Workload Dependent Performance Evaluation of the Btrfs and ZFS Filesystems

Dominique A. Heger
DHTechnologies, Austin, Texas
dheger@dhtusa.com

1.0 Introduction

The UNIX and Linux operating systems alike already provide a vast number of filesystems to the user community. In general, one of the major IO challenges faced today is scalability, especially in the lights of the very large storage IO subsystems being utilized by most IT data centers. Filesystems have to scale in their ability to address, and efficiently manage large IO storage subsystems, as well as in their ability to effectively detect, manage, and repair errors and faults in the IO channel. Two of the newer filesystem technologies that aim at addressing today's IO challenges are SUN's ZFS and the Linux filesystem Btrfs. The goal of this paper is threefold. 1st, to introduce the design, architecture, and features of Btrfs and ZFS, respectively. 2nd, to compare the two filesystem architectures and to elaborate on some of the key *performance by design* concepts that are embedded into the IO frameworks. 3d, to conduct an actual empirical analysis, comparing the performance behavior of Btrfs and ZFS under varying workload conditions, utilizing an identical hardware setup for all the benchmarks. As an additional reference point, the factual IO performance is compared to the behavior of a more traditional filesystem implementation, ext4.

2.0 Filesystem Technologies - Btrfs & ZFS

The Btrfs filesystem was originally designed and developed by Chris Mason at Oracle, and was just recently merged into the 2.6.29 Linux kernel [1]. Despite being part of the Linux kernel now, the Btrfs filesystem is still undergoing major development changes. As a matter of fact, the actual disk format has not yet been finalized. In a nutshell, the Btrfs filesystem provides and supports writable snapshots, sub-volumes, object-level mirroring and stripping, data checksums, compression, online filesystem checks and defragmentation features [2]. With solid-state drive (SSD) technologies becoming increasingly common, there is also an SSD optimized mode that aims at increasing the performance potential of Btrfs.

Sun's ZFS file system addition to Solaris 10, as well as the OpenSolaris release of ZFS, is considered an innovative, ground-up redesign of the traditional UNIX file systems [4]. *Note: As of writing this paper, there was also a FUSE (filesystem in user space) ZFS Linux project that was still somewhat active. As of 2012, a native ZFS Linux solution is available.* Engineers from Sun and the open source community have drawn their ideas for the new ZFS file system from some of the best products currently on the market. Network Appliances' snapshots and Veritas' object-based storage management, transactions, and checksumming features influenced ZFS. Further, the engineers involved in the project contributed their own new ideas and expertise to develop a streamlined, cohesive approach to file system design [5]. With ZFS, Sun addresses the important issues of integrity and security, scalability, as well as the difficulty of user administration that often impacts some of the other UNIX file systems. Btrfs and ZFS are considered a technological step-up from filesystems such as ext3, XFS, or UFS, and both provide similar features and functionalities. As ZFS was released prior to Btrfs (and hence is more mature), the conclusion may be drawn that Btrfs *borrowed* some of the ideas. While overlapping functionalities are common in UNIX and Linux filesystems, some of the similarities in ZFS and Btrfs are more of an artifact of the age of the filesystems, as both were designed and developed just recently. Having said that, there are still significant differences (not only in the actual IO performance behavior) that are elaborated on in this paper. In other words, while the features are similar, the implementation techniques are completely different.

3.0 Architecture & Design – ZFS

Among the most important components of any file system are data integrity and security. It is imperative that information on disk do not suffer from bit rot, silent corruption, or even malicious or accidental tampering. In the past, file systems have had various problems overcoming these challenges and providing reliable and accurate data. Most UNIX based file systems build on older technologies (such as UFS or

HFS) and overwrite blocks when modifying in-use data [10]. If a power failure occurs during a write operation, the data is corrupted, and the file system may lose some of the block pointers. To circumvent that issue, the fsck command scans for dirty blocks and reconnects the information where possible. Unfortunately, the fsck operation scans the entire file system, resulting in runtimes measured in minutes or hours to complete (depending on the size of the file system). To accelerate the recovery process, many file systems incorporate a journaling or logging feature [11]. But in the case of a corrupted journal, fsck still has to be invoked to repair the file system. Further, most journaled file systems do not log actual user data (an overhead issue).

For reliability purposes, organizations moved to disk or file system mirroring (by utilizing a volume management solution). In case some corruption occurs, one half of the mirror resyncs to the other (even if only a few blocks are in question). Not only is IO performance degraded during the resync operation, but also the system can not always accurately predict which copy of the data is uncorrupted. Sometimes the system chooses the wrong mirror to trust, and the bad data overwrites the good data. To address the performance issues, some volume managers introduced dirty region logging (DRL). DRL allows to only resync the areas where write operations were in flight at the time of the power loss. This addresses the performance issue, but it does not address the problem with detecting which side of the mirror has the valid data. ZFS tackles these issues by processing transaction-based copy-on-write modifications and constantly checksumming every in-use block in the file system [5].

3.1 Copy-on-write transactional model

The ZFS design represents a combination of a file system and a volume manager [4]. The file system commands require no concept of the underlying physical disks (because of the storage pool virtualization). All of the high-level interactions occur through the data management unit (DMU), a concept that is similar to a memory management unit (MMU) for disks instead of memory (see Figure 1). All of the transactions committed through the DMU are atomic, and therefore the data is never left in an inconsistent state.

In addition to being a transaction-based file system, ZFS only performs copy-on-write operations [6]. This implies that the blocks containing the data (that is in use) on disk are never modified. The changed information is written to alternate blocks, and the block pointer to the data in use is only moved once the write transactions are completed. This scenario holds true all the way up the file system block structure to the top block, which is labeled the *uberblock* [4]. In the case that the system encounters a power outage while processing a write operation, no corruption occurs as the pointer to the good data is not moved until the entire write operation completes. It has to be pointed out that the pointer to the data is the only entity that is moved. This eliminates the need for journaling or logging, as well as for an fsck or mirror resync when a machine reboots unexpectedly.

To summarize, ZFS uses a copy-on-write, transactional object model. All block pointers within the file system contain a 256-bit checksum of the target block, which is verified when the block is read. Blocks containing active data are never overwritten in place; instead, a new block is allocated, modified data is written to it, and then any metadata blocks referencing it are similarly read, reallocated, and written. To reduce the overhead of this process, multiple updates are grouped into transaction groups, and an intent log is used when synchronous write semantics are required.

3.2 End-to-End Checksumming

To avoid accidental data corruption, ZFS provides memory-based end-to-end checksumming [4]. Most checksumming file systems only protect against bit rot, as they use self-consistent blocks where the checksum is stored with the block itself. In this case, no external checking is done to verify validity. This style of checksumming will not prevent issues such as:

- Phantom write operations where the write is dropped
- Misdirected read or write operations where the disk accesses the wrong block
- DMA parity errors between the array and server memory (or from the device driver). The issue here is that the checksum only validates the data inside the array
- Driver errors where the data is stored in the wrong buffer (in the kernel)
- Accidental overwrite operations such as swapping to a live file system

With ZFS, the checksum is not stored in the block but next to the pointer to the block (all the way up to the *uberblock*). Only the *uberblock* contains a self-validating SHA-256 checksum. All block checksums are done in memory, hence any error that may occur up the tree is caught. Not only is ZFS capable of identifying these problems, but in a mirrored or RAID-Z ZFS configuration, the data is actually self-healing.

3.3 ZFS Scalability

While data security and integrity is paramount, a file system has to perform well [11]. The ZFS designers either removed or greatly increased the limits imposed by modern file systems by using a 128-bit architecture, and by making all metadata dynamic. ZFS further supports data pipelining, dynamic block sizing, intelligent prefetch, dynamic striping, and built-in compression to improve the actual IO performance behavior.

3.4 The 128-Bit Architecture

Current trends in the industry reveal that the disk drive capacity roughly doubles every nine months [12]. If this trend continues, file systems will require 64-bit addressability in about 10+ years. Instead of focusing on 64-bits, the ZFS designers implemented a 128-bit file system. This implies that the ZFS design provides 16 billion times more capacity than the current 64-bit file systems (such as XFS). According to Jeff Bonwick (ZFS chief architect) *"Populating 128-bit file systems would exceed the quantum limits of earth-based storage. You couldn't fill a 128-bit storage pool without boiling the oceans."*

3.5 Dynamic Metadata

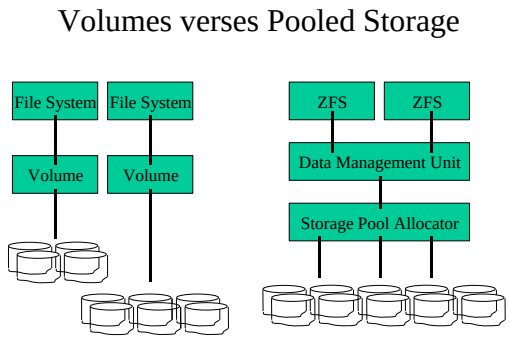
In addition to being a 128-bit based solution, the ZFS metadata is 100 percent dynamic. Hence, the creation of new storage pools and file systems is extremely efficient. Only 1 to 2 percent of the write operations to disk are metadata related, which results in large (initial) overhead savings. To illustrate, there are no static inodes, therefore the only restriction to the number of inodes that (theoretically) can be used is the size of the storage pool.

3.6 RAID-Z

ZFS implements RAID-Z, a solution that is similar to RAID-5, but uses a variable stripe width to circumvent the RAID-5 write hole issue. With RAID-5, write operations are performed to 2 or more independent devices, and the parity block is written as a component of each stripe [4]. Since these write operations are non-atomic, a power failure between the data and parity transactions results in the possibility of data corruption. Some vendors address this issue with parity region logging (a concept that is similar to dirty region logging, only for the parity disk) or via battery-backed NVRAM solutions.

With the NVRAM solution, the data is written into the NVRAM, and afterwards the data and parity writes are made to disk. Finally, the contents of the NVRAM is released. NVRAM is expensive, and can sometimes turn into the IO bottleneck component. RAID-Z reflects a data/parity scheme that utilizes a dynamic stripe width based approach. Hence, every block reflects a RAID-Z stripe, regardless of the block-size. This implies that every RAID-Z write represents a *full-stripe write operation*. Combined with the copy-on-write transactional semantics of ZFS, this approach completely eliminates the RAID-5 write hole issue. Hence, RAID-Z may perform more efficiently than a RAID-5 setup, as there is no read-modify-write cycle. ZFS also supports *RAID-Z2* (double parity) and the *mirrored storage pool* (which can be used for RAID-10 setups), respectively [14].

Figure 1: ZFS Storage Pool



The challenge faced by RAID-Z revolves around the *reconstruction process* though. As the stripes are all of different sizes, an *all the disks XOR to zero* based approach (such as with RAID-5) is not feasible. In a RAID-Z environment, it is necessary to traverse the file system metadata to determine the RAID-Z geometry. It has to be pointed out that this technique would not be feasible if the file system and the actual RAID array were separate products. Traversing all the metadata to determine the geometry may be slower than the traditional approach though (especially if the storage pool is used-up close to capacity).

Nevertheless, traversing the metadata implies that ZFS can validate every block against the 256-bit checksum (in memory). Traditional RAID products are not capable of doing this; they simply XOR the data together. Based on this approach, RAID-Z supports a self-healing data feature. In addition to whole-disk failures, RAID-Z can also detect and correct silent data corruption. Whenever a RAID-Z block is read, ZFS compares it against its checksum. If the data

disks do not provide the expected checksum, ZFS (1) reads the parity, and (2) processes the necessary *combinatorial reconstruction* to determine which disk returned the bad data. In a 3d step, ZFS repairs the damaged disk, and returns *good data* to the application.

3.7 Storage pools

Unlike a traditional UNIX file system that either resides on a single device or uses multiple devices and a volume manager, ZFS is implemented on top of virtual storage pools, labeled the *zpools* (see Figure 1). A pool is constructed from virtual devices (*vdevs*), each of which is either a raw device, a mirror (RAID-1), or a RAID-(10|Z) group [14]. The storage capacity of all the *vdevs* are available to all of the file systems in the *zpool*. To limit the amount of space a file system can occupy, a quota can be applied, and to guarantee that space will be available to a specific file system, a reservation can be set.

3.8 ZFS Snapshots

The ZFS copy-on-write model has another powerful advantage; when ZFS writes new data, instead of releasing the blocks containing the old data, it can instead retain them, creating a snapshot version of the file system. ZFS snapshots are created quickly, as all the data comprising the snapshot is already stored. This approach is also space efficient, as any unchanged data is shared among the file system and its snapshots. Writable snapshots (clones) can also be created, resulting in 2 independent file systems that share a set of blocks. As changes are made to any of the clone file systems, new data blocks are created to reflect those changes, but any unchanged blocks continue to be shared, no matter how many clones exist.

3.9 Dynamic striping

Dynamic striping across all devices to maximize throughput implies that as additional devices are added to the *zpool*, the stripe width automatically expands to include them, thus all disks in a pool are used, which balances the write load across them.

4.0 Variable block sizes

ZFS utilizes variable-sized blocks of up to 128 kilobytes [13]. The currently available ZFS code allows tuning the maximum block size, as certain workloads do not perform well with large blocks. Automatic tuning to match workload characteristics is contemplated. If compression is enabled, variable block sizes are used. If a block can be compressed to fit into a smaller block size, the smaller size is used on the disk to use less storage and improve IO throughput. This is accomplished though at the cost of increased CPU usage for the compression and decompression operations.

4.0 Architecture & Design – Btrfs

The Btrfs was designed and implemented with simple, proven, and rather well known components (such as ‘modified/optimized’ btree structures [3]) as the main building blocks.

4.1 Dynamic inode allocation

Dynamic inode allocation implies that when creating the filesystem, only a few inodes are established. Based on the actual workload, additional inodes are created and allocated ad hoc. Btrfs inodes are stored in *struct btrfs_inode_item*. The Btrfs inodes store the traditional stat data for files and directories. The Btrfs inode structure is relatively small, and will not contain any embedded file data or extended attribute data [2].

4.2 Btrfs Snapshots & Subvolumes

Btrfs subvolumes reflect a (named) btree that stores files and directories [3]. Subvolumes can be given a quota of blocks, and once this quota is reached, no new write operations are allowed. All of the blocks and file extents inside of the subvolumes are reference counted (to allow snapshot operations). Currently, up to 264 subvolumes can be created per Btrfs filesystem.

In Btrfs, snapshots are identical to subvolumes, but their root block is initially shared with another subvolume [1]. When the snapshot is taken, the reference count on the root block is increased, and the copy-on-write Btrfs transaction mechanism ensures changes made in either the snapshot or the source subvolume are private to that root. Snapshots are writable (active updates to the snapshot are possible), and they can be snapshotted again (any number of times). If read-only snapshots are required, their block quota is set to 1 at creation time [2]. To reiterate, the Btrfs snapshots can basically be utilized either as backups or as fast emergency copies of the existing data set. Btrfs provides vast flexibility in regards to snapshot functionalities.

4.3 Btrfs Copy-On-Write

Data, as well as metadata in Btrfs are protected with copy-on-write logging [3]. Once the transaction that allocated the space on disk has committed, any new write operations to that logical address in the file or btree will go to a newly allocated block, and block pointers in the btrees and superblocks will be updated to reflect the new location. Btrfs also utilizes the copy-on-write mechanism in conjunction with snapshot updates.

4.4 Btrfs RAID Support

At the time of this report, Btrfs supports RAID-0, RAID-1, and RAID-10, respectively. Btrfs allows adding devices (physical disks) to the file system after the original filesystem has been created (the dynamic inode allocation feature is paramount to support this). Further, Btrfs allows for dynamically removing devices from an existing, mounted filesystem. Btrfs further provides file system check and defragmentation options that can be invoked while the filesystem is in use. From a metadata perspective, the following options are supported right now:

- RAID-0 - the metadata is appended across all devices
- RAID-1 - the metadata is mirrored across all devices
- RAID-10 the metadata is appended and mirrored across all devices
- Single – the metadata is placed on a single device

4.5 Encryption and Compression

With most filesystems on the market, add-ons can be utilized to provide encryption (such as the *dm-crypt* interface provided by recent 2.6 Linux kernels). Btrfs has encryption built into the filesystem framework. In addition to encryption, Btrfs further provides compression mechanisms focusing on saving disk space, and potentially improving IO performance (the *zlib* kernel capabilities are being used right now). Some of the main mount options available for Btrfs (that *may* have a rather significant impact on actual IO performance are:

- nodatacow – there is no data copy on write
- nodatasum – there is no data checksums
- compress – compression is turned on

5.0 Btrfs & ZFS – Performance in Design

ZFS as well as Btrfs reveal some design and implementation components that ultimately govern their respective IO performance. The next few paragraphs outline and elaborate on some of the features. In general, a volume manager represents a layer of software that groups a set of block devices (for protection and/or device aggregation purposes). A filesystem represents an abstraction layer that manages such a block device by utilizing a portion of physical memory. From a user space perspective, applications issue read and write requests to the filesystem, and the filesystem generates IO operations to (in this scenario) the block devices. ZFS collapses these 2 functions into a single entity.

ZFS manages a set of block devices (the leaf *vdev* components), normally groups them into so-called protected devices (such as RAID-Z), and aggregates the top-level *vdevs* into pool components. The top-level *vdevs* can be added to a pool at any time. Objects that are stored in a pool are dynamically striped onto the available *vdevs*. Associated with the pools, ZFS manages a number of lightweight filesystem objects. A ZFS filesystem can basically be described as a set of properties that are associated with a given mount point [6]. The properties of a ZFS filesystem include the quota (maximum size) and reservation (guaranteed size) as well as attributes such as compression [5]. The ZFS (as well as Btrfs) filesystems are labeled lightweight, as the settable filesystem properties can be adjusted dynamically.

To illustrate, the ZFS recordsize can be described on a per ZFS filesystem instance. ZFS files smaller than the recordsize are stored by utilizing a single filesystem block (FSB). The FSB is of variable length but is determined in multiple of a disk sector. Larger files are stored utilizing multiple FSB's, each FSB of recordsize bytes. Currently, the default FSB size equals to 128KB. In other words, the FSB reflects the basic unit as managed by ZFS (the checksum is applied to the FSB). In the case a file grows beyond the recordsize (a.k.a. multiple FSB's are required), adjusting the ZFS recordsize property will not impact the file in question. A copy of the file will inherit the tuned/adjusted recordsize. From a performance perspective, benchmarks utilizing a workload scenario where small updates (< 128KB) to large files are pervasive revealed that adjusting the recordsize to a smaller value significantly alleviates the pressure on the IO subsystem and improves IO performance.

5.1 Transaction Groups & Intend Logs

The basic mode of operation for ZFS write requests (with no synchronous semantics such as O_DSYNC or fsync), is described as absorbing the IO operation in a per-host system cache labeled as the Adaptive Replacement Cache (ARC). As there is only one host memory subsystem but potentially multiple ZFS pools, all the cached data from all pools has to be processed by a single ARC. That ARC itself manages the data by balancing a Most Frequently Used (MFU) and a Most Recently Used (MRU) approach. One of the ZFS ARC's interesting properties is that a scan of a large file does not destroy most of the cached data.

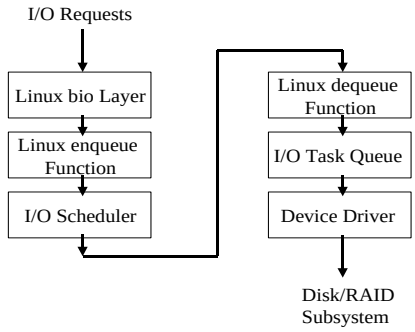
Each file write operation is associated with a certain transaction group (TXG). At regular intervals (the default equals to 5 seconds), each TXG will shutdown, and the pool will issue a sync operation for that group. A TXG may also be shutdown when the ARC indicates that there are too much dirty memory pages currently being cached. As a TXG closes, a new one immediately opens, and any potential file modifications (write operations) will be associated with the new TXG. If the active TXG is in a shutdown mode while a previous TXG is still in the process of syncing data to the storage, the user applications will be throttled back until the running sync operation completes. Hence, in a scenario where one TXG is in a sync state, while the other TXG is closed due to memory limitations or the 5-second clock (and is basically waiting to sync as well), the applications are throttled, waiting to write to a third TXG.

Such a scenario is only possible though due to a sustained saturation of the storage subsystem or some memory constraint situation. For file modification scenarios that require immediate data integrity (such as O_DSYNC or fsync), ZFS maintains a per-filesystem intent log (labeled the ZIL). The ZIL flags each (write) operation with a log sequence number. When a synchronous request is received (such as an fsync), the ZIL will commit the data blocks up to the last sequence number. When the ZIL is in the process of committing data, further commit operations will be stalled until the active ones are completed. This mechanism allows the ZIL to aggregate multiple (smaller) transactions, economizing on the number of necessary IO operations.

5.2 IO Scheduler

Next to the actual workload behavior and HW setup, the Btrfs IO performance is impacted (as any other Linux filesystem) by the Linux IO scheduler [8],[9]. To illustrate, the IO scheduler in Linux forms the interface between the generic block layer and the low-level device drivers. The block layer provides functions that are utilized by the file systems and the virtual memory manager to submit IO requests to block devices (see Figure 2). In other words, the Linux filesystems reside above the Linux IO schedulers. The filesystem IO requests are transformed by the IO scheduler and made available to the low-level device drivers. The device drivers consume the transformed requests and forward them (by using device specific protocols) to the actual device controllers that perform the IO operations. As prioritized resource management seeks to regulate the use of a disk subsystem by an application, the IO scheduler is considered an imperative kernel component in the Linux IO path. It is further possible to regulate the disk usage in the kernel layers above and below the IO scheduler. Adjusting the IO pattern generated by the file system or the virtual memory manager (VMM) is considered as an option. Another option is to adjust the way specific device drivers or device controllers consume and manipulate the IO requests. The various Linux 2.6 IO schedulers (CFQ, noop, deadline, and anticipatory) can be abstracted into a rather generic IO model.

Figure 2: Linux 2.6 IO Stack



The IO requests are generated by the block layer on behalf of threads that are accessing various file systems, threads that are performing raw IO, or are generated by virtual memory management (VMM) components of the kernel such as the *kswapd* or the *pdflush* threads. The producers of IO requests initiate a call to *__make_request(),* which invokes various IO scheduler functions such as *elevator_merge_fn()*. The *enqueue* functions in the IO framework intend to merge the newly submitted block IO unit (a *bio*) with previously submitted requests, and to sort (or sometimes just insert) the request into one or more internal IO queues. As a unit, the internal queues form a single logical queue that is associated with each

block device. At a later stage, the low-level device driver calls the generic kernel function *elv_next_request()* to obtain the next request from the logical queue. The *elv_next_request()* call interacts with the IO scheduler's dequeue function *elevator_next_req_fn()*, and the latter has an opportunity to select the appropriate request from one of the internal queues. The device driver processes the request by converting the IO submission into scatter-gather lists and protocol-specific commands that are submitted to the device controller. From an IO scheduler perspective, the block layer is considered as the producer of IO requests and the device drivers are labeled as the actual consumers. Please see [8] for a comprehensive discussion on the Linux 2.6 IO schedulers and their respective performance behavior.

ZFS (on Solaris) on the other hand reflects an IO construct where the IO scheduler is embedded into the filesystem framework [11]. ZFS maintains all the pending IO requests, but only issues a certain number (default equals to 35) at a time to the (disk) controllers. This allows the controllers to operate efficiently/effectively while never overflowing the IO queues. By limiting the IO queue size, in general, the service times of the individual disk requests can be kept at reasonable values.

When IO completes, the ZFS IO scheduler decides on the next one (priority based). Similar to the deadline Linux IO scheduler [8], the ZFS priority scheme is time based, where a read request is prioritized over a write requests that is issued in the last (approximately) 0.5 seconds. The fact that ZFS will limit each leaf devices IO queue to 35 is considered one of the reasons why the *zpool* should be constructed by utilizing *vdevs* that represent individual disks (or at least volumes that map to a small number of disks). Otherwise the ZFS self-imposed limits may become an artificial performance bottleneck.

5.3 IO System Calls

For ZFS, in the case that a read request can not be serviced by the ARC cache subsystem, ZFS will issue a *prioritized IO* for the data. In other words, even if the IO subsystem is processing a write intensive IO load at that time, there are actually only a maximum of 35 IO requests that can be outstanding. As soon as the set of 35 IO operations completes, the ZFS IO scheduler will issue the read request to the controller. However, to avoid write IO starvation, in scenarios where there is a write IO backlog, the write requests priority gets adjusted accordingly and the requests are processed. As already discussed, ZFS never over-writes active data on-disk, and always outputs complete records that are validated by a checksum.

Hence, to partially over-write a file record, ZFS has to have access to the corresponding data in memory. If the data is not available in the cache, ZFS will issue a read request prior to the write to partially modify the file record. With the data now in the cache subsystem, more write requests can target the same blocks. Before the actual write, ZFS will checksum the data. In the case of a full/complete record overwrite scenario, the read into cache step is not required. For ZFS, actual read-ahead scenarios are managed by the filesystem itself, whereas for Btrfs, the read-ahead framework embedded into the Linux 2.6 kernel is still being utilized.

5.4 Caching Scenarios

In any scenarios, IO operations are considered slow compared to the throughput potential of the CPU, cache, and the memory subsystems, respectively [15]. Hence, the only good and fast IO read request is considered as the one the system does not have to execute all the way down to the disks. While any filesystem (design) can decide where to place the write date on disk, read requests are generated by the user applications, and if the data is not available in memory, the application threads will have to stall. A workload behavior with small (mostly random) read requests may reflect one of the worst-case IO scenarios. In most circumstances, the response time for small random read requests is dominated by the seek time. The ZFS design addresses this issue by amortizing these operations and executing rather large IO requests (64KB by default). The larger request sizes may improve IO performance in scenarios where there is a good locality of reference. The actual data is loaded into a per-device cache (sized at 20MB). This device cache can be invaluable in economizing on the number of IO operations, but similar to the ZFS recordsize, if the inflated IO behavior causes a storage channel saturation, this Soft Track Buffer (STB) can act as an actual IO performance throttle.

6.0 Benchmark Environment

To assess and quantify the performance potential of the Btrfs and ZFS filesystems, respectively, a set of IO benchmarks were executed against a single SSA disk and a RAID setup consisting of 8 Seagate drives. To further position the status quo of Btrfs and ZFS, the same set of benchmarks were executed against

an ext4 setup [16]. For the Linux and OpenSolaris benchmarks, kernel versions 2.6.30 and Nexenta Core Platform 2, respectively were being used. For the Linux RAID benchmarks, a RAID-10 setup (with 8 disks) was configured. For all the Linux benchmarks, the CFQ IO scheduler was utilized. For the Solaris benchmarks, the ZFS *zpool* was created with 4 *vdevs* consisting of 2 disks per *vdev* (mirrored storage pool – RAID-10). The Btrfs, ZFS, and ext4 filesystems were created with the default options. The ext4 filesystem was mounted with the *extent* option. The hardware setup for all the conducted benchmarks can be summarized as:

- 2 Dual-Core Xeon Processors (2.5GHz)
- 16GB RAM (to avoid excessive caching scenarios, only 2GB (single disk) and 4GB (RAID) were used for the benchmarks)
- Single internal 10,000RPM 300GB SAS drive (serial attached SCSI)
- Areca ARC-1220 SATA PCI-express 8x controller with 8 Seagate 7,200RPM 500GB SATA II drives

All the empirical studied were executed by utilizing the Flexible FileSystem Benchmark (FFSB) IO benchmarking set [8],[15]. FFSB represents a sophisticated benchmarking environment that allows analyzing performance by simulating basically any IO pattern imaginable. The benchmarks can be executed on multiple individual file systems, utilizing an adjustable number of worker threads, where each thread may either operate out of a combined or a thread-based IO profile. Aging the file systems, as well as collecting systems utilization and throughput statistics is part of the benchmarking framework. For the single disk (4 concurrent IO threads), as well as the RAID setup (8 concurrent IO threads), the following benchmarks were conducted:

- *Sequential Read*, 40MB files, 4KB read requests
- *Sequential Write*, 100MB files, 4KB write requests
- *Random Read*, 40MB files, 4KB read requests, 1MB random read per IO thread
- *Random Write*, 40MB files, 4KB write requests, 1 MB random write per IO thread
- *Mixed*, 60% read, 30% write, 10% delete, file sizes 4KB to 1MB, 4KB requests

6.1 Benchmark Results

Tables A1 and A2 (see Appendix A) disclose the single disk and the RAID (raw) performance results for the 3 filesystems, respectively. Each benchmark was executed 10 times, and the results reflect the mean values over the sample set [12]. For all the conducted benchmarks, the coefficient of variation (CV) was less than 4%, hence the statement can be made that all the runs are reproducible with the resources at hand. For all the benchmarks, an efficiency value, representing the number of IO operations divided by the CPU utilization was calculated. In other words, the efficiency value fuses the throughput potential and the corresponding CPU demand/consumption.

6.2 Single Disk IO Performance

Based on the conducted benchmarks and the HW resources at hand, for the single disk setup, from a throughput perspective, the ZFS file system outperformed Btrfs in all but one (sequential write operations) workload scenarios. From an efficiency perspective, the ZFS filesystem provided substantially higher IO operations to CPU utilization ratios than Btrfs.

Figure 3: Single Disk - Sequential IO Throughput

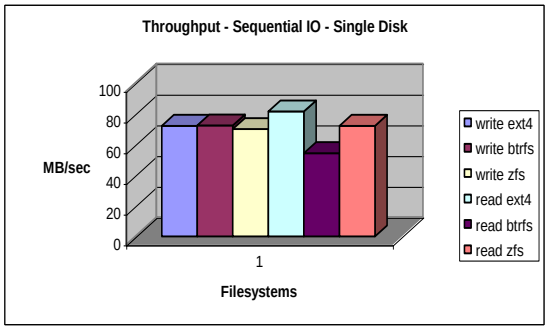
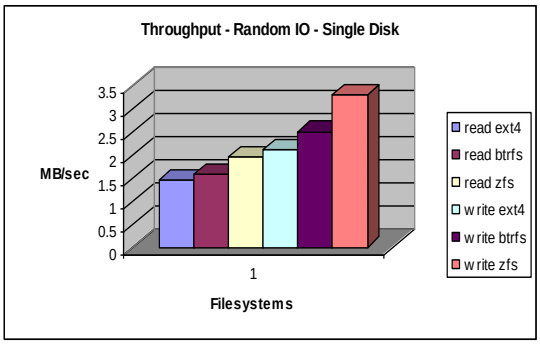


Figure 4: Single Disk – Random IO Throughput



Comparing the ZFS and the Btrfs filesystems to ext4, the ext4 filesystem outperformed the other 2 solutions for the sequential read and the mixed workload

scenarios. For all the conducted benchmarks, ext4 revealed the most promising IO operations to CPU utilization (efficiency) ratios.

The conducted single disk benchmarks further disclosed ZFS' high potential to perform well under random read and write workload conditions (see Figure 4). The study also showed that in a Linux 2.6 single disk environment (except for the sequential read and the mixed IO operation scenarios), the Btrfs and the ext4 filesystems behave similar from a throughput perspective (see Figures 3 & 5). From a CPU utilization point of view though, the Btrfs generates a much higher CPU demand to process the same workload, resulting into a much lower efficiency behavior (see Figure 6).

Figure 5: Single Disk – Mixed IO Throughput

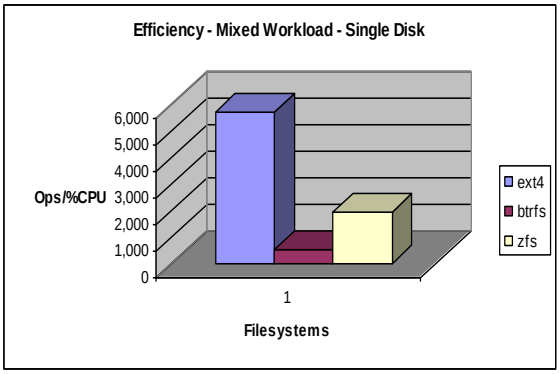
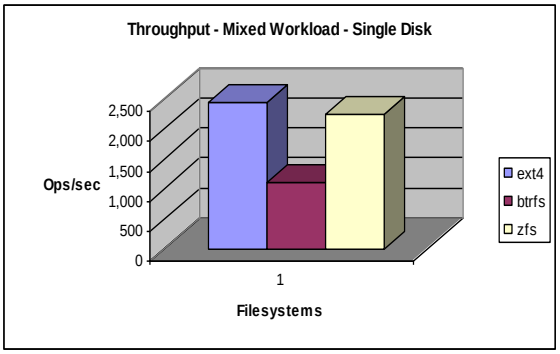


Figure 6: Single Disk – Efficiency Mixed IO



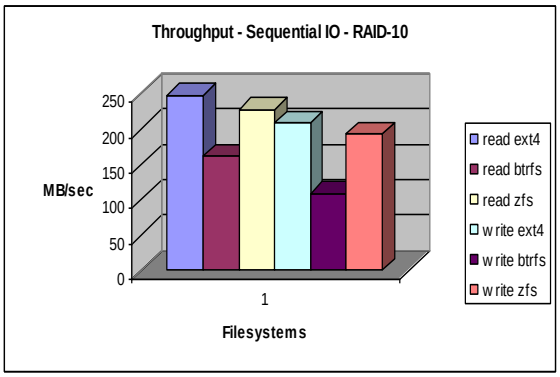
For the single disk setup, additional benchmarks utilizing the Btrfs mount options *nodatacow* and *nodatasum* were conducted. In most scenarios, the additional mount options resulted in only slightly better throughput results (see Section 7.0, IO Performance Conclusions, for more information). This observation holds true except for the mixed workload scenario where the mount options did not have a 'measurable' impact on the throughput behavior. As expected (see Benchmark Conclusions), the additional Btrfs mount options provided much better efficiency values though (up to 3 times compared to a Btrfs setup with no additional mount options). To further evaluate the efficiency behavior and the impact on actual performance, additional latency tests for the random read and random write workload scenarios were conducted. The efficiency values for ext4 and ZFS for the random read tests are 2,200 and 1,100, whereas the values for the random writes are 2,080 and 612, respectively. In both benchmarks though, ZFS outperformed ext4 from a throughput perspective.

The experienced throughput behavior was confirmed via some additional single disk latency tests that revealed average response time values for ext4 and ZFS of 26 milliseconds and 20 milliseconds for random read, and 29 milliseconds and 18 milliseconds for random write operations, respectively. In other words, while encountering a higher CPU demand, the throughput and the latency behavior of ZFS is superior (based on these random IO workload scenarios) compared to the ext4 setup.

6.3 RAID IO Performance

In the benchmarked RAID setup, except for the random read workload, the study disclosed a rather large throughput delta between Btrfs and ZFS/ext4. The conducted empirical studies revealed that in this RAID environment, the Btrfs filesystem was not able to provide a performance behavior that can compete right now with either ext4 or ZFS. Further, for all the RAID benchmarks, the tested Btrfs solution provided the lowest efficiency values.

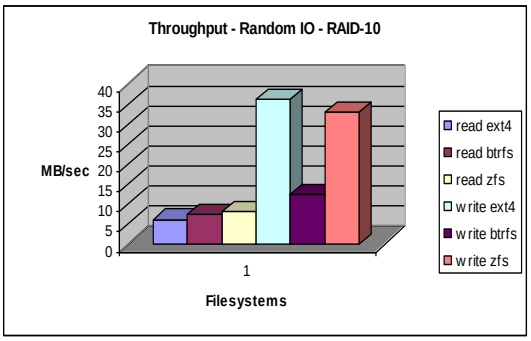
Figure 7: RAID - Sequential IO Throughput



The ZFS filesystem showed a very good random read and write performance behavior (Figure 8), whereas ext4 excelled in the sequential read and write, as well as in the mixed workload benchmarks (Figure 7 & 9). Across the

benchmark set, ext4 delivered the best efficiency values (Figure 10). To further investigate and analyze the rather large performance delta between ext4 and Btrfs, actual Linux kernel traces were taken and analyzed for both filesystem solutions.

Figure 8: RAID – Random IO Throughput



The traces taken on the Linux server system revealed that Btrfs is spending much more time in the Linux bio layer compared to ext4 (see Figure 1). Further, the comparison disclosed a higher lock contention (mutex – synchronization object) with Btrfs compared to ext4.

Figure 9: RAID – Mixed IO Throughput

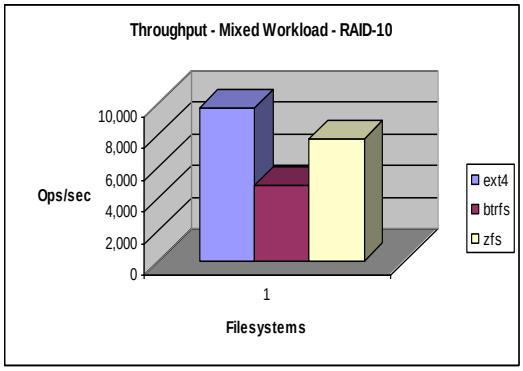
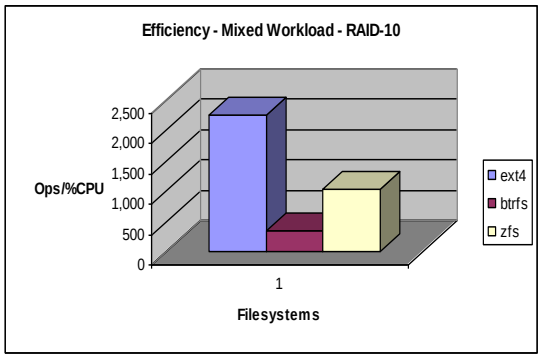


Figure 10: RAID – Efficiency Mixed IO



7.0 IO Performance – Conclusions

In addition to the benchmarks already discussed in this report, additional studies with different mount options for Btrfs and ZFS were conducted. The next few paragraphs in this report summarize some of the findings and conclusions (made based on the collected performance data).

Based on the conducted benchmarks (workload scenarios) and the HW at hand, for the single disk studies, utilizing the default Btrfs (mount) options resulted into good throughput numbers relative to ext4. The same holds true for ZFS, where the default mount options resulted into a performance behavior similar or better than the ext4 filesystem solution. For this study, additional (parallel) *dd* benchmarks were conducted to establish an upper IO performance bound (labeled the IO potential). Including a 10% performance buffer (due to protocol overhead), showed that in a single disk environment, for the sequential read and write workload scenarios, all 3 filesystems were able to *approach* the performance potential of the SSA disk drive. The only exception was the sequential read throughput of the Btrfs filesystem that only approached approximately 66% of the HW potential.

For the RAID benchmarks, Btrfs was not able to provide throughput results comparable to ext4 or ZFS. The ZFS performance behavior was better than Btrfs, but still trailed ext4 in all but 1 workload scenario (random read requests). Comparing the sequential read and write performance results to a (parallel) *dd* based upper IO performance bound (including a 10% protocol overhead), really disclosed some of the current limitations in Btrfs. The parallel *dd* benchmarks executed against the RAID system showed an upper IO throughput bound of approximately 295MB/second and 240MB/second for read and write IO requests, respectively. So to illustrate, ext4 was able to get up top 93% and 96% of the IO potential of the RAID system for sequential read and write IO requests. ZFS revealed results of 86% and 90% for read and write requests, whereas Btrfs only approached 61% and 51% of the IO throughput potential of the HW (including the discussed protocol overhead).

Disabling the data checksum and copy-on-write Btrfs features resulted into only slightly better throughput performance compared to the raw data shown in Appendix A. As these features reflect some of the major 'selling points' for Btrfs, losing the added protection of the data checksum and copy-on-write features is not worth the minimal throughput gain.

The conducted IO studies cleared reveled a much higher CPU demand (with the default mount options) for Btrfs and ZFS compared to ext4. Hence, on server systems where the workload is already CPU intensive, the higher CPU demand of ZFS or Btrfs (compared to a filesystem solution such as ext4) has to be taken into consideration while conducting a capacity analysis.

The study further outlined that some of the ZFS and Btrfs features are expensive from a CPU and memory perspective. In other words, to provide some of the ZFS and Btrfs features (that are not part of ext4), CPU and memory intensive operations revolving around checksum generation and validation, compression, encryption, as well as data placement and component load balancing are necessary. These features result into a higher CPU demand for ZFS and Btrfs, and based on the conducted studies, drive the efficiency values down compared to ext4. *The encountered Btrfs and ZFS CPU behavior was expected, and nicely supplements the design and architecture discussion in this report.*

8.0 Summary

The rapid development cycle, the on-going evolution process, all the inventions and the overall progress of the Linux operating system over the last few years have been remarkable. For a lot of companies though, one of the main aspects for not fully embracing the Linux operating system as the defacto OS has revolved around issues with the existing file systems. Many organizations require a filesystem solution that provides enterprise-level features, reliability, and scalability potentials similar to filesystems available for some UNIX server systems (such as J2 for AIX or ZFS for Solaris).

The conducted performance studies revealed that based on the benchmarked workload scenarios and the HW resources at hand, Btrfs has the potential to be developed into that defacto Linux filesystem. The study also disclosed though that the status quo, in a RAID setup, is not up to par yet compared to ext4 or ZFS solutions. It has to be pointed out that Btrfs already provides a vast number of nice features (similar to ZFS) that are necessary to become the Linux filesystem of choice. In conclusion, as Btrfs is still undergoing heavy design and development efforts, it has to be considered as not being ready for prime time yet (a statement that is in-line with comments made by the Btrfs community). To reiterate, Btrfs was just recently merged into the mainline kernel (2.6.29), and hence has to be still considered as being experimental. The ZFS filesystem performed well in the single disk and RAID setup (compared to ext4). Based on the different kernel architectures, while a ZFS to ext4 (or Btrfs) comparison on the same hardware has lots of merit, the results have to be deciphered by assessing the big picture, incorporating the entire kernel IO subsystem features (or lack thereof) into the analysis.

References

1. Mason, C., "*The Btrfs Filesystem*", Oracle Corporation, 2007
2. Kerner, M., "*A Better File System For Linux*", InternetNews, 2008
3. Rodeh, O., "*B-trees, Shadowing, and Clones*", IBM Haifa Research Lab, 2007
4. Bonwick, J., "*ZFS Block Allocation*", Online, 2007
5. Bonwick, J., "*ZFS: The Last Word in Filesystems*", Sun Microsystems, 2006
6. Sun, "*ZFS On-Disk Specification*", Sun Microsystems, 2006
7. Heger, D., Jacobs, J., McCloskey, B., Stultz, J., "*Evaluating Systems Performance in the Context of Performance Paths*", IBM Technical White Paper, Austin, 2000
8. Heger, D., Pratt, S., "*Workload Dependent Performance Evaluation of the Linux 2.6 IO Schedulers*", OLS 04, Ottawa, 2004
9. Johnson, S., "*Performance Tuning for Linux Servers*", IBM Press, 2005
10. Mauro, J., "*Solaris Internals – Core Kernel Architecture*", Prentice Hall, 2001
11. McDougall, R., "*Solaris Internals - Solaris 10 and OpenSolaris Kernel Architecture*", 2nd Edition, Sun Press, 2007
12. Hennessy, J., Patterson, D., "*Computer Architecture, a Quantitative Approach*", Third Edition, Morgan Kaufmann, 2003
13. McCarthy, S., Leis, M., Byan, S., "*Large Disk Blocks – Or Not*", USENIX 02, Santa Cruz, 2002
14. Sun Microsystems "*ZFS – Best Practices Guide*", Solaris Internals (Wiki), 2009
15. Heger, D., "*Quantifying IT Stability*", iUniverse Press, 2008
16. Bhattacharya, "*The ext4 file system*", FOSS, IBM Lab, 2006

Appendix A: Raw Performance Data

A1: Single Disk Performance

Sequential Writes (4 threads)	*TP MB/sec*	*Efficiency (Ops/%CPU)*
ext4	72.1	**6,100**
btrfs	**72.27**	1,200
zfs	69.9	4,800
Sequential Reads (4 threads)	*TP MB/sec*	*Efficiency (Ops/%CPU)*
ext4	**81.45**	**17,800**
btrfs	54.3	5,600
zfs	72.1	4,900
Random Reads (4 threads)	*TP MB/sec*	*Efficiency (Ops/%CPU)*
ext4	1.45	**2,200**
btrfs	1.58	1,100
zfs	**1.95**	1,200
Random Writes (4 threads)	*TP MB/sec*	*Efficiency (Ops/%CPU)*
ext4	2.1	**2,080**
btrfs	2.5	254
zfs	**3.3**	612
Mixed Workload (4 threads)	*Ops/second*	*Efficiency (Ops/%CPU)*
ext4	**2,430**	**5,700**
btrfs	1,100	550
zfs	2,241	1,955

A2: RAID Performance

Sequential Writes (8 threads)	*TP MB/sec*	*Efficiency (Ops/%CPU)*
ext4	**208.6**	**1,600**
btrfs	109.12	1,000
zfs	193.74	1,300
Sequential Reads (8 threads)	*TP MB/sec*	*Efficiency (Ops/%CPU)*
ext4	**248.1**	**15,500**
btrfs	162.52	5,400
zfs	227.74	8,900
Random Reads (8 threads)	*TP MB/sec*	*Efficiency (Ops/%CPU)*
ext4	6.1	**2,580**
btrfs	7.5	1,250
zfs	**8.1**	1,050
Random Writes (8 threads)	*TP MB/sec*	*Efficiency (Ops/%CPU)*
ext4	**36.1**	**2,100**
btrfs	12.48	450
zfs	32.9	1,400
Mixed Worklaod (8 threads)	*Ops/second*	*Efficiency (Ops/%CPU)*
ext4	**9,660**	**2,250**
btrfs	4,800	350
zfs	7,728	1,040