1 Designing SSD-Friendly Applications

1.1 Adopt Compact Data Structures

In this lecture we analyzed how exactly SSDs function, and deduced from that knowledge how we should write our programs in order to fully employ its capacity. On slide 13, we conclude the first paradigms for SSD friendly code:

First, we should keep in mind the page size of the SSD, which typically is 4KB. Hence, small read and write operation below this mark will have the same cost of a 4KB one, and in case of a write operation we may not fully utilize the space as the page we are writing to flash itself is not fully utilized. Thus, we should combine smaller IO operations in two fewer bigger ones, provided that these operations are sequential, which directly leads us to the second conclusion:

We should prefer compact data structures over scattered ones. This is due to the fact that we utilized the pages better and the controller of the SSD can better optimize this. Practically speaking, it is advisable to use fewer files (maybe even only one) to store our data, and in this file we should group related data into sequential sections.

Following these advises will result in more efficient IO operations and will also reduce the wear of the SSD and hence contribute to its longevity.

1.2 Designing SSD-friendly Applications for Better Application Performance and Higher IO Efficiency

This paper introduces the reader to the core functioning of SSDs and presents different optimization techniques to improve performance or longevity of the SSD. One optimization presented in the paper directly follows on from the previous discussion. The other one I chose to present analyzes the page distribution inside the SSD and its effect on performance.

Seperate hot data from cold data We have already established that we should group related data. But additionally, or rather more importantly, we should group hot data sequentially (i.e. data accessed very often) as well as cold data. The optimum for all data would be a smooth transition between these two extrema. The reason for this is very similar as before, as when accessing data from the SSD we also access unwanted data due the page size. Optimally, we also require this additional data, otherwise this data section is accessed redundantly. To maximize the chance of the first scenario, we organize our data in the proposed manner.

Avoid full SSD usage When we assume that the used pages distributed equally among the entire SSD, and the SSD is occupied by a factor of A ($A = 0.5 \implies$ half the pages are in use), then for each block on average we have $\frac{no\_occupied\_pages\_in\_block}{no\_pages\_in\_block} = A$. Thus, we see that the garbage collector will have to do way less data movement when A is low. In fact, $\frac{1}{1-A}$ blocks will need to be considered when trying to free a block on average, and also note that for each of these blocks, $A \cdot no\_pages\_in\_block$ pages will need to be copied. Thus, we face a double penalty for bigger values of A ($\frac{A \cdot no\_pages\_in\_block}{1-A}$ will need to be moved; in the paper P is used for $no\_pages\_in\_block$).