

Writing Disk-IO efficient applications

Adrian Chadd <adrian@freebsd.org>

What will be covered?

- A quick overview of disk systems
- Various kinds of user-space disk IO paradigms
- Operating system differences
- Example: web serving / caching filesystem hackery
- Future stuff

What won't be covered

- Hardware in too much depth - its assumed you know stuff works
- Operating system stuff in too much depth - OS VM/Buffer cache stuff is interesting and would take days to talk about, not hours
- Expensive / new stuff: SSDs, for example, and large disk arrays - not anything I have access to!

Disk Basics

- The standard stuff if you're not a flash drive:
 - Disks rotate
 - They generally have one set of moving heads
 - Modern disks may or may not have some kind of tagged queue operation
 - Modern disks generally use “sector sliding”

Disk Basics

- What if you're a flash drive?
- For “camera-grade” flash:
 - Very fast read rate
 - Somewhat fast write-if-erased rate(?)
 - Very slow erase rate (almost “sync”)
- SSD's try to compensate through intelligent controllers, but I don't have stuff yet to stress test and understand..

Controller Basics

- Each controller can speak from 1 to \$ {LOTS} of disks
- Most modern controllers work with disks to implement tagged queues
- Some storage systems have multi-level controllers (eg { Disks } -> RAID controller -> one LUN -> PC controller -> PC)
 - This may impact on performance..

Kernel Basics

- Current kernels pretend a “disk” has some abstracted set of operations and behaviour
- “low level”: {read,write,seek,sync,ioctl}
- Disk operations are submitted into a work queue which is sorted and fed into the driver
- GEOM: Or into another processing layer, which feeds it into another layer, etc..

Old Filesystem Basics

- Generally sits on top of the kernel “buffer” or “VM” layer (depending upon vintage)
- Reads generally happen in process context
- Writes generally happen via buffer layer and are tried to be done in the syncer
 - .. as we’ll see, this isn’t always the case!
 - .. and what about updating atime, etc?

New Filesystem Basics

- Current Filesystems:
 - Don't necessarily do all of their write IO via the syncer
 - Don't necessarily delay their writes until file close
 - .. so is close() potentially blocking?
 - .. so is write() potentially blocking?

Userland Basics

- Traditional interface - synchronous operations
- May or may not involve read and/or write buffering
- More complicated stuff is available for trying to handle large files, or async operations, or guaranteed IO, or ordered IO, or ..

Complicated Stuff

- How to implement scalable async IO?
- How to implement small/large file IO?
- How to work with the filesystem?
- How to minimise disk operations?
- How to maximise cachability?
- How to handle peak periods?

Sync IO

- POSIX read(); write(); etc
- PROS:
 - Simple, understandable
 - Syscall return == valid and private copy of data
- CONS:
 - Does not scale with massively parallel applications (w/out threading..)

Sync IO + threads

- So, lets create worker threads to do the work; they can call POSIX sync IO calls
- PROS:
 - It actually works very well
 - Is quite portable (see Squid)
- CONS:
 - It doesn't map 1:1 to the semantics of sync IO

Sync IO + Threads

- `aio_open("path", O_MODE, callback, cbdata)`
... processing happens ...
`callback(cbdata, status, error, new_fd)`
- `aio_write(fd, buffer, len, callback, cbdata)`
... processing happens ...
`callback(cbdata, status, error, new_fd)`

POSIX AIO

- `aio_read()` / `aio_write()` / etc
- Schedule disk operations to occur some time in the future
- Calling process is free to do whatever it likes; just needs to periodically check the completion of POSIX operations
 - `aio_error()` / `aio_return()`

POSIX AIO

- PROS:
 - Generally works very well where there's native support
- CONS:
 - The specification did -not- cover an efficient way to check/reap pending events; scales poorly on “default” implementations
 - Not 1:1 semantics with sync IO

MMAP IO

- `mmap()` files to read and write
- Avoids a “copy” for data coming in and out of kernel-space
- Allows for sharing of memory between processes - less memory footprint
- VM's generally “cache” more data than buffer caches
- .. but, not everything can be cached

MMAP IO

- So what's "cached" in the VM?
 - Object data - yes
 - Metadata? Not always - FreeBSD VMIO hackery, for example
- So very busy filesystems may still see buffer cache churn from metadata updates
- ... but using mmap lets you use VM for data and buffer cache for metadata

MMAP IO

- Hinting to the kernel? `madvise()`
 - `SEQUENTIAL` - sequential access
 - `RANDOM` - random access
 - `WILLNEED` - will need this soon!
 - `DONTNEED` - won't need this again soon?
 - `FREE` - free immediately

MMAP IO

- Syncing to the file: `msync()`
 - sync a memory region
 - `ASYNC` - sync in background
 - `SYNC` - sync in foreground
 - `INVALIDATE` - invalidate all cached data

MMAP IO

- General process is:
ptr = mmap(NULL, len, prot, flags, fd, offset)
madvise(ptr + offset, len, behaviour)
<fiddle with data at ptr as needed>
madvise(if required)
msync(ptr + offset, len, flags)
munmap(ptr, len)
- Generally - do mmap's of regions as required, do as much data fiddling as possible before msync/munmap

MMAP IO

- First major limitation: VM region size
- 32-bit OS: mmap()'ing regions are limited to KVM space, or even less under Linux (~ 512m contiguous? 1GB?)
- ... so you need to map in and out regions of the file for large files or direct IO
- 64-bit OS: mmap()'ing regions much more sensible, do that!

MMAP IO

- Second limitation: IO scheduling
- How do you convince the OS layer to do IO in larger chunks?
 - For buffer cache: just do larger IO, up to MAXPHYS or your filesystem block size?
 - For VM cache: you need to dirty contiguous pages and hope the syncer gets to them in time to coalesce them!

External process IO

- Use some kind of IPC to push disk IO into some external process
- Useful for simple stuff like logfile reading/writing
- Squid “diskd” uses this for general storage disk IO, rather than logfile writing

External process IO

- Example: Squid Logfile Writing
 - Using stdio for logfile writing
 - Assumption: writing is non-blocking as long as the buffer cache isn't blown
 - Assumption: writing happens -only- in syncer process
 - Reality: Hahahahahaha!

External process IO

- Squid, ctd
 - In reality: about 300-400 logfile lines a second before the main process blocked on UFS IO locks
 - .. way, way before the buffer cache was full.
 - .. way, way before the syncer got to run, and the syncer didn't fix things
 - So why? I'm not sure! I had to fix it!

External process IO

- The solution - simple - write logs to a pipe, in 64k chunks, and the process at the other end does the sync writes
- Falling behind in disk IO == filled pipe, and can be handed non-blocking in the main process (ie, drop new log entries on full queue.)
- Result: ~ 50,000 HTTP requests/second logging on a single disk - ~ 8 mbyte/sec

sendfile()

- Bypass the trip through userspace (mostly) entirely - just glue a VM object to a socket filedescriptor
- Works great for workloads where the content doesn't require transformation - so, say, HTTP static content services
- Works poorly for dynamic workloads - the data needs to be manipulated first in userland

sendfile()

- Isn't implemented the same on all platforms!
- Some let you insert arbitrary data before/after the sendfile, some don't
- Some let you glue non-VM FDs to socket FDs, some don't
- All platforms implement sendfile as a blocking syscall
 - So parallelism == threads or processes

sendfile()

- Is it worth it?
 - Yes - don't have to do the copyin/copyout (mmap() saves copyin btw)
 - Yes - when done right, you avoid needing temporary memory to copy data to; pages -should- be just glued into socket mbuf's
 - .. hm, is that done on all platforms? I Should check!

Operating System Diffs

- POSIX AIO:
 - FreeBSD has great native support, scales great across ~ 20 disks, thousands of ops/sec, `aio_waitcomplete()` / `kqueue()`
 - Linux: implemented in userspace; one pending operation per fd at a time
 - Solaris: also has great support, but no way to scale pending ops until event ports were available in Solaris 10

Operating System Diffs

- `sendfile()`
 - Not implemented the same on all platforms!
 - Not available (at least the last time I checked) under Solaris!
 - Issues with changing underlying data, will cover it soon

Operating System Diffs

- Raw device cachability:
 - FreeBSD - raw disk devices, uncached
 - (But: GEOM g_cache?)
 - Linux - raw disk devices, buffer cached by default
 - `O_DIRECT` or something to disable
- Very important to remember when implementing raw data stores!

Operating System Diffs

- `mmap()` ! (Ask Matt Dillon about this!)
 - Best example is the recent thread on FreeBSD-stable re: `mmap` and `rrdtool`
 - Linux: handles `mmap/madvise/msync/munmap()` per `rrd` update file
 - FreeBSD: doesn't handle it so well
 - Linux: `msync()` of the whole file works fine; FreeBSD: doesn't handle it so well

Sync vs Async IO

- POSIX sync IO is simple to understand
 - `read()` data gives you a private copy on return
 - `write()` data is assumed to have been written when `write()` returns
- Aborting the operation is simple - just don't do it in the first place! (Heh.)
- But async.. hm!

Sync vs Async IO

- Async IO - you “submit” some “thing” to happen, and it’ll happen sometime in the future
- .. when you’re then told about it
- You generally have to keep some “stuff” static between operation submission and completion
 - read buffers have to stay allocated
 - write buffers have to stay constant

Sync vs Async IO

- First gotcha: cancellation
 - You can't cancel IO events
 - Sync events - you can schedule them, then delete them before the syscall
 - Async events - will be scheduled, and will run at some point in the future
 - So abort becomes “try to cancel, wait for completion” - the userland needs to cope!

Sync vs Async IO

- Second gotcha: race conditions
 - For example - read/write race conditions between processes
 - Process A reads 4k, writes it to the network
 - Process B writes 4k of new data into where process A read for
 - Whats the behaviour?

Sync vs Async IO

- The race condition exists only during the sync read
- ie, once the data from process A has been read, its private copy can be sent to the network at leisure
- .. what about for async IO?
 - The race occurs right until the socket layer TX'es the data
 - ie, process B will modify the in-use page!

Sync vs Async IO

- Most commonly seen with `mmap()` and `sendfile()`
- The OS doesn't define any sort of "transactional" semantics here!
- Its all up to the userland processes/threads to sort out themselves
- Its less encountered with sync IO in threads/processes, so programmers may not be aware of the issue until they "async" stuff!

Filesystem?

- Learn how the filesystem lays out stuff
- Learn how the filesystem implements various file operations
 - Does your `open()` or `read()` syscall translate to underlying device writes?
- What about your filesystem block size? Fragment size? Whats the worst case scenario (eg, lots and lots of fragmentation, packed fragments, etc)

Filesystem?

- One popular question: one big filesystem, or lots of little filesystems?
- Big filesystem: easier to code for, but what about parallelism?
- Little filesystems: difficult to code for, but explicit parallelism
- Question is - what parallelism is needed for your application?
 - ..and are you lucky? Does your FS know?

Filesystem?

- UFS(1, 2) ; EXT(2, 3) - just “see” a block device
- Mirrored disks: writes remain the same speed, but reads can be load balanced across the disks in the set
- Some people with high-read loads will just add extra disks to the mirror set to increase read throughput

Filesystem?

- Striped volumes: different areas of the disk can be accessed in parallel
- But the user has no idea what those boundaries are!
- So hm, RAID0? RAID1? RAID5? Or lots of little filesystems? Or combinations of both?
- .. the answer is .. well, painful.

Filesystem?

- All things equal, you should be able to just look at your workload and figure it out
- However, there are confounds..
 - RAID controllers may implement great high-throughput RAID5, but crappy individual RAID0 disk IO
 - .. what if you run out of LUNs? RAID5/ RAID0 let you reduce the LUN count
 - What about intelligent shelves?

Filesystem?

- .. heck, what about mostly-dumb shelves which lock up with too many pending disk ops?
 - So choosing the most optimal value may end up causing instability!
- In practice, the same stuff applies: model, benchmark, test.

Filesystem?

- How would you use multiple filesystems?
 - Well, the more useful question is “what the hell are you putting on there, and what does the change/access pattern look like?”
- This is where one large filesystem may provide a “good enough” service..
- An example relating to web caches follows.

Filesystem?

- “Object locality”
- ie, objects on the same page are generally downloaded at about the same time
- Would you put them across different disks, or would you put them on the same disk and read them in at once?
- (hint: this is a mostly dissected, understood and solved problem.)

Filesystem?

- Squid: chooses a filesystem for an object based on either “round robin” or “least-load”
- Not optimal - objects for a given page end up across multiple disks
- Remember - doing larger IO reduces op/sec throughput **much slower** than the increase in transfer rate
- So: packing objects together == win

Filesystem?

- So, your choice of where you place objects will influence how you read them back
- .. and thus, influence how well things scale.

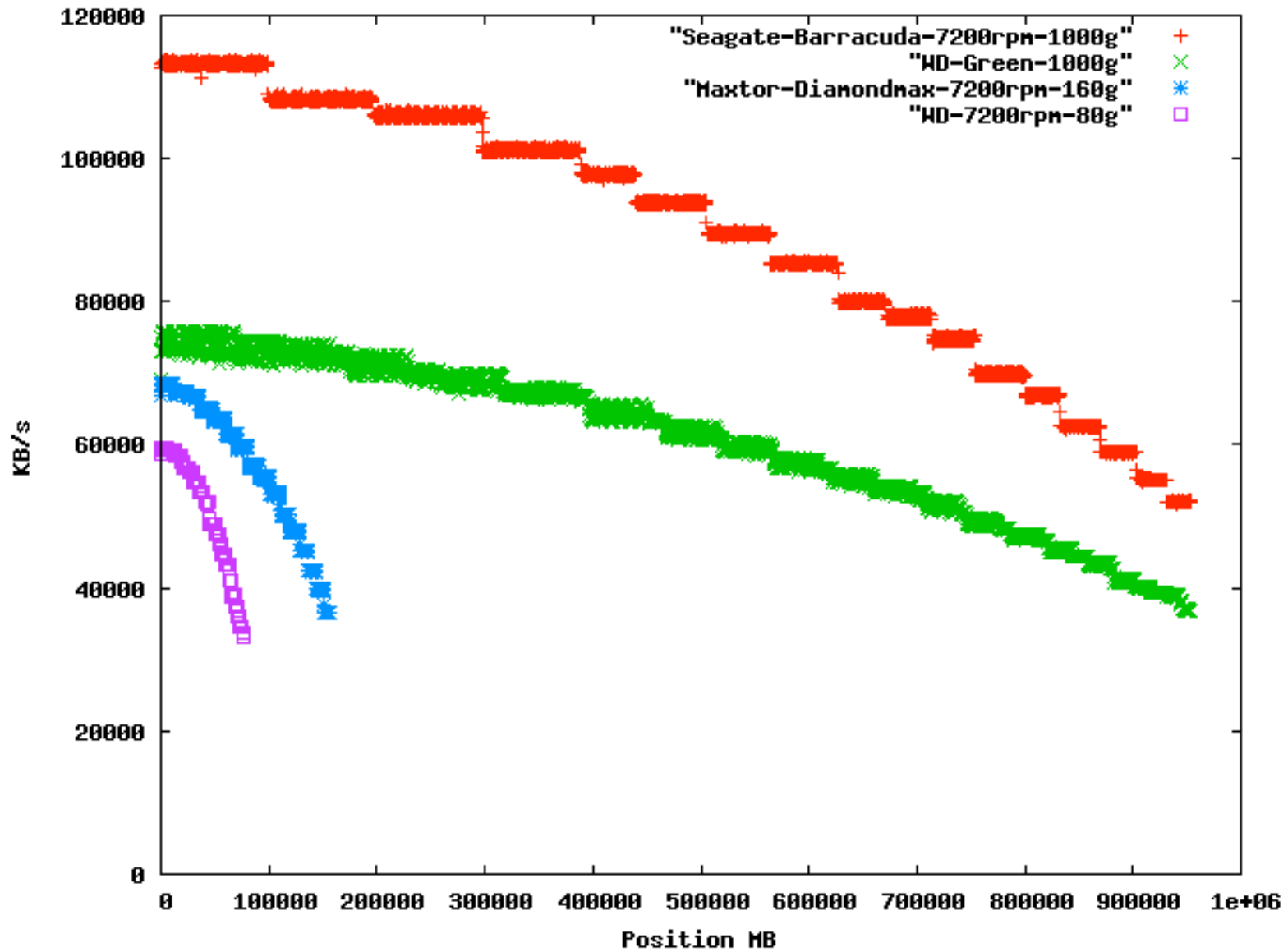
Filesystem?

- One final point: disk cluster/fragment size
- Again, relatively well-understood, won't get into it here in too much depth
- Gather stats on your file sizes, pick a filesystem setup which somewhat matches it
- (Lots available about this, go hit up your favourite search engine)

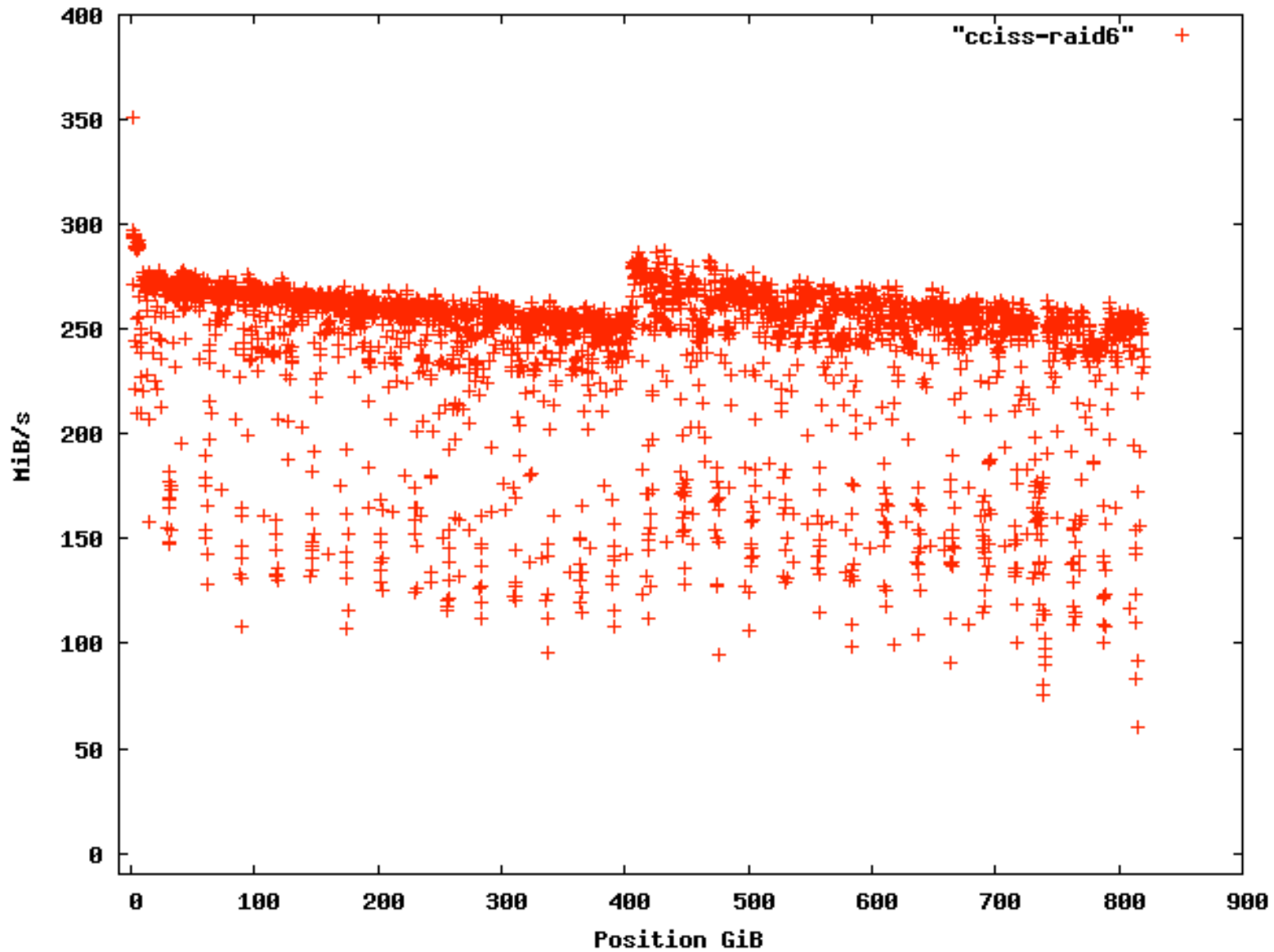
Disk Operations?

- Need to maximise “bang for buck”
- Standard, 7200rpm, SATA/SCSI: random IO?
~ 150 - 250 req/sec
- Why the variance?
 - Disk layout - some parts of disk transfer data faster than others
 - More information: benchmark your disks
 - eg “ZCAV” by the Bonnie++ author

Disk Operations?



Disk Operations?



Disk Operations?

- What about non-random type IO?
- Ie, what about random IO with increasingly larger buffer sizes, larger than a single block)
- Simple hacky benchmark - `src/tools/aio/aiop/`
`in -current`

Disk Operations?

HP 36.4G MAN3367MC 10000rpm U160 SCSI
16384 read ops; 256 concurrency

Op Size	TPS	KByte/Sec
512b	580	300kbyte/sec
1024b	579	592kbyte/sec
2048b	569	1167kbyte/sec
4096b	522	2139kbyte/sec
8192b	469	3849kbyte/sec
16384b	395	6475kbyte/sec
32767b	309	10146kbyte/sec
65536b	236	15467kbyte/sec

Disk Operations?

WDC WD2500JS-00MHB0 7200rpm SATA I 50
16384 read ops; 256 concurrency

Op Size	TPS	KByte/Sec
512b	120	61kbytes/sec
1024b	119	122kbytes/sec
2048b	118	243kbytes/sec
4096b	118	483kbytes/sec
8192b	117	961kbytes/sec
16384b	115	1890kbytes/sec
32767b	111	3637kbytes/sec
65536b	103	7274kbytes/sec

Cachability?

- Try to maximise the effectiveness of the VM/
buffer cache
- `mmap()` would be a logical choice, if not for
the strange operating system related issues,
and reliance on the syncer
- .. but if your load is almost all-reads, `mmap()`
may be your friend

Cachability?

- What do the various bits cache?
 - VM: mmap() style access
 - FreeBSD (at least): almost everything happens through the VM layer
 - Buffer cache: read/write
 - metadata, POSIX IO read/write

Cachability?

- VM: works on PAGE_SIZE chunks, not smaller or larger
- Buffer cache: in theory, works on variable sized objects
 - So useful for small objects and metadata
- When is which good?

Cachability?

- Buffer cache
 - Generally limited in size, not as efficient as just straight VM page wiring
 - But handles smaller objects fine, so you can throw tiny objects through the buffer cache
 - Larger objects - well, it works..

Cachability?

- VM
 - Larger pages - PAGE_SIZE - so larger reads/writes
 - Putting small objects into the VM is wasteful!
 - .. and your filesystem probably won't pack small objects together

Cachability?

- So a few tips..
 - .. pack your objects in memory before writing them
 - .. read them back in larger chunks, rather than little ones
 - keep in mind your replacement policy (generally LRU unless you're running ZFS) - don't do linear file walks, for example!

Handling peak periods

- Peak, burst workloads can do a few things
 - temporary increase in write workload, affects read workload
 - temporary increase in working set size, blowing away caching
- The general result: IO queues increase; your service times increase; everything snowballs to hell

Handling peak periods

- A common trick is “tiered” storage, this is an old trick
- You could view a VM system treating memory as “disk cache”, and everything eventually living on backend storage
- So you could use smaller amounts of high throughput disk/SSD, then larger amounts of medium-throughput disk

Handling peak periods

- What can the userland application developer do when things go pear-shaped?
- Measure general service times - note when things go “out of profile”, and stop handling requests
- This is great if you have a load balancer to throw other requests to when you’re screwed..

Cachability?

- In general:
 - Remember how your VM/buffer cache works - write test cases!
 - Don't do dumb crap like create lots of small files; pack them where appropriate (will discuss this later)
 - If using `mmap()` - use `madvise()` where appropriate (but test!)
 - 64-bit machines!

Case study: Squid

- Squid (and Harvest before it) started with straight sync IO
 - ie, disk IO was done in batches using read () and write()
- The main program didn't proceed whilst disk IO was being performed
- Interestingly, network IO could still occur, so existing socket buffers could fill/drain as appropriate..

Case study: Squid

- Even with interrupt network processing, throughput suffered
- ~ 80 or so disk requests/sec on P-II class hardware
- Disk IO was moved into threads, “asyncops”
- “asyncops” provided a callback-driven API for standard disk ops -
{open,close,unlink,read,write}

Case study: Squid

- Most of the **development** occurred on Solaris 2.5 - 2.7
- Most of the **deployment** occurred on Linux-2.2
- Solaris-2: with UFS, write() and close() could be sync
- Linux (with default ext2 mount and 2.2 VM): write() almost ever blocked, neither did close()

Case study: Squid

- The default async operations were thus {open,read,unlink}
- {write,close} were kept sync
- Worked great under Linux-2.2, provided slower disk throughput under FreeBSD and Solaris

Case study: Squid

- Why was the disk throughput slower under load?
 - Solaris? No idea, wasn't involved..
 - FreeBSD: soft-updates: sometimes, the buffer cache filled up, and the squid process blocked waiting for disk buffers to become free
 - remember, that whole “syncer” thing doing bits of data and metadata IO..

Case study: Squid

- Async disk IO:
 - Generally gives ~ 150 disk ops a sec per disk - faster on very recent 15k SAS..
 - Generally translates to ~ 80 req/sec per disk
 - Why? Filesystem overhead
 - One object per filesystem file entry limits throughput quite substantially

Case study: Squid

- Tuning filesystems!
 - HTTP objects on older forward proxies: mean object size ~ 15k, SD ~ 3k
 - (figures - from memory)
 - (yes, before .mp3 and .flv)
 - How does this map to actual on-disk operations?

Case study: Squid

- Well, you have to, at a minimum:
 - seek() and read() the directory entries - three level dir hash, so say 3 x (1 seek to the directory inode, then 1 seek to the directory data)
 - Then another seek to the file inode
 - Then another seek for the file contents
 - times however many non-contig disk blocks there are

Case study: Squid

- Remember, 7200rpm disks do ~ 150 - ~ 200 random IO ops a second, so 6 seeks to different parts of disk $\Rightarrow 25/33$ ops/sec
- Caching directories, inodes, and file/inode bitmaps helps significantly
 - There's no way to lock these into RAM
 - FreeBSD: pre-VMIO metadata caching?
Busy caches evicted metadata from buffer cache..

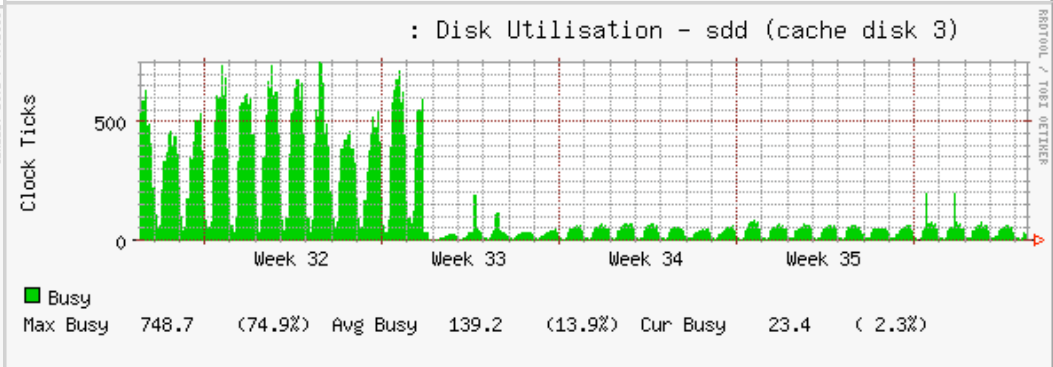
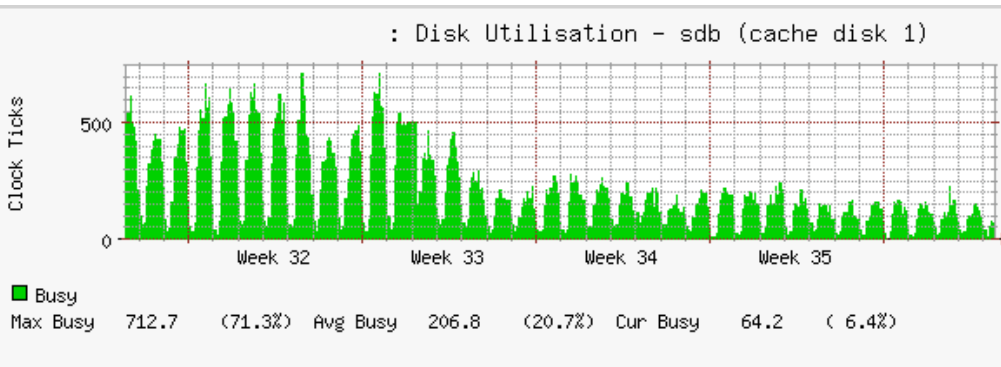
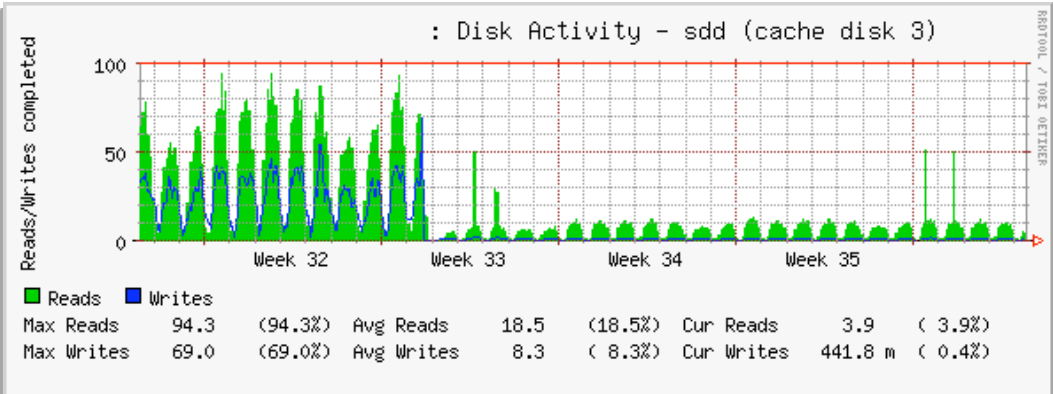
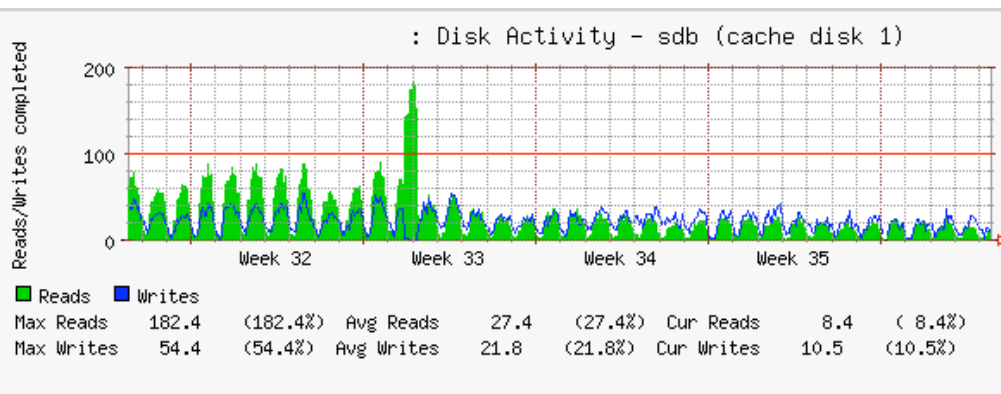
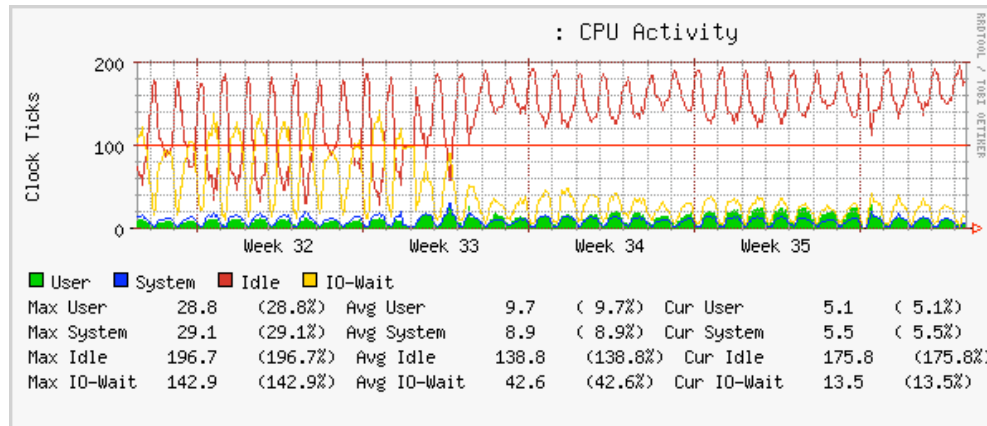
Case study: Squid

- Replacement Filesystem: COSS
 - Cyclic Object Squid Storage
- “On-disk LRU”
- Log Structured Filesystem
- Hot objects are rewritten to a later part of the disk
- Recycling occurs when writing out objects

Squid: COSS

- Objects are read in their entirety
- Objects are deleted and written in large chunks (typically megabyte sizes, perhaps larger)
 - Objects “underneath” the area being written to are deleted
- Minimises disk IO operations

Squid: COSS



Squid: COSS

- COSS (ctd)
 - No object locality - still one seek + read per object, even if they're groupable
 - Squid doesn't cache disk data (leaves it to the buffer cache), so COSS on raw FreeBSD disk doesn't involve memory caching (gah!)
 - Should write very frequent objects to the faster part of disk? (Bluecoat?)

Case study: Varnish

- Varnish, on the other hand, uses worker threads which implement sync operations
 - Extra worker threads are created to handle incoming connections if the current thread pool is all busy
- Works great in some situations, works poorly in other situations
- The trick is understanding the **why**..
- (and no, I'm not bashing Varnish. Honest.)

Case study: Varnish

- Varnish Storage Overview (correct at time of investigation - late Varnish 1.x)
- Kept an in-memory cache..
 - .. that was then disk backed either via `mmap()` to a file, or `malloc()` for anonymous backing (ie, “swap”)
- No particular object layout

Case study: Varnish

- Data is written to the network either via `writenv()` or `sendfile()`
- All operations are blocking - new worker threads spawned as required
 - Disk, Network, etc.

Case study: Varnish

- Works great when the majority of the workload is in RAM and generally static
 - (ie, the standard “reverse proxy”)
- Works poorly with larger objects or very small objects; works poorly with a workload is not in RAM
- This difference in workload generally dictates whether Squid or Varnish is used.

Case study: Varnish

- Varnish trusts the VM is doing the “right” job when scheduling disk IO
- But the VM has absolutely no clue about the application IO patterns, and fails miserably
- It doesn't know to toss pages in a certain way; it doesn't know to fetch data back from disk in a certain way, it doesn't know how objects on-disk are related
- Varnish -could- organise data to try and hint!

Further work

- Disk operation info: dtrace?; geom_nop
 - geom_nop needs some improved logging! (Will look into it..)
 - say, 100,000 ops/sec, 4 byte op|flags|result, 8 byte size, 8 byte offset - 20 bytes
 - --> 2 megabytes/sec of logging data
 - (Ie, perfectly easy.)

Further work

- People seem to hate the idea of custom filesystems
- (Unless you're a commercial product, for some reason)
- Perhaps some workload-specific raw filesystem stuff for evaluation/testing?
 - ..and it'd be nice to have a simple API
 - Like, I dunno, what Google do..? :P

Stuff To Look At

- Examples of different kinds of efficient file IO - obvious slant towards web services
 - Squid - async helper thread IO
 - Varnish - mmap
 - Lighttpd - mmap, sendfile, read/write, POSIX AIO in later versions?

Stuff To Look At

- “Object Locality”
 - basically, any paper between 1998 and 2003 regarding Web Cache Filesystems
 - Still very very applicable for current static file web serving loads

Stuff To Look At

- “Disk Behaviour”
 - Basically, check the last 10 years of IEEE
 - Will publish links to papers (when I find them) on automatically detecting and determining the disk behaviour, and adapting behaviour to suit
 - mmap()
 - For what **not** to do - the FreeBSD-stable thread on rrdtool tuning

Questions?

Thanks!

Adrian Chadd <adrian@freebsd.org>