Performance of User-Level Workloads on Solid State Drives

Pulkit Kapoor

pkapoor@cs.wisc.edu

Department of Computer Sciences, University of Wisconsin-Madison

Abstract- This paper performs a detailed analysis of the performance of different user-level workloads on Solid State Drives(SSD). A pair of workloads have been created that adhere and disobey the rules mentioned in 'The Unwritten Contract of Solid State Drives' [1] paper. This demonstrates how different workloads of user applications can stress the underlying SSD, and thus can be used to better construct applications to realize robust and sustainable performance.

1. Introduction

Storage systems have evolved rapidly in recent years and modern SSDs have replaced HDDs that differ a lot in performance and reliability characteristics. Thus, the way in which user applications and file systems utilize the underlying SSD interface affect overall throughput and latency.

The goal of this paper is to perform a detailed analysis on how different workloads of user-applications behave on top of file system/SSD stack.

The following rules of the unwritten SSD contract have been analyzed in detail in the paper –

1. **Request Scale** - SSD provide a high degree of internal parallelism through different channels thus providing better random as well as sequential performance. Hence, SSD clients (user applications atop file system) should issue large or outstanding requests to exploit the internal parallelism.
2. **Locality** - SSD should be accessed with locality to reduce the translation cache misses between logical to physical page mappings.
3. **Grouping by Death Time** – SSD cannot overwrite existing data, hence garbage collection is required to reclaim invalid data. SSD clients should group requests by death time to reduce the cost of background garbage collection.

To perform the analysis, WiscSee, an analysis tool has been used along with WiscSim, a discreet event SSD Simulator [1]. The block traces from the application running atop file systems are extracted and fed into WiscSim to study and understand the performance of the last run workload.

The following paper has been organized as follows. Section 2 provides a pseudo-code on how sequential and random application workloads can be generated in C. It also provides details on how random deletion workload can be generated. Section 3 presents the Experiments which describes the intuition behind the workload selection against and in favor of the rule. It also provides detailed analysis of the performance of the SSD with applied workload using the appropriate rule measurer. Section 4 concludes the paper.

2. Workload Generation

C has been used to generate different application workloads. This section provides details on how certain common workloads have been generated in C. Most of the current applications in production today are multithreaded, hence the application workloads generated in this paper for analyzing SSD performance involve multiple threads. POSIX API's have been used to implement multithreading. This section gives an introduction on how workloads are generated for different experiments in Section 3.

2.1 Random Workload

Below is the pseudo-code to generate a random read workload.

```
void * readFile(void *arg){
        numIO = fileSize/iosize;
        srandom(time(NULL));
        for(i=0;i<numIO;i++){
                randOffset = random()% (fileSize – iosize);
                if(pread(fd,buf,iosize,randOffset) == -1){
                        fprintf(stderr,"Read Error.\n");
                        pthread_exit(NULL);
                }
        }
}
void readFiles(char *path, int iosize, int numFiles, int numThreads, int fileSize){
        for(i=0;i<numThreads;i++){
                pthread_create(&tids[i], NULL, readFile, &targ[i]);
        }
}
```

The parameters passed to the function are the path of the file, I/O size, number of files, number of threads and file size. A number of threads are created where each thread reads the chunk size specified at a random offset of a file. pread() is used to read in C as it does not update the file offset after the read operation and thus helps in random operations. The number of I/O's are request size divided by the I/O size.

Below is the pseudo-code to generate a random write workload –

```
void  * writeFile(void *arg){
        numIO = fileSize/iosize;
        srandom(time(NULL));
        for(i=0;i<numIO;i++){
                fsyncReq--;
                randOffset=random()%(fileSize - iosize);
                if(pwrite(fd,buf,iosize,randOffset) == -1){
                        fprintf(stderr,"Write Error.\n");
                        pthread_exit(NULL);
                }
                if(fsyncReq == 0){
                        if(fsyncFlag == 1){
                                fsync(fd);
                        }
                }
        }
}
void writeFiles(char *path, int iosize, int numFiles, int numThreads, int
randFlag, int fileSize, int fsyncFlag){

        for(i=0;i<numThreads;i++){
                pthread_create(&tids[i],NULL, writeFile, &targ[i]);
        }
}
```

The parameters to generate a random write workload is similar to the random read workload generation except for the 'fsync' flag. This feature allows all the modified in-core data of the file referred by the file descriptor fd, to be pushed to the storage device. This feature holds utmost importance as seen in the further section of the paper. The parameter 'fsyncReq' specifies the number of requests post which a fsync is issued to the underlying storage device. This value is passed as a parameter to the function, that allows us to create different workloads where we can vary the request size being write buffered before being pushed to the disk. pwrite() is used for random write as it does not update the file offset.

2.2 Sequential Workload

Below is the pseudo-code to generate a sequential read workload.

```
void * readFile(void *arg){
        numIO = fileSize/iosize;
        for(i=0;i<numIO;i++){
                if(read(fd,buf,iosize) == -1){
                        fprintf(stderr,"Read Error.\n");
                        pthread_exit(NULL);
                }
        }
}
void readFiles(char *path, int iosize, int numFiles, int numThreads, int
fileSize){
        for(i=0;i<numThreads;i++){
                pthread_create(&tids[i], NULL, readFile, &targ[i]);
        }
}
```

The parameters passed to the function are the path of the file, I/O size, number of files, number of threads and file size. A number of threads are created where each thread reads the chunk size sequentially using the read() system call. The Number of I/O's are request size divided by the I/O size.

Below is the pseudo-code to generate a random write workload –

```
void  * writeFile(void *arg){
        for(i=0;i<numIO;i++){
                fsyncReq--;
                if(write(fd,buf,iosize) == -1){
                        fprintf(stderr,"Write Error.\n");
                        pthread_exit(NULL);
                }
                if(fsyncReq == 0){
                        if(fsyncFlag == 1){
                                fsync(fd);
                        }
                }
        }
}
void writeFiles(char *path, int iosize, int numFiles, int numThreads, int
randFlag, int fileSize, int fsyncFlag, int fsyncReq){
        for(i=0;i<numThreads;i++){
                pthread_create(&tids[i],NULL, writeFile, &targ[i]);
        }
}
```

The parameters to generate a sequential write workload is similar to the sequential read workload generation except for the fsync flag. The parameter ‘fsyncReq’ specifies the number of requests post which a fsync is issued to the underlying storage device.

2.3 Deletion Workload

Below is the pseudo-code to generate a random file deletion workload.

```
void deleteFiles(char *path, int numFiles, int numDelFiles){
        srand(time(NULL));
        while(i < numDelFiles){
                int file = rand()%numFiles;
                char fName[10];
                sprintf(fName,"%d.txt",file);
                strcat(path,fName);
                printf("Deleting the following file %s\n",path);
                if(unlink(path) == -1){
                        printf("File deletion error.");
                }
        }
}
```

The function takes directory path, the total number of files and number of files to be deleted as the parameter. It selects a random file and unlinks it. unlink() is used to delete the file in Linux which reduces the link count to the file by 1. If the link count becomes 0, and no process has the file open, then all the resources associated with the files are reclaimed.

3. Experiments

Experimental Platform – All experiments are run on a testing machine with two Intel E5-2660 v3 10-core CPUs at 2.60GHz (Haswell EP) processors and 160GB ECC Memory. The operating system is 64-bit Ubuntu-14.04-linux-4.5.4, and the file system used is ext4. The SSD used is Intel DC S3500 480 GB 6G SATA SSD.

3.1 Request Scale

Modern SSD’s have multiple independent units that can work in parallel. To exploit this parallelism, a common technique is to send large requests that can be subsequently striped into sub-requests and send to different channels. Modern SSD’s support Native Command Queuing to concurrently process multiple requests. A

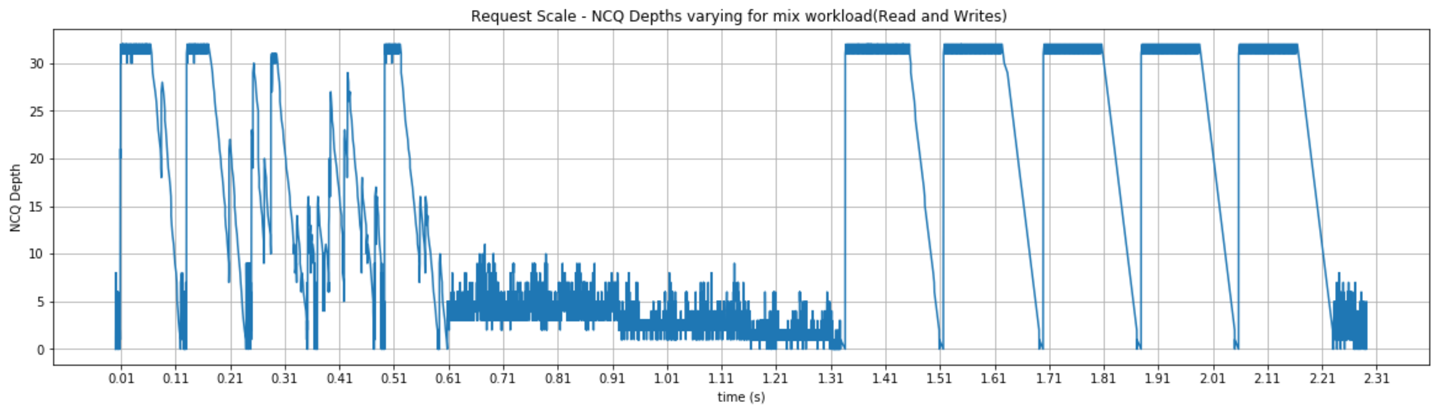

Figure 1: Request Scale – NCQ Depths for a mix workload. The lower NCQ depths are for read requests while higher ones are from write requests. The frequent draining of NCQ depths from 32 to 0 for the write workload is seen due to fsync() calls after every 10 requests. The workload corresponding to time interval 0.01s – 0.61s is random write with fsync() calls after every 10 requests each of size 512KB. The following time interval between 0.61s and 1.31s corresponds to random read workload. The time interval between 1.31s and 2.21s corresponds to sequential write workload with fsync() calls after every 10 requests each of size 512KB.

typical maximum queue depth of modern SATA SSD is 32 requests.

Following is the rule from the unwritten contract –
Request Scale*: SSD clients should issue large data requests or multiple concurrent requests. [1]*

3.1.1 Intuition

Frequent data barriers in the applications reduce request scale. A barrier has to wait for all previous requests to finish. While waiting, SSD bandwidth is wasted and hence, an application workload with frequent fsync() or read() calls will reduce the number of requests that can be concurrently issued.

3.1.2 Experiment Setup

The underlying SSD used for this experiment is of size 512MB with ext4 file system running on top of it. The NCQ depth has been set to 32 requests.

Preparation Workload

100 4MB files have been created using a chunk size of 128KB and 10 threads.

Actual Workload

Workload which adheres to the rule – Sequential writes are done in 100 files with small I/O size of 4KB using 10 threads.

Workload which disobeys the rule – Sequential writes are done in 100 files with small I/O size of 4KB using 10 threads. Frequent fsync() calls have been introduced in the code after every 10 requests each of size 4KB.

Mix Workload

A mix workload of write and read requests is also generated by alternating writes in 50 files with chunk size of 512KB and small reads in 50 files of 32KB chunk each. The reads are done using 5 threads while writes are done using 10 threads each. fsync() has been introduced in the write calls after every 10 requests of 512KB each. The first set of write requests generated are random overwrites and the next set of write requests are a sequential workload. The mix workload experiment is done on a SSD size of 1024MB with 100 files each of size 8MB. Figure 1 shows the results for this experiment.

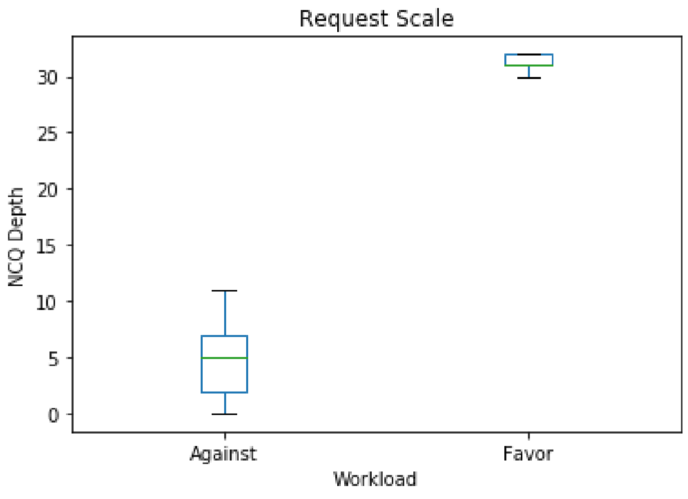

Figure 2: Request Scale Boxplot – Distribution of NCQ Depth. The X-axis indicate the I/O patterns – against the rule and in favor of the rule. Inside each panel, the top and bottom border of the box show the third and first quartile. The heavy lines in the middle indicate the medians. The whiskers indicate roughly how far the data points exceed.

Using fsync()	Workload Duration (s)	Max Request Size
No	0.518331	1 MB
Yes	0.847506	40960 KB

Table 1: **Request Scale –** Workload Duration and Maximum request size for both the I/O patterns – against the rule and in favor of the rule.

3.1.3 Results

Figure 2 shows the distribution of NCQ depths for I/O workloads – one in favor of request scale and one against the rule. Multiple concurrent write requests can be buffered, generating large write requests. These requests can be striped to multiple sub-requests, that generates a large request scale.

However, introducing synchronous data-related system calls like fsync() results in maximum request size of 40960KB being generated (10 requests of 4KB each post which a fsync() is done). This reduces the number of requests that can be concurrently issued and hence, lower NCQ depths. Table 1 shows the workload duration and maximum request size for both the workloads – one involving barrier and one without barrier.

Figure 1 shows the variance of NCQ depths for a mix workload. The high spikes seen in the plot are write requests which generally can be buffered and enlarged. However, read requests cannot be batched or concurrently issued due to dependencies. Thus, the scale of read requests is low. The left part of the plot depicts a random write workload and right part depicts a sequential write workload, both with a chunk size of 512KB and fsync() calls after 10 requests. fsync() calls drain the NCQ depths from 32 to 0. Sequential write requests provide better buffering and thus can be enlarged, hence large request scale and higher NCQ depths. The mid area in the plot depicts a random read workload that shows low NCQ depths for reasons mentioned above.

3.2 Locality

SSD does not allow in-place updates, hence a translation mapping is needed between logical and physical pages. More the number of pages in SSD, more the memory required for translation cache that stores these dynamic mappings. On-demand FTLs stores the mapping in flash and cache them in RAM, hence reduces the amount of RAM required for mappings. The mapping is loaded into memory only when it is required and evicted to make room for other translations. Thus, pages should be accessed with locality for this translation cache to work.

Following is the rule from the unwritten contract –
***Locality**: SSD clients should access with locality. [1]*

3.2.1 Intuition

Small reads at random offsets in a file will result in more translation cache misses because of less probability of the required mapping to be in the cache. As we increase the I/O size/chunk size, more data will be accessed in each request thereby increasing locality and a higher probability of required translation mapping to be present in the cache. Sequential reads will have better locality than random reads and hence result in fewer cache misses.

3.2.2 Experiment Setup

The underlying SSD used for this experiment is of size 256MB with ext4 file system running on top of it. Cache has been set to cover 1%, 2%, 5%, 10%, 50% and 100% of the logical space. Entire SSD space has been utilized to study the results better for this experiment.

Preparation Workload

100 2MB files have been created using a chunk size of 128KB and 10 threads.

Actual Workload

Workload which adheres the rule – Sequential reads are made to files with a chunk size of 512B, 1KB, 16KB, 32KB and 64KB using 5 threads.

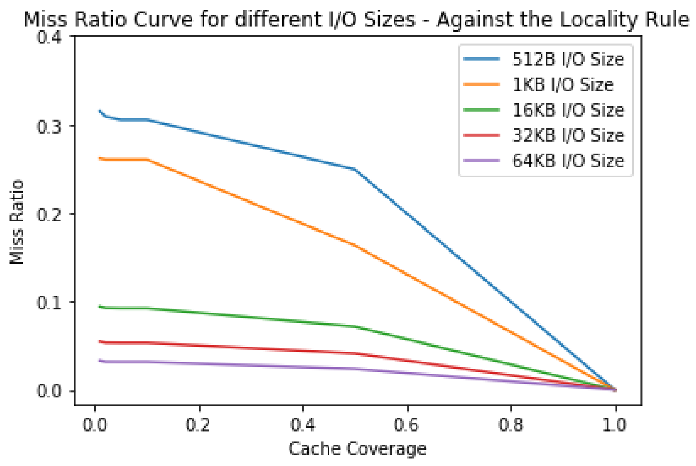

Figure 3a: **Against the Locality – Miss Ratio Curves for Random Read Workload.** Maximum observed Miss Ratio is 0.315.

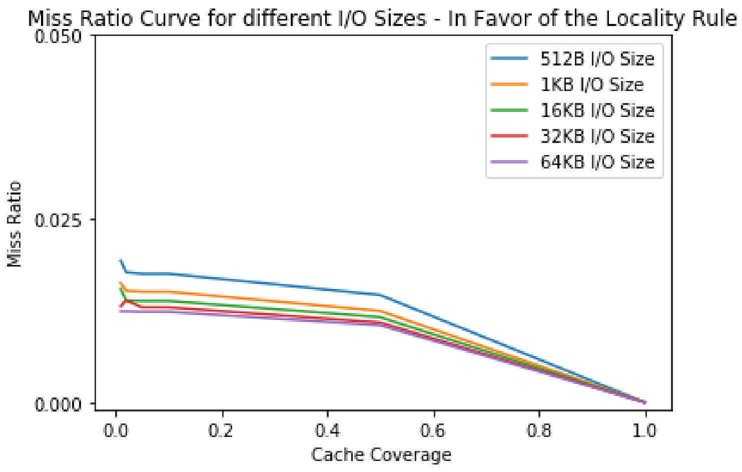

Figure 3b: **In favor of Locality – Miss Ratio Curves for Sequential Read Workload.** Maximum observed Miss Ratio is 0.019.

Figure 3: The X-axis indicate the cache coverage and Y-axis indicates the Miss Ratio. Cache has been set to cover 1%, 2%, 5%, 10%, 50% and 100% of the logical space. The I/O size is inversely proportional to the miss ratio as increasing the request size will result in better locality and hence, less cache misses.

Workload which disobeys the rule – Random reads are made to files with a chunk size of 512B, 1KB, 16KB, 32KB and 64KB using 5 threads each.

3.2.3 Results

Miss ratio curves have been used to show the results of this experiment obtained from WiscSim with an on-demand page-level mapping scheme. Figure 3a shows a random read workload which accesses data of different chunk sizes at random offsets in a file. This results in more cache miss due to less locality. As increasing the request size will result in more probability of the required mapping to be in the cache, we see fewer cache misses with increasing I/O size. The same is observed in the plots. Figure 3b

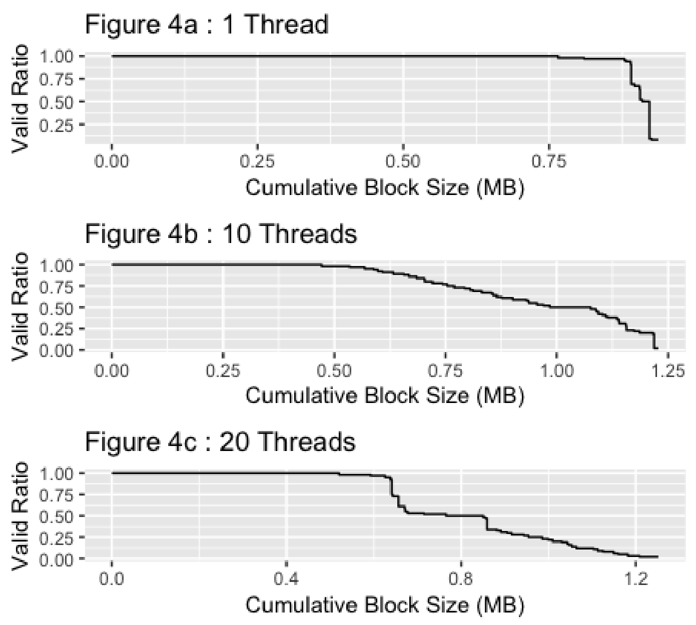

Figure 4 – **Large Sequential Write Workload – Zombie Curve**: The X-axis indicate cumulative block space and Y-axis indicates the Valid Ratio. The curve goes beyond 1 in Fig 4b and Fig 4c, which implies that the total size of occupied blocks is more than the logical space size. The area under curve 4a is more than 4c which implies almost all of the data on logical space is considered valid by the SSD, unlike the case in Fig. 4b and Fig. 4c.

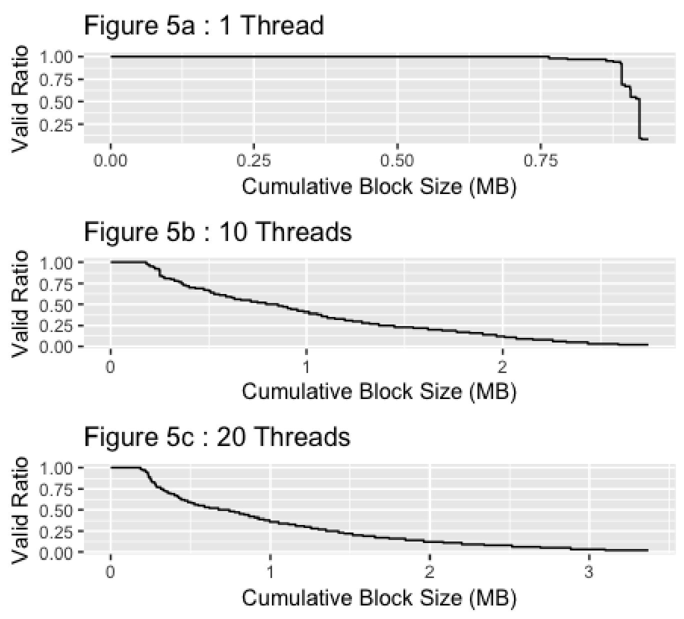

Figure 5 - **Small Random Write Workload – Zombie Curve**: The X-axis indicate cumulative block space and Y-axis indicates the Valid Ratio. The curve goes far beyond 1 in Fig 5b and Fig 5c, which implies that the total size of occupied blocks is a lot more than the logical space size. The area under curve 5b and 5c are too less which implies a small amount of valid data in the simulated SSD.

shows the miss ratio curve for a sequential read workload. Since, sequential access maintains locality, we see very few cache misses in the plot.

The read duration for the sequential read workload with chunk size of 64KB (Fig 3a) was observed to be 0.17s for 50% cache coverage while the random read workload with I/O size of 512 bytes provided the workload duration of 1.36s for 50% cache coverage.

3.3 Grouping by Death Time

Zombie window is the time window between the first and last invalidation of a page within which both live and dead data reside in the block. Such blocks are called *Zombie blocks [1]*. Large zombie windows result in higher probability of a partially valid/zombie block to be selected for garbage collection and incurring costly data movement. This is required as FTL must move the live data in a block to a new block and erase the victim block to free out SSD space. Zombie Windows can be reduced by combining similar death time data in a block. It can be achieved using 2 ways - Grouping by order which requires the host to reorder writes such that data with similar death times are placed in a write sequence. The second option is to place different death groups in different logical segments. Clients of segmented FTL's can group by order and space.

An ideal workload should be shown as a vertical cliff in the zombie curve, indicating zero partially valid blocks, thus simplifying the task of the garbage collector that can simply reuse these erase block without costly data movement.

The workload that violates the rule will have a large and long tail indicating multiple zombie blocks. If available flash space is limited, the garbage collector has to move data in zombie blocks to make free space.

3.3.1 Intuition

Mixing files with different death times in a flash block and deleting a partial set of these files will result in more zombie blocks and hence more victims for background garbage collection. Also, more the parallelism, more data with different death time will be mixed in a flash block.

However, creating files that fill multiple flash blocks completely through sequential workload and deleting a partial set of those files, will result in few zombie blocks being created and less overhead on the garbage collector.

3.3.2 Experiment Setup

The workloads are run on WiscSim for a long time and frequent snapshot of the valid ratio of used flash blocks are taken. The workloads are run for long time forcing the SSD to trigger garbage collection. The valid ratios provide important insight on zombie (partially valid) blocks that result in major overhead during background garbage collection.

The underlying SSD used for this experiment is of size 128MB with ext4 file system running on top of it. Size of each flash block is 128KB with a non-segmented FTL.

Actual Workload

Workload which adheres to the rule - 120 files each of size 1 MB are created with a chunk size of 512KB each. Sequential overwrites are done to these file in chunk size of 512KB followed

Threads	Workload Duration (s)	Write Bandwidth (MB/s)
1	0.84952	125.95
10	1.22472	118.39
20	1.39122	117.16

Table 2 – **SSD Performance – Large Sequential Write Workload**: The table indicates the duration and write bandwidth for a large sequential workload specified in Figure 4. As multithreading increases, it leads to more intermixing of dissimilar death time data in a block, which suggests need for garbage collection.

Threads	Workload Duration (s)	Write Bandwidth (MB/s)
1	0.8633	125.10
10	10.7178	32.28
20	14.85568	29.61

Table 3 – **SSD Performance – Small Random Write Workload**: The table indicates the duration and write bandwidth for a random write workload specified in Figure 5. As multithreading and randomness increases, it leads to more intermixing of dissimilar death time data in a block. This suggests need for background garbage collection and hence relatively poor write bandwidth.

by random deletion of 30 files amongst the total. This workload is chosen as it will avoid intermixing of different files in the flash block (if done with minimum parallelism as seen in the results section). Once, a file is deleted, it will empty the flash block which can be reclaimed by the garbage collector without any costly data movement.

Workload which disobeys the rule - 120 files each of size 1 MB are created with a chunk size of 1KB. Random overwrites are done to these files in chunk size of 1KB followed by random deletion of 30 files amongst the total. This workload is chosen as it will lead to intermixing of different files in the flash block.

3.3.3 Results

Zombie curve analysis is used to study group by death time rule. As the number of threads increase (Figure 4 and Figure 5), it gives rise to parallel writes and thus results in more intermixing of data that die at different times within a flash block.

Large sequential and small random write performance is identical to this rule in case of 1 thread as shown in the Figure 4a and Figure 5a. This is because, in case of 1 thread, random writes can be buffered and written sequentially, which minimizes the mixing of data from different files with different death times within a flash block. However, with increasing parallelism, sequential writes outperform random workload. As seen in Figure 4, files accommodate multiple flash blocks completely with minimal intermixing in sequential workload, hence the valid ratio stays longer at 1. This is followed by overwriting of files again with a chunk size of 512KB and deletion, thus freeing the occupied flash blocks completely (vertical cliff in Fig. 4a).

Figure 5b and 5c depicts a workload that violates the rule. Files are created with small chunk size and multiple threads that lead to mixing of files within a flash block. Further, files are overwritten randomly with a small chunk size that mixes the data within a flash block. Following this, we delete 30 files randomly. This results in a large and long tail due to many zombie blocks left behind.

4. Conclusion

SSD performance depends on a number of things: application workloads as well as the interaction between the file systems and SSD interface. In this paper, a set of example application workloads have been provided that adhere to the rules mentioned in the unwritten contract as well as violate the mentioned rules. This simulation-based analysis provides an insight into application level problems that might impact the overall throughput and latency of the system. This analysis can help in better design and optimize the application workloads to achieve a better immediate and sustainable performance from the underlying SSD and could benefit workload analysis on a simulated SSD environment.

5. References

[1] The Unwritten Contract of Solid State Drives Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

[2] Filebench. https://github.com/filebench/filebench

[3] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Operating Systems: Three Easy Pieces. 0.91 edition, May 2015

[4] Linux Generic Block Layer. https://www.kernel.org/doc/Documentation/block/biodoc.txt.

[5] C Reference Manual - Dennis M. Ritchie https://www.bell-labs.com /usr/dmr/www/cman.pdf

[6] ext4 Wikipedia - https://en.wikipedia.org/wiki/Ext4

[7] Rosenblum, M. and Ousterhout, J. The Design and Implementation of a Log-Structured File System