

# Tuning OLTP RAID IO

Paul Tuckfield

# I'm talking OLTP, and just IO

- You need to make sure you've done other things first, and have minimized your IO load, and arrived at the point where you must improve IO performance
- You bought a multidisk controller, you should make sure it's actually using **multiple disks** concurrently.

# 4 ways to make OLTP IO faster

- 0.) Do not do IO for the query
- 1.) Do IO only for the rows the query needs
- 2.) if you must do lots of IO, do it sequentially (read ahead) but only in the DB not in the fs or raid
- 3.) make sure you're really doing concurrent IO to multiple spindles (many things can prevent this)

digression: Sometimes it's faster to stream a whole table from disk, ignore most of the blocks, even tho an index might have gotten only the blocks you want. Where is the tradeoff point? in OLTP almost never, because **it flushes caches**

# 4 ways to make OLTP IO faster

- 0.) Do not do IO <- **Tune your queries first**
- 1.) Do as few IO as possible <- **Tune DB cache first**
- 2.) if you must do IO, do it sequentially <- **not OLTP workload, make sure filesystem and raid arent trying to “help” you by doing readahead**
- 3.) make sure you're really doing concurrent IO <- **know when you're serializing or getting concurrent IO then tune IO stack**

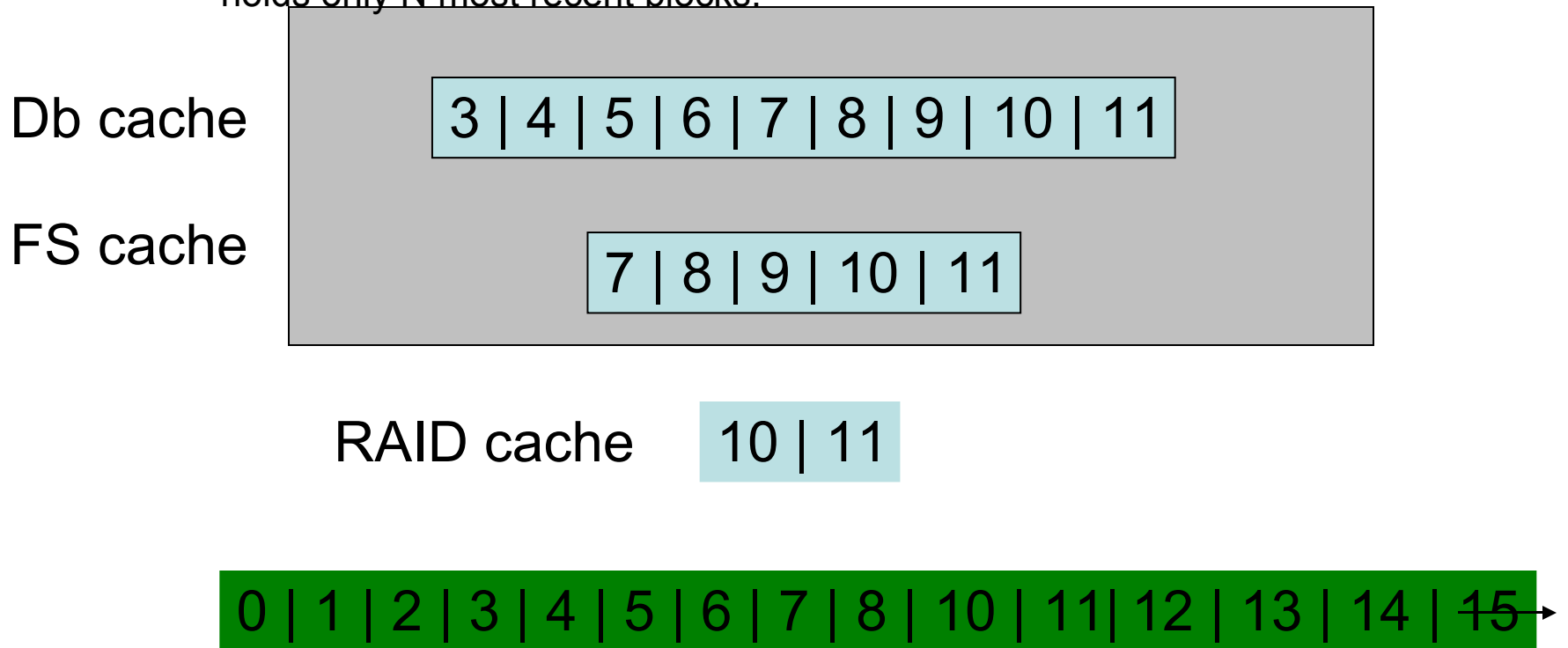
digression: flash drives. how will they change this?

# Generic rules for OLTP IO tuning

- Never cache anything below the DB
- Cache **only** writes in the raid controller (the sole exception to rule 1, assuming battery backed raid cache)
- Never do read ahead anywhere but in the DB similar to not caching outside db but then is the DB doing the right thing?
- Make sure your stripe/chunk size is much larger than db blocksize
- make sure you're **actually doing** concurrent IO

# Only DB should cache reads

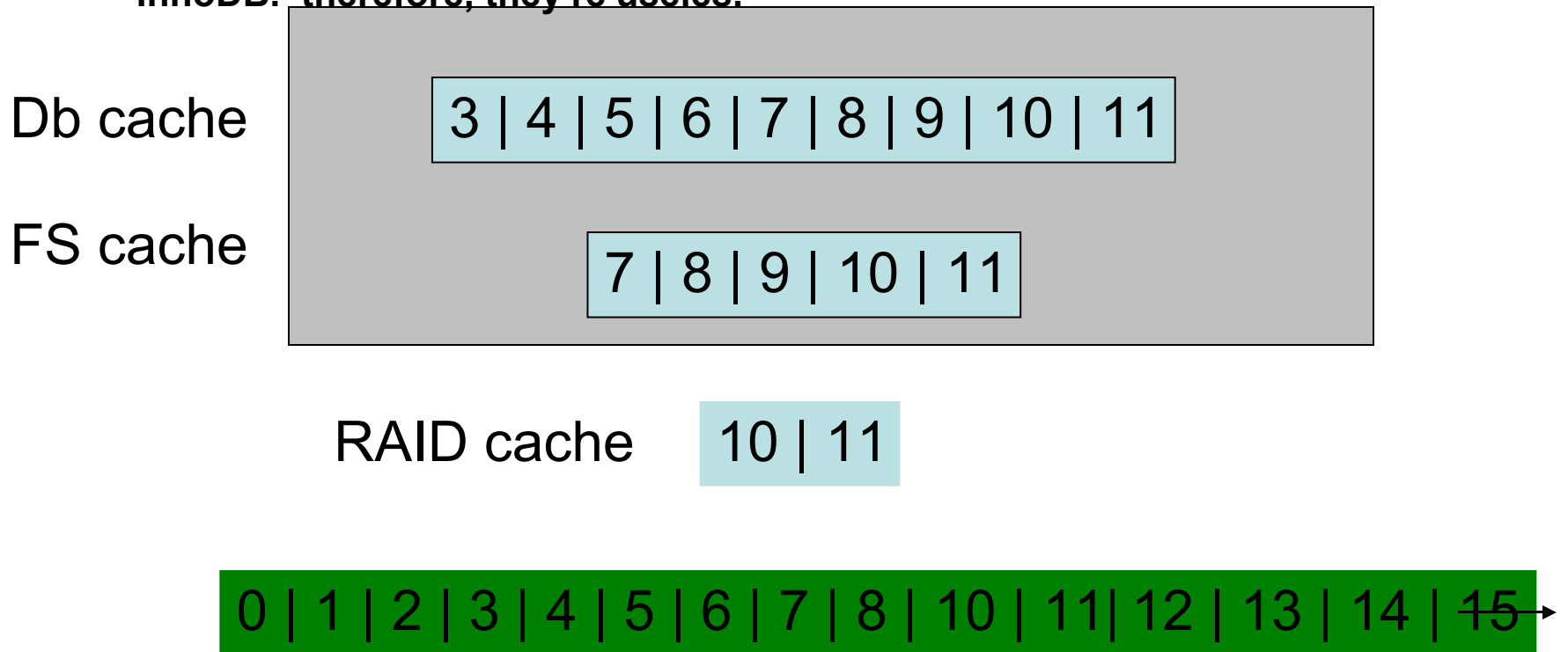
In a reasonable InnoDB setup, your db cache will be smaller than you disk based dataset, the filesystem cache (if you insisted on having one against my advice) should be smaller, the raid cache will be smaller still (ram is cheaper than a disk array) Imagine the db has read blocks 1 thru 11 , this is what the caches look like. Each cache holds only N most recent blocks.



# Only DB should cache reads

Suppose a read of any block: If it's in any of these caches, it will also be in the DB cache, because the db cache is the largest. **therefore the read caches of the file system and raid controler will never be used usefully on a cache hit.**

Suppose it's an InnoDB cache miss on the other hand: **it won't be in fs or raid cache then, since they're smaller and cant have a longer LRU list than InnoDB. therefore, they're useles.**

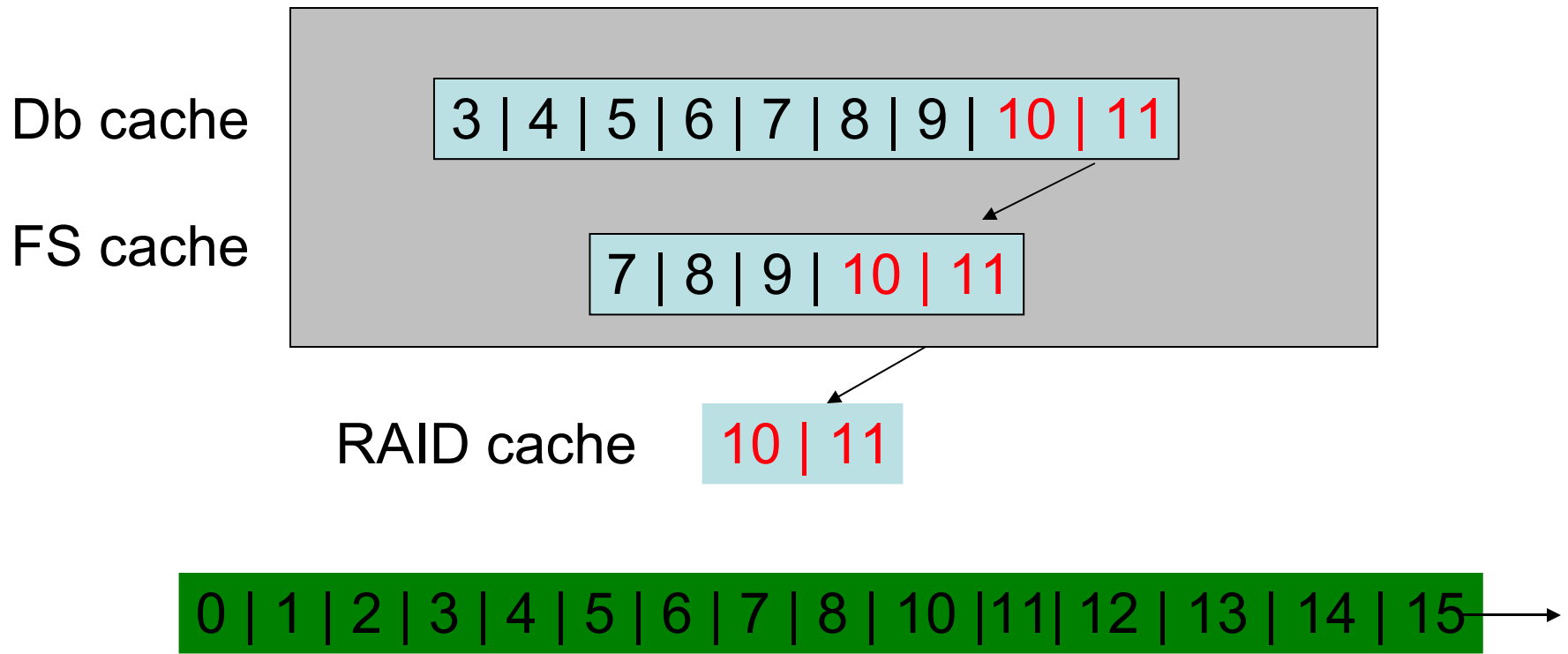


# Only the DB should readahead

- the db **knows** or at least has far better heuristic data to know when it needs to read ahead( Innodb method is questionable IMO)
- filesystem, raid are making much weaker guesses if they do readahead, it's more or less hard coded, not good. nominal case should be a single block.
- When they do readahead, they're essentially doing even more read caching, which we've established is useless or even bad in these layers.
- Check “avgrq-sz” in iostat to see if some upper layer is doing > 1 block (16k in the case of innodb) read

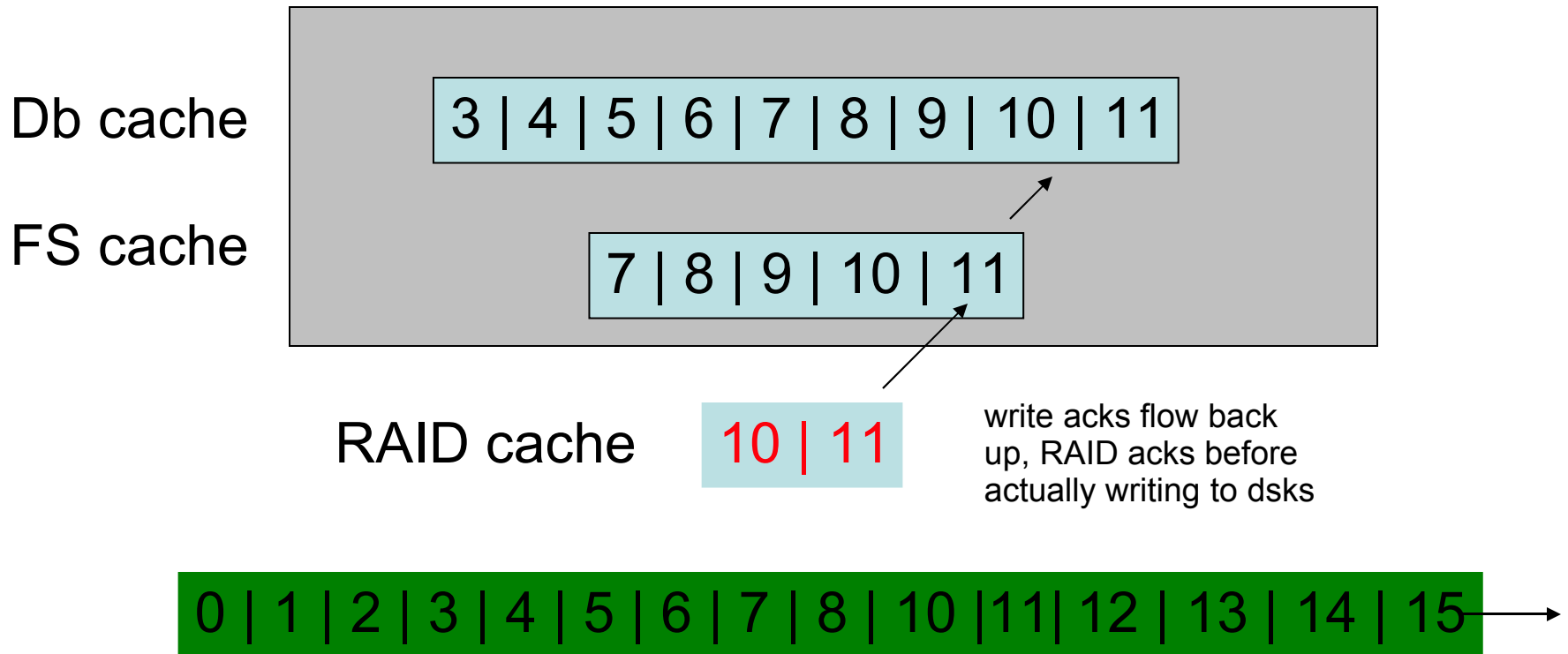
# Only cache writes in controller

suppose after reading 0 thru 11, the db modifies blocks 10 and 11. the blocks are written to the FS cache then to the raid cache. Until acknowledgments come back, they're "dirty pages" signified in red



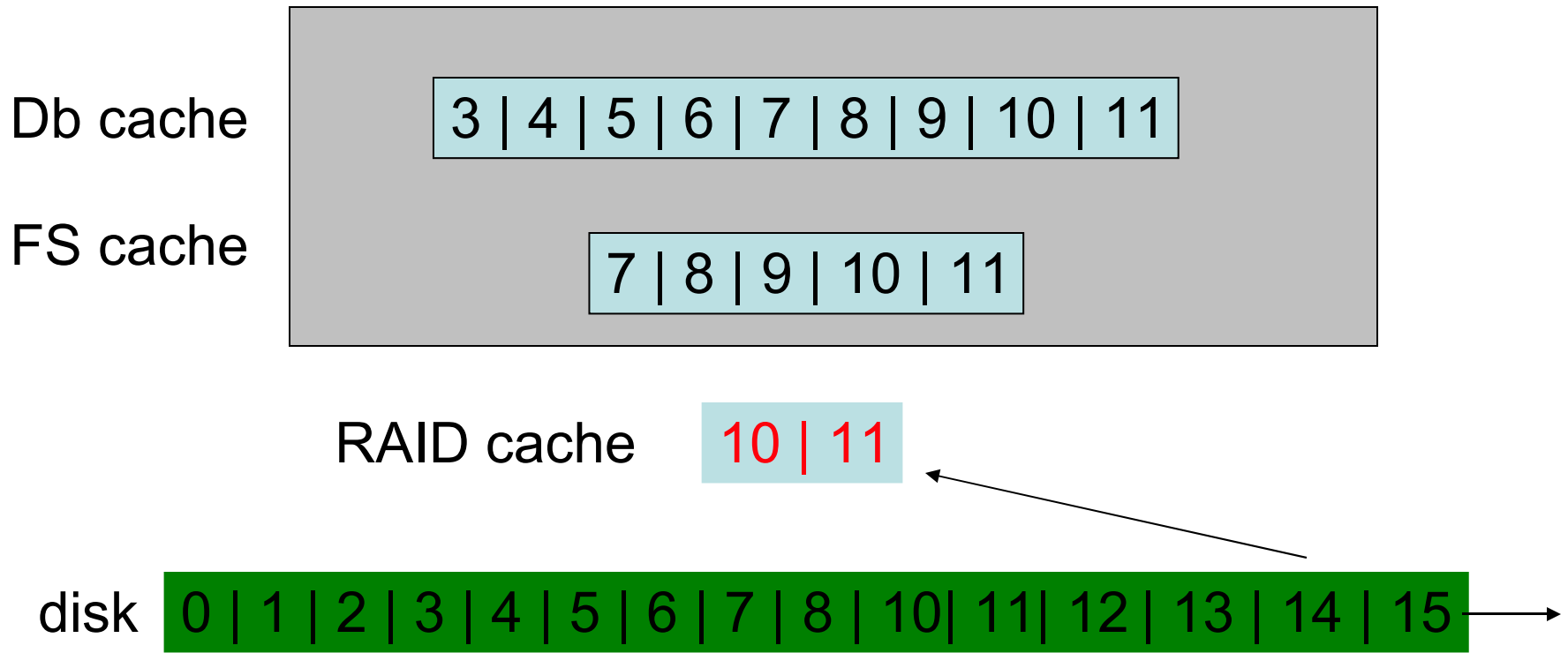
# Only cache writes in controller

When the raid puts the blocks in ram, it tells the FS that the block is written, likewise the fs tells the db it's written. Both layers then consider the page to be "clean" signified by turning them black here.



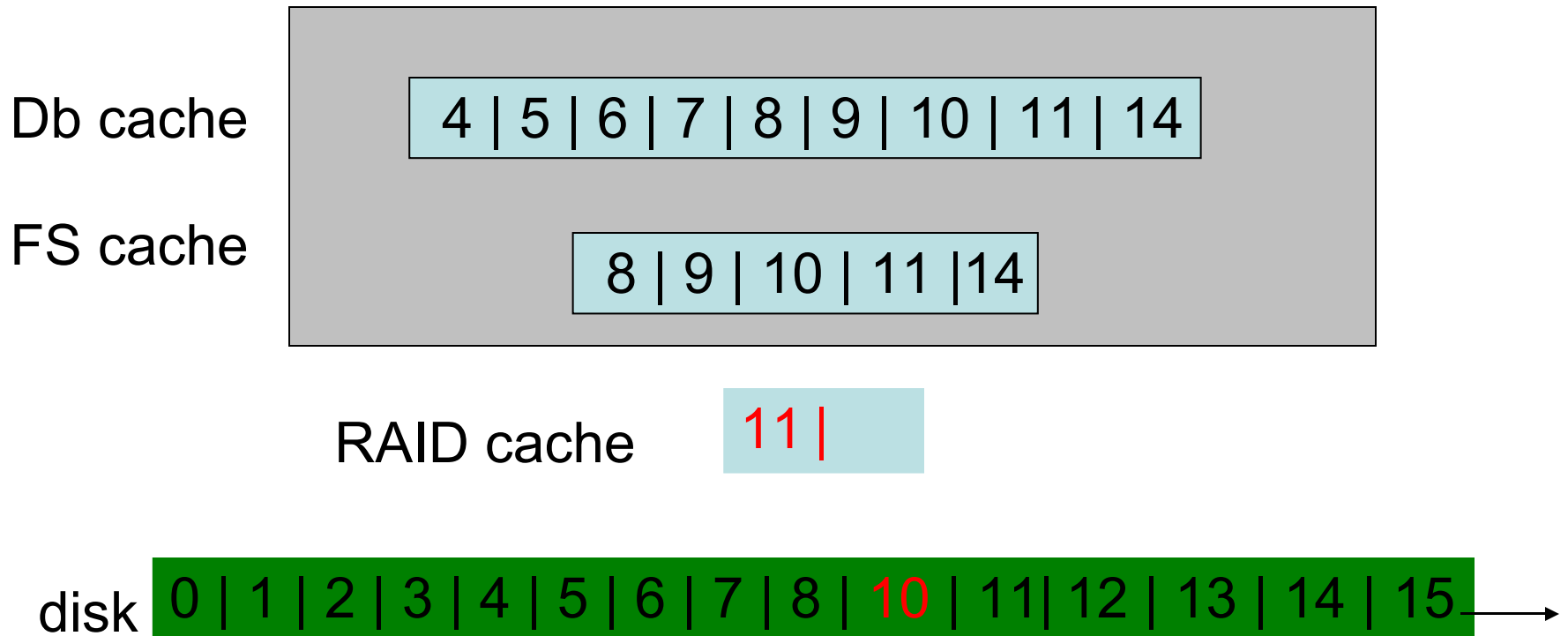
# Only cache writes in controller

The database can burst lots of writes to the raid cache before they ever get committed to disk, which is good: the raid offloads all the waiting and housekeeping of writing to persistent storage. Now Suppose a cache miss occurs requesting block 14. It can't be put into cache until block 10 is written, and the LRU is "pushed down"



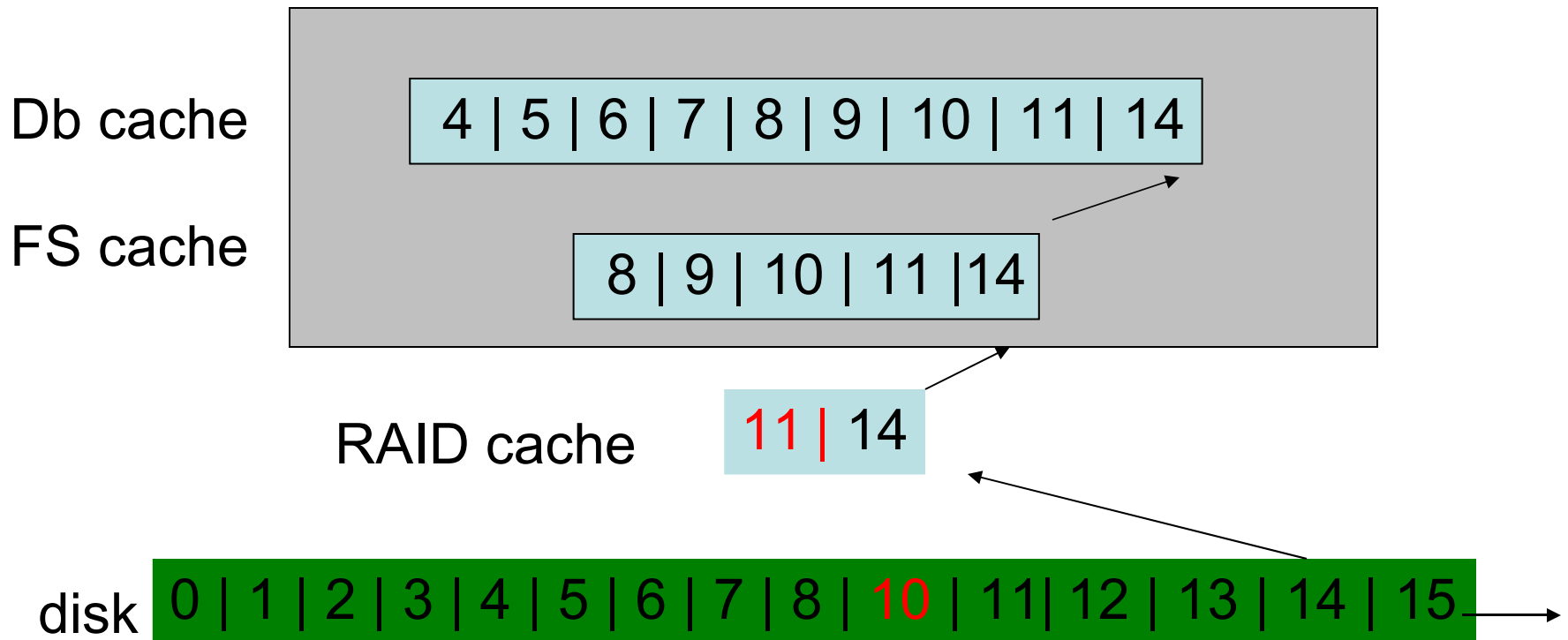
# Only cache writes in controller

The block 10 is retired at the cost of 1 IO before 14 is ever accessed.

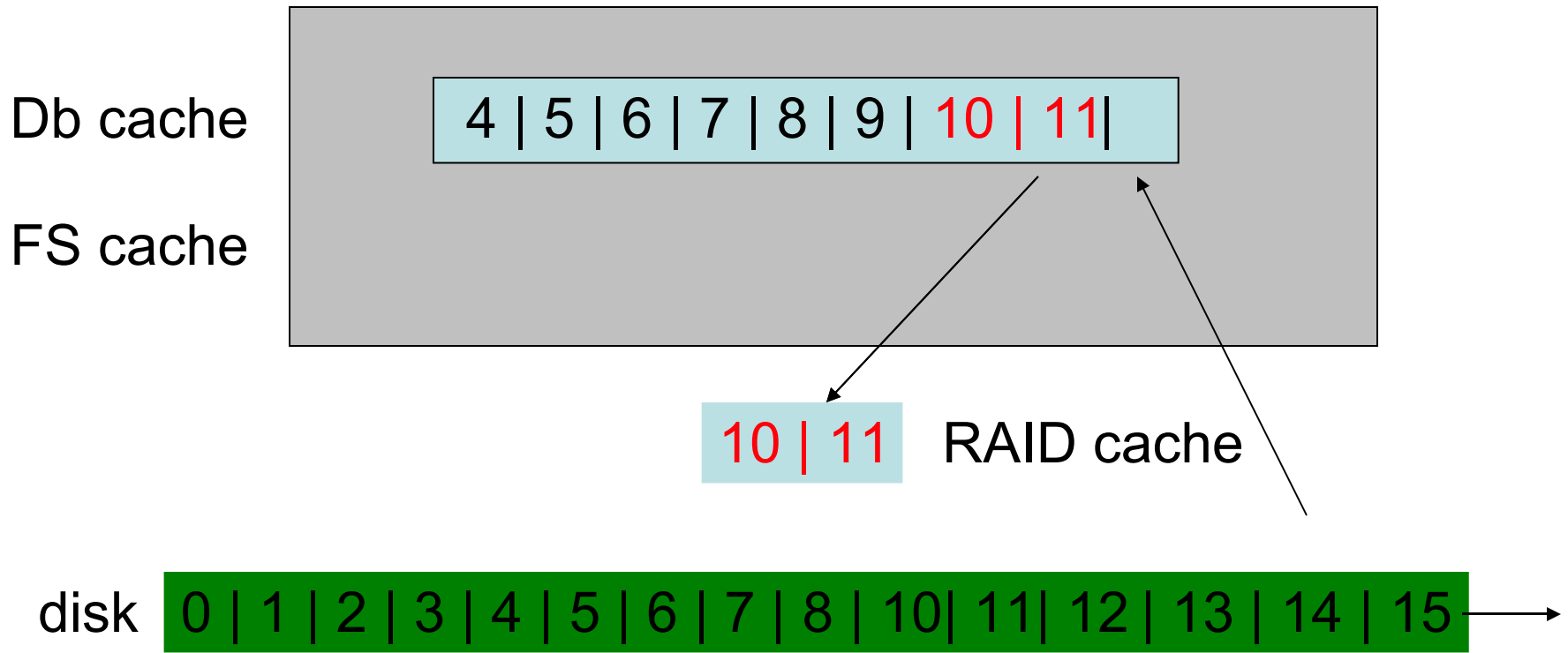


# Only cache writes in controller

Now, 14 can be put into the raid cache, a cost of 1 more IO, before sending on to the host



This is the ideal I seek : no filesystem cache at all, and only caching writes on the raid. In the same scenario, you can see how the read of block 2 will not serialize behind the writes of block 10.



# Only cache writes in controller

A real RAID cache is more complex than this example, sometimes involving high water marks or some scheme for splitting the cache between dirty and clean pages. But the principle is the same: after a burst of writes, the raid will finally decide that it shouldn't allow more writes or reads until it can evict some dirty pages. I like to think of this as “borrowing” from future bandwidth when doing a burst of writes.

Now, we've already established that the read cache is useless, because any cache hit will happen in the upper, much larger layers. If dirty pages are at the tail of an LRU cache, that means any reads will also have to wait for the dirty page to be written before proceeding (filling a cache buffer in the raid controller before passing up to the host) therefore, it's great to cache writes as long as you do not cache reads in the controller.

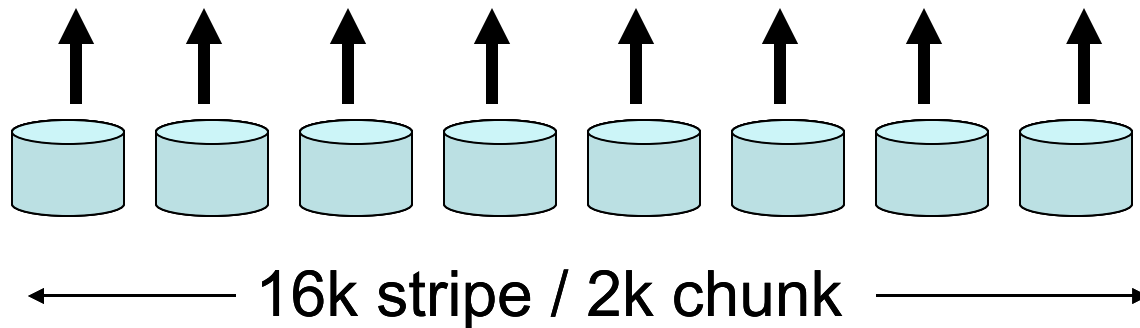
Some controllers (I think emc for example) may partition dirty and clean pages, and also partition by spindles, so that reads still don't serialize behind writes and no one spindle “borrows” too much bandwidth, leaving a cache full of dirty pages that must be retired to only one spindle for example. but again, we've established that raid read cache is useless in a nominal setup, cause it's always smaller than the host side cache, therefore may as well just give all the space to writes.

# Only cache writes in controller

- Burst writes “borrow” bandwidth by writing to raid cache
- at some point, new ios cannot continue, because controller must “pay back” IO freeing up cache slots.
- if reads don't go thru this cache they never serialize here. Some controllers partition, which is good enough.
- Various caching algorithms vary this somewhat, but in the end they at best are no better than not caching reads, at worst, hurt overall performance

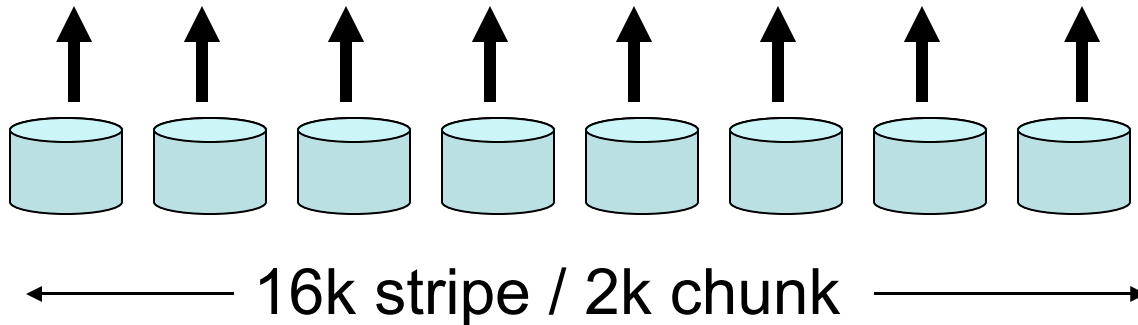
# Make stripe size >> blocksize

- suppose you read a 16k block from a striped (or raid10) with a small chunksize. You “engage” every spindle to service one IO.



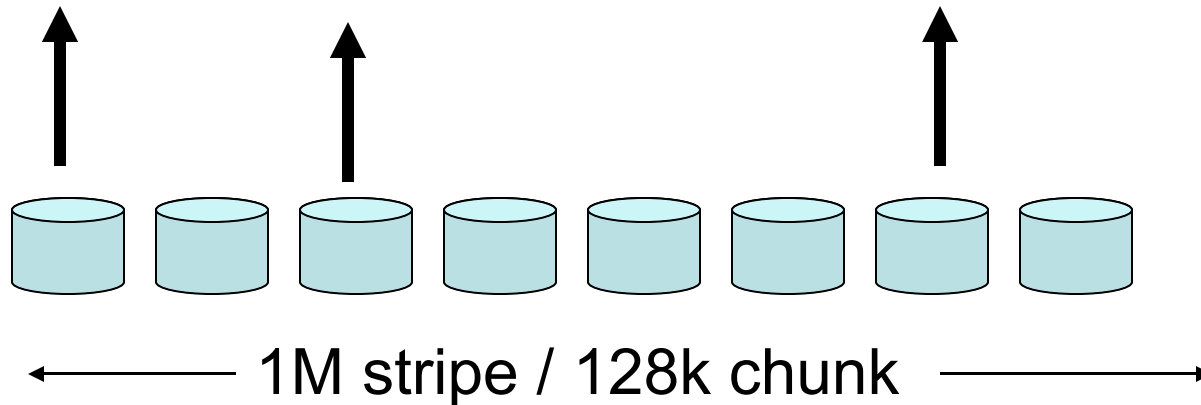
# Make stripe size >> blocksize

- now suppose 3 reads are issued by the host
- Seek /rotate on all drives (5 ms)
- Deliver 2k in parallel from all drives (<1ms)
- Seek /rotate on all drives (5 ms)
- Deliver 2k in parallel from all drives (<1ms)
- Seek /rotate on all drives (5 ms)
- Deliver 2k in parallel from all drives (<1ms)
- **THIS IS SERIALIZATION THAT SHOULD BE PARALLEL**



# Make stripe size $\gg$ blocksize

- issue 3 concurrent IO : Seek/rotate on 3 random drives drives (5 ms)
- Deliver 3x16k in parallel from 3 different drives (<1ms): each block “fits” on a single spindle.
- **Not that smaller chunk sizes also increase the likelihood of misaligned reads engaging 2 spindles instead of 1**
- A write of a given block, of course , engages 2x spindles in a mirrored config



# Generic rules for OLTP IO tuning, with “answers”

- **Never cache anything below the DB**
  - `innodb_flush_method=O_DIRECT` eliminates the FS cache.
  - If you must use or cant turn off fs cache, look into `vm.swappiness` parameter to prevent paging
  - I've seen serialization happen at this layer for reasons I dont understand, is suspect related to `O_DIRECT`. but worked around by mirroring in hardware and striping in md

# Generic rules for OLTP IO tuning, with “answers”

- Cache only writes in the raid controller, no readahead:
  - **adaptec (currently our controller for live dbs):**
    - `/usr/local/sbin/arcconf setcache 1 logicaldrive 1 roff`
    - Implements my ideal of caching writes but not reads, and not doing readahead

# Generic rules for OLTP IO tuning, with “answers”

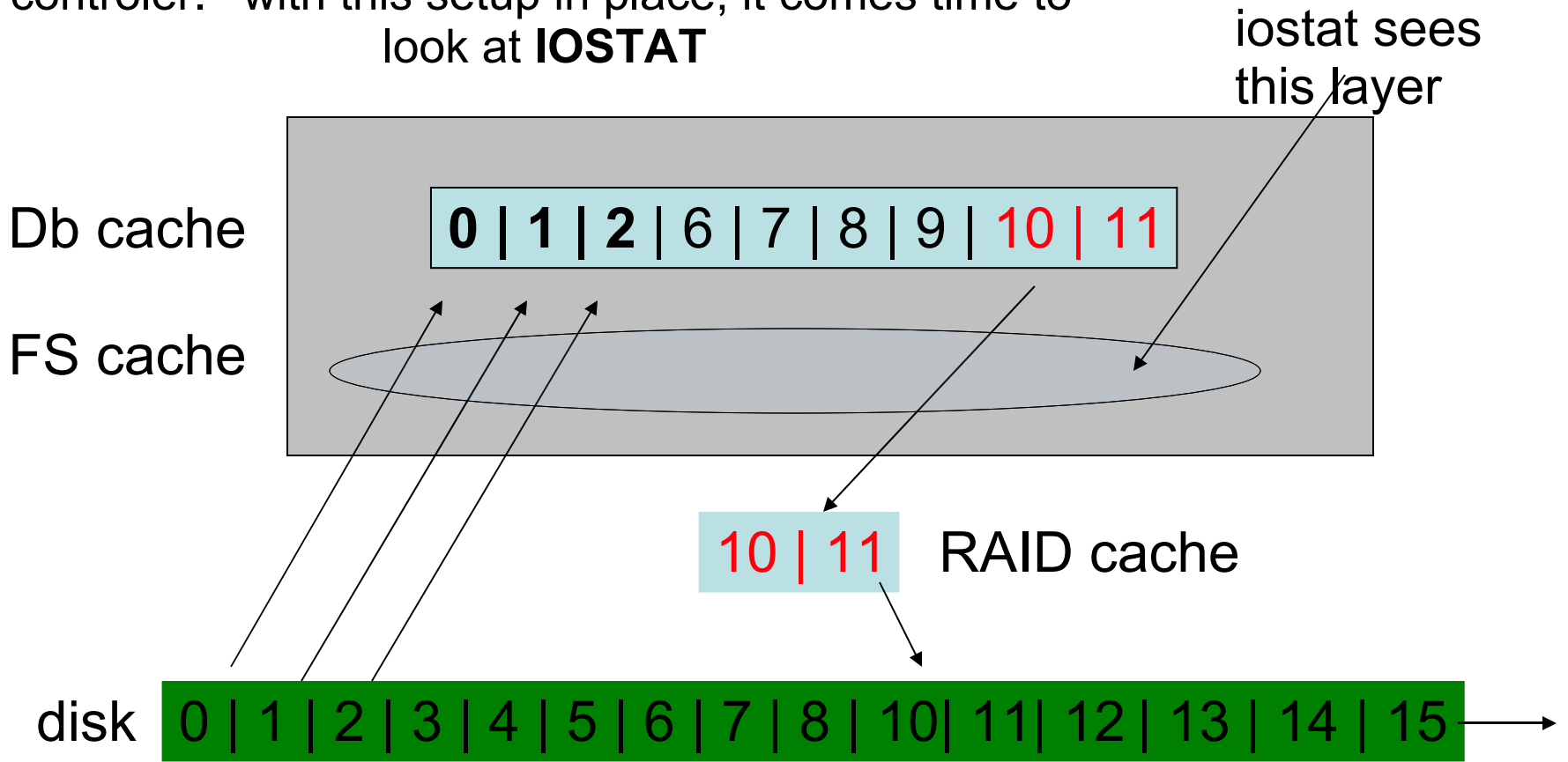
- Cache only writes in the raid controller, no readahead
  - MegRAID (our controller thru 2006):
    - write Policy = WRBACK
    - Read Policy = NORMAL
    - Cache Policy = DirectIO
  - These settings were what I had before we switched to adaptec. Never decided whether best to allow read/write cache w/o readahead, or turn both off with DirectIO

# Generic rules for OLTP IO tuning, with “answers”

- Cache only writes in the raid controller, no readahead
  - 3ware (I have a 9500S, never put a real workload on one)
    - no aparent ability to shut off readahead, not sure it's doing readahead
    - can shut of write caching, but not read (opposite of what I want)
    - my memory is vague on this controler, i used it for a fileservr a while back, but did token testing on it with database in mind

# My ideal

No FS cache at all, and only caching writes in the controller. with this setup in place, it comes time to look at **IOSTAT**

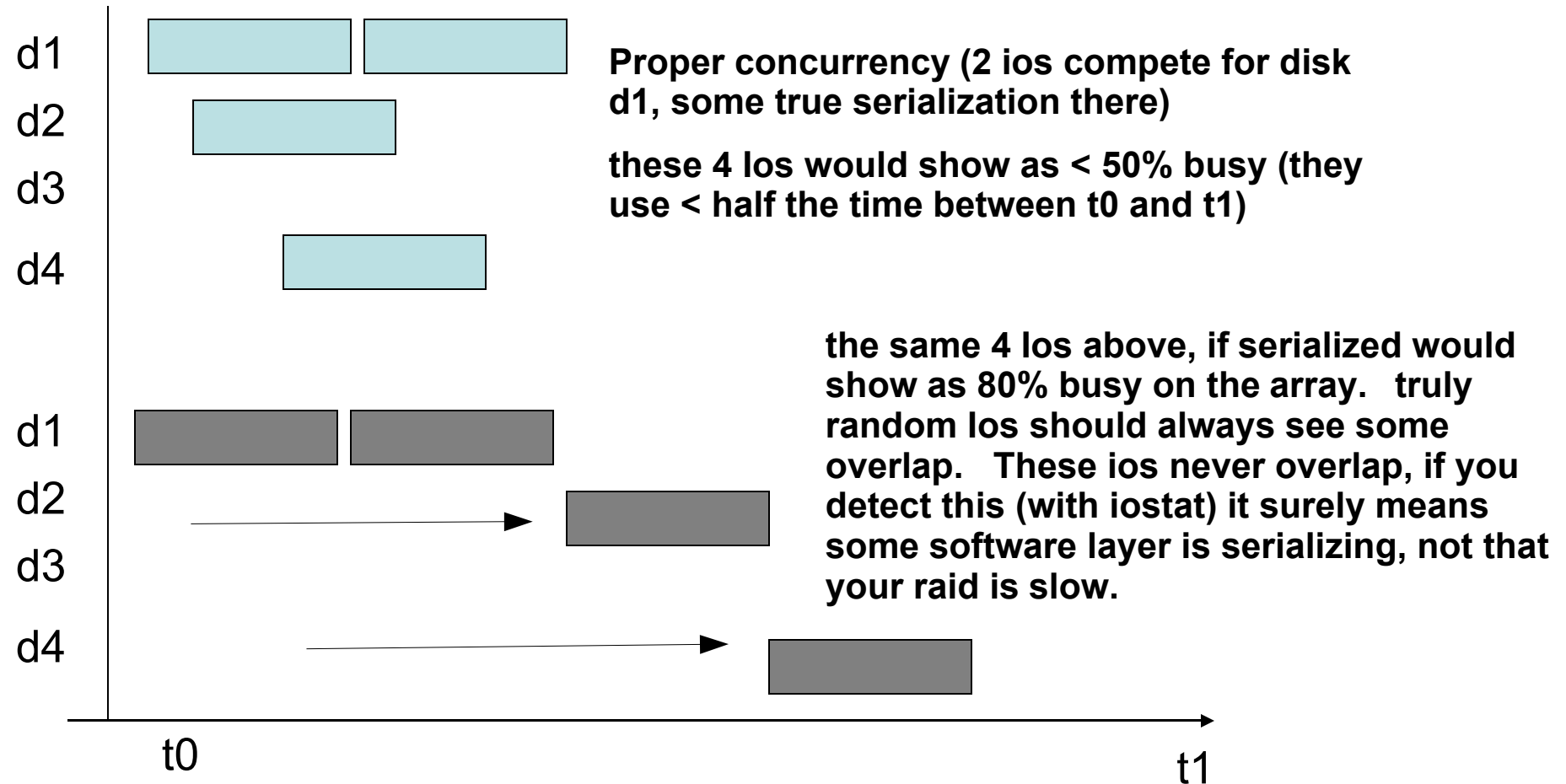


# Detecting serialization on multidisk arrays

- I've observed well tuned disk arrays serializing io for some reason, when they should be parallelizing (doing multiple ios to multiple disks concurrentl) and concluded it's somewhere in the filesystem/driver layers of linux
- diagnosis: If you're > 80% busy in iostat, and have a multidisk controller look **CLOSELY** at other numbers
  - Is reads/sec times avg wait < 1? Then you are **serializing IO** not getting full bandwidth of your spindles
  - If you're near 100% busy on a multi disk controler, you should be seeing the thruput of multiple disks.

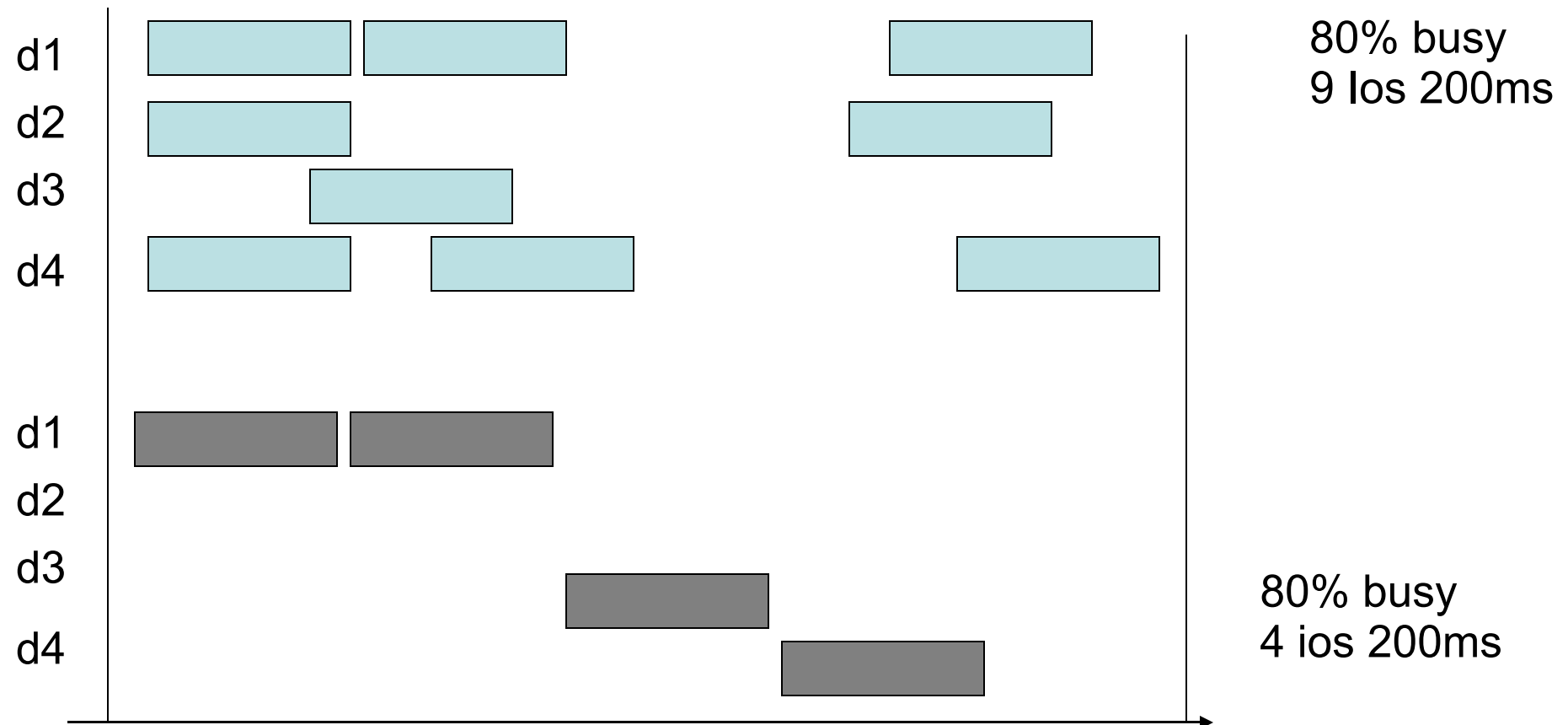
# Concurrent vs serialized

in this example, suppose youve taken 4 spindles in a disk array, and presented them as a single drive to linux by doing raid0. Linux “sees” one drive, /dev/sda but should still be willing to issue concurrent IO s . But is really issuing concurrent IO or is it serializing them and issuing 1 at a time?



# Concurrent vs serialized: both 80% busy

when you see a multidisk array at 80%, you have to make sure the IO/sec and response times crosscheck with the % busy figure.



# Formula for detecting serialization: **$(r/s + w/s) * svctm < 1$ with %busy near 100**

- remember that svctm is in milliseconds, so divide by 1000
  - ex 1 :  $(r/s + w/s) = 9$  so
    - $(9) * .200 = 1.8$
    - yes, concurrent ios on the array
  - ex 2:  $(r/s + w/s) = 4$  so
    - $(4) * .200 = .8$
    - NO, the array is not doing concurrent IO
  - BOTH AT 80% busy, but 1 is doing double the IO!

# Example iostat -xkd exhibiting serialization

This is real world iostat output on a system exhibiting some kind of “false” serialization in the filesystem/kernel layer. I reproduced it using rsync, so don't think this tells you anything about youtube scale

columns omitted for readability

Device:	...	r/s	w/s	...	avgqu-sz	await	svctm	%util
sda	...	0.00	11.91	...	0.37	31.13	1.09	1.30
sdb	...	2663.26	12.11	...	112.86	42.08	0.37	99.96

reads per second  
writes per second

time waited in  
kernel before  
issue to device

time actually taken by  
the device, after IO was  
finally issued to it

# Example iostat -xkd exhibiting serialization

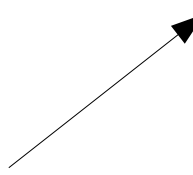
is  $(r/s+w/s) * svctm > 1$ ?  
 $(2663+12)*.00037 = .98$

therefore, this is serializing “above” the raid device. avgqu-sz further suggests it's serializing in kernel, not application (Innodb, or rsync in this case)

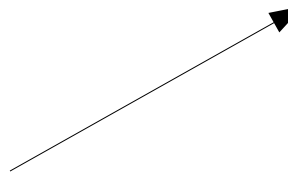
A large spread between await and svctm means things wait in queue in the kernel for a long time. this means higher layers are issuing things concurrently but the array cant soak them up fast enough to keep the queue low

Device:	...	r/s	w/s	...	avgqu-sz	await	svctm	%util
sda	...	0.00	11.91	...	0.37	31.13	1.09	1.30
sdb	...	2663.26	12.11	...	112.86	42.08	0.37	99.96

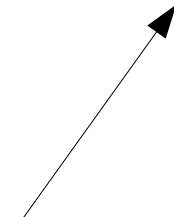
reads per second  
writes per second



If you see this much bigger than 32 sectors, which == 16k, the innodb blocksize. larger numbers point to readahead somewhere



Most disks should stay below 10ms. This output is weird because it's not from a db its from rsync



# Working around for serialization on multidisk array

I'm not sure why, but when I presented mirrored drives to linux, then striped in md (software raid) the serialization went away. I believe this is related to O\_SYNC, but maybe just the filesystem or scsi queuing side effects .

