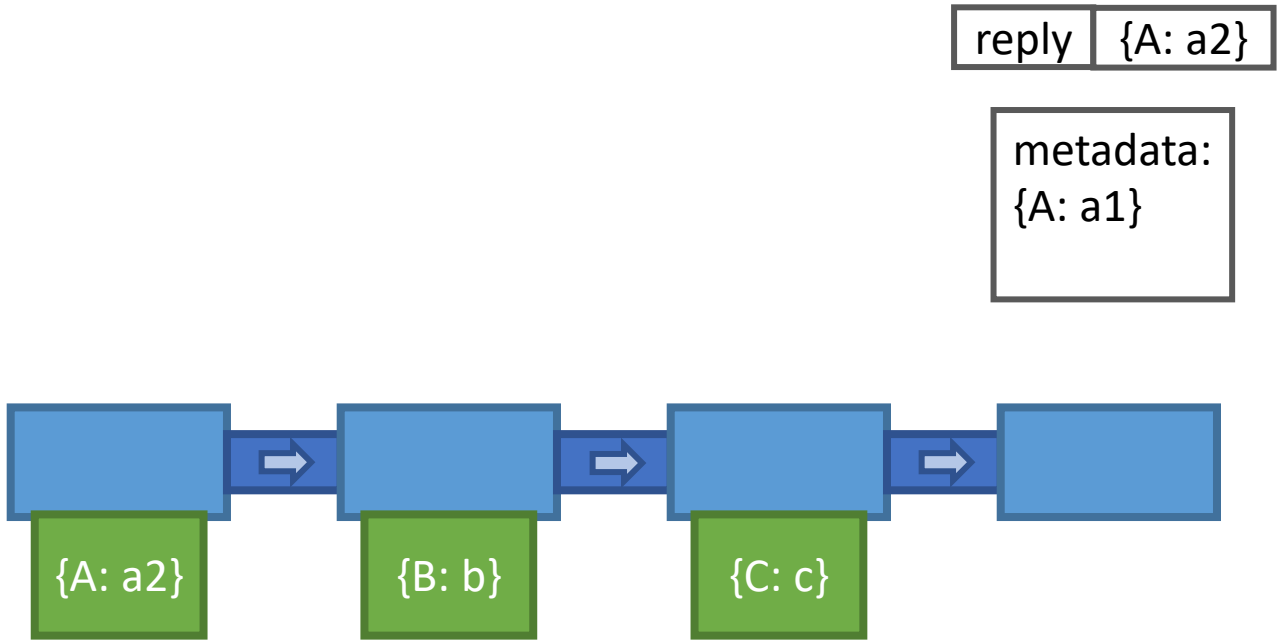

```
func update(K, V, idx):
    var meta_K, meta_V
    meta_K = cached_K[1][idx]
    meta_V = cached_V[1][idx]
    cached_K[1][idx] = K
    cached_V[1][idx] = V
    for i = 2 to num_of_cache:
        if K != evict_K:
            swap(meta_K, cached_K[i][idx])
            swap(meta_V, cached_V[i][idx])
```

Attempt 2: Set-Associative Cache w/ LRU policy




```
func update(K, V, idx):
    var meta_K, meta_V
    meta_K = cached_K[1][idx]
    meta_V = cached_V[1][idx]
    cached_K[1][idx] = K
    cached_V[1][idx] = V
    for i = 2 to num_of_cache:
        if K != evict_K:
            swap(meta_K, cached_K[i][idx])
            swap(meta_V, cached_V[i][idx])
```

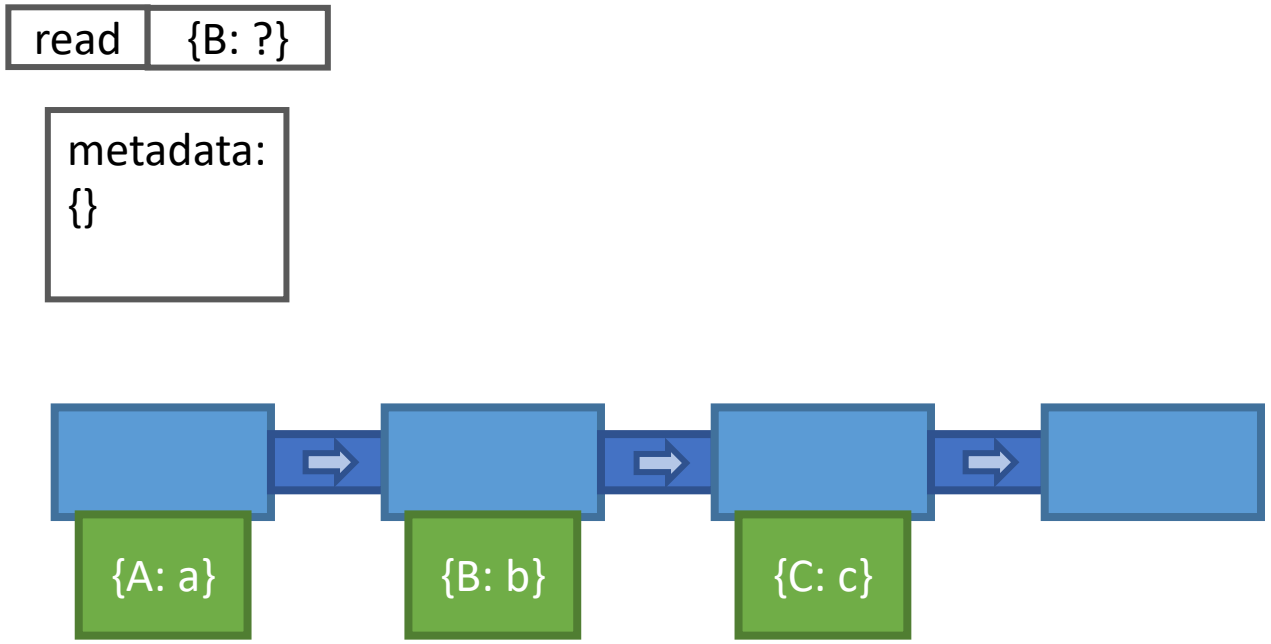
Attempt 2: Set-Associative Cache w/ LRU policy




```
func update(K, V, idx):
    var meta_K, meta_V
    meta_K = cached_K[1][idx]
    meta_V = cached_V[1][idx]
    cached_K[1][idx] = K
    cached_V[1][idx] = V
    for i = 2 to num_of_cache:
        if K != evict_K:
            swap(meta_K, cached_K[i][idx])
            swap(meta_V, cached_V[i][idx])
```

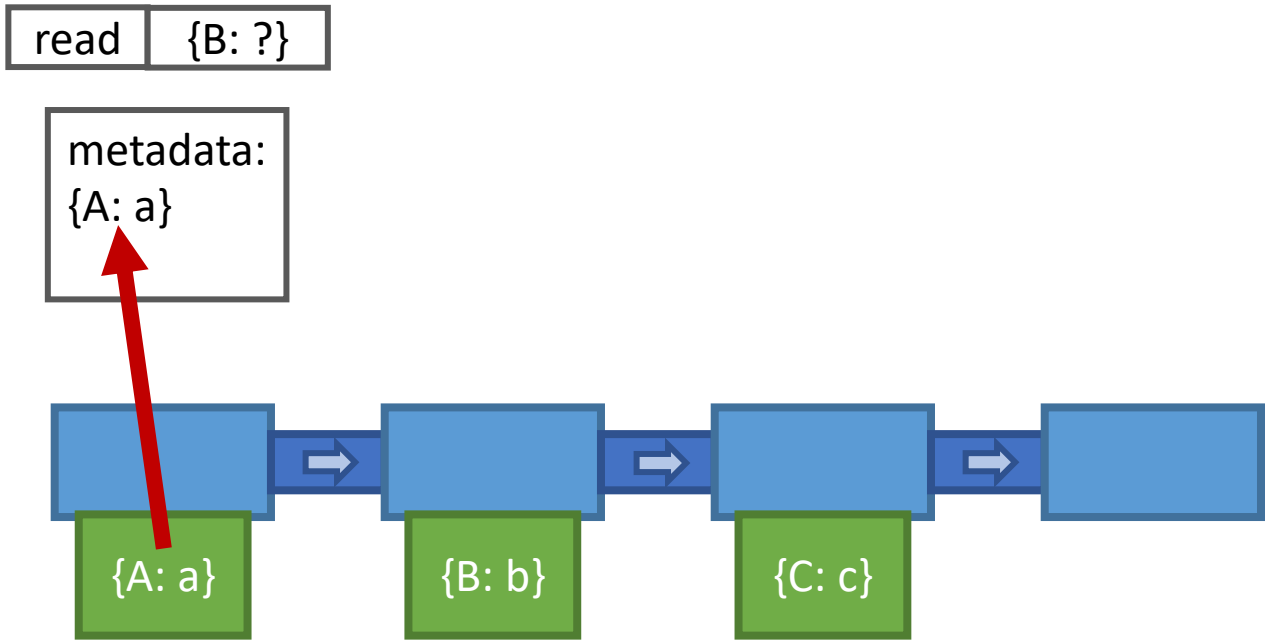
The point: the packet needs to traverse the whole pipeline to complete update()

Attempt 2: Set-Associative Cache w/ LRU policy



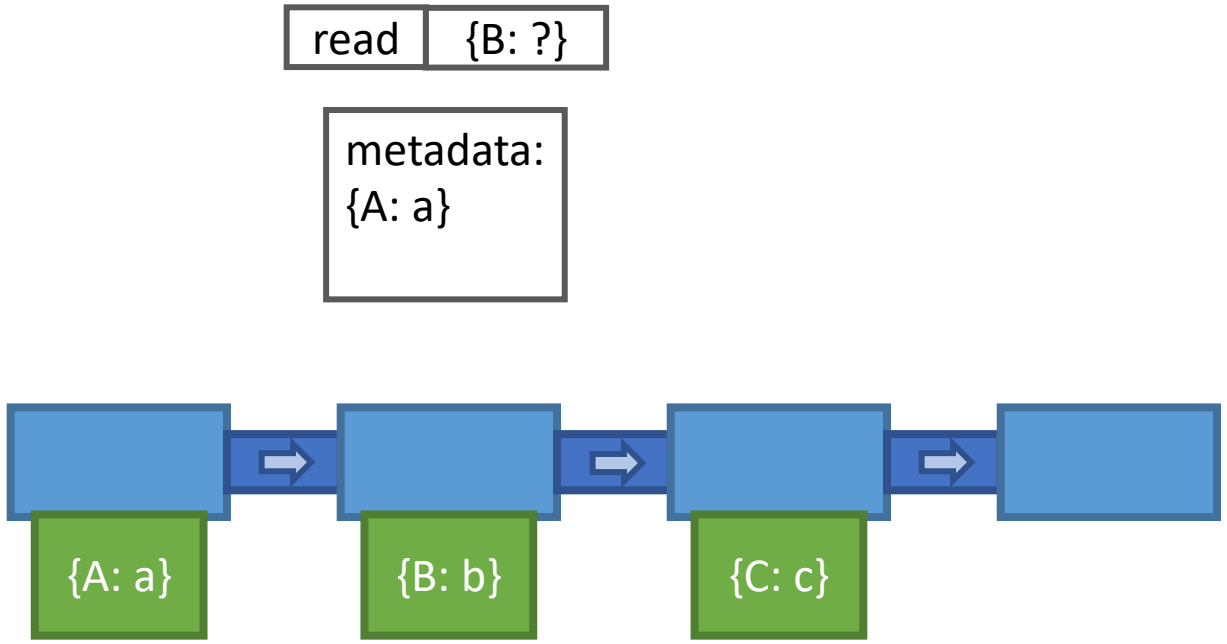

```
func find(K, V, Type, idx):
    var meta_K, meta_V
    meta_K = cached_K[1][idx]
    meta_V = cached_V[1][idx]
    for i = 2 to num_of_cache:
        if meta_K != K:
            meta_K = cached_K[1][idx]
            meta_V = cached_V[1][idx]
    if meta_K == K:
        V = meta_V
        return FOUND, V
    else:
        return NOT_FOUND, _
```

Attempt 2: Set-Associative Cache w/ LRU policy



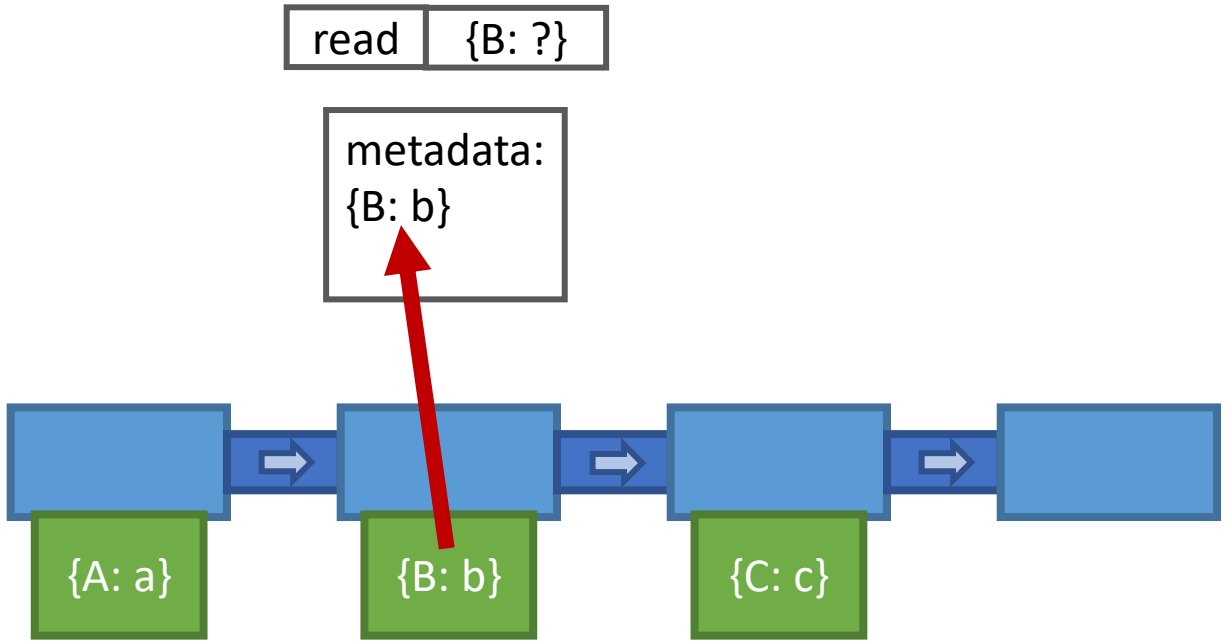

```
func find(K, V, Type, idx):
    var meta_K, meta_V
    meta_K = cached_K[1][idx]
    meta_V = cached_V[1][idx]
    for i = 2 to num_of_cache:
        if meta_K != K:
            meta_K = cached_K[1][idx]
            meta_V = cached_V[1][idx]
    if meta_K == K:
        V = meta_V
        return FOUND, V
    else:
        return NOT_FOUND, _
```

Attempt 2: Set-Associative Cache w/ LRU policy



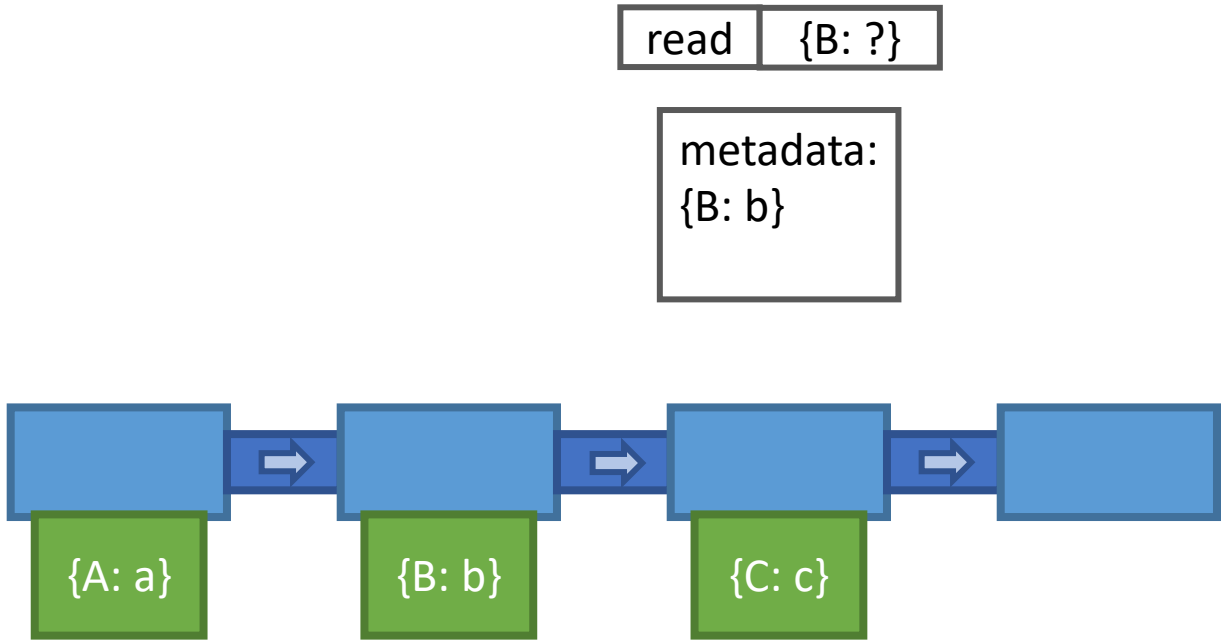

```
func find(K, V, Type, idx):
    var meta_K, meta_V
    meta_K = cached_K[1][idx]
    meta_V = cached_V[1][idx]
    for i = 2 to num_of_cache:
        if meta_K != K:
            meta_K = cached_K[1][idx]
            meta_V = cached_V[1][idx]
    if meta_K == K:
        V = meta_V
        return FOUND, V
    else:
        return NOT_FOUND, _
```

Attempt 2: Set-Associative Cache w/ LRU policy



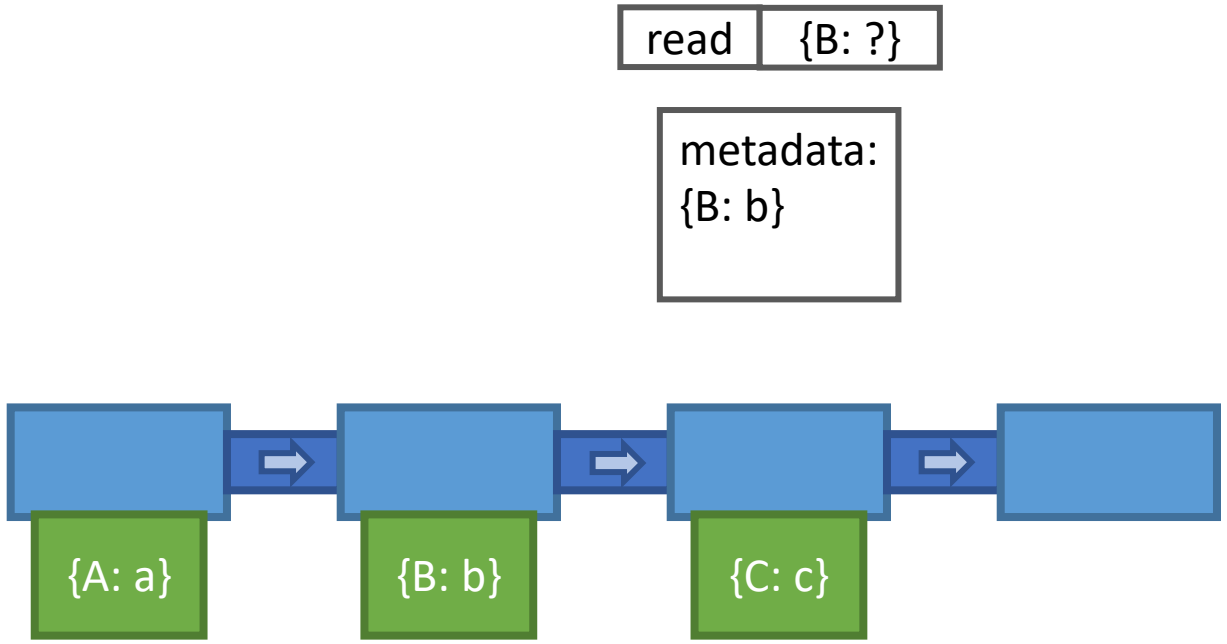

```
func find(K, V, Type, idx):
    var meta_K, meta_V
    meta_K = cached_K[1][idx]
    meta_V = cached_V[1][idx]
    for i = 2 to num_of_cache:
        if meta_K != K:
            meta_K = cached_K[1][idx]
            meta_V = cached_V[1][idx]
    if meta_K == K:
        V = meta_V
        return FOUND, V
    else:
        return NOT_FOUND, _
```

Attempt 2: Set-Associative Cache w/ LRU policy



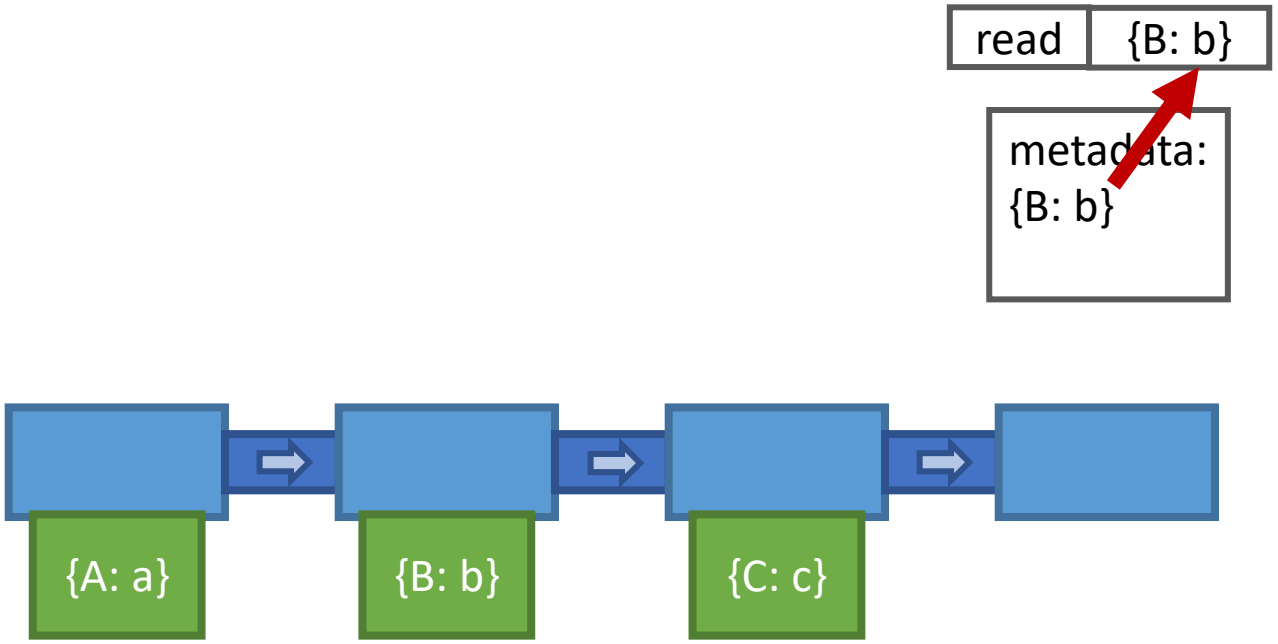

```
func find(K, V, Type, idx):
    var meta_K, meta_V
    meta_K = cached_K[1][idx]
    meta_V = cached_V[1][idx]
    for i = 2 to num_of_cache:
        if meta_K != K:
            meta_K = cached_K[1][idx]
            meta_V = cached_V[1][idx]
    if meta_K == K:
        V = meta_V
        return FOUND, V
    else:
        return NOT_FOUND, _
```

Attempt 2: Set-Associative Cache w/ LRU policy




```
func find(K, V, Type, idx):
    var meta_K, meta_V
    meta_K = cached_K[1][idx]
    meta_V = cached_V[1][idx]
    for i = 2 to num_of_cache:
        if meta_K != K:
            meta_K = cached_K[1][idx]
            meta_V = cached_V[1][idx]
    if meta_K == K:
        V = meta_V
        return FOUND, V
    else:
        return NOT_FOUND, _
```

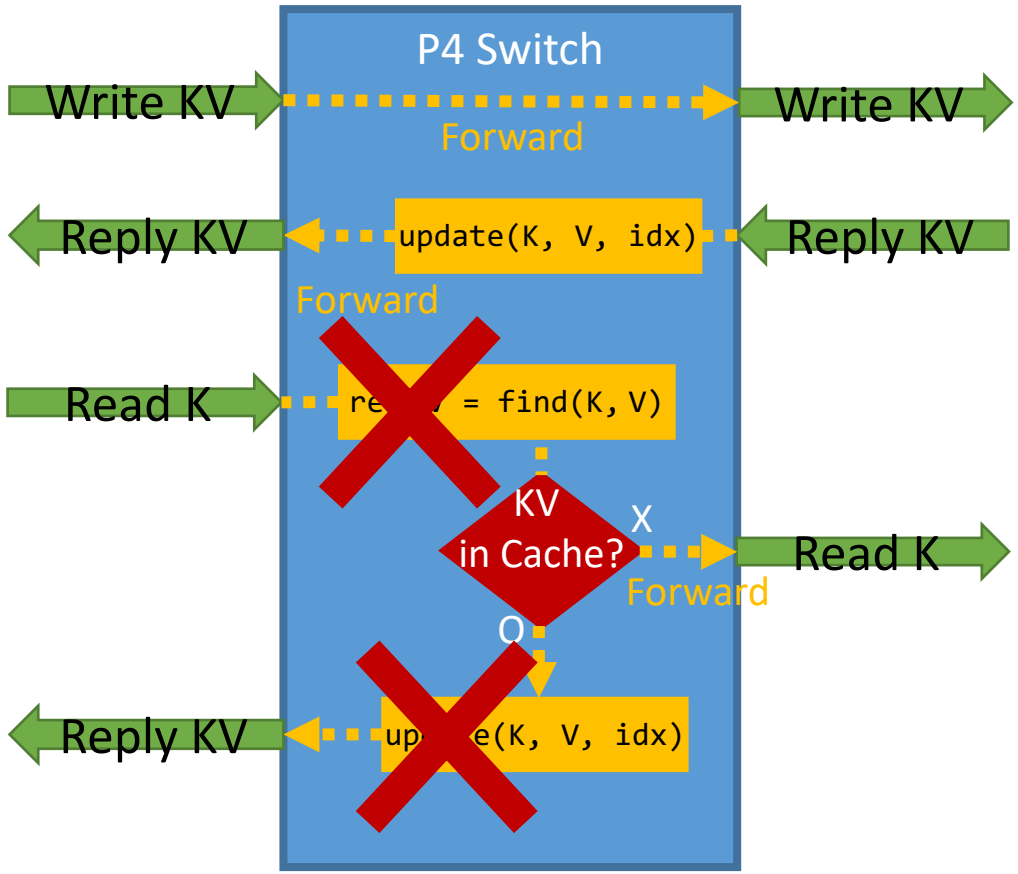
Attempt 2: Set-Associative Cache w/ LRU policy




```
func find(K, V, Type, idx):
    var meta_K, meta_V
    meta_K = cached_K[1][idx]
    meta_V = cached_V[1][idx]
    for i = 2 to num_of_cache:
        if meta_K != K:
            meta_K = cached_K[1][idx]
            meta_V = cached_V[1][idx]
    if meta_K == K:
        V = meta_V
        return FOUND, V
    else:
        return NOT_FOUND, _
```

The point: the packet needs traversing the whole pipeline to complete find() as well!

Attempt 2: Set-Associative Cache w/ LRU policy

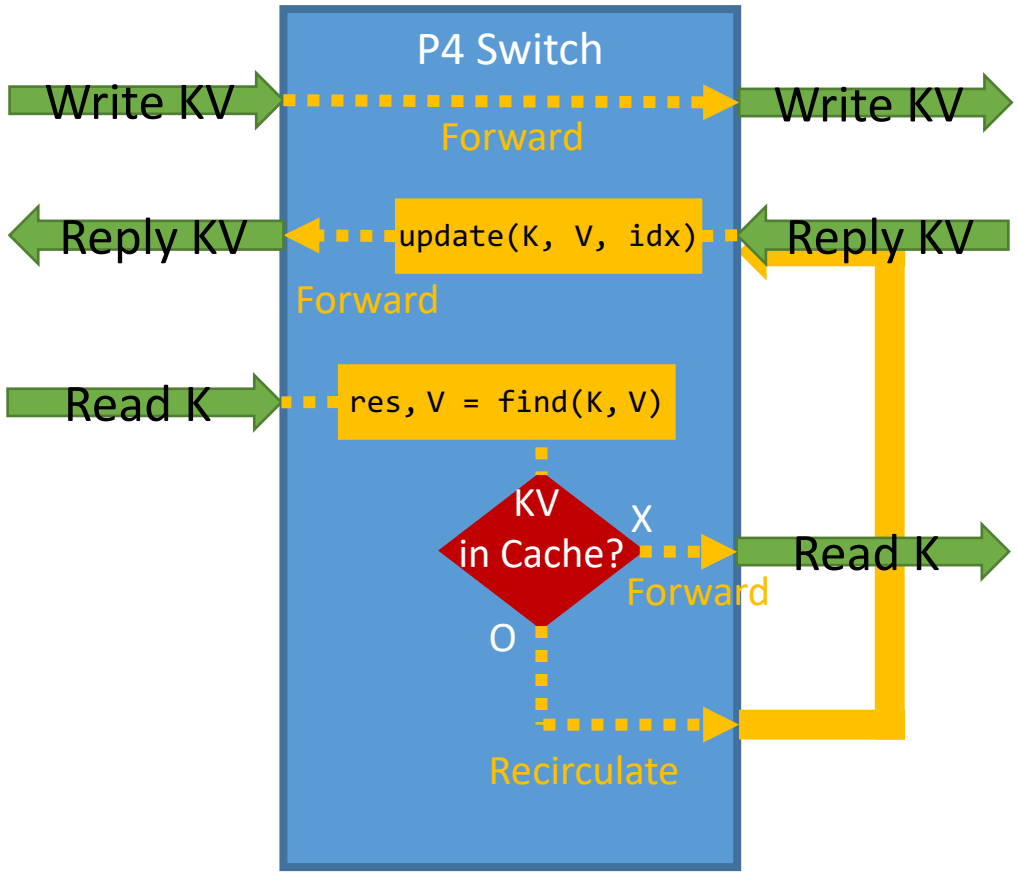



```
for each pkt(K, V, Type):
    idx = Hash(K)
    switch Type:
        case Write:
            forward(K, V, Write)
        case Reply:
            update(K, V, idx)
            forward(K, V, Reply)
        case Read:
            res, V = find(K, idx)
            if res == NOT_FOUND:
                forward(K, _, Read)
            else:
                update(K, V, idx)
                reply(K, V)
```

We cannot call both find() and update() at a packet arrival.

Solution: recirculate

Attempt 2: Set-Associative Cache w/ LRU policy

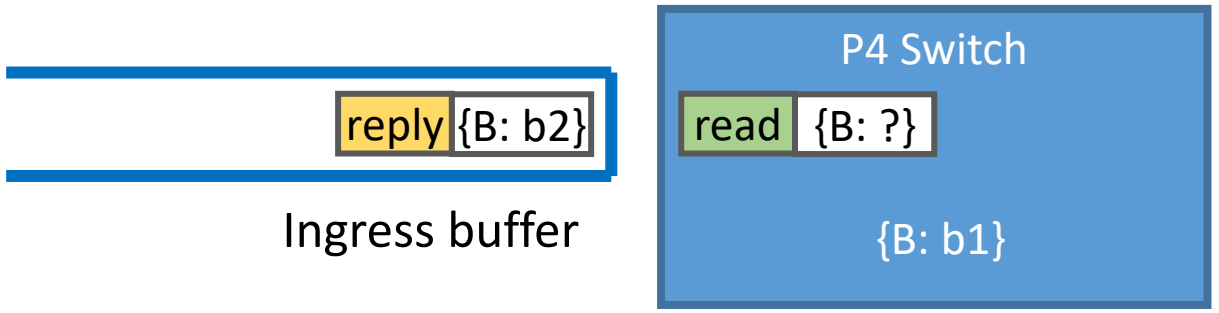



```
for each pkt(K, V, Type):
    idx = Hash(K)
    switch Type:
        case Write:
            forward(K, V, Write)
        case Reply:
            update(K, V, idx)
            forward(K, V, Reply)
        case Read:
            res, V = find(K, idx)
            if res == NOT_FOUND:
                forward(K, _, Read)
            else:
                recirculate(K, V, Reply)
```

Recirculate the processed packet to ingress buffer and let pipeline process the packet as a Reply

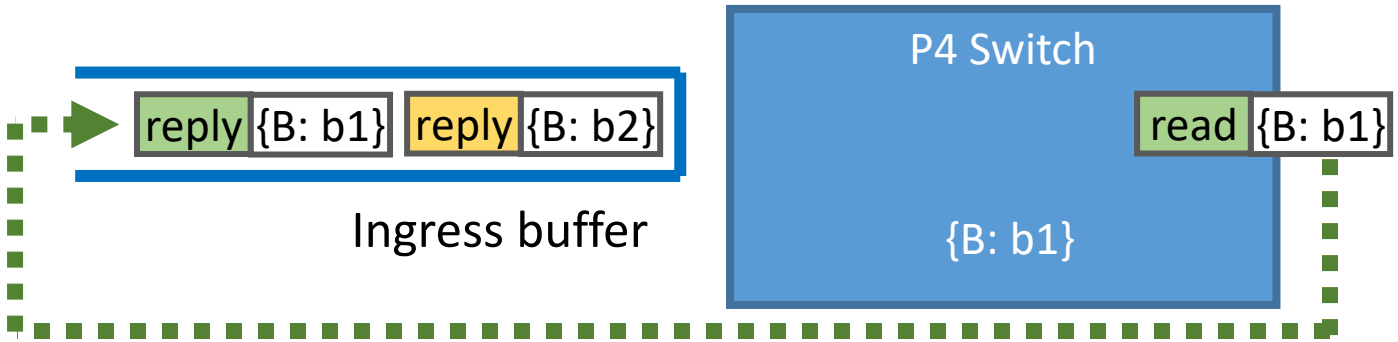
Attempt 2: Observation

- A serious problem of recirculation
  - The switch processes read B and found value b1



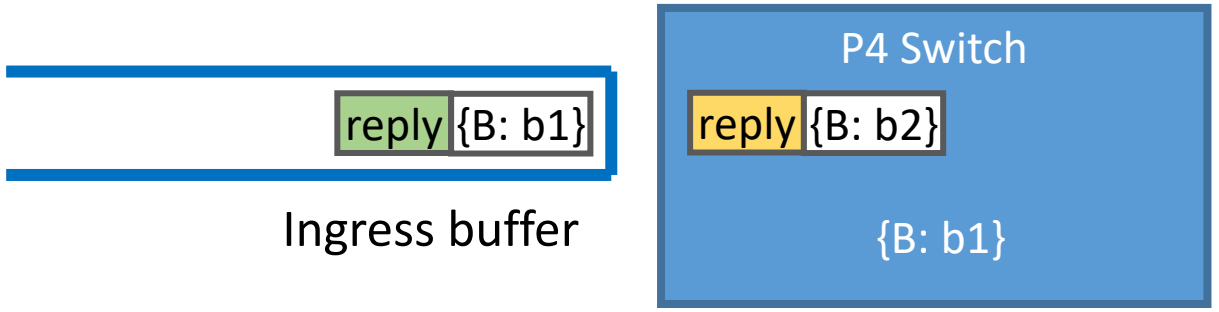
Attempt 2: Observation

- A serious problem
  - The switch processes read B and found value b1
  - Read B is recirculated as reply B:b1 and put in the buffer



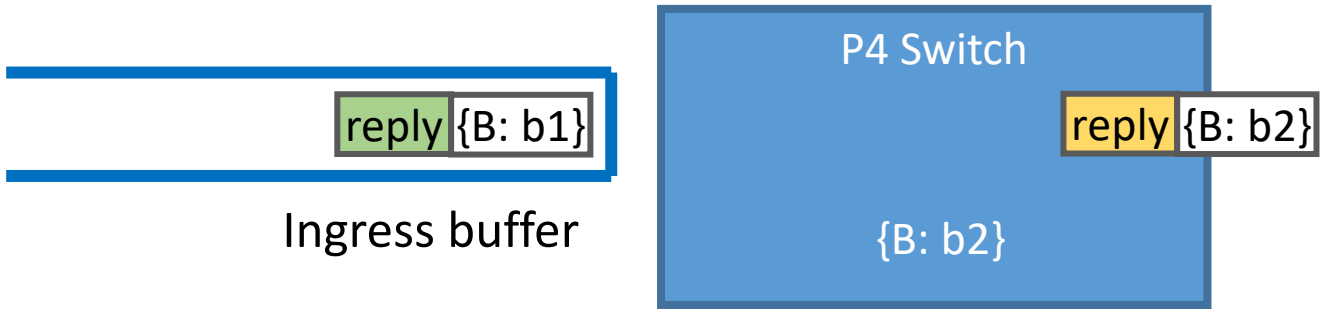
Attempt 2: Observation

- A serious problem
  - The switch processes read B and found value b1
  - Read B is recirculated as reply B:b1 and put in the buffer
  - The switch processes another reply B:b2 from server update



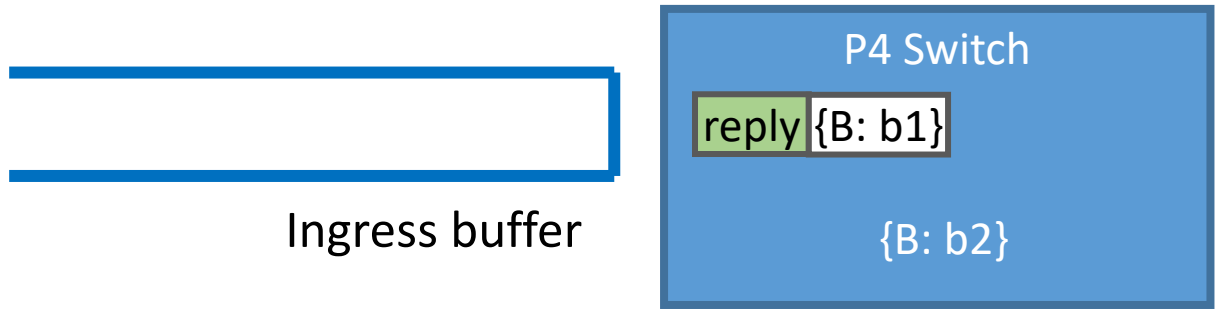
Attempt 2: Observation

- A serious problem
  - The switch processes read B and found value b1
  - Read B is recirculated as reply B:b1 and put in the buffer
  - The switch processes another reply B:b2 from server update
  - Now the switch has B:b2



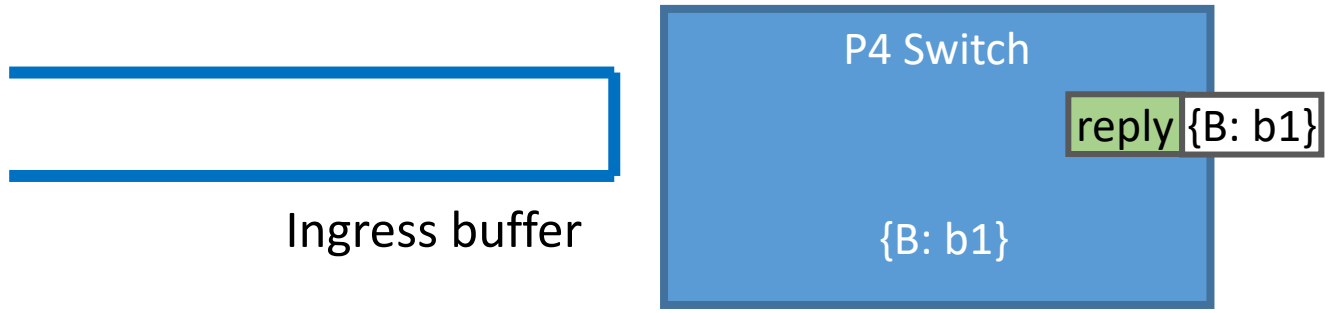
Attempt 2: Observation

- A serious problem
  - The switch processes read B and found value b1
  - Read B is recirculated as reply B:b1 and put in the buffer
  - The switch processes another reply B:b2 from server update
  - Now the switch has B:b2
  - The switch processes recirculated reply B:b1



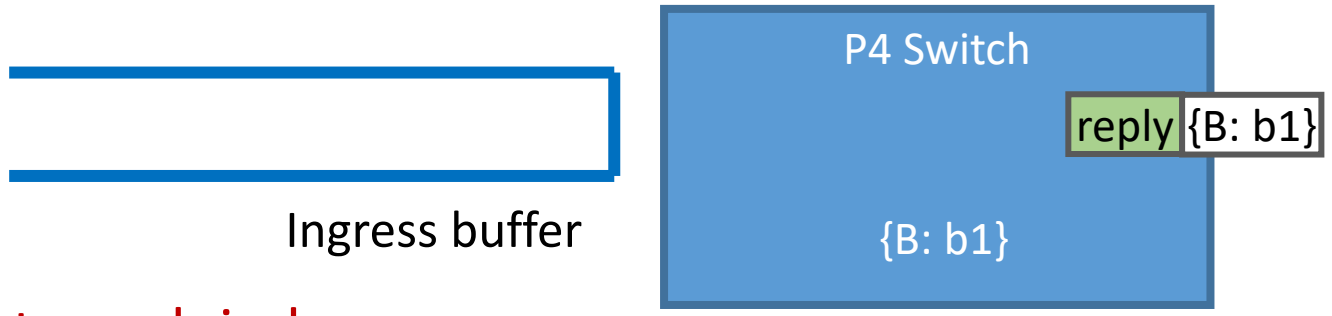
Attempt 2: Observation

- A serious problem
  - The switch processes read B and found value b1
  - Read B is recirculated as reply B:b1 and put in the buffer
  - The switch processes another reply B:b2 from server update
  - Now the switch has B:b2
  - The switch processes recirculated reply B:b1
  - The switch now has stale value B:b1!



Attempt 2: Observation

- A serious problem
  - The switch processes read B and found value b1
  - Read B is recirculated as reply B:b1 and put in the buffer
  - The switch processes another reply B:b2 from server update
  - Now the switch has B:b2
  - The switch processes recirculated reply B:b1
  - The switch now has stale value B:b1!



Possible Solution: add version number to each index

Attempt 3: Version Number of each index

- A P4 Switch has an array `version[]` that record the current version of indices of LRU lists
- When receiving Reply packet from server (not recirculated), increment `version[idx]` and update LRU list
- When receiving Read packet, read `version[idx]`. If KV is in the cache, recirculate the packet including the read version number
- When receiving recirculated Reply packet, check if `version[idx]` is still the same. If same, update the LRU list and forward to client. Otherwise:
  - Option 1: treat the packet as Read packet and process it
  - Option 2: do nothing, just forward the packet

Attempt 3: Version Number of each index

- Which one has a lower overhead?
- Treat the recirculated packet as Read packet and process it again
  - Pro: The target KV is guaranteed to be promoted to the front of the list
  - Con: Need more recirculations, more bandwidth overhead
- Just forward the packet
  - Pro: The packet is recirculated only once
  - Con: Target KV is not promoted therefore more likely to be evicted

Conclusion & Lessons

- It’s possible to implement LRU cache on P4 Switches
- Performance: I can only run P4 program on simulated environment (mininet and bmv2) where performance is far lower than hardware (~80,000 pps vs. ~5,000,000,000 pps)
- To design programs on P4 switch:
  - Aware of which operation should be done in which pipeline stage
  - Avoid writing programs with variable runtime or with non-deterministic loop
  - If using recirculation, the order of the packets should be considered
- Source code and my demo video:
  - https://github.com/RaymondHuang210129/P4KVCache

Reference:

- Hauser, Frederik, et al. "A survey on data plane programming with p4: Fundamentals, advances, and applied research." *arXiv preprint arXiv:2101.10632* (2021).
- Bmv2 Official Performance Document. https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md

Related Works:

- Jin, Xin, et al. "Netcache: Balancing key-value stores with fast in-network caching." *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017.
- 2021 P4 Workshop, Extending P4 to Realize a Scalable Flow Caching Mechanism. https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Angelo-Tulumello-Slides.pdf