ScaleCache:
A Scalable Page Cache for Multiple Solid-State Drives

Yongseok Son

Department of Computer Science and Engineering,

Chung-Ang University

Motivation

❖ **Emerging High-performance storage devices**
- NVMe-based SSDs with GB/s-level I/O bandwidth are widely adopted to satisfy increasing performance requirements

Seagate FireCUDA 530

Random read: 3.1 GB/s
Random write: 3.9 GB/s

Samsung PM1743

Random read: 6.6GB/s
Random write: 1.4GB/s

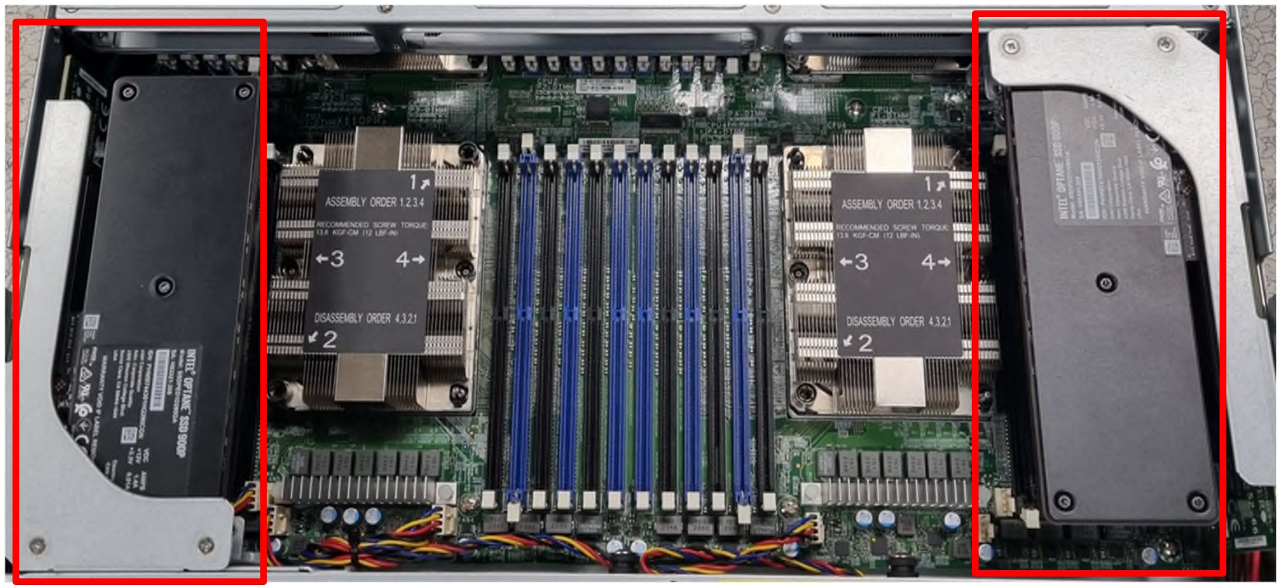
Intel Optane SSD 900P

Random read: 2GB/s
Random write: 2GB/s

Motivation

❖ **Data-intensive Applications on Multiple SSDs (RAID)**

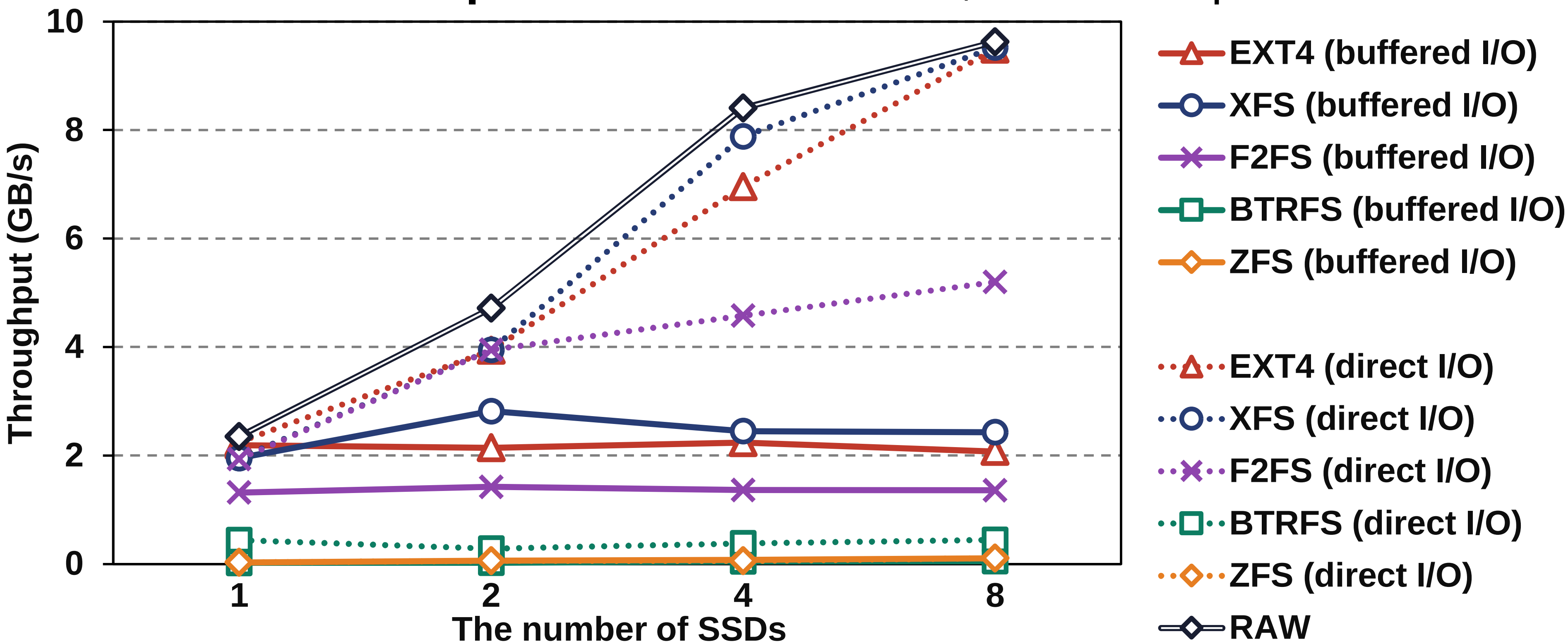
- Recent trends advocate using many SSDs for higher throughput in
  - Supercomputing [Patel et al., FAST'20]
  - Big data and graph processing [Wang et al., ATC'20]
  - Enterprise storage and Cloud services [Tong et al., ATC'21]
  - High-performance large-scale file server [Maneas et al., FAST'22]
- RAID configurations are attractive as they increases I/O performance, reliability and capacity



Motivational Evaluation

❖ **Write-intensive performance of diverse file systems**

- Unlike direct I/O, buffered I/O performance mostly does not scale well with the number of SSDs or even decreases
  - **FIO workload**: 64 threads, 3GB file size per thread, 4KB request size (commonly used in most OS and applications)
  - **RAID Setup**: RAID-0 with 8 SSDs, 512KB stripe size



SySLAB Systems and Storage Laboratory

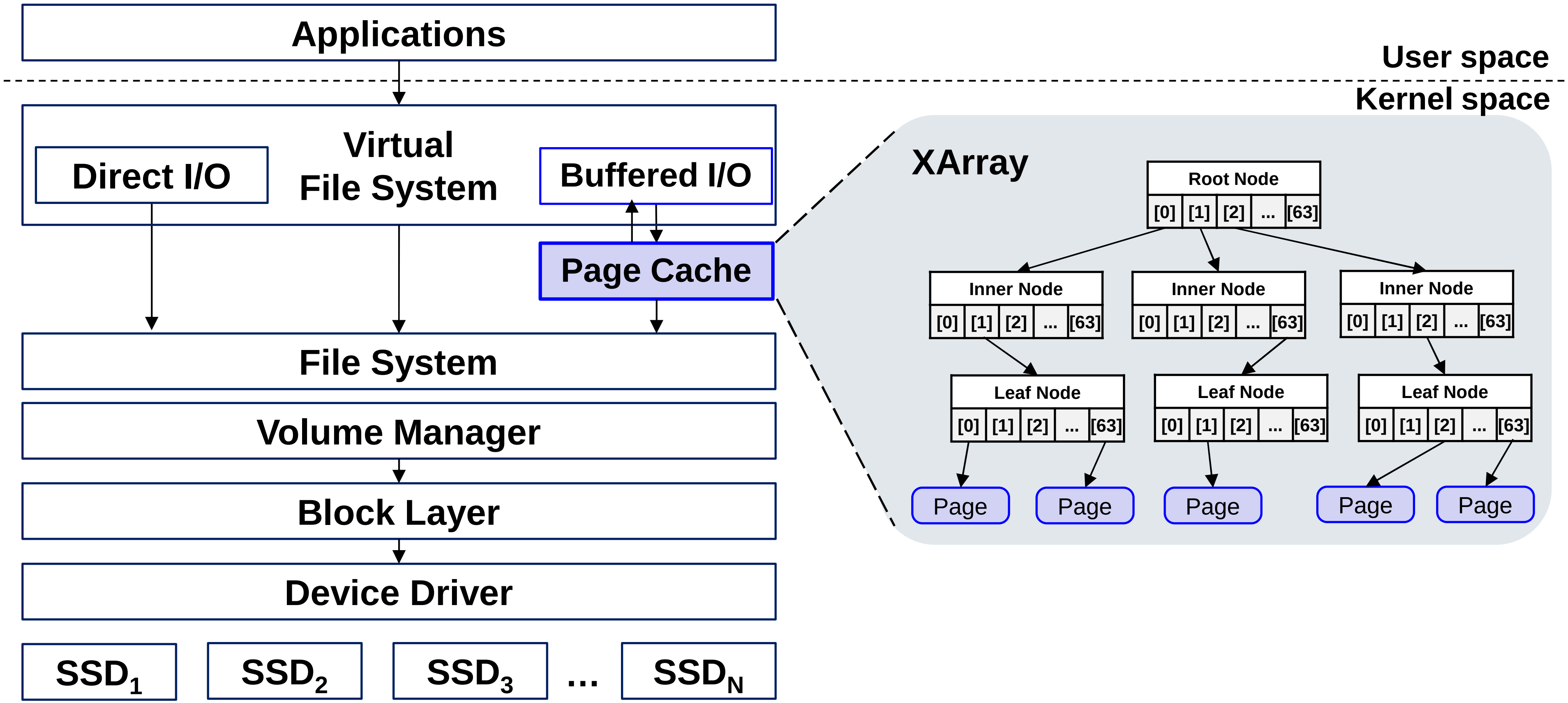
Background: Buffered I/O benefit

- **Linux kernel adopts Buffered I/O by default**
  - Page cache on the memory keeps data of files

- **Buffered I/O offers several benefits**
  - It minimizes I/O operations and provide low latency
    - In realistic scenarios
      - ✓ I/O fluctuations between heavy and non-heavy I/O activities
      - ✓ Better to maintain the advantages of buffered I/O while achieving performance comparable to direct I/O
  - It can be helpful to SSD lifespan by reducing the number of I/O
  - It does not require applications to align their I/O size

Background: Buffered I/O Flow

❖ Linux I/O Flow

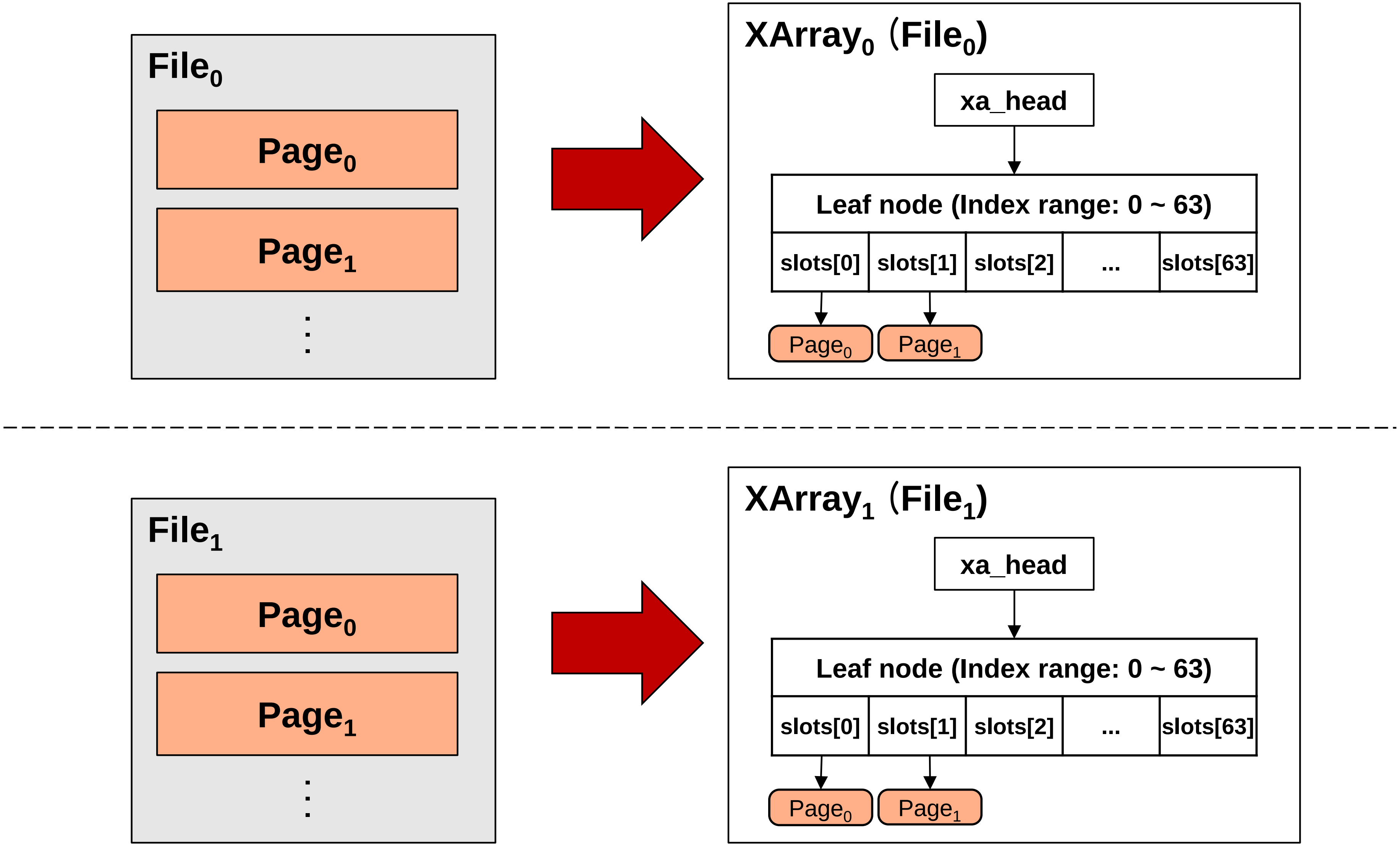


Background: XArray Definition

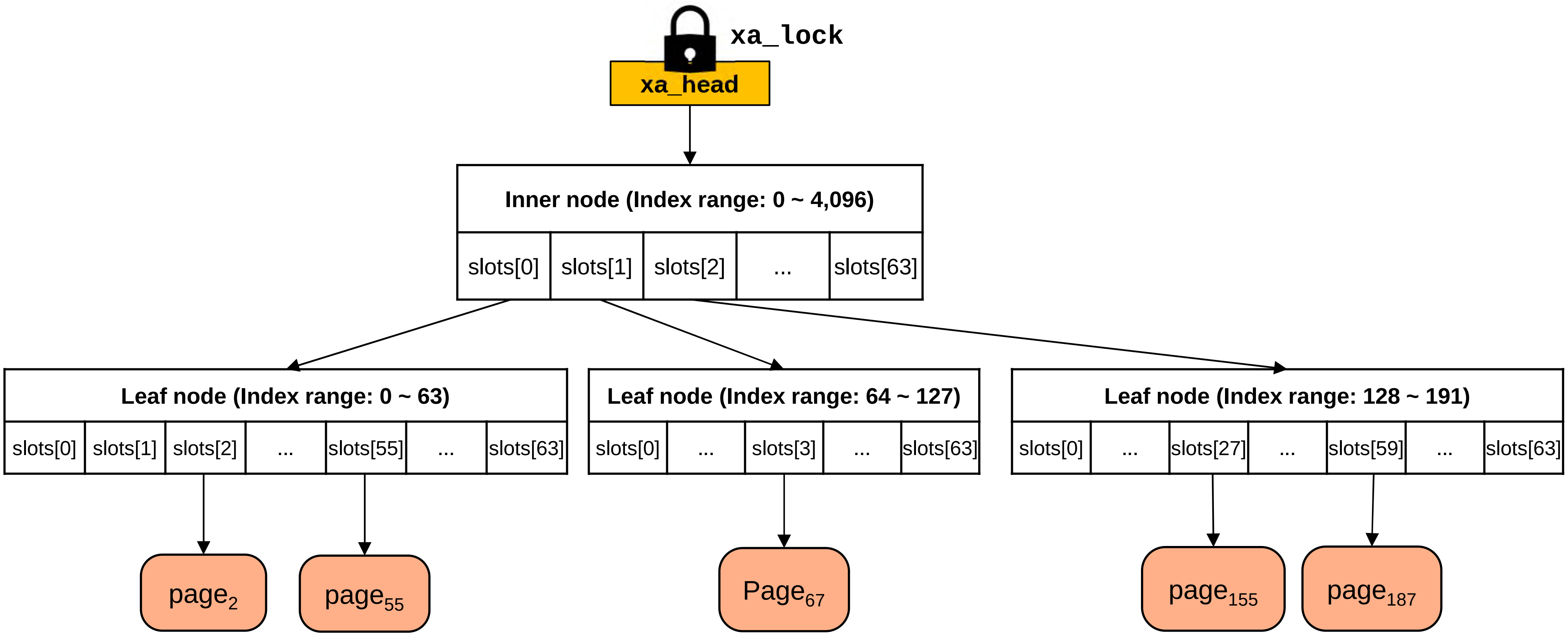
- **What is XArray? eXtensible Arrays**
  - XArray in the Linux kernel is a data structure which behaves like a very large array of pointers
  - **Dynamic-sized Array**
    - Unlike regular arrays, XArray does not have a fixed size.
    - It dynamically adjusts its size, making it suitable for managing large sets of objects or data within the kernel.
  - **Flexible Indexing**
    - XArray uses zero-based integer indexing to store and retrieve data quickly at specific locations.
    - The flexible nature of indexing ensures that inserting or accessing data at a particular index is efficient.
    - XArray uses "entries" to store data, which can be pointers, integers, or unique values, making it versatile for various data types.
  - **Minimized Locking**
    - Uses RCU and an internal spinlock to synchronize access

Background: Page, File, XArray

❖ Relationship between page index, file, XArray

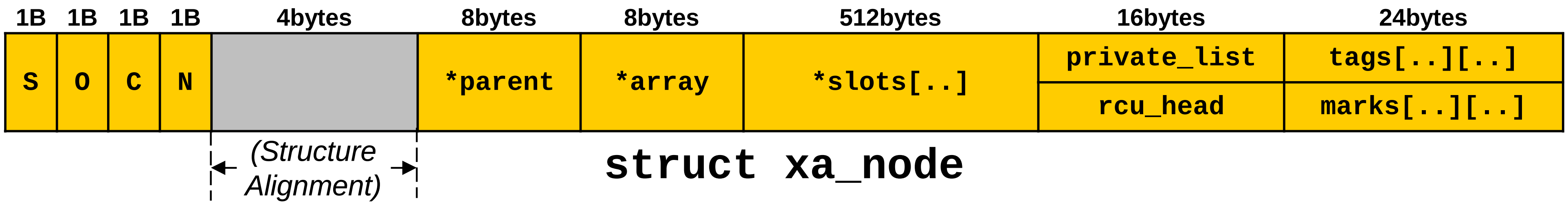


Background: XArray Overview



Background: XArray node

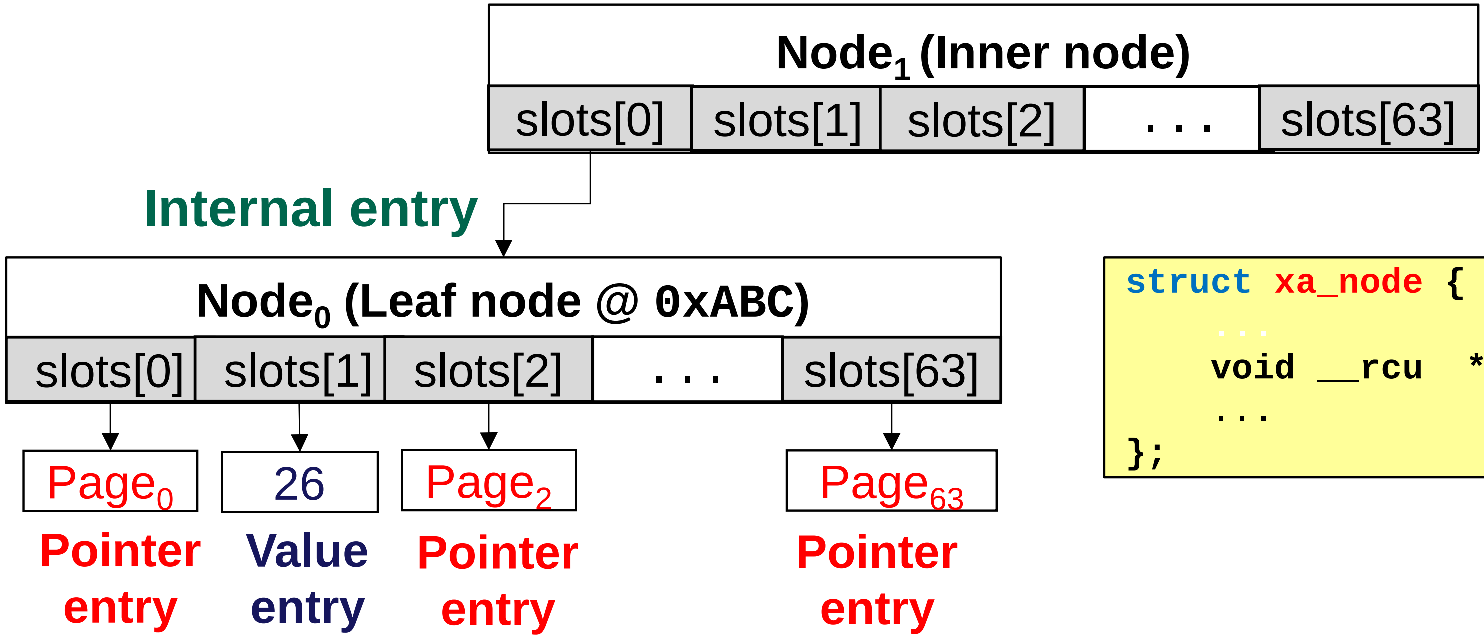
- **XArray node**
  - 64bit Architecture
  - 576 bytes per one xa_node structure
    - 576 % 8 = 0; ***8-byte aligned***
  - 4KB page can have up to 7 nodes
    - (576bytes * 7 nodes = 4,032bytes) < 4KB page
    - **S**: shift, **O**: offset, **C**: count, **N**: nr_values



shift: the bit in each node (6)
offset: the slot offset in parent
count: the count of element in the slots
nr_values: the count of a value entry
array: the xarray that the nodes belong to
slots: an array saving children nodes or elements
→ The number of slots is 64 by default

Background: XArray node slot

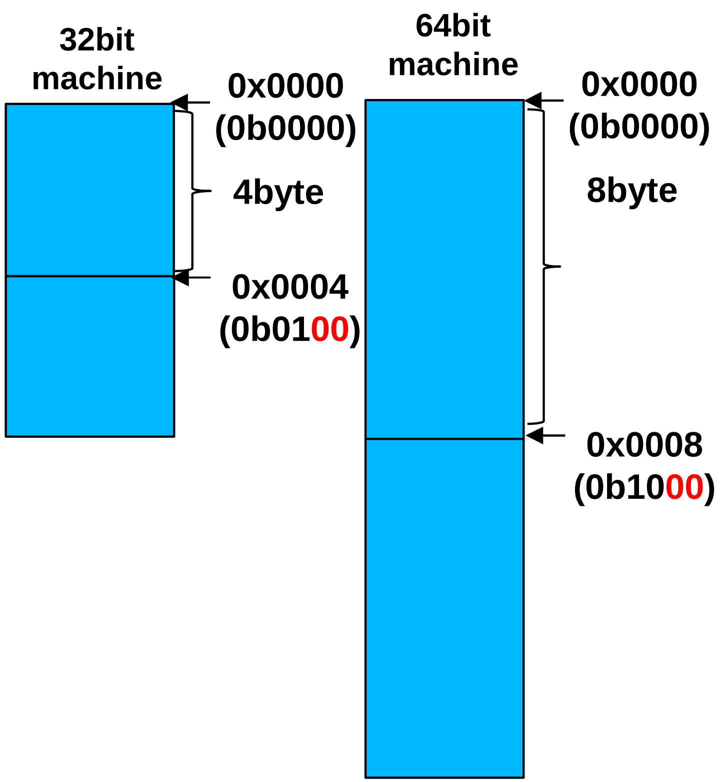
- **XArray node slots contains entries**
  - XArray has 64 slots by default
  - There are 3 types of entries in Xarray
    - **Pointer entry**
    - **Internal entry**
    - **Value entry**

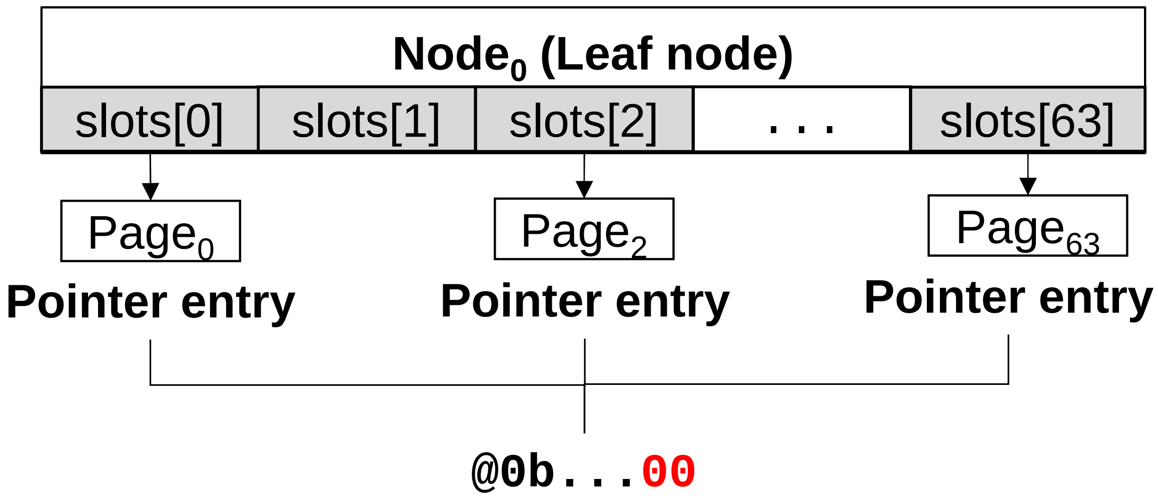



```
struct xa_node {
    ...
    void __rcu  *slots[CC_XA_CHUNK_SIZE];
    ...
};
```

Background: Pointer entry

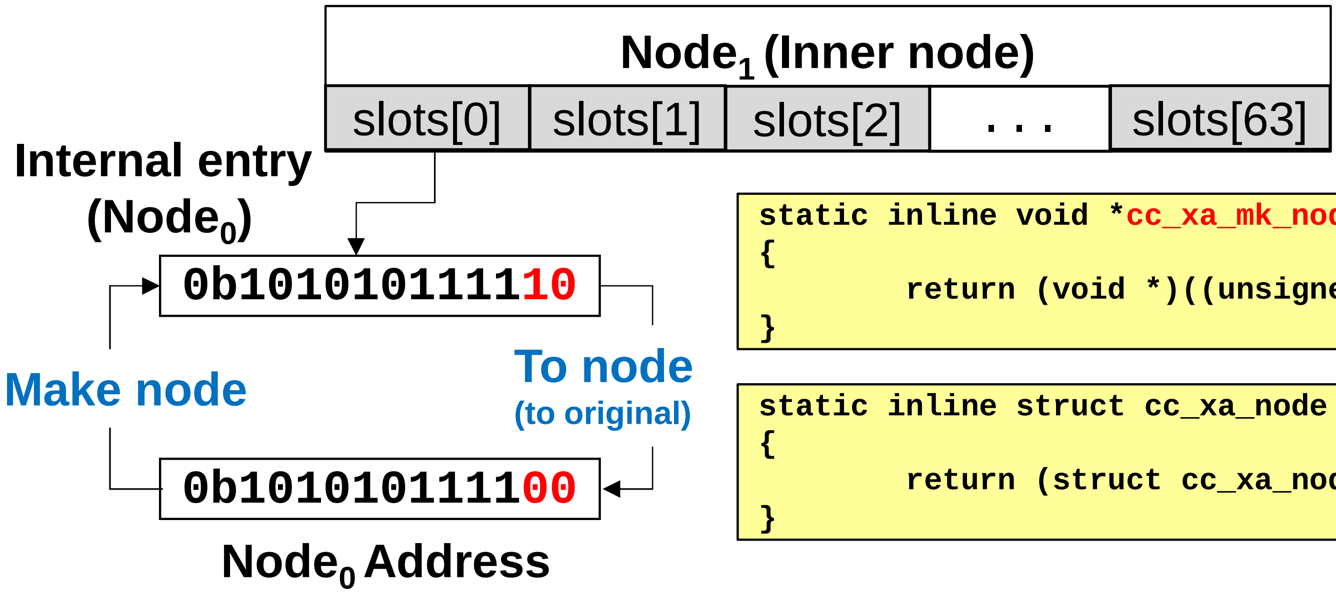
- **The last two bits of the entry determine how the XArray interprets the contents**
  - **0b00**: Pointer entry





Background: Internal entry

- The last two bits of the entry determine how the XArray interprets the contents
  - **0b10**: Internal entry




```
static inline void *cc_xa_mk_node(const struct cc_xa_node *node)
{
        return (void *)((unsigned long)node | 2);
}
```

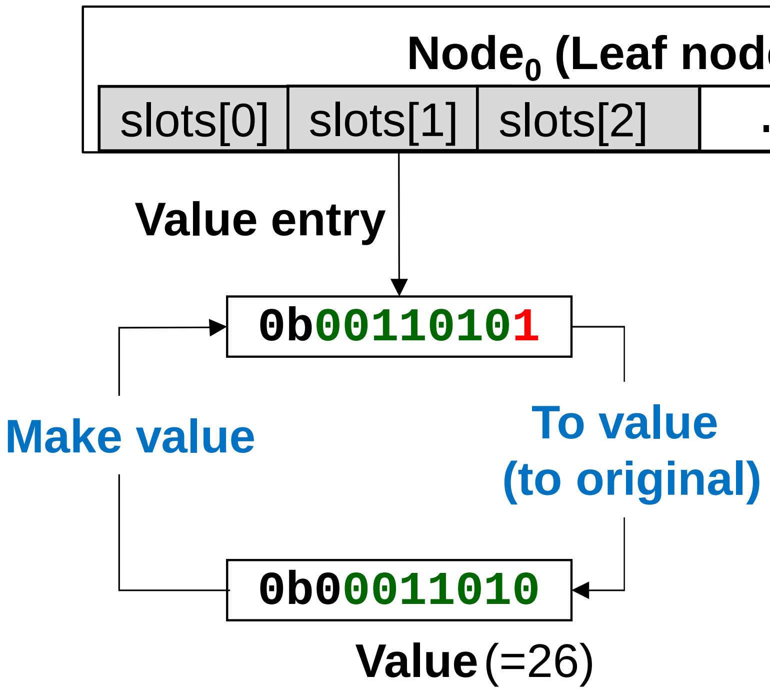
```
static inline struct cc_xa_node *cc_xa_to_node(const void *entry)
{
        return (struct cc_xa_node *)((unsigned long)entry - 2);
}
```

```
static inline bool cc_xa_is_internal(const void *entry)
{
        return ((unsigned long)entry & 3) == 2;
}
```

The last two bits should be 0b10

Background: Value entry

- **The last two bits of the entry determine how the XArray interprets the contents**
  - **`0bx1`**: Value entry

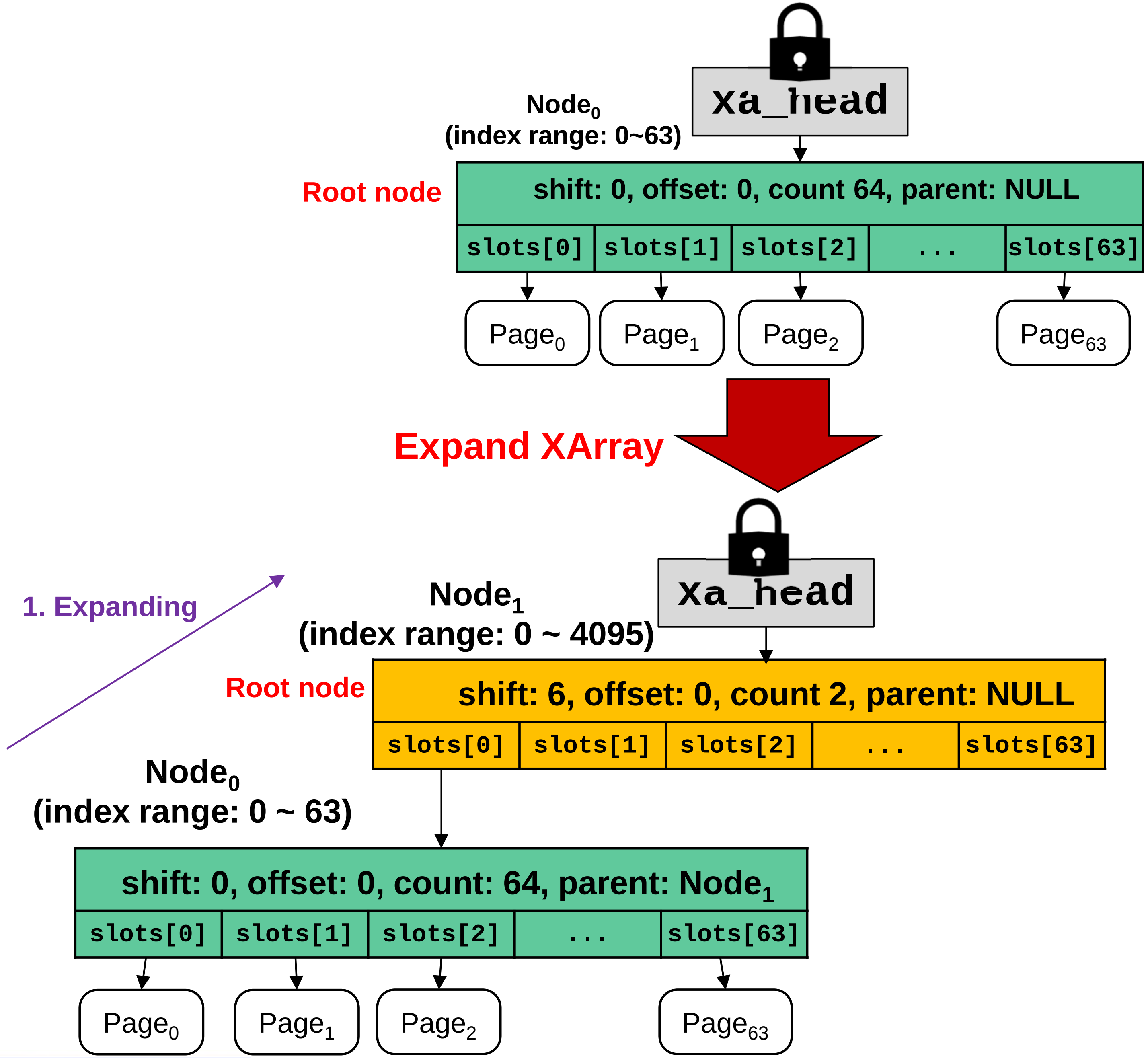



```c
static inline void *cc_xa_mk_value(unsigned long v)
{
        WARN_ON((long)v < 0);
        return (void *)((v << 1) | 1);
}
```

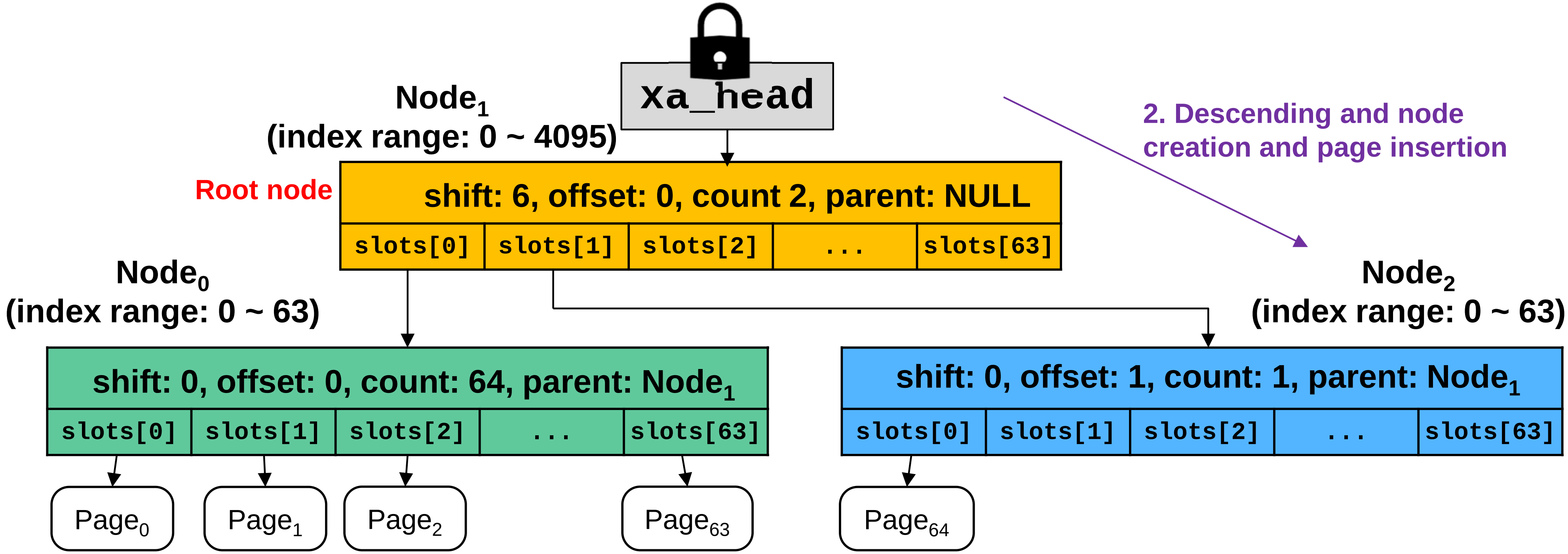
```c
static inline unsigned long cc_xa_to_value(const void *entry)
{
        return (unsigned long)entry >> 1;
}
```

```c
static inline bool cc_xa_is_value(const void *entry)
{
        return (unsigned long)entry & 1;
}
```

Background: XArray node insertion (1)



Background: XArray node insertion (2)

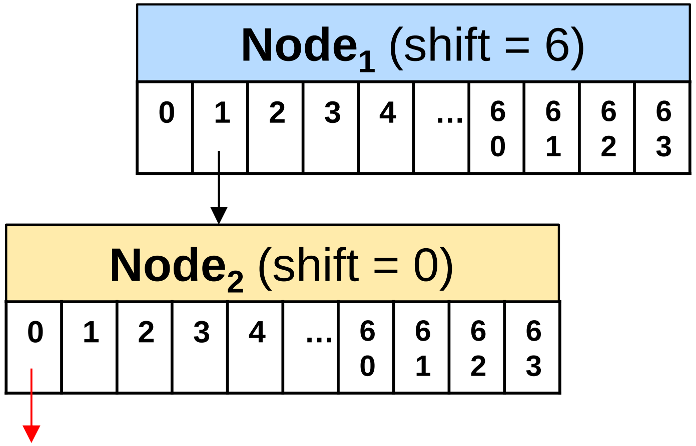


Background: Searching a page

Ex) Descending and Searching for a page index: 64 (000001 000000)

shifts — 6 bits →

index	000001 (1)	000000 (0)



Page (index 64)

lib/xarray.c

```
static unsigned int get_offset(unsigned long index, struct xa_node *node)
      return (index >> node->shift) & XA_CHUNK_MASK;
```

Ex1) When getting an offset in $Node_1$ for the page index,

1) `000001 000000 >> 6`

2) `000000 000001 & 000000 111111` XA_CHUNK_MASK

→ `000001(1)` (last 6 bits)

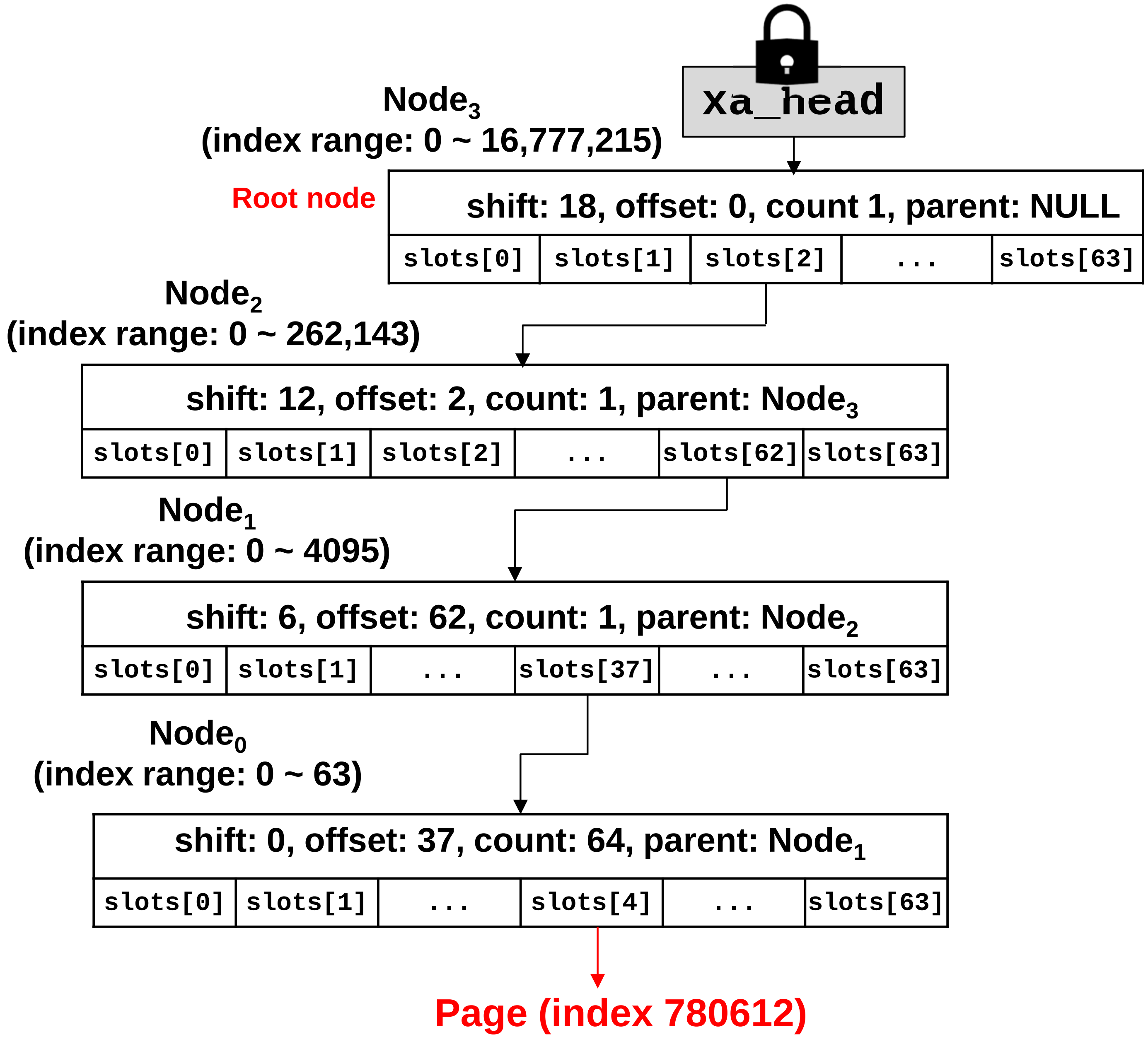
Ex2) When getting an offset in $Node_2$ for the page index,

1) `000001 000000 >> 0`

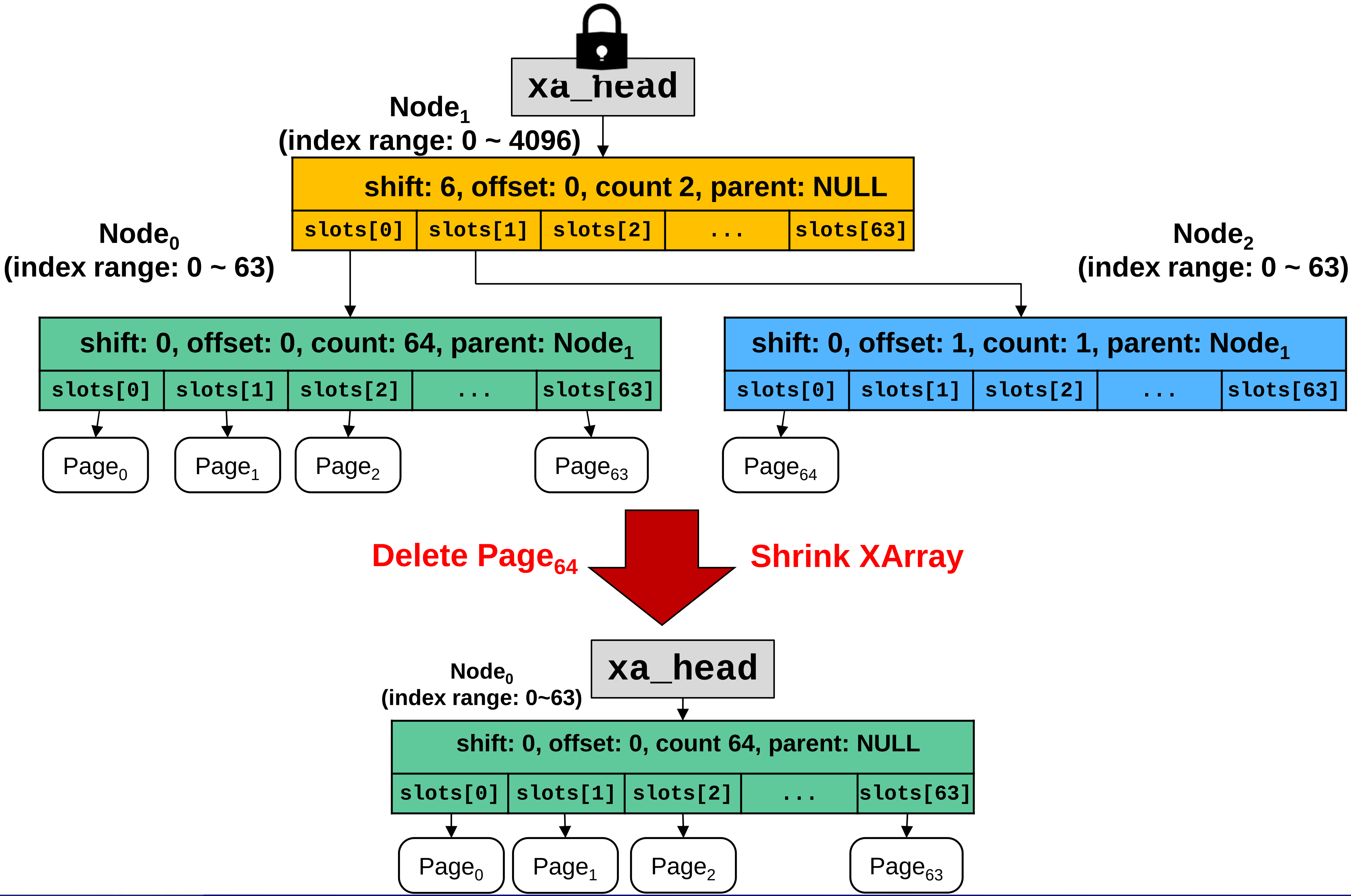
2) `000001 000000 & 000000 111111` XA_CHUNK_MASK

→ `000000(0)` (last 6 bits)

Background: Searching a page

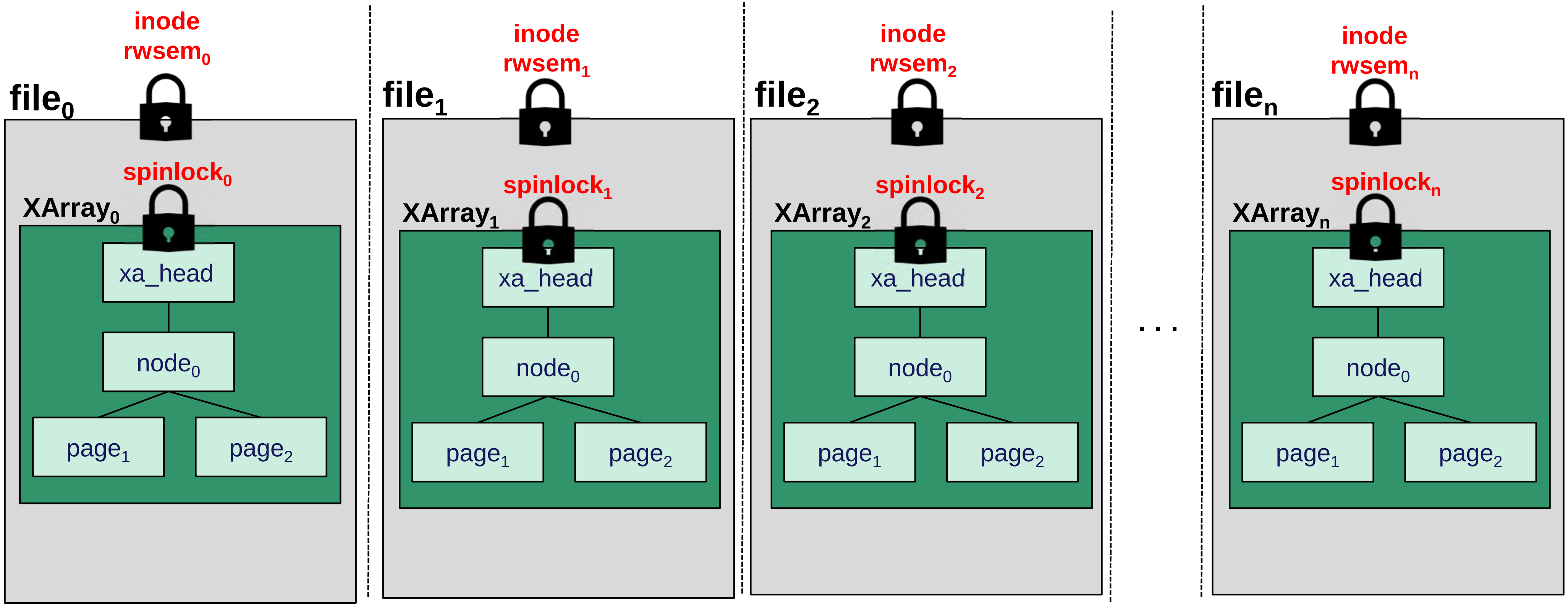


Background: XArray node deletion



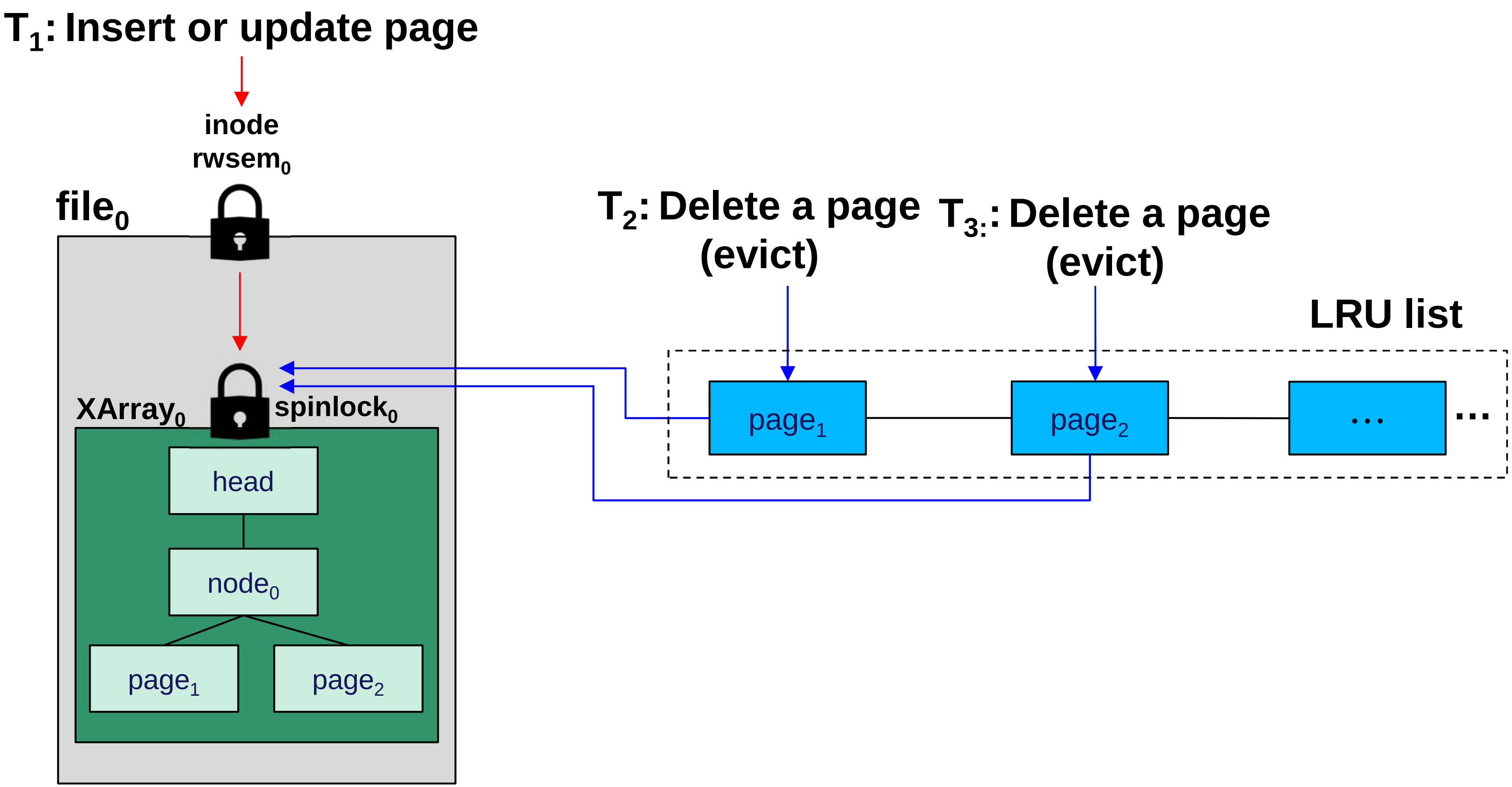
Background: XArray and File system

- **Page cache is managed based on Per-inode (Per-file)**
  - XArray is Per-file data structure



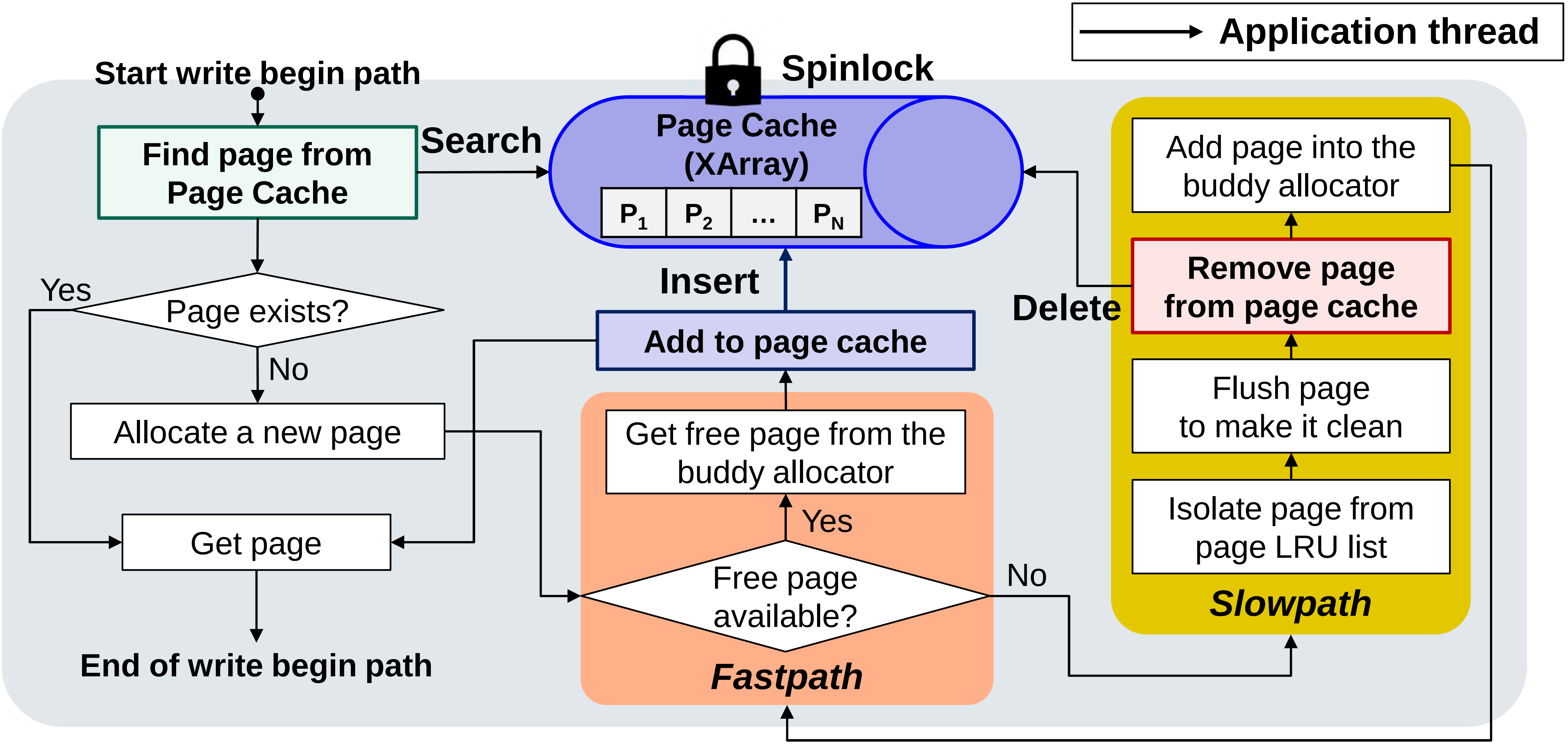
Background: XArray and File system

- **Page cache is managed based on Per-inode (Per-file)**
  - XArray is Per-file data structure
  - When a page is evicted to flush, the page is evicted based on LRU



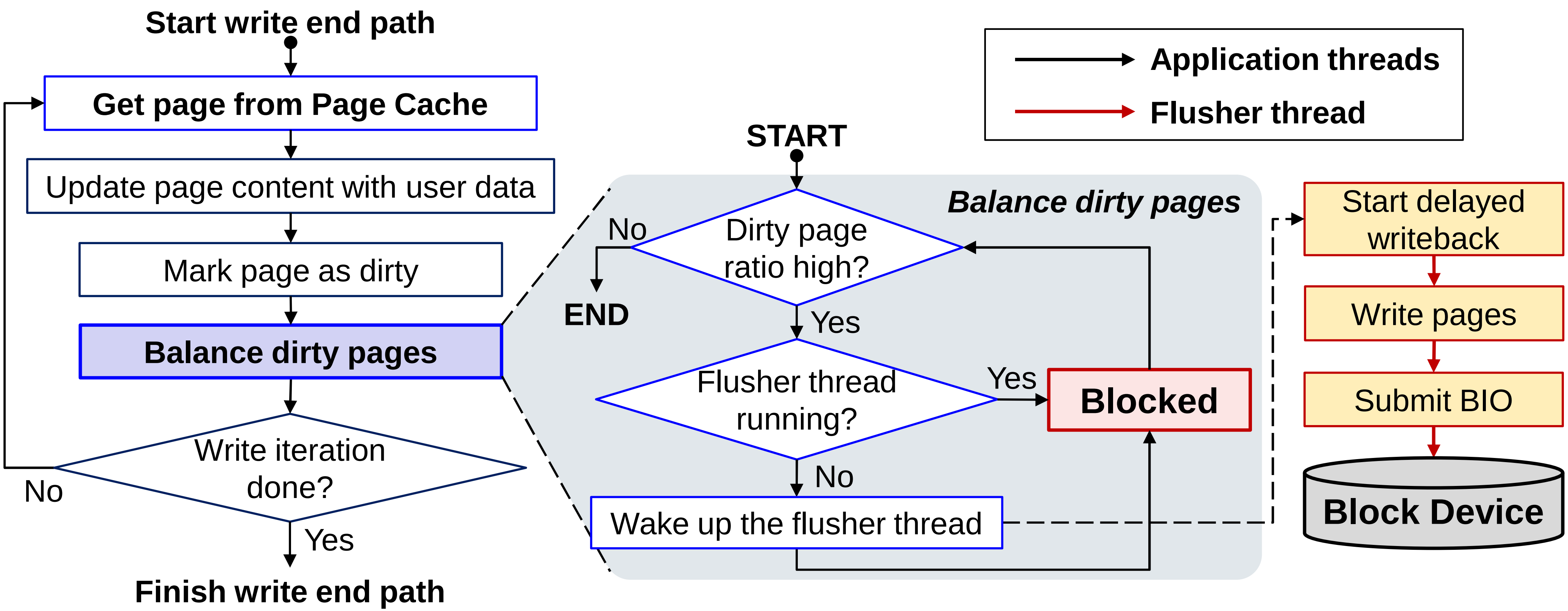
Background: Linux Buffered I/O Flow

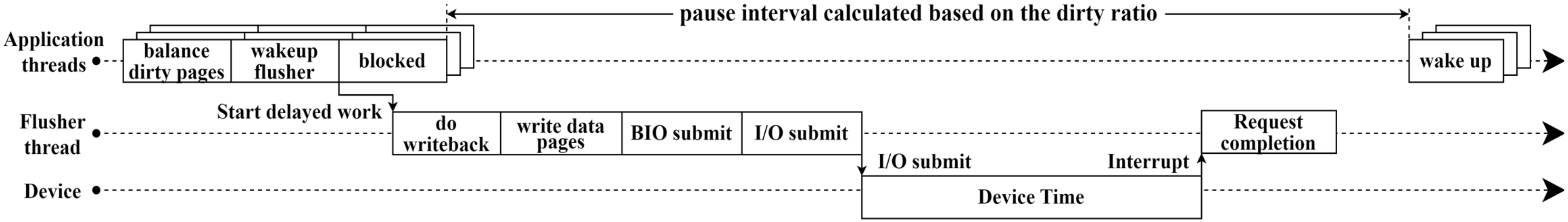
- **Simplified Page Cache operations**
  - **Search operation**: RCU lookup
    - Supports multiple readers with **a single writer**
  - **Insert or Delete operation**: Spinlock (xa_lock)
    - Resolves conflicts between multiple writers



Background: Linux Buffered I/O Flow

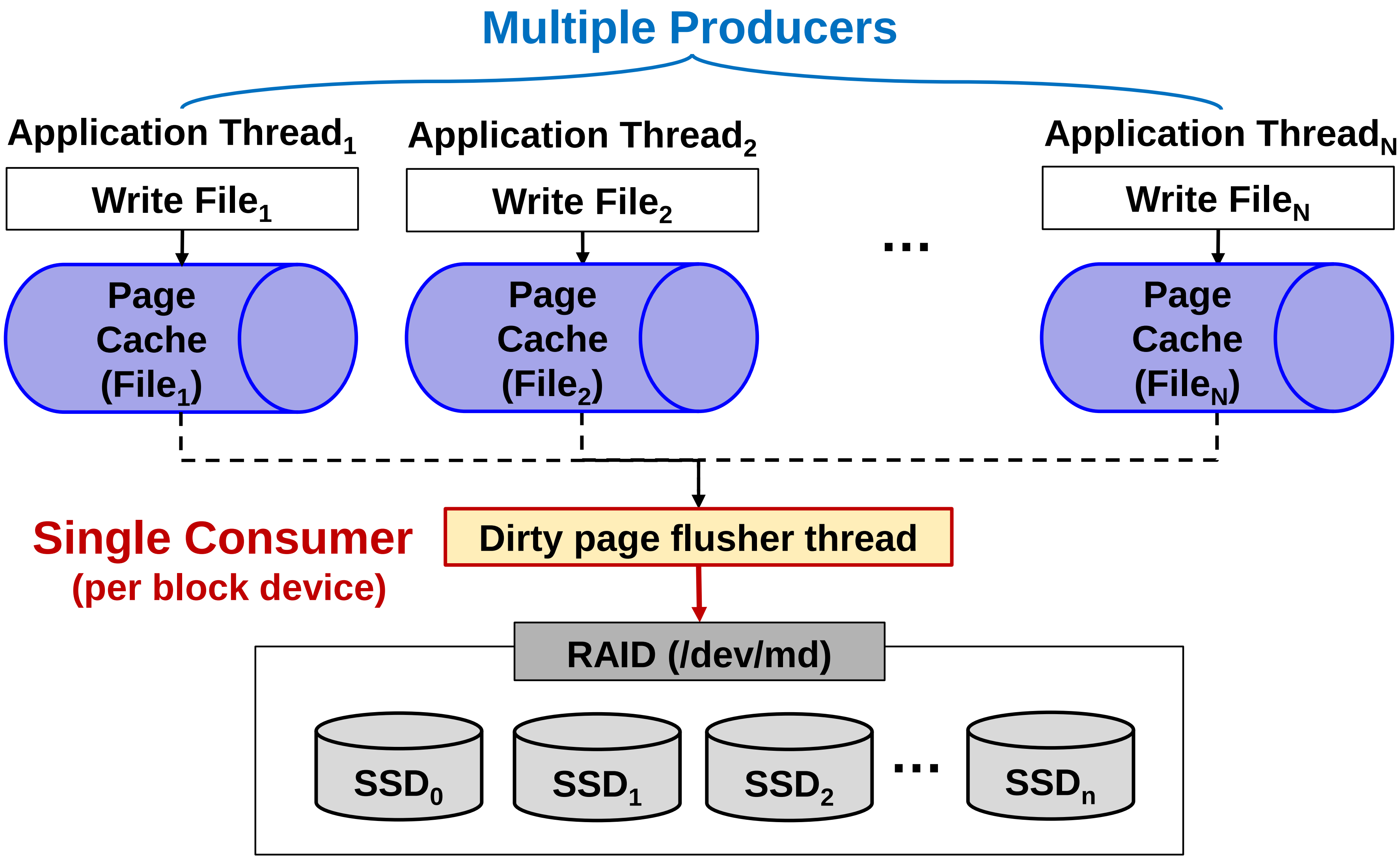
❖ Simplified Balancing dirty pages





Background: Linux Buffered I/O Flow

- **Limited I/O parallelism**
  - A single flusher per block device



Challenges

- **Two Challenges in Page Cache Management**
  - **Limited concurrency**
    - Application threads frequently insert/update/delete page cache under non-scalable spinlock
    - This spinlock serializes multiple writers, resulting in high lock contention
  - **Limited I/O parallelism**
    - A single flusher performs I/O operation even if multiple SSDs are used
    - This limits the I/O parallelism offered by the multiple SSDs

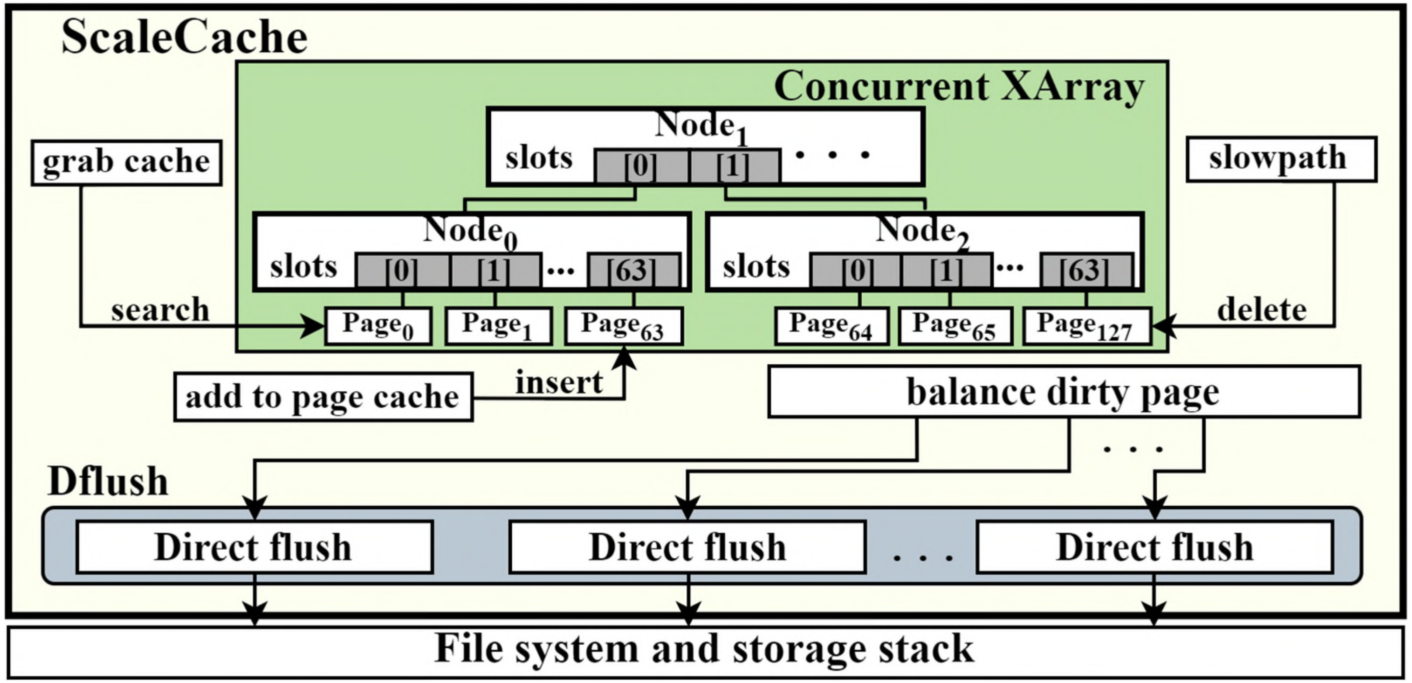
Our Goal

- **Achieving higher SSD scalability on multi-cores**
  - Scaling buffered I/O performance close to Direct I/O
    - Reducing the lock contention in the page cache
    - Maximizing I/O parallelism in the page cache

Overall Architecture

❖ **Concurrent and I/O parallelized page cache**

- A concurrent XArray – ***ccXArray***
  - Application threads can update page cache concurrently without being spin-locked and serialized
  - Concurrent Multiple Writers
- A direct dirty page flush – ***dflush***
  - Application threads directly balance dirty pages in the system instead of being blocked and wait for a single flusher thread
  - Multiple Producers with Multiple Consumers



Strategy of ScaleCache

- **Four main key strategies to design *ccXArray*:**
  - **Strategy 1**: Winner strategy
  - **Strategy 2**: Per-thread node access tracking
  - **Strategy 3**: Efficient design of ccXArray node
  - **Strategy 4**: Lazy node deletion and reuse
- **One main key strategy to design dflush**
  - **Strategy:** one(thread)-to-one(inode) model

How to make concurrent data structure

Combination of Atomic instructions

- GCC built-in functions for atomic memory access
  - __sync_fetch_and_add (type *ptr, type value)
  - __sync_lock_test_and_set (type *ptr, type value)
  - __sync_val_compare_and_swap (type *ptr, type oldval, type newval)
  - etc
- We can use this GCC built-in atomic functions when we use GCC compiler

How to make concurrent data structure

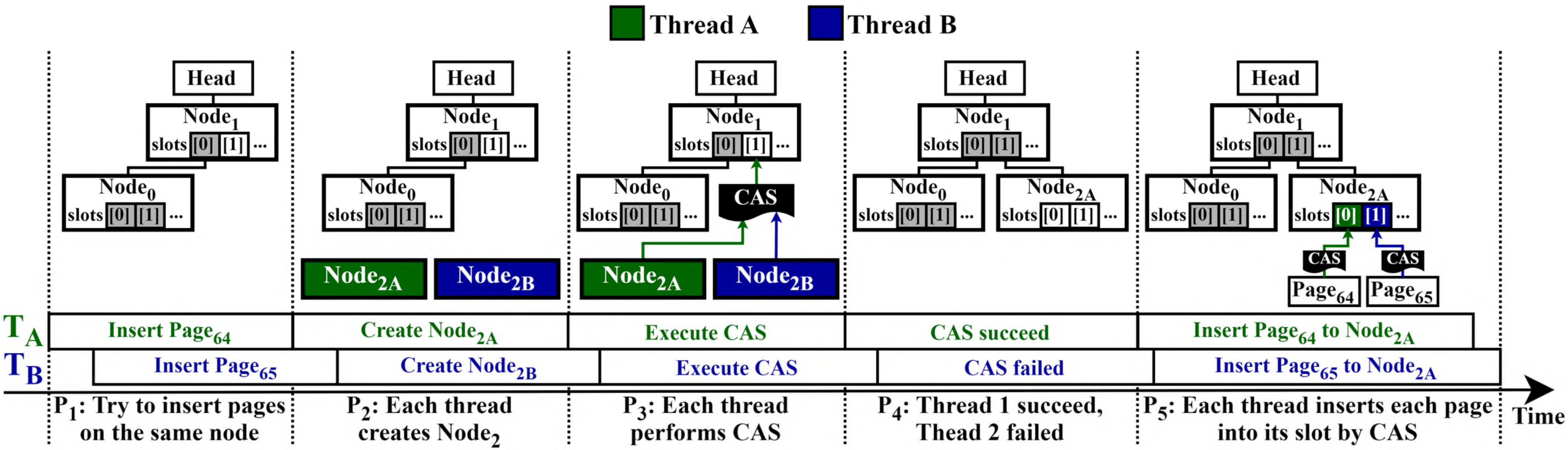
❖ Simple Example

```
new_node = alloc();
entry = CAS(current_node.slots[offset], NULL, new_node)
If(entry =! NULL){
        free(new_node)
        current_node = getnode();
}
 else
        current_node = new_node
```

Concurrent creation and insertion

❖ Node creation and page insertion (Winner Strategy)

- **Using Compare-And-Swap**: Elect a winner among threads concurrently running on ccXArray
- Allow the winner to insert/delete a page within the node or create/insert a node in ccXArray



Winner strategy: Concurrent page insertion and node creation

Concurrent node creation

```
while shift ≠ 0 do                                ▷ Descend until reaching the target leaf node
    shift ← shift − XA_CHUNK_SHIFT
    if entry == NULL then                                             ▷ Create child node
        new_node ← GET_NODE(alloc_node())
        new_node.shift ← shift
        entry ← CAS(curr_node.slots[offset], NULL, new_node)
        if entry ≠ NULL then
            free_node(new_node)
            curr_node ← GET_NODE(entry)
        else
            curr_node ← new_node
            atomic_add(curr_node.parent.count, 1)
        end if
    else                                                              ▷ Follow child node
        curr_node ← GET_NODE(entry)
        while atomic_read(curr_node.ldflag) == ON do
            Wait for logical node deletion
        end while
        if CAS(curr_node.del, ON, OFF) == ON then                             ▷ Reuse
            atomic_add(curr_node.parent.count, 1)
        end if
    end if
```

Winner strategy

1) If a target node (e.g., inner or leaf node) does not exist, each thread creates and gets its own node and tries to insert its own node at the slot using a CAS operation.
2) Only the CAS-succeeded thread can insert its created node to the slot

The CAS-failed threads cannot insert their node, free the created nodes, and use the node of the CAS-succeeded thread to descend

Lazy deletion

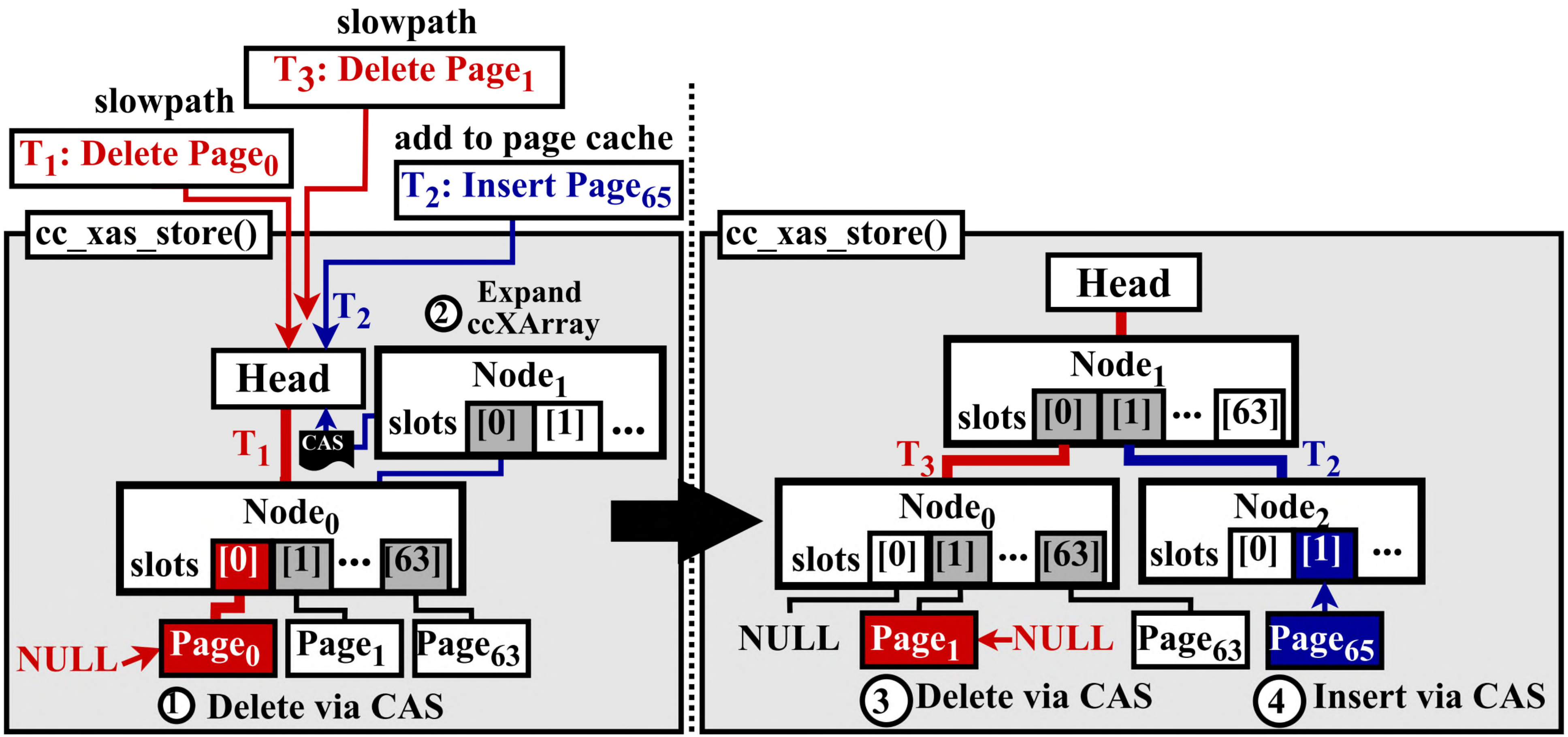
1) If a node already exists at the slot, they get the existing node and check if this node is tried to be deleted logically according to our lazy node deletion strategy.
2) If so, they wait for the logical node deletion procedure to be finished.

Reuse

If the target node is already logically deleted, according to our reuse strategy, the node can be reused again via CAS operation

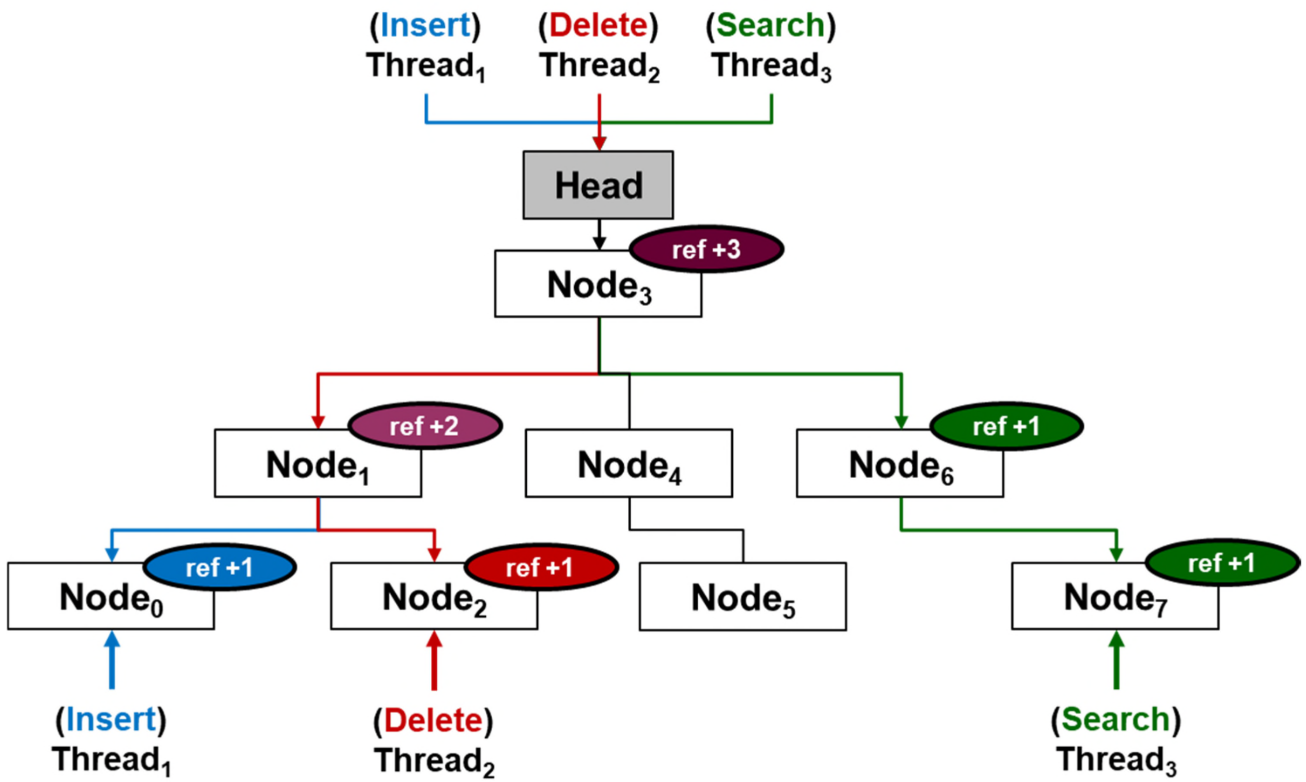
Concurrent expand operation

- **Expanding XArray**
  - In spite of during expanding nodes, ccXArray does not block the insert or delete operations



Per-thread node access tracking

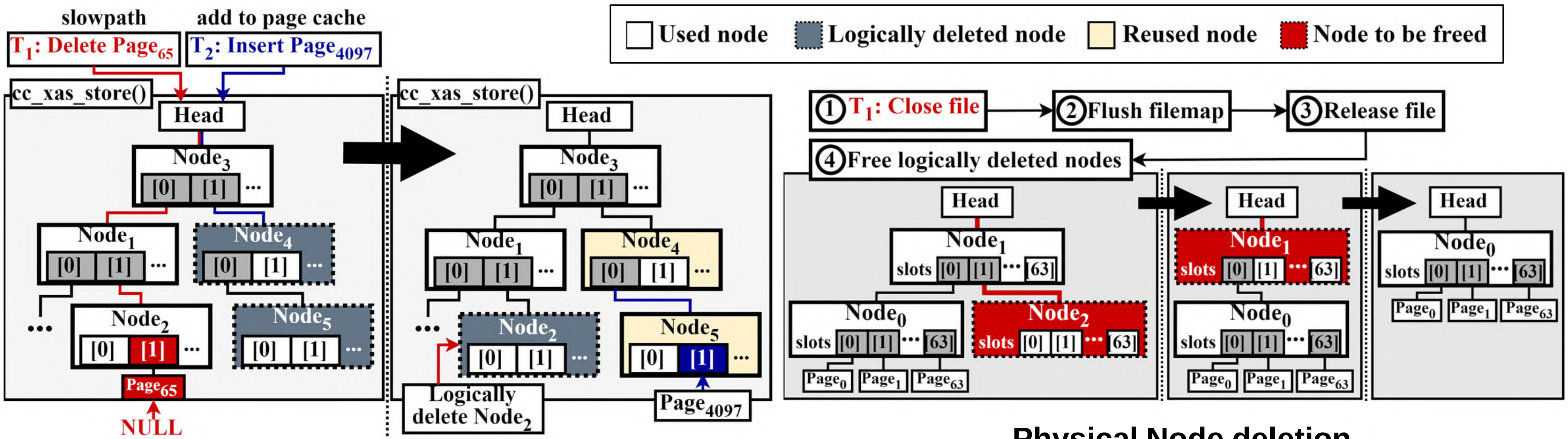
- **To avoid read/write and write/write conflicts**
  - Whenever the threads access or update to ccXArray nodes, we track the access of all the nodes by inserting the node into the per-thread list in an access order.
  - Always Increase from top to bottom



Lazy node deletion and reuse

❖ Logically and physically deletion and reuse

- Atomically mark the node as logically deleted when no pages in the ccXArray node
- Reuse node if it is requested before being deleted physically
- Delete the node physically in a certain situation where any page cannot be inserted / searched in the page cache



Logical Node deletion and reuse

Physical Node deletion

Efficient design of ccXArray node

❖ To support our deletion and tracking

- Three new indicators at the unused area in the node:
  - **Del flag:** indicates a logically deleted node
  - **LD flag**: a node is undergoing logical deletion
  - **Ref count**: tracks number of threads referencing the node
- Original XArray node design is intact for fully utilizing cache-line memory efficiency and the compatibility
  - In 64-bit systems, an XArray node is 576bytes
    - ✓ Up to 7 nodes in a 4KB page

1B	1B	1B	1B	1B	1B	1B	1B	8bytes	8bytes	512bytes	16bytes	24bytes
S	O	C	N					*parent	*array	*slots[..]	private_list	tags[..][..]
											rcu_head	marks[..][..]

(Structure Alignment)

Original Xarray node Structure

1B	1B	1B	1B	1B	1B	2bytes	8bytes	8bytes	512bytes	16bytes	24bytes
S	O	C	N	D	L	RF	*parent	*array	*slots[..]	private_list	tags[..][..]
										rcu_head	marks[..][..]

ccXarray node Structure

S: shift, O: offset, C: count, N: nr_values, D: delete flag, L: ldflag, RF: reference count

Logical node deletion

- **Simplified procedure of logical node deletion**

```
function CC_LOGICAL_DELETE_NODE(node)
    while node is not root node do
        if atomic_read(node.count) ≠ 0 then
            return
        else if CAS(node.ldflag, OFF, ON) ≠ OFF then
            return
        else if atomic_read(node.refcnt) > 1 then
            CAS(node.ldflag, ON, OFF)
            return
        else if CAS(node.del, OFF, ON) ≠ OFF then
            return
        end if
        atomic_sub(node.parent.count, 1)
        CAS(node.ldflag, ON, OFF)
        node ← node.parent
    end while
end function
```

Recheck for any potential page insertion

Informs other **upcoming threads** the target node will be tried to be deleted logically

Check the target node reference count
If refcnf > 1, stop logical deletion since other threads can access the target node

The target node can be deleted logically only if the conditions are satisfied

The number of nodes of the target node's parent decreases

The logical deletion for the target node is finalized

Node and page search

❖ Simplified search procedure

```
function CC_XAS_LOAD(index, head)
    entry ← head
    while entry is a node do
        node ← GET_NODE(entry)
        while atomic_read(node.ldflag) == ON do
            Wait for logical node deletion
        end while
        if atomic_read(node.del) == ON then
            return NULL
        end if
        offset ← (index ≫ node.shift) & XA_CHUNK_MASK
        entry ← atomic_read(node.slots[offset])
        if node.shift == 0 then
            goto found
        end if
    end while
found:
    PUT_ALL_NODES()
    return entry
end function
```

Increase reference count of nodes from root node to leaf

This ensures that the searching thread does not proceed further until the deleting thread finishes its work.

Only after the node is not occupied by the deleting thread, the searching threads check if the node is logically deleted

Leaf node is found

Direct flush

- **Throttling and balancing mechanism**
  - To flush the pages to HDD, the Linux kernel adopts a throttling mechanism with a page flusher which adjusts the number of I/Os, collects the I/Os, and submits them by considering the dirty page ratio in the page cache.
  - This leads to many benefits for single-channel HDD.
    - Specially, the throttling mechanism makes serialized I/O and sequential I/O patterns and reduces the amount of I/Os to HDD as much as possible.
    - In addition, the mechanism blocks the application threads for the flushing operations. There are two reasons for the blocking operation as follows.
      - ✓ (1) It prevents application threads from generating dirty pages anymore to get free pages.
      - ✓ (2) Multiple flushing operations with multiple flushers can negatively affect the performance of a single-channel device.

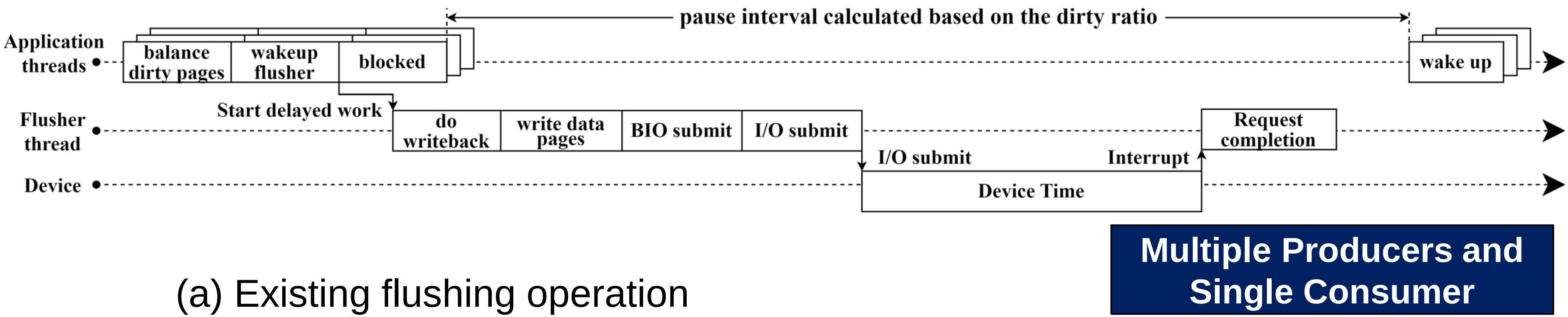
Direct flush

- There are three potential negative effects of throttling mechanism on multiple SSDs as follows.
  - The existing blocking operation which blocks the application threads hinders the opportunity to flush more dirty pages per unit time.
  - The blocking time which blocks application threads can be longer than the time required for I/O operation in the case of low-latency SSDs.
- Dflush
  - dflush opportunistically allows the application threads to perform the flush operation directly and parallelize the I/O operations instead of being blocked and waiting for I/O completion

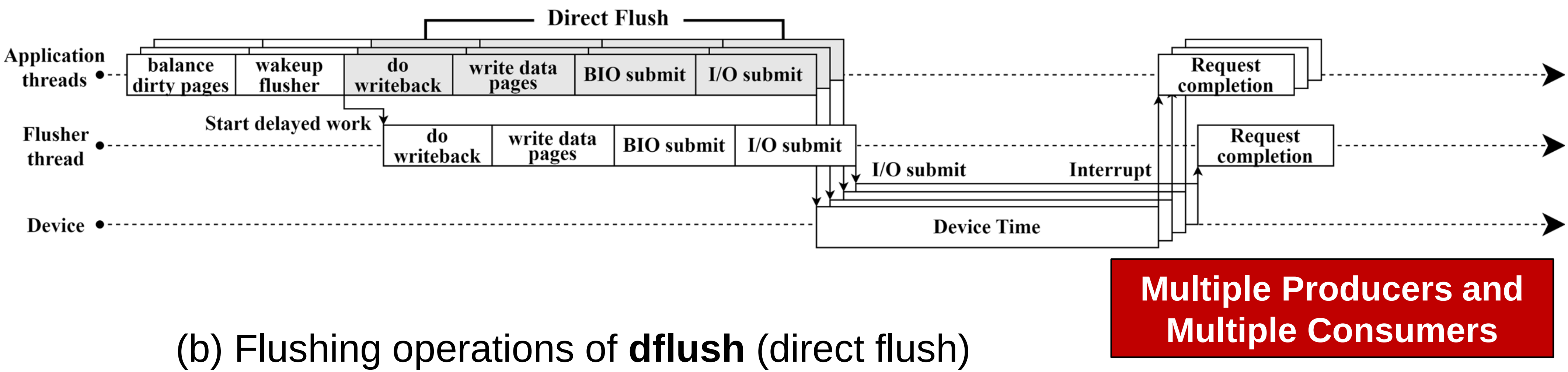
Direct flush

❖ A direct dirty page flush – *dflush*

- One(thread)-to-one(inode) model



(a) Existing flushing operation



(b) Flushing operations of **dflush** (direct flush)

Evaluation

❖ Experimental Setup

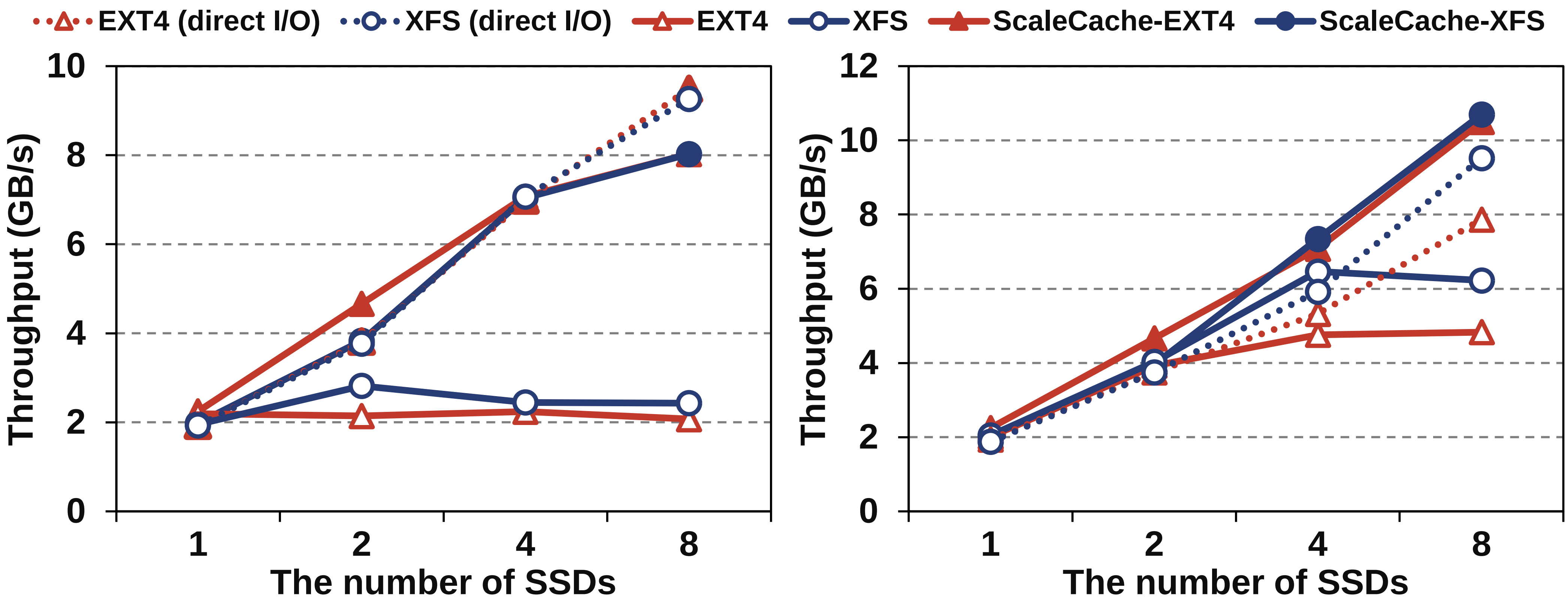
OS	Ubuntu 20.04 LTS
Base kernel	Linux 5.4.147
CPU	4 x Intel Xeon Gold 6242 (totally 64 physical cores, HT disabled)
Memory	DDR4 64GB
SSD	8 x Intel Optane 900P (NVMe, 2GB/s stable read/write)

❖ Workloads

- Micro-benchmark: FIO benchmark
- Macro-benchmarks
  - Filebench workloads: Fileserver, Varmail and Videoserver
  - FFSB
- Real-world Application: YCSB on RocksDB

Evaluation: Micro-benchmark

- Random and sequential writes w/ various # of SSDs
  - **FIO workload**: 64 threads, 3GB file size per thread, 4KB request size, QD=1
  - **RAID Setup**: RAID-0, 512KB stripe size, # of SSDs varies
  - **Improvement:** 3.87x and 3.30x compared with EXT4 and XFS

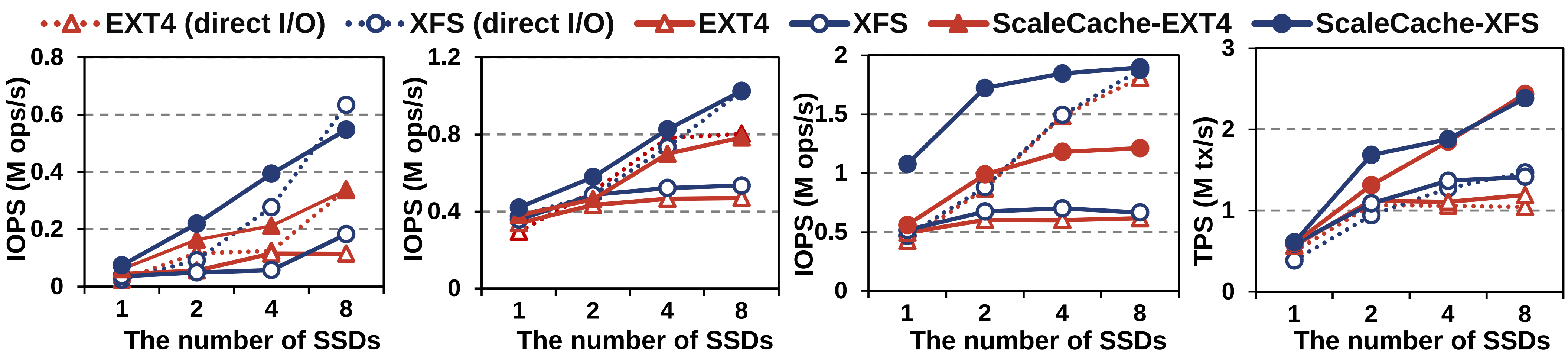


(a) Random write

(b) Sequential write

Evaluation: Macro-benchmark

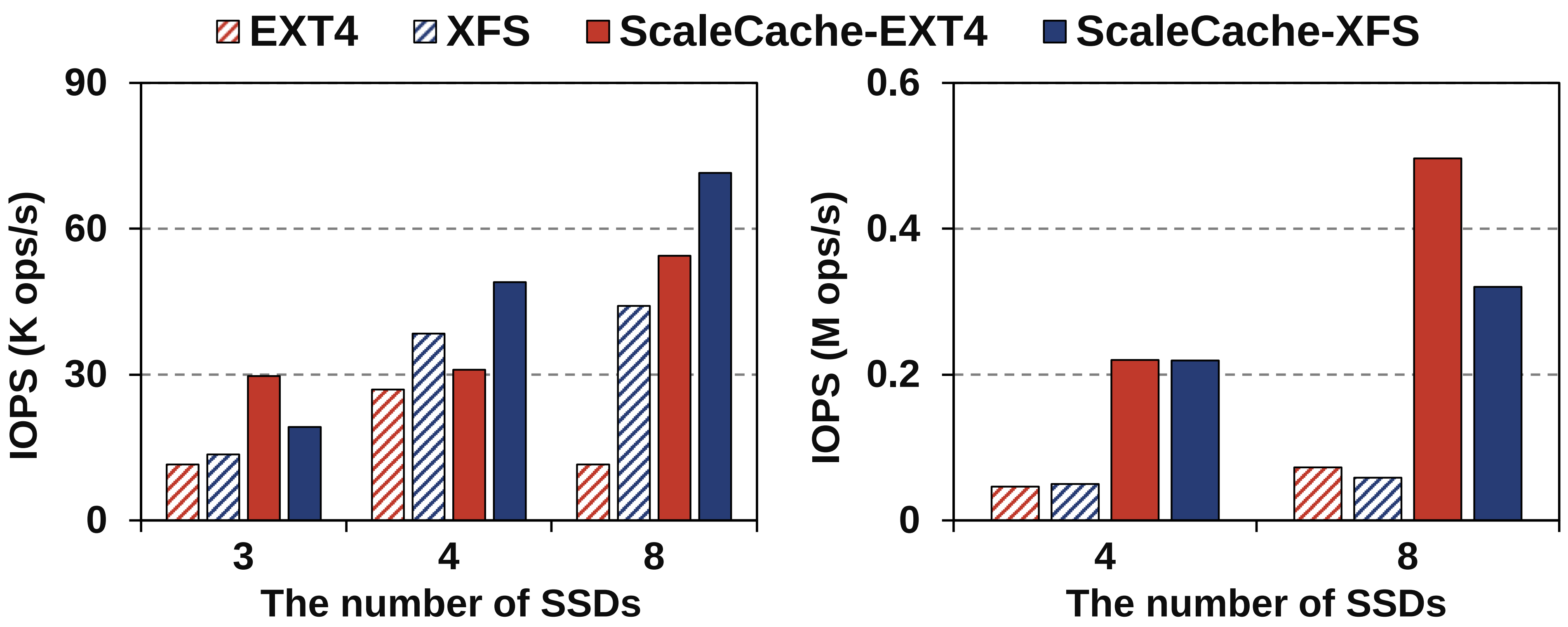
- **With various number of SSDs**
  - **Benchmarks:**
    - **Filebench** (fileserver, varmail, and videoserver workloads)
    - Flexible filesystem benchmark (**FFSB**)
  - **Workload:** 64 threads, 64 files, 3GB file size, 4KB request
  - **RAID Setup**: RAID-0, 512KB stripe size, #SSDs varies
  - **Improvement:** 6.81x (fileserver), 1.92x (varmail), 2.85x (videoserver), 2.04x (FFSB)



(a) Fileserver (b) Varmail (c) Videoserver (d) FFSB

Evaluation: Macro-benchmark

- Various RAID level configurations
  - **Fileserver workload**: 64 threads, 64 files, 3GB file size, 4KB request size
  - **RAID Setup**: RAID-5 and RAID-10, 512KB stripe size each, # of SSDs varies

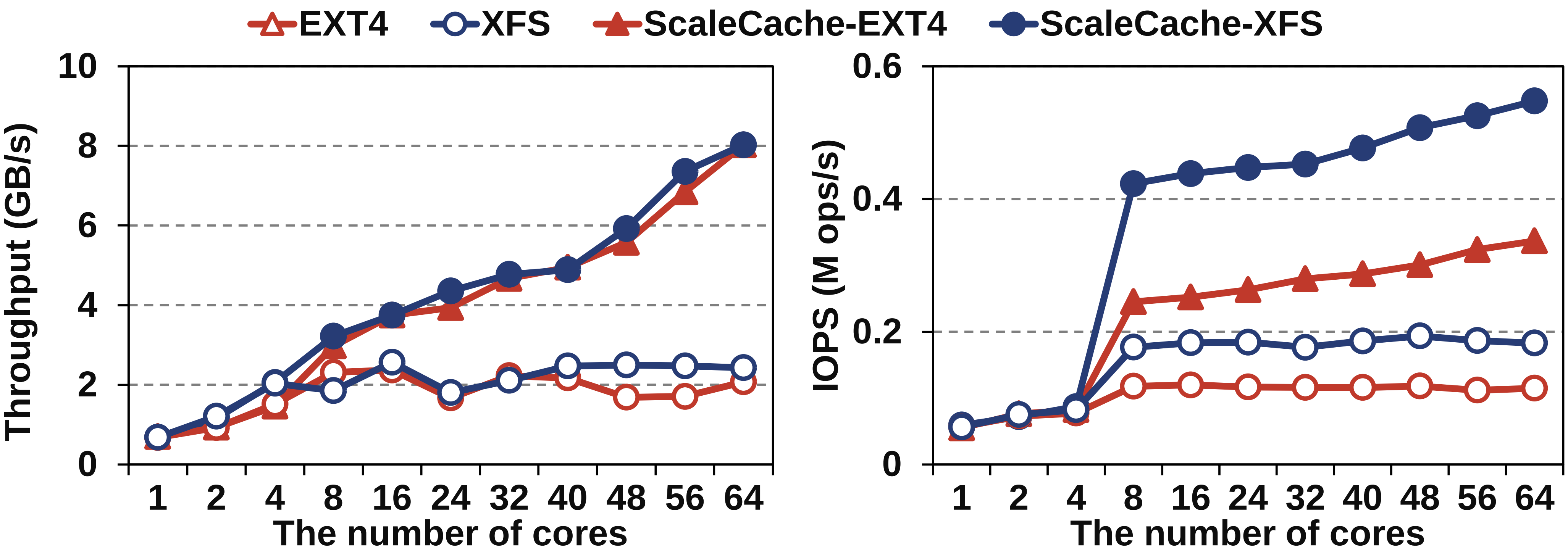


(a) RAID-5 (b) RAID-10

Evaluation: Core Scalability

❖ **Various number of CPU cores**

- **FIO Workload**: 64 threads, 3GB file size per thread, 4KB request size, random write, QD=1
- **Fileserver workload**: 64 threads, 64 files, 3GB file size, 4KB request size
- **RAID Setup**: RAID-0 with 8 SSDs, 512KB stripe size

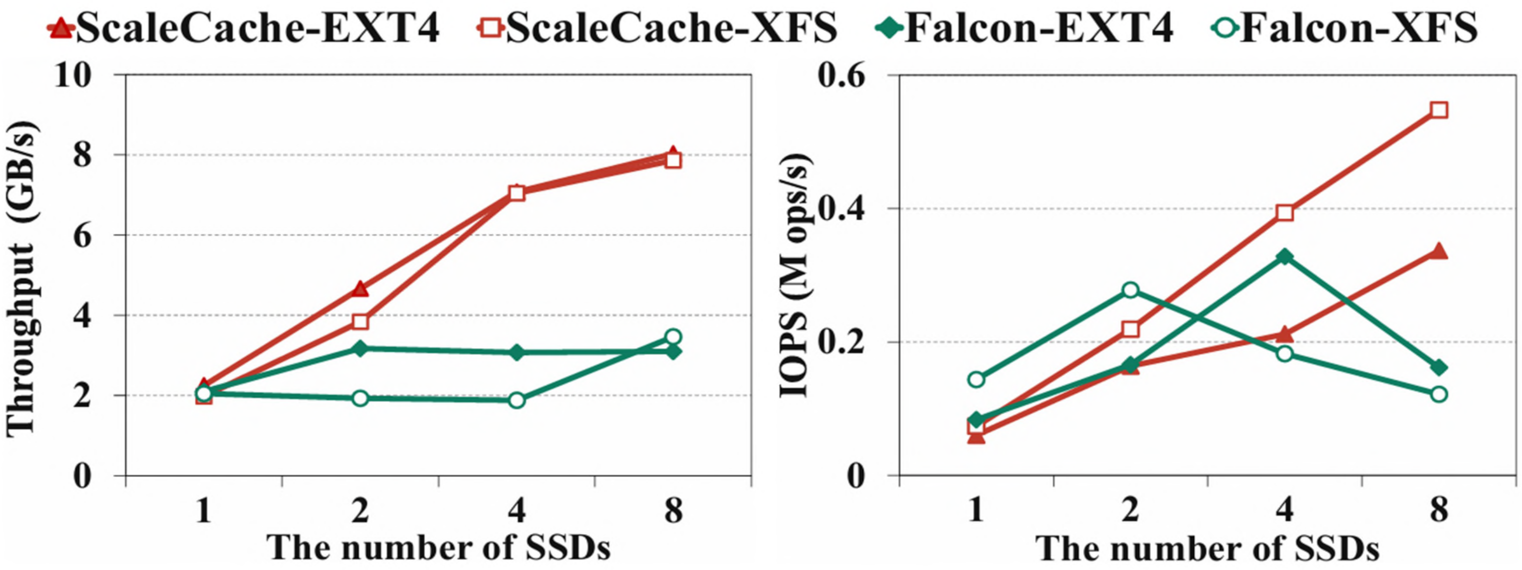


(a) FIO results

(b) Fileserver results

Evaluation: Comparing with a Scalable Scheme

- **Falcon (ATC'17)**
  - A scalable block layer scheme for multiple SSDs
    - Falcon parallelizes I/O operations in the block layer for multiple SSDs using per-drive I/O processing
      - ✓ Only one flusher thread in the page cache when balancing dirty pages
      - ✓ The lock-based XArray limits the concurrency of the page cache
  - **Improvement:** 2.59x (FIO), 4.5x (fileserver)



(a) FIO results (b) Fileserver results

Conclusion

- **ScaleCache** consists of two synergistic components**:**
  - ***ccXArray***: enables concurrent access to the data structure of the page cache
  - ***dflush***: presents a direct page flush in a parallel and opportunistic manner
- **ScaleCache outperforms**
  - Linux page cache by up to 6.81×
  - Existing scalable scheme by up to 4.50×

- **Please refer to the paper for further details**
  - https://dl.acm.org/doi/abs/10.1145/3627703.3629588
- **ScaleCache** is open source now:
  - https://github.com/syslab-cau/ScaleCache

Q&A

Thank you for your attention