Angel Ortiz & Zach Kennedy

Report OF Project 6

Keeping Track Of Everything

The data structures we used to keep track of the necessary information were very simple arrays, all stored in the disk structure, including the following:

1. dtf (disk_to_flash), which keeps track of the current flash page corresponding to each disk block. This makes disk reads very simple, as all that is needed is to look up the flash page in this array and read from that page.
2. flash_pages, which tracks the current state of each flash page. This includes an enumeration to tell if each page is stale, empty, or holds a disk block. The integers >= 0 are used in this array to denote the disk block that a flash page is holding. This is technically extraneous information from dtf, but it makes copying blocks for garbage collection much easier, and we need to track the state of each flash page anyways.
3. flash_blocks, which stores the current state of each flash block, including an enumeration to tell if each block is the current copy block, in a normal state, or contains stale pages. We used -1 to mark the copy block, 0 to mark a normal state with all live data and/or empty pages, and any positive integer denotes the number of stale pages currently in the block, which is used in the cleaning policy.

We will now go over a few examples of reads and writes, assuming a small example of ./flashsim with 4 disk blocks, 8 flash pages, and 2 pages per flash block. (./flashsim 4 8 2).

disk_write: block 0
Let's say that the write policy selects page 5 to write this block to. The dtf array will be updated to mark dtf[0] = 5, and the flash_pages array will be updated to denote flash_pages[5] = 0.

disk_write: block 1

Again, a flash page will be selected to write to, and dtf and flash_pages will be updated accordingly. Let's say page 4 is selected, in which case dtf[1] = 4, and flash_pages[4] = 1.

disk_read: block 0

Straightforward, we do a flash read from the flash page number stored in dtf[0].

disk_write: block 1

Before any write, we first check if the corresponding dtf[x] is already >= 0. Now, this is the case, meaning that we are updating data, and the old flash page will become stale, so we must do some extra accounting. Let's say that flash page 2 is selected to be written to. The following will occur:

1. flash_pages[4] = STALE
2. flash_blocks[3] = GARBAGE
3. dtf[1] = 2
4. flash_pages[2] = 1

disk_read: block 1

Again, super easy, just do a flash read from the flash page number stored in dtf[1].

disk_write: block 0

This will be another case where old data must be marked stale, and new data written to an empty page. Let's say that flash page 3 is selected to be written to. The following will occur:

5. flash_pages[5] = STALE
6. flash_blocks[3] = GARBAGE + 1
    a. The flash_blocks array keeps track of the number of stale pages it holds. In this case, it will be updated to a value of 2.
7. dtf[0] = 3
8. flash_pages[3] = 0

Selecting Which Page To Write

We use a random policy to select which page to write. A random index in the range of 0 <= index <= nflash_pages is chosen using rand(). Starting at this index, all flash pages are linearly checked, looping around to the start of the array, ensuring every flash page is checked. Whenever the first empty flash page is found, it is selected to be written to.

This policy is great for wear leveling, as very random writes will spread out writes very well in the long term. It is also simple to implement and understand. The tradeoff comes in a few forms:

1. It can be slow to search all flash pages linearly, in the worst case when there are no empty pages it will take O(n) time.
2. It may cause worse read times in a real flash device. Real devices are faster with sequential read access usually due to the fact that they read out an entire flash block at a time. With our strategy, it would be very costly to read sequential accesses, as it would require jumping around the flash device for each read.

When and How To Clean Blocks

The only time that blocks are cleaned are when it is necessary. This occurs when a disk_read needs to happen, but there are currently no valid flash pages that can be written to. The write policy will check each flash page, call the garbage collection function, then re-attempt the write.

The garbage collection function that we wrote will first check each block and store the block that has the highest number of stale pages. In this way, we make garbage collection more efficient, as blocks with more stale pages require less reads/writes for copying before erasure. During initialization, the last block is marked as the copy block. The copy block is always kept open, and writes are never allowed to flash pages that are in the copy block. This method ensures that we can always do garbage collection,

as we need open flash pages to copy live data out of to ensure that we can clean stale pages.

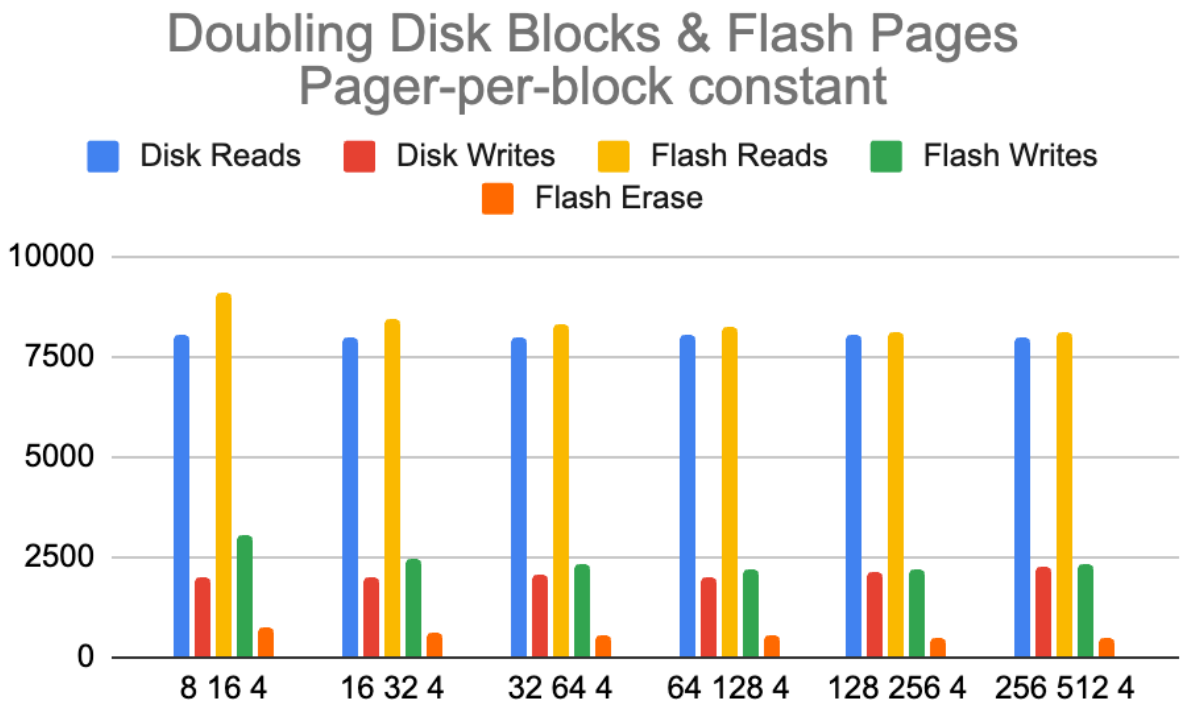
Once the garbage block to be erased and the copy block have been identified, then each live page in the garbage block is read into a buffer, and written to a page in the copy block. The dtf and flash_pages arrays are updated accordingly. The garbage block is then erased, and becomes the new copy block for the next round of garbage collection.

This works well, and is efficient as it only cleans when necessary. It will always clean the block that uses the least read/writes to copy, and will free the most flash pages, delaying the next garbage collection for as long as possible. However, it comes with a tradeoff of time, as each time we need to search for the block with the most stale pages is an O(n) linear search.

Wear Leveling

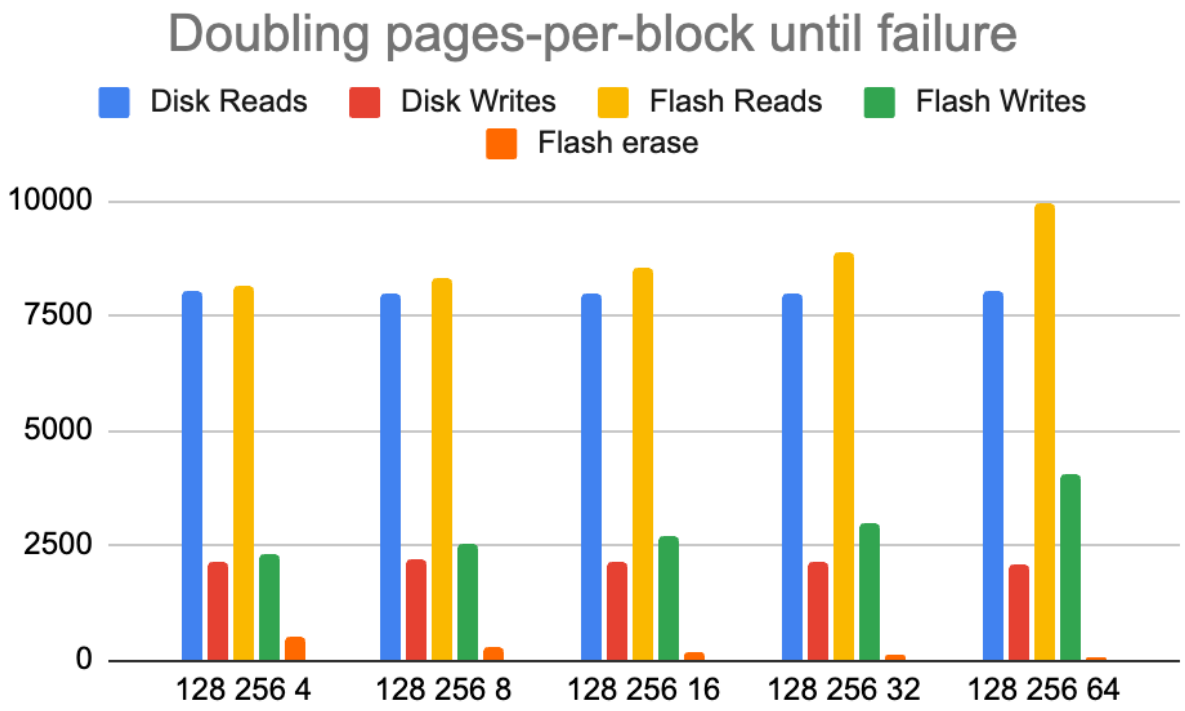
Our strategy for wear leveling involves random writes and erases, ensuring that each page has an equal chance of being written/erased at any time. When selecting a flash page to write, we start the search at a randomly selected index, meaning that any flash page is equally likely to be selected for a write. As well, during garbage collection, when searching for the block with the most stale pages, we start the search at a randomly selected block. This ensures that any blocks with the same number of stale pages are equally likely to be selected to be erased. This strategy works very well for random workloads. The tradeoff comes in the form of time, as a less random approach could utilize strategies to make writes not have to search the entire arrays to occur. As well, sequential workloads may be less efficient, as blocks that are live and not touched could make some blocks be only used very few times.

Plot 1



A low number of disk blocks means that they will be overwritten more often, leading to more flash reads/writes and necessarily more garbage collection needing to be done. This is seen in the graph, as the disk blocks and flash pages go up, the number of flash reads level out to be very close to the number of disk reads. As well, the number of erases goes down continually. Although the ratio of disk blocks to flash pages does not change, less disk blocks means that the same disk blocks will be written to more often, leading to more stale pages and more cleaning necessary.

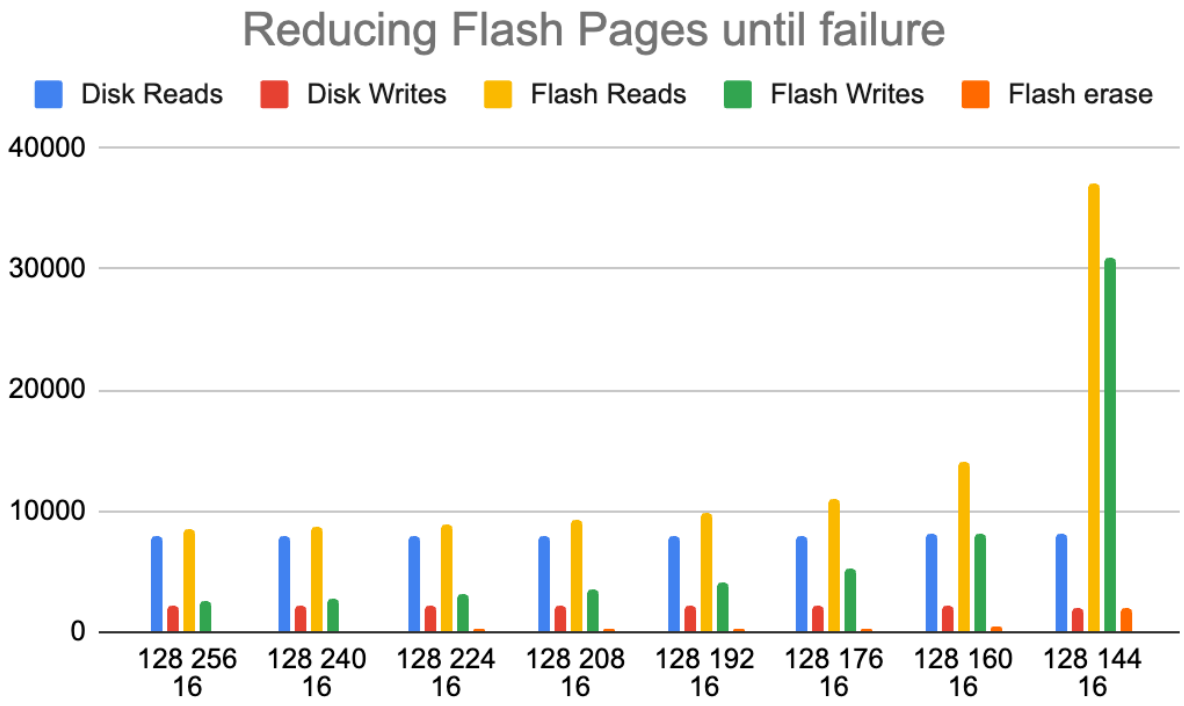
Plot 2



With larger flash blocks, every garbage collection cycle will be much more inefficient, as more pages will need to be copied. The device may need to copy a large number of live pages in order to free only a few pages, leading to many more flash writes/reads. Looking at the graph, this trend is followed, as the blocks become bigger, the number of flash reads/writes continues to increase. However, the number of flash erases does go down, as each erase is more likely to free a larger number of pages. Depending on the specifications of the hardware, it may be faster to have larger blocks. For example, if flash erases are very costly, and reads/writes are not, then larger blocks could make a system run faster, despite the number of reads/writes actually increasing.

The system fails when the number of disk blocks is equal to the number of pages per block in this case. This makes sense, as the system could write every disk block to a flash page, then not have room for any more writes due to the presence of the copy block.

Plot 3



As the number of flash pages is decreased, there are less available flash pages for stale pages to take up, leading to many more reads, writes, and erases. Looking at the graph, this trend is followed. The interesting bit is the exponential growth at the end. This would be due to the very limited number of extra blocks in the flash storage as the flash pages are reduced. For example, the last configuration only has a single extra block, meaning that after the first 128 writes, an erase is required, but only a single block is available each time, leading to very many garbage collection cycles to occur. The need to keep a block available for copying means the system cannot function with any lower amount of flash pages.