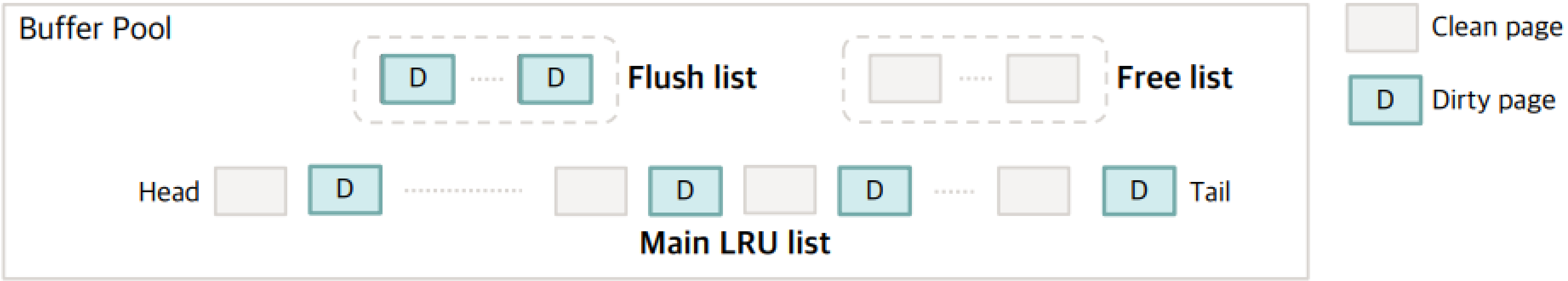
Buffer Management of MySQL/InnoDB (2) Write Requests Part 1

Bo-Hyun Lee
lia323@skku.edu

Index

- Three Types of Disk Write
- Single Page Flush
- Page Cleaner Thread and LRU Tail Flush
- How to Monitor Disk Writes

Lists of Buffer Blocks



- **Free list**
  - Contains free (empty) buffer frames
- **LRU list**
  - Contains all the blocks holding a file page
- **Flush list**
  - Contains the blocks holding file pages that have been modified in the memory but not written to disk yet (i.e., **dirty**)

Disk Read & Write

- Disk read:
  - Reading file page from the disk <u>upon buffer miss</u>

- Disk write:
  - Activity of writing **dirty pages** in memory to disk
  - InnoDB has limited space in the buffer pool and redo log
  - Thus, InnoDB tries *to avoid synchronous write* by:
    - ✓ Securing free buffer frames in the free list or using clean pages for replacement
    - ✓ Flushing dirty pages continually

When do We have to Perform Disk Write? 

- When page is evicted from buffer pool:
  - The memory version of the page and disk version is different
  - So, we need to reflect the changes to the disk

- In case of unexpected failure, we cannot lose all changes
  - But the data memory is gone!
  - Thus, the data in **redo logs** will be read only in the case of recovery
  - During recovery, the modified pages will be reconstructed with the latest version of the page based on redo log

Three Types of Disk Writes

- **Single Page Flush:**
  - A single write request issued by the foreground user process
  - Used as a victim for replacement
- **LRU Tail Flush:**
  - Asynchronous write requests issued by the background process
  - For cold page eviction
- **Checkpointing:**
  - Asynchronous write requests issued by the background process
  - For database recovery upon failure

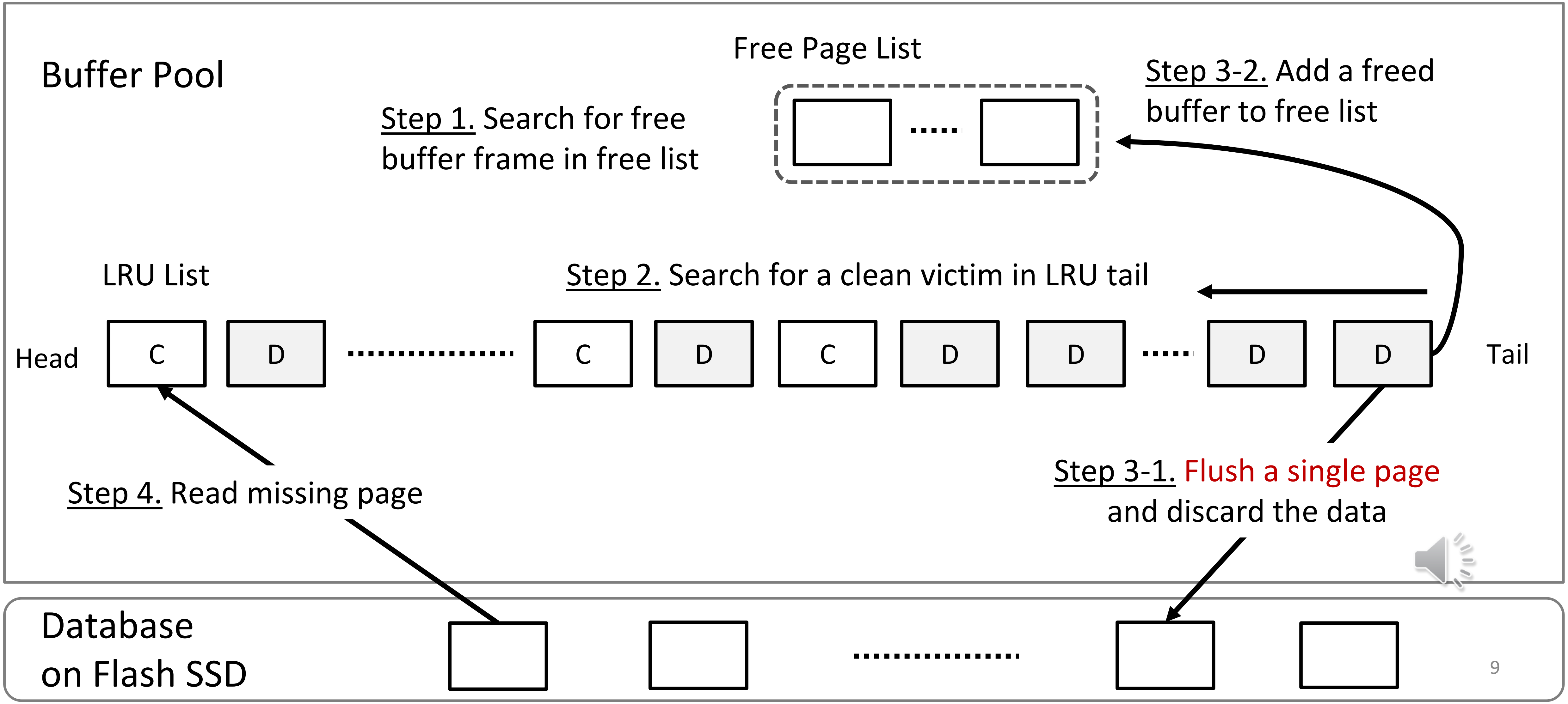
Challenges of Flushing

- If flushing occurs too aggressively,
  - It can result in degraded transaction throughput
- If flushing occurs too lazily,
  - We can run out of redo log space
  - Recovery time increases upon system failure
- Balancing the degree of flushing is important!

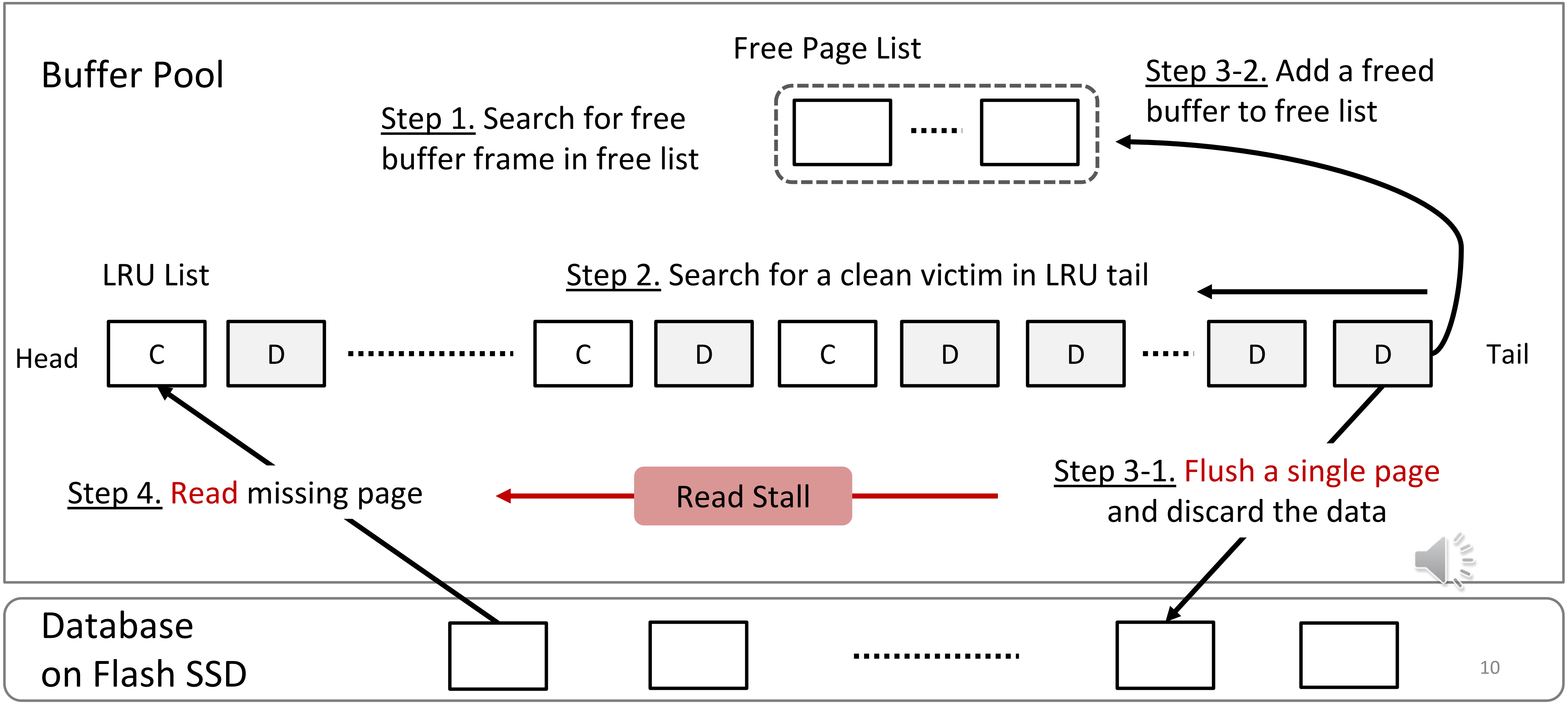
Disk write type 1:

Single Page Flush

Recall Victim Selection Upon Buffer Miss



Recall Victim Selection Upon Buffer Miss



LRU Get Free Block: Single Page Flush

- buf/buf0lru.cc: buf_LRU_get_free_block()

```
if (!buf_flush_single_page_from_LRU(buf_pool)) {
    MONITOR_INC(MONITOR_LRU_SINGLE_FLUSH_FAILURE_COUNT);
    ++flush_failures;
}

srv_stats.buf_pool_wait_free.add(n_iterations, 1);

n_iterations++;

goto loop;
```

If we failed to find a free buffer frame or a clean page, do a single page flushing

LRU Get Free Block: Single Page Flush

- `buf/buf0flu.cc: buf_flush_single_page_from_LRU()`

Start iteration from the last LRU page until it succeeds single page flush

Full scan

```
for (bpage = buf_pool->single_scan_itr.start(), scanned = 0,
     freed = false;
     bpage != NULL;
     ++scanned, bpage = buf_pool->single_scan_itr.get()) {

    ut_ad(buf_pool_mutex_own(buf_pool));

    buf_page_t* prev = UT_LIST_GET_PREV(LRU, bpage);

    buf_pool->single_scan_itr.set(prev);

    BPageMutex* block_mutex;

    block_mutex = buf_page_get_mutex(bpage);

    mutex_enter(block_mutex);
```

LRU Get Free Block: Single Page Flush

- `buf/buf0flu.cc: buf_flush_single_page_from_LRU()`

```
if (buf_flush_ready_for_replace(bpage)) {
    /* block is ready for eviction i.e., it is
    clean and is not IO-fixed or buffer fixed. */
    mutex_exit(block_mutex);

    if (buf_LRU_free_page(bpage, true)) {
        buf_pool_mutex_exit(buf_pool);
        freed = true;
        break;
    }
```

Check whether the current bpage can be immediately used for replacement without flushing

If so, we just simply discard the data inside bpage and return the emptied frame into the free list

LRU Get Free Block: Single Page Flush

- `buf/buf0flu.cc: buf_flush_single_page_from_LRU()`

```
} else if (buf_flush_ready_for_flush(
                bpage, BUF_FLUSH_SINGLE_PAGE)) {

        /* Block is ready for flush. Try and dispatch an IO
        request. We'll put it on free list in IO completion
        routine if it is not buffer fixed. The following call
        will release the buffer pool and block mutex.

                                                 is page has actually
                                                 lushed to disk */

        freed = buf_flush_page(
            buf_pool, bpage, BUF_FLUSH_SINGLE_PAGE, true);

        if (freed) {
            break;
        }

        mutex_exit(block_mutex);
```

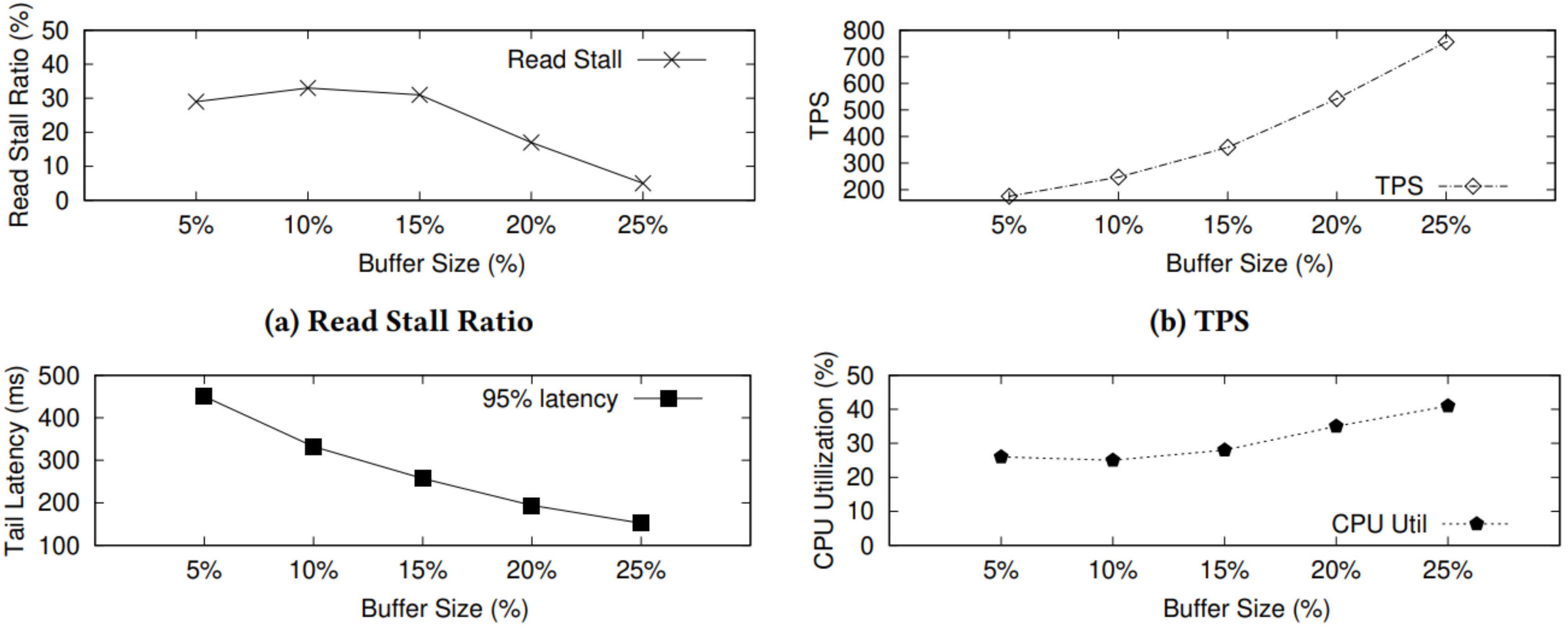
Check whether the current bpage can be flushed

If so, try to flush it to the disk

Synchronous write (true)

Effect of Single Page Flush on Performance

- When read stall caused by single page flush occurs frequently,
  - **TPS**(Transactions Per Second) and **tail latency** worsens
  - Not able to fully utilize CPU and internal parallelism of SSD



(a) Read Stall Ratio

(b) TPS

(c) Tail Latency

(d) CPU Utilization

TPC-C Evaluation on MySQL/InnoDB

Why does Single Page Flush Aggravate Performance?

- From the perspective of foreground user process, read I/O request is stalled until flush completes
  - If a victim is found via Step 1 or Step 2, it would take only about several hundred microseconds**(e.g., 500µs)**…
  - However, when a victim is replaced by Step 3, then it would take few milliseconds **(e.g., 5ms, which is 1,000 times more!!)**
  - In other words, if a single page flush is avoided, we can process 1,000 more read operations

- Also, it *blocks other I/O requests* issued by other foreground processes and page cleaner threads due to the buffer pool mutex acquisition

Disk write type 2:

LRU Tail Flush

Page Cleaner Thread(s)

- Background thread that *asynchronously* writes dirty pages to the disk
  - Wakes up once per second to perform two types of flushing
  - **LRU tail flush:** Flushes pages from the LRU tail
  - **Checkpointing:** Flushes pages from the flush list

- Asynchronous write vs. synchronous write
  - Synchronous write:
    - ✓ Process that performs synchronous write <u>waits until I/O request completes</u>
    - ✓ e.g., single page flush
  - Asynchronous write:
    - ✓ Process that performs asynchronous doesn't have to wait until I/O request completes
    - ✓ Usually performed by the background thread
    - ✓ e.g., LRU tail flush and checkpointing

Page Cleaner Thread(s)

- `buf/buf0flu.cc: buf_flush_page_cleaner_coordinator()`

```
extern "C"
os_thread_ret_t
DECLARE_THREAD(buf_flush_page_cleaner_coordinator)(
/*===========================================*/
	void*	arg MY_ATTRIBUTE((unused)))
			/*!< in: a dummy parameter required by
			os_thread_create */
{
	ib_time_monotonic_t next_loop_time = ut_time_monotonic_ms() + 1000;
	ulint	n_flushed = 0;
	ulint	last_activity = srv_get_activity_count();
	ulint	last_pages = 0;

	my_thread_init();
```

Wakes up every 1000ms

Page Cleaner Thread(s)

- `buf/buf0flu.cc: buf_flush_page_cleaner_coordinator()`

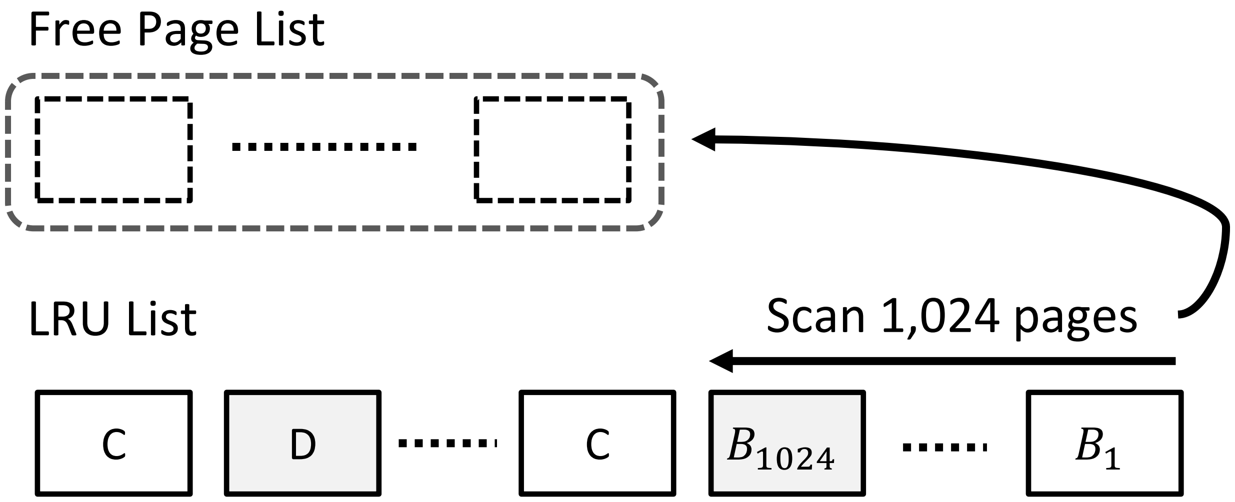
```
switch (recv_sys->flush_type) {
case BUF_FLUSH_LRU:
    /* Flush pages from end of LRU if required */
    pc_request(0, LSN_MAX);
    while (pc_flush_slot() > 0) {}
    pc_wait_finished(&n_flushed_lru, &n_flushed_list);
    break;

case BUF_FLUSH_LIST:
    /* Flush all pages */
    do {
        pc_request(ULINT_MAX, LSN_MAX);
        while (pc_flush_slot() > 0) {}
    } while (!pc_wait_finished(&n_flushed_lru,
                    &n_flushed_list));
    break;
```

LRU tail flush

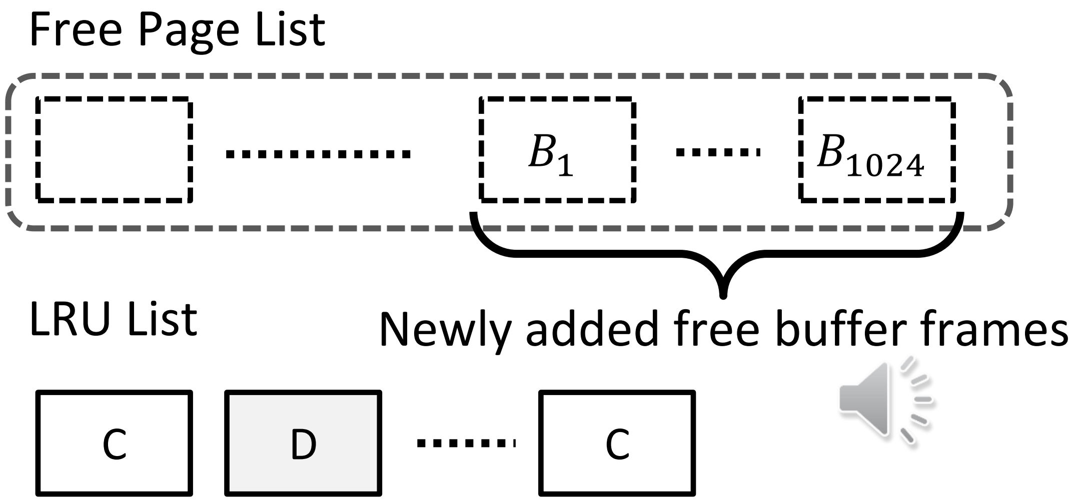
Flush list flush

LRU Tail Flush

- LRU Tail Flush:
  - **Evicting/flushing pages from LRU list and returning them to the free list**
  - Purpose is to evict cold pages to produce free buffer frames for page read
  - Write pages in a *batch* up to `innodb_LRU_scan_depth` (default: 1024 pages)



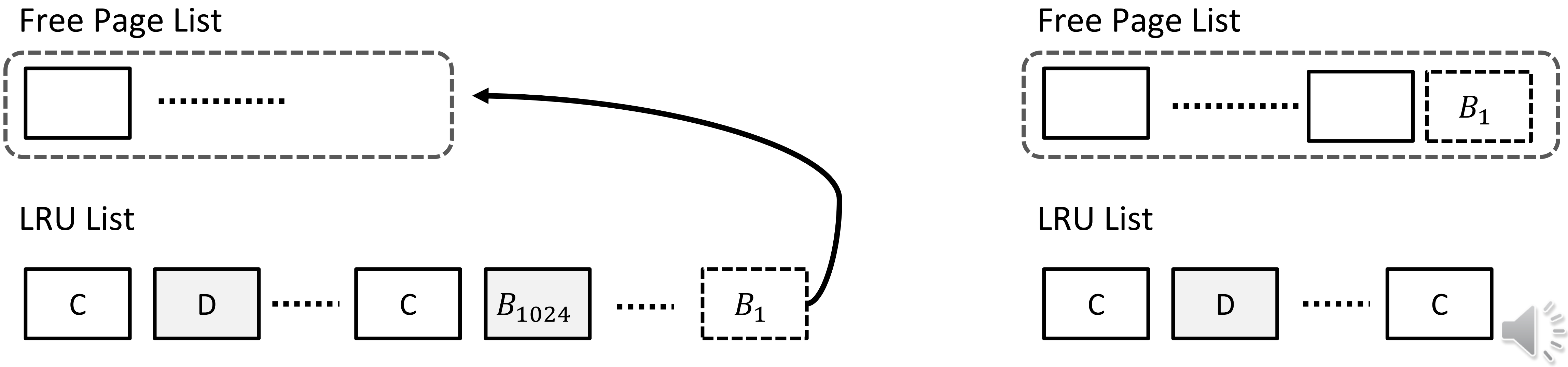
Before LRU Tail Flush



After LRU Tail Flush

LRU Tail Flush

- If the current bpage is **clean**:
  - Step (1): simply discard the data and evict from the LRU list
  - Step (2): return a free buffer frame to the free page list

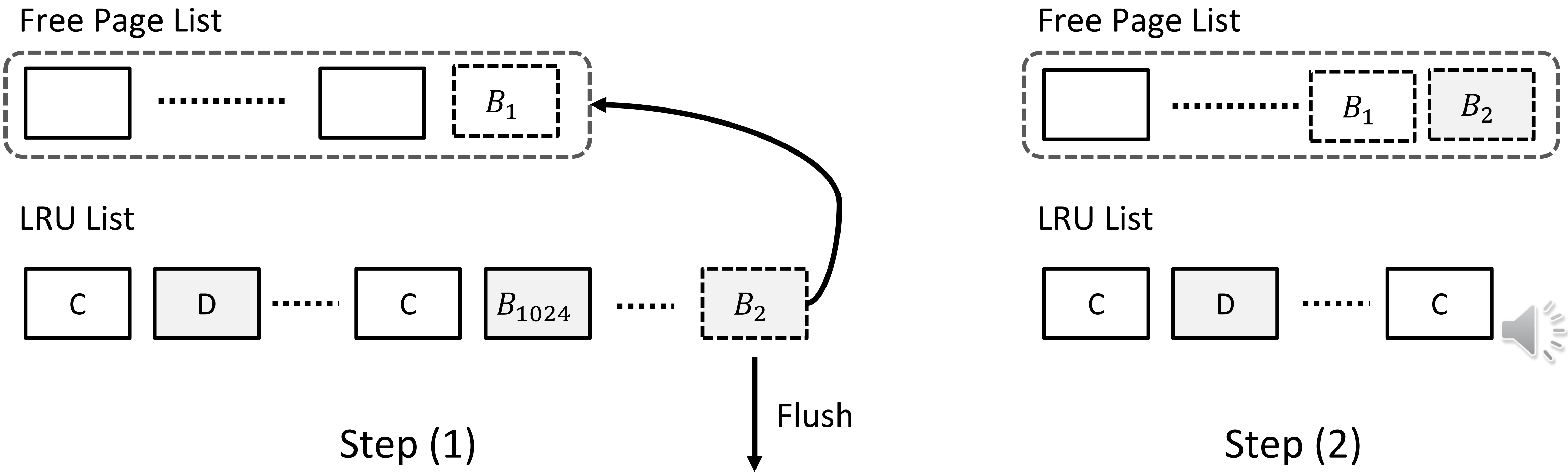


Step (1)

Step (2)

LRU Tail Flush

- If the current bpage is **dirty**:
  - Step (1): flush a page and evict from the LRU list
  - Step (2): return a free buffer frame to the free page list



LRU Tail Flush

- buf/buf0flu.cc: buf_flush_LRU_list_batch()

```
for (bpage = UT_LIST_GET_LAST(buf_pool->LRU);
     bpage != NULL && count + evict_count < max
     && free_len < srv_LRU_scan_depth + withdraw_depth
     && lru_len > BUF_LRU_MIN_LEN;
     ++scanned,
     bpage = buf_pool->lru_hp.get()) {

    buf_page_t* prev = UT_LIST_GET_PREV(LRU, bpage);
    buf_pool->lru_hp.set(prev);

    BPageMutex* block_mutex = buf_page_get_mutex(bpage);

    mutex_enter(block_mutex);

    if (buf_flush_ready_for_replace(bpage)) {
        /* block is ready for eviction i.e., it is
        clean and is not IO-fixed or buffer fixed. */
        mutex_exit(block_mutex);
        if (buf_LRU_free_page(bpage, true)) {
            ++evict_count;
        }
```

Get the last bpage of LRU

Check whether the current bpage can be immediately used for replacement without flushing

If so, we just simply discard the data inside bpage and return the emptied frame into the free list

LRU Tail Flush

- `buf/buf0flu.cc: buf_flush_LRU_list_batch()`

```
} else if (buf_flush_ready_for_flush(bpage, BUF_FLUSH_LRU)) {
    /* Block is ready for flush. Dispatch an IO
    request. The IO helper thread will put it on
    free list in IO completion routine. */
    mutex_exit(block_mutex);
    buf_flush_page_and_try_neighbors(
        bpage, BUF_FLUSH_LRU, max, &count);
} else {
    /* Can't evict or dispatch this block. Go to
    previous. */
    ut_ad(buf_pool->lru_hp.is_hp(prev));
    mutex_exit(block_mutex);
}
```

Check whether the current bpage can be flushed

If so, try to flush it to the disk asynchronously

Relation between LRU Flush and Single Page Flush

- Fundamentally, why does a single page flush occur? 🤔
  - When using flash SSDs, *read-write asymmetry* exists
  - Read speed is much (e.g., 5 to 10.4 times) faster than the write

- How does this affect single page flush?
  - Foreground processes *consume* free buffer frames with read speed
  - Page cleaner threads *produce* free buffer frames with write speed (LRU tail flush)
  - Thus, single page flush occurs very frequently <u>when using flash SSDs</u>

I/O Speed Asymmetry in Storage Devices

Storage Device	Random IOPS (16KB) Read	Random IOPS (16KB) Write	Asym. Ratio (*Read*/*Write*)
SSD-A†	190,622	33,866	5.6
SSD-B¶	169,681	33,811	5.0
SSD-C◇	39,860	3,830	10.4
SSD-D*	80,433	12,536	6.4
HDD♯	280	216	1.3

† Intel P4101 NVMe SSD 1TB, ¶ Samsung 970Pro NVMe SSD 512GB, ◇ Micron Crucial MX500 250GB, * WD Blue SN570 500GB, ♯Western Digital WD10EZEX 1TB

Quantifying the Cost of a Single Page Flush

- This week, you will quantify the cost of a single page flush
- You will first add some codes to measure the wait time of a single page flush
- After running TPC-C with modified MySQL, calculate the average time and the total time it took to perform single page flushes
- Refer to week4 for the experiment guide
  https://github.com/LeeBohyun/SWE3033-S2023

References

[1] Bryan Harris and Nihat Altiparmak. 2020. Ultra-low latency SSDs' impact on overall energy efficiency. In Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'20). USENIX Association, USA, Article 2, 2.

[2] MySQL document: https://dev.mysql.com

[3] Mijin An, MySQL Buffer Management, https://www.slideshare.net/meeeejin/mysql-buffer-management

[4] An et.al., “Avoiding Read Stalls on Flash Storage”, SIGMOD2022, https://dl.acm.org/doi/pdf/10.1145/3514221.3526126