# WHICH SDCARD?

Peter Chubb

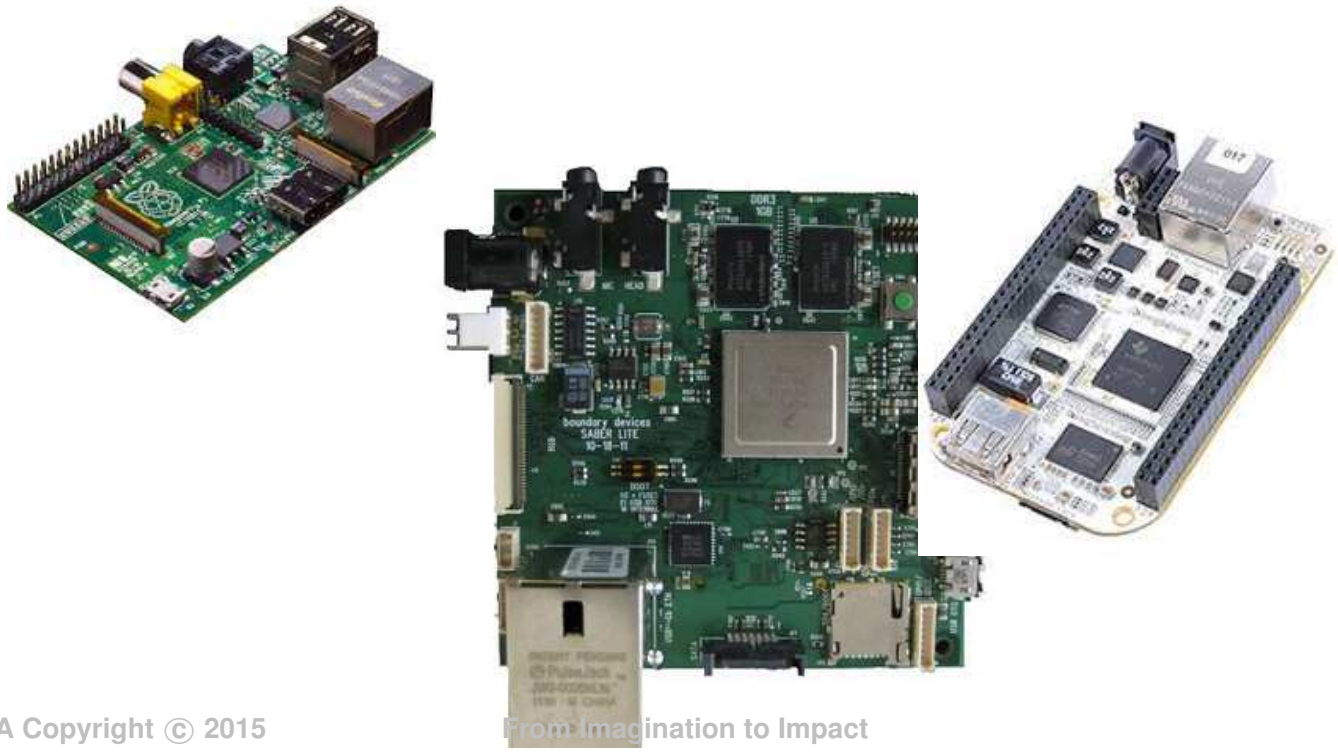
peter.chubb@nicta.com.au

# IT ALL STARTED WHEN...

NICTA



This story started about 2 years ago when we acquired a couple of Samsung TVs to display CI data and other useful things in our kitchen area at work. The in-built browser was fairly limited in what it could do, so, having a spare Raspberry Pi on my desk, I pressed it into service.
Performance sucked. And I traced that to the SD card.





More and more embedded systems that interest me come with only an SD card slot for their primary storage. So getting good performance from an SD card is fairly important to me.


# Buy a faster card!

NICTA

# Go Faster

# Marketing Buzzword

# 95MB/s!!!

# (with snazzy go-faster packag



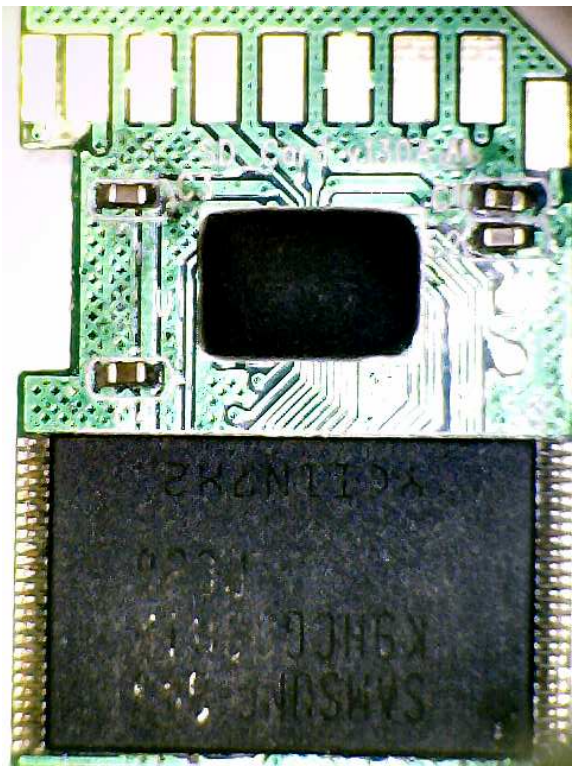

Spent lots of $$

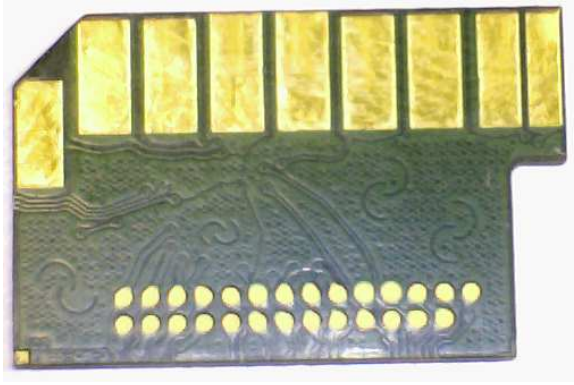
# Performance still sucks!

# INSIDE THE CARDS

NICTA





Here are some pictures. The one on the left is a fake Lenxs 16G SD card; you can see the 8Gb (Samsung) NAND flash chip, and the Chip-On-Board (COB) controller under the epoxy blob.
The one on the right is a SanDisk 1G card. Everything is encapsulated between the layers of the card, so there's not a lot to see.
I've been told that SanDisk design and fabricate their own flash and controllers; some other companies use third-party controllers and flash. This one is an AX215 from AppoTech.

# INSIDE THE CARDS


Bunnie Huang reverse engineered it

`http://bunniefoo.com/bunnie/sdcard-30c3-pub.pdf`

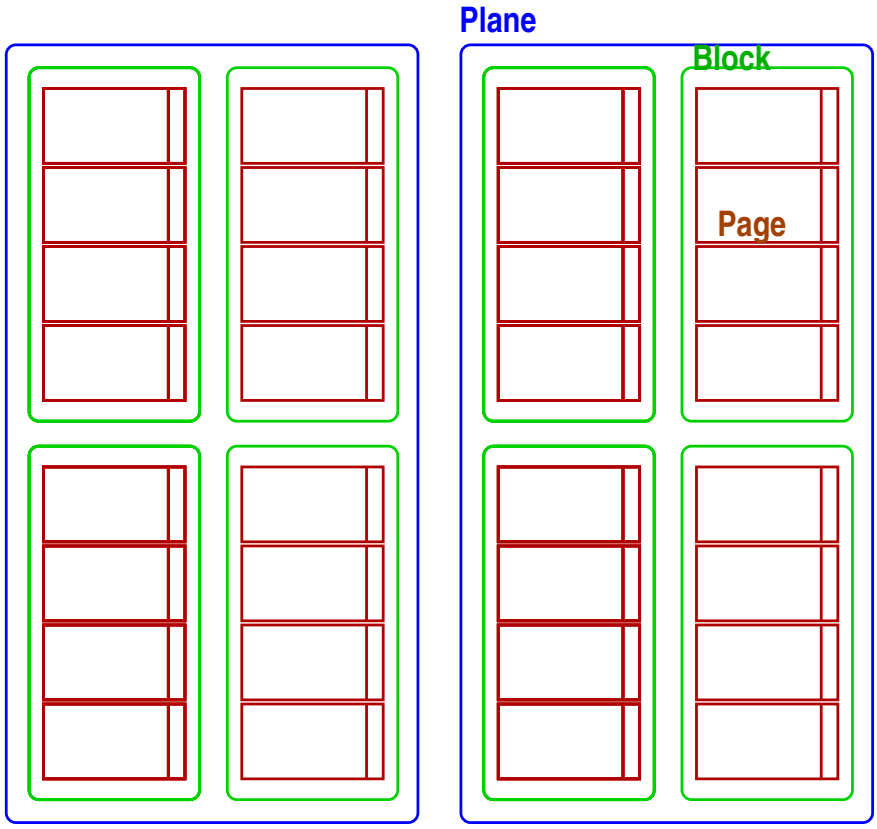
`https://github.com:xobs/ax2xx-code.git`

I thought at first I was going to have to desolder the chip and read the flash. But someone else beat me to it: `http://bunniefoo.com/k` and you can get the firmware from `https://github.com:xobs/ax2`

# FLASH MEMORY




**Typical:**
**Plane: 16G+ECC**
**Block: 4M+ECC**
**Page: 2k+112bytes**

There are basically two kinds of Flash memory: NOR flash, and NAND flash. Check the Wikipedia article if you want a more detailed view of how they differ; from a systems perspective, NOR flash is byte oriented so you can treat it like (slow) RAM; NAND flash is block oriented, so you have to read/write it a block at a time, like a disk.

I'm not going to talk any more about NOR flash; NAND flash memory has much higher density, and is the common cheap(ish) flash used on-board in embedded systems, SD cards, USB sticks and SSDs.

NAND flash comes with a variety of system interfaces. The most common are the Memory Technology Device interface (MTD), the MMC (Multi-Media Card) interface (a JEDEC standard, used for eMMC on-board flash and for SD cards) and a standard disk interface (such as used in USB sticks and in SSDs).



NAND flash is accessed as *pages*, typically 520, 2160 or 4320 bytes. A page is divided into payload and out-of-band data; the controller on the card usually uses the OOB bytes for ECC and other management functions.

Writes write only zeroes, so data has to be erased before it is written. The actual organisation of pages on the flash is usually proprietary, and depends on the precise detail of the flash architecture.

Erasure happens in larger units — an erase block can be two or four megabytes, or even more.

# TYPICAL TIMINGS

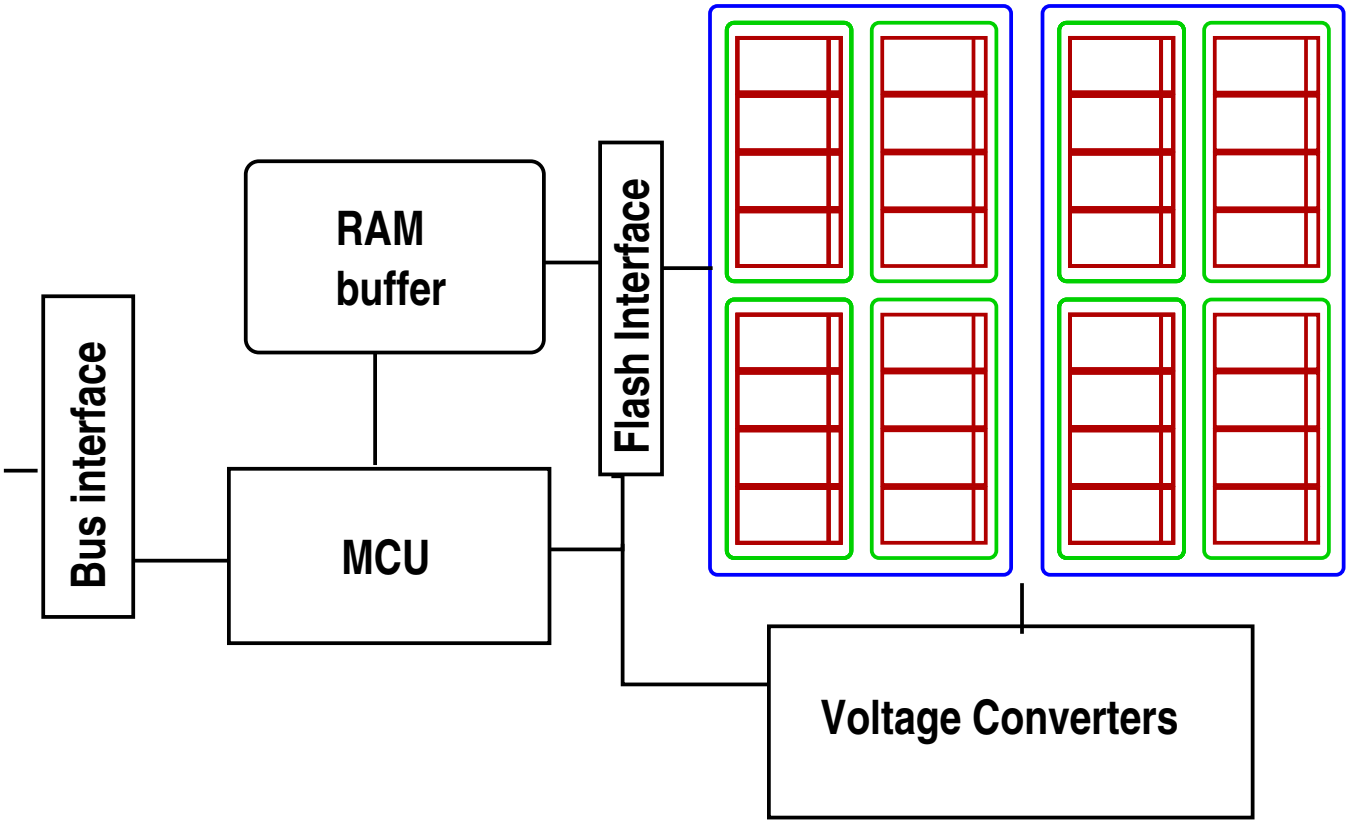
NICTA

- Write a page: $\approx$ 200 $\mu$s
- Read a 4320 byte page: $\approx$ 3 $\mu$s
- Transfer page to/from bus: $\approx$ 3nS/byte (12 $\mu$s)
- Erase a 138240 byte block: $\approx$ 3.5ms



These are typical timings for the current generation of raw flash. It's a bit hard to get definitive numbers, because so many flash chip manuals are available only under NDA. What's more, manufacturers keep improving things, so what's current may differ from this.

# INSIDE THE CHIP



Flash is really cheap in part because all flash is sold. It's cheap to put in a controller that maps out more than 80% of the area as bad blocks, and sell a nominal 16G chip as a 2G chip.
So in the same package with the flash, is a small controller that maps out bad blocks, and controls the erase/program/read state machines. It has a small amount or RAM, typically one or two pages, that act as a register to hold the page currently being read or written. It also presents the illusion of a contiguous set of erase blocks to the host.
To ease programming, when a chip has multiple planes, it is possible (especially on the higher-end chips) to read from one plane and write to another without going through the host interface. This eases garbage collection (see later).

# FLASH CHARACTERISTICS

NICTA

- Unreliable:
  - single/multi bit errors
  - Bad blocks
  - Read disturb
  - Limited life (maybe 100 000 erase cycles)
- Bit-serial access within a line

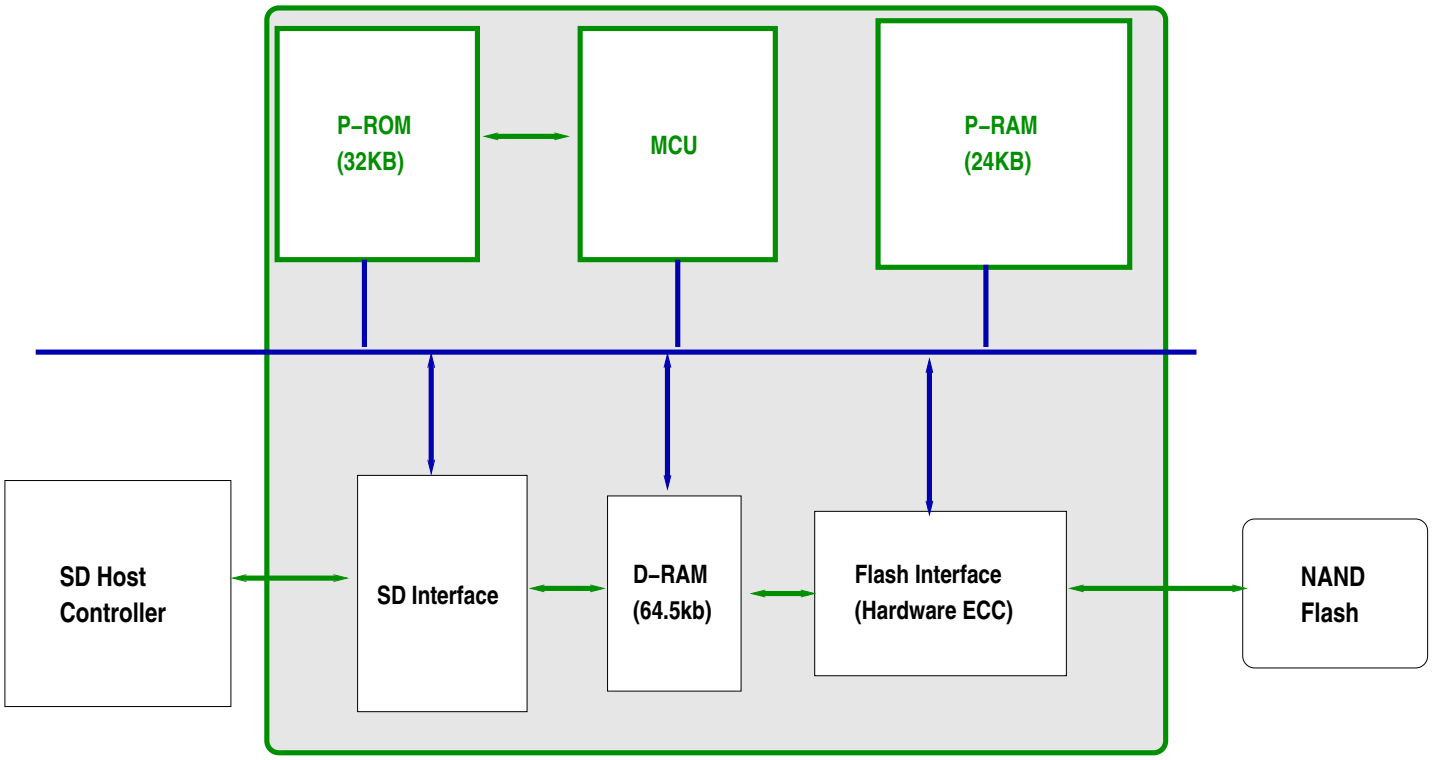


There are some other gotchas.
Blocks wear out. Currently available SLC NAND chips have around 100 000 erase cycles before they wear out (MLC chips have around half this); there is some research into adding heaters onto the chip to anneal and restore the cells, which would give three orders of magnitude better lifetime, but such NAND flash is not yet commercially available.
In addition, reading can disturb adjacent cells. Read-Disturb operates over a few hundred thousand read operations, and will cause errors, not in the cells being read, but in adjacent cells. Erasure fixes this problem.
These two characteristics together mean that flash has to be managed, either by a filesystem that understands FLASH characteristics (e.g., JFFS2, YAFFS), by an operating system component such as the UBI layer in Linux, or by a wear-levelling

translation layer and a garbage collector running on an embedded controller.
Flash technology changes fast enough, and best practice for managing the flash changes fast enough, that using managed flash is generally the best way to go for new designs.
What's more, manufacturers can merge the flash controller and management into a single MCU, not only saving costs, but also insulating the user from change in underlying technology.
Managed flash is what you get on an SD card, a chip with eMMC interface, or a USB stick.
Which is why so few hobbyist dev boards come with on-board flash any more.



# THE SD CONTROLLER



This picture is of a SiliconMotion SD controller. It's fairly typical of the low-end controllers. It has a modified 8051 8-bit microprocessor, and hardware accelerated ECC generation and checking. The 'Program ROM' is actually NOR flash, and used for housekeeping information such as where in the flash are the block maps, free maps and so on.
This particular controller has a single 64k buffer. This seems quite common in the cheaper 3rd party controllers, and matches quite well the 128-sector cluster size that the default FAT file system uses.
You'll have noticed that the available capacity on an SD card is significantly less than the rated capacity. Although the flash inside an SD card or USB stick is always a power-of-two size, it's quoted in thousands of megabytes, not 1024s of megabytes. The 'spare' capacity is used for holding metadata, erased blocks



for use on writes, and (perhaps) firmware for the controller.
Large flash manufacturers such as Toshiba, SanDisk and Samsung manufacture their own controllers. There are however a surprisingly large number of other companies that offer controllers. Third party controllers, because they are designed to interface with arbitrary flash chips, often offer in-system programming. This is used, on the one hand, by fraudsters wanting to sell you a card that identifies as 32 Gb when it only contains 8gb of flash; but also by manufacturers to allow sourcing from many different Flash manufacturers according to what's cheapest on the spot market.

# THE SD CONTROLLER


- Presents illusion of 'standard' block device
  - buffering, erase, serial-to-parallel, different block size, parallel access to different planes, etc., etc.
- Manages writes for wear levelling
- Manages reads to prevent read-disturb
- Performs garbage collection
- Performs bad-block management

Mostly documented (if at all) in Korean patents referred to by US patents!



The controller has to do a number of things, at speed. Total power consumption is fairly small — up to 2.88 Watts for a UHS-1 card; much less for standard cards. So it tends to be a fairly limited mmu-less microcontroller, such as the MCS-51 (or more usually, one of its variants). Higher end cards use an ARM processor; the cards that also do Bluetooth or WiFi often run Linux internally.
The main thing the controller has to do in its firmware is to present the illusion of a standard block device, while managing (transparently) the fact that flash cannot be overwritten in-place.
It also has to be aware when an SD card or USB stick is wrenched from its socket. The power pins on the card are longer than the others; this gives a few milliseconds of power to finalise writes, and to update the controller's NVRAM with the block address of any metadata in the Flash.

And finally, it does power management, to go to sleep when nothing is happening.



## WEAR MANAGEMENT


Two ways:

- Remap blocks when they begin to fail (bad block remapping)
- Spread writes over all erase blocks (wear levelling)

In practice both are used.
Also:

- Count reads and schedule garbage collection after some threshold

There are two ways to extend the limited write lifetime of a flash block. The first is to use ECC to detect cells going bad, and remap a block when errors start to appear. The second is to spread out writes over all blocks. By combining with garbage collection, such *wear levelling* can be achieved with low overhead.



# PREFORMAT

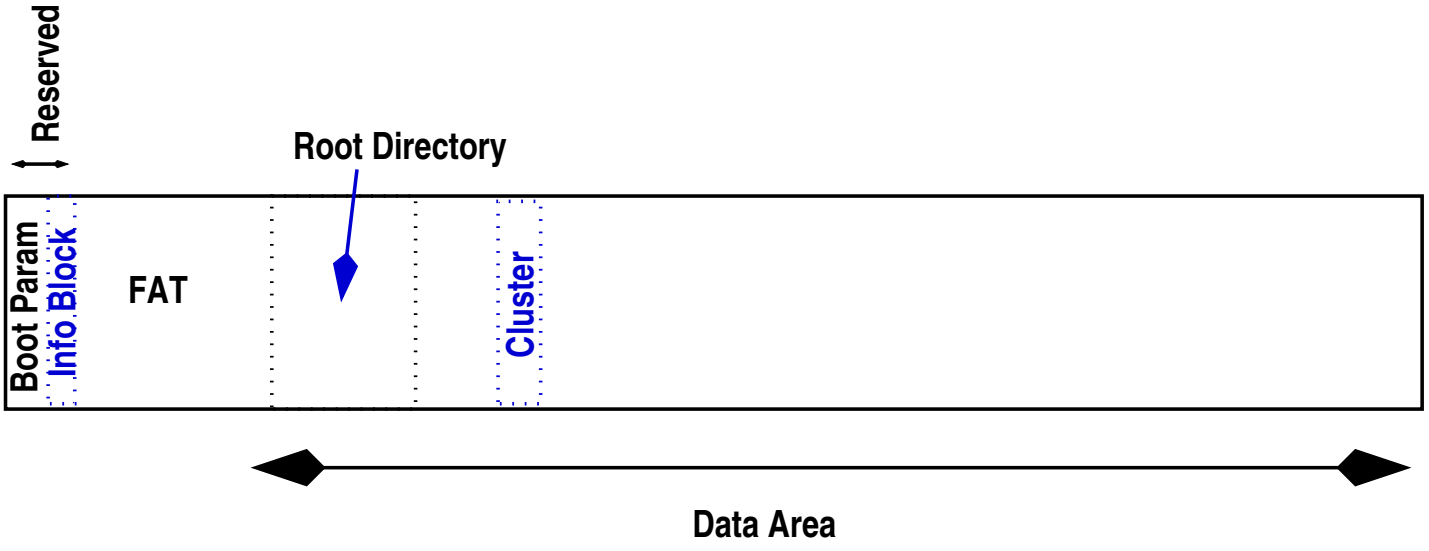

- Typically use FAT32 (or exFAT for sdxc cards)
- Always do cluster-size I/O (64k)
- First partition segment-aligned

  Conjecture Flash controller optimises for the preformatted FAT fs

Removable Flash devices almost always are preformatted with a FAT file system. Typically, the first partition starts on an erase boundary (thus being a good hint for the size of the erase block), and uses a cluster size that is a good compromise for the allocation unit size used by the controller. It's likely that the controller will optimise for the write patterns experienced when using a FAT fs.



# FAT FILE SYSTEMS



The reserved area at the start of the filesystem contains a Boot Parameter Block (essentially the same as a superblock on a UNIX file system). Key parameters are the location, size, and number of FATs (file allocation tables), the location of the root directory, and the cluster size.
All disk allocation and I/O is done in cluster-size chunks.
We looked at a lot of different preformatted cards. All used a 64k cluster size, and had the FATs in the second erase block of the disk. The first erase block was used only for the MBR.
The Directory entry for a file contains the index of its first cluster. The first cluster index is then used to look up the next cluster in the FAT, as a chain. So extending a file involves writing the data into the data area, and cluster indices into the FAT, and finally updating the directory entry with the file size.



# INTERFACE SPEED


- SPI mode: 3 Mb/s
- Standard SDHC cards: up to 25MB/s (DDR); SD cards 12.5MB/s
- UHS-1 up to 104MB/s (DDR)

When an SD card is first plugged in, it works at 3.3V, and in SPI interfacing mode. The host has to query the card's capabilities, then switch it into a higher-speed mode if possible. Cards have 4 data pins; higher speed cards can transfer a nibble on both rising and falling clock edges (Double Data Rate) giving at 25MHz clock speed a 25 MB/s transfer rate. UHS=1 cards can drop the voltage to 1.8V, and up the clock to 104MHz, giving up to 104MB/s transfer rate.
There are also UHS-2 and UHS-3 standards, but I haven't seen any host adapters available for them yet.
Cards are generally downward compatible. UHS-1 cards will work at SDHC and SD speeds. However, to gain the full advantage of the extra speed, you need the appropriate host controller.



# INTERFACE SPEED


Class 2, 4, 6 — write speed of $\geq$ 2, 4 or 6 MB/s when fragmented

Class 10 — write speed of $\geq$ 10MB/s when unfragmented

UHS-1 – cards usually have higher rated speeds than Class 10.

For Linux — often get better long-term performance from a class 4 or 6 card than a cheap class 10.

Note that a class 10 card matches pretty well its intended use in a camera or similar — it gets written to when in the camera, read once to upload, then reformatted to start again.



# MANAGING FLASH


- Controller has RAM buffer
  — Size of buffer seems to increase with price of card
- Has some number of ‘open’ blocks.
- Reading/writing to an open block cheaper than to a non-open block.

Thanks to Arnd Bergman for flashbench

`git://git.linaro.org/people/arnd/flashbench.g`

Given that the SD I/O speed is higher than the speed to write to Flash, the controller must have RAM to buffer the writes.
Given also that these controllers are very very cheap, they have strictly limited RAM for buffering. Given also that they want to minimise the overhead in address translation and garbage collection,
We timed writes at various offsets from each other to determine the size of the buffer (we expect that two adjacent writes within the same open buffer will be fast, but when the buffer finally is committed to flash, it'll be slower — which is what we found), and to discover how many write loci could be handled simultaneously.
There's a tool written by Arnd Bergman that does this for you semi-automatically, called 'flashbench'. It's in the Debian repository so easy to install.



# MANAGING FLASH


```
$ flashbench -a /dev/mmcblk1p1
align 67108864  pre 1.9ms       on 2.73ms       post 1.85ms     diff 858µs
align 33554432  pre 1.9ms       on 2.73ms       post 1.84ms     diff 858µs
align 16777216  pre 1.9ms       on 2.73ms       post 1.84ms     diff 861µs
align 8388608   pre 1.9ms       on 2.73ms       post 1.85ms     diff 855µs
align 4194304   pre 1.87ms      on 2.7ms        post 1.85ms     diff 840µs
align 2097152   pre 1.81ms      on 2.21ms       post 1.84ms     diff 381µs
align 1048576   pre 1.81ms      on 2.21ms       post 1.85ms     diff 377µs
align 524288    pre 1.81ms      on 2.2ms        post 1.84ms     diff 375µs
align 262144    pre 1.81ms      on 2.2ms        post 1.85ms     diff 376µs
align 131072    pre 1.81ms      on 2.21ms       post 1.85ms     diff 380µs
align 65536     pre 1.81ms      on 2.21ms       post 1.84ms     diff 379µs
align 32768     pre 1.82ms      on 2.18ms       post 1.82ms     diff 358µs
```

With `-a`, flashbench reads 3 1k blocks around a number of power-of-two boundaries, first ending at the boundary, then across the boundary, then just after the boundary. Crossing an erase block boundary should be slightly more expensive than if all blocks are in the same page. In this case that happens at the 4M boundary. Not shown on the figure is you can also discover, or at least intuit, the page size, and often the number of planes. And the tool has other functions to work out the number of open allocation units.



# NORMAL FILE SYSTEMS


- Optimised for use on spinning disk
- RAID optimised (especially XFS)
- Journals, snapshots, transactions...

Most ‘traditional’ file systems optimise for spinning disk. They arrange the disk into regions, and try to put directory entries, inodes and disk blocks into the same region.
Some also take RAID into account, and try to localise files so that a reasonably-sized single read can be satisfied from one spindle; and writes affect only two spindles (data plus parity).
Many use a journal for filesystem consistency. The journal is either in a fixed part of the disk (ext[34], XFS, etc) or can wander over the disk (reiserFS); in most cases only metadata is journalled.
And more advanced file systems (XFS, VxFS, etc) arrange related I/O operations into transactions and use a three-phase commit internally to provide improved throughput in a multiprocessor system.



# TESTING SDHC CARDS

NICTA

We ran fairly extensive benchmarks on four full-sized cards using a SabreLite: a Kingston 32Gb class 10, a Toshiba 16Gb class 10, and two different SanDisk UHS-1 cards: an Extreme, and an Extreme Pro.
The SabreLite does *NOT* have a UHS-1 controller, so transfer times are limited to SDHC rates.

# SD Card Characteristics

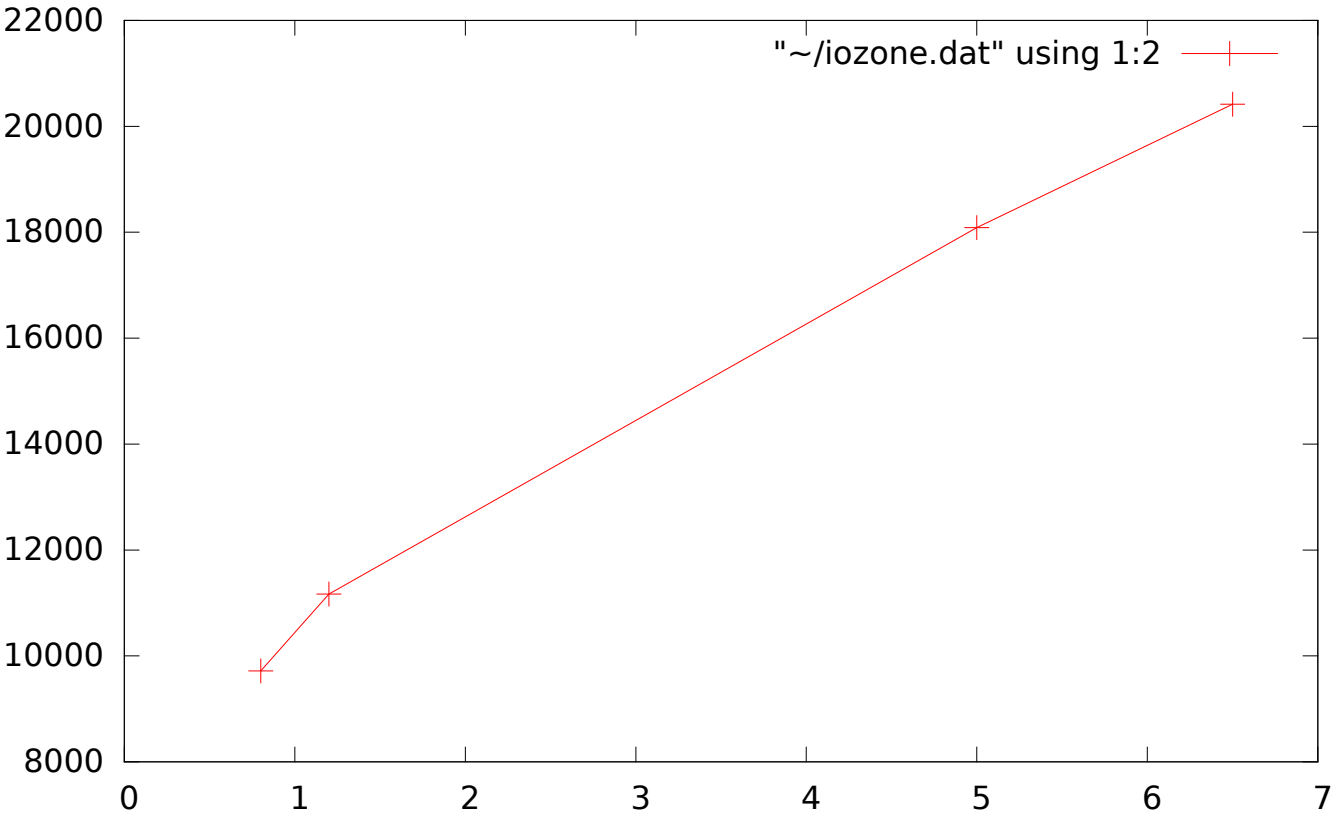

| **Card** | **Price/G** $ | **#AU** | **Page size** | **Erase** Size |
|---|---|---|---|---|
| Kingston Class 10 | $0.80 | 2 | 128k | 4M |
| Toshiba Class 10 | $1.20 | 2 | 64k | 8M |
| SanDisk Extreme UHS-1 | $5.00 | 9 | 64k | 8M |
| SanDisk Extreme Pro UHS-1 | $6.50 | 9 | 16k | 4M |

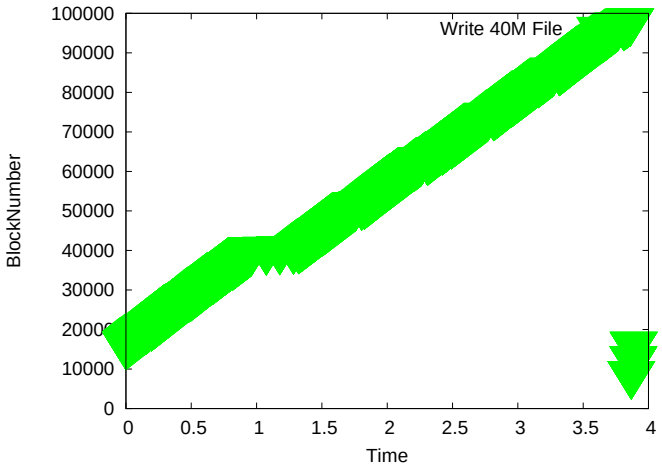
The Toshiba card we measured had two open allocation units, but didn't seem to treat the FAT area specially.
SanDisk and Samsung cards had between six and nine allocation areas, and didn't seem to treat the FAT area specially.
For the Kingston cards, which have only two open allocation units, one of them appears to be pinned to the FAT area. So you can do fast writes to a single open file, extending it in the FAT area, and in the data area. But multiple files are slow, and any filesystem that doesn't use the FAT area in the same way will be slow.

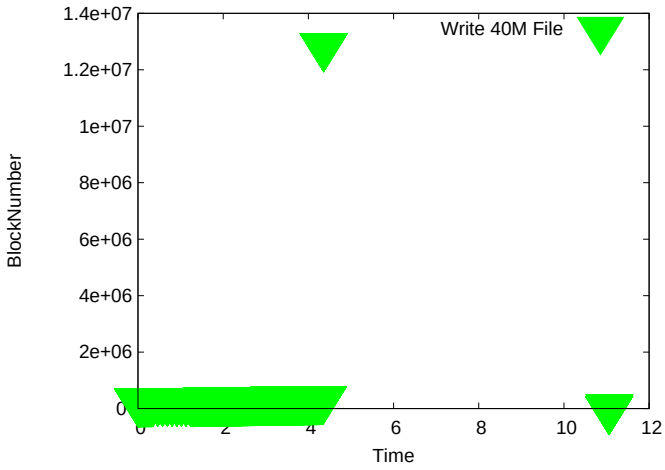


# SD Card Characteristics



# WRITE PATTERNS: FILE CREATE




(On Toshiba Exceria card)



You can see from the blktrace plots that creating a file on a FAT32 file system is almost ideal for a two open-allocation-unit system with a 64k buffer. All writes are in 64k chunks; the FAT remains open for extensions; and the buffer becomes available for the directory entry after closing off the last write to the file's data. However, ext4 doesn't behave like this at all. Ext4 on an unaged filesystem is pretty good at maintaining locality, but has some writes a while after the file is closed: one to update the free-block bitmap, one for the inode, and one for the journal — and these are non-local with respect to the last write.

# F2FS

NICTA

- By Samsung
- ‘Use on-card FTL, rather than work against it’
- Cooperate with garbage collection
- Use FAT32 optimisations



The question is, can better be done taking into account what the flash controller does?
F2FS was designed by Samsung for SD cards and other higher-level Flash devices. It works reasonably well on the cheaper USB sticks and SD cards.

# F2FS

NICTA

- 2M Segments written as whole chunks — always writes at log head
  — aligned with FLASH allocation units
- Log is the only data structure on-disk
- Metadata (e.g., head of log) written to FAT area in single-block writes
- Splits Hot and Cold data and Inodes.



It is designed to work *with* the flash translation layer. It understands the way that garbage collection might happen, and that full-segment writes are the most efficient. It uses log-based allocation to make the most of the FLASH characteristics.
It also divides files up based on their type. Files that are likely to be long lived, and written only once (e.g., photos, movies), are marked as ‘cold’ and stored in a different area of the file system from other files. This helps the garbage collector.
F2FS works best on cards that can write to at least 6 places at once.

# NILFS2

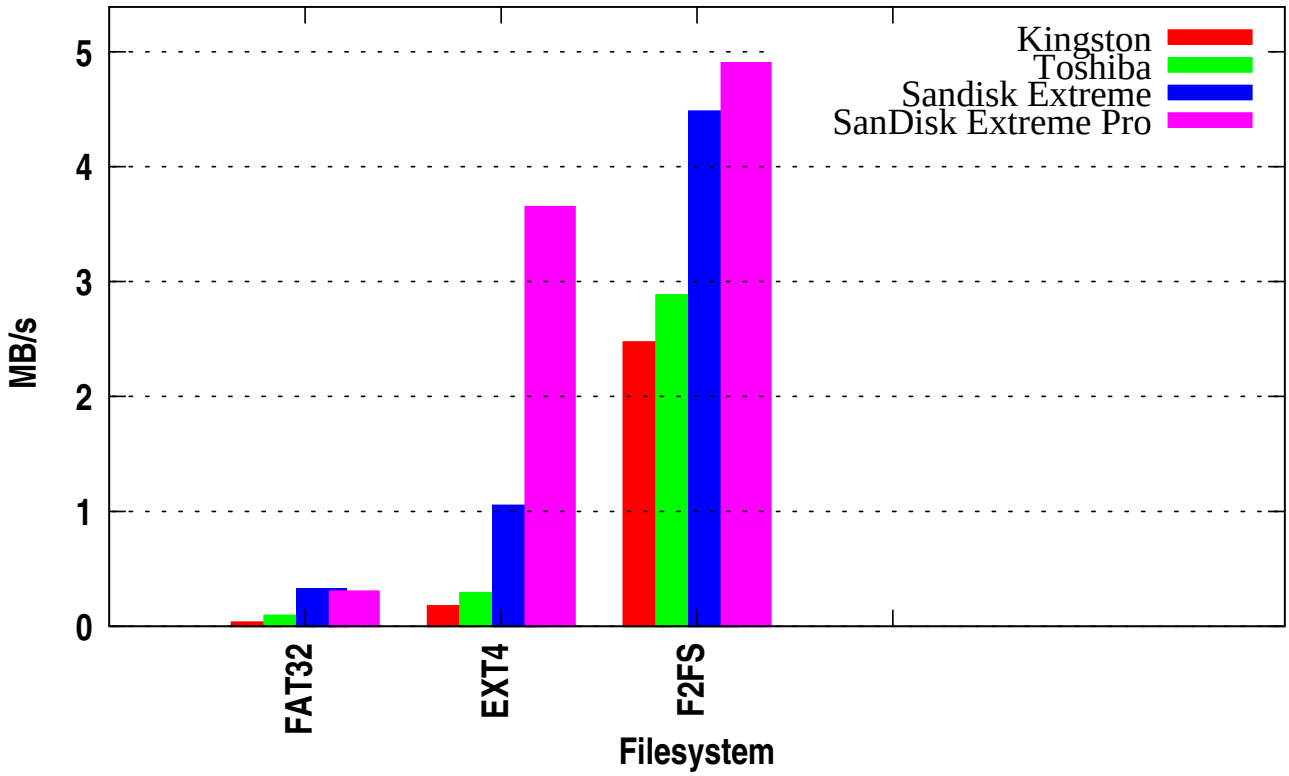
NICTA

- Log structured
- Defers garbage collection to user process: lower latency than F2FS
- All data is in the log; only one write locus.



NILFS2 is another file system that is meant to be flash-friendly. It has been in the mainstream kernel for longer than F2FS, and so ought to be more mature; but I don't know how much use it has had.
It also is a log structured file system, but is only written to at one point, rather than splitting a write stream into up to 6 places. All data goes into the log; the log is written in segments. The size of a segment is controlled at mkfs time; typically 8M (but you can gain performance by making it the same as the erase block size). Each segment has a summary at its start, that says what each block in the segment is.
The filesystem does continuous snapshotting; a user-mode daemon deletes old snapshots according to the preservation-time set at mount time, typically 120s.

# BENCHMARKS: POSTMARK 32K READ



Postmark, which writes lots of small files, showed massive differences between the cards. As its read and write sizes are much less than the page size, it forces a program/erase or a garbage collection on every write — making it worse case for the cards. The file systems that hide this (F2FS) do much better, even on the cheapest card.

# FS SUMMARY

NICTA

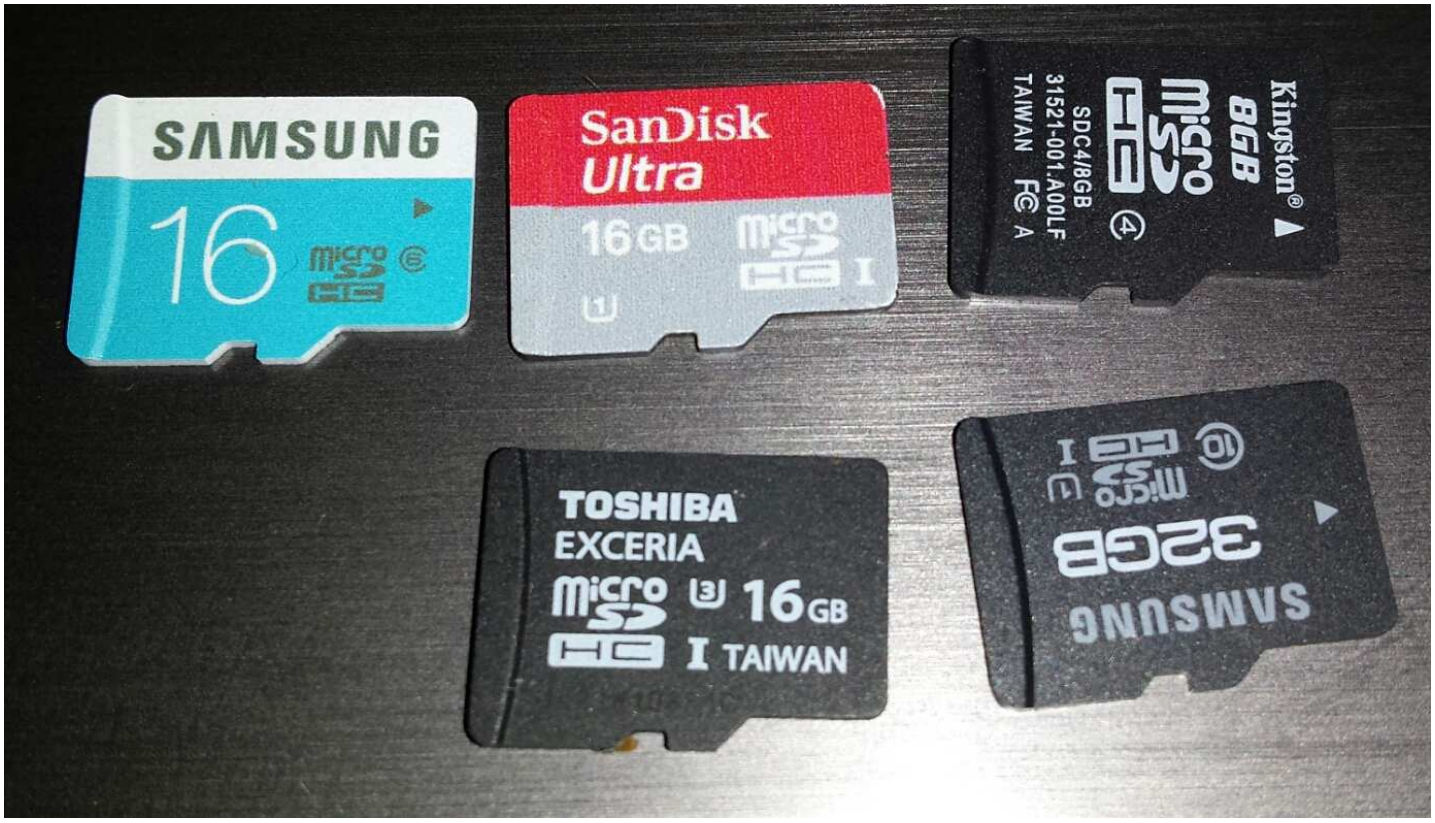
| File System | Min Writes per file | Allocation unit |
|---|---|---|
| ext4 | 4 | 4k |
| nilfs2 | 1 | 8M |
| f2fs | 1 | 2M |
| FAT32 | 2 | 64k |

To compare FS then, ext4 writes at at least four places on the flash for every file creation: in the journal, the file data, the directory entry and the inode.
The other two Linux FS write only to their logs.
And FAT32 writes to the FAT and to the data area.

# MORE CARDS






The next set of hardware we bought used micro-SD cards, but had a UHS-1 controller.

# CHARACTERISTICS

NICTA

| Card | Price, RRP $/Gb | Erase Size Mb | Open AU |
|---|---|---|---|
| Samsung EVO class 6 16G | 0.62, 1.87 | 4 | 7 |
| Sandisk Ultra 16G | 0.87, 3.12 | 2 | 4 |
| Kingston 8G | 0.93, 8.74 | 4 | 6 |
| Toshiba Exceria 32G | 1.23, 2.49 | 128 | 8 |
| Samsung Pro 32G | 1.34, 3.12 | 4 | **24** |

I bought a handful of different micro-SD cards in September last year. Prices per gigabyte reflect what I paid for them — but prices are really volatile, and street prices do not reflect Recommended Retail Prices (RRP). I put in the RRP as well, for information only.

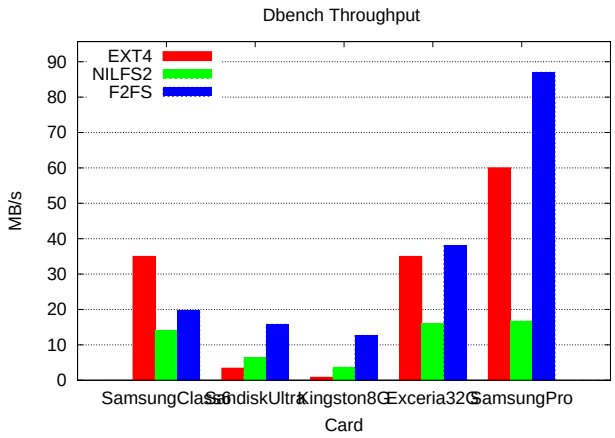
The two cards that cost more than $1 per gigabyte are the most interesting. The Toshiba one has a huge erase block size — it's possibly using TLC with 8-bits per cell, and probably has more than two planes. The Samsung Pro was hard to pin down as to its characteristics. It gave a slight timing glitch around the 4M mark, and so I'm assuming a 4M erase block size. But testing for the number of open blocks using that size gives around 24. So I'm thinking the controller may be doing something clever; the measurement may be reflecting running out of buffer space rather than running out of open AUs.

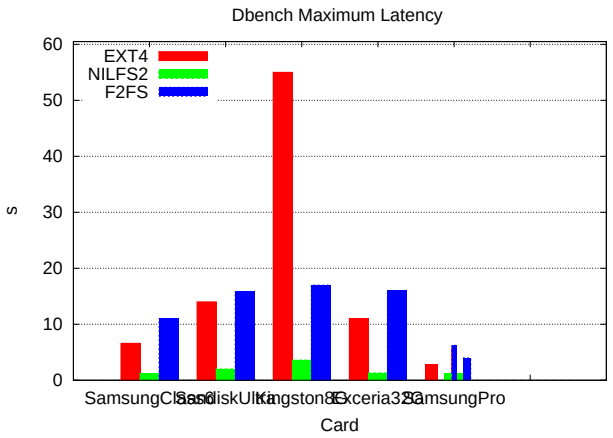
 

The Sandisk Ultra is also interesting. Although it has a smaller erase size, and fewer open allocation units, it outperforms some of the other cards.



# BENCHMARKS



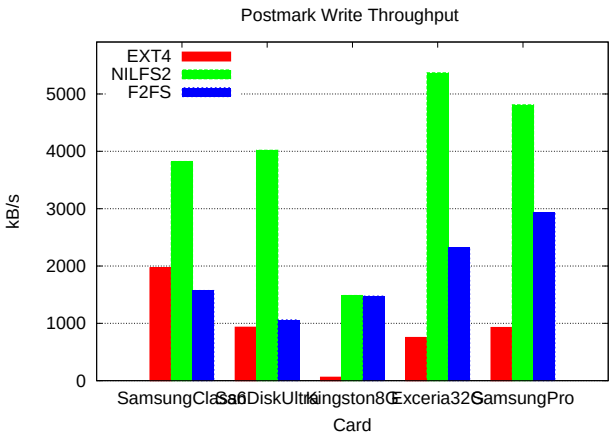



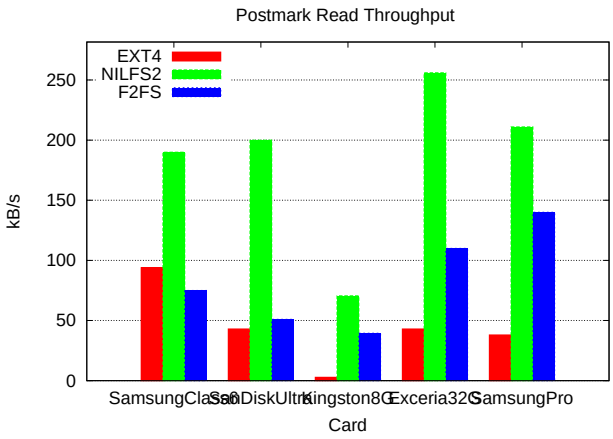
I ran dbench on all the cards using all the filesystems we used before.
On the graphs, the cards are ordered by increasing price per gigabyte across the bottom. What's clear is that different file systems behave differently on different cards; and that NILFS-2 trades latency for throughput.
Also clear is that the Kingston card is overpriced for its performance.



# BENCHMARKS



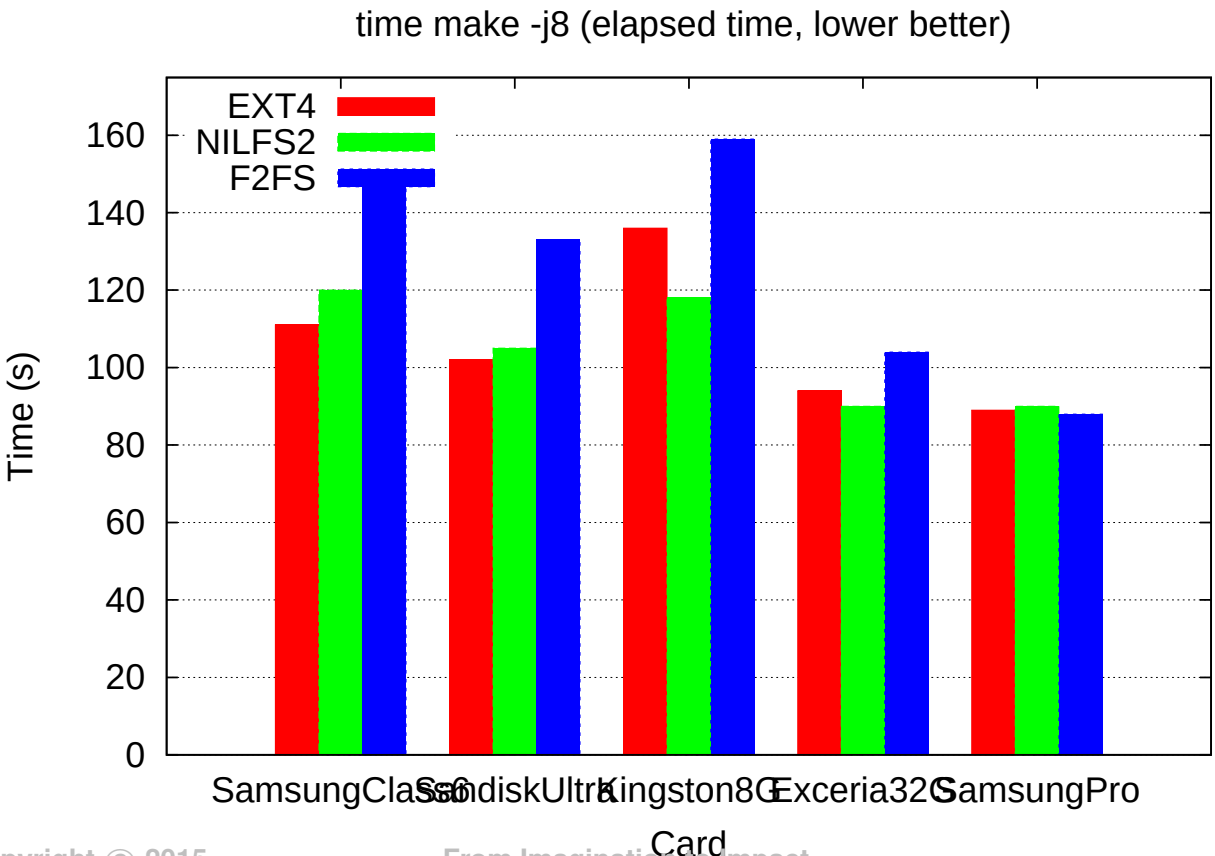



Postmark shows the reverse. NILFS2 works much better for the lots of small files case than F2FS or EXT4.

# KERNEL ALLNOCONFIG BUILD TIMES



But what of something I actually care about? The kernel build times (each average of three runs; variances are too low to show on the graph) show that the filesystems don't make as much difference on the more expensive cards (as expected), but also that ext4 isn't a bad choice.
The faster cards are fast enough that the kernel compile moves from being I/O to CPU bound. So you can't say much from this.
F2FS performs badly on the slower cards primarily because it has not been tuned for multi-threaded operation. Using lock_stat shows excessive hold times on some of its semaphores leading to a maximum wait time, in some cases, of over 6 seconds during a dbench run, with a maximum hold time of over 23 seconds.
Also



# OTHER GOTCHAS


- Some card/FS have very high latencies.
- NILFS2 cleaner can die; flash fills up
- No FS works well when flash is full
- F2FS still fairly immature

I'd also bought a Samsung EVO UHS-1 $\mu$SD card. It showed massive (more than 30s) latencies when attempting to run the various benchmarks, which caused the kernel to drop the transactions. It looked like the controller/garbage collector just wasn't coping with the load, under F2FS. Other filesystems were OK. On measuring it turned out this card draws too much current for the onboard v1.8V regulator on the Odroid XU3 — the card is however still within spec (a UHS-1 card is allowed to draw up to 2.88W, or 1.6A).

We've been running NILFS as the root FS for a Raspberry PI for the last eighteen months or so. Every now and then, either the nilfs-clean daemon doesn't start on mount, or it dies (not sure which); then the flash card fills up with snapshots and eventually the system grinds to a halt.

All the FSs tested degrade when the flash gets full, as they can



no longer use their optimised allocation algorithms. Instead they write to wherever there's room.

And as seen, F2FS has some issues with multithreaded workloads. I expect this will improve over time.

# SECURITY CONSIDERATIONS

NICTA

- MITM attacks possible
- Bunnie Huang,
  `http://www.bunniestudios.com/blog/?p=3554`
- Don't trust random cards you find in the street!



Finally, the controllers in many cards offer in-system reprogramming. This is so that the manufacturer can program the controller to match the flash that was actually installed in the card (and to allow fraudsters to adjust the capacity up before selling as a fake). However, because the manufacturers often do not turn off the facility, anyone can install new firmware.
This offers the possibility of man-in-the-middle attacks, where the file you get is not the one you think you're getting (allowing virus and Trojan injection).
Moreover, it may be possible to get an SD card to identify as an SDIO device, and fool the host in a similar way to the USB hacks we know about. This would be a little harder as fewer systems allow SDIO by default; but the same basic attack issues are present. An SDIO device allows configuration of up to eight virtual devices, allowing the expected memory card to be visible

at the same time as a network device, say.



## SECURITY CONSIDERATIONS


The hack code for AppoTech is at:
`https://github.com/wom-bat/ax2xx-code`.
Runs on Novena, Odroid XU3 and SabreLite.

I ported Bunnie and Xobs's code to run on the Odroid XU3 (for UHS-1 $\mu$SDcards) and on the Sabre Lite (for standard SD cards). To run them, you need to disable the appropriate SD card host in the flattened device tree before booting the board.
So far I haven't found any cards that use this controller.



# CONCLUSIONS


- Buy a suitable card – look for bargains
- Align partitions to erase blocks — Linaro images align too small
- Use a suitable filesystem — NILFS2 or EXT4 or F2FS depending on card
- Have fun attempting to hack cards
- But beware fakes.

Questions?

So in conclusion: try to get a real high-quality card, work out what its erase size and number of open allocation units is, and tune your FS to match.
And have fun attempting to hack the cards!