A Reconfigurable Hardware to Accelerate Directory Search

Venkatanathan Varadarajan*, Sanath Kumar R*, Arun Nedunchezhian* and Ranjani Parthasarathi**

Department of Computer Science and Engineering,
College of Engineering Guindy,
Anna University,
Chennai, India.
{venk1989, dayanandasaraswati,arunn3.14}@gmail.com, rp@annauniv.edu

Abstract. There is a large proliferation of data in the Internet leading to increase in the number of files in the Web Servers and Database Systems. In the current scenario, it is common to have millions of files in a single directory. It is highly essential to devise a faster execution of file access operation in such large directories that additionally poses memory latency problems. We investigate and analyze a method to reduce the memory bottleneck and efficiently perform a directory search using a reconfigurable hardware. Such a mechanism does not require the processor to execute the file access operations but delegates such accesses to this reconfigurable hardware.

Keywords: Directory Search, Field Programmable Gate Array, Ext2, Unix Based File systems, Flash Memory, Inode Search

1 Introduction

Files are the basic elements for storage and manipulation of data in a computer system. File read/write operations are very frequent in any computing environment. Many studies [6, 14] corroborate the fact that there is an unprecedented increase in number of files, file size, directory size and depth of the directory in a file system. Today, even medium scale web servers house billions of files. Similar situations prevail in database servers where a file system's utility goes far beyond just storing files. File retrieval becomes a mission-critical task in such applications and it is the file system's responsibility to provide proper service. The need and use of files are imperative in the current computing scenario.

Many file systems have been proposed to efficiently search and retrieve files from the storage device. All of these efforts are focused on software solutions for file system optimization. In this paper we propose a reconfigurable hardware based solution to accelerate the directory search. We use *ext2* [4] as the reference file system to showcase our idea. Ext2 is a file system used in a UNIX based systems where inode is used to represent a file/directory in a file system. In such file systems, given an absolute path to the file, an inode corresponding to the file must be retrieved. Traditional implementations first retrieve the inode corresponding to the root directory. The inode number of the next directory in the path is located within the root directory's inode using a linear search through the linked list as shown in Figure 1. With the inode number, the inode is read from the hard disk and this process repeats until the inode corresponding to the requested file is obtained. It is evident that such a search is extremely expensive because of the hard disk access latency and linear search operation. Although file systems provide indices to make the above operations faster, access times are still in the order of milliseconds. We propose a hardware search mechanism to accelerate this process.

This paper is organized as follows. Section 2 presents a brief overview of the related work done in the area of file system enhancements. Section 3 provides an elaborate discussion on the design of the reconfigurable hardware that specifically solves the file access limitations of *ext*2 file system. Section 4 includes the analysis of our system's performance. Section 5 provides concluding remarks and possible future work.

* Senior undergraduate students of Department of Computer Science and Engineering, CEG.
** The student team's research mentor, Professor in Department of Computer Science and Engineering, CEG

2 Related Work

Ever since the development of Unix File System(UFS) many high-performance file systems have been developed. The list includes Extended File system(ext), ReiserFS [3], Journaling File System(JFS) of IBM and XFS for IRIX operating system of Silicon Graphics [15]. First enhancement to the BSD(Berkley Software Distribution) Unix File System was proposed by Marshall et al [10]. These provided mechanisms to improve file I/O by clustering sequentially accessed data and changing the file representation at the block level. Further optimizations were proposed on the FFS implementation by Ian Dowse and David Malone [7]. Rémy Card mimicked the Minix File System with some significant modifications and came up with the first Extended File system (ext). Some of the limitations in the ext file system were overcome in the second Extended File system (ext2) [4] which was later equipped with a hash based indexing mechanism [13]. A data structure called HTree was utilized to enhance the search within a directory. The directory indexing, first of its kind, was implemented in the third version of the Extended File system (ext3) [16]. Mikulas Patocka came up with a new file system called SpadFS [12] which is characterized by high performance file system I/O, continuous allocation of directories and read-aheads.

It is evident that there have been constant advancements in design and implementation of a file system. Our contribution is the idea of adding a dedicated reconfigurable hardware(File system Access Accelerator) that can perform the file systems operations faster than the software counterparts. Such an architecture has a number of advantages as opposed to the traditional implementations. Since host processor is now freed from the task of searching for a file's inode in the file system, it can continue doing other operations. Also a dedicated hardware completes the task much faster than the host processor.

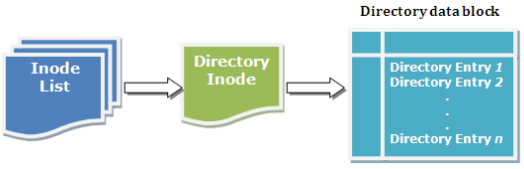

Fig. 1. Directory structure and directory entries

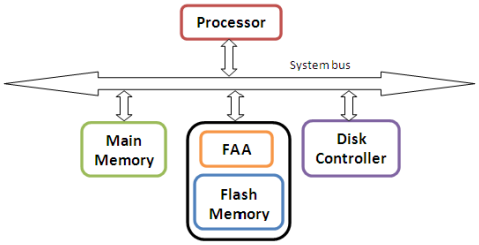

Fig. 2. System Configuration

3 File system Access Accelerator (FAA)

The goal of the FAA is to speed up the search of an inode in a large directory. Hence the input to the system is an absolute path to the file and the output is the corresponding inode. In order to search for the inode, we use a flash memory that holds the metadata of the file system. Two main factors which favor the choice of a flash memory are: 1.) File system metadata often is larger than what can be accommodated in the reconfigurable fabric. 2.) Flash memories have faster read times than the HDD [9, 5, 8] and hence will improve performance. The proposed system configuration is shown in figure 2.

3.1 System Configuration

The flash memory and FAA together act as an independent subsystem. This is attached to the host computer system through the Front Side Bus (FSB). The FAA connects directly to FSB and the flash memory is controlled by the FAA. The FAA has input/output buffers that are memory mapped to the host processor. This configuration provides two advantages:

1. The system can issue independent load operations to the disk controller that loads the inode found by the FAA into the primary memory.
2. The host processor can directly read/write to the I/O buffers without any special communication protocol as the system is memory mapped to the processor.

The file system metadata which consists of list of inodes, data blocks of directory, etc. except the file data are maintained in the flash memory module interfaced with the reconfigurable hardware. This flash memory is the primary data source for the FAA to work with.

3.2 FAA Architecture

The FAA needs to handle the loading of the large set of directory entries from the flash memory for searching. Each directory entry has inode number (4Bytes), record length (2B), filename length (1B) and filename (1-255B) as shown in Figure 4 [4]. Loading sets of such large directory entries (8-262B) from a flash memory and searching the linked-list of directory entries efficiently are the main objectives of the FAA.

Searching a file/directory in a large set of directory entries of a directory demands more time as the number of files/directories in a directory increases. We utilize a hash based indexing technique similar to the HTree [13] where the search range is reduced to a single block by hashing. The file/directory is searched in this block by traversing the linked-list of directory entries and if a match is found, the inode present in the directory entry is returned to the host processor, else the file/directory is not available.

The FAA thus consists of two major modules, the memory access module and File/directory search module.

3.3 Memory Access Module

The memory access module is responsible for bringing in the block that needs to be searched from the flash memory into the memory blocks (block RAMs) in the FAA. This is handled by the flash memory controller (FMC). Given a file/directory name to be searched in the current directory (whose inode is available), the FMC hashes the file/directory name using the Hash Unit (HU). The HU's output is a hash value which decides which block in flash memory contains the required file/directory using the hash index maintained in the parent inode [13]. FMC then initiates the block transfer from the flash to the block RAMs in the FAA. The size of the flash blocks is the same as the file systems block size (1,2,4 or 8KB) [1].

A number of block RAMs are configured to house a single flash block in order to reduce the memory bottleneck. By interleaving data from a flash block into several block RAMs with multiple ports, memory accesses are parallelized. For instance, in Spartan and Virtex series of FPGAs [2], which support block RAMs with two independent read/write ports, configurations to transfer 32bits to 64 bits (4 or 8 Bytes) in a clock cycle can be obtained. A *Block Start Address* register is used to hold the address from which the data is to be read. The data read is transferred to an internal buffer register for processing. If the internal register buffer is K bytes in size and N block RAMs are available each with M read/write ports then, this configuration enables to load K bytes in a time equivalent to loading $\left(\frac{K}{N*M}\right)$ bytes of memory (by parallelly loading through M ports from N RAMs). Thus, access time from the block RAM is reduced by a factor of $N * M$. Once the register buffer present in the processing unit is loaded, the file/directory searching logic is initiated.

3.4 File/Directory Search Module

This module is responsible for comparing the filename written into an input buffer by the host processor, with the data available in the internal register buffer. The challenge here is in handling comparisons of varying sizes. The register buffer may be loaded with a few directory entries or a part of a directory entry (since the size of the internal buffer K can be less than the maximum size of a directory entry). The inode number, record length and filename length (that form the header of the directory entry, as shown in figure 4) are latched into respective registers whenever a new directory entry is read into the buffer. The record length provides information about the starting address of the next directory entry. Similarly, filename length (FL) determines the number of times the buffer needs to be refilled from the block RAMs, $\lceil \frac{FL}{K} \rceil$. These values can be immediately calculated and stored in the control unit's registers as soon as the buffer is loaded (Figure 3). A K byte comparator is used to compare a part of the filename with the filename in the input buffer. The bytes that are needed from the input is selected by the data path selector logic using the offset (size of the header information that needs to be avoided during first cycle) and filename count (used to select which K byte of the input to use). Further, the input from the register buffer is masked based on the offset value. If the current K or less bytes match and the filename count is not zero, the block RAM controller is asked to fetch the continuation of the same Directory Entry from *(Block Start Address* $+K$). The filename count is decremented by K. The procedure is repeated until filename count becomes zero. If the filename count is zero and there is a match, the file is found and the control unit latches the inode number into the output

buffer. On the other hand if there is a mismatch at any point of time, the block RAM controller is asked to fetch K bytes from the next directory entry address and the process is repeated until all the directory entries are searched.

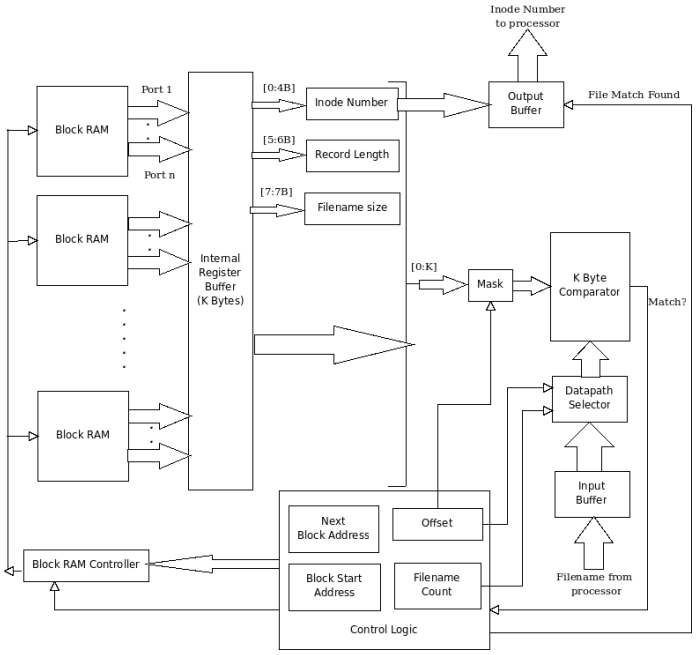

Fig. 3. FAA Architecture

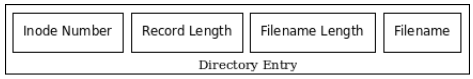

Fig. 4. Directory Entry Format

4 Performance Analysis

Performance of our model is measured based on the Time For Search(TFS) metric. It represents the time taken to find a file/directory name match within a given directory.

$$TFS = T_h + T_f + (T_m + T_p) * d \tag{1}$$

where T_h is the time to perform a hash computation by the HU, T_f is the time taken for a block read from the flash memory, T_m is the time taken to load the internal register buffer from the block RAMs, T_p is the time taken to compare a single filename and d is the average number of directory entries needed to be searched before locating the target file.

$$T_m = \left(\frac{K}{N * M}\right) * \frac{C_m}{F} \tag{2}$$

$$T_p = \left\lceil \frac{AVG(F_L)}{K} \right\rceil * \frac{C_c}{F} \tag{3}$$

where K is the size of the register buffer and comparator (in bytes), N is the number of block RAMs, M is the number of read/write ports in a block RAM, $AVG(F_L)$ is the average filename length (in bytes), C_m is the number of cycles required by a single block RAM to fetch $\frac{K}{N*M}$ bytes and C_c is the number of cycles taken by the comparator to complete a comparison operation. F the clock frequency is the same for both the comparison as well as the memory access operations from block RAM [2].

From equation (1), it can be seen that the one-time investment of the $T_f + T_h$ can be sufficiently hidden by the time taken to fetch and search the directory entries using read ahead techniques. Further, the memory access time for K bytes from RAM is reduced by a factor of $N*M$ using this parallel memory fetch operation. The characteristic equation (1) of the FAA reveal a linear relationship between number of directory entries and time required to do the search.

For example, if $T_f = 30\mu s$ (for a 4KB flash block obtained from [11, 5, 8]), $T_h = 5$ cycles, $K = 32$ bytes, $N = 4$, $M = 2$, $C_m = 1$ clock cycle for a 32 bits (4 bytes) transfer (for Xilinx's Block RAM [2]),

$F = 500MHz$, $C_c = 1$ (for Xilinx Virtex 4 FPGA Series [2]) with $d = 100K$ and $AVG(F_L) = 64$ bytes (from [1]), the parameters $T_h + T_f$ is around $30\mu s$ and $(T_m + T_p)$ is about $12ns$ approximately. Thus, for a single directory search, $TFS = 1.1ms$ for a directory size of 10^5 directory entries and with an average filename length of 64 bytes. This when compared to the traditional software implementation that utilize HDD gives better performance as the latter involves disk read latencies (which itself is in the order of milliseconds apart from delay due to search).

5 Conclusion and Future work

In this paper, we propose a new reconfigurable hardware implementation for increasing the efficiency of file accesses in any UNIX based file system. We have illustrated the design for the *ext*2 file system. The obvious advantages of such an implementation is CPU time is not wasted on these frequent file access operations Further, the access is made efficient by maintaining the file system metadata in a flash memory and parallelizing the memory fetch of directory entries. Since the FAA is designed on a reconfigurable fabric, it can be configured with a specific design for other file systems as well. Performance analysis reveal the time for search in our model is of the order of milliseconds with a directory size of 10^5 directory entries and with an average filename length of 64 bytes.

Future work involves validation of this model and extension to support regular expression based file search operations.

References

1. Linux kernel documentation: Filesystem: Ext2. http://www.kernel.org/doc/Documentation/filesystems/ext2.txt.
2. Using block ram in spartan-3 generation fpgas, March 2006. Xilinx Documentation: XAPP463.
3. Florian Buchholz. The structure of the reiser file system, January 2006. http://homes.cerias.purdue.edu/ florian/reiser/reiserfs.php.
4. Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux*.
5. Intel Corp. Intel flash memory data sheets and specification updates.
6. John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 59–70, New York, NY, USA, 1999. ACM.
7. Ian Dowse and David Malone. Recent filesystem optimisations on freebsd. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 245–258, Berkeley, CA, USA, 2002. USENIX Association.
8. Samsung Electronics. Nand flash memory and smartmedia data book, 2002.
9. Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*, pages 13–13, Berkeley, CA, USA, 1995. USENIX Association.
10. Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, 1984.
11. Chanik Park, Jeong-Uk Kang, Seon-Yeong Park, and Jin-Soo Kim. Energy-aware demand paging on nand flash-based embedded storages. In *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*, pages 338–343, New York, NY, USA, 2004. ACM.
12. Mikulas Patocka. An architecture for high performance file system i/o. *World Academy Of Science, Engineering and Technology (WASET)*, 29:250–255, 2007.
13. Daniel Phillips. A directory index for ext2. In *ALS '01: Proceedings of the 5th annual Linux Showcase & Conference*, pages 20–20, Berkeley, CA, USA, 2001. USENIX Association.
14. Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 4–4, Berkeley, CA, USA, 2000. USENIX Association.
15. Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *ATEC '96: Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.
16. Theodore Y. Ts'o and Stephen Tweedie. Planned extensions to the linux ext2/ext3 filesystem. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 235–243, Berkeley, CA, USA, 2002. USENIX Association.