THE CASE FOR A HARDWARE FILESYSTEM

by

Ashwin Mendon

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Electrical Engineering

Charlotte

2012

Approved by:

Dr. Ron Sass

Dr. James M. Conrad

Dr. Bharat Joshi

Dr. Zongwu Cai

ABSTRACT

ASHWIN MENDON. The case for a hardware filesystem.
(Under the direction of DR. RON SASS)

As secondary storage devices get faster with flash based solid state drives (SSDs) and emerging technologies like phase change memories (PCM), overheads in system software like operating system (OS) and filesystem become prominent and may limit the potential performance improvements. Moreover, with rapidly increasing on-chip core count, monolithic operating systems will face scalability issues on these many-core chips. Future operating systems are likely to have a distributed nature, with a separation of operating system services amongst cores. Also, general purpose processors are known to be both performance and power inefficient while executing operating system code. In the domain of High Performance Computing with FPGAs too, relying on the OS for file I/O transactions using slow embedded processors, hinders performance. Migrating the filesystem into a dedicated hardware core, has the potential of improving the performance of data-intensive applications by bypassing the OS stack to provide higher bandwidth and reduced latency while accessing disks.

To test the feasibility of this idea, an FPGA-based Hardware Filesystem (HWFS) was designed with five basic operations (open, read, write, delete and seek). Furthermore, multi-disk and RAID-0 (striping) support has been implemented as an option in the filesystem. In order to reduce design complexity and facilitate easier testing of the HWFS, a RAM disk was used initially. The filesystem core has been integrated and tested with a hardware application core (BLAST) as well as a multi-node FPGA network to provide remote-disk access. Finally, a Serial ATA IP core was developed and directly integrated with HWFS to test with SSDs. For evaluation, HWFS's performance was compared to an Ext2 filesystem, both on an FPGA-based soft processor as well as a modern AMD Opteron Linux server with sequential and random workloads. Results prove that the Hardware Filesystem and supporting infrastructure

provide substantial performance improvement over software only systems. The system is also resource efficient consuming less than 3% of logic and 5% of the Block RAMs of a Xilinx Virtex-6 chip.

ACKNOWLEDGMENTS

Several individuals have played an important part in the successful completion of this dissertation.

First and foremost, I would extend my gratitude to my adviser Dr. Ron Sass for his guidance, encouragement and help throughout the course of my PhD at UNC Charlotte.

My peers at the Reconfigurable Computing Systems Lab for supporting me. A special thanks to Andy, Sid and Bin with whom I have had the good fortune to work on several projects.

I have been blessed to have wonderful parents who have always stood by me and I hope that I have made them proud !

Shweta for her devotion, love and patience to whom I dedicate this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AIREN	Architecture Independent Reconfigurable Network
ALL	AIREN Data Link Layer
ATA	Advanced Technology Attachment
BLAST	Basic Local Alignment Search Tool
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DDR	Double Data Rate
DIMM	Dual in-line Memory Module
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
EOF	End of Frame
EMI	Electromagnetic Interference
FIS	Frame Information Structure
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FTL	Flash Translation Layer
HPC	High Performance Computing
HWFS	Hardware Filesystem
IP	Intellectual Propoerty
LBA	Logical Block Address
LFSR	Linear Feedback Shift Register
MGT	Multi Gigabit Transceiver
MPI	Message Passing Interface
NPI	Native Port Interface
OOB	Out of Band Signalling Controller

OS	Operating System
PCIE	Peripheral Component Interconnect Express
PCM	Phase Change Memory
PLL	Phase Locked Loop
RAID	Redundant Array of Independent Disks
SATA	Serial Advanced Technology Attachment
SOF	Start of Frame
SRAM	Static Random Access Memory
SSD	Solid State Drive
STTM	Spin Transfer Torque Memory
UFS	Unix Filesystem

CHAPTER 1: INTRODUCTION

Non-volatile Winchester style storage devices have been painfully slow in providing adequate I/O performance for High Performance Computing applications. As a result system designers have gone to great lengths to try to mitigate this poor performance. Processors are built with large caches to hold frequently used data. To preserve large working sets in fast main memory, most supercomputers today are built with huge amount of DRAM. This is both expensive and energy consuming. Operating systems have complex schedulers and I/O intensive applications perform sophisticated buffer management and optimizations to minimize I/O or overlap it with computation.

Flash-based non-volatile solid state disks (SSDs) have shown great promise in reducing the gap in computation and I/O performance [1, 2, 3]. These flash devices having access times in microseconds, offer reduced latency and increased bandwidth. Other emerging technologies such as Phase Change Memory (PCM) [4] and Spin-transfer torque memory (STTM) [5], broadly classified as storage class memory [6, 7], are even faster with speeds close to that of main memory technologies (nanoseconds). To leverage the full potential of these technologies, would require overcoming the legacy of disk based systems which have been designed assuming storage is slow.

Research conducted at the Non-Volatile Systems Lab in University of California San Diego [8], indicates that for solid state devices, overheads in the operating system stack exceed the hardware access time (It takes 20000 instructions to issue and complete a 4 KB IO request under Linux). Many of the software optimizations which are beneficial for mechanical disks are unprofitable for SSDs. The operating system's I/O request schedulers which aim to reduce the impact of rotational and seek delays for hard drives, only add software overhead for SSDs which have no moving parts. In

addition, the system call/return interface and data copies to/from user space to kernel space add substantial overhead. The filesystem also adds about 4 to 5 μs of latency to each I/O request [8]. Many filesystems employ prefetching to hide the long latency of I/O operations on disks. On SSDs, there is much less latency to hide. Hence, many workloads that may have benefitted from prefetching on disks, would not see similar gains on SSDs. Thus, with faster secondary storage devices, bottlenecks arise in the system software during I/O transactions. To derive optimum performance out of these non-volatile memory technologies will require significant changes to operating systems and system architecture.

In an era of abundant transistors and the end of frequency scaling, computer hardware is changing rapidly with increasing number of cores per chip [9, 10]. Operating systems have however remained monolithic, with a centralized kernel, posing question marks over their scalability across 100's and 1000's of cores. They have mainly relied on efficient cache coherence for communication of data structures and locks. However, the increasing core count and the complexity of interconnecting them has indicated that hardware cache coherency protocols would be expensive to scale and future many core chips are unlikely to have hardware support for cache coherence. Instead, they may have message passing buffers in hardware[10, 11]. This has spawned a resurgence in distributed operating systems research [12, 13, 14] with micro kernels communicating explicitly through messages. They make a case for restructuring the operating system such that some of its services can be run on dedicated cores. This would avoid contention for resources such as such as caches and TLBs between the working sets of the OS and application, leading to better utilization [15]. Moreover, several aggressive microarchitectural features of modern processors such as deep pipelines and speculation meant for improving application performance have not shown similar benefits for operating system execution [16]. OS codes have frequent branches which affects branch prediction logic performance and causes pipeline flushing. (Nellans et

al. have conducted an experiment by running an OS on a 3 GHz Pentium 4 with 31 stage pipeline and a 33 MHz 486 processor with a 5 stage pipeline and found that that they are close in performance for OS code !) Moreover, these complex processing cores prove to be energy inefficient while running operating system code [17]. Thus, there is a strong case for exploring the idea of having a specialized, dedicated core for certain operating system functions which have a significant contribution to the total execution time.

In the domain of High Performance computing with accelerators too, relying on the operating system for I/O transactions hinders performance. Modern High Performance Computing Systems frequently use Field Programmable Gate Arrrays (FPGAs) as compute accelerators. As Integrated Circuit technology advances, the programmable logic resources on FPGA devices continue to grow as well. Many of these devices, in addition to programmble logic resources and flip-flops, are also rich in special purpose blocks such as processors, Block RAMs, DSP cores and high speed-serial transceivers. This allows for greater integration of computing systems with Systems-on-Chip designs running a mainline Linux kernel. In fact several high-performance computing researchers are currently investigating the feasibility of using Platform FPGAs as the basic compute node in parallel computing machines [18, 19, 20]. If successful, there is an enormous potential for reducing the size, weight, and cost while increasing the scalability of parallel machines. The Multi-Gigabit Transceiver (MGT) cores on these devices are especially interesting because they allow for a wider range of high-speed serial peripherals, such as disk drives, to be directly connected to the devices. This has the potential to speed up data-intensive applications. However, the embedded processor cores on these FPGAs are clocked at relatively slow frequencies (100 MHz) as compared to modern high end processors (3 GHz). Data intensive application accelerator cores running on programmable logic resources of the FPGA have to go through the traditional layers of an operating system (system call, file sys-

tem, device driver and interrupt handling layer) for accessing storage devices using these slow sequential processors. This would result in substantial software overhead for file I/O transactions particularly for faster solid state storage devices. Results with running filesystem benchmarks such as Bonnie++ on faster Intel Pentium 4 processor also show that for I/O intensive applications, the Operating System overhead overshadows its userspace components (contributes about 86% of instructions for file I/O) [16].

Thus, with fast disks, cheap transistors, abundant cores, operating system overhead for filesystem operations and its inefficient use of a general purpose processor's complex architecture, there is a strong case for migrating filesystem operations to a dedicated hardware core. The central question that we propose to answer in this thesis is: *As secondary storage devices get faster and the number of on chip cores increase rapidly, will migrating the filesystem service from the operating system into a dedicated hardware core, responsible for managing all accesses to disks, provide I/O performance improvement for HPC applications ?*

The main purpose of a computing system is to create, manipulate, store, and retrieve data. As such, filesystems have been central to most modern computing systems. Filesystems are typically implemented in *software* as part of the operating system. To eliminate the overhead of traversing the operating system stack for file I/O operations, we have built infrastucture in the form of a hardware filesystem (HWFS) interfaced with an on-chip Serial ATA host controller core (also developed as part of this project). The simplest filesystems organize the sequential fixed-size disk sectors into a collection of variable-sized *files*. Of course, most modern filesystems are much more complex and also include a large amount of meta-information and further organize files into a hierarchy of directories. The design presented here, however, is narrowly defined to support high-performance computing. This is not a particularly serious weakness since SRAM-based FPGA devices can be reprogrammed to incor-

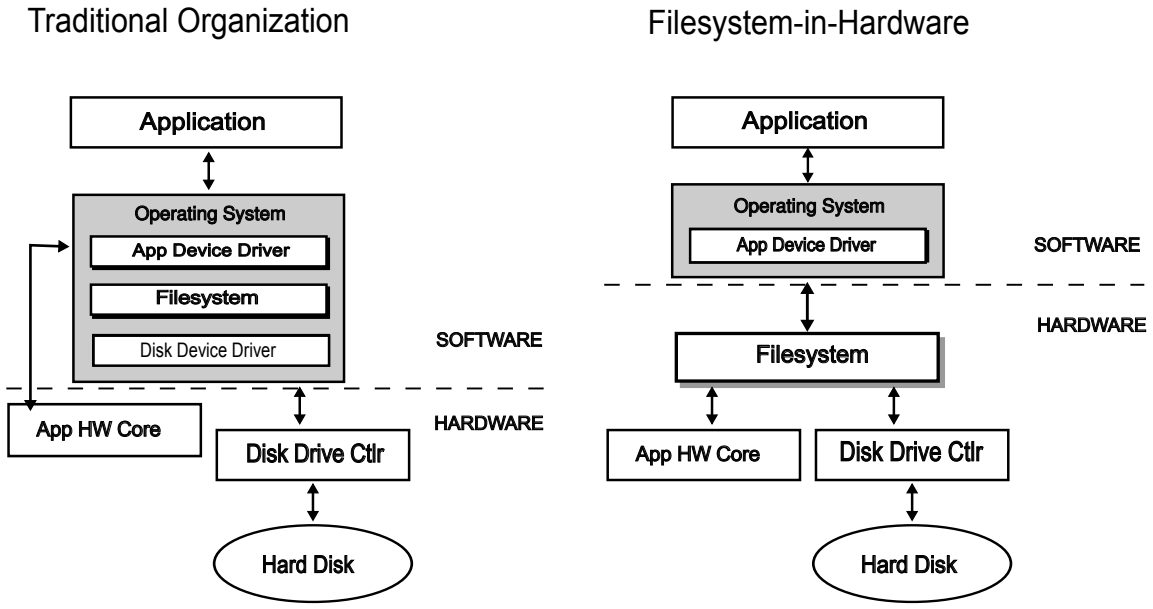


Figure 1.1: (a) traditional filesystem (b) filesystem migrated into hardware

porate new features in hardware. Figure 1.1 illustrates the concept of a hardware filesystem. On the left is the traditional organization with the filesystem and device driver implemented in software. Here the operating system plays the central role in coordinating an application's and hardware core's disk access. On the right is the filesystem migrated into hardware which enables more direct access between hardware cores and secondary storage.

Such an architecture is important to high-performance computing applications for a number of reasons. It allows FPGA computational cores to consume data directly from disk without interrupting the processor (or traversing the operating system's internal interfaces). This frees the processor from handling I/O requests and avoids the use of off-chip memory bandwidth to buffer disk data. This is particularly useful for streaming applications [21, 22] which have little temporal locality. It also reduces the number of interrupts which has been shown to negatively impact very large parallel systems [23, 24]. It allows the introduction of simple striped on-chip multi-disk controllers (without the cost or size of peripheral chipsets). Finally, it is possible to coordinate disks attached to multiple discrete FPGA devices — again,

without depending on the processor. (Again lowering the number of interrupts the processor sees and avoids wasting memory bandwidth to buffer data between disk and network subsystems.)

In short, this approach has the potential of increasing the bandwidth from disk to core, lowering the latency, and reducing the computational load on the processor for a large number of FPGA devices configured for high-performance computing. In the case of multi-core chips too, a dedicated filesystem core (managing its own filesystem metadata buffers) could communicate with applications cores using a low latency on-chip network and hardware message passing buffers.

Based on this approach, the metrics that we need to investigate in this thesis are as follows:

- Bandwidth *If the filesystem component is migrated into hardware, will this give performance improvements over a software fileystem for solid state drives?*
- Resource Utilization *Can the improvements in performance justify the cost of extra on-chip resources dedicated to the Hardware Filesystem and SATA disk controller cores?*
- Design Scalability *Can the filesystem core support multiple disks? Will the performance of the core scale with multiple disks?*

An ancillary benefit of this work is its potential use in checkpointing the system state for HPC applications. In traditional checkpointing, either the application or in some cases a library periodically stops the program and writes the application's critical data structures to non-volatile storage. The CPU which is running the program has to spend time in the checkpointing process using a traditional software filesystem. This process could be offloaded to special FPGA-based hardware cores using the hardware filesystem in the background without needing to stop the application

CPU. In case of a CPU crash, the FPGA based filesystem could still recover the data from the disk.

The rest of this dissertation is organized as follows: Chapter 2 provides background material to familiarize the reader with the proposed work. A section of related work in academia and industry is included here. The design goals and implementation of the Hardware Filesystem and the SATA Host controller core are presented in Chapter 3. The results and the evaluation of the thesis metrics are covered in Chapter 4. Chapter 5 concludes, summarizing the research.

CHAPTER 2: BACKGROUND

This chapter briefly covers the structure of the Unix Filesystem, the Serial ATA disk controller protocol and Solid State Drives. We also include a section on related work.

2.1 Disk Subsystem

FILESYSTEMS Filesystems are responsible for managing and organizing files on a nonvolatile storage medium. Files are composed of bytes and the filesystem is responsible for implementing byte-addressable files on block-addressable physical media, such as disk drives. Key functions of a filesystem are: (1) efficiently use the space available on the disk, (2) efficient run-time performance, and (3) perform basic file operations like create file, read, write and delete. Of course most filesystems also provide many more advanced features such as file editing, renaming, user access permission, and encryption to name a few.

The hardware filesystem implemented is loosely modeled after the well-known UNIX filesystem (UFS) [25]. The disk layout of the filesystem is shown in Figure 2.1

super block: describes state of the filesystem such as blocksize, filesystem size, number of files stored and free space information

inode list: list of pointers to data blocks

data blocks: contain actual file data

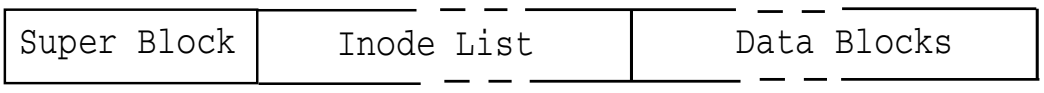


Figure 2.1: Filesystem layout

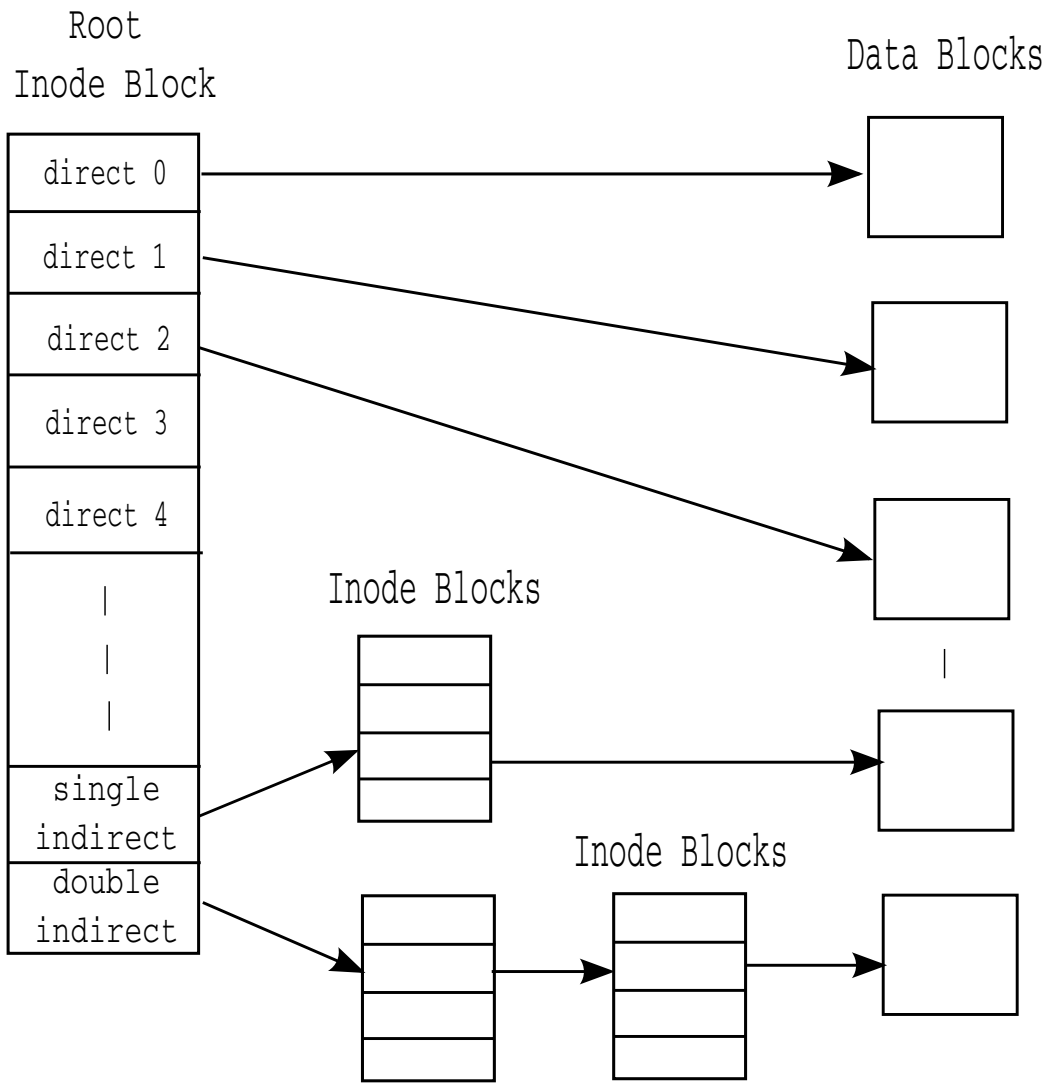


Figure 2.2: UNIX Inode structure

UFS uses logical blocks of 512 bytes (or larger multiples of 512). Each logical block may consist of multiple disk sectors. Logical blocks are organized into a filesystem using an Inode structure that includes file information (such as file length), a small set of direct pointers to data blocks, and a set of indirect pointers. The indirect pointers point to logical blocks that consist entirely of pointers. UFS uses a muti-level indexed block allocation scheme which includes a collection of direct, single indirect, double indirect, and triple indirect pointers in the Inode. The filesystem layout is as shown in the Figure 2.2.

SERIAL ATA Normally, the filesystem is designed to be independent of the disk controller. From the operating system's perspective, the disk controller is typically a device driver that is responsible for communicating with the physical media and responds to block transfer commands from the filesystem. On most systems the host side disk controller is an ASIC connected to the I/O bus of a processor. It communicates with the disk side controller using an ATA protocol. For expediency, the work here focuses on the most common, commodity drives available today: Serial ATA (SATA). SATA Gen 1 has a transmission speed of 150 MB/s, which was increased to 300 MB/s in SATA Gen 2. (The SATA Gen 3 specification is available and supports speeds upto 600 MB/s. However, we do not have access to FPGA devices compatible with SATA Gen 3 yet.) Serial ATA was designed to overcome a number of limitations of parallel ATA. The improvements include a four-wire point-to-point high-speed serial interface that supports one device per controller connection. Each device gets a dedicated bandwidth and there is no master/slave configuration jumper issues as with parallel ATA drives. The pincount is reduced from 80 pins to seven pins. Three ground lines are interspersed between four data lines to prevent crosstalk. Also, smaller cables result in less clutter, better routing, and improved airflow.

The SATA protocol stack is divided into five layers as shown in Figure 2.3. The application layer consists of the programming interface to the SATA HBA. Traditionally, a driver in the host processor builds the SATA specific data structures in main memory. The command layer defines the sequence of Frame Information Structures (FISs) exchanged between the host and device while executing an ATA command. The transport layer formats and decomposes FISs, manages flow control and retransmits buffered FISs in case of error. The link layer is responsible for sending and receiving frames, decoding primitives, handling transmission errors etc.

Several FPGA devices include high-speed serial transceivers. For example, the Xilinx Virtex-4, Virtex-5 and Virtex-6 device families have members that include

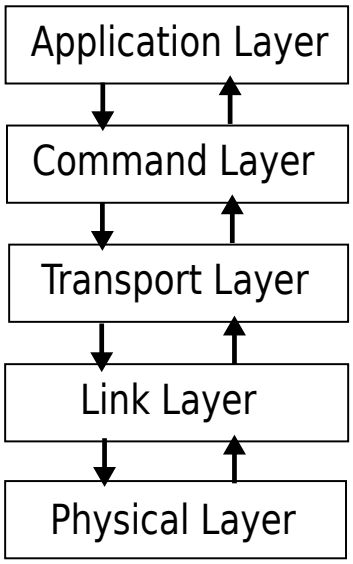


Figure 2.3: SATA protocol layers

multi-gigabit transceiver cores. These cores can be configured to communicate via the SATA protocol at the physical layer. There are commercial IP cores available to do this. However, we have implemented our own SATA Host Bus Adapter which includes command layer, transport and link layers in the FPGA fabric. The physical layer provides a high-speed electrical interface between host and device as well as the logic to encode/decode, serialize/deserialize the data. This capability is provided by the FPGA transceivers. The link initialization sequence for this layer is implemented in FPGA logic. Details of this design and can be found in Chapter 3

Solid State Drives Flash-based solid state drives have risen to prominence over the past few years as high performance data storage devices. The key components in an SSD are the memory to store data and the controller to manage it. The controller is typically an embedded processor which provides a bridge to the host computer and performs functions such as *wear leveling* and *garbage collection*. For this purpose it uses Flash Translation Layer (FTL) algorithms to deal with flash memory's idiosyncrasies.

The basic unit of read/write operations in a solid state drive is a 4KB page.

However, unlike traditional hard drives, the data on SSD needs to be erased in units of 128KB to 512KB (depending on manufacturer) before they can be rewritten to. This adds a penalty to the write operation in SSDs. There is also a limit on the number of write/erase cycles during the lifetime of a flash device. To mitigate these impediments to the performance and reliability of SSDs, the SSD controller use a Flash Translation Layer which provides indirection between the logical block addresses from the host controller and physical location of data on the flash device. This process spreads out the writes on the device providing wear leveling to increase the life time of the device. Also, before a block can be erased, the valid data must be relocated by a process called garbage collection. Modern SSD contollers uses efficient mechanisms for this process by employing background garbage by using idle time or in parallel with the host writes.

2.2 Related Work

The implications of fast, cheap non-volatile memories on the structure of operating systems has been examined in [26, 27]. This supports our argument that faster storage devices will necessitate re-examining the way storage is accessed by I/O systems and improvements in I/O subsystem architecture.

The hardware filesystem architecture described in this paper is, to the authors' knowledge, novel and unique. However, there are several research efforts pursuing related goals. These efforts are described below.

Work at the University of California, Berkeley describes BORPH's kernel filesystem layer [28] which enables hardware cores to access disk files by plugging into the software interface of the operating system via a hardware system call interface. However, the cores still have to traverse the software stack of the OS. The approach proposed in our work allows the hardware cores direct access to disk by implementing the filesystem directly in hardware.

The Reconfigurable parallel disk system implemented in the RDisk project [29]

provides data filtering near disks for bioinformatics databases by using a Xilinx Spartan 2 FPGA board. While this is relevant for scan algorithms which read in large datasets, it does not provide the capabilities of a filesystem such as writing and deleting files. A few other research groups too are using FPGAs with storage devices for investigating active disks approaches [30, 31].

Using FPGAs to mitigate the I/O bandwidth bottleneck has been of interest commercially among server vendors such as Netezza (now IBM) [32] and Bluearc (now Hitachi Data Systems) [33]. Netezza database storage servers have a tight integration of storage and processing for SQL-type applications by having FPGAs chips in parallel Snippet Processing Units (SPUs). These provide initial query filtering to reduce the I/O and network traffic in the system. Bluearc's Titan 3000 network storage server uses a hardware accelerated filesystem to speed up the I/O interface.

Well known RAID storage solutions have either hardware or software controller managing data across multiple disks. However, these solutions operate on a single I/O channel or bus [34] and still traverse the operating system's software stack. While this can be used to improve disk performance, it does not necessarily improve disk to compute accelerator performance. Moreover, the approach proposed here has the ability to be directly integrated into the network subsystem of a parallel machine — allowing multiple I/O channels in a parallel filesystem implementation.

CHAPTER 3: DESIGN

To mitigate risk, the design and implementation of the Hardware Filesystem (HWFS) was staged. The first stage focused on a software reference design and RTL simulations to judge the feasibility. This was reported in [35, 36]. The work here describes a design that synthesizes and runs on an FPGA. It was interfaced and tested first with a RAM Disk emulation system. Support for multiple disks in a RAID0 configuration has been provided. The HWFS has also been tested with a hardware application core across a multi-node FPGA cluster to provide remote disk access. Finally, a Serial ATA Host Bus Adapter core has been developed and directly interfaced with HWFS to measure performance with Solid State Drives.

3.1 Hardware Filesystem Core

As mentioned in the previous section, the layout of the UNIX filesystem was the initial starting point for the HWFS described here. However, UFS was designed to be general-purpose whereas the aim of this work is more narrowly focused on feeding streams of data to compute accelerators. This has led to a number of differences. First, the Hardware Filesystem uses only direct and single indirect pointers in its inodes Essentially, after the initial pointers in the inode are exhausted, the system reverts to a linked-list structure for very large files. This layout is shown in Figure 3.1. A second difference in the Hardware Filesystem is that the file names are merged into the *Super Block* along with filesystem metadata such as freelist head and freelist index. The UFS supports a hierarchy of directories and sub-directories but the HWFS described here is flat.

A high-level block diagram of the HWFS is shown in Figure 3.2. It consists of a single large state machine as the control unit and a datapath of metadata buffers.

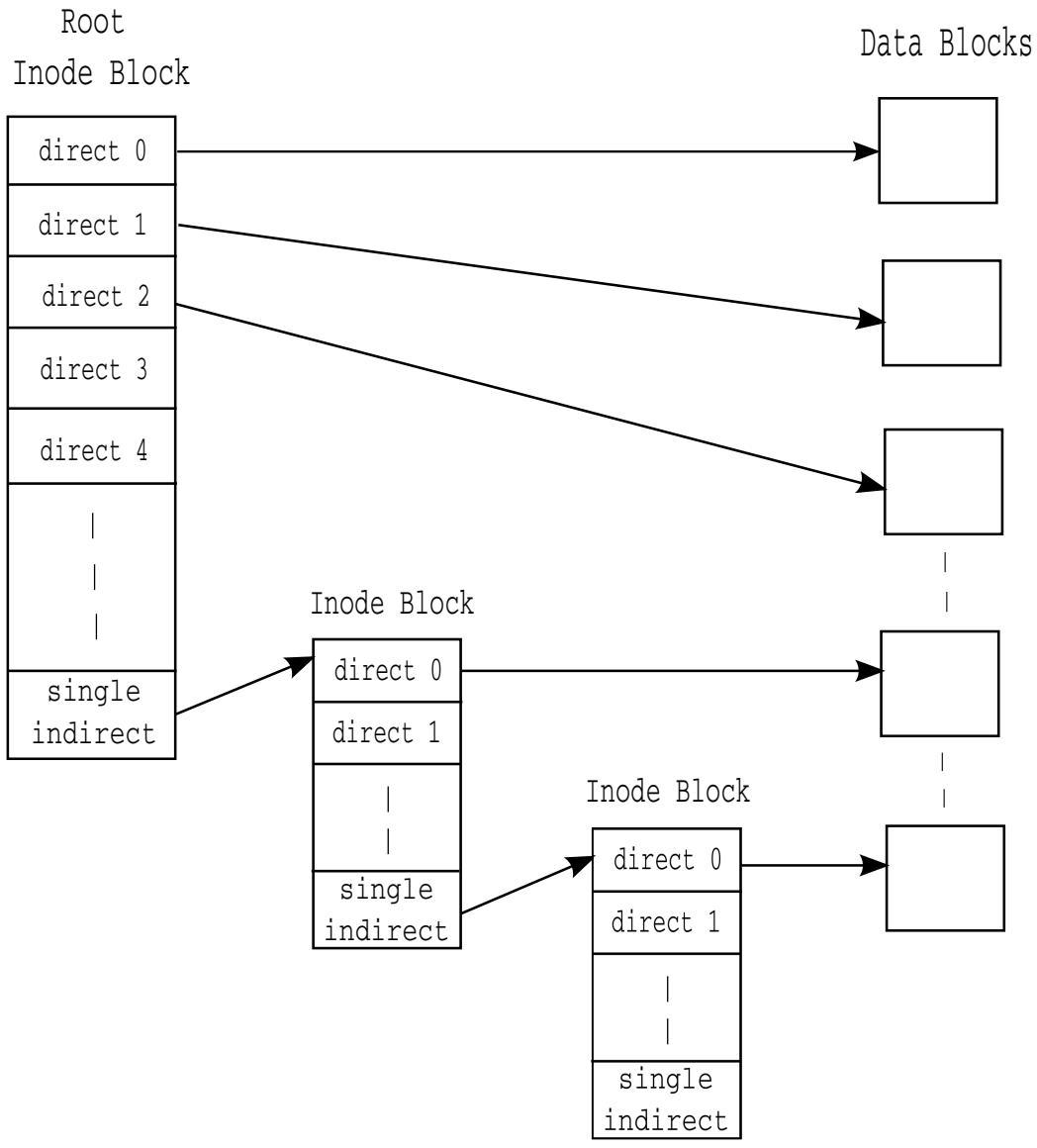


Figure 3.1: HWFS Inode Structure

Specifically, a buffer for the *Super Block*, a buffer for the currently open *Inode*, and a *Freelist* buffer. The HWFS performs basic file operations like open, close, read, write and delete. Details can be found in [35]. A simple interface consisting of command, status and data signals has been developed to allow compute accelerators to directly interface to HWFS. This minimizes complexity typically associated with I/O connectivity. A *filename*, *operation type file length* and *command start* are the signals needed from application core for file I/O transactions. HWFS is responsible for making disk controller block requests for the necessary file operation and delivering that data to the compute accelerator.

The HWFS core has additional functional improvements over what was reported in [35]. First is support for the *seek* operation. HWFS takes in a byte offset in addition to a Read operation command. Since, the filesystem only does block transactions with disk, the byte offset is first converted to a block request. The filesystem then uses the BLOCK_SIZE generic to seek to the relevant start byte within the first block fetched. The file length parameter is used to calculate the number of blocks to fetch after the start byte from disk. Since HWFS was conceived for large sequential file transfers, the initial design only used direct and single indirect pointers in its inode blocks. For seeking to offsets at large distances from the start of the file, this structure involves a penalty of reading and traversing the inode block linked list to reach the inode block of interest. To reduce this overhead, we add an additional data structure: a double indirect inode block which holds a list of single indirect pointers of a file. The double indirect inode blocks are in turn connected by a linked list and the pointer to the root double indirect inode is added to the superblock mapping it to the filename. To support this, an additional buffer was added to HWFS's datapath which holds the double indirect inode blocks. During a write file operation, the FSM stores the single indirect inodes of a file (the pointers that link the original inode block linked list) in the buffer and writes it out to the disk when full. During a seek operation, this data

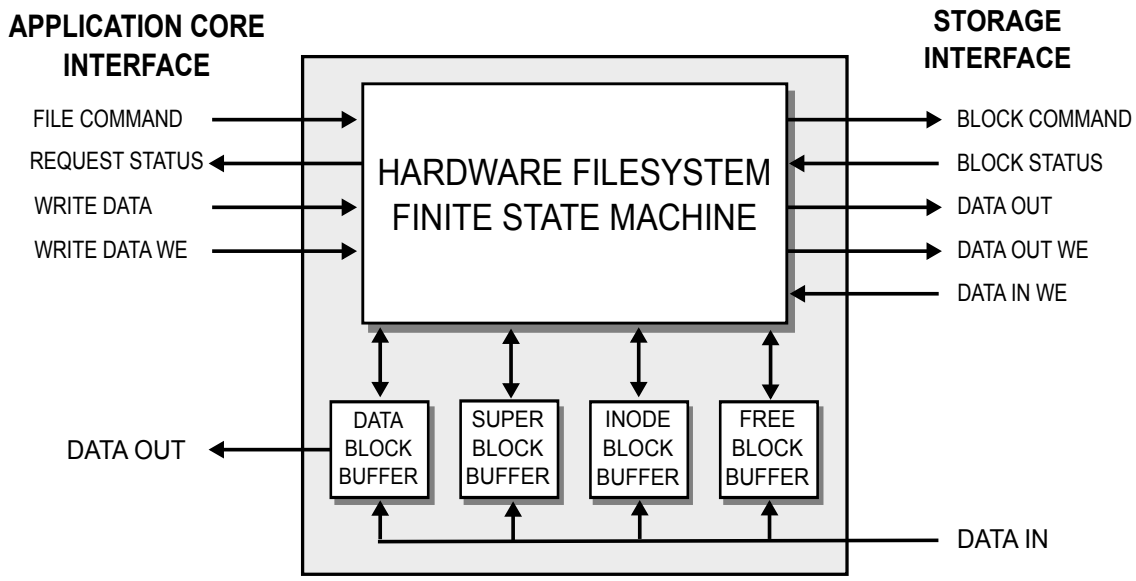


Figure 3.2: Hardware Filesystem Core: Block Diagram

structure is fetched from disk to get the pointer to the inode block of interest and avoiding the penalty of a sequential traversal of the original inode list. Thus, this design modification helped to avoid changing HWFS's original inode data structure with the only additional FPGA resource overhead for the extra on-chip buffer.

Support for multiple disks has been provided through the use of split transactions on disk controller. This allows multiple block requests to be issued to the memory subsystem. An internal counter in the FSM keeps track of outstanding block transactions. Section **??** discusses the multi-disk controller and integration with the HWFS in more detail.

3.2 Multi-Disk and RAM Disk Support

To further explore the feasibility and functionality of the HWFS core, a synthesized and operational design was required. However, commercial SATA disk controller cores are expensive and difficult to justify for a feasibility study. To make experiments — especially experiments with multiple disks — more feasible, a RAM Disk core was developed.

Figure 3.3 illustrates a high level block diagram of the system using the hardware

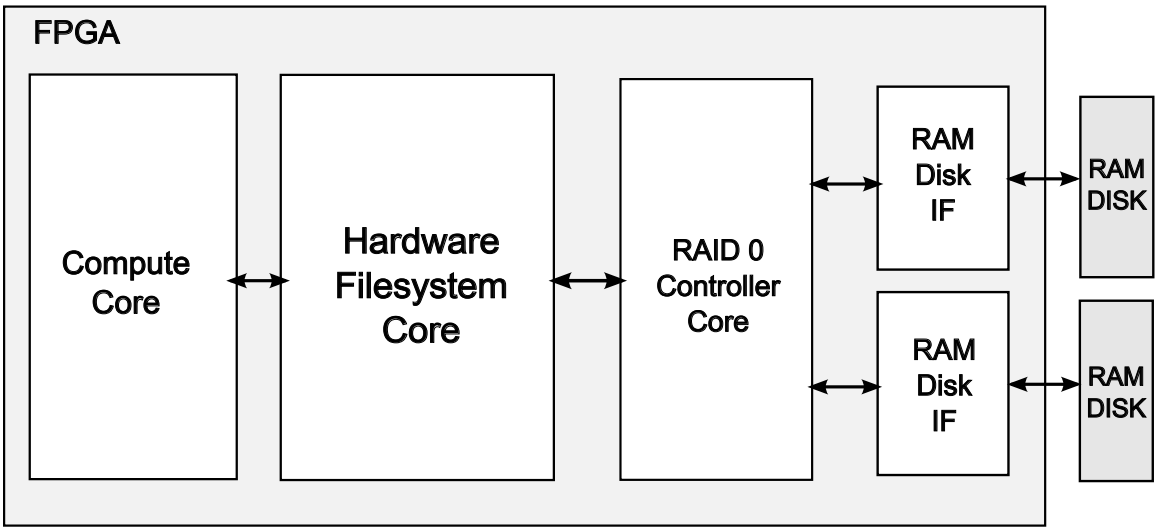


Figure 3.3: System level interface between HWFS and RAM Disk

filesystem and RAM Disk. The processor and computation core are both capable of interfacing with the hardware filesystem across the system bus. While the HWFS is targeted for a SATA hard disk, the HWFS core itself is designed with a generic interface to increase the number of devices that can be potentially interfaced with, beyond a hard disk. The Xilinx ML-410 FPGA board [37] provides interfaces for both ATA and SATA disks; however, to focus on the HWFS development the more complex ATA and SATA interfaces have been replaced with a RAM Disk.

3.2.1 Purpose of the RAM Disk

When presenting a hardware filesystem, it would be assumed the data would be stored on a hard disk. In the initial tests we have opted to use a specially designed RAM Disk in place of the SATA hard disk. There are several reasons why this approach was chosen. First and foremost is the cost of the SATA IP core. While SATA IP cores are currently for sale [38, 39], it is prohibitively expensive to purchase outright without any indication that the money would be well spent. Second is the design complexity of having to both create a hardware filesystem and integrate it with the SATA core in order to test even the simplest of file operations. Finally, while SATA may currently be the forerunner in the market, trends may soon shift to

alternative disks and interfaces which could cause another re-design of the system.

We have attempted to minimize initial cost and risk by focusing first on the design of the hardware filesystem. In simulation creating a simple SATA stub, which mimics some of the simple functionality of the SATA interface, enables a more rapid development of the hardware filesystem. In hardware there is no SATA stub, instead a fake disk must be created. External SDRAM presented itself as the ideal candidate with its easy and well documented interface. This RAM Disk is not targeted to be competitive with an actual hard disk, nor is it the long term goal of the Hardware Filesystem to include the RAM Disk. It simply provides an interface to large, off-chip storage that would allow for better testing of the Hardware Filesystem running on an actual FPGA. The data stored within the RAM Disk — super, inode, data and free blocks — are the same as the data that would be stored on that of a SATA disk. The key differences being the on-chip controller's interface and the data being stored in DDR2 instead of a physical disk.

As a result of the RAM Disk interface, we are now able to support any storage device by bridging the Hardware Filesystem's interface with the storage device's interface. This can be seen in Figure 3.4. While the complexity of the interfaces might be difficult to design, it should not be impossible, merely time consuming. The advantage of such an approach is with a working hardware filesystem the disk interface would take focus, reducing the number of unknowns in the design.

Finally, the RAM Disk based system is not aimed for performance. It should be obvious that the time to access a hard disk (rotational delay + seek time) will be constant between both a typical operating system's filesystem and the hardware filesystem. The RAM Disk system enabled us to measure the efficiency of the hardware filesystem.

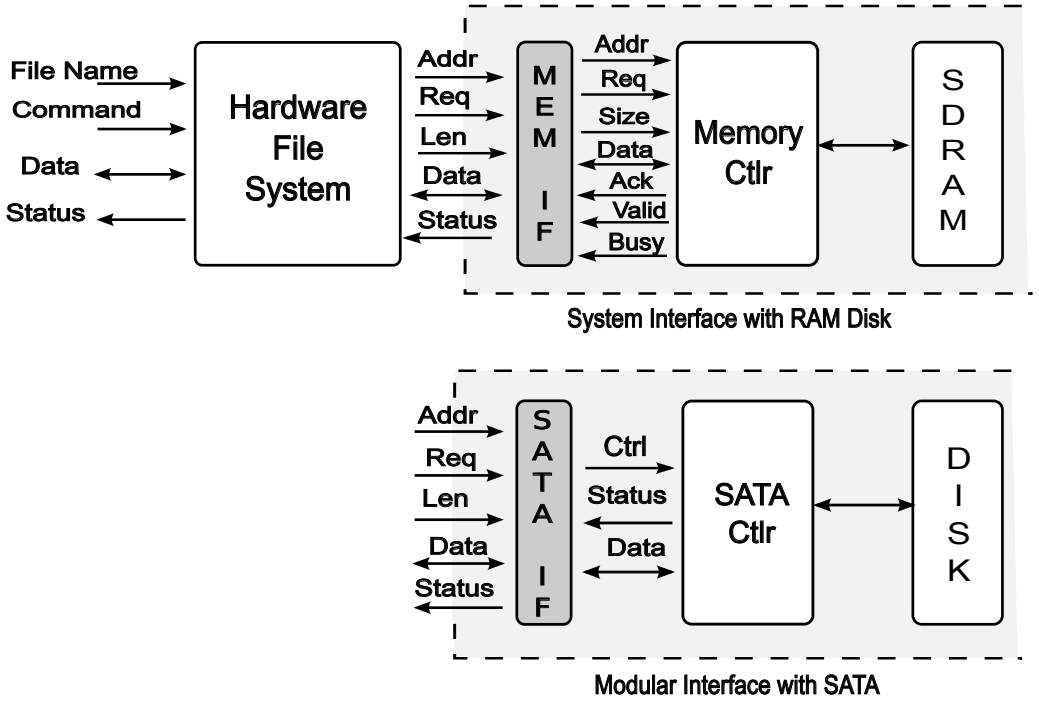


Figure 3.4: (a) Interface with RAM Disk (b) Modular Interface with SATA

3.2.2 Modular Interface to Disk

Figure 3.4(a) depicts the high level interface between the HWFS and the RAM Disk. Between the HWFS and the RAM Disk lies the Native Port Interface (NPI) to provide a custom, direct interface to the memory controller. The memory controller is a conventional soft IP core which communicates with the external memory. Requests from the HWFS are in the form of block transfers and it is the NPI which converts those block transfers into physical memory transfers.

Figure 3.4(b) highlights the flexibility of the HWFS core's design. Creating a simple interface between the HWFS and the SATA controller core is all that is necessary to port the RAM Disk implementation to a SATA implementation. Likewise, for any additional secondary storage the same process would apply.

3.2.3 Adding Multiple Disk Support

To support multiple disks a Redundant Array of Independent Disks (RAID)[40] Level 0 controller has been designed and synthesized for the FPGA. RAID 0 stripes

data across n number of disks, but does not offer fault-tolerance or parity. RAID 0 was chosen for this design as a first order proof of concept to investigate the question, how hard is it to add multiple disk support to the current Hardware Filesystem design? The initial design of the Hardware Filesystem core only supported access to a single disk, not a limitation, but instead a design choice to focus on the HWFS's internal functionality.

To provide support to multiple disks a handshaking protocol was established between the HWFS and the RAID 0 controller. Since the number of disks in the RAID system is unknown to the HWFS, requests should be issued as generically as possible. The handshaking protocol requires the HWFS to wait for a request acknowledge from the RAID controller before issuing subsequent requests. Initial designs with a single disk did not require this handshaking since only one request was in process at any given moment.

To illustrate the RAID 0, Figure 3.5 shows the Hardware Filesystem connected to the RAID 0 controller which is connected to two disks — (support for N disks is provided in the design). The stripe size in this design is one full block, but subblocks could be just as easily used. The RAID controller has been designed with a generic interface to allow easy support of any number of disks, limitations on the Xilinx ML-410 forced physical tests on the FPGA to two disks. More extensive tests of systems with greater than two disks have been performed and verified in simulation.

For a RAID controller with multiple disks, each read or write transaction could be to the same disk or to a different disk. For requests to the same disk the transactions are serialized, requiring the first transaction to complete before the second transaction can commence. For two requests to two separate disks, both requests can be issued in parallel. On a read request the RAID controller must also make sure the blocks are returned in the correct order since it is possible for two concurrent requests to be returned out of order.

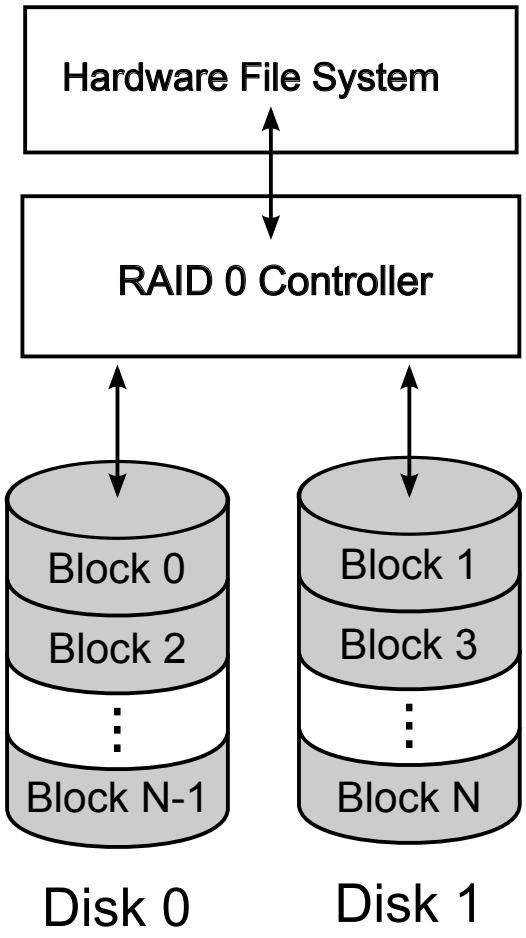


Figure 3.5: HWFS connected to the RAID 0 Controller for striping

With the successful integration of the RAID 0 controller, it is feasible to integrate more sophisticated controllers which offer parity, fault-tolerance, and mirroring of data in future designs. These higher RAID levels would still likely use the same interface to the HWFS core as the RAID 0, the difference would be the functionality within the RAID controller core itself.

3.3 Integration with BLAST and AIREN

The Reconfigurable Computing Cluster (RCC) project at UNC Charlotte [18] is exploring novel parallel computing architectures in High Performance Computing using FPGAs. While this cluster currently is comprised of 64-nodes (Xilinx ML410 Virtex4), it provides a valuable test bench for investigating the feasibility of using FPGAs exclusively as the computing platform to accelerate HPC applications. As part of this research, an investigation on accelerating and scaling I/O bound streaming applications was carried out. In particular, an FPGA implementation of the NCBI BLASTn (Basic Local Alignment Search Tool) algorithm was used. For communicating between FPGA nodes, the RCC cluster uses AIREN (Architecture Independent Reconfigurable Network), a custom integrated on-chip and off-chip network [41]. This provides low latency, high bandwidth connectivity of application cores without involving processors for network transactions. However, for accessing disk drives, application hardware cores would have to use a Linux software filesystem running on a slow embedded PowerPC processor operating at 300 MHz. This puts a limitation on the performance and scalability of applications implemented on FPGAs. To overcome this and enable direct access to secondary storage, HWFS was incorporated in a tightly integrated system with BLAST and AIREN.

To evaluate the scalability of this system, three types of nodes were constructed: head node, disk node and BLAST nodes. These were configured and tested with first a tree topology (reported in [42]) and a 4-ary 3-cube torus topology. The head node, shown in Figure 3.6 (a), consists of the BLAST accelerator cores, the Hardware

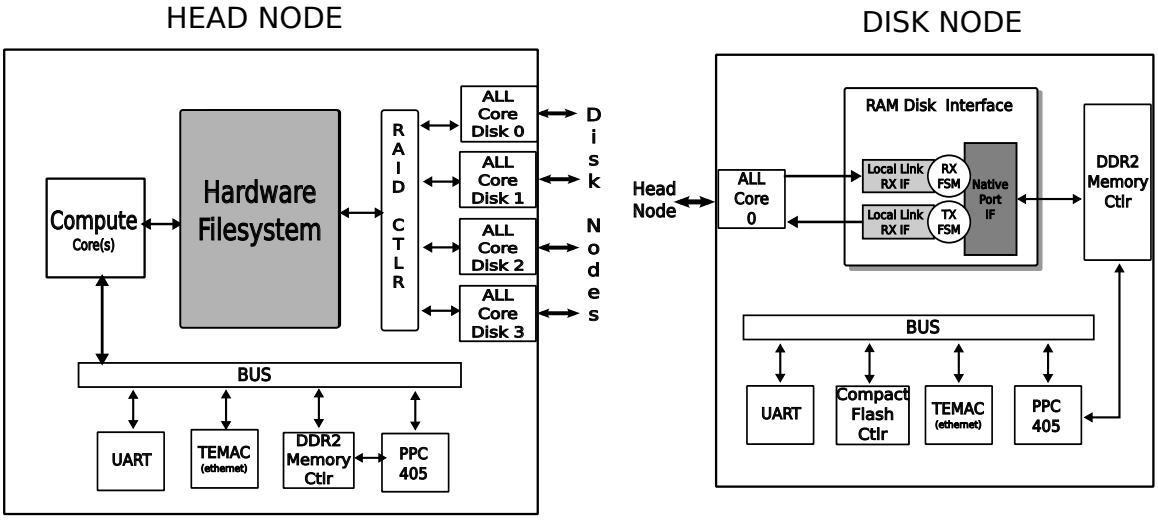


Figure 3.6: (a) Head Node with HWFS, BLAST and AIREN (b) Disk Node

Filesystem core, the RAID controller core and the Airen Local Link (ALL) interface. In addition, it also includes common System-on-Chip (SoC) components (processor, memory controller, system bus, etc) for a fully running Linux 2.6 kernel on the 300 MHz PowerPC 405 processor. Linux is primarily used to provide MPI support.

3.3.1 HWFS-BLAST Interface

The head node is responsible for initiating the retrieval of the databases from the HWFS on request by BLAST cores. The NCBI bioinformatics databases are formatted by HWFS and stored as files on the disk node. A simple direct interface is used to connect BLAST with HWFS. The HWFS core waits for a start command from the BLAST core in the idle state of its finite state machine. The BLAST core initially issues the *open* operation along with the new command signal to trigger the HWFS. Once, the database file is opened, HWFS reverts to the idle state. The BLAST core then issues the *read* operation, sends a desired database length (in bytes) and asserts the *new command* signal. Since the filesystem makes block transactions to the disk, the length information is first converted into the number of blocks to fetch for each sequence. The database is streamed in to the on-chip data FIFO and forwarded to BLAST. After the requested database is fetched, the HWFS issues a

done signal to the BLAST core and waits for the next request. On completion of all read transactions for every sequence of the database, the BLAST core issues a *close* command to the HWFS which transitions it to the idle state.

3.3.2 HWFS-RAID-AIREN interface

On the storage interface side, the HWFS-RAID interface described previously remains unchanged. The RAID-RAM Disk interface was split to relocate the RAM Disk logic to the disk node. A RAID-AIREN interface was created to enable the HWFS to access disks on remote FPGA nodes. This provides a point-to-point access to each disk node with 3.2 Gbps bidirectional bandwidth. Figure 3.6 (b) illustrates the disk node which contains a single bidirectional AIREN Local Link (ALL) network interface to the head node's RAID controller, and an interface to a RAM Disk (512 MB per RAM Disk). The filesystem is loaded on each node's RAM Disk from CompactFlash when the disk node is powered on and resides there until the system is shutdown. The initial design was implemented on a single node with two RAM Disks on local DDR and DDR2 memories. The HWFS-RAID-AIREN integration allowed testing with 4 and 8 disks. For this work again, RAM Disk was used as a storage medium in place of conventional disks due to unimplemented SATA disk controllers at that time. Further details of this work are reported in [43]

3.4 SATA Core

Although the functionality of the hardware filesystem was proven with a RAM Disk, it was still untested with real disk drives. For accessing non-volatile secondary storage, a Serial ATA Host Bus Adapter IP core was developed. The core is now open source and can be found at `https://opencores.org/project,sata_controller_core`.

This section highlights our design goals, gives an overview of the SATA core's user interface, and then provides the internal details of the design. (By "user" we are referring to the person designing an FPGA-based application that would like to use

the core.)

3.4.1 Design Goals

The SATA core has been designed primarily with the intention of providing users with an easy-to-use interface and the ability to interface directly to a high-bandwidth, non-volatile storage system. Commercial SATA IP cores in general keep the command layer and part of the transport layer (that deals with encoding a command into a Frame Information Structure) in software [38, 39]. A host processor generally builds the command FIS in system memory and transfers it to the SATA core via DMA and FIFOs. This enables flexibility in terms of supporting the full range of SATA commands. However, to allow FPGA cores to access disks directly, we have implemented the command layer in hardware and support the minimum subset of the ATA commands necessary for enabling features, status, reading, and writing sectors. Last, to support soft processors and operating systems, a bus interface to the command layer was needed. A block device kernel driver makes the mass storage available as conventional secondary storage sub-system.

3.4.2 SATA Core Interface and Modules

We have implemented the link, transport and command layers of the Serial ATA communication protocol in our design. Modern FPGAs have Multi-Gigabit Transceivers (MGTs) which support a variety of high-speed serial protocols. The Physical Layer of the SATA protocol is implemented using a wrapper around the Xilinx RocketIO GTP transceivers [44] in Virtex 5 and GTX transceivers [45] in Virtex 6.

The SATA core, shown in Figure 3.7, provides a simple user interface for reading and writing to storage devices. A user core sets the start *sector address*, *number of sectors*, and *type of request* (a read or write). It checks the *ready for cmd* signal and triggers the *new cmd*. Data can be sent/received through a 32-bit interface with supporting FIFO like handshaking signals (*full*, *empty*, *write en*, *read en*). The *command done* and *command failed* signals indicate the completion status.

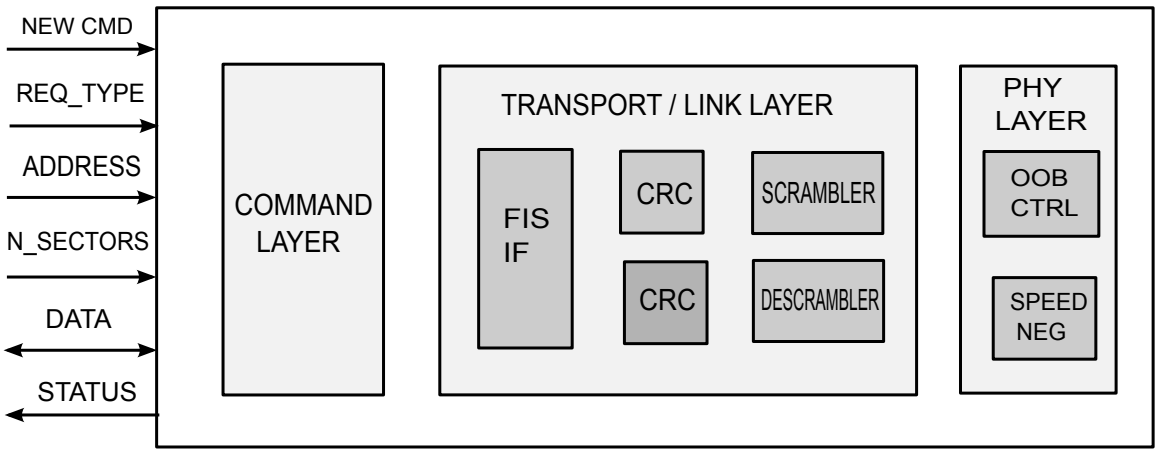


Figure 3.7: Serial ATA Host Bus Adapter Core

The modules in the design are described next, starting from the outermost layers (application side).

3.4.2.1 Command Layer Module

A subset of the ATA command set (Read DMA Ext, Write DMA Ext, FPDMA Read, FPDMA Write, Set Features and Identify Device) has been implemented using an FSM at this layer. The command layer module decodes read/write commands from the top level entity to issue the appropriate read/write Sector commands to the transport layer. Each command execution is a sequence of special data structures called Frame Information Structures (FIS) exchanged between the SATA host and drive. An example command sequence for Write DMA Ext command is shown in Figure 3.8

3.4.2.2 Transport Layer Module

The transport layer constructs and decomposes the FISs requested by the command layer. These deliver command, data, status and control information. An example of the Register Host to Device FIS for the Read DMA Ext command is shown in Table 3.1. The user parameters such as command type, sector address and number of sectors are encapsulated according to the ATA format. FIS type is indicated by the Frame Information type field located in byte 0 of first Dword (32 bits) of pay-

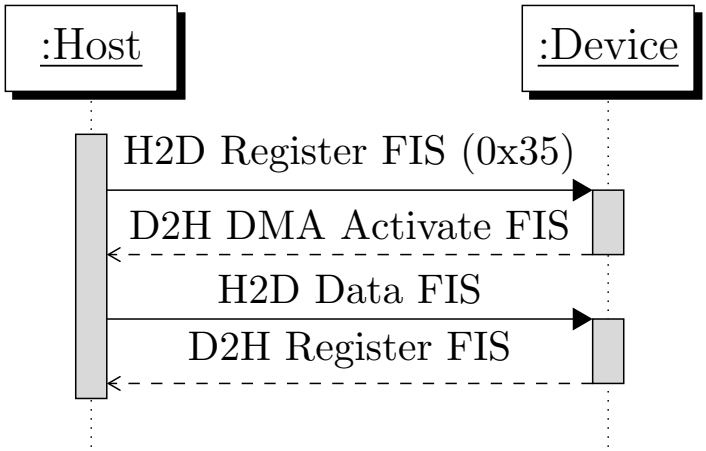


Figure 3.8: Write DMA Ext command sequence

Table 3.1: Register H-D FIS for Read DMA Ext

DWORD	**Byte3**	**Byte2**	**Byte1**	**Byte0**
0	Features	Command (25h)	Interrupt	FIS Type (27h)
1	Device	LBA High	LBA Mid	LBA Low
2	Features (exp)	LBA High (exp)	LBA Mid (exp)	LBA Low (exp)
3	Control	Reserved	Sector Count (exp)	Sector Count
4	Reserved	Reserved	Reserved	Reserved

load. The FISs used in our implementation and their characteristics are listed in (Table 3.2). These act as payloads of a Frame which is transmitted by the link layer. FIS transmission successes and errors are reported by the device through a Register Device to Host FIS. The transport layer FSM decodes the FIS and checks for errors in the ATA status and error fields of the FIS (same as command and feature field in Table 3.1). The FIS transmit buffer retains a copy of each command FIS (Register Host to Device) and retransmits it in case of an error. Data FISs which are bigger in size (maximum of 8196 bytes) are not retained due to cost considerations of the transmit buffers.

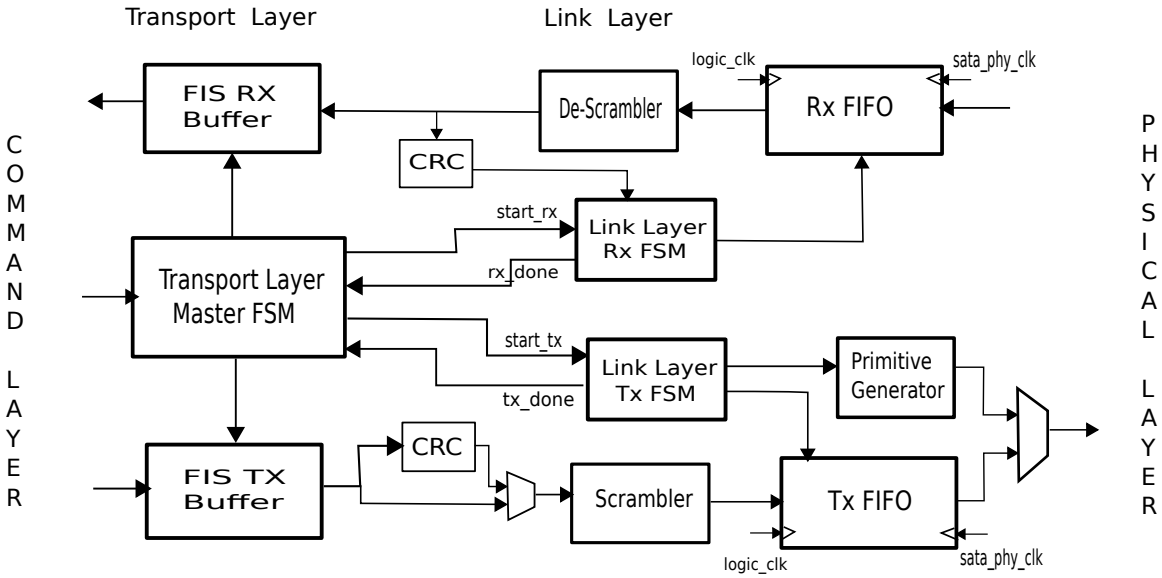
3.4.2.3 Link Layer Module

The link layer is essentially concerned with the framing and delivery of each FIS (created by the transport layer) by following a frame transmission protocol. For this purpose, it uses special control words called primitives which are unique patterns

Table 3.2: FIS types and characteristics

FIS type	ID	Size	Direction
Register FIS	27H	5 DWs	Host to Device
Register FIS	34H	5 DWs	Device to Host
DMA Activate	39H	1 DW	Device to Host
FPDMA Setup	41H	7 DWs	Device to Host
Set Device Bits	A1H	2 DWs	Device to Host
Data	46H	2049 DWs	Bidirectional

defined in the protocol. These are used for managing the flow of a frame as well as for frame construction. Figure 3.9 depicts the control units and data paths of the transport and link layer modules which work together to create and control the delivery of each FIS. The transport layer FSM creates a command FIS (Register Host to Device) and stores it in the FIS transmit buffer. It then issues a request to the link layer FSM to process it. The link layer RX and TX FSMs deal with frame construction and de-construction process. The TX FSM reads the FIS buffer, calculates a 32-bit CRC (Cyclic Redundancy Check) and appends it at the end of the frame payload. The next step is to add SOF (Start of Frame) and EOF (End of Frame) primitives to mark the frame boundaries as shown in Figure 3.10). This helps the receiver in identifying each FIS from the stream of information on the SATA link. In order to prevent Electromagnetic Interference (EMI), the frame needs to be sent through a scrambler circuit. Scrambling is performed by XORing the data to be transmitted with the output of a linear feedback shift register (LFSR). This process spreads out the noise over a broader frequency spectrum. However, the primitives are not subject to scrambling since the data patterns that define each primitive must be detected by the receiver's physical layer. Hence, to simplify the implementation, we first pass the FIS with the appended CRC through the scrambler and then add SOF and EOF primitives at the output of the transmit FIFO i.e. a mux is used to switch between the primitive generator and scrambled FIS as shown in Figure 3.9


Transport Layer
Link Layer
COMMAND LAYER
PHYSICAL LAYER
FIS RX Buffer
De-Scrambler
Rx FIFO
logic_clk
sata_phy_clk
CRC
start_rx
Link Layer Rx FSM
Transport Layer Master FSM
rx_done
start_tx
Link Layer Tx FSM
Primitive Generator
tx_done
FIS TX Buffer
CRC
Scrambler
Tx FIFO
logic_clk
sata_phy_clk

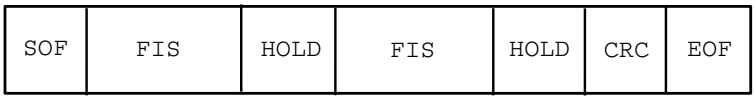


Figure 3.10: SATA Frame structure

Figure 3.11 depicts the frame transmission protocol between the host and device. Initially, when the SATA link is idle, a stream of SYNC primitives are continuously transmitted by the host and device. When the host is ready to transmit a frame, it sends X_RDY primitives (transmitter ready). The device responds with R_RDY(receiver ready). Next, the host begins frame transmission by sending the SOF and FIS to the device. Once the device recognizes a valid frame, it sends R_IP (reception in progress) primitives to the host (Note that the SATA protocol is half duplex). The host terminates the frame by sending the CRC and an EOF primitive and waits for a frame verification response from the device by sending WTRM (wait for frame termination). The device sends an R_OK or R_ERR to signal successful or failed frame transfer. The link is returned to the idle state by sending SYNC primitives. On the receive side, the frame from the physical layer is de-constructed and passed through the de-scrambler. The link layer RX FSM then calculates the CRC and checks it against the appended CRC. In case of a CRC error during reception or a transmission error detected by R_ERR, the link layer notifies the transport layer which retransmits the FIS as explained in Subsection 3.4.2.2 Additionally, the link layer also performs flow control to prevent buffer underflow and overflow conditions on the transmit and receive interfaces using HOLD and HOLDA primitives.

3.4.2.4 Physical Layer Module

The Hard IP transceivers are responsible for serializing/de-serializing the data as well as 8bit to 10bit encoding/decoding. However, before a data communication link can be established the SATA protocol requires that a reset, synchronization and link initialization process take place through the use of Out-of-Band (OOB) signals. This is handled by an FSM in FPGA logic. While developing the SATA core, we have used

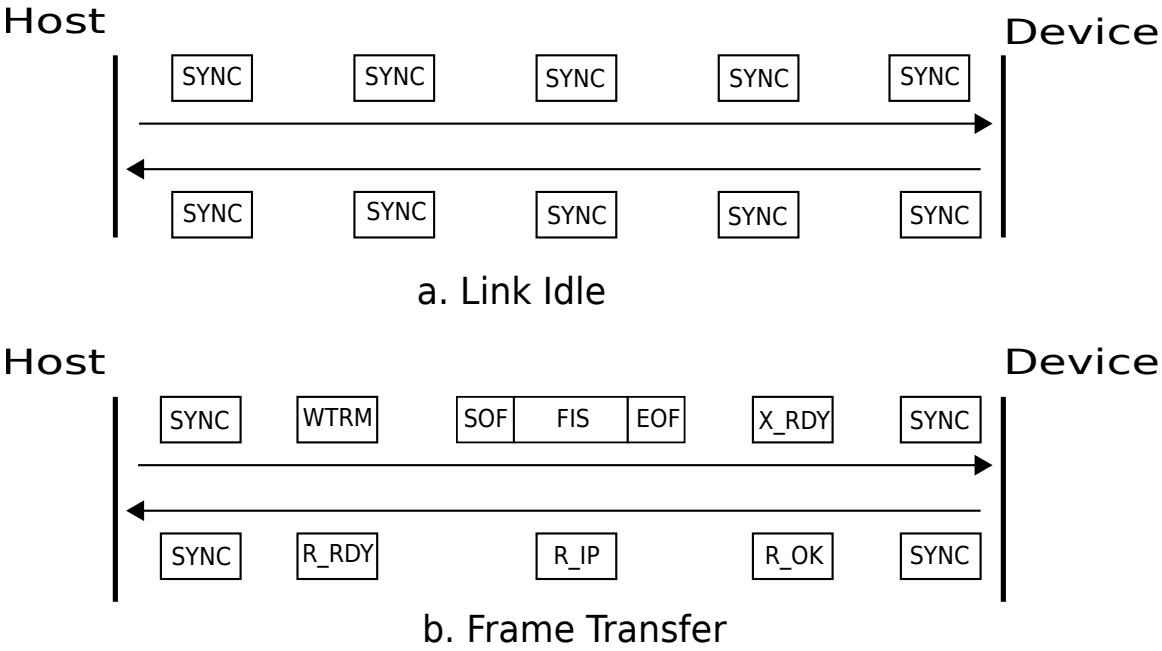


Figure 3.11: Frame Transmission Sequence

the SATA core link initialization reference design provided by the Xilinx Application note [46] as a starting point. However, this reference design is compatible with the GTP transceivers (16 bit data) of the Virtex 5 FPGA. In order to support Virtex 6, we first generated a GTX wrapper using Xilinx Coregen and configured the transceiver's OOB parameters for SATA2. The GTP and GTX transceivers differ in the data path width (16-bit for GTP and 32-bit for GTX) as well as the ports used for OOB signals (although the sequence of OOB signals remains the same). Hence, we had to rewrite the OOB signalling controller state machine and the clocking modules and interface it to the GTX wrapper.

3.4.3 Native Command Queueing

Native Command queueing (NCQ) is a mechanism for optimizing read and write operations by queueing up several requests on the drive. This was initially conceived for hard disks which could internally optimize the order of commands to minimize drive head movement (seek time). Solid State Drives having multiple flash chips also stand to benefit from NCQ by using it for concurrently accessing these flash chips.

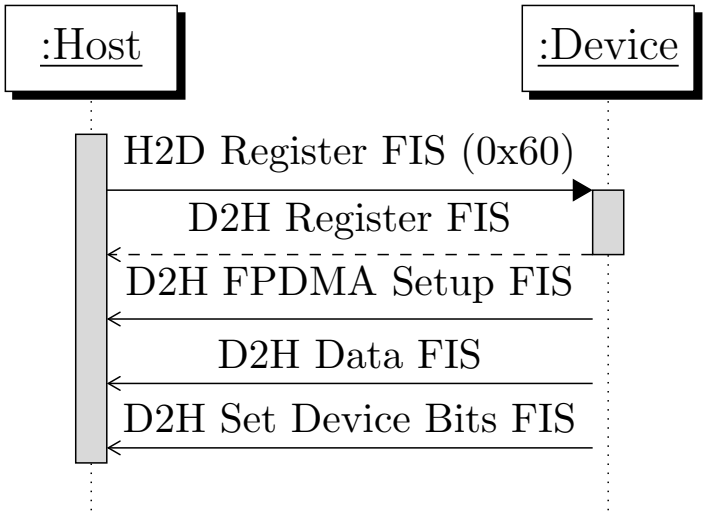


Figure 3.12: Read FPDMA command sequence

SATA NCQ supports a maximum queue depth of 32 outstanding commands.

The SATA protocol provides special commands: FPDMA Read and FPDMA Write for NCQ. A 5-bit tag is added to the command frame (H2D Reg FIS) and sent along with each queued command. The drive acknowledges that the command has been enqueued by sending a Device-to-Host Register FIS. At this point, the host is free to issue more commands to the device. When the device is ready to send or receive data, it sends a DMA Setup FIS with a tag corresponding to the command being completed. In case of an FPDMA Read (Figure 3.12), the drive sends the Data FIS followed by a Set Device Bits FIS to indicate command completion. (This FIS basically contains a 32-bit bitmap to correspond to each of the 32 NCQ commands). Multiple sequences of DMA Setup FIS - Data FIS - Set Device Bits FIS are sent by the device to complete all the enqueued commands. For an FPDMA Write, the drive sends a DMA Activate FIS after the DMA Setup FIS to notify the host that it is ready to receive data. The host delivers a Data FIS and the device sends back a Set Device Bits FIS after completing the write. Again, the above sequence repeats until all the queued commands are completed.

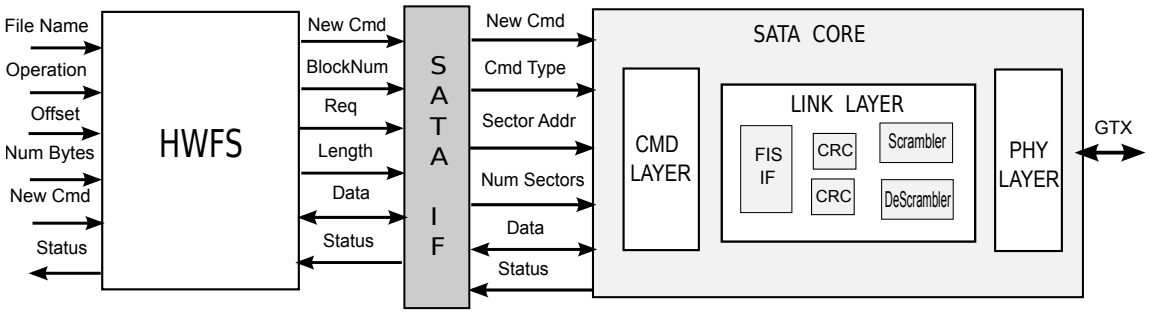


Figure 3.13: HWFS-SATA system

3.4.4 Linux Block Device Driver

Nowadays, it is common for one or even more microprocessors to be embedded in an FPGA as either hard cores or soft cores. Among many operating systems capable of running on an FPGA platform, Linux is arguably the most popular one. Therefore, besides the ability to directly interface with FPGA-based cores, we also provide an optimized Linux block device driver that makes our SATA core available to the operating system. Detailed Linux I/O stack has been covered in [47]. However, a traditional hard-disk-centric design in Linux I/O stack can not fully leverage the capability of a SSD mainly due to two reasons [27]. First, some layers of abstraction are unnecessary such as SCSI emulation for ATA drives. Second, queue schedulers, which are useful for hard disks, will add CPU load and delays to a SSD-based system. More aggressive tuning of block device driver to minimize software overheads was reported in [8]. Our device driver uses the existing kernel function called `make_request` to avoid invoking the queue scheduler. Nor does this block device driver implement SCSI/ATA emulation. Instead, the driver directly interacts with the command layer implemented in the SATA core.

3.5 System Integration

The final step was to integrate the Hardware Filesystem and SATA Cores to create an end-to-end system for evaluating performance with real disks. For this purpose, an interfacing layer (SATA IF) was created as shown in Figure 3.14. This converts

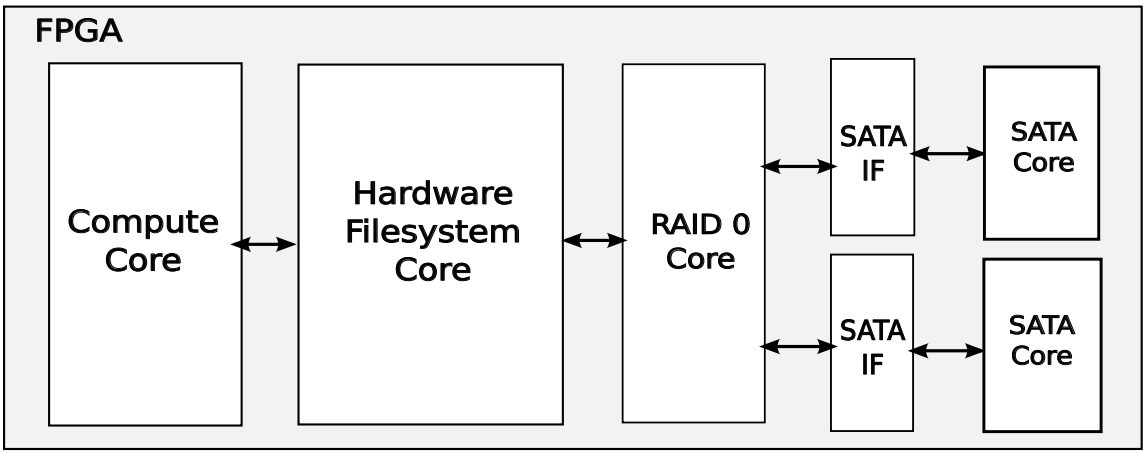


Figure 3.14: HWFS-RAID-SATA system

the file block requests from the HWFS core into sector requests by issuing sector address, number of sectors and the appropriate read/write command and manages the flow of data between the cores. (This essentially does the job of a software device driver layer which translates block commands from a regular software filesystem to ATA commands.) The HWFS had been already augmented with the ability for split transactions described in Subsection 3.2.3. With the SATA Core's support for Native Command Queueing, the interfacing logic enqueues the commands from HWFS and sends acknowledgements to HWFS for the respective commands. When the SATA core is ready to process a new request, it sends out the commands from the queue. It also makes sure that HWFS receives the requested blocks in the correct order from the SATA Core. This system minimizes the command processing overhead and helps in utilizing the available bandwidth from a single SATA channel efficiently.

With the single disk system in place, to test HWFS's performance scalablity with multiple SATA disks, the system shown in Figure **??** was built. The RAID controller described previously was ported and integrated with the SATA IF.

CHAPTER 4: EVALUATION

To establish whether implementing a filesystem directly in hardware provides performance improvements over a software filesystem while utilizing reasonable on-chip resources, we synthesized the HWFS-SATA core subsystem and conducted experiments first with a RAM Disk on a Xilinx ML410 and then with Solid State Drives on a Xilinx ML605. Details of the previous simulation experiments are available in [35, 36]. The experimental setups of the synthesized design, results obtained, analysis and evaluation of the thesis metrics follows.

4.1 Experimental Setup

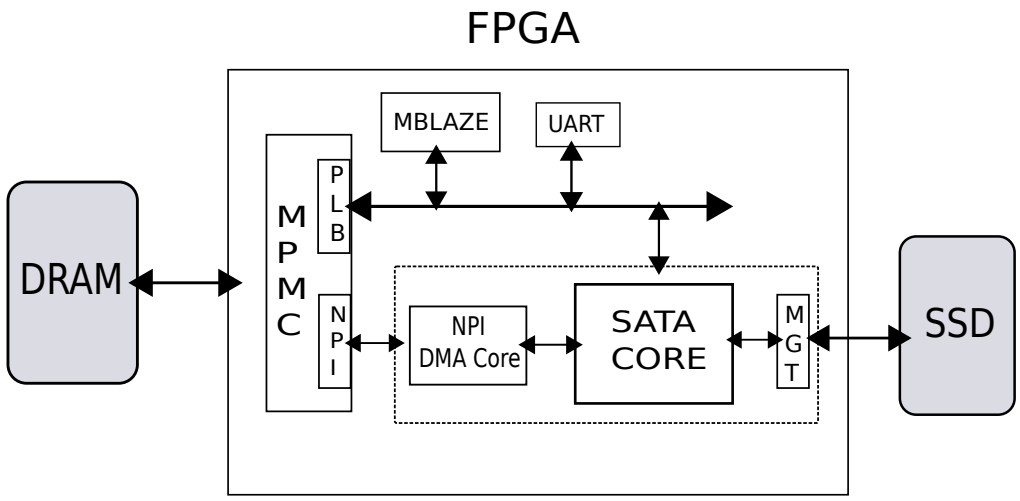
Since the design and implementation of HWFS and the supporting infrastructure was staged, we describe four different experimental setups for measuring efficiency, bandwidth and comparison with a software filesytem.

4.1.1 Setup 1 : HWFS - RAM Disk on ML-410

The first setup for the system running on the ML-410 builds upon the description given in Chapter3. A hardware base system was created using Xilinx EDK, consisting of a processor (PowerPC), system bus, on-chip memory, external memory, and the HWFS core (100 MHz) with RAM Disk interface. The test begins with the Embedded CPU initializing the Disk with the empty root filesystem. Once the Disk has been initialized, the test application exercises HWFS by issuing multiple *open, read, write* and *delete* commands. After the test finishes, the PowerPC reads the RAM Disk to verify the successful completion of the test.

4.1.2 Setups 2 and 3 : SATA Core and HWFS-SATA on ML605

The transceivers on the ML410-boards are incompatible with the physical layer of the SATA protocol. Hence, in order to test the functionality and performance of the



MGT : Multi Gigabit Transceiver
MPMC : Multi-port Memory Controller
NPI : Native Port Interface
MBLAZE : Microblaze Embedded CPU
SSD : Solid State Drive

Figure 4.1: SATA Core Test Setup

SATA core independently as well as the integrated HWFS-SATA system, two different base systems were created for the ML605 board shown in Figures 4.1 and 4.2 Both systems include FPGA SOC components like CPU (MicroBlaze), system bus (PLB), memory, DMA interface and UART. Since the ML605 board itself does not include SATA connectors, an FPGA Mezzanine Card (FMC XM104 [48]) was attached.

In Setup 2, a test application on the Microblaze issues sector requests to the SATA core. While the SATA core's FIFOs could be read and written through the PLB's slave register interface, this would not be optimum in terms of performance. Another option, was to use the Xilinx Central DMA Controller IP which saves processor time, but still uses the system bus for copying data between system memory and the SATA core. To test the peak performance of the SATA core, it was necessary to supply and consume data at a higher rate. Hence, we developed a custom DMA core with direct access to DDR via high bandwidth Native Port Interface (NPI) to the memory controller. This enables the use of NPI channel for supplying test data form DDR while relying on the slower MicroBlaze-PLB interface for command and status. A

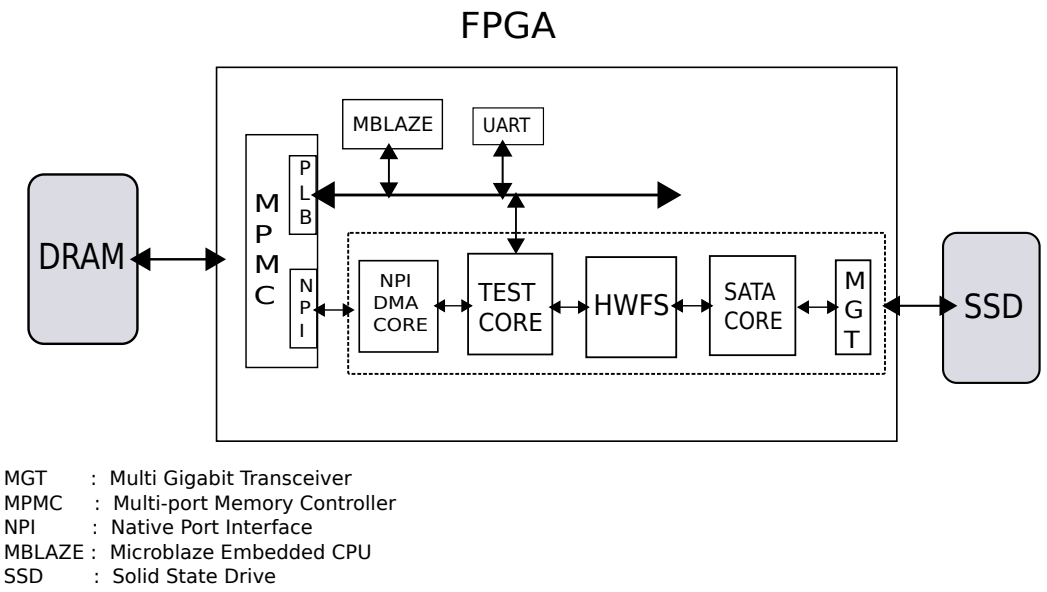


Figure 4.2: HWFS-SATA Test Setup

linux block driver was also added to allow the SATA core to appear as a block device to the Operating System.

In the third Setup, a workload generating test core was developed to emulate file I/O requests from FPGA-based applications. This core directly interfaces to HWFS's generic application-side interface with FIFOs for data transfer. Filesystem calls are made by issuing multiple *open, read, write* and *delete* commands to HWFS with sequential and random access patterns. A C test application on the Microblaze controls the workload generating test core, gathers performance numbers and checks the completion status. Here again, the DMA channel is used to transfer data from system memory to the test core.

4.1.3 Setup 4 : CPU - SATA on Linux Server

In order to compare the performance of HWFS with a software filesystem, the same OCZ SSD was attached to a modern Linux Server. The machine used is a 2.1GHz Quad core AMD Opteron CPU with 16 GB of system memory and an Nvidia SATA2 Host Bus Adapter chipset.

4.2 Results

The HWFS RAM Disk system is first used for measuring efficiency of HWFS and comparing with ideal disk efficiency achieved in previous simulation experiments. Next, bandwith numbers of the SATA core and the HWFS-SATA subsystem for sequential and random workoads and benefits of Native Command Queueing with supporting analysis are provided. Then, in order to evaluate the thesis metric of Bandwidth, we provide a performance comparison of HWFS-SATA with EXT2 filesystem on a modern Linux Server. The scalability of HWFS with multiple disks using the RAID controller is also shown. Lastly, sizes of the cores in terms of FPGA resource count are measured and reported to evaluate the metric of size.

4.2.1 Performance

4.2.1.1 HWFS Efficiency

To evaluate the amount of overhead induced by the filesystem itself for metadata operations, the execution times of sequential read and write operations were measured with single and two RAM Disks. For comparison with an ideal disk (zero delay), the simulation efficiency plot is shown (4.3). The filesystem's efficiency was computed as the ratio of the time taken to transfer raw data blocks of a file between the HWFS and disk to the total transfer time with the filesystem's processing overhead.

$$\text{eff} = \frac{\text{raw block transfer time}}{\text{filesystem block transfer time}}$$

$$\text{raw block transfer time} = \frac{\text{file size x clock cycle time}}{\text{bytes per clock cycle}}$$

The overhead includes the time taken to read the Super Block, find a file name match, get its root inode block (open file operation), read the inode blocks of the file (read file operation) and read/write free blocks and inode blocks (write file operation). Figure 4.3 shows a plot of the sequential read and write efficiencies for 64 B, 512 B

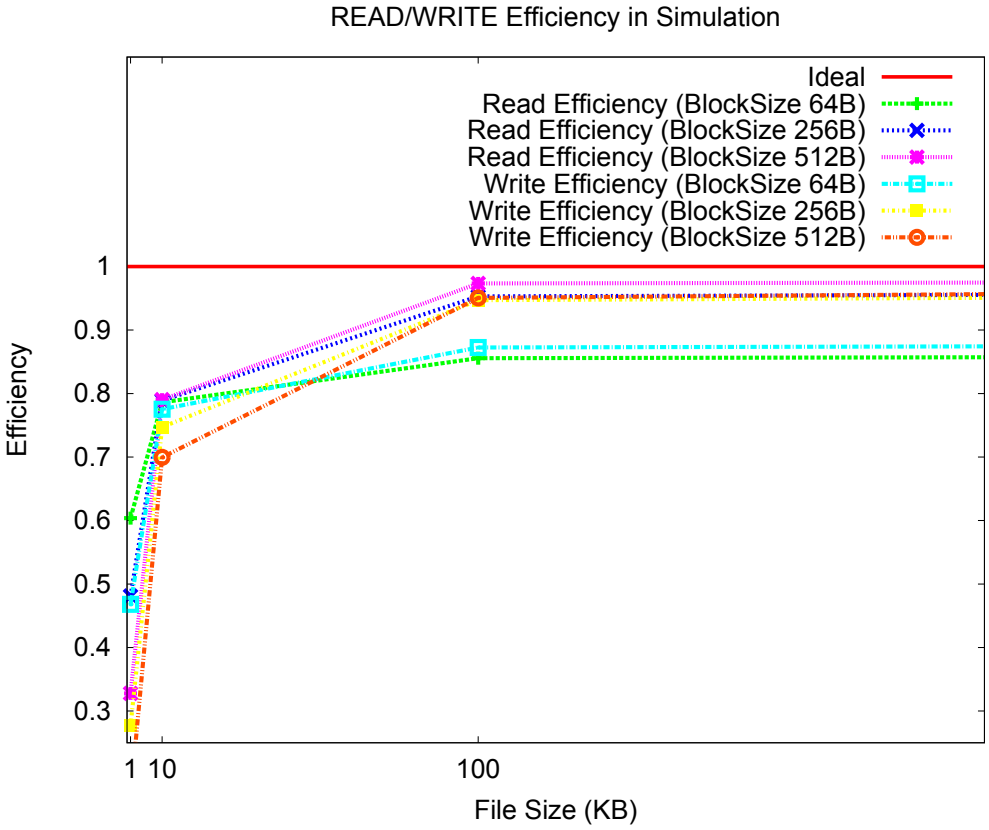


Figure 4.3: HWFS Sequential Read/Write Efficiency in simulation

and 1024 B sized blocks plotted against different file sizes. It is observed that for small files (1 KB to 10 KB) the efficiency is below 80% . It increases to 95% for 100 KB files and saturates for very large files (shown by a flattening of the plot for file sizes beyond 100 KB). This is due to the overhead having little effect on the execution times for large files thereby achieving efficient run-time performance. (To emphasize the transition in efficiency, the x-axis is restricted to 250 KB in the Figure).

Single RAM Disk Results Table 4.1 gives the execution measurements for read/write operations with a single RAM Disk synthesized and run in hardware. Unlike the simulation tests, the RAM Disk is not an ideal disk and the execution times increase accordingly. For a real SATA disk these numbers would again increase; however, the importance of this test is to show that running in actual hardware produces similar trends to simulation when taking into account the storage media's

Table 4.1: HWFS Read/Write Execution Time with single RAM Disk

	Read			Write		
File Size (Bytes)	**64 B**	**512 B**	**1024 B**	**64 B**	**512 B**	**1024 B**
1 KB	9.28 μs	12.54 μs	19.62 μs	9.4 μs	28.3 μs	51.77 μs
10 KB	73.59 μs	45.8 μs	51.28 μs	52.3 μs	59.55 μs	83.02 μs
100 KB	709.84 μs	380.97 μs	366.32 μs	483 μs	391.56 μs	396.65 μs
1 MB	7.18 ms	3.76 ms	3.55 ms	4.9 ms	3.57 ms	3.54 ms
10 MB	71.8 ms	37.44 ms	35.35 ms	48.97 ms	35.48 ms	34.82 ms
100 MB	717.93 ms	374.33 ms	353.32 ms	489.65 ms	354.53 ms	347.69 ms

Table 4.2: HWFS Execution time for a 1 KB file, 64B block size

Operation	**Total**	**HWFS**	**RAMs**
Write	9.29 μs	5.54 μs	3.75 μs
Read	9.16 μs	4.32 μs	4.84 μs
Delete	5.27 μs	2.66 μs	2.61 μs

access times.

Table 4.2 is presented to highlight the time taken by the filesystem to process data in comparison with the RAM Disk memory transaction time. For a write operation the execution time of the Hardware Filesystem is 5.54 μs compared to the simulation time of 5.47 μs (refer [35, 36]). This shows that the Hardware Filesystem is able to maintain the same performance with a RAM Disk as with the simulation's ideal disk. The same holds true for the read operation.

The efficiency of the Hardware Filesystem with a single RAM Disk is shown in Figure 4.4. The HWFS stalls until both the block requests to and from memory are satisfied. Due to this added memory transaction latency, the efficieny graph shows a dip in performance as compared to the simulation efficiency in Figure 4.3.

Multiple RAM Disks Results The split transactions implemented for multi-disk support provides an improvement over the single disk efficiency. Test results and the efficiency graph for read/write operations over two RAM Disks are shown in Table 4.3 and Figure 4.5.

For 64 byte blocks, the memory channel bandwidth is underutilized. Ideal transactions would be bursts of 128 bytes or larger. It is observed from Figure **??**, that

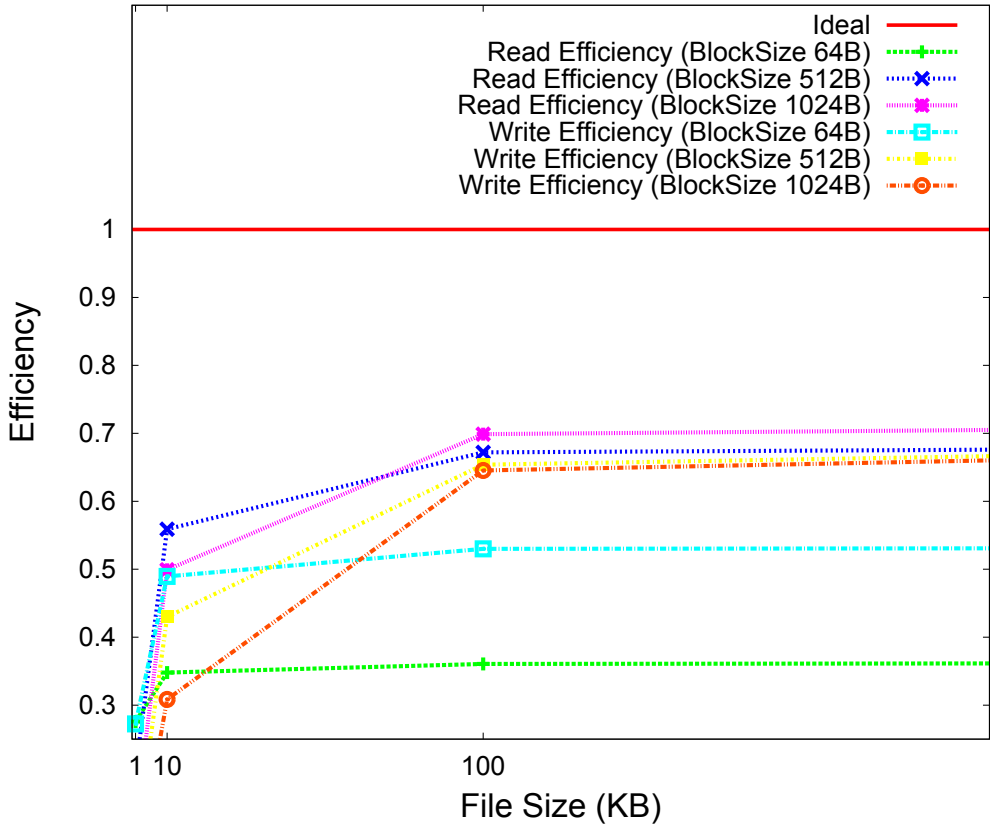


Figure 4.4: HWFS Sequential Read/Write Efficiency with single RAM Disk

Table 4.3: HWFS Read/Write Execution Time with two RAM Disks

	Read			**Write**		
File Size (Bytes)	**64 B**	**512 B**	**1024 B**	**64 B**	**512 B**	**1024 B**
1 KB	8.4 μs	10.69 μs	17.05 μs	13.08 μs	21.87 μs	36.86 μs
10 KB	66.17 μs	34.47 μs	40.48 μs	78.33 μs	50.85 μs	62.87 μs
100 KB	636.63 μs	274.35 μs	274.45 μs	734.69 μs	344.09 μs	321.7 μs
1 MB	6.4 ms	2.75 ms	2.69 ms	7.4 ms	3.37 ms	3.02 ms
10 MB	64.3 ms	27.47 ms	26.7 ms	74.7 ms	33.54 ms	29.86 ms
100 MB	643.28 ms	274.68 ms	267.84 ms	746.8 ms	335.3 ms	298.38 ms

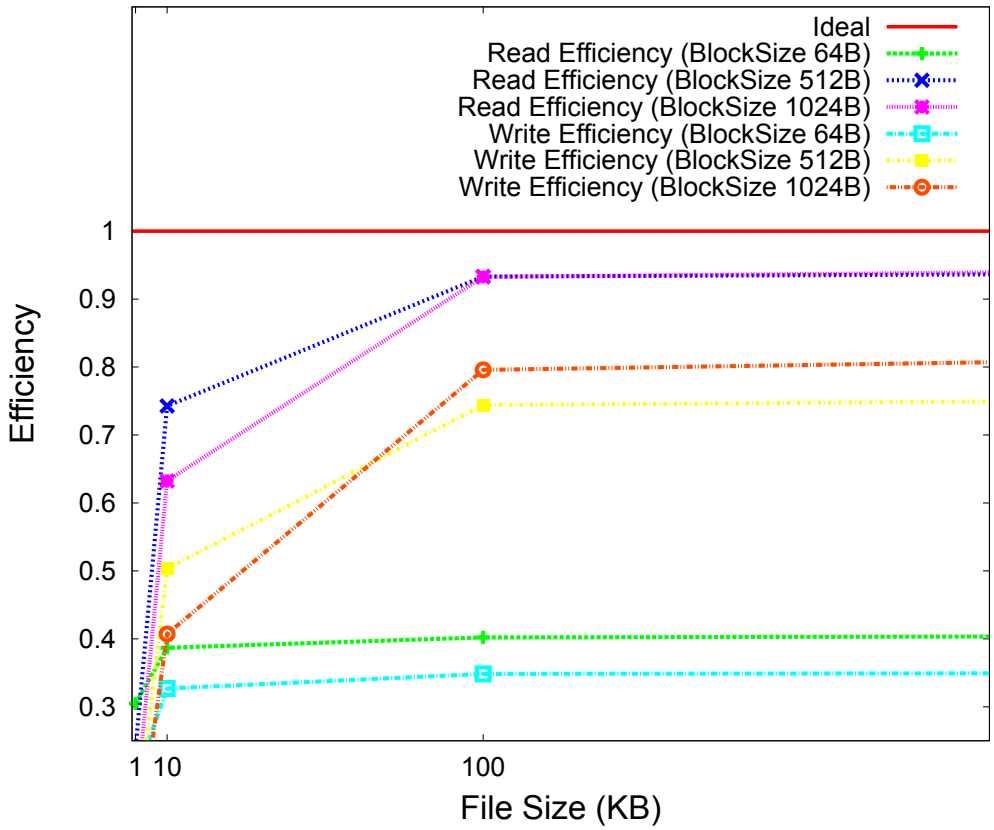


Figure 4.5: HWFS Sequential Read/Write Efficiency with two RAM Disks

the efficiency increases with the size of the block for the same file size. This is due to the improvement in the data transfer bandwidth with the HWFS block size. Using block sizes larger than 1024 B increases the BRAM usage for the core's metadata buffers without providing any substantial improvement in efficiency. Adding multiple disk support allowed two transactions to be processed in parallel from each memory channel, increasing overall efficiency. Given these trade-offs, 1024 B blocks prove to be ideal for the RAM Disk system.

4.2.1.2 SATA Core Bandwidth

In this sub-section, we report raw bandwidth numbers of the SATA host bus adapter core with both a traditional Winchester style Hard Disk (160GB Western Digital Caviar Blue) as well as flash-based Solid State Drives (64GB OCZ Agility 2). This gives us an indication about the overhead induced by the SATA core itself and

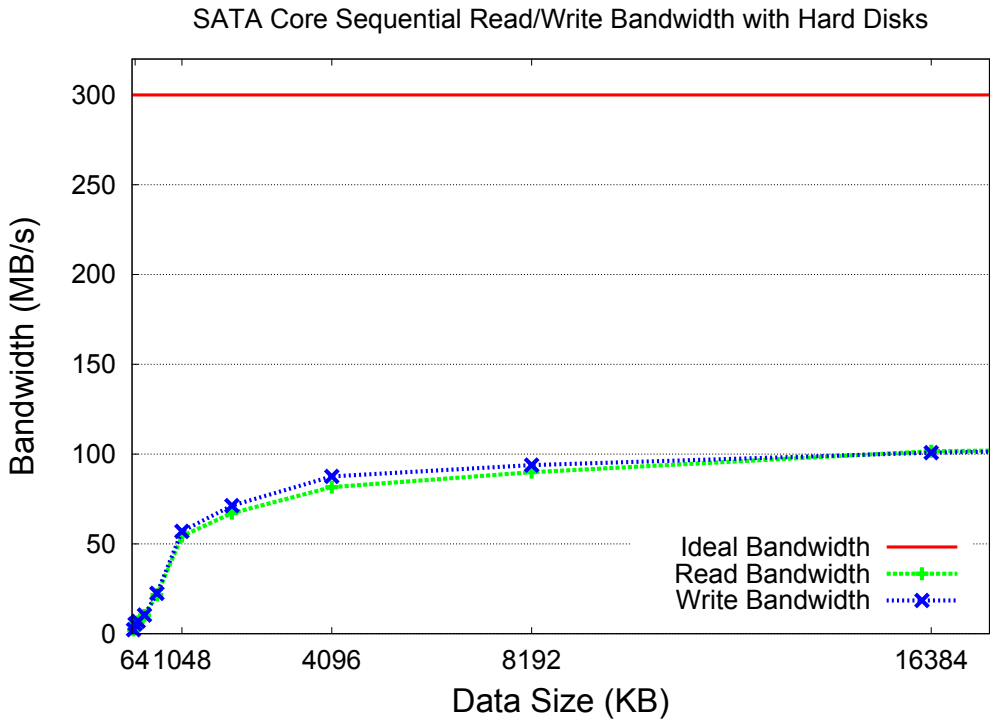


Figure 4.6: SATA Sequential Read/Write Bandwidth with Hard Disk

the peak bandwidth capabilities of the storage devices and the SATA protocol. Setup 2 (4.1.2) was used and counters were introduced in the HDL source code to collect the performance numbers. The SATA core is running at a 75 MHz clock with a 32-bit data path which gives a theoretical bandwidth of 300 MB/s (SATA Generation 2 limit).

Figure 4.6 shows the bandwith of the SATA core with traditional hard disks. A maximum read/write bandwidth of $\approx$ 107 MB/s (Figure 4.6) is obtained for sequential data transfer sizes of 32 MB and beyond. In this case, the peak bandwidth capabilities of the SATA2 protocol (300 MB/s) are underutilized. This is due to the limitation of mechanical disk drives.

Solid state drives on the other hand have no moving parts and can achieve reduced latencies and increased bandwidth. The SATA Gen 2 OCZ SSDs have a rated peak performance of 285 MB/s for reads and 275 MB/s for writes. Faster SSDs compatible with the 6 Gb/s (600 MB/s) SATA Gen 3 protocol are available but current

Xilinx devices (up to Virtex 6) only support the SATA2 protocol (300 MB/s) on the transceivers. Hence the OCZ SSDs are currently sufficient for our tests. Figure 4.7 shows the bandwidth of the SATA core with SSDs for transfer sizes between 4KB to 8MB (we tested upto 1GB in practice). With the low latency of SSDs, the core achieves performance close to the rated peak at 1024 KB. The performance for writes is better than reads for transfer sizes between 4KB-512KB. There are three reasons for this. Firstly, although flash memory has a limitation of erase-before-write, SSDs employ a sophisticated flash translation layer (FTL). FTLs uses log structured approaches where a data is only appended to a clean block for write requests with garbage collection in the background (akin to automatic defragmentation on idle disks) [49, 50]. This avoids degrading the write performance on SSDs. Secondly, most SSDs also employ RAM caches to buffer data which improves the write latency [51]. Also, the flash controller on the OCZ SSD used in these tests adopts a write compression technique [52], which reduces the amount of data written to the flash. The performance saturates with a peak of 279 MB/s for reads and 276 MB/s for writes beyond 8 MB transfer sizes. This shows that the SATA core by itself induces minimum overhead and efficiently utlilizes the available disk and SATA channel bandwidth.

Next, we tested the random read/write performance of the core and the benefits of using NCQ on an SSD. The basic unit of read/write operations on a flash SSD is a 4KB page (compared to a 512 byte sector on a hard disk). Also, a filesystem's standard block size is 4KB. Thus, 4KB random read/write results are an important metric for getting the worst case performance of an SSD. One must note here that the SATA protocol which was originally intended for traditional hard disks still uses sector addresses which refer to a 512 byte region. If the sector requests are not aligned to the page boundaries of a flash device, the SSD device processor ends up reading/writing across multiple flash pages which reduces performance. In our tests, the sector offsets were aligned to the page boundaries. To exploit full potential of

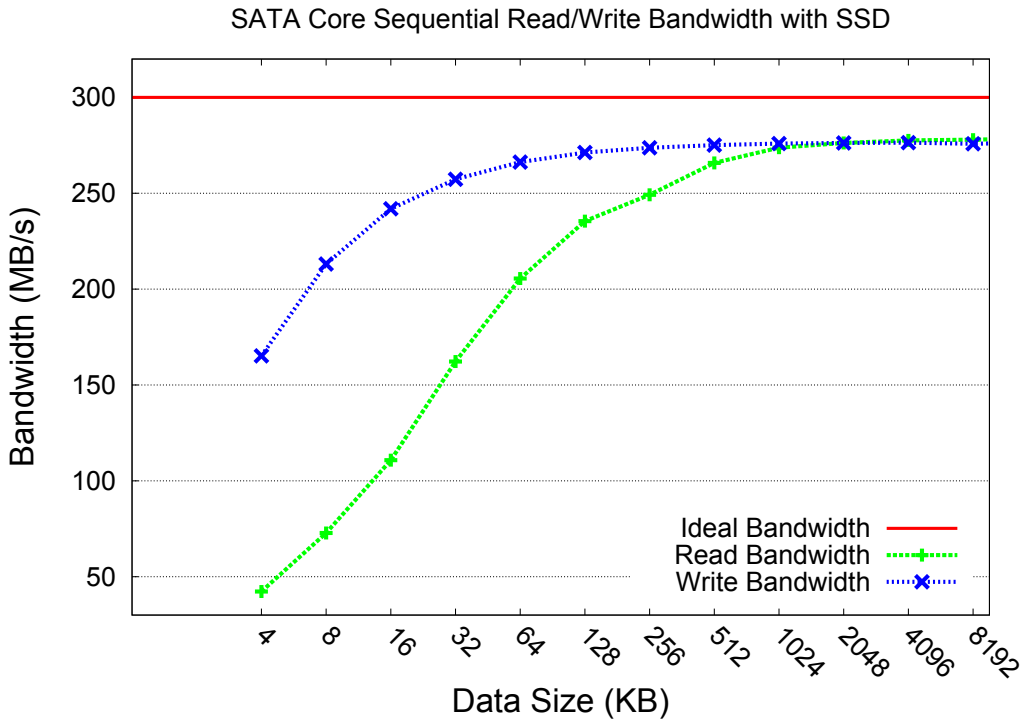


Figure 4.7: SATA Sequential Read/Write Bandwidth with SSD

NCQ, we also ensured that the SATA core had enough commands in its queue to issue to the drive.

We use 4K IOPS as a metric to report random read/write performance of the SATA core and SSD. This specifies how many 4 KB operations the drive could handle per second with each block (page) being read or written to a random position.

$$\text{IOPS} = \frac{\text{bytes per second}}{\text{data size in bytes}}$$

Figure 4.8 shows random read and write IOPS vs queue depth. The read IOPS scales almost linearly with the queue depth. For 4KB random writes tested with the OCZ SSD, it was observed that the device would send DMA Setup FISs (indicating that it is ready to receive data from host) after receiving the first three commands from the host. The SATA core then has to send the data associated with the currently enqueued commands before sending the 4th command request. Hence the random

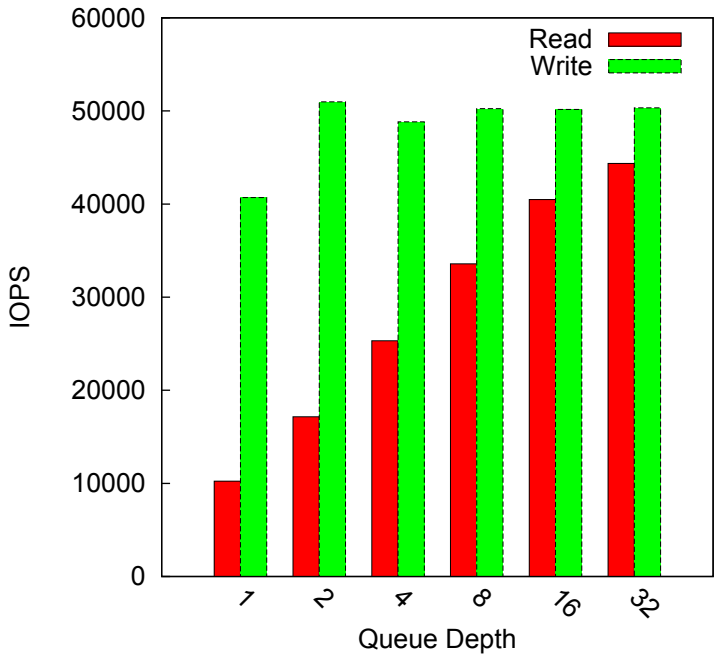


Figure 4.8: SATA 4K Random Read/Write IOPS

write IOPS saturates beyond a queue depth of four. The OCZ Agility 2 60 GB SSD has a rated 4KB random write performance of 10,000 IOPS, and the SandForce SF-1200 processor in these drives has 30,000 IOPS for 4KB reads and writes, but our SATA core exceeds both these published specs reaching a maximum of 50,337 IOPS for writes and 44,366 IOPS for reads at a queue depth of 32 .

4.2.1.3 HWFS-SATA Bandwidth

The HWFS-SATA subsystem was tested with filesystem block sizes ranging from 4KB to 16 KB. The SATA Core was configured with support for Native Command Queueing. Due to the limited performance achieved by the SATA Core itself with mechanical hard disks (Subsection 4.2.1.2), we have used SSDs to report all measurements in this subsection using Setup 3 (4.1.2).

Sequential Workload For sequential workload evaluation of the HWFS-SATA system, file sizes of 1GB were used. The execution time for reading and writing files were measured to calculate Bandwidth. (The time to open and close a file, which includes operations such as reading and writing the Superblock, has been taken into

account in these measurements). With a block size of 4 KB and no queuing support on SATA core (QD1), peak throughputs of 72 MB/s for Read and 96 MB/s for Writes are obtained at a file size of 1 MB as shown in Figure 4.9. From previous results shown in Figure 4.7, it was observed that for a 1 MB data transfer size the raw performance of the SATA Core is close to the rated peak of the device. With the HWFS-SATA system, although the file size is 1MB, the HWFS issues request in units of 4KB blocks. Also, even though the HWFS is capable of issuing multiple requests, the SATA core services one request at a time. Due to these two factors, the SATA channel bandwidth is underutilized. Increasing the queue depth, allows the SATA core to process and service multiple requests from HWFS. These requests are in turn enqueued and issued by the SATA core to the disk side controller which allows for efficient utlization of the single SATA channel. With 32 commands enqueued, this system provides a read bandwidth of 170 MB/s and write bandwdith of 181 MB/s. As described in Subsection 4.2.1.2), the write performance scales and saturates quicker than the read performance due to the SSD controller and device characteristics. (Similar trend is observed with the raw performance of SATA core system).

Figure 4.10 and Figure 4.11 show the performance improvements obtained by increasing the filesystem block sizes to 8KB and 16 KB along with queueing support from the SATA core. Bigger block sizes increases the data transfer rate of the system. For the base case without queueing, the read/write performance saturates at 112 MB/s and 148 MB/s respectively with 8KB blocks. With 16 KB blocks, this increases to 152 MB/s for reads and 190 MB/s for writes. With a best case configuration of 16 commands enqueued and a block size of 16 KB, the bandwidth of the system reaches a peak of 241 MB/s for reads and 219 MB/s on writes. This amounts to 85% of the peak read and 80% of the peak write performance of the SSD. Although the standard block size for most software filesystems is 4KB most HPC filesystems support bigger blocks. Hence using 16 KB block sizes is reasonable, given the performance

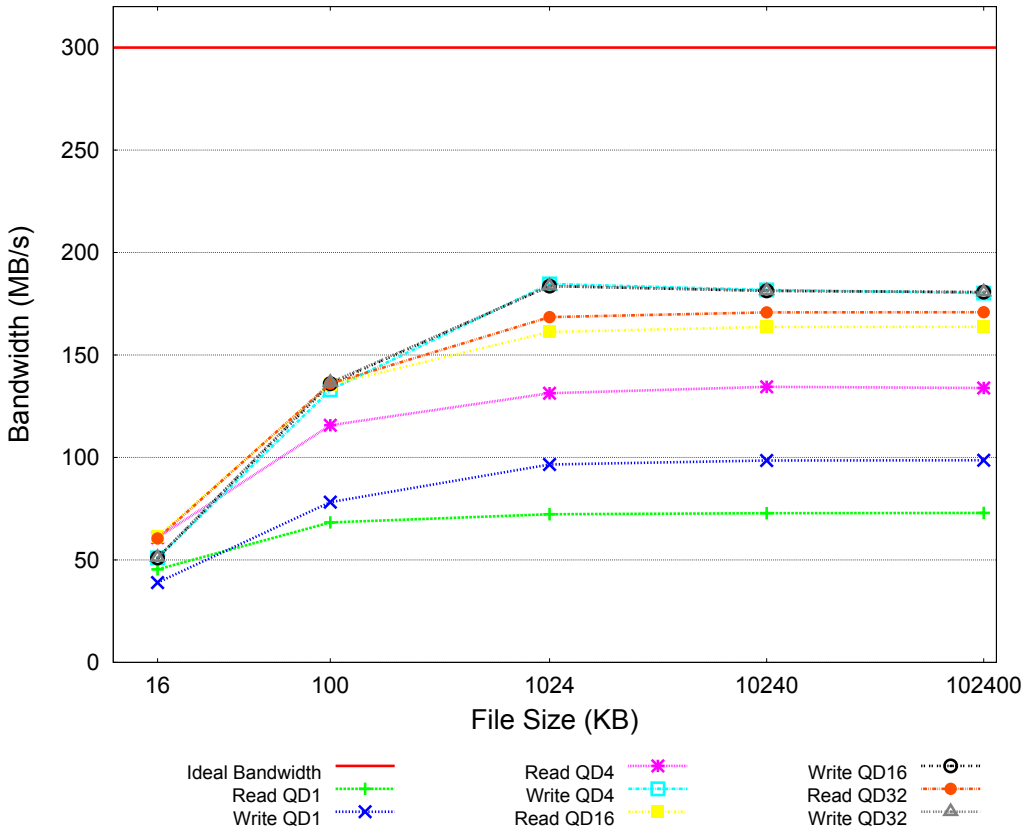


Figure 4.9: HWFS-SATA Sequential Read/Write Bandwidth, 4 KB Blocks

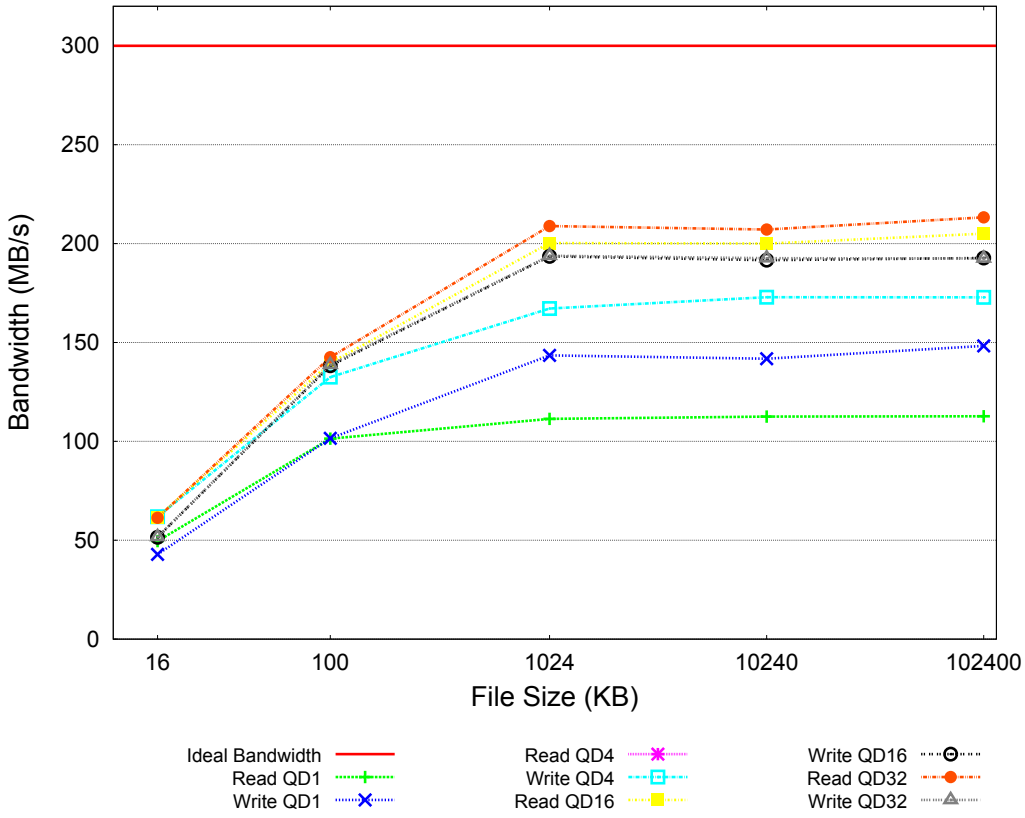


Figure 4.10: HWFS-SATA Sequential Read/Write Bandwidth, 8 KB Blocks

improvements obtained. In Subsection 4.2.1.2), the additional FPGA resource cost associated with supporting bigger block sizes is shown.

RANDOM WORKLOAD The workload generator core was then configured for random read/writes. The random workloads use request sizes ranging from 4KB to 128 KB in a 1 GB file seeking to random offsets after each request. Figure 4.12 depicts the random read and random write bandwidth with varying request sizes for 4KB filesystem blocks.

Unlike hard disks, SSDs do not suffer from expensive disk head seeks and have almost uniform random access latencies. To take full advantage of flash memory's fast random access performance it was improtant to minimize filesystem's seek overhead. The design enhancement to HWFS for the *Seek* operation described in Section 3.1 clearly benefits the system. For a 4KB block size, the HWFS Inode block holds 255

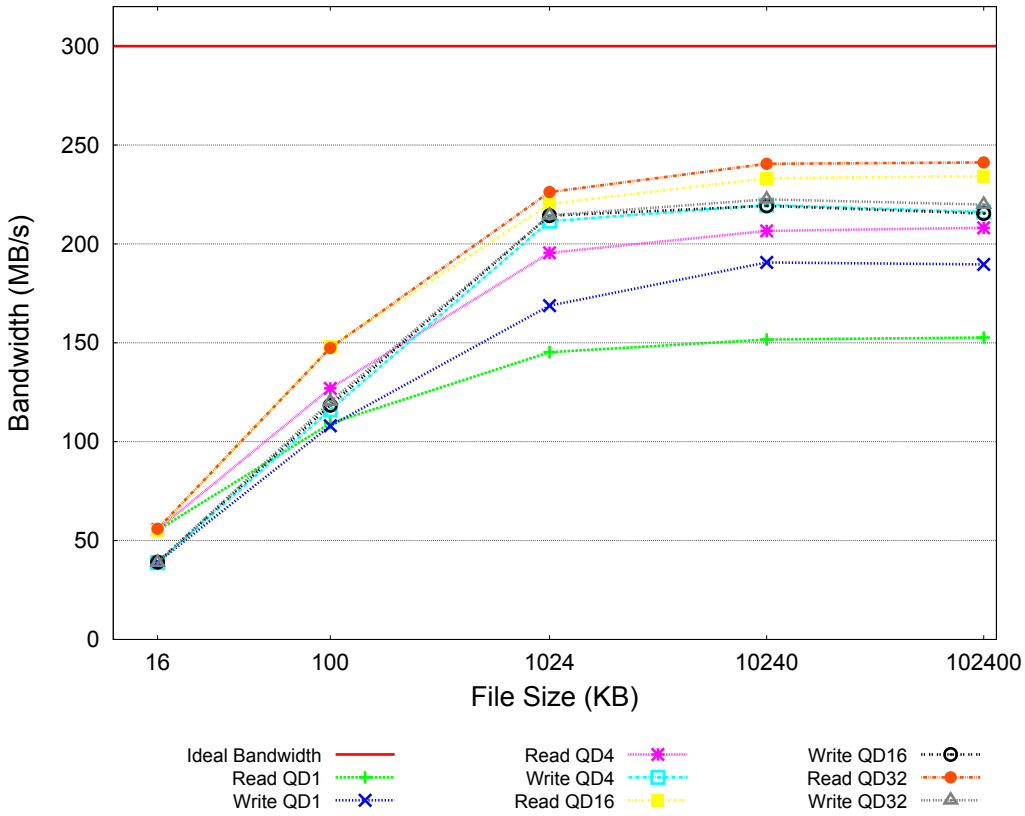


Figure 4.11: HWFS-SATA Sequential Read/Write Bandwidth, 16 KB Blocks

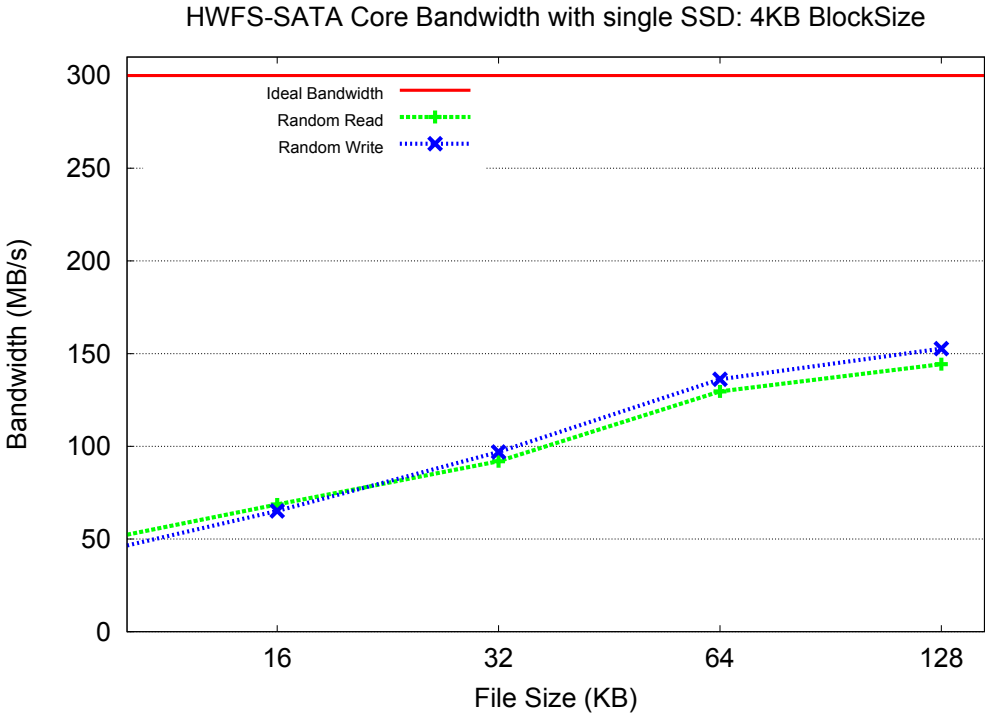


Figure 4.12: HWFS-SATA Random Read/Write Bandwidth

direct data block pointers (and a single indirect pointer), allowing it a seek space of 4 MB per Inode block. With an additional fetch of a file's double indirect Inode block (which holds a list of all single indirect pointers of a file) from disk, HWFS can seek anywhere within a 1GB file space with a single level of indirection. The random read/write performance of the system comes close to the sequential workload performance for small files with the only additional overhead of calculating block offsets from byte offsets.

4.2.1.4 Comparison with a Software Filesystem

In order to compare the Hardware Filesystem's performance to a Software Filesystem running on a CPU, Setup 4 (4.1.3) was used. We first used the Linux utility *dd* on the raw block device (no filesystem) with a request size of 1024KB over a 1 GB transfer and obtained a peak write performance of 174 MB/s and a peak read performance of 197 MB/s. For the same request size, the SATA core achieves performance close to the SSD's rated peak (4.2.1.2). This suggests that the Operating System

introduces overhead even during raw block access. Next, the SSD block device was formatted with the Linux Ext2 filesystem. A benchmarking tool, `fio` [53] was run on this system for performance measurements with Ext2. This has the ability to use the Linux asynchronous I/O engine `libaio`, spawn multiple processes for I/O transfers and generate sequential and random workloads. We used fio's `direct` flag to bypass the operating system's buffer cache and noop scheduler on the block device.

Table 4.4 provides a bandwidth comparison of the Ext2-CPU-SSD system to our HWFS-SATA-SSD solution. For sequential workloads, both systems performed 1 GB file reads and writes. HWFS was configured with a 16 KB block size with a maximum queue depth of 32 on the SATA core. HWFS performs better than the CPU on both sequential reads and writes despite running at a much slower clock frequency. The performance advantage is significant for writes (more than 2x). For random workloads, a 64 KB request size was used on both systems seeking over a 1 GB file. Again, HWFS outperforms the CPU on random reads and writes. The SATA protocol provides support for Native Command Queueing which has the potential of benefitting SSDs through concurrent accesses to flash chips. HWFS with a tightly coupled SATA core makes optimum use of this concurrency, whereas the latency of the Operating System's software stack prevents exploiting the full potential of SSDs.

We also used Setup 2 to measure the SATA Core's performance with an Ext2 filesystem running on Microblaze. A sequential read and write bandwidth of 22.15 MB/s and 13.41 MB/s was obtained. Such a significant underutilization compared to the HWFS-SATA system can be attributed to the low frequency of MicroBlaze (100 MHz) as well as the overhead of the Linux software stack.

The positive results obtained from the experiments conducted successfully answers the following thesis question: *If the filesystem component is migrated into hardware, will this give performance improvements over a software fileystem for solid state drives?*

Workload	**Microblaze-Ext2** 100 MHz	**Opteron-Ext2** 2.1 GHz	**HWFS** 75 MHz	**HWFS Speedup vs**	
				Microblaze	**Opteron**
Seq Read (MB/s)	22.15	187.62	241.20	10.89×	1.29×
Seq Write (MB/s)	13.41	102.16	219.85	16.39×	2.15×
Rnd Read (MB/s)	11.37	88.19	129.60	11.40×	1.47×
Rnd Write (MB/s)	8.56	64.62	136.18	15.90×	2.10×

Table 4.4: HWFS-SATA vs Ext2 on CPU : Speedup

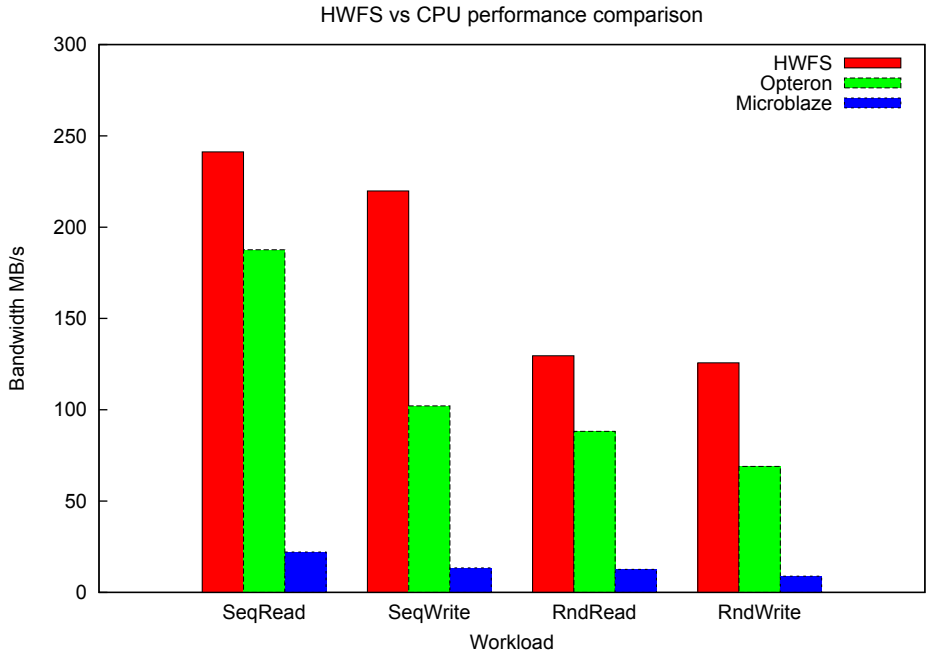


Figure 4.13: HWFS-SATA vs CPU: Bandwidth

4.2.2 Scalability with Multiple Disks using RAID

In order to check the performance scalability of HWFS with muliple SSDs, we measured bandwidth of the HWFS system with a RAID controller and two SATA cores connected to two SSDs. In this system, both the SATA cores run at 75 MHz, each with 32-bit data paths. HWFS runs at 150 MHz with a 32-bit data path which gives the system a theoretical peak of 600 MB/s. The RAID controller core uses FIFOs for crossing between the two clock domains.

Figure 4.14, shows the performance comparison of the two SSD system with a single SSD system. HWFS has been configured with 4 KB, 8 KB and 16 KB block sizes and a maximum queue depth of 32 for these tests. The bandwidth of the system scales almost linearly with two SSDs reaching a peak of 430.56 MB/s on sequential reads (1.78x over 1 Disk System) and 383.29 MB/s (1.73x over 1 Disk System) on sequential writes for 16 KB blocksizes.

Although the ML605 board has 8 transceivers available for SATA on its High Pin Count (HPC) interface, the FMC connector that we used provides access to only two. Hence, the tests were limited to two disks. Even if the infrastructure was available for building a system with four disks, scaling the system performance beyond the theoretical peak of 600 MB/s would involve clocking HWFS at a higher frequency (more than 150 MHz). The HWFS-RAID performance with multiple disks is limited by the clock frequency of HWFS. Hence, this investigation concludes that the performance of the system scales upto two SSDs.

These results successfully answer the following thesis questions: *Can the filesystem core support multiple disks? Will the performance of the core scale with multiple disks?*

4.2.3 Size

The HWFS, SATA and interfacing logic were synthesized using the Xilinx Synthesis Tool (XST) available in the Xilinx ISE design suite, version 12.2, for the target device XC6VLX240T from the Virtex-6 family. This FPGA has 150,720 LUTs,

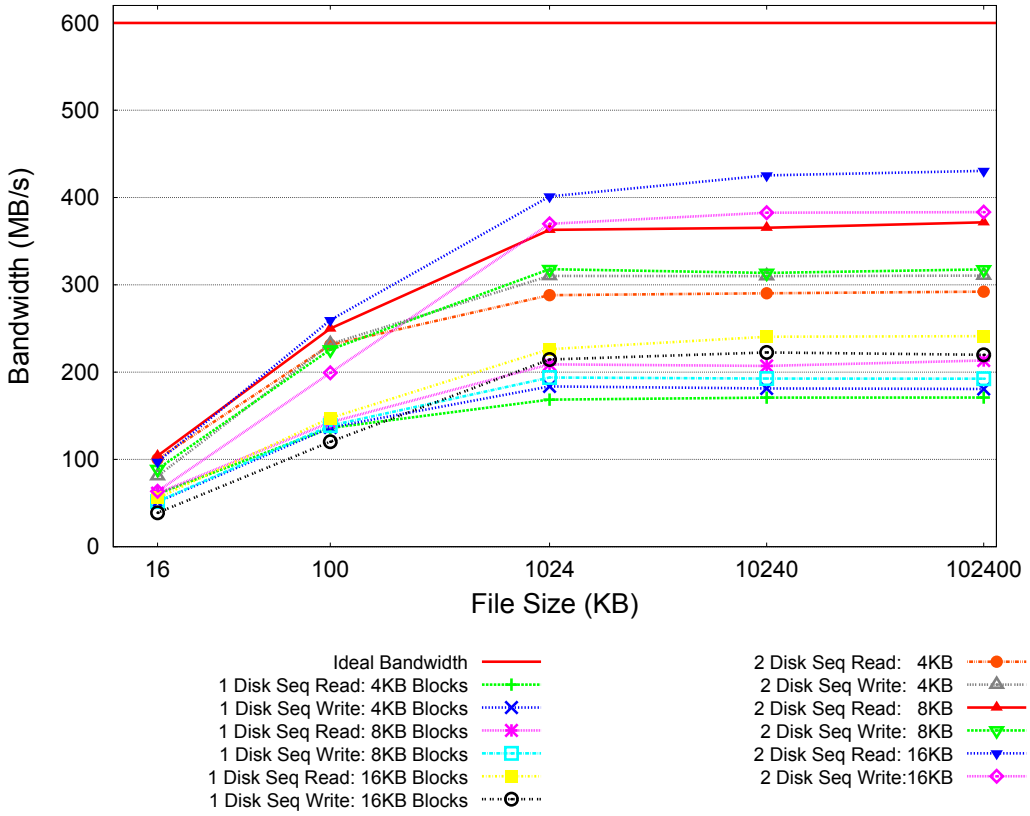


Figure 4.14: HWFS-SATA Bandwidth with two SSDs

Table 4.5: HWFS resource utilization, synthesized for XC6VLX240T

Block Size	**LUTs**	**F/Fs**	**BRAMs**
4 KB	2587	636	6
8 KB	2554	619	9
16 KB	2601	623	15

301,440 Flip/Flops, 416 BRAMs and 20 transceivers.

Table 4.5 shows the resource utilization statistics of HWFS for different block sizes. Since the on-chip metadata buffers for super block, inode block, freelist block and double indirect inode block are mapped onto BRAMs, the logic utilization (LUT) is independent of the block size. A slight variation in LUT count is observed due to the BRAM buffer's address width variations and the synthesis tool's speed optimization efforts.

Using a block size of 16 KB with queueing support gives optimum performance for sequential workloads in our system. This entails an increase in the BRAM usage but it is still within 4% of the devices BRAMs. Additionally, we have restricted our superblock which holds filename-inode mappings to 4 KB making it independent of the block size. This would support 240 filename-inode pairs, sufficient for a small number of large files.

Table 4.6 shows the size of HWFS, SATA and interfacing logic for a 16KB Block Size. The total logic utilization of the system is a modest 2.8%. The SATA core's FIS buffers and frame transmit-receive FIFOs are mapped onto the dedicated on-chip Block RAMs occupying three BRAMs. The SATA interfacing logic uses an additional two BRAMs for buffering and flow control between HWFS and SATA cores. The total BRAM usage for the system is still less than 5% of the device. A single transceiver is used for the physical layer of SATA. To provide a frame of reference for these results, the microblaze embedded processor running at 100 MHz takes up 4600 LUTs (and 3541 F/Fs) which is more than the LUT count of HWFS, SATA and interfacing logic combined. A high end processor would consume even more resources.

Table 4.6: HWFS-SATA resource utilization, synthesized for XC6VLX240T

Resources	**HWFS**	**SATA**	**SATA IF**	**HWFS+SATA+SATA IF**	**% Device**
LUTs	2601	1,334	290	4225	2.8%
F/Fs	623	894	328	1845	0.6%
BRAMs	15	3	2	20	4.8%
MGT	0	1	0	1	5%

The small size of the HWFS-SATA system validates our thesis metric of resource efficiency and answers the following thesis question positively: *Can the improvements in performance justify the cost of extra on-chip resources dedicated to the Hardware Filesystem and SATA disk controller cores?*

CHAPTER 5: CONCLUSION

Faster non-volatile memories like Solid State Drives promise I/O performance improvements. However, the traditional operating system stack prevents exploiting their full potential. In order to answer the thesis question 'Will migrating the filesystem into a dedicated hardware core improve performance over software approaches?', a Hardware Filesystem (HWFS) was designed and implemented on an FPGA with support for the fundamental filesystem operations: open, read, write, delete and seek. To minimize risk, a RAM Disk was used initially for testing the filesystem and measuring efficiency. Subsequently, support for split transactions and striping across multiple disks was added to HWFS. To measure performance with SSDs, a SATA Host Bus Adapter core was designed with the ability to directly interface with other IP cores. This has been realeased as an open source project.

The HWFS-SATA system was evaluated with sequential and random workloads to measure bandwidth. HWFS makes optimum use of the SATA core's support for Native Command Queueing achieving 85% of peak read and 80% peak write performance of the SSD for sequential workloads. Comparison with Linux's Ext2 filesystem reveals that HWFS provides a performance improvement of 10.89× for sequential reads, 16.39× for sequential writes, 11.4× for random reads and 15.9× for random writes over a microblaze processor running at 100 MHz. Compared to a 2.1 GHz AMD Opteron CPU with Ext2, HWFS achieves a speedup of 1.29× for sequential reads, 2.15× for sequential writes, 1.47× for random reads and 2.1× for random writes. The system is also resource efficient consuming less than 3% of logic and 5% of the Block RAMs of a Xilinx Virtex-6 chip.

In short, the Hardware Filesystem and supporting infrastructure provides low la-

tency, high bandwidth access to fast non-volatile storage and performs significantly better over traditional software filesytems. Moreover, with current non-volatile storage bandwidth trends and the relatively flat gains in processor frequency, it seems the overhead lost to software will likely increase over time with traditional approaches.

REFERENCES

[1] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications," in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '09. New York, NY, USA: ACM, 2009, pp. 217–228. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508270

[2] I. Koltsidas and S. D. Viglas, "Flashing up the storage layer," *Proc. VLDB Endow.*, vol. 1, pp. 514–525, August 2008. [Online]. Available: http://dx.doi.org/10.1145/1453856.1453913

[3] D. Roberts, T. Kgil, and T. Mudge, "Integrating nand flash devices onto servers," *Commun. ACM*, vol. 52, pp. 98–103, April 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498791

[4] G. Burr, M. Breitwisch, M. Franceschini, D. Garetto, K. Kailash Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. Lastras, A. Padilla, B. Rajendran, S. Raoux, and R. Shenoy, "Phase change memory technology," *Journal of Vacuum Science and Technology B*, vol. 28, no. 2, pp. 223–262, March 2010.

[5] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano, "A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram," in *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*, dec. 2005, pp. 459 –462.

[6] R. F. Freitas and W. W. Wilcke, "Storage-class memory: the next storage system technology," *IBM J. Res. Dev.*, vol. 52, pp. 439–447, July 2008. [Online]. Available: http://dx.doi.org/10.1147/rd.524.0439

[7] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM J. Res. Dev.*, vol. 52, pp. 449–464, July 2008. [Online]. Available: http://dx.doi.org/10.1147/rd.524.0449

[8] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 385–395. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2010.33

[9] Tilera, "Tilepro64 processor," uRL: http://www.tilera.com/products/processors/TILEPRO64.

[10] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson, "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, feb. 2010, pp. 108 –109.

[11] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *Micro, IEEE*, vol. 27, no. 5, pp. 15 –31, sept.-oct. 2007.

[12] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 29–44. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629579

[13] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 76–85, April 2009. [Online]. Available: http://doi.acm.org/10.1145/1531793.1531805

[14] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: an operating system for many cores," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 43–57. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855745

[15] A. Belay, D. Wentzlaff, and A. Agarwal, "Vote the os off your core," Computer Science and Artificial Intelligence Lab (CSAIL), MIT, Tech. Rep. MIT-CSAIL-TR-2011-035, 2011. [Online]. Available: http://dspace.mit.edu/handle/1721.1/64977

[16] D. Nellans, R. Balasubramonian, and E. Brunv, "A case for increased operating system support in chip multiprocessors," in *Proceedings of the 2nd IBM Watson Conference on Interaction between Architecture, Circuits and Compilers*, 2005.

[17] J. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar, "Using asymmetric single-isa cmps to save energy on operating systems," *Micro, IEEE*, vol. 28, no. 3, pp. 26 –41, may-june 2008.

[18] R. Sass, W. V. Kritikos, A. G. Schmidt, S. Beeravolu, P. Beeraka, K. Datta, D. Andrews, R. Miller, and D. S. Jr., "Reconfigurable computing cluster (rcc)

project: Investigating the feasibility of FPGA-based petascale computing," in *FCCM '07: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'07)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 127–138.

[19] C. Pedraza, E. Castillo, J. Castillo, C. Camarero, J. L. Bosque, J. I. Martinez, and R. Menendez, "Cluster Architecture Based on Low Cost Reconfigurable Hardware," in *Proceedings of the 2008 International Symposium on Field Programmable Logic and Applications*, Sep 2008.

[20] M. Saldana, E. Ramalho, and P. Chow, "A Message-Passing Hardware/Software Co-simulation Environment to Aid in Reconfigurable Computing Design Using TMD-MPI," in *Proceedings of the 2008 International Conference on Reconfigurable Computing and FPGAs*, Dec 2008.

[21] S. Datta *et al.*, "RCBLASTn: Implementation and evaluation of the BLASTn scan function," in *FCCM 2009*, 2009.

[22] S. Datta and R. Sass, "Scalability studies of the blastn scan and ungapped extension functions," in *ReConFig 2009*, 2009.

[23] A. Nataraj, A. Morris, A. D. Malony, M. Sottile, and P. Beckman, "The ghost in the machine: observing the effects of kernel operation on parallel application performance," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007.

[24] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[25] M. J. Bach, *The Design of the UNIX Operating System*. Prentice Hall, September 1991.

[26] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating system implications of fast, cheap, non-volatile memory," in *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, ser. HotOS'13. Berkeley, CA, USA: USENIX Association, 2011, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1991596.1991599

[27] E. Seppanen, M. T. O'Keefe, and D. J. Lilja, "High Performance Solid State Storage Under Linux," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/MSST.2010.5496976

[28] H. K.-H. So and R. Brodersen, "File System Access from Reconfigurable FPGA Hardware Processes in BORPH," in *Proceedings of the 2008 International Symposium on Field Programmable Logic and Applications*, Sep 2008.

[29] D. Lavenier, S. Guyetant, S. Derrien, and S. Rubini, "A reconfigurable parallel disk system for filtering genomic banks," in *Proceedings of 2003 International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2003.

[30] N. Abbani, A. Ali, D. Al Otoom, M. Jomaa, M. Sharafeddine, H. Artail, H. Akkary, M. Saghir, M. Awad, and H. Hajj, "A Distributed Reconfigurable Active SSD Platform for Data Intensive Applications," in *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, sept. 2011, pp. 25 –34.

[31] T. Li, M. Huang, T. El-Ghazawi, and H. H. Huang, "Reconfigurable Active Drive: An FPGA Accelerated Storage Architecture for Data-Intensive Applications," in *Symposium on Application Accelerators in High Performance Computing*, July 2009.

[32] Netezza, "The Netezza Data Appliance Architecture," www.redbooks.ibm.com/redpapers/pdfs/redp4725.pdf.

[33] Bluearc, "The bluearc file system technology," uRL: http://www.hds.com/bluearc/.

[34] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-performance, reliable secondary storage," *ACM Computing Surveys*, vol. 26, no. 2, pp. 145–185, 1994.

[35] A. A. Mendon, "Design and Implementation of a Hardware Filesystem," Master's thesis, University of North Carolina at Charlotte, Aug. 2008.

[36] A. A. Mendon and R. Sass, "A hardware filesystem implementation for high-speed secondary storage," in *2008 IEEE International Conference on Reconfigurable Computing and FPGA's*, 2008.

[37] Xilinx, Inc., "ML410 embedded development platform user guide," September 2008.

[38] DGWAY, "Serial ata ip core," uRL: http://www.dgway.com/products/IP/SATA-IP/index2-E.html.

[39] Intelliprop, "RTL Storage Interface Cores," http://www.intelliprop.com/products-storage_interface_cores.htm.

[40] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (raid)," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 1988, pp. 109–116.

[41] A. G. Schmidt, W. V. Kritikos, R. R. Sharma, and R. Sass, "Airen: A novel integration of on-chip and off-chip fpga networks," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, vol. 0, pp. 271–274, 2009.

[42] A. G. Schmidt, S. Datta, A. A. Mendon, and R. Sass, "Productively Scaling I/O Bound Streaming Applications with a Cluster of FPGAs," July 2010.

[43] ——, "Investigation into scaling I/O bound streaming applications productively with an all-FPGA cluster," *International Journal on Parallel Computing*, Dec 2011.

[44] Xilinx, "Virtex-5 fpga rocketio gtp transceiver user guide," uRL: www.xilinx.com/support/documentation/user_guides/ug196.pdf.

[45] ——, "Virtex-6 fpga rocketio gtx transceiver user guide," uRL: www.xilinx.com/support/documentation/user_guides/ug366.pdf.

[46] Matt DiPaolo and Simon Tam, "Serial ata physical link initialization with the gtp transceiver of virtex-5 lxt fpgas," uRL: http://www.xilinx.com/support/documentation/anstorage.htm.

[47] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly Media, 2005.

[48] Xilinx, "FMC XM104 Connectivity Card," http://www.xilinx.com/products/boards-and-kits/HW-FMC-XM104-G.htm.

[49] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ser. ATC'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 57–70. [Online]. Available: http://dl.acm.org/citation.cfm?id=1404014.1404019

[50] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '09. New York, NY, USA: ACM, 2009, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508271

[51] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '09. New York, NY, USA: ACM, 2009, pp. 181–192. [Online]. Available: http://doi.acm.org/10.1145/1555349.1555371

[52] Jeffrey B. Layton, "On-the-fly Data Compression for SSDs," http://www.linux-mag.com/id/7869/.

[53] Jan Axboe, "Fio-flexible IO tester," http://freecode.com/projects/fio.