Proceedings of the
Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

A Reliable and Portable Multimedia File System

Joo-Young Hwang, Jae-Kyoung Bae, Alexander Kirnasov,
Min-Sung Jang, Ha-Yeong Kim
Samsung Electronics, Suwon, Korea
{jooyoung.hwang, jaekyoung.bae, a78.kirnasov}@samsung.com
{minsung.jang, hayeong.kim}@samsung.com

Abstract

In this paper we describe design and implementation of a database-assisted multimedia file system, named as XPRESS (eXtendible Portable Reliable Embedded Storage System). In XPRESS, conventional file system metadata like inodes, directories, and free space information are handled by transactional database, which guarantees metadata consistency against various kinds of system failures. File system implementation and upgrade are made easy because metadata scheme can be changed by modifying database schema. Moreover, using well-defined database transaction programming interface, complex transactions like non-linear editing operations are developed easily. Since XPRESS runs in user level, it is portable to various OSes. XPRESS shows streaming performance competitive to Linux XFS real-time extension on Linux 2.6.12, which indicates the file system architecture can provide performance, maintainability, and reliability altogether.

1 Introduction

Previously consumer electronics (CE) devices didn't use disk drives, but these days disks are being used for various CE devices from personal video recorder (PVR) to hand held camcorders, portable media players, and mobile phones. File systems for such devices have requirements for multimedia extensions, reliability, portability and maintainability.

Multimedia Extension Multimedia extensions required for CE devices are non-linear editing and advanced file indexing. As CE devices are being capable of capturing and storing A/V data, consumers want to personalize media data according to their preference. They make their own titles and shares with their friends via internet. PVR users want to edit recorded streams to remove advertisement and uninterested portions. Non-linear editing system had been only necessary in the studio to produce broadcast contents but it will be necessary also for consumers. Multimedia file system for CE devices should support non-linear editing operations efficiently. File system should support file indexing by content-aware attributes, for example the director and actor/actress of a movie clip.

Reliability On occurrence of system failures(*e.g.* power failures, reset, and bad blocks), file system for CE devices should be recovered to a consistent state. Implementation of a reliable file system from scratch or API extension to existing file system are difficult and requires long stabilization effort. In case of appending multimedia API extension (*e.g.* non-linear edit operations), reliability can be a main concern.

Portability Conventional file systems have dependency on underlying operating system. Since CE devices manufacturers usually use various operating systems, file system should be easily ported to various OSes.

Maintainability Consumer devices are diverse and the requirements for file system are slightly different from device to device. Moreover, file system requirements are time-varying. Maintainability is desired to reduce cost of file system upgrade and customization.

In this paper, we propose a multi-media file system architecture to provide all the above requirements. A research prototype named as "XPRESS"(eXtensible Portable Reliable Embedded Storage System) is implemented on Linux. XPRESS is a user-level database-assisted file system. It uses a transactional Berkeley database as XPRESS metadata store. XPRESS supports zero-copy non-linear edit (cut & paste) operations. XPRESS shows performance in terms of bandwidth and IO latencies which is competitive to Linux XFS. This paper is organized as follows. Architecture of XPRESS is described in section 2. XPRESS's database schema is presented in section 3. Space allocation is detailed in section 4. Section 5 gives experimental results and discussions. Related works are described in section 6. We conclude and indicate future works in section 7.

2 Architecture

File system consistency is one of important file system design issue because it affects overall design complexity. File system consistency can be classified into metadata consistency and data consistency. Metadata consistency is to support transactional metadata operations with ACID (atomicity, consistency, isolation, durability) semantics or a subset of ACID. Typical file system metadata consist of inodes, directories, disk's free space information, and free inodes information. Data consistency means supporting ACID semantics for data transactions as well. If a data transaction to update a portion of file is aborted due to some reasons, the data update is complete or data update is discarded at all.

There have been several approaches of implementing file system consistency; log structured file system [11] or journaling file system [2]. In log structured file system, all operations are logged to disk drive. File system is structured as logs of consequent file system update operations. Journaling is to store operations on separate journal space before updating the file system. Journaling file system writes twice (to journal space and file system) while log structured file system does write once. However journaling approach is popular because it can upgrade existing non-journaling file system to journaling file system without losing or rewriting the existing contents.

Implementation of log structured file system or journaling is a time consuming task. There have been a lot of researches and implementation of ACID transactions mechanism in database technology. There are many stable open source databases or commercial databases which provide transaction mechanism. So we decide to exploit databases' transaction mechanism in building our file system. Since we aimed to design a file system with performance and reliability altogether, we decided not to save file contents in database. Streaming performance can be bottlenecked by database if file contents are stored in db. So, only metadata is stored in database while file contents are stored to a partition. Placement of file contents on the data partition is guided by multimedia optimized allocation strategy.

Storing file system metadata in database makes file system upgrades and customization much easier than conventional file systems. XPRESS handles metadata thru database API and is not responsible for disk layout of the metadata. File system developer has only to design high level database schema and use well defined database API to upgrade existing database. To upgrade conventional file systems, developer should handle details of physical layout of metadata and modify custom data structures.

XPRESS is implemented in user level because user level implementation gives high portability and maintainability. File system source codes are not dependent on OS. Kernel level file systems should comply with OS specific infrastructures. Linux kernel level file system compliant to VFS layer cannot be ported easily to different RTOSes. There can be a overhead which is due to user level implementation. XPRESS should make system calls to read/write block device file to access file contents. If file data is sparsely distributed, context switching happens for each extent. There was an approach to port existing user level database to kernel level[9] and develop a kernel level database file system. It can be improved if using Linux AIO efficiently. Current XPRESS does not use Linux AIO but has no significant performance overhead.

Figure 1 gives a block diagram of XPRESS file system. XPRESS is designed to be independent of database. DB Abstraction Layer (DBAL) is located between metadata manager and Berkeley DB. DBAL defines basic set of interfaces which modern databases usually have. SQL or complex data types are not used. XPRESS has not much OS dependencies. OSAL of XPRESS has only wrappers for interfacing block device file which may be different across operating systems.

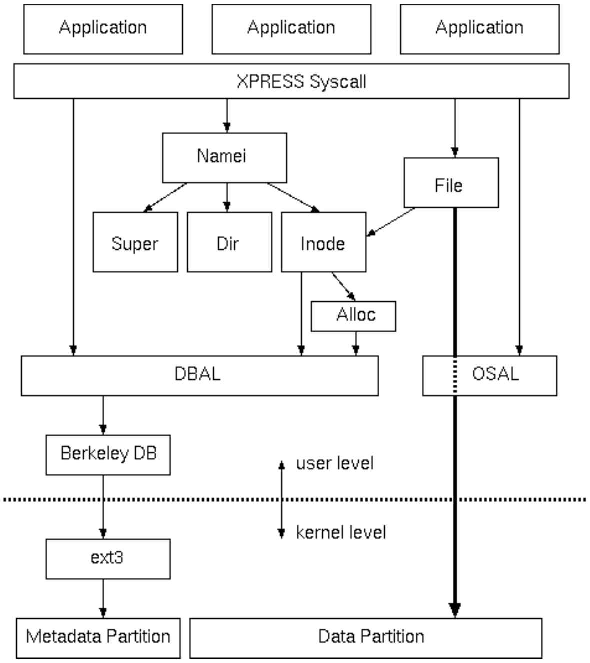


Figure 1: Block Diagram of XPRESS File System

There are four modules handling file system metadata; superblock, directory, inode, and alloc modules. Directory module implements the name space using a lookup database (`dir.db`) and `path_lookup` function. The function looks up the path of a file or a directory through recursive DB query for each token of the path divided separated by '/' character. The flowing is the simple example for the process of this function. Superblock module maintains free inodes. Inode module maintains the logical-to-physical space mappings for files. Alloc module maintains free space.

In terms of file IO, file module calls inode module to updates or queries `extents.db` and reads or writes file contents. The block device file corresponding to the data partition (say "`/dev/sda1`") is opened for read/write mode at mount time and its file descriptor is saved in environment descriptor, which is referred to by every application accessing the partition. XPRESS does not implement caching but relies on Linux buffer cache. XPRESS supports both direct IO and cached IO. For cached IO, the block device file is opened without `O_DIRECT` flag while opened with `O_DIRECT` in case of direct IO.

Support for using multiple partitions concurrently, XPRESS manages mounted partitions information using environment descriptors. A environment descriptor has information about a partition and its mount point, which is used for name space resolution. Since all DB resources (transaction, locking, logging, etc) corresponding to a partition belongs to a DB environment and separate logging daemon is ncessary for separate environment, a new logging daemon is launched on mounting a new partition.

3 Transactional Metadata Management

3.1 Choosing Database

As the infrastructure of XPRESS, database should conform to the following requirements. First, it should have transactional operation and recovery support. It is important for implementing metadata consistency of XPRESS. Second, it should be highly concurrent. Since file system is used by many threads or processes, database should support and have high concurrency performance as well. Third, it should be light-weight. Database does not have to support many features which are unnecessary for file system development. It only has to provide API necessary for XPRESS efficiently. Finally, it should be highly efficient. Database implemented as a separate process has cleaner interface and maintainability; however library architecture is more efficient.

Berkeley DB is an open source embedded database that provides scalable, high-performance, transaction-protected data management services to applications. Berkeley DB provides a simple function-call API for data access and management. Berkeley DB is embedded in its application. It is linked directly into the application and runs in the same address space as the application. As a result, no inter-process communication, either over the network or between processes on the same machine, is required for database operations. All database operations happen inside the library. Multiple processes, or multiple threads in a single process, can all use the database at the same time as each uses the Berkeley DB library. Low-level services like locking, transaction logging, shared buffer management, memory management, and so on are all handled transparently by the library.

Berkeley DB offers important data management services, including concurrency, transactions, and recovery. All of these services work on all of the storage structures. The library provides strict ACID transaction semantics, by default. However, applications can relax the isolation or the durability. XPRESS uses relaxed semantics for performance. Multiple operations can be grouped into a single transaction, and can be committed or rolled back atomically. Berkeley DB uses a technique called two-phase locking to support highly concurrent transactions, and a technique called write-ahead logging to guarantee that committed changes survive application, system, or hardware failures.

If a BDB environment is opened with a recovery option, Berkeley DB runs recovery for all databases belonging to the environment. Recovery restores the database to a clean state, with all committed changes present, even after a crash. The database is guaranteed to be consistent and all committed changes are guaranteed to be present when recovery completes.

DB	unit	key	data	structure	secondary index
super	partition	partition number	superblock data	RECNO	-
dir	partition	parent INO/file name	INO	B-tree	INO (B tree)
inode	partition	INO	inode data	RECNO	-
free space	partition	PBN	length	B-tree	length(B tree)
extents	file	file offset	PBN/length	B-tree	-

Table 1: Database Schema, INO: inode number, RECNO: record number, PBN: physical block number

3.2 Database Schema Design

XPRESS defines five databases to store file system metadata and their schema is shown in Table 1. Each B-tree database has a specific key-comparison function that determines the order in which keys are stored and retrieved. Secondary index is used for performance acceleration.

Superblock DB The superblock database, `super.db`, stores file system status and inode bitmap information. As overall file system status can be stored in just one record, and inode bitmap also needs just a few records, this database has RECNO structure, and does not need any secondary index. Super database keeps a candidate inode number, and when a new file is created, XPRESS uses this inode number and then replaces the candidate inode number with another one selected after scanning inode bitmap records.

Directory DB The dir database, `dir.db`, maps directory and file name information to inode numbers. The key used is a structure with two values: the parent inode number and the child file name. This is similar to a standard UNIX directory that maps names to inodes. A directory in XPRESS is a simple file with a special mode bit. As XPRESS is user-level file system, it does not depend on Linux VFS layer. As a result it cannot use directory related caches of Linux kernel (*i.e.*, dentry cache), instead, database cache will be used for that purpose.

Inode DB The inode database, `inode.db`, maps inode numbers to the file information (*e.g.*, file size, last modification time, etc.). When a file is created, a new inode record is inserted into this database, and when a file is deleted, the inode record is removed from this database. XPRESS assigns inode numbers in an increasing order and upper limit on the number of inodes is determined when creating the file system. It will be possible to make secondary indices for `inode.db` for efficiency (*e.g.*, searching files whose size is larger than 1M). But currently no secondary index is used.

Free Space DB The free-space database, freespace.db, manages free extents of the partition. Initially free-space database has one record whose data is just one big extent which means a whole partition. When a change in file size happens this database will update its records.

Extents DB A file in XPRESS consists of several extents of which size is not fixed. The extents database, `extents.db`, maps file offset to physical blocks address of the extent including the file data. As this database corresponds to each file, its life-time is also same with that of a file. The exact database name is identified with an inode number; `extents.db` is just database file name. This database is only dynamically removable while all other databases are going on with the file system.

3.3 Transactional System Calls

A transaction is atomic if it ensures that all the updates in a transaction are done altogether or none of them is done at all. After a transaction is ether committed or aborted, all the databases for file system metadata are in consistent. In multi-process or multi-thread environment, concurrent operation should be possible without any interference of other operations. We can say this property isolated. An Operation will have already been committed to the databases or is in the transaction log safely if the transaction is committed. So the file system operations are durable which means file system metadata is never lost.

Each file system call in XPRESS is protected by a transaction. Since XPRESS system calls are using the transactional mechanism provided by Berkeley DB, ACID properties are enabled for each file system call of XPRESS. A file system call usually involves multiple reads and updates to metadata. An error in the middle of system call can cause problems to the file system. By storing file system metadata in the databases and enabling transactional operations for accessing those databases, file system is kept stable state in spite of many unexpected errors.

An error in updating any of the databases during a system call will cause the system call, which is protected by a transaction, to be aborted. There can be no partially done system calls. XPRESS ensures that any system call is complete or not started at all. In this sense, a XPRESS system call is atomic.

Satisfying strict ACID properties can cause performance degradation. Especially in file system, since durability may not be strict condition, we can relax this property for better performance. XPRESS compromises durability by not syncing the log on transaction commit. Note that the policy of durability applies to all databases in an environment. Flushing takes place periodically and the cycle of flushing can be configurable. Default cycle is 10 seconds.

Every XPRESS system call handles three types of failures; power-off, deadlock, and unexpected operation failure. Those errors are handled according to cases. In case of power-off we are not able to handle immediately. After the system is rebooted, recovery procedure will be automatically started. Deadlock may occur during transaction when the collision between concurrent processes or threads happened. In this case, one winning process access database successfully and all other processes or threads are reported deadlock. They have to abort and retry their transactions. In case of unexpected operation failure, on-going transaction is aborted and system calls return error to its caller.

3.4 Non-linear Editing Support

Non-linear editing on A/V data means cutting a segment of continuous stream and inserting a segment inside a stream. This is required when users want to remove unnecessary portion of a stream; for example, after recording two hours of drama including advertisement, user wants to remove the advertisement segments from the stream. Inserting is useful when a user wants to merge several streams into one title. These needs are growing because consumers capture many short clips during traveling somewhere and want to make a title consisting of selected segments of all the clips.

Conventional file systems does not consider this requirement, so to support those operations, a portion of file should be migrated. If front end of a file is to be cut, the remaining contents of the file should be copied to a new file because the remaining contents are not block aligned. File systems does assume block aligned starting of a file. In other words, they do not assume that a file does not start in the middle of a block. Moreover, a block is not assumed to be shared by different files. This has been general assumptions about had disk file system because disk is a block based device which means space is allocated and accessed in blocks. The problem is more complicated for inserting. A file should have a hole to accommodate new contents inside it and the hole may not be block-aligned. So there can be breaches in the boundaries of the hole.

We solve those problems by using byte-precision extents allocation. XPRESS allows physical extent not aligned with disk block size. After cutting a logical extent, the physical extents corresponding to the cut logical extent can be inserted into other file. Implementation of those operations involves updating databases managing the extents information. Data copy is not required for the editing operations, only extents info is updated accordingly.

4 Extent Allocation and Inode Management

In this section, the term "block" refers to the allocation unit in XPRESS. The block size can be configurable at format time. For byte-precision extents, block size is configured to one byte.

4.1 Extent Allocation

There were several approaches for improving contiguity of file allocation. Traditionally (FAT, ext2, ext3) disc free space was handled by means of bitmaps. Bit 0 at position n of the bitmap designates, that n-th disc block is free (can be allocated for the file). This approach has several drawbacks; searching for the free space is not effective and bitmaps do not explicitly provide information on contiguous chunks of the free space.

The concept of extent was introduced in order to overcome mentioned drawbacks. An extent is a couple, consisting from block number and length in units of blocks. An extent represents a contiguous chunk of blocks on the disk. The free space can be represented by set of corresponding extents. Such approach is used for example in XFS (with exception of real-time volume). In case of XFS the set of extents is organized in two B+ trees, first sorted by starting block number and second sorted by size of the extent. Due to such organization the search for the free space becomes essentially more efficient compared to the case of using bitmaps.

One way to increase the contiguity comes from increasing the block size. Such approach is especially useful for real time file systems which deal with large video streams in combination with idea of using special volume specifically for these large files. Similar approach is utilized in XFS for real-time volume. Another way to improve file locality on the disc is preallocation. This technique can be described as an allocation of space for the file in advance before the file is being written to. Preallocation can be accomplished whether on application or on the file system level. Delayed allocation can also be used for contiguous allocation. This technique postpones an actual disc space allocation for the file, accumulating the file contents to be written in the memory buffer, thus providing better information for the allocation module.

XPRESS manages free space by using extents. Allocation algorithm uses two databases: `freespace1.db` and `freespace2.db`, which collect the information on free extents which are not occupied by any files on the file system. The `freespace1.db` orders all free extents by left end and the `freespace2.db`, which is secondary indexed db of `freespace1.db`, orders all free extents by length. Algorithm tries to allocate entire length of req_len as one extent in the neighborhood of the last_block. If it fails, then it tries to allocate maximum extent within neighborhood. If no free extent is found within neighborhood, it tries to allocate maximum free extent in the file system. After allocating a free extent, neighborhood is updated and it tries the same process to allocate the remaining, which iterate until entire requested length is allocated. Search in neighborhood is accomplished in left and right directions using two cursors on `freespace1.db`. The neighborhood size is determined heuristically proportional to the req_len.

Pre-allocation is important for multi-threaded IO applications. When multiple IO threads try to write files, the file module tries to pre-allocate extents enough to guarantee disk IO efficiency. Otherwise, disk blocks are fragmented because contiguous blocks are allocated to different files in case that multiple threads are allocating simultaneously. Pre-allocation size of XPRESS is by default 32Mbytes which may be varying according to disk IO bandwidth. On file close, unused spaces among the pre-allocated extents are returned to free space database, which is handled by FILE module of XPRESS.

4.2 Inode Management

File extents are stored in `extents.db`. Each file extent is represented by logical file offset and corresponding physical extent. Both offset and physical extent are specified with byte precision in order to provide facilities for partial truncation that is truncation of some portion of the file from a specified position of the file with byte precision.

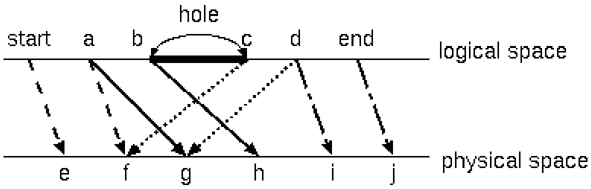


Figure 2: Logical to physical mapping

Let us designate a logical extent starting from the logical file offset a mapped to a physical extent [b,c] as [a,[b,c]] for explanations in the following. A logical file interval may contain some regions which do not have physical image on the disc; such regions are referred as holes. The main interface function of the INODE module is `xpress_make_segment_op()`. The main parameters of the function are type of the operation (READ, WRITE and DELETE) and logical file segment, specified by logical offset and the length.

`extents.db` is read on file access to convert a logical segment to a set of physical extents. Figure 2 shows an example of segment mapping. Logical segment [start, end] corresponds to following set of physical extents; {[start,[e,f]], [a,[g,h]], [b,[]], [c,[f,g]], [d,[i,k]]}, where the logical extent [b,c] is a file hole. In case of read operation to `extents.db`, the list of physical extents is retrieved and returned to the file module.

When specified operation is WRITE and specified segment contains yet unmapped area - that is writing to the hole or beyond end of file is accomplished, then allocation request may be generated after aligning the requested size to the block size since as was mentioned allocation module uses blocks as units of allocation. In Figure 3, blocks are allocated when writing the segment [start,end] of the file.

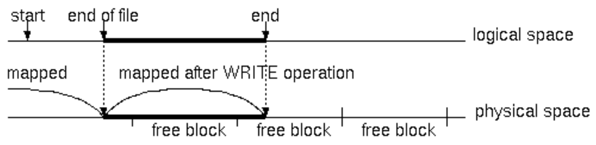
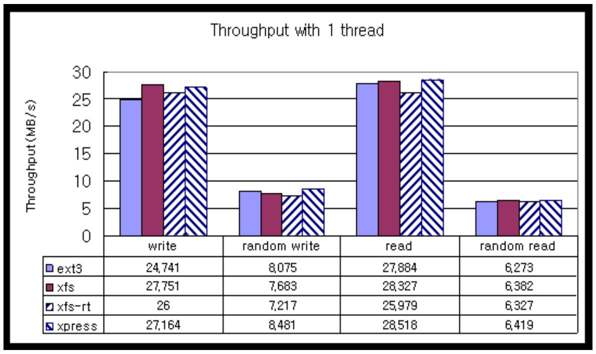


Figure 3: Block allocation details



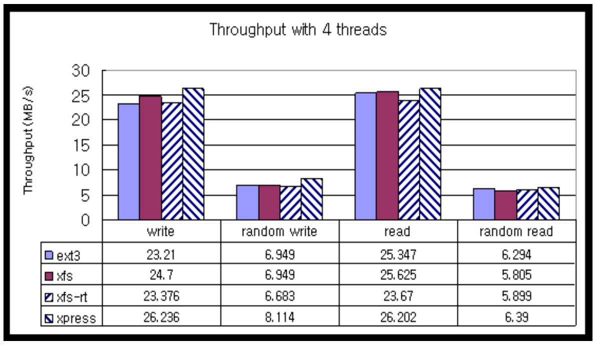
	write	random write	read	random read
▫ ext3	24,741	8,075	27,884	6,273
▪ xfs	27,751	7,683	28,327	6,382
▨ xfs-rt	26	7,217	25,979	6,327
▧ xpress	27,164	8,481	28,518	6,419

Figure 4: Throughput with 1 thread

Xpress allows to perform partial file truncate operation as well, as cut and paste operation. First operation removes some fragment of the file, while the latter also inserts the removed file portion into specified position of another file. On partial truncate operation, a truncated logical segment is left as a hole. On cut operation, logical segment mapping is updated to avoid hole. On paste operation, mapping is updated to avoid overwriting as well.

5 Experimental Results

Test platform is configured as following. Target H/W : Pentium4 CPU 1.70GHz with 128MB RAM and 30GB SAMSUNG SV3002H IDE disk
Target O/S : Linux-2.6.12
Test tools : `tiotest`(tiobench[1]), `rwrt`



	write	random write	read	random read
▫ ext3	23.21	6.949	25.347	6.294
▪ xfs	24.7	6.949	25.625	5.805
▨ xfs-rt	23.376	6.683	23.67	5.899
▧ xpress	26.236	8.114	26.202	6.39

Figure 5: Throughput with 4 threads

XPRESS consistency semantic is metadata journaling and ordered data writing which is similar to XFS file system and ext3 with ordered data mode. Hence we chose XFS and ext3 as performance comparison targets. As XFS file system provides real-time sub-volume option, we also used it as one of comparison targets. By calling it XFS-RT, we will distinguish it from normal XFS. `tiotest` is threaded I/O benchmark tool which can test disk I/O throughput and latency. `rwrt` is a basic file I/O application to test ABISS (Adaptive Block IO Scheduling System)[3]. It performs isochronous read operations on a file and gathers information on the timeliness of system responses. It gives us I/O latencies or jitters of streaming files, which is useful for analyzing streaming quality. We did not use XPRESS's I/O scheduling and buffer cache module because we can get better performance with the Linux's native I/O scheduling and buffer cache.

IO Bandwidth Comparison
Figure 4 and Figure 5 show the results of I/O throughput comparison for each file system. These two tests are conducted with `tiotest` tool whose block size option is 64KB which means the unit of read and write is 64KB. In both cases, there is no big difference between all file systems.

IO Latency Comparison
We used both `tiotest` and `rwrt` tool to measure I/O latencies in case of running multiple

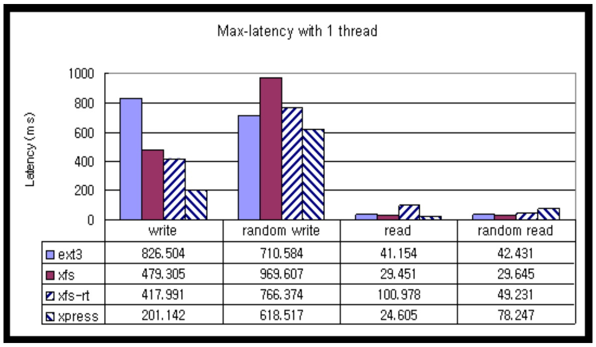


Figure 6: MAX latency with 1 thread

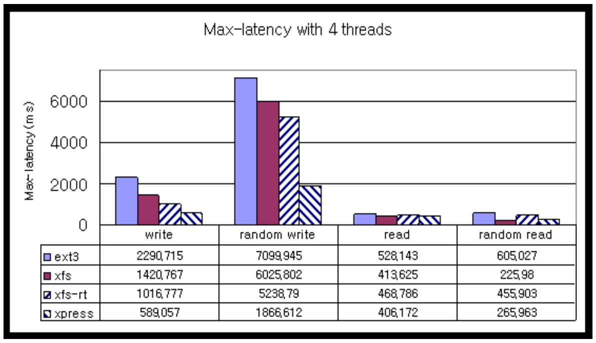


Figure 7: MAX latency with 4 threads

concurrent I/O threads. The `rwrt` is dedicated for profiling sequential read latencies and tiobench is used for profiling latencies of write, random write, read, and random read operations.

Figure 6 and Figure 7 show the maximum I/O latencies for each file system obtained from `tiotest`. The maximum I/O latency is a little low on XPRESS file system. In terms of write latency, XPRESS outperforms others while maximum read latencies are similar.

To investigate read case more, we conducted the `rwrt` with the number of threads from 1 to 8 and measure the read latencies for each read request. Each process tries its best to read blocks of a file. The results of these tests include the latencies of each read request whose size is 128Kbyte. Table 2 shows the statistics of the measured read latencies for 2, 4, and 8 threads. The average latencies are nearly the same for all experimented file systems. XPRESS show slightly smaller standard deviation than others and improvement regarding the maximum latencies. Please note that XPRESS maximum latency is 413 milliseconds while the max latency of XFS-RT is 524 milliseconds.

File System	Average	Std Dev	Max
	2 threads		
Ext3	8.47	26.03	512
XFS	8.62	25.15	239
XFS-RT	8.96	25.66	260
XPRESS	8.62	24.93	262
	4 threads		
Ext3	17.9594	72.70	877
XFS	17.8389	72.70	528
XFS-RT	18.7506	74.33	524
XPRESS	17.8467	71.79	413
	8 threads		
Ext3	36.38	166.47	1144
XFS	36.35	166.86	1049
XFS-RT	38.34	171.39	1023
XPRESS	36.33	166.38	923

Table 2: Read Latencies Statistics. All are measured in milliseconds.

Jitters during Streaming at a Constant Data Rate

The jitters performance is important from user's point of view because it leads to a low quality video streaming. We performed `rwrt` tool with ABISS I/O scheduling turned off. The `rwrt` is configured to read a stream with a constant specified data rate, say 3MB/sec or 5MB/sec. Table 3 shows jitter statistics for each file system when running single thread with 5MB/sec rate, four concurrent threads at 5MB/sec rates, and six concurrent threads at 3MB/sec rates, respectively. The results of these tests include the jitters of each read re-

File System	Average	Std Dev	Max
1 thread (5MB/s)			
Ext3	3.97	2.44	43
XFS	3.91	2.03	46
XFS-RT	3.94	1.79	25
XPRESS	3.89	1.92	40
4 threads (each 5MB/sec)			
Ext3	18.00	41.72	694
XFS	15.67	30.25	249
XFS-RT	19.22	34.63	267
XPRESS	17.93	38.83	257
6 threads (each 3MB/sec)			
Ext3	24.80	48.43	791
XFS	23.81	42.20	297
XFS-RT	26.57	43.48	388
XPRESS	23.66	45.31	337

Table 3: Jitter Statistics. All are measured in milliseconds.

quest whose size is 128Kbyte. Mean values of experimented file systems are nearly the same. XPRESS, XFS, and XFS-RT show the similar standard deviation of jitters which is much less than that of ext3.

Metadata and Data Separation
Metadata access patterns are usually small requests while streaming accesses are continuous bulk data transfer. By separating metadata and data partitions on different media, performance can be optimized according to their workload types. XPRESS allows metadata to be placed in separate partition which can be placed on another disk or NAND flash memory. Table 4 summarizes the effect of changing the db partition. This is test for extreme case since we used ramdisk, which is virtual in-memory disk, as a separate disk. However, we can identify the theoretical possibility of performance enhancement from this test. According to the result, the enhancement happens mainly on write test by 11%. We expect using separate partition will reduce latencies significantly, which are not shown here.

configuration	write	read
same disk	25.20	27.82
separate disk	28.20	28.53

Table 4: Placing metadata on ramdisk and data on a disk. All are measured in MB/s.

Non-linear Editing
To test cut-and-paste operation, we prepared two files each of which is of 131072000 bytes, then cut the latter half (= 65,536,000 bytes) of one file and append it to the other file. In XPRESS, this operation takes 0.274 seconds. For comparison, the operation is implemented on ext3 by copying iteratively 65536 bytes, which took 3.9 seconds. The performance gap is due to not copying file data but updating file system metadata (`extents.db` and `inode.db`). In XPRESS, the operation is atomic transaction and can be undone if it fails during the operation. However our implementation of the operation on ext3 does not guarantee atomicity.

6 Related Works

Traditionally file system and database has been developed separately without depending on each other. They are aimed for different purposes; file system is for byte streaming and database is for advanced query support. Recently there is a growing needs to index files with attributes other than file name. For example, file system containing photos and emails need to be indexed by date, sender, or subjects. XML document provides jit(just-in-time) schema to support contents based indexing. Conventional file system is lack of indexing that kind of new types of files. There has been a few works to implement database file systems to enhance file indexing; BFS[5], GnomeStorage[10], DBFS[6], kbdbfs[9], and WinFS[7]. Those are not addressing streaming performance issues. For video streaming, data placement on disk is important. While conventional database file systems resorts to DB or other file systems for file content storage, XPRESS controls placement of files content by itself.

Compared to custom multimedia file systems (*e.g.* [12], [8], [4]), XPRESS has a well-defined file system metadata design and implementation framework and file system metadata is protected by transactional databases. Appending multimedia extensions like indexing and non-linear editing is easier. Moreover since it is implemented in user level, it is highly portable to various OSes.

7 Conclusions and Future Works

In this paper we described a novel multimedia file system architecture satisfying streaming performance, multimedia extensions (non-linear editing), reliability, portability, and maintainability. We described detailed design and issues of XPRESS which is a research prototype implementation. In XPRESS, file system metadata are managed by a transactional database, so metadata consistency is ensured by transactional DB. Upgrade and customization of file system is easy task in XPRESS because developers don't have to deal with metadata layout in disk drive. We also implement atomic non-linear editing operations using DB transactions. Cutting and pasting A/V data segments is implemented by extents database update without copying segment data. Compared to ext3, XFS, and XFS real-time subvolume extension, XPRESS showed competitive streaming performance and more deterministic response times.

This work indicates feasibility of database-assisted multimedia file system. Based on the database and user level implementation, it makes future design change and porting easy while streaming performance is not compromised at the same time. Future works are application binary compatibility support using system call hijacking, appending contents-based extended attributes, and encryption support. Code migration to kernel level will also be helpful for embedded devices having low computing power.

References

[1] Threaded i/o tester. `http://sourceforge.net/projects/tiobench/`.

[2] M. Cao, T. Tso, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas. State of the art: Where we are with the ext3 filesystem. In *Proceeding of Linux Symposium*, July 2005.

[3] Giel de Nijs, Benno van den Brink, and Werner Almesberger. Active block io scheduling system. In *Proceeding of Linux Symposium*, pages 109–126, July 2005.

[4] Pallavi Galgali and Ashish Chaurasia. San file system as an infrastructure for multimedia servers. `http://www.redbooks.ibm.com/redpapers/pdfs/redp4098.pdf`.

[5] Dominic Giampaolo. *Practical File System Design with the Be File System.* Morgan Kaufmann Publishers, Inc., 1999. ISBN 1-55860-497-9.

[6] O. Gorter. Database file system. Technical report, University of Twente, aug 2004. `http://ozy.student.`

utwente.nl/projects/dbfs/dbfs-paper.pdf.

[7] Richard Grimes. Revolutionary file storage system lets users search and manage files based on content, 2004. http://msdn.microsoft.com/msdnmag/issues/04/01/WinFS/default.aspx.

[8] R. L. Haskin. Tiger Shark — A scalable file system for multimedia. *IBM Journal of Research and Development*, 42(2):185–197, 1998.

[9] Aditya Kashyap. *File System Extensibility and Reliability Using an in-Kernel Database*. PhD thesis, Stony Brook University, 2004. Technical Report FSL-04-06.

[10] Seth Nickell. A cognitive defense of associative interfaces for object reference, Oct 2004. http://www.gnome.org/~seth/storage/associative-interfaces.pdf.

[11] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[12] Philip Trautman and Jim Mostek. Scalability and performance in modern file systems. http://linux-xfs.sgi.com/projects/xfs/papers/xfs_white/xfs_white_paper.%html.