Key ingredients to achieve effective I/O

Sébastien Ponce
sebastien.ponce@cern.ch

CERN

Thematic CERN School of Computing 2019
In the previous episodes...

- We’ve found out how to store data efficiently
- And how to distribute it
- We’ve learnt how to detect data corruptions
- And made sure our data was safe and consistent
In the previous episodes...

- We’ve found out how to store data efficiently
- And how to distribute it
- We’ve learnt how to detect data corruptions
- And made sure our data was safe and consistent

Today

How do we use them? Efficiently
Key ingredients to achieve effective I/O

Outline

1. Asynchronous I/O
   - Latency
   - Asynchronous I/O interfaces
   - Message queues

2. I/O optimizations
   - Optimizing network transfers
   - Optimizing local transfers

3. Influence of data structures on I/O
   - Measuring I/O efficiency of algorithms

4. Caching
   - Principles
   - Policies
   - Distributed Caches

5. Conclusion
Asynchronous I/O

1. Asynchronous I/O
   - Latency
   - Asynchronous I/O interfaces
   - Message queues

2. I/O optimizations

3. Influence of data structures on I/O

4. Caching

5. Conclusion
Sources of I/O latency

Physical constraints

- preparing an SSD (∼ 100 µs)
- moving disk’s arm (∼ 10 ms)
- mounting and rotating tapes (∼ 1 min)
Sources of I/O latency

**Physical constraints**
- preparing an SSD (∼ 100 µs)
- moving disk’s arm (∼ 10 ms)
- mounting and rotating tapes (∼ 1 min)

**Network infrastructure**
- number of switches routers on the way
- 200 µs delay per switch/router
### Sources of I/O latency

#### Physical constraints
- preparing an SSD ($\sim 100\,\mu s$)
- moving disk’s arm ($\sim 10\, \text{ms}$)
- mounting and rotating tapes ($\sim 1\, \text{min}$)

#### Network infrastructure
- number of switches routers on the way
- $200\,\mu s$ delay per switch/router

#### “slow” speed of light
- “speed” of light is slower in a fiber (refractive index 1.47)
- that gives around $200\,\text{m}\,\mu\text{s}^{-1}$ or $20\,\text{cm}\,\text{ns}^{-1}$
- Budapest’s ping time from CERN is $\sim 21\,\text{ms}$ for $\sim 2500\,\text{km}$
Key ingredients to achieve effective I/O

Impact of I/O latency

Typical I/O pattern

Source  Destination

\[ \text{time} \]

\[ \text{sending time} = \text{sending time} + \text{ping time} \]
Impact of I/O latency

Typical I/O pattern

- send a packet to destination

Source → Destination

First bit sent

time

efficiency = sending time + ping time
Impact of I/O latency

Typical I/O pattern

- send a packet to destination

Source

Destination

First bit sent

Last bit sent

time

efficiency = sending time + ping time
Key ingredients to achieve effective I/O

Impact of I/O latency

Typical I/O pattern

- send a packet to destination
- wait for ack

Source

Destination

First bit sent

Last bit sent

Ack

time

sending time

efficiency

= sending time + ping time
Key ingredients to achieve effective I/O

Impact of I/O latency

Typical I/O pattern

- send a packet to destination
- wait for ack
- go to next block

Impact of I/O latency

Efficiency = sending time + ping time
Key ingredients to achieve effective I/O

Impact of I/O latency

Typical I/O pattern

- send a packet to destination
- wait for ack
- go to next block

Source

Sending time

Destination

First bit sent

Last bit sent

Ack

Time

Efficiency = Sending time + Ping time
Impact of I/O latency

Typical I/O pattern

- send a packet to destination
- wait for ack
- go to next block

\[
\text{efficiency} = \frac{\text{sending time}}{\text{sending time} + \text{ping time}}
\]
Some mathematics

Definitions

\[
\text{efficiency} = \frac{\text{sending time}}{\text{sending time} + \text{ping time}} \tag{1}
\]

\[
\text{sending time} = \frac{\text{data size}}{\text{speed}} \tag{2}
\]

\[
\text{ping size} = \text{speed} \times \text{ping time} \tag{3}
\]
Some mathematics

Definitions

\[
\text{efficiency} = \frac{\text{sending time}}{\text{sending time} + \text{ping time}} \tag{1}
\]

\[
\text{sending time} = \frac{\text{data size}}{\text{speed}} \tag{2}
\]

\[
\text{ping size} = \text{speed} \times \text{ping time} \tag{3}
\]

Gives

\[
\text{efficiency} = \frac{1}{1 + \frac{\text{ping size}}{\text{data size}}} \tag{4}
\]

\[
\text{data size} = \frac{\text{efficiency}}{1 - \text{efficiency}} \times \text{ping size} \tag{5}
\]
## Some (bad) numbers

### Consequences for 10KB blocks

<table>
<thead>
<tr>
<th>Usage</th>
<th>Speed</th>
<th>Latency</th>
<th>Ping Size</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>1 GB s⁻¹</td>
<td>100 µs</td>
<td>10 kB</td>
<td>50%</td>
</tr>
<tr>
<td>CC</td>
<td>10 GB s⁻¹</td>
<td>100 µs</td>
<td>100 kB</td>
<td>9%</td>
</tr>
<tr>
<td>WAN</td>
<td>1 GB s⁻¹</td>
<td>10 ms</td>
<td>1 MB</td>
<td>1%</td>
</tr>
<tr>
<td>WAN</td>
<td>10 GB s⁻¹</td>
<td>10 ms</td>
<td>10 MB</td>
<td>1‰</td>
</tr>
<tr>
<td>UK-JP</td>
<td>10 GB s⁻¹</td>
<td>250 ms</td>
<td>250 MB</td>
<td>0.04‰</td>
</tr>
</tbody>
</table>
More bad numbers

Data size for decent efficiency

<table>
<thead>
<tr>
<th>Usage</th>
<th>Speed</th>
<th>Latency</th>
<th>50% efficiency</th>
<th>91% efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>1 GB s$^{-1}$</td>
<td>100 µs</td>
<td>10 kB</td>
<td>100 kB</td>
</tr>
<tr>
<td>CC</td>
<td>10 GB s$^{-1}$</td>
<td>100 µs</td>
<td>100 kB</td>
<td>1 MB</td>
</tr>
<tr>
<td>WAN</td>
<td>1 GB s$^{-1}$</td>
<td>10 ms</td>
<td>1 MB</td>
<td>10 MB</td>
</tr>
<tr>
<td>WAN</td>
<td>10 GB s$^{-1}$</td>
<td>10 ms</td>
<td>10 MB</td>
<td>100 MB</td>
</tr>
<tr>
<td>UK-JP</td>
<td>10 GB s$^{-1}$</td>
<td>250 ms</td>
<td>250 MB</td>
<td>2.5 GB</td>
</tr>
</tbody>
</table>

Remember maximum TCP packet size is 64 KiB
More bad numbers

Data size for decent efficiency

<table>
<thead>
<tr>
<th>Usage</th>
<th>Speed</th>
<th>Latency</th>
<th>50% efficiency</th>
<th>91% efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>1 GB s(^{-1})</td>
<td>100 µs</td>
<td>10 kB</td>
<td>100 kB</td>
</tr>
<tr>
<td>CC</td>
<td>10 GB s(^{-1})</td>
<td>100 µs</td>
<td>100 kB</td>
<td>1 MB</td>
</tr>
<tr>
<td>WAN</td>
<td>1 GB s(^{-1})</td>
<td>10 ms</td>
<td>1 MB</td>
<td>10 MB</td>
</tr>
<tr>
<td>WAN</td>
<td>10 GB s(^{-1})</td>
<td>10 ms</td>
<td>10 MB</td>
<td>100 MB</td>
</tr>
<tr>
<td>UK-JP</td>
<td>10 GB s(^{-1})</td>
<td>250 ms</td>
<td>250 MB</td>
<td>2.5 GB</td>
</tr>
</tbody>
</table>

Remember maximum TCP packet size is 64 KiB
The solution: asynchronous I/O

Do not wait the acknowledgment!

- call an API to express what should be transferred
- this immediately returns without doing much
- get called back when transfer has been done/has failed
The solution: asynchronous I/O

Do not wait the acknowledgment!

- call an API to express what should be transferred
- this immediately returns without doing much
- get called back when transfer has been done/has failed
The solution: asynchronous I/O

Do not wait the acknowledgment!

- call an API to express what should be transferred
- this immediately returns without doing much
- get called back when transfer has been done/has failed
The solution: asynchronous I/O

Do not wait the acknowledgment!

- Call an API to express what should be transferred.
- This immediately returns without doing much.
- Get called back when transfer has been done/has failed.

Source

Destination

- First bit
- Last bit
- Ack

Time
The solution: asynchronous I/O

Do not wait the acknowledgment!

- call an API to express what should be transferred
- this immediately returns without doing much
- get called back when transfer has been done/has failed
The complexity of asynchronous I/O

Some consequences

- involves multi-threaded / multi-process approach
  - with a background task for the actual transfer.
- involves out of order transfers
  - no guarantee on the order in which transfers will arrive to destination
  - especially in case of failures
- involves queue management on the caller side
  - to not send too much in parallel
  - to adapt to network efficiency
The complexity of asynchronous I/O

Some consequences

- involves multi-threaded / multi-process approach
  - with a background task for the actual transfer.
- involves out of order transfers
  - no guarantee on the order in which transfers will arrive to destination
  - especially in case of failures
- involves queue management on the caller side
  - to not send too much in parallel
  - to adapt to network efficiency

Libraries

Do not try to do all that by hand
Use a library that does (most of) it for you
POSIX asynchronous I/O

**Old interface based on signals**

```c
struct aiocb {
    int aio_fildes; /* fd */
    off_t aio_offset; /* offset */
    volatile void *aio_buf; /* buffer */
    size_t aio_nbytes; /* length */
    int aio_reqprio; /* priority */
    struct sigevent aio_sigevent; /* cb method */
    int aio_lio_opcode; /* operation */
};

int aio_read (struct aiocb *aiocbp);
int aio_write (struct aiocb *aiocbp)
```

The application can elect to be notified of completion of the I/O operation in a variety of ways: by delivery of a signal, by instantiation of a thread, or no notification at all.
Internal tweaks

- The proc file system contains two virtual files that can be tuned for asynchronous I/O performance:
  - The `/proc/sys/fs/aio-nr` file provides the current number of system-wide asynchronous I/O requests.
  - The `/proc/sys/fs/aio-max-nr` file is the maximum number of allowable concurrent requests. The maximum is commonly 64KB, which is adequate for most applications.
Key ingredients to achieve effective I/O

Ceph asynchronous I/O

More modern interface with callbacks

typedef void (*rados_callback_t)(rados_completion_t cb, void *arg);

int rados_aio_create_completion(void *cb_arg, rados_callback_t cb_complete,
                                  rados_callback_t cb_safe,
                                  rados_completion_t *pc);

int rados_aio_write(rados_ioctx_t io, const char *oid,
                     rados_completion_t completion,
                     const char *buf, size_t len, uint64_t off);

int rados_aio_wait_for_safe(rados_completion_t c);

Handling of threading is done for you
Key ingredients to achieve effective I/O

Ceph asynchronous I/O

example code

```c
void commit_cb(rados_completion_t comp, void *arg) {
    struct timeval *t = (struct timeval *) arg;
    gettimeofday(&t, NULL);
}

int main() {
    ... // connect to ceph
    struct timeval commit_end;
    rados_completion_t comp;
    rados_aio_create_completion
      ((void*) &commit_end, NULL, commit_cb, &comp);
    rados_aio_append(ioctx, obj_name, comp, data, len);
    rados_aio_flush(io); // wait for call back
    // print commit_end
}
```
Modern asynchronous I/O internals

Architecture

- based on message queuing
- both for requests and responses

Client

Request queue

Request Handling

Storage

Response queue

Response Handling

submit

pop

I/O

call back

pop
Advantages of message queues

Architecture

- disentangles client API calls from actual processing
- allows optimizations on both sides
  - asynchronous I/O on client side
  - reordering of requests on server side
- allows easy scalability, parallelizing the handlers
  - using thread safe or even distributed queues
I/O optimizations

1. Asynchronous I/O

2. I/O optimizations
   - Optimizing network transfers
   - Optimizing local transfers

3. Influence of data structures on I/O

4. Caching

5. Conclusion
Optimizing network I/O

Golden rules

- async I/O is a must
- setsockopt is often needed
  - as default parameters may not be optimal for your case
  - e.g. TCP_NODELAY or SND_BUF
The TCP buffer

Actual steps writing to a TCP socket

1. write to the TCP buffer
2. create a packet and actually send it to the network device
The Nagle algorithm of TCP

Reason of being
- avoid sending too many small packets when write to TCP buffer are very small
  - as there is 40 bytes overhead (20 for TCP, 20 for IPv4)
- initially for key pressing in telnet sessions over ARPANET (1984 !)

Algorithm
- if enough data \((\geq\) maximum segment size), send a packet
- else if there is unconfirmed data still in the pipe
  - enqueue data in the buffer until an acknowledge is received
- else send data immediately
Potential bad consequences

TCP delayed acknowledgment
- allows to combine several acknowledgments into a single response to reduce protocol overhead
- ACK may wait up to 500 ms

Collision of features
- suppose you use a socket in “write-read-read” schema
  - e.g. send control packet “give me more” and read until nothing left
- a delay of 500 ms will be introduced
Bypass Nagle via TCP_NODELAY

Code example

```c
int flag = 1;
int result =
    setsockopt(sock, /* socket */
               IPPROTO_TCP, /* TCP level */
               TCP_NODELAY, /* option */
               (char *)&flag, /* value */
               sizeof(int)); /* length */
if (result < 0) { ... error handling ... }
```
A note on the size of the TCP buffer

**Standard case**

- default size is used (16 kB)
- allows to create packets up to 16 kB
- already not able to use the maximum of 64 kB
A note on the size of the TCP buffer

**Standard case**
- default size is used (16 kB)
- allows to create packets up to 16 kB
- already not able to use the maximum of 64 kB

**Bad case**
- via setsockopt and SND_BUF option, the TCP buffer is shortened
  - a bug strikes, size is set to 0, which will be corrected into 1448
- many small packets will be sent
- latency will kill efficiency
  - e.g. to ≤ 5 MB s\(^{-1}\)

Key ingredients to achieve effective I/O
A note on the size of the TCP buffer

**Standard case**
- default size is used (16 kB)
- allows to create packets up to 16 kB
- already not able to use the maximum of 64 kB

**Bad case**
- via `setsockopt` and `SND_BBUF` option, the TCP buffer is shortened
  - a bug strikes, size is set to 0, which will be corrected into 1448
- many small packets will be sent
- latency will kill efficiency
  - e.g. to $\leq 5 \text{ MB s}^{-1}$
- and CMS may call you for complaining....
Anatomy of local I/O in Linux

1. Client requests bytes from VFS
   Virtual File System

   Client
   ↓
   VFS
Anatomy of local I/O in Linux

1. Client requests bytes from VFS (Virtual File System)
2. Kernel checks whether they are available in cache

Diagram:
- Client
- VFS
- Empty blocks

Steps:
1. Client requests bytes from VFS
2. Kernel checks whether they are available in cache
3. If not, reads a 4K block from device and updates cache
4. Bytes are read from cache and given to application
5. Subsequent reads skip step 3
Anatomy of local I/O in Linux

1. Client requests bytes from VFS (Virtual File System)
2. Kernel checks whether they are available in cache
3. If not, it reads a 4K block from device and updates cache
Anatomy of local I/O in Linux

1. Client requests bytes from VFS (Virtual File System).
2. Kernel checks whether they are available in cache.
3. If not, it reads a 4K block from device and updates cache.
4. Bytes are read from cache and given to application.
Anatomy of local I/O in Linux

1. Client requests bytes from VFS (Virtual File System)
2. Kernel checks whether they are available in cache
3. If not, it reads a 4K block from device and updates cache
4. Bytes are read from cache and given to application
5. Subsequent reads skip step 3
The write case

1. Client sends bytes to VFS

Client

VFS

Device

Possible data loss between step 2 and 3, data are only in cache. You need to use fsync to sync to the device.
The write case

1. Client sends bytes to VFS
2. Kernel overwrite block(s) in cache and marks the cache dirty

Possible data loss between step 2 and 3, data are only in cache you need to use fsync to sync to the device.
Key ingredients to achieve effective I/O

The write case

1. Client sends bytes to VFS
2. Kernel overwrite block(s) in cache and marks the cache dirty
3. “Later”, the cache is synced to disk

Possible data loss between step 2 and 3, data are only in cache. You need to use fsync to sync to the device.
The write case

1. Client sends bytes to VFS
2. Kernel overwrite block(s) in cache and marks the cache dirty
3. "later", the cache is synced to disk

Possible data loss
- between step 2 and 3, data are only in cache
- you need to use fsync to sync to the device
More data loss...

**fsync**
- only forces the cache to be synced to the device
- that is the content of the cache to be sent to the device

**But...**
- there may be many other caches involved
  - in the controller
  - in the hardware, e.g. inside the disk itself
- so after syncing, data may still be at risk
More data loss...

**fsync**
- only forces the cache to be synced to the device
- that is the content of the cache to be sent to the device

**But...**
- there may be many other caches involved
  - in the controller
  - in the hardware, e.g. inside the disk itself
- so after syncing, data may still be at risk

In practice, massive reboots $\Leftrightarrow$ loss of data!
Block size matters

Back to our read case:

1. Client requests bytes from VFS (Virtual File System)
2. Kernel checks whether they are available in cache
3. If not, it reads a **4K block** from device and updates cache
4. Bytes are read from cache and given to application
Block size matters

Back to our read case:

1. client requests bytes from VFS (Virtual File System)
2. kernel checks whether they are available in cache
3. if not, it reads a **4K block** from device and updates cache
4. bytes are read from cache and given to application

- scattered tiny reads will transform into 4K blocks anyway
- so scanning events, reading only the 4 bytes of their type may not be very efficient!
Block size: the real killer

Remember RAID 5 - aka Parity

Disk 1  Disk 2  Disk 3  Disk 4

$A_1$  $A_2$  $A_3$  $P_A$

Remember RAID 5 - aka Parity

Block size: the real killer

Key ingredients to achieve effective I/O

Block size: the real killer

Key ingredients to achieve effective I/O

Block size: the real killer

Key ingredients to achieve effective I/O
Block size: the real killer

Remember RAID 5 - aka Parity

Disk 1  Disk 2  Disk 3  Disk 4

4 bytes write into into $A_1$

- the parity will have to be recomputed
  - either $A_1$ and $P_A$ or all $A_i$ will be read
  - and $A_1$ and $P_A$ will written back

- that’s a minimum of 8K reads + 8K writes for 4 bytes!

- can be much worse for RAID6 or erasure coding
Combined RAID levels

RAID 60

- RAID levels can be put on top of each other
- 60 means a RAID 0 on top of a RAID 6
- Let’s see how a file A is stored on such a configuration
Key ingredients to achieve effective I/O

Combined RAID levels

RAID 60
- RAID levels can be put on top of each other
- 60 means a RAID 0 on top of a RAID 6
- Let’s see how a file A is stored on such a configuration

Layout of file A

RAID 0
- Virtual Disk 1
  - A1
- Virtual Disk 2
  - A2
Combined RAID levels

**RAID 60**
- RAID levels can be put on top of each other
- 60 means a RAID 0 on top of a RAID 6
- Let’s see how a file A is stored on such a configuration

**Layout of file A**

![Diagram of RAID 60 configuration]

- Virtual Disk 1
  - Disk 1: $A_{1.1}$
  - Disk 2: $A_{1.2}$
  - Disk 3: $P_A$
  - Disk 4: $Q_A$

- Virtual Disk 2
  - Disk 5: $A_{2.1}$
  - Disk 6: $A_{2.2}$
  - Disk 7: $P_A$
  - Disk 8: $Q_A$
Key ingredients to achieve effective I/O

**Block size and combined RAID levels**

2 levels - 2 chunk sizes, ideally like this

![Diagram of RAID levels and block sizes](image-url)
Incoherent chunk sizes

Consequence of block size incoherence

- suppose you have the same chunk size for both levels
- writing a chunk to the RAID 0 will only fill one of the chunks in the RAID 6 stripe
- it will trigger the reading of the rest to recompute the parities
- leading to 3 reads and 3 writes
- to fill a stripe of the RAID 6, this will be repeated twice
Incoherent chunk sizes

Consequence of block size incoherence

- suppose you have the same chunk size for both levels
- writing a chunk to the RAID 0 will only fill one of the chunks in the RAID 6 stripe
- it will trigger the reading of the rest to recompute the parities
- leading to 3 reads and 3 writes
- to fill a stripe of the RAID 6, this will be repeated twice

Bottom line

- 6 reads and 6 writes instead of 4 writes!
- a slow down of a factor 3 minimum
Incoherent chunk sizes

Consequence of block size incoherence

- suppose you have the same chunk size for both levels
- writing a chunk to the RAID 0 will only fill one of the chunks in the RAID 6 stripe
- it will trigger the reading of the rest to recompute the parities
- leading to 3 reads and 3 writes
- to fill a stripe of the RAID 6, this will be repeated twice

Bottom line

- 6 reads and 6 writes instead of 4 writes!
- a slow down of a factor 3 minimum
- in case of 8+2 disks in the RAID6, the slow down is of a factor 5!
**O_DIRECT: bypass the kernel cache**

**When / Why?**
- when reading large amounts of data sequentially
  - so that you avoid wiping the cache
- if you’re handling caching by hand
  - typical use case is databases

**How**

```c
int fd = open("myfile", O_RDONLY | O_DIRECT, ...);
```
Data duplication

Back again to our read case:

1. client requests bytes from VFS
2. kernel checks the cache
3. kernel transfers a 4K block from device to cache
4. bytes are read from cache and **given to** application
Back again to our read case:

1. Client requests bytes from VFS
2. Kernel checks the cache
3. Kernel transfers a 4K block from device to cache
4. Bytes are read from cache and *given to application*

- "given to application" $\Leftrightarrow$ "copied to user buffer"
  - It takes CPU time
  - It hurts the CPU cache
  - Only to duplicate the data into memory!
Memory-mapped files

**Idea**

- allows direct use of kernel cache, mapped to user buffer
- for up to 30% gain compared to regular file read
Memory-mapped files

Idea
- allows direct use of kernel cache, mapped to user buffer
- for up to 30% gain compared to regular file read

Usage
```c
int *map; /* mmapped array of int's */
int fd = open(FILEPATH, O_RDONLY);
if (fd == -1) { ... }
map = mmap(0, SIZE, PROT_READ, MAP_SHARED, fd, 0);
if (map == MAP_FAILED) { ... }
/* access file through map array */
if (munmap(map, FILESIZE) == -1) { ... }
close(fd);
```
slightly more complicated for writing but still easy
Read-ahead mechanism

**Principle**
- when you read a block of a file, there is high chance you will read the next few ones
- cache misses in the kernel cache are expensive (disk seeks)
- so why not read the next few blocks from the start?

**practical usage**
- just do nothing, this is on by default
- give advices to kernel for special cases:
  - use posix_fadvise, posix_madvise or madvice (non POSIX)
  - advices include POSIX_FADV_SEQUENTIAL, POSIX_FADV_RANDOM, POSIX_FADV_NOREUSE
Influence of data structures on I/O

1. Asynchronous I/O
2. I/O optimizations
3. Influence of data structures on I/O
   - Measuring I/O efficiency of algorithms
4. Caching
5. Conclusion
Algorithm efficiency

Standard efficiency metric
- speed, that is time spent per data
- mostly CPU time
- mapping to number of operations, e.g. $O(n^2)$
Algorithm efficiency

Standard efficiency metric

- speed, that is time spent per data
- mostly CPU time
- mapping to number of operations, e.g. $O(n^2)$

Well... what about I/O?

- e.g. scan 1GB of logs and count nb Errors
- is CPU dominant?
Algorithm efficiency

Standard efficiency metric
- speed, that is time spent per data
- mostly CPU time
- mapping to number of operations, e.g. \( O(n^2) \)

Well... what about I/O?
- e.g. scan 1GB of logs and count nb Errors
- is CPU dominant?
- depends whether data is in RAM or on disk...
Data access latencies

Remember the numbers of Danilo

<table>
<thead>
<tr>
<th>Device</th>
<th>Latency (time)</th>
<th>Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg</td>
<td>0.4 ns</td>
<td>1</td>
</tr>
<tr>
<td>L1</td>
<td>1.5 ns</td>
<td>4</td>
</tr>
<tr>
<td>L2</td>
<td>4 ns</td>
<td>10</td>
</tr>
<tr>
<td>L3</td>
<td>14 ns</td>
<td>35</td>
</tr>
<tr>
<td>Mem</td>
<td>80 ns</td>
<td>200</td>
</tr>
<tr>
<td>SSD</td>
<td>12 µs</td>
<td>30 000</td>
</tr>
<tr>
<td>HDD</td>
<td>4 ms</td>
<td>10 000 000</td>
</tr>
</tbody>
</table>

Bottom line: never, ever go to disk if not absolutely needed! Care about cache hits in general.
Data access latencies

<table>
<thead>
<tr>
<th>Device</th>
<th>Latency (time)</th>
<th>Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg</td>
<td>0.4 ns</td>
<td>1</td>
</tr>
<tr>
<td>L1</td>
<td>1.5 ns</td>
<td>4</td>
</tr>
<tr>
<td>L2</td>
<td>4 ns</td>
<td>10</td>
</tr>
<tr>
<td>L3</td>
<td>14 ns</td>
<td>35</td>
</tr>
<tr>
<td>Mem</td>
<td>80 ns</td>
<td>200</td>
</tr>
<tr>
<td>SSD</td>
<td>12 μs</td>
<td>30 000</td>
</tr>
<tr>
<td>HDD</td>
<td>4 ms</td>
<td>10 000 000</td>
</tr>
</tbody>
</table>

Bottom line:
- never, ever go to disk if not absolutely needed!
- care about cache hits in general
Algorithm I/O efficiency

- we use mem and disk here
- but it applies to any two levels

Notations

- $N$ number of items in the problem
- $M$ number of items fitting in memory
- $B$ number of items fitting in a disk block
Algorithm I/O complexity

Fundamental Bounds of I/O complexity

- scanning: \( O\left(\frac{N}{B}\right) \)
- searching: \( O\left(\log_B N\right) \)
- sorting: \( O\left(\frac{N}{B} \log_M \frac{N}{B}\right) \)
Algorithm I/O complexity

Fundamental Bounds of I/O complexity

- scanning: \( O\left(\frac{N}{B}\right) \)
- searching: \( O\left(\log_B N\right) \)
- sorting: \( O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right) \)

Importance of \( B \)

- \( N = 256 \times 10^6, B = 8000, \) 1ms disk access time
- \( N \) I/Os take \( 256 \times 10^3 \) sec = 71 h
- \( \frac{N}{B} \) I/Os take 32 s
Real life example: linked list

standard implementation
- nodes are stored in an array and contain pointer to next
- new nodes are inserted at the end of the array
- but in the middle of the linked list
- so most of links are across disk blocks
Real life example: linked list

Standard implementation

- Nodes are stored in an array and contain pointer to next
- New nodes are inserted at the end of the array
- But in the middle of the linked list
- So most of links are across disk blocks

Complexity

- Insertion: \(O(1)\)
- Scan: \(O(N)\)
Real life example: linked list

**standard implementation**

- nodes are stored in an array and contain pointer to next
- new nodes are inserted at the end of the array
- but in the middle of the linked list
- so most of links are across disk blocks
Real life example: linked list

standard implementation

- nodes are stored in an array and contain pointer to next
- new nodes are inserted at the end of the array
- but in the middle of the linked list
- so most of links are across disk blocks

23 20 9 11 5 19
Key ingredients to achieve effective I/O

Real life example: linked list

Standard implementation

- Nodes are stored in an array and contain a pointer to the next node.
- New nodes are inserted at the end of the array, but in the middle of the linked list.
- So most of the links are across disk blocks.

Complexity:

- Insertion: $O(1)$
- Scan: $O(N)$
Real life example: linked list

Standard implementation:

- Nodes are stored in an array and contain pointer to next
- New nodes are inserted at the end of the array
- But in the middle of the linked list
- So most of links are across disk blocks

Complexity:

- Insertion: $O(1)$
- Scan: $O(N)$
Real life example: linked list

Standard implementation:
- Nodes are stored in an array and contain pointer to next
- New nodes are inserted at the end of the array
- But in the middle of the linked list
- So most of links are across disk blocks

Complexity:
- Insertion: $O(1)$
- Scan: $O(N)$
Improved linked list

Ordered linked list

- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in “list order”, blocks contain consecutive elements
- most of the links point to same block

Complexity:
- insertion: $O(NB)$
- scan: $O(NB)$
**Improved linked list**

**Ordered linked list**

- Nodes are stored in an array and contain a pointer to the next.
- New nodes are inserted in the middle of the array.
- Nodes are kept in “list order”, blocks contain consecutive elements.
- Most of the links point to the same block.

Complexity:
- Insertion: $O(N_B)$
- Scan: $O(N_B)$
Improved linked list

Ordered linked list

- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in “list order”, blocks contain consecutive elements
- most of the links point to same block

Complexity

- insertion: $O(NB)$
- scan: $O(NB)$
Key ingredients to achieve effective I/O

Improved linked list

Ordered linked list

- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in “list order”, blocks contain consecutive elements
- most of the links point to same block

Complexity

- insertion: $O(N_B)$
- scan: $O(N_B)$
**Improved linked list**

**Ordered linked list**
- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in “list order”, blocks contain consecutive elements
- most of the links point to same block
Improved linked list

Ordered linked list

- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in “list order”, blocks contain consecutive elements
- most of the links point to same block

Complexity:
- insertion: $O(N_B)$
- scan: $O(N_B)$
Improved linked list

Ordered linked list

- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in “list order”, blocks contain consecutive elements
- most of the links point to same block
Key ingredients to achieve effective I/O

Improved linked list

Ordered linked list

- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in “list order”, blocks contain consecutive elements
- most of the links point to same block
**Improved linked list**

**Ordered linked list**
- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in “list order”, blocks contain consecutive elements
- most of the links point to same block

![Linked List Diagram]

**Complexity**
- insertion $O\left(\frac{N}{B}\right)$
- scan $O\left(\frac{N}{B}\right)$
Key ingredients to achieve effective I/O

## I/O Efficient linked list

### Ordered sparse linked list

- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse
I/O Efficient linked list

Ordered sparse linked list

- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse
Key ingredients to achieve effective I/O

I/O Efficient linked list

Ordered sparse linked list

- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse
Key ingredients to achieve effective I/O

I/O Efficient linked list

Ordered sparse linked list

- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse
I/O Efficient linked list

Ordered sparse linked list
- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse
I/O Efficient linked list

Ordered sparse linked list

- keep same ideas
  - nodes put in "list order"
  - links point mainly to same block
- but allow blocks to be sparse
Key ingredients to achieve effective I/O

I/O Efficient linked list

Ordered sparse linked list

- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse
I/O Efficient linked list

Ordered sparse linked list

- keep same ideas
  - nodes put in "list order"
  - links point mainly to same block
- but allow blocks to be sparse
I/O Efficient linked list

Ordered sparse linked list

- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse

Complexity:
- insertion: \( O(1) \)
- scan: \( O(NB) \)
Key ingredients to achieve effective I/O

I/O Efficient linked list

- Ordered sparse linked list
  - keep same ideas
    - nodes put in “list order”
    - links point mainly to same block
  - but allow blocks to be sparse
Key ingredients to achieve effective I/O

I/O Efficient linked list

Ordered sparse linked list
- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse

Complexity
- insertion $O(1)$
- scan $O\left(\frac{N}{B}\right)$
I/O complexity in general

Applies to more complex structures / algorithms
- queues, stack, trees
- searching, sorting

Is always a trade off with CPU complexity
- ordered linked list means $O(n)$ CPU complexity at insertion
- a tree of nodes stored in an array may be a better alternative

Libraries exist
- In particular STXXL, the STL for XXL Data Sets
Key ingredients to achieve effective I/O

1. Asynchronous I/O
2. I/O optimizations
3. Influence of data structures on I/O
4. Caching
   - Principles
   - Policies
   - Distributed Caches
5. Conclusion
## Caching principles

### Why to use (custom) caches

- to optimize access to a costful resource (network, storage, ...)
- to take benefit of particular access patterns
Caching principles

Why to use (custom) caches
- to optimize access to a costful resource (network, storage, ...)
- to take benefit of particular access patterns

When to use custom caches
- when you have a particular, well defined access pattern
  - e.g. web browsing, with mostly static pages
  - e.g. tape/archive handling
  - e.g. proxy architecture for software distribution
- for specific data structures that would benefit from a cache
  - e.g. trees in the root framework
Caching usage

Where to use caching

- server side to benefit of cross client hits
- client side to avoid network traffic
  - think of your browser cache
- in a proxy to lower load on the server
- locally, in front of “expensive” storage
  - tape → disk cache
  - disk → ssd/memory cache
Anatomy of a cache

Architecture
- it has a limited size
- so must use some garbage collection algorithm

Policies
Replacement policy which item to discard when cache is full and new item is coming
Writing policy how to handle writes of new items to the system
Write-miss policy how to handle “write misses”
Replacement policies

**FIFO - First In First Out**
- most naive and easiest to implement (simple list)
- not very effective in practice

**LRU - Least Recently Used**
- still easy to implement, just needs some “age bits”
- one of the most common

**Others**
- **LFU** Least Frequently Used
  - refinement on LRU where number of accesses is taken into account
- **LIRS** Low Inter-reference Recency Set (LIRS)
  - using reuse distance as a metric for dynamically ranking accessed pages
Writing policies (1)

Write-through

- write is done synchronously both to the cache and to the backend store
- no caching gain on writes
- backend store is always up to date
Writing policies (2)

Write-back

- Writing is done only to the cache initially.
- The write to the backing store is postponed until the cache blocks containing the data are about to be modified/replaced by new content.
- Writes are faster.
- But backend store is not up to date.
- Data may be at risk.

Diagram:
- Data Request
  - Cache Hit? Yes: Read from cache, Done
  - Cache Hit? No: Dirty block?
    - Yes: Write back to backend
    - No: Retrieve from backend, mark clean
  - Dirty block? No: Write into cache, mark dirty

- Write back to backend
- Retrieve from backend
- Write into cache mark dirty
- Read from cache
Write-miss policies

**Write allocate**
- datum at the missed-write location is loaded to cache, followed by a write-hit operation
- write misses are similar to read misses

**No write allocate**
- datum at the missed-write location is not loaded to cache
- it is written to the backend
- no cache for writes
Key ingredients to achieve effective I/O

**Write-miss policies**

**Write allocate**
- datum at the missed-write location is loaded to cache, followed by a write-hit operation
- write misses are similar to read misses

**No write allocate**
- datum at the missed-write location is not loaded to cache
- it is written to the backend
- no cache for writes

---

Data Request

```
Data Request

<table>
<thead>
<tr>
<th>Cache Hit?</th>
<th>Dirty block?</th>
<th>Write back to backend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
```

- **Cache Hit?**
  - Yes: Proceed with write-hit operation
  - No: Continue with write-miss policy

- **Dirty block?**
  - Yes: Write back to backend
  - No: Proceed with write-miss policy

- **Write back to backend**

- **Retrieve from backend**

- **Mark clean**

- **Read from cache**

- **Write into cache**

- **Mark dirty**

- **Write to backend**

- **Done**
**Write-miss policies**

**Write allocate**
- datum at the missed-write location is loaded to cache, followed by a write-hit operation
- write misses are similar to read misses

**No write allocate**
- datum at the missed-write location is not loaded to cache
- it is written to the backend
- no cache for writes
Parallelization and caches

The problem

- you scale horizontally by adding more servers/processes
- each server has its own cache
- all caches will tend to have the same content
- so the cache size will not scale
Parallelization and caches

The problem

- you scale horizontally by adding more servers/processes
- each server has its own cache
- all caches will tend to have the same content
- so the cache size will not scale

Solutions

- use shared caches and tackle the synchronization issues
  - locally via shared memory
  - across servers via shared filesystem
- do nothing, you’re fine with it
  - you have automatic replication of hot items
  - your caches are big enough
Key ingredients to achieve effective I/O

Cache consistency

The problem

- with distributed caches, you main have several copies of one data
- do you have the same version in all cases?
  - e.g. in case it was just overwritten in one cache with write-back policy
Cache consistency

The problem

- with distributed caches, you mainly have several copies of one data
- do you have the same version in all cases?
  - e.g. in case it was just overwritten in one cache with write-back policy

Solutions

- use write-through policy
- not caring
  - e.g. in case of web browsing for pretty static pages
- add cleverness
  - inform all caches when setting a dirty flag
  - syncing all caches for the given data when retrieving from backend
Conclusion

1. Asynchronous I/O
2. I/O optimizations
3. Influence of data structures on I/O
4. Caching

5. Conclusion
Conclusion

Key messages of the day

- Take care of latencies and use asynchronous I/O when needed
- In case of bad performance, optimize your I/O
- Make sure that you don’t do more I/O than needed
- Caches may help at all levels