CEWG / Linux Foundation Open Projects

eMMC/SSD File System Tuning Methodology

Rev. 1.0

May 24, 2013

Abstract

This document describes methods of file system performance measurement and tuning for eMMC/SSD storage media, including open source tools and file system options. It also covers longevity aspects of eMMC/SSD media with certain file systems and file system power-fail tolerance in the presence of performance-tuning options.

Audience

Audience of this document is people optimizing I/O performance of embedded systems with managed FLASH storage devices.

Document Status

Draft

Contact Information

Send your comments, questions and report any other issues related to this document to: max.filippov@cogentembedded.com and artemi.ivanov@cogentembedded.com.

Cogent Embedded, Inc. contact information is available at http://cogentembedded.com

Revision History

Rev.	Date	Author	Changes
0.1	October 15, 2012	M. Filippov	Initial Revision
0.2	November 1, 2012	M. Filippov	Add preliminary results and benchmark scripts overview
0.3	November 12, 2012	M. Filippov	More results, more scripts details
0.4	January 12, 2013	M. Filippov	Add IO schedulers results
0.5	May 13, 2013	M. Filippov, K. Kozhevnikov	Add Wear Intensity and Power-Failure Tolerance sections. Update results section and clean-up text.
1.0	May 24, 2013	D. Semyonov	Describe power-failure testing results. Clean-up text, styles, and paragraph ordering. Update logo.

Issues

None

Table of Contents

1 Introduction

1.1 Overview

This document provides a methodology for performance tuning of a file system operating on a managed FLASH device, like SSD/SD/eMMC. It focuses on ext3/ext4, BTRFS and f2fs open source file systems (list can be extended). A set of performance characteristics is defined and tools for workload simulation / performance measurement are proposed. Procedures for wear intensity estimation of tuned file systems and power-fail tolerance are described.

The document is accompanied with a set of scripts that allow conducting performance testing and tuning for exact target/storage configuration in accordance with the methodology.

1.2 Acronyms and Abbreviations

Term	Definition
atime	Access time (one of the file time stamps)
eMMC	Embedded MultiMedia Card
FS	File System
IO, I/O	Input/Output
relatime	Relative access time (atime, that is only updated if the previous atime was earlier than the current modify or change time)
SD	Secure Digital (non-volatile memory card format)
SSD	Solid State Drive

2 Setup

2.1 Hardware Setup

No specific requirements for test hardware are imposed, though it is expected to have CPU performance high enough to keep IO benchmarking IO-bound rather than CPU-bound.

Our selection of ARM/Cortex-A9 platform is explained by the fact that it is one of the most popular choices for hardware configuration these days. (If necessary, different platforms – e.g. Intel – can be easily added too.)

The conducted testing was performed using the following evaluation boards:

- TI OMAP 4460-based PandaBoard ES (with external SD cards)
- Exynos 4412-based ODROID-X board (with swappable Toshiba eMMC)

2.2 Software Setup

The following software configuration was used for the testing:

- **kernel:** mainline linux-3.8;
- **userland:** Ubuntu 12.04 for ARM;
- **rootfs:** mounted over NFS, common for both PandaBoard and ODROID-X. NFS allows to minimize interference of system activities with benchmarking and minimize setup time for new media;
- **additional tools:** btrfs-progs (git tip: v0.20-rc1-37-g91d9eec), f2fs-tools-1.1.0, fio (2.0.10);
- **host tools:** gnuplot (graphical result comparison), expect (power switch control), gawk (parsing test logs).

3 Performance Benchmarking and Optimization

A number of factors influence file system performance. Some of them (like file system cache size or file data predictability) do not depend on underlying media, others (like media access pattern) do. In order to tune file system performance for specific media it needs to be put into an environment where media-dependent factors would prevail. The following methods are proposed to achieve that:

- use direct IO. This should be used only in cases when IO pattern is to be controlled by the benchmark;
- pin down memory with mlock. This method is proposed for workloads that would use FS cache in real life. Memory available for FS cache should be limited to 1/2...2 times total file size used by the benchmark; -- unfortunately, this method proved to be very fragile: task is either unable to mlock 1/2 of the free memory, or it is being killed by the oom-killer once it could;
- limit available machine memory at boot time;
- use random data for file IO. This would mitigate data compression and de-duplication at file system and device levels.

3.1 Approach

The following iterative approach for file system tuning is proposed:

1. file system without any tuning, (that is, with default options), is set up and its performance parameters are measured;
2. each of the remaining tuning options is applied to the current file system setup and performance parameters of the new setup are measured;
3. parameter difference is evaluated and the option that gives the biggest performance gain is added to the current file system setup, step 2 is repeated until there are no more options or no option gives performance gain.

Each benchmark provides a method for visual performance parameters comparison. Benchmarking script also provides hints about optimal file system parameters/IO scheduler for each workload and generally for a file system based on either cumulative throughput or worst latency.

3.2 Measured Performance Parameters

The following parameters are chosen for FS performance evaluation:

1. ***aggregated throughput.*** This parameter is important for all tasks as it directly determines available IO bandwidth and thus time needed to complete the task;
2. ***access latency.*** This parameter is important for interactive tasks.

The following benchmarks are chosen for FS performance evaluation:

1. linear read/write single-threaded throughput with a range of fixed IO block sizes;
2. random read/write single-threaded throughput and access latency with a range of fixed IO block sizes;

3. aggregated throughput of multithreaded workloads. This parameter shows FS performance that may be expected by real tasks. The following workload types are chosen:
   3.1. linear read from multiple files (grep, archiving);
   3.2. linear read + linear write to multiple files (unarchiving);
   3.3. multithreaded linear read from multiple files + linear write (compilation);
   3.4. multithreaded random read from multiple files + random write to multiple files (file server);
   3.5. multithreaded random read/write to memory-mapped file (simple database).

3.3 Results Comparison

Benchmarking tools produce result vectors that may be plotted together or compared numerically.

Numerical comparison is performed by calculation of merit value for each combination of benchmark + file system options + IO scheduler, and ordering them accordingly. Merit value may be calculated either with respect to aggregated throughput or to worst latency:

$m_{throughput}=(\Sigma\log(throughput_i))/n$

$m_{latency}=1/\max(latency_i)$

Each benchmarking run yields several data points (for different IO block size or IO size to cache size ratio), summation and maximum determination are performed for all these points. Winning file system options + IO scheduler combination for each benchmark is the one with the highest merit. Winning file system options + IO scheduler combination for the medium is the one with the highest sum of merit for each benchmark.

3.4 Benchmarking Tools and Workload Generators

The following tools are chosen for the benchmarking:

- flashbench;
- FIO.

The following candidates were also evaluated for the benchmarking:

1. IOZone: not flexible enough, functionality is superseded by FIO;
2. bonnie++: predictable test data, functionality is superseded by FIO;
3. filebench: not very stable, functionality is superseded by FIO.

3.4.1 Flashbench

Although not file system benchmarking tool, flashbench allows to measure FLASH media characteristics such as erase block/segment size which in turn allow to tune file system geometry accordingly. It is only used during initial evaluation of media characteristics.

The following approach for FLASH media geometry determination is proposed in the flashbench README:

> “*This is a simple read-only test doing small reads across boundaries of various sizes. Example:*

```
$ sudo ./flashbench -a /dev/mmcblk0  --blocksize=1024
align 134217728 pre 735µs       on 1.08ms       post 780µs      diff 324µs
align 67108864  pre 736µs       on 1.05ms       post 763µs      diff 300µs
align 33554432  pre 722µs       on 1.04ms       post 763µs      diff 294µs
align 16777216  pre 727µs       on 1.05ms       post 772µs      diff 302µs
align 8388608   pre 724µs       on 1.04ms       post 768µs      diff 299µs
align 4194304   pre 741µs       on 1.08ms       post 788µs      diff 317µs
align 2097152   pre 745µs       on 950µs        post 811µs      diff 171µs
align 1048576   pre 745µs       on 945µs        post 807µs      diff 169µs
align 524288    pre 743µs       on 936µs        post 799µs      diff 165µs
align 262144    pre 746µs       on 948µs        post 809µs      diff 171µs
align 131072    pre 737µs       on 935µs        post 804µs      diff 165µs
align 65536     pre 735µs       on 925µs        post 796µs      diff 159µs
align 32768     pre 735µs       on 925µs        post 800µs      diff 157µs
align 16384     pre 745µs       on 911µs        post 781µs      diff 148µs
align 8192      pre 785µs       on 808µs        post 725µs      diff 53.3µs
align 4096      pre 784µs       on 788µs        post 779µs      diff 5.85µs
align 2048      pre 787µs       on 793µs        post 789µs      diff 4.65µs
```

> *This shows the access times to do two 1024 byte reads around the boundaries of power-of-two aligned blocks. Reading at the end of a 128 MB unit takes around 735 microseconds, reading the last block of this unit together with the first block of the next one takes about 1080 microseconds and reading the first two blocks in a 128 MB unit takes around 780 microseconds.*
>
> *The most interesting number here is the last one, the difference between the second number and the average of the first and the third is 324 microseconds. These numbers all stay roughly the same for all units between 4 MB and 128 MB.*
>
> *However, from 2 MB down to 16 KB, the last column has a much lower value. This indicates that whatever the memory card does on a 4 MB boundary does not happen at other boundaries. The educated guess here is that 4 MB is the erase block size, also called the segment or allocation unit size. This erase blocksize will need to be used in other tests following this one.*
>
> *Similarly, both 16 KB and 8 KB boundaries are special. The logical explanation for this is that the card has 8 KB pages, but can use multi-plane accesses to read two 8 KB pages simultaneously.*”

Benchmarking scripts automate FLASH geometry guessing with the described method by finding two rows in the 'flashbench -a' output with the biggest $(D_{row+1}-D_{row})/D_{row}$, where D_i is the value of the last output column, and printing corresponding block sizes.

3.4.2 FIO

FIO is described as a tool that “would be able to simulate a given io workload without resorting to writing a tailored test case”. It is highly configurable: there can be any number of processes or threads involved, and they can each be using their own way of generating IO.

FIO is driven by job files. A job file may contain any number of threads and/or files - the typical contents of the job file is a global section defining shared parameters, and one or more job sections describing the jobs involved. When run, fio parses this file and sets everything up as described. If we break down a job from top to bottom, it contains the following basic parameters:

1. **IO type:** defines the io pattern issued to the file(s). We may only be reading sequentially from this file(s), or we may be writing randomly. Or even mixing reads and writes, sequentially or randomly;
2. **block size:** in how large chunks we are issuing IO. This may be a single value, or it may describe a range of block sizes;
3. **IO size:** how much data we are going to be reading/writing;
4. **IO engine:** how we issue IO. We could be memory mapping the file, we could be using regular read/write, we could be using splice, async io, syslet, or even SG (SCSI generic sg);
5. **IO depth:** if the IO engine is async, how deep queue we maintain;
6. **IO type:** should we be doing buffered IO, or direct/raw IO;
7. **num files:** how many files we are spreading the workload over;
8. **num threads:** how many threads or processes we should spread this workload over.

FIO measures a lot of job parameters, most interesting for us are:

1. the distribution of IO completion latencies;
2. average bandwidth.

FIO has the following shortcomings that limit the range of useful workloads:

1. it cannot measure read and write throughput of readwrite job separately: for jobs with equal read and write data amount reported throughput numbers are also equal;
2. it cannot generate metadata intensive activity.

3.5 File Systems and Their Performance Tuning Options

The following common file system tuning options are chosen:

- **noatime** (mount -o noatime)
  - historically, when a file is read, the access time (*atime*) for that file must be updated in the inode metadata, which involves additional write I/O. If accurate *atime* metadata is not required, file system may be mounted with the *noatime* option to eliminate these metadata updates. In most cases, however, atime is not a large overhead due to the default relative *atime* (or relatime) behavior. The relatime behavior only updates *atime* if the previous *atime* is older than the modification time (*mtime*) or status change time (*ctime*);
- **discard** (ext4, BTRFS, F2FS; mount -o discard)
  - this option controls whether file system should issue discard/TRIM commands to the underlying block device when blocks are freed. Time spent issuing TRIM commands may be compensated by lower write operation delays if media is able to take advantage of bigger free blocks pool.

File system specific options are listed for each evaluated file system.

3.5.1 Ext3/Ext4

- **no journal** (tune2fs -O ^has_journal)
    - this option turns off file system journal. It should be safe to do so on battery-backed device with a stable kernel;
- **data=writeback** (mount -o data=writeback)
    - this option specifies the following journaling mode for file data: data may be written into the main file system after its metadata has been committed to the journal. This is rumored to be the highest-throughput option. It guarantees internal file system integrity, however it can allow old data to appear in files after a crash and journal recovery;
- **nobarrier** (mount -o nobarrier)
    - next to “no journal” option, should be tried in case “no journal” is not acceptable; may be skipped if “no journal” is selected;
    - this option disables the use of write barriers in the jbd code. Write barriers enforce proper on-disk ordering of journal commits, making volatile disk write caches safe to use, at some performance penalty. In case of battery-backed media disabling barriers may safely improve performance;
- **fs geometry** (mount -o stripe=, mkfs -E stripe-width=)
    - this option controls the number of file system blocks that mballoc will try to use for allocation size and alignment. As in RAID5/6 case proper IO block size and alignment may speed up FLASH write operations. Hint for the stripe width may be deduced from the flashbench tool output.

3.5.2 BTRFS

- **fs geometry** (mkfs –leafsize=)
    - this option controls the leaf size, the least data item in which btrfs stores data. As in RAID5/6 case proper IO block size and alignment may speed up FLASH write operations. Hint for the leaf size may be deduced from the flashbench tool output;
- **SSD mode** (mount -o ssd, mount -o ssd_spread)
    - these options enable SSD-optimized allocation schemes.

3.5.3 F2FS

- **fs geometry** (mkfs -s -z):
    - these options control number of segments per section and number of sections per zone. Sizes that match chip parameters may speed up FLASH write operations. Hint for the segment and zone sizes may be deduced from the flashbench tool output;

3.6 I/O Schedulers

The following schedulers are proposed to be tested:

- *noop*, as basic configuration;
- *deadline*, as it provides I/O request merging opportunity with guaranteed latency;
- *cfq*, as it is probably best suited for multithreaded workloads;
- *row*, as it prioritizes read requests (usually synchronous and latency-critical) over write requests.

3.7 Expected Results

The following performance characteristics are generally expected:

- direct IO workloads throughput has little dependency on underlying file system and is close to that of raw medium;
- IO workloads that go through file system cache have the following characteristics:

	big cache	**small cache**
read	• aggregate throughput: nearly constant, as with raw medium • interference with other reads: device bandwidth is divided between all readers • interference with writes: low • IO scheduler effect: low	• aggregate throughput: depends on file system, IO scheduler and concurrent write activity • interference with other reads: device bandwidth is divided between all readers • interference with writes: high • IO scheduler effect: high
write	• aggregate throughput: high, independent of medium • interference with reads: low • interference with writes: low • IO scheduler effect: low	• aggregate throughput: depends on file system, IO scheduler and concurrent read and write activity • interference with reads: high • interference with writes: high • IO scheduler effect: high

3.8 Observed Results

The following devices have been benchmarked:

- 16 GB class 10 SD card (Kingston);
- 16 GB class 10 SD card (Transcend);
- 16 GB class 10 SD card (SanDisk);
- 16 GB eMMC chip (Toshiba);

with the following results of automatic throughput-oriented analysis:

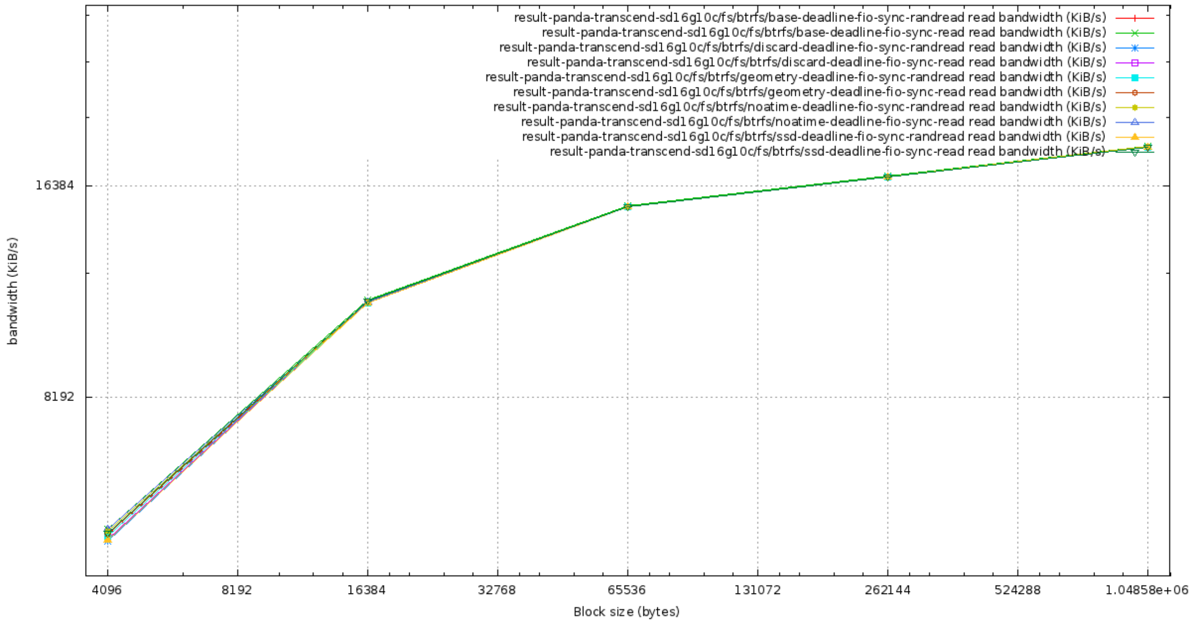
Medium	**BTRFS**	**ext4**	**f2fs**
16 GByte class 10 SD card (Kingston)	ssd deadline 102979 noatime cfq 102788 discard row 102727 base row 102174 ssd cfq 102128	noatime cfq 99513 geometry cfq 98519 writeback cfq 98376 nojournal row 98314 nobarrier cfq 97969	base row 107223 discard cfq 106212 base cfq 106108 noatime row 105754 noatime deadline 105643
16Gbyte class 10 SD card (Transcend)	ssd cfq 107477 ssd row 107142	nojournal cfq 103284 noatime cfq 103246	base row 111643 discard row 111342

	geometry cfq 106774 discard cfq 106711 ssd deadline 106474	noatime row 102373 nojournal row 102325 writeback cfq 102193	discard cfq 111272 noatime noop 111221 noatime cfq 111168
16 Gbyte class 10 SD card (SanDisk)	ssd cfq 106681 base cfq 106528 geometry cfq 106253 noatime cfq 106192 ssd row 106190	nojournal cfq 102254 noatime cfq 101709 writeback cfq 101313 nobarrier cfq 101182 discard row 101026	noatime row 108446 base cfq 108419 discard row 107453 noatime cfq 107254 discard cfq 106571
16 GByte eMMC chip (Toshiba)	base cfq 117721 noatime cfq 117324 geometry cfq 117060 discard cfq 116861 ssd row 116843	discard cfq 117597 noatime cfq 117584 writeback cfq 117520 base row 117515 nojournal cfq 117411	noatime cfq 117043 base cfq 116992 discard cfq 116644 discard row 116136 base row 116131

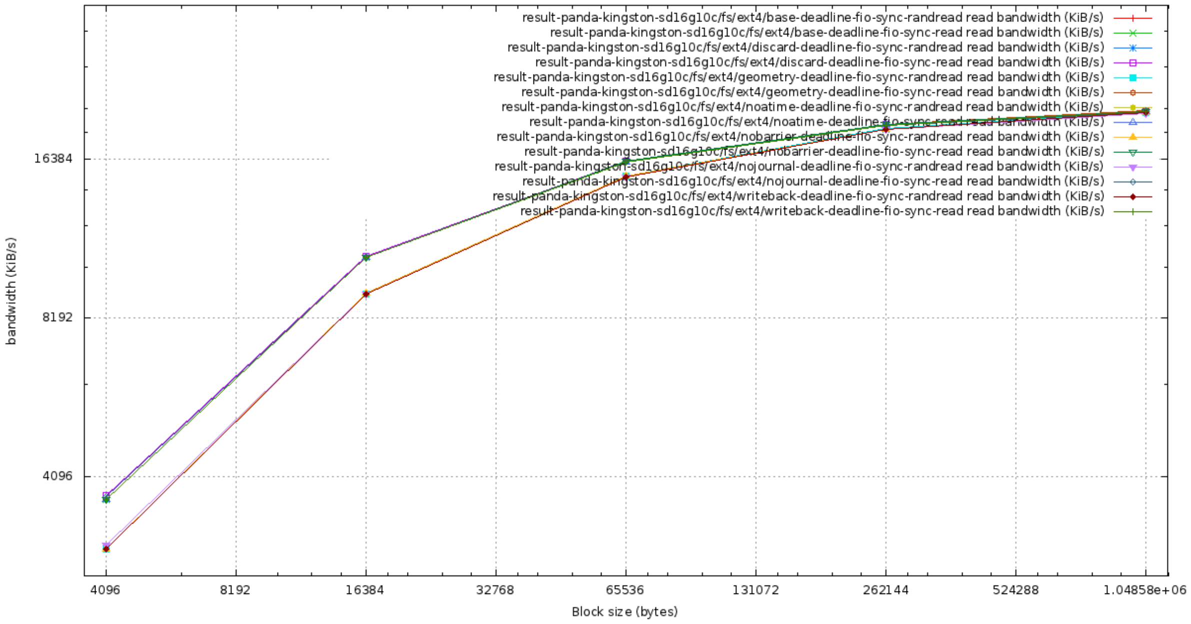
- BTRFS is consistently faster with SSD mode enabled on devices that support TRIM;
- ext4 prefers noatime/nojournal and CFQ scheduler. Strangely it works (slightly) faster with discard enabled on eMMC without TRIM support;
- f2fs runs best with its basic settings, or with noatime enabled;
- CFQ scheduler dominates the table followed by the ROW.

3.8.1 Linear and Random Direct Reading

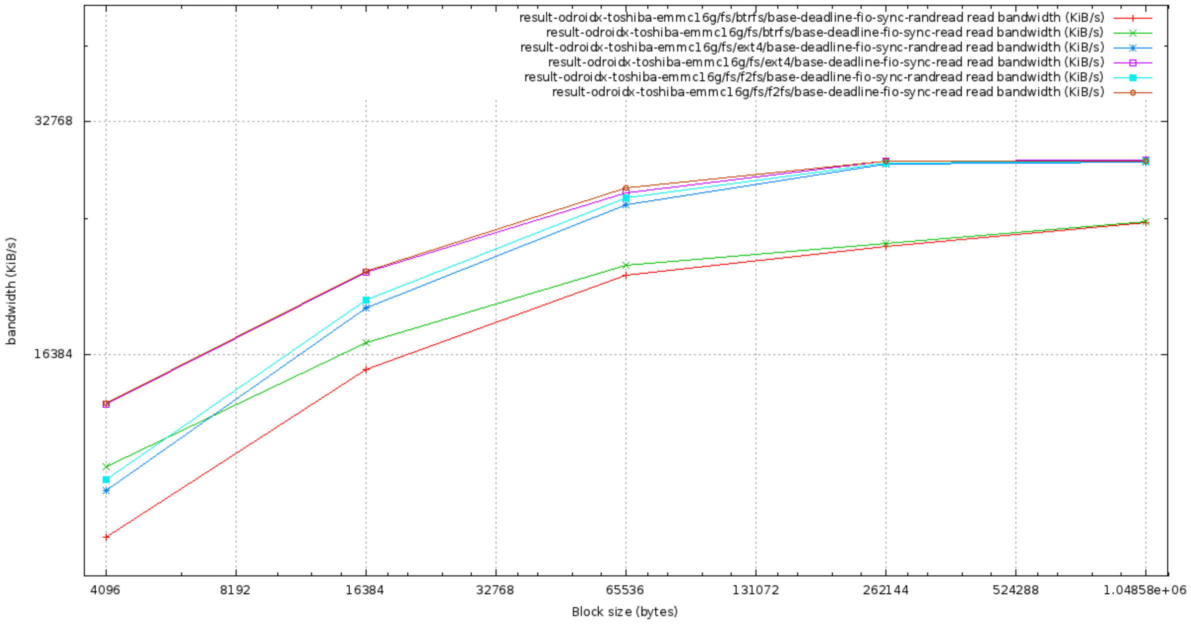
The following two types of read performance are observed on tested media:



first type, where random reads are as fast as sequential reads, on all file systems,



and the second type, where random reads are slower than sequential reads for small block sizes, but the difference gets lower as block size grows. In both cases performance is very little affected by file system options or IO scheduler selection, but is different for different file systems:



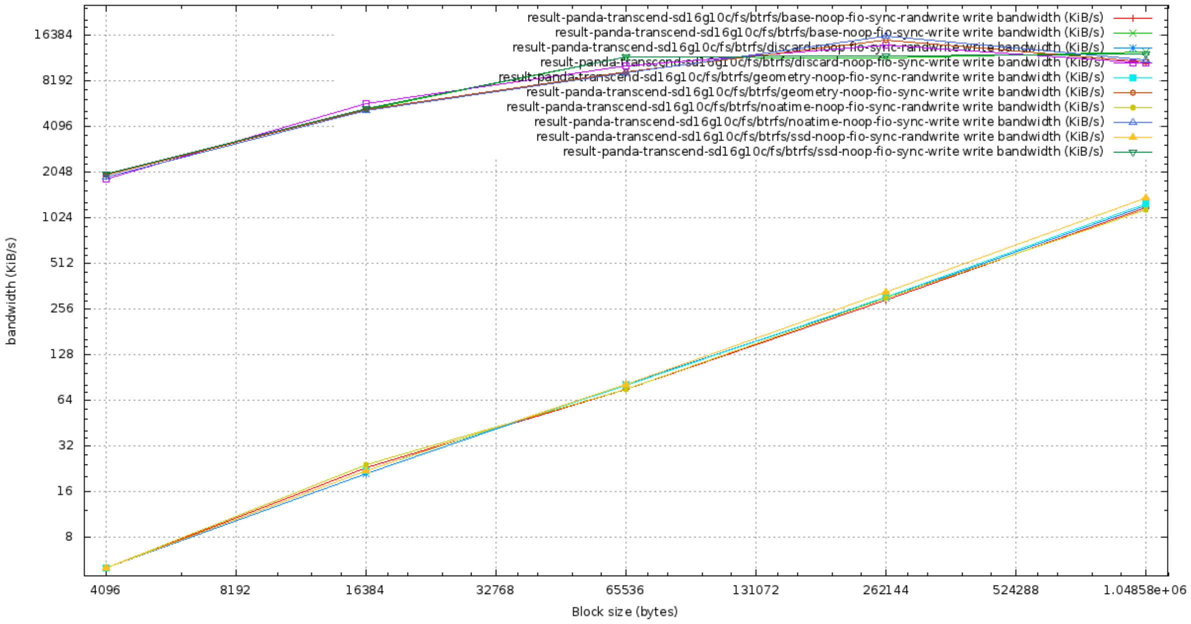
Optimal settings for linear direct reading:

Medium	BTRFS	ext4	f2fs
16 GByte class 10 SD card (Kingston)	noatime row 9313 noatime noop 9311 noatime deadline 9311 noatime cfq 9310 ssd deadline 9306	discard noop 9397 discard deadline 9397 noatime noop 9396 geometry noop 9395 geometry row 9394	noatime noop 9395 noatime deadline 9395 base noop 9395 base deadline 9395 discard noop 9394
16Gbyte class 10 SD card (Transcend)	noatime noop 9427 geometry noop 9424 geometry cfq 9423 base noop 9423	noatime noop 9531 base noop 9531 writeback noop 9530 writeback deadline 9530	base noop 9529 base deadline 9529 discard noop 9528 discard deadline 9528

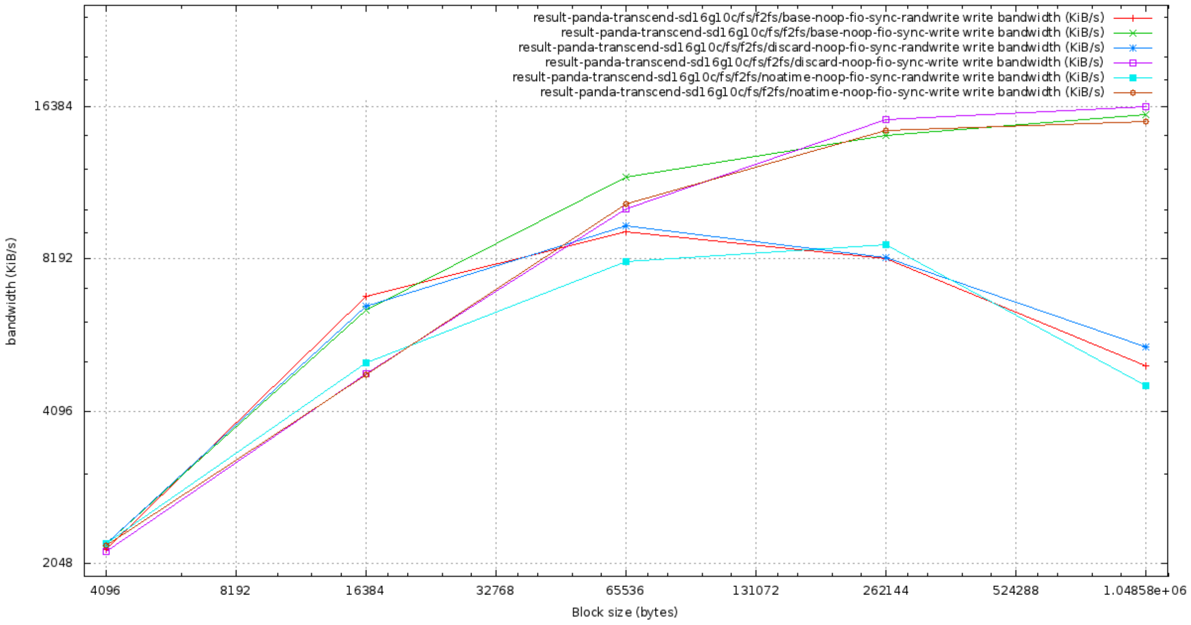
	base deadline 9423	nojournal noop 9530	discard cfq 9527
16 Gbyte class 10 SD card (SanDisk)	noatime noop 9513 ssd noop 9509 ssd deadline 9507 ssd cfq 9507 noatime deadline 9507	nobarrier noop 9625 noatime noop 9625 noatime deadline 9625 writeback noop 9624 nojournal noop 9624	noatime noop 9625 base noop 9625 base deadline 9625 discard deadline 9624 noatime row 9623
16 GByte eMMC chip (Toshiba)	ssd noop 9849 ssd deadline 9846 geometry noop 9846 discard noop 9846 ssd cfq 9845	geometry deadline 10050 base deadline 10050 writeback deadline 10049 nojournal deadline 10049 noatime deadline 10047	noatime deadline 10056 discard noop 10056 noatime row 10055 noatime cfq 10055 base noop 10055

3.8.2 Linear and Random Direct Writing

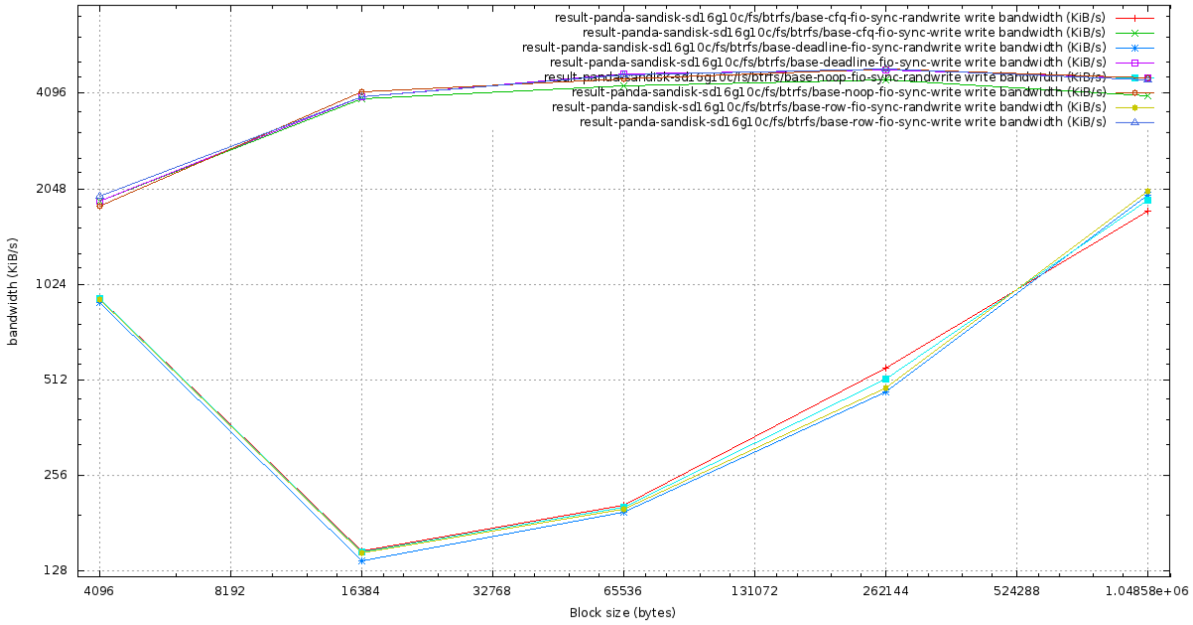
The following two types of write performance are observed on tested media:



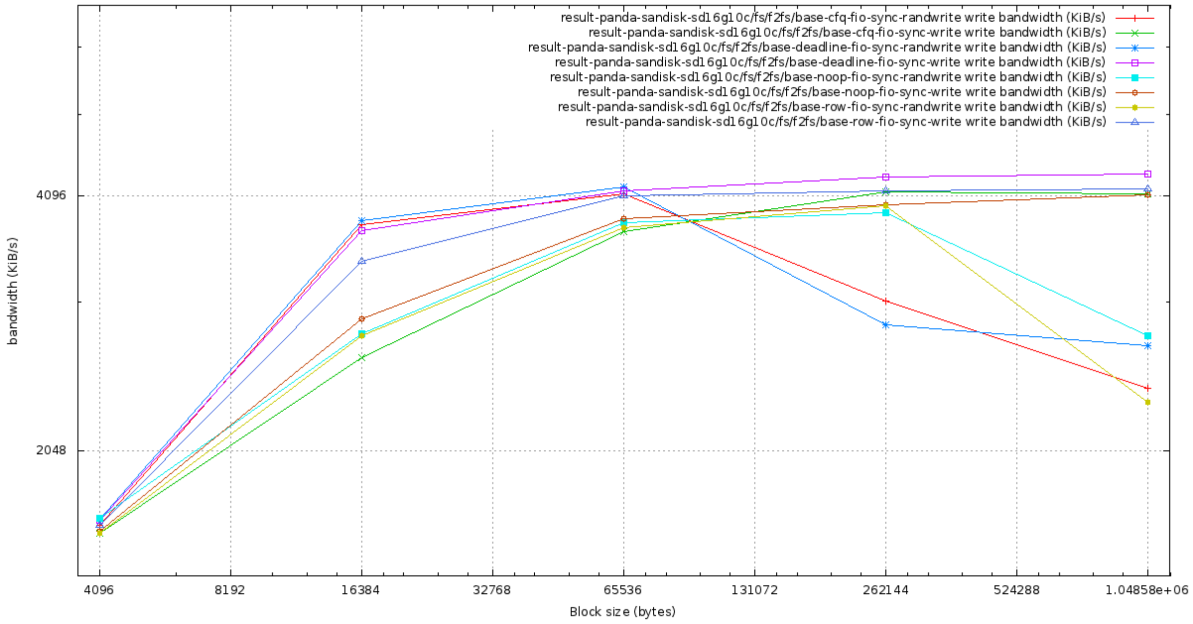
first type, where random write is significantly slower than sequential write and grows linearly with IO block size. However it depends on the file system type, and even on such medium f2fs performs significantly better than btrfs or ext4, keeping random writes on par with sequential writes for IO block sizes up to 64KiB:



and the second type, where random writes with small and large IO block size are fast, but with medium IO block size are slow:



On such media f2fs also performs well with medium IO block size:



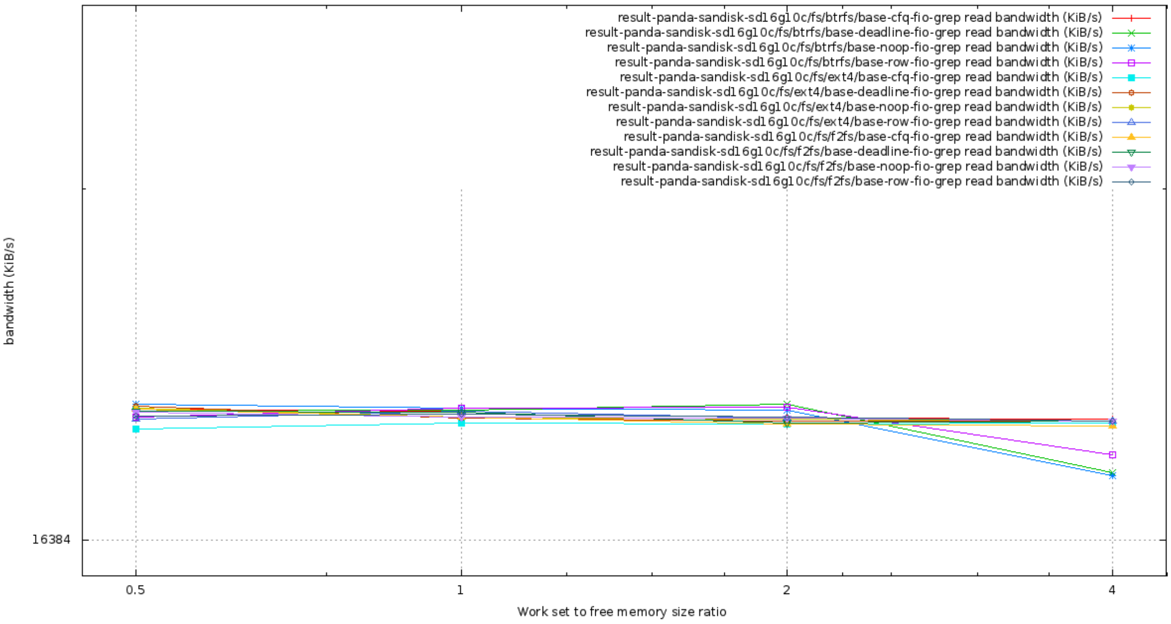
Optimal settings for linear direct writing:

Medium	**BTRFS**	**ext4**	**f2fs**
16 GByte class 10 SD card (Kingston)	ssd noop 8768 ssd row 8766 ssd deadline 8766 geometry row 8755 noatime row 8751	writeback row 8967 writeback noop 8966 writeback deadline 8966 writeback cfq 8961 noatime row 8957	noatime noop 8864 discard noop 8809 base row 8799 base noop 8589 noatime deadline 8588
16Gbyte class 10 SD card (Transcend)	discard deadline 8906 ssd cfq 8877 discard cfq 8873 ssd noop 8870 ssd deadline 8868	geometry deadline 9139 nobarrier row 9126 noatime noop 9122 geometry cfq 9118 geometry noop 9116	discard cfq 9035 base cfq 9032 base noop 9023 discard noop 8951 discard row 8931

16 Gbyte class 10 SD card (SanDisk)	discard cfq 8250 ssd cfq 8246 discard deadline 8244 base row 8244 discard noop 8242	base noop 8315 base deadline 8314 discard noop 8303 base row 8302 geometry deadline 8301	noatime noop 8263 noatime cfq 8201 noatime deadline 8199 noatime row 8193 base deadline 8148
16 GByte eMMC chip (Toshiba)	geometry row 9188 geometry deadline 9186 ssd row 9184 noatime row 9184 noatime deadline 9183	writeback cfq 9160 nojournal cfq 9155 base cfq 9152 discard cfq 9147 geometry cfq 9144	discard noop 9226 noatime noop 9225 base deadline 9225 discard cfq 9224 noatime row 9222

3.8.3 Linear Reading from Multiple Files (grep)

All tested media show the following type of performance for this workload:



File system options and IO scheduler selection effect on performance reach up to 20% when work set to free memory ratio is big.

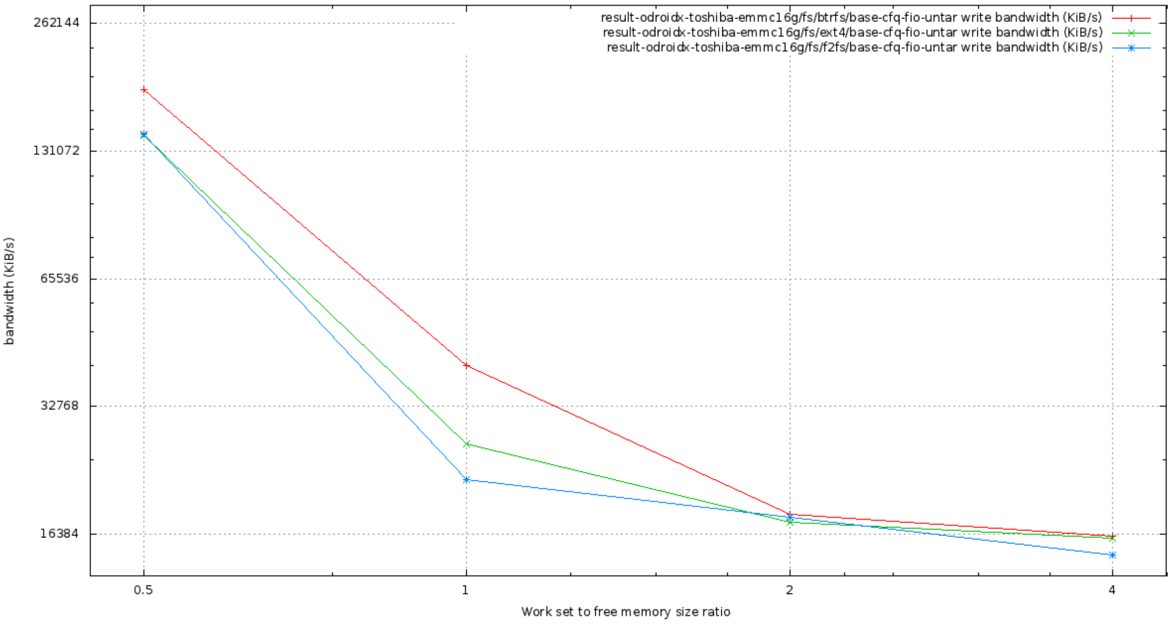
Optimal settings for this workload:

Medium	**BTRFS**	**ext4**	**f2fs**
16 GByte class 10 SD card (Kingston)	ssd deadline 9782 geometry cfq 9767 discard deadline 9766 noatime deadline 9764 base noop 9763	nojournal cfq 9712 nojournal deadline 9711 nojournal noop 9698 nojournal row 9693 base row 9674	discard noop 9708 noatime noop 9707 base deadline 9707 noatime deadline 9701 discard row 9701
16Gbyte class 10 SD card (Transcend)	base row 9797 noatime row 9793 geometry noop 9793 geometry row 9789 discard deadline 9786	nojournal deadline 9741 nojournal row 9727 base deadline 9727 nobarrier cfq 9722 noatime noop 9721	discard noop 9747 discard deadline 9745 base noop 9743 base row 9739 discard row 9736
16 Gbyte class 10 SD card (SanDisk)	geometry noop 9860 discard row 9858 geometry row 9854 geometry deadline 9854	noatime noop 9855 nobarrier deadline 9850 nojournal row 9849 nobarrier noop 9848	noatime deadline 9851 noatime noop 9847 discard noop 9847 discard deadline 9847

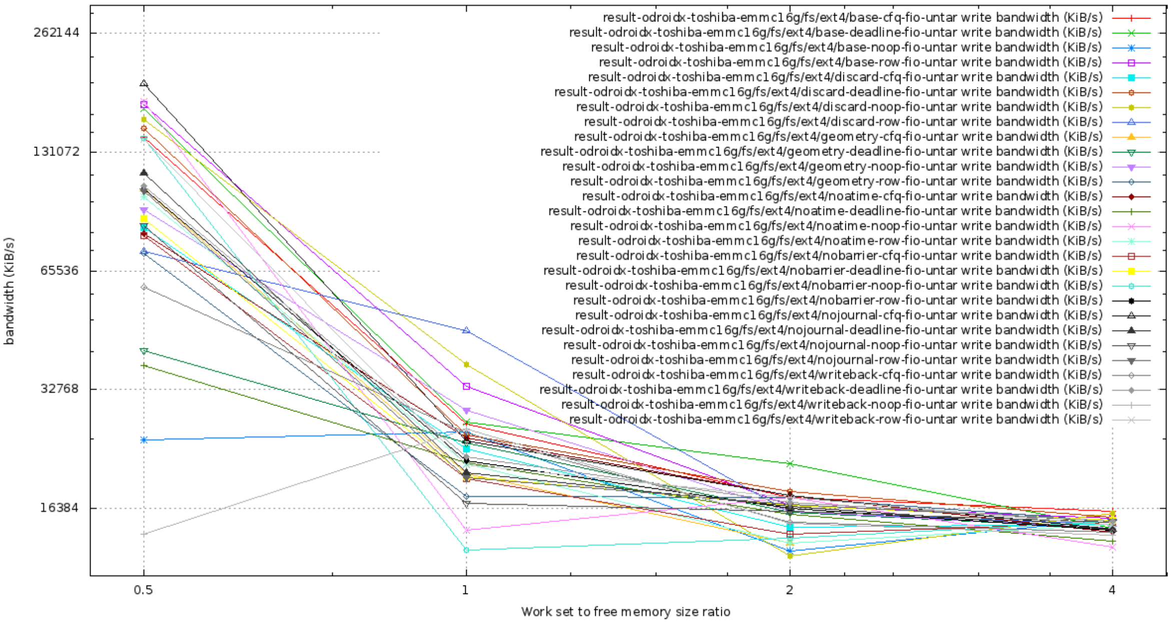
	geometry cfq 9853	discard row 9848	base row 9846
16 GByte eMMC chip (Toshiba)	ssd cfq 10407 base cfq 10407 discard row 10401 discard cfq 10401 geometry row 10398	nojournal cfq 10397 nobarrier noop 10394 base deadline 10394 geometry cfq 10389 discard row 10384	discard noop 10422 base noop 10417 discard cfq 10416 noatime noop 10415 discard deadline 10414

3.8.4 Linear Writing to Multiple Files (untar)

All tested media show the following type of performance for this workload:



File system options and IO scheduler selection have significant impact on performance, especially when work set is less than free memory:



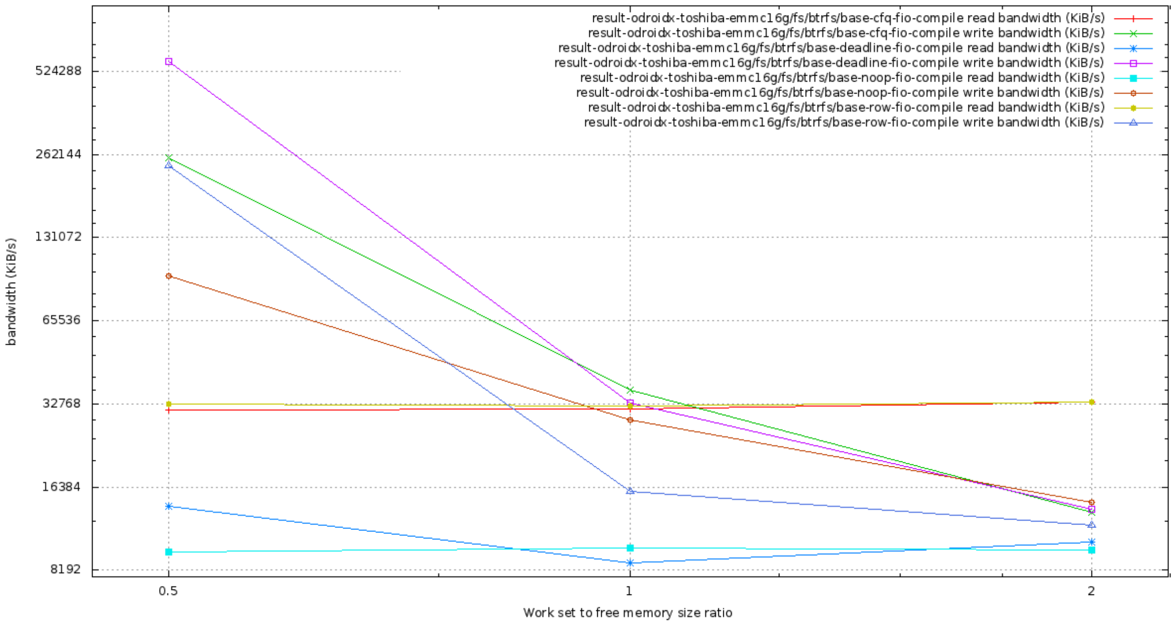
Optimal settings for this workload:

Medium	**BTRFS**	**ext4**	**f2fs**

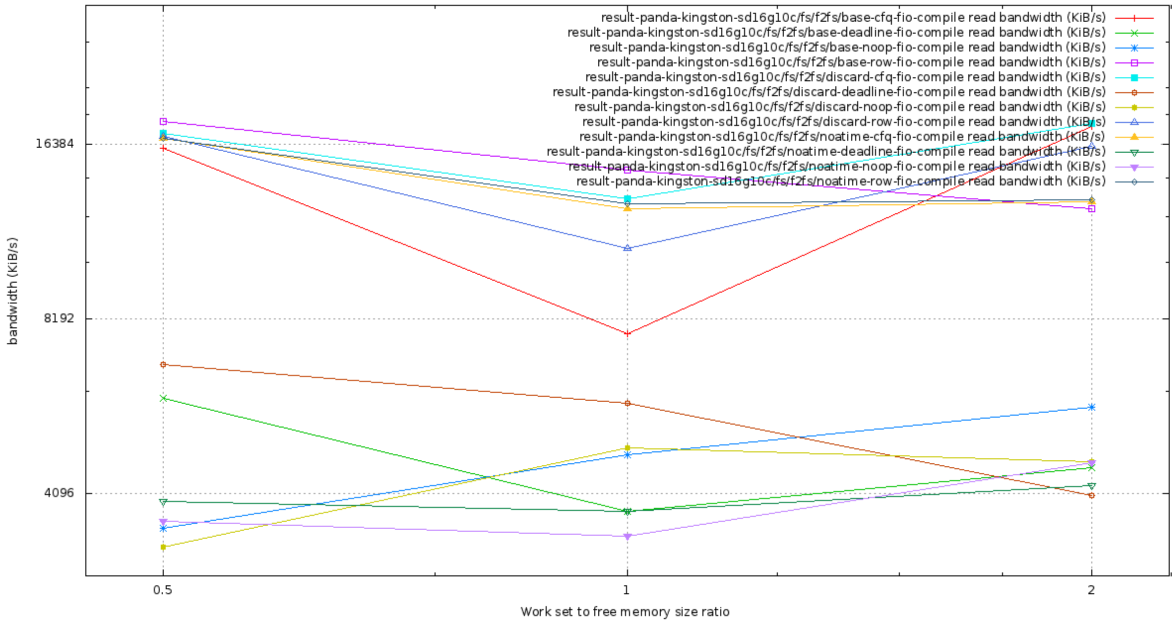
16 GByte class 10 SD card (Kingston)	base row 10203 discard cfq 10088 base cfq 10075 base noop 9793 ssd deadline 9752	discard deadline 9592 writeback cfq 9406 nobarrier deadline 9381 noatime deadline 9335 nojournal noop 9308	base noop 9867 discard noop 9743 discard deadline 9735 base deadline 9627 noatime deadline 9614
16Gbyte class 10 SD card (Transcend)	ssd cfq 10148 noatime cfq 10143 base cfq 10130 geometry cfq 10127 ssd row 10091	nobarrier row 9865 discard row 9848 base noop 9848 writeback cfq 9800 noatime noop 9790	base noop 10200 discard noop 10154 base deadline 10044 discard deadline 9919 base row 9863
16 Gbyte class 10 SD card (SanDisk)	geometry noop 9375 geometry deadline 9375 discard deadline 9359 ssd row 9341 noatime deadline 9340	base row 9435 noatime row 9297 discard cfq 9281 base noop 9274 nojournal noop 9265	noatime deadline 9261 discard noop 9250 noatime row 9071 discard deadline 9002 base cfq 8998
16 GByte eMMC chip (Toshiba)	base cfq 10561 ssd noop 10440 noatime cfq 10440 discard noop 10439 noatime deadline 10436	base row 10461 base deadline 10451 nojournal cfq 10412 discard noop 10406 base cfq 10380	base cfq 10318 noatime cfq 10207 discard deadline 10205 noatime deadline 10194 base deadline 10153

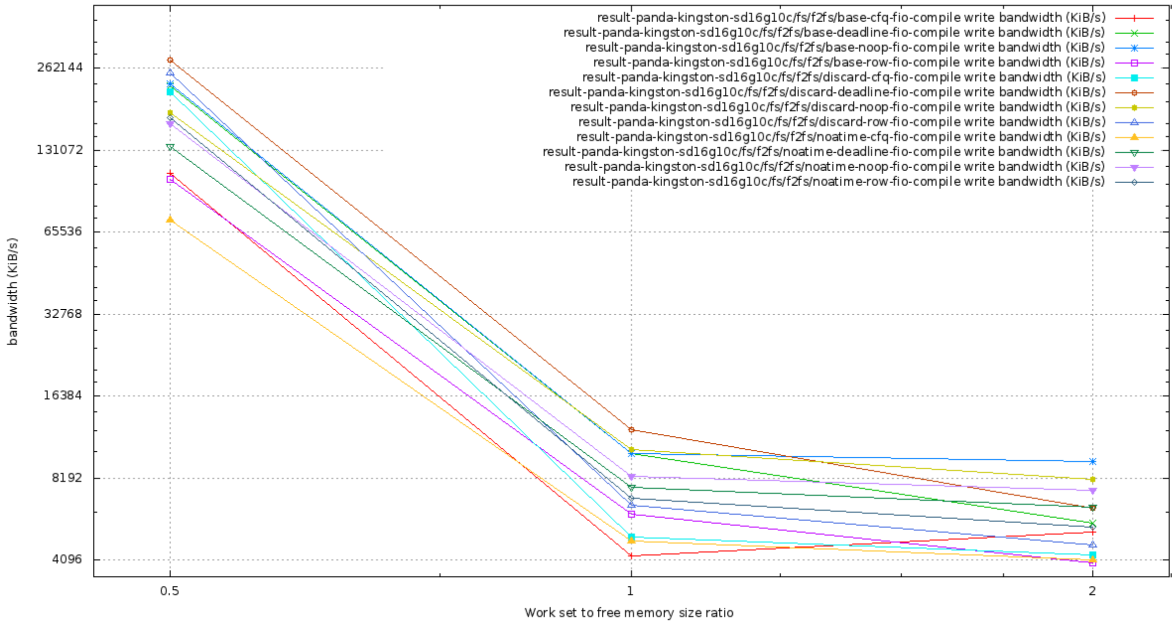
3.8.5 Linear Reading and Writing Multiple Files (compile)

All tested media and all file systems have the following type of performance for this workload:



Write performance starts with very high numbers when work set is small and drops to nearly direct linear medium write speed at big work set size. Read performance varies insignificantly with work set size change, but highly depends on selected IO scheduler and a little less on file system options. Noop scheduler shows the worst aggregated throughput, CFQ and ROW show the best. Both read and write throughput variation with file system options and IO scheduler selection reaches 400%:





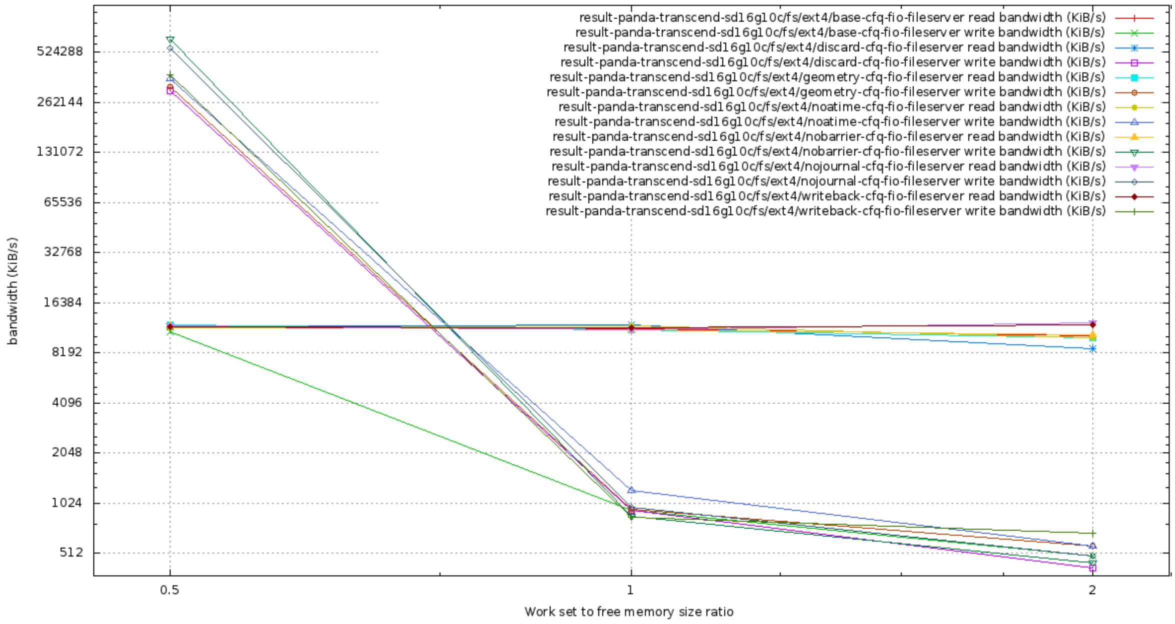
Optimal settings for this workload:

Medium	**BTRFS**	**ext4**	**f2fs**
16 GByte class 10 SD card (Kingston)	ssd row 19869 discard row 19654 base row 19537 geometry cfq 19504 ssd cfq 19501	nojournal cfq 18935 noatime cfq 18660 noatime row 18582 nojournal row 18494 discard row 18016	discard row 19461 noatime row 19393 discard cfq 19391 base row 19130 base cfq 18973
16Gbyte class 10 SD card (Transcend)	geometry row 20321 ssd row 20230 base cfq 20164 geometry cfq 20162 noatime row 20123	noatime cfq 19425 nojournal cfq 19255 noatime row 19109 base cfq 18833 base deadline 18571	discard row 20119 base row 19944 discard cfq 19419 discard noop 19224 base noop 19207
16 Gbyte class 10 SD	base cfq 20033	nojournal cfq 19410	noatime row 20069

card (SanDisk)	ssd row 19977 geometry row 19951 geometry cfq 19948 discard row 19884	noatime cfq 19161 noatime row 19099 nojournal row 18957 writeback row 18780	base row 19995 base cfq 19983 discard cfq 19964 noatime cfq 19797
16 GByte eMMC chip (Toshiba)	base cfq 21189 ssd cfq 21133 noatime cfq 21008 ssd row 20972 discard row 20895	writeback cfq 21492 base noop 21447 noatime cfq 21330 base deadline 21296 discard cfq 21246	noatime cfq 20891 base cfq 20885 discard row 20879 base row 20801 discard cfq 20781

3.8.6 Random Reading and Writing Multiple Files (fileserver)

All tested media and all file systems have the following type of performance for this workload (similar to one for linear reading and writing, but with bigger write throughput degradation on big work set size):



Throughput varies greatly with file system options and IO scheduler selection.

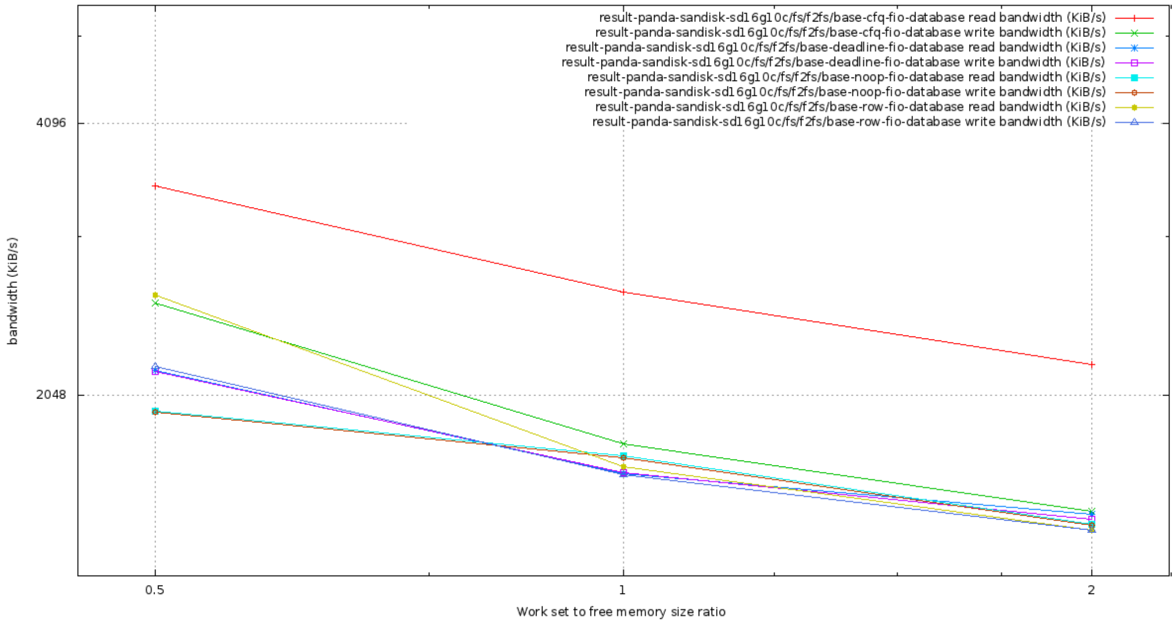
Optimal settings for this workload:

Medium	BTRFS	ext4	f2fs
16 GByte class 10 SD card (Kingston)	noatime cfq 18391 ssd deadline 17845 discard row 17843 base row 17462 ssd cfq 16668	noatime cfq 17275 geometry cfq 17024 nojournal row 16871 writeback cfq 16732 nobarrier cfq 16652	base row 18226 noatime deadline 17679 base cfq 17226 discard cfq 17087 noatime cfq 17074
16Gbyte class 10 SD card (Transcend)	geometry cfq 19490 ssd cfq 19484 discard cfq 19473 ssd row 19083 base cfq 18858	nojournal cfq 18149 writeback cfq 18077 noatime cfq 18072 nobarrier cfq 18064 geometry cfq 17939	base row 19457 discard cfq 19427 discard row 19099 base cfq 18734 discard deadline 18674
16 Gbyte class 10 SD card (SanDisk)	ssd cfq 19021 base cfq 18923 discard cfq 18858 noatime cfq 18761	nojournal cfq 18098 discard row 17915 writeback cfq 17846 nobarrier cfq 17769	base cfq 18975 noatime row 18969 discard row 18729 base deadline 17862

	geometry cfq 18640	noatime cfq 17764	noatime noop 17859
16 GByte eMMC chip (Toshiba)	base cfq 20468 noatime cfq 20417 geometry cfq 20336 discard cfq 20271 ssd row 20240	base cfq 20392 discard cfq 20383 geometry row 20284 noatime cfq 20265 writeback row 20116	noatime cfq 19232 base cfq 19165 discard cfq 19129 base row 19074 noatime row 19040

3.8.7 Random Reading and Writing Multiple Memory-mapped Files (database)

All tested media and all file systems have the following type of performance for this workload:



Optimal settings for this workload:

Medium	**BTRFS**	**ext4**	**f2fs**
16 GByte class 10 SD card (Kingston)	ssd cfq 14668 ssd row 14603 ssd noop 14501 ssd deadline 14495 discard cfq 14368	nojournal cfq 12293 base cfq 12255 noatime cfq 12253 nobarrier cfq 12247 discard cfq 12219	discard cfq 14806 base cfq 14800 noatime cfq 14760 base row 14642 discard deadline 14635
16Gbyte class 10 SD card (Transcend)	ssd deadline 15973 ssd cfq 15930 ssd row 15907 ssd noop 15880 discard cfq 15654	nojournal cfq 14015 writeback cfq 13979 base cfq 13910 nojournal row 13888 noatime cfq 13839	discard cfq 16304 base cfq 16291 base deadline 16194 discard noop 16149 base noop 16135
16 Gbyte class 10 SD card (SanDisk)	ssd cfq 15322 discard cfq 15295 noatime cfq 15289 base cfq 15284 geometry cfq 15235	nojournal cfq 13112 writeback cfq 12937 discard row 12822 writeback row 12804 noatime cfq 12799	noatime cfq 15576 base cfq 15479 discard cfq 15377 noatime row 15150 noatime deadline 15054

16 GByte eMMC chip (Toshiba)	discard cfq 17396 base row 17366 ssd cfq 17323 geometry cfq 17323 base cfq 17305	nobarrier cfq 17561 writeback cfq 17537 discard cfq 17529 geometry row 17518 nojournal row 17505	noatime cfq 17825 base cfq 17754 discard cfq 17739 base row 17324 discard row 17296

3.9 Conclusion

Benchmarking results look mostly as expected: read bandwidth is limited by media, random write bandwidth is great with big FS cache and gets limited by media as I/O work set gets bigger than FS cache.

There's no single set of file system options/IO schedulers optimal for all media, each medium has its own performance profile.

Unidirectional (read only, write only) workloads are less sensitive to file system options and IO scheduler selection than bidirectional workloads.

4 SSD Wear Intensity

The physical media has a limited resource of operation in terms of, at least, maximal amount of block erase/write cycles. In case of SSD media a range of S.M.A.R.T. attributes describe device health, most notable are erase/program cycles counter (the total count of erase/program cycles for entire flash memory in its entire lifetime), and worst case erase counter (the maximum number of erase operations performed on a single flash memory block). When erase counter of particular block reaches the safety threshold (e.g. 100000 erase operations), the block cannot be used for writing any longer, so it should be either marked as read-only (if it contains valid data) or marked as non-available (cannot be selected by flash controller for writing new data, and therefore effective storage capacity lowers). Particularities of flash management algorithm are unknown to external observer and constitute the know-how of flash device manufacturer. However we can (safely) choose the following comparison criterion: we will say that of two file system configurations the one that gives slower ascending of total erase counter estimation is "better". In other words, we believe that any “decent” flash management algorithm will distribute the workload (almost) uniformly across all blocks in the storage, and reducing of total erase counter will proportionally increase expected device lifetime.

4.1 Approach

Straightforward comparison procedure is to retrieve total-erase-counter parameter before and after performance tests run, and to use relative growth of the counter serves as a score (to achieve the certain level of confidence the simulation may be repeated multiple times and received score is averaged). Of two file system configurations (different file systems or different configuration parameters of single file system) the one with lower averaged score observed in such way is considered as a preferred.

In practice, system designer is interested in expected lifetime duration of device that naturally varies with regard to intended usage scenario. The workload which is not specific to device may easily kill its storage quickly (e.g. tablet used as mail-server). Therefore it is clear that benchmark used for estimation of wear intensity shall be chosen to reflect “typical” usage. Such typical usage can be modeled roughly as number of primitive operations performed daily / weekly / monthly, so there is known relation between single benchmark cycle and “modeled” device run time (e.g. one cycle corresponds to 3 months of “normal” device usage). Basing on observed wear intensity and characteristics of underlying physical storage it is possible to derive expected device lifetime for given file system configuration, and that information may be used by system designer to make a decision on what file system to use. For example, if two configurations yield expected lifetime duration 30 and 100 years respectively, probably the parameter is not so important and system designer should base his/her decision on file system performance characteristics; in contrast, if it is 1 and 3 years, probably importance of that parameter becomes quite high.

4.2 Results

Unfortunately, none of the tested samples supports S.M.A.R.T. diagnostic interface, and described comparison method is not applicable. As there is no way to execute flash controller firmware in simulated environment (for instance, nandsim used as underlying flash media), the only way to compare two file system configurations is to wear off the device completely and to compare the time-until- failure. Such brute-force comparison procedure is too costly (in all senses) to perform, and therefore it has not been performed.

5 Power-Fail Tolerance

The aim of power-failure tests is to make sure that file-system and data integrity is preserved after unexpected power outages (or in some cases, after sudden system reset, e.g. triggered by hardware watchdog). It is expected outcome that data which existed only in operating system caches will get lost; more important is what happens to information that has been previously committed to persistent storage. Corruption of data that was successfully flushed to disk, or corruption of data that was not accessed at all, or even corruption of entire file system (i.e. inability to mount disk on next boot) are major faults that may forbid usage of specific file system configuration or even particular storage media unless hardware modifications are added that (practically) guarantee absence of power-failures.

From hardware perspective, certain operations are expected to be atomic. For instance, block erase operation or page writing operation all should be atomic. If operation is not completed for some reason, the state of flash cell array is, strictly speaking, unknown – a block may be erased just partially, or page programming may be incomplete. It is known that abnormal termination of such atomic operations may put flash array in internally unstable state (subsequent reading of the “partially-written” bits may get different values, and writing into “partially-erased” pages is not guaranteed to be correct), and suggested resolution is to repeat aborted procedure. As operating system is not managing flash (the task is performed by flash controller firmware), there is no real control over such “low-level” errors from file system standpoint, and consequently it is entire responsibility of the firmware to assure proper handling of events like these. File system, in turn, is responsible for consistency of its metadata and data (in other words, it shall recover from errors if underlying flash management layer is “reliable”), and it is expected that different configurations may show different behavior under power-failures.

There are a number of parameters that may get affected by power-failure. For instance, file system recovery procedure may get triggered on next boot resulting in significantly longer start-up time; or first file access time after power-failure may be longer. All such aspects may be important from general system operation standpoint, however in this section we will concentrate only on file system metadata and data integrity – something that we can assess relatively easily.

5.1 Approach

Power-failure test script (located at git://git.cogentembedded.com/pub/celf/pf-test) performs unattended simulation of power failures and analyzes file system and data integrity check results. Detailed execution log is saved for further examination. Power-fail test server residing on host manages the execution of various remote tests on target with particular file system configuration set up, and controls power source to initiate power outages at random time instants during remote test execution. Test server connects to the target and retrieves logging and diagnostic information.

The following steps are executed in a loop for each remote test and file system:

1. connect to the target;
2. perform mandatory file system check and recovery;
3. perform data integrity check if applicable;
4. mount current file system configuration;
5. start background test;
6. initiate power outage after random interval.

The following file system workloads are proposed:

- fsstress, that would stress file system metadata;
- custom SQLite test scripts, that would reveal occasional file data corruption.

Fsstress proved to be quite effective in detecting various file system implementation issues in the past. And SQLite is known for its extra care about power failures. Therefore, if SQLite database becomes corrupted, the most likely reason for this lies in file system or physical storage layers.

5.2 Results

Power-failure testing is time consuming process. Depending on required confidence degree, the complete testing round may take from one day to several weeks. (Of course, really bad configurations will be detected much earlier.) Therefore, this section provides just a few samples of what you get from power-failure tests, without trying to cover all possible configurations exercised by performance tuning procedure.

We used two sets of parameters for each file system: “default” and “the recommended performance-wise”, and executed the tests on PandaBoard with 16 GB class 10 SD card (Kingston).

It should be noted that optimal parameters from performance point of view may be the worst possible ones from power-failure tolerance perspective. This is especially true for ext4 which delivers the best performance with crucial safety features (barriers, journal) disabled.

5.2.1 Ext4

This proved to be the most robust file system among the tested ones, but only at the expense of improved performance. (In the absence of journal, `fsck` fails to auto-recover the file system after just a couple of power failures. Thus, this mode is not covered in the following performance-related comparison table.)

Default journaling mode, with and without ***nobarrier*** mount option enabled randomly, survived 1406 power cycles before `fsck` detected unrecoverable failure.

Note: “Manual recovery” `fsck` feature still allowed to bring ext4 partition into consistent state by blindly answering “yes” to very low-level and highly technical questions. But in real life this would likely mean loss of data.

It is interesting that the last mount command before the failure did not use ***nobarrier*** option. This may indicate that it is safe to use it in our configuration to improve performance without affecting power-fail tolerance. Although, more testing rounds are required to prove this.

5.2.2 BTRFS

The test fails after just a few power cycles because `fsck` finds errors which it is not able to auto-recover. This happened both with default mount options, and with performance-oriented ones. Since no safety-related parameters were disabled during the tests, the conclusion is to avoid using btrfs in that particular configuration if power loss events are anticipated.

5.2.3 F2FS

Unfortunately, F2FS does not have the `fsck` utility. Therefore, “mandatory file system check and recovery” step could not be executed for F2FS. This is probably the reason why it demonstrated the most troublesome behavior: hanged `sync` command during one round of testing, and severe file system issues revealed by manual checking of basic file system operations, (although, `mount` operation alone reported no problems), during another round. Thus, similar to btrfs, F2FS is not recommended for usage in this particular configuration in the presence of power failures.

5.2.4 Fsck, Mount, and Unmount Performance

The following table contains statistics for file system-related operations which may affect OS startup and shutdown times. Btrfs, f2fs, and ext4 rows represent statistics with default mount options, and corresponding *perf rows – with performance-oriented options: btrfs(ssd, noatime), f2fs(discard, noatime), ext4(nobarrier).

```
                      :    min |    max |   mean | median | count
btrfs      fsck   :   0.05 |   0.36 |   0.20 |   0.20 | 3
btrfsperf  fsck   :   0.03 |   0.12 |   0.06 |   0.04 | 5
f2fs       fsck   :    -   |    -   |    -   |    -   | 173
f2fsperf   fsck   :    -   |    -   |    -   |    -   | 1496
ext4       fsck   :   0.02 |   4.44 |   2.51 |   2.45 | 714
ext4perf   fsck   :   0.36 |   4.70 |   2.53 |   2.42 | 694

btrfs      mount  :   0.17 |   0.20 |   0.18 |   0.17 | 3
btrfsperf  mount  :   0.14 |   1.77 |   0.40 |   0.17 | 7
f2fs       mount  :   0.02 |   3.49 |   1.23 |   0.85 | 345
f2fsperf   mount  :   0.02 |   4.64 |   1.28 |   1.01 | 2990
ext4       mount  :   0.01 |   1.24 |   0.16 |   0.17 | 1426
ext4perf   mount  :   0.01 |   0.63 |   0.16 |   0.17 | 1387

btrfs      umount :   0.43 |   0.43 |   0.43 |   0.43 | 1
btrfsperf  umount :   0.29 |   0.37 |   0.31 |   0.31 | 5
f2fs       umount :   0.03 |   2.05 |   0.38 |   0.22 | 171
f2fsperf   umount :   0.03 |   4.35 |   0.74 |   0.78 | 1498
ext4       umount :   0.03 |   2.14 |   0.19 |   0.18 | 713
ext4perf   umount :   0.03 |   1.96 |   0.19 |   0.18 | 694
```

Since we got more-or-less trustful results only for ext4 file system, others are provided only as samples of what you can get from power-failure test logs.

An interesting observation is that maximum values could be much larger than the median ones. This should be taken into account while designing system with tight startup/shutdown time budget.

Appendix I. Benchmarking Scripts

Git tree with benchmarking scripts is available at
git://git.cogentembedded.com/pub/celf/flashopt

Tree Structure

```
./
├── benchmark/                       # benchmarks directory
│   ├── common                       # common parts of benchmark scripts
│   ├── benchmark-script             # script instance
│   ├── benchmark-script.config      # instance configuration
│   ├── benchmark-script.job         # FIO job definition (for fio scripts)
│   └── ...
├── fs/                              # file systems directory
│   ├── btrfs/                       # file system instance
│   │   ├── config                   # common parts of configuration scripts
│   │   ├── fs-configuration-script  # fs configuration script instance
│   │   ├── fs-configuration-script.config  # instance configuration
│   │   └── ...
│   ├── ...
│   └── common                       # common parts of all file system scripts
├── iosched/                         # IO schedulers directory
│       ├── config                   # common parts of configuration scripts
│       ├── ioscheduler-script       # IO scheduler configuration script instance
│       └── ...
├── result-xxx/                      # results directory
│   └── fs/
│       ├── <fs>/                    # results for the file system <fs>
│       │   ├── <fs-config>-<benchmark>     # result for <benchmark> in <fs-config>
│       │   └── ...
│       └── ...
├── config                           # common configuration parameters
└── run*                             # topmost run wrapper
```

All scripts in the tree are bash scripts, configuration files are sourced.

Configuration

./config file is the main configuration file, it defines parameters of the used media and file system options:

```
device=/dev/mmcblk0p3                # device used for testing
mount_dir=/media/mmc                 # test mount point
result_dir=result-kingston-sd8g4c    # result directory location
media_page_size=16384                # media page size
media_erase_block_size=4194304       # media erase block size

ext4_default="nobarrier"             # current configuration for ext4
ext4_exclude="geometry"              # list of excluded configurations for ext4
btrfs_default="noatime ssd"          # current configuration for btrfs
btrfs_exclude="geometry"             # list of excluded configurations for btrfs
f2fs_default="noatime"               # current configuration for f2fs
f2fs_exclude="geometry"              # list of excluded configurations for f2fs
```

A hint on the media geometry parameters may be obtained from ./run geometry.

Execution

Once ./config is filled ./run script may be started. For each missing file under ${result_dir} it invokes fs configuration scripts not mentioned in *_default variables to create and mount corresponding file system and then it invokes benchmarking scripts in turn and put their output into the ${result_dir}.

Once ./run is complete result comparison graphs may be built by running ./run gdiff.

Comparison graphs for arbitrary (but compatible) result sets may be built by calling benchmarking scripts with parameter 'gdiff' and a list of result files to compare.

Sample Usage

1. fill in ./config for the test:

   ```
   $ cat > config
   device=/dev/mmcblk0p3
   mount_dir=/media/mmcblk0p3
   result_dir=result-kingston-sd8g4c
   ext4_default=""
   btrfs_default=""
   f2fs_default=""
   ^D
   ```

2. Determine flash media parameters:

   ```
   # ./run geometry | tee -a ./config
   #
   # Guessed media geometry
   #
   media_page_size=16384
   media_erase_block_size=4194304
   ```

   Note that run geometry is essentially guessing based on the flashbench output, it may be unstable and inaccurate.

3. If the step 2 result does not look sane try to analyze flashbench output manually. Please see section 3.4.1 for the recommended way of doing that.

   ```
   # flashbench -a /dev/mmcblk0 --blocksize=1024
   align 134217728 pre 735µs    on 1.08ms    post 780µs    diff 324µs
   ...
   align 4194304   pre 741µs    on 1.08ms    post 788µs    diff 317µs
   align 2097152   pre 745µs    on 950µs     post 811µs    diff 171µs
   align 1048576   pre 745µs    on 945µs     post 807µs    diff 169µs
   ...
   align 16384     pre 745µs    on 911µs     post 781µs    diff 148µs
   align 8192      pre 785µs    on 808µs     post 725µs    diff 53.3µs
   align 4096      pre 784µs    on 788µs     post 779µs    diff 5.85µs
   ...
   ```

   That output suggests that media_erase_block_size should be 4194304 and media_page_size should be 16384. If the flashbench output looks chaotic and it's impossible to guess media geometry then geometry variation may be disabled:

   ```
   # cat >> config
   ```

```
ext4_exclude='geometry'
btrfs_exclude='geometry'
f2fs_exclude='geometry'
^D
```

4. run it:

   ```
   # ./run
   ```

5. see the results:

   1. plot performance parameters:

      ```
      $ ./run gdiff
      ```

      This will pop up a bunch of gnuplot windows with performance parameters for every benchmark that was run.

   2. see automatic analysis hint:

      ```
      $ ./run analyze throughput
      ```

      or

      ```
      $ ./run analyze latency
      ```

      This will print the best file system option, IO scheduler and associated performance metric for every file system and every benchmark.

6. choose file system options to apply, add them to the corresponding *_default parameters in the config file, e.g. set

   ```
   ext4_default=”nojournal noatime”
   btrfs_default=”ssd noatime”
   f2fs_default=”geometry”
   ```

7. change result directory, e.g. set

   ```
   result_dir=result-kingston-sd8g4c-iteration2
   ```

8. repeat from step 4.