A Hardware Filesystem Implementation for High-Speed Secondary Storage

Ashwin A. Mendon Ron Sass
Electrical & Computer Engineering Department
University of North Carolina at Charlotte
9201 University Blvd., Charlotte, NC 28223
{aamendon,rsass}@uncc.edu

Abstract

Platform FPGAs are capable of hosting entire Linux-based systems including standard peripherals, integrated network interface cards and even disk controllers on a single chip. Filesystems, however, are typically implemented in software as part of the operating system. This presents a challenge as some applications are very sensitive to file I/O latency and Platform FPGA processor cores are clocked at relatively slow frequencies. This paper describes a design and implementation of a filesystem in hardware*. A hardware implementation offers several features that have potential of improving certain classes of applications.*

The filesystem implemented is a simplified version of the well-known UNIX filesystem and specifically designed to handle a relatively low number of very large files. The design synthesizes but lacks a SATA host controller needed to test it. Instead, Modelsim was used to verify the functionality of four basic operations: open, read, write and remove. Synthesis results show that the core uses a modest 3% of the slices (and 3 BRAM blocks) of a Xilinx Virtex-4 FX60 device. By using a behavioral model of a SATA disk controller, sequential read bandwidth simulations achieved over 3 Gb/s for block sizes of 512 bytes. Since physical disks are much slower, these results suggest that a hardware filesystem core offers several benefits with little cost and no loss of performance.

1. Introduction

As Integrated Circuit (IC) technology advances, the programmable logic resources available on FPGA devices continue to grow as well. This allows, among other things, greater integration of computing systems. Indeed, it is now feasible to integrate network interface cards [7], disk controllers [8], and other conventional peripherals [9] onto a single Platform FPGA device running Linux. These developments present an enormous potential for reconfigurable, high-performance computing — especially for out-of-core applications and applications that need to stream very large data sets.

This paper describes the implementation of a *hardware* filesystem. Filesystems are typically implemented in *software* as part of the operating system. The primary role of the filesystem is to organize the sequential fixed-size disk sectors into a collection of variable-sized *files*. By moving this functionality into hardware, the proposed system gives computational accelerator cores (implemented in the programmable logic of an FPGA) direct access to the files on a disk. This also has the potential of increasing the bandwidth from disk to core, lowering the latency, and reducing the computational load on the processor.

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 is explicitly moved between secondary storage and main memory. (These so-called out-of-core applications are far more efficient than relying on the OS to swap.)

Alternatively, if part of the computation is performed by accelerators implemented in the programmable logic of an FPGA, then the data does 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 without buffering the data in main memory.

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 single processor performance continues to double every 18 months, biological databases are growing even faster. Figure 1 shows both growth rates of between 1994 and 2004 on a semi-log graph. The nucleotide data points come from GenBank [6], a public collection of sequenced genomes. A line fitted to this data shows a compound annual growth

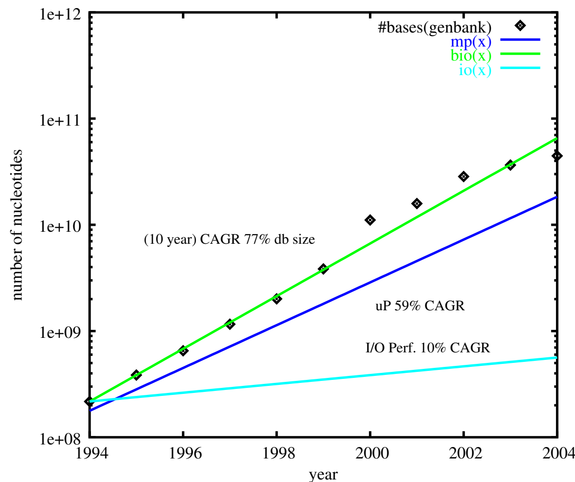


Figure 1. compound annual growth rate of problem size, single processor performance, and I/O subsystem performance

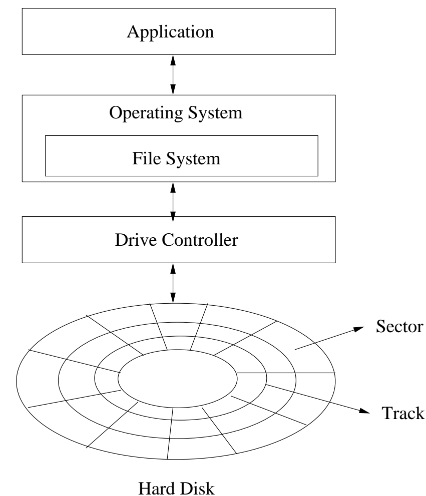


Figure 2. disk interface to operating system

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 [1] suggest a 10% compound annual growth rate in performance while others [3] 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 is a first step towards multi-disk solutions are better handled in hardware.

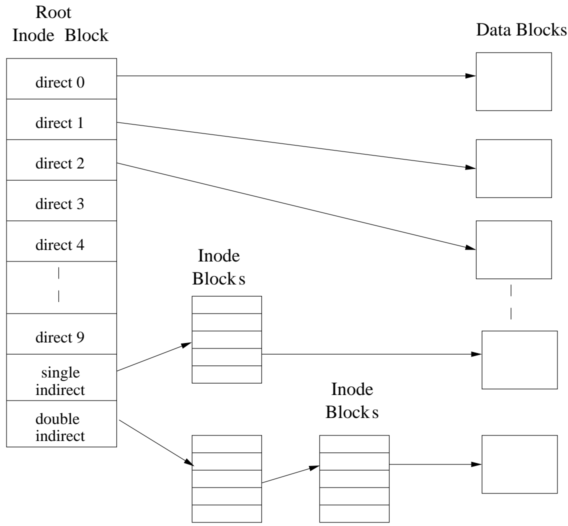
The rest of this paper is organized as follows. In the next section, a brief description of filesystem requirements and FPGA technology needed to implement the proposed core. In Section 3, an overview of the proposed hardware filesystem is described. Its functionality, performance and resource utilization have been tested in simulation and with synthesis; these results are reported in Section 4. The paper concludes with a brief summary and future directions.

2. Background

The main purpose of a computing system is to create, manipulate, store, and retrieve data. As such, filesystems have been central to all modern computing systems. Filesystems are responsible for managing and organizing files on a nonvolatile storage medium, such as a Winchester-type disk drive (commonly 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 hard 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. Figure 2 illustrates the high-level relationship between an application and nonvolatile storage.

Our goals are specifically to support the RCC project [7]. The proposed filesystem core needs to support relatively few, very large files. Ownership, access control, and even the concept of subdirectories are not very important. Fast sequential access is essential and reasonably fast random read access is important. After researching the architecture of the existing filesystems, we chose to model our design by adapting the UNIX filesystem (UFS) [2]. UFS uses logical blocks of 512 bytes (or larger multiples of 512). (Logical blocks may consist of multiple disk sectors.) The logical blocks are organized into a filesystem. The filesystem uses an I-node 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 of pointers to data blocks. UFS includes a collection of direct, single indirect, double indirect, and triple indirect pointers in the I-node. The file system layout is as shown in the Figure 3. Our Filesystem only uses direct and single indirect pointers. However, the indirect logical blocks include a pointer to another indirect block — essentially reverting to a linked-list structure for very large files.

Normally, the filesystem is designed to be independent



Direct and Indirect Inode Blocks

Figure 3. UNIX I-node structure

of the disk controller. For expediency, we have focused 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 gets a dedicated bandwidth and there are no master/slave configuration jumper issues as with parallel ATA drives. The pincount 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 transcievers. For example, the Xilinx Virtex II, Virtex-4, and Virtex-5 device families have members that include Multi-Gigabit Transceiver cores (MGTs). These cores can be configured to communicate via the SATA protocol at the physical layer.

3. Design and Implementation

To evaluate the feasibility and performance of a hardware filesystem core, it was necessary to set up an experimental apparatus. Figure 4 shows the three main components: a testbench to exercise the hardware filesystem, the hardware filesystem itself, and a behavioral model of a SATA host controller.

3.1. SATA Host Controller

Although SATA host controllers exist [8], the core is expensive and not required to test the feasibility of a hardware filesystem core. Thus our approach was to create a behavioral simulation of the SATA host controller. First, `mkafs.vhd` is run in a simulator to create an empty

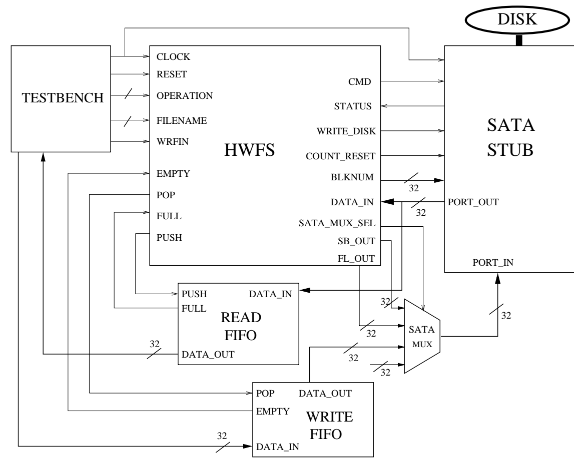


Figure 4. Experimental Apparatus

filesystem which is written to a local disk file on the workstation (`virtdisk.bin`). This includes the *SuperBlock* metadata and an interleaved linked freelist of all the data blocks. In our current implementation, the *Superblock* is 4 blocks wide and can store up to 15 files. This is easily scaled up if the filesystem needs to accomodate more files.

During simulation, the SATA stub opens the empty disk file `virtdisk.bin` and loads it into a 2D array in memory. The dimensions of the array are defined by the disk size and block size. The column indices are used as logical block numbers of the disk. Depending on the command and block number received from the controller state machine, disk blocks are copied to and from read and write buffers. These buffers interface to the 32-bit input and output ports of the SATA stub. We did not attempt to model the actual characteristics of a disk drive; data is immediately available when the HWFS requests it.

3.2. HWFS Core

The Hardware FileSystem (HWFS) core consists of controlling State Machine and a Datapath as illustrated in Figure 5. The datapath consists of two Block RAMs, a comparator, three 32-bit 4:1 multiplexers, two 32-bit 2:1 multiplexers, and FIFOs. It interfaces to the SATA host controller and, in our example, the testbench. The latter interface is simple enough that it can be directly connected to a computational accelerator or attached to a bus.

The state machine implements the Open, Read, Write and Remove file operations. The 2-bit *operation* port is driven by a testbench to select from the four operations. The FSM issues an internal *command* signal and a 4-byte *blockid* to read from or write to the appropriate location in the *satastub* array. The *satastub* sends out the block 4-bytes at a time through *portout*. After completing a block transac-

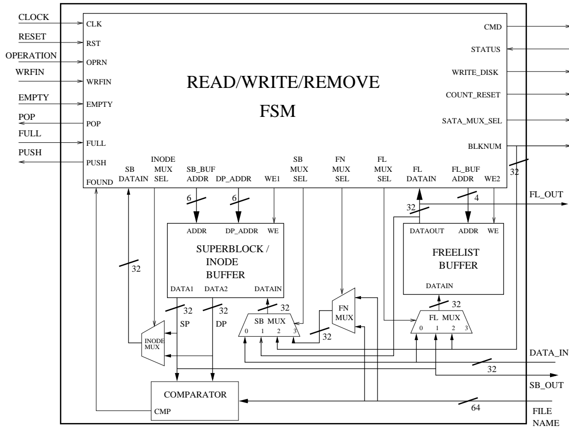


Figure 5. Hardware File System: State Machine and Components

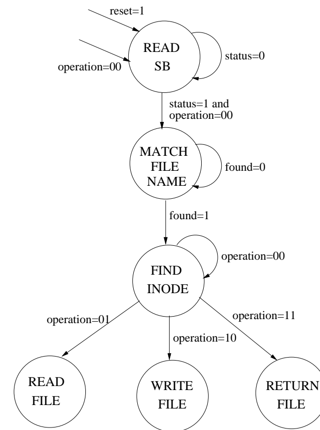


Figure 6. Open File State Machine

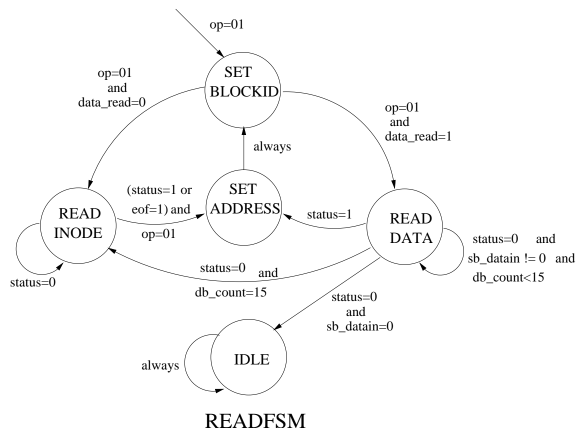


Figure 7. Read File State Machine

tion, it asserts the *status* signal. *WriteDisk* signals *satastub* to write the entire 2D array to disk file.

Block RAMs are used as buffers for storing the *SuperBlock*, *InodeBlocks* and the *FreeListBlocks*. The superblock buffer, which is 4 blocks wide to accomodate a 4 block SuperBlock, is dual ported to allow for 8-byte filename comparison and also in checking for an end of file primitive. Blockids are transferred between the two buffers for the purpose of creating inode blocks in *writefile* and returning inodes to freelist in *removefile*. The 64-bit equality comparator is used for finding a match between the testbench filename and the superblock filename.

3.3. Example Operations

The Mealy-type controlling state machine was implemented to handle all four filesystem operations: *openfile*, *readfile*, *writefile*, and *removefile*. It has 17 states in all, some of which are common between read, write and remove operations. For clarity, we describe just two operations, *openfile* and *readfile*, here. Full details of the whole 17-state state machine and each operation are documented elsewhere [4].

Open File: Open File takes in a filename alongwith an open command from the testbench and returns the file's root inode location.

The state machine starts from the *Read Super Block* state and reads blocks 0-3 from *satastub* into the *SuperBlock* 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 superblock 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 superblock and a *filenotfound* signal is asserted. In find inode state, the state machine captures the file's root inode location from the filename-inode mapping in the superblock.

Read File: Once the file is opened, Read File uses the file's root inode block location to read in the file contents into the Read FIFO.

Read File begins with the *Read Inode* state. The root inode block of the file is first fetched from *satastub* into the SuperBlock/Inode buffer using the inode location number as a blockid. The state machine then transitions to the *Read Data* state via two intermediate states: *Set Address* and *Set Blockid* which are used to account for the delay between setting the BRAM address and reading the output blockid. In *Read Data* state, the inode block in the Superblock/Inode buffer is read sequentially, 4-bytes at a time. The output

inodes are fed back to the state machine and used as block-ids for fetching data blocks from the *satastub* 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 till an inode 0 is found which signals the end of file. The state machine then goes to the *idle* state.

4. Evaluation and Analysis

4.1. Experimental Setup

To test the timing, area and performance metrics described in Section 1, we used ModelSim and Xilinx ISE for simulation and synthesis purposes.

The XST manual served as a guideline for writing synthesizable VHDL code. The controller state machine was decomposed into a two process Mealy model. Mealy state machines use fewer states than Moore models. This helps in reducing the number of flip-flops and decreases the latency of operations. Block RAMs were chosen in the design for the superblock/inode buffer and freelist buffer for efficient resource utilization. The buffer sizes vary with the disk blocksizes of the filesystem. Hence, for large blocksizes mapping this logic on BRAMs saves on slice resources. A structural VHDL code instantiates and binds the synthesizable components in the design: the state machine, superblock buffer, freelist buffer, the comparator and the multiplexers.

The VHDL design description was synthesized using the Xilinx Synthesis Tool (XST) [12] available in the Xilinx ISE design suite, version 10.1 [10], for the target device XC4VFX60-11ff1152 from the Virtex-4 family [11]to generate the Xilinx specific NGC files.

For simulation purposes, the *satastub* behavioral model and a VHDL testbench file was included to test the functionality of the design. 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 filename 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 remove. On completing the read/write/remove file operations the state machine transitions to the idle state and asserts the stop-simulation signal. The testbench checks for this signal and reports a "Testbench Successful" message alongwith the iteration time.

ModelSim verification environment, version 6.3b, [5] running on a Linux Workstation was used for simulation and debugging.

Table 1. Statistics for HWFS resource utilization with different block sizes

Block Size	Slices	LUTs	F/Fs	BRAMs
64 B	759	1471	343	2
128 B	724	1369	345	2
256 B	749	1446	349	2
512 B	783	1502	353	2
1024 B	762	1463	356	3
4096 B	779	1476	364	10

4.2. Results and Analysis

The design was synthesized for different block sizes and it was observed that the logic resource utilization is independent of the block size. The superblock and freelist memory buffers which vary with the block size are mapped onto BRAMs. Hence, the number of slices is not affected by increasing block sizes. For a 512 byte block, the design uses 783 slices, which takes up just 3 percent of the chip area. Table 1 shows that two 18-Kbit Block RAMs are utilized for block sizes from 64 B to 512 B. This increases to 3 BRAMs for 1024 B blocks and goes upto 10 BRAMs for 4096 B blocks. The slice utilization varies between 724 to 783 slices. This is because, the logic resources used for decoding the BRAMs addresses varies with the block size and the Xilinx Synthesis Tool tries to optimize the design for speed.

The maximum frequency estimate of the design was found to be 144.35 MHz, which meets our target clock frequency of 100 MHz. The timing report also gave satisfactory delay estimates on the four netlist domains: register to register, input to register, register to output and input to output.

The filesystem performance for reads and writes was checked using file sizes ranging from 1 kilobytes to 5 gigabytes. A 1 GB file read with 512 Byte blocksizes takes 2.7 seconds. Hence, the sequential read bandwidth can be estimated as 3.01 Gb/s. We recorded the read and write latencies during simulation and computed the efficiency of the filesystem using the equation stated below

$$\text{eff} = \frac{\text{data latency}}{\text{(data + overhead) latency}}$$

The data latency is the time taken to transfer raw data blocks of a file between the file system and the sata array. The read overhead includes the time taken to read a superblock, find a filename match, get it's root inode block (open file operation) and read the inode blocks of the file (read file operation). Figure 8 shows a plot of the sequential read and write efficiencies for 64 B, 256 B and 512 B sized blocks.

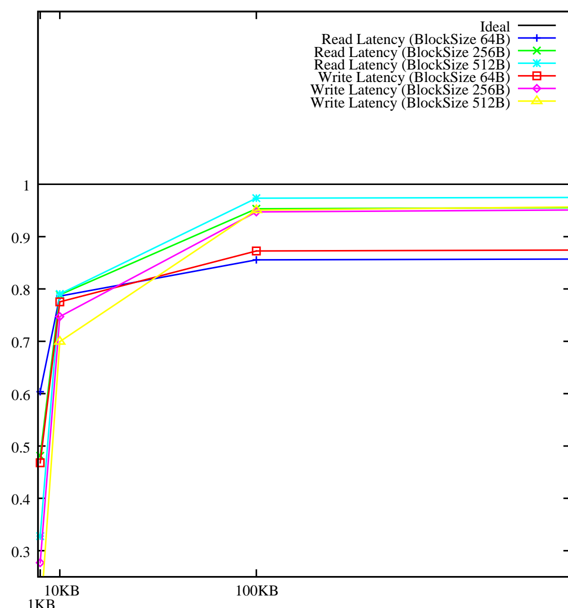


Figure 8. File Read/Write Efficiency plotted against different file sizes(10 KB-5GB)

It is observed that for small files (1 KB to 10 KB) the efficiency is below 80 % . It goes upto 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 because, the overheads have a lesser effect on the latencies for large files thereby achieving efficient run-time performance. For larger blocks, the plot shows that there is no substantial improvement in efficiency. Table 1 shows that using 512 B blocks consumes 2 BRAMs and for larger blocks, the cost of BRAMs increases (10 BRAMs for 4 KB blocks). Hence, using 512 B blocks gives us the ideal tradeoff between area and performance for our design.

It is also observed from figure 8, that the efficiency increases with the size of the block for the same filesize. This is because, using larger blocks improves the data transfer bandwidth. But having larger blocks on the disk increases the block fragmentation, leaving large portions of the disk unused. Our file system will primarily deal with very large sized files existing in biological databases and hence fragmentation due to large block sizes will not be a critical issue.

5. Conclusion

The goal of this work was to verify the feasibility of implementing a filesystem directly in hardware for high performance computing. The novel achitecture proposed and implemented in this project avoids the sequential system software bottleneck by giving the computation cores on the FPGA the opportunity to bypass the operating system entirely. It enables the cores to get direct, high bandwidth access to large data sets. The design, correctly implements the four basic filesystem operations: open, read, write and remove. The filesystem was synthesized for an ML410 Virtex-4 platform FPGA with a 3 % slice utilization. The read/write efficiencies measured during simulation, improve with larger disk block sizes due to higher data transfer rates and smaller overhead.

The design currently provides sequential access to a file. It can be enhanced to incorporate random reads and writes by implementing the lseek operation. Also, once the SATA IP core is purchased, the filesystem can be interfaced to it for measuring the actual File I/O performance. The hardware file system can then be replicated on each node of the 64-node cluster. This design will serve as an important first step to develop and evaluate a parallel filesystem which will co-ordinate file access from multiple disks accross the cluster and present a single filesystem image for parallel applications.

References

[1] T. Agerwala. System trends and their impact on future microprocessor design. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. Invited keynote talk.

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

[3] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[4] A. A. Mendon. Design and implementation of a hardware filesystem. Master's thesis, University of North Carolina at Charlotte, Aug. 2008.

[5] MentorGraphics, Inc. Modelsim. URL: http://www.model.com.

[6] NCBI User Services. Genbank overview, Aug. 2005. http://www.ncbi.nlm.nih.gov/Genbank/.

[7] 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 *Proceedings of the 2007 International Symposium on Field Programmable Custom Computing Machines*, April 2007.

[8] S. Tam and L. Jones. Embedded serial ata storage system. Technical Report XAPP716(v1.0), Xilinx, Inc., oct 2006.

[9] Xilinx, Inc. Xilinx EDK design suite 10.1. URL: www.xilinx.com/edk.

[10] Xilinx, Inc. Xilinx ISE design suite 10.1. URL: http://www.xilinx.com/ise.

[11] Xilinx, Inc. Xilinx virtex-4 fpga's. URL: http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/-virtex4/index.htm.

[12] Xilinx, Inc. Xst user guide.