Benchmarking Linux Filesystems for Database Performance Using the 3n+1 Problem

K.S. Bhaskar
Senior Vice President, FIS
ks.bhaskar@fisglobal.com
+1 (610) 578-4265

Summary

This reports the results of an apples to apples comparison of several Linux file systems using the free / open source FIS GT.M™ transaction processing database engine driven by an update intensive workload and highly random access.

Unsurprisingly, the best performance was observed on ext2 and the worst on btrfs.

Background

As the developers of FIS GT.M (http://fis-gtm.com), a database engine that is used in mission-critical applications at large health care and financial institutions we are extremely interested in file system performance for transaction processing database applications. To a first approximation, the fraction of database accesses that are updates is much higher in a transaction processing application than with other database applications.

We previously developed and released io_thrash (http://sourceforge.net/projects/fis-gtm/files/Benchmarking/io_thrash/io_thrash_20081110.tgz) as a workload simulating a transaction processing database application against a POSIX API to compare disparate systems. io_thrash remains a useful tool, especially on systems without a GT.M port.

More recently, to evaluate the performance of the GT.M engine on a variety of file systems, we conducted a benchmark of several Linux file systems under an identical GT.M workload. This benchmark is also freely available.

FIS GT.M

FIS GT.M is a hierarchical (multidimensional key-value, "NoSQL") database engine. Unlike better-known database engines, GT.M uses a daemonless architecture in which application processes accessing the database cooperate with one another to manage the database. Processes run with normal user and group ids, and are able to access a database file only if access is permitted by file ownership and access permissions. Instead of routing database accesses through a small number of daemon processes, a large number of user processes (potentially thousands at large sites) concurrently access the database.

A database consists of a number of *regions*, each of which contains a current *journal file* and a *database file*. When committing an update to the database, a process writes to the journal file sequentially (each process performs a seek to and write at the end of the journal file), followed by updates to database blocks cached in a shared memory segment. Randomly selected processes flush blocks in the cache to disk to avoid stale blocks or buffer full conditions. Periodically (every 300 seconds by default), some process ensures no dirty blocks remain and fsync's to create a checkpoint called an *epoch*. The first update of a block after an epoch places a *before image* record of that block in its pre-modification state in the journal file. In the event of a system crash, GT.M can recover the database from the journal file using *backward recovery* – using the before image records in the journal file to roll the database back to the last epoch then applying the updates in the journal file to restore the database state to that at the time of the crash.

When an updating process finds a journal file has reached its maximum size (typically 4GB), the process renames the file (with a time-stamp affixed to the file name) and creates a new current journal file with a back-pointer to the predecessor journal file. The bulk of the IO is journal IO – for example, on the jfs test, the database grew to 5.3GB, but there were 15 prior generation journal files of 4GB each, and one current journal file of 1.2GB. Note that slower file systems generate more journal records, since a before image record is generated when a block is first updated in an epoch, and longer running tests have more epochs. When the longest benchmarks were run on the slowest file systems, prior generation journal files had to be deleted with the benchmark underway in order to avoid filling up the available file system with journal files. The impact of this was not material, since the tests in question ran for tens of thousands of seconds, and the file deletion took no more than tens of seconds.

In this benchmark, we configured the database with just one region, with a database block size of 4KB, and a shared memory cache with room for 65,536 blocks. We left all other database parameters at their defaults.

The GT.M version was the 64-bit executable of V5.4-001 for Linux on x86 architectures downloadable from Source Forge (http://sf.net/projects/fis-gtm).

3n+1 sequence

The 3n+1 sequence for a positive integer is the number of steps in the following sequence that it takes to reach 1:

- if the number is even, divide it by 2, and
- if the number is odd, multiply it by 3 and add 1.

The as yet unproven Collatz conjecture (http://en.wikipedia.org/wiki/Collatz_conjecture) holds that for every integer, the 3n+1 sequence reaches 1 after a finite number of steps. In the benchmark, a number of parallel worker processes cooperate using a database to find the longest 3n+1 sequence starting with any integer in a range, as well as the largest integer encountered in the course of calculating those sequences. The benchmark divides the range of starting integers into sub-ranges of numbers and each worker process computes the sequence for each number in one sub-range at a time – of course, this will take it to numbers outside the sub-range. If the database contains an answer to the sequence length for a number the process moves on to the next number. After the last number in its sub-range, it works on the next sub-range of numbers not yet claimed by another worker process. If the database does not contain an answer to the sequence length for a number, the process stacks the number, computes the next number in the sequence and checks the database for that number's sequence, continuing till it finds a number whose sequence length is in the database or till it encounters 1 (whose sequence length is zero). Then it pops its stack of numbers one by one, and enters the sequence length for each number in the database, the sequence length for each number being one more than that of the previous number popped. Each process tracks its counts of logical reads and writes, and contributes to the maximum number value handled in the course of the benchmark. When all worker processes terminate, the parent process prints the results and then reads the next line of input and kicks off the processing it specifies. The journaling configuration ensures that, should the system crashes at any point in the benchmark, GT.M can recover the database, and resume the computation by restarting the program with the same input range it was working on at the time of the crash. The resumed computation would use results computed and stored in the database from before the crash.

A complete functional specification for the program can be found at http://ksbhaskar.blogspot.com/2010/06/3n1-nosqlkey-valueschema-freesche.html and the actual program used can be found at http://sourceforge.net/projects/fis-gtm/files/Benchmarking/threeen1/threeen1f.tgz

Instructions for you to run the program yourself can be found at https://docs.google.com/document/pub?id=1OO2TG4pAuW3MrEPSlzv1zAkIlsADq9PpI46DilM5lQ8

The input file used for the benchmark was:

```
1 100000 8 5000
1 1000000 8 50000
1 10000000 8 50000
1 20000000 8 50000
1 40000000 8 50000
```

Regardless of the number of worker processes, block size, elapsed time, total number of database accesses or accesses per second, the following results demonstrate functional correctness:

Start	Finish	Longest sequence	Largest number encountered
1	1,000,000	524	56,991,483,520
1	10,000,000	685	60,342,610,919,632
1	20,000,000	704	306,296,925,203,752
1	40,000,000	744	474,637,698,851,092

The number of reads and number of updates varied slightly between runs – with multiple parallel worker processes, it is quite possible for more than one process to compute the length of the 3n+1 sequence for an integer. The standard deviation as a percentage of the average in the number of reads ranged from 0.16% to 0.005%, and that for the number of updates ranged from 0.28% to 0.001%, with the largest percentages for the runs with the smallest final integer. So, any variation was not material.

Computer system

- CPU – Quad Core AMD Phenom™ 9850 at 2500MHz
- Cache – L1: 128KBx4; L2: 512KBx4; L3: 2MBx1
- RAM – 4GB DDR2 800
- Disks – Twin Samsung HD103SJ 1TB 3Gbps SATA. Each had four partitions. Root occupied one partition on one drive (/dev/sda1). A volume group was built from one 705GB partition on each drive (/dev/sda3 and /dev/sdb3). Other partitions were mounted but not accessed during the benchmark.
- OS – 64-bit Ubuntu 10.04.1 (desktop) with Linux kernel 2.6.32-25.
- Benchmark file systems – from the volume group, multiple 100GB file systems, one for each type tested, were created, each striped across the two drives. All file systems were created with default settings.

A complete hwinfo report on the system is available on request – please e-mail the author.

Results

When monitoring the runs with atop (http://www.atoptool.nl), the runs were all CPU limited for the first line (through 1,000,000). During the second (10,000,000), the system would start CPU limited, and transition to IO limited (avio times in the small number of milliseconds, disks "red-lined" and usually more than 80% busy and most of the time more than 95% busy). The third and fourth would spend virtually all of their time IO limited.

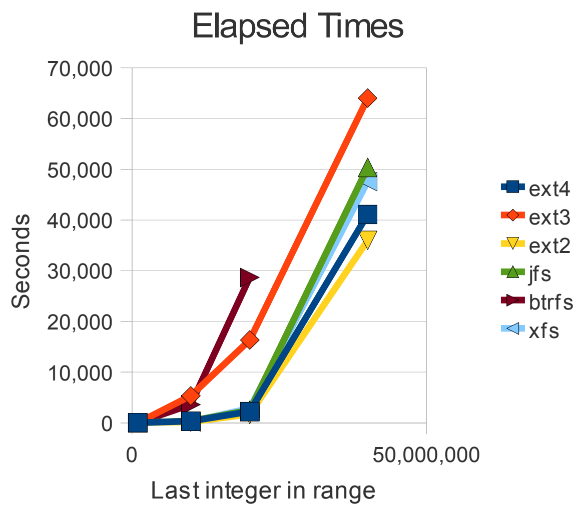
The btrfs run with a final integer of 40,000,000 was terminated after more than a day and a half when it was estimated to be only about 80% through the run.

Elapsed time

The elapsed times in seconds for each file system are:

Finish	ext4	ext3	ext2	jfs	btrfs	xfs
1,000,000	15	14	13	13	15	14
10,000,000	319	5,312	177	374	3,571	220
20,000,000	2,213	16,323	1,680	2,484	28,667	2,834
40,000,000	41,087	64,016	35,989	50,378		47,571

Elapsed times are graphed below:

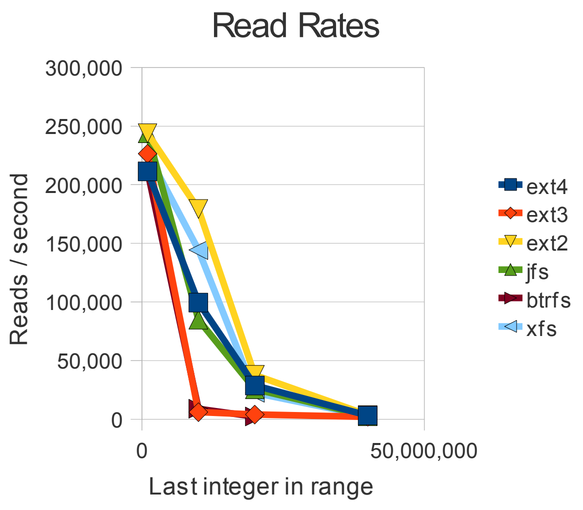


Read Rate

The read rates for each file system were:

Finish	ext4	ext3	ext2	jfs	btrfs	xfs
1,000,000	211,365	226,421	243,805	243,870	211,341	226,581
10,000,000	99,479	5,974	179,277	84,845	8,886	144,271
20,000,000	28,675	3,888	37,773	25,547	2,214	22,392
40,000,000	3,089	1,983	3,526	2,519		2,668

The read rates are graphed below:

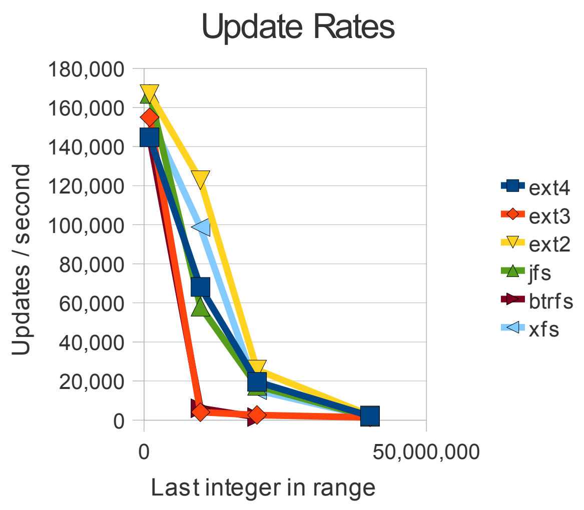


Update Rates

The update rates per file system were:

Finish	ext4	ext3	ext2	jfs	btrfs	xfs
1,000,000	144,698	154,992	166,882	166,947	144,674	155,152
10,000,000	68,131	4,091	122,779	58,107	6,086	98,816
20,000,000	19,638	2,662	25,868	17,495	1,516	15,335
40,000,000	2,115	1,358	2,415	1,725		1,827

The update rates graphed are:



Discussion

It comes as no surprise that ext2, as a non-journaled file system, consistently outperforms all the other file systems. The performance of btrfs, as a copy-on-write file system, is probably also to be expected. That the performance of ext4 beats ext3 is perhaps also to be expected, given the file systems' lineage.

jfs, ext4 and xfs were in the middle, with ext4 leading the other two.

Future Work

In the future, we plan to continue the benchmark to look at other types of IO subsystems. For example, below are elapsed times with the benchmark on a jfs file system on an SSD.

Finish	Magnetic	SSD	Ratio
1,000,000	13	9	1.4
10,000,000	374	193	1.9
20,000,000	2,484	631	3.9
40,000,000	50,378	2,341	21.5

This is not an apples to apples comparison because the CPU, cache and RAM differ between this system and the one reported on earlier. Nevertheless, the fact that the CPU-limited smaller runs have the same elapsed times suggests that they are broadly comparable. But the disk-limited largest run is dramatically faster on the solid state disk.

We also intend to look at the impact of filesystem, database and journal tuning parameters as well. GT.M has a sync_io flag, which when used causes processes to open the journal file with O_DIRECT and O_DSYNC. The environment variable $gtm_fullblockwrites causes a GT.M process to write database blocks that are an integer number of filesystem blocks, at an offset within the file that is also a multiple of that size. The idea is that such a write when a database block is not full, even if the extraneous data written is not meaningful, may permit the underlying IO subsystem to perform just a write operation rather than a read-modify-write operation. Below is a comparison of the effect of these flags on elapsed times for ext4 and jfs on a 1 to 20,000,000 run.

File system	Sync io	Full block writes	Elapsed times	Relative Speed
jfs	No	No	2,484	100%
jfs	Yes	No	1,733	143%
jfs	No	Yes	2,273	109%
jfs	Yes	Yes	1,743	143%
ext4	No	No	2,213	100%
ext4	Yes	No	3,137	71%
ext4	No	Yes	2,122	104%
ext4	Yes	Yes	3,050	73%

It it interesting that both these speed up jfs – almost to ext2 speeds, but not quite – whereas sync_io slows ext4 and $gtm_fullblockwrites has only a slightly beneficial effect. There are a small number of other parameters as well within GT.M, although not too many: part of our philosophy over the years has been that if we create a tuning parameter, we should also endeavor to tune it automatically and dynamically.