On-flash, in-kernel key-value store

Muhammad Usman Nadeem Talha Ghaffar

May 11, 2017

Abstract

Flash memory has some inherent characteristics that make it very different from storage media such as magnetic hard disks. These distinguishing characteristics include the inability to write to an already written page without performing an erase operation, inability to erase just one page, limited erase/program cycles for a flash block (flash wear) and reliability of the flash. In order to efficiently utilize the flash memory, flash storage systems need to consider these constraints in their design. In this project, we develop a storage system for the flash memory in the Linux kernel. The storage system is simply a key-value store. We start with a provided prototype implementation of this storage system. The provided implementation has severe limitations in its functionality and performance. We choose three key limitations of the provided prototype and implement a mechanism to perform (out-of-place) update and delete operations. To support out-of-place updates and deletes we implement a garbage collector to manage the invalid data created by the these operations to enable full flash utilization. We implement a block selection policy for writes and garbage collection that ensures that the flash blocks do not wear out prematurely. We also introduce an indexing mechanism to maintain metadata for the stored data and cache this metadata in RAM to make our read and write operations faster. We write this metadata to the flash using an asynchronous kernel thread to ensure data persistence in the flash. Finally, we validate the correctness of our implementation and compare its performance with the provided prototype to determine its efficiency. We observe that our implementation is scalable and offers huge performance over the prototype.

1 Introduction

Flash memory is a non-volatile storage device where data can exist even when the device is not powered on. A flash memory chip is organized to contain a certain number of blocks and each block contains a fixed number of pages. Like other storage media, flash memory also supports the common read, write and erase operations. However, flash memory has certain constraints that a flash storage system needs to manage in its implementation. The main limitations of Flash memory are:

1. Write can only be performed on an erased page. To perform write to a non-erased page, the page has to be erased first. Since erase cannot be performed at page granularity, the entire block has to be erased. This constraint has implications in the implementation of update and delete operations.

2. Writes can only be performed sequentially in a flash block. This constraint means that a flash storage system needs to write a block completely before moving to the next block.

3. A block can only be erased a finite number of times. Once that limit is reached, the block is worn out and is rendered useless. This constraint has implications in the block selection policy for write and erase operations.

4. Flash memory has some reliability constraints as well, because upon data read/write on flash, bit-flips can happen. A flash storage system may need to handle such errors with error correcting codes.

For this project, we are concerned with NAND flash memory that is more commonly used. In the project, we implement a key-value storage system in the kernel for Flash memory. We are provided with a a functional prototype of the storage system with some severe limitations. We identify

some of the important limitations of the prototype. After identifying the prototype's limitations we selected and resolve three critical limitations of the storage system. Now, in this section, we briefly describe the current implementation of the prototype before enlisting the identified limitations in the next section.

1.1 Prototype Implementation

1.1.1 Storage System

The core of the storage system is implemented as a kernel module. It uses the flash driver 'mtd' to perform read, write and erase operations on the flash. The storage system exports a virtual device 'lkp_kv' in the /dev directory. This virtual device is the interface to the storage system for the user-space applications.

The listing below shows the main functions. The module currently implements only get, set and format functionality. A brief description is included in the code comments.

```
// core.c
int init_config(int mtd_index);
// launched on mount time
// prepares the in-memory metadata by going over and reading every page in the disk.

int write_page(int page_index, const char *buf);
// writes data in the buffer to the page specified by the index
// we can only write one full page at a time, no more no less
// we can only write to a page if it is empty

int read_page(int page_index, char *buf);
// reads data into the buffer from the page specified by the index
// we can only read one full page at a time, no more no less

/* actual operations */
int set_keyval(const char *key, const char *val);
// sets key-val pair ONLY if not already present and there is free space
// updates the metadata in memory

int get_keyval(const char *key, char *val);
// get the value for the specified key and returns a code

int format(void);
// formats the disk
```

1.1.2 Storage Format

The prototype is implemented to write one key-value pair on a single flash page. Both keys and values are only character strings. Each key-value pair is stored as header + data on the flash page. Header consists of two 32-bit integers representing the size of the key and value respectively. Key is stored following the header and the Value is stored after the key. The terminating null character is not stored for the key and value strings.

1.1.3 Interface with user-space

In the user-space, the prototype provides a C library that can be included in user applications to request services from the storage system. Following listing shows the important functions defined in that library:

```
// kvlib.c
/* writing a key/value couple */
int kvlib_set(const char *key, const char *value);

/* getting a value from a key */
int kvlib_get(const char *key, char *value);
```

```
/* fomat */
int kvlib_format();
```

2 Prototype Limitations

2.1 Missing Functionality

1. No Update or Delete operations:

   The prototype, currently, cannot update or delete any key-value pair. This means that any data written to the disk is final and cannot be modified. A user can only add/read data to the flash and once it's full, the flash is set to read-only. In order to write additional data, the user would need to format/erase all the existing data.

2. No bad block management:

   Bad blocks can be introduced in the flash because of bit-flips during read/write to the flash. The prototype implementation has no support for bad block management.

3. A single page is limited to a single key-value pair:

   This means that if size(key + value + header) is less than the size of page, then the remaining page will be empty, resulting in a wastage of space.

4. Size of key-value pair limited to size of a page:

   In the current prototype implementation we cannot have size(key+value+header) $>$ flash page size. The implication is that only a limited kind of data can be written to the disk.

5. No permissions and user management:

   Anyone can read/write a key-value pair to the disk because we do not have any metadata related to permission and user management.

2.2 Performance and Scalability

1. No indexing - No metadata on disk:

   We currently need to scan every block and every page to create metadata every time we mount the disk. This has a high latency. Apart from that every time we want to read something we have to scan the entire disk to find the page containing the relevant key-value pair. During the write operation we have to do the same to ensure that we are not creating a duplicate key. This increases the read, write and mounting latency.

2. No particular Wear Leveling or Garbage Collection:

   The prototype currently does not implement any Wear Leveling or a Garbage Collection policy. They are required to implement a feasible Delete/Update function and to prolong the life of the disk (more details in the sections that follow).

3. No concurrency support:

   Flash Drives can support access to multiple memory channels at the same time. This way we can write to multiple channels at the same time and read from multiple channels at the same time leading to higher throughput.

3 Resolved Limitations

3.1 No indexing - No metadata on disk

3.1.1 Description

We currently need to scan every page to create metadata every time we mount the disk. This has a high latency.

Apart from that, since we do not have any index table in the metadata that we keep, every time we want to read something we have to scan the entire disk to find the page containing the relevant key-value pair. During the write operation we have to do the same to ensure that we are not creating a duplicate key. This increases the read, write and mounting latency.

3.1.2 Solution

There is a two part solution to this problem:

1. Introduce an indexing mechanism, and
2. Write that metadata to the disk for persistence.

```
// mapping and state for a single key
typedef struct {
   __u32 keyHash;
   int block;
   key_state state; // KEY_DELETED | KEY_VALID
   int page_offset;
} directory_entry;
```

To fix this limitation we only to make changes to core.c and core.h.

The listing above shows the main structure we added to core.h to implement indexing. To save space and to make the key comparison more efficient we convert the key to an integer hash (collisions are unlikely so we do not worry about them, we have not faced any collisions during our tests, they can easily be mitigated). For that hash we keep the location of the key-value pair on the disk, in terms of block and page offset. We have one copy of the structure for every key. Since, we only store one key-value pair in one page, we know that the disk can only contain a fixed number of pairs i.e. equal to the number of pages - the size of the metadata. Which is why we create a fixed sized array to store the directory_entries.

This way we only have to do a linear number of searches and just a single integer comparison per search (as compared to a string comparison in the prototype) to find the location of the key-value pair. Since this structure is actually cached in memory, the search itself is much faster than accessing the disk to find the key.

Indexing reduces both, the read and the write latencies.

To reduce the mount time latency we write all the metadata, including the directory list and the old metadata from the prototype at the start of the disk. For the default disk size of 10 blocks the metadata takes $<= 1$ block but we reserve the entire block for metadata purposes.

At mount time we read the appropriate number of pages and try to recreate the metadata. In the first few pages we have some parameters that contain data specific to the disk e.g. block size, number of pages etc. If the values for these parameters read from disk matches the information we get from the mtd structure then we assume that the metadata is correct and continue to read the remaining data. This is faster than the prototype implementation because we only have to read a few pages (depending on disk size and metadata size for that disk) compared to the whole disk as in the prototype implementation.

In the case that those parameters do not match with the ones we get from mtd we assume that either the metadata on the disk is corrupted or that this is a new disk. So we format the disk and reinitialize the metadata parameters as-if the disk were empty.

Both of these are much faster operations as compared to reading the whole disk.

3.1.3 Limitations of current solution and Discussion

One of the limitations of this approach is that we use a fixed size array for the directory. This results in a linear number of comparisons. An alternative would be to use a map. Linux implements maps using a binary search tree. This would have resulted in a less number of comparisons but not only would it have resulted in more space but the implementation would also have been much more complex.

One of the complexities come from the fact that we have to write the metadata to the disk. In order to do that we have to serialize all the structures. Serializing a map would have added to the complexity because we cannot just copy the memory to the disk, we have to pay special attention

to the pointers and "flatten" the data structure when writing to the disk. On reading from the disk we have to recreate all the pointers.

The trade-off here was speed vs memory usage and implementation complexity but it would be interesting to see comparison of a map vs an array implementation. We leave it as one of our future works.

Another limitation is the amount of wear the first block gets due to the frequent update of metadata on disk. Our current implementation always writes to the first block, but we could use multiple different blocks in an alternative fashion to equally divide the wear count. It was discussed in the lectures that the blocks containing the metadata are more resilient than the other blocks but even then we limit the the rate at which write the metadata to the disk. We have a kernel thread that runs every 500ms and checks if there has been any changes to the metadata or not. It writes the metadata back to the disk only if there were changes. This can cause consistency issues and even though that is outside the scope of our project we have discussed it toward the end of this report.

In short our solution provides these benefits:

1. Faster reads because the location is stored in an index.
2. Faster writes because we can consult the index to tell us if the key already exists.
3. Since we have the index in memory we don't have to access the disk, saving precious time.
4. We reduce the comparison complexity and save metadata space by converting the key into a hash.

3.2 Update/Delete Operation

3.2.1 Description

The current prototype writes key-value pairs to the disk through the set operation. It has no update or delete operation and the set operation cannot be used for an already existing key. Since a key once written to the disk is permanent, this severely limits the usability of the key-value store file system.

3.2.2 Solution

A new delete operation has been implemented and changes have been made in the code that sets a key-value pair in the core.c and core.h files. Other than that the user-space API has also been adjusted to accommodate the new functionality.

To implement Delete(Key) we just have to find the corresponding directory_entry* for that key and once have that structure we just mark its state as KEY_DELETED. Once the key has been marked deleted it wont be accessible through Get operations. But that is not all, we still have to free up the space on disk. For this purpose we just access the block and page index stored in that directory_entry* and mark it as PG_DELETED, this lets the garbage collector know that this page is free.

Frequent deletion operations result in scattered deleted pages over different blocks. Limitations of flash storage hardware prevents us from just freeing a single page, we are only able to free a full block. This is why We need to use a garbage collector, which runs occasionally and has to reshuffle some data pages, so that after garbage collection we have at least one writable block and move minimal amount of valid data. More details about garbage collection are under the Wear Leveling section.

The update operation is built on the delete and the set operation. It first calls delete(key) and then calls set(key, val). This is due to the limitation that we cannot modify an already written page. Although update has been implemented as a separate operation in the user-space API, the existing set operation includes all of the update's functionality. For example, we can use both user-space update() and set() to update the key-value pair.

The set function in core.c has been modified to add a check for the condition when the key already exists in the disk. If this is the case then we call the delete(key) operation followed by a normal set, which updates the directory_entry* structure (location of block and page where the key-value is stored) corresponding to the key and also updates the new page's state from PG_FREE to PG_VALID to mark it as used.

3.2.3 Limitations of current solution

From what we can tell there are no limitations to the update and delete operation's implementation.

3.3 Wear Leveling and Garbage Collection

3.3.1 Description

As has been discussed earlier, in-place updates are not possible in Flash memory without sacrificing performance. The reason is the limitation that the flash memory has to be erased before any data can be written to it again. Since in this key-value flash filesystem, we are implementing delete and update operations, the updates have to be performed out-of-place in order to achieve reasonable performance. However, out-of-place updates introduce invalid data in the flash blocks. For example, for deletion, we mark the key as invalid and for update, we mark the previous entry as invalid and write the updated data at the next selected page. To fully utilize the flash memory, this invalid data (garbage) has to be removed and pages have to be freed in the flash. Therefore, a flash filesystem needs to provide a garbage collector that can manage the valid/invalid pages in the flash.

A critical limitation of the Flash memory is that the Flash memory cells can only be erased a finite number of times. Repeated use of a particular block (for writes) will cause the block to wear out. For garbage collection, we need to move the data around to create free space for full flash utilization. If it so happens that some blocks are selected more than others for garbage collection, those blocks will wear out prematurely. Thus block selection policy (both, for garbage collection and for writes) becomes crucially important. Additionally, a wear leveling policy is needed which ensures that the blocks do not get 'victimized' unfairly for garbage collection and also that the block selection for writes is fair.

Garbage collection and wear leveling policies are quite closely tied to each other. In the following section, we describe our implementation of these policies.

3.3.2 Garbage Collector Implementation

As evident from the name, a "garbage collector" collects garbage, essentially freeing up space in memory and thus enabling full utilization of the Flash. A garbage collector has to perform two key steps:

1. Identify free space in the Flash
2. Reclaim invalid pages by moving valid data to free space

We implement a garbage collection policy that can be categorized as 'greedy garbage collection'. In our implementation, the first step a garbage collector does is to identify free space in the flash. While selecting free space for writing data after garbage-collection, there are two scenarios to consider **a)** if a free block exists and **b)** if there is no free block in the flash. Our policy for selecting free space also takes wear-leveling into account. Since each block's metadata has BLK_USED or BLK_FREE information, we use this metadata to identify free pages in the flash. If a free page exists, we select a page that has the lowest erase count. If, however, there was no free block in the flash (all the blocks have some data), we greedily select a block that has lowest number of valid pages to extract most free space. While identifying such a block, the garbage collector may encounter a scenario where none of the blocks have any free space (in other words, all the blocks have valid data and the flash is actually full). In such a case, the garbage collector sets a flag to indicate that the flash is full and some data has to be deleted before any further write operations can be performed.

After identifying some free space to write 'garbage-collected' data to, the garbage collector needs to identify victim block(s) for garbage collection. At this point, since the garbage collector knows how much free space is available for it to write to (either a full free block or a block with minimum valid pages), it tries to identify victim blocks. To select a victim block, we consider both the erase_count of the block and the number of invalid_pages in the block. We calculate the ratio erase_count/invalid_pages for each block with BLK_USED state and select a block with the lowest value. This enables us to find a balance between maximizing the free space, performance and maintaining wear leveling. The higher the invalid pages in the victim block, the more the free space we get and the less data we have to move. Less data to move equals fast operation. If more

than one blocks are identified as victim (because their valid data can be moved to identified free block), the garbage collector will select all of them as the victim blocks. The algorithm for victim block selection is described in the listing Algorithm 1.

Algorithm 1 Victim block(s) selection

```
1: victim_blocks[]
2: while true do
3:     for each non-metadata block do
4:         blk_victim_potential = UINT_MAX
5:         if block.state == BLK_USED then
6:             if block.invalid_pages == 0 then
7:                 continue
8:             end if
9:             new_blk_victim_potential = (block.erase_count * 100) / block.invalid_pages;
10:            if (new_blk_victim_potential <= blk_victim_potential then
11:                if block.valid_pages <= req_pages then
12:                    Select as victim block
13:                    blk_victim_potential = new_blk_victim_potential
14:                end if
15:            end if
16:        end if
17:    end for
18:    req_pages -= req_pages - selected_block's valid pages
19:    Add selected victim block to victim_blocks
20: end while
```

After the victim block(s) have been identified, the garbage collector needs to move data from the victim block(s) to the free space. We do so by initially copying valid data (PG_VALID page state) from the victim block(s) into a buffer. Here, we need to consider two cases **1)** where the GC found a free block **2)** where the GC could not find a free block, but identified a block with least valid data. We cover case 1 by copying all the valid data to the free block and then erase the old block. To cover case 2, we first have to save the data erase the new block and then write it with the data from the buffer.

After writing the data to the block, the garbage collector updates the current_block, current_page_offset in the metadata so that following writes can be served. Moreover, the GC sets the block's state to BLK_USED to indicate that the block is no longer free and also sets the number of free pages in the block accordingly. Note that if the number of pages written to selected block are less than page_size of the block, all the following writes to the flash will be done in this block.

While writing data to the block, we also need to update the in-memory metadata for this data so that read/write operations can still be performed. For every page that we write to the new flash block, we read key and value from the buffer that we're writing from. Based on the key read from the buffer, we compute the hash and locate that entry's metadata based on the computed hash. After that, we update the entry's metadata with the new block and page offsets of this entry (key-value pair).

During garbage collection, a scenario might occur where the GC could not find any victim block (all the blocks had valid data) but (at least) one free block. In this scenario, the GC would update metadata to reflect that this block should be used for any following writes to the flash. Note that the selected free block is the block with lowest erase count among all the free blocks.

3.3.3 Garbage Collector Invocation

In our implementation, the garbage collector runs synchronously with the writes to the flash. After a write to the flash, we determine the page and block IDs so that future writes to the flash can be serviced. While updating the page and block IDs, we check the number of free blocks left the flash. If this number falls below 20%, we invoke the garbage collector. The garbage collector, then frees up some space from the used blocks and updates the page and block IDs in the metadata.

The garbage collector also needs to manage scenarios where some data was deleted by the user after the flash has become full (no more free pages). In order to achieve that, we use a lazy

approach with garbage collector invocation. We maintain a field invoke_gc which is set after a delete operation was performed on a full flash. Now, when a write (set or update) is performed, if invoke_gc is found to be set, the garbage collector will be invoked and some page(s) will be freed so that write can complete successfully.

3.3.4 Wear leveling Implementation

In section 3.3.2, we have discussed how we ensure that wear leveling is performed during garbage collection. Besides that, in order to ensure wear leveling among flash blocks, we use a block selection policy that is based on the erase count of the flash blocks. In this policy, whenever we need to select a block for a write operation, we select a free block with lowest erase count. By doing this, we ensure that wear is spread out among all the flash blocks and no block will prematurely wear out.

3.3.5 Limitations of current solution

The main limitation for the current implementation of the garbage collector is that it is invoked synchronously with the write operations which can slow down these write operations. This limitation can be avoided to some extent by invoking the garbage collector asynchronously from a kernel thread and making sure that the GC and the user do not write concurrently to the flash pages.

The implemented wear-leveling scheme considers that all the data written to the flash is dynamic (hot) and that there's no 'cold' data. This assumption is true to some extent within the context of a key-value store but generic flash storage implementations would need to take hot/cold data into account as well. A cold block may never have invalid pages and will have a very low wear count.

4 Experimental Setup

To perform our experiments, we simulated the NAND Flash using the nandsim simulator. The configurations for the simulated flash memory chip are listed in Table 1:

Table 1: Simulated Flash

# of blocks	10
Pages per block	64
Page size	2 kB

Table 2: Simulated Flash with a higher disk size

# of blocks	100
Pages per block	64
Page size	2 kB

Table 3: Computer Specifications

# CPU	i5-3210m
RAM	12GB
RAM (VM)	4GB
RAM (QEMU within VM)	2GB
HDD	128GB SSD
HOST OS	Windows 10
Guest OS	Linux 4.4.0-75-generic
QEMU OS	Linux debian 4.0.9 x86_64

5 Validation

In this section, we describe the tests that were used to validate the implementation of our key-value storage system. Mainly we validate four aspects of the storage system **1)** correctness of the

Table 4: Simulator

Name	The NAND flash simulator (nandsim)
Author	Artem B. Bityuckiy
License	GPL
SRC Version	593EAF1D507E55697FC80CD

read, write, delete, update, format operations **2)** correctness of garbage collector implementation **3)** correctness of the data/metadata persistence on the flash **4)** correctness of our wear-leveling implementation.

5.1 Operation correctness

In the tested configuration, we have 9 blocks (after excluding the metadata block) and 64 pages per block, hence, we can write 576 key-value pairs in this implementation. To perform operation validation, our test application first fills up the flash with key-value pairs and validates whether each write was performed successfully. Then, it reads all the key-val pairs, thus validating that the writes were successful and also that the read operation works correctly. After this, the test application updates values for all the keys in the flash and then validates the update operation by reading and verifying the updated values. After update, we test the delete operation by deleting all the keys and then validating the delete operation by testing that no keys were read from the flash.

5.2 Garbage collection

To test garbage collection, our test application tries to fill all the blocks in the flash. During these write operations, it creates some invalid data in each page by deleting some keys. Since, the garbage collector is invoked whenever the remaining free blocks are less that 20% of the total blocks, during these flash writes, garbage collector will be invoked multiple times and free up some space so that next writes can be serviced. Note here that the flash won't be full since the garbage collector would have freed up some space (1 free block + 17 free pages at the end of these writes). We verify the correctness by testing that the write operation was successfully performed and then reading the written key-val pair from the flash. After all the writes have been performed, we read all the written key-value pairs again to test for correctness of the writes (and consequently the garbage collector).

We, then, fill the flash completely. In a full flash, we perform some writes and verify that the writes cannot be done. Then, we delete some keys so that some invalid data can be created for garbage collector to free up some space. We, then, try to perform some writes to the flash so that the flash is full again and verify that writes were completed successfully.

Now that the flash is full again, we delete some keys so that upon next invocation, enough data would be invalid for the garbage collector to be able to free more than one block and write all the valid data to only one block (Note that, we verified this by printing information from the garbage collector since we cannot actually get 'freed' or 'garbage-collected' blocks' information in the user-space). The garbage collector was invoked upon attempt to write some data to the flash.

5.3 Persistence

To test for persistence, we need to ensure that the data is read successfully when the storage system is remounted. We test this by first filling up the flash and then removing the module. Later, we insert the module again and verify that the data we wrote previously can be read successfully.

5.4 Wear-leveling

To test wear leveling, we need to observe the erase_count of the flash blocks after a lot of writes/updates operations have been performed on the flash. To achieve this, in our wear-leveling test, we write the flash to 80% of its capacity. After that, we randomly select keys and update them. In one iteration, we perform TOTAL_PAGES * 0.8 updates, each page only contains one key-val pair. So if we initially wrote 200 values, we overwrite 200 randomly chosen values in each iteration. This process is done in a loop that runs 100 times. We chose this process to simulate random updates

in the flash to get close to real wear-leveling estimates for our implementation. We exit from the module after this test and check the erase_count of the flash blocks (visible on dmesg output on module exit). We verify that the wear is distributed almost evenly across all the flash blocks. Figure 1 shows the histogram for the wear count of different blocks of the flash. It can be seen that the wear is evenly spread out among all the flash blocks.

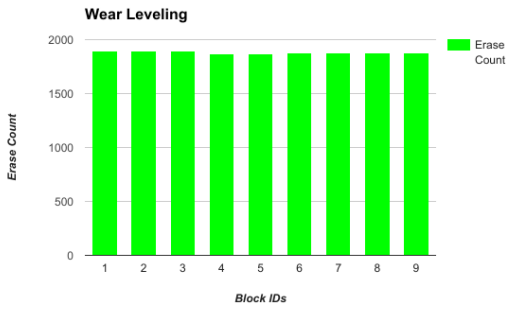

Figure 1: Wear Leveling

6 Performance Evaluation/Scalability

6.1 Our implementation vs the Prototype

We test using the default disk size of 10 blocks. Figure 3 shows the time taken by different operations by our implementation and the prototype. We only test read and write here because the prototype has these operations common with our implementation.

6.1.1 Write

We write till the disk is 25% 50% and 100% full. There is a negligible increase, almost constant, in the time taken for our implementation (discussed later) but for the prototype we see a steep increase in the time taken. This is because at every write operation they do "n x m" comparisons. Where n is the size of the key and m is number of keys on the disk. Another source of the overhead is reading the disk pages on every iteration. As discussed above above our number of comparisons is reduced to only "m" and we do not have to access the disk.

As the disk usage increases, there are more keys on the disk, hence more number of comparisons and more time taken.

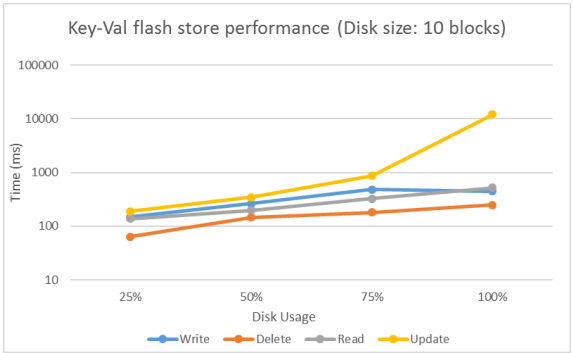

Figure 2: Key-Val Flash store performance (Disk Size: 10 blocks)

6.1.2 Read

Even though on hardware level read is faster than write we do not see much performance difference between the reads and the writes. This might be because most of the time is consumed in

computation and not the actual IO. To compare the prototype and our implementation we see the same almost the same difference we saw between the writes of the two. The reason is also the same. The prototype does a lot of reads and comparisons to make sure there is no duplication.

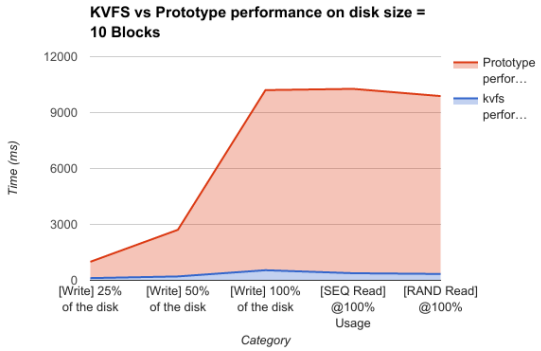

Figure 3: KVFS vs Prototype performance

6.2 Wear Leveling

We filled the disk till 80% of its capacity (suppose n key-val pairs). Then we randomly choose n keys and updated their values (update = delete+set). We chose random keys to make sure there is a good distribution of invalid/deleted pages among the blocks. We repeated this 100 times. Figure 1 shows the results. It shows how many times the block (excluding the first metadata block) was erased. The minimum and maximum are 1867 and 1894 respectively. The low variation shows that the blocks are wearing out fairly.

6.3 Mount Latency

We could not test mount latency because there was a very high variation in multiple mounts. Technically we needed to compare the first time mount, second time mount, and mount with the disk full.

For the prototype all mounts would have resulted in the same time and higher than our implementation because they scan the whole disk. For our implementation the all the mounts would have been almost the same time too (but less than the prototype) because we read the first few pages in any case to confirm if we have valid metadata or not. The third case of mounting at full disk usage would also be fast because our mount time does not depend on the amount of data but on the size of metadata, which would remain same irrespective of the amount of data.

6.4 Scalability

Looking at figures 2 and 4 we can see that our implement scales very well with an increased size disk. The respective increases in the times for both disk sizes have the same linear trend.

For the difference between the read and write we see a bigger difference when we have a lot of data because at that point IO time becomes greater than time used computing.

The increase in trends is because of the increased comparisons (we have a linear array for the index) and we see a linear increase in all operations except update. For updates we have an almost exponential increase as the %age of disk usage increases. This is because of delete+write nature of update and the limitation of flash, that we first have to move the pages to create free space. The more the usage the less number of free pages we have and the more the deletion the more scattered invalid pages would be.

7 Other Limitations and potential solutions (future work)

1. A single page is limited to a single key-value pair:

   This means that if size(key + value) is less than the size of page, then the remaining page will be empty, resulting in a wastage of space.

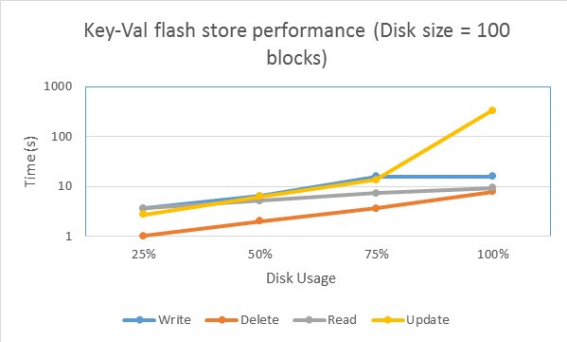

Figure 4: Key-Val Flash store performance (Disk Size: 100 blocks)

To solve this problem we can cache the KV pairs before writing them to the disk. Since we can only write one page in its entirety, a possible solution would be to keep caching the pairs in RAM until the total data size is >= 1 page. Then we commit that to the disk and update the relevant metadata.

For example if we have KV pairs each of size 0.6*page_size we will keep caching them until our buffer reaches size >= one page_size. In this case it will be two KV pair with a total size of 1.2 pages. We will commit the first page and keep the remaining portion of the second KV pair (size = 0.2*page_size) in a cached buffer. We will write the remaining portion of the second KV pair when our buffer again reaches size >= one page_size.

There are two problems with this approach. The first one is that we will have to keep more metadata per KV pair to keep track of the actual location on the disk since a pair can be divided into two pages, which may or may not be contiguous in the disk (depending on how you choose the next page to write).

The second problem is that most of the times there will be some data in the buffer which could mean that a KV pair is only partially committed to the disk because the size of data in the buffer was not a multiple of page size. This might lead to a data corruption on power loss. A possible solution would be to have the size of KV pairs such that multiple pairs can fit perfectly into a single page, otherwise we will either have partially written KV pairs or wastage of space.

2. Size of key-value pair limited to size of a page:

   In the current prototype implementation we cannot have size(key-value) > flash page size. The implication is that only a limited kind of data (limited by length of keys and values) can be written to disk.

   The solution would be add metadata to keep track of a) number of pages this key-value pair is stored in, and b) direct location to each of those pages.

   If the maximum size of key-value pair is variable or unknown we could also also add a pointer at the end of every page to contain the next page's location.

   This solution could also be used in conjunction with (2) above to enable multiple pages per KV pair AND multiple KV pairs per page. The only drawback is that, on garbage collection/on reshuffling data we would have to make a lot more changes.

3. No permissions and user management:

   Anyone can read/write a key-value pair to the disk, since we do not have any metadata related to permission and user management.

   This can simply be solved by modifying the metadata in the kernel module to keep track of userIDs and associated permissions and to do a check on every operation.

4. Consistency:

   Since in out current implementation we write KV pairs as soon as possible but keep caching the associated metadata in the RAM for a while, consistency is an issue on power loss.

Meaning that, if there is an unexpected power cut the metadata might not be consistent with the data written on the disk.

There are two scenarios, we assume data is always written successfully:

(a) We have written the data on the disk but metadata is cached and there is a power loss.
In a normal magnetic disk this would just result in a loss of data (due to it being inaccessible from the index) but in our case of using a NAND drive it is a little more complicated.
There will be loss of data but we would still be able to mount the disk. On reboot our metadata would think of the previously written (but uncommitted) data pages as FREE but they, in fact, have some data written on them. Since a flash memory can only write data on FREE pages we will get an error the next time we choose that page to write.
This can potentially be solved using one of the following techniques with a trade-off of mount-time vs write-time performance..

i. On every mount scan the entire drive and if any pages are not consistent with the metadata we can update the metadata to mark those pages invalid, to be later cleaned during garbage collection.
ii. On every write, read the page to make sure that it is empty. If it is not empty then choose a new page to write on.
iii. Keep a undo log, with a hash at the end of every successful commit. On every write, first commit the changes to the undo log and then do the actual write. On mounting recheck with the last change on the undo log and make changes as necessary.

(b) We have written the data to disk but when we are writing the metadata there is power loss.
Since we only keep one copy of the metadata on disk and if we want to modify this metadata we first erase the block and then rewrite it with new one, if there is a power cut we will lose our only copy of the metadata. We will not be able to mount our FS, even though all our real data (Key-Value pairs) is intact.
We propose two solutions for this condition with a trade-off of mount-time vs write-time performance.

i. On every mount (if the metadata is corrupted) scan the entire drive to recreate the meta data from scratch. This will only work if there are no invalid pages (marked to be deleted). Since only the metadata contains this information and there could be duplicate KV pairs on disk (one deleted version and one latest version) and no way to differentiate between them.
ii. Instead of keeping only one copy of the metadata we can keep two or more.
We could reserve certain blocks for metadata and hard-code their location either in the kernel module or on the first (read-only) block of the disk.
Every metadata block can have a hash and a time-stamp at the end to mark successful commit of the block as well as to see which block is the latest one. Other than that we can write alternatively between the two blocks to make sure that we always have at-least one working/non-corrupt copy of the metadata.

8 Conclusion

For this project, we were provided a prototype implementation of a key-value store for NAND flash. The provided prototype was very limited in its functionality and performance. We solved three important limitations of the provided storage system and implemented a mechanism to perform (out-of-place) update and delete operations. We implemented a garbage collector to manage the invalid data created by the out-of-place updates and deletes and enable full flash utilization. We implemented a block selection policy for writes and garbage collection that ensures that the flash blocks do not wear out prematurely. We also introduced an indexing mechanism to maintain metadata for the stored data and cache this metadata in RAM to make our read and write operations faster. We write this metadata (and data) to the flash using an asynchronous kernel thread to ensure data persistence in the flash.

We validated the correctness of our implementation of the key-value flash store. We verify that all the read, write, update and delete operations can be performed successfully. We also verified the correctness of our garbage collector implementation and validated our wear leveling implementation. We showed that our garbage collector can help ensure full utilization of the flash and also that the wear is almost evenly spread out among all the flash blocks. We showed that our data/metadata can persist across disk remounts, with theoretically lower mount latency, significant performance and scalability advantages over the prototype.