1. INTRODUCTION

1.1 PROJECT IDEA

The software will be an I/O scheduler for Flash Based Devices in order to optimize the throughput of block device for read write operations. The scheduler will reside in the kernel I/O subsystem. It will accept requests from the user space and reorder the requests accordingly. The existing I/O Schedulers are designed for Hard Disk Drives with moving mechanical components. It will leverage the inherent parallelism, lower access time and less latency of SSDs.

1.2 MOTIVATION

Flash based SSDs have the potential to alleviate the ever-existing I/O bottleneck problem in data-intensive computing environments, due to their advantages over conventional HDDs in aspects of performance, energy, reliability, etc. SSDs differ from traditional mechanical HDDs in various respects. The most distinguishing feature is that they are built upon semiconductors exclusively, completely being free from the rotational latency which dominates the disk access time of HDDs, which results in SSDs' operational speed being one or two orders of magnitude faster than HDDs. However, on the other hand, due to the long existence of HDDs as persistent storage devices, the existing I/O scheduling algorithms have been specifically designed or optimized based on the assumption of HDDs' characteristics. As a consequence, replacing conventional HDDs with SSDs in the storage systems without taking optimizing other relating components into account, we may not be able to make the best use of SSDs, squandering the promising performance they can provide. We propose to improve the performance and lifetime of SSDs by leveraging the rich inherent parallelism within SSDs, which has been well observed.

2. PROBLEM DEFINITION AND SCOPE

2.1 PROBLEM STATEMENT

To build an I/O scheduler for Flash Based Devices in order to optimize the throughput of block device for read write operations.

2.1.1 Goals and objectives

- To provide a scheduler for Flash Based Devices instead of existing schedulers which are designed for Hard Disk Drives with moving mechanical components.
- To make efficient use of available hardware and software resources.
- To leverage the inherent parallelism, lower access time and less latency and of SSDs.

2.1.2 Statement of scope

A description of the software with Size of input, bounds on input, input validation, input dependency, I/O state diagram, Major inputs, and outputs are described without regard to implementation detail.

Major Input:

I/O requests from user space comprising of Read or Write requests

Major Output:

Modified queue of I/O requests according to Veloces Algorithm

2.1.3 Software context

For scheduling of I/O to optimize the performance of SSD.

2.1.4 Major constraints

Any constraints that will impact the manner in which the software is to be specified, designed, implemented or tested are noted here.

- Data dependency – Maintaining the data dependency when changing the request order
- Write amplification problem of SSDs
- Lifetime of SSD
- Possible interference with operation of existing intelligent SSD Controllers

2.1.5 Hardware Resources Required

Hardware	Minimum	Suggested
CPU	1GHz	2 x 2GHz
Memory	2GB	4GB
Disk	SATA SSD	SATA SSD
Disk Space	5GB	20GB

2.1.6 Software Resources Required

- Ubuntu 12.04 Ext-4 formatted.
- Emacs IDE 24.3
- gcc Compiler-4.4

2.1.7 Area of Project

D.4 Operating Systems

D.4.2 Storage Management

3. PROJECT PLAN

3.1 INTRODUCTION

3.1.1 Problem Definition

To build an I/O scheduler for Flash Based Devices in order to optimize the throughput of block device for read write operations.

3.1.2 Management and technical constraints –

- Time Limitations.
- Memory Limitations.
- Design for Unix based operating system.

3.1.3 Purpose of the Document

It is to give an overview of the plan, project schedule, risk estimates, team structure and roles.

3.2 PROJECT ESTIMATES

3.2.1 Reconciled Estimate

- **Estimated Cost:** As were working using open source software, and partly hardware independent software there is hardly any finance needed.
- **Estimated Time:** The project took around 4 to 5 months split as 2 months on design and algorithm studies and remaining on actual implementation.

3.2.2 Project Resources

3.2.2.1 Hardware Resources Required -

- **Hardware Required for Host -**

The host machine with following requirements

Hardware	Minimum	Suggested
CPU	1GHz	2 x 2GHz
Memory	2GB	4GB
Disk	SATA SSD	SATA SSD
Disk Space	5GB	20GB

Table 3.1 Hardware requirement

3.2.2.2 Software Resources Required -

- Ubuntu 12.04 Ext-4 formatted.
- Emacs IDE 24.3
- gcc Compiler-4.4

3.3 RISK MANAGEMENT

3.3.1 Project Risks -

- **Technical Risks –**

  I. Solid State Device might wear out due to excessive workloads.

- **Management Risks -**

  I. Lack of planning.
  II. Lack of management experience and training.
  III. Communication problems.

IV. Lack of authority.
V. Control problems.

3.3.2 Risk Table

No.	Risk	Category	Probability
1	Changes in Design regarding choice of underlying algorithm,	Technical – Design	70%
2	Lack of communication among team members due to clashes in schedule.	Project	5%
3	Ambiguous Requirements that may change, i.e study of the complex SSD Architecture and Controller.	Project	20%
4	Performance.(Something taking too long or too heavy on resources.)	Technical	30%
5	Release of a similar product by another team.	Business	30%
6	Experience with development language/ platform/ tools.	Project	10%

Fig. 3.2 Risk Table

3.3.3 Overview of Risk Mitigation, Monitoring, Management

- **Risk Mitigation:** Schedule the project in such a fashion that measure modules are completed first so that they get enough time for testing and debugging.

- **Risk Monitoring & Management:**
  1. Keep a track of the project by lines of code and complexity.
  2. Keep track of the number of man hours spend.
  3. Modify the algorithm for efficiency as the project develops and it becomes clearer as the development phase progress.

3.4 PROJECT SCHEDULE

Sr. No.	Deliverables	Submission Date	Review Date
1	Group Formation	2nd Week of July	--
2	Synopsis	2nd Week of August	3rd Week of August
3	Guide Allocation		
4	Detail problem Definition, Literature survey , Platforms, SRS(Software Requirement Specifications)-Analysis of Existing Schedulers, Study of properties of SSDs	3rd week of Sept.	3rd week of Sept.
5	Project Plan-Design of Basic Algorithm	4th week of Sept	4th week of Sept
6	High Level Design Document-Tracing Flow of Request	1st week of Oct.	1st week of Oct.
7	Submission for I term Report consisting of(2,4,5,6)	2nd week of Oct.	2nd week of Oct.
8	Detail Design Document-Adding Tunables,	3rd week of Jan.	4th week of Jan.

9	Test Plan-Comparison of Veloces with existing Schedulers, Benchmarking for various workloads	3rd week of Feb.	3rd week of Feb.
10	Review of Project with Demo	Last Week of March	Last Week of March
10	Report (Soft Copy)	1st week of April	1st week of April
11	Report (Hard Copy)	2nd week of April	2nd week of April

Table 3.3 Project Schedule

3.4.1 Project task set

The different task sets that are critical in the project are as follows: –

Sr. no	Task ID	Name of Task
1.	T-1	Study of SSD Architecture and SSD Controller
2.	T-2	Looking for existing solutions.
3.	T-3	Study of NOOP. Deadline, CFQ (Existing Algorithm)
4.	T-4	Preparation of first prototype.
5.	T-5	Revision of the prototype.
6.	T-6	Final deliverable
7.	T-7	Testing and documentation.

3.4.2 Task network

- Timeline chart

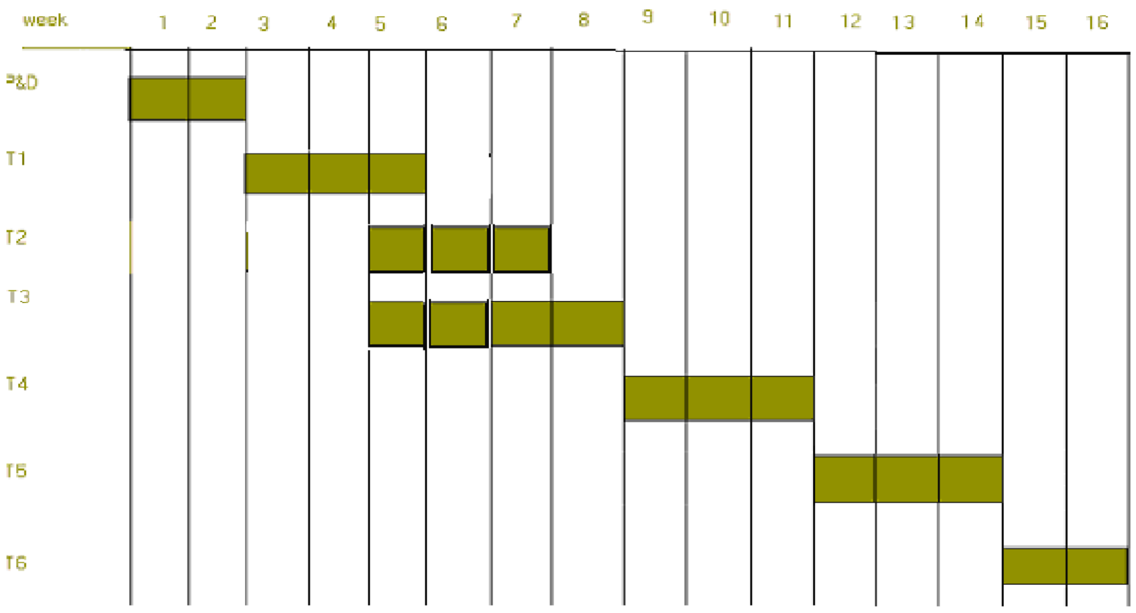


Fig.3.1 Timing Diagram

3.5 STAFF ORGANIZATION

3.5.1 Team structure:

Role	Responsibilities	Participant(s)
Project Sponsor	1.Ultimate decision-maker and tie-breaker 2.Provide project oversight and guidance 3.Review/approve some project elements	L3CUBE -Swapnil Pimpale -Nafisa Mandliwala
Project Guide	4.Manages project in accordance to the project plan 5.Provide overall project direction 6.Direct/lead team members toward project objectives	Prof. Girish Potdar
Project Participants	7.Communicate project goals, status and progress throughout the project to personnel in their area 8.Coordinates participation of work groups, individuals and stakeholders 9.Assure quality of products that will meet the project goals and objectives 10.Identify risks and issues and help in resolutions	-Vishakha Damle -Amogh Palnitkar - Om Pawar -Sarvesh Rangnekar

Table 3.4 Team Structure

3.5.2 Management Reporting and Communication

The communication began in July with initial assignments being given, which were in generalized domain. Each assignment was checked and accordingly reviewed and corrected by the external guide. In August, the focus is on finding problem definition and going through the detailed study as how to go about the project. On an average, we have couple of meetings per week with external guide as well as internal guide.

3.6 TRACKING AND CONTROL MECHANISMS

Techniques to be used for project tracking and control are identified.

3.6.1 Quality assurance and control

Software quality is defined as conformance to explicitly state functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed seawares. Software quality assurance (SQA) is an activity that is applied throughout the software process. It involves various steps like:

- Create a set of activities that will help ensure that every software engineering work product exhibits high quality.
- Perform quality assurance activities on every software products.
- Use matrix to develop strategies for improving your software process and as consequence the quality of the product.
- Everyone involved in the software engineering process is responsible for the quality of product.

4. SOFTWARE REQUIREMENTS SPECIFICATION

4.1 INTRODUCTION

4.1.1 Purpose and Scope of Document

- To present a detail of the software requirements and analysis of what is intended to design. A more detailed view of the project will follow which will include the development phases and the deliverables, as well as the project plan.

- To present an analysis of the perceived problem from which a list of specific unambiguous requirements will be formulated from the user's point of view. The list will include functional, external interface, performance and other requirements.

- Design constraints and attributes leading to the finished product will be specified.

- To discuss the approach adopted to carry the project through the development process to completion. This will include the analysis and design technique used.

- To provide an overview of the interaction between the user and the system, design of the visual appearance of the interface. A description of the interface between the user interface and the software will be included in the subsequent documents.

4.1.2 Overview of Responsibilities of Developer

The responsibilities of the Developer can be broadly overviewed as follows:

- Analyzing the requirements of the project.

- Design for recommendations for the workload specified.

- Provide an easy and effective user interface.

- Coding and testing the tool with predefined results and check for efficiency.

4.2 PRODUCT OVERVIEW

The scheduler will be used for scheduling of I/O requests in the block layer to optimize the throughput of SSD. The scheduler will reside in the block I/O layer. It reorders and merges the requests before sending them to the SSD controller. SSDs work best in big data centers where speed, ruggedness, form factor, noise, or fragmentation are important factors.

4.3 USAGE SCENARIO

This section provides a usage scenario for the software. It is the organized information collected during requirements elicitation into use-cases.

4.3.1 User profiles

1. Actor Name : Process

Actor Functions:

I. System Call

2. Actor Name : Linux Kernel

Actor Functions:

I. Valid Request

II. Invalid Request

III. Check Cache

3. Actor Name : Block Layer

Actor Functions:

I. Send Data

II. Send Request

4. Actor Name : Cache

Actor Functions:

I. Cache Hit

II. Cache Miss

III. Send Data

5. Actor Name : Scheduler

Actor Functions:

I. Set priority

II. Classify Requests

III. Reorder Requests

IV. Merge Requests

4. Actor Name : SSD Controller

Actor Functions:

I. Fetch Data

4.3.2 Use-cases

All use-cases for the software are presented. Description of all main Use cases using use case template is to be provided.

1. Use Case Name: System Call

   Primary Actor: Process

   Secondary Actor: Linux Kernel

   Pre Condition: Valid Process

   Main Flow:

   1. Process sends an I/O request as a System Call to the Linux Kernel.

   Alternate Flow:

   Post Condition: System Call regarding I/O is sent to the Linux Kernel.

2. Use Case Name: Invalid Request

   Primary Actor: Linux Kernel

   Secondary Actor: Process

   Pre Condition: System Call is sent to the Linux Kernel

   Main Flow:

   1. Process sends a System Call to the Linux Kernel.

2. The Linux Kernel on receiving the System Call analyses it.
3. Invalid System Calls are ignored and an Error Message is sent to the Process.

Alternate Flow:

Valid requests are serviced.

Post Condition: Error Message is sent to the Process.

3. Use Case Name: Valid Request

   Primary Actor: Linux Kernel

   Secondary Actor: Block Layer

   Pre Condition: System Call

   Main Flow:

   1. Process sends a System Call to the Linux Kernel.
   2. The Linux Kernel on receiving the System Call analyses it.
   3. Valid request is serviced.

   Alternate Flow:

   Invalid System Calls are ignored and an Error Message is sent to the Process.

   Post Condition: Cache is checked.

4. Use Case Name: Check Cache

   Primary Actor: Linux Kernel

   Secondary Actor: Cache

   Pre Condition: System Call and Valid Request.

   Main Flow:

   1. Process sends a System Call to the Linux Kernel.
   2. The Linux Kernel on receiving the System Call analyses it.
   3. If the request is valid cache is checked.

   Alternate Flow:

   Post Condition:

5. Use Case Name: Cache Hit

   Primary Actor: Cache

   Secondary Actor: Linux Kernel

   Pre Condition: Valid Request and Cache is checked.

   Main Flow:

1. The Linux Kernel checks the cache.
2. If it is Cache Hit, data is retrieved from the Cache.

Alternate Flow:

Post Condition:

6. Use Case Name: Cache Miss

   Primary Actor: Cache

   Secondary Actor: Block Layer

   Pre Condition: Valid Request and Cache is checked.

   Main Flow:

   Alternate Flow:

   Post Condition:

7. Use Case Name: Send Request

   Primary Actor: Block Layer

   Secondary Actor:

   Pre Condition: Valid Request and Cache Miss

   Main Flow:

   Alternate Flow:

   Post Condition:

8. Use Case Name: Set Priority

   Primary Actor: Scheduler

   Secondary Actor:

   Pre Condition: Request is sent from Block Layer to Scheduler.

   Main Flow:

   Alternate Flow:

   Post Condition:

9. Use Case Name: Classify Requests

   Primary Actor: Scheduler

   Secondary Actor: Request is sent from Block Layer to Scheduler.

   Pre Condition:

   Main Flow:

Alternate Flow:

Post Condition:

10. Use Case Name: Merge Requests

    Primary Actor: Scheduler

    Secondary Actor: Request is sent from Block Layer to Scheduler.

    Pre Condition:

    Main Flow:

    Alternate Flow:

    Post Condition:

11. Use Case Name: Reorder Requests

    Primary Actor: Scheduler

    Secondary Actor: Request is sent from Block Layer to Scheduler.

    Pre Condition:

    Main Flow:

    Alternate Flow:

    Post Condition:

12. Use Case Name: Dispatch Requests

    Primary Actor: Requests are classified, merged and reordered.

    Secondary Actor: SSD Controller

    Pre Condition:

    Main Flow:

    Alternate Flow:

    Post Condition:

13. Use Case Name: Send Data

    Primary Actor:

    Secondary Actor: Process

    Pre Condition: Cache hit or Data is received from Block Layer.

    Main Flow:

    1. On receiving valid request, the Linux Kernel checks the cache.
    2. On Cache Hit the data is fetched directly from the cache.

Alternate Flow:

Post Condition: Requested data is sent to the Process.

4.3.3 Use Case View

Use Case Diagram

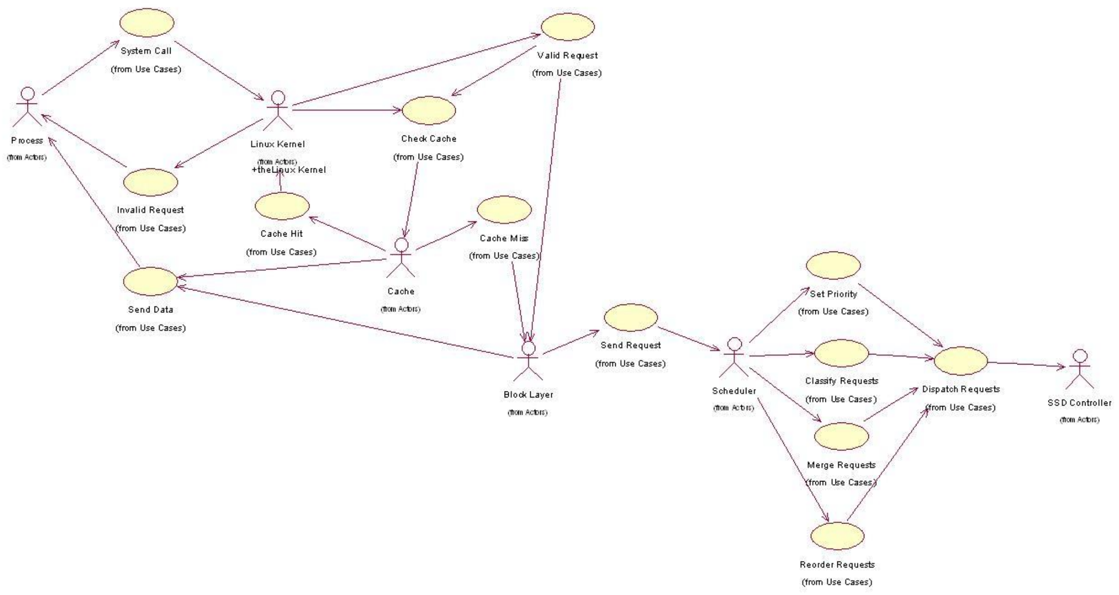


Fig. 4.1 Use Case – Business Model

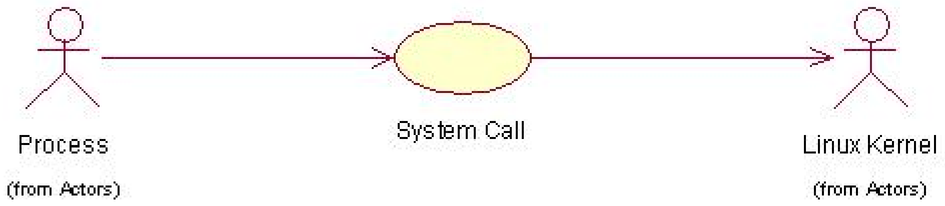


Fig. 4.2 Use Case – Issue Request

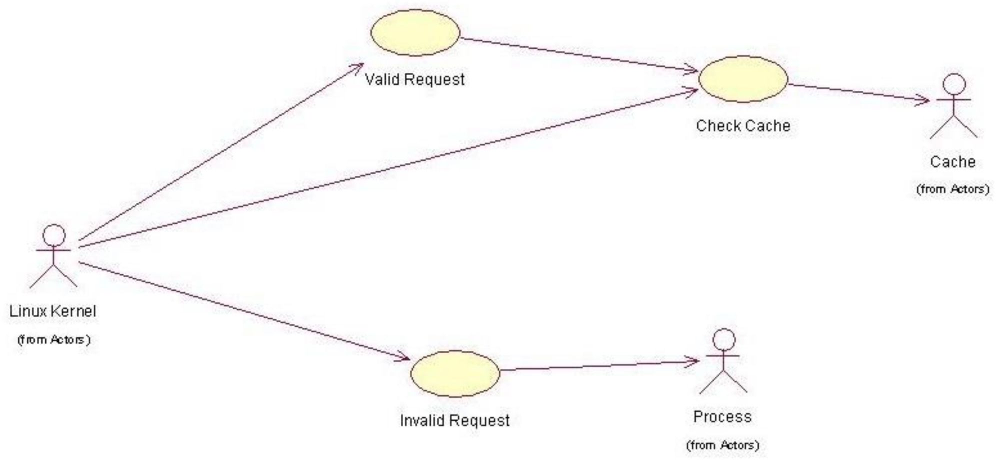


Fig. 4.3 Use Case- Validate

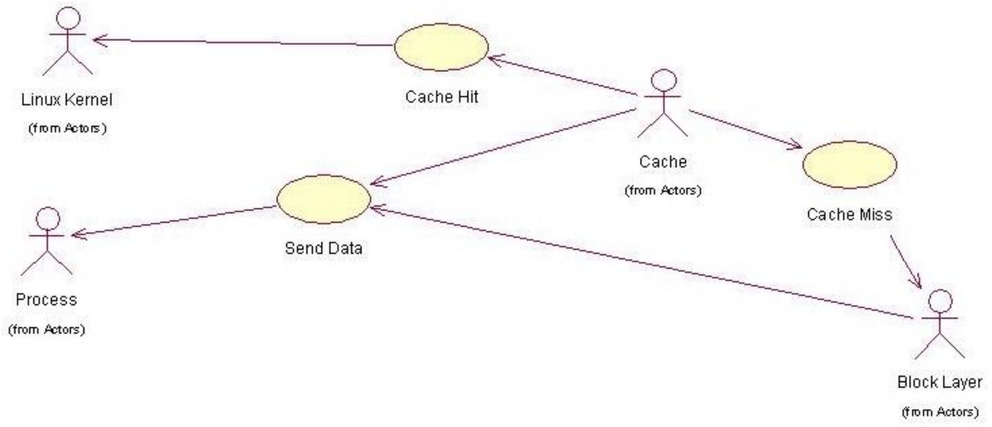


Fig. 4.4 Use Case- Return Data

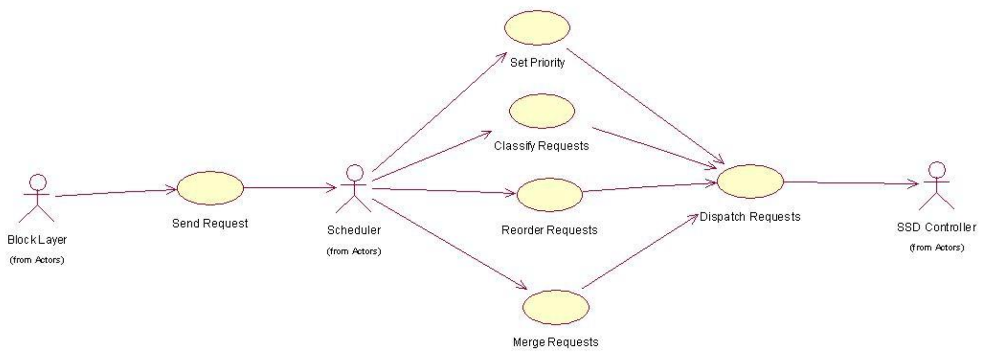


Fig. 4.5 Use Case- Scheduler

4.4 SPECIAL USAGE CONSIDERATIONS

Special requirements associated with the use of the software are presented.

I. Scheduler relevant only to a Solid State Drives.
II. Scheduler will work only for Linux Ext4 File System.

4.5 DESIGN CONSTRAINTS

- Exclusive scheduler for EXT4 file system.
- Design only for Unix based operating system.

4.6 SOFTWARE INTERFACE DESCRIPTION

4.6.1 External Machine Interfaces:

- External machine interface won’t be required.

4.6.2 External System Interfaces:

- The File system itself is the interface for the system.

4.6.3 Human Interface

- Human Interface won’t be required.

4.7 RESTRICTIONS, LIMITATIONS AND CONSTRAINTS

- Write amplification problem of SSDs
- Lifetime of SSD
- Possible interference with operation of existing intelligent SSD Controllers

5. SOFTWARE DESIGN SPECIFICATION

(HIGH LEVEL DESIGN)

5.1 INTRODUCTION

The purpose of this Detailed Design Document is to add the necessary detail to the current project description to represent a suitable model for coding. This document is also intended to help detect contradictions prior to coding, and can be used as a reference manual for how the modules interact.

The Detailed Design Document documentation presents the structure of the system, such as the architecture, application architecture (layers), application flow (Navigation), and technology architecture. The Detailed Design Document uses non-technical to mildly-technical terms which should be understandable to the administrators of the system.

5.1.1 Overview of Product

The software will be an I/O scheduler for Flash Based Devices in order to optimize the throughput of block device for read write operations. The existing I/O Schedulers are designed for Hard Disk Drives with moving mechanical components. It will leverage the inherent parallelism, lower access time and less latency of SSDs.

5.2 ARCHITECTURAL DESIGN

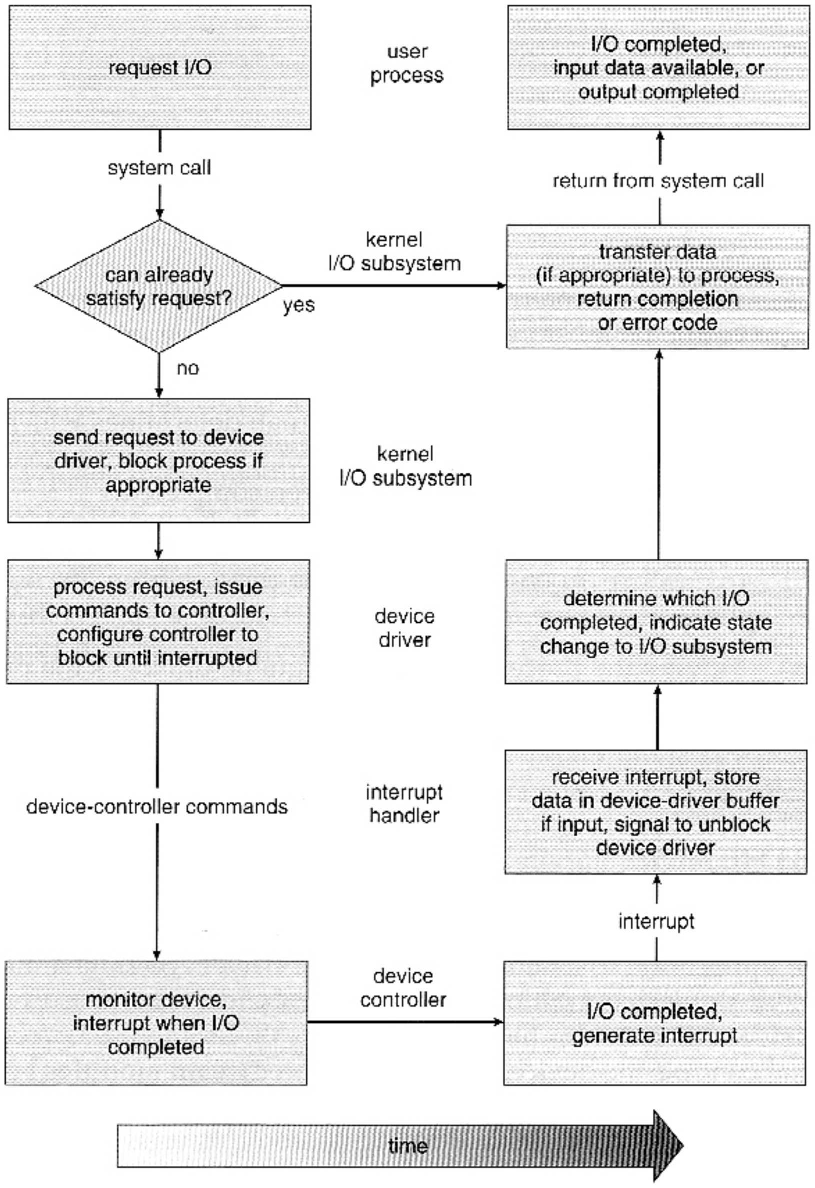
I. The process issues an I/O request system call. The kernel checks whether the parameters are valid.

II. If the data is already present in the cache, then the requested data is returned to the process.

III. Otherwise, the block device needs to be accessed. The I/O scheduler in the kernel subsystem receives the requests and schedules the request accordingly.

IV. The device driver operates the device hardware to perform the data transfer.

V. On completion, the device driver signals the kernel I/O subsystem that request has been completed. It then transfers the data to the requesting process.


request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
kernel I/O subsystem
yes
transfer data (if appropriate) to process, return completion or error code
no
send request to device driver, block process if appropriate
kernel I/O subsystem
process request, issue commands to controller, configure controller to block until interrupted
device driver
determine which I/O completed, indicate state change to I/O subsystem
device-controller commands
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt
time

`

Fig 5.1 I/O Request

5.3 DATA DESIGN

5.3.1 Internal software data structure

a. Request

Data structure to store the information contained in a request. It maintains the information about the request such as tag, length of data, retires, next request.

b. Request_queue

It is the data structure to store the requests in a queue. It consists of the list head and list of the requests.

c. Elevator_type

It is the data structure to store the specific details of the scheduler and the functions used by the scheduler. Identifies the elevator type. It contains the elevator options, elevator name.

5.3.2 Global data structure

a. BIO

It is the main unit of I/O for the block layer and lower layers (ie drivers and stacking drivers). It stores information about block device such as max segments size, flags and device address.

5.4 COMPONENT DESIGN

5.4.1 Program Structure

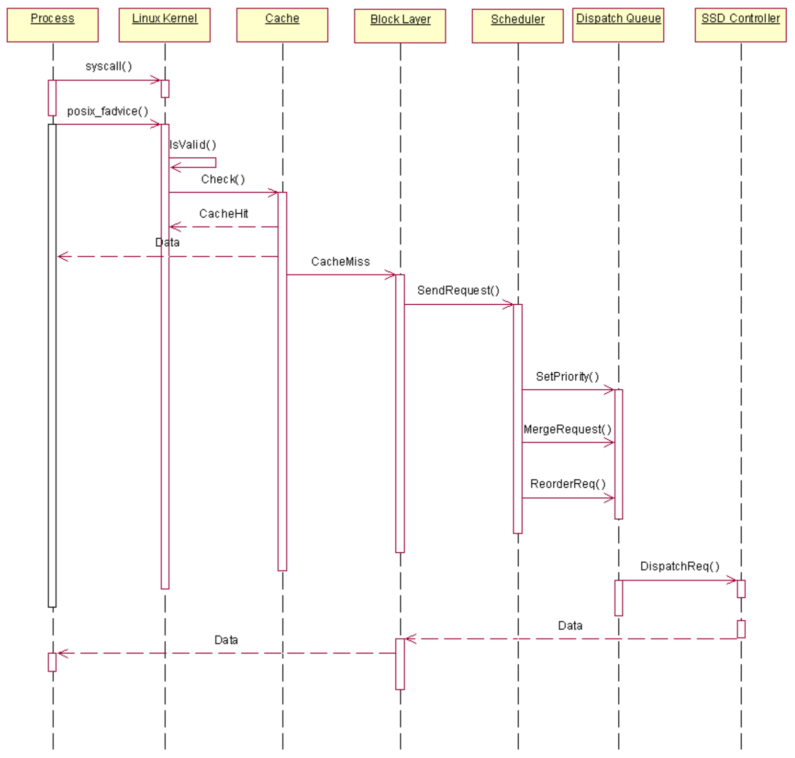


Fig. 5.2 Interaction Diagram

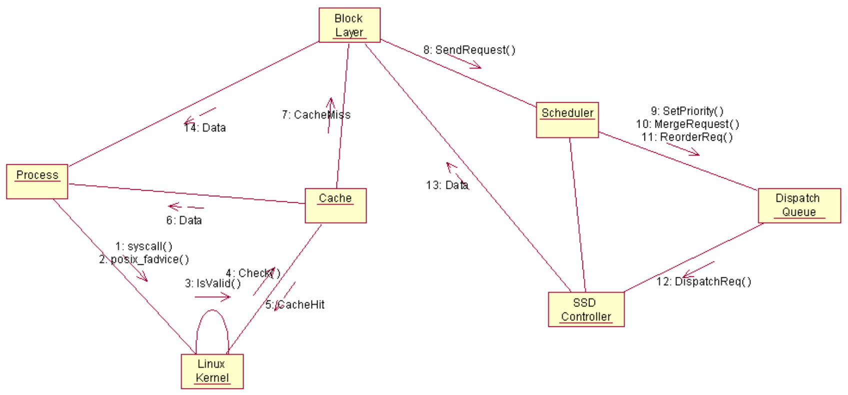


Fig. 5.3 Collaboration Diagram

5.3 SOFTWARE INTERFACE DESCRIPTION

5.3.1 External Machine Interfaces:

- External machine interface won't be required.

5.3.2 External System Interfaces:

- The File system itself is the interface for the system.

5.3.3 Human Interface

- Human Interface won't be required.

5.4 RESTRICTIONS, LIMITATIONS AND CONSTRAINTS

- Write amplification problem of SSDs
- Lifetime of SSD
- Possible interference with operation of existing intelligent SSD Controllers

5.5 REFERENCES

[1] UML basics: The sequence diagram
http://www.ibm.com/developerworks/rational/library/3101.html

[2] Dan Pilone, Neil Pitman. UML 2.0 in a Nutshell published by O'Reilly

6. TEST SPECIFICATION

6.1 INTRODUCTION

The Document seeks to provide the Test plan, test strategy and the test procedure for the project, "Veloces : An Efficient I/O Scheduler for Solid State Devices " , developed as a part of Final year Engineering Project at Pune Institute of Computer technology with purpose of obtaining a degree in Computer Science at the Pune University. This document will be used by faculty of computer Engineering department and the external guide. This document provides detailed description of test specifications with regard to unit testing, integration testing, system performance testing etc.

6.1.1 Goals and objectives

Following are the major goals and objectives that are fulfilled with this document:

i. Each unit of code is thoroughly tested so that common bugs are eliminated.
ii. The functioning of every tested unit is tested when they are integrated together.
iii. The performance increase/decrease is calculated against common benchmarks.

6.1.2 Statement of scope

The given feature of scheduler is tested against its implementation limited to the Solid State Devices. The tests are performed to check whether all the different aspects of the scheduler are working for the Solid State Device.

6.2 TEST PLAN

This section describes the overall testing strategy and the project management issues that are required to properly execute effective tests.

6.2.1 Software (SCIís) to be tested

We are going to test the project "Veloces : An Efficient I/O Scheduler for Solid State Devices" which aims at implementing the implementing the scheduler for solid state devices. An I/O Scheduler is the one which decides the order of block I/O operation and resides in the block layer of the Linux kernel. The I/O Scheduler ensures that the I/O requests are issued within a particular deadline and that requests are completed within minimum time.

6.2.2 Testing strategy

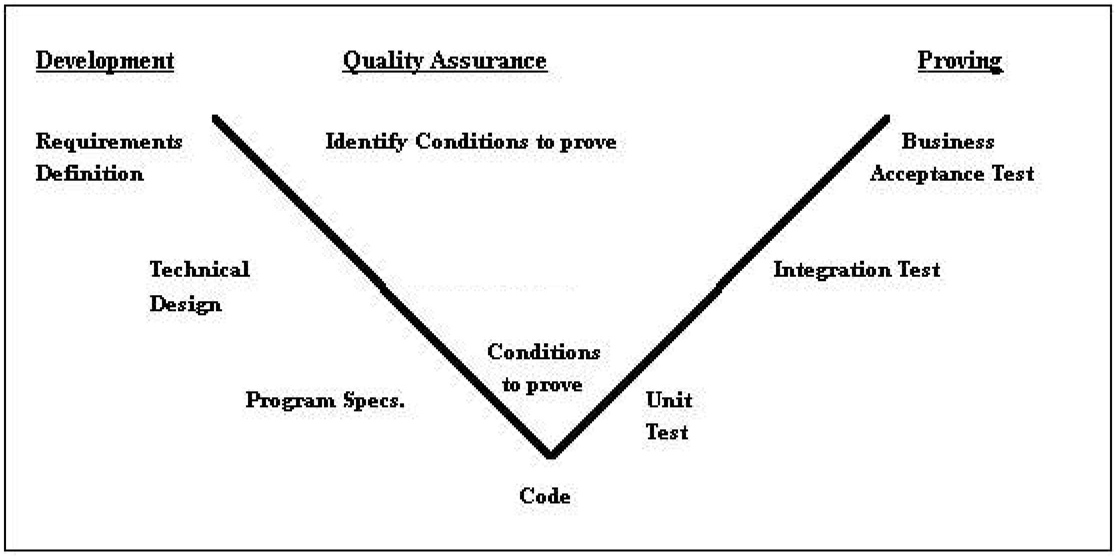


Fig 6.1 Testing Strategy

The above V Model shows the optimum testing process, where test preparation commences as soon as the Requirements Catalogue is produced. System Test planning commenced at an early stage, and for this reason, the System test will benefit from Quality initiatives throughout the project lifecycle.

Designing a system test involves identifying test cycles, test cases, expected results. In general, test conditions / expected results will be identified by the Test Team in conjunction with the Project Business Analyst or Business Expert. The Test Team will then identify Test Cases and the Data required. The Test conditions are derived from the Business Design and the Transaction Requirements Documents.

6.2.2.1 Unit testing

Each component is tested separately. A bottom up approach is used for testing. Components for unit testing:

i. Get the input request
ii. Enqueue the request to read or write queue
iii. Schedule the request
iv. Dispatch the request

Criteria for selection

Components for unit testing are selected so that this individual unit of code can be tested separately for their working without depending on other components of the system.

6.2.2.2 Integration testing

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.

Integration testing is performed after unit testing. It focuses on the problems related to integration of units that function correctly when used individually.

An integration testing may follow either a top-down approach or a bottom-up one. Here we have used bottom-up testing i.e. beginning with construction and testing of atomic modules. Integration testing documentation involves:

i. Find system as well as scheduler stability in terms of Pure read requests.
ii. Find system as well as scheduler stability in terms of Pure write requests.
iii. Find system stability as well as scheduler stability as a combination of above two cases.

After integration testing is complete and the errors detected are fixed, regression testing is required to ensure that the changes made to the software have not introduced new errors.

6.2.2.3 Validation testing

The following validation criteria are to be met by the software:

i. Validating that requests are added to appropriate read and write queues.
ii. Validating that requests are not indefinitely starved.
iii. Validating that all scheduled requests are dispatched.

6.2.3 Testing resources and staffing

Resources used for testing are:

i. Intel x86 core 2 Duo machine with Ubuntu 12.04
ii. Linux kernels 3.7.0
iii. Kingston 60GB SSD
iv. Filebench utility
v. TIOBench
vi. Dbench
vii. FIO

6.2.4 Test work products

The work products produced as a consequence of the testing strategy are identified.

6.2.5 Test record keeping

Mechanisms for storing and evaluating test results are specified.

6.2.6 Testing tools and environment

The coding for this project has been completely done in the ext4 file system of the Linux kernel. We have used print statements for debugging wherever possible.

The use of printf statements was primarily used for debugging the code that we have written in the ext4 file system. In order to actually perform stress related tests we have performed stress tests using Filebench, FIO, tioBench. Different tests were performed to measure the performance of scheduler for various workloads such as sequential reads, sequential writes, random reads, random writes, Fileserver, Mailserver and Webserver.

6.2.7 Test schedule

A detailed schedule for unit, integration, and validation testing as well as high order tests is described.

Task	**No. of people involved**	**Man hours required**	**No. of days**	**Start date**	**End date**
Unit Testing	4	15	3	25 February, 2014	27 February, 2014
Integration Testing	4	20	6	1 March, 2014	6 March, 2014
Validation Testing	4	25	8	10 March, 2014	17 March, 2014

Table 6.1: Test Schedule

6.3 Test Procedure

This section describes as detailed test procedure including test tactics and test cases for the software.

6.3.1 Software (SCIís) to be tested

We are going to test the project “Veloces : An Efficient I/O Scheduler for Solid State Devices” which aims at implementing the I/O scheduler for solid state devices. An I/O Scheduler is the one which decides the order of block I/O operation and resides in the block layer of the Linux kernel. The I/O Scheduler ensures that the I/O requests are issued within a particular deadline and that requests are completed within minimum time.

6.3.2 Testing procedure

The overall procedure for software testing is described.

6.3.2.1 Test Items:

i. Process user input
ii. Compare Algorithm
iii. Interpret kernel stream
iv. Generate difference

6.3.2.2 Integration testing

The integration testing procedure is specified.

Testing Procedure for Integration

Bottom up integration testing begins construction and testing with atomic modules i.e. components at the lowest levels in the program structure.

Individual components after unit testing are integrated and tested. Each time a new component is added to the system an integration test is performed. Integration is done in hierarchy of how components are built.

Test Cases and their purpose and their expected results

Test Case No.	**Description**	**Result**
1	Check whether application is running after integration.	PASS
2	Check individual components working.	PASS
3	Retest Individual components after integration.	PASS
4	Complete system test.	PASS

Table 6.2 Integration Testing

6.3.2.3 Validation testing

Pass/fail criterion for all validation tests

Test Case No	**Description**	**Result**
1	**Check whether utility is running without any errors**	**PASS**
2	**Check whether requests are added to appropriate read or write queue**	**PASS**
3	**Check whether the scheduled requests are dispatched**	**PASS**
4	**Check whether no requests are starved infinitely**	**PASS**
5	**Check whether there are no drop of requests**	**PASS**

Table 6.3: Validation Testing

6.3.2.4 High-order testing (System Testing)

The high-order testing procedure is specified. For each of the high order tests specified below, the test procedure, test cases, purpose, specialized requirements and pass/fail criteria are specified.

6.3.2.5 Performance testing

The test setup for performance of the proposed system includes a single machine, M1. A ext4 filesystem running on 60 GB Solid State Device. The performance test is carried out with a view of having a comparison of the current I/O Schedulers and the proposed I/O Scheduler. For various performance test operations like creation, deletion, rename etc are performed between the schedulers.

The following are the results obtained in M1 by selecting Noop, Deadline and CFQ schedulers and once with Veloces. The Fig 3.2 shows a comparison of the Input Output Operations performed per second (IOPS) on the three existing schedulers compared with the Veloces scheduler for various workloads Mailserver, Fileserver and Webserver. The Fig 3.3 shows a comparison of the bandwidth (MB/s) on the three existing schedulers compared with the Veloces scheduler for varying number of threads running together.

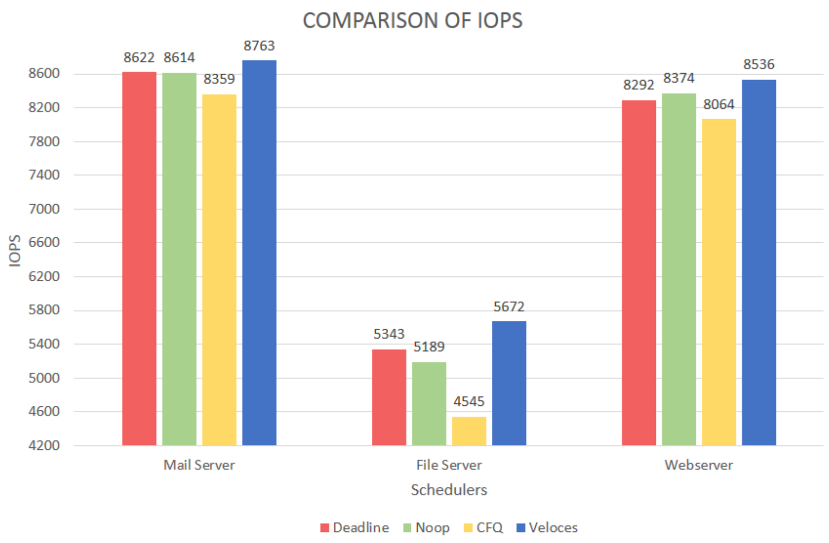


Fig. 6.2 Comparison of IOPS

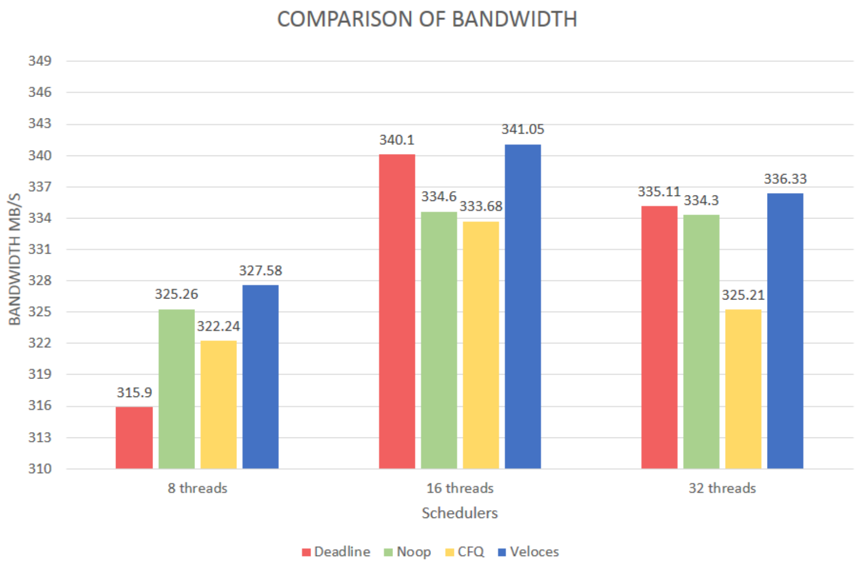


Fig. 6.3 Comparison of Bandwidth

7. FUTURE ENHANCEMENT

- In the future, we plan to add the multi-queue capability in order to leverage the inherent parallelism of SSDs.

- We plan to make use of the I/O hints that can be obtained from the applications. The I/O hints give information regarding the nature of data whether is sequential or random.

8. SUMMARY AND CONCLUSION

In conclusion, Flash-based storage devices are capable of alleviating I/O bottlenecks in data-intensive applications. However, the unique performance characteristics of Flash storage must be taken into account in order to fully exploit their superior I/O capabilities while offering fair access to applications.

We observed that the lack of seek/rotation overhead eliminates the performance benefit of anticipatory I/O, but proper I/O anticipation is still needed for the purpose of fairness. Based on these motivations, we designed a new Flash I/O scheduling approach that contains three essential techniques to ensure fairness with high efficiency—read preference, selective bundling of write requests and front merging of the requests.

We implemented the above design principles in our scheduler and tested it using FileBench as the benchmarking tool. The performance of our scheduler was consistent across various workloads like File Server, Web Server and Mail Server.

9. REFERENCES

1 *M. Dunn and A.L.N. Reddy, A new I/O scheduler for solid state devices. Tech. Rep. TAMU-ECE-2009-02, Department of Electrical and Computer Engineering, Texas A&M University, 2009.*

2 *J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S.H. Noh, Disk schedulers for solid state drivers. In Proc. EMSOFT (2009), PP. 295-304.*

3 *Wang H., Huang P., He S., Zhou K., Li C., and He X. A novel I/O scheduler for SSD with improved performance and lifetime. Mass Storage Systems (MSST), 2013 IEEE 29th Symposium, 1-5.*

4 *S. Kang, H. Park, C. Yoo, Performance enhancement of I/O scheduler for solid state device. In 2011 IEEE International Conference on Consumer Electronics, 31-32*

5 *S. Park and K. Shen, "Fios: A fair, efficient flash i/o scheduler," in FAST, 2012.*

6 *Y. Hu, H. Jiang, L. Tian, H. Luo, and D. Feng, "Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity," in Proceedings of the 25th International Conference on Supercomputing (ICS'2011), 2011.*

7 *J. Axboe. Linux block IO — present and future. In Ottawa Linux Symp., pages 51–61, Ottawa, Canada, July 2004.*

8 *S. Park and K. Shen. A performance evaluation of scientific I/O workloads on flash-based SSDs. In IASDS'09:Workshop on Interfaces and Architectures for Scientific Data Storage, New Orleans, LA, Sept. 2009.*

9 *J. Lee, S. Kim, H. Kwon, C. Hyun, S. Ahn, J. Choi, D. Lee, and S. H. Noh. Block recycling schemes and their cost-based optimization in NAND Flash memory based storage system. In EMSOFT'07: 7th ACM Conf. on Embedded Software, pages 174–182, Salzburg, Austria, Oct. 2007.*

10 *S. Park , E. Seo , J. Shin , S. Maeng and J. Lee. Exploiting Internal Parallelism of Flash-based SSDs. In IEEE computer architecture letters, Vol. 9, No. 1, JANUARY-JUNE 2010*

10. ANNEXURE

ANNEXURE A

PROJECT ANALYSIS OF ALGORITHMIC DESIGN

MATHEMATICAL MODEL OF SYSTEM

Relevant mathematics associated with the Project -

S= {F, LK, SA, I, O}

F= File System Structure.

LK= Linux Kernel.

SA=Scheduler Algorithm.

I = {I/O Request Queue, I/O Hints}

I/O Request Queue = {I/O Request 1, I/O Request 2 …… I/O Request n}

O = {I/O Dispatch Queue}

I/O Dispatch Queue = {I/O Dispatch Request 1, I/O Dispatch Request 2 …… I/O Dispatch Request n}

ANALYSIS

P-CLASS – A PROBLEM belongs to P-class if it can be solved in deterministic polynomial time, or a DTM(Deterministic Turing Machine) can be constructed in polynomial time.

NP-CLASS – A problem belongs to NP-class if it can solved in non-deterministic polynomial time, or a NTM(Non-deterministic Turing Machine) can be constructed in polynomial time.

Our problem definition belongs to P-class since it can be solved in deterministic polynomial time.

The solutions provided to the problem are deterministic in nature and were completed in polynomial time.

ANNEXURE B

PROJECT QUALITY AND RELIABILITY TESTING OF PROJECT DESIGN

OVERLOADING AND MORPHISM

Since Linux is written in C programming language, it does not directly support function overloading or morphism. However, it is achieved through the use of function pointers.

1. The header file elevator.h defines the functions as –
   typedef <return type> (name) (parameters list)

2. Next, a function pointer is created for the given function.
   name *func_ptr

3. Each scheduler defines its own set of functions. These functions are accessed by the func_ptr, when the scheduler is in use.
   *func_ptr = function_name

4. Thus, function overloading is achieved in C.

Eg:

```
typedef int (elevator_dispatch_fn) (struct request_queue *, int);
elevator_dispatch_fn *elevator_dispatch_fn;
```

For noop scheduler

```
elevator_dispatch_fn        = noop_dispatch
```

Similarly for cfq,

```
elevator_dispatch_fn        = cfq_dispatch
```

DESIGN TESTING STRATEGIES

1. **Boundary value analysis**

The boundary between two partitions is the place where the behavior of the application changes and is not a real number itself. The boundary value is the minimum (or maximum) value that is at the boundary. Boundary value analysis does not require invalid partitions.

2. Decision table

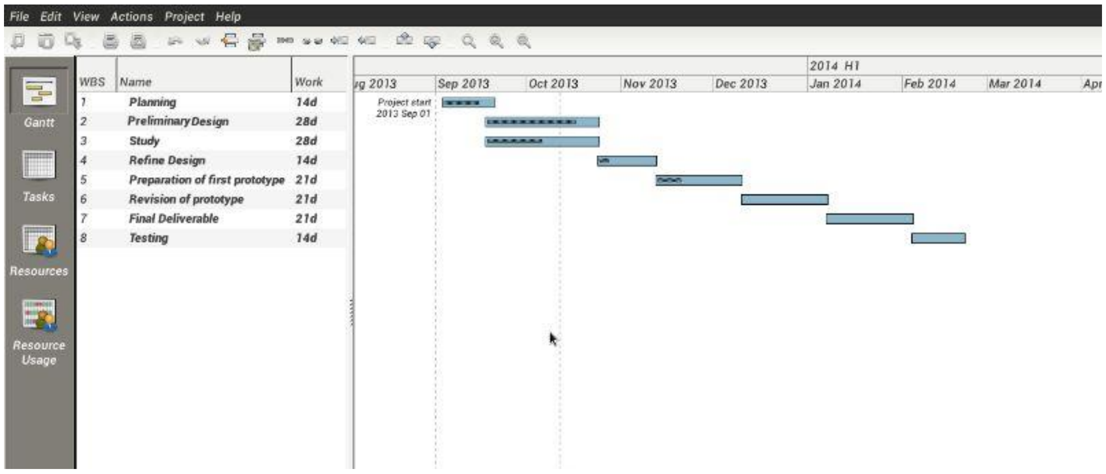
Some decision tables use simple true/false values to represent the alternatives to a condition, other tables may use numbered alternatives, and some tables even use fuzzy logic or probabilistic representations for condition alternatives. In a similar way, action entries can simply represent whether an action is to be performed (check the actions to perform), or in more advanced decision tables, the sequencing of actions to perform.

3. Design testing using use cases

Approaches to documenting use cases that are more formal cover not only typical and exceptional workflows, but also explicit identification of the actor, the preconditions, the post conditions, the priority, the frequency of use, special requirements, assumptions, and potentially more. The formal approach might also entail the creation of a use case diagram that shows all the actors, all the use cases, and the relationship between the actors and the use cases.

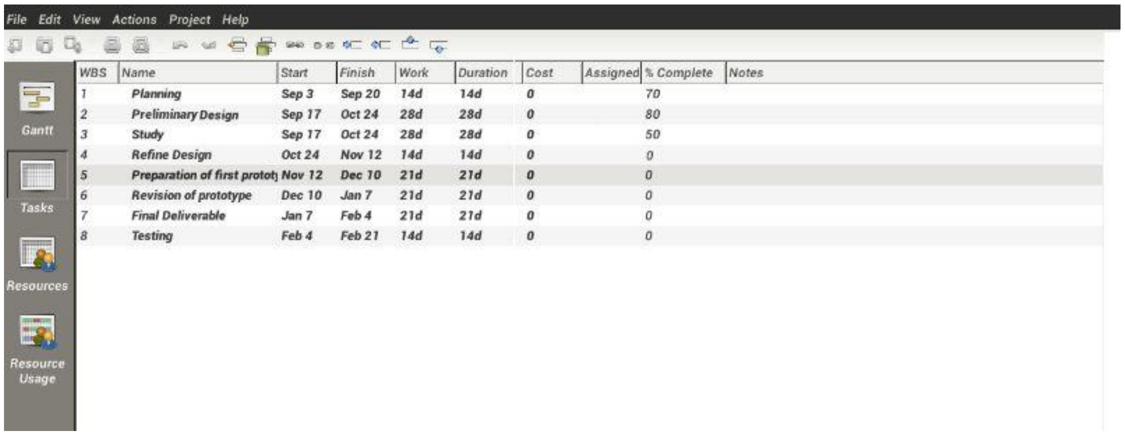
ANNEXURE C

PROJECT PLANNER



WBS	Name	Work
1	Planning	14d
2	Preliminary Design	28d
3	Study	28d
4	Refine Design	14d
5	Preparation of first prototype	21d
6	Revision of prototype	21d
7	Final Deliverable	21d
8	Testing	14d

Fig 10.1 Project Timeline



WBS	Name	Start	Finish	Work	Duration	Cost	Assigned	% Complete	Notes
1	Planning	Sep 3	Sep 20	14d	14d	0		70	
2	Preliminary Design	Sep 17	Oct 24	28d	28d	0		80	
3	Study	Sep 17	Oct 24	28d	28d	0		50	
4	Refine Design	Oct 24	Nov 12	14d	14d	0		0	
5	Preparation of first protot	Nov 12	Dec 10	21d	21d	0		0	
6	Revision of prototype	Dec 10	Jan 7	21d	21d	0		0	
7	Final Deliverable	Jan 7	Feb 4	21d	21d	0		0	
8	Testing	Feb 4	Feb 21	14d	14d	0		0	

Fig 10.2 Task set

ANNEXURE D

REVIEW OF DESIGN, BLACK BOX TESTING AND CORRECTIVE ACTION IN DESIGN

INTRODUCTION

Black-box testing is a method of where we test the functionality of an application regardless its internal structures or workings. Specific knowledge of the application's code/internal structure and programming knowledge in general is not required.

Considering the above principle we tested our project in various different aspects under black box testing. Aspects of black box testing:

1. Spike test:

The input to the system was changed rapidly and in very high values. Test: The system was tested over 4-5 VM with a single core. The memory and time required was well-balanced without any intermittent spikes.

2. Load Test:

A load test is usually conducted to understand the behavior of the system under a specific expected load. Test: We tested the system with sufficient load; diverse, independent strings for grammar generation and many number of processes for multi-processing aspect. The results obtained were satisfactorily optimized.

3. Configuration Test:

The system was tested with changes in configuration of the system. viz it was tested with selected test cases on a dual core 2.5ghz , core i5-2100 CPU at 3.1ghz and with multiple processes running on system.

2. Integration Testing:

Test: Testing whether the different modules of grammar generation, core scheduler, process handling work in tandem successfully or failure happens. The demo run of the project included different test cases, and for each test case, all modules were run sequentially as per design.

3. System testing

4. GUI testing: Not required.

5. Load testing: A load test is usually conducted to understand the behavior of the system under a specific expected load. Test: We tested the system with sufficient load; diverse, independent strings for grammar generation and many number of processes for multi-processing aspect. The results obtained were satisfactorily optimized.

ANNEXURE E

PROJECT WORKSTATION AND INSTALLATION REPORT

Project Workstation Selection

1. Hardware Requirements:

Hardware	Minimum	Suggested
CPU	1GHz	2 x 2GHz
Memory	2GB	4GB
Disk	SATA SSD	SATA SSD
Disk Space	5GB	20GB

Table 10.1 : Hardware Requirements

2. Software Requirements:

- Ubuntu 12.04 Ext-4 formatted.
- Emacs IDE 24.3
- gcc Compiler-4.4

Installation and Setup

Format external drive with ext4

```
Mkfs ext4 <path to device>
```

Compile the linux-master folder

```
1. make –j 2
2. make modules
3. make modules_install
4. make install
```

Change the scheduler

1. echo veloces > /sys/block/sda/queue/scheduler

Check the current scheduler

1. cat /sys/block/sda/queue/scheduler

ANNEXURE F

MODULE DEVELOPMENT PLAN AND MODULE DESCRIPTION

No.	Name of Module	Parts in the Module	Name of Student
1.	Accept Request	- Study of existing code. - Understanding parameters Involved. - Functions to be added.	Vishakha Damle
2.	Process Request	- Study of existing scheduler. - Understanding the function calls and usage. - Modifications and new functions to be added.	Amogh Palnitkar
3.	Schedule Request	- Working and using it through calls to be made from the block layer.	Sarvesh Rangnekar
4.	Performance Testing	- Benchmarking against existing schedulers. - Using filebench command and its options. - Generating graphs with the data.	Om Pawar

Table 10.2 Module Development Plan and Description

ANNEXURE G

WHITE BOX TEST PLAN, TESTING AND TEST REPORT

There are six basic types of testing: unit, integration, function/system, acceptance, regression, and beta. White-box testing is used for three of these six types:

I. Unit Testing: Unit testing is important for ensuring the code is solid before it is integrated with other code. Also, since the coder writes and runs unit tests him or herself, there is no track of the unit test failures in organizations.
Test: The each module function was unit tested separately.
   a. Result: Pass.

II. Integration testing:
   a. Test: Test cases were written which explicitly examine the interfaces between the various units viz. user command preprocessing, kernel stream decoder.
   b. Result: Pass

III. Regression testing: This can be tested with both black box and white box methodologies.

Test cases of white box testing:

Using the white-box testing we designed test cases that
I. Exercise independent paths within a module or unit.
   A stream of read requests to check if the read requests are handled correctly.
   A stream of write requests to check if the write requests are handled correctly.

II. Exercise logical decisions on both their true and false side.
   Mixture of read write requests to check that requests are properly classified

III. Input to functions are provided such that the logical decisions are tested for both true and falsity of the function logic.
   Various workloads were simulated to test whether the read preference and bundling operations were performed correctly

IV. Execute loops at their boundaries and within their operational bounds.
   The functions were tested with the inputs to them close to the boundary conditions of the data-types. We used the block boundary concept and the requests were sent to the scheduler which were interleaved.

Test Data

Measurement of IOPS for schedulers on various workloads using Filebench. (All readings are in IOPS)

	Deadline	Noop	CFQ	Veloces
Mail Server	8622	8614	8359	8763
File Server	5343	5189	4545	5672
Webserver	8292	8374	8064	8536

Table 10.3 Comparison of IOPS

Table 10.3 shows a comparison of the Input Output Operations performed per second (IOPS) on the three existing schedulers compared with the Veloces scheduler for various workloads Mailserver, Fileserver and Webserver. The measurement is done using Filebench. The results show that Veloces shows a consistent improvement over the existing schedulers for all the workloads.

Measurement of Bandwidth for schedulers using Dbench. (All readings are in MB/s)

	Deadline	Noop	CFQ	Veloces
8 threads	315.9	325.26	322.24	327.58
16 threads	340.1	334.60	333.68	341.05
32 threads	335.11	334.3	325.21	336.33

Table 10.4 Comparison of Bandwidth

Table 10.4 shows a comparison of the Bandwidth(MB/s) on the three existing schedulers compared with the Veloces scheduler for various number of threads running at a time. The measurement is done using Dbench. The results show that Veloces shows a consistent improvement over the existing schedulers in all the cases.

ANNEXURE H

PAPER

PAPER TITLE: Disk Scheduler for Solid State Devices

NAME OF CONFERENCE: Ottawa Linux Symposium, Canada.

PAPER ACCEPTED/ REJECTED: Paper accepted.

Veloces : An I/O Scheduler for Solid State Devices

Vishakha Damle, Amogh Palnitkar, Om Pawar, Sarvesh Rangnekar
Pune Institute of Computer Technology, India
{vishakha.22,amogh1337,om.pawar1992,sarveshr7}@gmail.com

Abstract

Solid State Devices (SSD) use NAND-based Flash memory for storage of data. They have the potential to alleviate the ever-existing I/O bottleneck problem in data-intensive computing environments, due to their advantages over conventional Hard Disk Drives (HDD). SSDs differ from traditional mechanical HDDs in various respects. The SSDs are built upon semiconductors exclusively, and have no moving mechanical parts. Hence, they are completely free from the rotational latency which dominates the disk access time of HDDs. This results in SSDs' operational speed being one or two orders of magnitude faster than HDDs.

However, on the other hand, due to the long existence of HDDs as persistent storage devices, conventional I/O schedulers are largely designed for HDDs, to mitigate the high seek and rotational costs in mechanical disks, through elevator-style I/O request ordering and anticipatory I/O. As a consequence, just by replacing conventional HDDs with SSDs in the storage systems without taking into consideration other properties like low latency, minimal access time and absence of rotary head, we may not be able to make the best use of SSDs.

We propose to implement an I/O scheduler which will leverage the inherent properties of SSDs. Since, SSD performs read operations faster than write operations, we designed the proposed scheduler to provide preference of reads over writes. Secondly, writing in the same block of the SSD is faster than writing to different blocks. Therefore, we plan to bundle write requests belonging to the same block. Lastly, serving a single large request is much more efficient than serving multiple small requests. So, both front and back merging of the requests are employed by our scheduler. This will enhance the overall performance of SSDs.

Index Terms—Operating System, SSD, I/O scheduler

I. INTRODUCTION

Solid State Devices (SSD) use NAND-based Flash memory for storage of data. They are very fast and have the potential to alleviate the ever-existing I/O bottleneck problem in data-intensive computing environments, due to their advantages over conventional Hard Disk Drives (HDD). SSDs differ from traditional mechanical HDDs in various respects. The SSDs are built upon semiconductors exclusively, and have no moving mechanical parts. Hence, they are completely free from the rotational latency which dominates the disk access time of HDDs. This results in SSDs' operational speed being one or two orders of magnitude faster than HDDs. However, on the other hand, due to the long existence of HDDs as persistent storage devices, the existing I/O scheduling algorithms have been specifically designed or optimized based on characteristics of HDDs.

Current I/O schedulers in the Linux Kernel are designed to mitigate the high seek and rotational costs in mechanical disks, through elevator-style I/O request ordering and anticipatory I/O. SSDs have many operational characteristics like low latency, minimal access time and absence of rotary head which need to be taken into account while designing I/O schedulers. As a consequence, by just replacing conventional HDDs with SSDs in the storage systems without taking into consideration their properties, we may not be able to make their best use, squandering the promising performance they can provide.

The rest of this paper is structured as follows. In Section II, SSD characteristics are elaborated. In Section III the existing schedulers are studied and their characteristics and features are compared. In Section IV, we elaborate on the design details of the proposed scheduler. Section V focuses on results and performance evaluation of our scheduler for different workloads. Finally, in Section VI we conclude this paper. Section VII covers the acknowledgements.

II. SSD CHARACTERISTICS

SSDs have been evolved from EEPROM (Electrically Erasable Programmable Read-Only Memory) which gives it distinctive properties. It consists of a number of blocks which can be erased independently. Block consists of pages. Read and write operations are performed at page level whereas erasing is done at block level. Overwriting is not allowed in SSDs. This makes the writes expensive. SSDs can only write to empty or erased pages. If it does not find any empty pages it finds an unused page and has to erase the entire block containing the page. Then it has to write the previous as well as the new page content on the block [10]. This makes SSDs slower over a period.

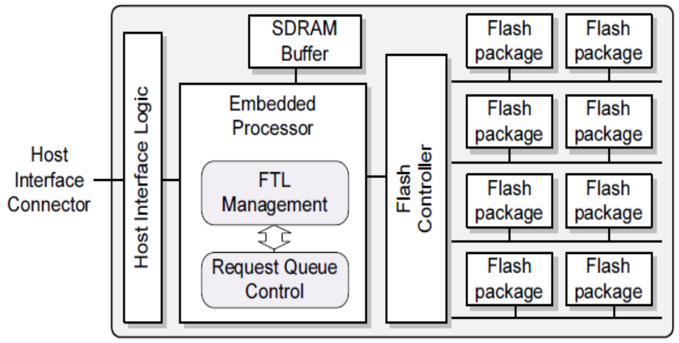


Fig. 1.1 SSD Model

SSDs make use of Flash Translation Layer (FTL) to provide a mapping between the Logical Block Address (LBA) and the physical media [10]. FTL helps in improving the SSD performance by providing Wear Leveling and Garbage Collection. Wear Leveling helps in even distribution of data over the SSDs so that all the flash cells have same level of use. Garbage collection keeps track of unused or 'stale' pages and at an opportune moment erases the block containing good and stale pages, rewrites the good pages to another block so that the block is ready for further writes.

Though SSDs provide a substantial improvement in I/O performance over the conventional HDDs, there is sufficient discrepancy between the read-write speeds. This is primarily due to erase-before-write limitation. In Flash based devices it is necessary to erase a previously written location before overwriting to the said location. This problem is further aggravated by erase granularity which is much larger than the basic read/write granularity. As a result read operations in SSDs tend to be relatively faster than writes.

As mentioned earlier, SSDs do not possess rotary drive. Therefore, access times of I/O operations are relatively less affected by spatial locality of the request as compared to traditional HDDs. However it has been observed that the I/O requests in the same block tend to be slightly faster than I/O request in different block.

We have considered these features of SSDs while designing our scheduler.

III. STUDY OF EXISTING SCHEDULERS

A. Noop scheduler

The name Noop (No Operation) defines the working of this scheduler. It does not perform any kind of sorting or seek prevention, thus is the default scheduler for flash based devices where there is no rotary head. It does not do anything to reduce the seek latency. It performs minimum operations on the I/O requests before dispatching it to the underlying physical device [1].

The only chore that a NOOP Scheduler performs is merging, in which it coalesces the adjacent requests if any. Besides this it is truly a No Operation scheduler which merely maintains a request queue in FIFO order [7].

As a result, it is suitable for SSDs, which can be considered as random access devices. However this might not be true for all workloads.

B. Completely Fair Queuing Scheduler

CFQ scheduler attempts to provide fair allocation of the available disk I/O bandwidth for all the processes which requests an I/O operation.

It maintains per process queue for the processes which request I/O operation and then allocates time slices for each of the queues to access the disk [1]. The length of the time slice and the number of requests per queue depends on the I/O priority of the given process. The asynchronous requests are batched together in fewer queues and are served separately.

C. Deadline Scheduler

Deadline scheduler aims to guarantee a start service time for a request by imposing a deadline on all I/O operations.

It maintains two FIFO queues for read and write operations and a sorted Red Black Tree (RB Tree). The queues are checked first and the requests which have exceeded their deadlines are dispatched. If none of the requests have exceeded their deadline then sorted requests from RB Tree are dispatched.

IV. PROPOSED SCHEDULER

In this section, we will discuss the implementation of our proposed scheduler.

A. Read Preference

As mentioned earlier, Flash-based storage devices suffer from erase-before-write limitation. In order to overwrite to a previously known location, the said location must first be erased completely before writing new data. The erase granularity is much larger than the basic read granularity. This leads to a large read-write discrepancy [5][8]. Thus reads are considerably faster than writes.

For concurrent workloads with mixture of reads and writes the reads may be blocked by writes with substantial slowdown which leads to overall degradation in performance of the scheduler. Thus in order address the problem of excessive read blocked by write, reads are given higher preference.

B. Bundling of Write requests

In SSDs, it is observed that sequential writes are faster than random writes. This comes from the fact that random writes impose an additional overhead of visiting various logical blocks to write the data whereas sequential writes can be done by writing to a single block [2][9]. Therefore, we propose a time bounded bundling of write requests where write requests belonging to the same logical block are bundled together and written sequentially. Also, synchronous write requests are serviced as soon as possible.

Bundling time of these write requests can be adjusted according to the workloads to further optimize the performance.

C. Front Merging

Request A is said to be in front of request B when the starting sector number of request A is larger than the ending sector number of request B.

Correspondingly, request A is said to be behind request A when the starting sector of request B is greater than the ending sector of request B.

The current I/O schedulers in the Linux Kernel facilitate merging of contiguous requests into a larger request before dispatch because serving a single large request is much more efficient than serving multiple small requests. However only back merging of I/O requests is performed since they take into account the direction of the rotational head of the hard disk.

As SSDs do not possess rotational disks there is no distinction between backward and forward seeks. So, both front and back merging of the requests are employed by our scheduler.

V. EXPERIMENTAL EVALUATION

A. Environment

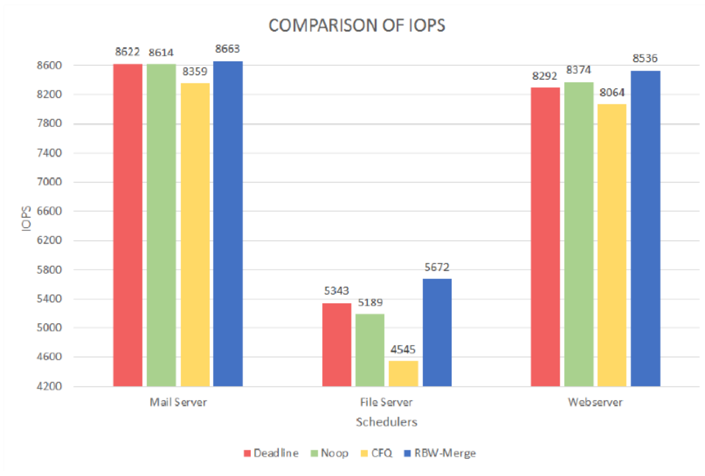
We implemented our I/O Scheduler with parameters displayed in Table 1.

Type	Specifics
CPU/RAM	Intel Core 2 Duo 1.80GHz
SSD	Kingston 60GB
OS	Linux-Kernel 3.12.4 / Ext-4 File System
Benchmark	Filebench Benchmark for Mail Server, Webserver and File Server workloads
Target	Our Scheduler and existing Linux I/O Schedulers

Table 1: System Specifications

B. Results

We used the FileBench benchmarking tool which generates workloads such as Mail Server, File Server and Webserver. The results of the benchmark are shown in Fig. 5.1



VI. CONCLUSION

In conclusion, Flash-based storage devices are capable of alleviating I/O bottlenecks in data-intensive applications. However, the unique performance characteristics of Flash storage must be taken into account in order to fully exploit their superior I/O capabilities while offering fair access to applications.

We observed that the lack of seek/rotation overhead eliminates the performance benefit of anticipatory I/O, but proper I/O anticipation is still needed for the purpose of fairness. Based on these motivations, we designed a new Flash I/O scheduling approach that contains three essential techniques to ensure fairness with high efficiency—read preference, selective bundling of write requests and front merging of the requests.

We implemented the above design principles in our scheduler and tested it using FileBench as the benchmarking tool. The performance of our scheduler was consistent across various workloads like File Server, Web Server and Mail Server.

VII. ACKNOWLEDGMENT

We would like to take this opportunity to thank our internal guide Prof. Girish Potdar, Pune Institute of Computer Technology for giving us all the help and guidance we needed. We are really grateful to him for his kind support throughout this analysis and design phase.

We are also grateful to other staff members of the Department for giving important suggestions.
Our external guides Mr.Swapnil Pimpale and Ms. Nafisa Mandliwala were always ready to extend their helping hand and share valuable technical knowledge about the subject with us.

We are thankful to PICT library and staff, for providing us excellent references, which helps us lot to research and study topics related to the project.

REFERENCES

[1] M. Dunn and A.L.N. Reddy, A new I/O scheduler for solid state devices. Tech. Rep. TAMU-ECE-2009-02, Department of Electrical and Computer Engineering, Texas A&M University, 2009.

[2] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S.H. Noh, Disk schedulers for solid state drivers. In Proc. EMSOFT (2009), PP. 295-304.

[3] Wang H., Huang P., He S., Zhou K., Li C., and He X. A novel I/O scheduler for SSD with improved performance and lifetime. Mass Storage Systems (MSST), 2013 IEEE 29th Symposium, 1-5.

[4] S. Kang, H. Park, C. Yoo, Performance enhancement of I/O scheduler for solid state device. In 2011 IEEE International Conference on Consumer Electronics, 31-32

[5] S. Park and K. Shen, "Fios: A fair, efficient flash i/o scheduler," in FAST, 2012.

[6] Y. Hu, H. Jiang, L. Tian, H. Luo, and D. Feng, "Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity," in Proceedings of the 25th International Conference on Supercomputing (ICS'2011), 2011.

[7] J. Axboe. Linux block IO — present and future. In Ottawa Linux Symp., pages 51–61, Ottawa, Canada, July 2004.

[8] S. Park and K. Shen. A performance evaluation of scientific I/O workloads on flash-based SSDs. In IASDS'09:Workshop on Interfaces and Architectures for Scientific Data Storage, New Orleans, LA, Sept. 2009.

[9] J. Lee, S. Kim, H. Kwon, C. Hyun, S. Ahn, J. Choi, D. Lee, and S. H. Noh. Block recycling schemes and their cost-based optimization in NAND Flash memory based storage system. In EMSOFT'07: 7th ACM Conf. on Embedded Software, pages 174–182, Salzburg, Austria, Oct. 2007.

[10] S. Park , E. Seo , J. Shin , S. Maeng and J. Lee. Exploiting Internal Parallelism of Flash-based SSDs. In IEEE computer architecture letters, Vol. 9, No. 1, JANUARY-JUNE 2010

ANNEXURE I

SYNOPSIS

Group Id: 18

Project Title: Veloces: An Efficient I/O Scheduler for Solid State Devices

Sponsorship: L3Cube

External Guide: Mr.Swapnil Pimpale, Ms.Nafisa Mandliwala

Internal Guide: Prof. G. P. Potdar

Title of the project:

Technical Key Words (Ref ACM Keywords):
D.4 Operating Systems
D.4.2 Storage Management
D.4.3 File Systems Management

Relevant mathematics associated with the Project:

S= {F, LK, SA, I, O}

F= File System Structure.

LK= Linux Kernel.

SA=Scheduler Algorithm.

I = {I/O Request Queue}

I/O Request Queue = {I/O Request 1, I/O Request 2 …… I/O Request n}

O = {I/O Dispatch Queue}

I/O Dispatch Queue = {I/O Dispatch Request 1, I/O Dispatch Request 2 …… I/O Dispatch Request n}

Names of at least two conferences where paper can be submitted:

1. Usenix

   www.usenix.com

2. Csail

   www.csail.com

3. Linux Symposium

   www.linuxsymposium.org/

4. ACM

5. IEEE Conference

   http://www.radarcon2014.com

Review of Conference/Journal Papers supporting project idea
(at least 10 papers + white papers or web references)

1. M.Dunn and A.L.N. Reddy, A new I/O scheduler for solid state devices. Tech. Rep. TAMU-ECE-2009-02, Department of Electrical and Computer Engineering, Texas A&M University, 2009.
2. J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S.H. Noh, Disk schedulers for solid state drivers. In Proc. EMSOFT (2009), PP. 295-304.
3. Wang H., Huang P., He S., Zhou K., Li C., and He X. A novel I/O scheduler for SSD with improved performance and lifetime. Mass Storage Systems (MSST), 2013 IEEE 29th Symposium, 1-5.
4. S. Kang, H. Park, C. Yoo, Performance enhancement of I/O scheduler for solid state device. In 2011 IEEE International Conference on Consumer Electronics, 31-32
5. S. Park and K. Shen, ”Fios: A fair, efficient flash i/o scheduler,” in FAST, 2012.
6. Y. Hu, H. Jiang, L. Tian, H. Luo, and D. Feng, Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity, in Proceedings of the 25th International Conference on Supercomputing (ICS2011), 2011.
7. J. Axboe. Linux block IO present and future. In Ottawa Linux Symp., pages 5161, Ottawa, Canada, July 2004.
8. S. Park and K. Shen. A performance evaluation of scientific I/O workloads on flash-based SSDs. In ASDS09:Workshop on Interfaces and Architectures for Scientific Data Storage, New Orleans, LA, Sept.
   2009.
9. J. Lee, S. Kim, H. Kwon, C. Hyun, S. Ahn, J. Choi, D. Lee, and S. H. Noh. Block recycling schemes and their cost-based optimization in NAND Flash memory based storage system. In EMSOFT07: 7th ACM Conf. on Embedded Software, pages 174182, Salzburg, Austria, Oct. 2007.
10. S. Park , E. Seo , J. Shin , S. Maeng and J. Lee. Exploiting Internal Parallelism of Flash-based SSDs. In IEEE computer architecture letters, Vol. 9, No. 1, JANUARY-JUNE 2010.