Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 572860, 13 pages
doi:10.1155/2009/572860

Research Article

A Hardware Filesystem Implementation with Multidisk Support

Ashwin A. Mendon, Andrew G. Schmidt, and Ron Sass

Reconfigurable Computing Systems Lab, University of North Carolina at Charlotte, 9201 University City Blvd., Charlotte, NC 28223-0001, USA

Correspondence should be addressed to Ashwin A. Mendon, aamendon@uncc.edu

Received 16 March 2009; Accepted 4 August 2009

Recommended by Cesar Torres

Modern High-End Computing systems frequently include FPGAs as compute accelerators. These programmable logic devices now support disk controller IP cores which offer the ability to introduce new, innovative functionalities that, previously, were not practical. This article describes one such innovation: a filesystem implemented in hardware. This has the potential of improving the performance of data-intensive applications by connecting secondary storage directly to FPGA compute accelerators. To test the feasibility of this idea, a Hardware Filesystem was designed with four basic operations (open, read, write, and delete). Furthermore, multi-disk and RAID-0 (striping) support has been implemented as an option in the filesystem. A RAM Disk core was created to emulate a SATA disk drive so results on running FPGA systems could be readily measured. By varying the block size from 64 to 4096 bytes, it was found that 1024 bytes gave the best performance while using a very modest 7% of a Xilinx XC4VFX60's slices and only four (of the 232) BRAM blocks available.

1. Introduction

Many FPGA devices available today are rich in special-purpose blocks. These fixed-function cores, implemented on the die, add capabilities to the ever-growing capacity of these devices. The Virtex-4 FX devices, for example, in addition to the conventional programmable logic and flip-flops, include processors, Block RAMs, DSP48 cores, and high-speed serial transceivers. The Multigigabit Transceiver (MGT) cores 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.

These advances enable a designer to implement highly integrated computing systems. For example, it is feasible to integrate video, networking interfaces, disk controllers [1], and other conventional peripherals onto a single Platform FPGA device 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 [2–4]. If successful, there is an enormous potential for reducing the size, weight, and cost while increasing the scalability of parallel machines.

Within the context of parallel computing, the integration of disk drives is especially interesting because of its potential to speed up data-intensive parallel applications. In particular, out-of-core applications (MPI-IO) needing tightly integrated secondary storage or streaming very large data sets would benefit. Tight integration, specifically, is important to these 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). It also allows the introduction of simple striped multidisk 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.

Filesystems are typically implemented in *software* as part of the operating system. This paper describes the implementation of a *hardware* filesystem. Figure 1 illustrates this concept. Figure 1(a) is the traditional organization with the filesystem and device driver implemented in software, Figure 1(b) is the filesystem migrated into hardware. The simplest filesystems organize the sequential fixed-size disk sectors into a collection of variable-sized *files*. Of course,

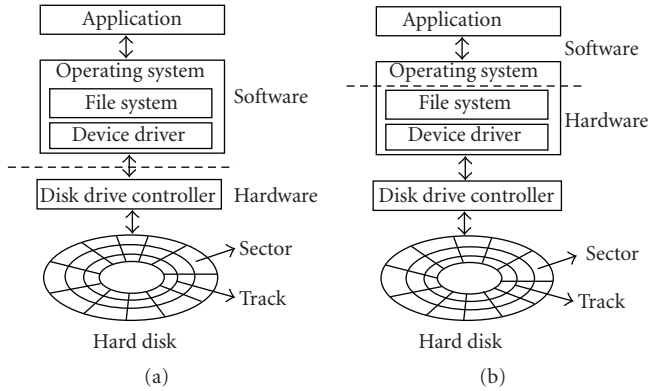

Figure 1: (a) Traditional filesystem implementation. (b) Filesystem migrated into programmable logic.

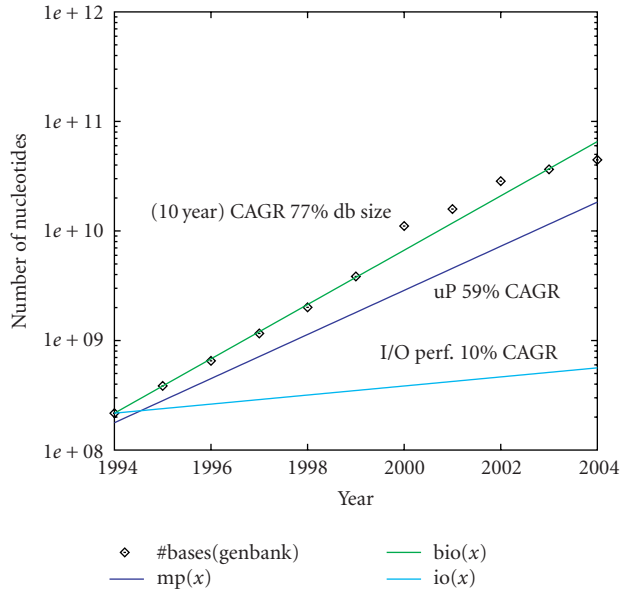

Figure 2: Compound annual growth rate of problem size, single processor performance, and I/O subsystem performance.

most modern filesystems are much more complex and also include a large amount of metainformation 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 incorporate new features in hardware.

For some scientific applications, these features are extremely valuable. For example, in some cases, the resolution of experiment or simulation is limited by the main memory available to store the data structures. In order to increase the detail of the simulation, computational scientists are forced to code their algorithms so that data are explicitly moved between secondary storage and main memory. (These so-called out-of-core applications are far more efficient than simply relying on the OS to swap memory to secondary storage.)

Alternatively, if part of the computation is performed by accelerators implemented in the programmable logic of an FPGA, then the data do not necessarily have to go through all of the traditional layers of an OS (device driver, filesystem interface) just to have the application then forward it to the core. Instead, the core can simply open the file and access it directly. This frees the processor from handling I/O requests and avoids the use of off-chip memory bandwidth to buffer disk data as in other approaches [5]. It also reduces the number of interrupts which has been shown to negatively impact very large parallel systems [6, 7]. Finally, by migrating part of the filesystem operations to hardware, it becomes feasible to handle remote disk access directly in hardware. (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 a third case, some applications are facing datasets that are growing faster than computer speeds. Consider processor speeds and the size of bioinformatic databases. Suppose that single processor performance continues to double every 18 months and biological databases are growing even faster. Figure 2 shows both growth rates of between 1994 and 2004 on a semilog graph. The nucleotide data points come from GenBank [8], a public collection of sequenced genomes. A line fitted to this data shows a compound annual growth rate of 77% (compared to the 59% annual growth rate of processors). Now consider the performance gains of I/O subsystems (disk and interface). Secondary storage is not keeping pace with processor speeds, let alone the growth rate of the biological databases. The most aggressive estimates [9] suggest a 10% compound annual growth rate in performance while others [10] suggest a more modest 6% growth rate. Regardless, the consequence is profound: the same question (e.g., *is this sequence similar to any known gene*?) will take longer and longer every year. In short, the problem size is growing so fast; the bottleneck is simply I/O bandwidth. A filesystem implemented in hardware will not directly address

this issue; however, it does offer a first step towards alleviating high I/O bandwidth needs.

Our approach to this investigation is broken down into three steps: first, a software implementation as a basic proof of concept of how the filesystem in hardware will operate, second, a simulation of the hardware filesystem, and third, a synthesized implementation running on an FPGA.

The rest of this paper is organized as follows. The next Section is a background section on filesystems, specifically the Unix filesystem. In Section 3, the three designs (software, simulation, and synthesized) implementations are presented. Section 4, explains the testing methodology followed by the presentation and analysis of the results. The paper concludes with a brief summary and future directions.

2. Background

2.1. Disk Subsystem. 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 responsible for managing and organizing files on a nonvolatile storage medium, such as a Winchester-type disk drive (also known as a hard disk or hard drive). 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) [11]. 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 mutilevel 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 3.

Normally, the filesystem is designed to be independent of the disk controller. The disk controller is typically a device driver in an operating system that is responsible for communicating with the physical media and responds to block transfer commands from the filesystem. For expediency, the work here focuses on the most common, commodity drives available today: Serial ATA (SATA). SATA provides a 4-wire point-to-point configuration, supporting one device per controller connection. Each device has dedicated bandwidth and there are no master/slave configuration jumper issues as with parallel ATA drives. The pin count is reduced from 80 pins to 7 pins having 3 ground lines interspersed between 4 data lines to prevent crosstalk. Several FPGA devices include high-speed serial transceivers. For example, the Xilinx Virtex II, Virtex-4, and Virtex-5 device families have members that include multigigabit 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.

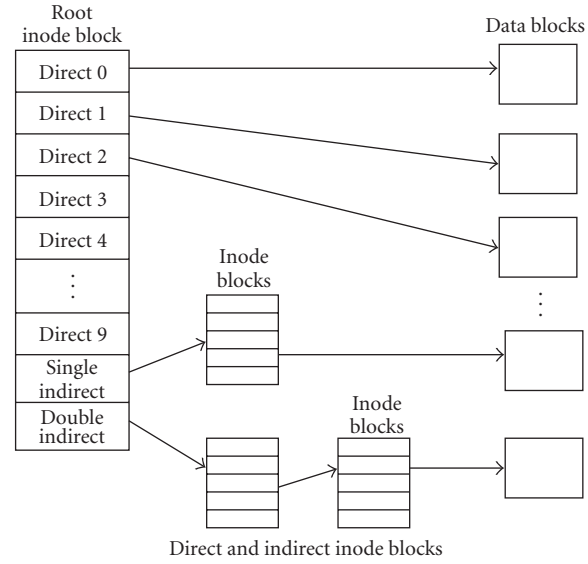

Figure 3: UNIX Inode structure.

2.2. Related Work. 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 [5] 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 here 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 [12] 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.

Using FPGAs to mitigate the I/O bandwidth bottleneck has been of interest commercially among server vendors such as Netezza [13] and Bluearc [14]. 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-based filesystem to speed up the I/O interface.

Finally, 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 [15] 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.

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 hardware simulations to judge the feasibility. This was reported in [16]. The work here describes a design that correctly synthesizes for an FPGA, includes a RAM Disk core to emulate the behavior of a disk controller, and has support for multiple disks in a RAID0 configuration. Below is a high-level description of the Hardware Filesystem followed by a description of the RAM Disk core and multidisk extensions.

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 computational accelerators. This has led to a number of differences. First, the Hardware Filesystem uses only direct and single indirect pointers in its inodes and the last indirect logical block points to another block of indirect pointers. 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 4. 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 subdirectories but the HWFS described here is flat.

A high-level block diagram of the HWFS is shown in Figure 5. It consists of a single large state machine, a buffer for the *Super Block*, a buffer for the currently open *Inode*, and a *Freelist* buffer. A compute accelerator presents commands and parameters to the core; the core is responsible for making disk controller block requests and delivering that data associated with a compute accelerator's request.

Specifically, the state machine implements the Open, Read, Write, and Delete file operations. The *operation* port is driven by the compute accelerator to select from the four main operations. The state machine asserts *request* and *new blknum* signals to issue a new block request to data from the disk controller. A *command* signal is used to distinguish between read and write block requests. The logical block numbers are issued from the *blocknum* port to address the appropriate memory location. The HWFS core waits on a *cmd ack* signal before issuing the next block request. On completion of a block transaction, the memory interface asserts the *blk xfer done* signal.

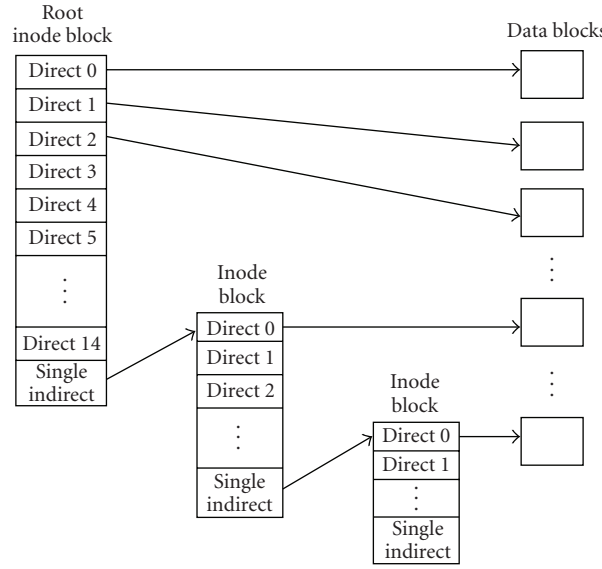

Figure 4: HWFS Inode Structure.

The HWFS core has an additional functional improvement over what was reported in [16]. 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. Subsection 3.2 discusses the multidisk controller and integration with the HWFS in more detail.

The rest of this subsection describes the various operations and the relevant states in the top-level state machine.

Open File. Open File takes in a file name along with an open command and returns the file's *Root Inode* location. The open command is required before issuing the first read or write command for a file. After the file is open, any number of reads and writes can be issued.

The state machine starts from the *Read Super Block* state and reads blocks 0–3 from *disk* into the *Super Block* buffer 4 bytes at a time. After reading 4 blocks, it transitions to the *Match Filename State* which starts a linear search for the 8 byte filename. The FSM sequences through the Super Block buffer, starting at the first file slot and reads the BRAM contents into a 64-bit equality comparator. If a match is found with the required filename, the comparator sends a *found* signal to the FSM which transitions to the *Find Inode* state. If the search is unsuccessful, the file does not exist in the filesystem Super Block and a *file not found* signal is asserted. In *Find Inode* state, the state machine captures the file's *Root Inode* location from the filename-inode mapping in the Super Block. Figure 6 depicts the open operation.

Read File. Once the file is opened, Read File uses the file's root inode block location to read in the file contents into a Read FIFO. Read File (Figure 7) begins with the *Read Inode* state. The *Root Inode* block of the file is first fetched from *disk* into the *Inode* buffer using the inode location

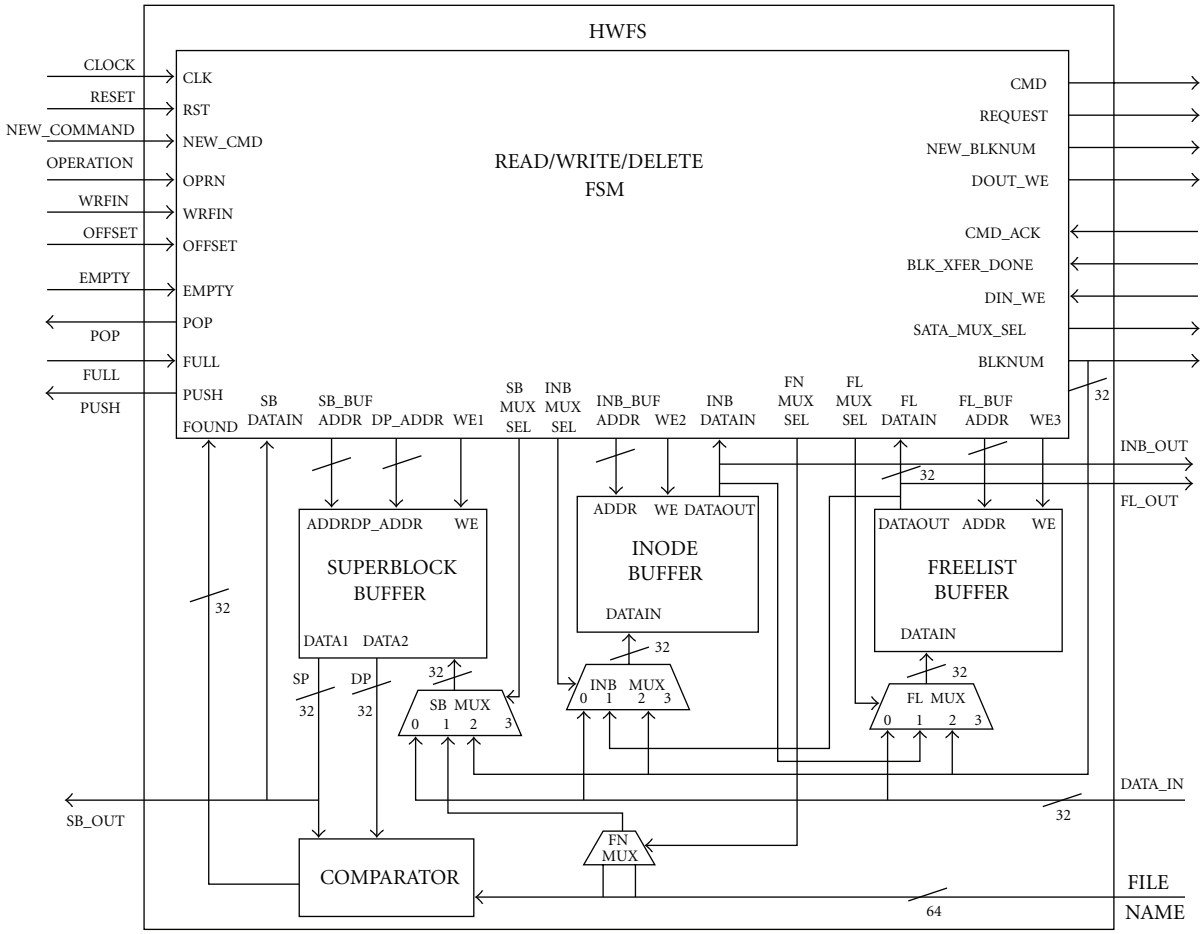

Figure 5: Hardware filesystem core: block diagram.

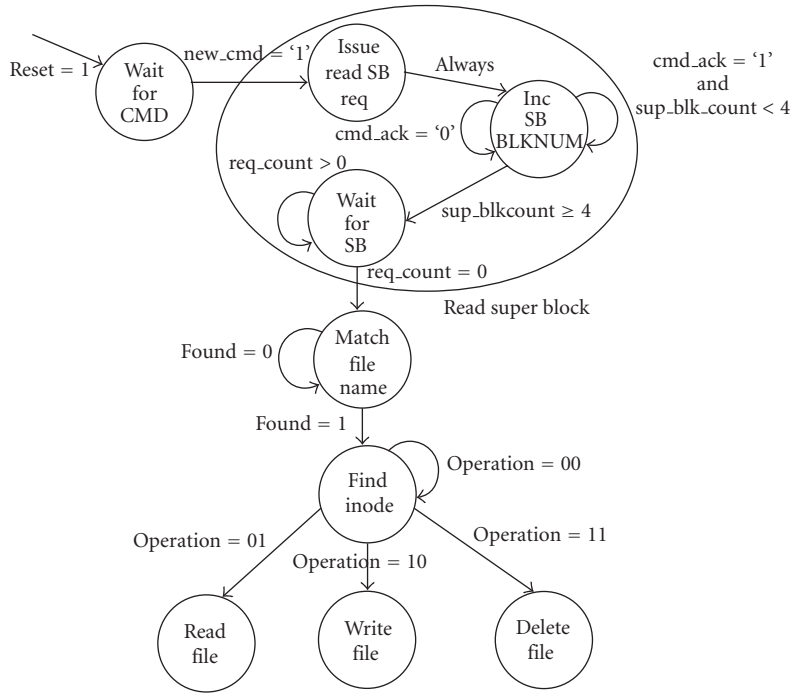

Figure 6: Open file state machine.

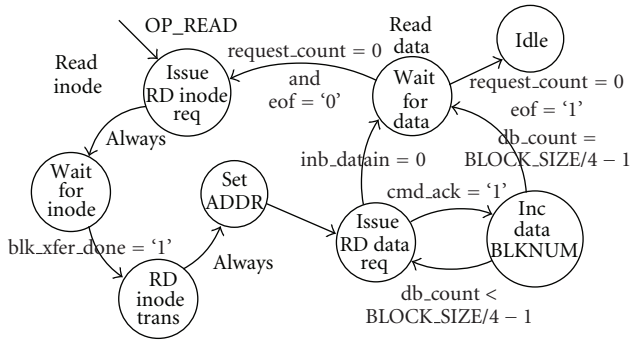

Figure 7: Read file state machine.

number as a blockid. The state machine then transitions to the *Read Data* state via an intermediate state: *Set Address* which accounts for the delay between setting the BRAM address and reading the output blockid. In *Read Data* state, the inode block is read sequentially, 4-bytes at a time. The output inodes are fed back to the state machine and used as blockids for fetching data blocks from *disk* into the Read FIFO. The last inode in the root inode block links to the next inode block of the file. The FSM uses this link inode for fetching the next inode block of the file by cycling back to the *Read Inode* state. The data blocks are read until an inode 0 is found which signals the end of file. The state machine then returns to the *idle* state.

Write File. Write File shown in Figure 8 either creates a file if it does not exist on the disk or appends data to an existing file. Once the file has been opened, the state machine extracts the filesystem metadata from the *Super Block* buffer in *Read FSMD* state. The *Read Free Block* state reads the first block from the freelist into the *Freelist* buffer. The FSM then goes to *Write Data Block* state where the data blocks are written from the *Write FIFO* out to disk using free blockids from the freelist buffer. The blockid is also simultaneously stored as an *Inode* in the *Inode* buffer. In *Write Inode Block*, the inode buffer's contents are written to disk and a *Root Inode* block for the file is created. As the file size increases, subsequent inode blocks of the file are created and added as a linked list to the root inode block. The file's inode blocks thus exist in the form of a linked list interspersed among the data blocks and freelist blocks. After the data blocks and inode blocks are written, the filesystem metadata and super blocks are written to close the file so that HWFS is ready for the next operation.

Delete File. In Delete file, shown in Figure 9, the first block from the freelist is read into the *Freelist* buffer and the *Root Inode* block is read into the *Inode* buffer. The blockids from *Inode* buffer are then transferred to *Freelist* buffer starting from the freelist index, until it is full of free blockids. Next, the contents of the freelist buffer are stored on disk. The FSM reads the Super Block to delete the file name and update the filesystem metadata. The new *freelist head* and *freelist index* are written to the Super Block buffer and then transferred to *disk*.

3.2. Multidisk 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 10 illustrates a high-level block diagram of the system using the hardware filesystem and a SATA 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 [17] 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.

Purpose of the RAM Disk. When presenting a hardware filesystem, it would be assumed that the data would be stored on a hard disk. In these tests we have opted to use a specially designed RAM Disk in place of the SATA hard disk. One might ask why to read about a disk-less hardware filesystem. To this seemingly simple question, we would like to explain our reasoning for the lack of a hard disk in the currently implemented design. First and foremost is the cost of the SATA IP core. While SATA IP cores are currently for sale [18], they are 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 redesign of the system.

Our implementation attempts 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

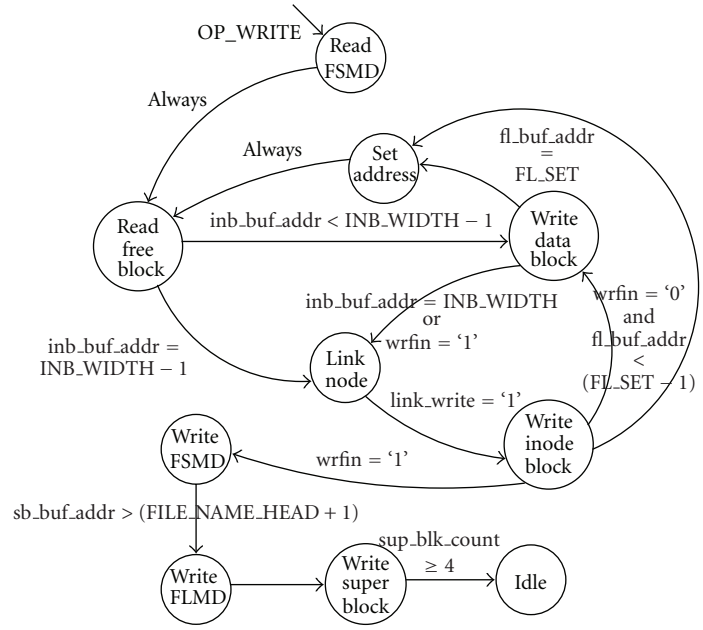

Figure 8: Write file state machine.

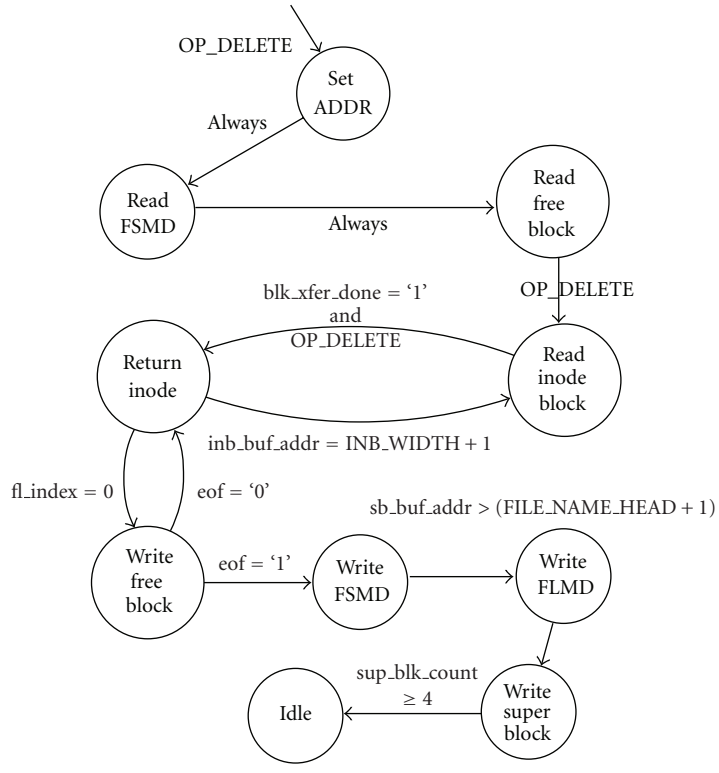

Figure 9: Delete file state machine.

Filesystem's interface with the storage device's interface. This can be seen in Figure 11. 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 that with a working hardware filesystem the disk interface would take focus, reducing the number of unknowns in the design.

Finally, we do not aim to use the RAM Disks 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. Therefore, a straightforward test between the two filesystems is not immediately possible. Instead, what we show is the efficiency of the hardware filesystem.

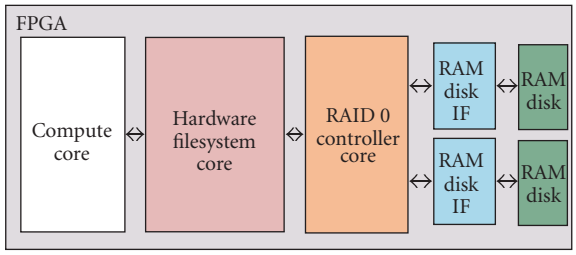

Figure 10: Hardware system level interface between the SATA core and hardware filesystem.

System Implementation. A hardware base system has been created consisting of a processor, system bus, on-chip memory, external memory, and the HWFS core. Linux 2.6 has been compiled and configured to run with this hardware base system; however, it is important to note that Linux is being used as the test bench and is not using the HWFS core. Eventually, considering the results from these tests, Linux will use the HWFS core in place of its current software file system.

Figure 11(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 11(b) highlights the flexibility of the HWFS core's design. Purchasing a SATA controller IP core and creating a simple interface between the HWFS and the SATA controller 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.

Adding Multiple Disk Support. To support multiple disks a Redundant Array of Independent Disks (RAID) [19] 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.

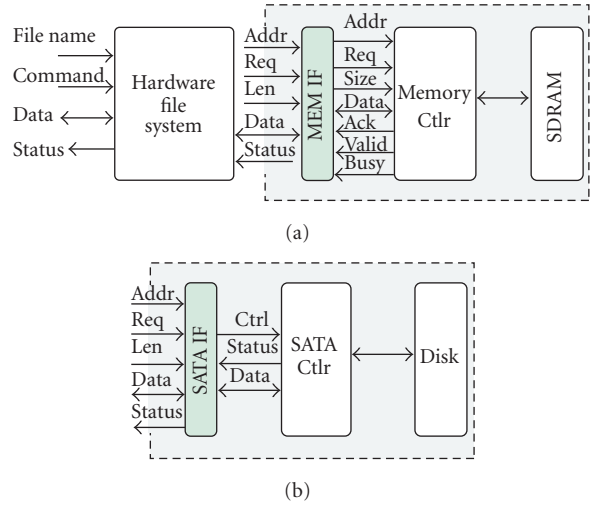

Figure 11: (a) System interface with RAM disk (b) Modular interface with SATA.

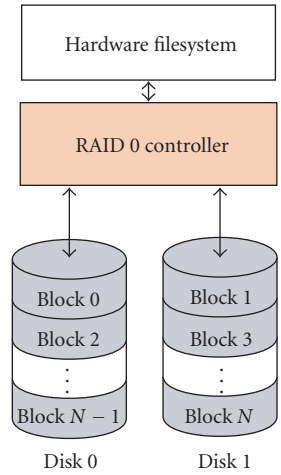

Figure 12: Hardware Filesystem connected to the RAID 0 Controller which stripes the data blocks across two disks.

To illustrate the RAID 0, Figure 12 shows the Hardware Filesystem connected to the RAID 0 controller which is connected to two disks—this paper presents support for N disks, but only two RAM Disks have been tested running on the FPGA at the time of this writing. 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 that the blocks are returned in the correct order since it is possible for two concurrent requests to be returned out of order.

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.

4. Experimental Setup and Results

To establish whether implementing a filesystem directly in hardware provides sufficient improvements in latency and bandwidth while utilizing limited chip resources, we simulated and synthesized the VHDL design code and ran the design on a Xilinx ML-410 FPGA Development board. The experimental setup, results obtained, and analysis follows.

4.1. Simulation Setup and Results. The functionality of the design was first verified in simulation with a VHDL testbench and a *satastub* behavioral model. The testbench instantiates the top level structural VHDL module of the design. It then creates a 100 MHz master clock signal to synchronize the design and provides a reset pulse to initialize the state machine. Next, a test process generates an input test sequence to exercise the design. This includes a 64-bit file name for opening the required file from the disk and a 2-bit operation signal to select from one of the four operations: open, read, write, and delete. The state machine transitions to the idle state and asserts the stop-simulation signal on completing the operations. The testbench checks for this signal and reports a "Testbench Successful" message along with the iteration time. ModelSim verification environment, version 6.3b, running on a Linux Workstation was used for simulation and debugging.

To evaluate the amount of overhead induced by the filesystem itself the execution times of sequential read and write operations were measured in simulation with an ideal disk for file sizes ranging from 1 kilobytes to 5 gigabytes shown in Table 1.

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:

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

$$\text{raw block transfer time} = \frac{\text{file size} \times \text{clock cycle time}}{\text{bytes per clock cycle}}. \quad (1)$$

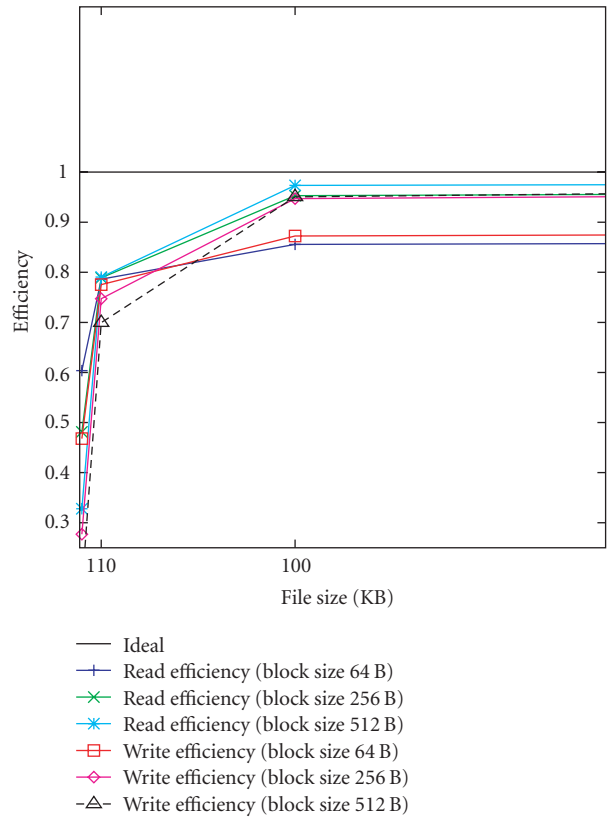

Figure 13: HWFS sequential read/write efficiency in simulation plotted against different file sizes.

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 13 shows a plot of the sequential read and write efficiencies for 64 B, 256 B, and 512 B sized blocks. 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).

4.2. Synthesis Setup and Results. The setup for the system running on the ML-410 builds upon the description given in Section 4.1. The test is running on a Linux-based system which requires a device driver to allow the test application to communicate with the HWFS core. The test begins with the PowerPC initializing the RAM Disk with the empty root filesystem. The PowerPC communicates directly with the RAM Disk since it is a volatile storage device and it is necessary to format the RAM Disk. Once the RAM Disk has been initialized, the PowerPC's test application exercises the HWFS via the device driver. The test application simply issues multiple *open*, *read*, *write*, and *delete* commands to the

Table 1: HWFS sequential read/write execution time in simulation with different block sizes.

	Read			Write		
File Size (Bytes)	64 B	256 B	512 B	64 B	256 B	512 B
1 KB	4.24 μs	5.32 μs	7.81 μs	5.47 μs	9.24 μs	15.52 μs
10 KB	32.56 μs	32.45 μs	32.39 μs	33.02 μs	34.26 μs	36.6 μs
100 KB	299.2 μs	268.6 μs	263 μs	293.4 μs	270.3 μs	269.3 μs
1 MB	3.03 ms	2.71 ms	2.67 ms	2.96 ms	2.7 ms	2.65 ms
10 MB	30.12 ms	26.8 ms	26.6 ms	29.6 ms	26.5 ms	26.5 ms
100 MB	300.4 ms	266.4 ms	264.3 ms	295 ms	263 ms	262.5 ms
1 GB	2.98 s	2.7 s	2.69 s	2.96 s	2.65 s	2.63 s
5 GB	14.6 s	13.6 s	13.5 s	14.4 s	13.2 s	13 s

Table 2: Hardware filesystem with a single disk resource utilization synthesized for the XC4VFX60.

Block Size	Slices	LUTs	F/Fs	BRAMs
64 B	1378	2546	871	2
128 B	1302	2442	872	3
256 B	1254	2350	874	3
512 B	1335	2515	882	3
1024 B	1357	2559	887	4
4096 B	1317	2483	904	14

Table 3: Hardware filesystem with multiple disk resource utilization with Block Size 1024 synthesized for the XC4VFX60.

Resources	HWFS	RAID Ctlr	RAM Disk IF
Slices	1357	343	678
F/Fs	887	255	601
LUTs	2559	626	1253
BRAMs	4	0	2

HWFS core. After the test finishes, the PowerPC reads the RAM Disk to verify the successful completion of the test.

The VHDL design description was synthesized for varying block sizes between 64 and 1024 bytes using the Xilinx Synthesis Tool (XST) available in the Xilinx ISE design suite, version 10.1, for the target device XC4VFX60-11ff1152 from the Virtex-4 family to generate the Xilinx specific NGC files. Table 2 shows the resource utilization statistics with these varying block sizes. Since the super block, inode, and freelist buffers are mapped onto BRAMs, the logic resource (slice) utilization is independent of block size. A slight variation in slice count is observed due to the BRAM buffer's address width variations and the synthesis tool's speed optimization efforts. Based on the synthesized resource utilization results, the largest block size without excessive BRAM usage is 1024 Bytes. At a block size of 4096 Bytes a total of 14 BRAMs are used. In a filesystem with a large number of small files, 4096 Byte blocks would possibly introduce fragmentation; however, the HWFS focuses on opening relatively few large files, orders of magnitude greater than the block size. As a result the block size is less of a restriction as the BRAM resource utilization.

Table 3 shows the resource utilization breakdown for the Hardware Filesystem, RAID Controller, and RAM Disk interface with a block size of 1024 Bytes accessing two disks. In this design the RAID Controller connects to two RAM Disk interface cores which each connect to two external SDRAM DIMMs. In this configuration the HWFS and RAID Controller use a modest 7% of the slices while only using four BRAMs. The RAM Disk interface uses two BRAMs to buffer sending and receiving data between the HWFS and RAM.

Single RAM Disk Results. Table 4 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 hard 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 access times.

Table 5 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 microseconds compared to the simulation time of 5.47 microseconds (Table 1). 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 14. The HWFS stalls until both of the block requests to and from memory are satisfied. Due to this added memory transaction latency, the efficiency graph shows a dip in performance as compared to the simulation efficiency in Figure 13.

Multiple RAM Disks Results. The split transactions implemented for multidisk 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 6 and Figure 15. The limiting factor on the number of RAM Disks in the multiple RAM Disk test is based on the Xilinx ML-410 development board consisting of two

Table 4: Hardware filesystem read/write execution time with single RAM Disk synthesized for the XC4VFX60.

	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

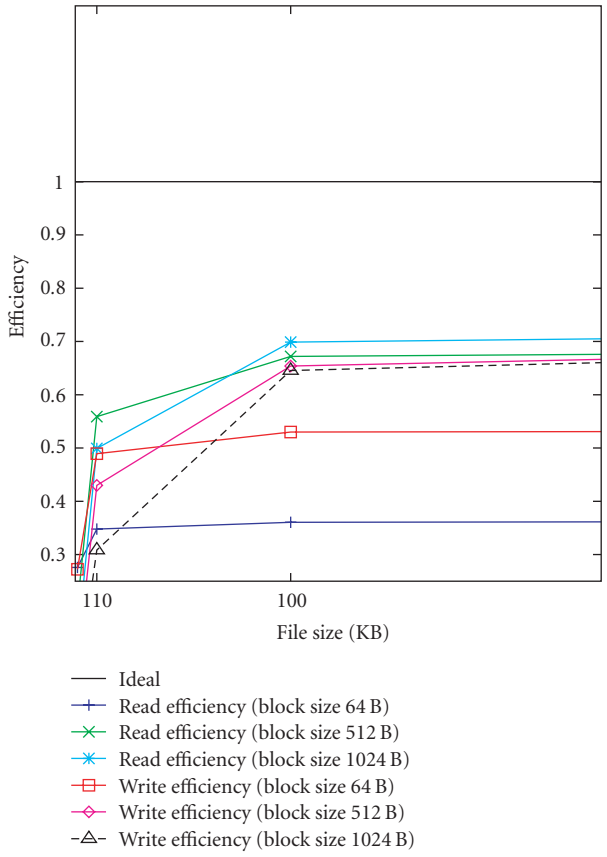

Figure 14: HWFS sequential read/write efficiency with single RAM disk plotted against different file sizes.

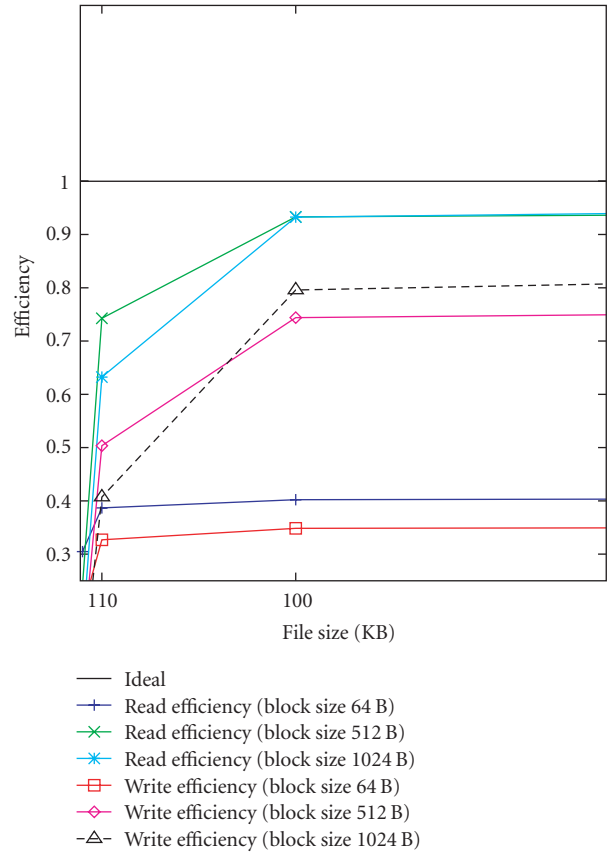

Figure 15: HWFS sequential read/write efficiency with two RAM disks plotted against different file sizes.

Table 5: Hardware filesystem execution time for a 1 KB file with a single ram disk with 64 byte 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

external memory channels. While it would be possible to further model multiple disks by subdividing up the external memory, two disks provide sufficient results to motivate the research to investigate the HWFS with actual SATA disks.

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 15 that the efficiency increases with the size of the block for the same file size. This is due to using larger blocks which improve the data transfer bandwidth. Using large block sizes 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, increasing overall efficiency. Given these trade-offs, 1024 B blocks prove to be ideal for this design.

Table 6: Hardware filesystem read/write execution time with two RAM disk synthesized for the XC4VFX60.

	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

5. Conclusion and Future Work

This paper evaluates the feasibility, functionality, and performance of a hardware filesystem implemented on an FPGA device. The HWFS core provides a generic interface to storage media and was evaluated with a RAM Disk. The design was synthesized and run on an ML410 developer board (Xilinx Virtex-4 device). By adding a RAID controller, split transactions to multiple RAM Disks are also supported, yielding additional performance benefits by allowing concurrent requests in parallel to separate disks.

Synthesis results show that the HWFS and RAID cores in total use ≈7% of the slices for an XC4VFX60 device. The design correctly implements the four basic filesystem operations: open, read, write, and delete. The filesystem, which was designed for situations that require relatively few very large files provides efficient run-time performance for file sizes greater than 100 KB as the metadata overhead has little effect on the access times for files larger than that threshold. The sequential read/write efficiencies improve with larger disk block sizes due to higher data transfer rates and smaller overhead.

The novel architecture proposed and implemented in this project has the potential of increasing the disk to core bandwidth by bypassing the sequential software stack of the OS, avoiding the use of main memory bandwidth and reducing the processor's computational load.

Current results are limited to just RAM Disks but once a SATA IP core is acquired, a simple interface can be created to port to support SATA drives. This will allow the hardware filesystem to be evaluated with actual File I/O performance using HPC I/O benchmarks. Thus, this filesystem core is an important first step in testing a parallel hardware filesystem for coordinating file access from multiple, distributed disks.

Acknowledgment

This project was supported in part by the National Science Foundation under NSF Grant CNS 06-52468 (CRI). The opinions expressed are those of the authors and not necessarily those of the foundation.

References

[1] S. Tam and L. Jones, "Embedded serial ATA storage system," Tech. Rep. XAPP716(v1.0), Xilinx, San Jose, Calif, USA, October 2006.

[2] R. Sass, W. V. Kritikos, A. G. Schmidt, et al., "Reconfigurable Computing Cluster (RCC) Project: investigating the feasibility of FPGA-Based petascale computing," in *Proceedings of the International Symposium on Field Programmable Custom Computing Machines (FCCM '07)*, April 2007.

[3] C. Pedraza, E. Castillo, J. Castillo, et al., "Cluster architecture based on low cost reconfigurable hardware," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 595–598, September 2008.

[4] 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 International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 265–270, December 2008.

[5] H. K.-H. So and R. Brodersen, "File system access from reconfigurable FPGA hardware processes in BORPH," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 567–570, September 2008.

[6] 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 ACM/IEEE Conference on Supercomputing (SC '07)*, 2007.

[7] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *Proceedings of the ACM/IEEE Conference on Supercomputing (SC '08)*, 2008.

[8] NCBI user services, "Genbank overview," August 2005, http://www.ncbi.nlm.nih.gov/Genbank.

[9] T. Agerwala, "System trends and their impact on future microprocessor design," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO '02)*, IEEE Computer Society Press, Los Alamitos, Calif, USA, November 2002.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, Calif, USA, 1996.

[11] M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1991.

[12] D. Lavenier, S. Guyetant, S. Derrien, and S. Rubini, "A reconfigurable parallel disk system for filtering genomic banks," in

Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '03), pp. 154–163, 2003.

[13] Netezza, "Netezza data warehouse appliances," http://www.netezza.com/data-warehouse-appliance-products/dw-appliance.aspx.

[14] Bluearc, The bluearc file system technology, http://www.bluearc.com/html/products/file_system.shtml.

[15] 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.

[16] A. A. Mendon and R. Sass, "A hardware filesystem implementation for high-speed secondary storage," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 283–288, December 2008.

[17] Xilinx Inc., "Xilinx ml410 board," http://www.xilinx.com/ml410-p.

[18] ASICS World Services, "Serial ATA Host IP Core," December 2007, http://www.asics.ws/.

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