Nathan Tung

CS111 Sections of Chapter 6 Outline #15

Chapter 6, section 6.1 (pages 300-321), section 6.3.4 (pages 360-362)

- Performance: usually a result of a bottleneck (stage in computer system that takes too long on a task)
  - Solutions: exploit workload properties, concurrent executions, speculation, batching
- Designer for Performance
  - Tradeoffs between physical/technical limits (power, heat, chip size, algorithm)
  - Tradeoffs for device sharing with little overhead
  - **When in doubt, use brute force**
    - Better to use simple brute force than complex, badly defined algorithms
    - Direct corollary of d(technology)/dt curve
  - Keep it simple
- Performance Metrics
  - Capacity, Utilization, Overhead, and Useful Work
    - Capacity: measure of a system's size of resource amount
    - Utilization: percentage of capacity used for some request workload
    - Layered system
      - Layers below do overhead work, layers above do useful work
  - Latency: delay between change of input and output from system
    - Latency of pipelined operations is more than the sum of the individual ones
    - Because passing the requests causes more latency
  - Throughput: measure of rate of useful work for some request workload
    - Throughput of pipelined operations is less than sumo of the individual ones
    - Because passing the requests causes more overhead
    - Throughput = 1/Latency (only when serial; no pipeline direct relationship)
- A Systems Approach to Designing for Performance
  - To improve throughput, all stages need to be improved
  - Law of diminishing returns: don't focus so much on optimizing individual stages
  - Identify if performance enhancement is needed; identify performance bottleneck; predict enhancement; measure new implementation; repeat if needed
- Reducing Latency by Exploiting Workload Properties
  - Limits make reducing latency of certain algorithms (like getting from hash table) impossible
  - Use a fast path for most things, and a slow path for uncommon ones
  - **Design to optimize common case: use caches or multilevel memory**
- Reducing Latency using Concurrency
  - Not all computations are equal in amount of work
  - Also, there's overhead in passing the requests around, splitting them/merging them
- Improving Throughput: Concurrency
  - Hide latency by overlapping concurrent programs
    - **Instead of reducing latency, hide it**
      - Some things are intrinsically at their limit (speed of light)
  - Unfortunately, some stages are slower than others (queue builds up)
  - Several requests must be available in the first place
  - Interleaving requests is a solution; run n instances concurrently to different instances at full speed
- Queuing and Overload
  - When stage operates at max capacity, a queue of requests forms behind busy stage
  - Service time: time taken to process a request
  - Queuing theory tells us average delay will be 1/(1-p), where p is service utilization
  - Offered load: rate of arrival of service requests; it is overloaded when over capacity
  - Persisting overload can result in two choices
    - Increase system capacity
    - Shed load
  - Use a bounded buffer
  - Crude solution: put a quota on max number of outstanding requests

- Fighting bottlenecks
  - Batching, dallying, speculation
  - Batching: perform several requests as a group, reducing overhead from setup
    - Can possibly allow stage to avoid work; can reorder request processing to improve latency
  - Dallying: delay processing request in case it doesn't need to be done (or allow more batching)
    - AKA write absorption for writing operations (when request 1 is taken by request 2)
  - Speculation: performing operation in advance in case it's requested
    - Deliver results with less latency and less setup overhead
    - Use idle resources so extra operations aren't wasted
    - Or use busy resource to perform an operation
    - Usually used to execute the next (n+1) instruction
    - Sometimes, speculation can also increase load for later stages
- Challenges with Batching, Dallying, and Speculation
  - Introduce complexity; might need to coordinate references to shared variables if applicable
- An Example: The I/O Bottleneck
  - Mechanical problems in magnetic disk causes bottlenecks
  - Incommensurate scaling rule: seek/rotational latency improvements are lagging behind processors
  - Example: bits read from a disk
    - Mechanical limit: rate at which bits spin under disk heads on way to buffer
    - Electrical limit: rate at which I/O channel/bus transfers buffer data to computer
  - RAID has multiple disks, allowing interleaving read/write requests
  - Buffer without write-through can improve performance but reduces reliability and makes more complex
  - Cache introduction needs more constraints on coordination

- Case Study: Scheduling the Disk Arm
  - Thread scheduling has become less important due to processor improvements
  - Now the problem lies with disk arm scheduling
    - Goal: optimize overall throughput, not individual request delays
  - Order of processing requests
    - FIFO is not good
    - Instead, we can sort the processes based on track number
  - In practice, it's more complex; when new processes arrive, (how) do we reorder them?
  - Two options
    - Shortest seek first
    - Elevator algorithm
    - Many disk controllers use a combination of both of these
    - Might be smart to bound time limit for these algorithms to prevent "starvation"
      - Prevent back and forth processing between two much-closer tracks
        - Good for disk systems
        - Bad for people/elevators (use elevator algorithm)