Iterator Interface Extended LSM-tree-based KVSSD for Range Queries

Youngjae Kim (PhD)

2023.10.18.

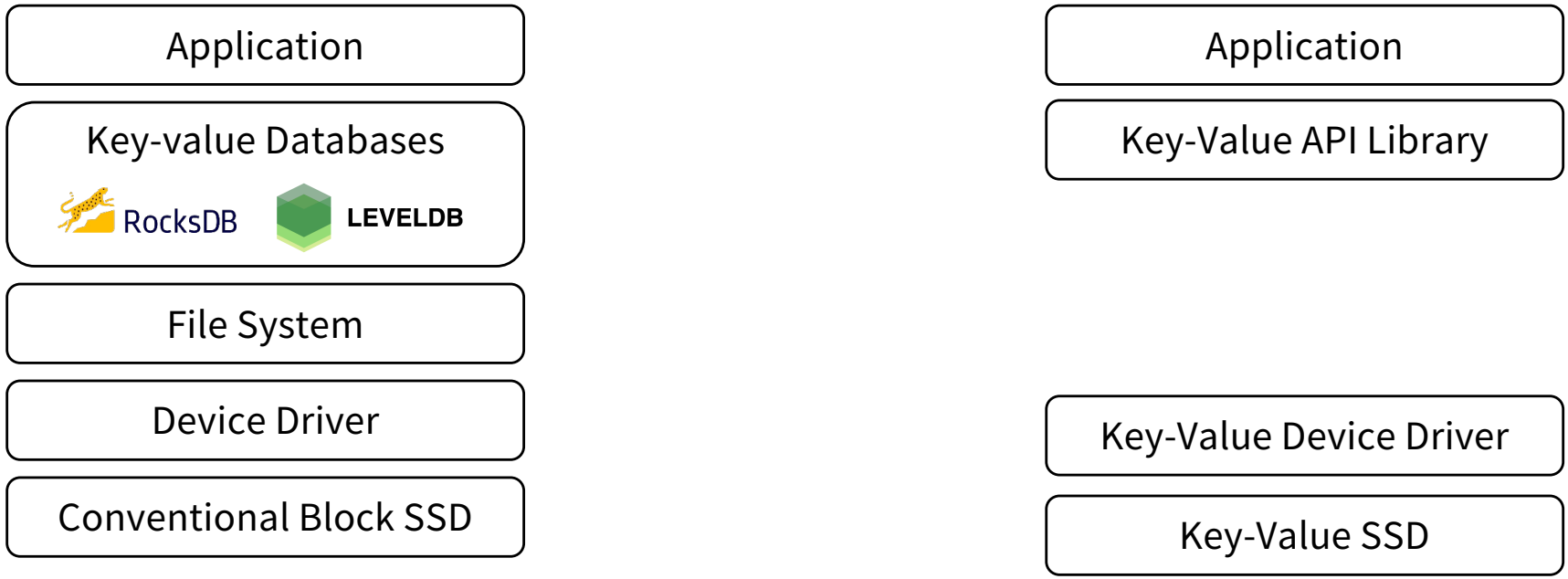
Seungjin Lee, Chang-Gyu Lee, Donghyun Min, Inhyuk Park, Woosuk Chung,
Anand Sivasubramaniam, Youngjae Kim

16th ACM International Systems and Storage Conference (**SYSTOR**) (2023), Haifa, Israel, June 2023

NVRAMOS'23

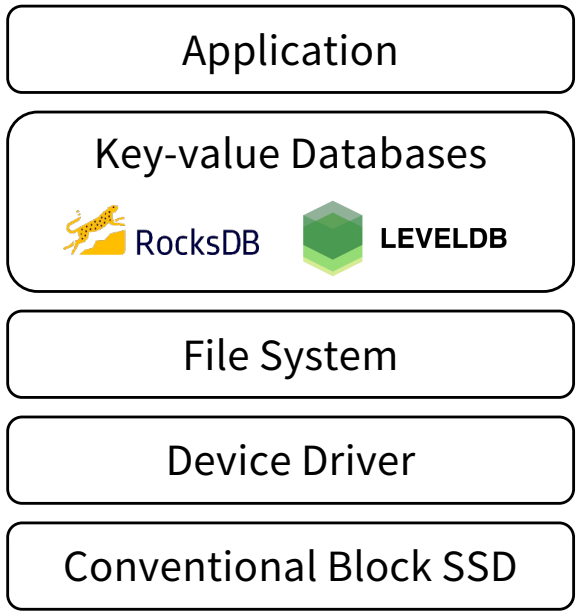
Key-Value SSD

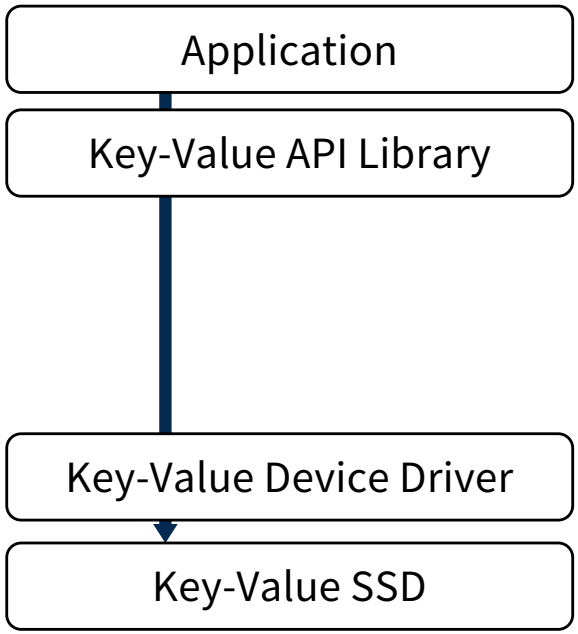
- **Key-Value SSD: Removing the host software I/O stack**



Key-Value SSD

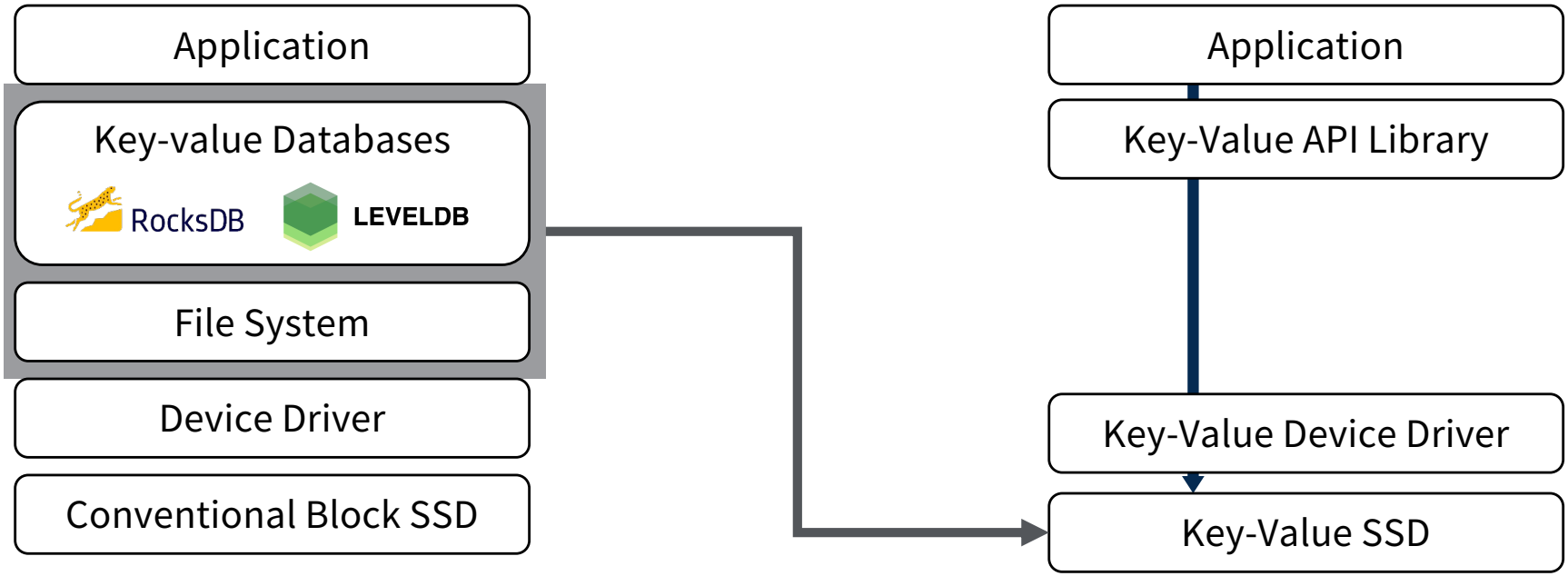
- **Key-Value SSD: Removing the host software I/O stack**





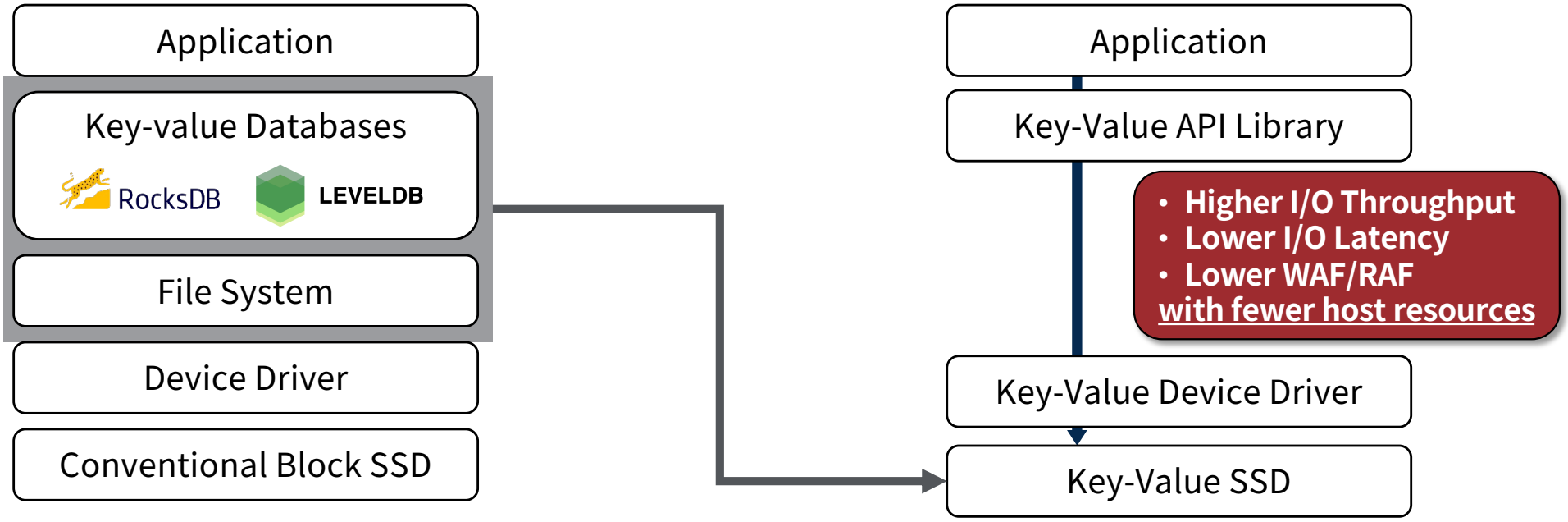
Key-Value SSD

- **Key-Value SSD: Removing the host software I/O stack**



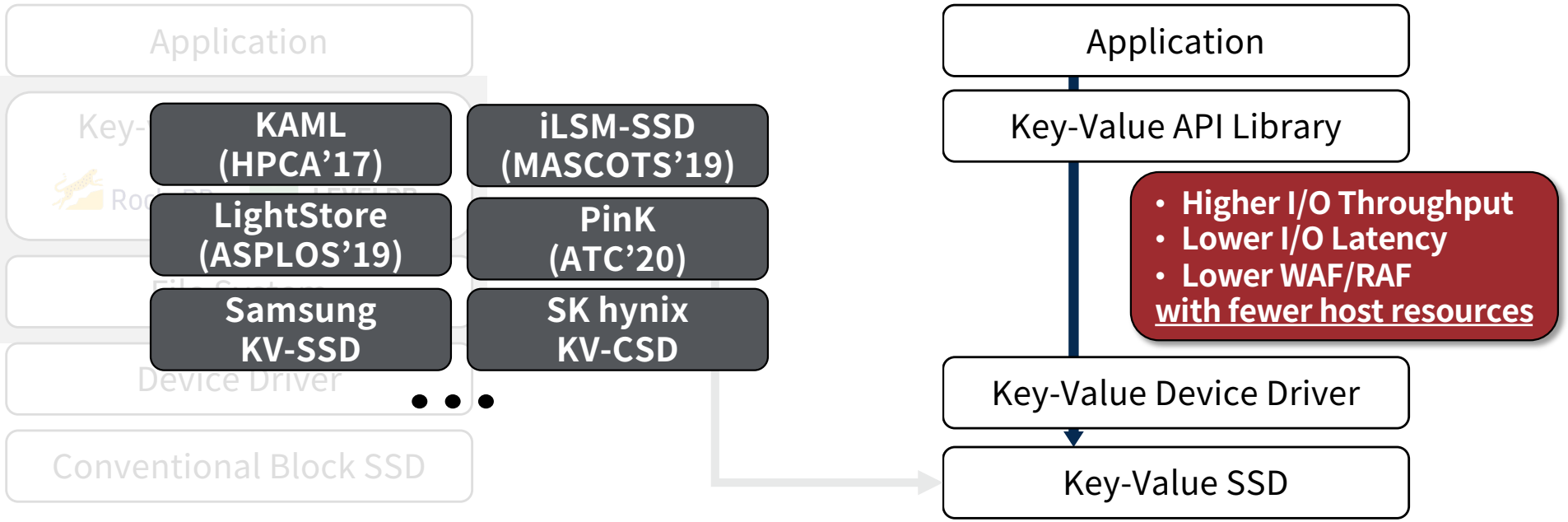
Key-Value SSD

- **Key-Value SSD: Removing the host software I/O stack**



Key-Value SSD

- **Key-Value SSD: Removing the host software I/O stack**



Range Query for KVSSD

[1] PinK: High-speed In-storage Key-value Store with Bounded Tails, USENIX ATC 2020
[2] Modernizing File System through In-Storage Indexing, USENIX OSDI 2021

Range Query for KVSSD

- **Existing KVSSDs mostly focus on Point Query (Put/Get)**
  - Index Pinning[1], Index Compression[2]
  - H/W Accelerator for Compaction[1]
  - …
- **What about Range Query (Seek/Next)?**
  - Using a sorted data structure makes range query implementation simple.
  - However, previous studies have not addressed the design details of range queries.

[1] PinK: High-speed In-storage Key-value Store with Bounded Tails, USENIX ATC 2020
[2] Modernizing File System through In-Storage Indexing, USENIX OSDI 2021

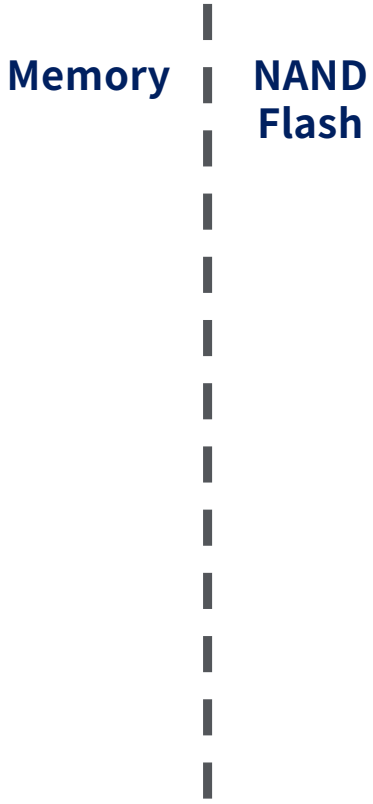
Range Query for KVSSD

- **Existing KVSSDs mostly focus on Point Query (Put/Get)**
  - Index Pinning[1], Index Compression[2]
  - H/W Accelerator for Compaction[1]
  - …
- **What about Range Query (Seek/Next)?**
  - Using a sorted data structure makes range query implementation simple.
  - However, previous studies have not addressed the design details of range queries.

[1] PinK: High-speed In-storage Key-value Store with Bounded Tails, USENIX ATC 2020
[2] Modernizing File System through In-Storage Indexing, USENIX OSDI 2021

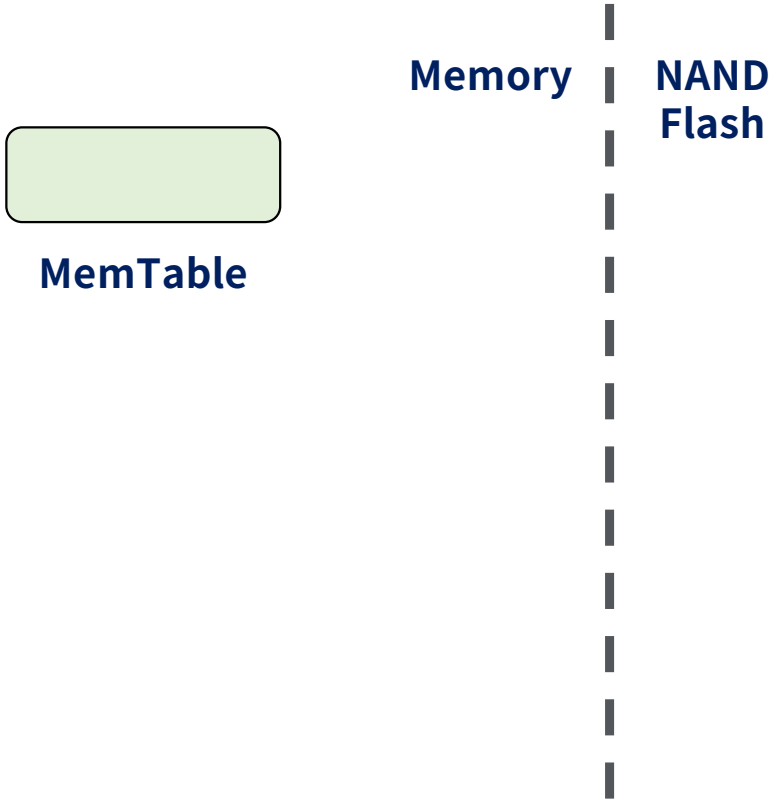
Key-Value SSD Internals

- **LSM-tree-based Key-Value SSD**



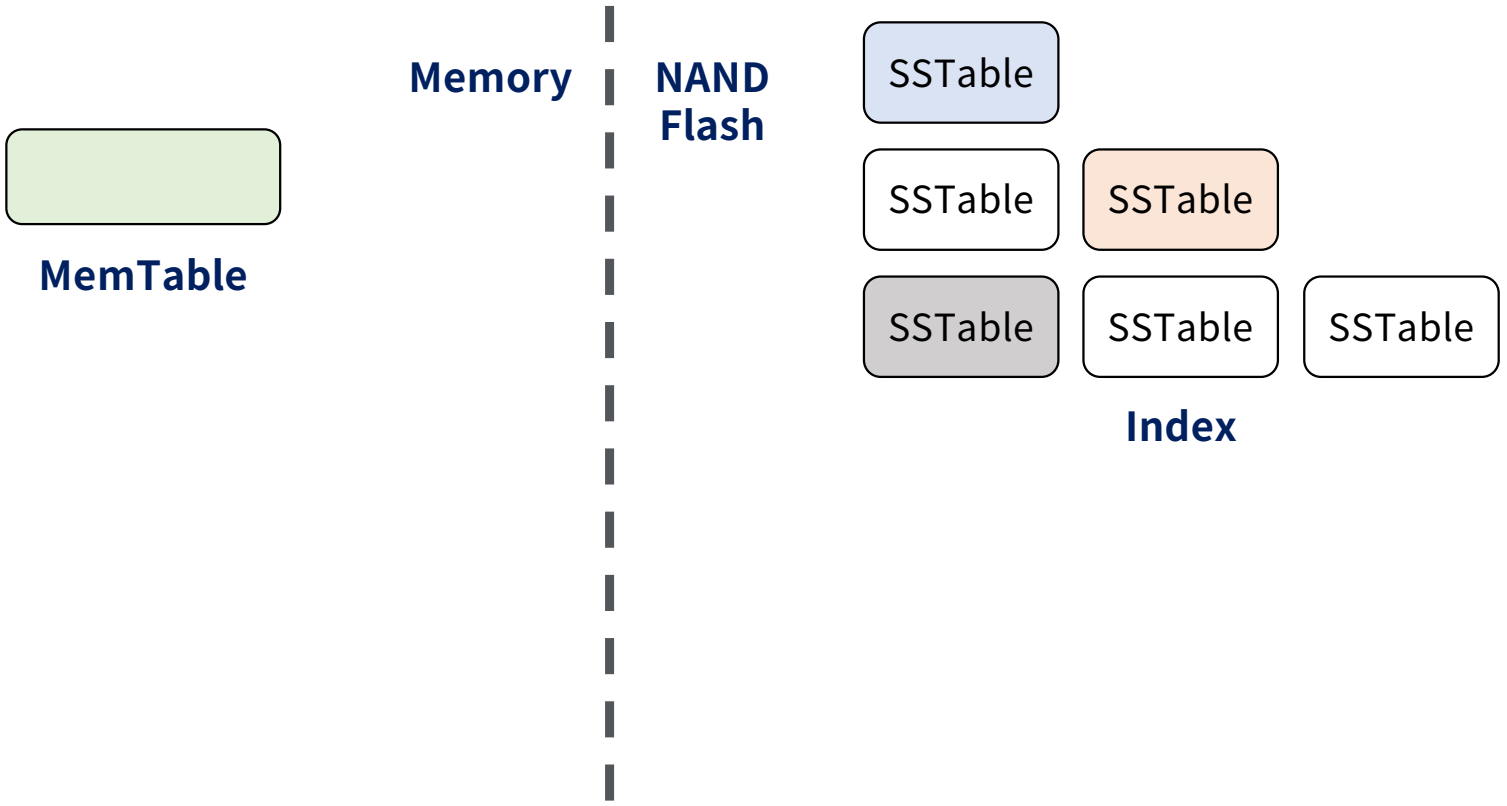
Key-Value SSD Internals

- **LSM-tree-based Key-Value SSD**



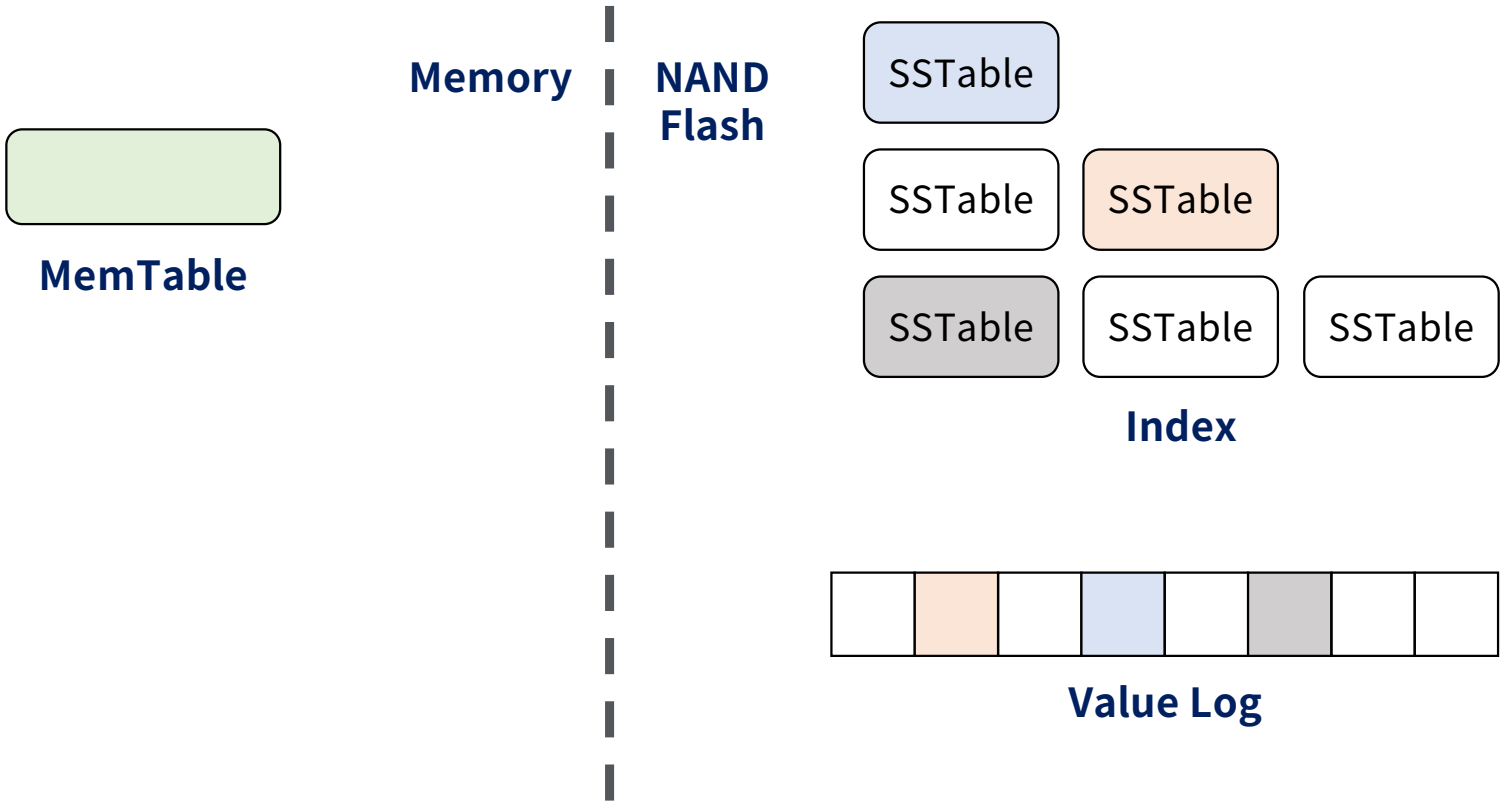
Key-Value SSD Internals

- **LSM-tree-based Key-Value SSD**



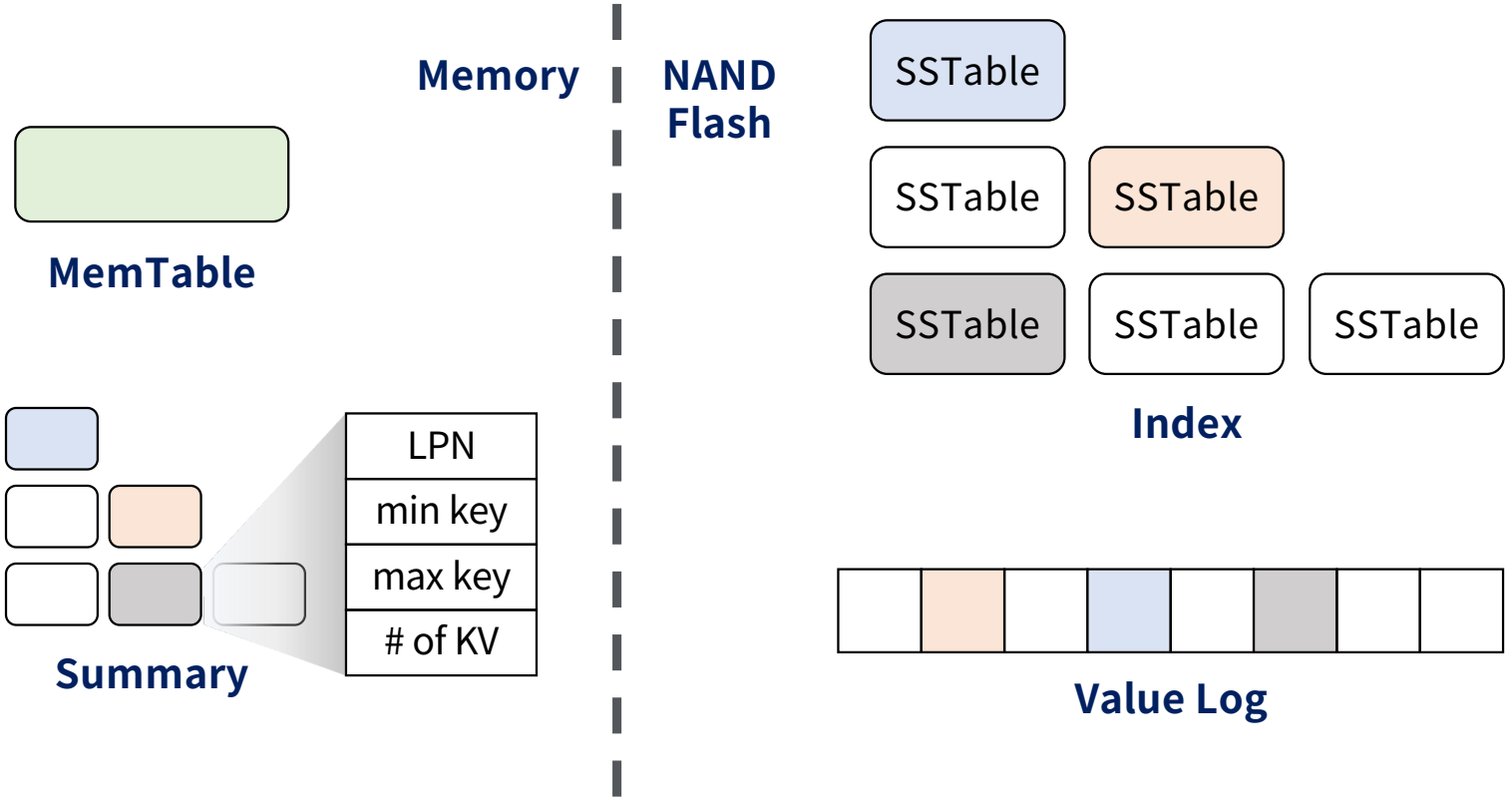
Key-Value SSD Internals

- **LSM-tree-based Key-Value SSD**



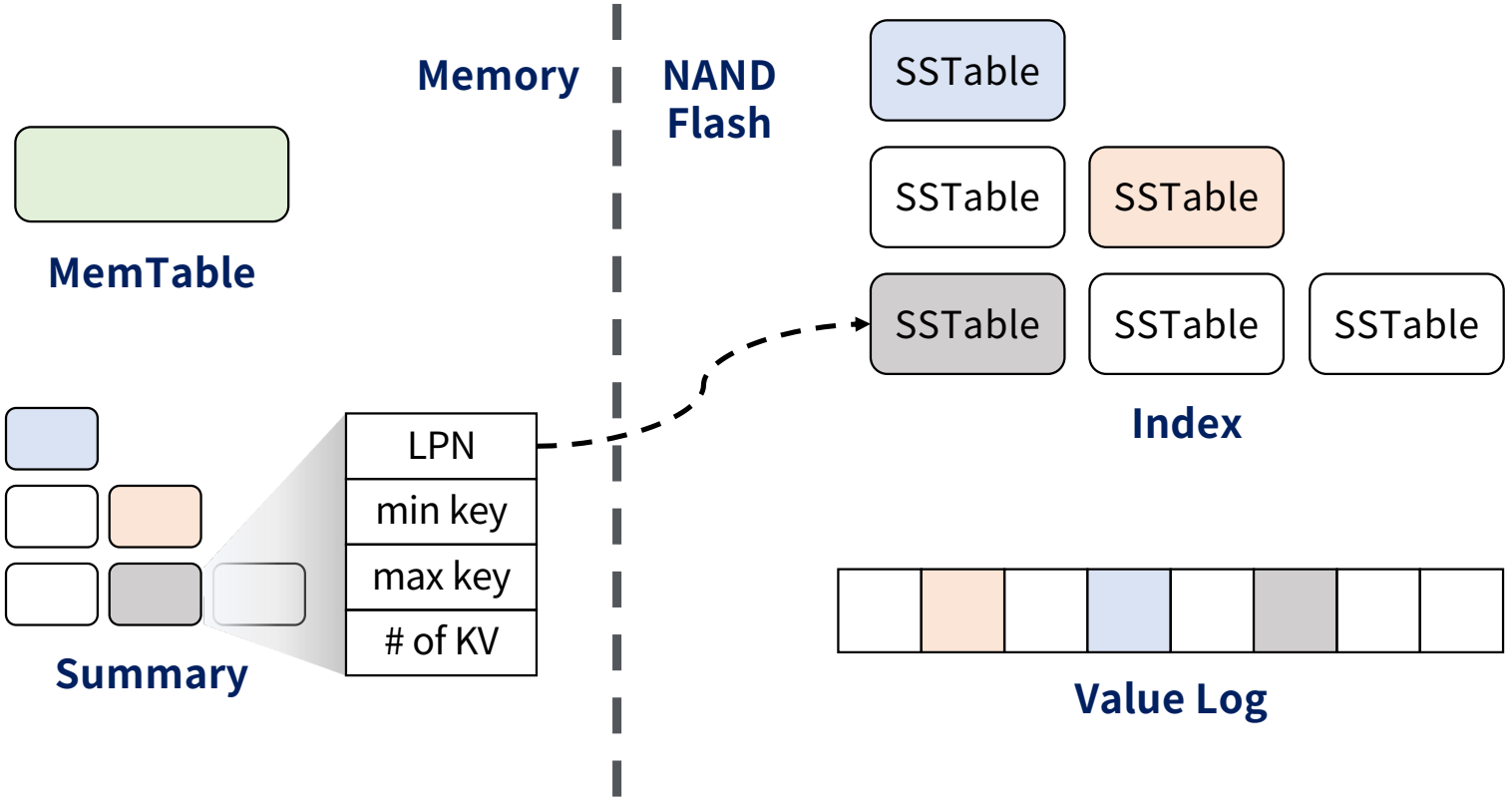
Key-Value SSD Internals

- **LSM-tree-based Key-Value SSD**



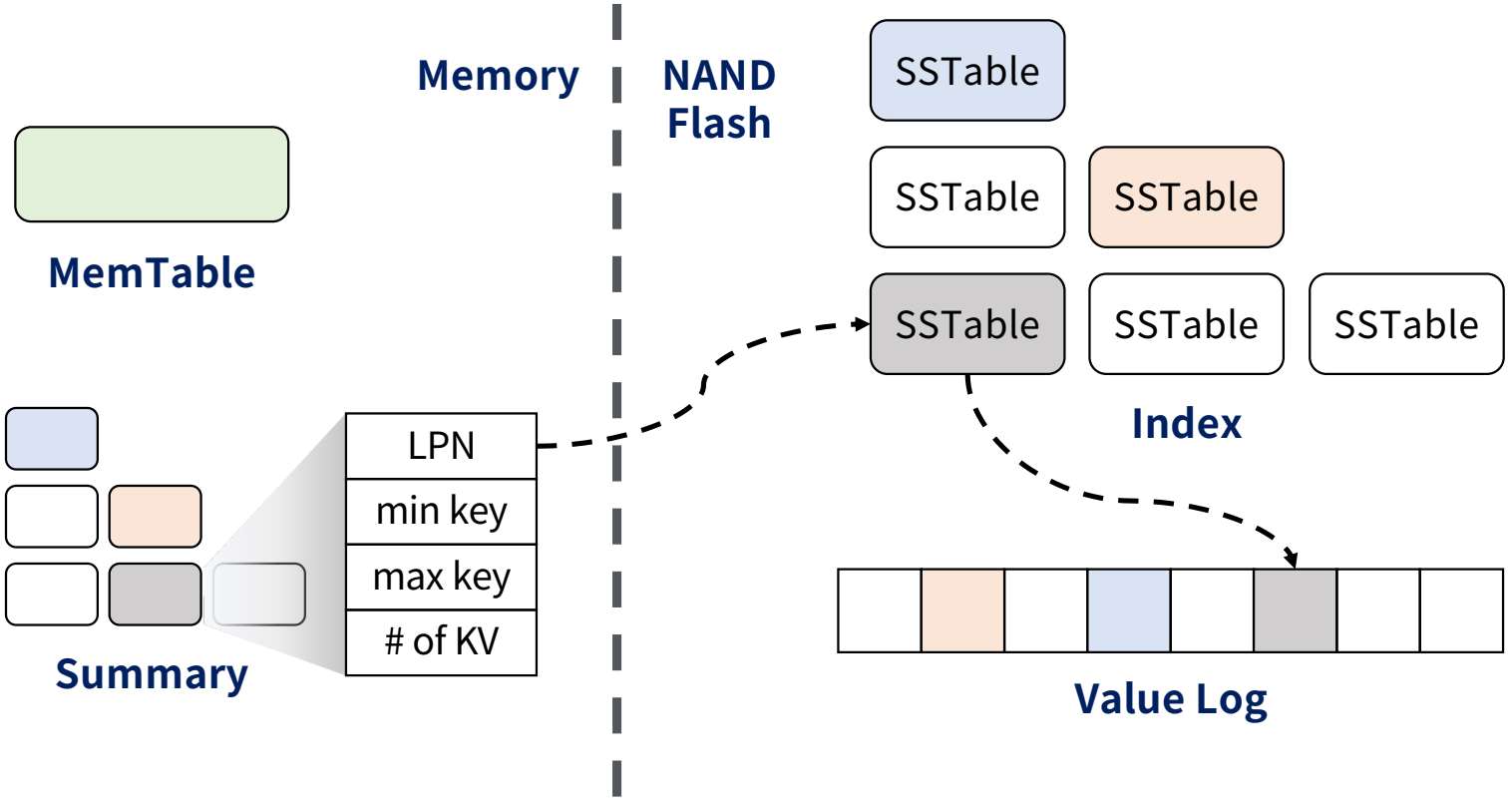
Key-Value SSD Internals

- **LSM-tree-based Key-Value SSD**

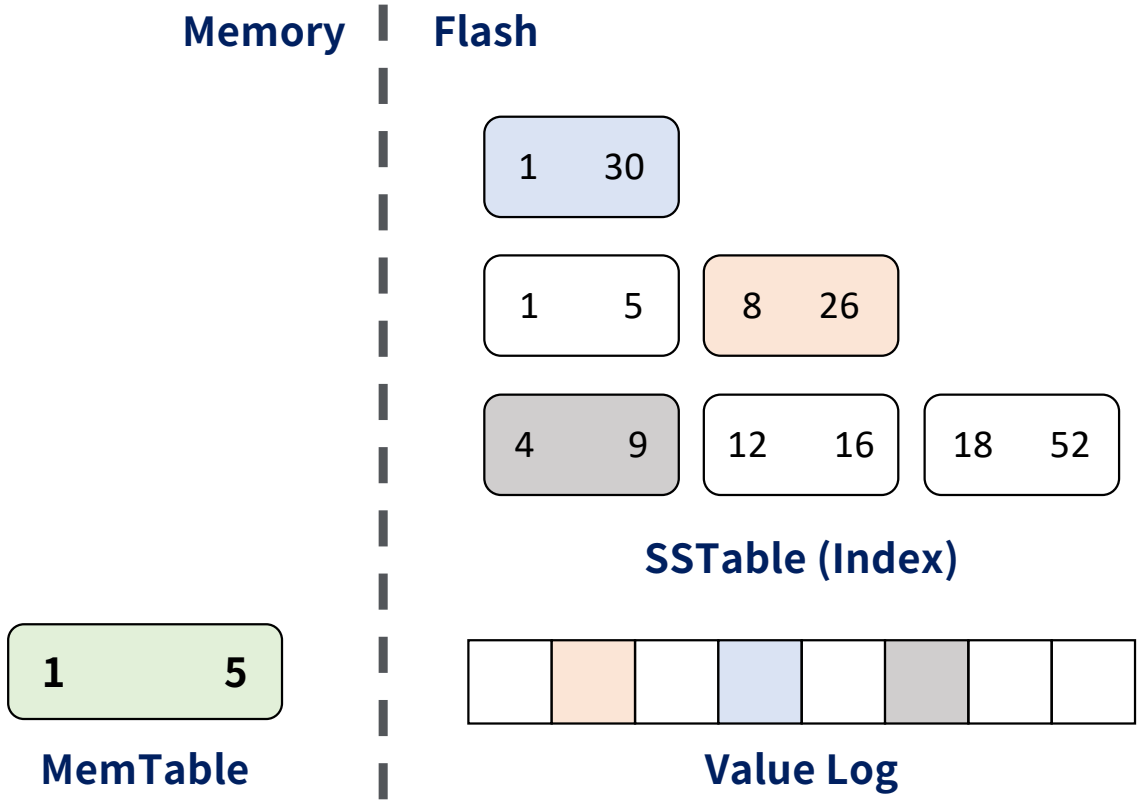


Key-Value SSD Internals

- **LSM-tree-based Key-Value SSD**

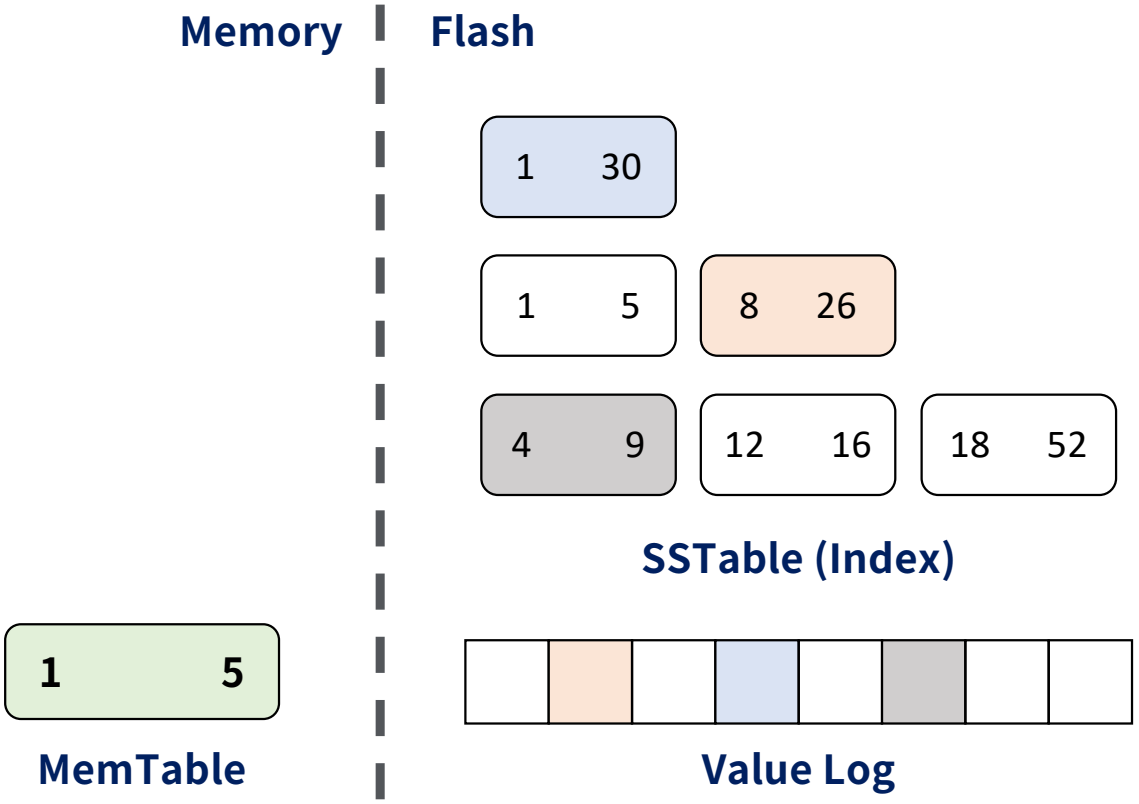


Range Query in LSM-tree-based KVSSD



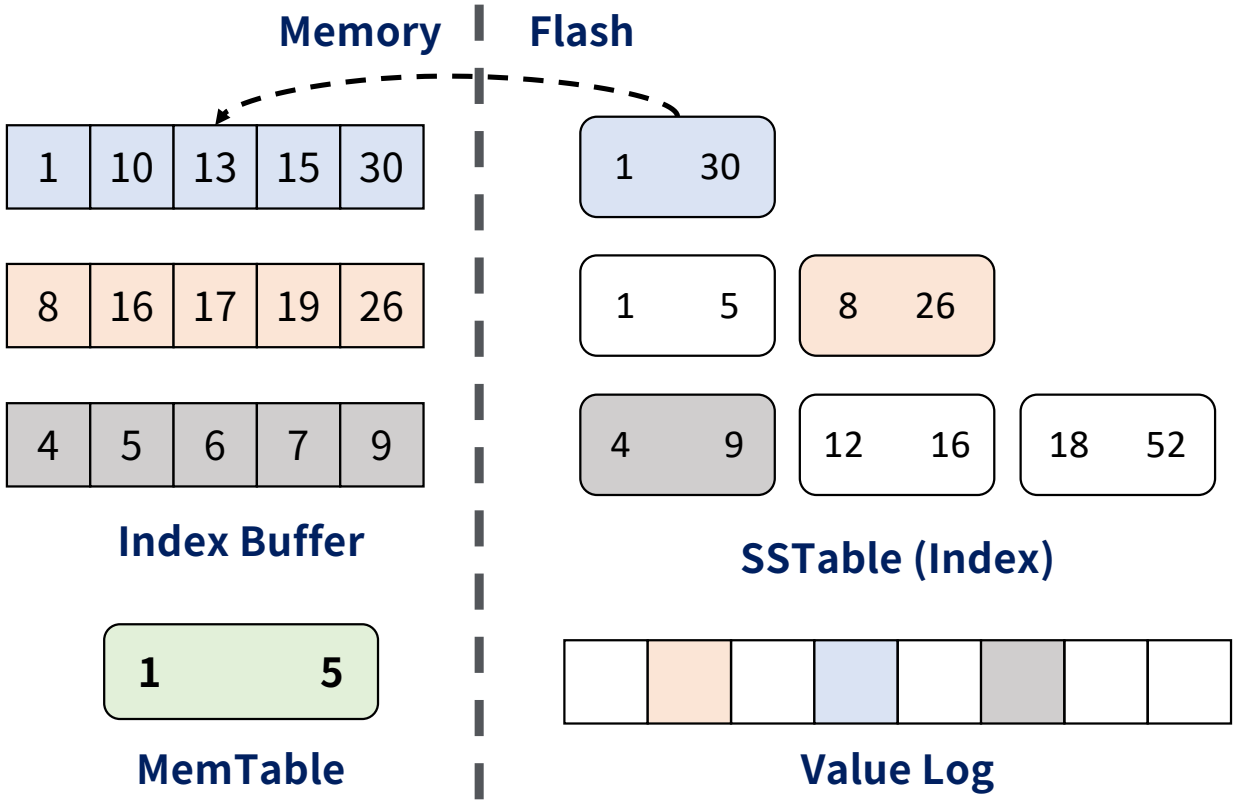
Range Query in LSM-tree-based KVSSD

- **Range Queries are often served as Iterator Interface**
  - Seek() and Next()



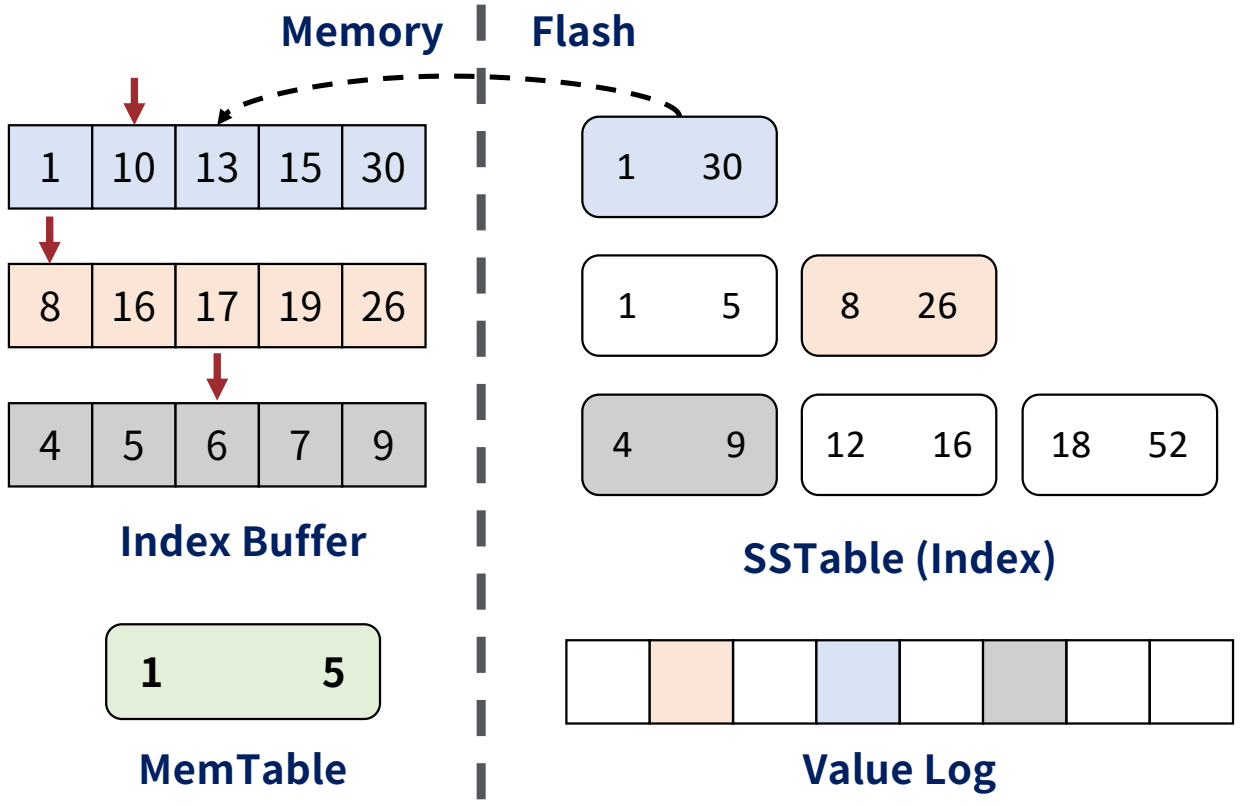
Range Query in LSM-tree-based KVSSD

- **Range Queries are often served as Iterator Interface**
  - Seek() and Next()



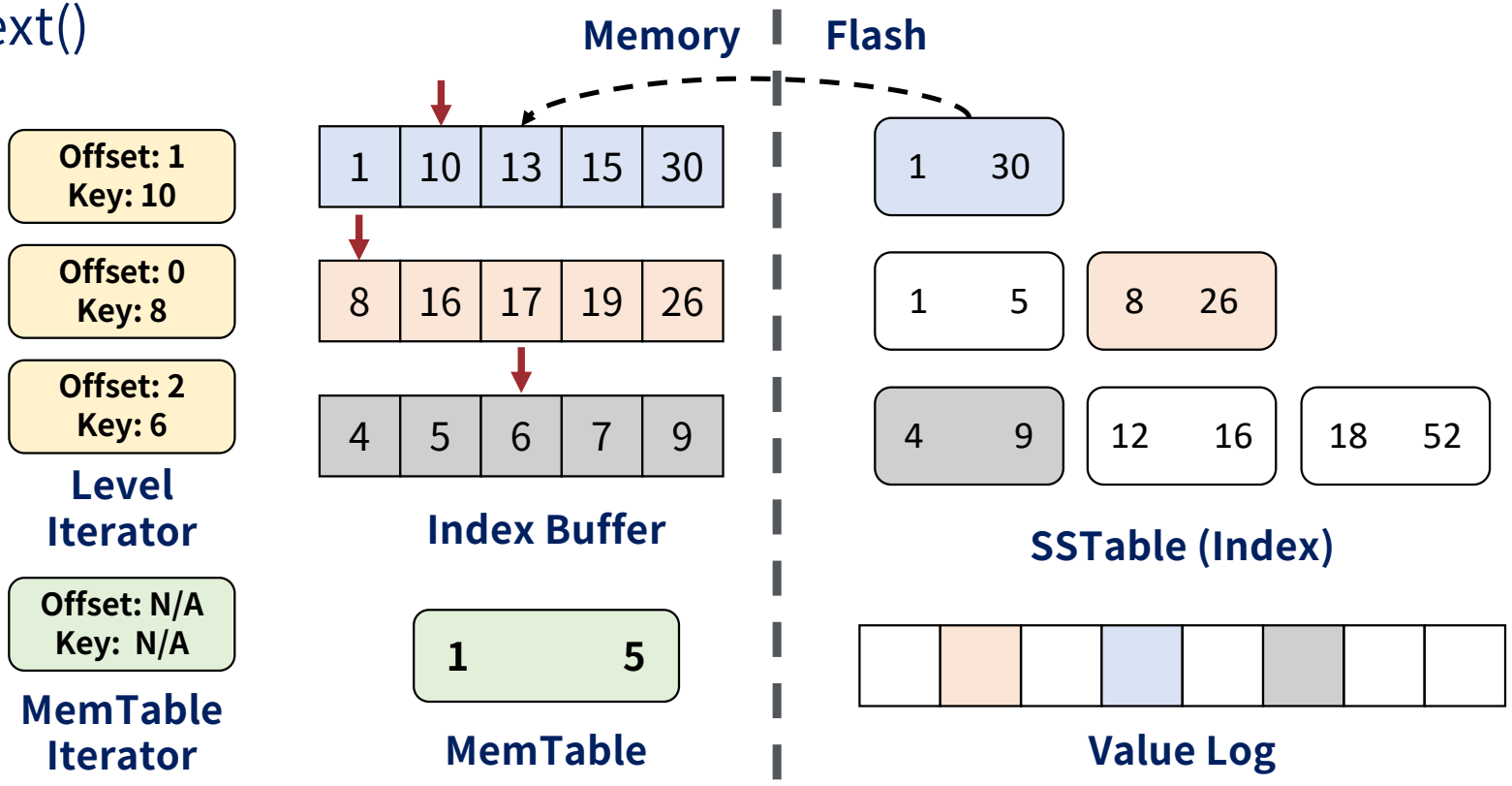
Range Query in LSM-tree-based KVSSD

- **Range Queries are often served as Iterator Interface**
  - Seek() and Next()



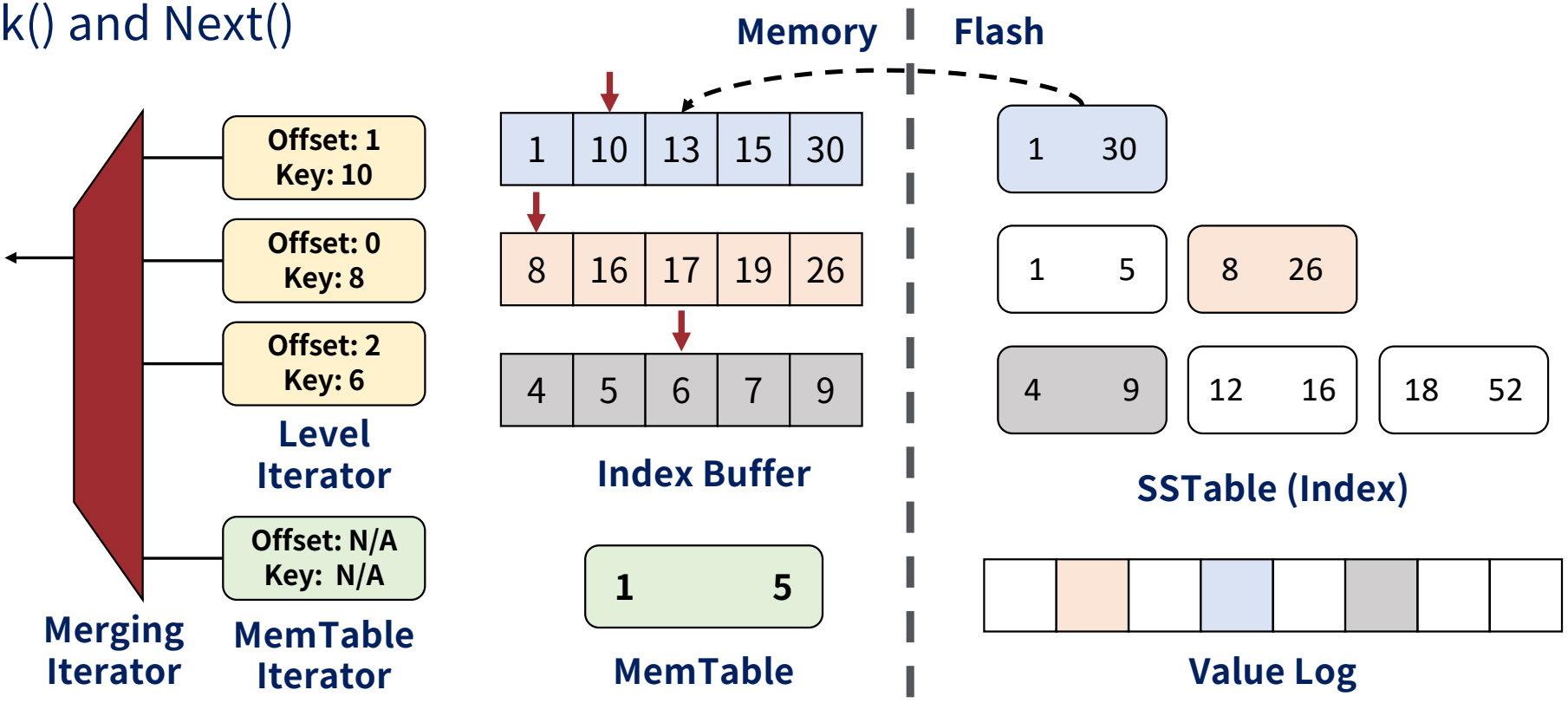
Range Query in LSM-tree-based KVSSD

- **Range Queries are often served as Iterator Interface**
  - Seek() and Next()



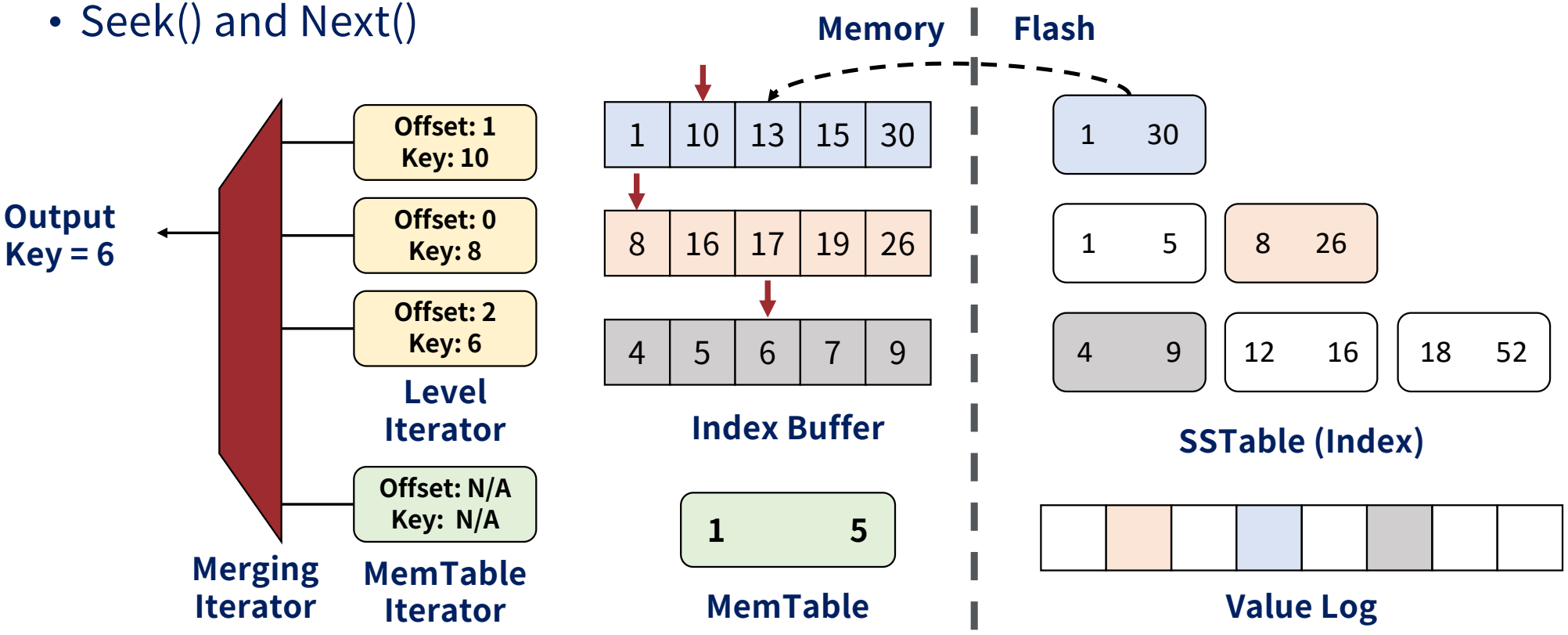
Range Query in LSM-tree-based KVSSD

- **Range Queries are often served as Iterator Interface**
  - Seek() and Next()



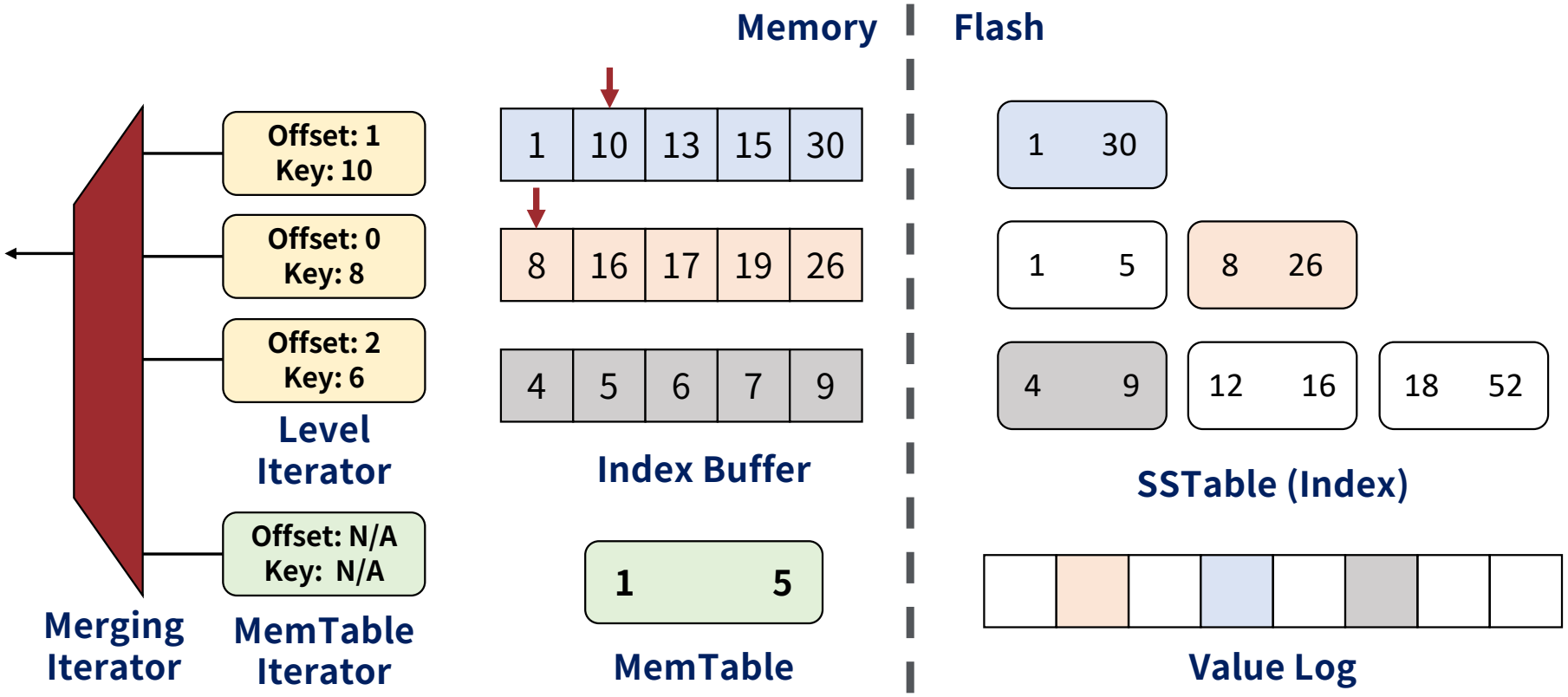
Range Query in LSM-tree-based KVSSD

- **Range Queries are often served as Iterator Interface**
  - Seek() and Next()



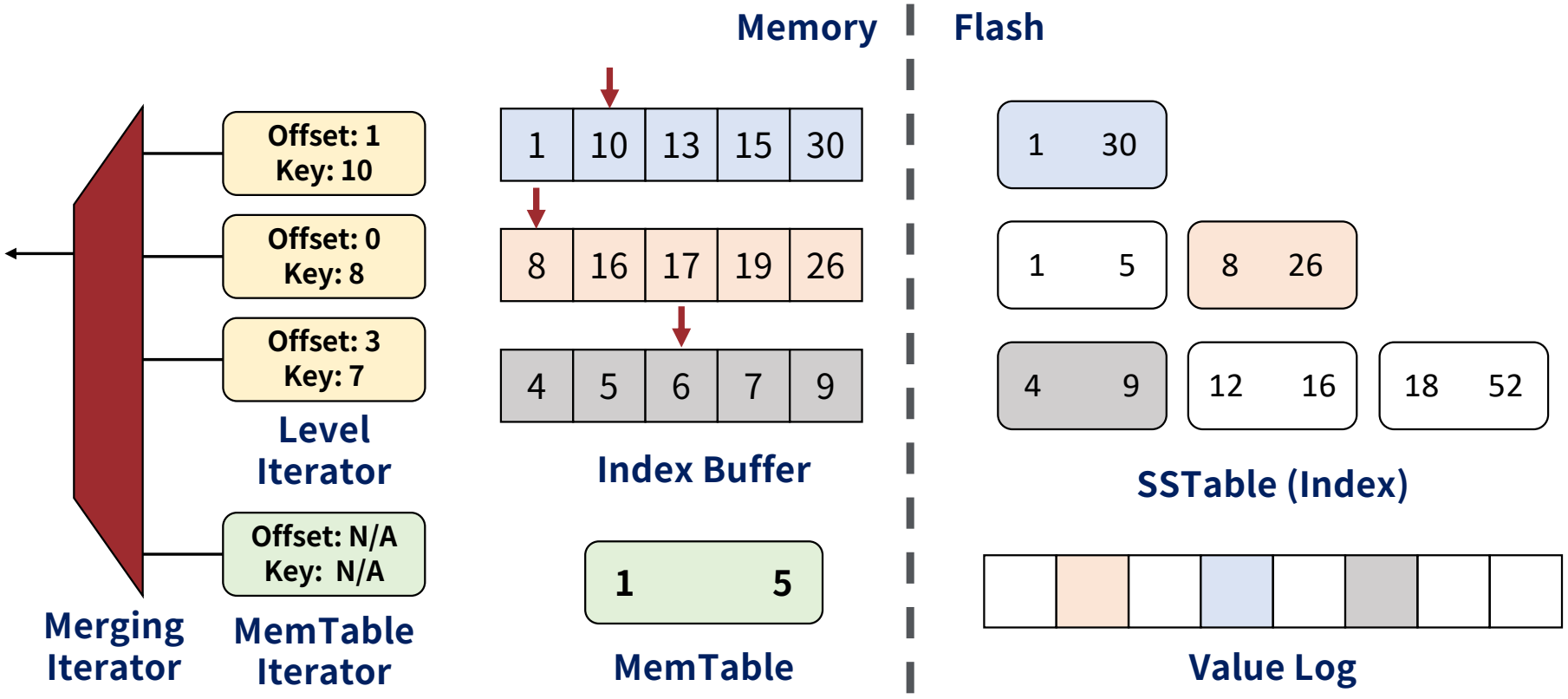
Range Query in LSM-tree-based KVSSD

- **On Next()**



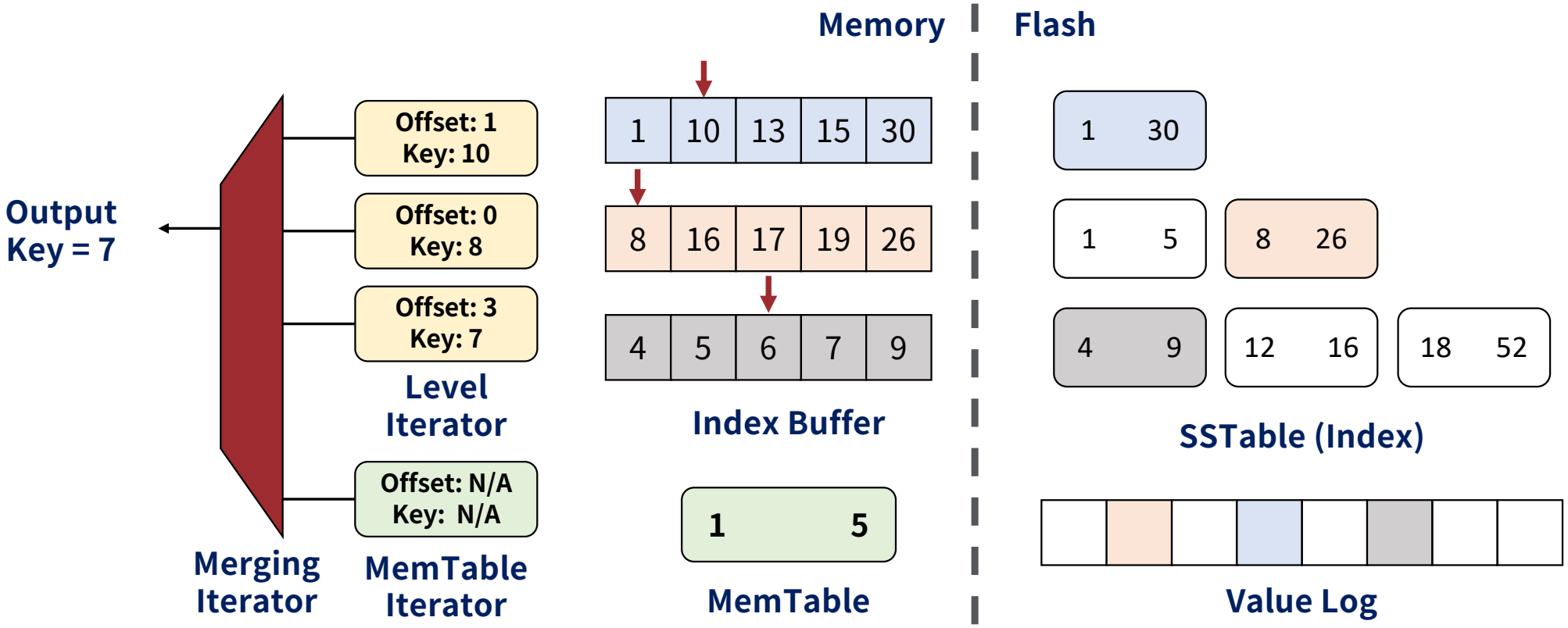
Range Query in LSM-tree-based KVSSD

- **On Next()**



Range Query in LSM-tree-based KVSSD

- **On Next()**



Problem Definition

Problem Definition

- **Problem #1 – Inconsistent Range Query**
  - Range queries are executed through **multiple iterator commands**
  - During range queries, LSM-tree can change by Put, Delete commands
  - How can the change in LSM-tree structure be handled?
- **Problem #2 – Long Tail Latency Problem**
  - During range queries, Iterator interface sometimes requires Index Read
  - This NAND access (Index Read) incurs long tail latency
- **Problem #3 – Poor Range Query Performance**
  - Every Seek, Next command entails Value Read from Value Log
  - This NAND Access (Value Read) incurs poor overall performance

Problem Definition

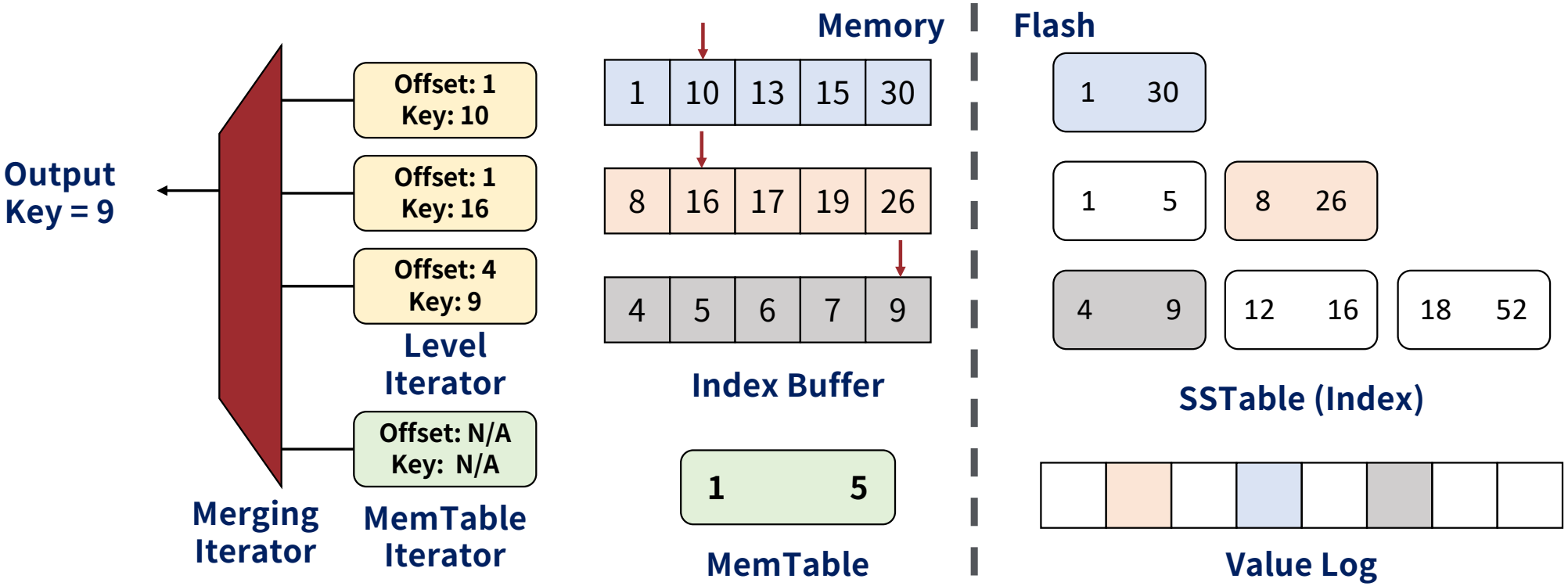
- **Problem #1 – Inconsistent Range Query**
  - Range queries are executed through **multiple iterator commands**
  - During range queries, LSM-tree can change by Put, Delete commands
  - How can the change in LSM-tree structure be handled?
- **Problem #2 – Long Tail Latency Problem**
  - During range queries, Iterator interface sometimes requires Index Read
  - This NAND access (Index Read) incurs long tail latency
- **Problem #3 – Poor Range Query Performance**
  - Every Seek, Next command entails Value Read from Value Log
  - This NAND Access (Value Read) incurs poor overall performance

Problem Definition

- **Problem #1 – Inconsistent Range Query**
  - Range queries are executed through **multiple iterator commands**
  - During range queries, LSM-tree can change by Put, Delete commands
  - How can the change in LSM-tree structure be handled?
- **Problem #2 – Long Tail Latency Problem**
  - During range queries, Iterator interface sometimes requires Index Read
  - This NAND access (Index Read) incurs long tail latency
- **Problem #3 – Poor Range Query Performance**
  - Every Seek, Next command entails Value Read from Value Log
  - This NAND Access (Value Read) incurs poor overall performance

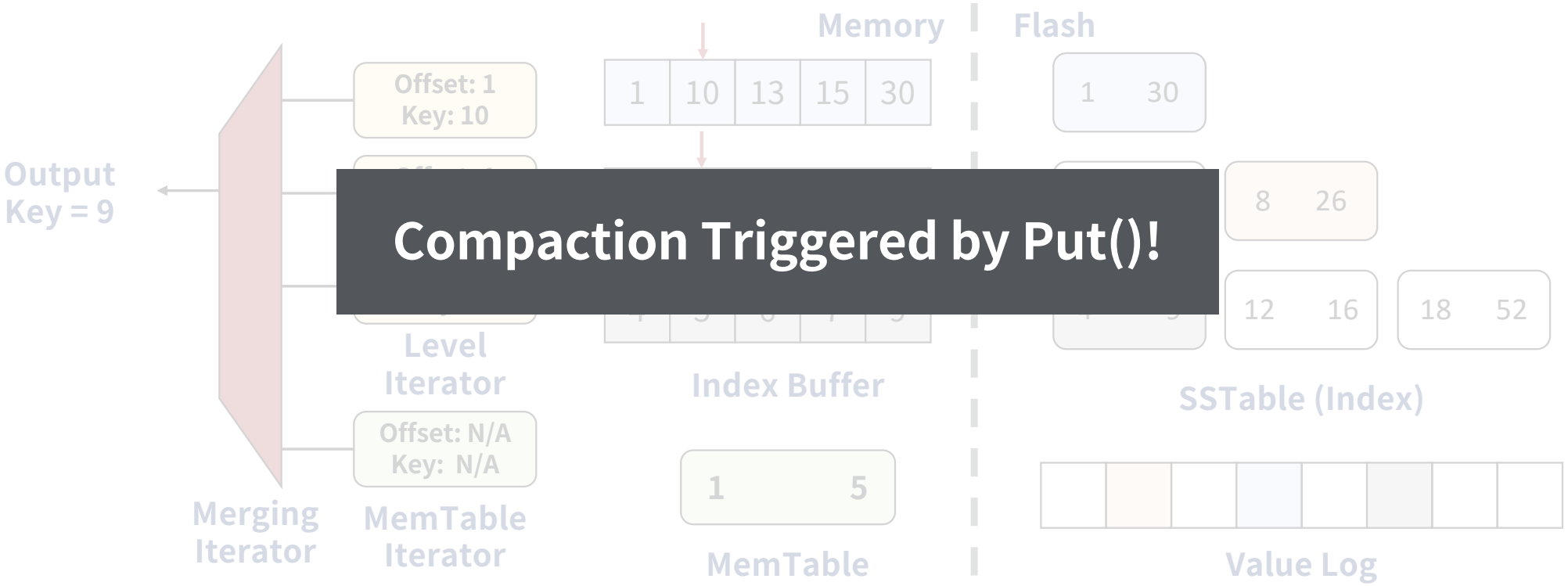
Problem #1 - Versioning

- **Versioning Problem**
  - Put(), Delete() can be issued in the middle of range query



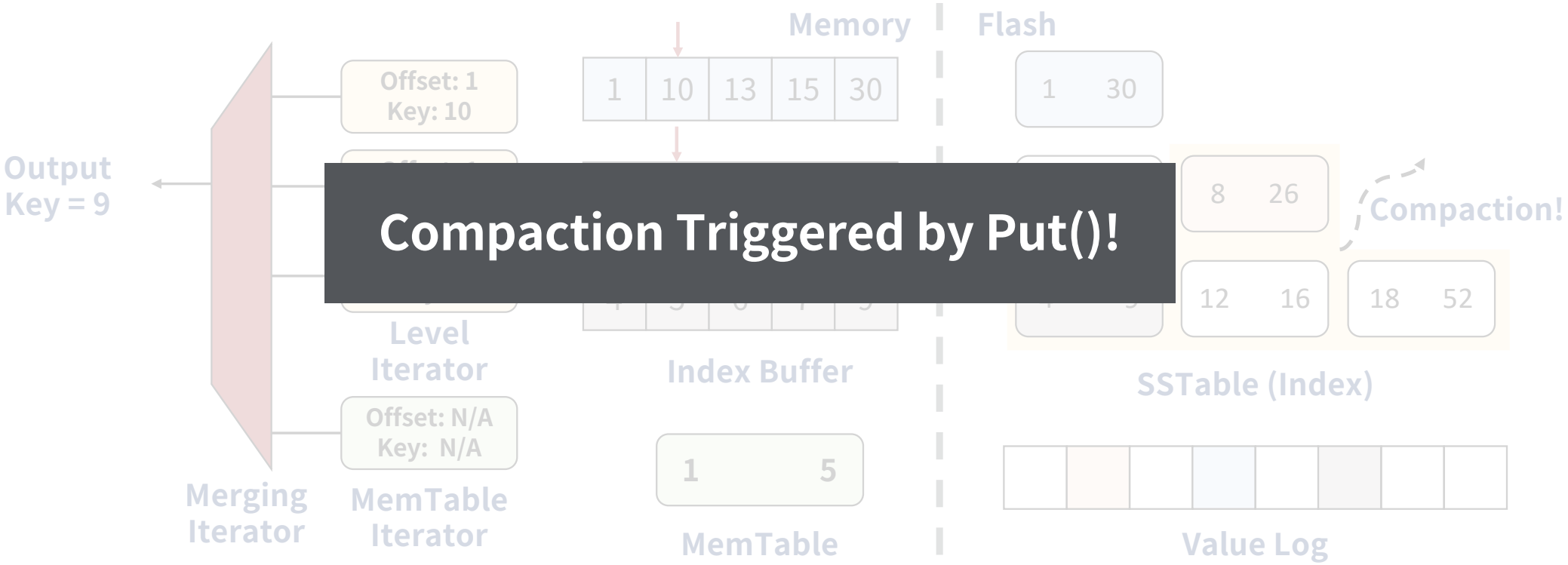
Problem #1 - Versioning

- Versioning Problem
  - Put(), Delete() can be issued in the middle of range query



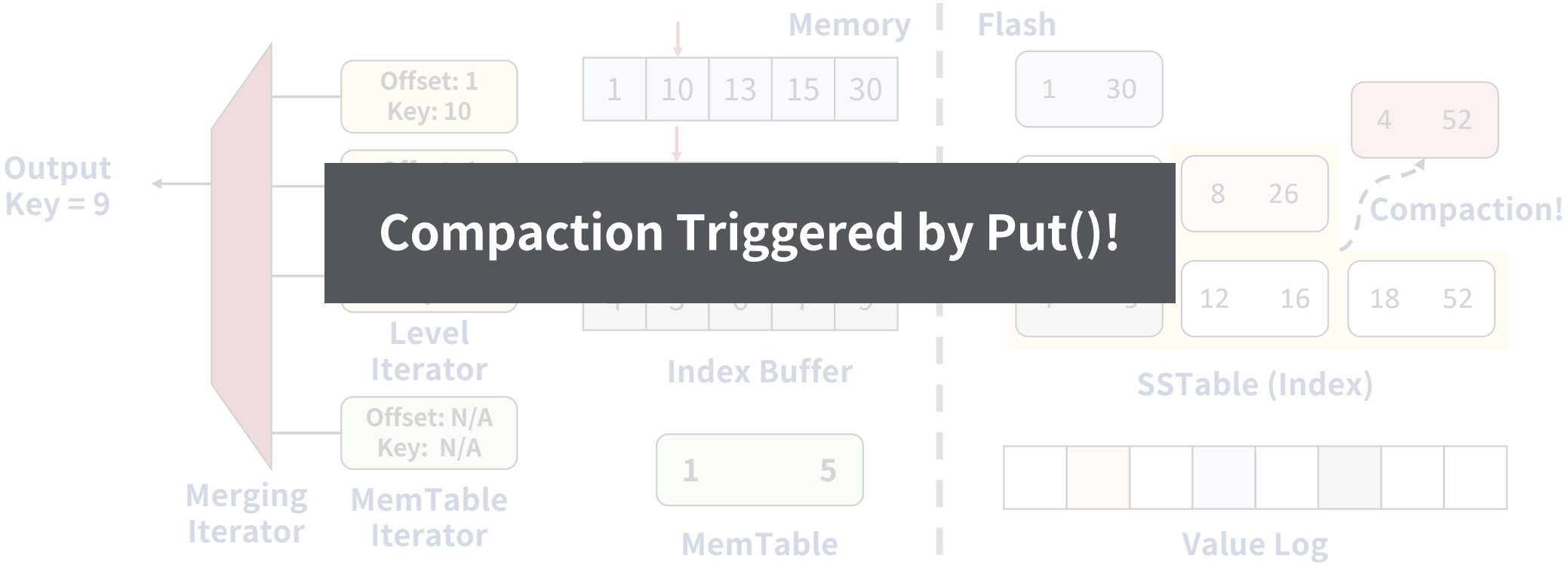
Problem #1 - Versioning

- Versioning Problem
  - Put(), Delete() can be issued in the middle of range query



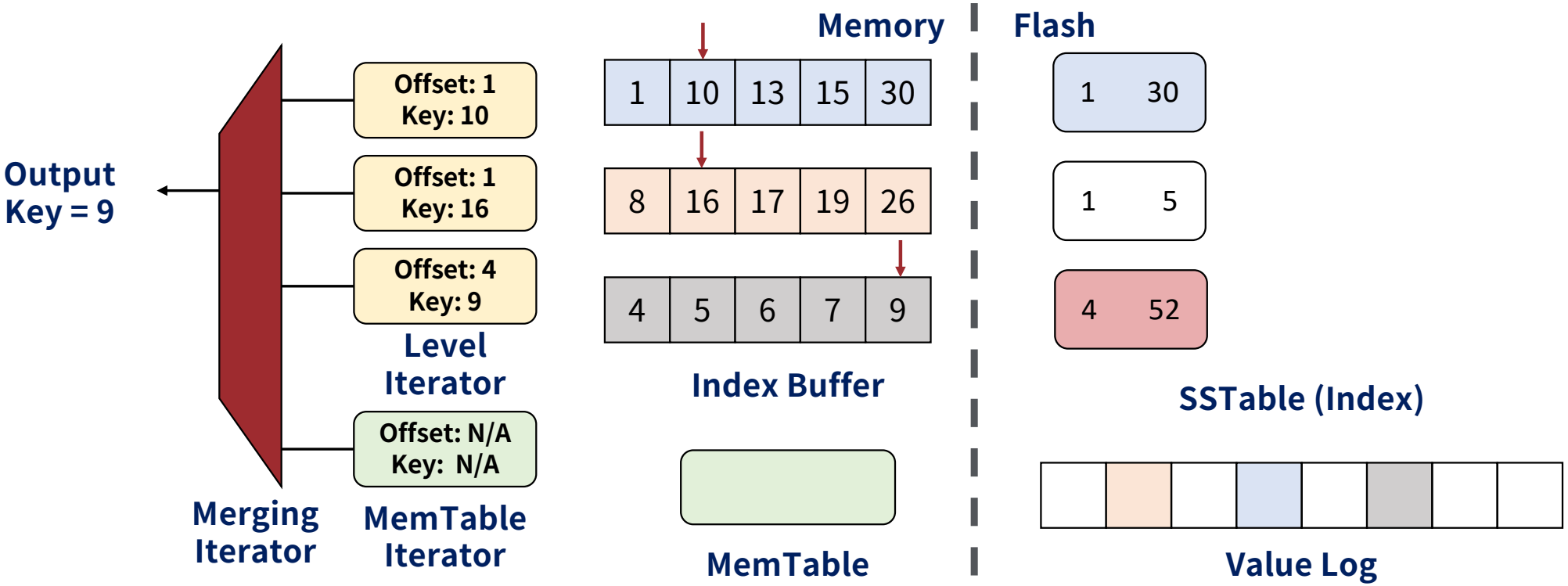
Problem #1 - Versioning

- Versioning Problem
  - Put(), Delete() can be issued in the middle of range query



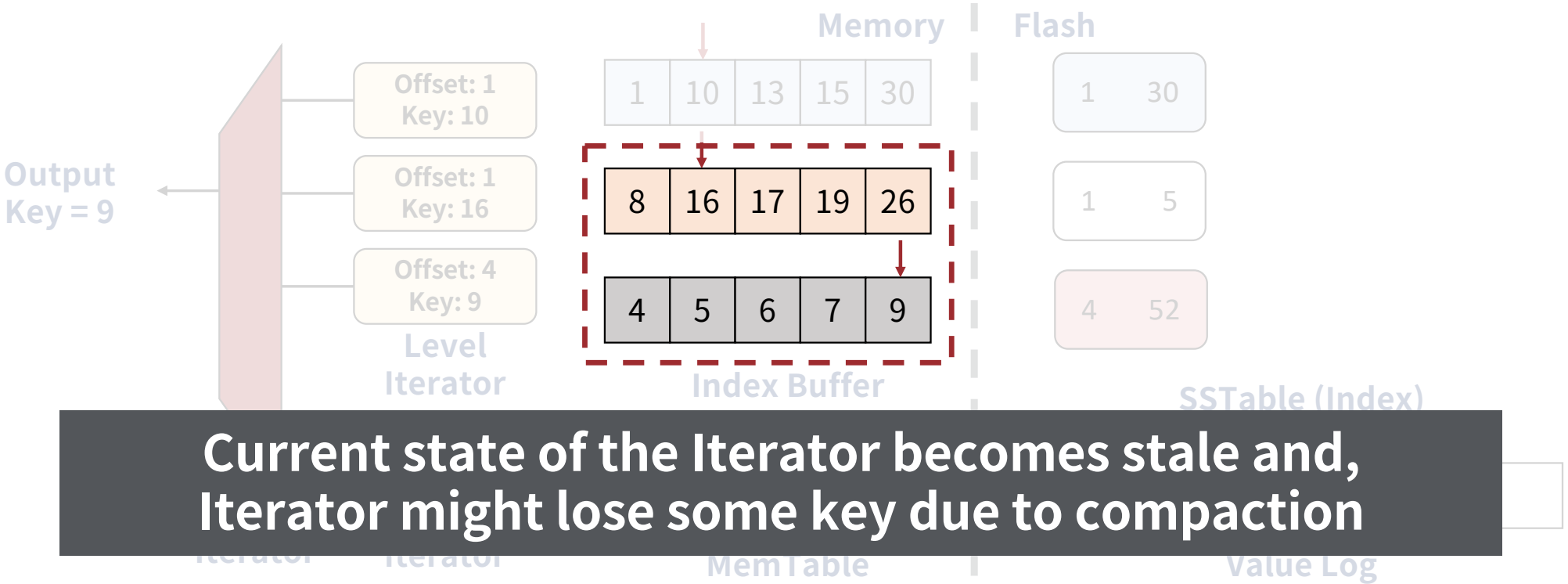
Problem #1 - Versioning

- **Versioning Problem**
  - Put(), Delete() can be issued in the middle of range query



Problem #1 - Versioning

- Versioning Problem
  - Put(), Delete() can be issued in the middle of range query



Current state of the Iterator becomes stale and, Iterator might lose some key due to compaction

Problem #1 - Versioning

- **Versioning Problem**
  - Put(), Delete() can be issued in the middle of range query
  - For this reason, host-side Key-Value stores support versioning in general
  - An iterator needs to see the version of the LSM-tree at its creation time.

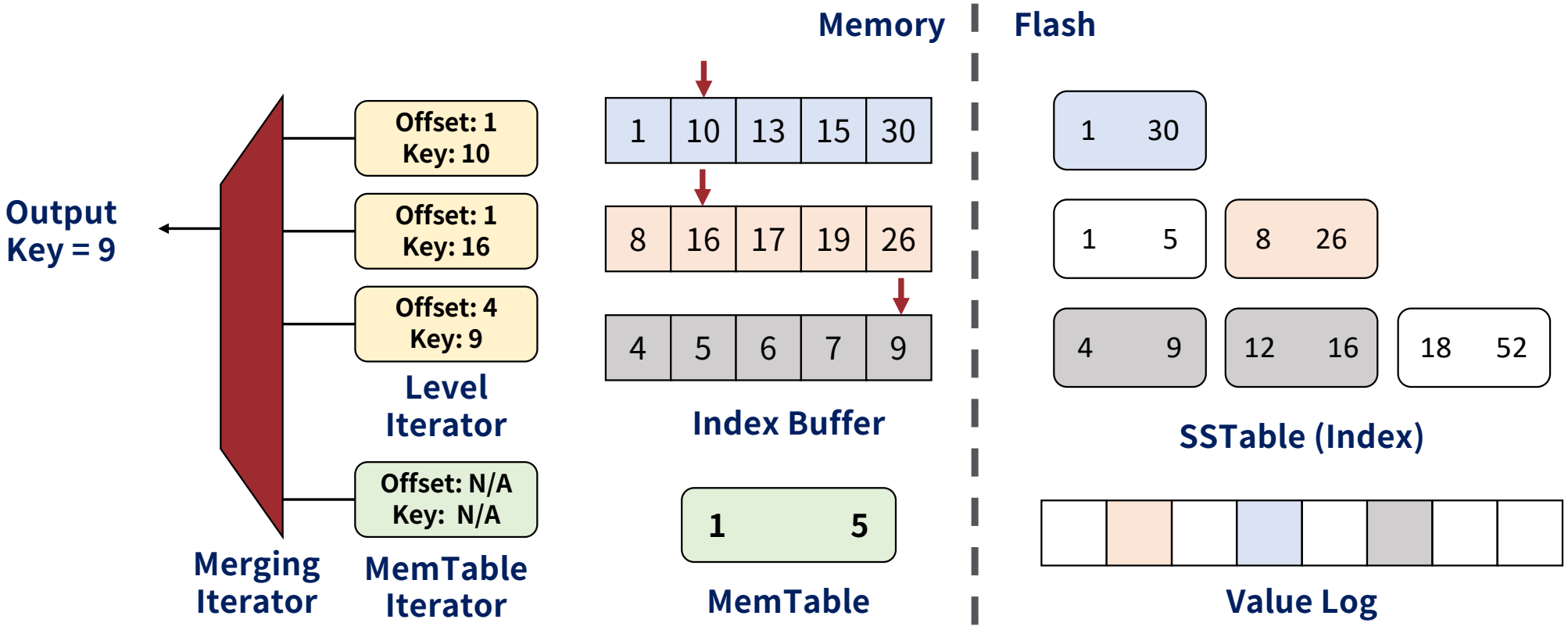
Problem #1 - Versioning

- **Versioning Problem**
  - Put(), Delete() can be issued in the middle of range query
  - For this reason, host-side Key-Value stores support versioning in general
  - An iterator needs to see the version of the LSM-tree at its creation time.

But,we need memory-efficient versioning inside the device!

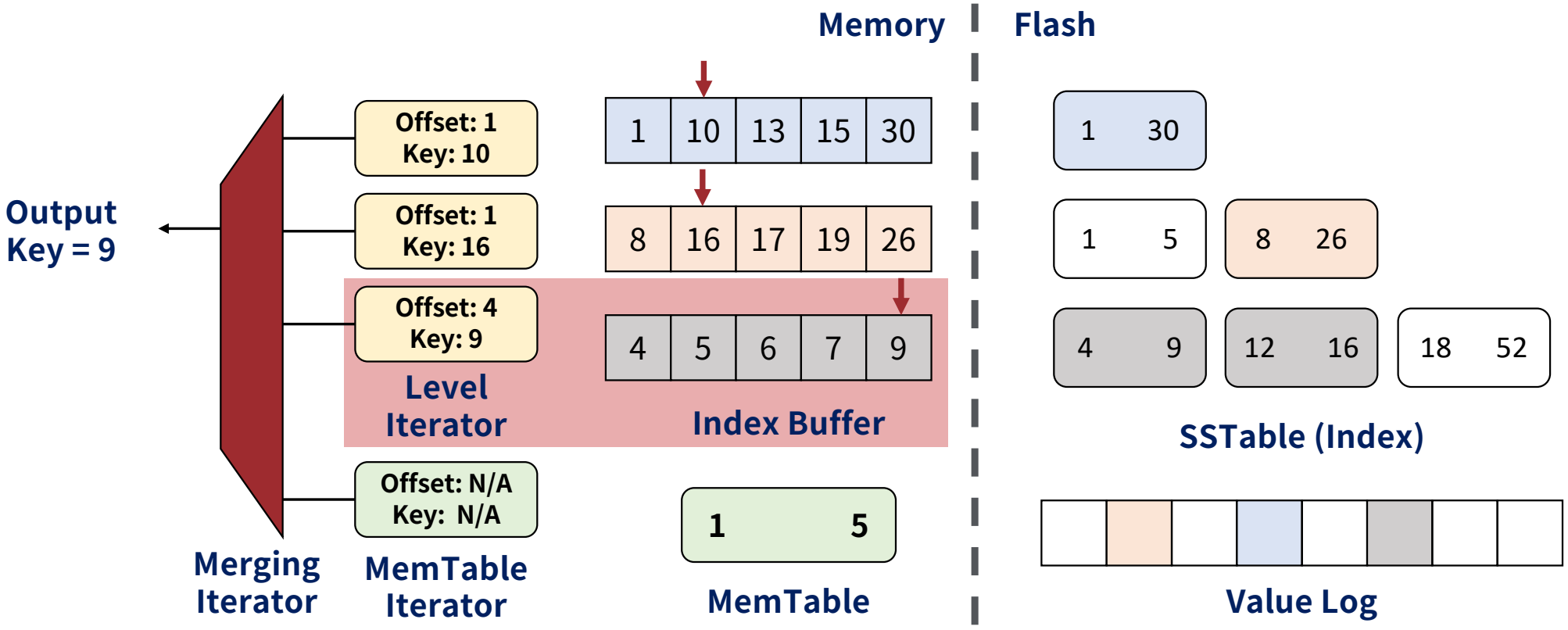
Problem #2 – Synchronous Index Read

- **Synchronous NAND Flash Access for Index Read**



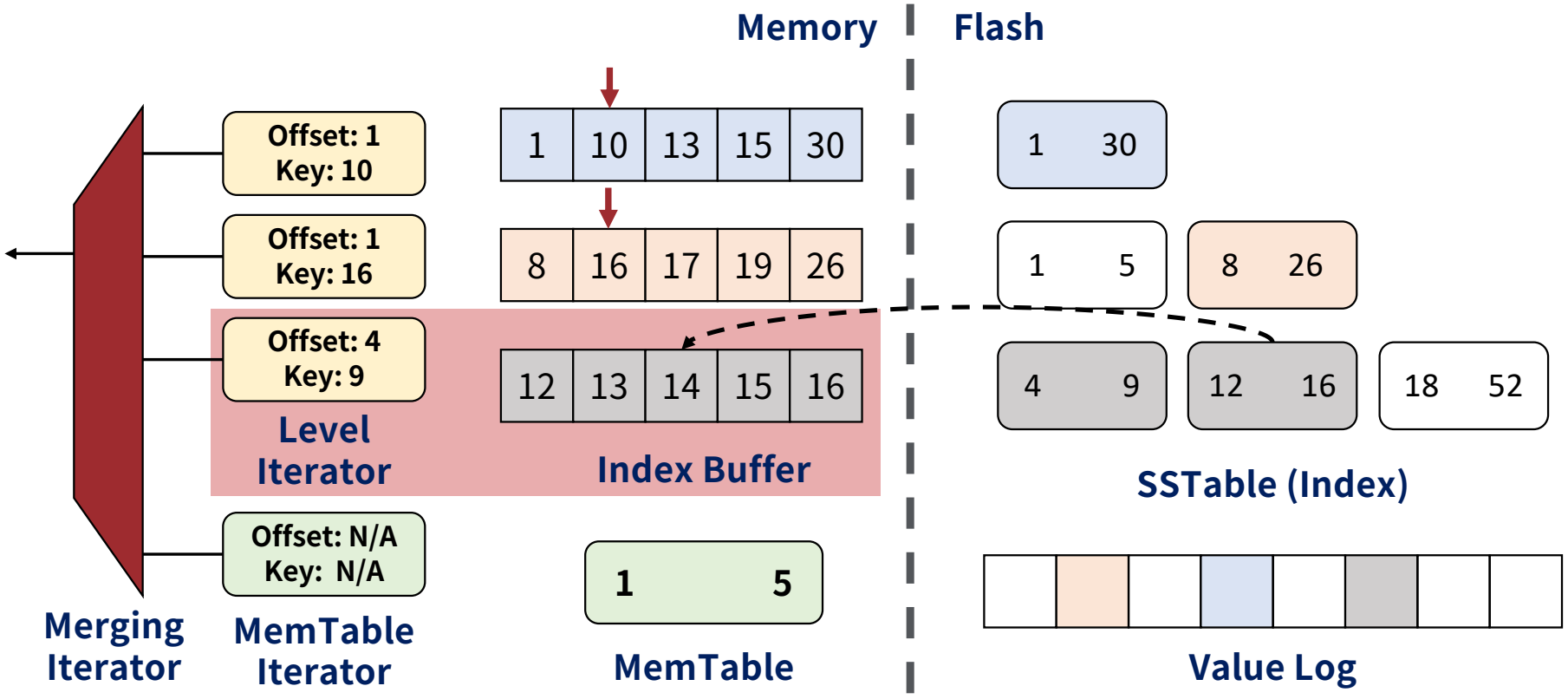
Problem #2 – Synchronous Index Read

- **Synchronous NAND Flash Access for Index Read**



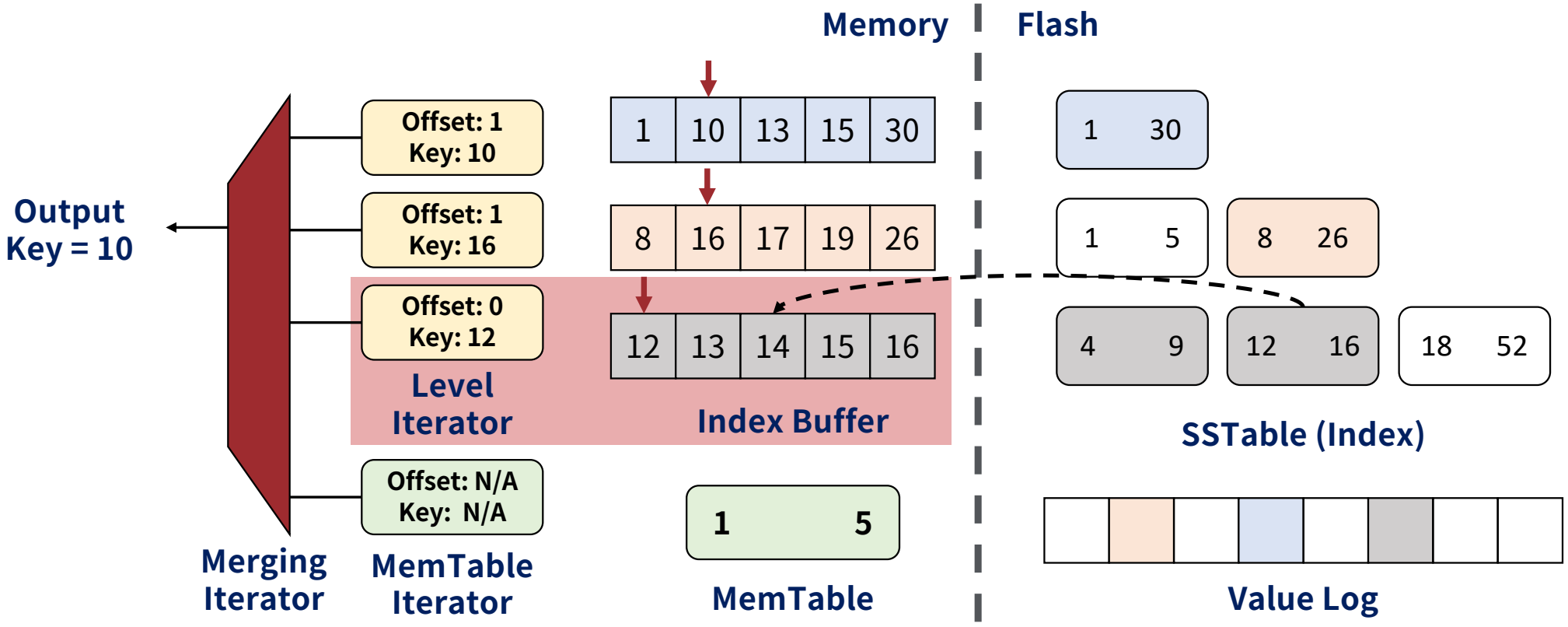
Problem #2 – Synchronous Index Read

- **Synchronous NAND Flash Access for Index Read**



Problem #2 – Synchronous Index Read

- **Synchronous NAND Flash Access for Index Read**



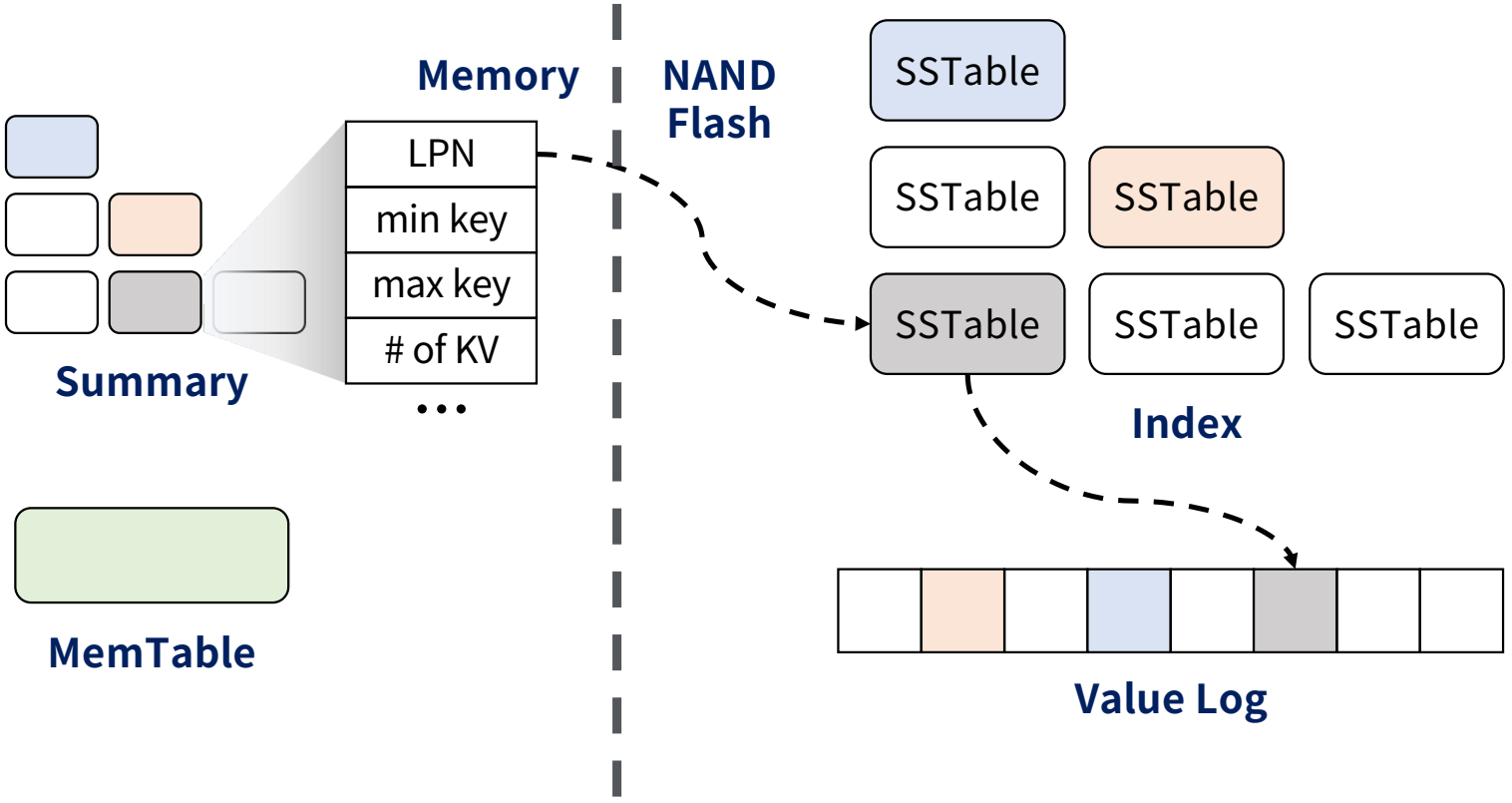
Problem #3 – Synchronous Value Read

- **Design Challenge #3 – NAND Flash Access for Value Read**
  - **Every Seek() and Next() command** requires NAND Flash Access for Value
  - Considering that NAND Flash access is much slower than the other steps, synchronous NAND Flash access for Value may woefully aggravate the overall performance

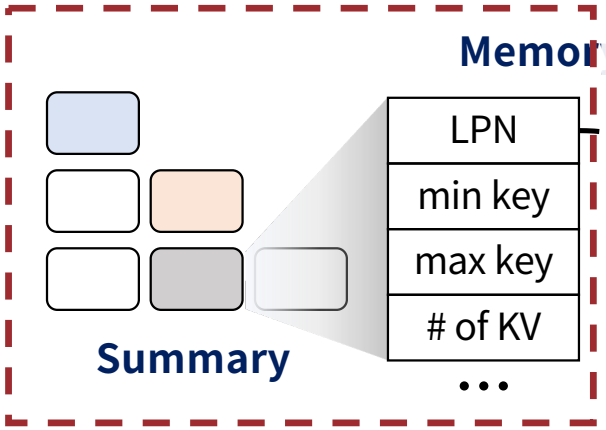
Design of IterKVSSD

Memory Efficient Versioning Data Structure

- How to support Versioning inside the device?

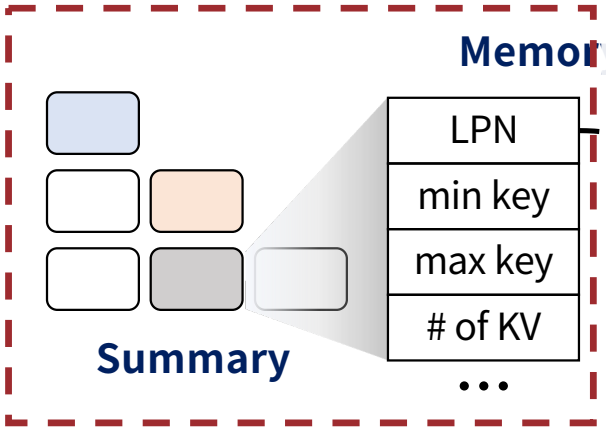


Memory Efficient Versioning Data Structure



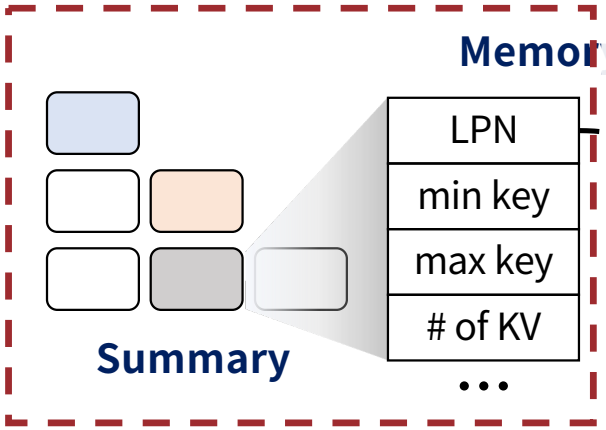
- Summary represents the state of LSM-tree
- Summary Size $\propto$ # of SSTables
- In our setup,
  1. Summary Entry Size = 44B
  2. # of SSTables $\approx$ 65,000
  3. Total Size $\approx$ 2.7MB

Memory Efficient Versioning Data Structure



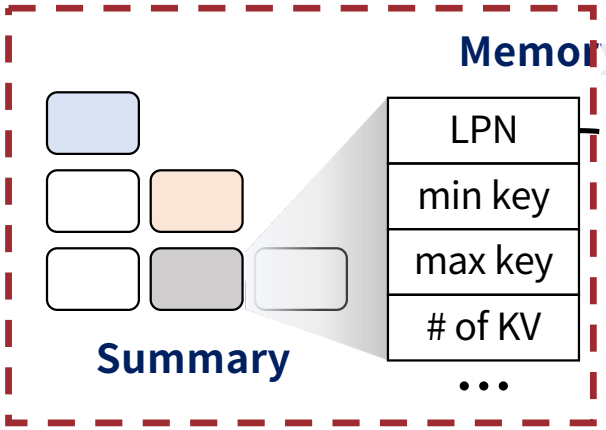
- Summary represents the state of LSM-tree
- Summary Size $\propto$ # of SSTables
- In our setup,
  1. Summary Entry Size = 44B
  2. # of SSTables $\approx$ 65,000
  3. Total Size $\approx$ 2.7MB

Memory Efficient Versioning Data Structure



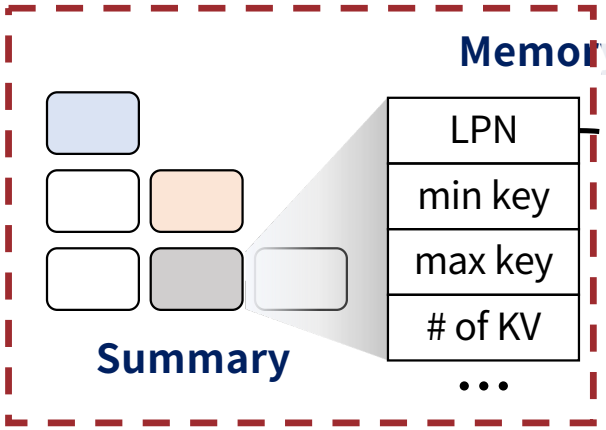
- Summary represents the state of LSM-tree
- Summary Size $\propto$ # of SSTables
- In our setup,
  1. Summary Entry Size = 44B
  2. # of SSTables $\approx$ 65,000
  3. Total Size $\approx$ 2.7MB

Memory Efficient Versioning Data Structure



- Summary represents the state of LSM-tree
- Summary Size $\propto$ # of SSTables
- In our setup,
  1. Summary Entry Size = 44B
  2. # of SSTables $\approx$ 65,000
  3. Total Size $\approx$ 2.7MB

Memory Efficient Versioning Data Structure

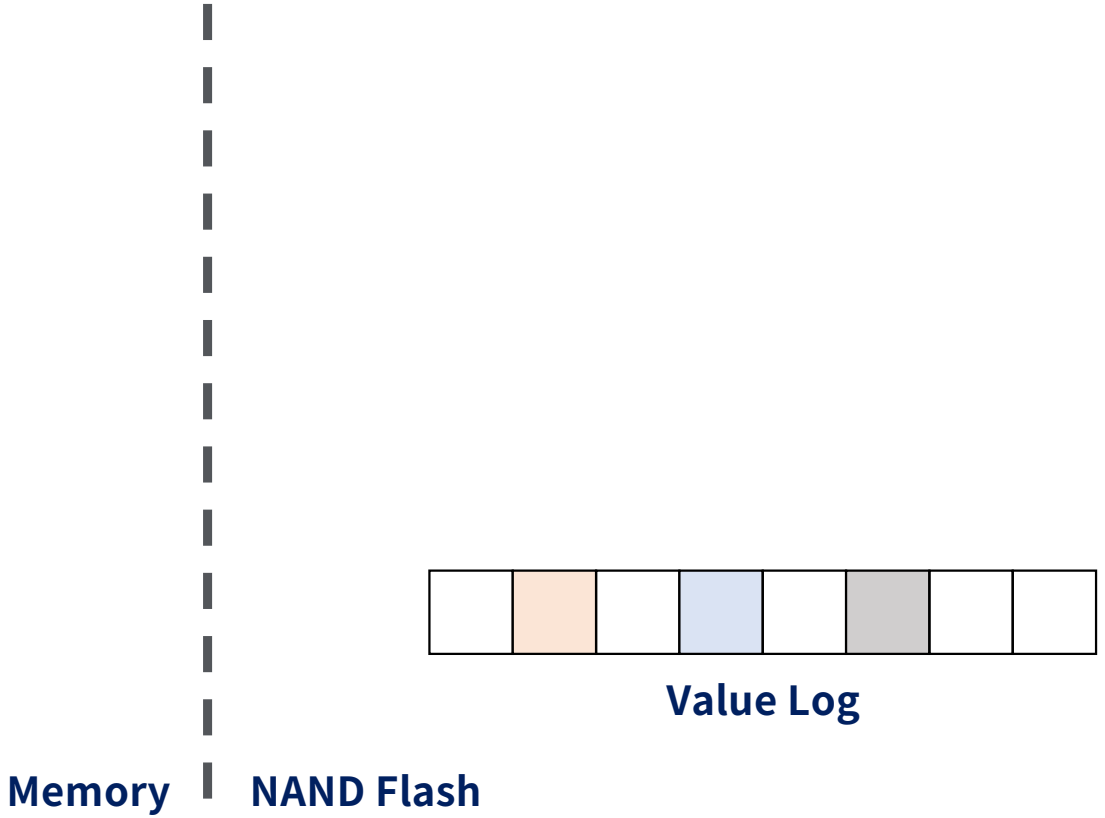


- Summary represents the state of LSM-tree
- Summary Size $\propto$ # of SSTables
- In our setup,
  1. Summary Entry Size = 44B
  2. # of SSTables $\approx$ 65,000
  3. Total Size $\approx$ 2.7MB

Keeping Summary for every Iterator is too expensive!

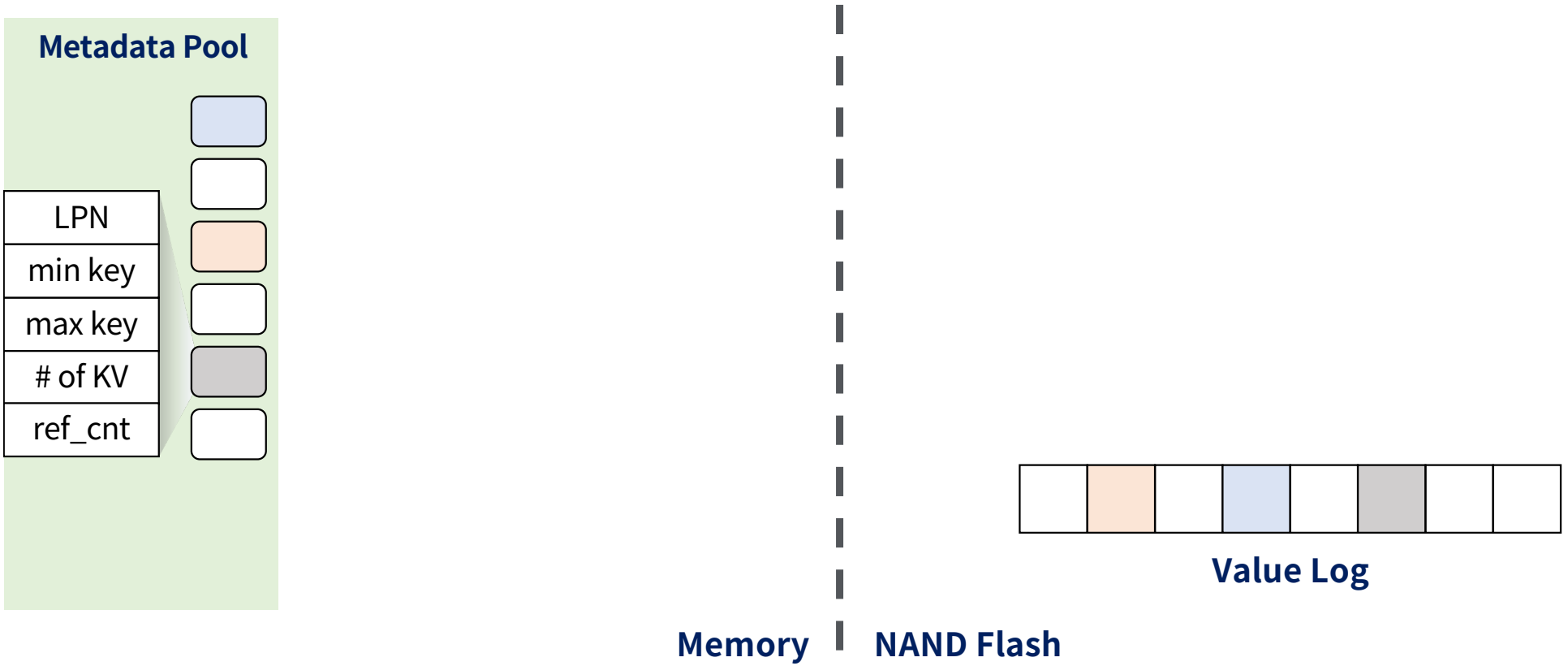
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



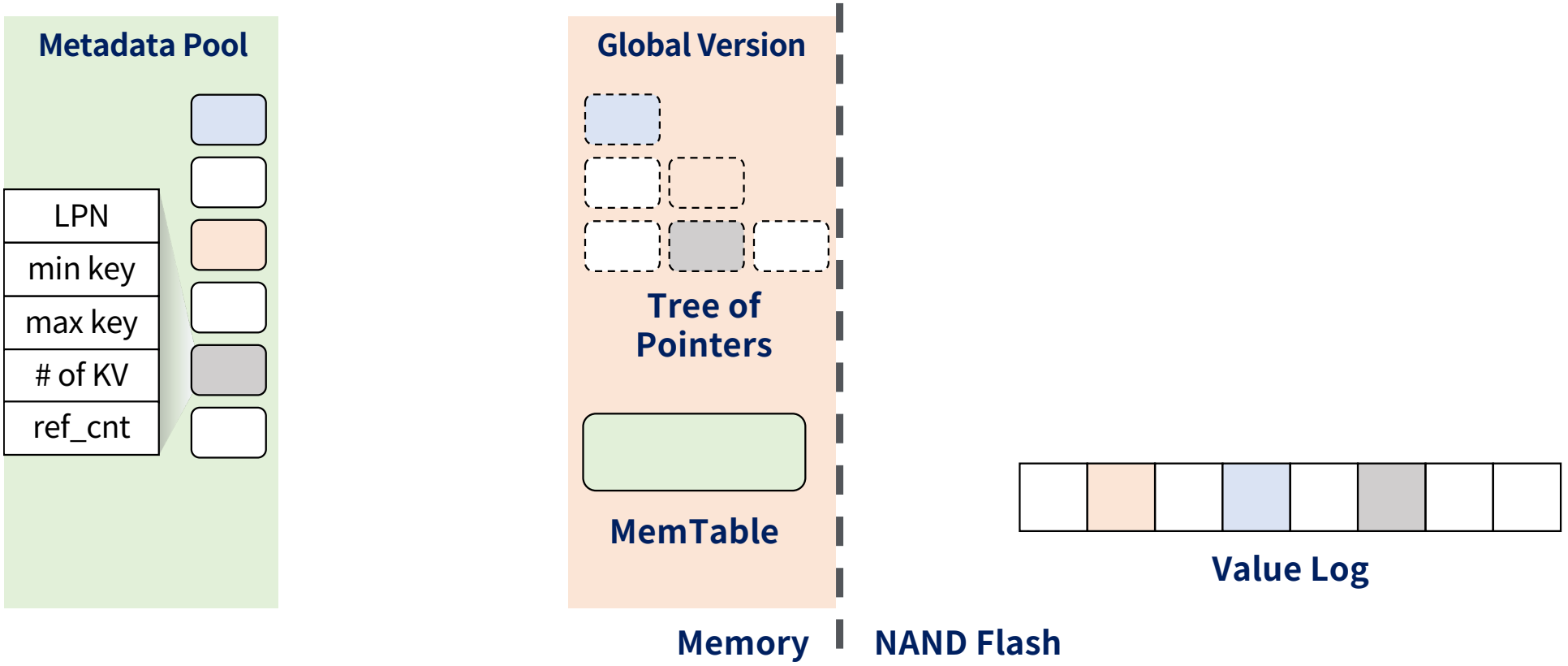
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



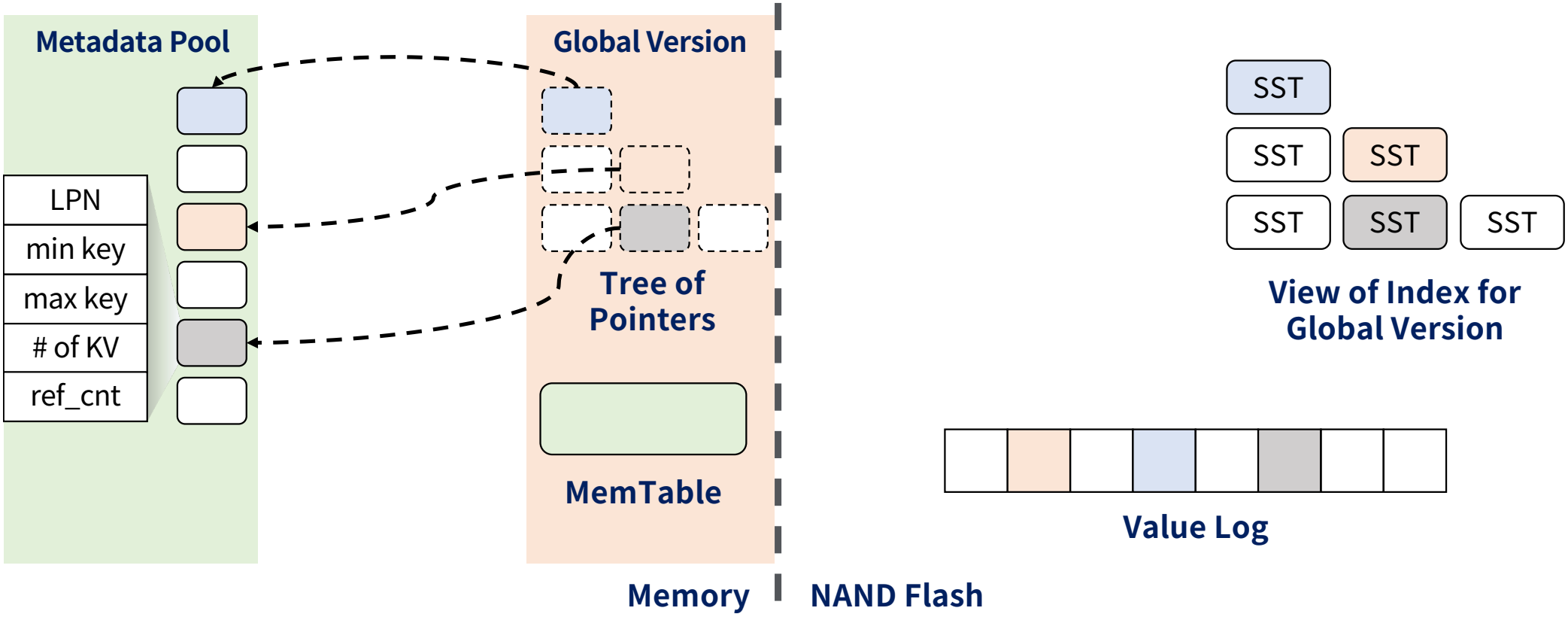
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



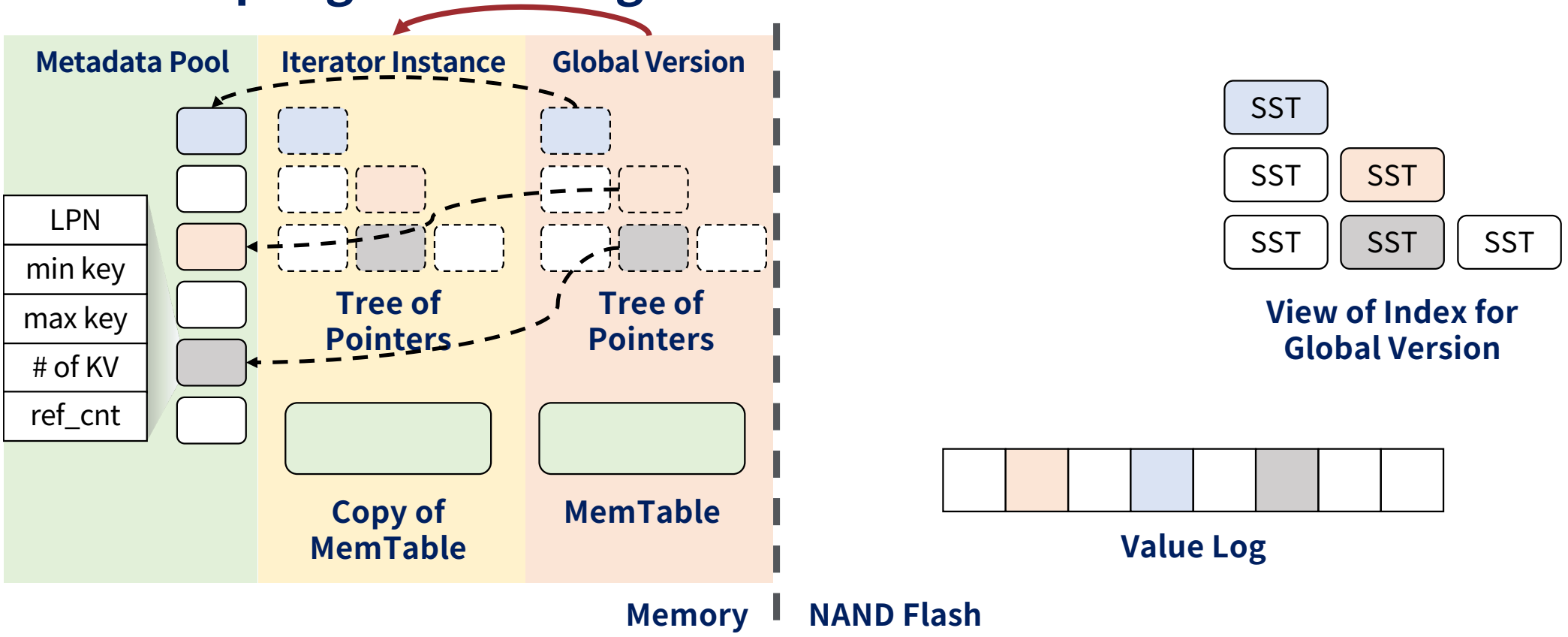
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



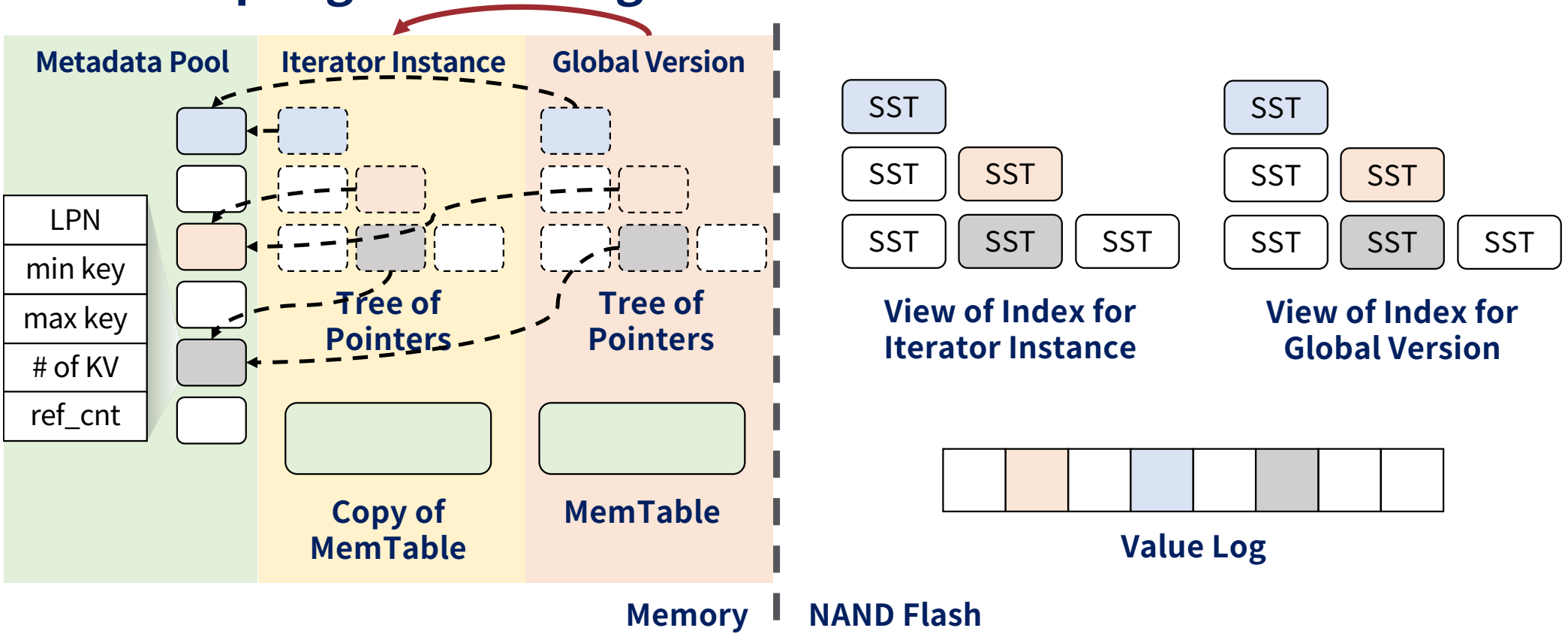
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



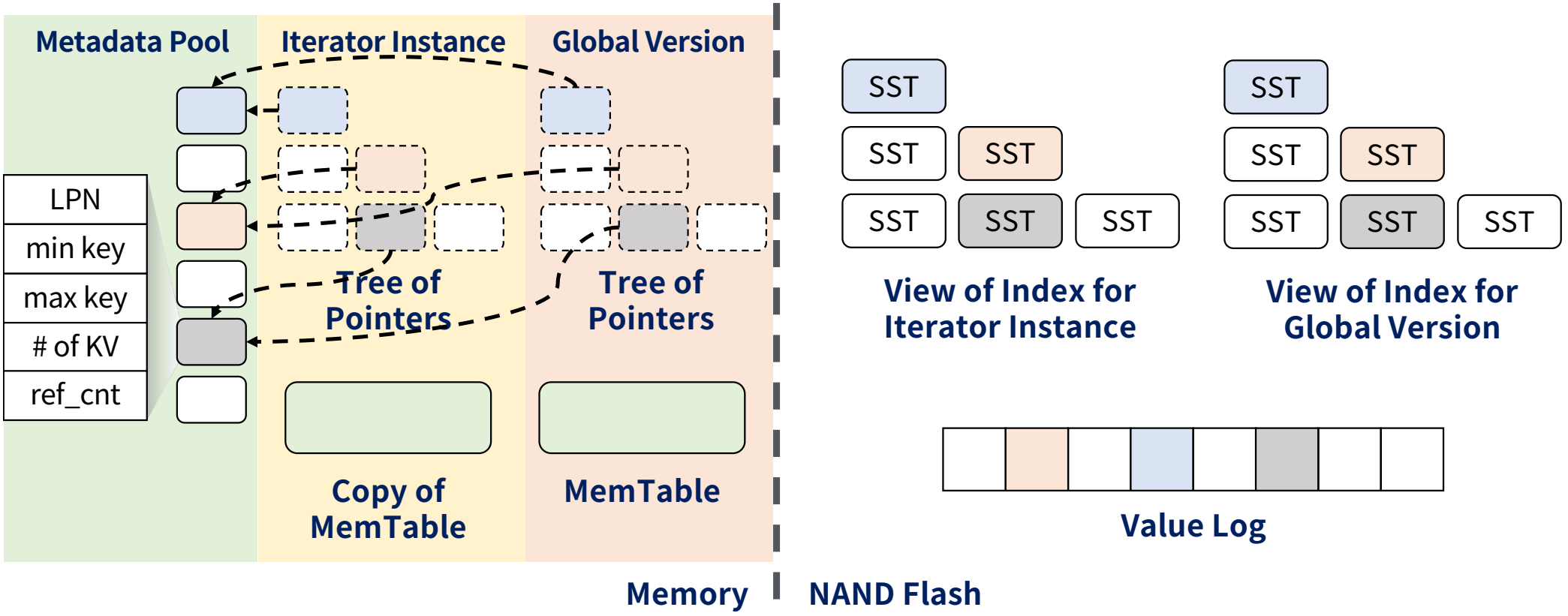
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



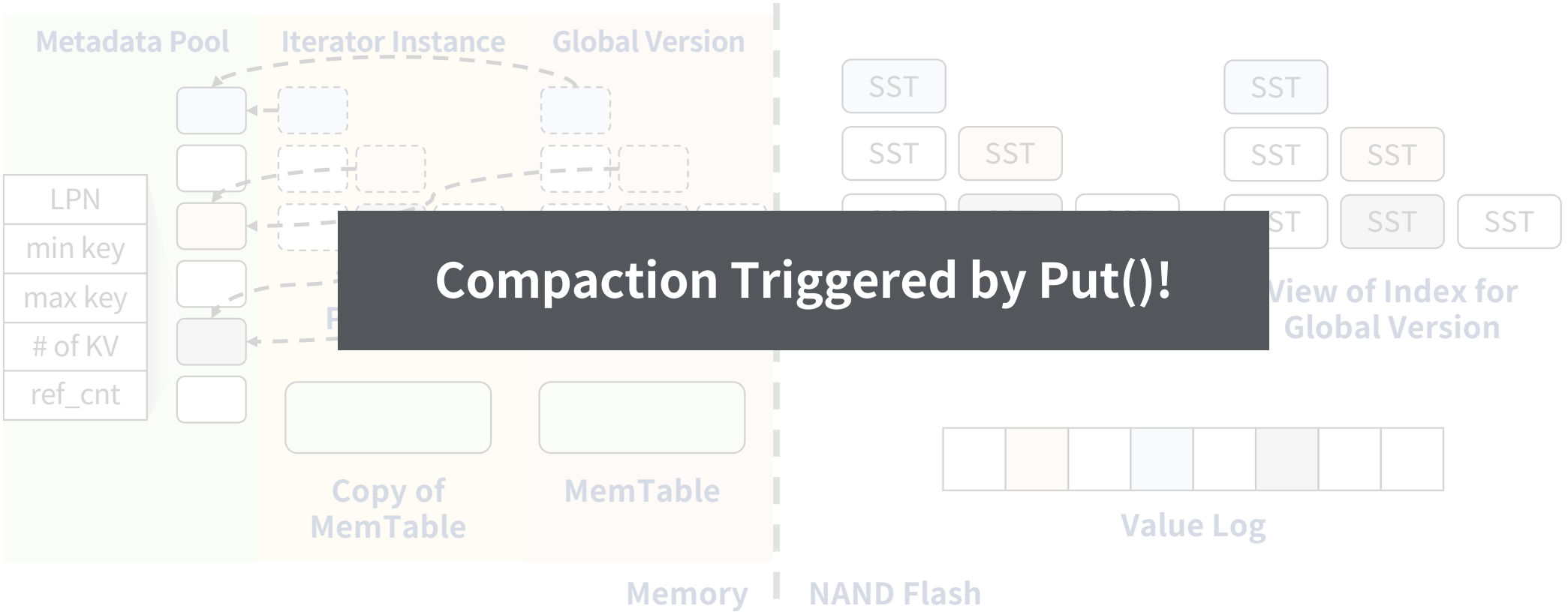
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



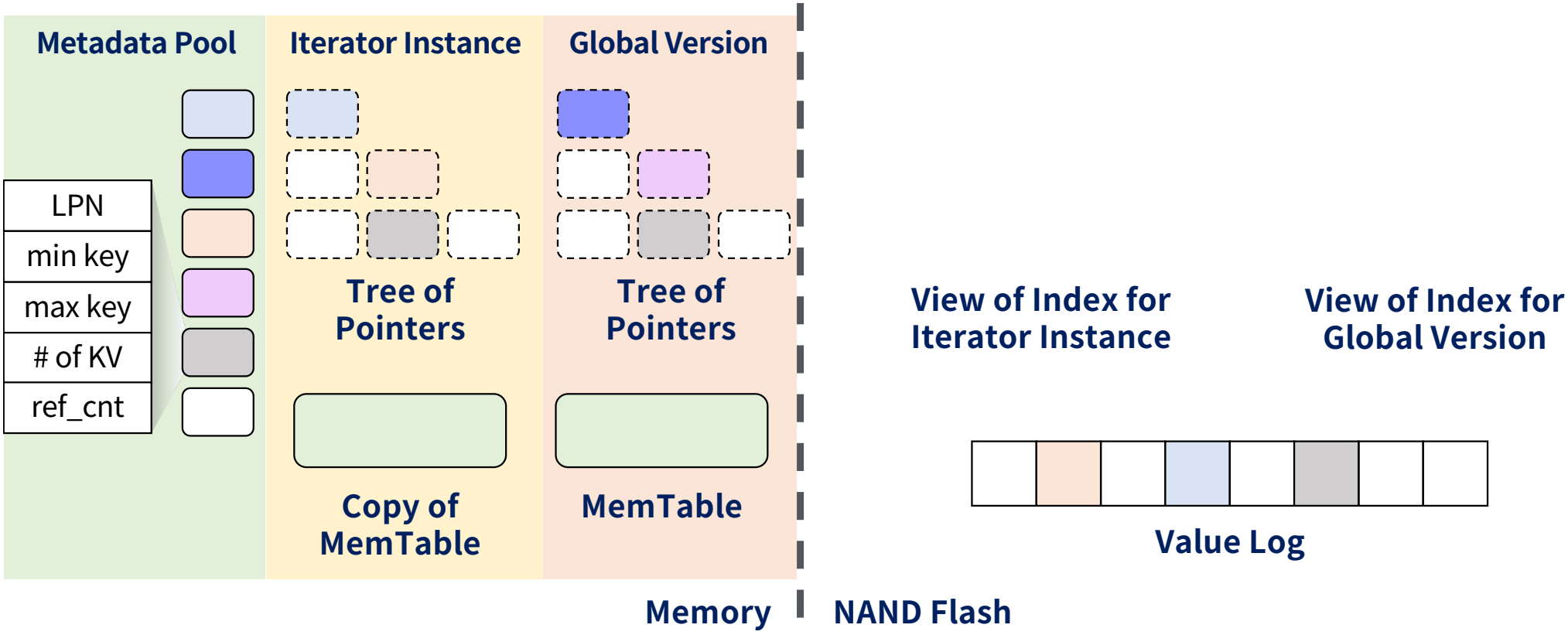
Memory Efficient Versioning Data Structure

- Decoupling and Pooling Metadata from Version Data Structure



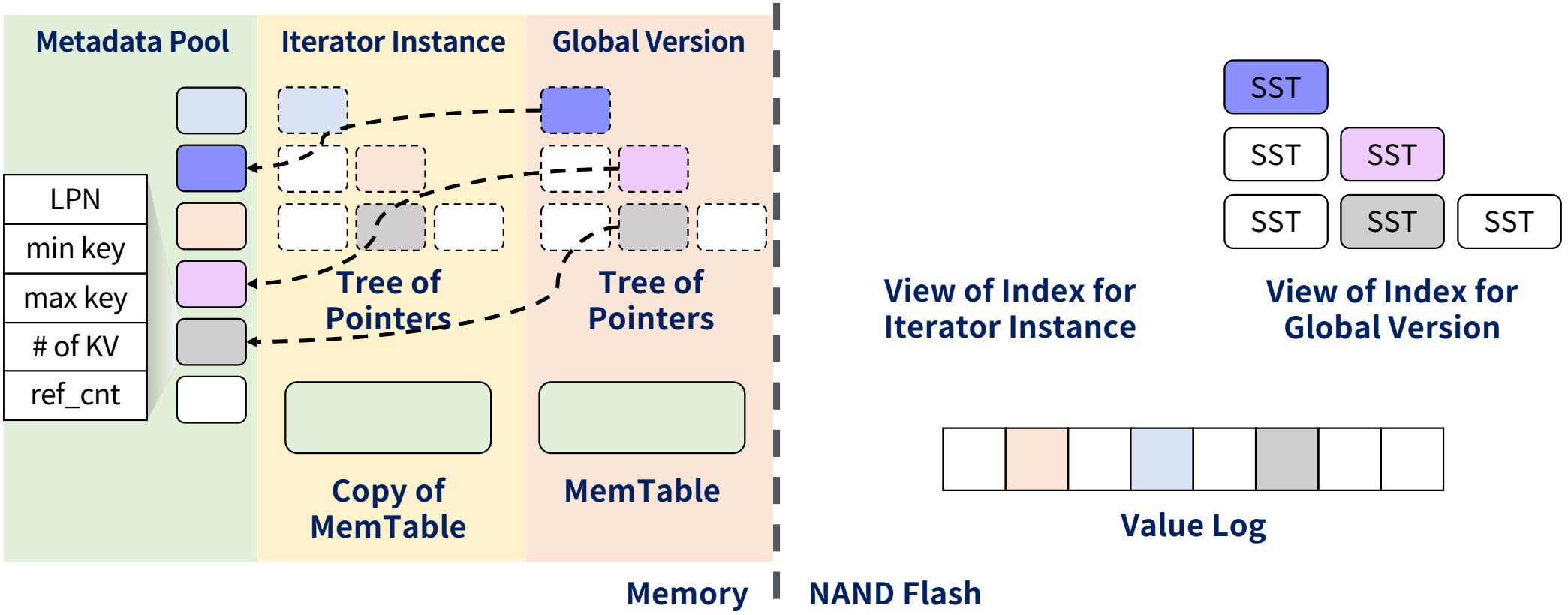
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



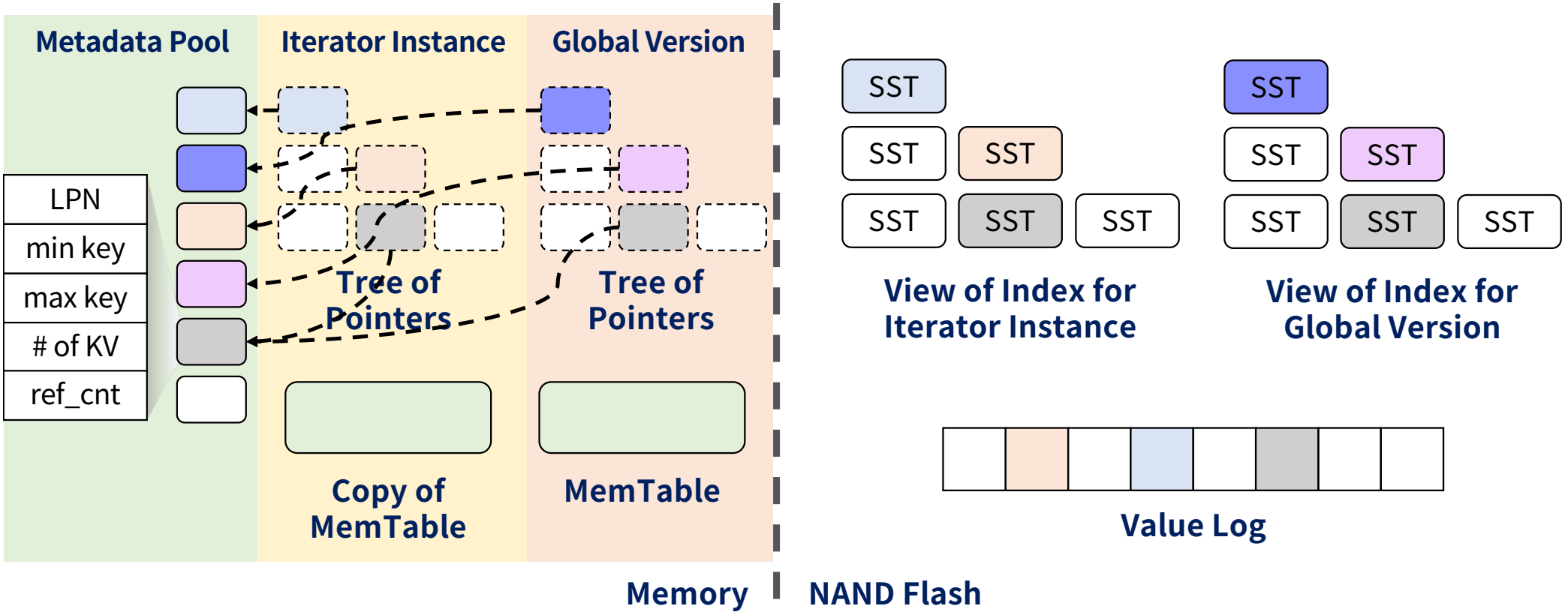
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



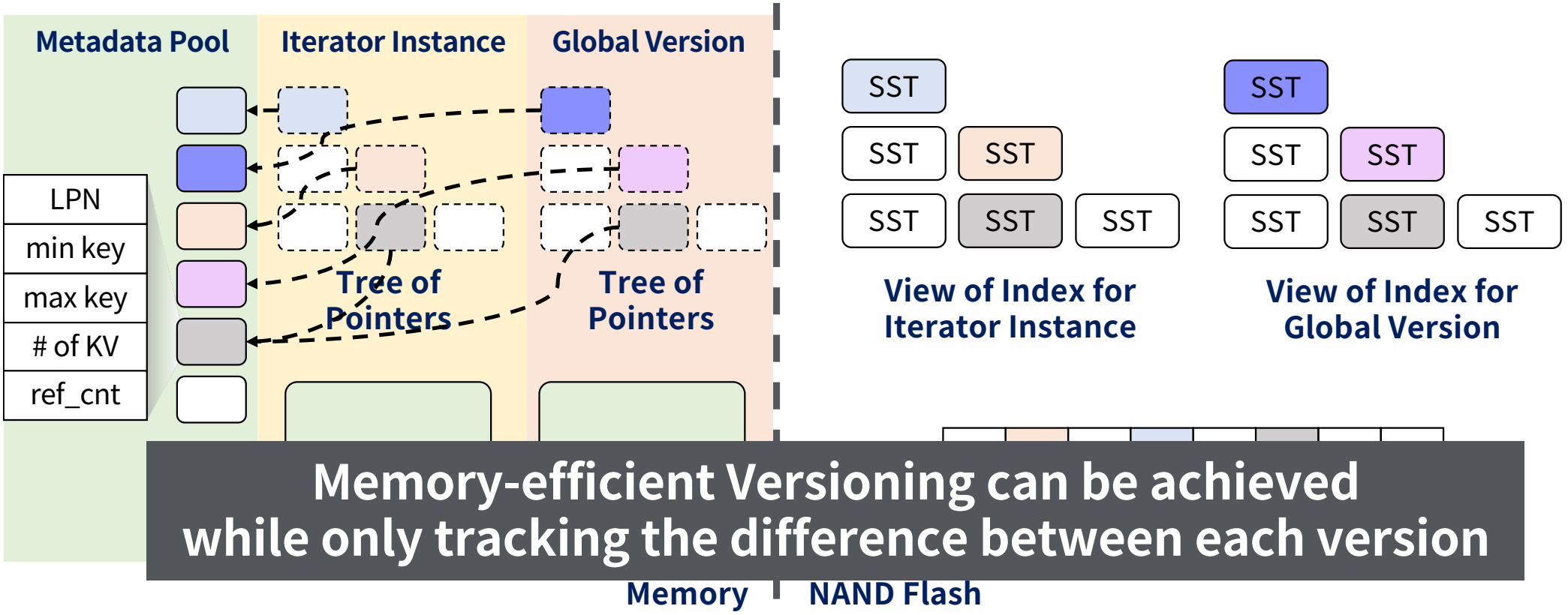
Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



Memory Efficient Versioning Data Structure

- **Decoupling and Pooling Metadata from Version Data Structure**



Memory-efficient Versioning can be achieved while only tracking the difference between each version

How to Reduce NAND Flash Access Cost?

How to Reduce NAND Flash Access Cost?

- **Key ideas to minimize synchronous NAND Flash access penalty**

1. **Range query sequentially retrieves Key-Value pairs in a row**
   - KVSSD knows what is the next key to read
2. **Key-Value semantic is enabled inside the device**
   - KVSSD knows where in physical memory the next Key-Value pairs are stored
3. **Exploit multiple independent channel controllers**
   - KVSSD can overlap NAND Flash access with processing storage protocol and the other steps in parallel

How to Reduce NAND Flash Access Cost?

- **Key ideas to minimize synchronous NAND Flash access penalty**

1. **Range query sequentially retrieves Key-Value pairs in a row**
   - KVSSD knows what is the next key to read
2. **Key-Value semantic is enabled inside the device**
   - KVSSD knows where in physical memory the next Key-Value pairs are stored
3. **Exploit multiple independent channel controllers**
   - KVSSD can overlap NAND Flash access with processing storage protocol and the other steps in parallel

How to Reduce NAND Flash Access Cost?

- **Key ideas to minimize synchronous NAND Flash access penalty**

1. **Range query sequentially retrieves Key-Value pairs in a row**
   - KVSSD knows what is the next key to read
2. **Key-Value semantic is enabled inside the device**
   - KVSSD knows where in physical memory the next Key-Value pairs are stored
3. **Exploit multiple independent channel controllers**
   - KVSSD can overlap NAND Flash access with processing storage protocol and the other steps in parallel

How to Reduce NAND Flash Access Cost?

How to Reduce NAND Flash Access Cost?

- **On every Seek and Next commands,**

1. **Compute** for processing the storage protocol command
2. **Read Index** from NAND Flash, **if necessary**
3. **Search and Compare** the Index for each level
4. **Read Value** from NAND flash
5. **DMA transfer over PCIe** to return the result Key-Value pair

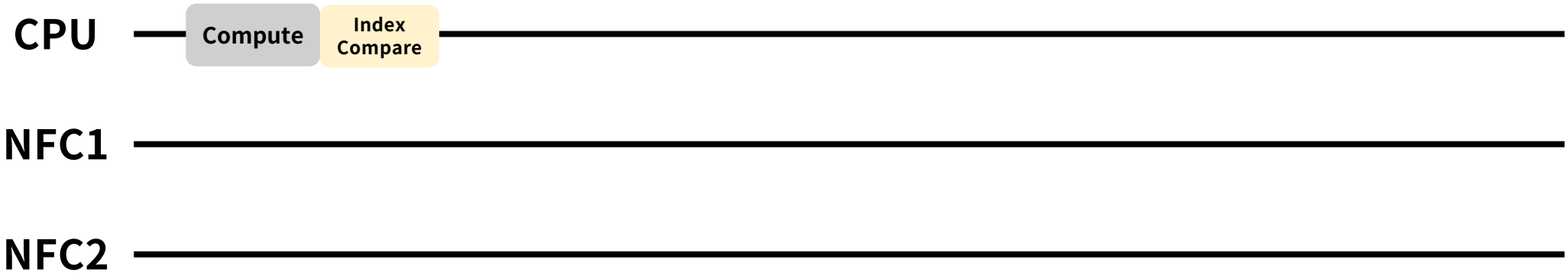
How to Reduce NAND Flash Access Cost?

- **On every Seek and Next commands,**

1. **Compute** for processing the storage protocol command
2. **Read Index** from NAND Flash, **if necessary**
3. **Search and Compare** the Index for each level
4. **Read Value** from NAND flash
5. **DMA transfer over PCIe** to return the result Key-Value pair

Index Prefetching

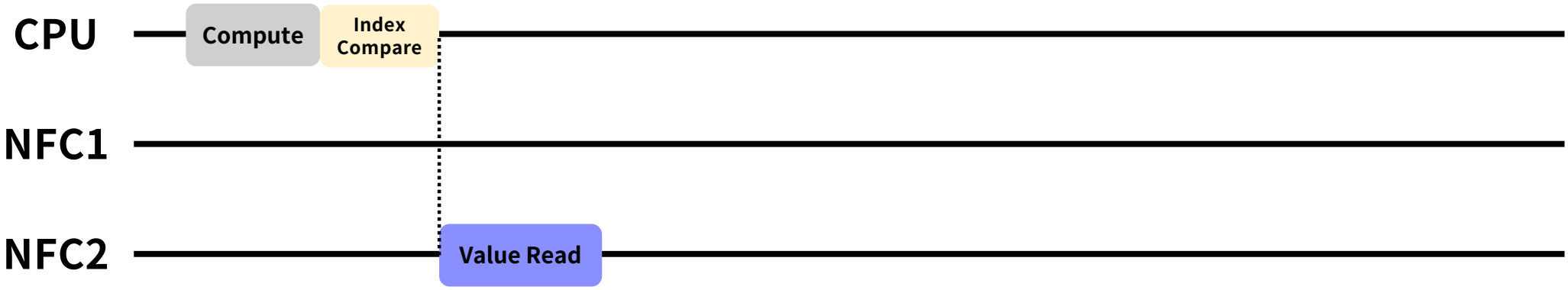
- **Index Prefetching**



- The device can accurately predict when to read an index from NAND.
- Index prefetch and other NAND accesses accessing the same channel can be avoided by using multiple NAND Flash Controllers (NFC).

Index Prefetching

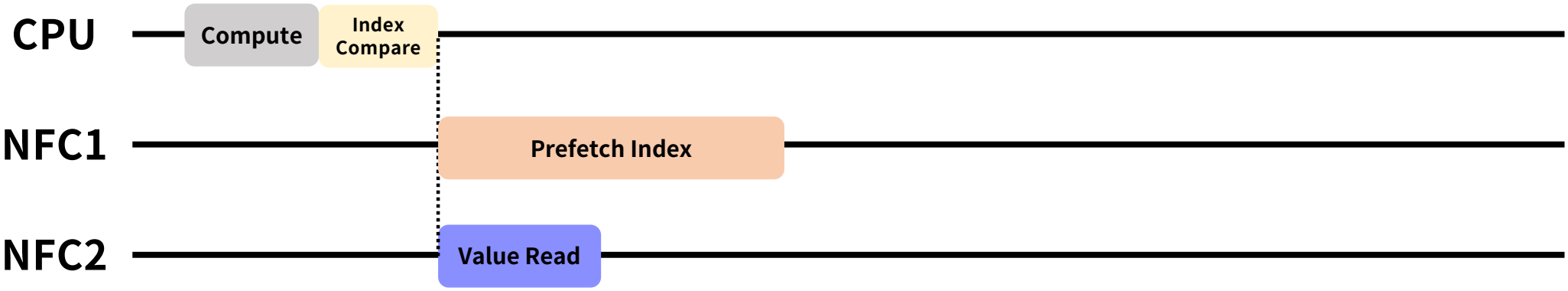
- **Index Prefetching**



- The device can accurately predict when to read an index from NAND.
- Index prefetch and other NAND accesses accessing the same channel can be avoided by using multiple NAND Flash Controllers (NFC).

Index Prefetching

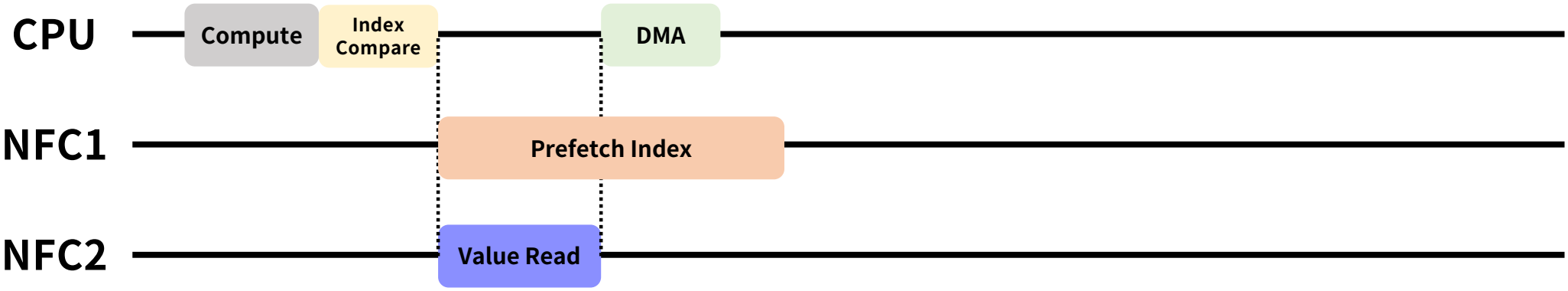
- **Index Prefetching**



- The device can accurately predict when to read an index from NAND.
- Index prefetch and other NAND accesses accessing the same channel can be avoided by using multiple NAND Flash Controllers (NFC).

Index Prefetching

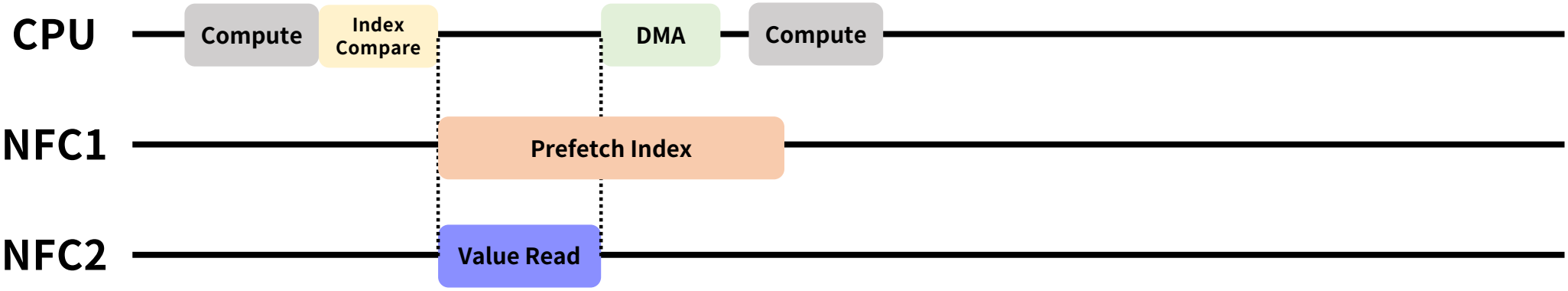
- **Index Prefetching**



- The device can accurately predict when to read an index from NAND.
- Index prefetch and other NAND accesses accessing the same channel can be avoided by using multiple NAND Flash Controllers (NFC).

Index Prefetching

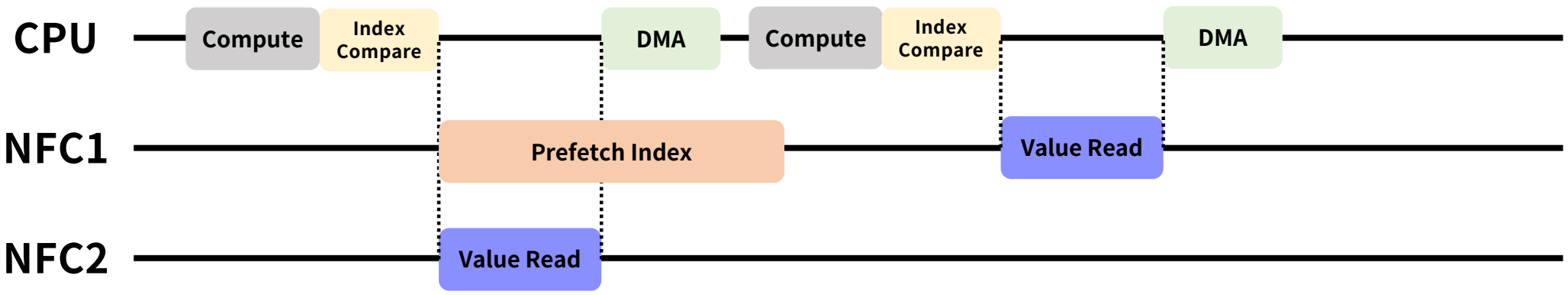
- **Index Prefetching**



- The device can accurately predict when to read an index from NAND.
- Index prefetch and other NAND accesses accessing the same channel can be avoided by using multiple NAND Flash Controllers (NFC).

Index Prefetching

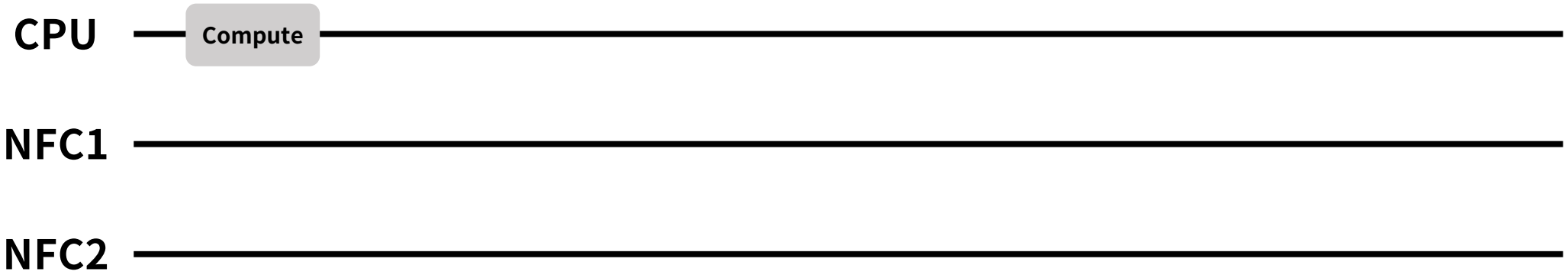
- **Index Prefetching**



- The device can accurately predict when to read an index from NAND.
- Index prefetch and other NAND accesses accessing the same channel can be avoided by using multiple NAND Flash Controllers (NFC).

Value Prefetching

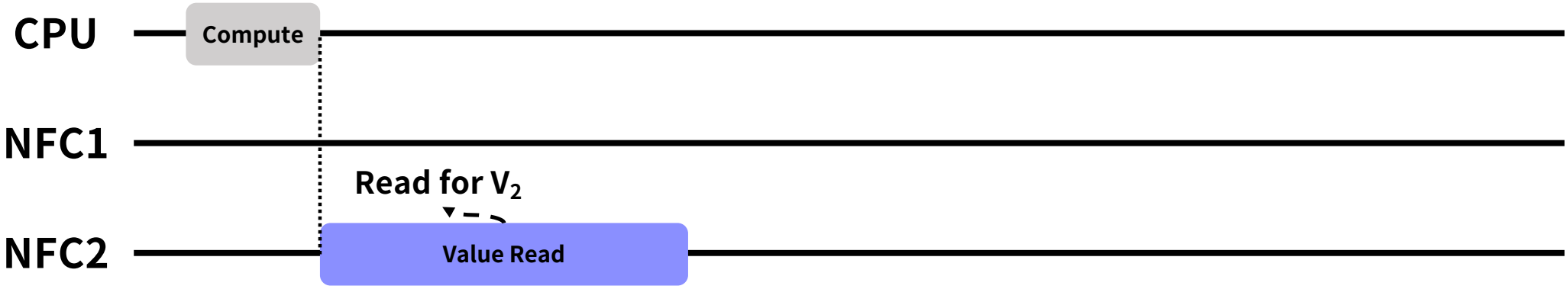
- **Value Prefetching**



- When processing **Nth** Next command, the device issues prefetch request for the **(N+D)th** Next command in the future (**D**: Prefetching Degree).
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

Value Prefetching

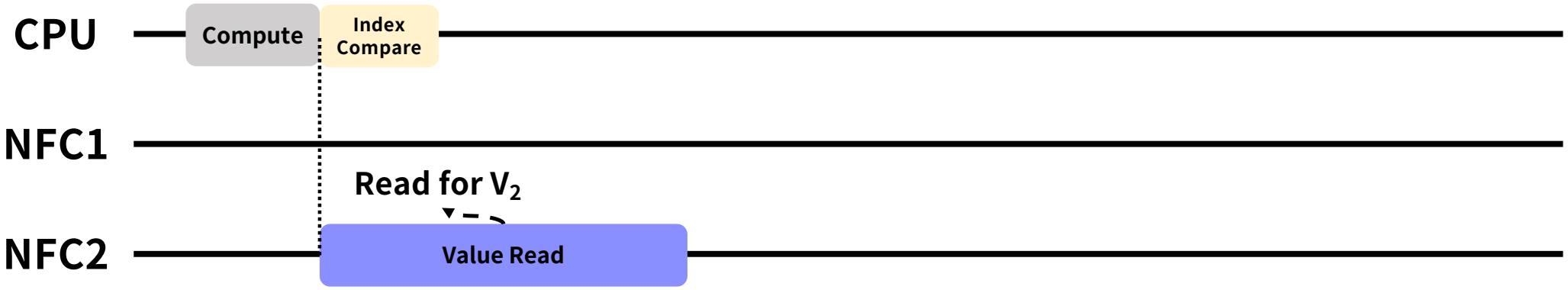
- **Value Prefetching**



- When processing **Nth** Next command, the device issues prefetch request for the **(N+D)th** Next command in the future (**D**: Prefetching Degree).
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

Value Prefetching

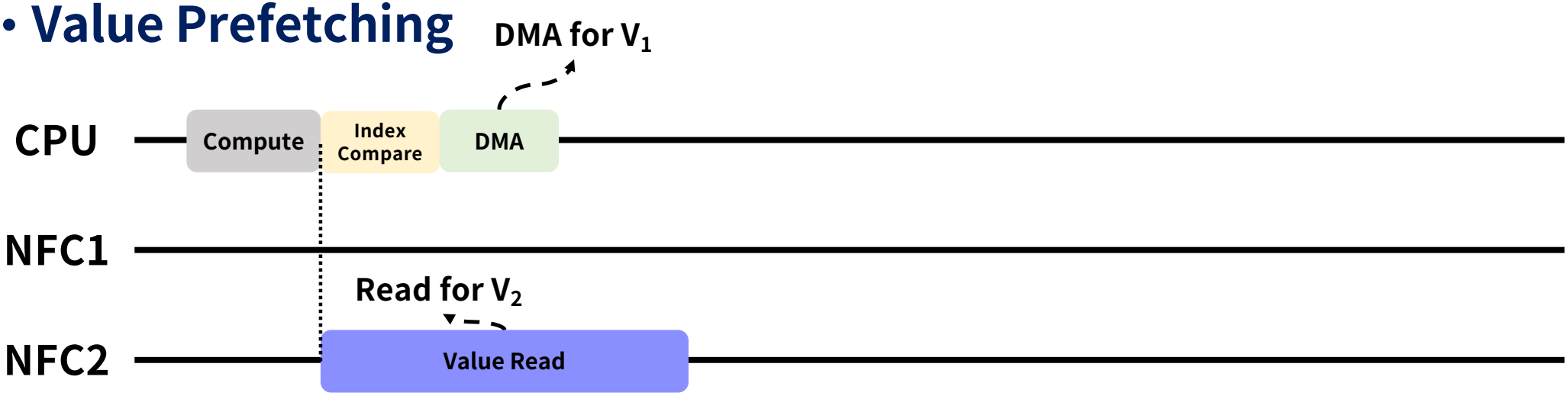
- **Value Prefetching**



- When processing **Nth** Next command, the device issues prefetch request for the **(N+D)th** Next command in the future (**D**: Prefetching Degree).
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

Value Prefetching

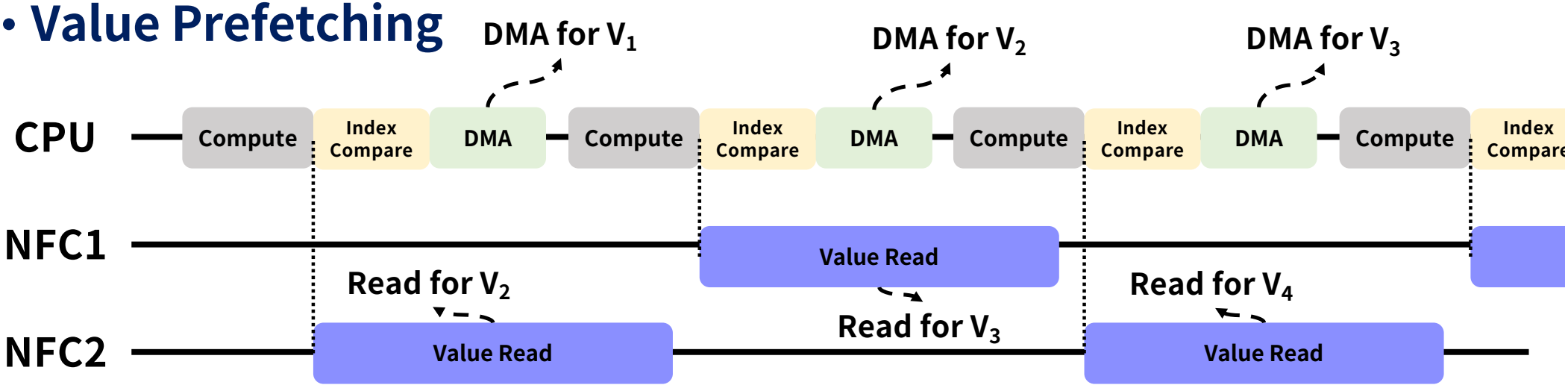
- **Value Prefetching**



- When processing **Nth** Next command, the device issues prefetch request for the **(N+D)th** Next command in the future (**D**: Prefetching Degree).
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

Value Prefetching

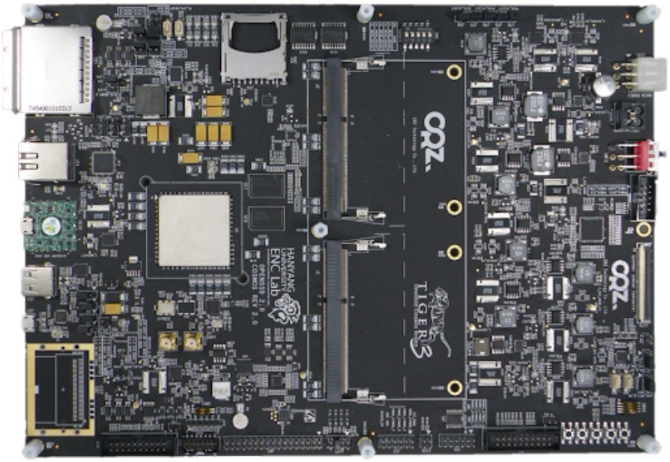
- **Value Prefetching**



- When processing **N**th Next command, the device issues prefetch request for the **(N+D)**th Next command in the future (**D**: Prefetching Degree).
- Therefore, it can overlap the reading time with other process in advance by exploiting hardware (NFCs).

Experimental Setup

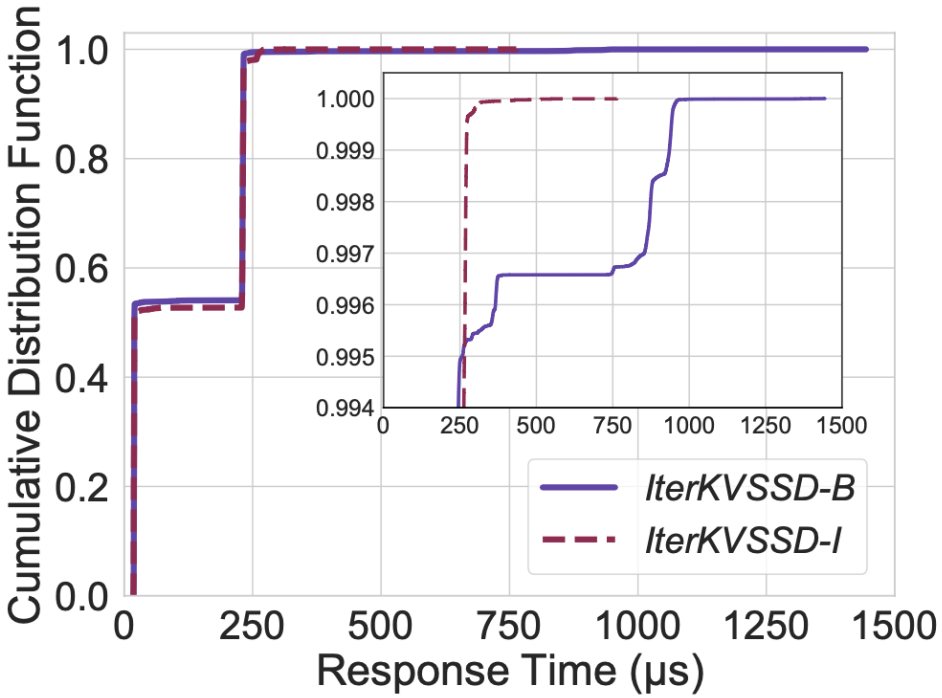
- **IterKVSSD prototyped on OpenSSD Cosmos+**
  - Extended NVMe Protocol for Iterator Command
- **Workloads**
  - RocksDB ***db_bench*** benchmark for ***SeekRandom*** workload
  - 3M Key-Value pairs with 4B Key
  - Varied scan length*, value size, and prefetch degree for value prefetch



scan length = the number of KV pairs retrieved during a range query

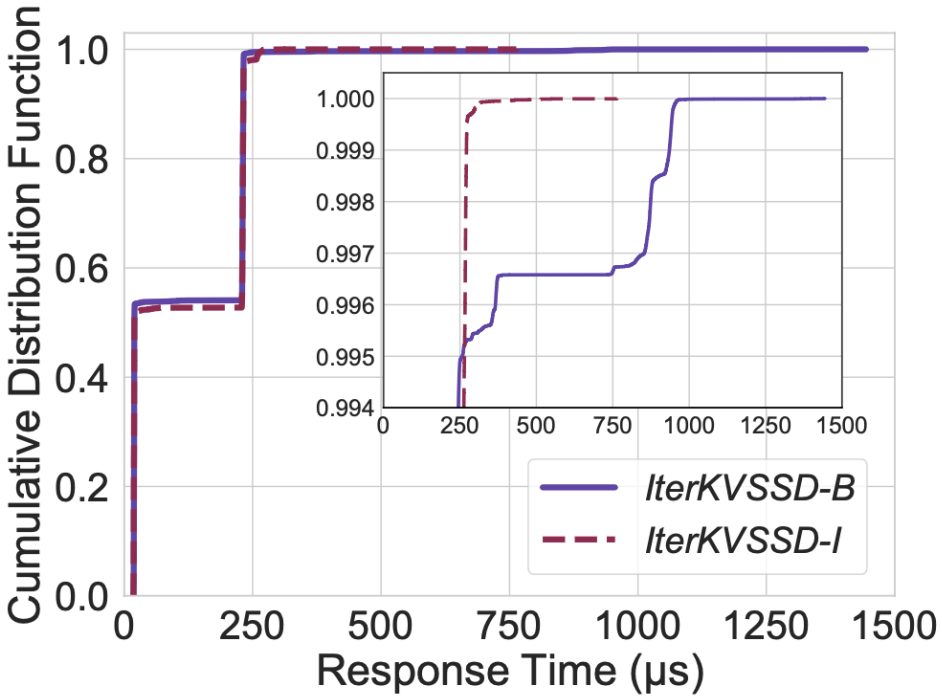
Evaluation – Index Prefetch

- **Effect of Index Prefetching**



Evaluation – Index Prefetch

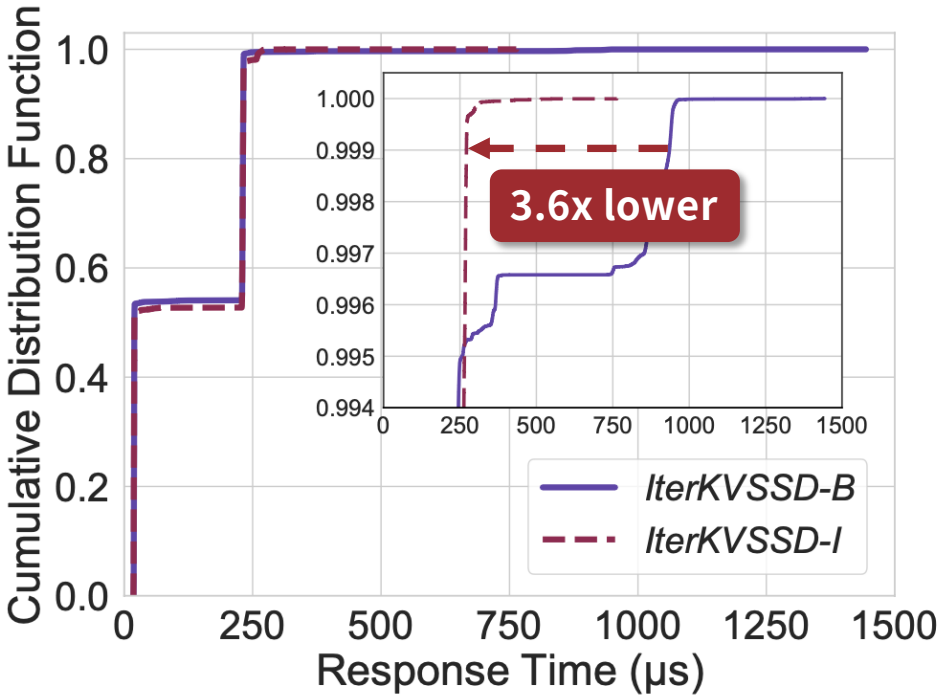
- **Effect of Index Prefetching**



- **IterKVSSD-B**: Baseline w/o prefetch
- **IterKVSSD-I**: w/ Index Prefetch + w/o Value Prefetch
- Scan Length = 200,000 which is enough to trigger Synchronous Index Read
- Show about 3.6x better P99.9 tail latency
- Channel conflict prevents it from being removed completely

Evaluation – Index Prefetch

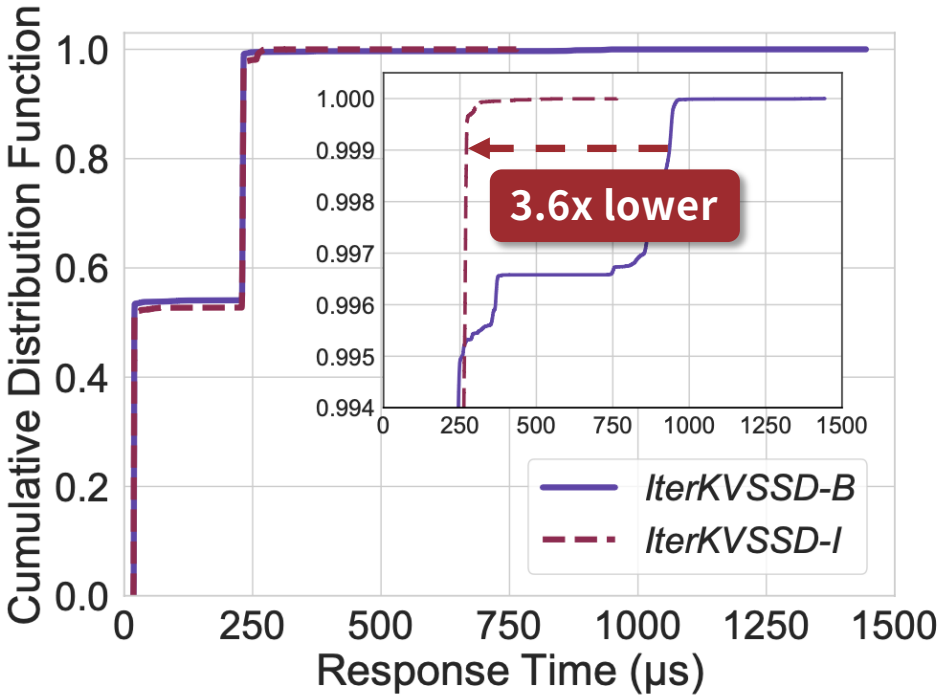
- **Effect of Index Prefetching**



- **IterKVSSD-B**: Baseline w/o prefetch
- **IterKVSSD-I**: w/ Index Prefetch + w/o Value Prefetch
- Scan Length = 200,000 which is enough to trigger Synchronous Index Read
- Show about 3.6x better P99.9 tail latency
- Channel conflict prevents it from being removed completely

Evaluation – Index Prefetch

- **Effect of Index Prefetching**

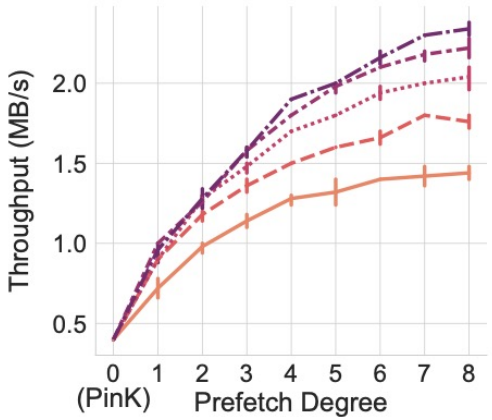


- **IterKVSSD-B**: Baseline w/o prefetch
- **IterKVSSD-I**: w/ Index Prefetch + w/o Value Prefetch
- Scan Length = 200,000 which is enough to trigger Synchronous Index Read
- Show about 3.6x better P99.9 tail latency
- Channel conflict prevents it from being removed completely

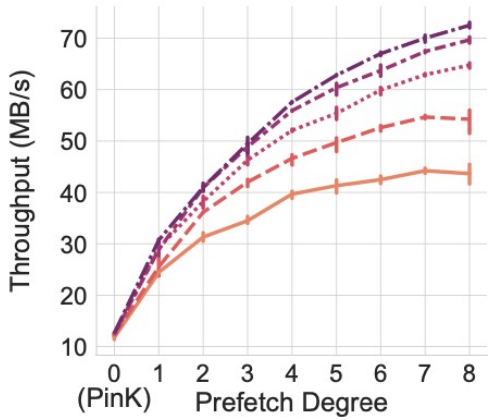
Evaluation – Value Prefetch

- **Effect of Value Prefetching**
  - Evaluated with ***SeekRandom*** workload
  - Prefetch Degree: 0 – 8
  - Value Size : 128B, 4KB, 16KB, 128KB
  - Scan Length: 128, 256, 512, 1024, 2048

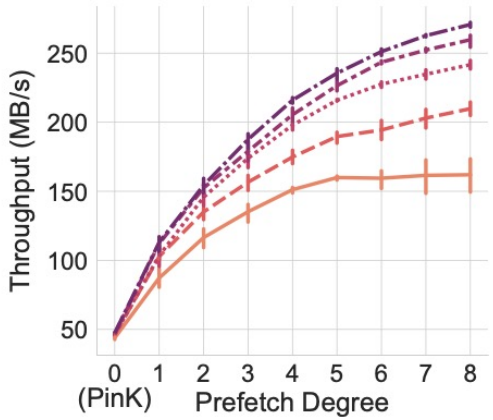
Evaluation – Value Prefetch



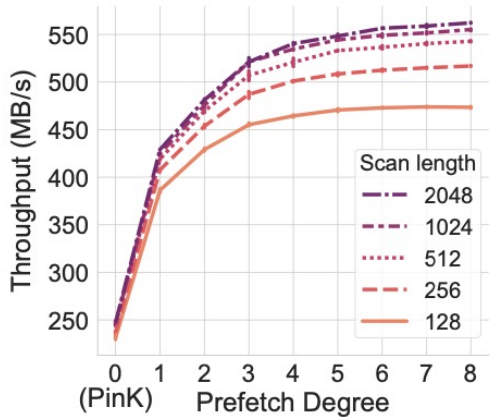
(a) 128 B



(b) 4 KB



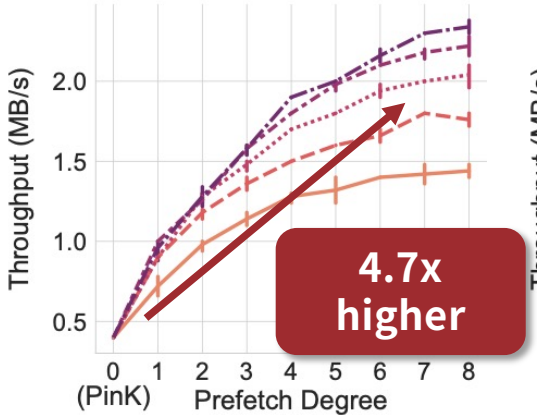
(c) 16 KB



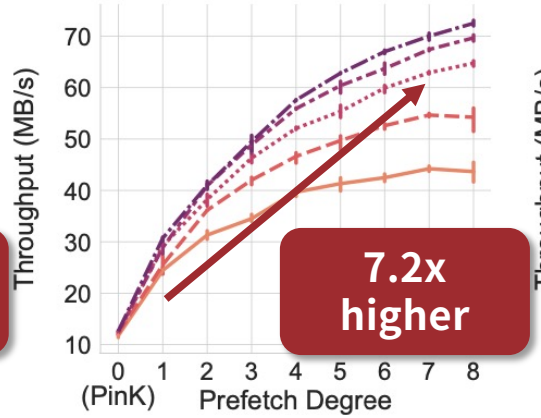
(d) 128 KB

Evaluation – Value Prefetch

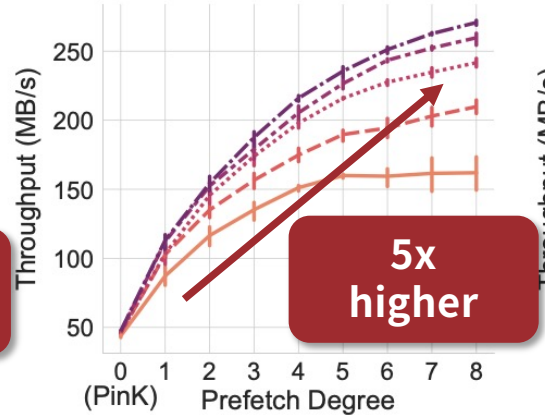
- **Effect of Prefetching Degree**
  - With higher prefetch degree, better I/O performance because small prefetch degree is not enough to hide NAND Flash access latency completely
  - However, prefetch degree over some degree will not improve performance
    - When internal bandwidth is saturated
    - When NAND Access can be fully overlapped with other steps



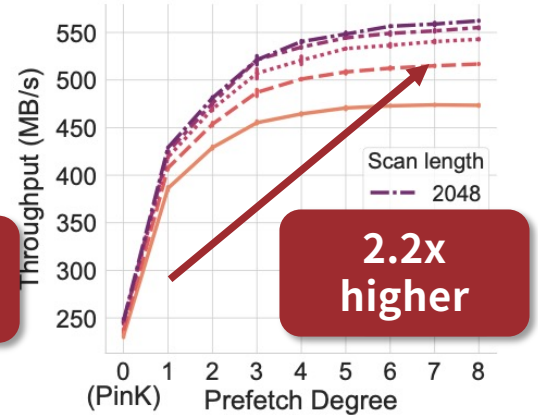
(a) 128 B



(b) 4 KB



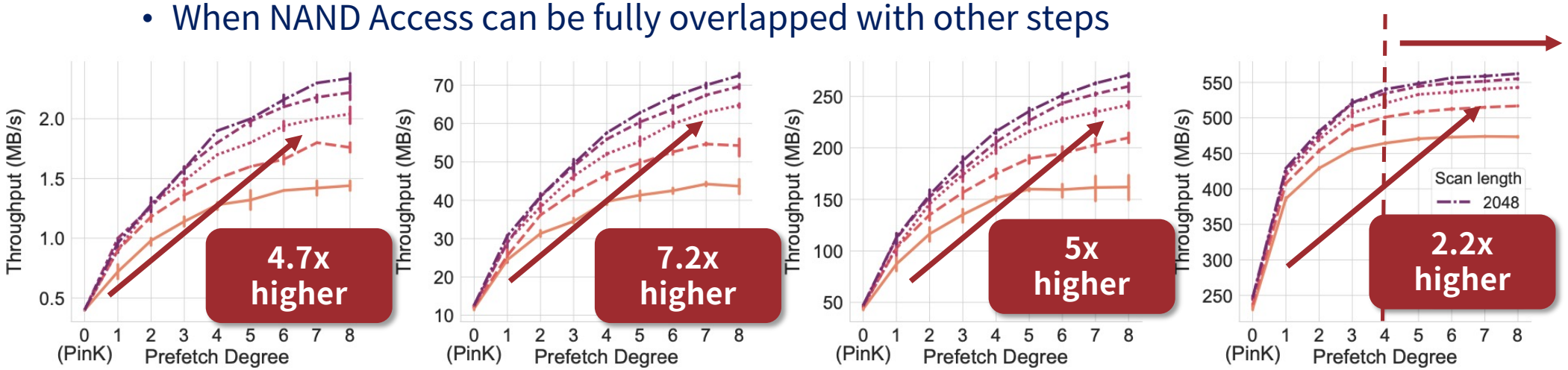
(c) 16 KB



(d) 128 KB

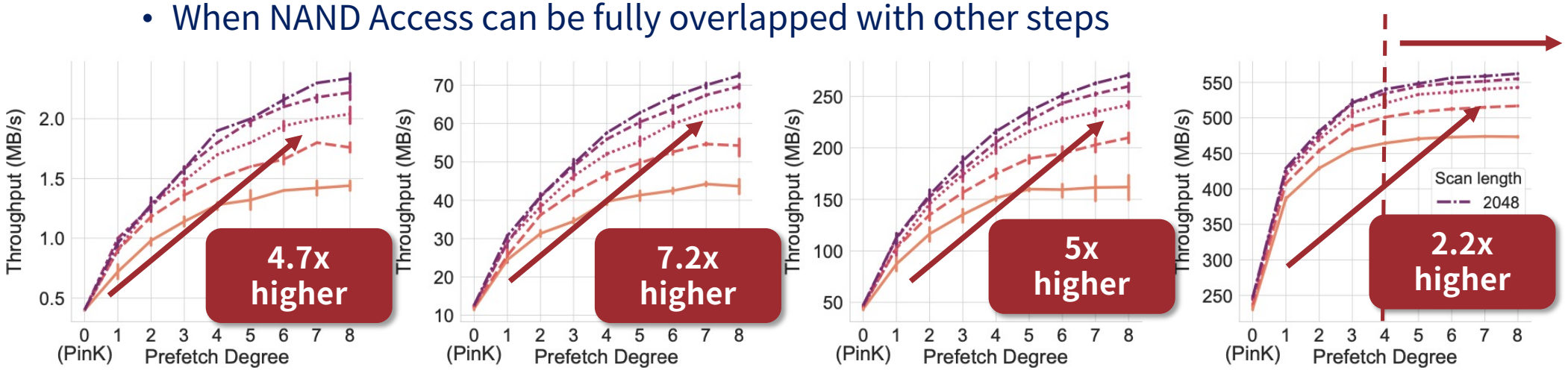
Evaluation – Value Prefetch

- **Effect of Prefetching Degree**
  - With higher prefetch degree, better I/O performance because small prefetch degree is not enough to hide NAND Flash access latency completely
  - However, prefetch degree over some degree will not improve performance
    - When internal bandwidth is saturated
    - When NAND Access can be fully overlapped with other steps



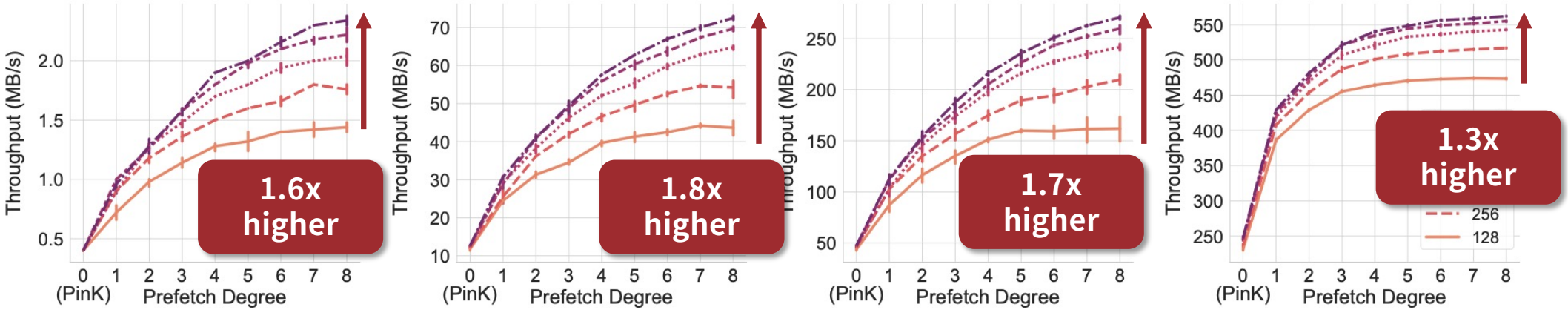
Evaluation – Value Prefetch

- **Effect of Prefetching Degree**
  - With higher prefetch degree, better I/O performance because small prefetch degree is not enough to hide NAND Flash access latency completely
  - However, prefetch degree over some degree will not improve performance
    - When internal bandwidth is saturated
    - When NAND Access can be fully overlapped with other steps



Evaluation – Value Prefetch

- **Effect of Scan Length in Range Query**
  - With higher scan length, better I/O throughput



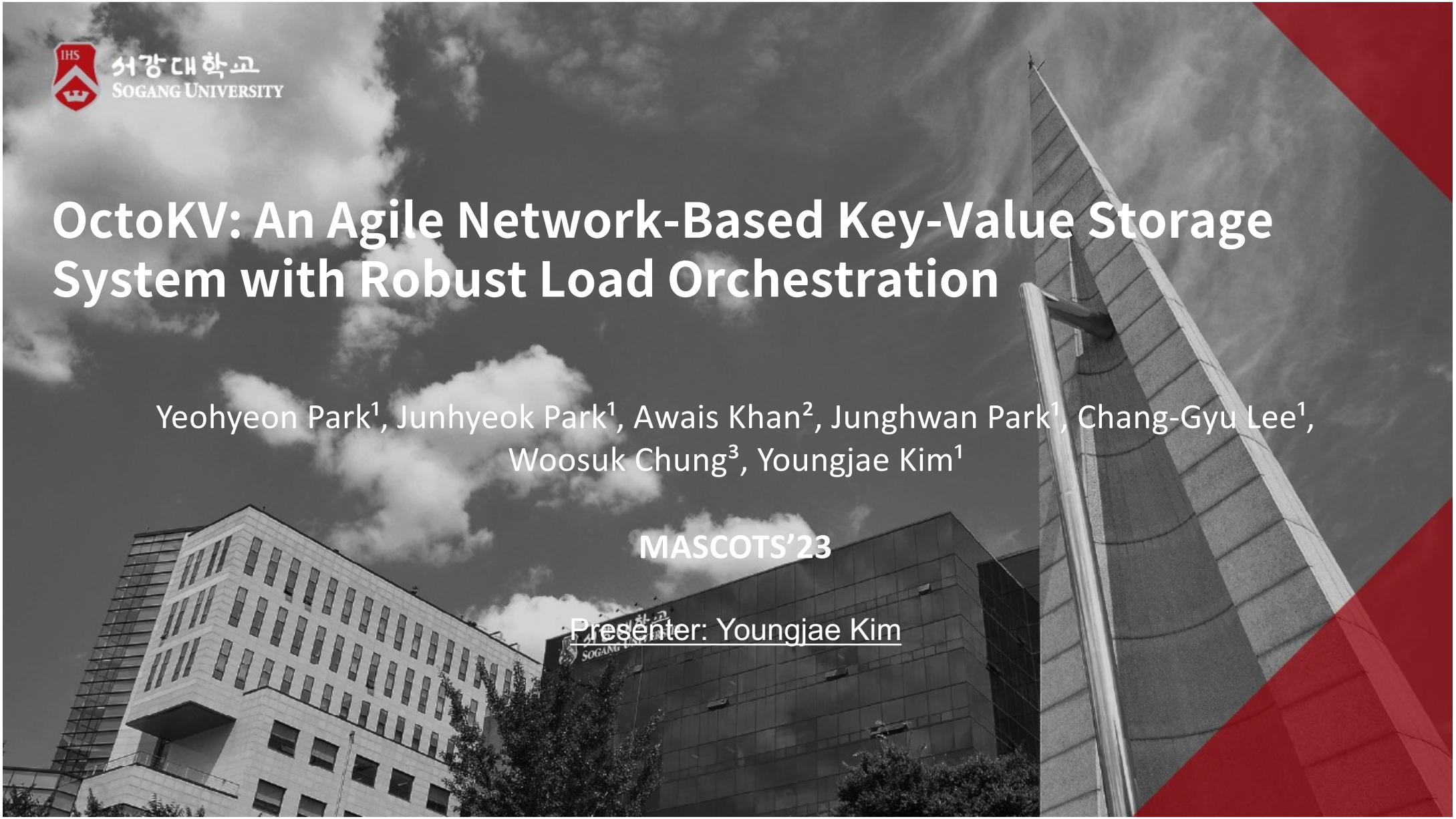
(a) 128 B (b) 4 KB (c) 16 KB (d) 128 KB

Conclusion

- **Explored three problems of current iterator interface**
  1. Versioning Problem
  2. Synchronous Index Read Problem
  3. Synchronous Value Read Problem

- **Proposed IterKVSSD**
  - Memory-efficient Versioning through decoupling and pooling metadata
  - Index/Value Prefetch to mitigate NAND Flash Access Penalty

- **Showed 2x lower P99.9 tail latency, up to 7.2x better I/O throughput**



OctoKV: An Agile Network-Based Key-Value Storage System with Robust Load Orchestration

Yeohyeon Park[1], Junhyeok Park[1], Awais Khan[2], Junghwan Park[1], Chang-Gyu Lee[1], Woosuk Chung[3], Youngjae Kim[1]

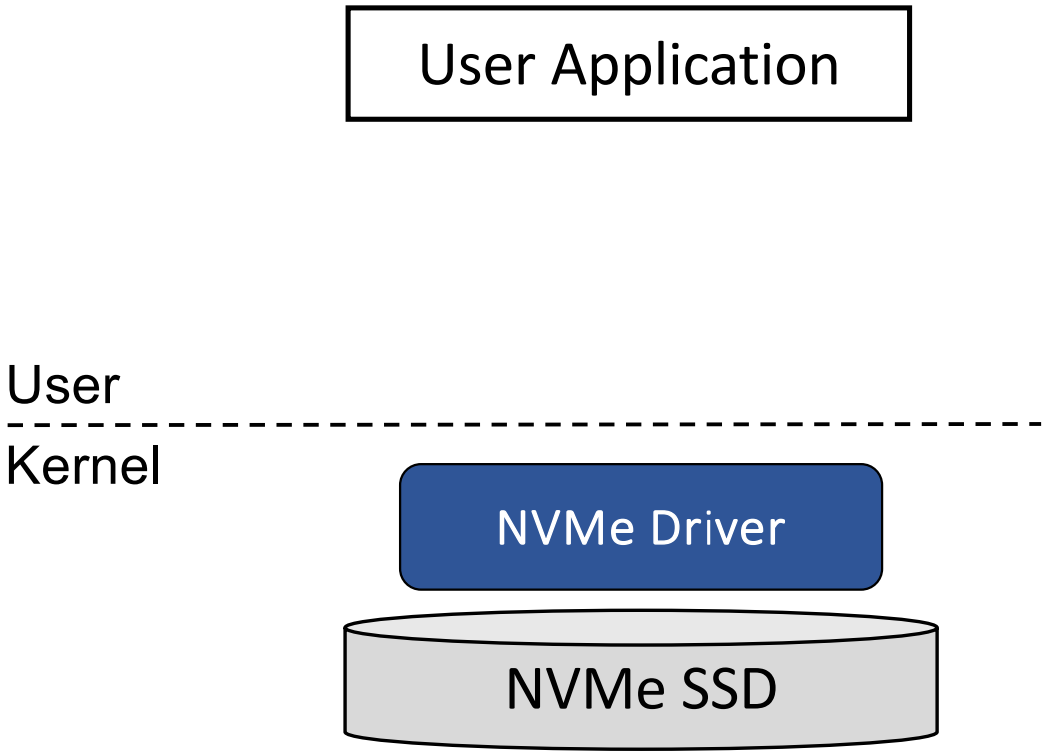
MASCOTS'23

Presenter: Youngjae Kim

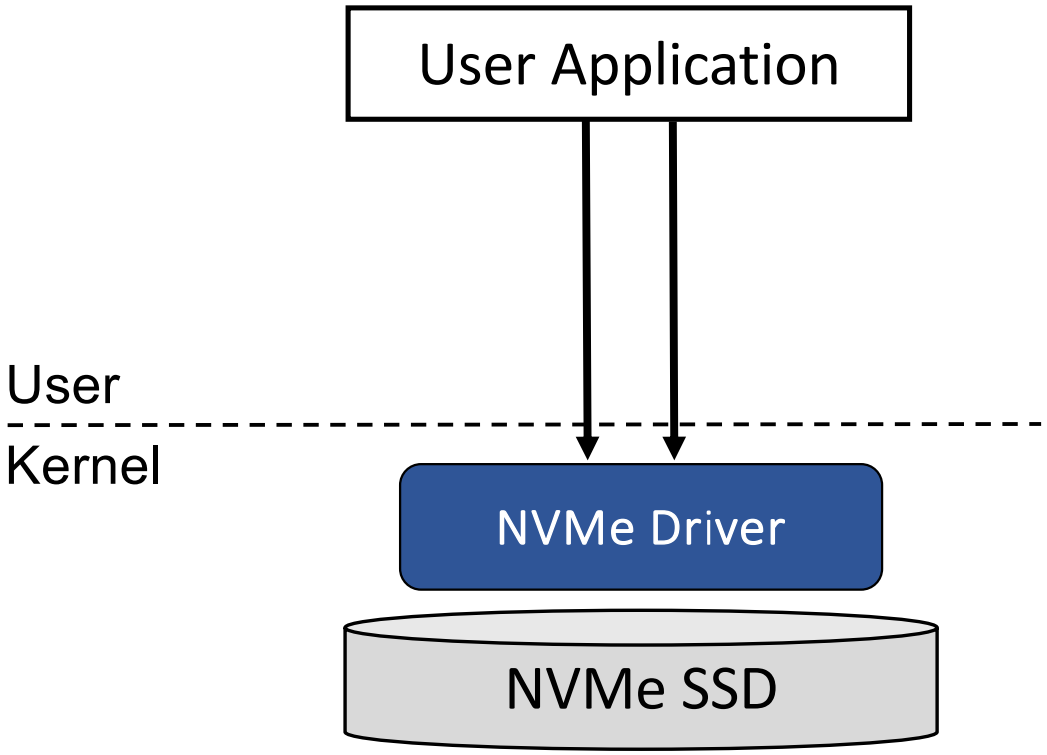
Content

- Background
- Problem Definition
- Motivational Experiments
- OctoKV: Design and Implementation
- Evaluation
- Conclusion

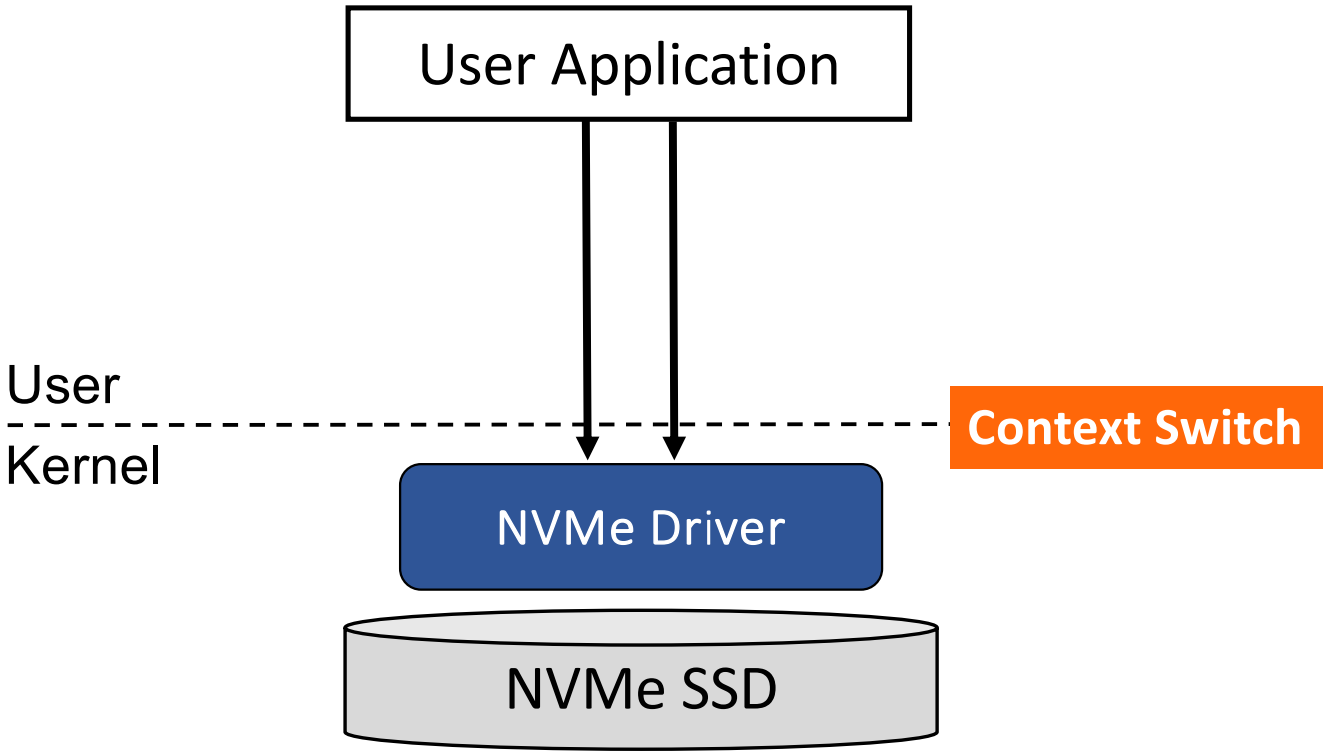
(1) User Space NVMe Driver



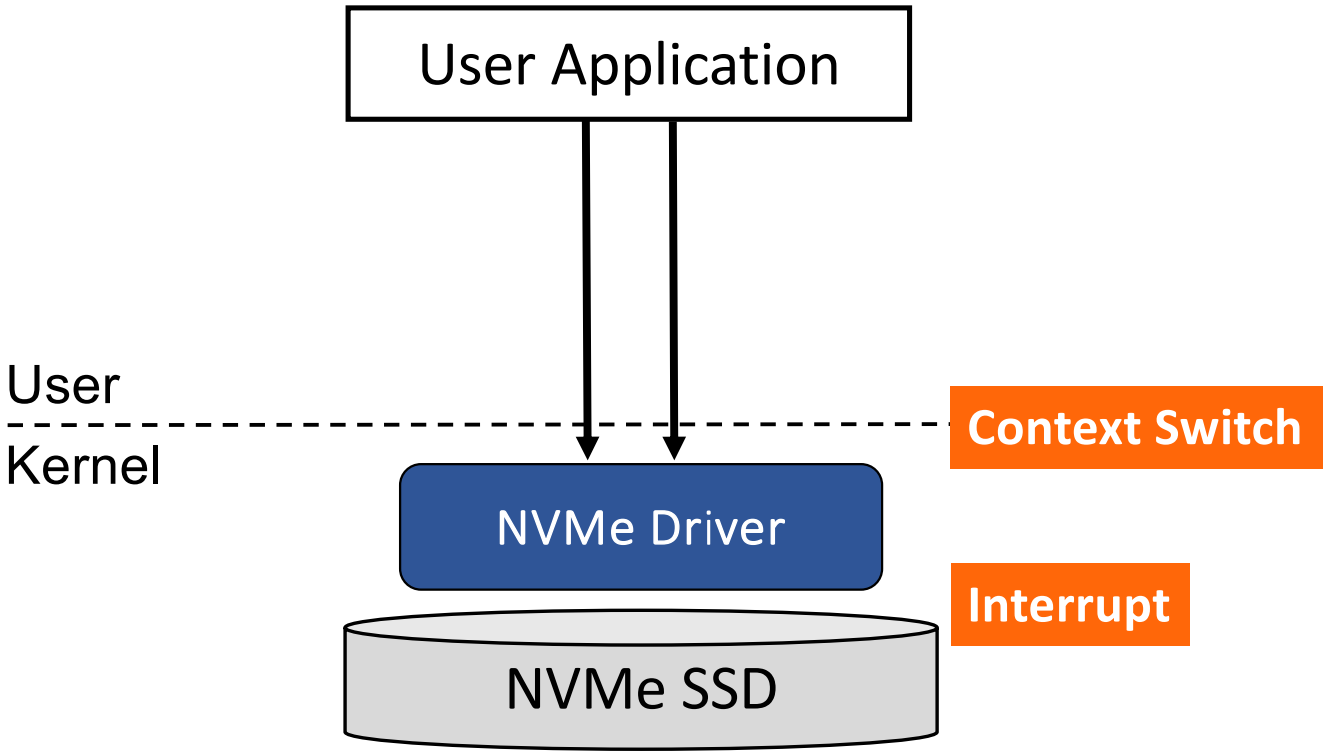
(1) User Space NVMe Driver



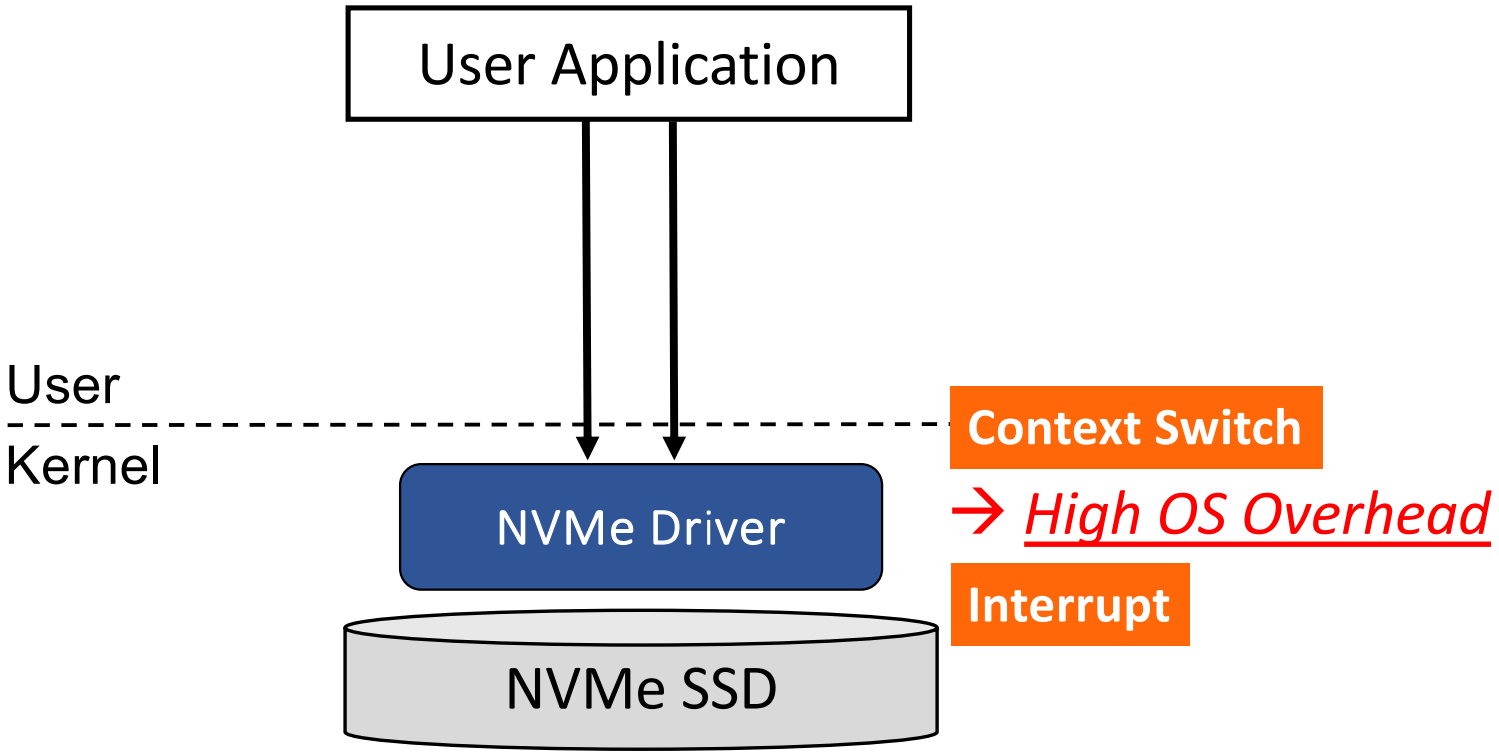
(1) User Space NVMe Driver



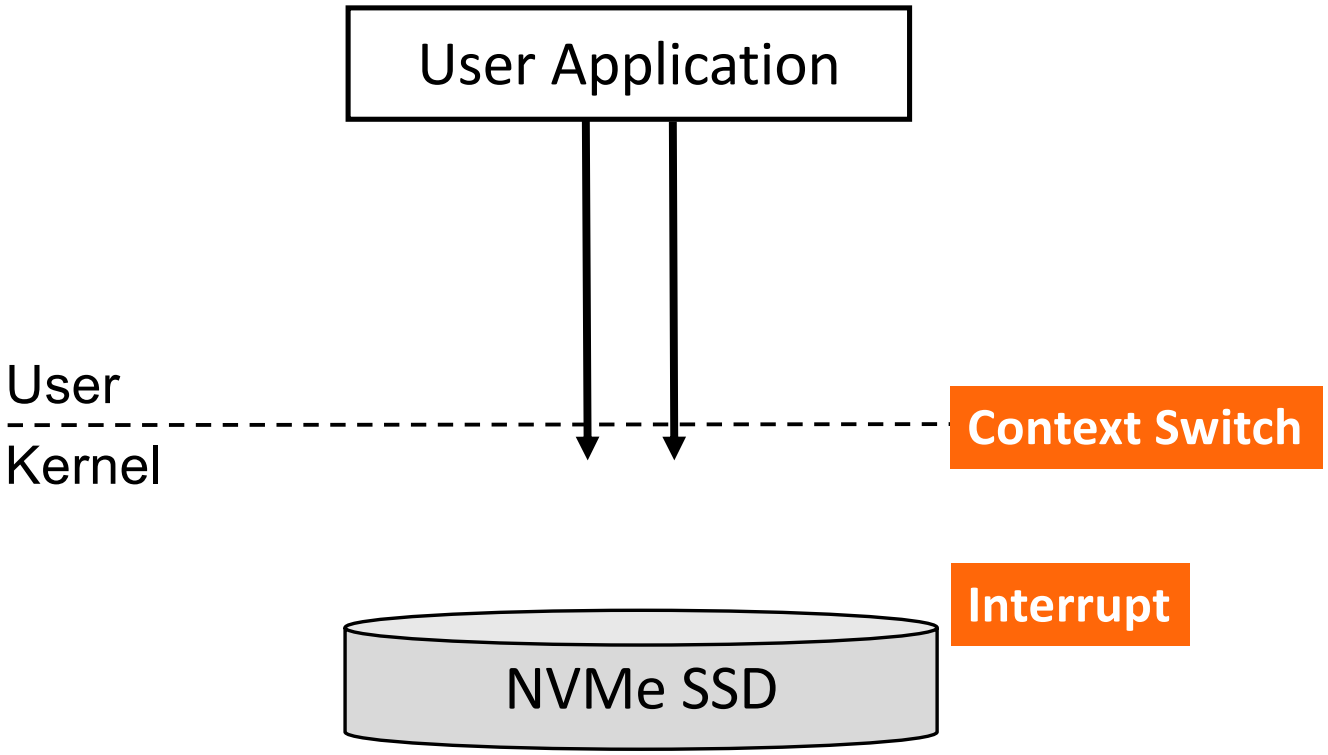
(1) User Space NVMe Driver



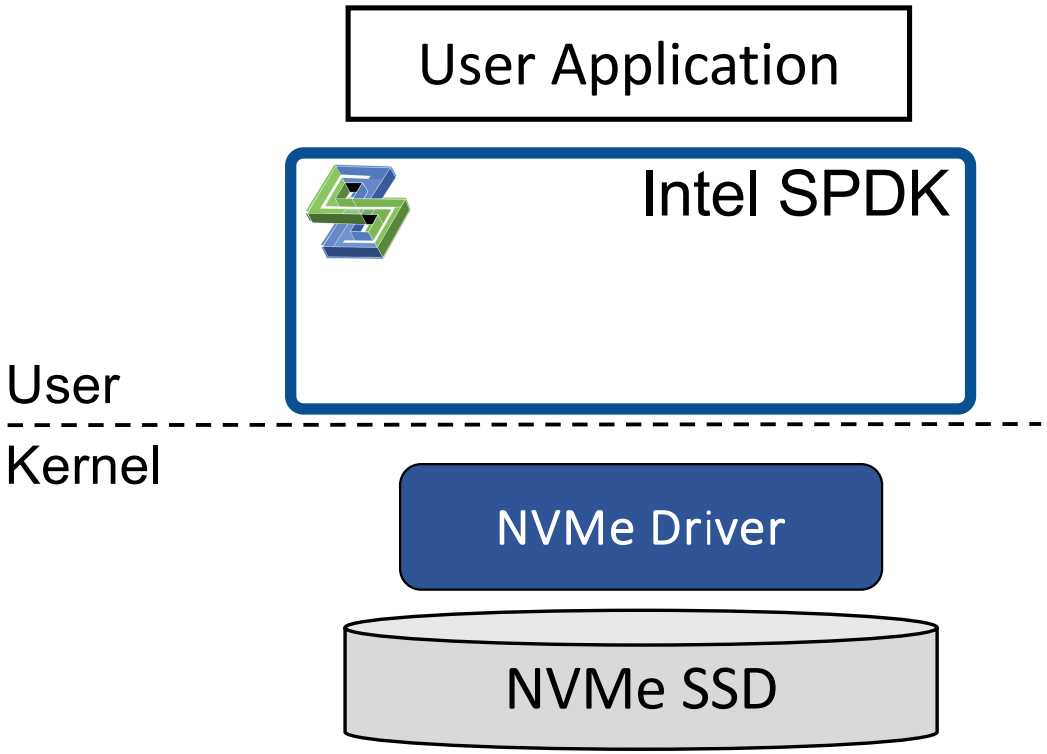
(1) User Space NVMe Driver



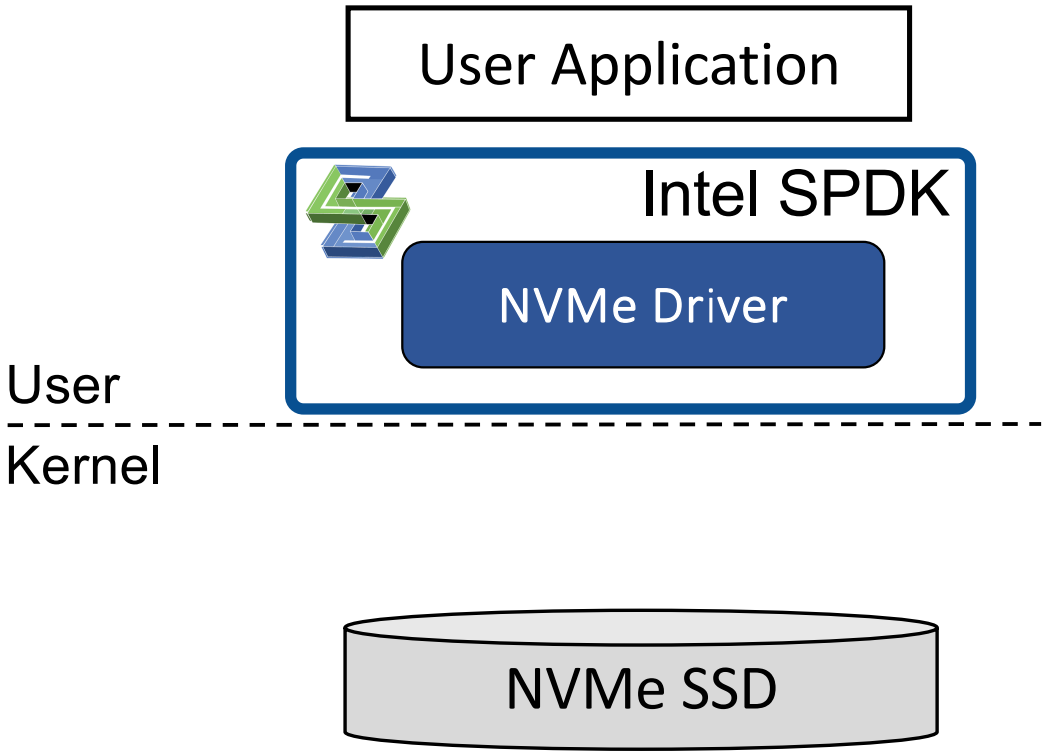
(1) User Space NVMe Driver



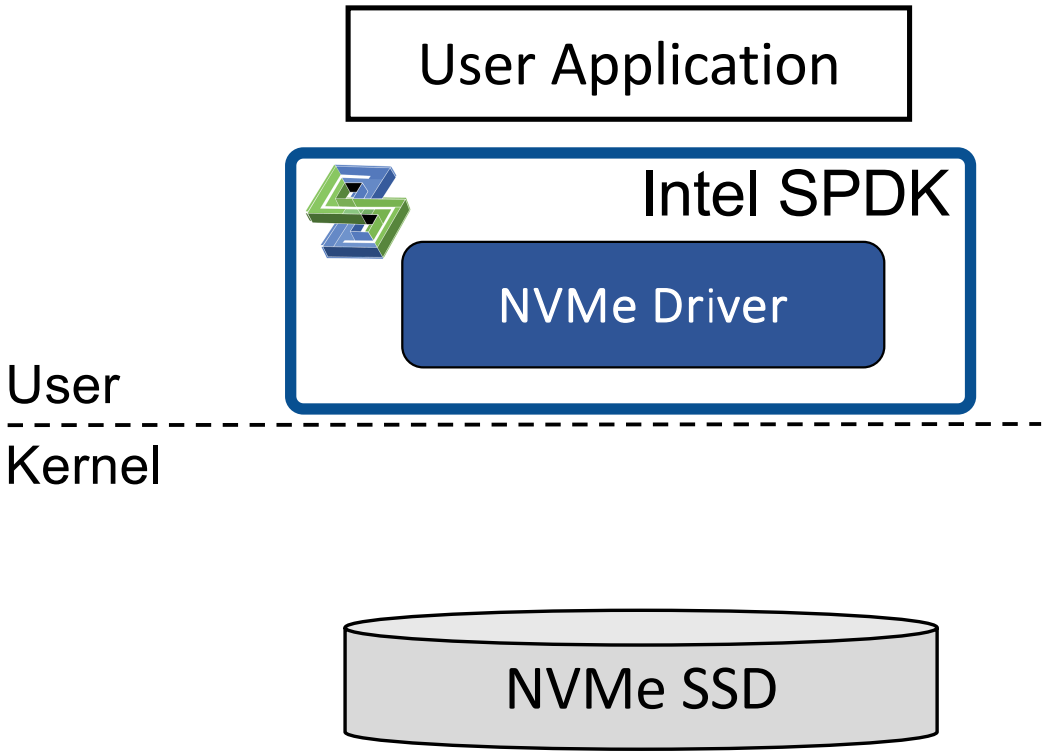
(1) User Space NVMe Driver



(1) User Space NVMe Driver

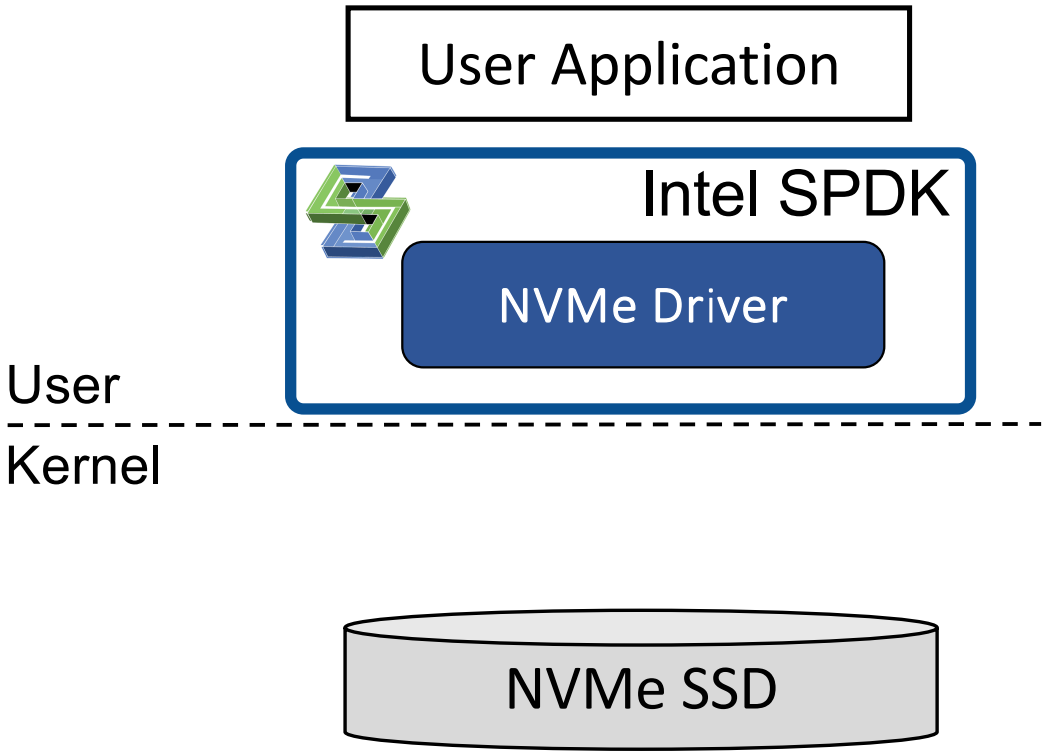


(1) User Space NVMe Driver



1. User level NVMe driver

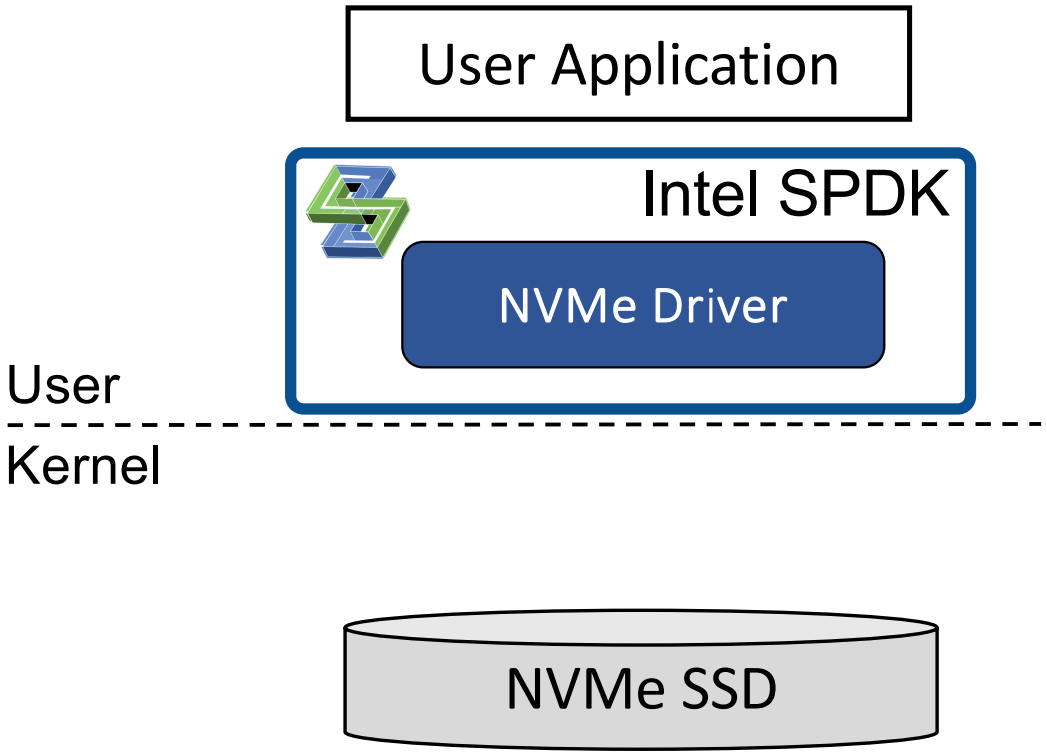
(1) User Space NVMe Driver



1. User level NVMe driver

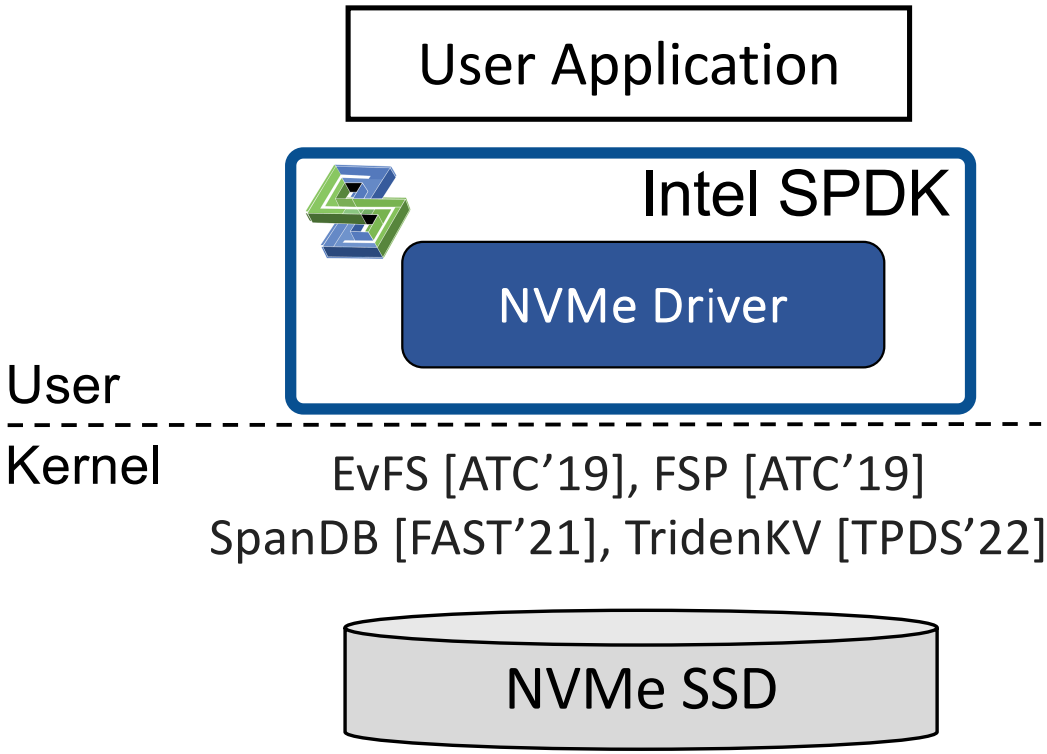
2. Bind I/O to a specific core

(1) User Space NVMe Driver



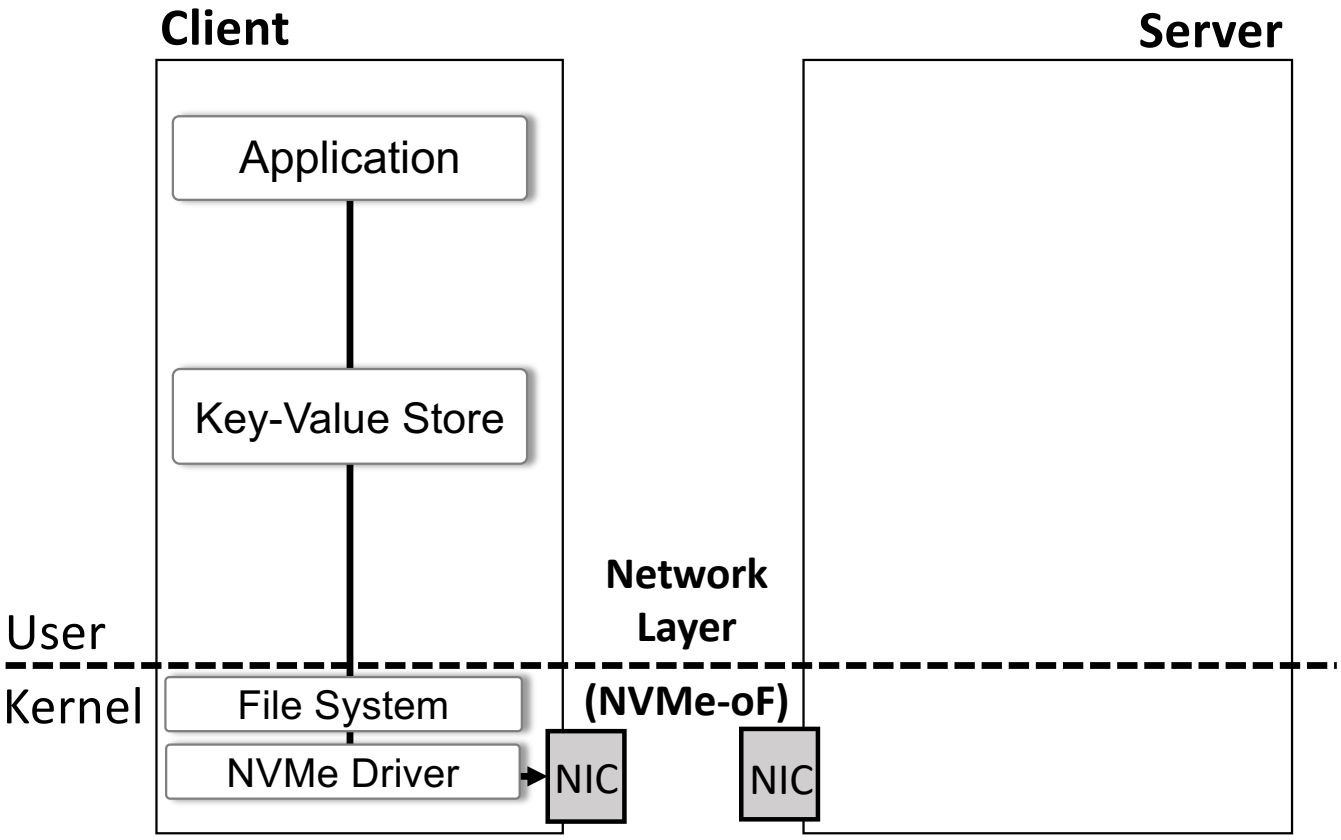
1. User level NVMe driver
2. Bind I/O to a specific core
3. Use polling for completion

(1) User Space NVMe Driver

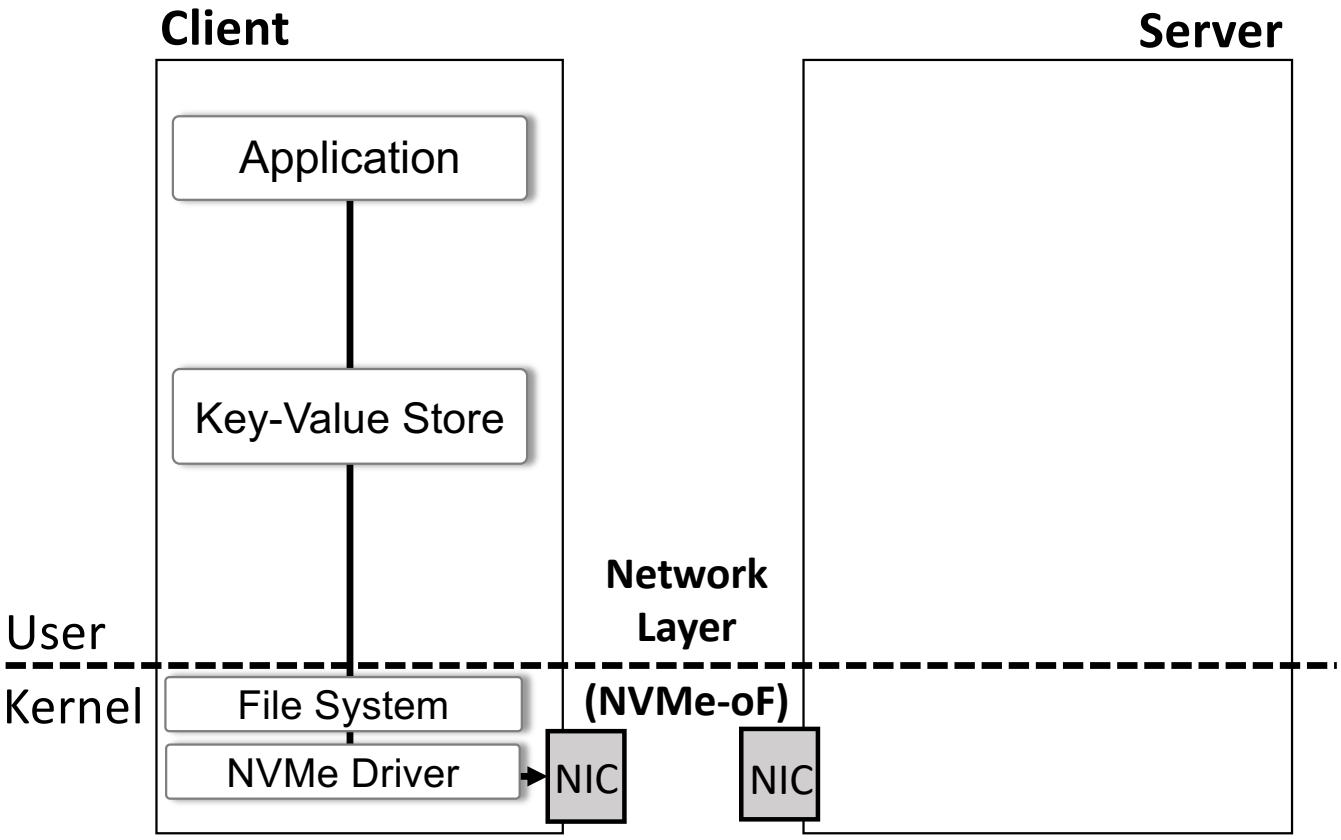


1. User level NVMe driver
2. Bind I/O to a specific core
3. Use polling for completion

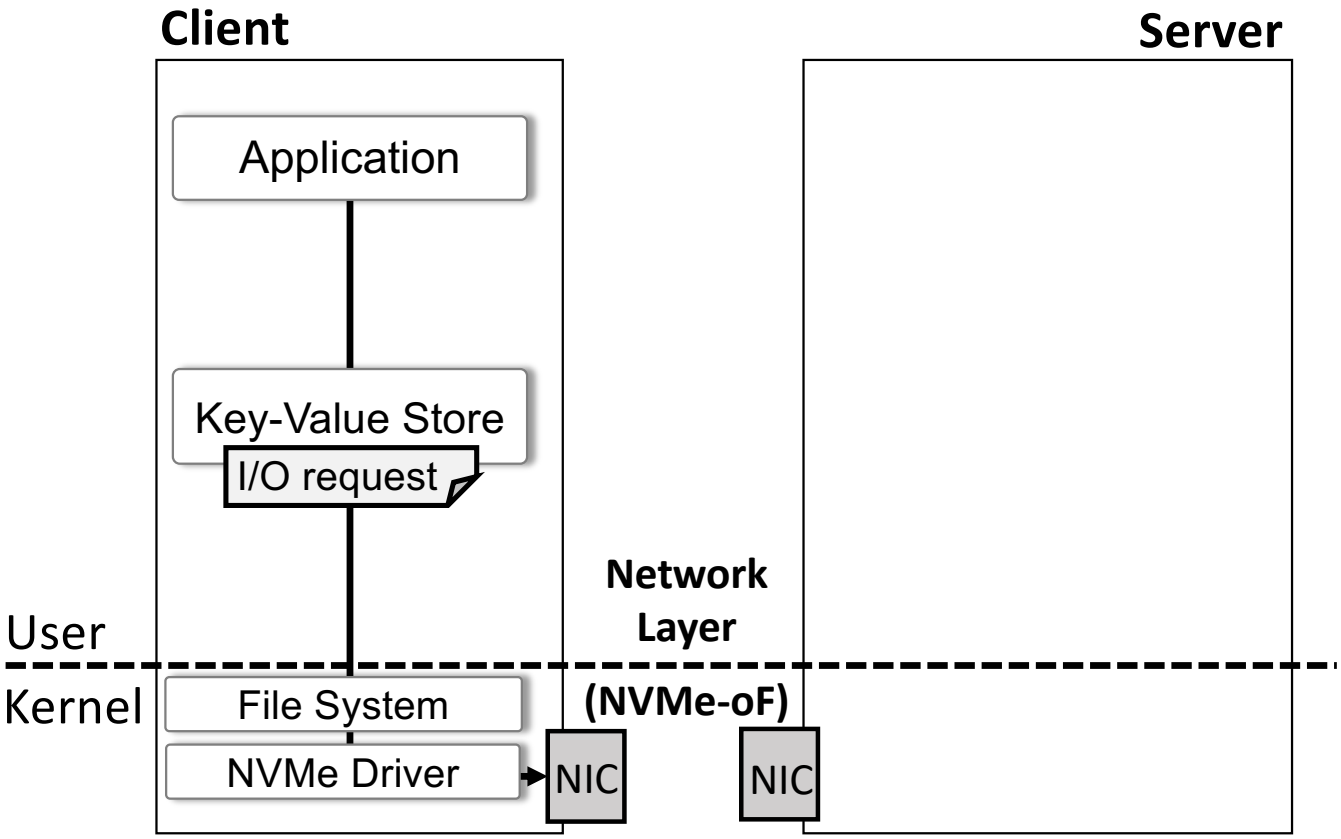
(2) SPDK-based Network-based Block Storage



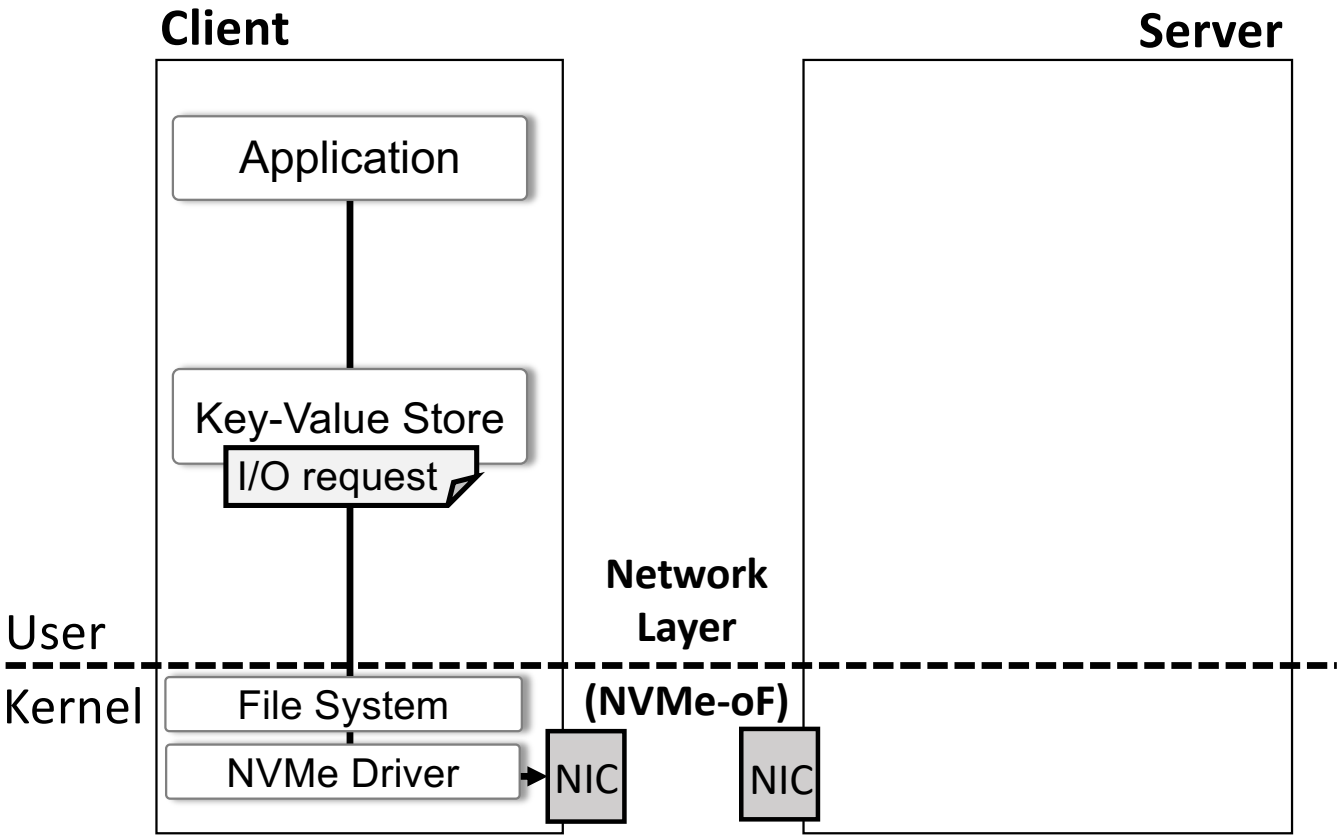
(2) SPDK-based Network-based Block Storage



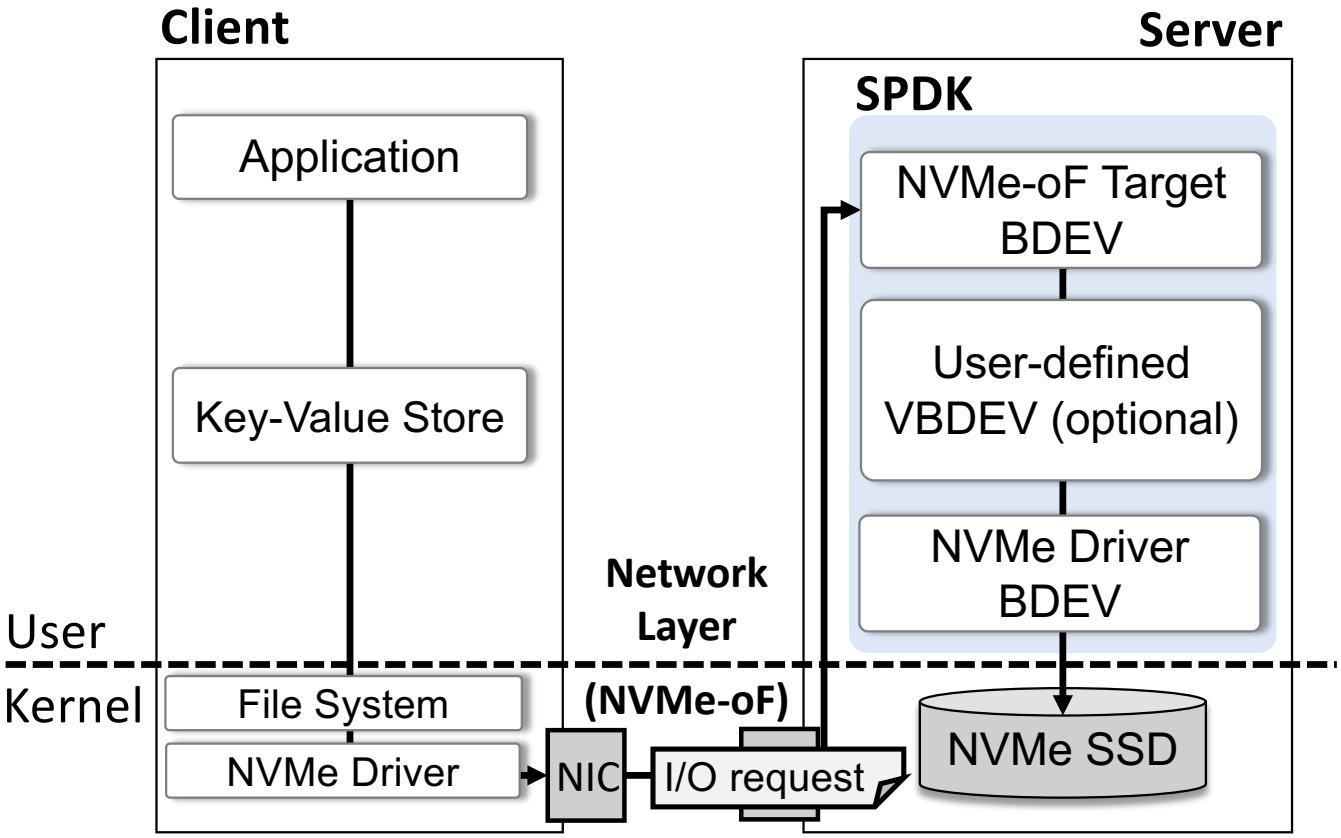
(2) SPDK-based Network-based Block Storage



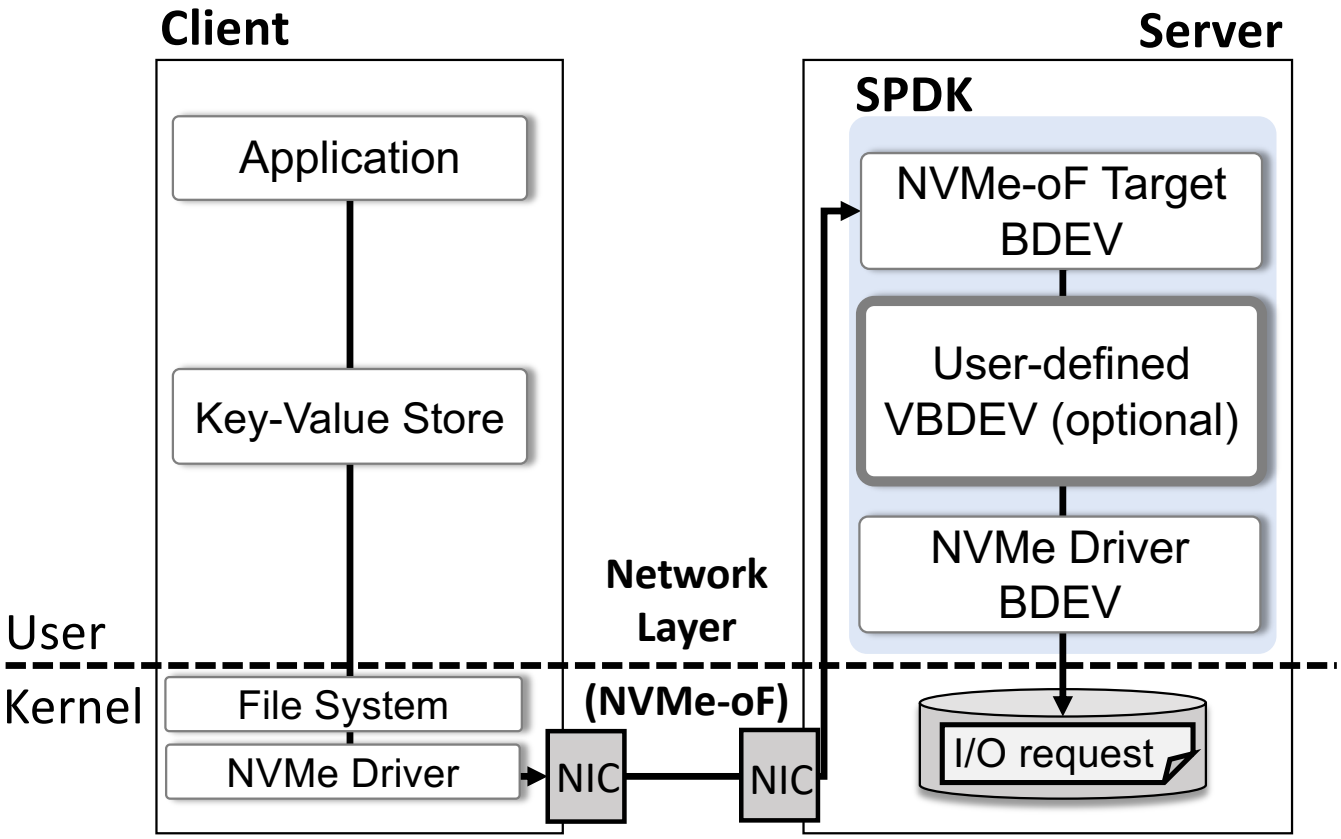
(2) SPDK-based Network-based Block Storage



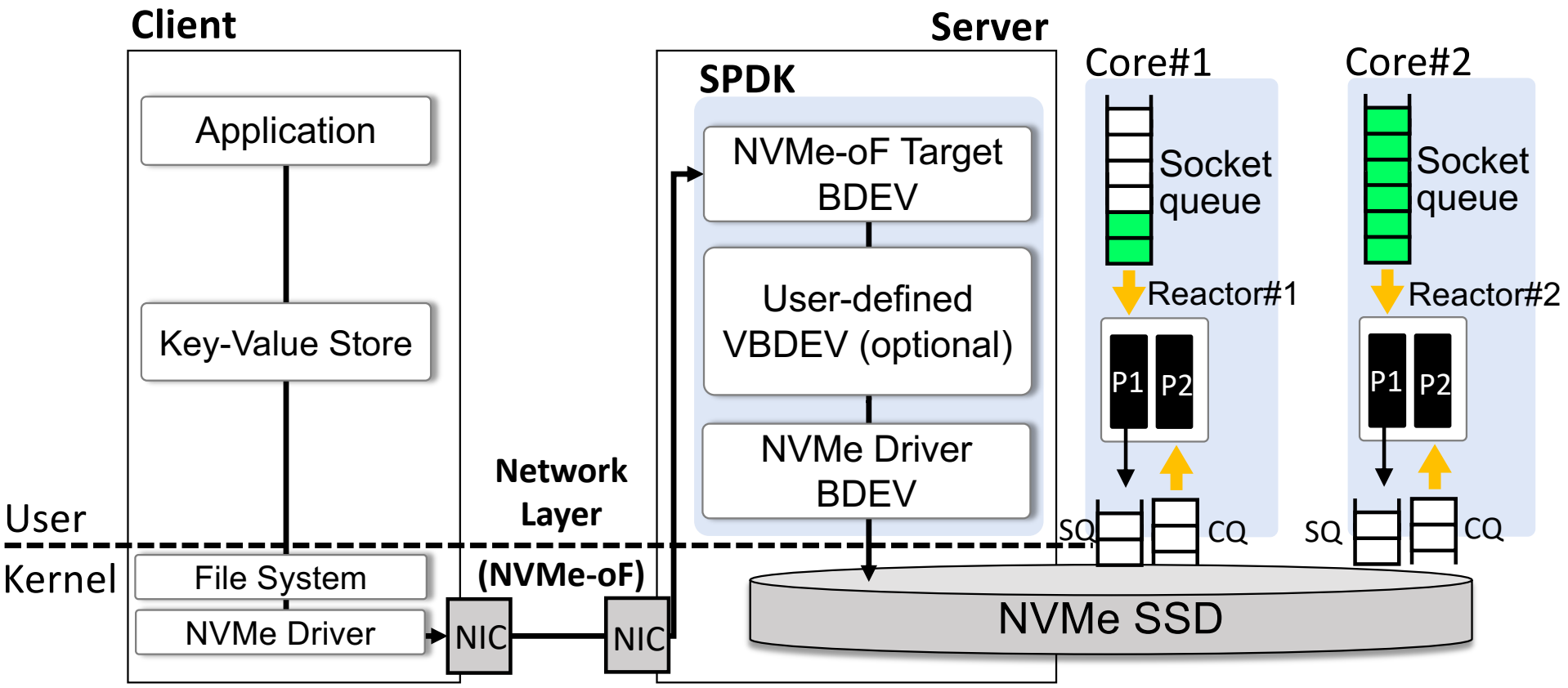
(2) SPDK-based Network-based Block Storage



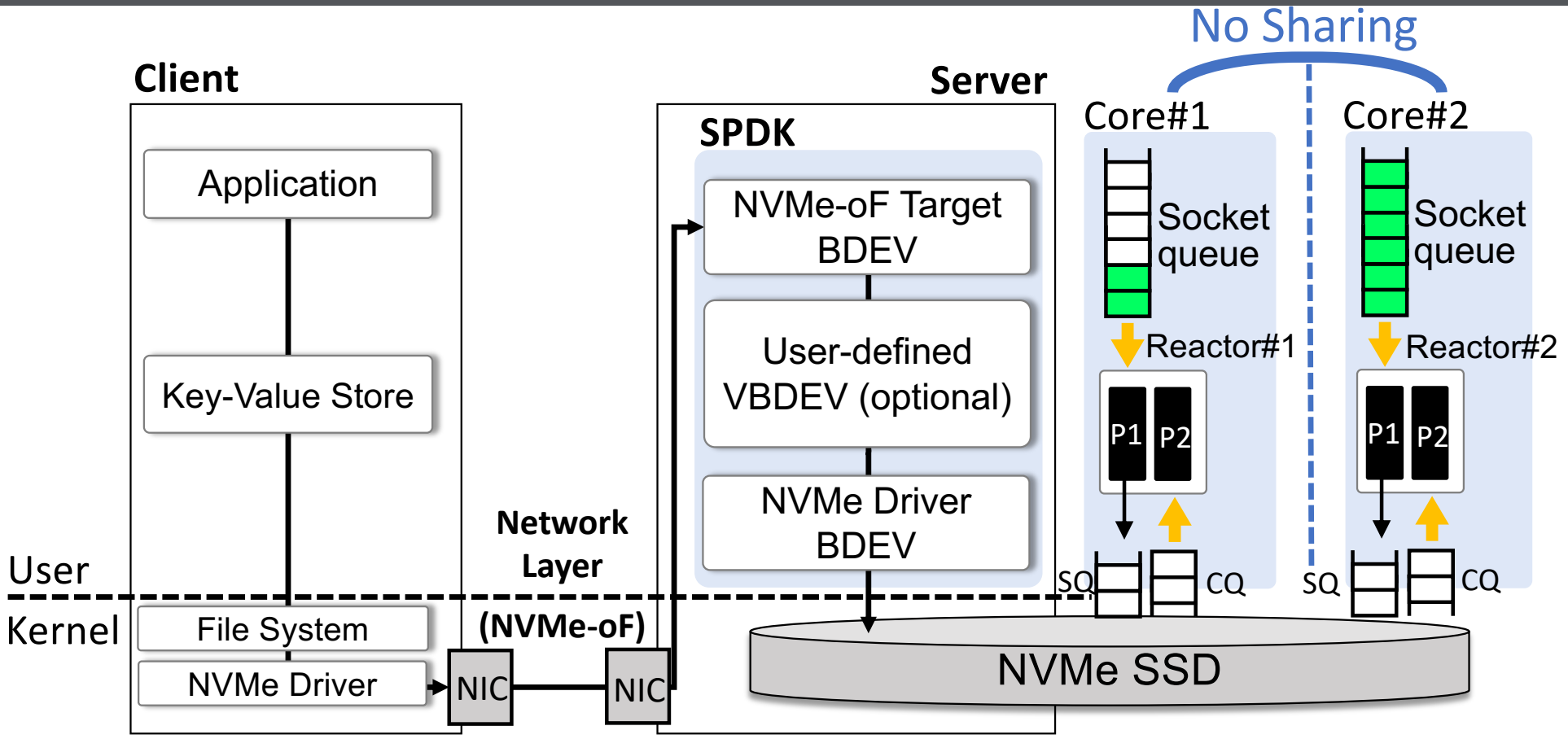
(2) SPDK-based Network-based Block Storage



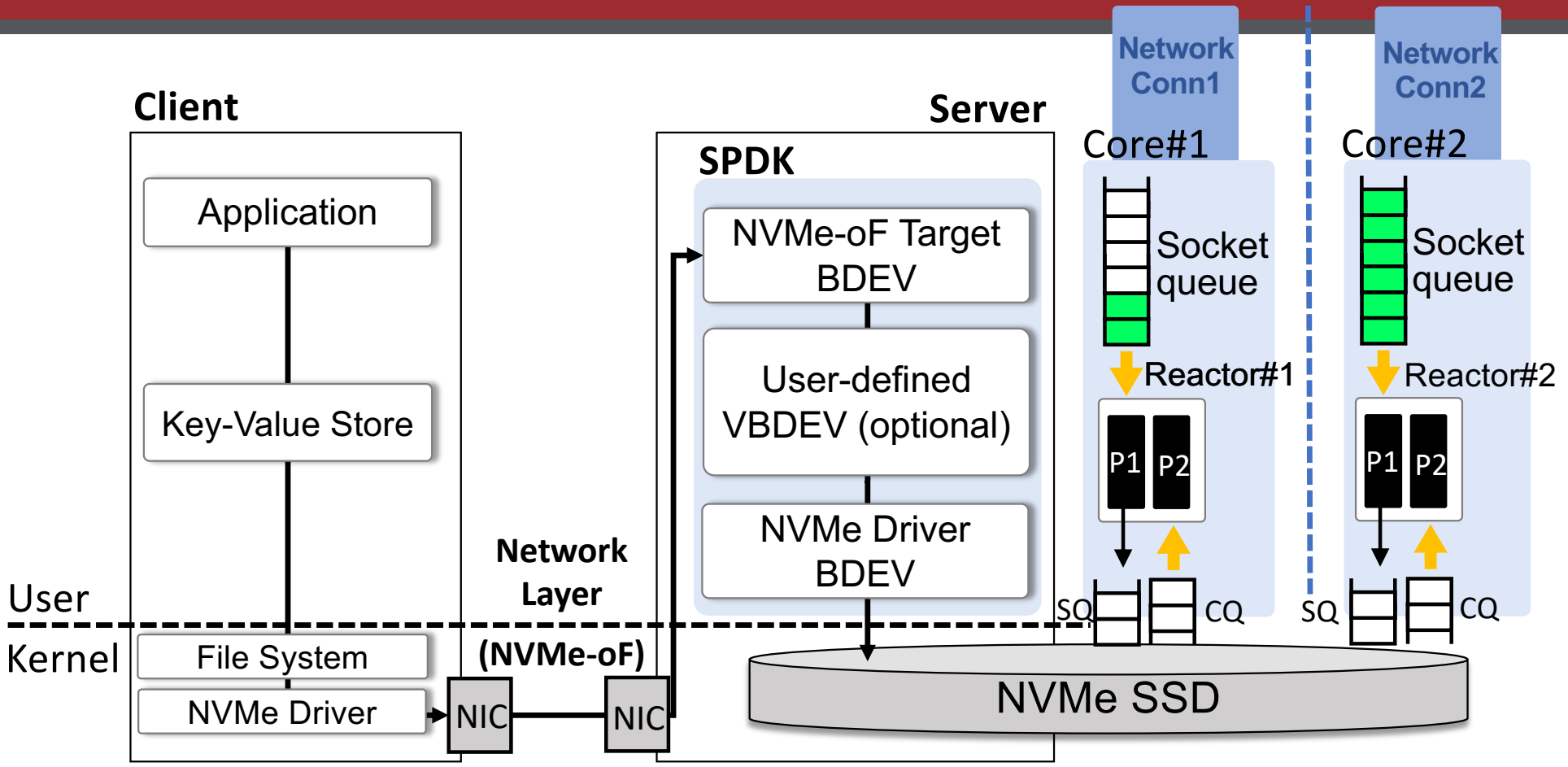
(2) SPDK-based Network-based Block Storage



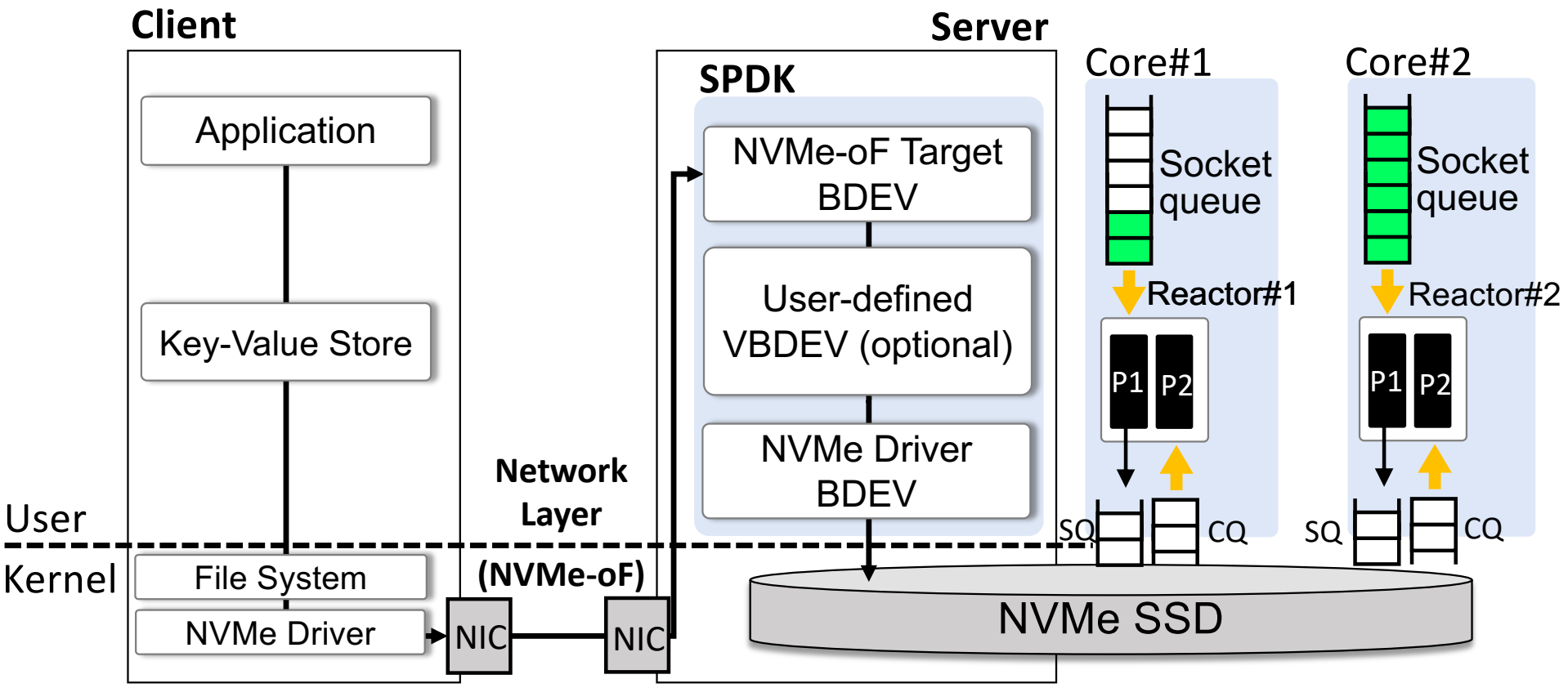
(2) SPDK-based Network-based Block Storage



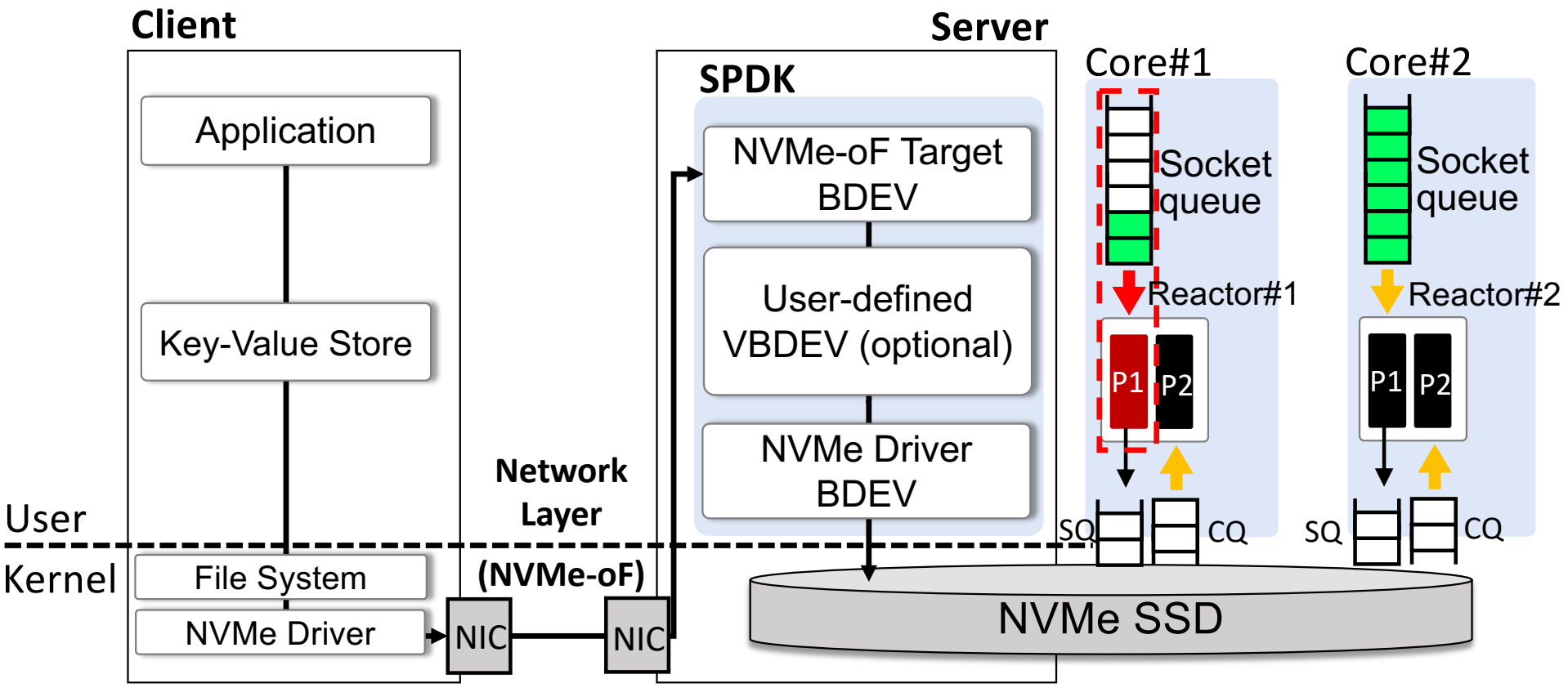
(2) SPDK-based Network-based Block Storage



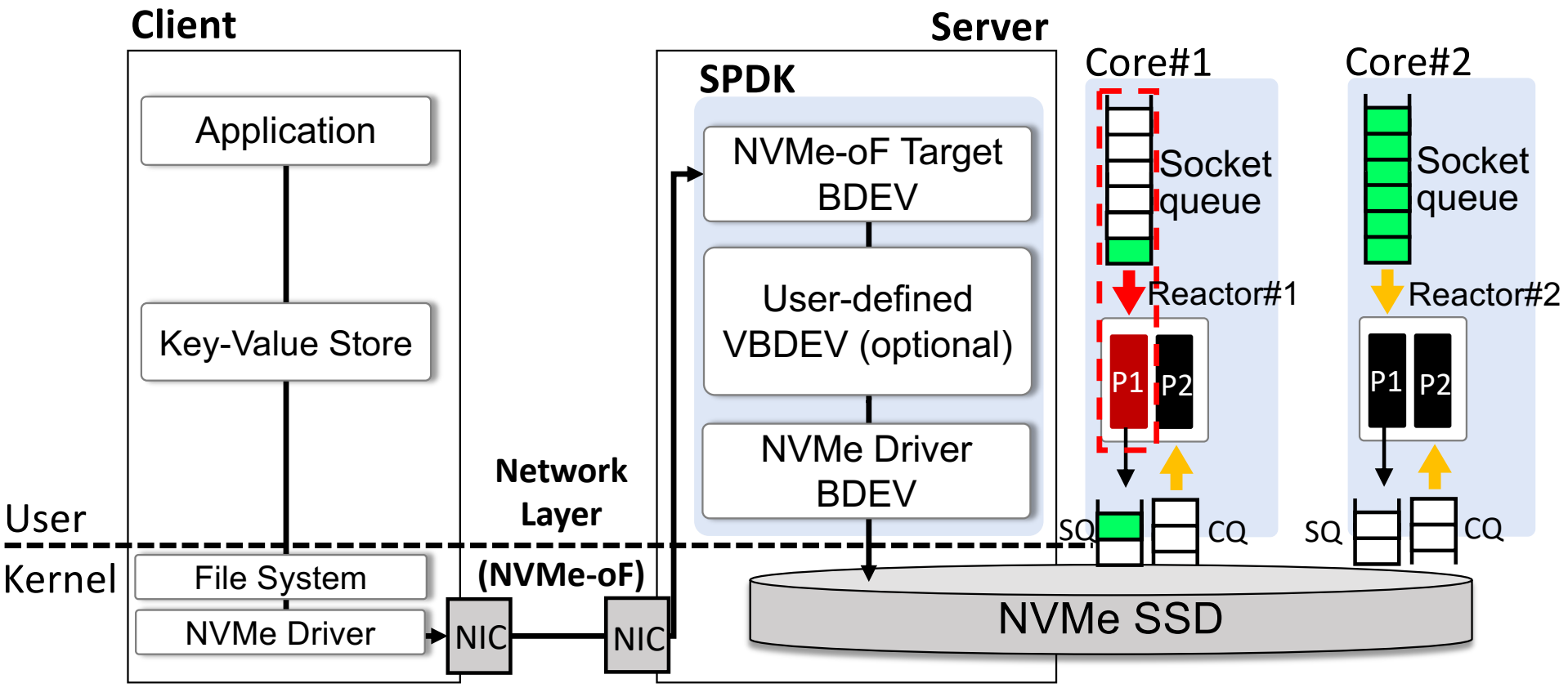
(2) SPDK-based Network-based Block Storage



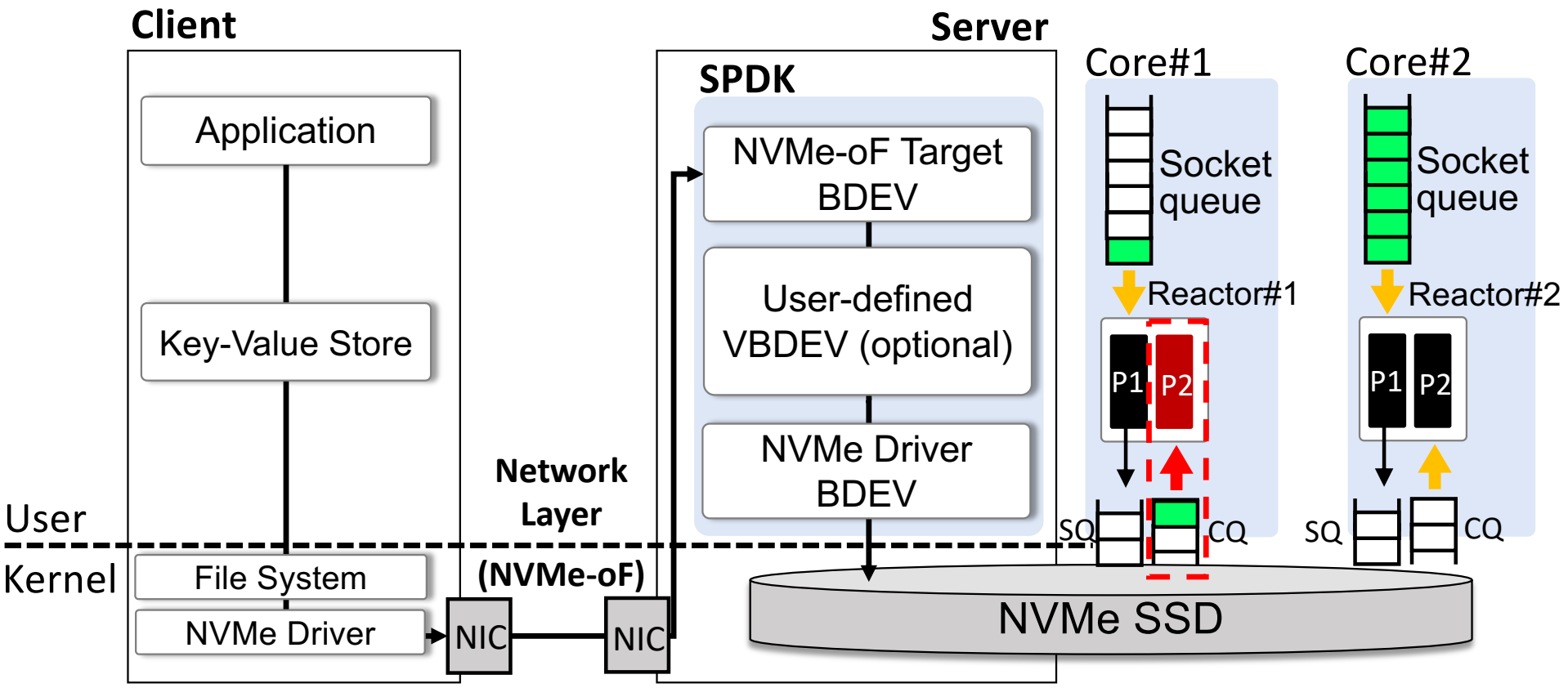
(2) SPDK-based Network-based Block Storage



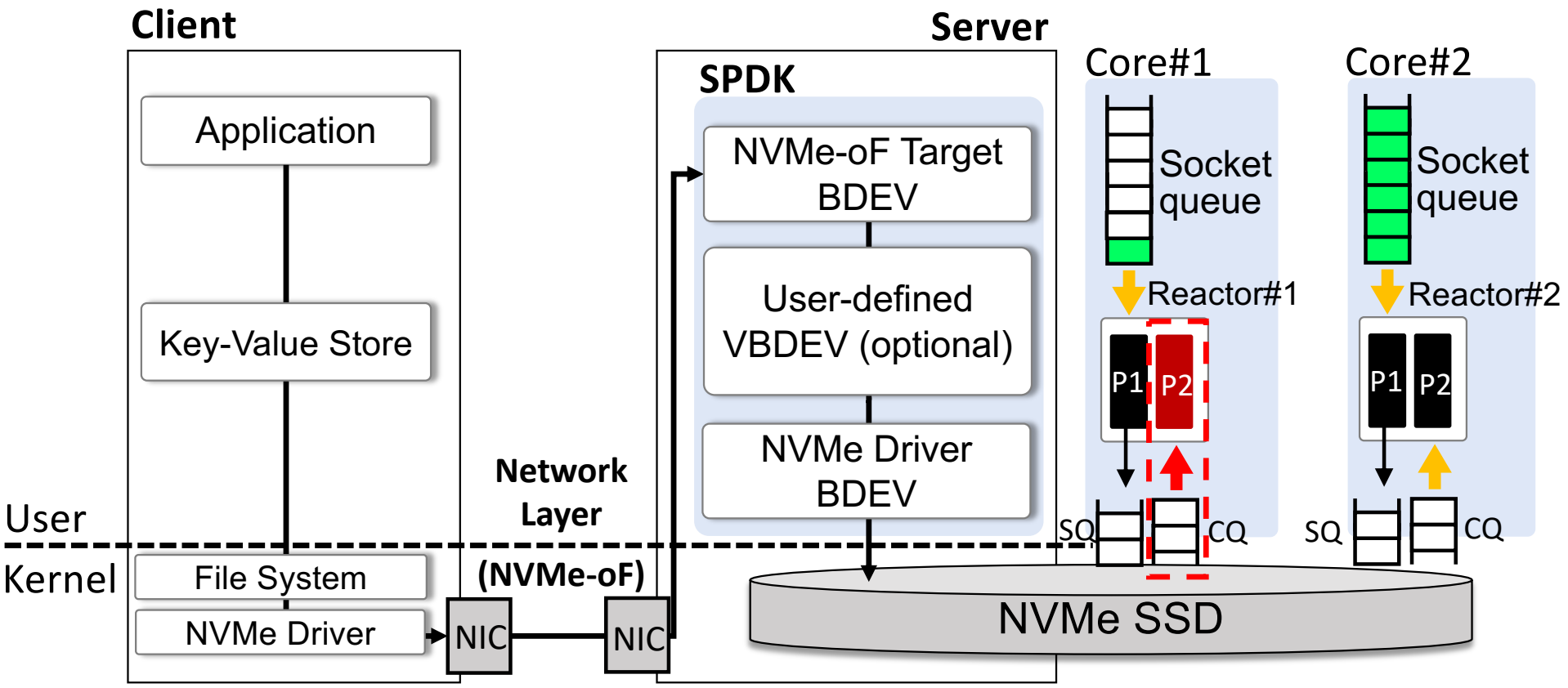
(2) SPDK-based Network-based Block Storage



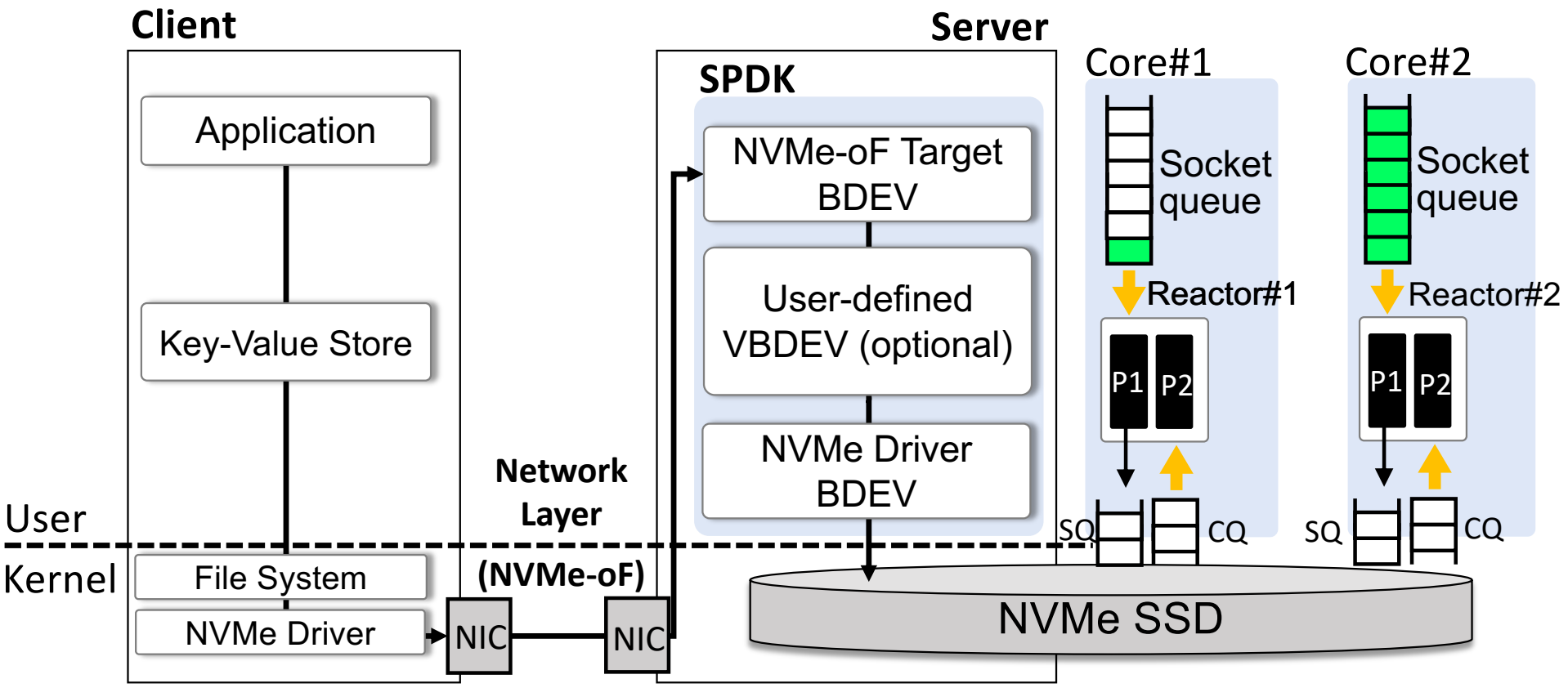
(2) SPDK-based Network-based Block Storage



(2) SPDK-based Network-based Block Storage

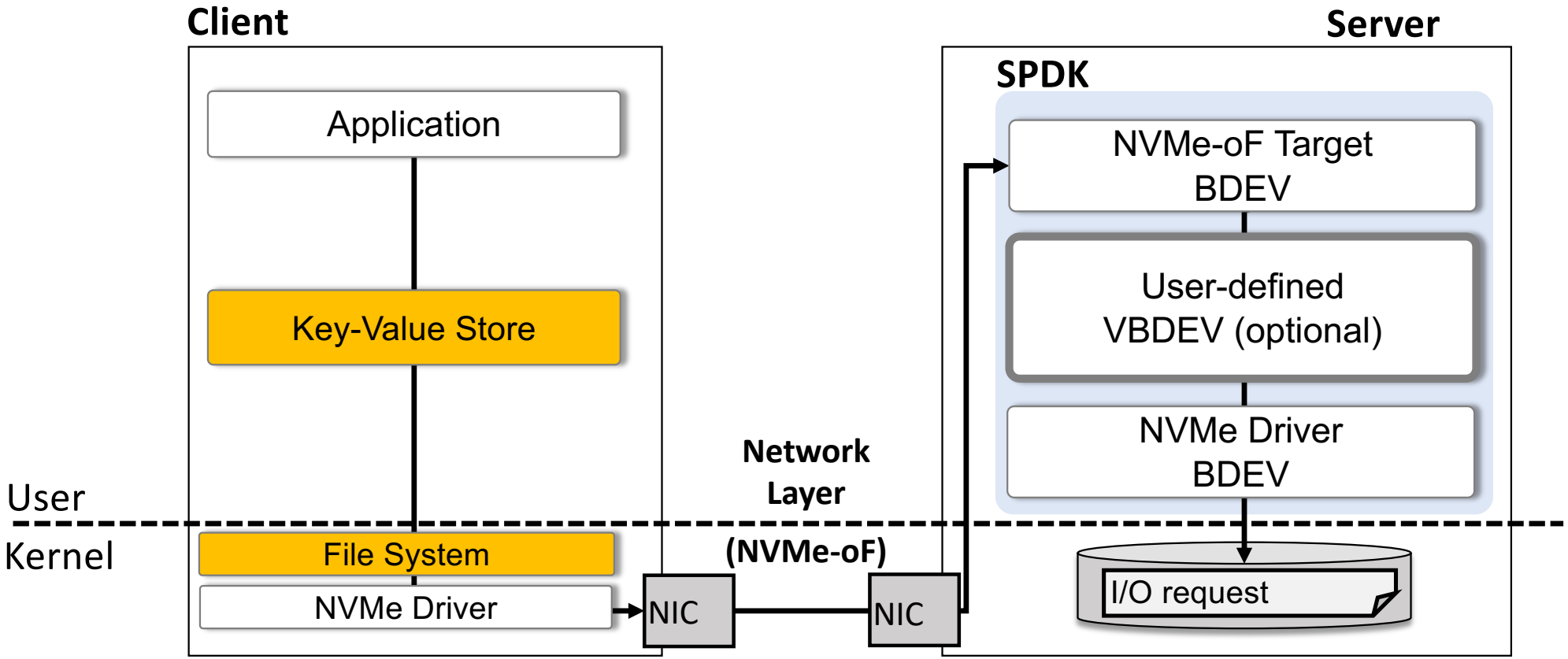


(2) SPDK-based Network-based Block Storage



Target Archiecture

Disaggregated Network-based Key-Value Storage System



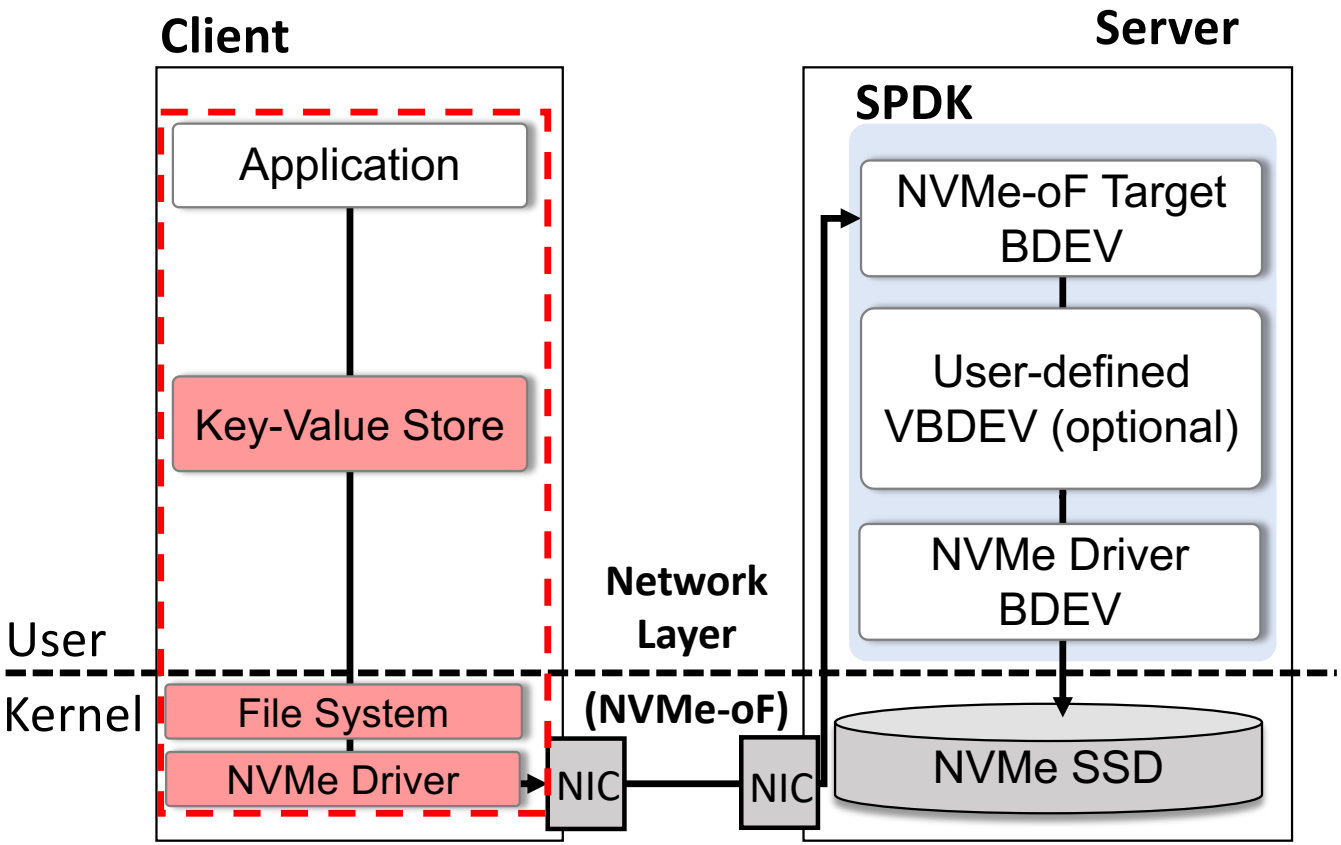
Problem Definition

Disaggregated Networked-based Key-Value Storage System has the following two problems

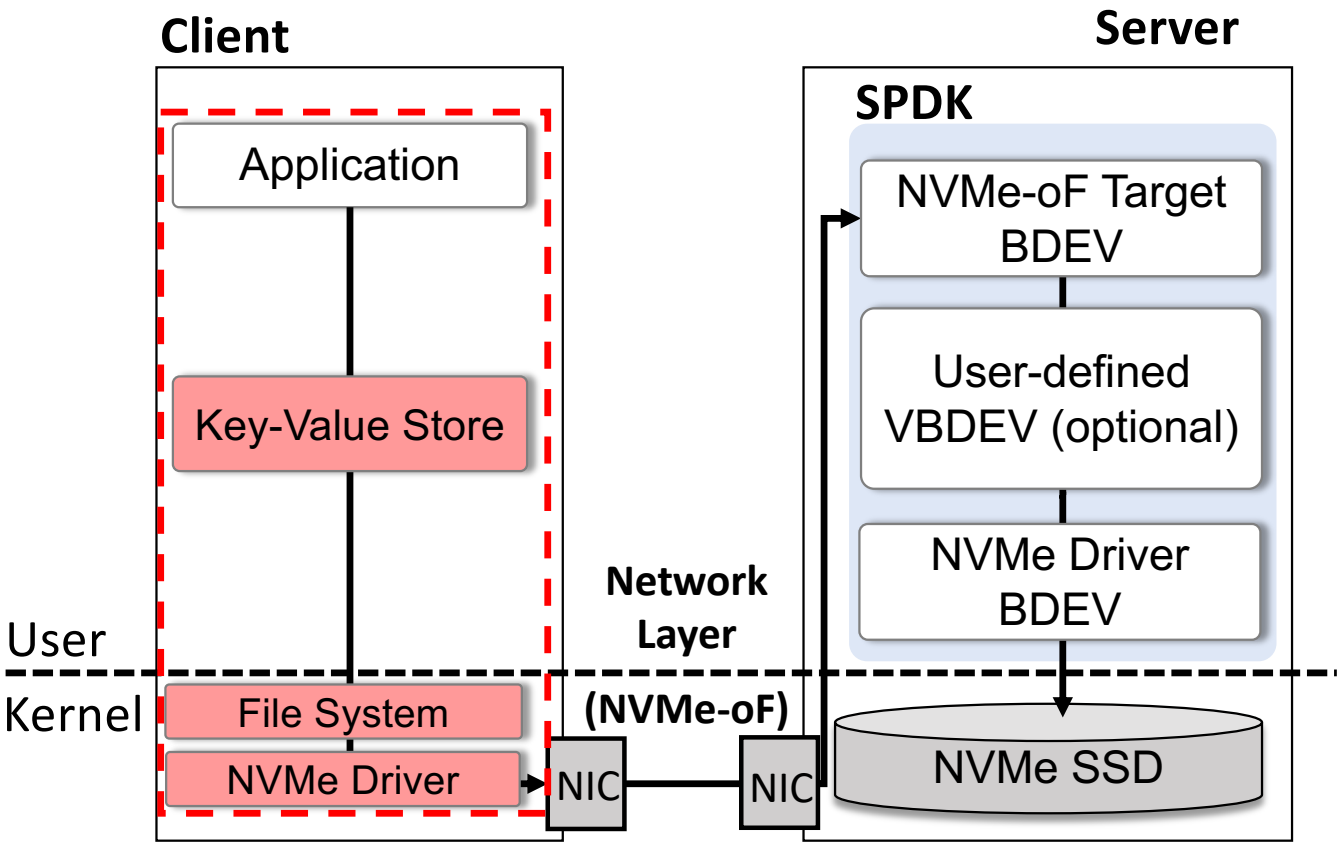
(1) High I/O Stack Overhead Problem

(2) Core Load Imbalance Problem

Problem#1: High I/O Stack Overhead

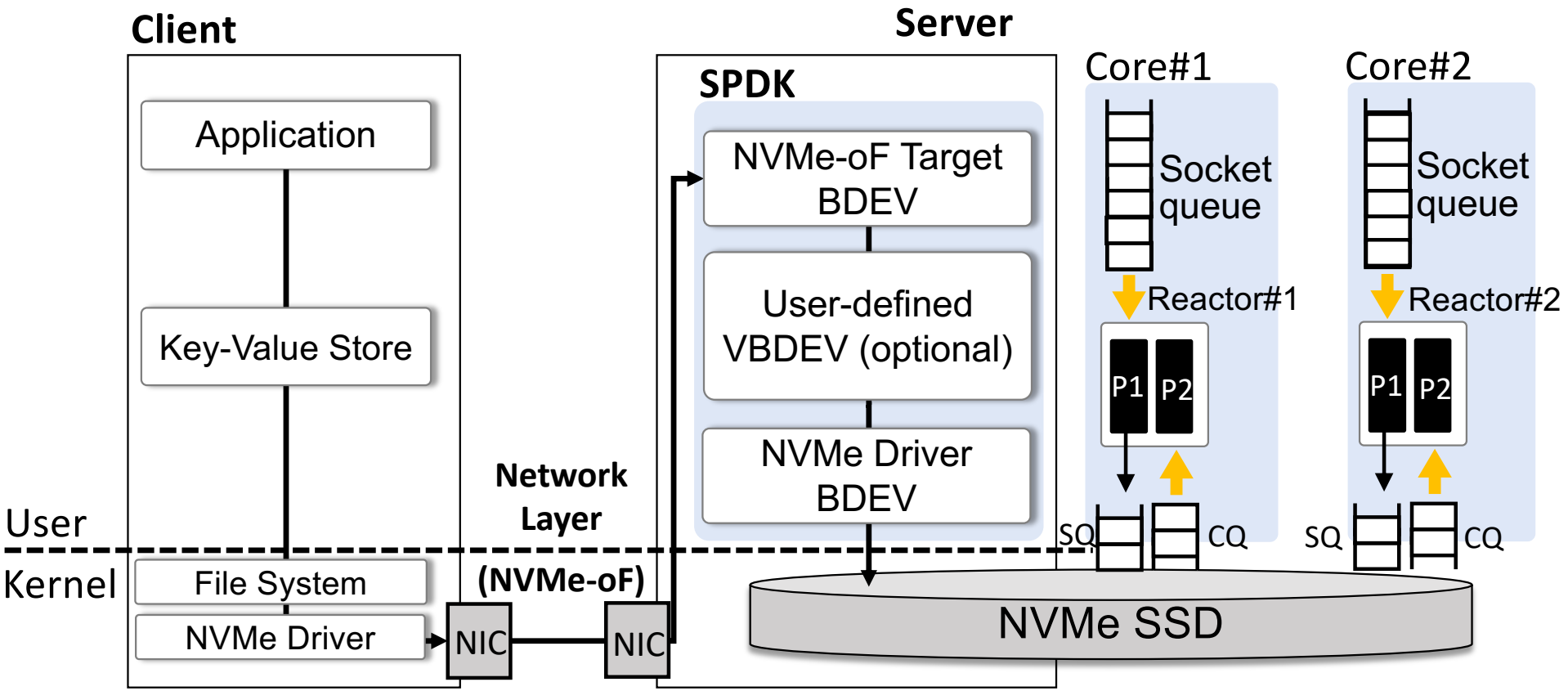


Problem#1: High I/O Stack Overhead

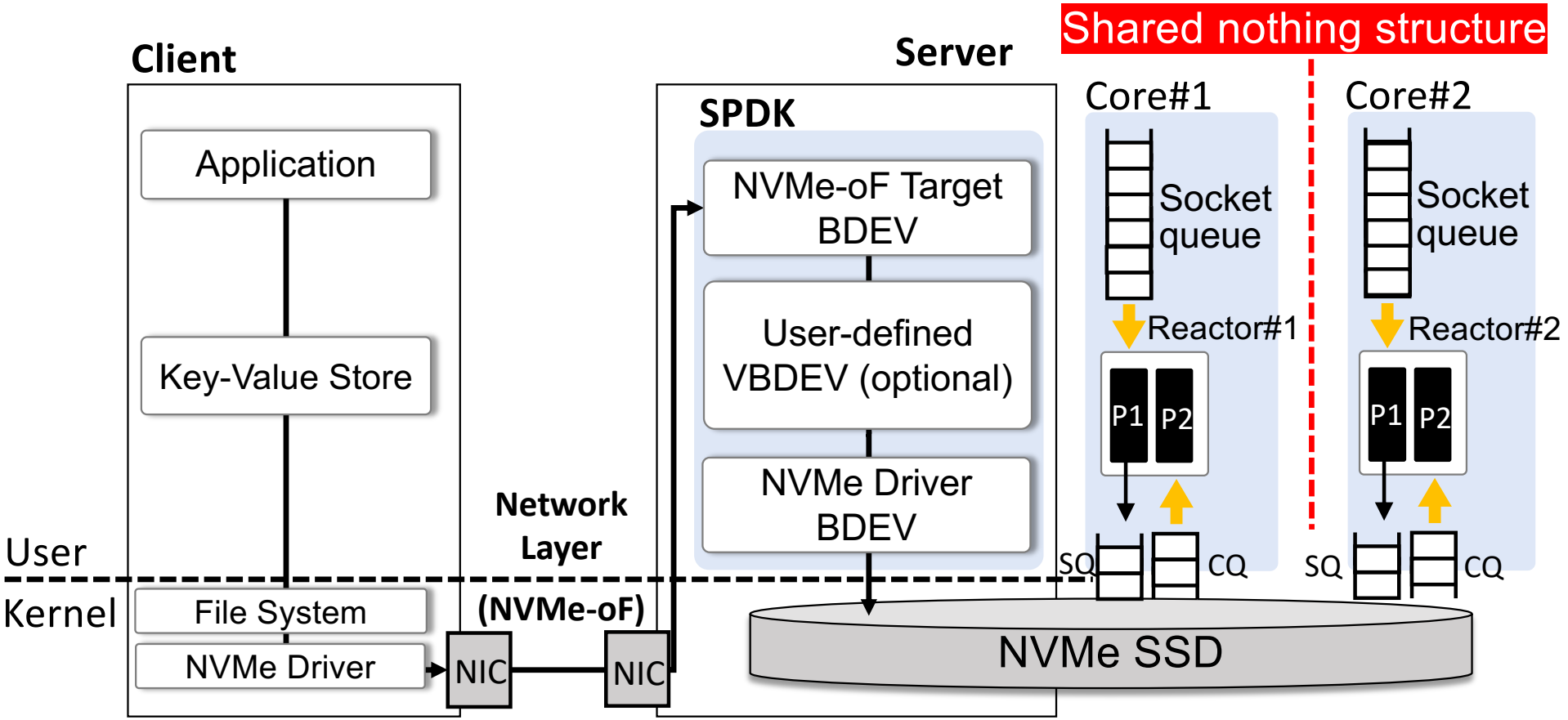


(1) KV to file, file to block address translation overhead
(2) User-to-kernel context switch overhead

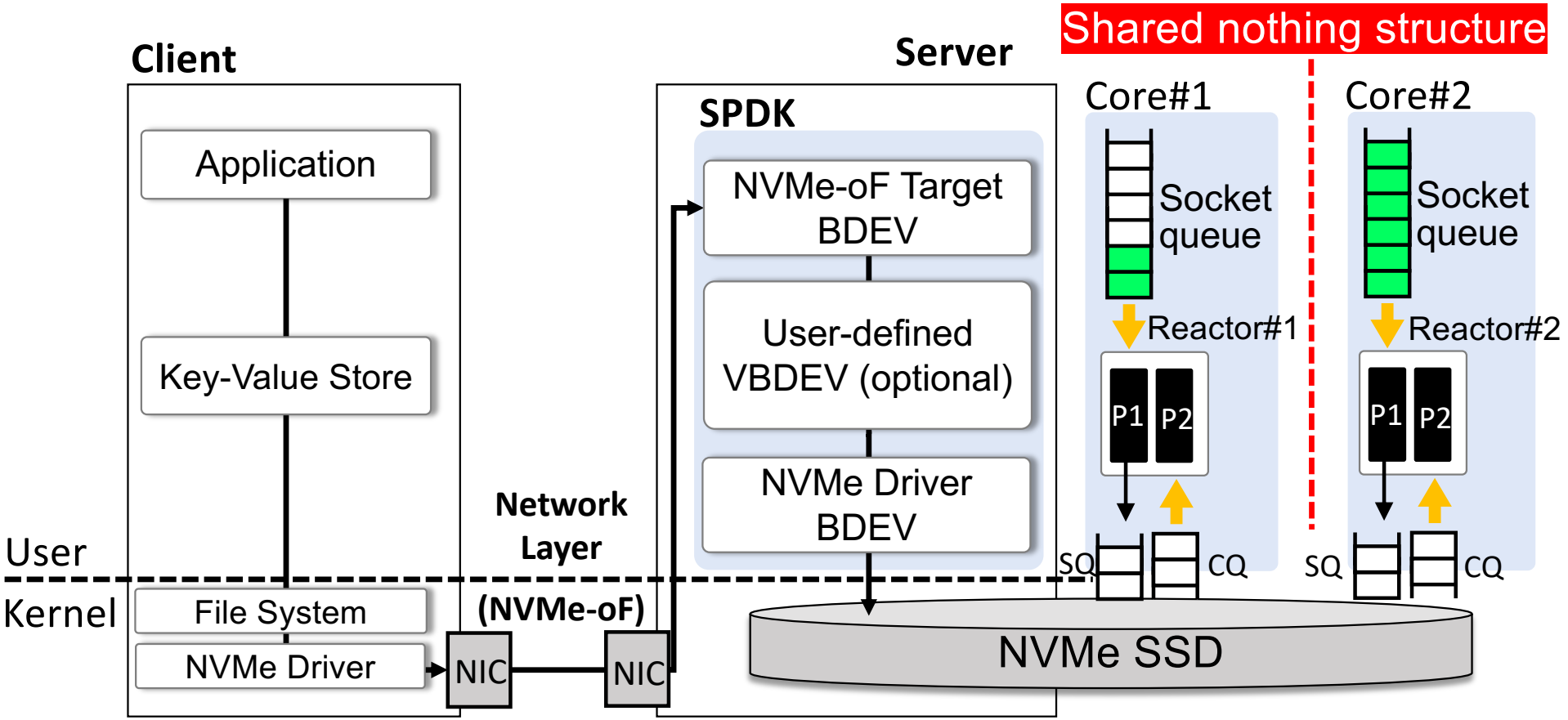
Problem#2: Core Load Imbalance



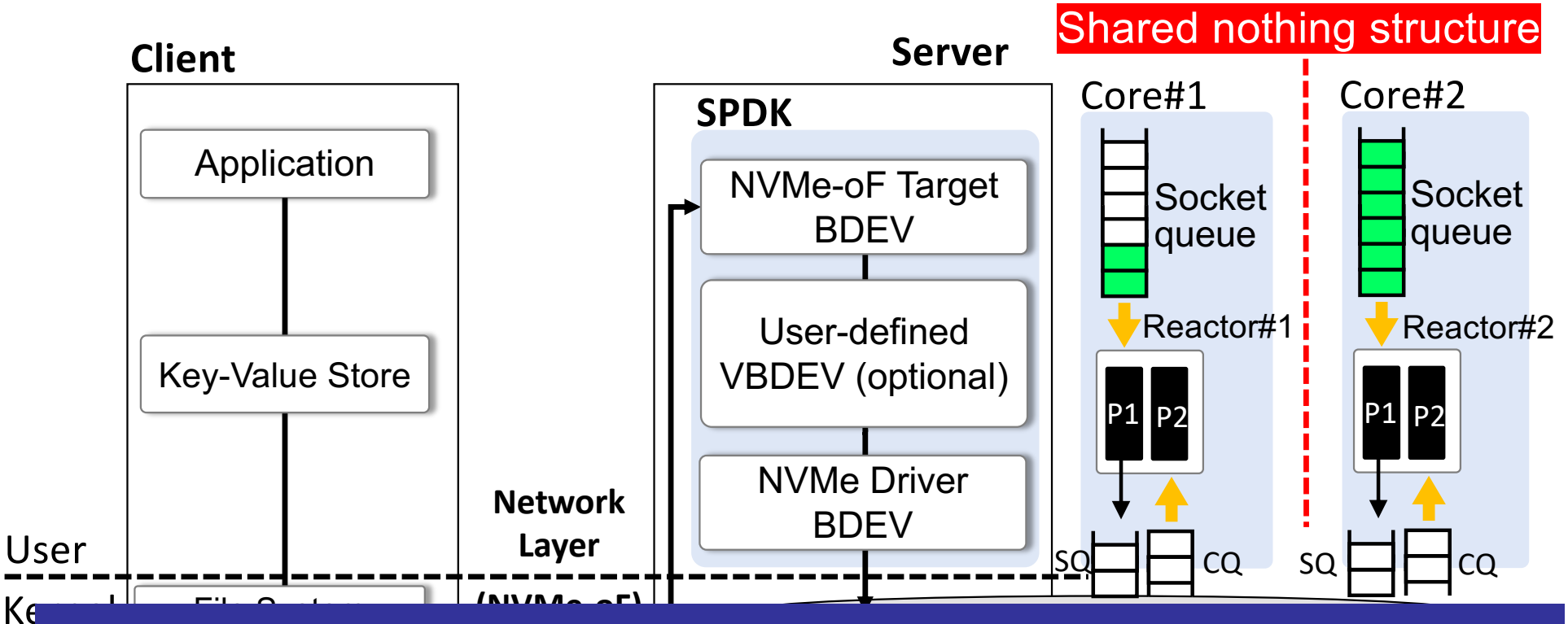
Problem#2: Core Load Imbalance



Problem#2: Core Load Imbalance



Problem#2: Core Load Imbalance



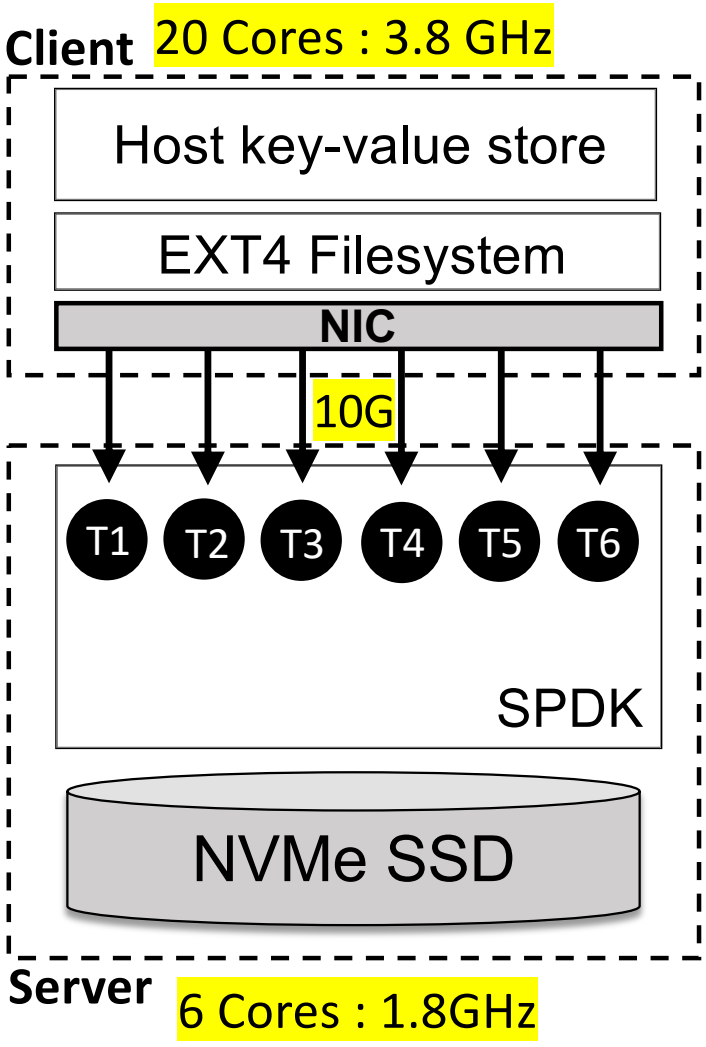
There is a lack of structures for sharing data between cores, resulting in load imbalance.

Content

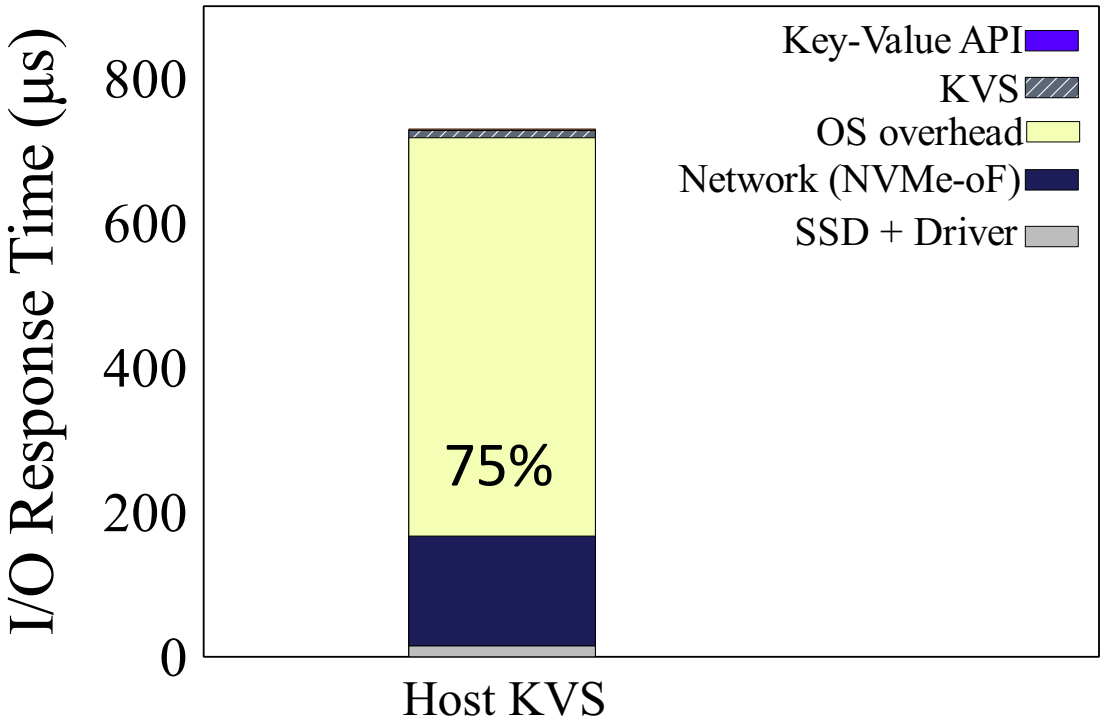
- Background
- Problem Definition
- Motivational Experiments
- OctoKV: Design and Implementation
- Evaluation
- Conclusion

Problem#1: High I/O Stack Overhead

- Client
  - § 20 CPU cores
  - § Running a host hash-based key-value store
- Server
  - § 6 CPU cores
  - § Running a Linux OS using Intel SPDK
- Workloads
  - § Running a db_bench
  - § I/O request size = 16KB



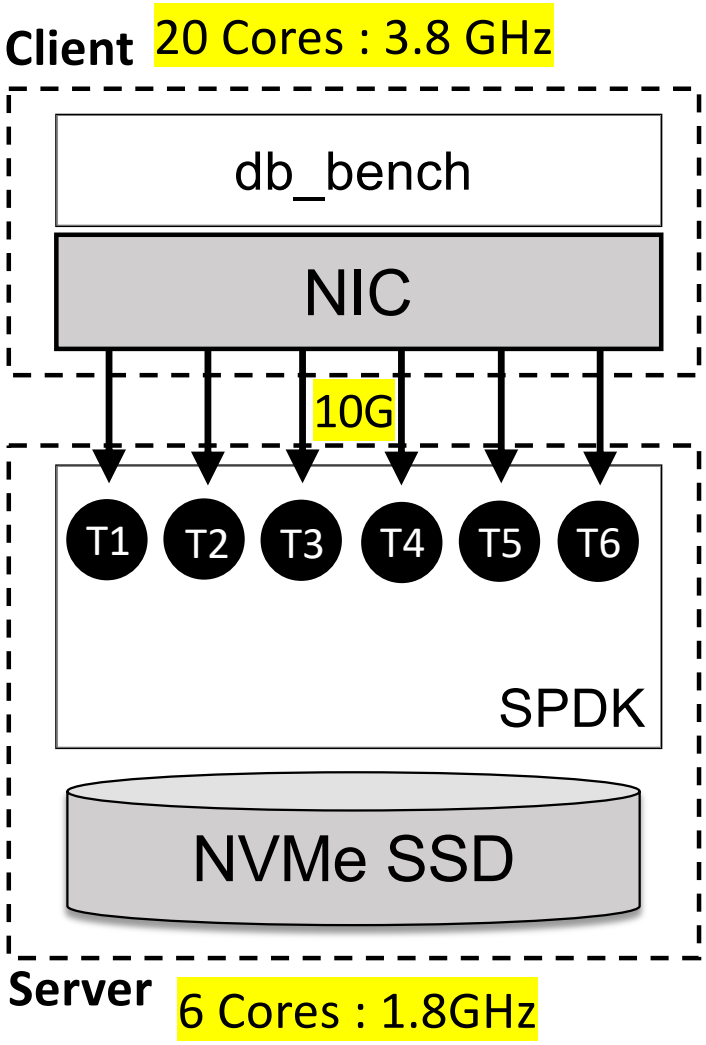
Problem#1: High I/O Stack Overhead



Running key-values store on top of the file system
in a disaggregated architecture has significant I/O overhead

Problem#2: Core Load Imbalance

- Workloads
  - § Running a db_bench
    - 1) Light workload
      - *7 I/O threads issue write I/Os*
    - 2) Heavy workload
      - *12 I/O threads issue write I/Os*
  - § I/O request size = 16KB



Problem#2: Core Load Imbalance

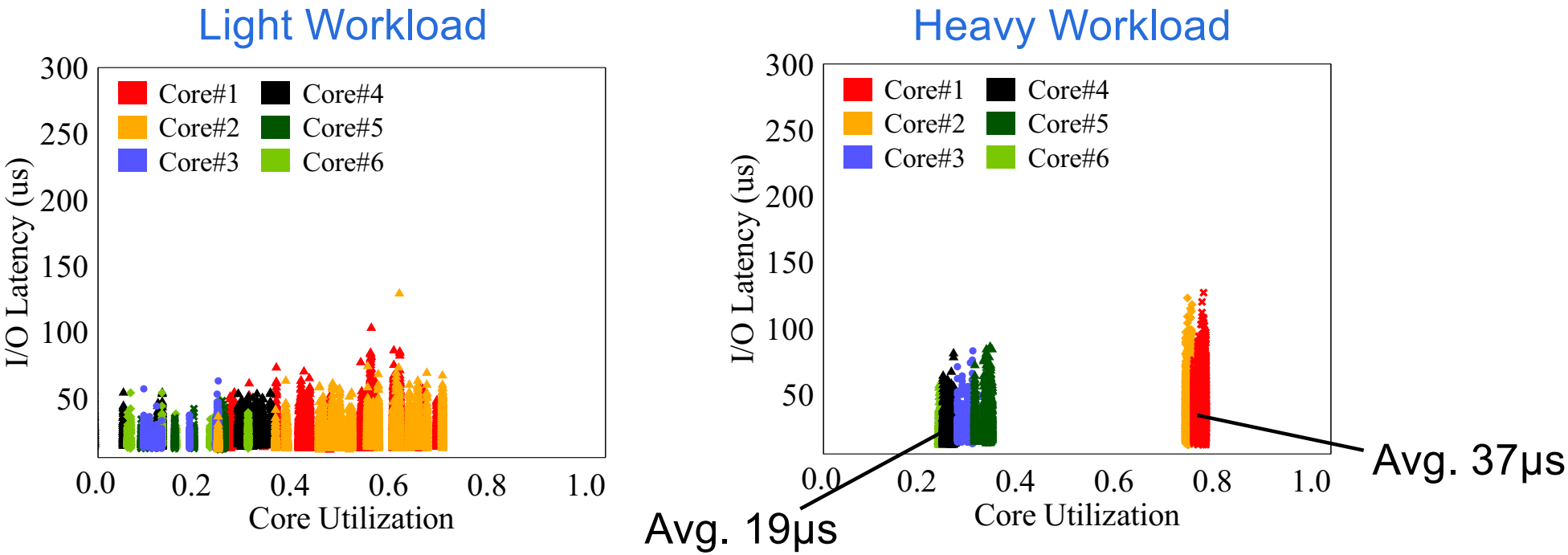
Thread Queue Depth for each core/SPDK thread

Queue Depth	Core#1	Core#2	Core#3	Core#4	Core#5	Core#6	Avg	Stdev
Light Workload (Put)	2.00	2.21	0.75	1.58	0.67	0.33	1.26	0.78
Heavy Workload (Put)	5.25	5.48	2.00	2.06	2.13	2.13	3.18	1.70
Light Workload (Get)	3.95	4.23	1.27	1.36	2.00	1.82	2.43	1.31
Heavy Workload (Get)	6.06	6.54	2.69	2.62	2.92	2.65	3.91	1.86

The amount of NVMe commands delivered to the core is imbalanced

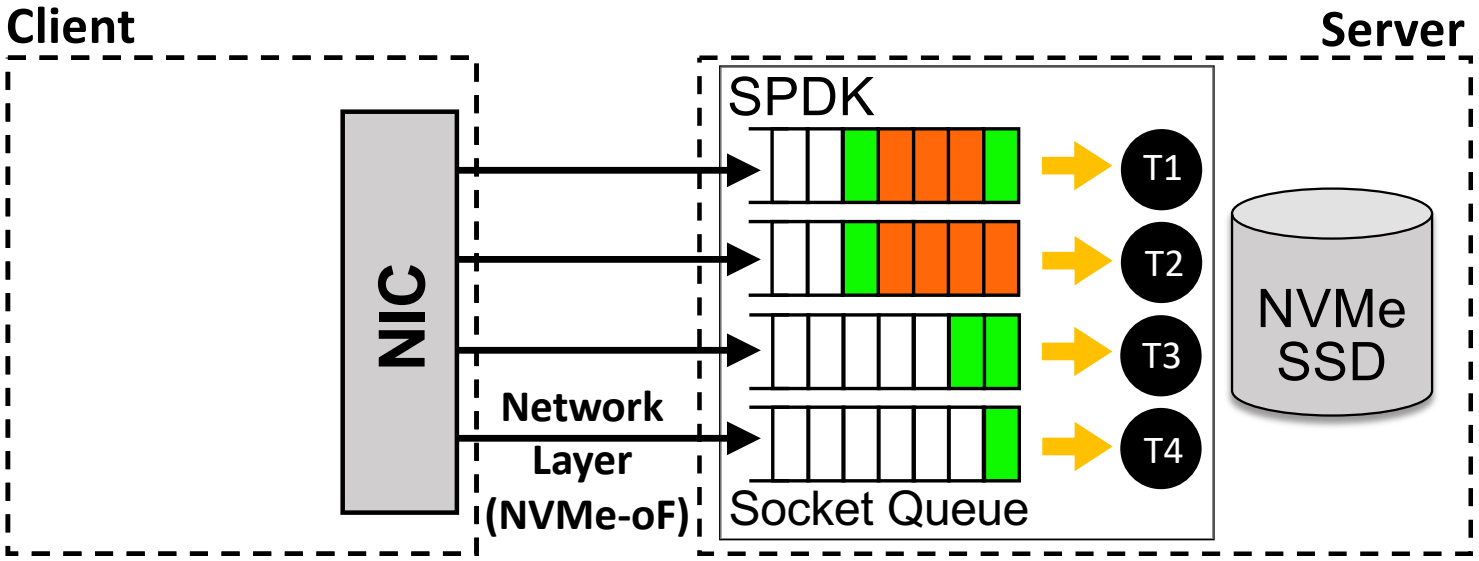
Problem#2: Core Load Imbalance

Core Utilization vs I/O Latency



Core utilization between cores forms a bimodal distribution,
I/Os processed on busy cores show high latency

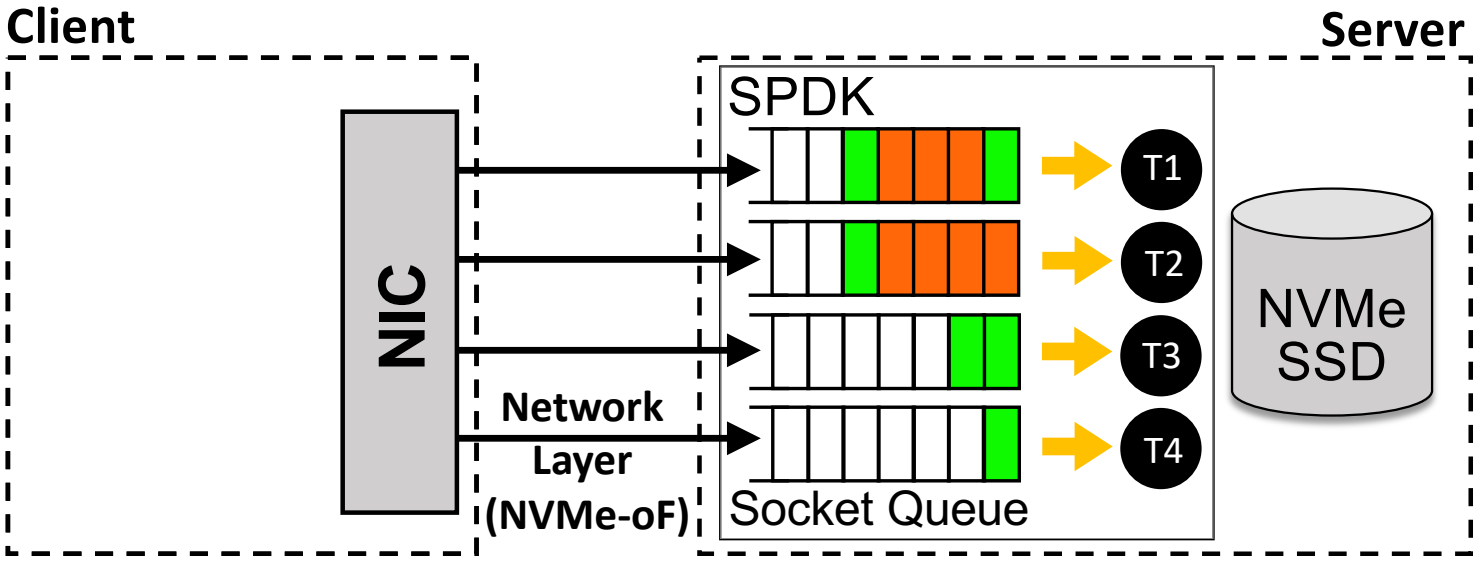
Problem#2: Core Load Imbalance



→ hash function, compaction, background task …

Ti SPDK thread → TCP connection NVMe request (Light) NVMe request (Heavy)

Problem#2: Core Load Imbalance



→ hash function, compaction, background task …

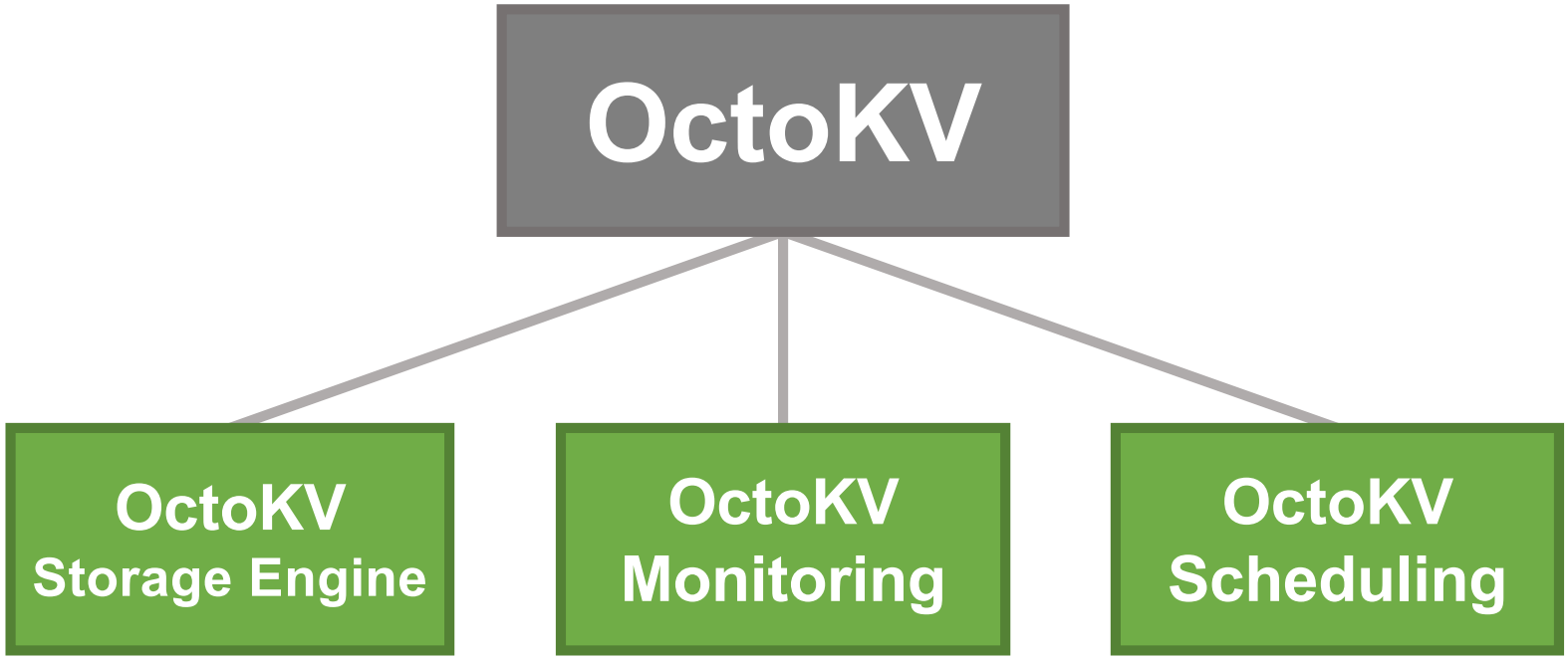
Performing compute-intensive tasks on the server increases the CPU load and increases the load imbalance problem

OctoKV: *An Agile Network-Based Key-Value Storage System with Robust Load Orchestration*

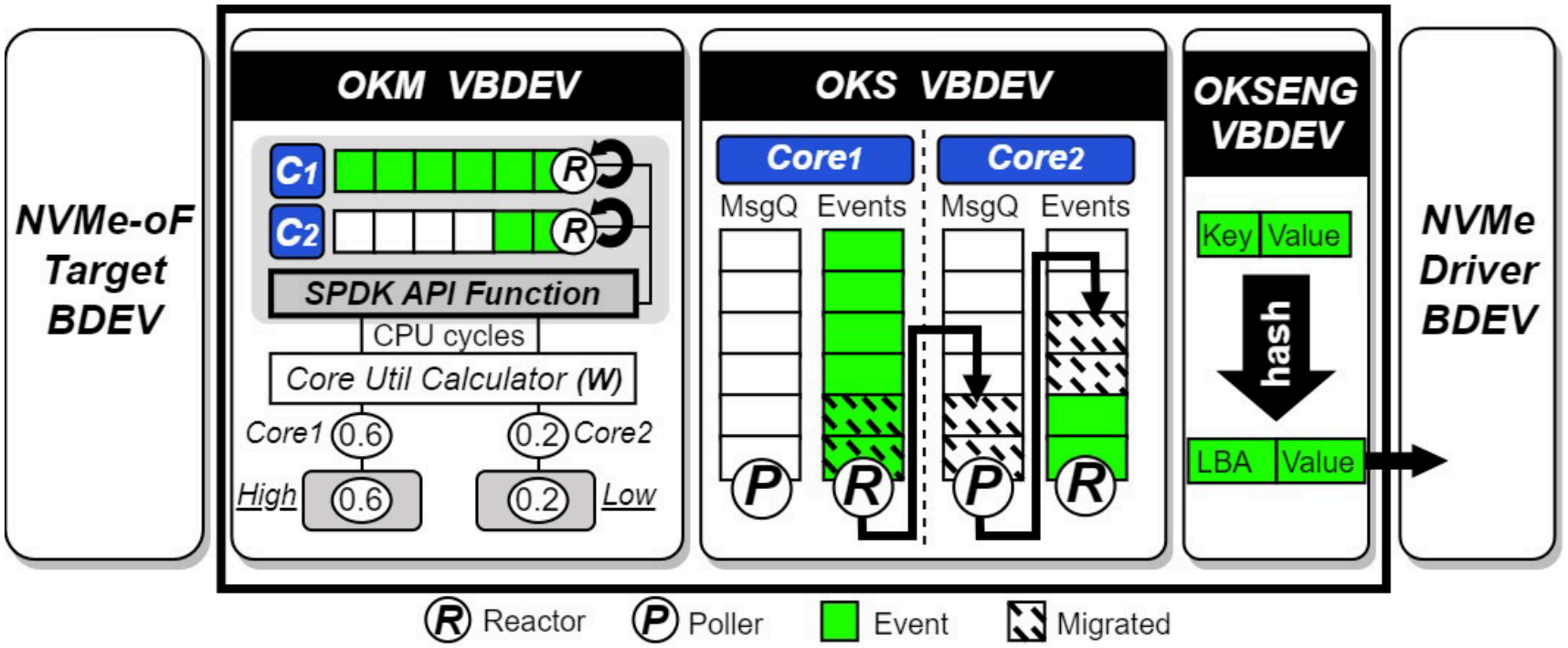
Content

- Background
- Problem Definition
- Motivational Experiments
- OctoKV: Design and Implementation
- Evaluation
- Conclusion

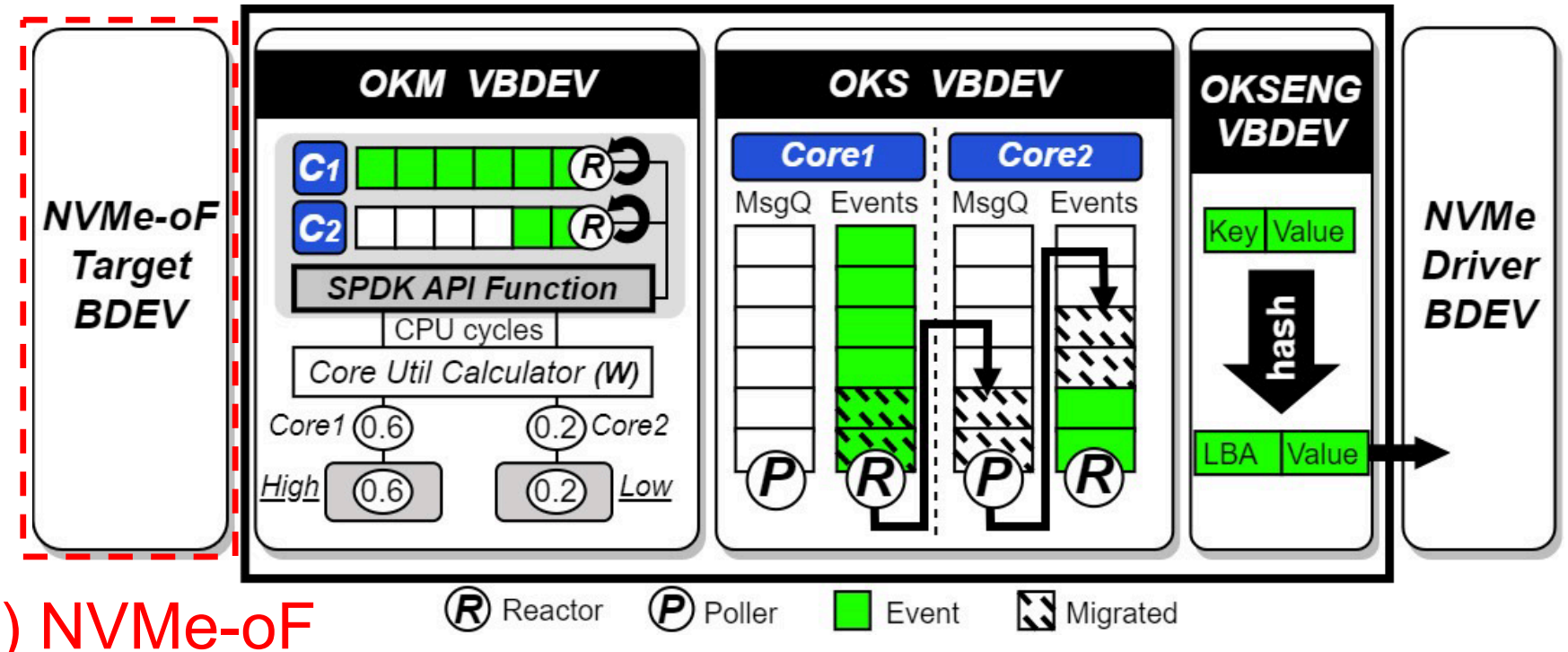
OctoKV Overview



OctoKV Overview

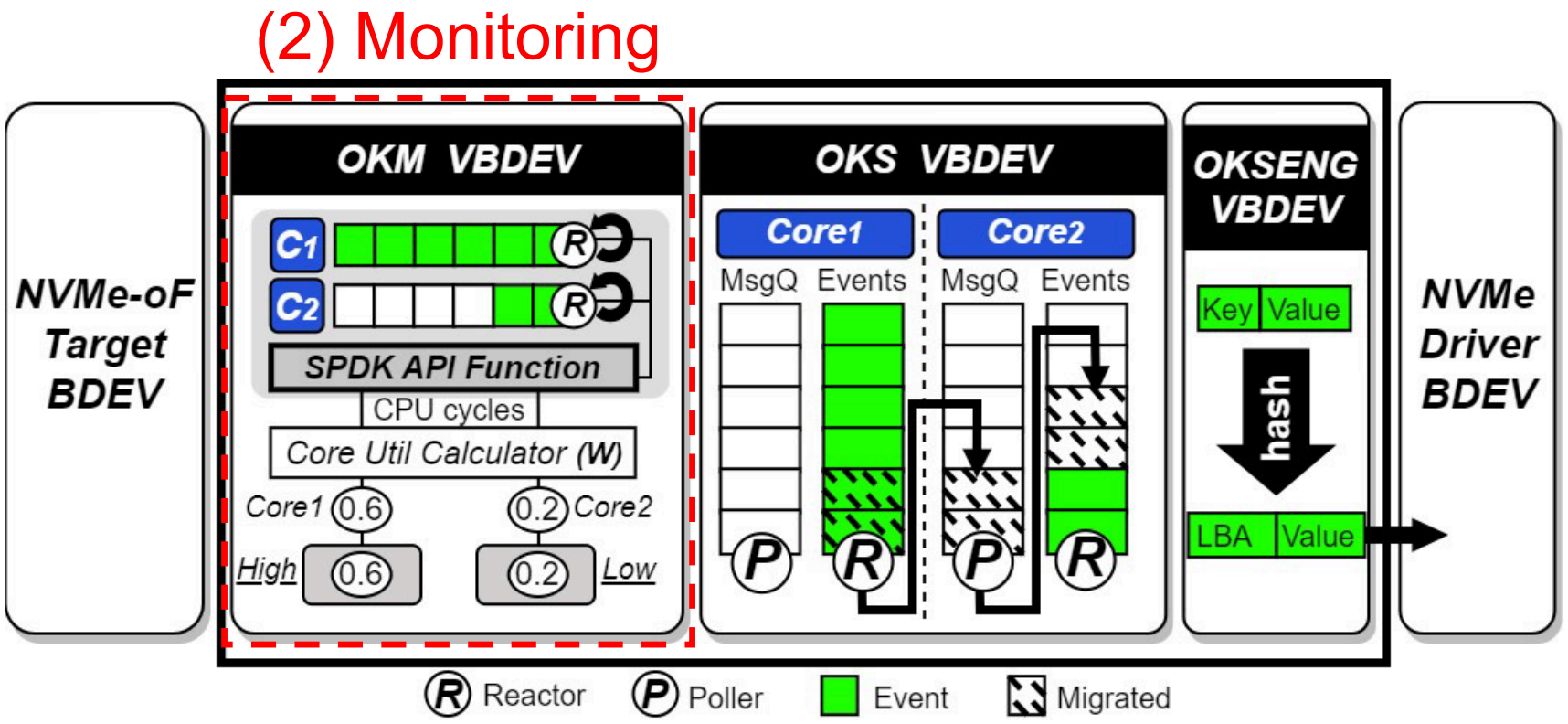


OctoKV Overview

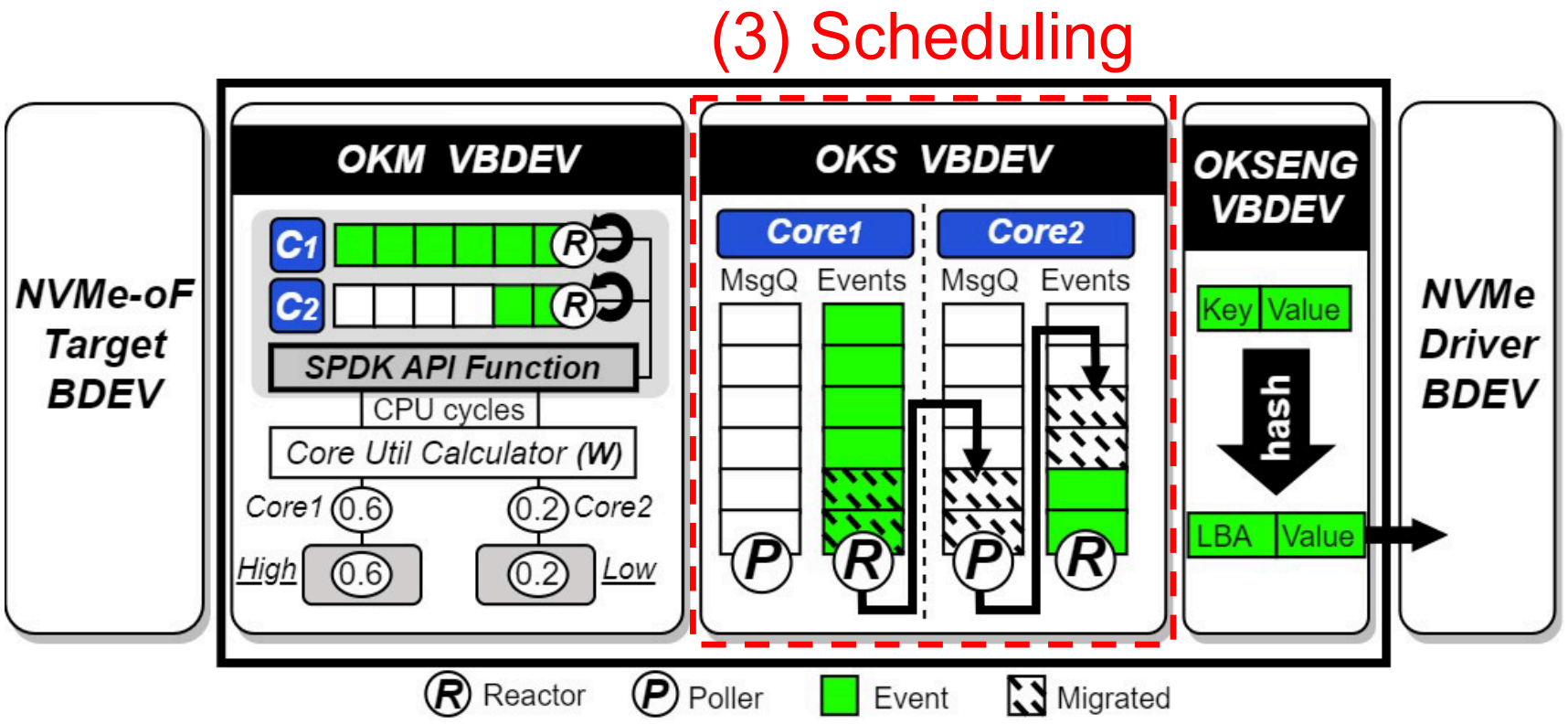


(1) NVMe-oF

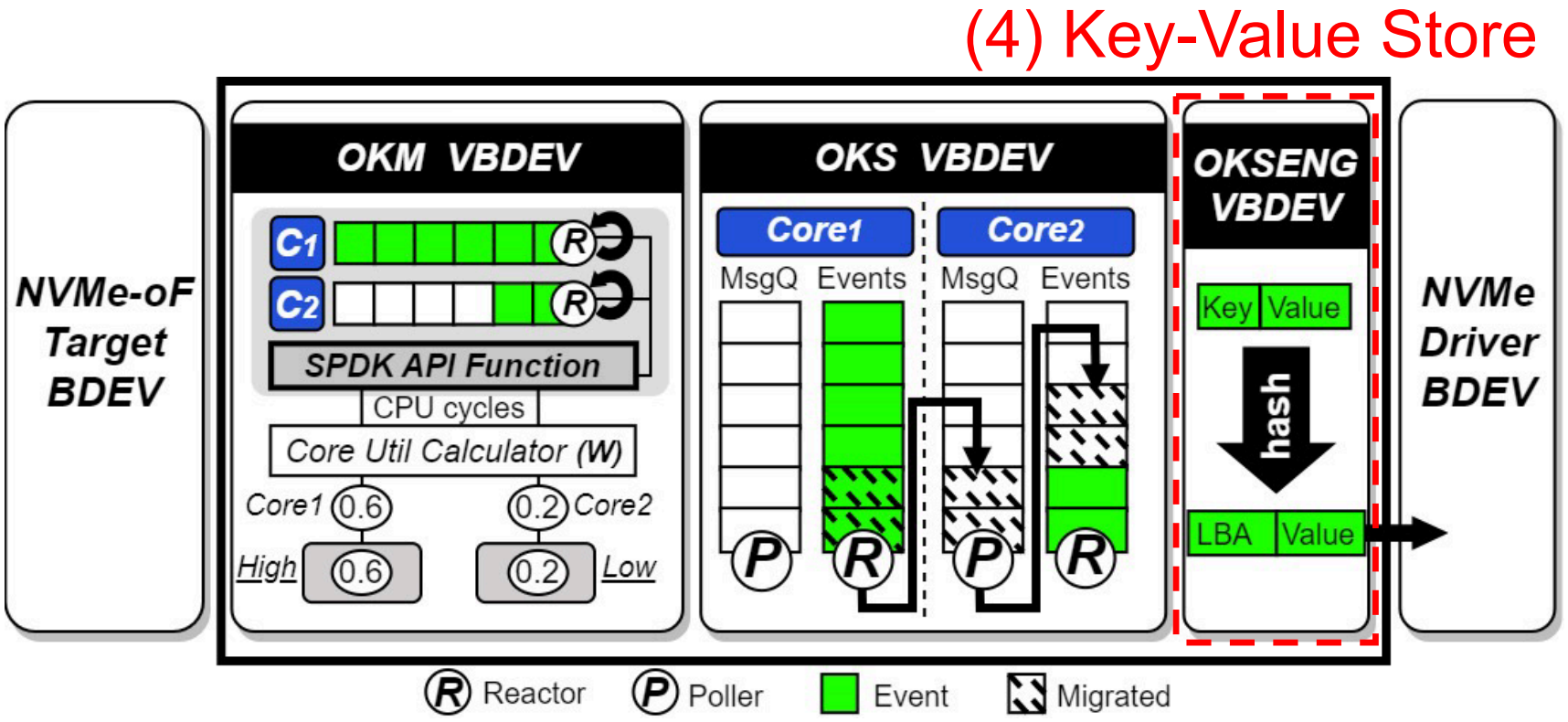
OctoKV Overview



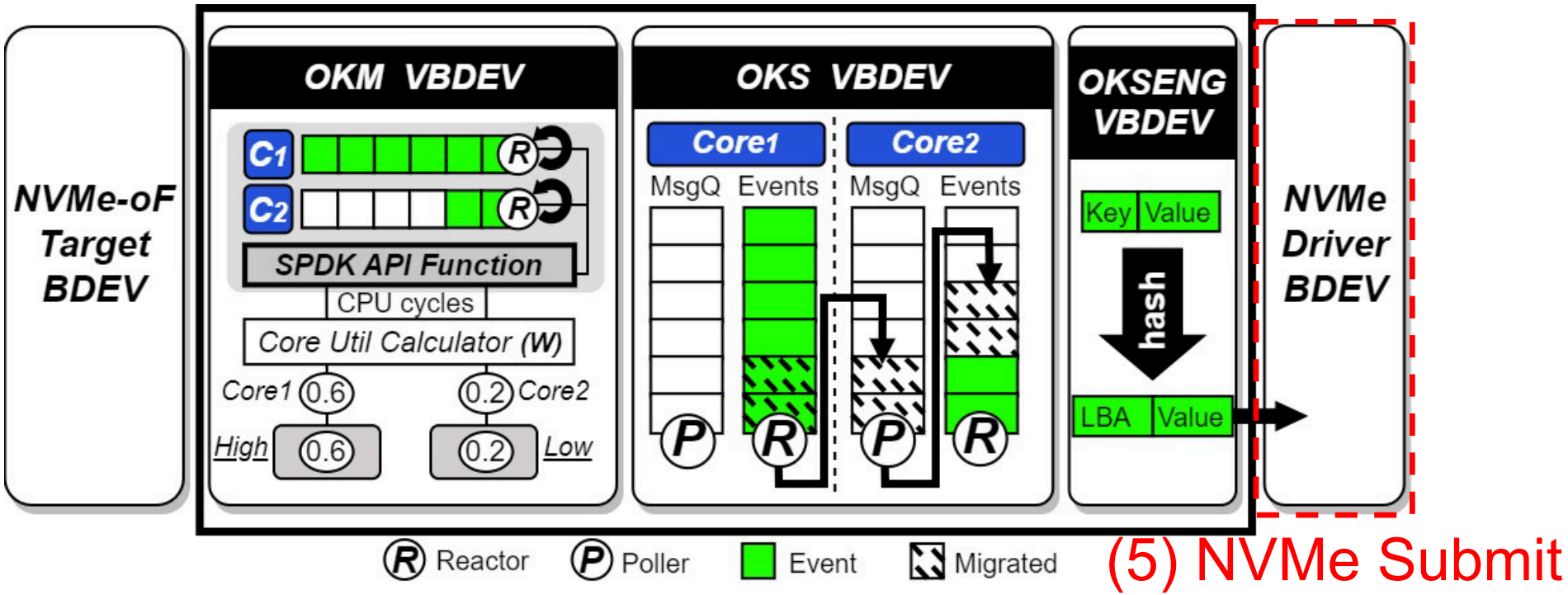
OctoKV Overview



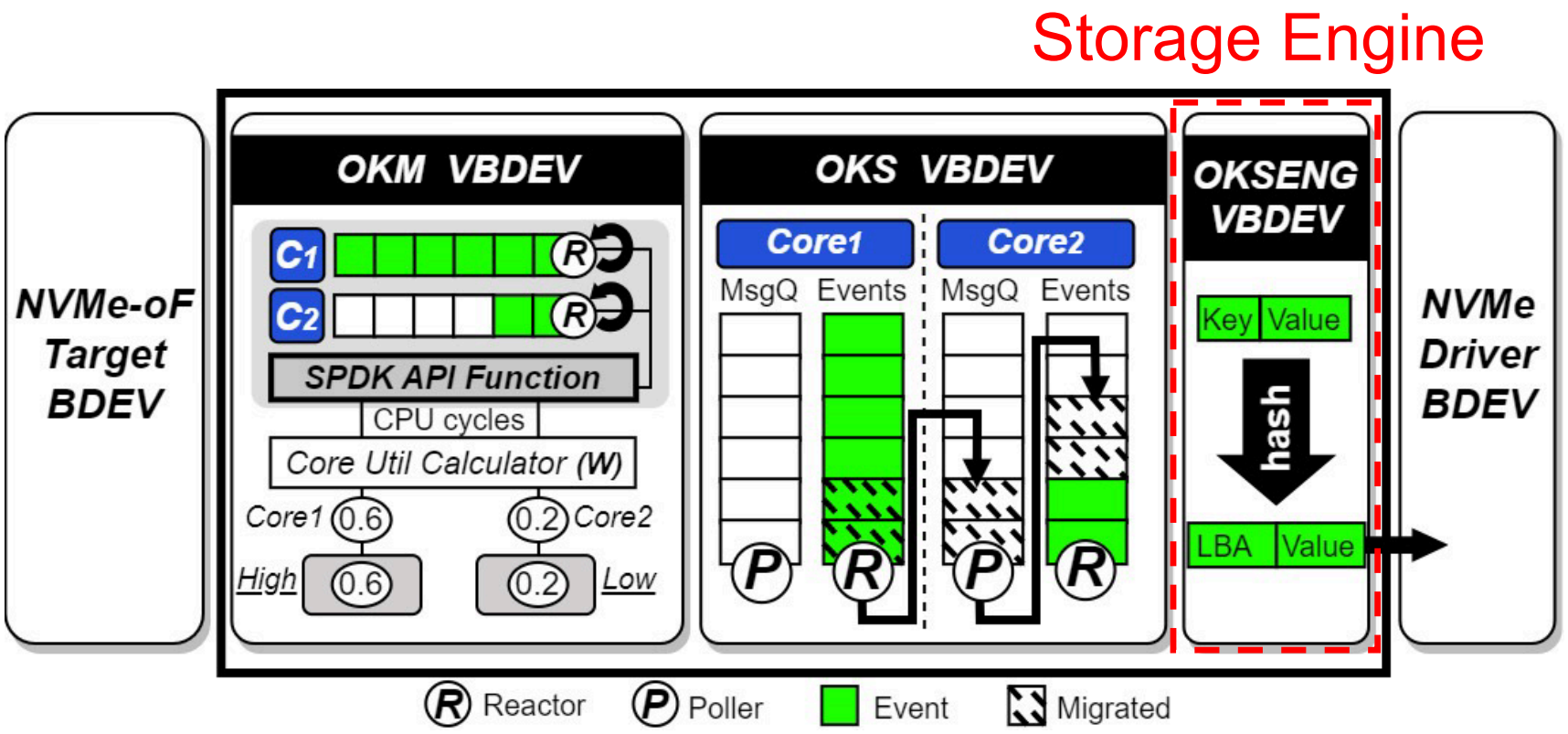
OctoKV Overview



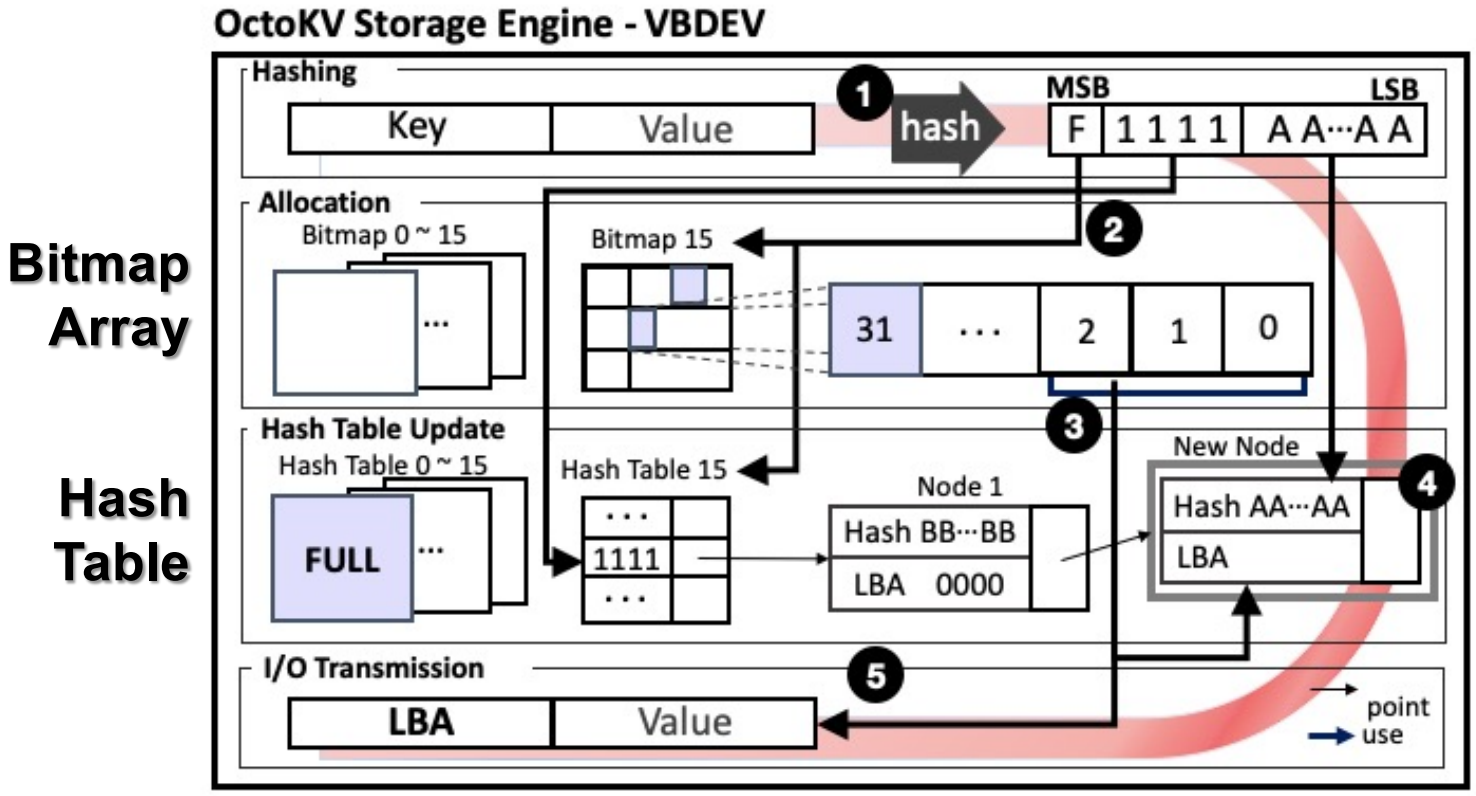
OctoKV Overview



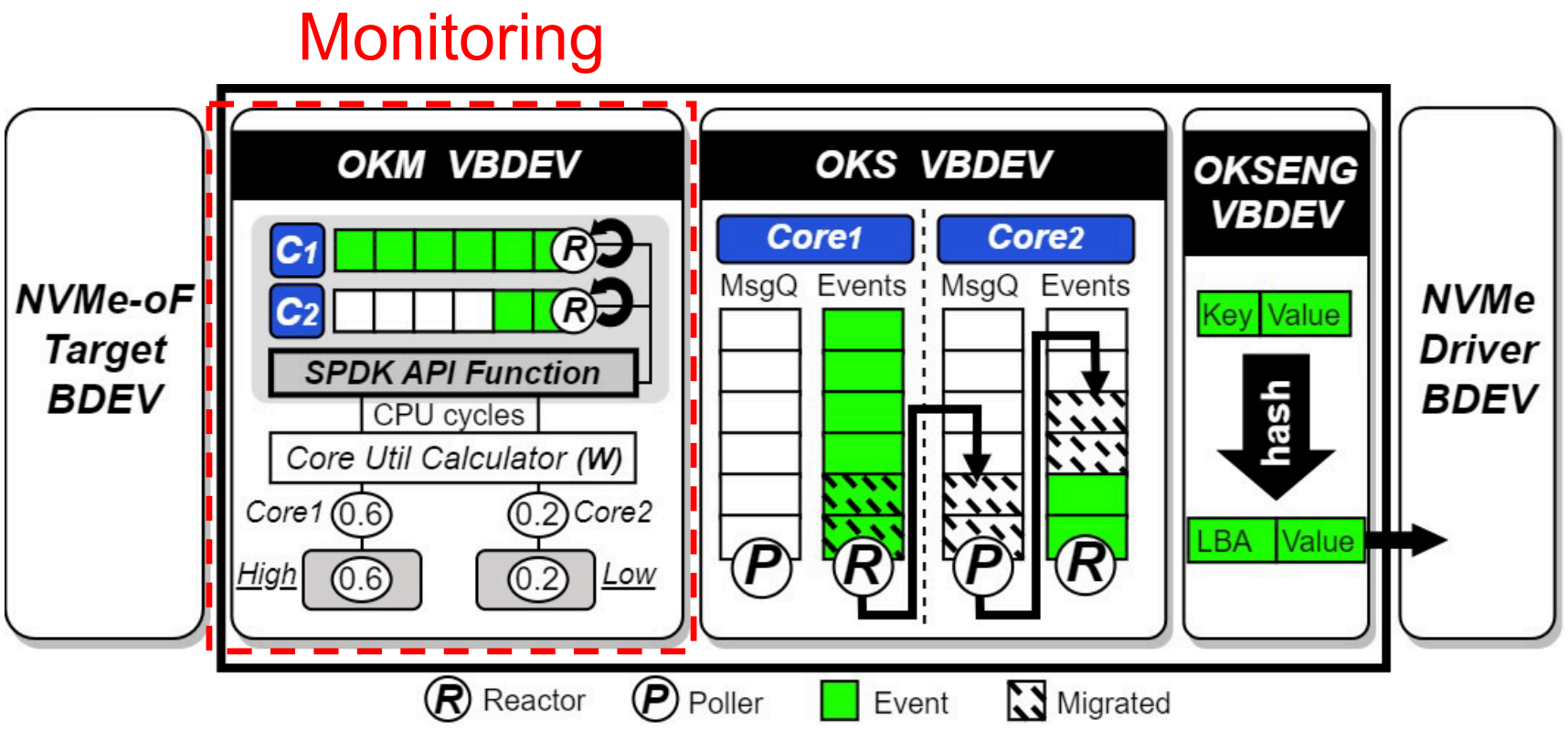
OctoKV: Design and Implementation



Module#1: Storage Engine

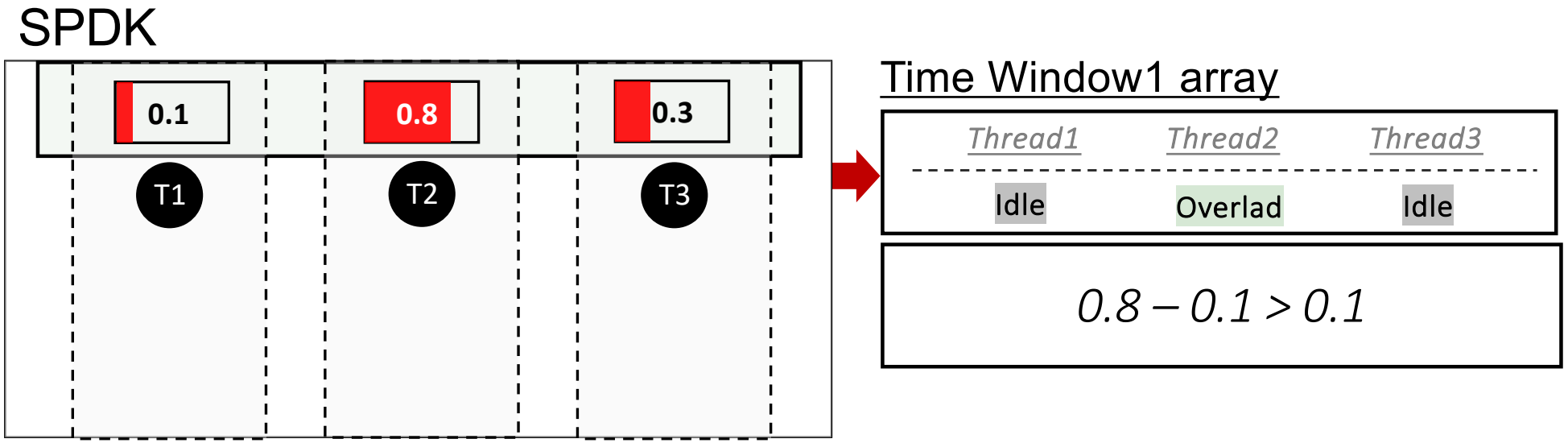


Module#2: Monitoring



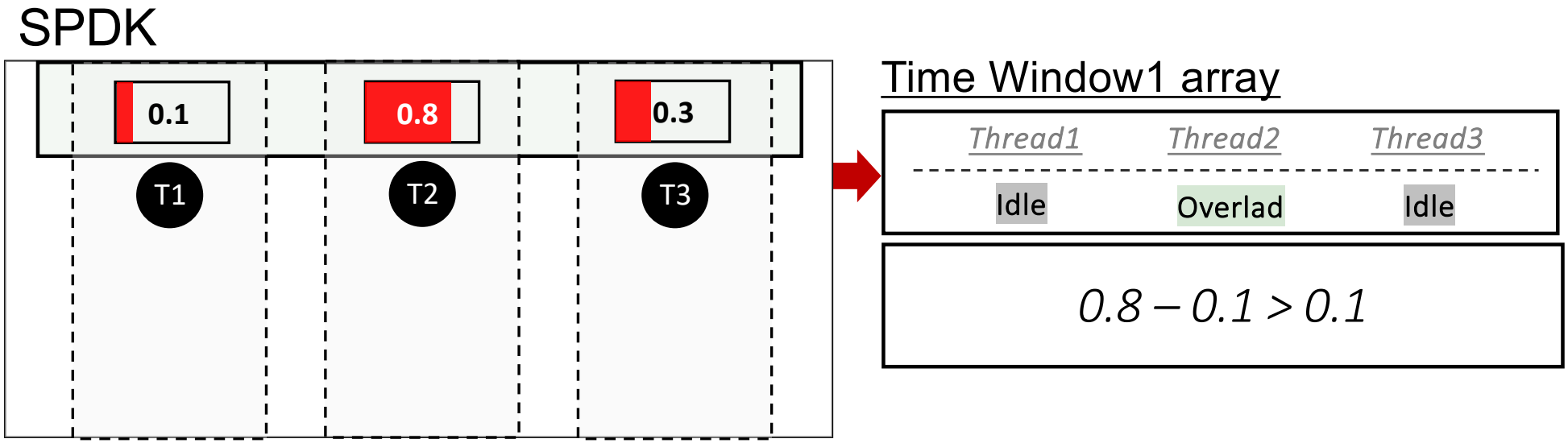
Module#2: Monitoring

- Monitors the utilization of each core



Module#2: Monitoring

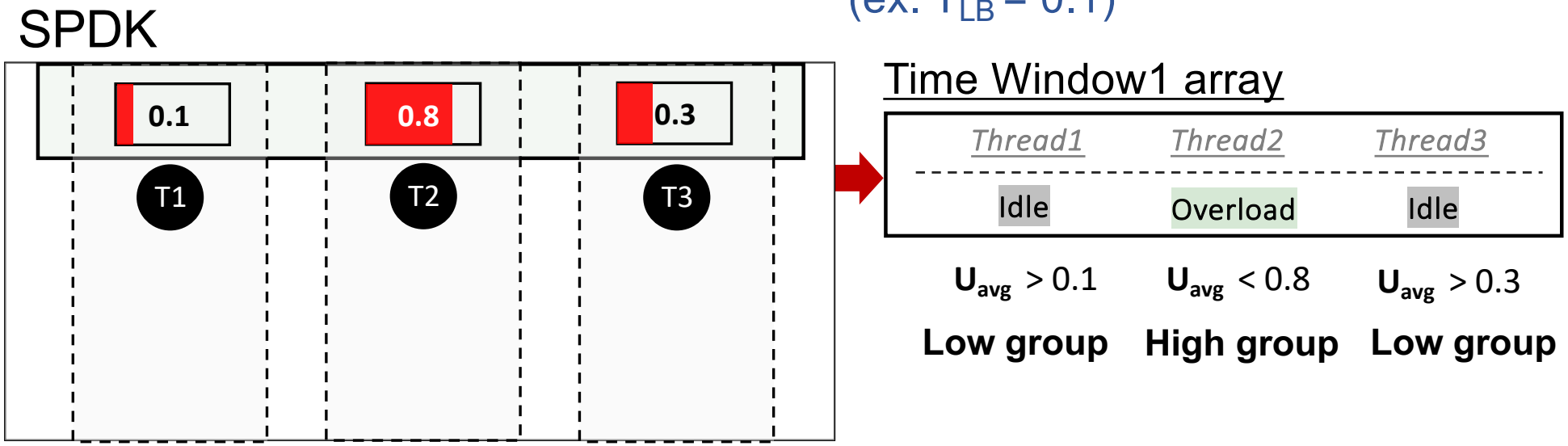
- Monitors the utilization of each core
  Condition#1: Core Overloading $F_{cutil}(C) > T_{OL}(T_{OL} = 0.4)$



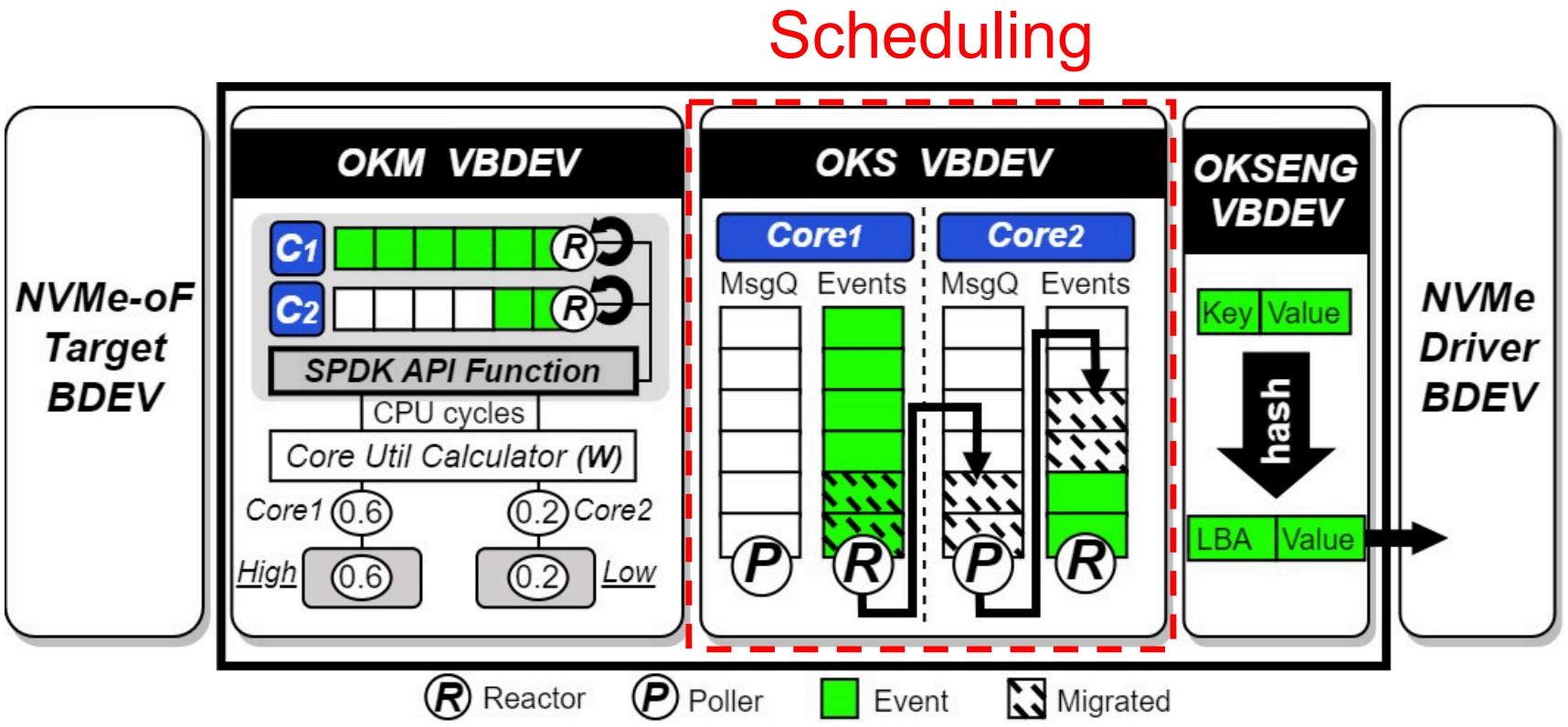
Module#2: Monitoring

- Monitors the utilization of each core
  Condition#1: Core Overloading
  Condition#2: Load Imbalance

$Max\{F_{cutil}(C)\} - Min\{F_{cutil}(C)\} > T_{LB}$
(ex. $T_{LB} = 0.1$)



Module#3: Scheduling



Module#3: Scheduling

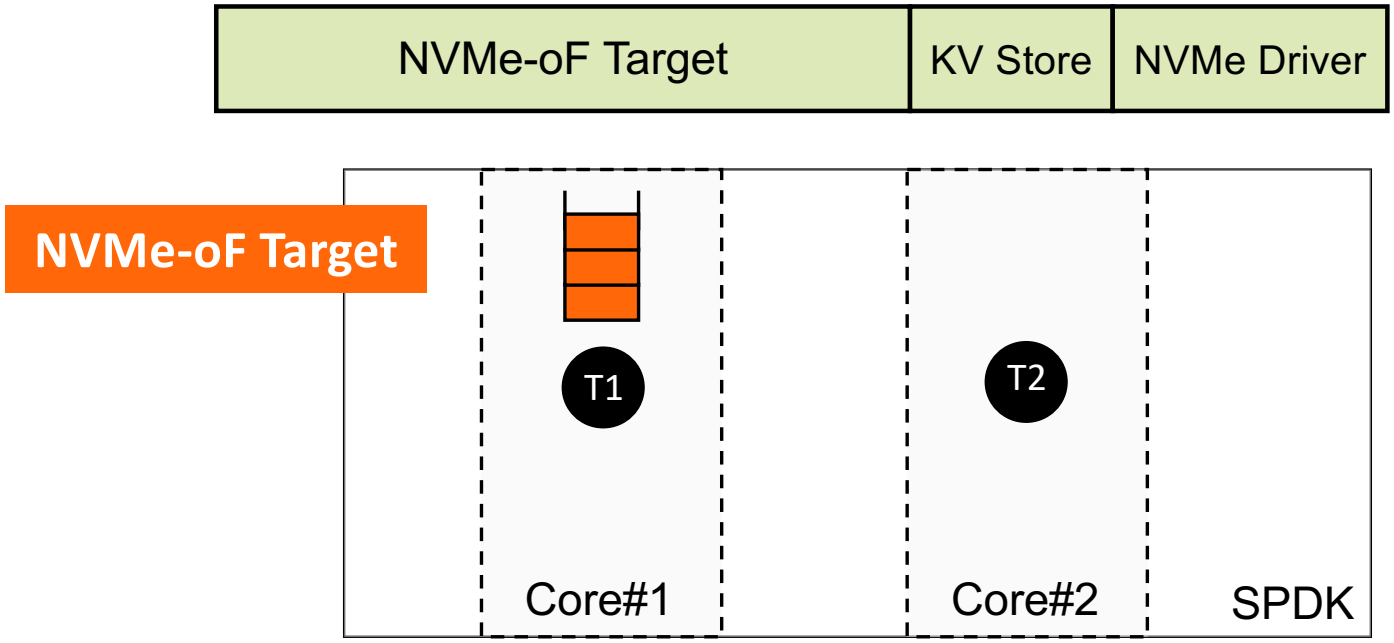
- OctoKV Scheduling Module migrates I/O requests from overloaded cores to idle cores
- A single I/O request consists of three stages

Module#3: Scheduling

- OctoKV Scheduling Module migrates I/O requests from overloaded cores to idle cores
- A single I/O request consists of three stages

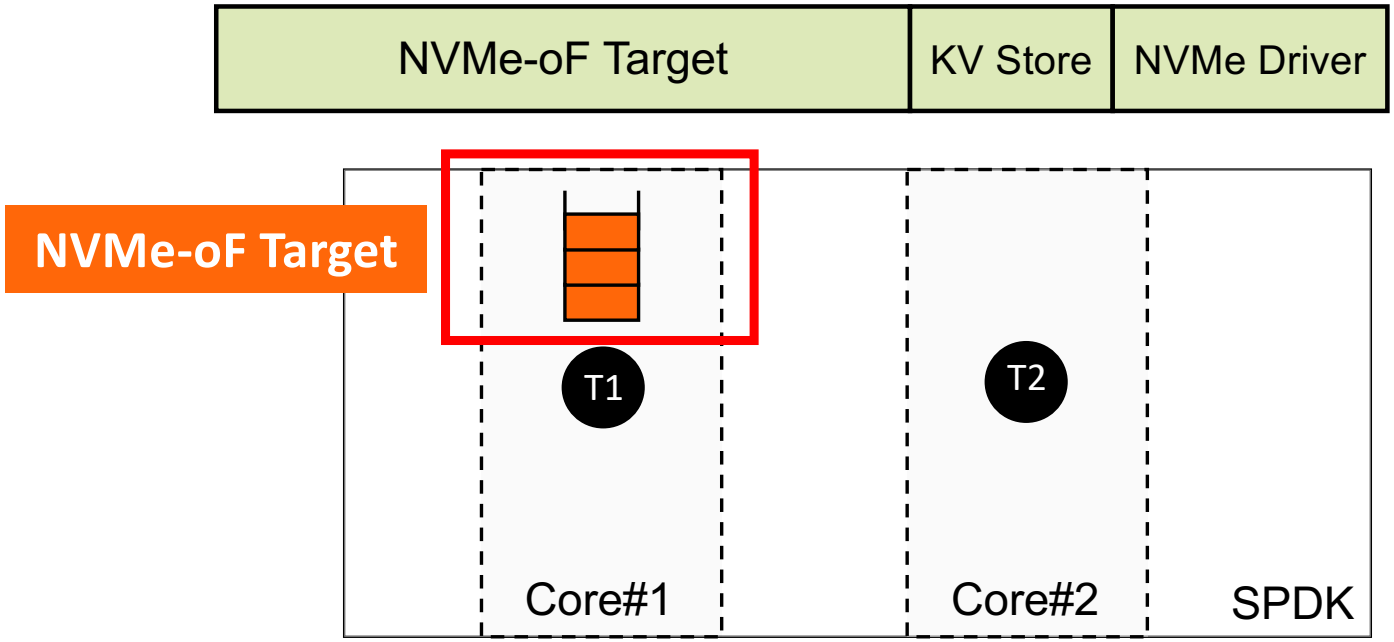
Module#3: Scheduling

- OctoKV Scheduling Module migrates I/O requests from overloaded cores to idle cores
- A single I/O request consists of three stages



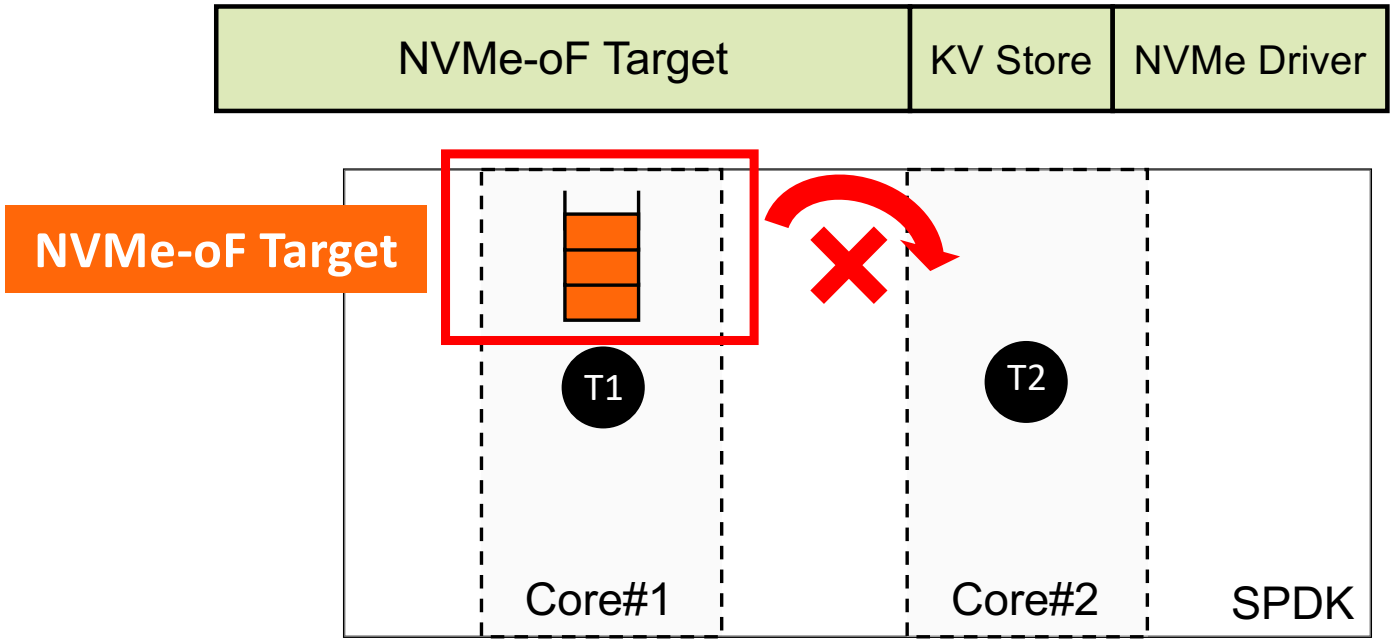
Module#3: Scheduling

- OctoKV Scheduling Module migrates I/O requests from overloaded cores to idle cores
- A single I/O request consists of three stages



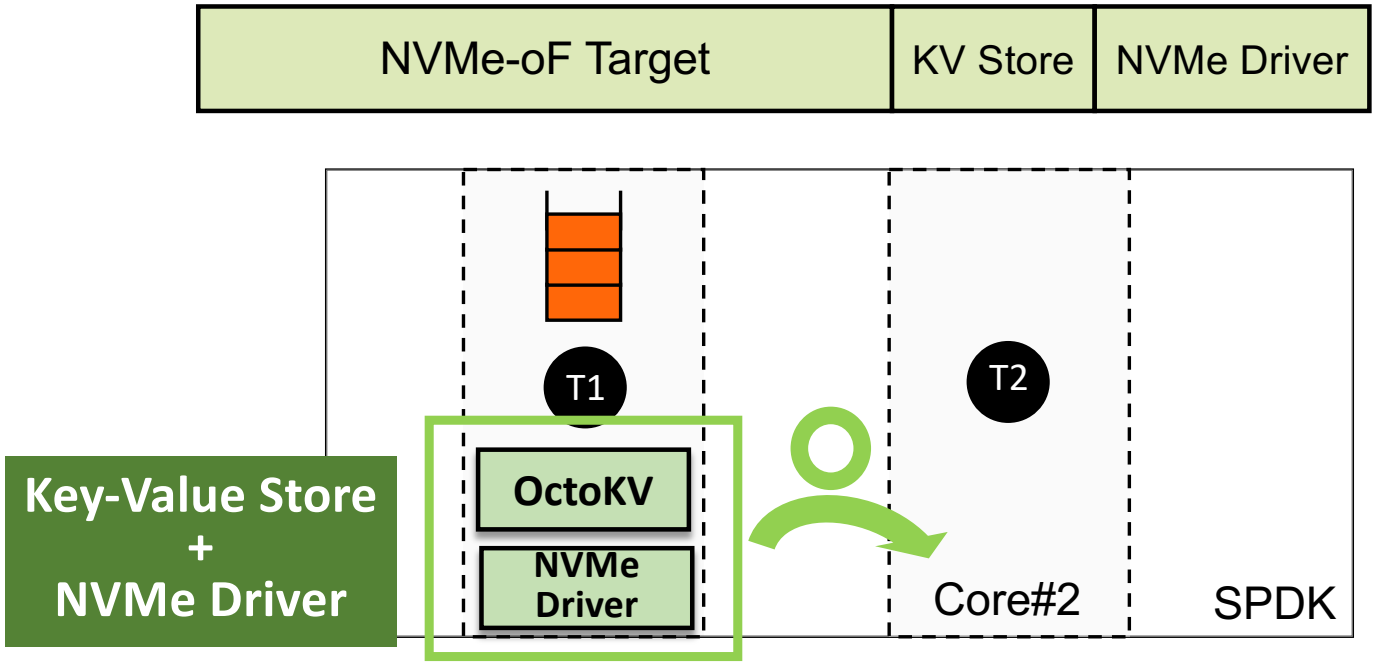
Module#3: Scheduling

- OctoKV Scheduling Module migrates I/O requests from overloaded cores to idle cores
- A single I/O request consists of three stages

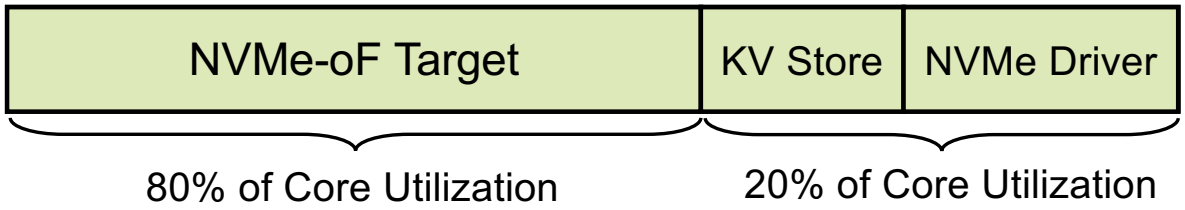


Module#3: Scheduling

- OctoKV Scheduling Module migrates I/O requests from overloaded cores to idle cores
- A single I/O request consists of three stages



Module#3: Scheduling



High Group

Thread2 : 0.8

➡ Movable core utilization
20% of 0.8 => 0.16

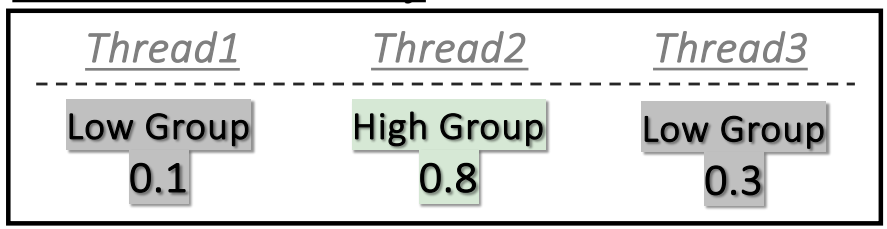
Low Group

Thread1 : 0.1

Thread3 : 0.3

➡ Acceptable core utilization
(U_{avg} – 0.1) + (U_{avg} – 0.3) => 0.4

Time Window1 array



All KV stores and NVMe Driver stages in the high group core can be moved to the low group core for processing.

Module#3: Scheduling

- Two heuristic algorithms determine
  how much I/O to migrate to each core of low group

(Ex) *Thread1: 0.1* *Thread3: 0.3* *U_{avg} : 0.4*

RoundRobin (RR)

Thread1 : Thread3 = 1 : 1

Proportional Share (PS)

Thread1 : U_{avg} - 0.1 = 0.3
Thread1 : U_{avg} - 0.3 = 0.1

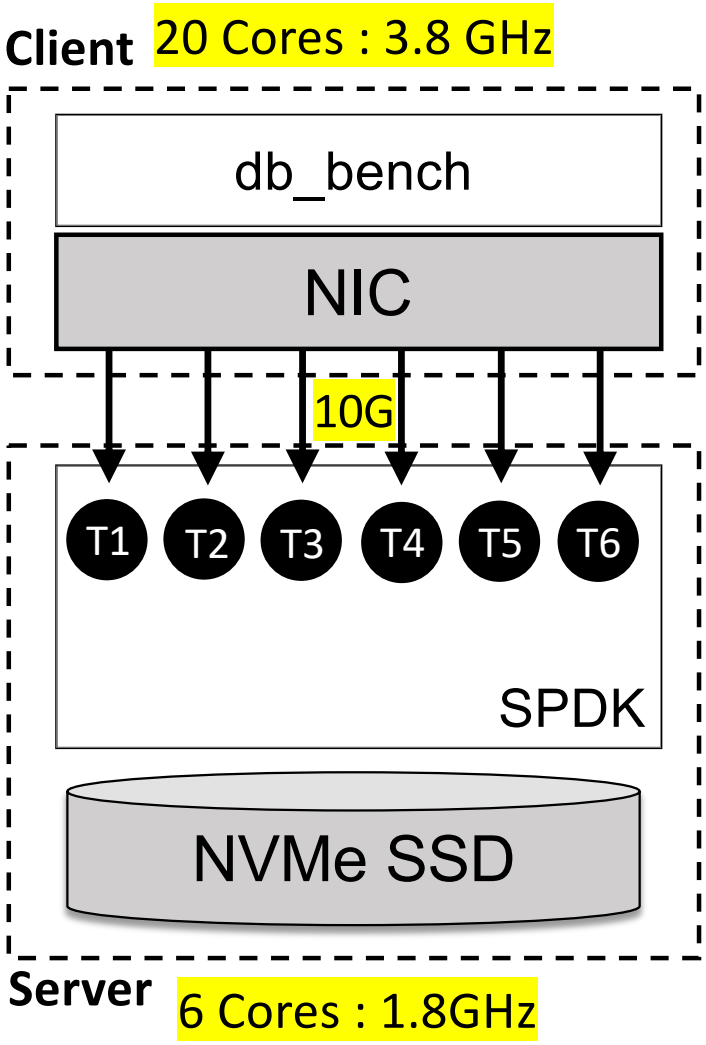
Thread1 : Thread3 = 3 : 1

Content

- Background
- Problem Definition
- Motivational Experiments
- OctoKV: Design and Implementation
- Evaluation
- Conclusion

Experimental Setup

- Client
  - § Running a db_bench benchmark
    - 1) Light workload
      - *7 I/O threads issue Put/Get I/Os*
    - 2) Medium workload
      - *10 I/O threads issue Put/Get I/Os*
    - 3) Heavy workload
      - *12 I/O threads issue Put/Get I/Os*
  - § I/O request size = 16KB
- Server
  - § 6-core device
  - § Running a Linux OS using Intel SPDK



Comparison

(1) **Host KVS**

→ A hash-based key-value storage engine running on the client, layered atop the kernel and file systems

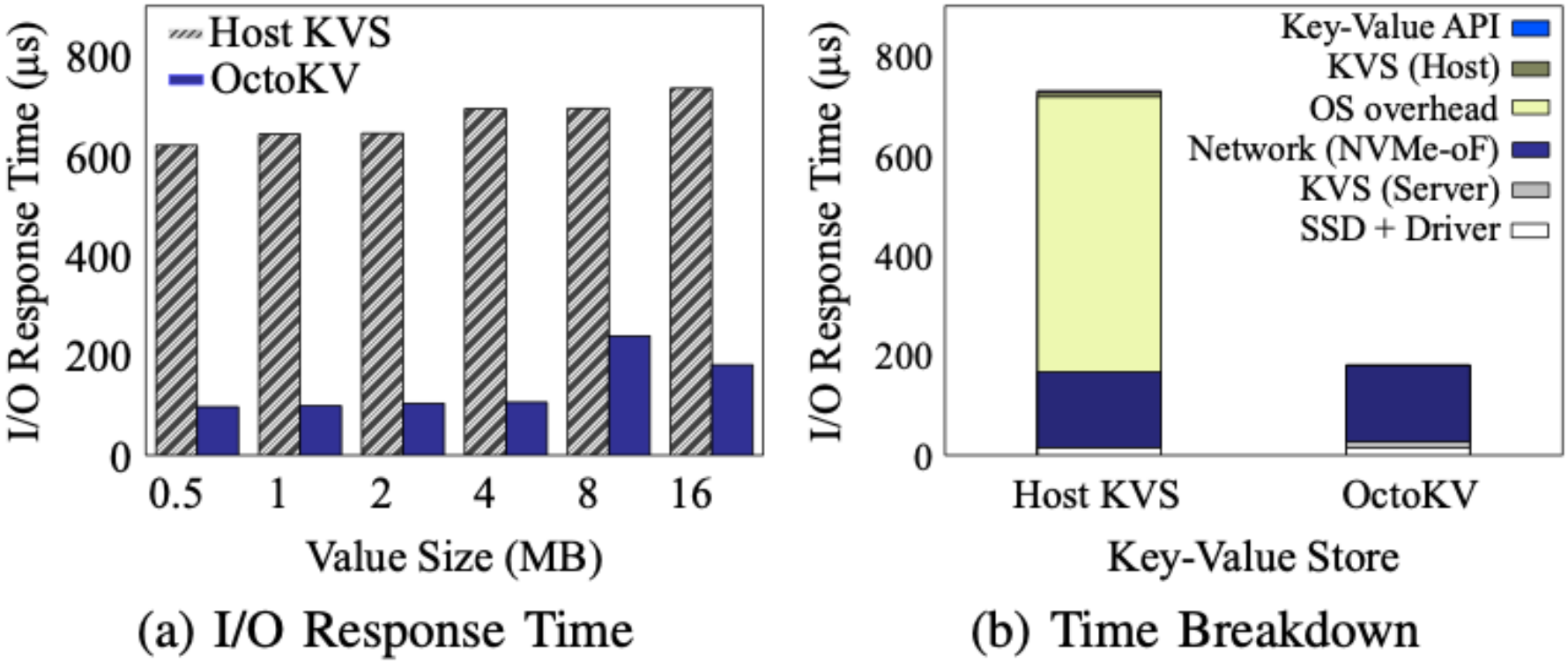
(2) **OctoKV**

→ The proposed system with only the key-value storage engine running on the server

(3) **OctoKV-LB**

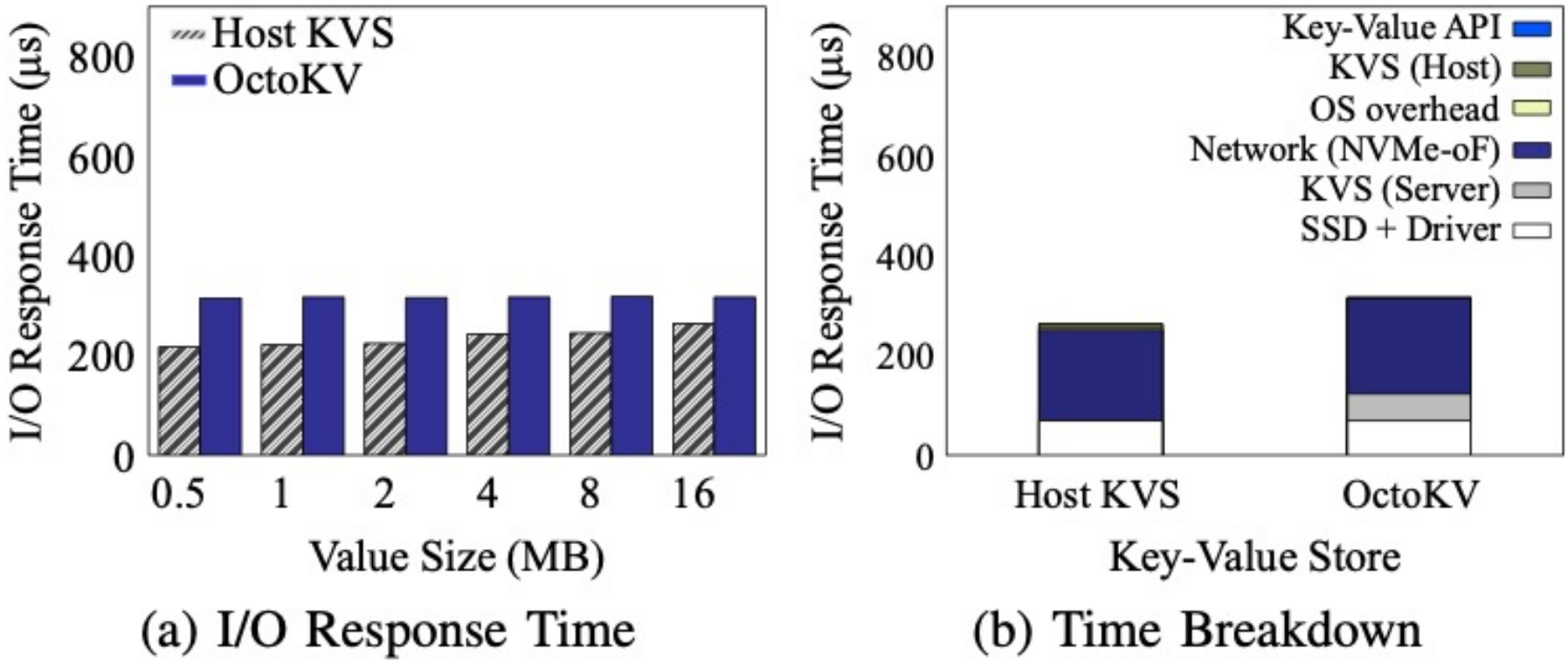
→ OctoKV with the load-aware balanced I/O scheduling

Evaluation – Put Workload



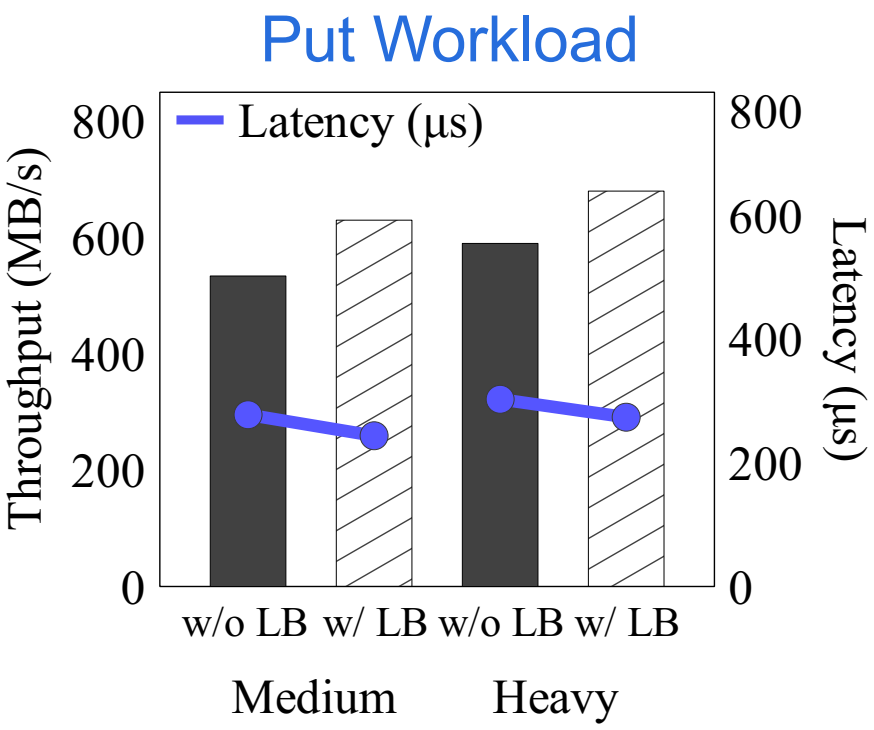
The fsync() overhead of the DB file is quite large.

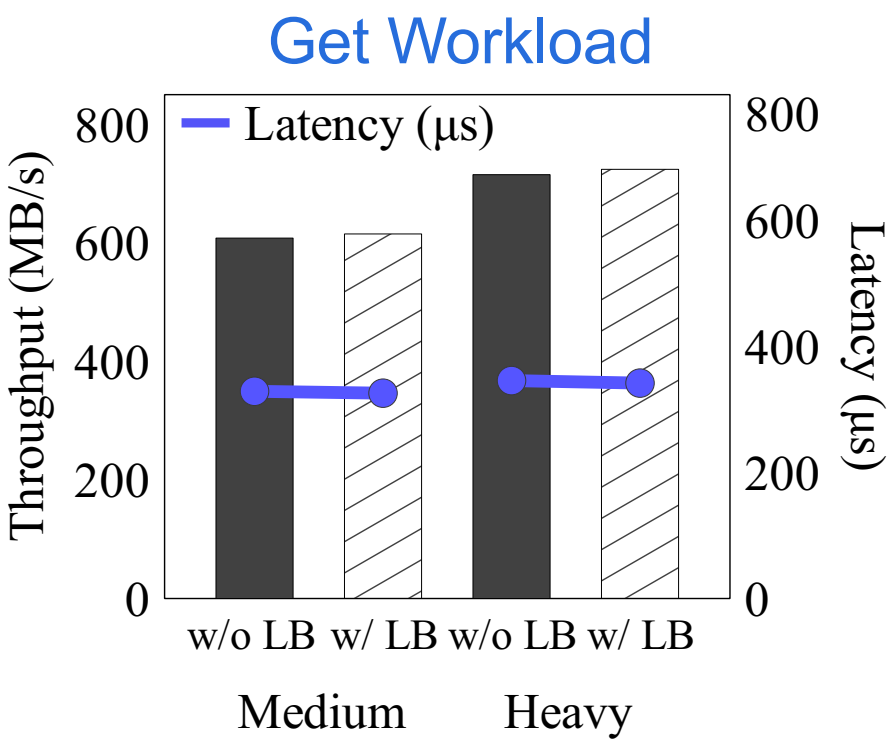
Evaluation – Get Workload



Storage servers have low CPU performance and are slow to run key-value stores.

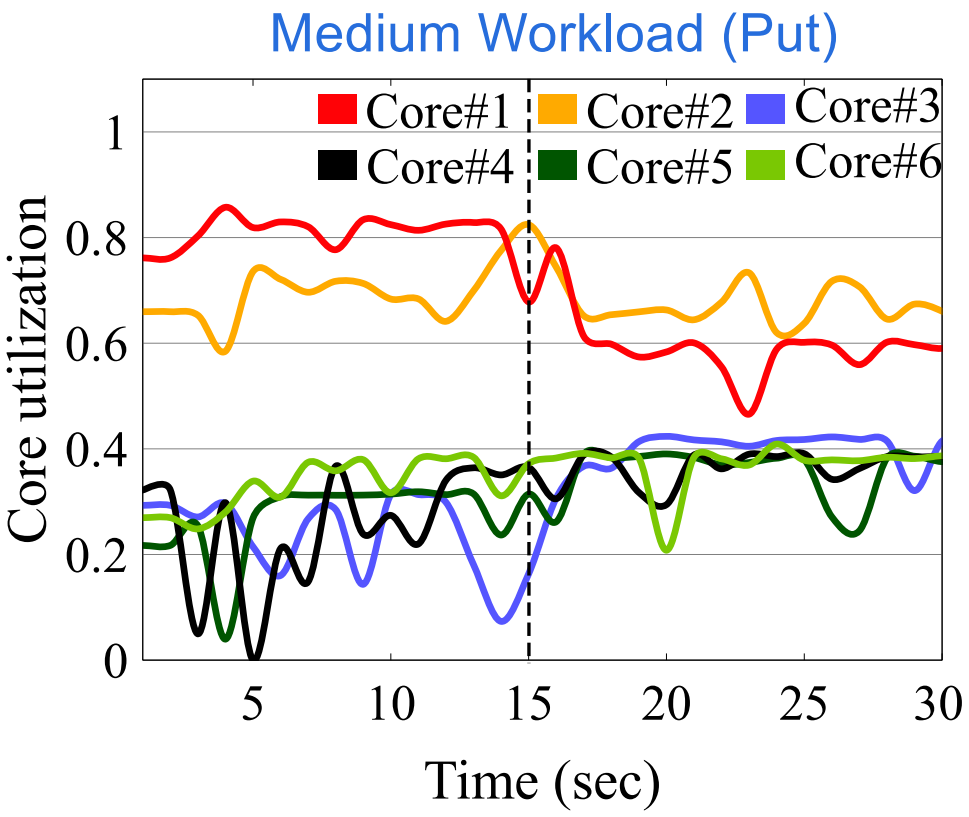
Evaluation – Load Balancing

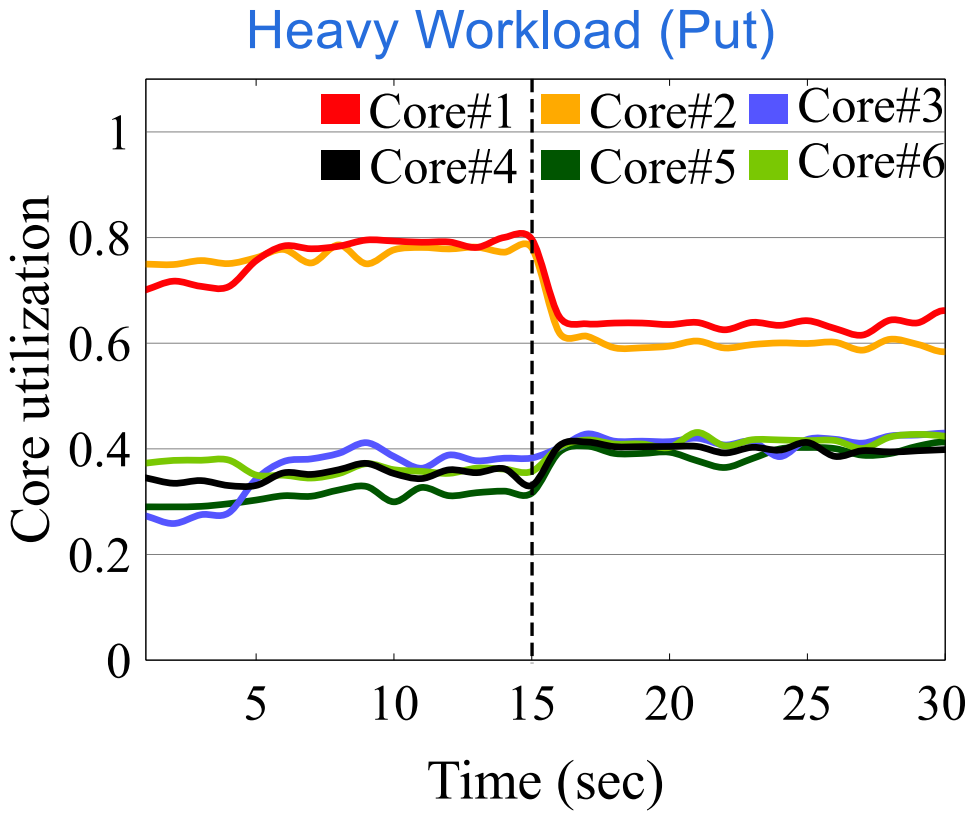




The performance gain was significant in the Put workload.

Evaluation – Load Balancing





Conclusion

- Proposed an **OctoKV, An Agile Network-Based Key-Value Storage System with Robust Load Orchestration**

  - OctoKV is a server-side key-value store that leverages the SPDK capabilities for high-performance in disaggregated storage

  - OctoKV has proposed a powerful load-aware balanced I/O scheduling


서강대학교
SOGANG UNIVERSITY
Thank You ☺
youkim@sogang.ac.kr
Youngjae Kim
서강대학교
SOGANG UNIVERSITY