EEL-4736/5737
Principles of Computer System Design

Lecture Slides 13

Textbook Chapter 6

Designing for Performance

Introduction

- We will study principles used to design for performance
- Approaches to scheduling

Designing for Performance

- Systems have performance bottlenecks
  - Physical limits
    - Speed of light
    - Power and heat dissipation
  - Sharing/contention
    - When a device/service is busy, subsequent operations are delayed
- Designing for performance requires understanding of bottlenecks, and cost,complexity/performance trade-offs

Example – single system

- Exposing memory abstraction, enforcing modularity with virtual memory – part of the requirements of many systems
- A naïve implementation using a single memory technology would be too slow
  - Design principles/hints used
  - Hierarchy of modules – caches, memory, storage
  - Make the common case fast – exploit empirical behavior of locality

Example – distributed system

- Network file system (NFS)
  - Initially target local-area general-purpose file systems (versions 2 and 3)
  - Desire to support wide-area environments, with significantly longer latencies, resulted in deep protocol changes (version 4)
    - No longer stateless, and with server-to-client "call-backs" to support aggressive caching
    - Request combining – coalescing multiple NFS calls in one RPC request

Metrics

- What does a designer want to improve or optimize?
- Example:
  - Camera generates continuous requests with frames of a video to a service
  - Service does digital signal processing to filter images and forwards to a storage service
  - Storage service stores frames on disk
  - Frames/second? Delay for the first frame to become available to other clients?

Capacity

- Measure a service’s size or amount of resources
  - Total number of bytes in storage system
  - Total number of processor cycles available in an interval
- Utilization
  - How many bytes are being used? How many cycles?
- Layered system – utilization from layer below is *overhead*
  - 95% CPU utilization – 70% application (useful work), 25% O/S (overhead)

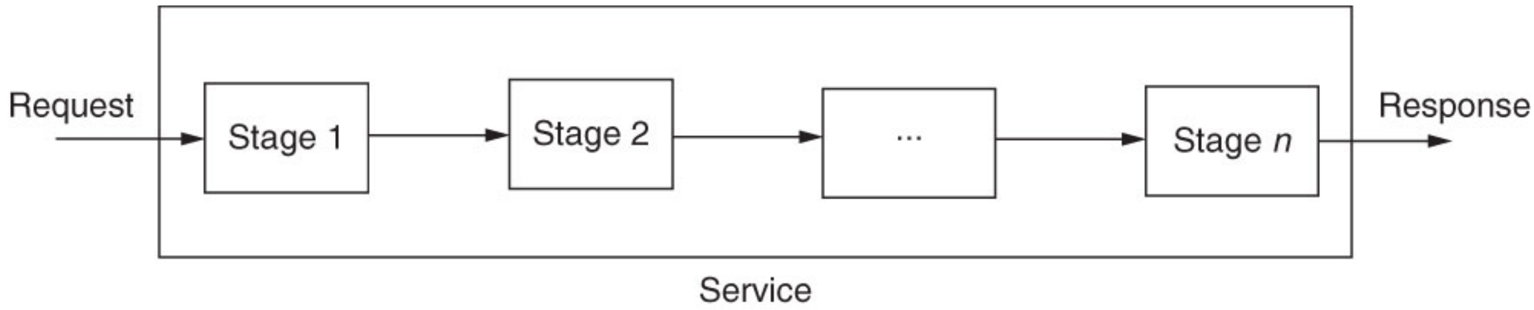
Latency

- Delay between change at the input of a system and corresponding change at output
  - E.g.: time elapsed between client stub issues an RPC call and receives a response
  - Modules that are invoked in sequence accumulate the overall latency: system call processing, marshalling, network, service processing, …

Throughput

- Measure of the *rate* of useful work performed by a service
  - E.g. frames/second processed by DSP service
  - Bytes/second archived in storage service
  - Bytes/second in the network segments connecting each of these devices

Service pipelines



- Latency >= sum (stage latencies)
  - Propagation delays
- Throughput <= min (stage throughputs)
  - One stage can be a bottleneck
- For each stage, on average:
  - Throughput α 1/latency

Latency and throughput

- If stages process requests serially
  - Average throughput of complete pipeline inversely proportional to average latency in the pipeline
- If stages process requests concurrently
  - Relationship between latency and throughput is not straightforward
  - Increase in throughput may not lead to decrease in latency
    - E.g. parallel network links

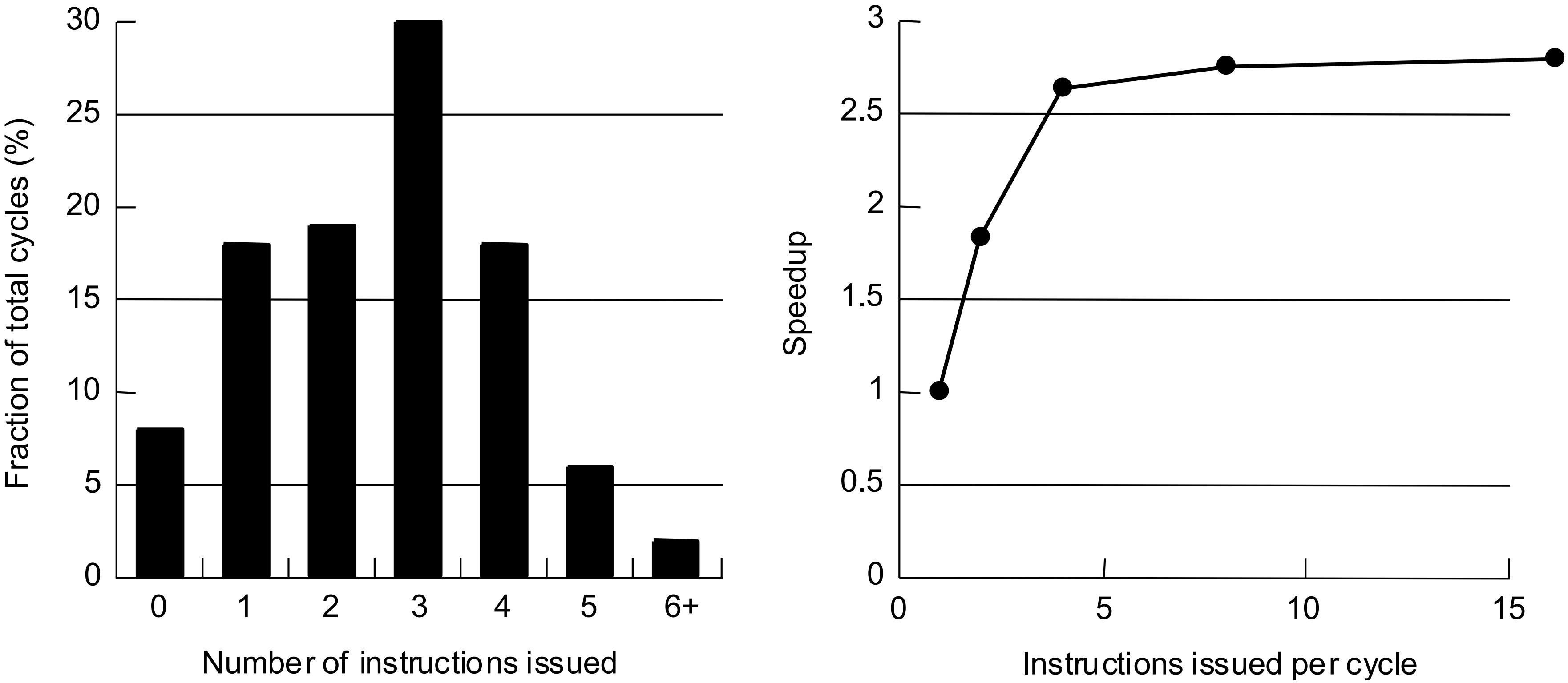
Designing for performance

- Assume there is a design to begin with that one wishes to improve
  - How much improvement can be achieved?
  - What are the stages where improvements should focus?
- Critical to identify bottlenecks
  - Otherwise, may improve a stage that results in small overall performance improvements
    - Diminishing returns
  - Quantitative approach is necessary: models, measurements, guided by experience and insight
    - “Premature optimization is the root of all evil” (Hoare)

Iterative approach

- 1) Measure the system
  – Is a performance enhancement needed? Which metric (latency, throughput)?
- 2) Identify bottlenecks
- 3) Predict impact of enhancement with back-of-the-envelope model
  – Modeling and prediction themselves take design time; d(technology)/d(t) may be sufficient
  – Unrealistic extreme cases (zero latency; infinite throughput) help determine next bottleneck if one is taken care of
- 4) Measure the new design
- 5) Iterate

Example – Superscalar uP



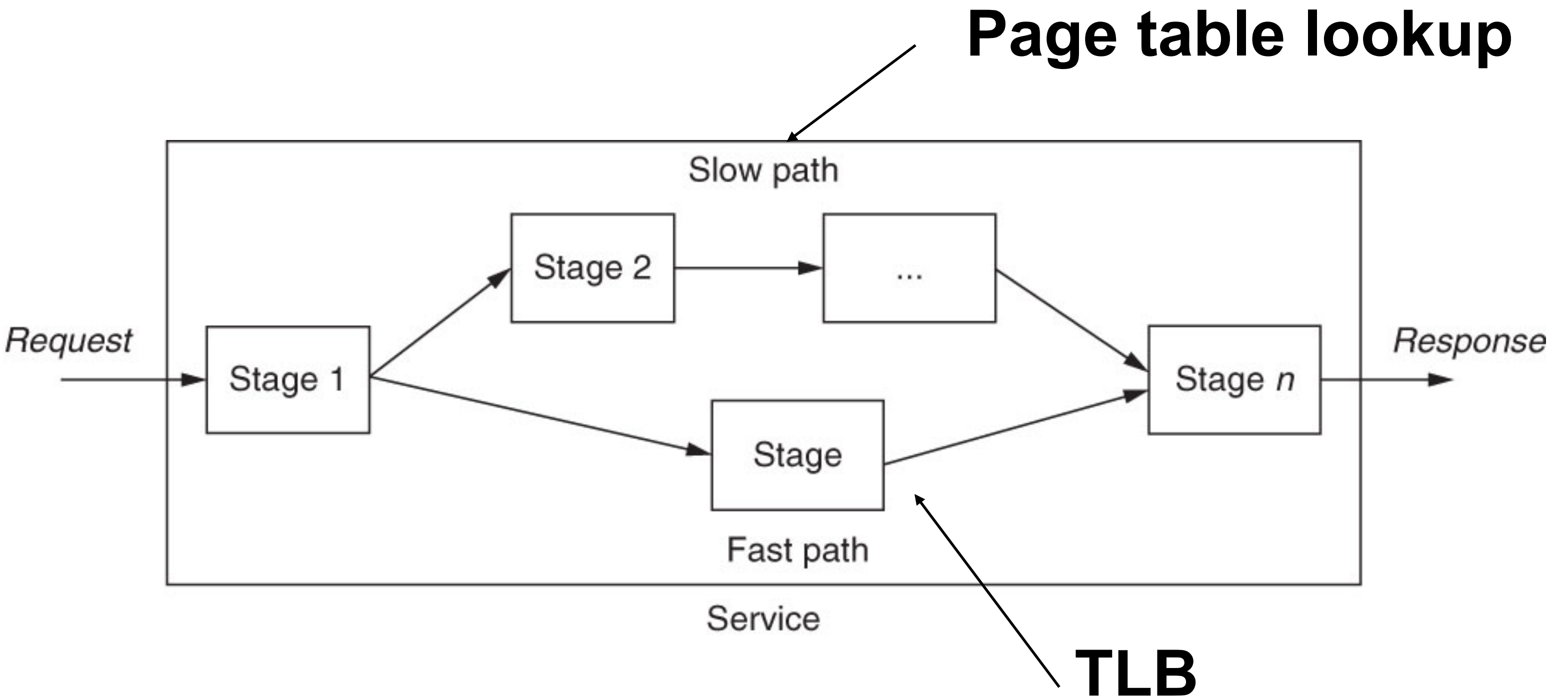
- Infinite resources and fetch bandwidth, perfect branch prediction and renaming
  - real caches and non-zero miss latencies

Reducing latency

- Always good, often difficult, sometimes not possible
  - Economic arguments
    - Bleeding-edge technology (e.g. fastest transistor) is more expensive
    - Larger structures also cost more (e.g. caches)
  - Algorithmic arguments
    - E.g. sequential sort is O(n*log(n))
  - Physical limitations
    - Speed of light; energy dissipation

Exploiting workload properties

- In many scenarios, some requests are more common than others
  - E.g. workload access to memory is not random
  - **Optimize for the common case**
  - Potentially slowing down less common cases



Average latency

- Average Latency =

  Freq_fastpath * Latency_fastpath +

  Freq_slowpath * Latency_slowpath

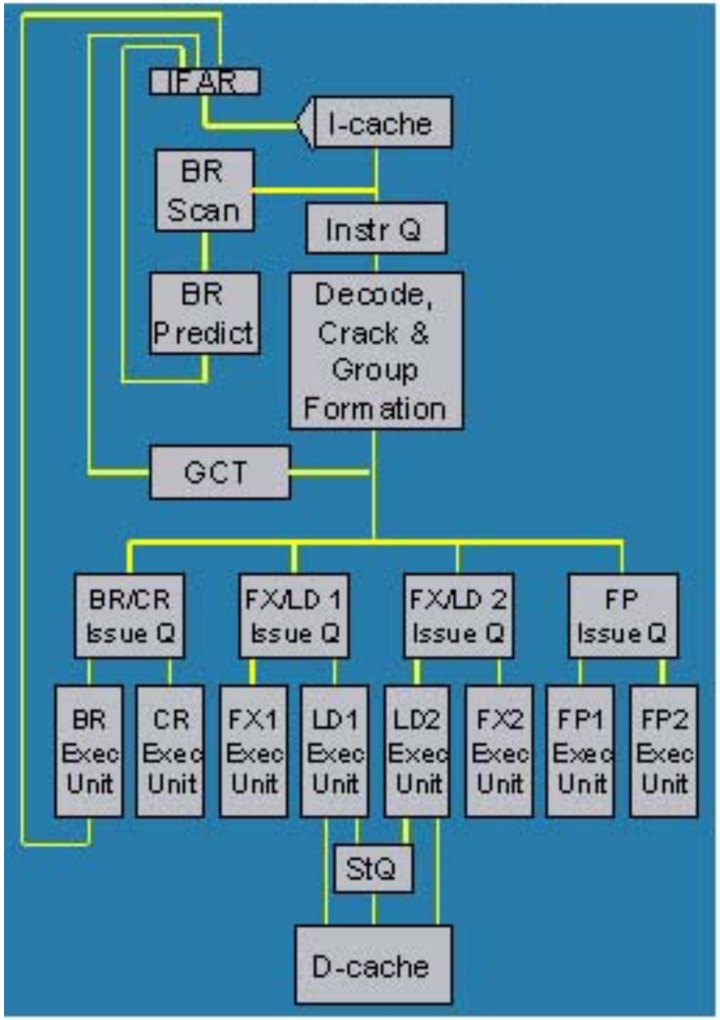
- E.g. caches, TLB
  - Freq_fastpath: upper 90%s
  - Latency_slowpath/Latency_fastpath: 10s, 100s

Reducing latency - concurrency

- Parallelize a stage
  – E.g. filter stage: split the camera image in “n” chunks and use “n” processors to work concurrently on each chunk
- Ideal speedup is “n”
  – Real speedup depends on algorithm, communication, synchronization needs
    - E.g. boundaries between image chunks require communication
    - Must synchronize concurrent threads before proceeding to next stage - locking

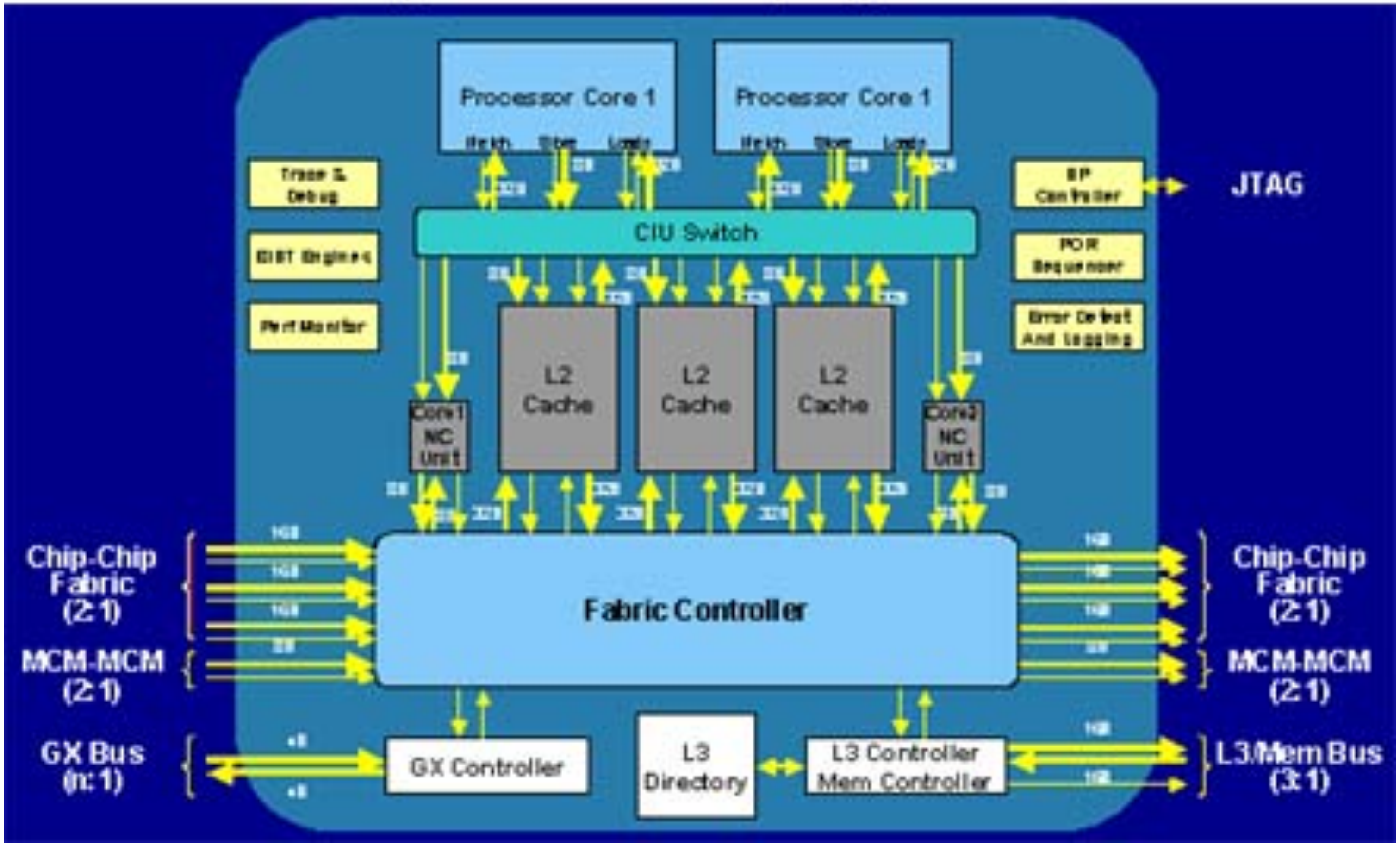
Example – Instruction parallelism

Figure 2: POWER4 Core



Example – Thread parallelism

Figure 1: POWER4 Chip Logical View



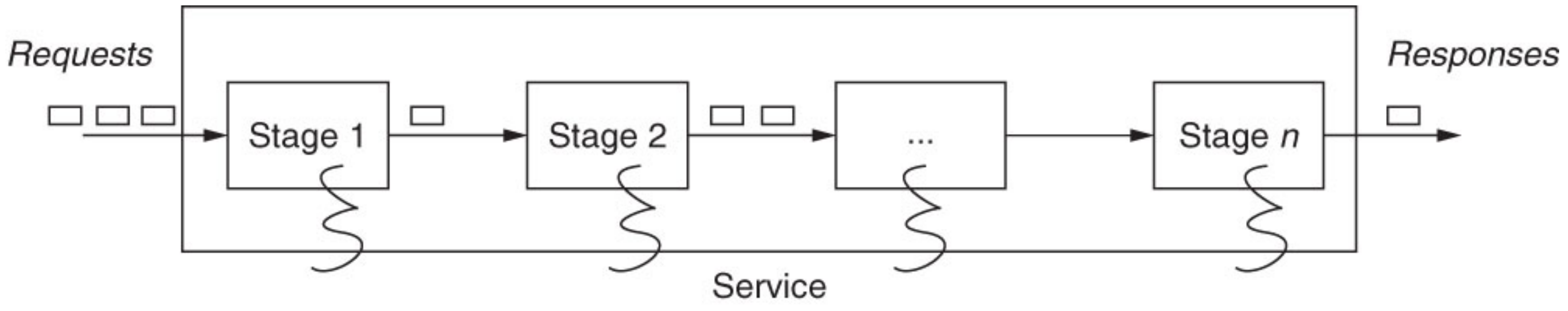
Example – ILP vs TLP

- Instruction-level parallelism (ILP): diminishing returns
- Multiple, independent processors: potential for higher performance gains
  – Requires applications to explicitly deal with parallelism – thread-level parallelism (TLP)
  – Compared to ILP, which focuses on concurrency within the scope of a single thread

Improving throughput: concurrency

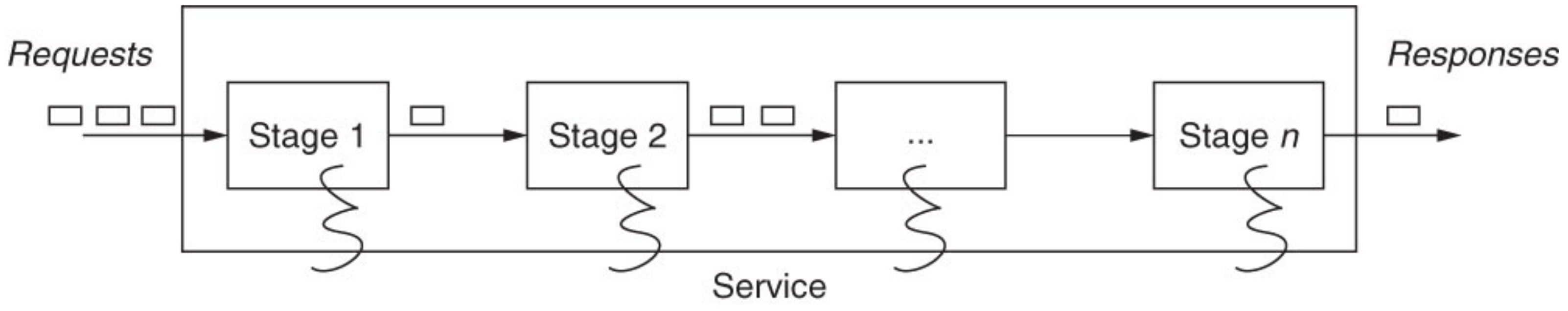
- Reducing latency may not be feasible because of limits
- Overlapping requests with other requests can *hide* latency and improve throughput

Assembly-line style
E.g.: simple RISC ("scalar") pipeline



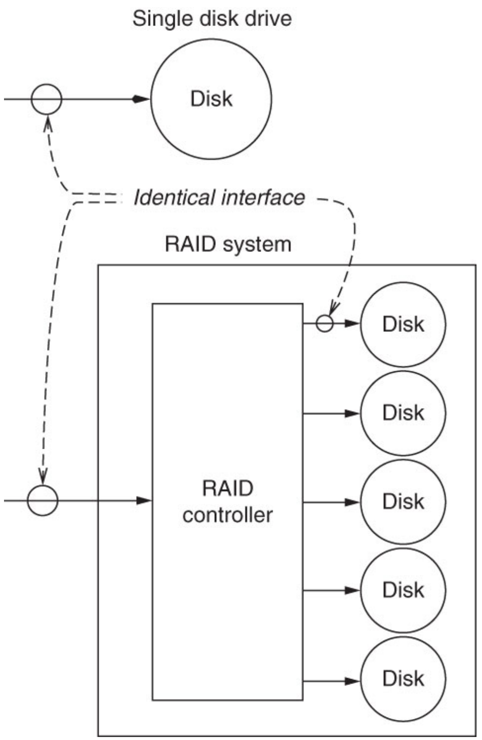
Challenges

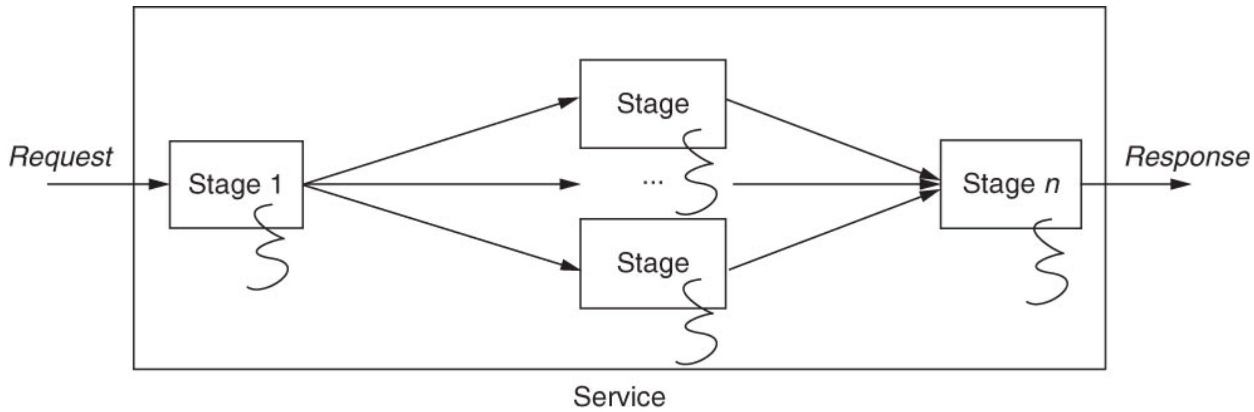
- Need multiple requests to keep pipeline busy
  - May push bottleneck to client
- Communication delay, queuing impose bound on throughput improvement



Interleaving

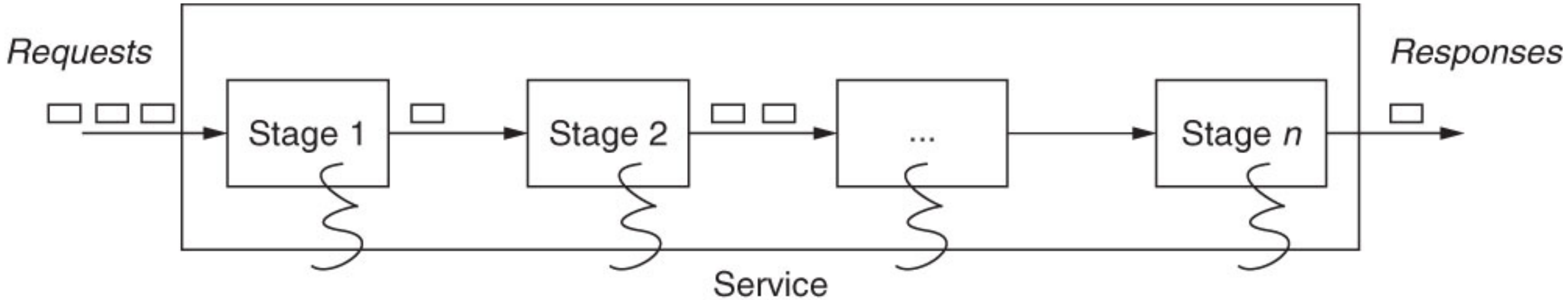
- Replicate bottleneck stage, interleave requests across replicas
  - E.g. RAID disk arrays





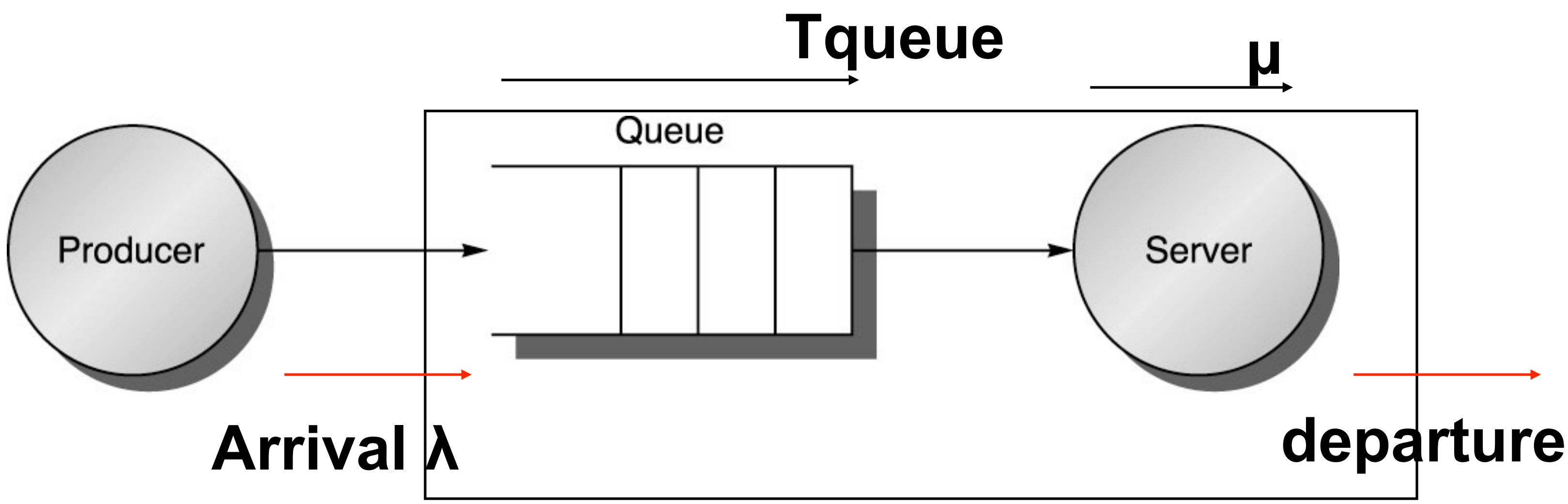
Queuing

- Asynchronous producer/consumer: requests may build up in a queue while other stages may be idle
  - E.g. short burst of requests that arrive at a faster rate than a stage can process



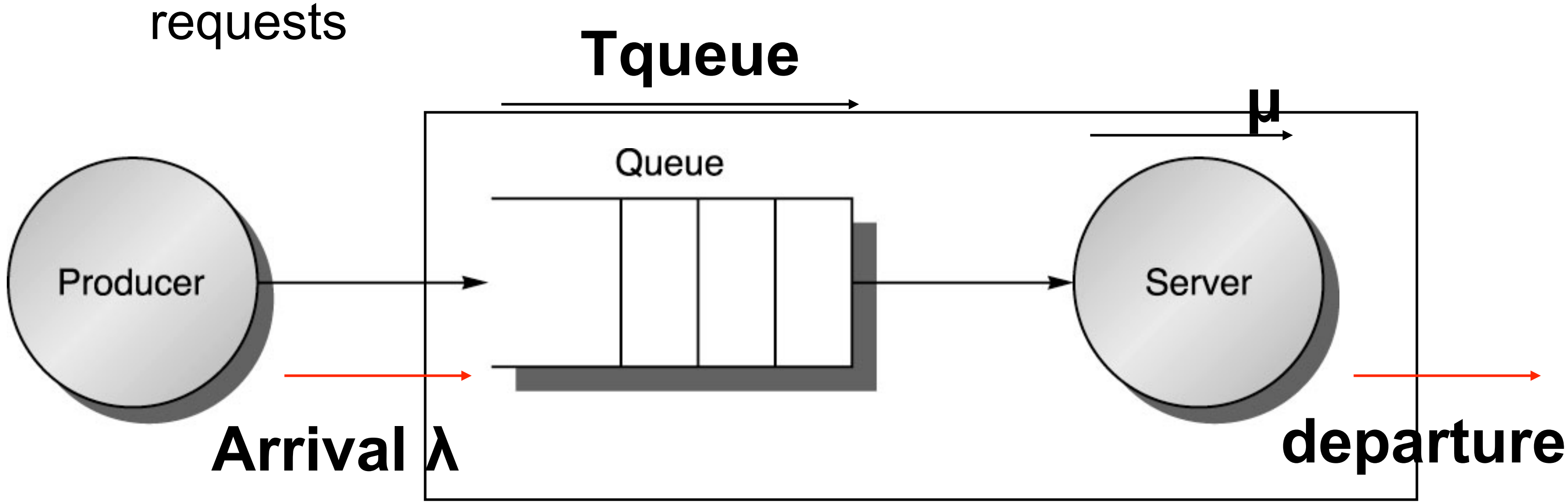
Queuing theory 101

- Simplest model based on random, memory-less arrival (λ); independent, exponentially distributed service times (μ); FIFO
  - Server utilization: $\rho = \mu * \lambda$
  - Average time in queue: Tqueue = $\mu * [\rho/(1 - \rho)]$
  - Average latency: Tqueue + μ
  - Length of queue: λ*Tqueue



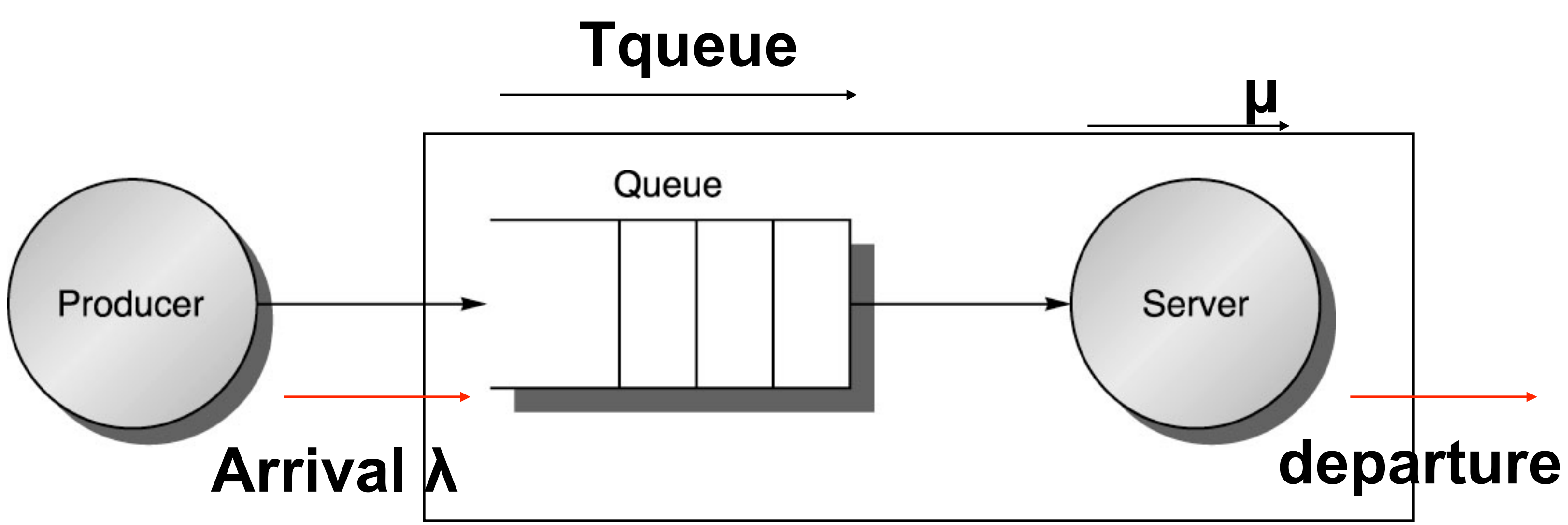
Example

- 20 disk requests per second (λ, exp. distrib)
- Average disk service time: 20ms (μ=0.02s)
  - Server utilization: ρ = 20*0.02 = 0.4 (40%)
  - Tqueue: μ*[ρ/(1- ρ)] = 13.33ms (0.0133s)
  - Average latency: 13.33ms + 20ms = 33.33ms
    - 40% of time on queue; avg. queue length = 0.27 requests



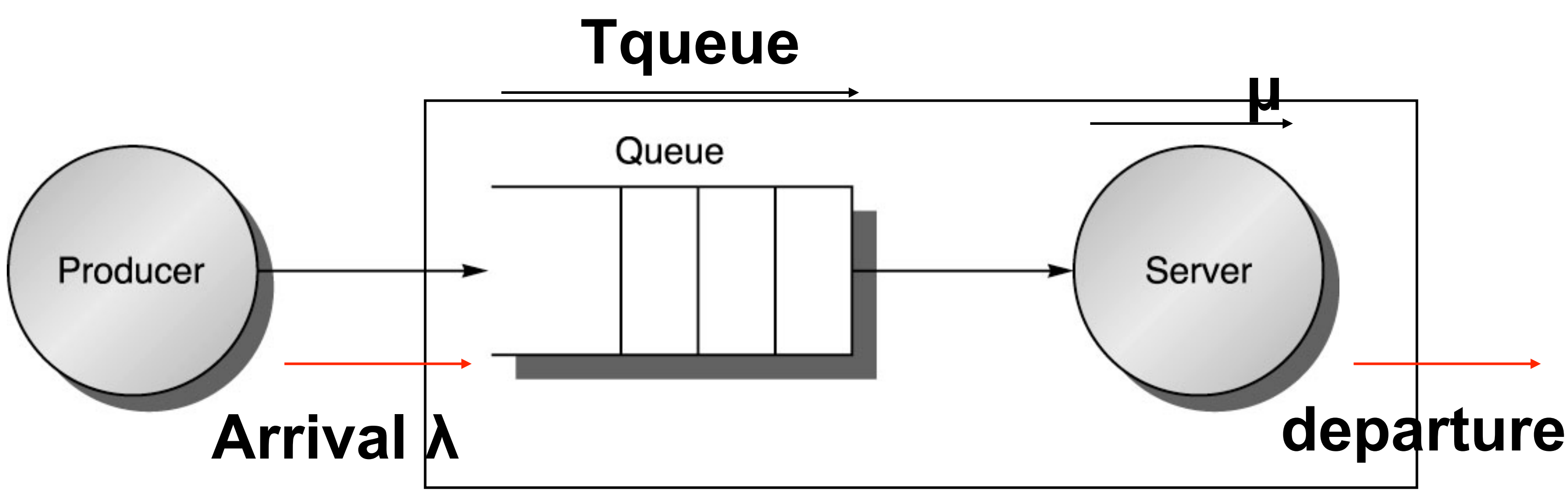
Example

- 40 disk requests per second (λ, exp. distrib)
- Average disk service time: 20ms (μ=0.02s)
  - Server utilization: ρ = 40*0.02 = 0.8 (80%)
  - Tqueue: μ*[ρ/(1- ρ)] = 80ms (0.08s)
  - Average latency: 80ms + 20ms = 100ms
    - 80% of time on queue; avg. queue length = 3.2 requests



Example

- 40 disk requests per second (λ, exp. distrib)
- Average disk service time: 10ms (μ=0.01s)
  - Server utilization: ρ = 40*0.01 = 0.4 (40%)
  - Tqueue: μ*[ρ/(1- ρ)] = 6.67ms (0.067s)
  - Average latency: 6.67ms + 10ms = 16.67ms
    - 40% of time on queue; avg. queue length = 2.7 requests



Tradeoff

- Having requests in queue is good for throughput
  - Avoid server to become idle
- Not having requests in queue is good for latency
  - Server can work on a request immediately
- Queuing theory, policies a topic of its own (offered next semester)

Overload

- Queuing delays grow without bound as arrival rate (offered load) approaches service rate (capacity)
  - $\rho=1$
- In some constrained systems, it may be possible to plan capacity so it matches offered load
- In many computer systems, it is not possible to precisely determine the offered load
  - E.g. Web site, flash crowds

Dealing with overload

- Short-term bursts are dealt with by queues
  - If overload time is comparable to service time, queue delays requests until a later time when offered load reduces
- If overload persists, queue grows indefinitely

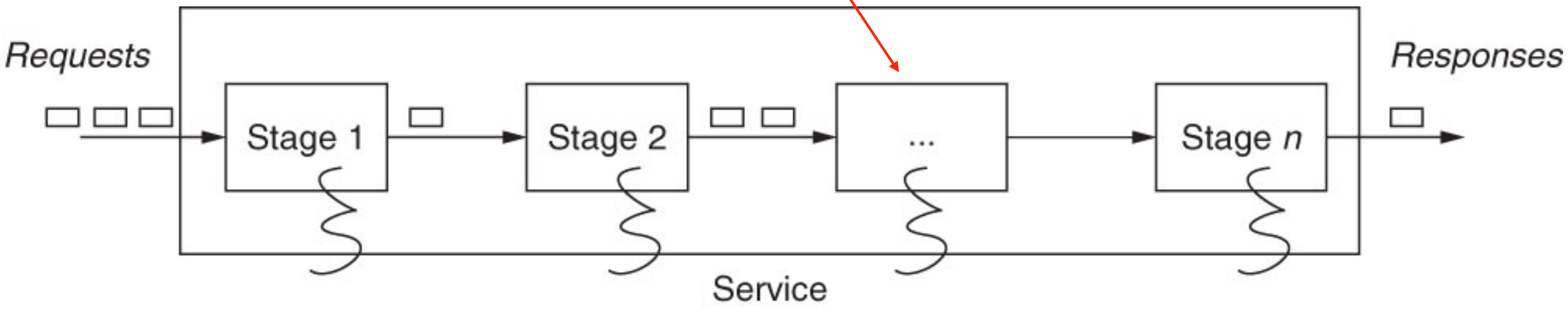
Dealing with overload

- Increase capacity of the system
  - E.g. faster processors; concurrency
- Shed load
  - Reduce/limit offered load

Increasing capacity

- Increasing capacity with dynamic provisioning
  - Interesting model: cloud computing – pay as you go

React dynamically when overloaded
E.g. spawn new servers

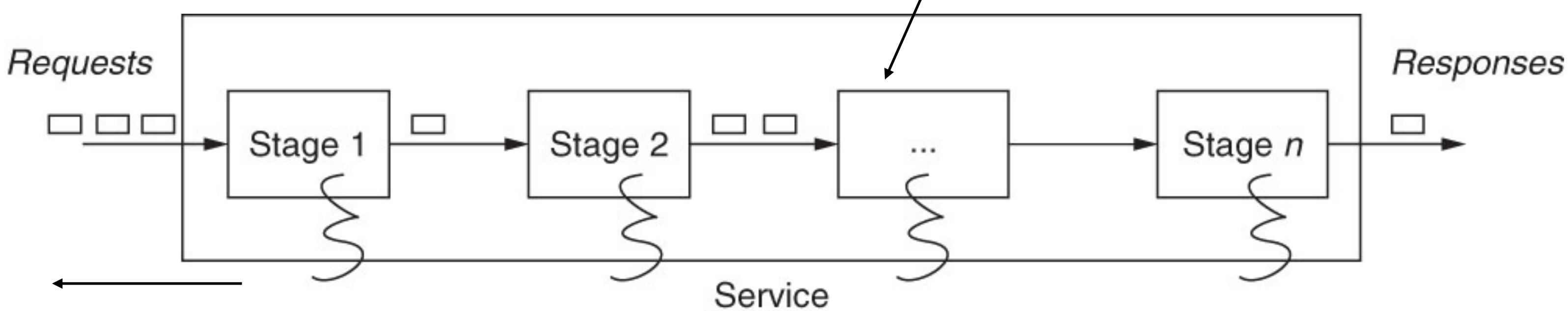


Shedding load

- Bounded buffers

Buffers may also
Fill up in earlier stages

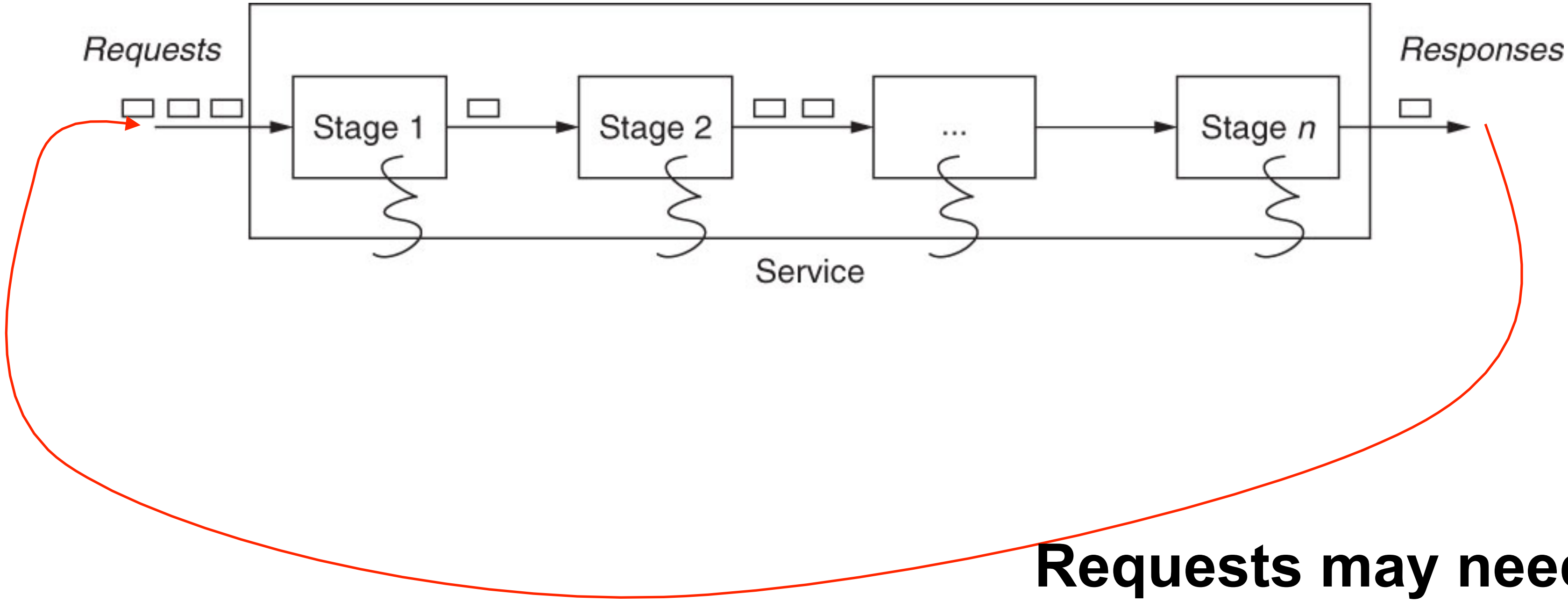
Buffer full;
Block incoming requests



Bottleneck may be pushed
to client

Shedding load

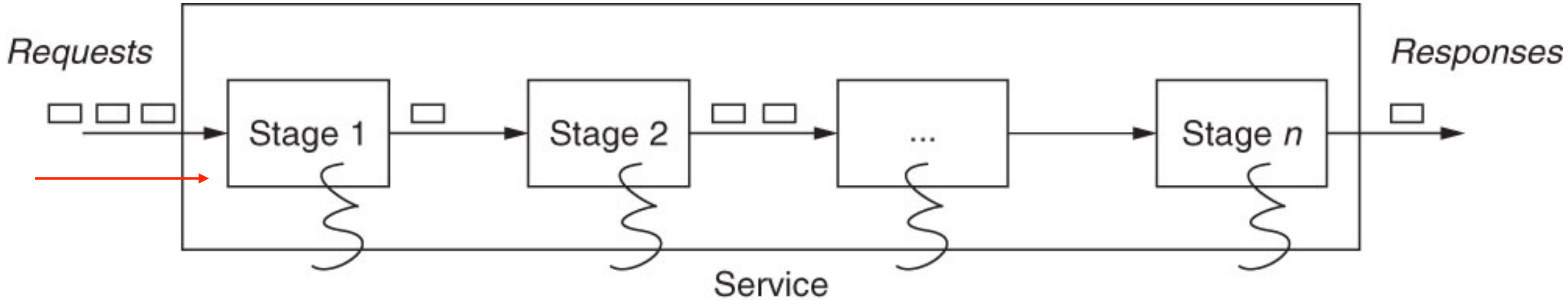
- Rate of requests may naturally be reduced due to nature of workload
  - E.g. single user, interactive



Requests may need Results – self-managing

Shedding load

- Source may continue with the requests
  - Place limits, deny requests
  - Reduce offered load when stage is overloaded – at the expense of quality of service
    - E.g. lowered frame rate, swapping virtual memory to disk



Admission control, quotas:
Limit outstanding requests

Batching

- Perform several operations as a group
  - When it is possible to avoid setup overhead by consolidating operations
- If the latency of a single request is a fixed setup time plus variable delay
  - Without batching: n*(f+v)
  - With batching: f + n*v
  - Example: combining several requests in a single message
    - Recall NFS version 4 discussion

Batching

- Additional benefits:
  - Coalescing requests
  - E.g.: subsequent write operations
    - Two writes to sub-sets of a disk block – combine sub-sets, single block write
    - Two writes to the same memory cell – avoid work by discarding the first write
    - Need to be mindful of interleaving reads; reads can also take data from the batch buffer
  - Reordering requests
    - If requests are independent and can be reordered, can make better scheduling decisions – e.g. disk arm scheduling

Dallying

- Delay a request on the chance that an operation may not be needed
- In combination with batching, delaying requests provides opportunities for more requests to be batched, and work to be reordered or absorbed
  - Potential to improve overall average latency at the expense of increasing the latency of some requests

Speculation

- Perform an operation in advance of a request, in the anticipation that the request will come through
  - Goal: deliver results with less latency
  - Hiding latency by starting earlier
  - May use resources that would otherwise be idle
- The challenge is, how to predict:
  - Which request is going to be issued in the future
    - If guessed wrong, work will be useless and may delay other useful work
  - When the request will be issued
    - If guess too late, also useless. Too early, the results may be dropped before they are used

Speculation

- Predictability is a function of the workload
  - If requests and their times are random, little opportunity for prediction
  - However, many workloads exhibit patterns that can be predictable to a degree sufficient to exploit this technique

Speculation

- Example: branch prediction
  - Crucial module of modern super-scalar processor pipelines
  - Two things can be predicted about branches quite accurately:
    - Is the branch taken or not taken?
    - What is the target address?
  - Think about the branch at the end of a loop that iterates 1000 times
  - Advanced branch predictors account for correlation with other branches

Speculation

- Example: pre-fetching
  - Given that workload accessed block “n” in the near past, it’s likely that it will access blocks “n+1”, “n+2” in the near future
    - Locality of references – memory, files
  - May go ahead and bring blocks “n+1”, “n +2” as soon as a request for block “n” is received

Speculation

- Example: sequences of instructions within a thread
  - Interpreter encounters a branch with hard to predict direction
  - Have two threads execute both “taken” and “not taken” paths
    - Pick the correct one when branch outcome is known
    - Discard the state generated by the incorrect one

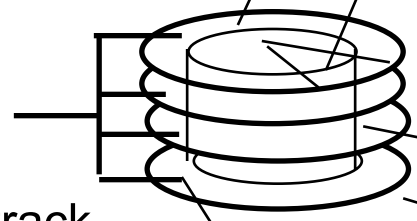
Challenges

- Batching, dallying, speculation – all introduce complexity
  - Introduce additional coordination of requests
    - Additional opportunities for undesirable situations to arise, e.g. race conditions
- Depending on workload, may degrade performance
  - E.g. aggressive pre-fetching, but access pattern is unpredictable, pre-fetched blocks may throw out useful blocks and cause contention on other stages

Example

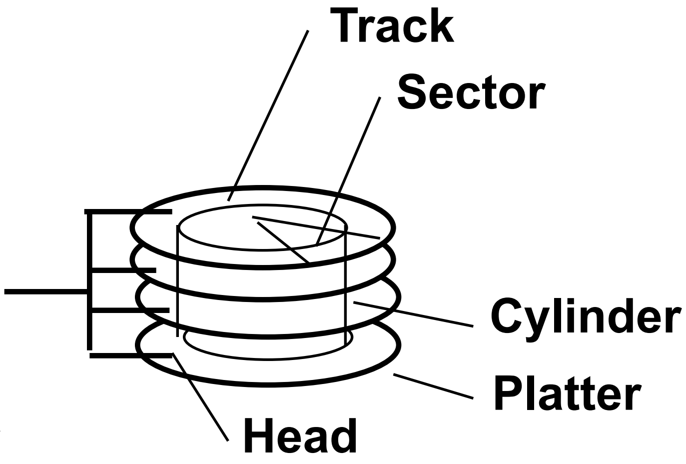
- Addressing I/O bottleneck of hard disks
  – Hard disks are very cost-effective in terms of price/bit
  – However, they are orders of magnitude slower than DRAM memory
    - Moving mechanical parts involved
  – Opportunities for designing for performance require understanding of workload and device characteristics

Magnetic Disk Characteristic



- Cylinder: all the tracks under the head at a given point on all surface
- Read/write data is a three-stage process:
  - Seek time: position the arm over the proper track
  - Rotational latency: wait for the desired sector to rotate under the read/write head
  - Transfer time: transfer a block of bits (sector) under the read-write head
- Average seek time:
  - Typically in the range of 3 ms to 14 ms
  - (Sum of the time for all possible seek) / (total # of possible seeks)
- Due to locality of disk reference, actual average seek time may:
  - Only be 25% to 33% of the advertised number

Magnetic Disk Characteristics



- Rotational Latency:
  - Most disks rotate at 5K-15K RPM
  - Approximately 4-12ms per revolution
  - An average latency to the desired information is halfway around the disk
- Transfer Time is a function of :
  - Transfer size (usually a sector): 512B-4KB / sector
  - Rotation speed (5K-15K RPM)
  - Recording density: typical diameter ranges from 2 to 3.5 in
  - Typical values: 30-80 MB per second
    - Caches near disk; higher bandwidth (320MB/s)

Example

- Consider the following parameters:
  - Average seek latency: 8ms
  - Rotation delay: 8.33ms (7200 RPM)
    - Average rotation latency: 4.17ms
  - Transfer rate our of disk media: 1.5MB per track, 120 tracks per second – up to 180MB/s
  - Transfer rate to memory – function of bus speed
    - IDE: 66MB/s would be bottleneck
    - Serial ATA: 3GB/s, disk mechanics the bottleneck

Random 4KB read

- Average latency:
  - Average seek time + average rotation latency + transmission delay
    - (ignoring queuing delays, bus latency)
  - 8ms + 4.17 ms + (4/(180*1024))*1000
  - 12.19ms
- Average throughput:
  - 4KB / 12.19ms = 336KB/s
- Opportunity: drive disk at peak transfer rate (180MB/s) instead of average random block rate

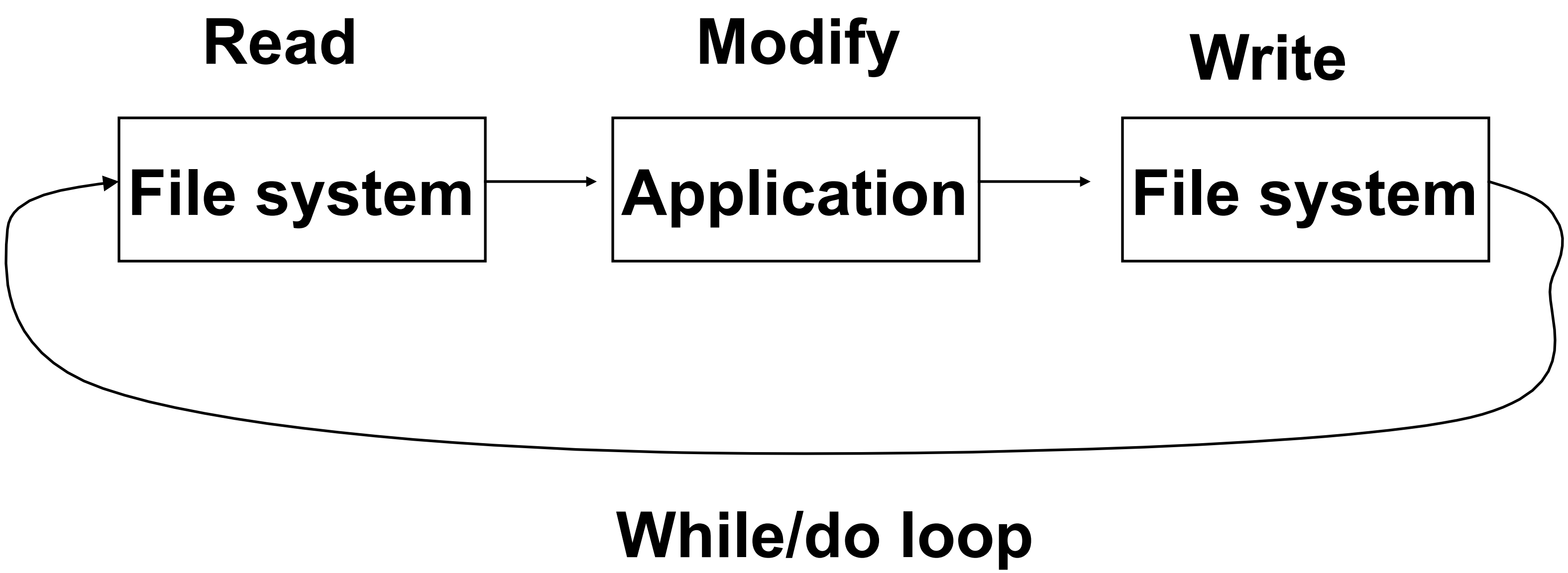
Example

- Common data processing pattern
  - Read file sequentially, process, write sequentially

```
in <- OPEN ("in", READ)
out <- OPEN ("out", WRITE)
while not ENDOFFILE (in) do
   block <- READ (in, 4096)
   block <- COMPUTE (block)  // assume 1ms
   WRITE (out, block, 4096)
CLOSE (in)
CLOSE (out)
```

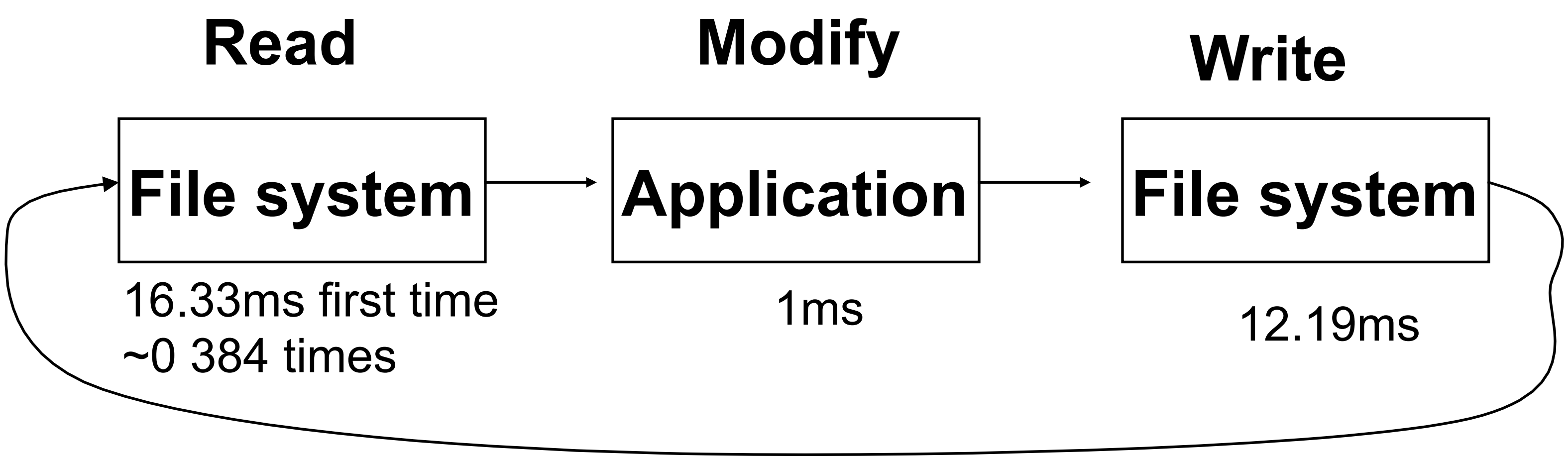
Pipeline

- Worst case: file system distributes blocks randomly across disk
  - 12.19ms + 1ms + 12.19ms = 25.38ms



Pipeline

- Improvement 1:
  - File system layer:
    - Distribute adjacent blocks sequentially along same track
    - *Speculation*: read pre-fetching of entire track
  - Now latency to read entire track
    - Average seek time + 1 rotational delay
    - 8ms + 8.33ms = 16.33ms
    - 1.5MB per track – sufficient for 384 loop iterations
      - Average time per iteration: (16.33 + 384*13.19)/384=13.23ms



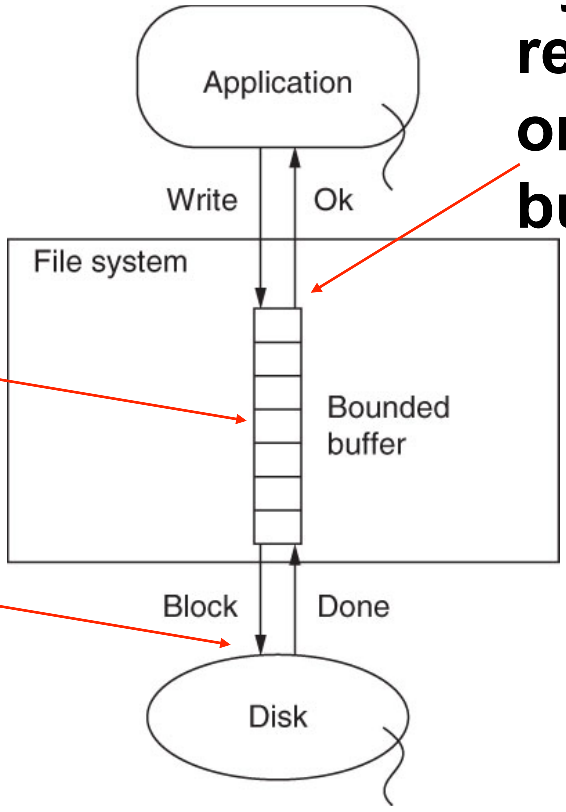
Pipeline

- Improvement 2:
  - Dallying and batching write requests using file system buffer

Syscall returns once in buffer

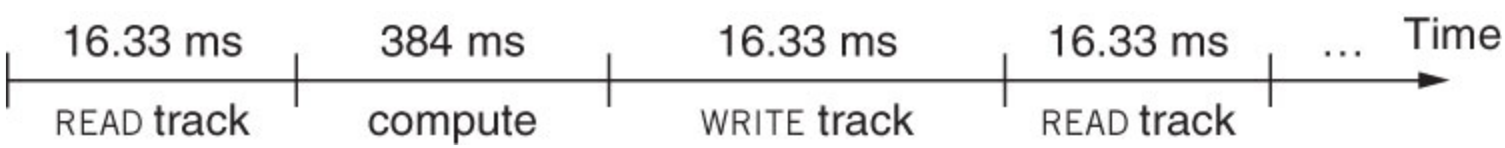
Delay until enough for a track

Write track all at once 384 blocks



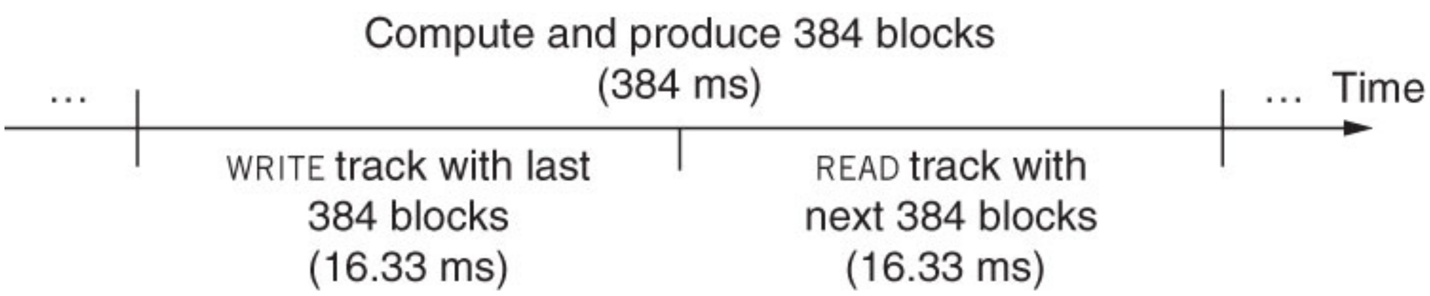
Improvement 2

- Now read and write track in one request, computation takes place 384 times without I/O bottleneck
- Average iteration latency:
  – (16.33+384+16.33)/384 = 1.09ms



Improvement 3

- File system pre-fetches next track as well
  – Only first READ in sequential path – subsequent READs overlap with computation
- File system buffer holds 2+ tracks
  – Fully overlap I/O with computation
  – Bottleneck is now computation time

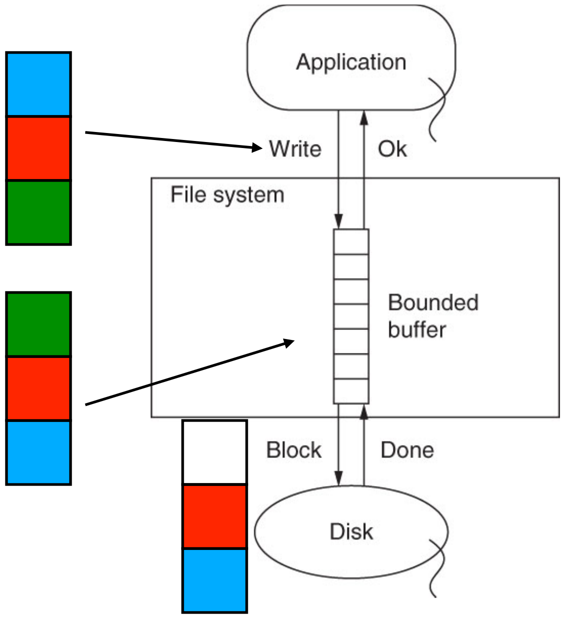


Discussion

- Assumption of a simple model of disk performance
  - Other factors include queuing, caching, disk layout/error maps
- Assumption of a single disk
  - RAID arrays provide concurrent access and additional opportunities for overlapping
- Emerging technologies can change performance parameters substantially
  - E.g. Flash SSD disks; uniform access times, but endurance for multiple writes is an issue

Discussion

- Reliability issues
  - Data in volatile storage, potentially with re-ordered requests
  - Crash recovery becomes more difficult



Discussion

- APIs for interacting with file system allow applications to force pending requests to stable storage
  - Linux: fsync(fd)
- Draining un-forced memory buffers to disk done transparently to applications based on buffer size, availability of idle cycles, timers
  - Linux: every x seconds, or # of “dirty” pages in memory buffers > y%; configurable
  - /proc/sys/vm/dirty_expire_centisecs (30s)
  - /proc/sys/vm/dirty_background_ratio (10%)

Reading

- Section 6.3