



# Achieving Breakthrough Performance with Tree-based File Systems

Using the right file system is one of the most critical factors in achieving system performance, and also the most often overlooked. This is true for desktops, servers and even more so for embedded devices. The amount of data stored and processed on devices is increasing rapidly, initially driven by consumer devices like Smartphones, MP3/video players, DVR, etc. This trend is now carrying over to other segments of embedded industry including Industrial, Military, and Automotive. Customers expect highly responsive devices regardless of the amount of data stored. As ever larger amounts of data bog down system performance, device manufacturers compensate by adding faster processors and more memory, which in turn increases device cost and impacts battery life.

In this paper, we will look at a more efficient way of achieving the device responsiveness – improving data management performance by using a highly optimized file system. The paper will first evaluate file system architectures based on certain key constructs that have significant impact on its performance and then provide an overview of a new file system that employs these constructs.

## Introduction to File System Architectures

There are several ways of classifying file system architecture. In this paper we will focus on three which have significant impact on the performance of embedded devices: data reliability architecture, disk allocation methodology, and metadata architecture.

## Data Reliability Architecture

Many devices operate in environments where uncontrolled power shutdown is a strong possibility, so file system developers over the years have employed different technical constructs to add resilience against data corruption when uncontrolled system shutdown occurs.

The most commonly used file system for embedded devices today, FAT, originated in the desktop environment where the risk of unexpected power interruption was minimal and therefore was not a critical requirement. FAT-based systems rely on a utility like CHKDSK to check the file system after every such event and determine if any corruption occurred. When corruption was found CHKDSK must be run in 'fix mode' but it does not guarantee recovery from all corruption issues.

Embedded developers have created many different methods of adding reliability measures to FAT, with varying degree of success. Unfortunately, the lack of reliability in its core architecture means that these measures will never assure of 100% logical file system reliability.

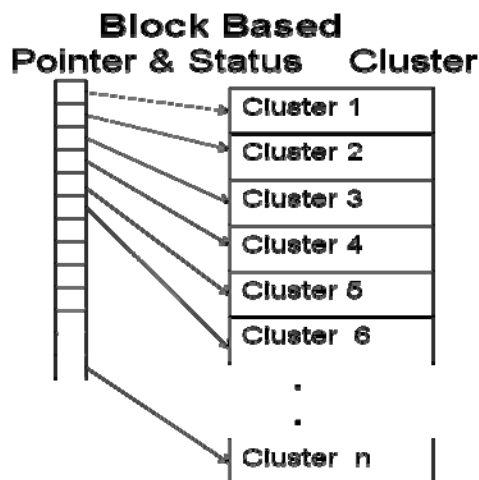
To address file system reliability in embedded systems, developers first tried the technique of *journaling*. File systems in this category include ext3, Jfile system, Reiserfile system, and Xfile system. These file systems were originally developed for use in Linux server environments, and were later adopted by embedded developers to address the issues of power loss and system crashes.

Journaling file systems do provide an improved level of data reliability, but also introduce overhead, the enemy of performance. In addition, the workings of the journal itself are usually abstracted from the application developer, making it difficult to control how journal transactions are impacting system performance.

Another category called *transactional* file systems provides reliability, but unlike journaling file systems, were specifically designed for resource-constrained embedded devices and require no CHKDSK or journal. Datalight Reliance Nitro is an example of such a file system, as is Microsoft Tex-FAT.

## Disk Allocation Methodology

Depending on how file systems allocate storage space for files, they can be categorized into two types – block-based and extent-based. Block-based file systems allocate space in



blocks which correspond directly to the block architecture provided by the underlying block device drivers or some multiple; those blocks are sometimes referred to as

## Performance Impact of Data Reliability Architecture

**FAT:** With simplistic architecture and low overhead, FAT based file systems have fast sequential I/O operations. Random I/O performance is significantly slower. Write operation speeds vary depending on media. Most FAT-based file systems do not perform well on resident flash memory.

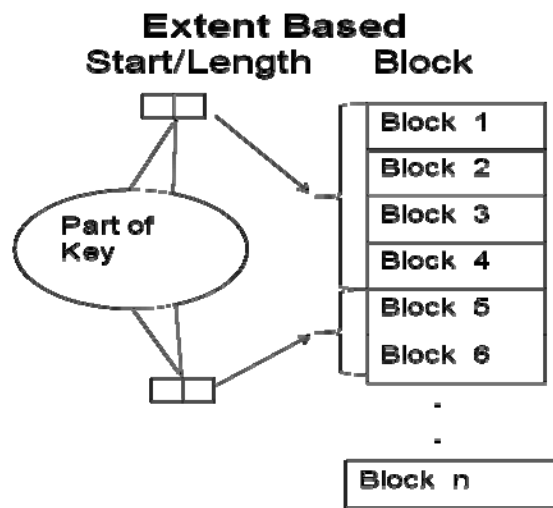
**Journaling file systems** provide decent performance in environments where resources are not limited, and require journal validation when loading. This can have a measurable impact on device boot times.

**Transactional file systems:** Performance depends on implementation. Datalight Reliance Nitro is architected for resource-constrained devices and provides fast mount times (can be mounted with just 3-5 logical block reads). Reliance also has special modes that allow it to perform faster on resident flash memory when used with Datalight FlashFX Pro.

	FAT	Journaling	Datalight Reliance
Mount Times	Med	Slow	Fast
Read Speeds	Fast	Fast	Fast
Write Speeds	Fast	Med	Fast

sectors. When requesting an allocation, the file system assigns blocks as required. While optimal for small or very fragmented files, this strategy uses a lot of metadata space, creating the need for allocation units of one or more sectors just to reduce the size of the tables to a manageable level. To accommodate large contiguous files, many allocation units must be designated, which has a negative impact on performance.

In contrast, extent-based file systems allocate a set of blocks (called as extent) when writing to a particular file. An extent consists of a data pair: a starting location and the extent length.



For example, (10, 20) would indicate a file starting at block 10 with an extent size of 20 blocks. Extents require smaller metadata overhead for large chunks

of file data, which improves performance. Random file access performance does not suffer as much when compared to a *FAT chain*, and extent-based systems typically also provide stronger performance for large files and sequential operations. On the downside, they are known to suffer in performance on smaller files and random operations, unless some specific optimizations are put in place.

## Performance Impact of Disk Allocation Methodology

**Block-based:** Random I/O operations can be faster than extent-based, but large contiguous files result in multiple allocation units, slowing overall performance.

**Extent-based:** Sequential reads and writes are faster because files are assigned contiguously, and therefore are less fragmented.

	Block-based	Extent-based
Sequential I/O	Med	Fast
Random I/O	Med	Med
Sustained performance	Med*	Fast

*\*(due to fragmentation)*

## Metadata Architecture

The most important category of tasks that a file system performs is metadata management. Metadata is the data associated with a file that describes it and allows for various ways to access the file. Examples of metadata include file path, date created, file size, permissions, etc. Given the intensive nature of metadata activity, how a file system organizes and manages metadata can have a huge impact on its performance.

The most common metadata architecture is *linear allocation*. In this method, the file system uses a linear array to store metadata information. This makes the file system simple and works fairly well for a system with a small number of files. The limitations of linear allocation start to show significant impact as the number of files in a system grows. Accessing large number of files through linear traversal of a folder takes a long time, leading to a serious degradation in performance.

Because of the problems with linear allocation, most modern file systems are moving to *tree-based allocations* for metadata management.

Tree-based allocations are a faster way to organize and find files, based on grouping information into addressable chunks, and assigning keys that direct the system where to find them. Basically, a tree can be simplified into a specialized array. Each array entry consists of a key and data pair. The array will be sorted in ascending order based on the key. The chain of leaf nodes can be thought of as an array. The intermediate nodes that point to the leaf nodes assist in quickly finding something in the leaf nodes. To store data in this array, a key is created that describes how the data is to be interpreted as well as what to associate the data is with.

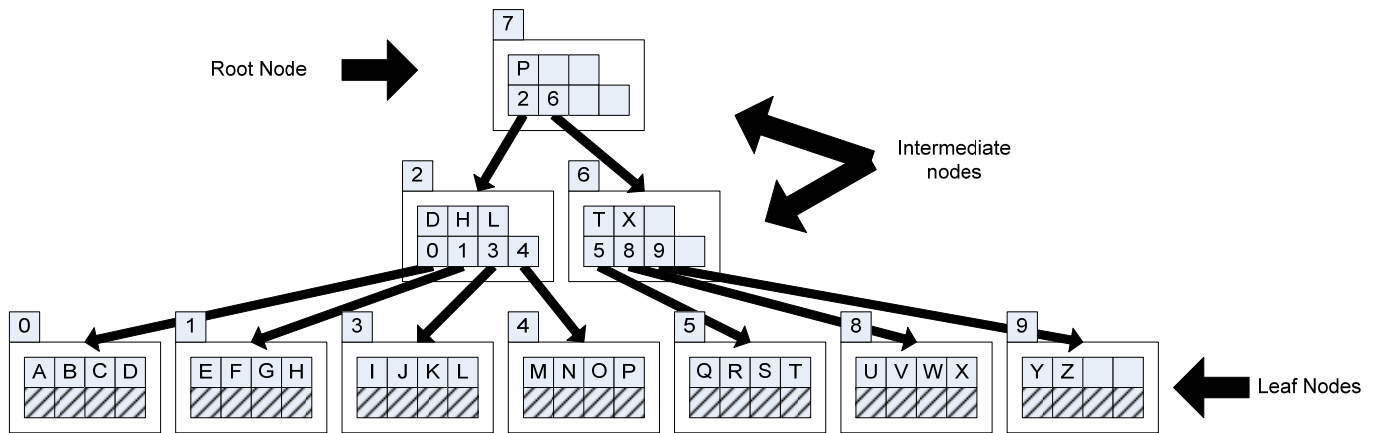
## Performance Impact of Metadata Architecture

**Linear Allocation** works well for a small number of files, but slows down significantly as the number of files grows.

**Tree-based Allocation** groups information into addressable chunks and assigns keys telling the system where to find them.

Performance benefit is especially pronounced when dealing with a large number of files.

	Linear Allocation	Tree Allocation
I/O Throughput (few files)	Fast	Fast
I/O Throughput (many files)	Med	Fast
File Operations (few files)	Med	Fast
File Operations (many files)	Very Slow	Fast



The Reliance Nitro file system is based on a highly specialized version of the tree-based approach, and as we will show in the architecture section later in this paper, this variation of tree-based architecture provides significant performance benefits.

## Reliance Nitro Overview

Before we dive deeper into the unusual architecture of Reliance Nitro, it is essential to understand the architecture of its predecessor, Reliance. This is because the strength of Reliance – 100% logical reliability against corruption, fast mount times, dynamic configurability of file system and ease of porting, are all carried forward in Reliance Nitro.

## Datalight Reliance Background

The Reliance file system has two distinct states, the *committed* (on-disk) state and the *working* state. The committed state is found on the media at initialization time. It is also the state of the file system as written to the media at the last completed transaction point.

Reliance uses a concept called the *metaroot* to refer to the start of the disk state. The metaroot is the base of a hierarchical structure of file metadata that represents everything stored on the disk. The committed state and the working state each have their own metaroot.

The working state consists of the file system's logical structures in memory, data in the disk cache, and blocks on disk that have been written from the disk cache. A critical point to understand is that even though the working state will consist of some data that is already written to disk, the ***committed disk state is not modified***. The working state only writes to blocks that are considered free.

## Executing a Metaroot Transaction Point

When a metaroot transaction point is performed, the disk cache is “flushed” so that all the user data and metadata from the working state (except the metaroot) is written to disk. After this is completed, the updated metaroot is written as an atomic operation. Once the working state’s metaroot is successfully written to disk, it becomes the committed state, and the previous committed state becomes the new working state.

At all times during the course of building the working state, the committed state on the media remains unchanged. The working state never writes anything to disk that coincides with blocks that the committed state is using, but rather only writes to free blocks. The benefit of maintaining two distinct states is that at any time during this process, the power may be lost without corrupting or losing anything from the committed state. Only operations from the interrupted working state will be lost.

## System Startup Logic

At system startup time, Reliance examines the two metaroot blocks to determine which one represents the valid committed state. If both metaroots happen to be valid, then the most recent metaroot is used as the committed state. This is a very quick operation because the two metaroots blocks are single logical disk blocks. Other file systems must replay a journal or scan the entire media with utilities such as CHKDSK, scandisk, or FSCK, to repair problems. Reliance requires no such utilities; rebuilding of its file structure is automatic.

## Improvement Potential

While Reliance provides a balanced combination of reliability and performance for embedded devices, there is always room for improvement. Building a file system for the next generation of embedded products required both incremental improvements and radical rethinking of how metadata was stored and managed. Our success factors included:

1. Accommodating the increasing storage capabilities of devices, particularly larger average file sizes. Extent-based file systems provide much better overall performance for larger files than block-based file system.
2. Finding an alternative to linear directory trees, which get increasingly inefficient as number of files increase.
3. Maintaining the reliability of transaction points while finding a way to mitigate any performance penalty. We had to sustain 100% reliability while reducing the overhead of a metaroot transaction point.
4. Optimizing I/O counts and improving code execution speed.

## Reliance Nitro

In creating Reliance Nitro, we set out to employ the best combination of elements discussed so far in this paper:

1. Transactional architecture for
  - a. 100% logical data reliability
  - b. Fast mounts
  - c. Fast I/O throughput
2. Extent-based to improve random I/O
3. Tree-based metadata management for fast file operation on volumes with a large number of files
4. *Delta Transactions* to reduce transaction point overhead

Since the original Reliance is both transactional and extent-based, and we've already described these attributes, let's focus on the new aspects of Reliance Nitro, namely its tree-based structure and use of delta transactions, both key factors in the performance benefit achieved by Reliance Nitro on embedded devices.

## Tree-based Metadata Management

The benefits that tree-based metadata structures have over linear ones have been discussed to a certain extent. Reliance Nitro builds on those benefits with a specific implementation that applies tree-based architecture to both the directory tree and the allocation tree.

Below is a snapshot of the way tree-structure is used in Reliance:

1. **Directory Tree:** The purpose of the directory tree is to associate a name with a unique file number within a parent directory. This association is implemented via a tree structure, which allows for a deterministic look-up time and near constant performance.
2. **Allocation Tree:** The purpose of the allocation tree is to associate allocated blocks with a file as well as to store information about the file itself. No longer is a single block allocated to manage a file. This metadata block is used by many files.

## Tree Root Node

Initially, each tree begins with a node known as the *RootNode*. Once each leaf node becomes full, an intermediate node is added. Intermediate nodes can point to multiple leaf or intermediate nodes.

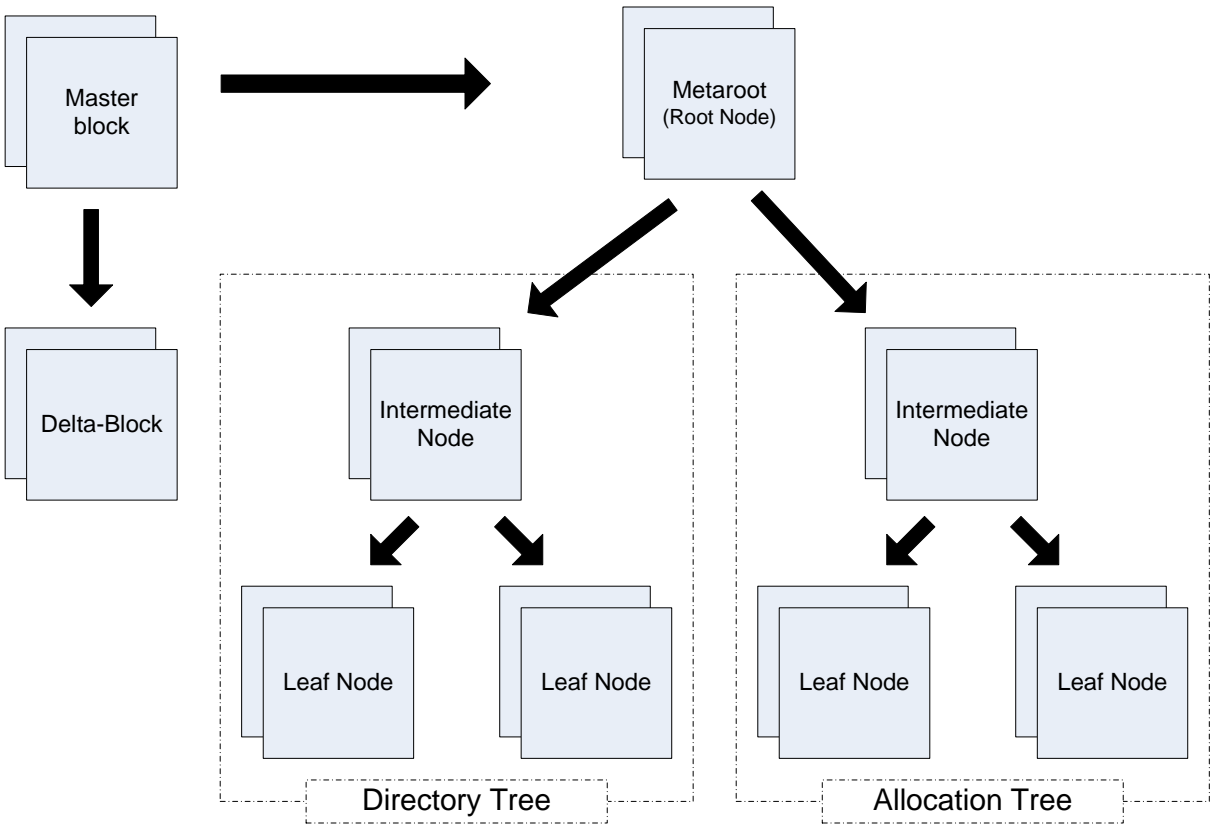
## Leaf Node

Leaf nodes are where the data associated with a key is actually stored.

## Intermediate Node

Intermediate nodes provide a method for the tree to grow dynamically. The data associated with intermediate node keys contain a logical block number for a leaf node or intermediate node:

# Nitro Architecture



Block Size in Bytes	512	1024	2048
Keys per Intermediate node	41	84	169
Keys per leaf node	15	31	63

Tree Depth	Number of Keys		
Level 1	15	31	63
Level 2	615	2,604	10,647
Level 3	25,215	218,736	1,799,343
Level 4	1,033,815	18,373,824	304,088,967

## Performance Advantages

For 512 bytes block size:

- On a linear file system (like TFAT), nodes traversed to access file #25,215 =  $25,215/8 = 3,151$
- On a tree-based file system like Reliance Nitro, nodes traversed to access file #25,215 = 3
- For cases where there are more than 17 files on the device, a tree-based file system will perform better than a linear one (assumes a 512 byte sector).

## Delta Transactions

As described earlier, a transactional file system employs the concept of a transaction point (sometimes called a metaroot transaction), where all uncommitted operations are written to the disk and the working and committed states are interchanged. This operation involves writing several metadata blocks including the metaroots for each state, a substantial source of system overhead with a potentially adverse effect on system performance. To mitigate the performance penalty of a transaction point, Reliance Nitro introduces a special data structure known as the *Delta-Block*.

A Delta-Block transaction allows a transaction to occur by only writing a single metadata block that encompasses all the metadata changes since the last transaction. The Delta-Block contains all the updates to the already transacted state of the media. So in essence, the Delta-Block records the difference between committed state and the working state. There are two Delta-Blocks employed in Reliance Nitro, each one is a single logical block. The logical block number of each Delta-Block is specified in the file system MasterBlock (the equivalent to the boot block in other file systems). The Delta-Blocks are written independently as part of a fast file system transaction. The two Delta-Blocks together enable transactions to be atomic. Either Delta-Block can be the current transaction metadata for the file system, but only one is valid at any time. The currently valid Delta-Block is never overwritten by the file system driver. Instead, when a Delta-Block transaction is being committed to the disk, the file system driver will write to the invalid Delta-Block. When the Delta-Block is completely written, its CRC and transaction counter can be used to identify it as the valid and most recent Delta-Block.

During a mount, the file system driver will compare the transaction counters to determine which Delta-Block is valid. If both Delta-Blocks pass a CRC check the 32-bit transaction counters will be sequential. The Delta-Block with a higher transaction counter is the valid one.

If a Delta-Block fails a CRC check or is otherwise unreadable, it is assumed that a power interruption or media removal occurred while performing a write operation. Thus, the incomplete Delta-Block is invalid and the other Delta-Block remains valid. Once the current Delta-Block is selected, the file system is mounted.

## Delta-Block Transaction

The Delta-Block intercepts calls to manipulate the allocation and directory trees. It stockpiles tree requests to add and remove keys (shown in Figure 6 and 7). If a transaction is called for and all of the “add” and “remove” requests made since the last transaction point can fit into a single Delta-Block, a Delta-Block transaction is performed. The Delta-Block is the only metadata block that needs to be committed as part of a Delta-Block transaction.

The Delta-Block has a finite amount of space to hold tree operations. Eventually the number of operations made to the tree will surpass the Delta-Block capacity at which time the changes are committed to the respective trees and a Metaroot transaction is performed as part of the next transaction.

When a Metaroot transaction is performed the Delta-Block will not be written. Instead the Delta-Block transaction counter will be set to the last good Delta-Block transaction counter. This will indicate on mount that the contents of the Delta-Block associated with the valid Delta-Block counter have already been incorporated into the tree.

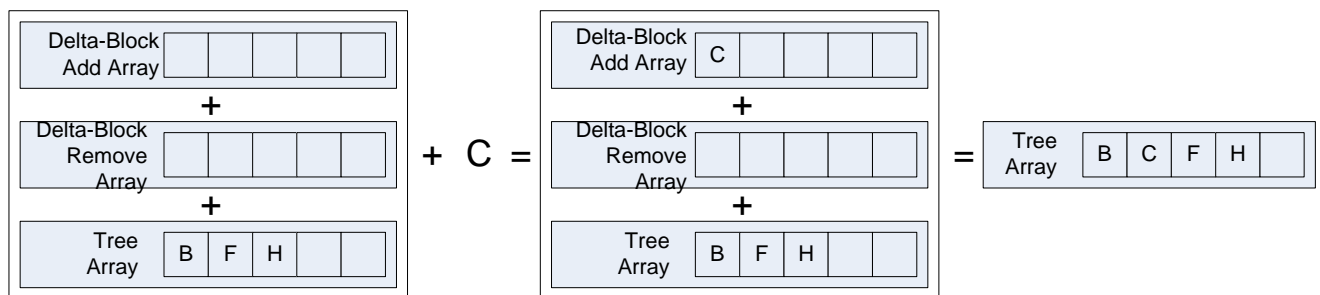


Figure 6: Delta-Block Add Operation

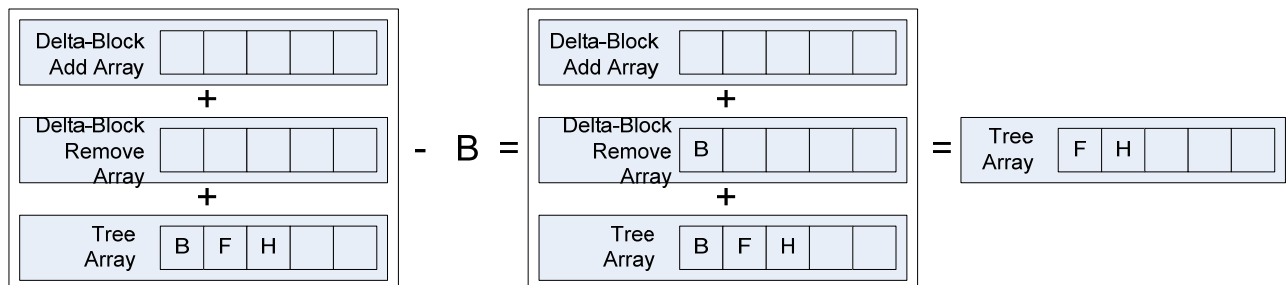


Figure 7: Delta-Block Remove Operation

## Performance Impact of Delta-block Transactions

Delta transactions significantly speed up the atomic transaction points in Reliance Nitro without compromising the 100% reliability benefit.

## Benchmarks

The architecture of Reliance Nitro brings significant performance improvement to embedded devices.

Microsoft T-exFAT was used for comparison with Reliance Nitro for the following reasons:

1. It is widely available as it is included with Windows CE 6
2. It employs the concept of transactions

	T-exFAT	Reliance Nitro
<b>Sequential read</b>	1405 KB/s	1447 KB/s
<b>Sequential write</b>	251 KB/s	1294 KB/s
<b>Sequential re-write</b>	291 KB/s	1434 KB/s
<b>Random read</b>	891 KB/s	910 KB/s
<b>Random write</b>	44 KB/s	137 KB/s

## Test Setup

The test media that was used was a 1GB SanDisk SD card using a 33Mb partition on a BSquare PXA320 development board. Datalight's platform-independent file system test suite (FSIO) was used to perform benchmarking using default configurations for both T-exFAT and Reliance Nitro

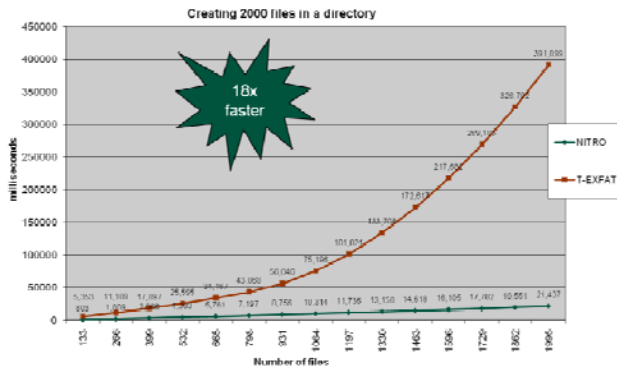
## File Operations

In these tests, we want to check the responsiveness of the file system in real world scenarios. The test benchmarks the time taken to Create, Open and Delete 15 samples of 133 files each (total 1995). This simulates scenarios such as:

1. Copying a lot of music files to the device (Create)
2. Sending a lot of text messages in a short amount of time (Create)
3. Indexing all emails stored on the device for faster search (Open)
4. Browsing through all available ringtones on the device (Open)
5. Browsing through pictures on a digital camera or photo frame (Open)
6. Clearing browser cache (Delete)

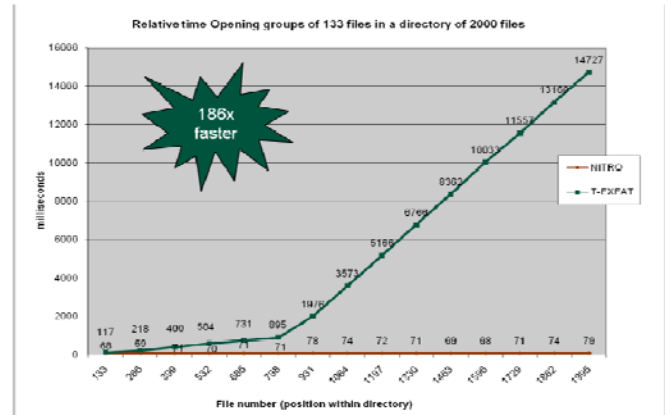
The graphs below plot the time taken to complete each sample set. Higher numbers mean the file system takes more time to complete the operation and performs more slowly.

## File Create



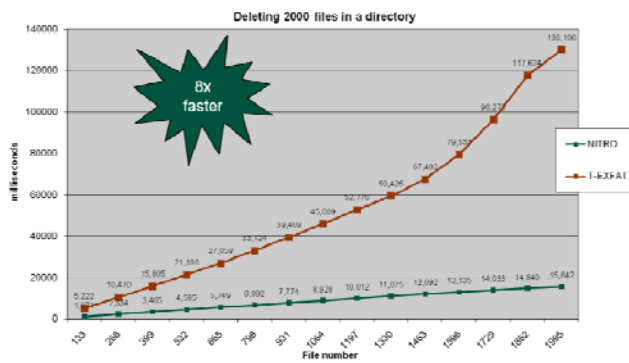
Depending on the number of files, Reliance Nitro can be 6 to 18 times faster than T-exFAT on creating files.

## File Open



Due to Reliance Nitro's tree-based allocation and directory design, the open times for each sample are very fast, such that in cases where there are fewer than 133 files in the directory, Reliance Nitro is approximately **80%** faster. At higher range, Reliance Nitro can be **186 times** faster than T-exFAT. Since 'file open' is one of the most often run operations by the file system, this can translate into significant performance gains for the device.

## File Delete



The above chart shows Reliance Nitro performance on File Delete operations compared to T-exFAT. Depending on the number of file in the directory, Reliance Nitro can be **2-8 times** faster than T-exFAT

## Conclusion

This paper highlights the impact architectural choices can have on the performance of a file system, specifically the differences between linear allocation and tree-based architecture. Many device developers underestimate the performance impact of the file system on overall device performance, and as the quality of the user's experience. Selecting a file system with an optimized architecture for your device scenarios is of paramount importance for device developers.

Reliance Nitro has been designed as an innovative variant of tree-based architecture to provide breakthrough performance and solid reliability for users of embedded devices. The throughput performance of Reliance Nitro is faster than other reliable file systems, but where it truly shines is in the area of file operations, beating other file systems by an order of magnitude in most cases.

## Appendix A: File System Basics

This white paper assumes some familiarity with file system basics. Basic file system definitions and concepts are outlined in this section.

A **file system** is a way to organize, store, retrieve, and manage information on a permanent storage medium such as a hard disk or flash memory.<sup>4</sup>

Each file system has a “**block size**.” The block size is defined to be the smallest unit that a file system can write. Everything a file system does is composed of operations done on blocks

A file system block is a logical unit rather than a physical unit. The logical block size of a file system is either the same size or a multiple of the sector size of the underlying storage medium. Selecting the right logical block size is a compromise between wasting as little disk space as possible and minimizing the number of blocks that have to be allocated to store a file.

**User data** is the named piece of information contained in a file. This piece of information may be any of the following: text such as a letter, text such as program source code, a database, or a graphic image.

**Metadata** is a piece of information about a file. Metadata includes file attributes like file name as well as other information such as its owner, creation time, size, and date of last modification.

An **i-node** is a location where a file system stores metadata about a file. The i-node also provides a pointer to the contents of the file on disk.

“**Sector size**” or block size is the granularity that the storage medium can read or write. The block or sector size of most modern hard disks is 512 bytes. Flash memory management software manages flash memory so that it appears as a hard drive with 512-byte sectors.

A file system **directory** is a way to name and organize files. The main purpose of a directory is to organize a list of files in a human readable way and to connect the name in the directory with the associated files.

**Basic file system operations** include initialize, mount, unmount, create a file, open a file, read a file, create a directory, write to files, read files, delete files, rename files, open directories, and read directories. These operations are fairly self explanatory with the exception of initialization, mounting, and unmounting which are defined below.

**Mounting a file system** consists of several tasks: accessing a raw storage device, reading the masterblock and other file system metadata, and then preparing the file system for access to a volume. A part of this preparation is verifying that the file system is valid. An alternate term for a valid file system is a “clean” or “consistent” file system meaning that the metadata and the user data are consistent with each other.

**Unmounting of a file system** involves flushing out to disk all in-memory state associated with the volume. Once all the in-memory data (data in RAM or other non-volatile memory) is written to the volume, the volume is said to be “clean.”

