# Android Application Context Aware I/O Scheduler

by

Sivasankaran Jeevan Prasath

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved September 2014 by the
Graduate Supervisory Committee:

Yann Hang Lee, Chair
Carole-Jean Wu
Aviral Shrivastava

ARIZONA STATE UNIVERSITY

December 2014

## ABSTRACT

Android has been the dominant platform in which most of the mobile development is being done. By the end of the second quarter of 2014, 84.7 percent of the entire world mobile phones market share had been captured by Android. The Android library internally uses the modified Linux kernel as the part of its stack. The I/O scheduler, is a part of the Linux kernel, responsible for scheduling data requests to the internal and the external memory devices that are attached to the mobile systems. The usage of solid state drives in the Android tablet has also seen a rise owing to its speed of operation and mechanical stability. The I/O schedulers that exist in the present Linux kernel are not better suited for handling solid state drives in particular to exploit the inherent parallelism offered by the solid state drives. The Android provides information to the Linux kernel about the processes running in the foreground and background. Based on this information the kernel decides the process scheduling and the memory management, but no such information exists for the I/O scheduling. Research shows that the resource management could be done better if the operating system is aware of the characteristics of the requester. Thus, there is a need for a better I/O scheduler that could schedule I/O operations based on the application and also exploit the parallelism in the solid state drives. The scheduler proposed through this research does that. It contains two algorithms working in unison one focusing on the solid state drives and the other on the application awareness. The Android application context aware scheduler has the features of increasing the responsiveness of the time sensitive applications and also increases the throughput by parallel scheduling of request in the solid state drive. The suggested scheduler is tested using standard benchmarks and real-time scenarios, the results convey that our scheduler outperforms the existing default completely fair queuing scheduler of the Android.

*To all who supported me*

## ACKNOWLEDGEMENT

I would like to take this opportunity to thank my Professor Dr.Yann Hang Lee, for his support, without which my thesis would not have been possible. It is because of his guidance and support that I was able to complete my research. I would also like to thank Professor Dr.Carole-Jean Wu and Dr.Aviral Shrivastava for their support. I would also like to thank my friend Mr.Kuan Hao Ken Chen for his support in setting up the lab equipment. Finally, I would like to thank my parents, for their motivation and encouragement to pursue this degree.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

# Chapter 1

# INTRODUCTION

The Linux kernel's I/O scheduler is designed for the hard disk drives that are used in a desktop environment. It is not optimized for solid state drives and does not have any specific implementations that would cater the need of the Android devices. But Android utilizes the Linux I/O scheduler for scheduling the I/O operations. This will result in under utilization of the potential available in the solid state drives and not being able to meet the needs of the Android system, of mobile devices. The above situation gives scope for changes and enhancements to be done in the kernel in order to make it best suited for the mobile and Android platform.

## 1.1 Motivation

**Demand for Faster I/O Operations:** The demand for a quick response and interaction by Android devices has always existed. The delay in loading a game or a lag while playing a video, resulting in not meeting the expected quality of service (QoS) is not acceptable to the end user. Hence many approaches are taken to improve the performance of the Android device, they include introducing high end processors, switching to solid state drives etc.,

**I/O Scheduler and Solid State Drives:** It is seen in the past that the I/O processing systems could be a source of concern in real-time applications. The I/O operation's performance is the combined effect of many things like the latency of the memory device, the speed of the processor, the interface bandwidth and so on. Through solid state drives the devices have become faster, as they have a faster access to the memory blocks than the traditional hard disk drives. The I/O scheduler is the time controlling valve in the communication between the memory device and the processor. The I/O scheduler software that exist in the present Linux kernel, is not designed with the solid state drives architecture in mind. They are designed and optimized for the hard disk drives [19]. This leads to a need for a new I/O scheduler, that could be used to enhance the throughput by exploiting the features that are present in the solid state drives.

**Application Awareness in Scheduling:** Consider a scenario, in which the Android is doing a background work of downloading an update or it is fetching a huge chunk of database from the memory, at the same time the user tries to play a stored video from the device. The present Linux I/O scheduler, the Completely Fair Queuing (CFQ), has the implementation such that the background process gets all the resource bandwidth it requests. This could lead to a lag in the video watching and reduce the QoS to the end user. The present Linux as mentioned above does not distinguish the applications based on their I/O operations. It does the allocation of the I/O bandwidth such that it can maximize the system throughput. This could result in the degradation in the performance of the time sensitive application [12]. The present implementation of the I/O scheduler contains a queue per process request, and each of these queues is serviced in a round robin manner. It works on the idea of maximum throughput and fair allocation of the disk resource. The CFQ scheduler is not aware of the characteristics of the application requesting the I/O operations and therefore there is a possibility of the time sensitive application to stall. Hence, there is a need for an I/O scheduler that incorporates information about the applications in its decision on I/O scheduling. This approach would enhance the responsiveness of the Android towards the end user.

# Chapter 2

# BACKGROUND

## 2.1 Fundamentals of Flash Memory

The advancement in the field of NAND-flash technology and the reduction in the manufacturing cost, have seen the deployment of the solid state drives in many fields. Solid state drives are gaining significance in the data server systems, desktops and the hand held devices. The usage of the solid state drives in the tablet devices has also seen a significant rise. There are two types of flash devices NAND and NOR. NOR is used for byte level random access while NAND is used for block-level random access. NAND is denser and lower in cost and hence used in more general applications. Our focus in only on the NAND devices [8].

NAND devices come in single-level cell (SLC), and multi-level cell (MLC) technology, where the naming is based on the bits that could be stored in a single cell. The basic unit of the NAND device is the NAND string. A NAND string is made up of 64 to 128 cells in series. It contains two selection transistors that provide the source and the connection to the bit line. The bit lines are connected to each cell's control gates. The NAND strings sharing the common bit lines constitute a block. In order to increase the density of the devices, the flash devices are manufactured by packaging several chips together. A package has the same I/O bus with separate chip select and read/busy signal for each chip. Every chip is composed of two or more dies. Each die contains several planes. Furthermore, every plane is comprised of thousands of flash blocks. Each flash block has around 64 to 128 pages, and each page is of a size of 4 Kilobyte,8 Kilobyte or 16 Kilobyte. A total of 8192 or more flash blocks makes an NAND array [11].

### *2.1.1 Solid State Drive Architecture*

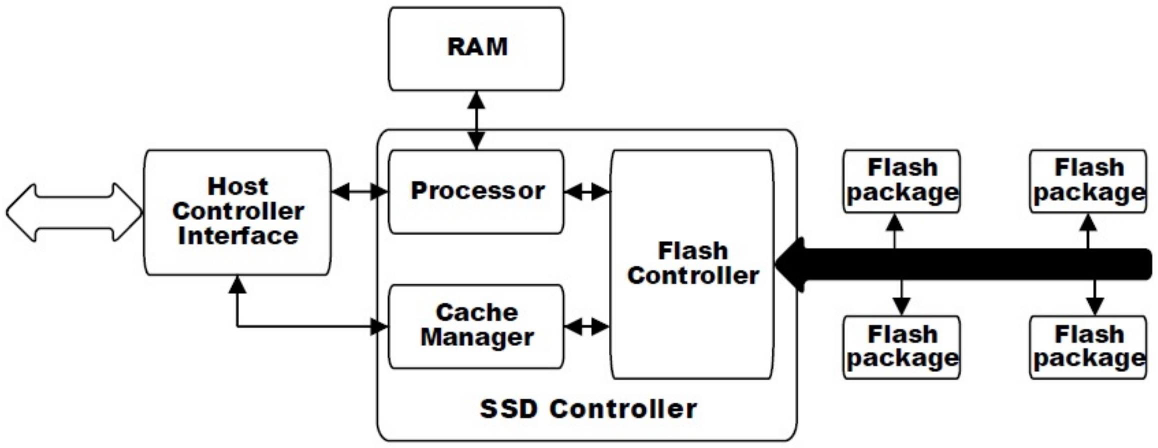

**Figure 2.1:** An Solid State Drive Architecture Adapted from [26]

A basic solid state drive is made of four major components namely (i)the host interface (ii)solid state drive controller (iii)DRAM or SRAM buffer (iv) Multiple channels [9].

**Host Interface:** The interaction between the solid state drive and the outside world is established through this interface. The interface is one of the connecting mediums that would be available like PCI express, SATA, USB or Fiber channel. The host interface has a critical role in the performance of the solid state drives. It is often seen as a bottleneck point, especially the bandwidth supported by the interface is expected to match that of the flash array. The host interface also contains command decoding circuit that is sent by the host (operating system) that handles the data flow from and to the flash memories.

**Solid State Drive Controller:** This plays a crucial role in the solid state drives. The roles performed by it are as follows: it act as a disk emulation to enable solid state drives replicate the hard disk drives. It contains a buffer manager that is responsible for maintaining the pending and serviced requests. The multiplexer of the flash is responsible for sending the commands and receiving the data from the flash packages. The important block that enables the solid state drives to function, as a magnetic disk is the flash file system. When the host requests a sequential access of memory that extends to multiple sectors forming a file. The solid state drives maintains it as a link list, which the host uses to form a file allocation table for itself. The flash file system is implemented as a firmware inside the controller with specific operation to each sub layer. The basic operations include the wear level management, garbage collection and block management. Wear leveling phenomenon is needed in order to maintain the long life of the blocks. The NAND flash devices have a life cycle that would last 10K to 100K writes and therefore it is necessary to have a management or controller to relieve the stress on the frequently accessed blocks. In order to mitigate the risks of the frequently accessed blocks, it is necessary to uniformly distribute the aging process of the blocks. The maximum life cycle of the blocks must also be considered while making this decision. The wear leveling techniques uses the logical address to physical address translation in a way to reduce the risk of the blocks. Whenever a host application requests updates to the same sector, the controller maps the sector to a different physical sector, and the obsolete copy is tagged as invalid and erasable. In this way, aging is maintained uniformly across all the blocks. The are two types of wear leveling techniques that have been adopted. The dynamic wear leveling technique refers to the process, where the first available erased block with the least erase count is used to write the new block. The static wear leveling technique, refers to the process wherein every block is eligible for the erase as soon as it's age deviates the average value coded. Garbage collection refers to the process in which the solid state drives get the free sectors even when the available free sectors fall below a threshold value. The wear leveling techniques need the free sectors for it to write the request, but when the availability of those free sectors falls below the threshold value they are compacted, and multiple obsolete copies are deleted. The garbage collection layer selects the blocks that have invalid sectors, copies the valid copy onto that free sectors and erases those blocks. As a performance improvement, the garbage collection is handled in the background. The bad block management is the sub layer wherein the solid state drives maintain a map that contains the list of bad blocks. These are the bad blocks that cannot be used further, for the usage of those blocks would not return a guaranteed result. The map in this is updated right from the testing phase and continues to be updated for the entire life time of the solid state drives based on its usage. Another unit that is present in the controller is the error correction code. The most popular codes used are the Reed-Solomon and the BCH codes, used for the error correction of the disk requests.

**SRAM/DRAM Buffer:** Based on the type of solid state drives, it contains a dedicated DRAM buffer or low cost SRAM. It is integrated in the controller to hold the data. Before they are being written to the flash or being transferred to the host.

**Multiple Channels:** In most of the solid state drives there exist multiple channels ranging from 2 to 10, used to connect the controller and the flash packages. The number of packages that share one channel may vary based on the solid state drives implementations.

### *2.1.2 Flash Commands and Basic Operations*

The solid state drives have three main functions read, write and erase. As the name suggests the read fetches the data from the flash, the write programs the data on to the flash and the erase resets all the target blocks to 1. The operations are executed by placing the correct code in the command register along with the address of the block that needs to be accessed in the address register. The address contains six segments: chip, address, die, plane, block and page. Page represents the least significant bits. In flash, there is one restriction that within a block, the pages must be programmed sequentially in the ascending order of the page address. Solid state drives also support few extra commands these are the extension of the basic commands.

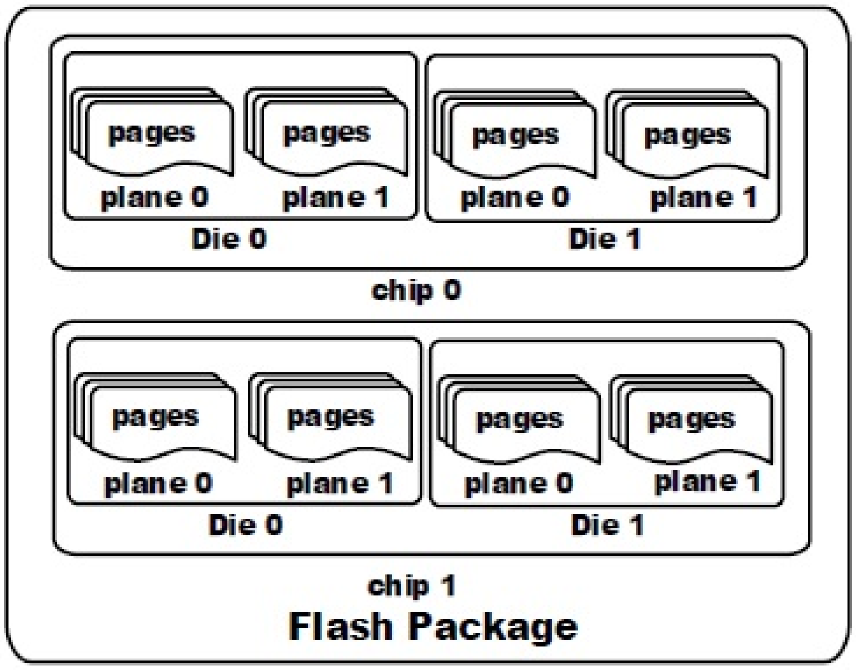

**Figure 2.2:** A Layout of the Solid State Drive Structure Adapted from [9]

**Internal Data Move:** Also called as the copy back, this command is used to move a data page from one location to another within the same plane without accessing the I/O bus. The restrictions are that the source and the destination page must be on the same chip, die and plane. The address of the source and the destination page must be either odd or both even.

**Multi Plane:** This command provides the option of multiple reads, write or erase operations on multiple planes of a die. The cost incurred is only of one operation. The constraint in the command is that the Multiplane can be executed only on blocks that have the same chip, die and block address.

**Interleaving:** This command is used to execute multiple reads,writes and erase operations on multiple planes of different dies that are on the same chip. The only restriction of this command is that the command is executed only on pages of different dies belonging to the same chip.

### *2.1.3 Parallelism in Solid State Drives*

The internal architecture of the solid state drives suggests that many levels of parallelism are possible with the solid state drives. Further the operations can be interleaved or paralleled at each level. The list of the parallelism possible and that could be exploited [13].

**Channel Level Parallelism:** In flash devices, the packages are connected through channels, and few high end solid state drives have error correction and flash controller for each channels in-order to boost its performance. Each of these channels can be operated in parallel or individually.

**Package Level Parallelism:** Making the packages that share a channel to be operated independently can optimize resource utilization. Interleaving of the opera-

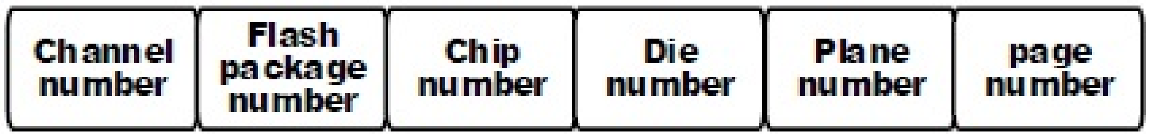

**Figure 2.3:** Addressing Format Within a Solid State Drive Adapted from [9]

tion between the packages of the same channel is also possible with the solid state drives.

**Die Level Parallelism:** In order to increase the throughput, each die of the package can be operated individually. This could be achieved in the flash devices by placing the appropriate die address and requesting data from the exact die.

**Plane-level level parallelism:** Chips are composed of multiple planes. High-end flash memories support performing similar operation on multiple planes simultaneously. There are some flash devices that provide caches to increase the parallelism in all levels.

## 2.2 Details on Android

The Android platform has three main components that are of our concern the Java application, the Android middleware and the Linux kernel [27]. Each Java application has its own Dalvik virtual machine and it runs in this instance. Every application has its own process identification number to differentiate it from other applications and platform threads. The middleware contains a list of libraries and services that are used by these applications. The middleware manages the life cycle of the application; it provides Java native libraries that could be used by the applications to access the device specific features and facilities. The middleware provides functionality like the activity manager controls the existence of the application. It is also responsible for services like the media service that is used to play videos on the mobile, the location manager for the GPS updates and so on. The application uses the Android inter process communication mechanism called the binders to communicate to these system services. An example would be the playing of the video in the mobile phones, the application requests the service manager for using the media service. The service manager is a process that maintains a service list available in the system. It is responsible for granting services to the application based on the requests and its availability. Once the request is placed by the application the service manager checks the media service availability and grants the application the permission. Upon receiving the permission the Android application informs the location of the video to the media player service, along with its type through a binder. It is the media service that plays the video by decoding it based on its media file type.

### *2.2.1 Foreground and Background Processing*

The Android application can run either in foreground or in background. The tasks that run in foreground are generally user interactive tasks.The background task does not need much interaction with the end user. The middleware is aware of the list of application that run in background and foreground, this information is passed on to the process scheduler and memory manager of the Linux kernel. The kernel has the CFS (Completely Fair Scheduler), as its process scheduler and it provides higher priority to the application that are running in the foreground when compared to the background. The Android memory management is used to provide memory by removing the background tasks in case there is a shortage of memory for the foreground. These techniques are not extended to the I/O scheduling.

## 2.3 I/O Scheduler

Input Output (I/O) scheduler is a piece of software that decides the ordering of the input and output block requests and when they are to be submitted to storage devices. It is a part of the kernel of the operating system. It is also referred to as disk scheduling. The I/O scheduler has to implement variants of the elevator algorithms in order to minimize the large seek time that is caused due to the random requests made to the disks. The I/O scheduler has the following responsibilities.

**Merging and Sorting Block Requests:** The main function of the I/O scheduler is to merge and sort the disk request. This is important owing to the nature of the disk and the way the data is organized in it. The seek time in disks is one of the bottlenecks in retrieving or writing data. It is highly affected by the seek distance and the moving speed of the disk arm. The merging and the sorting are needed in order to reduce the disk latency that is being caused due the movement of the disk arm. Sorting requests reduces this movement by making the disk arm service requests that are close to each other.

**Priority of Requests:** Not all the requests that are placed to the disk have the equal urgency, but it is important that all must be serviced within certain time. This gives the leverage for the kernel developers to prioritize the request and service them in the order that is best suited for the operating system. This prioritization is done based on certain parameter like the deadline for request, the priority of the task that is requesting the request, the availability of the bandwidth etc. All these decisions are made in the I/O scheduler.

**Bandwidth Sharing:** This bring fairness in the scheduler. It is the expectation that each request to the disks will be serviced immediately, but it is not feasible. It is necessary to make sure that all the process gets its share of the disk bandwidth. These I/O schedulers handle the responsibility of maintaining a balance in meeting all the demands of the processes and distributing the available bandwidth proportionally.

**Avoiding Starvation:** Another important responsibility of the I/O schedulers is to avoid starvation of the process that request for the disk data. In general this is implemented by the I/O schedulers by assigning each request with a default deadline. This deadline is the maximum time within which the I/O scheduler must service the request. By assigning this deadline to each request, it is taken care that each request gets serviced within that stipulated time.

### *2.3.1 Basic Types of I/O Scheduler*

**FCFS:** First Come First Serve [1], the most basic form of an I/O scheduler this scheduler has no specific function. It queues requests and services them one after the other. This type of service handling suffers badly in performance because it does not try to reduce the seek time by merging or sorting the requests. This is better suited for non rotational-disks like flash devices, but FCFS as an I/O scheduler is not used extensively.

**LIFO:** Last in first out I/O scheduler, it takes the most recent request and services them. The only advantage that could be seen is that, it could be used to service a sequence of reads at one time taking into account the locality factor. It runs into a high risk of starvation if there are highly intensive I/O operations.

**SSTF:** Shortest service/Seek time first seek time is the measure of the disk arms latency to settle at the correct track of the request to be serviced. Seek time depends on various factors like the distance the arm needs to move, the mechanical characteristics of the arm and the size of the disk. The SSTF algorithm tries to service the requests whose service/seek time is the lowest. That is from the present location of the disk arm, the next shortest distance that it has to travel to service the next request. Now for example,if there are three requests that needs to be serviced. Their track numbers are 34,55,62 respectively, assuming that the disk arm is at location 55 and still 34 and 62 are not serviced. Then as per the SSTF algorithm the track 62 will be serviced as it is only seven tracks ahead of the 55. This algorithm requires to always maintain the information about the disk arm. It is good if the requests show the property of temporal locality. But it causes starvation for the random requests and when the request patterns are located much farther apart.

**SCAN:** The elevator Algorithm, as the name suggests follows the servicing pattern of the elevator. All the disk requests that are coming in one direction are serviced first and then the disk arm changes its direction to service the request in reverse order. This algorithm is much better when compared to the previously discussed algorithm in servicing the disk request. Conditions are that the disk arm moves only in one direction, servicing all the request of that direction. If there is no request in the current direction then the direction of the arm is reversed. This scheduler also has some drawbacks if the requested block is in the track just traversed or just close in the other direction, it is not serviced immediately. It will be serviced only when all the requests in the current directions are serviced and then the disk arm reaches the requested track in the reverse direction after servicing the requests in between. It is more favored to the innermost tracks, the outermost tracks and the jobs that are arriving at the latest in the current direction.

**C-SCAN:** The circular scan is a variation of the SCAN algorithm. In this algorithm, the servicing of the request always follows a single direction. It scans in one direction and on reaching the end moves its disk arm to the other end and moves in the same direction. This algorithm eliminates the unfairness that was caused in the previous elevation algorithm wherein certain tracks had a probability of being serviced two times. This algorithm is fairer and it reduces the request time for those blocks, which are on the edges.

**F-SCAN:** The SCAN and SSTF causes the problem of arm stickiness. When there is a burst of requests for the same track, all the above mentioned algorithms will stop the progress of the disk arm. It would service the request with zero arm movement. This would yield unfairness to the tracks that are positioned in the edges or the other sides. In order to avoid this starvation the F-SCAN has two queues, where once the first queue is filled with the sorted requests, the next bunch of requests are placed in the other smaller queue. Only when the first queue is serviced the next queue is serviced, similar to double buffering. This would make the arm non-sticky, the disadvantage is that the disk on the other queue has to wait till the first queue is serviced.

**N-SCAN:** Similar to F-SCAN, that had two queues, the N-SCAN contains N queues of fixed smaller sizes and all the queues are serviced one after the other. Such an implementation guarantees servicing of requests and avoids starvation.

**LOOK:** This is similar to the elevator algorithm, with one addition that the algorithm sees if there are further requests in the current direction. If there are no requests then it reverses its direction from the current position and starts servicing the requests that are in the opposite direction. By doing this it reduces, the time taken for the disk arm to reach the latest request and the edge of the disks. LOOK is similar to the SSTF algorithm with the only addition that it causes a starvation problem because the LOOK algorithm is dependent on the track area or the location being serviced. LOOK still retains the problem of being biased to the outermost and the inner most edges just like the elevator algorithm. The LOOK algorithm also favors the recently arriving jobs.

**C-LOOK:** A variant of the CSCAN and LOOK algorithm. In this algorithm the scanning is unidirectional and unlike the CSCAN. It does go till the end of the disks, but only till the last request in that direction. Once it is serviced the disk arm is moved to the starting position. This relies on the factor that many disks can move the read/write disk arm at high speeds if it is moving across a large number of sectors.

**F-LOOK and N-LOOK:** This was designed to avoid the bias that LOOK had towards the recent jobs that are closer to the disk arms. The idea is to split the request queue into multiple queues of smaller size and process the queues in the order of the older ones first. F-LOOK contains only two queues like the double buffering technique. The advantage of this sub queuing is that it limits the maximum latency a process can expect before it is being serviced.

**S-LOOK:** S stands for the shortest here. This algorithm is also an extension of the LOOK algorithm. It is used to handle cases, wherein the disk head is positioned between two requests that are located on either end. Then the decision is made based on the shortest seek time between the two requests, before any further request arrives. This algorithm aims at reducing the seek time.

### *2.3.2 Linux I/O Schedulers*

**NOOP:** As the name suggests it is a no operational scheduler, it acts as a modified version of the FIFO scheduler [3]. The NOOP scheduler inserts the incoming requests into a queue and provides merging if they are requested consecutively. This scheduler is used when the host does not know whether the re-ordering of the request based on the sector number would result in an increase in the productivity. The following are the places wherein the scheduler can be used: The merging and the sorting are performed by some other module located in the upper or the lower layers of the stack. The request that could probably be optimized in the block device layer, by a host bus adapter like the Serial attached small computer system interface (SCSI) with a RAID controller or an external controller connected to a storage area network. Another possible reason could be the host does not have the details of the sector positions. This could be because the host does not know about the exact arrangement of how the sorted queue should look like, when being sent to the device. The other reason could be that the devices disk arm movement from the start position to the edge position or vice-verse, would not impact the performance of the disk. Hence, does not require the re-ordering of the request. This case is true for the non-rotational media like the flash devices solid state drives.

**Deadline Scheduler:** This scheduler is an extension of the C-SCAN scheduler. It contains additional deadlines that are soft to prevent the starvation that might happen to the request that are not at the innermost or outermost edges. This scheduler is the ideal schedulers for non rotational type of flash devices. The Implementation of the deadline scheduler is good to know because this forms the basic block for the other advanced type of schedulers [21].

Implementation of the deadline scheduler: The scheduler is based on the concept of the elevator algorithm. There are two types of queues: the elevator queue wherein the requests are arranged based on their sectors using a red black tree, such that the next immediate left leaf of the tree contains the next request to be serviced. The red black tree is a self-balancing search tree [10]. The other queue is the FIFO queue called the deadline list. The deadline list is a standard queue containing block requests of same deadline period inserted to the tail of the list. Whenever a request is made, they are placed in both the deadline list and the elevator queue. Further, both the data structure have two instances, one to store the read and the other to store the write requests. The reason for two separate queues is that it gives the ability for the scheduler to prioritize the read and write requests differently. Further, batching is allowed in this scheduler, that is, up to 16 requests in a row can be serviced from the elevator queue, once the value is reached or when there are no more requests in the queue, the direction(read to write or write to read) is reversed. Adding to this, the reads are serviced twice when compared to the writes. The reason for giving high priority to the read overwrite is because the processes, usually gets blocked on the read operations. Once the data direction is selected the head of the deadline list is seen to check whether there are any requests whose deadline time has expired. If any such request is found, it is chosen as the next request to be serviced. The other main functionality handled by this deadline scheduler is merging. The request needs to be ordered to increase the throughput. The existing Linux file system splits the read operations that are large, to multiple smaller requests. Before sending them to the schedulers and the merging. It is also possible to merge request of another processes if they are the consecutive requests. The elevator queue is generally used to take decisions about the merging of requests. One of the drawbacks of the deadline scheduler is that it does not have any quality of service support. All the deadlines maintained are global and is same to all process. Therefore they have equal priority to all the tasks. Though it has an option to change the priority of the read request overwrite but it is not very useful. All the deadlines are soft, they are made only to prevent starvation of the requests which may happen in these types of elevator algorithms.

**Anticipatory Scheduler:** It is no longer maintained in the Linux kernel and replaced with the CFQ scheduler discussed in the next section [15]. The Anticipatory scheduler, came in limelight to eliminate the drawbacks of the deadline scheduler. Namely, the reduced throughput in a highly spatial locality process, the cause of thrashing due to the work conserving nature of the scheduler. Deceptive idleness: It is a situation wherein the process whose data was currently processed gives the perception that it has done requesting meanwhile preparing for the next request data. The I/O scheduler owing to its nature switch to another process thereby creating possible thrashing and reduced merging. The reason for the deceptive idleness is that the deadline scheduler is making decisions, due to early or lack of future prediction. This problem is eliminated in the anticipatory scheduler by waiting for few milliseconds before dispatching the new request, once it has completed dispatching the current request. The Apache web server load seems to perform better with this type of scheduler. The delay provides the scheduler with an option to know the about the next coming request to gives it an option to better schedule the next request. The implementation of the anticipatory scheduler is similar to that of the deadline scheduler, it has two data structure, one deadline list and the other elevator queue. Once a request has been serviced, unlike the deadline scheduler before processing the next request, the anticipatory scheduler checks whether there are any requests from the same process or if there any request which is less than thousand sectors away from the present location of the disk arm. If found, it will be serviced. Otherwise the scheduler waits for a stipulated amount of time with the expectation that a new request may come from the same process or requests that are closer to the present disk arm location. This avoids the trashing affect of the deadline scheduler. In addition to this, the anticipatory scheduler also allows backward seeks if they are located very near to the present location. The main advantage of anticipatory scheduler is as follows: It is often seen that many processes make multiple blocks of read causing multiple submit request invocation. The underlying scheduler fails to see the complete scenario, there is a possibility for more to arrive soon. The other reason for the delay may be from the file system that may have to look for the indirect blocks or the physical location of the next block in large requests. This will require sometime to get the next request that should be issued. There could be a delay from the application or the process issuing the request, that is, it may request future data based on the some caclulation results obtained or may be the processor has a low memory footprint. The other condition may be wherein multiple reads are made on the files of the same directory that are located close to each other due to the file system organization.

**CFQ Scheduler:** The completely fair queuing scheduler (CFQ)[20], is the default scheduler for the Linux kernel. This scheduler tries to satisfy most of the shortcomings of the above mentioned schedulers. The detailed explanation of this scheduler are as followed

**Providing Fairness to All the Process:** This is brought about by assigning the processes of the same I/O priority class, equal time slices to access the I/O system. The interesting point here is that the fairness is obtained based on the time slice and not on the throughput obtained, that is, for a given time the random I/O operations produce less throughput than a sequential I/O operation. Prioritizing the tasks is crucial in providing some quality of service to the processes. It is obtained by dividing the processes into classes and assigning them with some priority based on the class. Similar to the anticipatory scheduler, the spatial locality of the process request are exploited by waiting for more requests from the current process with the idea to avoid long latency caused due to the seeks from far sectors. Latency is maintained equally across all the process by scheduling them periodically in the round robin manner. As explained above the main aim of the CFQ scheduler is to provide fair allocation of the disk I/O bandwidth to all the processes. Implementation wise CFQ allocates a queue for every process that requests for an I/O operation and there are separate queue maintained for the asynchronous requests. The underlying idea is that all reads are sync and all writes are async for this scheduler. The process that requests the data is given exclusive access to the block device for a fixed time. The time allocated depends on the priority and is calculated as $time\_slice = slice\_sync + (slice\_sync/(5 * (4 - prio)))$. The default value of the $slice\_sync$ is 100msec [2]. Thus providing priorities to the application tasks based on their disk priorities brings about the QoS. CFQ has three priority classes and every process may fall into any one of those priority classes.

**Real−Time:** Process can be set into this class only by the user with root privileges. The tasks, which fall into this category, are given the first access time among the other classes. The real-time (RT) classes have priority levels ranging from 0 to 7 classes. 7 is the lowest, with 0 being the highest.

**Best Effort:** All the process by default is assigned to this class. Like the RT class, this class too has seven priority levels with 4 being the default value. The I/O priority class is modified to that of the CPU's if its priority is set.

**Idle:** This the third class of scheduling priority. The root users usually place tasks in this process. These processes are serviced only when the RT and Best Effort (BE) classes are serviced. Hence in a heavily loaded system there is a possibility of starvation for these processes. The CFQ data structure consists of all the active process (its queue) arranged in the order of their classes with RT first, followed by the BE and then the IDLE class. The second level of arrangement is based on the expected service time. The time to service is just a soft deadline. A task whose service time has been expired does not preempt an active task. If the active process is not requesting any more data then, its I/O scheduling is stopped and the remaining time slice is used during the next iteration.

## 2.4 Blktrace: Linux I/O Operations Tracing Tool

The Blktrace is developed and maintained by Jens Axboe. It is used to know the path and the operations performed on the I/O requests submitted by the application layers, in the block I/O layer[5]. The Blktrace is the tracing mechanism that is used to measure the performance and the behavior of the I/O layers of the Linux operating system. The Blktrace works by storing all the I/O request related information in the debug file systems and thus it makes it less burdensome on the CPU working. Blktrace is not a tool used for analyzing, it is an event logging tool. The events logged are later parsed by other tools like the Blkparse.It provides detailed information of the number of requests submitted, the size of the request, the time taken for it to be dispatched and so on. The Blkrawverify could also be used to test the received log messages for errors.

Blktrace[4] as mentioned before is an event logging tool. It contains many trace points that are plugged into the file systems code, in the I/O platform stack and even in the I/O scheduler. They remain inactive unless externally triggered by the user application. But the modules must be activated during the kernel building process. The logging of the events is handled by the minimal overhead mechanism called the relay file system.

**Relay File System:** Relay file system is a feature in the Linux kernel. It could be seen as a memory buffer that could be used to log events within the kernel. The relay buffers are created per CPU and are tagged to each CPU based on its CPU id. They are debug files that could be mapped on the user space by the mmap call. The relay file system offers simple kernel logging for a large amount of data. It uses the concept of channel abstraction layer that consists of a set of buffers which are mapped per CPU, each of which buffers as mentioned before are represented as files. The kernel events are logged onto the channel using the specific write functions. Each logging is mapped directly to the corresponding CPU buffers. Once the logging is completed, the events can be read by the user space by mmap the data from those files. The relay file system does not impose any restriction on the type of data that is being logged onto these files thus making it simple. The relay file system supports system calls like open(),mmap(),close() ..

**Logged Events:** Request Queued(Q): This could be seen as the time in which the request sent from the user is being inserted into the request queue [6]. The happening of this event is mostly in the context of the requesting user i.e., the user process is the one responsible for placing this request in the queue.

Dispatch Request(D): This is the time instant at which the request queue that was inserted by the user is ready. It is being sent to the memory device in order to retrieve the data from the physical memory. This is the final action from the part of the CPU, in requesting the data.

Complete Request(C): This is the time taken for the request to be serviced. This is inclusive of the time it spends in the queue before being dispatched, the time it has taken in order to be processed by the physical device. This timestamp could be used to calculate the total time taken for the request posted in the queue to be processed. It must be noted that the users perform not all the request actions. The operations like dispatching and completion are handled by the kernel threads.

### *2.4.1 Blkparse*

As mentioned before, Blktrace is not an analyzing tool it just logs events. It is the responsibility of the Blkparse tool to parse those log messages and generate data in a human readable format. It could be used both as an offline tool on Blktrace data that is logged and stored in a file or even during the run time by piping the output from the Blktrace onto the Blkparse. Blkparse can be used to trace events mentioned above like the request, queued, dispatch and the completed request. It also provides the ability to format the output data [5].

The sample output from the I/O operation after being parsed would look like this

```
    8,0 1 15 0.081862000 4713 A R 2253851 + 8 ⟵ (8,2) 277856
8,0 1 15 0.081862000 4713 A R 2253851 + 8 ⟵ (8,2) 277856
8,0 1 15 0.081862000 4713 A R 2253851 + 8 ⟵ (8,2) 277856
8,0 1 15 0.081862000 4713 A R 2253851 + 8 ⟵ (8,2) 277856
```

```
8,2 1 16 0.081865666 4713 Q R 2253851 + 8 [.example.ioread]
8,2 1 16 0.081865666 4713 Q R 2253851 + 8 [.example.ioread]
8,2 1 16 0.081865666 4713 Q R 2253851 + 8 [.example.ioread]
8,2 1 16 0.081865666 4713 Q R 2253851 + 8 [.example.ioread]
8,2 1 17 0.081889666 4713 G R 2253851 + 8 [.example.ioread]
8,2 1 17 0.081889666 4713 G R 2253851 + 8 [.example.ioread]
8,2 1 17 0.081889666 4713 G R 2253851 + 8 [.example.ioread]
8,2 1 17 0.081889666 4713 G R 2253851 + 8 [.example.ioread]
8,2 1 18 0.081896333 4713 P N [.example.ioread]
8,2 1 18 0.081896333 4713 P N [.example.ioread]
8,2 1 18 0.081896333 4713 P N [.example.ioread]
8,2 1 18 0.081896333 4713 P N [.example.ioread]
```

**Details of the Trace:** Here the first column stands for the major and minor number of the device whose events are getting logged. The next column stands for the CPU id. The third column stands for the sequence of instructions happening. The following column is the time stamp of the event happening. Then followed by the PID, the actions that took place in our case (A: mapping of I/O requests to a different device, Q: I/O requests being queued by the I/O queues, G: getting the request, P: Plugging of the request). The next column is the read, write operation, followed by the block to be accessed and the process requesting for the block.

### *2.4.2 Blktrace and Our Research*

As a part of the research we need to the I/O request characteristics of the Applications. The I/O requests that are send by the applications, needs to analyzed for this. The analysis requires measuring the I/O requests size, the time spend on the request and dispatch queues before it being serviced. Blktrace tool provides a mechanism to measure those informations in the Linux kernel.

## 2.5 Ext4 Filesystem of Linux

### *2.5.1 Fundamentals of Ext4*

The Android internally uses the ext4 file system of the Linux kernel. Ext4 was born due to the limitations in the previous ext3 format. Ext3 format was designed to hold file systems up to a size of 16TB due to its 32bit block numbers and the present existing systems were reaching that ceiling limit set. The other reasons were the need to increase the maximum number of sub directories, which had a limit of 32,768. The time resolution from seconds to nanoseconds due to an increase in the speed of the processing. The ext4 system was developed in order to cater these problems. Android included this file system as part of its stack [7]. This chapter would try and explain the working and the organization of the ext4 file system. It is based on the understanding that the reader has a basic knowledge about file systems.

### *2.5.2 Virtual File System*

This is also called as a virtual file switch. It could be seen as an abstraction layer that provides an interface to the application layer and different file systems that are available in the kernel. The virtual file system(VFS) is present in the kernel layer. The VFS was implemented in order to make the access time of the files fast and the data consistent. The VFS buffers information about each file system after it being mounted in the memory. It must monitor the activities performed like file creation, deletion and modification and update the buffers carefully. Among the caches, the most important is the buffer cache, being responsible for the integration of the file system onto the block devices. These caches are used not only for the data but also for the interface with the block drivers asynchronously. Every file system once after initialization is registered to the VFS, during the boot. It is possible to have both dynamically linking modules and modules that are integrated into the kernel, the static type for file systems. Once the file system is mounted, the VFS file system reads the superblock of the file system. It is the responsibility of the file system to map its superblock and the main routines to that of the VFS's. The file system must also contain its topology and directory tree information, mapped onto that of the VFS of the superblock. The VFS contains a list of superblocks, each for a file system available. Each of the superblocks contains the address of the subroutine that should be invoked to perform specific operations. So when this read routine of the file system is invoked, it fills out the fields of the VFS inode. Further, the VFS superblock contains the pointer head to that of the first VFS inode. In case of the root file system that would be "/".

### *2.5.3 Ext4 File System Organization*

The disk is divided into a number of blocks of fixed sizes which is decided when the file system is created. A group of sequential blocks are considered as block groups. Each block group is represented through a block descriptor.

**Group Descriptors:** The block descriptors are stored in a group descriptor table. The descriptors contain the pointers to bitmaps that are used for inode allocation, pointers of the data, and the location of the inode table and the list of free bitmaps. Upon the file system being mounted the group descriptor table and the superblock of the file system are copied onto the RAM. The real group descriptor table

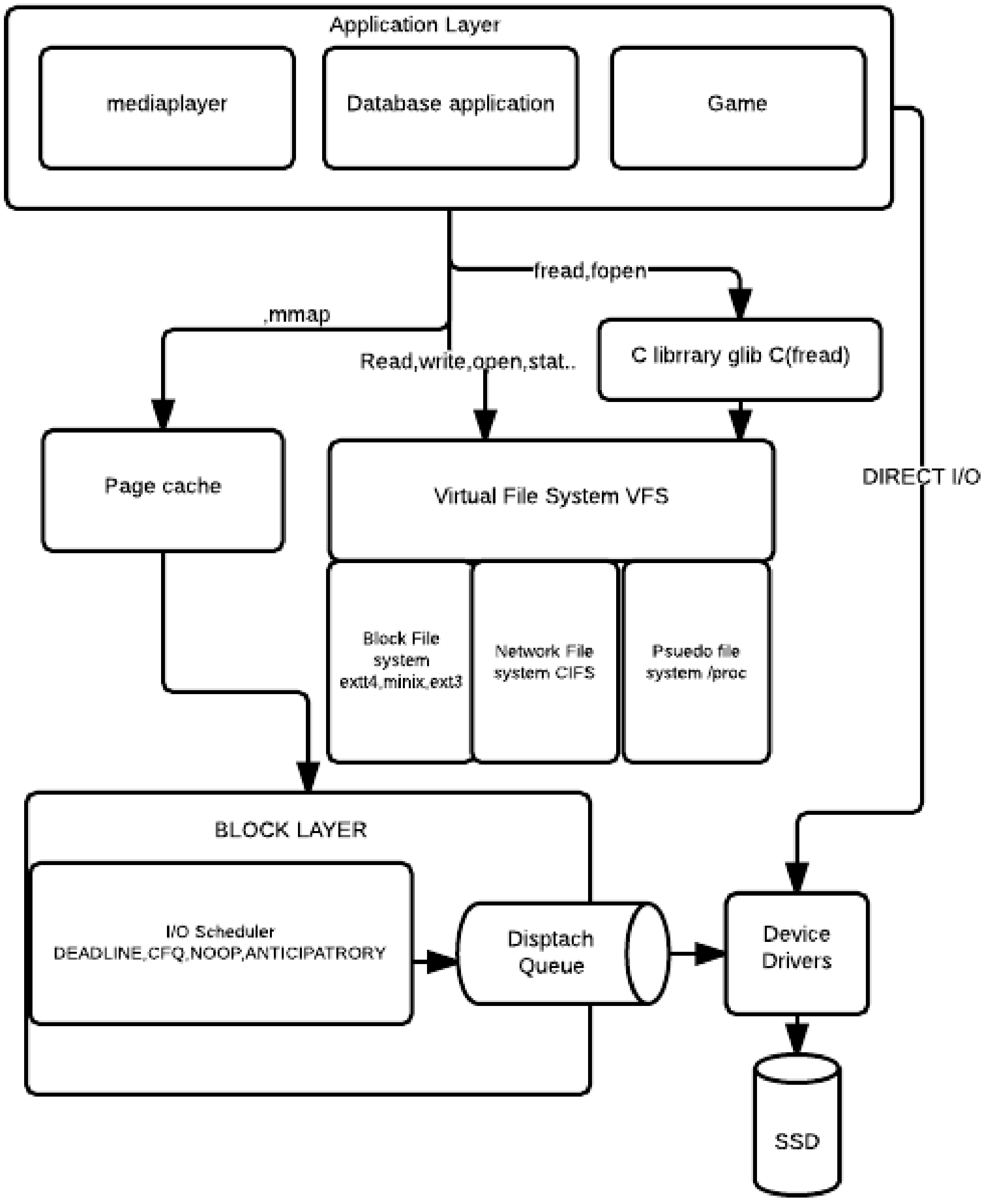

**Figure 2.4:** An Adapted Architecture Blocks of the Linux I/O System [17]

and the file systems superblock are copied in front of the block group 0. This is used for the file system recovery and its backup is stored in the group 1. The mkfs.ext4 also stores some additional space for the expansion of the file system.

**Inode and bitmaps:** The bitmaps are used to give us an indication whether a block or inode is empty or not. The bitmaps working is as follows: bit 0 is used for byte 0 if the bit is not set then it means its empty. Every block group contains two bitmaps, one for the inode and the other for the data blocks. The bitmaps are of size of one block, the default size of the block is 4KiB thus the bitmaps limits the size of the blocks in a block group to 128 Mib.

Inode: The inode is a structure that contains the information about the file or directory. It contains the information like the access permission, the last time it was changed, the address of the data blocks. The size of the inode in the ext4 data structure is 256 bytes. The data blocks that are used for storing the inode tables are pre-allocated during the initialization of the file system to fixed limit. The present ratio is one inode for every 8KiB of disk. The inode number represents the inodes. It is with this number that the inodes are located from the block array with any other additional information.

**Directory Block:** The directory block is a part of the ext4 that is used for mapping the files with the paths in which it is located. Thereby allowing the files to be accessed by their path instead of the inode index number. The directory block is a link list type of data structure. It contains the inode number, the type and the name of the file. The directories blocks are in form of the link list because they do not allow random access to the directory entries .i.e. the directories must be traversed and searched.

**Extents:** Extents could be seen as index pointer that could hold up 2Î5 continuous blocks of address information. The size of each block is around 4 KiB. Thus each extent can handle up to 128 Mib of data. The extent structure is provided below. Extents find its use in all the files and directories created using ext4. Based on the setting of the EXT4_ EXTENTS_ FL flag, the i_ node header i_ blocks are interpreted. If the flag is set then it points to the header of the extent ext4_ extent-header. The depth of the tree is held in the header structure variable eh_ depth. If the variable eh_ depth is set to zero, as in the case with the small files, then the extent points to direct disk block and the i_ node can hold 4 extents along with its header. This facilitates a direct addressing of 512MiB possible. For large files, the extents work based on the depth of the tree stored in the header struct. Thus inside a inode we may have a header followed by four ext4 index id called the ext4 _ extent_ idx, while the index node can access only 4 ext4 _ extent_ idx each ext4 _ extent_ idx points to a leaf node which is a block of 4Kib in size and this could access about 340 ext4 _ extent_ idx or ext4_ extent. Thus allowing each leaf to access around 340*128 MiB .i.e, 42.5 GiB. Thus the maximum file size for one block of size 4KiB is 16TiB.

**Journaling:** This is done in order to improve the response time and the reliability. It is a cyclic buffer that holds the important data that needs to be preserved in case of a crash. It works during the recovery by providing consistent data. The present ext4 has 128 MiB allocated to its file system as a journal. The working of the journal is as follows: the important data that is to be written onto the disk is first stored in the journal and once the data from the disk cache is cleared and the committed flag is set. The journal thread based on its scheduling will try and copy the contents onto the exact location in the disk. During this slow process if there is a system crash then the journal would be used to get all the data till the latest committed record. This guarantees that during the midway of the update even if there is a crash the data could be obtained correctly [22].

It is possible in the ext4 to specify the extent upto which the journal could be used, whereas in default case only the metadata of the files are stored and could be retried. The data of the files are not maintained in the journal so after a crash there is a possibility to lose some file data. But it is possible to allow the journal to back up all the file data by setting the flag data=journal.

## 2.6 FileBench I/O Benchmark

A benchmark that is used for storage, the filebench could be used to generate different types of workload. Using the workload modeling language it is possible to generate a wide variety of the workload that could be used to analyze the systems [23]. The main advantage of the filebench is the flexibility that it provides in creating different kinds of workloads and the ease with which they could be developed. The filebench by default provides some basic workloads like the mail server, web server and database server. The added advantage of the filebench is that it could be used for the micro benchmarking.

### *2.6.1 Salient Features*

- As said before the filebench supports testing of multiple workloads.
- It comes with some predefined workloads like the web, data base and mail servers.
- The flexibility to add personalities files or custom workloads.
- It has the ability to support multi-threaded and multi-process environment.
- Filebench comes with the ability to support configurable directories depths,

widths and even for files to provide a statistical distribution.

- Filebench provides statistical information like the throughput, latency and also the CPU cycle count per system call.
- Filebench works on all POSIX complainants operating system.

### *2.6.2 Workload Modeling Language*

Filebench works by creating synthetic modeling of the workloads. This can be used to create the behavior of the applications to a good extent thereby allowing us to analyze and predict real-time behavior with reduced time. The idea behind the filebench is to use the models described through its language, to create synthetic benchmarks that could be used to test the system in a manner much similar to the real applications [32]. It could be used to measure the performance of the system using these workloads. In order to simulate a real workload the benchmark works by creating the exact numbers of processes, thread and memory usage along with the inter process synchronization as available in the real systems. A description of the workload is obtained by defining it as a bunch of processes and thread with a defined work flow. Here every process could be seen as an address space containing many threads, with each thread specifying a certain sequence of operation. Each operation is of pre-defined behaviour as read, write etc.

### *2.6.3 Commands f Language*

The commands can be broadly classified into four main categories: commands, entities, attributes and flowops [23]. The entities constitute the specific resources like the file and threads. The executing actions are referred as flowops. The commands control the running of the benchmark. The attributes could be used to pass the commands to the entities and flowops as parameters.

**General Commands:**

**debug:** This command is used to decide the level of debugging required. If debug flag is not zero the output would be provided in a defined format starting with the process id followed by seconds from the start of the script to the debug message. By making the flag debug to 0, all the debugging messages could be suppressed. The default value of debug flag is set to 2, by making the debug flag 3 messages appear during the flowops with the statistics of the flowops.

Syntax

debug ”level of debug”

**echo:** As with every other language echo is used to send output to the standard output. Syntax must be such that message or text within the quotes will be echoed.

Syntax

echo ”information to be printed”

**exit:** This is the termination command use to stop the filebench

Syntax:

exit

**foreach:** This command is used to provide variables with values successively from a string list or a comma separated integer list. This works as simple for loop by assigning values to a variable and then executing the instructions that are written within the enclosed brackets.

Syntax:

foreach $testvar in $1,$2

//instructions

**quit:** Command is used to stop the filebench

Syntax

quit

**log:** This commands prints the value of the variables onto a log file and also to the terminal. In order to print the variables, they must be placed in the command line comma separated while the entire list of variables is placed in quotes.

Syntax:

log ”$testvar1,$testvar2”

**run:** This command is used to execute the filebench it could be supplied with the time in seconds for which the execution must be run. It resets the static variables and also initiates all the processes and creates the file set and files. Once it is done it goes to sleep and gets activated only at the time provided in the first parameter and stops the process while collecting all the necessary statistics.

run $testvar

**create:** This command is used to create file sets and processes.

**define:** used for defining the processes,random variables and the file sets

**set variable:** This command is used to search for the variable supplied as the first argument and if found assigns an integer, variable or string value in the variable. If the variable is not found it creates a variable and then assigns the second parameter.

Syntax:

set $testvar = \$testvar2| < integer > | < string >$

**shutdown:** This command is used to stop the process, list of processes provided. If there exist no such argument then its an error. Syntax:

shutdown Process | processes

**sleep:** This command is used to sleep the master process for the specified time . Syntax:

sleep < integer variable >

**system:** This command is used to execute the Linux command that is provided in the quotes.

Syntax:

system ”Linux command”

**Entities:**

**Var:** The workload can use variables, where the representation of the variables is a string of character followed by a dollar sign. The variables could be assigned values such as an integer, string or boolean by using the set variable. There are two types of commands regular variable and random variable. Regular variables are user variables that can be initialized using the set variable. The random variables are used wherein there is a need for a random value each time it is being used. These variables could be used as a regular user variables but each time it is used, it returns a random value.

**file:** This entity is used to provide information about a single file. They are defined using the syntax define file followed by the name of the file, the path and the size of the file. The f language also provides the option of reusing, filling of the file with null data and to allocate the file in parallel to other files.

**fileset:** This entity is defined using the define fileset command. The define fileset command provides the functionality to create a group of related files. The fileset command must provide a name for the fileset command, with its path to the directory where it must be created. Along with the number of files to be created, their average size, the depth of the directories and the number of the sub directories that contains the files of the fileset. The f language also provides the option of reusing, filling of the file with null data.

**Process and thread entities:** This entity is used to create a process, like in the operating system and thread that are contained in the process. The define keyword followed by process could be used to create processes. It provides the flexibility to create multiple instances and also providing the nice value to it could change the priority. The threads of the process are used to hold the attributes and other information about the operating system process, it is created within a process, it could be duplicated to multiple threads by specifying it. Threads could also be assigned regions of memory by specifying it or there is also a possibility for it to use the IPC shared memory.

**eventgen:** This is used to specify the rate in which the events must be triggered. There is only one defined value for the event generator.

**Flow operation:** The nature of the workload written is controlled by the flow operations called the flowops. The flowops are written after the defining the threads. Creating, reading, writing, opening files or filesets are the general flow operations available. The files are accessed in round robin manner.There is also a possibility in filesets to specify a specific file index number in the flowops and that file could be accessed directly. Flowops provides a wide variety of I/O operation. Flowops contains operations like read, write, read whole file, write whole file, append file, append, file rand, statfile, fsync, fsyncset, deletefile. There are directory flowops like the MakeDir, ListDir, RemoveDir. Flowops also provides the option to specify asynchronous I/O operations like the aiowrite.

# Chapter 3

# LITERATURE REVIEW AND RELATED WORKS

There has been a lot of work done to find an I/O scheduler that would be better suited for the solid state drives. These approaches varies from modifying the existing hard disk drive inclined schedulers to work better for solid state drives, defining a new scheduling algorithm. This section is written in order to list few of those approaches and the algorithms that have influenced the Scheduler that is implemented through this thesis.

## 3.1 Solid state drive Based I/O Schedulers:

FIOS: It works in order to provide fair scheduling to all the tasks, it works towards avoiding the read/write interference problem[25]. To ensure fairness it has it own fair time slice management, which fragments the quanta of time given to every request thereby ensuring fairness. It also have I/O requests anticipation to provide fairness. The other SSD specific scheduler is FlashFQ, this scheduler also works on providing fairness to all the tasks by using the start fair queuing time. This time is a virtual time based on which the requests are dispatched [28]. The internal parallelism is also exploited by sending multiple requests to the same region. The performance of the FIOS and the Flash FQ on fairness is the same. The Argon scheduler [30] is similar to the working of the FIOS. It also maintains a fair queuing technique by assigning a time quanta to each of the workload request. The incoming requests are queued and when its time to dispatch all the requests from the workload they are sent to the disk. It is continued till the quantum assigned to it expires. PASS: This scheduler is a simple scheduler that is designed to exploit the internal parallelism that exists the solid state drive [31]. This is achieved by queuing the request based on their requested address. The idea is to have multiple queues to dispatch requests. The Disk I/O Scheduling based on Energy is also discussed in the paper [33]. The sole idea behind the paper is to estimate the burst size of the request and then use that measure to schedule the task and remain idle in between the consecutive requests.

## 3.2 Application Aware Scheduling:

Rialto: This is an interesting attempt to combine both the real-time and non real-time system programs to co-exist in an architecture[16]. This approach design involves including both the strict time constrained real-time applications and the non real-time applications to work together. The scheduling of these tasks takes into account a factor called the "runbytime". This runbytime is the time by which the task must be scheduled in order to meet the deadline. SMART as the name suggests, this scheduler dynamically notifies the application the current load status, thereby making them aware and adapt to the present situation[24]. It also provides a mechanism to the user to decide the allocation of resources in case of runnings tasks having same priority. It dynamically stalls or evicts the real-time tasks in case of an overload. ACIOM:This is an application based scheduler, that prioritizes the application requesting access to the disk or the networks based on its request pattern[18]. The scheduler is for the Android platform and is implemented by modifying the Linux default scheduler the CFQ.

## 3.3 Scheduler for Android Working with a Solid State Drive:

The Scheduler that is to be designed must work for Android device that is equipped with solid state drives. The focus of our implementation is on the I/O Scheduling. There are various types of the solid state drive scheduling algorithms that are discussed above. There are quite a few algorithms that are mentioned above that could be used to schedule resources taking into account the behavior of the requester. In general, the solid state drives perform better with the deadline scheduler of the Linux kernel. The design decision was to reuse the existing scheduler and combine the two required features in it. The PASS scheduler was chosen, owing to solution it provides to the exploit the internal parallelism. Also the compatibility of it with the other application aware scheduler. The other scheduler that was chosen to be integrated onto the system is the ACIOM disk scheduler. The reason for this is, the I/O scheduling algorithm that is implemented in this is more focused on Android and meets our requirements. Thus, the thesis could be seen as the attempt to integrate the features of PASS and the ACIOM onto an Android platform and test to see whether the performance meets the expectations required.

# Chapter 4

# PROBLEM ANALYSIS AND REQUIREMENTS

The Problem that is being attempted to be resolved through this effort is of two folds. The first problem is focused on improving the bandwidth by increasing the parallelism in the Android devices that use solid state drive. The other focuses on the need to make the I/O requests rendering of the Android based on the application which requests the I/O operations. Rather, than just trying to improve the throughput by distributing the I/O system resource uniformly.

## 4.1 Exploiting Parallelism in Solid State Drives

The Android tablets are being attached with solid state drives for a lot of reasons like the speed, durability and lack of noise. The Android stack has Linux as its working kernel, and it does not have an I/O scheduler that caters the requirements of the solid state drives. The I/O Schedulers that are available in the Linux are elevator based schedulers are more focused towards the memory devices like the hard disk drives. The elevator approach to the hard disk drive is done so that it can service the requests better with minimum disk arm movement to avoid latency and wear and tear of the mechanical parts. The solid state drives as explained in the background chapter does not have any mechanical or rotating parts and their random seek time is as good as a sequential seek time. Thus the I/O Schedulers that are available in the Linux kernel although doesn't affect the working of the solid state drives in the system they do not do any good either.

### *4.1.1 Schedulers for Solid State Drives*

From the reasons mentioned above, it is clear that there is a need for a scheduler that could cater the solid state drives and exploit the parallelism that is offered by it. The following could be seen as the requirements that would be ideal for a new scheduler.

- Should not contain any elevator type approach
- Should be able to cater all the memory devices
- Need to exploit the parallelism available in the solid state drives.
- The solid state drives need sorting to increase the throughput as random reads are slower in solid state drives.
- Allow the writing and reading but avoid the read write interference problem.
- No changes must be done in the Android framework.

The reason for choosing these requirements is as follows

**Solid State Drive and Elevator Approach** The elevator algorithms are developed in order to cater the hard disk drives, the elevator algorithm makes sure that the request are serviced in sequential order. Elevator algorithm also supports a backward movement but only upto a certain limit. The reason for this being very advantageous to the hard disk drive is because the mechanical movement of the arm that read the disk plates takes time and it is always faster to service request with minimum arm movement. The other reason is if there is a request that is on the other side of the disk then the mechanical arm will have to travel more distance. More distance causes further delay. Thus taking advantage of the principal locality the hard disk drive performs better with elevator algorithms. This would not be the case with the solid state drives as it does not have any delay to get the requests from any block. Hence, the algorithm to be used on the solid state drives need not worry a lot about the principle of locality.

**Exploiting Parallelism in Solid State Drive** The background chapter clearly explains the inherent features of the solid state drives and their addressing modes. The solid state drives exhibit package level, die level and even plane level parallelism. So it is possible to make multiple planes work in parallel to gain more throughput. But the existing schedulers are unaware of it. The solid state drives also have a flash translation layer along with a buffer manager that is used for caching request and buffering the data based on the request. It tries to find the pattern of the request and prefetching the data. Hence sorting the request would yield a better result with the solid state drives.

**Read Write Interference** The read write interference problem in the solid state drives happens when the read operation and the write happen in the parallel. Due to background operations like cleaning or writing from the cache buffer, due to buffer constraints there could be conflicts between read and write. Though there are few existing I/O scheduling algorithms that avoid this by dispatching the request in separate queues, not all schedulers have something like this implemented.

## 4.2 Application Aware I/O Scheduler

Android provides an open source software for smart devices, which includes the Linux kernel, Java application and the middle layer. The middle layer is responsible for providing the services that are requested by the application and it is also responsible in providing the system libraries that are needed by the Application. The Android along with the middle layer maintains a good amount of information about the application. These includes its request to the resources like camera, media service, other information like applications is running in foreground or background. These information are not available to the Linux kernel. This results in the bursty application running in the background to receive the major share of the bandwidth and thereby starve the application running in the foreground. This will result in poor Quality Of Service to the end user. Adding to this there could be a unresponsiveness in the device caused due to the non-serviced requests from the foreground application or could cause a lag. Thus, there must be some mechanism to monitor the application bandwidth usage.

### *4.2.1 Features Required in an Application Scheduler*

The paragraph above focused on the need for the I/O Scheduler that enhances the QoS of time sensitive application.

The requirement for such a scheduler should be:

- Able to characterize the applications and assign requests based on their type.
- Should make sure that the time sensitive application are serviced within time
- Fairness among the time sensitive application must be maintained.
- The other application running in the background should not starve
- If the request is only from the background process they should be serviced without any delay.
- No changes must be done in the Android framework.

Though the reason for choosing the above requirements are self-explanatory here some more justification for these requirements

**Characterize Application** The scheduler must be able to do this because the priority assignments should be done only, based on the type of the application that need the requests.

**Service in Time** The scheduler should be responsive to the task that needs its request to be serviced within a fixed interval. For e.g. the media player, its request must be serviced with a specified time to avoid lag or distortion.

**Fairness** The scheduler must be fair to the application of the same type, this is also crucial because we cannot allow two applications running with the same specifications being serviced differently.

**Starvation** The scheduler must not cause the starvation of the background applications, they must run with lower priority but still should receive their request within due time. It is also the responsibility of the scheduler to allow the tasks to access the maximum bandwidth, if it is the only application running.

This algorithm makes the effort to bring those application details on to the Linux kernel and thereby make the I/O system's resource management of the kernel, take into the account the nature of the application. The algorithm suggested does not involve any changes in the application or the service layers. It characterizes the applications based on the services that it requests as time sensitive and bursts or plain. The idea of this scheduler is to provide a maximum bandwidth reservation for the time sensitive applications and put a limit on the bandwidth allocated to the bursty applications. The effectiveness of the algorithm lies in the management of the I/O system resource when the time sensitive and the bursty applications are running.

Chapter 5

# THE DESIGN OF CONTEXT-AWARE SSD I/O SCHEDULER

The Design of the our I/O Scheduler combines the effect of the parallelism in the solid state drives and includes application awareness during I/O scheduling. The scheduler must be integrated onto the Linux kernel that would work on the Android systems. The idea is to modify any one of the existing schedulers in order to integrate these above features on to the Linux kernel. The best candidate for this is chosen to be the deadline scheduler, the reason for choosing this Deadline scheduler is as below:

- It is simple and easy to adapt.
- It has separate data structures for handling both read and write.
- It has both time-based as well as address based sorting structures.
- It has a deadline associated with each request that could be used to guarantee fixed response times.

## 5.1 Concept of the Solid State Drive I/O Scheduler

The suggested scheduler is focused towards the solid state drives and the parallelism in it. The algorithm suggested here partitions the disk into subregions speculatively and then attaches a queue that could be used for dispatching the request to those regions. The requests are placed in the corresponding queues based on the addresses of the blocks being requested. The sub queues are serviced in a round robin manner one after the other [31]. The relationship between the addresses requested and the parallelism that exists within the solid state drives are utilized effectively through this algorithm. The results convey the same. In addition to this, the algorithm also enhances few other features of the solid state drives like the lifetime and reduction of the read-write interference [25]. The life-time of the solid state drives are improved by sorting the requests and thereby reducing the number of random writes which enhances the life of the solid state drive. The read-write interference problem is resolved by having a specific queue for each of the operations.

### *5.1.1 Design Details*

The idea as mentioned above is to split the entire disk region into a number of subregions of each fixed size, such that each subregion exploits the parallelism intrinsically present in the solid state drives. The requested address is used as an indicator to place them in the expected queues. The internal implementation of the scheduler includes two FIFO lists that are used to track the request based on the arriving times and two red black trees to track the same request, but sorted by the requested address. The data structure maintained here contains two of each type in order to handle read and write separately. The incoming request is placed on both the data structures. The servicing of the requests is done in the round robin manner. The assumption here is that the request would fall into different queues and therefore would be serviced in parallel by the underlying solid state drive. The idea of dividing the entire disk into number of subregions comes from the fact that the inherent parallelism in the solid state drive is optimal only for a fixed defined number of requests. Anything more, would degrade the performance. In other words, the subregion or the flash package can best serve only a fixed requests simultaneously. If the number of the requests exceeds the optimal point then, the performance would degrade due to resource contention [14]. The creation of these subregions would provide us with the ability to successfully control the number of requests that could be send to each subregion. By switching to another queue, we would be able to avoid overcrowding. The switching to another queue allows requests to be served in another queue within the same time thereby increasing performance, which otherwise must be spent on waiting for the over requested queue to be completed. To explain this lets assume that the maximum number of requests that could be serviced within a subregion is "A". If there are two requests "A1" and "A2" such that both are greater than A and let us assume that the requests fall in two different subregions "Q1" and "Q2" respectively. Then it is possible with our algorithm to service a part of "A1" that is equal to A and then by switching to another queue "A2" to service some more request. This approach is not possible in the existing algorithms which can service only "A" requests in one region at a given time.

**Deciding the Optimal Subregion Space:** This is one of the important procedure of the entire algorithm. The ideal size is determined by running tests on the solid state drive. The test is carried out by sending multiple requests on to the same region. By varying the size of the region and the number of requests being sent to it, the ideal point is obtained. Plotting the performance curve between requests and region sizes the optimal point gives us the region size and the batch value for every queue.

**Creation of the Subregion:** The algorithm calculates the total disk size. The splitting of the request based on the queue is done as following: let us say the total size of the disk is A in sectors and the size of the subregions is L in sectors. Then the total number of the queues that would be needed is A+L-1/L. The address regions for all the queues ranging from 0 to A/L would be (L*c,(c+1)*L-1). Where "c" is an index ranging from 0 to A/L. Whenever there is, a request received its address is

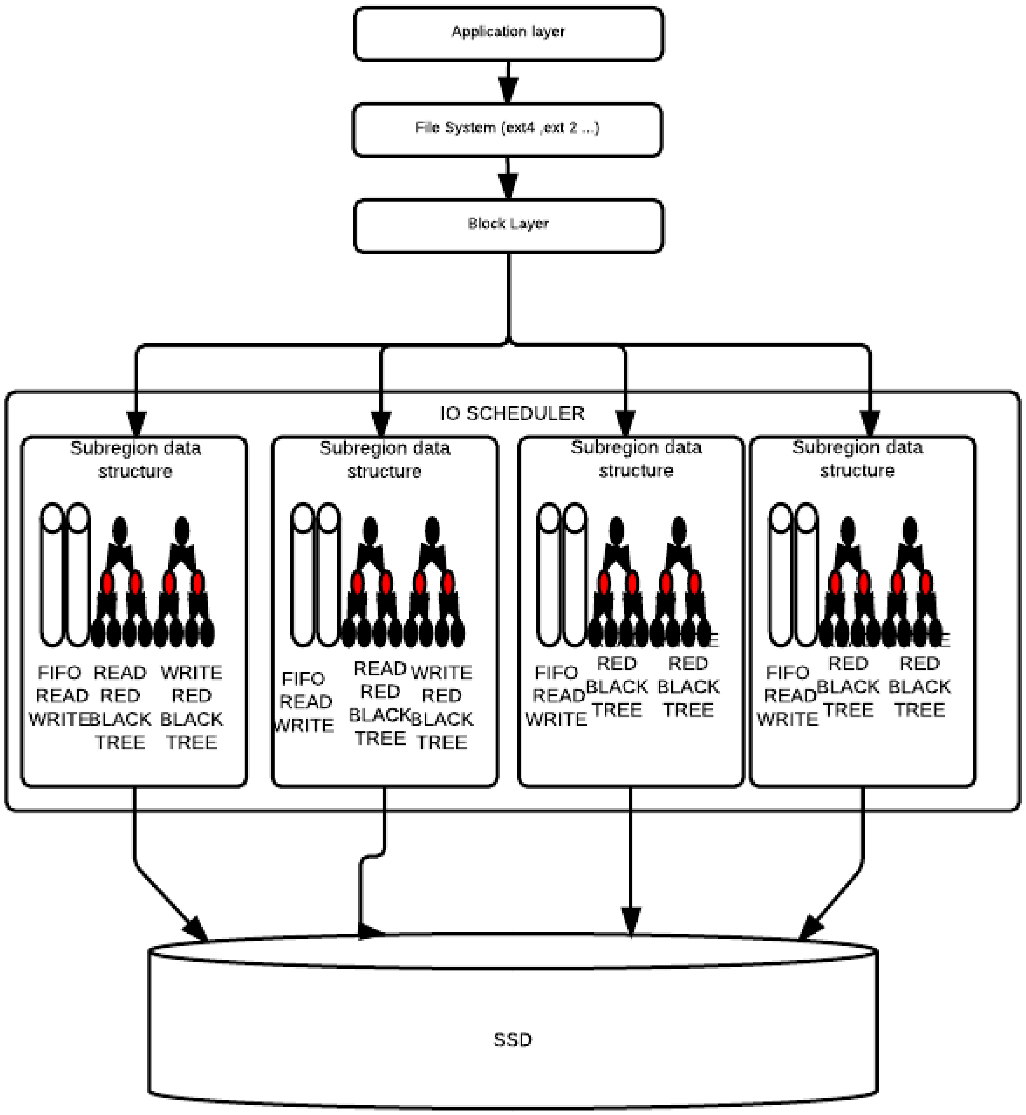

**Figure 5.1:** An Solid State Drive Scheduler for Internal Parallelism[31]

divided by the sector size of the subregion "L" and the corresponding number is the queue's index. The corresponding data is placed onto that queue's linked list and the red black tree.

**The Request Handler and Mechanism to Avoid Interference:** As mentioned before, there are several data structures that are used to track the request based on their requesting address. The two FIFO lists handle the requests based on the arrival order one for read and the other for write. The two red black trees are for ordering the requests based on the accessing order. Each incoming request is assigned with a deadline in order to serve the request, avoid starvation and guarantee responsiveness. The idea of the deadline is to try and make the request be serviced within that specified time. This servicing is achieved by monitoring the FIFO lists for their deadline expiry. The usage of the red black tree is to provide a sequential access pattern to the solid state drives which could improve the life of the solid state drives.

**Dispatching the Request:** The dispatching of the request happens by selecting the dispatching queues in a round robin manner. During the dispatch of a sub queue, based on the direction of the previous request, it continues in the same request order and dispatches the request, then changes its direction in the coming turn. There exist two conditions wherein the dispatch queue moves to another queue, when the dispatching queue is empty or the case when the number of requests it has dispatched exceeds the batch value. The idea of individual queues and batching avoids the read-write interference problem and also exploits the inherent parallelism that exists.

**Read-write Interference Avoidance:** The dispatching of the request follows the read-write request handling in such a way that the interference problem is avoided.

The dispatching queue checks if there are any pending request in its queue. If any request is found then it is serviced, provided the number of requests serviced in this queue is less than it batch value. If the request exceeds the threshold or there are no more requests in the queue, then the request pattern is changed from the previous one i.e. if the previous request was read then it is changed to write. Then the next queue is checked for the pending request in the write queue, if there are any requests they are serviced. If the requests in that direction do not exist or the number of requests exceeded the threshold, then it changes the direction of dispatching the request back to read and checks the next queue. On a closer look, we could understand that the request queues are serviced in the direction opposite to the adjacent queue. This avoids the read-write interface problem in the solid state drives.

## 5.2 Concept of the Application Aware I/O Scheduler

The I/O Scheduler suggested makes the effort to bring the application awareness on to the Linux kernel. Thereby make the I/O system's resource management of the kernel to taken into the account the nature of the application. It does not involve any changes in the application or the service layers. It characterizes the applications as time-sensitive or bursty or plain, based on the services that it requests. The idea of the algorithm is to provide a maximum bandwidth reservation for the time-sensitive applications and put a limit on the bandwidth allocated to the bursty application. The effectiveness of the algorithm lies in the management of the I/O system's resource during the situation when both time-sensitive and bursty applications are running.

### *5.2.1 Design Details*

This section describes the design to be adapted on the I/O scheduler to make it a priority based, with a bounded time guaranteed I/O operations to the time-sensitive application. This algorithm would provide a prompt and fixed delay bounded responses to the user interactive tasks and thereby improve the quality of service to the end user. The first part of this section attempts to provide an insight into how the applications are characterized as time-sensitive, bursty or plain.

**Classification of the Application Type Based on the Services:** The applications are characterized by their I/O requests access pattern as plain, bursty or time-sensitive. Applications could be said time-sensitive if they impose some restrictions on the timing requirements of the I/O requests. The video player is time-sensitive as it expects the data available to it at a particular rate of frames, any delay would cause a lag in the appearance of the video. Bursty requests are those type of applications that would make huge chunks of request in a bursty manner. A typical example of such an application would be a download manager service or a storage service. The plain type of applications would be services like power management which are not that intensive. There are few other application that are intensive when they are in the foreground while when in the background they are bursty, an example for that would be the SQL service. If this works for the foreground application then, it is time-sensitive,background then it is a bursty application.

**Information Provided by Android to Characterize the Application:** The application by itself does not provide any information to the Linux kernel about its nature. This makes it difficult for the Linux kernel to differentiate. The advantage of Android platform in this context is that it provides us with the information about the services that are being requested by the application. Based on that the characteristics of the application are derived. For example, an application that uses the download manager service routine could be seen as a bursty application. The download manager requests data in the bursty fashion both the disks and the networks are affected by these bursty requests. While, on the other hand, a foreground running app like a game could be seen as a time-sensitive app when it requests data from the disk for its processing. Thus, the classification is brought about in the application by deciding the type of services that are being requested by these applications.

### *5.2.2 Algorithm*

The algorithm should be able to reserve bandwidth for time-sensitive applications and at the same time restrict the bandwidth to the bursty applications. The amount of bandwidth that could be allocated to the time-sensitive application, in order to execute it smoothly and avoid any performance loss is not known. Through the algorithm, the bandwidth based on the previous request size and the time allocated for the requests is estimated. The algorithm takes a count of the disk requests along with its size. The algorithms first calculates the estimated time by which the present disk request should be serviced. For this it measures the arrival time of the requests, the requests size. The projected time period is represented as a weighted combination of the previously available values and the new data. Mathematically: If $R^i$ represents the ith request by an application that is time-sensitive. Let $E^i$ be the arrival time of the request and $RS^i$ be the request size. Then the projected time by which it must be serviced is given by

$$T^i = \lambda * \ T^{i-1} + (1 - \lambda) * E^i - E^{i-1}$$

Here $T^0$ represents the default time period set for read in the scheduler. The arrival time of the first instance is calculated as $E^0 = E^1 - \ T^0$. $\lambda$ is a history quotient it determines the how much previous value must be considered. It ranges from $[0, 1]$.

The request size is measured for the ith request as follows

$$RS^i = \gamma * \ RS^{i-1} + (1 - \gamma) * RS^i$$

. Here the $RS^0$ is $RS^1$, where $RS^1$ is the first size requested by the application i. The parameter $\gamma$ similar to $\lambda$. The bandwidth $BW^i$ is the measure of the quantity that the application requests on an average for its working[29].

$$BW^i = T^i * RS^i$$

Once the estimation of bandwidth for each application is available, the next task is to assign this bandwidth. In order to do it, the application's request size $RS^i$ is to be serviced within the time interval of $T^i$. This is achieved by using the deadline parameter attached to every request when inserted into the I/O schedulers queue.

Thus, the required bandwidth is attached to every request from a time-sensitive application as below

$$D^i = D^{i-1} + T^i$$

. Here $D^i$ is the request's deadline of the ith request and the initial value $D^0$ is made equal to $A^1$. Thus, by making sure that the ith request from the time-sensitive application is handled before its assigned deadline $D^i$ the bandwidth of minimum $B^i$ is provided to the application.

**Bursty Request** The bursty application, on the other hand, tends to consume all the available disk bandwidth, when they are scheduled which affects the time-sensitive application. Hence, a limit is placed on the bandwidth of the bursty applications such that a reserved bandwidth is always available for the time-sensitive application. The implementation involves maintaining a separate queue for bursty applications and all the bursty applications are placed in this queue for some default time and are then inserted into one of the I/O request queues.

The release time of these applications is decided based on the available remaining bandwidth once it has been allocated to all the time-sensitive applications. The total remainder bandwidth is first calculated and then equally distributed to all the bursty applications. This ensures that the application that are time critical receives the bandwidth required.

**Bursty Request Bandwidth Allocation:** Calculating the remaining bandwidth is done as follows

$$BWR = \frac{TBA - \sigma * B^i + SB}{TB}$$

where $BWR$ is the remaining bandwidth, TBA is the total maximum bandwidth possible in the system, T: the number of time-sensitive tasks and B; number of bursty tasks. $B^i$ stands for the aggregate of the total bandwidth consumed by the time-sensitive application and SB is the safety reserved bandwidth.

Once the remaining bandwidth is obtained, the next task is to assign the bandwidth to the bursty applications. In order to do this, the average request size is needed. This is obtained in a similar way as the time-sensitive application.

$$BS^i = \epsilon * \ BS^{i-1} + (1 - \epsilon) * BS^i$$

Where the $BS^i$ is the average requested size of the bursty application and $BS^i$ is the actual size. Once the average request is calculated, it is used to find a number of bursty request that could be serviced. This is obtained as $Q^i = BWR/\ BS^i$, this value is used to adjust the release time of the bursty applications.

$$RB^i = RB^{i-1} + \frac{1}{Q^i}$$

. The value of $RB^0 = E^1 - T^1$.

Chapter 6

# IMPLEMENTATION

The Linux kernel's block layer is modified in order to implement the algorithm. The deadline scheduler is modified in order to incorporate the change. The reason for choosing this scheduler is, it is better suited to include both the Android specific and the solid state drive specific algorithms. The Linux kernel version 3.0.05 was chosen for the modification. The Android version used for testing the algorithm is Jelly Bean 4.3. The test setup was build on SABRE SD from Freescale. The board has the following configuration: Freescale i.MX 6Quad, 1 GHz processor based on the ARM Cortex A9 core, 1 GB DDR3 SDRAM, 7-pin SATA data connector and 8GB flash memory storage. A 8GB solid state drive was connected via the SATA connector to the setup.

## 6.1 Solid State Drive I/O Scheduler

**Experimental Setup:** The solid state drives used is a Seagate 8GB in size, the sub region size for the solid state drive algorithm is decided by using the filebench benchmark. The exact sub region size is decided by modifying the workload fivestream-read. This workload contains five threads reading onto five different files. The file is changed to make the five threads read the same file. Slowly the size of the file and the number of threads is increased. A graph is plotted between the file size and the number of threads the point, which is optimum, is chosen. The optimal point gives us the size of the sub region and the batch value of that region. After our analysis, for the solid state drive we found that the optimal point is at 2GB and the batching size is 16. So the entire sector size of the solid state drive is created into the sub regions of size 2GB each, and the inherent parallelism is exploited using it. The filebench provides us with workloads of various types like the varmail, webserver, fileserver and the database, these workloads, as explained before could be used to emulate the exact conditions of the real workloads that exist in the systems.

## 6.2 Application Aware I/O Scheduler

**Experimental Setup:** The application aware scheduler needs to be configured with the below parameters. The TBA the total bandwidth available with the solid state drive this is taken from the specification of the disk and cross checked by running a micro bench of sequential read. The SB is the safety bandwidth and it was decided to be 0.1 times the value of the TBA. The values of $\lambda$ and $\gamma$ are chosen to be 0.99 to account for the previous value as much as possible. The exact value of $\lambda$ is chosen by repeated tests with various values ranging from 0.5 to 0.99. The default request deadline is based on the Linux kernel provided along with the board. The default request deadline was set to 50ms. The modification in the Deadline Scheduler required for implementing the application aware I/O Scheduler are the following: Two hashmap data structures one for the time sensitive application and the other for the bursty application was created. The hashmap data structure was implemented using the support functions provided in the Linux kernel. These structures hold the deadline information that are added to the request blocks when it is inserted into the queue. These data structures along with the deadline also maintains the bandwidth and the request sizes, these are used for the calculation. The insertion of the modified deadline is done at the "deadline_ add_ request" interface exposed by the elevator. The decision of selecting the dispatch queue is taken by the "deadline_ dispatch_-requests", which is modified to allow insertion of the bursty requests on to the request queue along with updating the dispatch request.

# Chapter 7

# EVALUATION

The Two algorithms the solid state drive parallelism and the Application awareness are combined in the deadline scheduler. The resultant scheduler has the features to exploit the parallelism in the solid state drives and schedule the I/O requests based on Application characteristics. The implementation mentioned in the previous chapter was tested.

## 7.1 Test Results from Solid State Drive I/O Scheduler

In order to test the solid state drives parallelism exploits, the filebench test works loads were used. The Comparison is made between the present default Linux scheduler the CFQ against our scheduler. The workloads used and their characteristics are defined:

**fileserver:** This workload could be closely related to the general workload of all mobile application. This workload does the following operations it creates files, deletes files, append files, read and write and has some attribute operations performed on the directory tree provided. It works by using up to 50 threads, It works similar to the specsfs benchmark.

**webserver:** This workload as the name explains emulates the web-server. It has around 100 threads that open read and close multiple files in the directory mentioned. It is a basic functionality repeatedly performed.

**mailserver:** This off the shelf workload is called the varmail. The mail server store files(e -mails) in a particular directory referred as the server. The operations that are performed consists of reading, deleting, create-append-sync, read-append-sync. There were 16 threads used and it resembles the postmark bench with multi-threading enabled.

**proxywebserver:** This workload is similar to the webserver with the additional operation of appending the files to simulate the logs generated in the proxy servers. Like the webserver, it has 100 threads created by default.

**Table 7.1:** Workload Analysis for CFQ and Our Scheduler

| S.No | Workload | CFQ(in OPS) | OurScheduler(in OPS) |
|---|---|---|---|
| 1 | FileServer | 3795.235 | 4039.433 |
| 2 | webserver | 4697.863 | 4720.331 |
| 3 | Mailserver | 4640.463 | 4722.6 |
| 4 | ProxyWebserver | 723.7 | 907.712 |

These workloads are considered because the are the closest the mobile application workloads. The workloads are ran in both the CFQ scheduler and our scheduler the table below gives the measure of operations per second performed. The table below provides the information about the behavior of the schedulers to the workloads

From the table and the figure it is clearly seen that our I/O Scheduler outperforms the CFQ in all the workloads thereby making it better suited for the devices fixed with the solid state drives. The main reason for our scheduler to perform better is because of selecting the exact sub region size and thereby allowing the requests to be serviced in parallel.

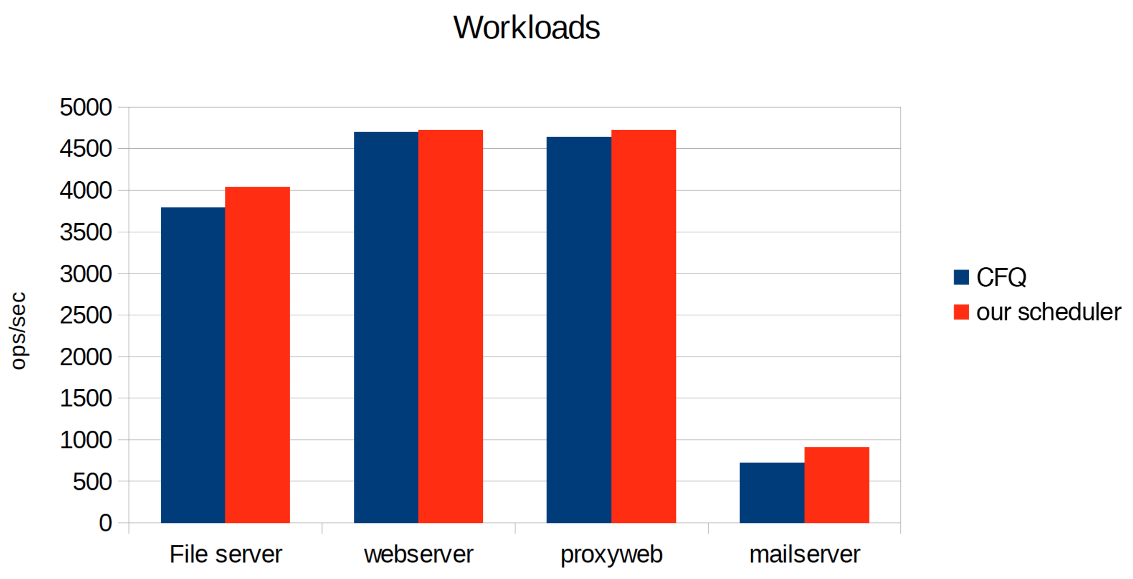

**Figure 7.1:** Workload Analysis of the CFQ(Default Linux Scheduler) and Our Scheduler

## 7.2 Test Results Application Aware I/O Scheduler

The Second part of the algorithm is focused on improving the responsiveness of the time sensitive application. The experimental setup consists of the two applications, the first application is a media player the Mx player: A free mobile app chosen for the testing purpose, the other application is a custom app called the IOREAD: It spawns a thread and reads a file sequentially in the background. The IOREAD app is written in JNI and C combined with the Java interface to provide the app like behavior. The test conditions is as follows, first the IOREAD app is ran and it makes sequential reads, after some time the media player is made to play a video. The Blktrace is used to measure the request traces posted by both the applications. The Blktrace traces are later post-processed using Blkparse and a parsing script, to get the exact request posted by each of the application. Two different quality of videos are used for the test purpose one video is 320x240 px and the other has a resolution of 1080px.

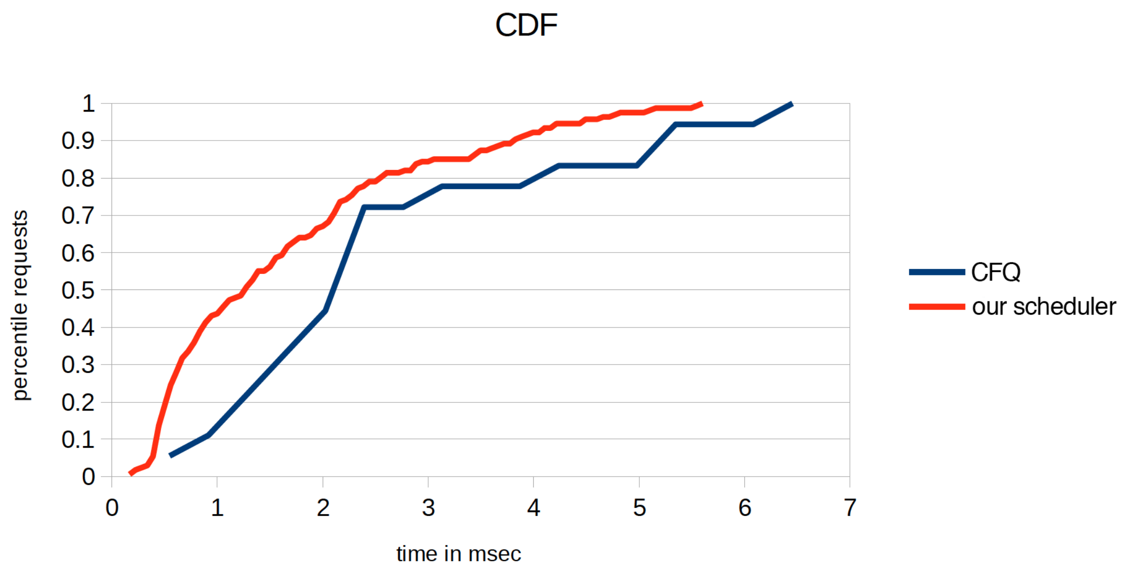

**Figure 7.2:** Cdf of the CFQ(Default Linux Scheduler) and Our Scheduler for the Media Player Playing Video of 320 Px Resolution

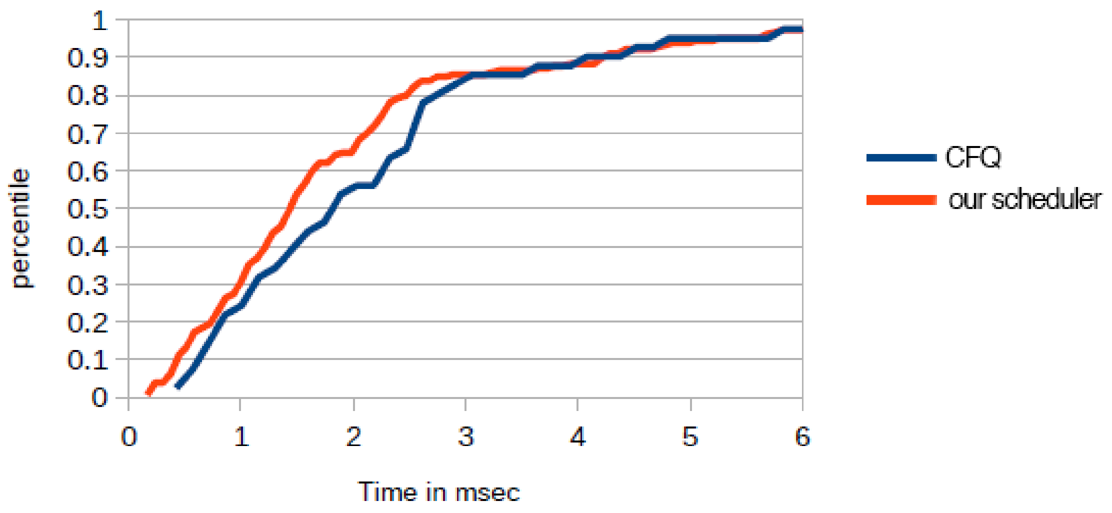

**Figure 7.3:** Cdf of the CFQ(Default Linux Scheduler) and Our Scheduler for the Media Player Playing Video of 1080px Resolution

The results are shown using the cumulative distribution function. The reason for choosing this distribution is that the request sequence and batching varies based on the type of the schedulers used and a direct comparison between these two schedulers could not be obtained using latency measurements.

The above two graphs shows the performance improvement with our scheduler in comparison to that of the CFQ. The latency measurement clearly shows that for any given video quality the performance of our scheduler stands out. The background task chosen is sequential continuous read. The reason for the faster response with our scheduler is because the video application is considered as a time sensitive application and allocated the resource first and then the background tasks receive the remaining.

## 7.3 Conclusion

Through the thesis an I/O Scheduler is developed, it provides us with a better performance and exploit the inherent parallelism in the solid state drives. It also improves the responsiveness of the applications based on their services they request. The inherent parallelism of the solid state drives are exploited using the exact sub-region size and dispatching requests to those regions based on the request address. Thereby, making them work in parallel. Since, the requests are being sent unidirectional, after they are sorted it provides a solution to the read-write interference problem. The results on the workloads have shown that out scheduler outperforms the CFQ scheduler. The second part of the scheduler is focused on improving the responsiveness of the mobile system for time sensitive applications. The results show that we are able to provide better responsiveness to the time sensitive application even during the existence of the background bursty application. Our scheduler combines the effects of parallelism in the solid state drives and better scheduling of the time sensitive applications. This is better suited for the Android system attached with solid state drive.

# Chapter 8

# FUTURE WORK

The I/O scheduler has been designed, implemented and tested successfully the future works that should be handled are as explained below:

The area that must be next focused is the networks. The application awareness that has been brought about in the I/O scheduler could be extended to the networks. The present network scheduler used in the Linux kernel is not application aware and works by serving the requests in the FIFO order. This would result in degraded performance on a n online game when in the background some downloading happens. By making the network scheduler aware of the application, we can avoid this problem.

The other area that could be focused is pre-launching of the application in the Android. Based on the context like the location, activity decision could be made to pre-launch the apps. For e.g. based on the location say at home few apps like the Games, Facebook and youtube could be pre-launched thereby avoiding the time taken to load the application. Similarly, when a person is running the exercise apps or the music apps could be pre-launched. These exact apps to be launched can be configured based on the user's request.

# REFERENCES

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.80 edition, May 2014.

[2] Jens Axboe. Trees ii: red-black trees, January 2002. URL `https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt`.

[3] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. 3.0 edition, 2005.

[4] A. D. Brunelle. Block I/O layer tracing: blktrace. In *Gelato-Itanium Conference and Expo*, April 2006.

[5] Alan D Brunelle. blktrace user guide, February 2007. URL `http://www.cse.unsw.edu.au/~aaronc/iosched/doc/blktrace.html`.

[6] Alan D Brunelle. btt user guide, April 2007. URL `http://stderr.org/doc/blktrace/btt.pdf`.

[7] Mingming Cao, Suparna Bhattacharya, and Ted Tso. Ext4: The next generation of Ext2/3 filesystem. In *Linux Storage & Filesystem Workshop,February*. USENIX Association, 2007.

[8] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09.

[9] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *17th International Conference on High-Performance Computer Architecture*, 2011.

[10] Jonathan Corbet. Trees ii: red-black trees, June 2006. URL `http://lwn.net/Articles/184495/`.

[11] Laura M Grupp, Adrian M Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H Siegel, and Jack K Wolf. Characterizing flash memory: anomalies, observations, and applications. In *Microarchitecture, 2009. 42nd Annual IEEE International Symposium on.*

[12] Chang-Hung Hsieh, Yu-Yu Chen, Chih-Chieh Yang, Shih-Lung Chao, and Hung-Yu Wei. POSTER: A smart scheduling mechanism for energy saving in Android system. In *Proc of the 10th International Conference MobiSys '12*. ACM.

[13] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shu Ping Zhang. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In *Proc of the 25th International Conference on Supercomputing*, 2011.

[14] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance. *Computers, IEEE Transactions on*, June 2013.

[15] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proc of the 18th ACM SOSP '01*. ACM, 2001.

[16] Michael B. Jones, Paul J. Leach, Richard Draves, and Joseph S. Barrera III. Modular real-time resource management in the rialto operating system. In *5th Workshop on HotOS-V*, 1995.

[17] M.Tim Jones. Anatomy of the linux file system, October 2007. URL `http://www.ibm.com/developerworks/library/l-linux-filesystem/`.

[18] Hyosu Kim, Minsub Lee, Wookhyun Han, Kilho Lee, and Insik Shin. Aciom: application characteristics-aware disk and network I/O management on Android platform. In *Proc of the 11th International Conference on EMSOFT*, 2011.

[19] Jaeho Kim, Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H Noh. Disk schedulers for solid state drivers. In *Proc of the 7th ACM international conference on Embedded software*. ACM, 2009.

[20] Robert Love. Kernel korner - I/O schedulers, February 2004. URL `http://www.linuxjournal.com/article/6931`.

[21] Robert Love. Linux kernel development, August 2010.

[22] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier, and Bull S. A. S. The new ext4 filesystem: current status and future plans. In *Ottawa Linux Symposium*, 2007.

[23] Richard McDougall and J Mauro. Filebench tutorial. *Sun Microsystems*, 2004.

[24] Jason Nieh and Monica S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. *SIGOPS Oper. Syst. Rev.*, 31, October 1997.

[25] Stan Park and Kai Shen. FIOS: a fair, efficient flash I/O scheduler. In *Proc of the 10th USENIX conference on FAST*, page 13, 2012.

[26] Abhishek Rajimwale, Vijayan Prabhakaran, and John D. Davis. Block management in solid-state devices. In *Proc of the 2009 Conference on*, USENIX'09.

[27] Rick Rogers, John Lombardo, Zigurd Mednieks, and Blake Meike. *Android Application Development: Programming with the Google SDK*. O'Reilly Media, Inc., 1st edition, 2009.

[28] Kai Shen and Stan Park. FlashFQ: A fair queueing I/O scheduler for flash-based ssds. In *USENIX Annual Technical Conference*, 2013.

[29] Cheng-Han Tsai, Edward T.-H. Chu, and Tai-Yi Huang. Wrr-scan: A rate-based real-time disk-scheduling algorithm. In *Proc of the 4th ACM International Conference on EMSOFT '04*.

[30] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance insulation for shared storage servers. In *5th USENIX Conference on FAST 2007*.

[31] Hua Wang, Ping Huang, Shuang He, Ke Zhou, Chun-hua Li, and Xubin He. A novel I/O scheduler for SSD with improved performance and lifetime. In *IEEE 29th Symposium on MSST*, pages 1–5, 2013.

[32] Andrew Wilson. The new and improved filebench. In *Proc of 6th USENIX Conference on File and Storage Technologies*, 2008.

[33] Youjip Won, Jongmin Kim, and Wonmin Jung. Energy-aware disk scheduling for soft real-time I/O requests. *Multimedia Syst.*, 13, 2008.