On-Flash, In-Kernel Key-Value Store

Project 6
ECE 5984 Linux Kernel Programming
Spring 2017

Ho-Ren Chuang, Christopher Jelesnianski, Beichen Liu, Mincheol Sung

Introduction

In this project, we were given the sources of a prototype implementation for a non-volatile storage system in the kernel (ver. 4.0.9). The system is a simple key-value pair store. The prototype came with basic functionality and serious limitations allowing us to build upon it and implement features conventionally found and relied on in a typical storage system that are missing in the prototype. The storage system prototype also lacked in overall performance and scalability. We were tasked with addressing these issues, solving the one main limitation of this prototype followed by choosing two secondary limitations. For the secondary limitations, the class together came up with a long list of limitations that could feasibly be addressed in the time allotted for this project after having studied the code. Specifically, the following questions were posed:

- *What are the usual functionalities of a storage system that are absent in the prototype?*
- *What is hindering the performance and the scalability of the prototype?*

Limitations Solved

In this project, our group decided to solve the following limitations:

- Limitation 1: (Mandatory) The current system cannot update a key with another value, nor can the current system delete a key/value pair previously written into the flash disk.
- Limitation 2: Read Latency does not scale. The system currently needs to scan the entire flash storage system to find a value. This affects mount time and writing as the storage system first needs to check to ensure a duplicate key is not being inserted.
- Limitation 3: The system does not employ any sort of wear-leveling scheme.
- Limitation 4 (BONUS): The current system does not utilize any concurrency for better overall performance.

To solve all the above limitations, our project went through three distinct phases as we solved the limitations one-by-one. To allow reading this report to be a little easier, we explain them here. In the first phase (Implementation 1), our flash storage project used 2 separate partitions; one just for data and the other only for metadata. In the first phase all logic was serial for easy debugging. In the second phase (Implementation 2), our implementation removed the metadata partition and now all information about the flash storage system was held within a single partition (metadata among side data blocks) as is found in a typical flash drive BUT all logic was STILL serial. Finally, in our third and last implementation phase, we upgraded the single partition with serial operations to be parallelized. By parallelizing our implementation, we could tackle the last limitation, introducing timer interrupts and adequate locking.

This paper is broken up as follows: we go over important design decisions for each limitation and including justification for each design choice (Design). We also present schematics to visually explain the layout with the inclusion of the features and small interaction diagrams to show how our mechanisms work also in this section. We then go over crucial functions that were implemented for each of the

discussed features and how they fit into the overall improved infrastructure (Implementation). We also designed unit testing for each of our features to showcase their performance when compared to the original prototype (Validation Process). Finally, we discuss future work and the avenues that would be worth exploring if more time was given for this project (Future Work) and conclude (Conclusion).

Design

In this section we go over each of the limitations, explain how our group approached each, explaining design decisions and

- Limitation 1: (Mandatory) The current system cannot update a key with another value, nor can the current system delete a key/value pair previously written into the flash disk

As flash memory data cannot simply be *re-written*, the only way to update a key with another value would be to find the current key/value pair within the storage system, invalidate it, and write a new entry for the same key into the storage system. The set_keyval() function was updated to automatically recognize the differences when a new key is being entered into the storage system versus when the key already exists within the system. This can be accomplished by a simple search before performing any operations to know whether the set is a writing a new key or updating an existing one. In the case it is an update, the set_keyval() goes through one additional step to invalidate the page containing the old key/value pair, before continuing on with the original write operation. By default, it is assumed that when a user sets a key/value pair of a key that is already registered, the user wants to update that key, and the old value is discarded.

To handle strict deleting of a key/value pair on the storage system, it is much simpler than a write or update. Since the prototype only initially had the write feature, we needed to add a delete operation. To function properly, the delete operation first searches a hash table (explained in limitation: read latency) for the given key. If the desired key is found, it is checked if it has already been marked for garbage collection (explained below), if not, it invalidates the key via our invalid_pg() function, records the hash table index of the deleted key, and finally returns the index. If it has already been marked for deletion, the delete function returns an error value. This extra check is in place so that if the user did not remember whether a key has been deleted and goes to try delete the key again, no actual data needs to be tied up for this operation.

Now that individual key/value pairs can be deleted, this should allow the storage system to store more values throughout time. In the given prototype implementation, once a value was written to the storage system, it could not be deleted or updated until the entire disk was formatted, limiting the amount of key/value pairs that could be effectively kept track of by the storage system. With the addition of update and delete key functions, this can now be avoided, but there needs to exist a mechanism able to recognize when a block contains mostly stale or unneeded key/value data. For this reason, a garbage collection scheme needs to additionally be implemented.

Garbage Collecting Algorithm:

Serial (Phase 2)

In phase 2, we implemented a serialized version of key value store which GC and flushes metadata sequentially after writing. This serialization helps to improve scalability. Theoretically, read or write takes O(1) irrespective of data size. It is also following first page header scheme (explained in limitation 2). The only difference between the one explained below is that the tag that signifies data blocks or metadata blocks. In Phase 2, the tag is “1990” which are integers. For each write, it checks whether there is a block whose number of invalid pages are over the GC threshold and if so, do GC. After GC, it also checks to flush metadata.

```
set_keyval(key, val)
{
      /* actual write */
      write(index, buffer);

      gc_check();

      time_now = gettime();
      time_elapsed = time_now - time_flush;
      if (time_elapsed >= 1)
            flush_metadata();
}
```

The serial implementation maintains a time value containing time of the last flush. If flush occurs more than 1 sec before, it will flush metadata. Otherwise, metadata not be flushed. Thanks to this scheme, we can get performance improvement by flushing on every write in case of sequential writes. However, it does have a limitation. When we write multiple key/value pairs in one instant but nothing afterwards, a flush will not be triggered until the next write. For this reason, it is possible to lose all the data written in this instant in case of a power cut. The only other way flushing occurs in the serial implementation is when we unmount the flash storage system module. This limitation can be solved by introducing a parallelized implementation of garbage collection.

Parallel (Phase 3)

Whenever the timer interrupt goes off for garbage collection is initiated. This interval defined in the core framework and can be adjusted to a shorter or longer interval as needed. As soon as garbage collection begins, the coarse-grained lock mentioned earlier is secured before moving forward. Below is an illustration showing the garbage collection process. After obtaining the lock, the garbage collection iterates through the entire disk looking for a prime candidate block to rewrite somewhere else. The candidate is selected upon 1 primary condition. If a block has majority of its blocks free, but containing stale data (as marked by invalid_pg()), it would good to renew the block and move the valid pages to another block with space for the data still currently valid in the candidate block. The worst case scenario is also considered: If no other blocks are free OR have enough free space to hold the candidate's valid data pages, the garbage collector can still nominate the current block being looked at to write back to itself, although this is avoided if possible to prevent uneven wear-leveling. It is also entirely possible that no suitable candidates are found, and thus garbage collection is aborted until the next timer interrupt. After a candidate has been found (1.), a target block needs to found to write the current valid data in the candidate block to. When looking for a target block, garbage collection primarily considers a blocks worn counter information will select the one with the lowest counter to help with wear-leveling; garbage collection will also try to find a block that is capable of holding all the valid data (1 primary reason simplicity in the algorithm). After a target block has been found, the candidate's valid pages are all copied into a temporary buffer, their hash table hashes are updated for the new target block, and the candidate block is erased (2.). Finally, the valid data is written to the target block on disk (3.) and the blocks meta data is updated to have the new key/value pairs as well. Additionally, if a free block was used as the target, the block is now marked as in use.

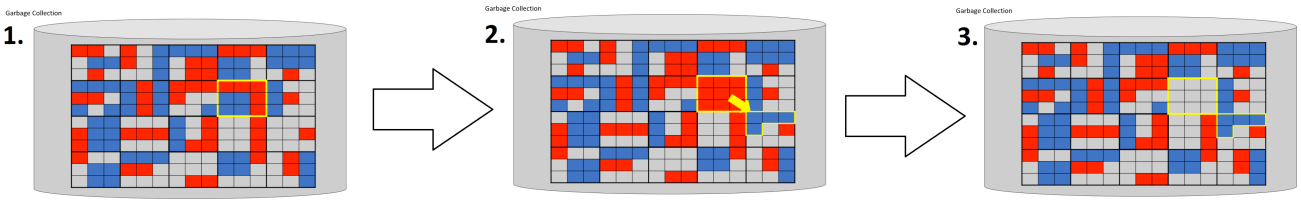


Figure 1. Garbage collection algorithm

Before moving on to the next limitation, we illustrate the operations supported in our implementation, and how they interact with the hash table and disk.

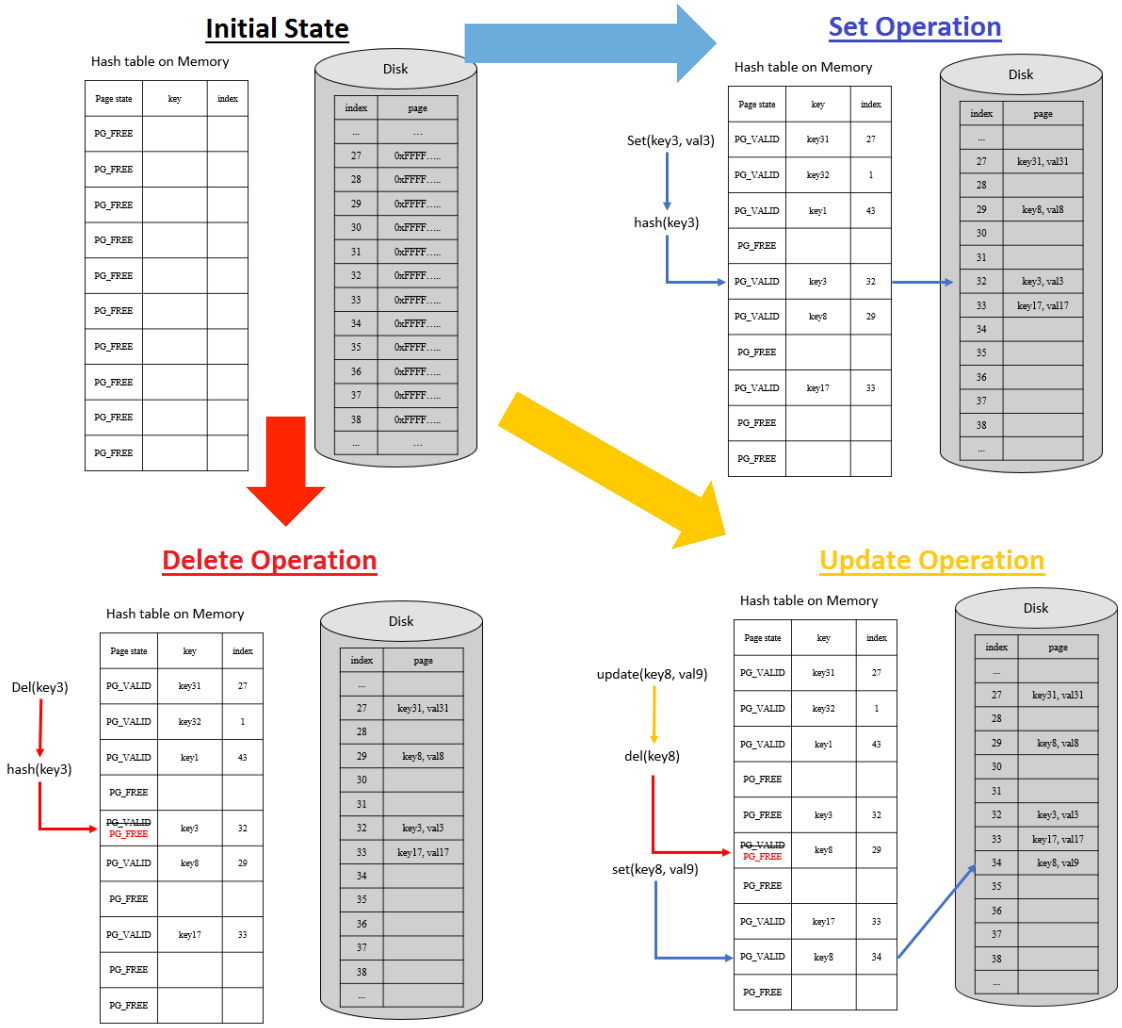


Figure 2. Interaction between hash table and disk operations

- Limitation 2: Read Latency does not scale. The system currently needs to scan the entire flash storage system to find a value. This affects mount time and writing as the storage system first needs to check to ensure a duplicate key is not being inserted.

In the given prototype implementation, the read latency was dependent on linearly iterating through the entire disk until the desired key was found. This initial approach was also used when checking if a given key already existed on the storage system. For larger storage system capacities, this approach simply does not scale. For this reason, our group threw out that entire implementation and decided to rely on a hash table implementation that utilized metadata. For a small compromise in storage capacity (since the hash table takes up a small amount of space and will also need to be stored on disk), the storage system can lookup values quickly no matter the amount of data.

To start, using a hash table requires implementing all the functionality needed use a hash table such as:

- Making a hash value for given input
- Adding elements to the hash table
- Searching the hash table for a given value

It should be noted that a variety of hash algorithms exist when creating a hash key, for simplicity we researched a little online and found the following, the djb2 algorithm, to be good enough to suite our use case:

---

djb2

this algorithm (k=33) was first reported by dan bernstein many years ago in comp.lang.c. another version of this algorithm (now favored by bernstein) uses xor: `hash(i) = hash(i - 1) * 33 ^ str[i];` the magic of number 33 (why it works better than many other constants, prime or not) has never been adequately explained.

```
unsigned long
hash(unsigned char *str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return hash;
}
```

Code 1. Hash function

In Phase 1, the storage system was split into 2 partitions. The first partition held all data (key/value pairs) information, while the second partition was a lot smaller and only held metadata information about the current state of all data within the first partition.

For the data partition, we used the same block & page scheme as in the prototype:

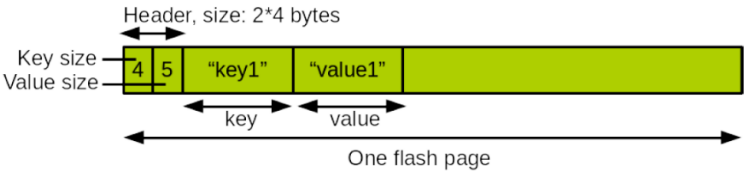


Figure 3. Key/value store scheme

For the metadata partition, we utilized two data structures keeping track of the following Information such as:

- Which blocks are free/used
- Current page offset for a given block indicating where the next page within that block should be written to
- Amount of times the block has been erased (worn count, which is important for effective leveling of the disk)
- Whether a page is free or containing valid data for a key/value pair

A closer look at the metadata partition is below (more information on specifics is in Implementation section)

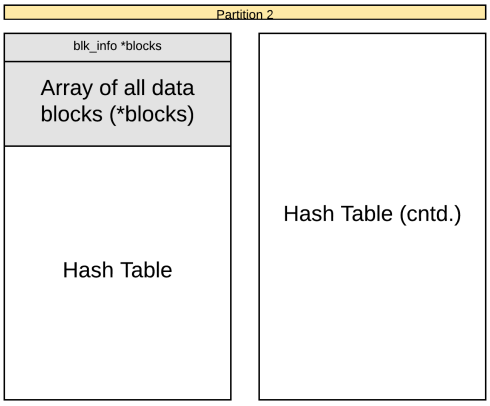


Figure 4. Flash disk partition layout in phase 1

Keeping the data and metadata information separated via different partitions allowed for manageable development of additional features for this project as it allowed us to handle 1 mechanism at a time. A figure of the first implementation is below.

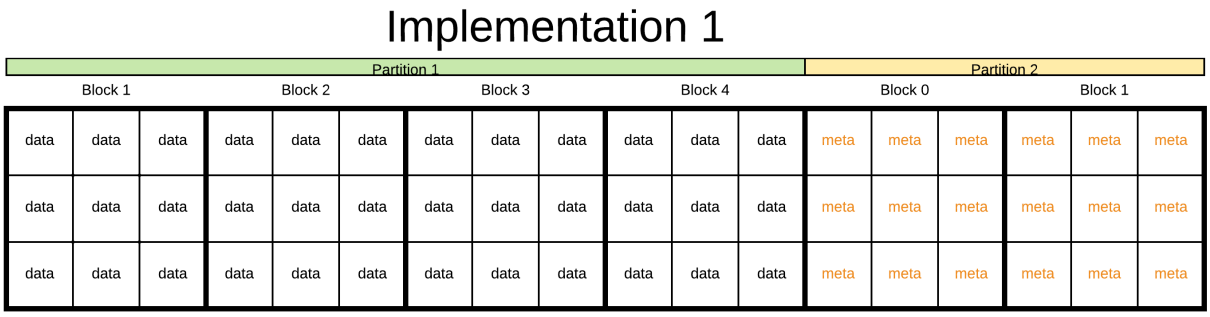


Figure 5. Implementation details in phase 1

In Phase 2, we addressed the hotspot issue by eliminating the second partition completely, allowing metadata blocks to mingle and be written among data blocks. This lessened the hotspot issue occurring the in second partition but does not solve it completely, as block assignment is still not completely randomized (refer to Future Work). This iteration required the merging of the two global attribute structures (lkp_kv_cfg and lkp_meta_cfg) responsible for data and metadata, respectively.

With this merge, a new problem arose. How to tell apart which blocks are for data and which are being used for meta data? As it is important not to mix the two kinds of information within the same block. For the given time frame, we felt it would be best to distinguish the two types of blocks within the single partition by having a header for each block; this was accomplished by reserving and utilizing the first page of every block as the header information. Initially, on an empty storage system no block will any header information as all blocks will be designated as BLK_FREE. Whenever new data (either a key/value pair or metadata information) is written to the storage system, before the data is written to disk, a free block is found, and a header creation function is run marking the block for use of data or metadata only! Only after a header has been established can the storage system continue to write its data to disk. Whenever garbage collection is run and a block has been freed for future use, the block is entirely cleared including its header page and will only have information that its state is BLK_FREE once again.

How does our implementation distinguish between a block for data and a block for metadata? We achieve this by applying different magic numbers to the very first page of the given block. The figure below shows the difference when block 44 is marked with a header for data (left) versus when it is marked as a block to be used for metadata (right).

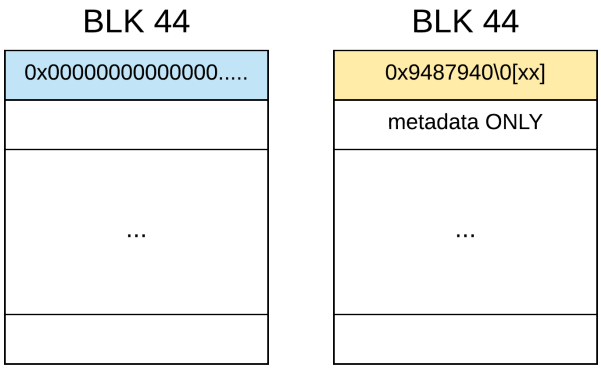


Figure 6. Flash disk layout in phase 2

- 0x000000000000: Signifies a data block
- 0x9487940\0[xx]: Signifies a meta data block, that holds chunk [xx]
    - [xx]: Signifies which chunk of the contiguous memory segment this block holds.

However, metadata block information is a little trickier because for our implementation the entire metadata partition is now split among various blocks, but in reality is a contiguous segment of memory. For this reason, we also need to keep track of the where each chunk was put. Therefore, we add this information as a suffix to our magic number of metadata. This suffix information is used whenever we restart the storage system and need to construct our metadata for the disk in RAM. When recreating the metadata, we simply iterate through the disk looking for metadata specific headers, store the metadata block suffixes into a temporary array, and then refer this array to order our data correctly. Another example of this implementation is below and can demonstrate how it works.

Implementation 2

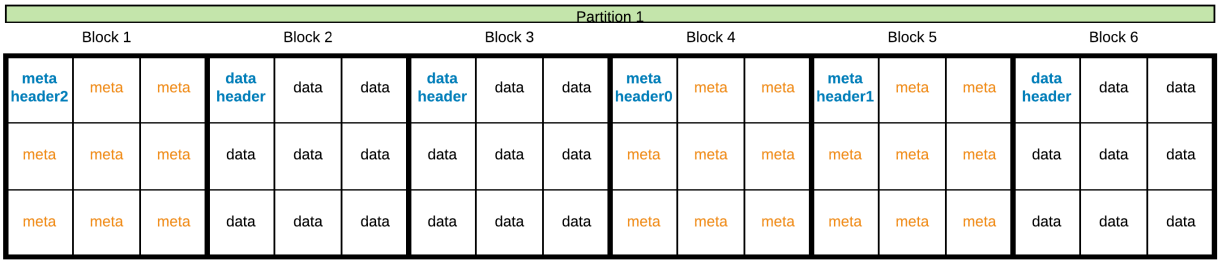


Figure 7. Implementation details in phase 2

Situation: Disk has just been rebooted and currently has this configuration. Time to reconstruct metadata information into RAM.

1. Iterate through all blocks looking for metadata headers, note suffixes and add to an array.
    a. Array initialized to: {-1, -1, -1, -1, -1, -1}
    b. First meta block found {3, -1, -1, -1, -1, -1}
    c. Second metadata block found {3, -1, -1, 0, -1, -1}
    d. Third Metadata block found {3, -1, -1, 0, 1, -1}
2. Order block information properly to buffer.

```
a. For( i = 0, j = 0 ; i < number_blocks; i++){
        If( array[i] == j ){
            memcpy( block[i] ) to buffer;
            j++;
        }
    }
```

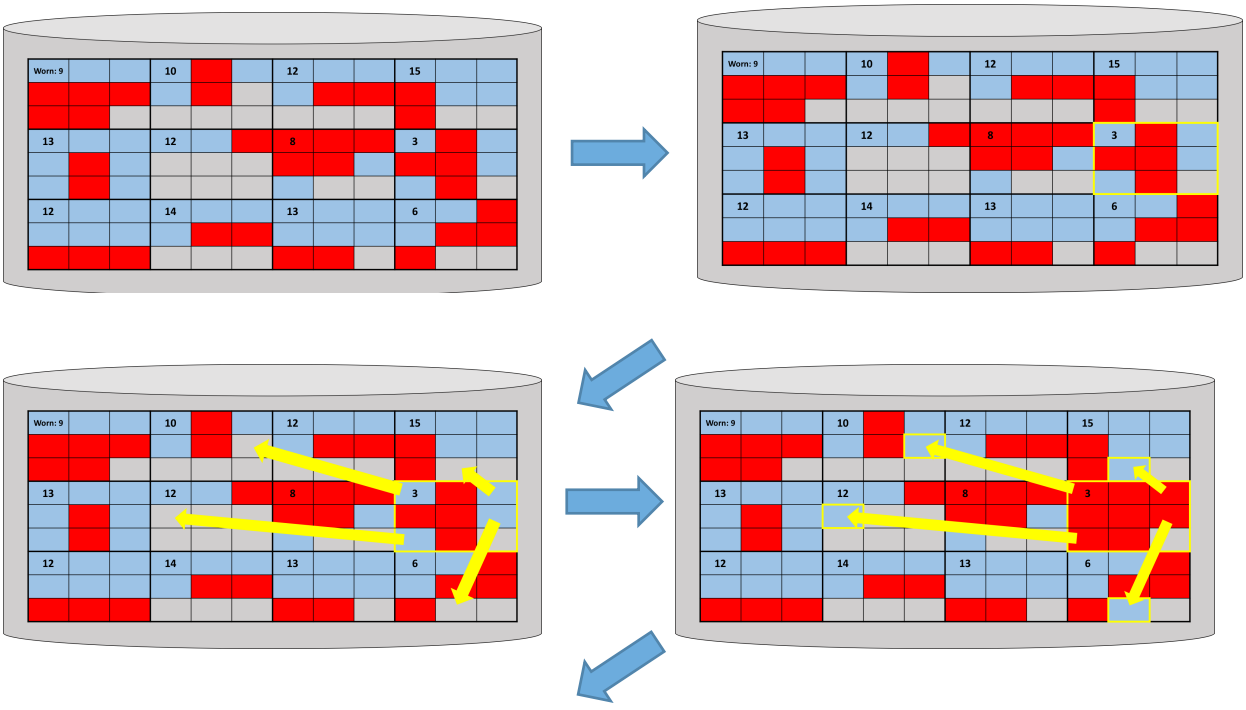
Finally, all that is needed is to save this information to the stack.

- Limitation 3: The system does not employ any sort of wear-leveling scheme.

Throughout the project, wear leveling was the last limitation we addressed, as we would have had to re-implement it every time we updated the base implementation (From phase 1, phase 2, and phase 3). So our wear leveling implementation works as follows.

Serial Version

In Phase 2, when a set_keyval occurs we need to write to disk and this also means the metadata needs to be updated. To begin this process, the least worn bock is chosen (Figure 8.2) for the data to be written to. Since metadata blocks that are flushed repeatedly would become hotspots, it is necessary to balance them among the all the other blocks within the flash storage system. Additionally, every time a flush of metadata to disk is requested we also perform wear leveling. Our algorithm uses the same mechanism as for data to choose the least worn blocks as metadata blocks. However, over time the flash storage system can get filled and fragmented, with a majority of blocks all holding some amount of valid data leaving no blocks completely free. In order for metadata to be written back, we need empty blocks. For this reason, we implemented two additional core functions, move_pages_to_other_block() and get_meta_block(). The function get_meta_block() is the function responsible for finding the best candidate (least worn) block for the updated metadata to be written to. However, the candidate block will not always be free (such as in Figure 8.1). Therefore, we also implemented the function move_pages_to_other_block() which will distribute the pages among other available blocks, erasing the candidate block, making it available to hold new data or metadata (Figure 8.3-8.4).



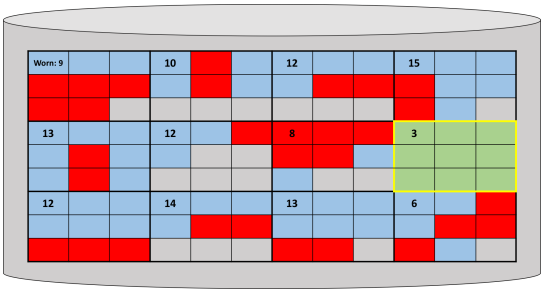


Figure 8. The algorithm for choosing metadata block in phase 2

Parallel Version

Now in the parallelized implementation (Phase 3), both garbage collection and flushing metadata functions always try to find the blocks having the smallest worn counter similar to our Phase 2 implementation. Therefore, any function looking for victims is actually helping wear-leveling. The wear-leveling is implemented by two parts. Firstly, the garbage collection tries to provide as many free blocks as possible in the system. To this end, we upgraded our garbage collection to support merging ability as shown in Figure 9. Secondly, when the flush_metadata() function runs, it looks for all free blocks and selects n blocks (where n depends on total meta data size) which have the smallest worn counter. Because of this, hot data (metadata blocks in our case) will be relatively evenly distributed among the entire flash storage system.

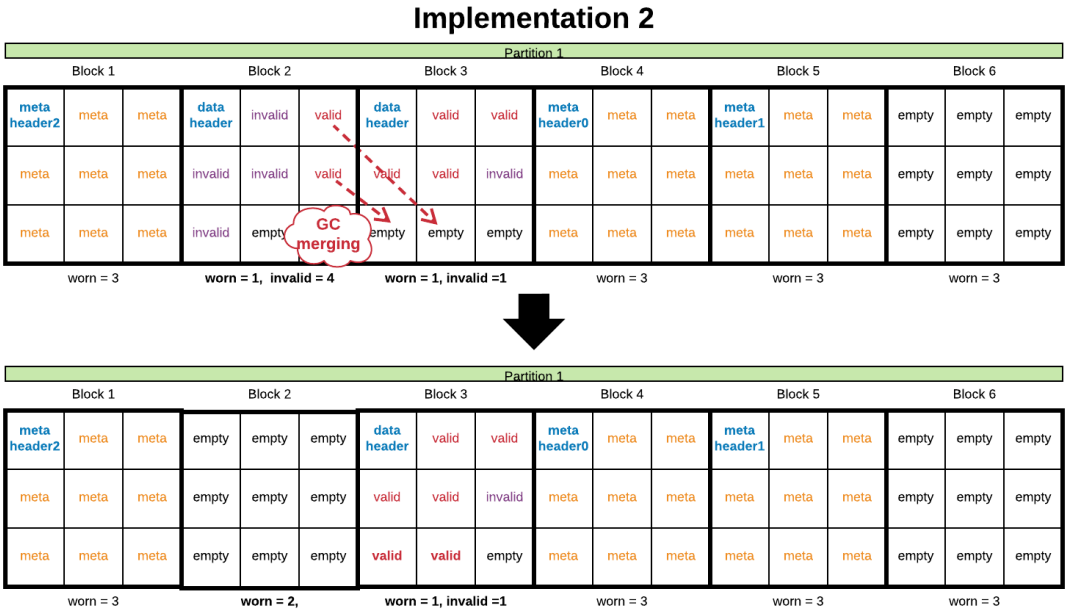


Figure 9. Garbage collection merging in phase - 3

Figure 9 gives an example of GC merging. Initially, our garbage collection checks if the invalid count in a block is larger than or equal to a user-defined threshold (4). If so, it makes it as our target block. Garbage collection then will try to move valid data from the target block (2) to another free block, which has the least worn counter in the system.

Performing garbage collection in this manner has 2 benefits:

1. Less data is moved in a long-term perspective: This implementation tries to strategically position all the valid data into as few blocks as is possible (thereby not scattering data everywhere). This will result in more space for flush_metadata() to select a certain number of blocks to fulfill the wear-leveling. Additionally, the frequency of flushing will be reduced compared to scattering the valid data into any valid block.
2. Performing merging together with garbage collection at once: We decided to implement our merging in garbage collection since, no matter what, we have to erase a block when a garbage collection is performed. For this reason, performing merging when garbage collection is the opportune time to do so.

- Limitation 4 (BONUS): The current system does not utilize any concurrency for better overall performance.

In the Phase 3 implementation, we implement garbage collection and flushing of metadata to run periodically as part of asynchronous timer interrupts. Having it run asynchronously allows us to isolate it and therefore get a boost to performance as it does not weigh down other user operations by running on its own in a timer interrupt. However, this design choice DOES add a layer of complexity as we now need to develop the storage system to be able to handle concurrency. Why? We need to guarantee that garbage collection and flushing metadata do not occur while another operation is being performed on either the disk or metadata in RAM. Without this guarantee in the implementation, it is possible that a state mismatch could occur between the actual disk state and what is recorded in the metadata (hash table saying where valid data is) leading to a fatal error such as overwriting a valid page, etc.

For this reason, we implement a few locks:

- A coarse grained lock (erase_lock): We put this lock around all instances of code that are called by developers/users and garbage collection, flushing metadata functions, such that if garbage collections occurs it will have to wait until other operations are completed and vice-versa that not disk operations and metadata modifications occur until garbage collection has been completed.
- A finer grain lock (list_lock): This lock specifically protects all metadata operations. e.g. the linked list associated with each block and keeps track of given key/value pair data with a given block (explained in limitation: read latency), the hash table, and the block information.
- A finer grain lock (one_lock): for protecting basic MTD _read & _write operations

Implementation

We now go over more specific implementation details for achieving solutions for the above listed limitations of the given prototype. We give short overviews of each new function added to the code for our implementation explaining objectives such as what it does and its overall goal is for our project implementation. We also will provide short code snippets where applicable.

hash.h

Synopsis: declares hash table functions, defines the max size of our hash table, defines the hash table bucket structure
Changes: New file

```
typedef struct {
    struct list_head p_list;
    int dirty;
    int index;
    char key[128];
    page_state p_state;
    blk_state *b_state;
} bucket;
```

Code 2. Structure of bucket

hash.c
Synopsis: implements hash table functions
Changes: New file

Input: key of key/value pair
Action: generate associated hash of input
Output: hash value of input
unsigned int hash(const char *str)

Input: global hash table, key of key/value pair, disk page index
Action: adds key and its index to global hash table
Output:
    hash_index of added key: success
    -1: error
int hash_add(bucket *hashtable, const char *key, int index)

Input: global hash table, key of key/value pair
Action: searches hash table for key & returns disk page index
Output:
    -1: Hash value not found
    -2: key/value pair has been marked for garbage collection, no longer valid
    0<x<MAX_PAGES: Success, found page index of input key
int hash_search(bucket *hashtable, const char *key)

device.c
Synopsis: implements code related to the virtual device.
Changes: Functionality was added. Specifically, we added additional IOCTL operations:

- IOCTL_DEL: Enables userspace to now be able to delete key/value pairs.
- IOCTL_GC: Enables userspace to perform manual garbage collection (for debugging)
- IOCTL_PRINT: Enables userspace to print current status of hash table

core.h
Synopsis: declares states for a flash block, flash page, status of a block, and global attributes for the storage system.
Changes: Added additional variables for the status of a block and added a new global attribute to utilize with metadata. Changes are indicated with associated comments.

```c
/* data structure containing the state, wear level, and the number of invalid pages of a flash block */
typedef struct {
    struct list_head *list;
    blk_state state;
    int worn;       /* for wear leveling */
    int nb_invalid;  /* for GC */
    int current_page_offset;
} blk_info;
```

Code 3.

NOTE: This global attribute structure was used together with lkp_kv_cfg in implementation 1. Implementation 2 eliminated this global attribute structure and fields missing in lkp_kv_cfg from lkp_meta_cfg were added.

```c
typedef struct {
    struct mtd_info *mtd;
    int mtd_index;
    int nb_blocks;
    int block_size;
    int page_size;
    int pages_per_block;
    blk_info *blocks;
    int format_done;
    int read_only;
    struct semaphore format_lock;
    int hashtable_size; /* Total size of hash table in bytes */
    int block_info_size;    /* Total size of Flash in bytes */
    int metadata_size;  /* Total size of hashtable_size + block_info_size */
    atomic_t recent_update; /* flag for interrupt to check when to flush to disk */
} lkp_meta_cfg;
```

Code 4. structure of lkp_meta_cfg

core.c
Synopsis: implements majority of functions for prototype functionality of flash storage system. Functions mentioned below are EITHER “NEW” or “UPDATED” for our project.

Updated Functions:

static int __init lkp_kv_init(void)
int init_config(int mtd_index, int meta_index)
int init_scan()
int set_keyval(const char *key, const char *val)
int get_keyval(const char *key, char *val)

New Functions:

Input: VOID
Output:
    0: Success
    -2 : meta_on_disk_format failed
    -1 : write_meta_page failed
Action: Writes out metadata partition (Implementation 1) from RAM to disk. When force true it will spin until it runs.
int flush_metadata(bool force)

Input: hash table index of page that user wants to update
Output: VOID

Action: Finds the page using hash_idx, updates that page data at given index to be invalidated via updating p_state back to PG_FREE and nb_invalid++

void invalid_pg(int hash_idx)

Input: VOID
Output: VOID
Action: This function counts the number of invalid pages (nb_invalid) within the flash storage system. If the threshold has been passed, this function kicks off garbage collection. (Garbage collection still occurs at normal intervals, but this compensates in cases of overloading the system with updates or delete and writes)

void gb_check(void)

Input: key value that user wants to delete
Output:
- -1: key not found in hash table OR deleting empty key
- 0 < x < MAX_HASH_INDEX: Successfully deleted key

Action; Searches hash table for given key, updates the page containing that key to be PG_FREE and returns index of key deleted if successful

int del_key(const char *key)

Input: VOID
Output:
- 1: Is READ-ONLY, all blocks are used
- 0: Is NOT READ-ONLY, a block has at least 1 free page able to be written to.

Action: Iterates through all blocks on disk and checks if all blocks current_page_offset is at the end meaning that every block is FULL

int is_read_only()

Input:
- page index of where header should be written (should always be page 0)
- MACRO that specifies if the header should be a data header or a metadata header
  - NAND_DATA: specifies this will be a block holding data
  - NAND_META_DATA specifies this will be a block holding metadata
  - Header will be flagged as 000000000 (kzalloc) for data block. Header will be flagged as META_HDR_BASE[X] for Metadata block X where [X] is the block of the set (Metadata is organized as more than 1 block)
- A number specifying which block is this header being written for the metadata (metadata must be accessed consecutively, since metadata takes up more than one block, the storage system needs to keep track of which block segment it is currently writing)

Output: VOID
Action: writes the block header to a block that is designated BLK_FREE

void write_hdr(int pg_idx, int data, int meta_blk_num)

Input: VOID
Output:
- 0 < x < MAX_NUM_BLOCKS: Success, returns the next block the flash storage system should write to
- -1: Flash storage system is full

Action: Called in set_keyval, determine and find block that will receive the next insertion.

int get_next_block_to_write()

Input: VOID
Output:
    0 < x < MAX_NUM_BLOCKS: Success, returns a blocks that is free
    -1: Flash storage system is full, no free blocks exist
Action: Iterates through the disk and finds a free block

int get_healthy_block()

Input: erase_info object used by the MTD driver
Output: VOID
Action: Is the call back function for the erase operation of formatting the metadata partition (Implementation 1)

void meta_format_callback(struct erase_info *e)

Input: VOID
Output:
    0: Success, all metadata has been formatted
    -1: Error, MTD driver error
    -2: Error, formatting error within _erase
Action: Will format/erase entire metadata partition (Implementation 1)

int meta_on_disk_format(VOID)

Input: block index of the block that should be formatted
Output:
    0: Success, block at given index has been formatted and is ready to be used again
    -1: Failure, Error with driver function _erase
Action: Erases a single block within the disk at index and resets metadata information about a given block

int format_single(int idx)

Input: VOID
Output:
    0: Success, a hrtimer timer interrupt has been started
    1: No effect, as timer was already active
Action: Initializes a hrtimer timer interrupt

static int init_flush_timer(void){

Input: VOID
Output: VOID
Action: Disables hrtimer interrupt for flushing RAM-to-disk

static void clear_flush_timer(void){

Input: hrtimer object
Output: Returns flag such that the hrtimer timer is RESET
Action: Function registered as the call back to when the hrtimer interrupt expires. We have set it flush the metadata to disk.

static enum hrtimer_restart flush_timer_callback(struct hrtimer *f_timer)

Input: VOID
Output: VOID
Action: Performs Garbage Collection

void gc(VOID)

Input: VOID
Output: index of target block

Action: Finds a least worn block.
Int get_meta_block(VOID)

Input: index of block to be written to
Output:
0: Success
-2: Error, Kmalloc error
-4: Error, read_page error
Action: moves pages within victim_block to other blocks with space
int move_pages_to_other_block(int victim_block)

Validation Process

We produce a few user-space test benchmarks and scripts to validate the functionalities of our implementations. If it's working correctly most of the time it should return 0. In the interest of time we did not do any performance testing for our Phase 1 Implementation (2 Partitions)

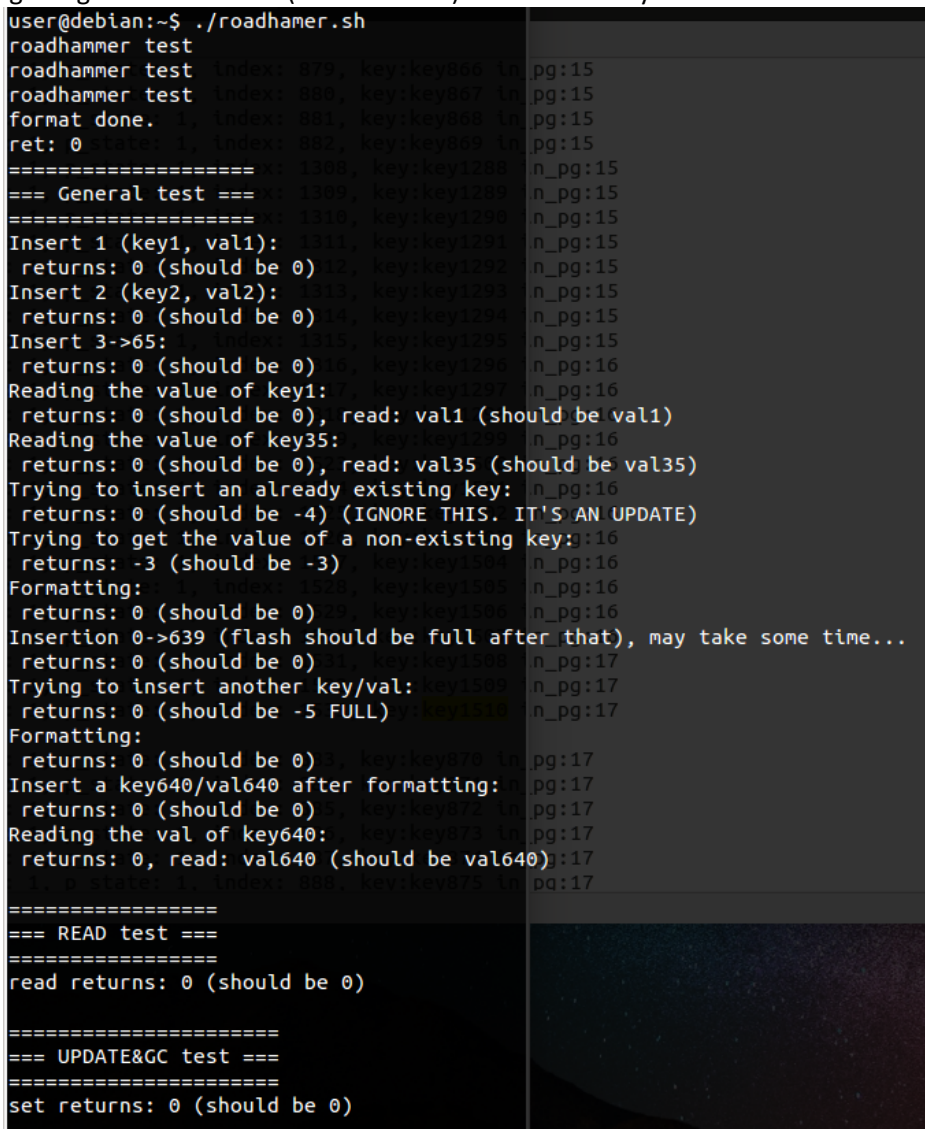
Functionalities validation

Here, we only show the result of running parallel implementation.

Script 1: Roadhamer.sh

General test (testbench): default test given by the instructor (set/get/update/del) (We didn't test read_only mode since we solved this limitation)

- Read test(readtest): read 640 keys
- Update & garbage collection test (testmincheol): invalidate "key100000" 100000 iterations



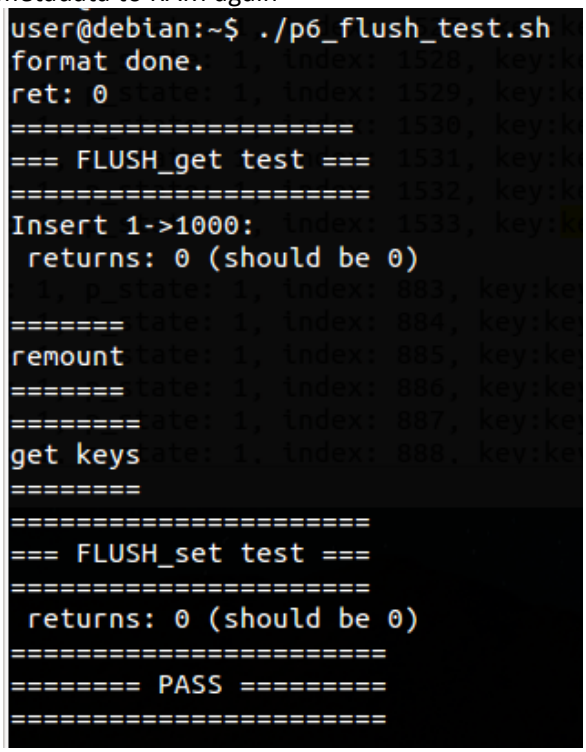
Script 2: p6_flush_test.sh

Flushing & reconstructing metadata

step 1. Flushing metadata back to disk

step 2. power cut (remount the driver)

step 3. reconstruct metadata to RAM again

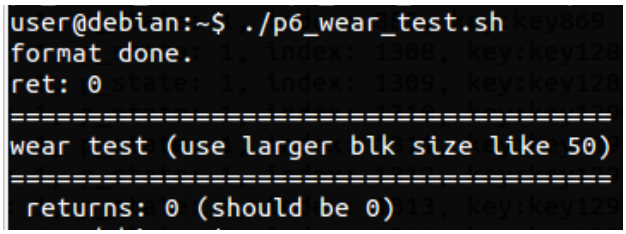


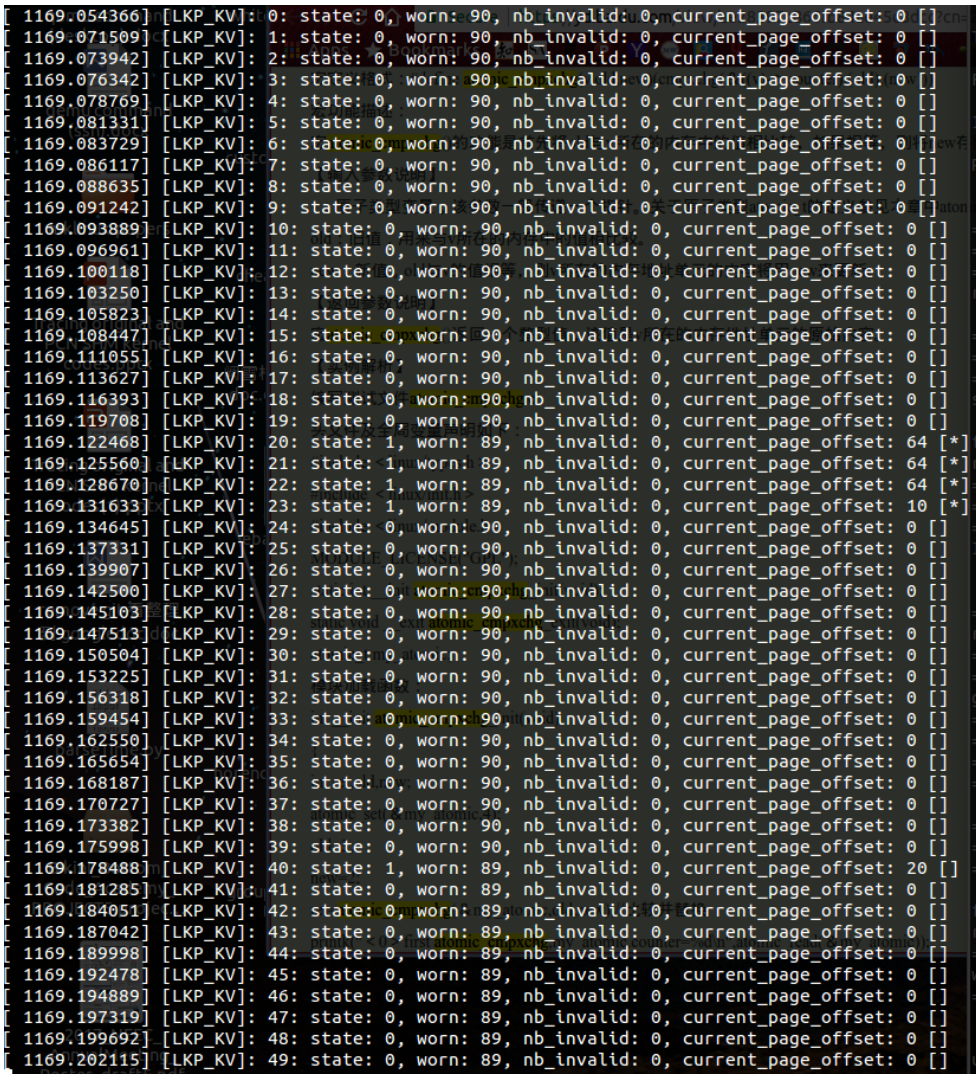
Flushing_meta is running as a demon.

Script 2: : p6_wear_test.sh

wear-leveling (parallel version)

Update key1~19 for 10,000 iterations.





The number of erase (worn counter) is evenly distributed among system.

FYI: script 1, 2, 3 can be all run at once by launching ./all.sh

Performance evaluations

READ experiment is specifically designed for limitation 2 (Read latency improvement - Hash table implementation), we simply compare the time for doing a series of read operations for three flash storage system implementations: the example code given by instructor with a few modifications (vanilla), serial implementation w/o concurrency (serial, phase 2), parallel implementation with concurrency (parallel, phase 3). Next, we compare write latency since the vanilla implementation does not support the "update" operation. Lastly, we compare our two implementations (phase 2 & 3) by doing a bunch of update operations, which will trigger a lot of delete and invalidate, WRITE, garbage collection, and flushing metadata operations. Our code and scripts for the evaluations are testbench_data.c, plot.py, read_write.sh, testUpdate.c, plot_update.py, update.sh (see folder "data_collection" in submitted tar file).

Evaluation Procedures:

For reads/writes evaluation, we use testbench_data.c ,plot.py, read_write.sh
1. change the output file name for different environment in testbench_data.c
2. make and make install to vm
3. launch simulator and insert prototype for different environment
4. change the parater in read_write.sh if needed.
5. run ./read_write.sh to generate csv files in different prototypes
6. use "python plot.py read_vallina.csv read_serial.csv read_parallel.csv" to generate a plot for read
7. use "python plot.py write_vallina.csv write_serial.csv write_parallel.csv" to generate a plot for write
8. remove the modules

For update evaluation, we use testUpdate.c, plot_update.py, update.sh
1. change the output file name for different environment in testUpdate.c
2. make and make install to vm
3. launch simulator and insert prototype for different environment
4. change the parater in update.sh if needed.
5. run ./update.sh to generate csv files in different prototypes
6. use "python plot_update.py update_serial.csv update_parallel.csv" to generate a plot for update time
7. remove the modules

Environment:

Host: AMD FX(tm)-8350 with 8 cores, 16GB RAM
KVM: 2 cores with 4GB RAM

Experimental Results:

vanilla: hash table(X), timer interrupt(X), flush(X), GC(X)
serial: hash table(O), timer interrupt(X), flush(O), GC(O - phase2)
parallel: hash table(O), timer interrupt(O), flush(O), GC(O - phase3)

1. READ: no flush and GC involved

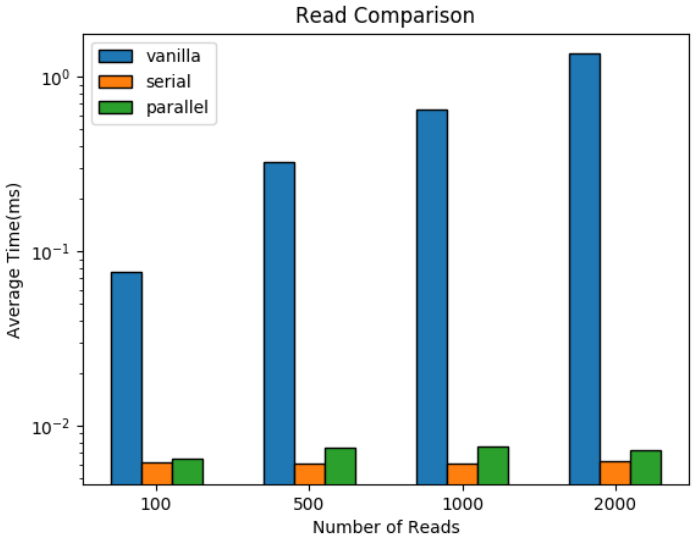


Figure 10. Read comparison

Figure 10 is a semi-log plot of read operations. It depicts that our hash table scales very well and beats the vanilla implementation by a large margin. Moreover, since our parallel implementation (phase 3) does have two more timer interrupt running under the hood (even if it is not actually performing the garbage collection/flushing), this makes parallel is a bit slower than serial implementation.

2. WRITE: no GC involved

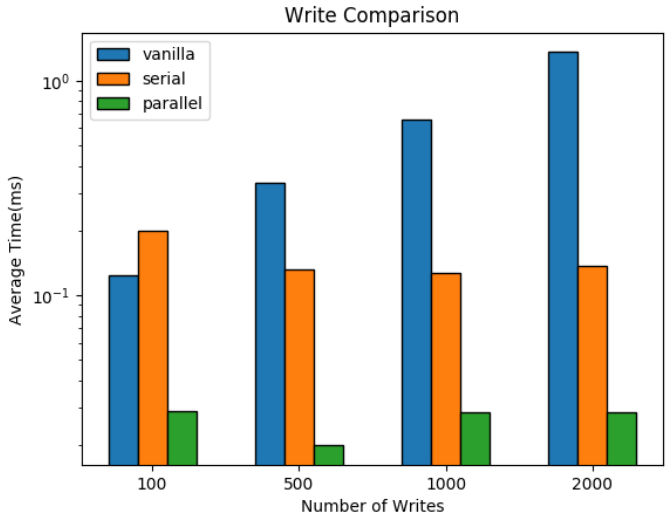


Figure 11. Write comparison

As we can observe in Figure 11, before performing a write, the vanilla implementation tries to iterate through the existing pages being used in order for checking a duplicated key with complexity (O(n)); our

two implementations (phase 2 & 3) exploit a hash table, taking only complexity O(1) for checking if there is a duplicated key. Since a write operation will trigger flushing metadata, our parallel version (phase 3) can efficiently postpone flushing and perform all changes at once with a timer interrupt. On the contrary, the serial version (phase 2) can only set a threshold for performing the flush needs another user operation to occur for the function to be called. For this reason, it can lose data if there is a power cut before reaching the threshold.

3. Update (= DEL + WRITE + Garbage Collection + Flushing metdata): everything is involved

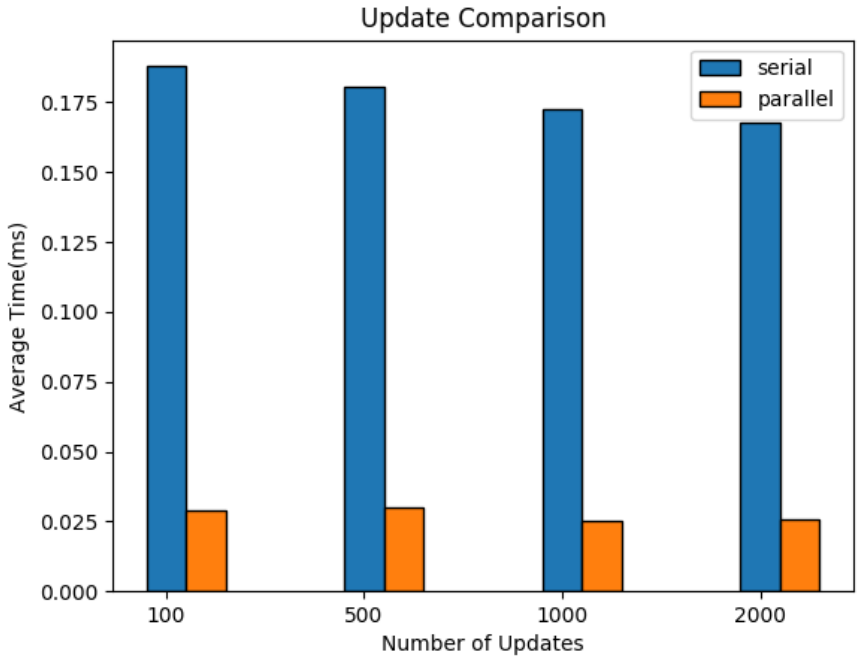


Figure 12. Update comparison

Figure 12 shows our parallel implementation (phase 3) is tremendously faster than our serial implementation (phase 2). The reason is that parallel version can perform more user operations at once, delaying background jobs to a later time and doing all updates at once.

Future Work

While our group focused on the updating of key/value pairs, caching in RAM with optimizations of storing data, developing an effective wear-leveling schema, and concurrency and performance problems, limitations within this prototype still remain.

Important feature lacking in the prototype is the capability to handle data whose size is larger than a single page. In today's world, data comes in all sizes; with the advancement of data storage density and precision of instrumentation, files have certainly evolved to be larger than a single page. In order to handle larger data, a strategy would have to be designed to keep track of individual fragments since the size of a page the system can move around the system largely does not change, but does vary among systems. A potential approach would be to have two fields within the metadata of the page or block. The first field would contain a unique id of the file (something similar to an inode id) while the second would either contain a pointer to the next fragment of the large data file or a NULL terminator indicating the end of the file data has been reached.

In addition, another typical feature within a non-volatile storage system is functionality which addresses soft and hard hardware failure. While unintended bit-flips (soft hardware errors) due to cosmic radiation are handed by the driver, hard hardware errors, such as a faulty block rendering it useless are not handled by the prototype implementation. This makes the prototype vulnerable to such failures as it will attempt to write to all blocks, without any knowledge of which blocks should be avoided and no longer used since no marking mechanism is in place. With flash storage (and unlike traditional HDDs), the hardware can only support a given number of erase/writes on a block before a block has been worn out and no long capable of storing information. Therefore, functionality is needed to detect and mark "bad blocks" within a flash storage device.

Conclusion

In conclusion, our group has taken a basic prototype of a non-volatile storage system in the kernel and enhanced it such that its overall performance and scalability have been improved.

Beyond the minimum features completed, we did the next largest performance gainer feature for this prototype would be to employ concurrency within the non-volatile storage system!!! In Phase 2, read and write operations are performed sequentially as in a pipeline, limiting the throughput and bandwidth the storage system can handle to a single operation at a time. However, concurrency was one of the most complicated to implement as it required a significant overhaul of the current infrastructure. Most of the effort went go into wrapping any operations manipulating data with an appropriate locking mechanism and figuring out dependencies. Not only would traditional reading and writing would need be addressed, but also the mechanism performing periodic garbage collection would need to be made safe. With this refactoring, we hypothesize that the non-volatile storage system would see a major improvement in read and write operation latency especially when dealing with larger amounts of data. However as the validation section shows, we actually take a slight performance hit. This degradation could be tweaked by varying the frequency of flushing and garbage collection interrupt timer intervals.

Specifically, we eliminated three limitations of the given prototype being:

- The system now is capable of performing update and deleting of key/value pairs. Garbage collection is also implemented to be able to take advantage of this freed space on disk and manage updates. Flushing metadata periodically save the all metadata, including block information & hash table, to disk when it's needed.

- The system now has a much better (and scalable) read latency by relying on a hash table with the djb2 hash algorithm to keep track of all data and retrieve its index via hash instead of iterating through the entire disk.
- The system relies on garbage collection and periodic flushes to disk using an algorithm that attempts to minimize uneven wear-leveling by looking at erase counts (worn level) and prioritizing blocks that do not have a lot of data on them.
- We take advantage of concurrency, making our garbage collection and flushing metadata as demons running in the background and make everything concurrency-safe.