An Exploration of Hybrid Hard Disk Designs Using an Extensible Simulator

Pavan Konanki

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Ali R. Butt, Chair
Wu-chun Feng
Calvin J. Ribbens

December 1, 2008
Blacksburg, Virginia

Keywords: Hybrid hard disk, Simulator, Storage

© Copyright 2008, Pavan Konanki
An Exploration of Hybrid Hard Disk Designs Using an Extensible Simulator

Pavan Konaniki

(ABSTRACT)

The growing performance gap between CPUs and sustainable disk I/O is a major hurdle in supporting modern applications. As the CPUs become faster, this gap is projected to worsen, thus making it a critical problem that should be addressed with high priority. Although efficient algorithms have alleviated this problem, the mechanical nature of the disk places physical limits on the achievable speedup. On the other hand, newer technologies such as flash memory promise significant improvements in access time, power consumption, and storage density. However, the mature disk technology offers the most favorable cost per bit ratio. Since replacing standard hard disks with flash disks is prohibitively expensive, hybrid hard disks augment the standard hard disk with a small amount of flash memory. By exploiting the beneficial aspects of both technologies they aim to provide breakthrough increase in performance. Nevertheless, hybrid hard disks pose several significant design challenges. Effective and efficient algorithms to manage the flash, the disk, and interaction between them are required.

To facilitate rapid and easy exploration of the design space for hybrid hard disk algorithms we present the design and implementation of a flexible and extensible simulator that models hybrid hard disks. The simulator is flexible in that it models several configurations in which the flash and the magnetic medium interact. The simulator is extensible in that it provides a
simple framework to plug in several algorithms to manage the hybrid hard disk. We validate our simulator and analyze the performance of the hybrid hard disk for real workloads.
Acknowledgments

I thank my adviser Dr. Ali R. Butt for his support, suggestions, and guidance throughout my research. I thank my committee members Dr. Wu-chun Feng and Dr. Calvin J. Ribbens for their support and valuable comments. The independent and critical thinking that I developed during my Master’s will help me throughout my career.

I thank Hari Pyla for his unfailing constructive criticism and suggestions. I thank all my friends who have made my stay in Blacksburg enjoyable and memorable. I thank all the members of the NDSSL team at Virginia Bioinformatics Institute. It was an enriching experience working as a Research Assistant at VBI. I thank Virginia Tech for providing a stimulating environment during the course of my degree.

I thank my family for their unconditional love. I thank God for providing me with the strength to complete this thesis.
## Contents

1 Introduction 1

1.1 Motivation ................................................. 3

1.2 Contributions .................................................. 6

1.3 Outline .............................................................. 7

2 Background ......................................................... 8

2.1 Flash Memory ..................................................... 8

2.2 Hybrid Hard Disks .............................................. 12

2.3 Disksim Simulator ............................................... 14

3 Modeling Hybrid Hard Disks 17

3.1 Modeling the Flash Medium ..................................... 21

3.2 Hybrid Hard Disk in Configuration 1 .......................... 24
7 Conclusion 66

8 Future Work 69

Bibliography 71
List of Figures

3.1 Architecture of a hybrid hard disk ........................................ 18
3.2 Hybrid hard disk in Configuration 1 ..................................... 19
3.3 Hybrid hard disk in Configuration 2 ..................................... 19
3.4 Disksim configuration to simulate a flash disk ......................... 22
3.5 A framework to plug in new FTL algorithms ............................ 24
3.6 A Disksim model representing a hybrid hard disk in the first configuration ............................ 25
3.7 Interface to plug in a flash management policy into the hybrid controller .................................. 28
3.8 A Disksim model representing a hybrid hard disk in the second configuration ......................... 29
4.1 Algorithm to service a write request under page level FTL ............ 34
4.2 Algorithm to service a write request under log block FTL .......... 35
4.3 Merge operation under the log block FTL ............................. 36
4.4 Algorithm to service a write request under ANAND FTL .......... 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Algorithm to service a read request under ANAND FTL</td>
<td>38</td>
</tr>
<tr>
<td>4.6</td>
<td>Algorithm to service a write request under FMAX FTL</td>
<td>39</td>
</tr>
<tr>
<td>4.7</td>
<td>Algorithm to service a read request under FMAX FTL</td>
<td>40</td>
</tr>
<tr>
<td>4.8</td>
<td>Algorithm to service a read request at the hybrid controller</td>
<td>41</td>
</tr>
<tr>
<td>4.9</td>
<td>Algorithm to service a write request at the hybrid controller</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>Observed and simulated response time for the Bonnie workload</td>
<td>50</td>
</tr>
<tr>
<td>5.2</td>
<td>Observed and simulated response time for the Download workload</td>
<td>50</td>
</tr>
<tr>
<td>5.3</td>
<td>Observed and simulated response time for the MP3 workload</td>
<td>51</td>
</tr>
<tr>
<td>5.4</td>
<td>Observed and simulated response time for the Untar workload</td>
<td>51</td>
</tr>
<tr>
<td>5.5</td>
<td>Observed and simulated response time for the Postmark benchmark</td>
<td>52</td>
</tr>
<tr>
<td>5.6</td>
<td>Observed and simulated response time for a hybrid hard disk in Configuration 1</td>
<td>52</td>
</tr>
<tr>
<td>5.7</td>
<td>Wear leveling statistics for writing a 85 MB file over a 100 MB partition</td>
<td>54</td>
</tr>
<tr>
<td>5.8</td>
<td>Wear leveling statistics for writing a 85 MB file fifteen times over a 1 GB partition</td>
<td>54</td>
</tr>
<tr>
<td>5.9</td>
<td>Response times for the standard and hybrid disk for the Firefox workload</td>
<td>58</td>
</tr>
<tr>
<td>5.10</td>
<td>Response times for the standard and hybrid disk for the Dia workload</td>
<td>58</td>
</tr>
<tr>
<td>5.11</td>
<td>Response times for the standard and hybrid disk for the MP3 workload</td>
<td>59</td>
</tr>
</tbody>
</table>
5.12 Response times for a standard and hybrid disk for the Untar workload . . . 59

5.13 Response times for a standard and hybrid disk for the Postmark benchmark 60
## List of Tables

4.1 Parameters characterizing the flash medium ........................................... 32

5.1 Characteristics of I/O benchmarks used for validation .............................. 49

5.2 Fraction of random write requests in each benchmark used for validation ... 49

5.3 Characteristics of I/O benchmarks used for analysis ............................... 57

5.4 Fraction of random write requests in each benchmark used for analysis ... 57
Chapter 1

Introduction

The growing performance gap between CPUs and sustainable disk I/O is a major hurdle in supporting modern applications. As the CPUs become faster, this gap is projected to worsen, thus making it a critical problem that should be addressed with high priority. Although efficient algorithms have alleviated this problem, the mechanical nature of the disk places physical limits on the achievable speedup.

On the other hand, newer technologies such as flash memory deliver significant improvements in access time, power consumption, and storage density. Flash memory [31] is non-volatile storage that persists data using the storage of charge. In flash memory, the presence of charge indicates a 0 or 1. The charge can be moved to or from the transistor with an erase or write operation. By electronically storing data flash memory eliminates expensive seek delays that arise from a mechanical head. Therefore, flash provides excellent random read
performance and significantly better random write performance.

However, the mature disk technology offers the most favorable cost per bit ratio. Since replacing standard hard disks with flash disks is prohibitively expensive, hybrid hard disks augment the standard hard disk with a small amount of flash memory, with the goal of creating a beneficial effect from the synergistic interaction between flash memory and the magnetic medium. The benefit usually is improved performance and/or reduced power consumption. The flash memory within the disk can be used for several purposes such as to reduce boot time, improve read response time for frequently accessed data, or to act as a write buffer for the magnetic medium. Further, flash memory can interact with magnetic storage in several configurations. For example, the flash memory can either be managed by the operating system or by the disk controller. Effective algorithms and favorable configurations for hybrid hard disks are actively researched. Understanding the characteristics of hybrid hard disks will not only aid in designing suitable algorithms but will also lay the foundation for designing the next generation of hybrid storage devices that could offer breakthrough I/O performance by combining the beneficial aspects of disparate storage technologies. For example, we can explore the possibility of coupling Micro Electro-Mechanical System (MEMS) based storage devices with standard hard drives or even flash drives.

To facilitate rapid and easy exploration of the design space for hybrid hard disk algorithms, we design and implement a simulator to model hybrid hard disks. Our simulator is implemented as a module that plugs into the Disksim simulator [16]. We chose to extend the Disksim simulator since it is a highly accurate storage system simulator. Disksim is widely
used in the research community \cite{13, 39, 15, 14, 33} and is under active development. Its most recent version includes a model of a MEMS-based storage device \cite{17, 37}.

Our simulator is flexible in that it models several configurations in which the flash and the magnetic medium interact. For example, in addition to all configurations supported by the Disksim simulator, the hybrid hard disk can be used in a configuration where the flash is exposed to the operating system or in a configuration where the flash is completely managed by the disk controller. The simulator is extensible in that it provides a simple framework to plug in several algorithms to manage the disk. More specifically, FTL and cache management algorithms can be easily plugged into the simulator. We validate our simulator and analyze the performance of the hybrid hard disk for real workloads.

1.1 Motivation

As computing speed increases much faster than disk speed, the disk has become the bottleneck for several applications. New storage technologies such as flash storage and MEMS-based storage promise increased storage density, reduced access times, and lower power consumption. However, the mature disk technology provides reasonable performance and the most favorable cost per bit ratio. To simultaneously utilize the beneficial aspects of two or more technologies, recent research explores the prospect of combining the newer flash technology with standard hard disks. The result of this coupling is the hybrid hard disk, which claims to provide improved performance and reduced power consumption. However,
the addition of flash memory to disks creates several design challenges.

For example, algorithms need to accommodate the fact that a block in flash needs to be erased before being rewritten. Further, the life of the device is restricted by the number of times a block can be erased, requiring FTL algorithms to aim at wearing down all blocks uniformly. Moreover, flash memory also exhibits access latencies that are very different from standard hard disks. Flash memory offers fast read access times for both sequential and random reads and sequential writes, whereas it suffers from a slow write access time for random writes since blocks need to be erased before they are rewritten. To accommodate such different characteristics, caches need to be flash aware and filesystems must be tailored to leverage the beneficial aspects of flash memory and mask its shortcomings. A FTL algorithm is an algorithm running over the flash to provide wear leveling and to emulate a block device interface similar to that of a standard hard disk. Designing such algorithms is also a complex task that requires considering the unique aspects of flash memory.

The marriage of flash memory with standard magnetic storage results in more complexity brought about by the interaction of these two disparate storage technologies. Since the flash memory on disk is much larger than the on-disk RAM cache, it is possible to use the flash for several purposes. In order to identify beneficial ways of using the flash cache, a clear understanding of the characteristics of both forms of media and their interaction is essential. The flash cache can be managed either by the operating system or by the disk controller. In either case, algorithms must address issues such as what data resides on the cache, what data is directly written to disk, when the data on flash is flushed to disk, how much of the
data is flushed at a time, and which of the pages must be evicted from the flash during a cache flush. Further, design choices pertaining to the use of flash memory as merely a write cache or as both a read and write cache must also be assessed.

Several of the discussed algorithms are complex to implement since they entail extensive changes to the operating system kernel or to firmware. An accurate, extensible, and flexible simulator is an excellent tool to study algorithms and configurations of hybrid hard disks since it provides total control over every subsystem, enables easy modification of algorithms at every level in the storage hierarchy, and provides common ground for comparison of various algorithms and configurations. For example, it would enable us to evaluate FTL algorithms in light of their wear leveling and access time characteristics or evaluate different caching strategies for the flash cache. Ultimately, the simulation environment will help us identify application classes that can benefit from the hybrid hard disk.

In addition to evaluating the hybrid hard disk in isolation, by virtue of the Disksim simulator, we can evaluate the performance of a hybrid hard disk in conjunction with other storage technologies supported by the simulator. For example, it is straightforward to simulate a RAID array of hybrid hard disks, or a combination of hybrid hard disks and standard hard disks. The insights gained from such studies can open up opportunities for a wide range of novel storage systems that combine several different storage technologies.

For the lessons learned from the simulator to apply to a real system, we need to ensure that it accurately models the real system. To this end, we validate our modules against observed behavior for real workloads. We also compare the performance of a hybrid hard disk with
that of a standard disk for real workloads.

1.2 Contributions

Specifically, we make these contributions in this thesis:

- Design and implementation of a flash model in Disksim to simulate solid state drives either as stand alone devices or in combination with other devices as part of a hybrid device.

- Design and implementation of a model of two configurations of a hybrid hard disk in Disksim. In the first configuration the flash memory within the hybrid hard disk is managed by the operating system and in the second, by the disk controller.

- An extensible framework to plug in algorithms to manage the hybrid hard disk. Algorithms that can be plugged into the simulator using our framework include flash translation layer algorithm, algorithms to selectively direct data to the flash, cache replacement policies for the flash and policies to flush the content of the flash to disk.

- Evaluation of the simulator. Specifically, we validate the flash model and hybrid hard disk model using various workloads. Further, we analyze the performance of the hybrid hard disk and compare it to that of a standard disk for various workloads.
1.3 Outline

Chapter 2 presents concepts, terminology, and technologies that serve as a background for our work. Chapter 3 discusses the process of modeling hybrid hard disks, the design of the hybrid hard disk simulator, and its extensible framework-based design. Chapter 4 details the algorithms that drive the hybrid hard disk model. Chapter 5 presents our evaluation of the simulator. We validate the simulator and analyze the performance of the hybrid hard disk against real workloads. Chapter 6 presents related work and prior research in this area. Finally, in Chapter 7 and Chapter 8 we conclude and discuss future directions for this work, respectively.
Chapter 2

Background

In this chapter, we present the background, define terminology, and introduce concepts relevant to this thesis. In Section 2.1 we introduce flash memory or solid state storage and the Flash Translation Layer. In Section 2.2 we discuss the hybrid hard disk. Finally, in Section 2.3 we discuss the Disksim simulator and its architecture.

2.1 Flash Memory

Flash memory [31] is non-volatile storage that persists data using the storage of charge. In flash memory, the presence of charge indicates a 0 or 1. The charge can be moved to or from the transistor with an erase or write operation.

Two kinds of flash memory are widely used: NOR flash and NAND flash [8]. While NOR
flash is used for code storage on portable devices, NAND flash is designed to have a low cost per bit ratio and is optimized for mass data storage. The characteristics of NOR flash are lower density, fast read speed, slow write speed, slow erase speed, and a random access interface. The characteristics of NAND flash are high density, medium read speed, high write speed, high erase speed, and an indirect or I/O like access [8]. In this thesis we focus on NAND type flash memory, the type used in mass storage devices.

In comparison to a standard disk, a storage device based on flash memory provides:

- Faster random read performance due to the lack of a moving head, eliminating the long seek delay present in standard hard disks [23]
- Comparable performance for sequential reads and writes [10]
- Slower random writes performance due to physical characteristics of the flash medium [23]
- Uniform latencies for accessing any part of device due to the lack of seek delays

While data can directly be written to unwritten portions of the flash medium, written portions must be erased before being rewritten. The minimum amount of data that can be read or written is referred to as a page. The minimum amount of data that can be erased, referred to as a block, consists of several pages since the erase operation is much slower than the read and write operations. The size of a page is between 1 byte to 2 KB and the size of a block is between 4 KB and 128 KB [24]. Further, a block on flash can be erased a limited number of times after which it becomes ineffective for the retention of charge. Each page
on the flash has some additional space, typically 128 bytes, to store metadata information. This space could hold information such as the logical to physical mapping, erase count, and error codes.

To provide wear leveling and an interface similar to that of a standard hard disk, the flash medium uses an additional software layer known as the Flash Translation Layer (FTL) [5]. The FTL provides wear leveling by exposing a logical address space and dynamically mapping it to the physical address space. This logical to physical mapping changes as data is read from and written to the flash memory. Since the same logical page is mapped to different physical pages over time, blocks are worn out uniformly even if the workload repeatedly writes to a specific set of blocks. Flash memory’s random write performance depends on the effectiveness of the FTL.

Several FTL algorithms that exploit the special characteristics of the flash medium to provide fast response times while working with the limited computing power of the controller and the limited on-disk RAM have been designed and published in research literature and patents. We discuss some of these algorithms in this section.

The page mapping FTL [20, 23] was used in earlier log-structure type [34] FTL algorithms that appended logical pages to a log and garbage collected old pages, keeping the updated logical to physical page mapping in memory. Page mapping FTL is flexible since a logical page can be mapped to any physical page. However, as the amount of flash managed runs into several gigabytes, the number of pages also increases, resulting in a large logical to physical mapping table. This increases the on-disk RAM requirement, raising the cost of the
device.

The block mapping FTL used in SmartMedia cards [11] maps a logical block to a physical block. The offset of a logical page within the logical block is the same as the offset of the physical page in the corresponding physical block. Since the FTL only keeps track of blocks, the RAM requirement is much lesser than that of the page mapping algorithm. However, block mapping FTL suffers from the inefficiency that modifying a page might require a whole block update.

The log block FTL algorithm [24] manages a small number of log blocks at the page level and the rest of the data blocks at the block level. A logical page can be mapped to any page within the log block. Writes to a data block are first directed to a log block. When the log block is full it is merged with the data block. A merge is either a full merge or a switch merge. In a full merge, the valid data from the log block and the data block are copied to a free block, and the free block becomes the new data block. The log block and the data block become free blocks. In cases when the log block is written from the first page to the last page sequentially, a quicker switch merge is possible. In a switch merge, the log block becomes the data block and the old data block becomes a free block. The log block algorithm is discussed in detail in Section 4.1.

In ANAND FTL [2], the logical address is broken down into a logical block number and an offset of a page within the logical block. Each logical block is mapped to a chain of physical blocks. To read page $x$ in logical unit $n$, ANAND FTL sequentially searches the chain associated with logical block $n$. The last unit to contain data in the $x^{th}$ page is returned. In
other words, all blocks before this block have old data for the \(x^{th}\) page and all blocks after this block have a clean \(x^{th}\) page. To write the page \(x\) in logical block \(n\), the data is written to the first block in the chain corresponding to logical block \(n\) that has a free page at offset \(x\). To reclaim space, the chain is folded into the last block of the chain by copying valid data for each page into the last block. All the other units of the chain become free blocks [12]. ANAND FTL is discussed in detail in Section 4.1.

**FMAX FTL** [2], a variation of ANAND FTL, limits the chain length to two. The first and second blocks in the chain are called the primary and replacement units, respectively. The logical offset of a page is the same as its physical offset in the primary unit. The replacement unit does not preserve the logical offset of a page. When a page in the primary unit is updated the new data is written to some page in the replacement unit. Thus the replacement unit can contain several copies of the same logical page of which only one is the most recent. To reclaim space, valid pages from both the primary and the replacement units are copied to a new block, which then becomes the new primary unit. The old primary and replacement units are erased. To read a page, the replacement unit is searched followed by the primary unit. FMAX FTL is discussed in detail in Section 4.1.

### 2.2 Hybrid Hard Disks

A hybrid hard disk is different from a standard hard disk in that it provides an on-disk flash memory buffer in addition to the RAM buffer and magnetic storage. The flash memory on the
hybrid hard disk can be used to improve performance and to reduce power consumption [7]. For example, by storing data required for booting on flash, the operating system does not have to wait for the disk to spin up. Since the flash provides comparable performance to magnetic medium for sequential reads and better performance for random reads, the time required to boot up the machine can be reduced by reading data required for booting from flash. Microsoft Windows Vista uses the name ReadyDrive to refer to this technique [6].

Manufacturers claim that the hybrid hard disk has several benefits. In a hybrid hard disk, data can be offloaded to flash and periodically flushed to disk, giving the magnetic medium the opportunity to remain in the spun down state for a longer duration. Since the flash medium consumes lesser power than the rotating magnetic medium [30], a hybrid hard disk can prove to be more power efficient than a standard hard disk. Since the flash medium provides superior performance for random reads compared to a standard hard disk, a hybrid hard disk can be used to speed up booting or to resume quickly from hibernate. By judiciously selecting the data to offload to flash, the benefits of both technologies can be exploited to develop a faster, low power storage device.

On the flip side, the cost of the hybrid hard disk increases with the amount of flash memory present on it. The presence of flash memory could also complicate recovery algorithms, since the data on flash memory that has not been flushed to disk must be considered in the event of a crash. Another unfavorable side effect is that read requests that miss in the buffer cache need to wait for the disk to spin up, extending the read response time. An efficient FTL layer is also required to overcome the slow random write performance of flash memory in
order to observe a significant performance improvement from the hybrid hard disk. While the above shortcomings can be overcome by efficient algorithms, designing such algorithms is a difficult problem.

2.3 Disksim Simulator

Disksim is an efficient, accurate, and configurable storage system simulator [16]. It is written in C and executes in user space. Disksim has a modular design, consisting of loosely coupled components that can be interconnected in several ways. It includes modules for most of the components in the storage hierarchy including device drivers, buses, controllers, adapters, and disk drives. It supports several trace formats and configurable synthetic workloads. Further, Disksim also provides an interface that enables it to be integrated into a system simulator.

A simulation begins by reading in the configuration file. The configuration file contains

- Parameters specification

Each module in Disksim is characterized by a set of parameters. While some modules such as the bus module are simple, others such as the disk drive module are extremely complicated requiring the specification of several parameters. Dixastrac [36], which was developed to solve this problem, is a tool to automatically extract about 100 performance critical parameters from SCSI drives. It runs in user space on Linux using the
/dev/sg generic SCSI interface.

• Component instantiations

An instance of each component required during the current simulation run is created.

• Topology specification

The topology specification describes the interconnections between components and is mostly independent of the components. The I/O driver module must be at the top of the hierarchy and a storage device must be at the bottom. The bus module is used to interconnect any two modules. A bus can have no more than 15 disks or controllers attached to it. In the current version, an instantiated component can occur only once in the topology specification, indicating that a device must be reachable from the I/O driver through exactly one path. Further, a controller supports a maximum of 4 back-end buses and one host side bus.

• Rotational synchronization delays

This section specifies rotationally synchronized devices that begin the simulation at the same rotational offset. Non-synchronized devices are assigned a random initial offset.

• Disk array organization

This section specifies striping and various RAID configurations.

• Synthetic workload specification

This section specifies representative synthetic workloads that drive the simulation.
The simulator runs as a single process with a single thread of execution. This approach increases the efficiency of the simulator [16]. The simulation maintains an internal list of events ordered by the simulated time, which acts as the monotonically increasing wall clock time in the simulator. As each event is extracted from the list of events, the simulated time is advanced. An event could trigger its own set of sub-events that are added to the global list of events and processed in a similar manner. The simulation ends when the last record in the trace file is read. Once the simulation ends, the statistics are finalized, formatted for printing, and written to an output file.

Implementing a new storage device in Disksim involves developing a new module that defines itself as a storage device and is capable of receiving messages from and responding to the controller module using the supported bus protocol. The flash memory module, which is one such device, is discussed in Section 3.1 and Section 4.1. Implementing an intelligent controller involves receiving messages from and responding to both the host side module and the storage device, complying with the bus protocol. Such a controller is discussed in Section 3.3 and Section 4.2.
Chapter 3

Modeling Hybrid Hard Disks

In this chapter, we discuss the design of the hybrid hard disk simulator.

A hybrid hard disk consists of flash memory in addition to magnetic storage and a DRAM buffer. Figure 3.1 shows the architecture of a hybrid hard disk. Current hybrid hard disks contain 80 GB or 160 GB of magnetic storage, 8 MB of DRAM buffer, and 256 MB of flash memory. In the following discussion, we refer to the magnetic storage in the hybrid hard disk as “disk”.

In recent years, new storage media that address some of the shortcomings of standard disks have emerged. Solid state memory or flash storage and MEMS-based storage are two examples. While flash storage has been a commercial success in consumer electronics, MEMS-based storage is still in the research pipeline. While standard disks still have the least cost per bit ratio, these new media can be coupled with standard disks to achieve an order of
magnitude improvement in response time, storage density, and power consumption. The hybrid hard disk is a result of such a coupling between flash and magnetic storage.

To enable a synergistic interaction between disparate technologies, efficient algorithms that exploit the benefits of each technology and mask their shortcomings are required. Algorithms to manage the newer media are actively being researched. Exploring algorithms for hybrid storage devices is difficult due to several reasons. First, many of the algorithms involved such as the FTL algorithm might be implemented in firmware. Further, since some of the algorithms are not revealed by the manufacturer, it is difficult to explore the complete design space and compare the performance of various algorithms. Finally, changing algorithms in different parts of the storage stack is a tedious and error prone task.

An accurate, extensible, and flexible simulator can aid in overcoming all these challenges.
Every part of the storage stack is exposed and easily modifiable in the simulator. It also provides common ground for comparing the performance of various algorithms. Most importantly it enables the exploration of novel combinations and configurations of storage media to aid develop the next generation of storage devices.

We support two different configurations of the hybrid hard disk:

- **Configuration 1:** Operating system manages the flash, i.e. the flash portion of the hybrid hard disk is exposed outside the disk. (See Figure 3.2)

- **Configuration 2:** The disk controller manages the flash; the hybrid hard disk appears as a standard hard disk to the operating system. (See Figure 3.3)

In the first configuration the operating system requires a driver that is specific to the hybrid hard disk. In this configuration, data can be selectively written to flash or directly to the
disk by the operating system depending on its semantics. For example, file metadata could be written to flash whereas file data could be directly written to the disk. Implementing such a scheme in the second configuration would be hard, if not impossible, since the controller would have no knowledge of files and file metadata. Another example is to store the data required during the booting process on flash. The operating system can then read this data from flash instead of waiting for the disk to spin up. This technique is used in Windows Vista [6].

The advantage of using the second configuration is that the disk can present an interface similar to that of a standard hard disk. The controller utilizes the flash to optimize performance transparent to the operating system enabling it to reuse the standard hard disk driver. If the performance provided by this configuration is comparable to that of the first configuration, storage vendors could provide the option of operating in the second configuration thereby significantly lowering the barrier to adoption of the hybrid hard disk.

These two configurations of the hybrid hard disk are incorporated in the Disksim storage system simulator. Disksim is designed based on the principle that the separation of components and their interconnections reduces the effort required to test and maintain components and to implement and integrate new components [35, 16]. We adopt these same design principles. The components we use to model a hybrid hard disk are loosely coupled and can be interconnected in several ways leading to a flexible and extensible design.

First, we design and implement a flash module that simulates the flash medium. This module can be coupled with the disk module to simulate a hybrid hard disk in the first configuration.
Further, we design and implement a hybrid controller that operates a simulated flash disk and a standard hard disk. The hybrid controller manages the flash medium and presents an interface similar to that of a standard hard disk, facilitating the simulation of the second configuration.

Section 3.1 discusses the design of the flash module and how it can be used to simulate flash disks. Section 3.2 illustrates the simulation of the first configuration of the hybrid hard disk. In Section 3.3 discusses the design of the hybrid hard disk controller. Section 3.4 discusses the simulation of the second configuration of the hybrid hard disk. Section 3.5 briefly discusses other configurations that can be simulated.

### 3.1 Modeling the Flash Medium

As discussed in Chapter 2, the minimum amount of data that can be read from or written to flash memory is known as a page. The minimum amount of data that can be erased on the flash medium is known as a block. A block consists of several pages. Due to the absence of a moving head, the flash medium provides a constant read and write access time. However, the effective read and write access time vary across blocks due to the property that a block must be erased before being reprogrammed. Depending on the FTL a read request for a page might require multiple pages to be read. Similarly, a write request directed to a page might translate to multiple block erases, page reads, and page writes.

A raw flash medium is modeled using a fixed read latency, which is the time taken to read
a page from the flash; a fixed write latency, which is the time taken to write a page to the
flash; and a fixed erase latency, which is the time taken to erase a block on the flash. One
of several FTL algorithms can be plugged onto the flash medium. Currently, the simulator
supports the page level FTL without wear leveling, log block based FTL, ANAND FTL,
and FMAX FTL. The flash module can also be used without any FTL, wherein the device
exposes a read/write/erase interface to the host system. This mode of operation can be used
by flash-aware file systems such as YAFFS [29] that are built directly over the flash medium.
In this case, the responsibility of erasing blocks before reprogramming them and providing
wear leveling rests with the file system software. The flash module in the simulator is capable
of being used both as a standalone device and in combination with other devices such as
standard disk drives The Disksim configuration used to model a stand alone flash device is
shown in Figure 3.4

![Disksim configuration to simulate a flash disk](image)

As flash memory replaces traditional storage technologies in several areas, newer and more
effective flash aware file systems and FTL algorithms are invented. To stay relevant and
useful with the newer FTL algorithms, the simulator provides an easy mechanism to swap out one FTL algorithm for another or add a new FTL algorithm. The FTL algorithm to be used is specified through a configuration file, providing an easy way to swap an FTL algorithm for another. We also provide a simple framework that enables one to implement and easily plug in a new FTL algorithm. Figure 3.5 shows an overview of the framework provided to add an algorithm to the simulator. During initialization, an FTL algorithm registers itself with the flash module. An FTL algorithm exports an interface consisting of the read_latency and write_latency methods, which are called when a read or write requests are received, respectively. An FTL algorithm manages its own data structures and is provided with a handle to the flash module data structures at the time of registration. Note that there is no erase_latency call. The FTL layer emulates a block device and provide a read/write interface to the host system. However, note that the FTL might erase one or more blocks and/or write one or more pages to satisfy a write request. Similarly, the FTL might read several pages to find the right page to be returned to a read request. This framework provides a flexible and simple method to add new FTL algorithms since the FTL designer can simply focus on the internals of the FTL algorithm and is insulated from the mechanics of the simulator.
3.2 Hybrid Hard Disk in Configuration 1

The flash module can be used in conjunction with the standard disk drive module in Disksim to derive a model that practically represents a model of a hybrid hard disk in the first configuration. Figure 3.6 presents the model of a hybrid hard disk operating in the first configuration, in which the flash and the disk is exposed to the operating system. In this mode of operation, the operating system offloads selected writes to the flash, manages the metadata to track the location of data, ensures that consistent data is returned on reads, and flushes the contents of the flash to disk when appropriate.

3.3 Modeling the Hybrid Hard Disk Controller

To simulate a hybrid hard disk in the second configuration, we designed and implemented a hybrid hard disk controller, referred to as the hybrid controller. The controller serves to manage the flash transparent to the operating system and presents an external interface
Figure 3.6: A Disksim model representing a hybrid hard disk in the first configuration similar to that of a standard hard drive. The hybrid controller selectively directs data to the flash or the disk, manages the metadata to track the location of data, ensures that consistent data is returned to read requests and eventually flushes the data on flash to disk. While the flash on the hybrid hard disk could be used as a read/write cache, we use the flash memory exclusively to offload write requests since a larger buffer cache in the operating system serves as an effective read cache.

For instance, consider the problem of selectively directing data to the flash. Offloading all writes to flash and later flushing them to disk could degrade performance in several cases. For example, while writing large sequential files to hybrid hard disk it is better to bypass the flash since disks currently provide a greater bandwidth to handle large sequential writes. Further, such an access pattern could quickly flood the flash creating an immediate flush that further delays subsequent requests in addition to evicting several useful pages from the flash. On the other hand, random writes to disk incur expensive seek times and temporally staggered writes might require the disk to spin up if the disk is spun down during idle periods.
Random writes and temporally staggered writes are handled better by flash. We use a simple scheme to detect large sequential writes. Write requests of size below a predefined threshold are treated as small writes and are directed to the flash. Whereas, writes above the threshold are directly written to disk. This scheme detects most of the large sequential writes since writing large amount of data to disk comprises several requests with the request size equal to the maximum size supported by the device. Thus by setting the threshold to be slightly less than the maximum request size supported by the device, most of the large sequential writes are captured.

In addition to selectively directing data to the flash, the hybrid controller must also maintain metadata to track the location of data in order to return consistent data to read requests. We ensure that the flash on the hybrid hard disk has the most recent data for the blocks that it holds. To satisfy a read request, say for sector $x$, the data from the flash is returned if sector $x$ is cached on the flash. If sector $x$ is not found on the flash, it is read from disk. A write to sector $x$ is written on flash, since it is a small write. When a large sequential write arrives at the hybrid hard disk, it is directed to the disk if none of the sectors are cached on flash, and to the flash otherwise. This scheme ensures that the flash always holds the most recent data and that the read requests are always presented with consistent data. Note that under this scheme, when a read request of the form $(b, n)$ where $b$ is the starting block number and $n$ is the number of blocks requested arrives at the hybrid controller, this request could be broken down into several child requests some of which are dispatched to the flash and the rest to the disk. Completion to the I/O driver is reported when all the child requests
complete. Algorithms to service reads and writes are discussed in detail in Section 4.2.

Since the size of flash is large, a system crash or a power outage could result in losing significant amount of data. If some of the data is file metadata the damage could be even worse. To guard against such losses, the metadata used to track the location of data must be made persistent. Suitable recovery mechanisms are also required to recover data from system crashes. While we model the parts required to maintain the consistency of data, we do not focus on failure recovery characterization of hybrid hard disks.

As writes are offloaded to flash, the free space on flash in the hybrid hard disk is eventually exhausted and data on flash needs to be flushed to disk to accommodate newer writes to flash. We implement an “on-demand” scheme wherein a cache cleanup is initiated when the flash is full and a write needs to be offloaded to the flash. In the current implementation, the cleanup operation flushes all dirty blocks on the flash to disk. A complete cache cleanup delays the next cleanup operation as much as possible. Since the disk might have to be spun up to perform a cleanup operation, this scheme enables the disk to remain in a low power state for a longer duration, thereby reducing the device’s power consumption. Further, the flushing efficiency is also increased by this scheme [3].

While we provide a policy to manage the flash on a hybrid hard disk, the hybrid controller is designed in a manner that allows a variety of schemes to be easily plugged in. For example, an alternate scheme is to flush just the required number of pages when there is insufficient space on flash to accommodate a write. Other dirty pages can be flushed to disk periodically when the disk is idle. This scheme eliminates the time consuming full flush operation. However,
on a workload with small bursts of I/O activity and large idle times, it results in the disk being spun up several times to write small amounts of data, leading to increased power consumption. Other schemes such as initiating a cleanup when the amount of free space in the flash drops below a low watermark could also be plugged in. When the flash is not flushed to the disk in its entirety, a page replacement algorithm is required to select victim pages during a cleanup operation. Our implementation provides an LRU scheme, which can easily be swapped out for another policy. Figure 3.7 illustrates the hybrid controller interface used to plug in a different flash management policy. A flash management policy registers itself with the hybrid controller using the register function. The update_cache function is called when a page is read from, written to, or added to the cache. The cache_cleanup function is called when a cleanup operation is initiated. The is_in_cache function returns 1 if a specified page is in the cache and 0 otherwise.

Figure 3.7: Interface to plug in a flash management policy into the hybrid controller
3.4 Hybrid Hard Disk in Configuration 2

The hybrid controller is the key component used to model a hybrid hard disk in the second configuration. Figure 3.8 presents the model of a hybrid hard disk operating in the second configuration. Everything below the hybrid controller appears as a single device to the I/O driver module. The hybrid controller receives requests from the I/O driver module and dispatches them to the flash module and the disk module. As discussed earlier, completion is reported to the I/O driver when all sub-requests are complete.

![Diagram of Hybrid Hard Disk Model](image)

Figure 3.8: A Disksim model representing a hybrid hard disk in the second configuration

3.5 Extending the Simulator for Other Configurations

While the simulator directly supports two configurations, several other combination can also be modeled with small code changes. For example, let us consider a configuration where the hybrid hard disk controller manages the flash within the disk and also allows the operating system to pin specific pages to the flash. This configuration can be simulated by extending
the second configuration with a request flag to indicate that a page must be pinned to
the flash. The flash management policy should also be modified to not evict pages that
are pinned to the flash. As discussed in this chapter, the framework-based design of the
simulator makes the implementation of such modifications easy.

In addition to these configurations, the hybrid hard disk can also be used in RAID configu-

rations. The flash module can be used to model solid state hard disks. Standard hard disks,
hybrid hard disks, and solid state disks could be combined to form heterogeneous storage
layers. Enabling the exploration of such a wide variety of storage technologies and their
interactions aids the design of the next generation of storage.
Chapter 4

Simulator Components

In this chapter, we discuss the flash and hybrid controller modules in further detail. The flash and the hybrid controller modules are implemented in C. These modules operate as part of the Disksim simulator. They run in user space and do not require any special system software or kernel modifications.

The hybrid hard disk is built using both the flash and disk modules in Disksim, using either a simple controller or a hybrid controller. All components are interconnected using the bus module. All the modules that we have implemented conform to the bus protocol used in Disksim [16], facilitating their interconnection with all other Disksim components.

The rest of this chapter is organized as follows. Section 4.1 discusses the parameters that characterize the flash medium and takes a closer look at the algorithms that simulate the various supported FTL algorithms. Section 4.2 presents, in detail, the algorithms that drive
Table 4.1: Parameters characterizing the flash medium

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block count</td>
<td>Number of blocks on the flash medium</td>
</tr>
<tr>
<td>Pages per block</td>
<td>Number of pages in each block</td>
</tr>
<tr>
<td>Page read latency</td>
<td>Constant delay to read a page</td>
</tr>
<tr>
<td>Page write latency</td>
<td>Constant delay to write a page</td>
</tr>
<tr>
<td>Block erase latency</td>
<td>Constant delay to erase a block</td>
</tr>
<tr>
<td>Read command overhead</td>
<td>Time elapsed between the instant when a read request is issued to the device and the instant when it is processed</td>
</tr>
<tr>
<td>Write command overhead</td>
<td>Time elapsed between the instant when a write request is issued to the device and the instant when it is processed</td>
</tr>
</tbody>
</table>

the hybrid controller.

4.1 The Flash Module

4.1.1 Flash Parameters

The flash module in Disksim is characterized by several parameters that are specified in the Disksim parameter file and fed to the simulator at the beginning of a simulation run. Some of the important parameters are listed in Table 4.1. A block is the minimum amount of data that can be erased by an erase operation on flash. A page is the minimum amount of data that can be read from or written to flash. Each block consists of a fixed number of pages. The block count and pages per block parameters together define the size of flash memory.
4.1.2 Flash Translation Layer Algorithms

In this section we take a closer look at the various FTL algorithms supported by the simulator. A request issued to the flash device is of the form \((p, n)\) where \(p\) is the starting page number and \(n\) is the number of pages to read/write. In the case of an erase request \(p\) refers to the block number.

The page level FTL without wear leveling serves as a basis for comparison with other schemes. The page level FTL algorithm for handling write requests is detailed in Figure 4.1. A page on flash is written directly if it is clean. If the page is dirty, the block containing the page is read into memory, the required modification is made in memory, and the block is erased and rewritten on flash. A few checks ensure that a block is not erased more than once for servicing a single request. Since a logical page is statically mapped to a physical page, this scheme does not provide wear leveling. Handling read requests is straightforward. A read request for logical page \(p\) is returned the physical \(p\).

The flash model supports a parameter known as the “Prewrite probability” that can be used with the page level FTL to begin the simulation with a “non-clean” flash. By a non-clean flash we imply that some of the pages in the flash contain valid data. This affects the write latencies since a write to a page that holds valid data must be preceded by an erase. Setting the Prewrite probability to 1.0 starts the simulation with a flash in which all the pages are assumed to be dirty. Thus, a write request would always read the entire block, make all the modifications, erase the block, and rewrite the block on flash. This behaves similar
Input: Write request of the form (p,n) where p is the starting page number and n is the number of pages

\[
\begin{align*}
    i &= p; \\
    \textbf{while } i < p + n \textbf{ do} \\
    &\quad \text{blkend} = \text{Page number of the last page in the block containing } p; \\
    &\quad \text{pageswritten} = \text{Min}((p + n - 1), \text{blkend}) - (i + 1); \\
    &\quad \textbf{if none of the pages from } i \text{ to } (i + \text{pageswritten} - 1) \text{ are written then} \\
    &\qquad \text{Write each page from } i \text{ to } (i + \text{pageswritten} - 1); \\
    &\quad \textbf{else} \\
    &\qquad \text{Read the whole block into memory;} \\
    &\qquad \text{Modify all required pages in the block;} \\
    &\qquad \text{Erase the block on flash;} \\
    &\qquad \text{Write the modified block back to flash;} \\
    &\quad \textbf{end} \\
    &\quad \text{Modify metadata to mark the dirtied pages as dirty;} \\
    &\quad i = i + \text{pageswritten}; \\
    &\textbf{end}
\end{align*}
\]

Figure 4.1: Algorithm to service a write request under page level FTL

to the algorithm in older Linux kernels used for certain flash media. Setting the Prewrite probability to a value \(0.0 \leq x \leq 1.0\), dirties approximately \(100x\%\) of the pages on the flash medium during initialization.

The log block FTL algorithm for handling write requests is detailed in Figure 4.2. This algorithm requires a certain number of blocks to be set aside as log blocks. Initially each logical block is assigned a data block and the remaining blocks are considered as free blocks. Note that if \(x\) blocks are set aside as log blocks, at any point in time the sum of the number of free blocks and the number of log blocks is \(x\). The log blocks are managed at the page level since the log block is written sequentially and the data blocks are managed at the block level, thus placing a manageable demand on memory. In essence, the log block FTL algorithm offloads writes to data blocks to a set of reserved blocks known as log blocks. When the log
The merge operation is key in understanding how the log block FTL performs wear leveling.

The merge algorithm is listed in Figure 4.3. One of two kinds of merge operations is possible; a switch merge or a full merge. A switch merge, which is quicker, simply makes the log block the data block and frees the data block. This kind of merge is possible only if the pages in the log block have been written sequentially from first page to last page. In all other cases a slower full merge is performed that writes all valid data to a new block and frees both the
log block and the data block. Since the full merge operation requires a free block, we must ensure that there is at least one free block available at all times. Observe that the merge also performs wear leveling by rotating the data blocks and the free blocks. Read requests are serviced by either reading pages from the assigned data block or the log block.

Let l be the log block to be merged and b be its associated data block;

\[
\text{if the logical page number of the } i^{th} \text{ page in block } l \text{ is } (\text{logical block number of block } b \times \text{pages per block} + i) \text{ for } i \in [0, \text{pages per block}) \text{ then}
\]

// Switch merge
Make block l the data block associated with the logical address of b;
Erase block b and add it to the list of free blocks;

else

// Full merge
Get a free block, say f;
Copy valid pages from block b and block l to block f;
Erase both block b and block l and add them to the list of free blocks;

end

Update the logical to physical mapping;

Figure 4.3: Merge operation under the log block FTL

Figure 4.4 illustrates the algorithm used to service a write request using the patented ANAND FTL. The ANAND FTL maps a virtual unit or logical address to several physical units that are chained together. The offset of a page within the logical block is identical to its offset in the physical block of the chain. A page \( p \) whose offset within the logical block is \( k \) is written to the first block in the chain whose \( k^{th} \) page is clean. If no such block exists, a block is added to the chain. When the chain length exceeds a threshold, it is “folded”.

A fold consists of copying the valid data from all the blocks in the chain to the last block in the chain, erasing all the blocks except the last block, and adding all erased blocks to the
**Input:** Write request of the form \((p, n)\) where \(p\) is the starting page number and \(n\) is the number of pages

**Figure 4.4:** Algorithm to service a write request under ANAND FTL

```
foreach page \(p'\) from \(p\) to \((p + n - 1)\) do
    \(v = \) Virtual unit corresponding to page \(p = p / \) pages per block;
    offset = \(v \mod (\) pages per block\);  
    if \(v\) does not have a chain assigned to it then
        if there are no free virtual units left then
            fold the longest chain;
        end
        Assign a free unit to \(v\), forming a chain of length 1;
    end
    Let \(v'\) be the first unit in the chain that has its \((\) offset\()^{th}\) page clean;
    if \(v'\) does not exist then
        if there are free units left and the chain is not too long then
            Add a free unit to the chain and write the current page at the \((\) offset\()^{th}\) page
            in the newly added free unit;
        else
            if the chain has only one unit then
                Read the unit into memory;
                Write \(p'\) on the \((\) offset\()^{th}\) page in memory;
                Erase the unit on flash;
                Rewrite the unit onto flash;
            else
                fold the chain and write \(p'\);
            end
        end
    else
        Write \(p'\) at the \((\) offset\()^{th}\) page in \(v'\);
    end
end
```

list of free blocks. The data is moved to the last block since the last block is guaranteed to have the most recent data in its dirty pages.

The algorithm to service a read request is illustrated in Figure 4.5. Briefly, to read page \(p\), the algorithm finds the last block in the corresponding chain that contains valid data for
Input: Read request of the form (p,n) where p is the starting page number and n is the number of pages

foreach page p’ from p to (p + n - 1) do
  v = Virtual unit corresponding to page p = p / pages per block;
  offset = v mod (pages per block);
  Locate the chain corresponding to the virtual unit v;
  if the chain is not initialized then
    // Read request for data we have not written
    Return empty data;
  end
  foreach block in the chain do
    if the offset\textsuperscript{th} page in the block is clean then
      The required page is the offset\textsuperscript{th} page of the previous block in the chain;
    end
  end
end

Figure 4.5: Algorithm to service a read request under ANAND FTL

In the ANAND FTL algorithm, folding a long chain requires several erases leading to a large response time for certain write requests. The patented FMAX FTL algorithm, which is an extension of the ANAND FTL algorithm remedies this situation. The algorithm to service write requests under the FMAX FTL is illustrated in Figure 4.6. The FMAX FTL limits the length of the chain to 2. The first unit in the chain is known as the primary unit in which the offset of a page in the logical block is preserved. The second unit of the chain, known as the replacement unit, does not have this property. A page is written to the replacement unit on any available physical page. By limiting the length of the chain to 2 units, the FMAX FTL limits the time taken to perform a fold operation.
**Input:** Write request of the form \((p, n)\) where \(p\) is the starting page number and \(n\) is the number of pages

```
foreach page \(p'\) from \(p\) to \((p + n - 1)\) do
    \(v = \) Virtual unit corresponding to page \(p = p / \) pages per block;
    \(\text{offset} = v \mod (\text{pages per block});\)
    if \(v\) has no chain assigned to it then
        if there are less than 2 free units then
            \text{fold} the first chain that has a replacement unit
        end
        Assign a free unit to \(v\), forming a chain of length 1 (primary unit);
    end
    if the \((\text{offset})\)th page in the primary unit is not dirty then
        Write \(p'\) on the \((\text{offset})\)th page of the primary unit of \(v\)'s chain;
    else
        if \(v\)'s chain has a replacement unit then
            Cycle through the pages from offset to \((\text{offset} - 1)\) looking for a clean page, say \(q\);
            if \(q\) exists then
                Write \(p'\) in \(q\);
            else
                \text{fold} the chain and write \(p'\);
            end
        else
            if there are less than 2 free units then
                Read the primary unit into memory;
                Write \(p'\) on the \((\text{offset})\)th page in memory;
                Erase the primary unit on flash;
                Rewrite the primary unit on flash;
            else
                Assign a new compount unit to \(v\)'s chain from the list of free units;
                Write \(p'\) to the \((\text{offset})\)th page on the newly assigned primary unit;
            end
        end
    end
end
```

Figure 4.6: Algorithm to service a write request under FMAX FTL

Figure 4.7 illustrates the algorithm to service a read request under FMAX FTL. In order to read a page, the replacement unit is first checked. If the page is not in the replacement unit,
it is returned from the primary unit. This ensures that the most recent data for a page is retrieved.

| Input: Read request of the form (p,n) where p is the starting page number and n is the number of pages |
| while all pages from p to (p + n -1) have not been read do |
|   Let v be the current virtual unit to be read; |
|   ptr = Pages to be read in the current virtual unit = Min(number of pages from the current page to the end of this block, number of pages from the current page to (p + n - 1)); |
|   if v does not have a replacement unit then |
|     Read ptr required pages from the primary unit; |
|   else |
|     // both primary and compound units exists |
|     foreach page to be read do |
|       if the page is not in the replacement unit then |
|         Read it from the primary unit; |
|       else |
|         Read it from the replacement unit; |
|       end |
|     end |
|   end |
| Advance the current page pointer to the beginning of the next virtual unit; |
| end |

Figure 4.7: Algorithm to service a read request under FMAX FTL

### 4.2 The Hybrid Controller

The hybrid controller interacts with the flash module and the disk module on one side and the I/O driver on the other side. The hybrid controller essentially consists of data structures to interact with the flash module and the disk module, a request queue to receive new I/O requests from the I/O driver, a flash queue and a disk queue to hold requests to be dispatched
to the flash module and the disk module, respectively, and data structures to interact with a pluggable flash management policy.

Input: Read request of the form \((p,n)\) where \(p\) is the starting page number and \(n\) is the number of pages

Add the request to the “request queue”;

if the controller is busy then
  return;
else
  Split the request into \(n\) individual blocks for blocks from \(p\) to \((p + n - 1)\);
  Initialize a flash list and a disk list;
  foreach of the \(n\) individual request do
    if the block is in cache then
      Add the request to the “flash list”;
    else
      Add the request to the “disk list”;
    end
    update_cache;
  end
  Merge sequential requests into a single request on the flash list and the disk list;
  Add the flash list to the “flash dispatch queue”;
  Add the disk list to the “disk dispatch queue”;
  dispatch_requests();
end

Figure 4.8: Algorithm to service a read request at the hybrid controller

An outline of the algorithm used by the hybrid controller to service read requests is presented in Figure 4.8. A request arriving from the I/O driver could be split into several sub-requests, some of which are directed to the flash and others to the disk. A completion for this request is reported to the I/O driver when all the sub-requests complete. The algorithm retrieves a block from disk only if it is not present in the flash. Note that for this algorithm to return consistent data, the algorithm to process write requests must ensure that the flash always contains most recent data for the blocks that is caches. The cache_update functions notifies
the cache of the read request. The updated data structures depend on the plugged in caching algorithm. The `dispatch_requests` function checks if the flash or the disk can accept more requests and dispatches requests to both devices. It also updates the metadata required to track the number of sub-requests dispatched.

```
Input: Write request of the form (p,n) where p is the starting page number and n is the number of pages
if n is too large and none of the pages from p to (p + n - 1) are on flash then
   // large sequential write
   Create a sub request (p,n) and add it to the disk dispatch queue;
else
   foreach page p’ from p to (p + n - 1) do
      if the number of free pages on flash is less that the threshold then
         cleanup_cache();
      end
      update_cache();
      Add a write request (p’,1) to the flash dispatch queue;
   end
   dispatch_requests();
end
```

Figure 4.9: Algorithm to service a write request at the hybrid controller

The algorithm to service a write request at the hybrid controller is shown in Figure 4.9. The algorithm detects a large sequential write as explained in Section 3.3 and adds it to the disk dispatch queue. If the request is a small write request or if some of the pages to be written are already cached on the flash, the write is directed to the flash. This ensures that the flash always contains the most recent data for the pages that it caches. A large write request in which some of the pages are already on flash is directed to the flash. The `cleanup_cache` and `update_cache` functions in the algorithm interact with the plugged in flash management algorithm.
The metadata maintained to manage the dispatch and completion of sub-requests is not shown in Figure 4.8 and Figure 4.9. This metadata is updated on receiving a notification from a storage device that a sub-request is complete. When all sub-requests corresponding to a parent request are processed, a completion interrupt is sent to the I/O driver.
Chapter 5

Evaluation: Validation and Analysis of Hybrid Hard Disks

In this chapter, we validate the accuracy of our simulator and analyze the performance of a few applications on a simulated hybrid hard disk. Section 5.1 discusses the experimental setup. Section 5.2 explains the process of extracting the parameters that characterize the flash medium. Section 5.3 presents the results to validate our flash and hybrid hard disk models in Disksim. Section 5.4 illustrates how the simulator can be used to compare the wear leveling characteristics of various FTL algorithms. Section 5.5 presents our analysis of the performance of a few applications on a hybrid hard disk. Finally, Section 5.6 presents the insights gained from our analysis of hybrid hard disks.
5.1 Experimental Setup

Our experimental setup consists of an Intel Core 2 Duo 2.4 GHz machine with 3 GB memory. The system contains 2 hard disks: (1) Seagate SATA disk - 7200 RPM, 250 GB, 8 MB on-disk RAM cache (2) Seagate SCSI disk - 15K.5 RPM, 73.4 GB, 16 MB on-disk RAM cache. The SCSI disk is connected to the system through an Adaptec ASC-29160N PCI Ultra 160 SCSI Controller.

We used blktrace [1], which is a block layer I/O tracing tool that provides detailed per I/O information up to user space, to collect I/O traces. Disk traces are obtained using the SCSI disk and flash traces are obtained using a Sandisk 1 GB USB flash stick. All traces are dumped on the SATA disk.

5.2 Parameter Estimation: Challenges and Solutions

Accurately validating the flash module is challenging for several reasons. First, the USB flash stick presents an interface similar to that of a standard hard drive, enabling it to work with a standard block device driver. The FTL algorithm that is responsible for providing a standard block device interface is implemented on a controller within the flash stick, making it hard to identify the FTL algorithm. Further, while some of the FTL algorithms are found in patents and research literature, others have remained trade secrets.

Depending on the FTL algorithm, a read could translate to several reads and a write could
translate to a combination of reads, writes, and erases. Since the FTL algorithm cannot be easily determined, it is hard to accurately determine the read, write, and erase latencies of the flash medium.

The third challenge pertains to modeling the state of flash memory. As a flash device is used, each block on the device broadly falls into one of three states: (1) it contains valid data (2) it does not contain valid data but needs to be erased before being rewritten or (3) it does not contain valid data and has been erased. Accurately capturing the observed behavior in the simulator would require that we capture the state of flash and preload the flash medium in the simulator with it before a simulation run. It is hard to determine the state of each block on the flash medium without support from the hardware.

The first challenge of determining the FTL algorithm was addressed by empirical comparison of the simulated and observed behavior of the flash medium for several micro benchmarks. The micro benchmarks involved reading and writing varying number of pages directly to the flash device, bypassing the buffer cache. We observed that the ANAND FTL best modeled the Sandisk flash device. ANAND FTL is an FTL algorithm patented by M-Systems Flash Disk Pioneers Ltd., which was later acquired by Sandisk Corporation.

To determine the read latency of the flash medium, the latency of the flash device to read varying number of pages was recorded. Since a read for a given page could result in one or more reads depending on the state of the flash, the experiment was repeated ten times and the minimum observed latency was chosen as this latency most probably directly resolved to the target page without superfluous reads. For a flash device operating under ANAND
FTL, the latency observed for a read request depends on the number of pages read and a constant command overhead that involves setting up registers in the device and for initiating and tearing down communication with the device. Thus the latency to read ‘n’ pages from the flash device is given by \( L = nRL + C \), where \( n \) is the number of pages written, \( RL \) is the time taken to read a single page from the flash, and \( C \) is the command overhead. \( RL \) and \( C \) are obtained as the slope and y-intercept of a linear trend line, which minimizes the square of the error, fitted to the points on a graph of block size \( (n) \) vs observed latency \( (L) \). The write latency of the flash medium, \( WL \), is determined in a similar manner. Once again, since a write could resolve to either a single write or to several writes and erases, the experiment was repeated ten times and the minimum write latency was chosen. The erase latency, \( EL \), is especially hard to estimate since the flash device emulates a block device interface, which does not export the erase operation. The erase latency is taken from published studies [24, 19].

To tackle the task of modeling a flash medium in which not all blocks are “clean”, let us first analyze the behavior of FTL algorithms. We consider a write to be sequential if it is directed to page \( n \) and the previous write was directed to page \( (n - 1) \). A write is a random write if it is not sequential. In general, FTL algorithms are optimized for sequential writes. For example, assume that block 10 is erased in order to write to its first page. The rest of the pages in block 10 can be written without incurring another erase. Thus for workloads with mostly sequential writes, the time taken to erase a block is amortized over several writes. However, in the case of workloads with random writes, the temporal separation
between erase operations could be much lesser when we begin with a flash where not all blocks are “clean”. In fact, this is the reason behind the poor random write performance of flash memory. To capture the behavior of a “non-clean” flash medium, we ran the Postmark I/O benchmark, which is comprised of 98% random write requests. Next we record the latencies of each individual request for a given block size. A write request for \( n \) blocks on a “non-clean” flash approximately translates to \( n \) writes and \( k \) erases. Thus, the observed latency \( L = n.WL + k.EL \), where \( n \) is the request size, \( WL \) is the write latency to write a single page, \( EL \) is the erase latency to erase a single block, and \( k \) is the mean number of erases incurred by the write request. This experiment is repeated for different block sizes, the mean value of \( k \) is obtained, and the correction is applied to the simulator.

5.3 Validation

We validate our flash module in Disksim against a 1 GB Sandisk USB flash stick. First, workloads are run on the USB flash stick and I/O traces are obtained using blktrace. The response time per I/O request is recorded. The response time of an I/O request is defined as the time elapsed between the issue of the request to the I/O driver and the completion of the request reported by the I/O driver. The trace is then converted to a Disksim friendly format, fed into the simulator, and the simulated response time is recorded. The observed and simulated response time curves are compared to gauge the accuracy of the flash module. The workloads used and their request size characteristics are shown in Table 5.1. Table 5.2
Table 5.1: Characteristics of I/O benchmarks used for validation

<table>
<thead>
<tr>
<th>Request size</th>
<th>Bonnie</th>
<th>Download</th>
<th>MP3</th>
<th>Untar</th>
<th>Postmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>0.22</td>
<td>0.39</td>
</tr>
<tr>
<td>16</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.03</td>
<td>0.24</td>
</tr>
<tr>
<td>32</td>
<td>0.03</td>
<td>0.02</td>
<td>0.01</td>
<td>0.04</td>
<td>0.20</td>
</tr>
<tr>
<td>64</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.07</td>
<td>0.10</td>
</tr>
<tr>
<td>128</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>0.11</td>
<td>0.04</td>
</tr>
<tr>
<td>256</td>
<td>0.95</td>
<td>0.97</td>
<td>0.95</td>
<td>0.53</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Table 5.2: Fraction of random write requests in each benchmark used for validation

<table>
<thead>
<tr>
<th>Randomness</th>
<th>Bonnie</th>
<th>Download</th>
<th>MP3</th>
<th>Untar</th>
<th>Postmark</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.061</td>
<td>0.064</td>
<td>0.105</td>
<td>0.669</td>
<td>0.98</td>
</tr>
</tbody>
</table>

displays the fraction of writes that were random in each workload. Bonnie is an I/O benchmark consisting of predominantly large requests with 6.1% of random writes. The Download workload involves downloading a 100 MB file from the Internet onto the flash stick. It is characterized by large requests with 6.4% random writes. The MP3 workload involves copying 209 MB of MP3 files from the SATA disk to the flash stick. It consists of large requests with 10.5% random writes. The Untar workload comprises untarring the Linux 2.6 kernel source tar ball. It involves a mix of both small and large request sizes with around 67% of random writes. Finally, the Postmark benchmark is an I/O benchmark from NetApp, consisting of a small requests with 98% random writes.

The observed and simulated response time curves for the Bonnie, Download, MP3, Untar, and Postmark workloads are shown in Figure 5.1, 5.2, 5.3, 5.4, and 5.5, respectively. The x-axis represents the response time in milliseconds and the y-axis represents the fraction of
Figure 5.1: Observed and simulated response time for the Bonnie workload

Figure 5.2: Observed and simulated response time for the Download workload
Figure 5.3: Observed and simulated response time for the MP3 workload

Figure 5.4: Observed and simulated response time for the Untar workload
Figure 5.5: Observed and simulated response time for the Postmark benchmark

Figure 5.6: Observed and simulated response time for a hybrid hard disk in Configuration 1
requests. A point \((x, y)\) on the curve indicates that \(100y\%\) of the requests had a response time less than \(x\) milliseconds.

We see that for the Bonnie, Download, and MP3 workloads the simulated response time closely matches the observed response time indicating that the flash module in the simulator accurately models the flash medium. The simulated response time for the Untar and Postmark workloads, which have a high degree of randomness, is also close to the observed response time, indicating that the ANAND FTL is an apt choice and that the correction discussed in Section 5.2 is also accurate.

Having validated the flash module, we now proceed to validate the hybrid hard disk model in Configuration 1. As discussed in Section 3.2, Configuration 1 augments the disk with flash memory. Since the disk, the standard controller, the I/O driver, and the bus modules are already validated, the validation of the hybrid hard disk in Configuration 1 logically follows.

To validate the hybrid hard disk empirically in Configuration 1, a 100 MB file was written to the flash stick, read from the flash stick, and then written to disk. This behavior mimics an attempt by the OS to temporarily offload data to the flash, and later flush it to disk. The observed and simulated response time curves, which match closely, are shown in Figure 5.6.

### 5.4 Wear Leveling Statistics

Since each block on the flash medium can be erased a limited number of times before it is rendered unusable, one of the chief functions of the FTL algorithm is to ensure that all
Figure 5.7: Wear leveling statistics for writing a 85 MB file over a 100 MB partition

Figure 5.8: Wear leveling statistics for writing a 85 MB file fifteen times over a 1 GB partition
blocks on the flash medium are worn out uniformly. This is achieved through a dynamic logical to physical block mapping.

Our simulator provides wear leveling statistics for an FTL algorithm by recording the number of times each block was erased during a simulation run. Figure 5.7 and 5.8 illustrate the wear leveling statistics for two simulation runs and various FTL algorithms. The x-axis represents the number of times blocks were erased and the y-axis represents the fraction of blocks erased. The graph in Figure 5.7 was generated by writing a 85 MB file over a 100 MB partition on the flash medium and feeding the resulting I/O trace to the simulator. Similarly, Figure 5.8 was generated by writing a 85 MB file fifteen times over a 1 GB flash medium. In both cases the simulation begin with the “clean” flash medium in which all blocks are erased and clean.

In Figure 5.7, we see that under ANAND FTL and FMAX FTL none of the blocks are erased since both algorithms use all unused blocks before reclaiming blocks. Blocks are erased 0, 1, or 2 times under the the page level FTL because the I/O requests do not fall on flash page boundaries resulting in the “read-erase-write” cycle occurring twice or thrice for certain blocks.

Figure 5.8 illustrates the wear leveling performance in the case where a small portion of the flash is repeatedly written. Since ANAND FTL and FMAX FTL utilize all blocks before reclaiming and erasing used blocks, once again we see that under these algorithms most blocks are erased once or twice respectively. Blocks are erased a greater number of times under FMAX FTL since the chain length is limited to 2. We see that log block FTL and
page level FTL, which show very similar wear leveling characteristics, are more reliant on the access pattern. Under these two algorithms, most blocks are not erased while few are erased several times.

5.5 Analysis of the Hybrid Hard Disks

In this section, we compare the performance of the hybrid hard disk with that of a standard hard disk for various workloads. The application traces are run through the SCSI disk and a hybrid hard disk in Configuration 2 that is modeled by augmenting the SCSI disk with a 256 MB flash medium, and their response time curves are compared.

The workloads used and their block size and random I/O characteristics are shown in Table 5.3 and 5.4. The Firefox workload, which involves mainly small requests and a few larger requests and a high degree of randomness, is an I/O trace of the Firefox web browser. The Dia workload is a trace of the Dia image manipulation program. Like the firefox workload, it involves mainly small requests and a high degree of randomness. The I/O trace for the MP3 workload, comprising mostly large sequential requests, was captured by copying several MP3 files. The Untar workload is an untar of the Linux 2.6 kernel source. It has a wide spread of request sizes leaning towards the larger end and a very high degree of randomness. The Postmark benchmark, which has the highest degree of randomness, mainly consists of small requests.

In Figure 5.9, we see that the hybrid hard disk (ANAND FTL) completes 50% of the requests
within 10 ms as opposed to 30% completed by the standard disk by offloading small random requests to the flash medium. However, a fraction of requests incurred a very high response time on the hybrid hard disk due to a burst of fold operations occurring in quick succession. The hybrid hard disk with FMAX FTL does better than the standard since FMAX caps the maximum time consumed by a fold operation.

The Dia workload is a workload that is light on I/O, chiefly consisting of small I/O write requests resulting from saving a file. Both the hybrid hard disk (ANNAD FTL) and the standard hard disk perform well, servicing 75% and 91% of the requests within 10 milliseconds respectively as shown in Figure 5.10. Once again, a few successive chain fold operations push the response time of a small fraction of requests towards the higher end of the scale. The hybrid hard disk (FMAX FTL) circumvents this problem and performs slightly better than the standard disk.
Figure 5.9: Response times for the standard and hybrid disk for the Firefox workload

Figure 5.10: Response times for the standard and hybrid disk for the Dia workload
Figure 5.11: Response times for the standard and hybrid disk for the MP3 workload

Figure 5.12: Response times for a standard and hybrid disk for the Untar workload
The hybrid hard disk (ANAND FTL) performs worse than the standard disk for the mostly sequential large-request MP3 workload as shown in Figure 5.11. While the hybrid hard disk (ANAND FTL) completes a larger fraction of small requests within 5 milliseconds, the larger requests complete quicker on the standard disk. We observed that while approximately 50% of write requests exceeded the threshold and were directly written to disk, the remaining large requests that fell below the threshold were offloaded to flash. It took longer to write these requests to flash than to directly write them to the disk, probably due to the sequential nature of these requests and the larger bandwidth provided by the standard disk compared with the flash stick. The hybrid hard disk (FMAX FTL) does slightly better than the hybrid hard disk (ANAND FTL), but is still outperformed by the standard disk.

For the Untar workload, as shown in Figure 5.12, both the hybrid hard disk (ANAND FTL)
and the standard disk completed approximately 50% of the requests. However, the small random writes offloaded to flash quickly flooded the flash, thereby initiating a costly flush operation that delayed the remaining requests. Hence, we see that a significant fraction of the requests have a higher response time. The hybrid hard disk (FMAX FTL) performs worse than the hybrid hard disk (ANAND FTL).

The hybrid hard disk (ANAND FTL) performs much better than the standard disk on the Postmark benchmark, which is a representative enterprise benchmark, since it takes lesser time to offload the small random writes to flash. We see that the hybrid hard disk (ANAND FTL) consistently services a greater fraction of requests in each time slot. The hybrid hard disk (FMAX FTL) further improves on its counterpart that uses ANAND FTL.

5.6 Discussion

By eliminating the large seek delay incurred by a moving head in standard disks, the hybrid hard disk offers superior performance for small random requests. While its performance lags standard disks for larger sequential requests, the gap could be bridged by improvements to flash technology. Although hybrid hard disks have primarily been considered for laptop computers as a means of reducing the power consumption of the storage component, it would be interesting to explore their relevance in enterprise environments given that the hybrid hard disk outperformed the standard disk for the Postmark benchmark, which is a representative enterprise benchmark.
The study in Section 5.5 emphasizes the importance of efficient algorithms to manage both the flash and interactions between the flash and the disk. For example, we see that when chain fold operations coincide with the arrival of a burst of I/O activity, several requests could be delayed. The FTL algorithm must aim to minimize such effects. In fact, the FMAX FTL algorithm, which is a close variant of the ANAND FTL, limits the length of the chain to two so that the time taken by the fold operation is reduced and bounded.

The policy used to migrate data between the flash memory and the disk when the hybrid hard disk is used in the second configuration also plays an important role in determining its performance. For example, performing a full flush of the flash in the midst of heavy I/O activity increases the response time of several requests. The flash management policy must aim to reduce such effects.

While we explored the strategy of using the flash memory as a temporary offloading area for writes, other strategies such as using the flash only to store metadata or for other specialized purposes could also be explored. Finally, improvements in flash technology can bolster the hybrid hard disk and help it outperform standard disks.
Chapter 6

Related Work

While prior research has focused on building efficient and accurate storage system simulators, to the best of our knowledge, our work is unique in designing and implementing an extensible, flexible, and accurate general purpose simulator for hybrid hard disks.

Pantheon [38] is a storage system simulator that aims to aid the rapid exploration of design choices in building a storage system. It supports various components such as disks, tapes, drivers, I/O schedulers, and array controllers. The simulator consists of independent components written in C++ that can be linked together in several ways. The Tcl language is used to instantiate components and specify interconnections. However, Pantheon does not model hybrid hard disks.

HRaid [9] is a flexible storage system simulator to model heterogeneous disks, storage hierarchies, and resource sharing. It is mainly targeted at aiding the design of storage systems for
cluster of workstations. The simulator is partly validated against real workloads and partly by comparing the results with the simulator developed by Kotz et.al. [28]

Flash backed I/O requests [4] is an I/O scheduling algorithm for hybrid hard disks that uses the on-disk flash memory to reduce the latency of write requests. Their algorithm focuses on random writes and selectively offloads writes by considering the efficiency of both media to service the I/O request. The authors simulate a hybrid hard disk using access latencies from the Hitachi EK1000 2.5in drive and a Sandisk Ultra II Compact Flash to analyze the performance of their algorithm. However, their simulator is not a flexible general purpose simulator and is not validated against real workloads.

Flushing policies for the non-volatile cache were proposed, addressing the questions of when and how the contents of the flash must be flushed to the disk in order to minimize the data synchronization overhead [3]. They conclude that flushing performance improves significantly by exploiting traditional I/O optimization techniques such as merging and reordering and by flushing the flash only when it is full. The same hybrid hard disk simulator used in [4] was used to validate their results.

An intelligent data pinning policy and a flash aware cache were proposed to improve performance and reduce power consumption of hybrid hard disks compared to standard hard disks [26]. SimHybrid, a hybrid hard disk simulator, was built to analyze the proposed techniques. However, it is not as flexible as our simulator. It neither supports various configurations nor provides an extensible framework. The accuracy of the simulator is not validated.
Several approaches [18, 21, 22, 25, 27, 30] have focused on using the beneficial aspects of flash memory in combination with standard disks to improve performance and/or power consumption. Novel algorithms [3, 4], to manage hybrid hard disks were also proposed. An accurate and extensible simulator can benefit such research by obviating the authors’ need to design and implement their own simulator to analyze the proposed algorithms.
Chapter 7

Conclusion

Current storage systems face two severe challenges. First, as the CPU speed increases at a much faster rate than disk speed, the disk becomes the bottleneck for several applications. Second, disks consume a significant fraction of the overall system power [32], increasing cost of running enterprise systems. On the positive side, new storage technologies such as flash memory and MEMS-based storage have emerged, which promise orders of magnitude greater storage density and faster access times while requiring only a fraction of the power consumed by standard disks. However, standard hard disks are more mature, better understood, and have a better cost per bit ratio than these emerging technologies. While flash memory or MEMS-based storage might not be set to replace standard hard disks, they could be combined with standard hard disks to significantly improve the performance and power consumption of hard disks. The hybrid hard disk is a result of such a coupling, augmenting the standard hard disk with flash memory.
The hybrid hard disk claims improved performance and reduced power consumption. The effectiveness of the hybrid hard disk depends on the algorithms used to manage the disk. Designing such algorithms is a challenging task. To ease the exploration of the design space, to allow easy modification of every part of the storage stack, and to provide common ground for evaluating various algorithms, we designed an extensible and flexible hybrid hard disk simulator.

We presented the design of the hybrid hard disk simulator and illustrated two different configurations of the hybrid hard disk that the simulator models. In the first configuration the flash memory is managed by the operating system and in the second, by the disk controller. We discussed the design of the flash module and how it can be used to simulate both a solid state disk and a hybrid hard disk in the first configuration. Next, we discussed the design of the hybrid controller and how it can be used to model a hybrid hard disk in the second configuration. We demonstrated our framework-based design and the interface used to plug in FTL algorithm and flash management algorithms. Finally, we discussed other configurations that are not directly supported but can be easily incorporated in the simulator through small code changes.

We explained the process of extracting the parameters required by the simulator. We validated our flash and hybrid hard disk (first configuration) models. Finally, we compared the performance of the hybrid hard disk with that of a standard disk for several workloads.

We observed that the hybrid hard disk offers superior performance for small random requests. While its performance currently lags that of standard disks for larger sequential requests,
the gap could be bridged by improvements to flash technology. Although hybrid hard disks have primarily been considered for laptop computers as a means of reducing the power consumption of the storage component, it would be interesting to explore their relevance in enterprise environments given that the hybrid hard disk outperformed the standard disk for the Postmark benchmark, which is a representative enterprise benchmark.
Chapter 8

Future Work

In this thesis, we focused on modeling hybrid hard disks to understand their performance characteristics by designing and implementing an accurate and extensible simulator. Our simulator can be enhanced and extended in a few ways.

Since hybrid hard disks claim to consume lesser power than a standard hard disk when managed efficiently, it would be useful to incorporate a model to track and estimate the power consumption of a hybrid hard disk during a simulation run. This would provide another dimension for the comparison of various algorithms. Algorithms that achieve a balance between performance and power consumption could make the hybrid hard disk useful in notebook computers where battery life is important.

Currently, we extracted the parameters for the flash medium after thorough empirical analysis. A tool to automatically extract parameters from a flash medium would greatly simplify
the process of plugging in a new flash medium into the simulator.

We approximately modeled a non-clean flash medium as discussed in Section 5.2. Devising a technique to accurately capture the state of the flash medium and then reconstructing that state within the simulator before a simulation run would enhance the accuracy of the simulator.

Finally, the simulator could be extended to support more configurations. For example, the operating system could delegate the management of the hybrid hard disk to the hybrid controller while retaining limited control by pinning certain pages to the flash. Some of these configurations were discussed in Section 3.5.
Bibliography


[38] John Wilkes. The Pantheon storage-system simulator, May 1996.