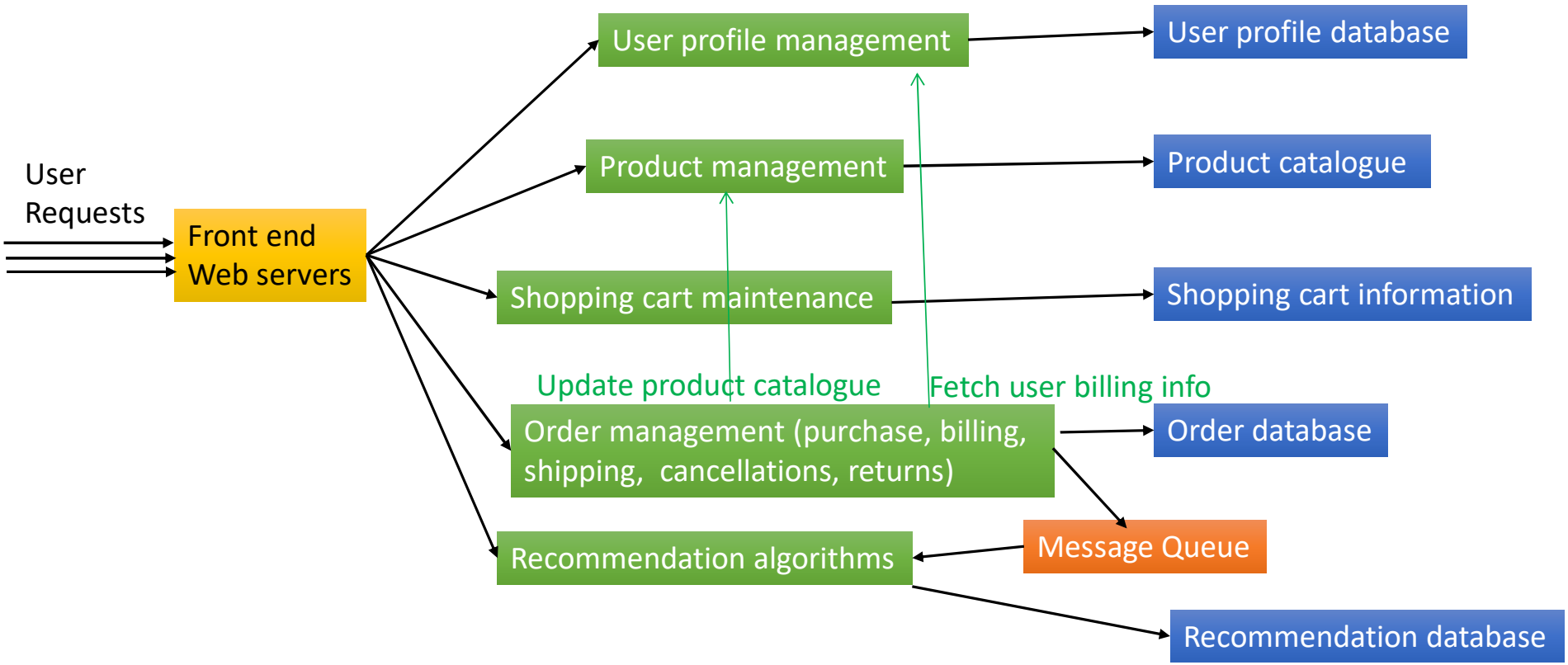
Performance Engineering

Mythili Vutukuru
CSE, IIT Bombay

Multi-tier systems

- Real-world computer systems are built as multi-tier applications
  - Multiple components/tiers distributed across several machines
- High-level architecture of a multi-tier application
  - Clients access applications hosted in organizations or public clouds
  - Front-end components (e.g., web servers) receive user requests, reply to user with responses, consult various application servers to build responses
  - App servers contain business logic to process different types of user requests
  - Application data is stored in several database servers in the backend
- Each application is typically a multi-threaded program, running on multiple cores

Example: e-commerce system



Performance: parameters and metrics

- Given a computer system, how to measure its performance?
- Input parameters on which performance depends / incoming load
  - Number of concurrent users/requests in the system
  - Rate (requests/sec) of incoming traffic
  - Mix of various types of requests (which require different amounts of work)
- Performance metrics / outputs measured
  - Throughput of the system in requests/sec (averaged over a time window)
  - Response time or latency (averaged, various percentiles)
  - Utilization of various resources at components (averaged)
  - Various kinds of errors and failures (counts)
- Load testing of a system: vary incoming load, measure performance

Performance bottleneck

- The performance of a system is limited by the slowest component in the system (bottleneck)
- Consider a web application servicing one type of requests
  - Front-end can serve 1000 req/s, app server can handle 5000 req/s, backend database can process 100 req/s
  - Max throughput of the system is 100 req/s (capacity)
  - Database component will be the performance bottleneck and will be fully occupied at peak input load (other components will not be as busy)
  - Database component takes approx. 1/100 seconds = 10 milliseconds to process each request (service demand)
  - Response time of system will be at least 10 millisecond + time needed at other components
- Sometimes, the network can also be a bottleneck, not any component
  - Maybe some router on path between clients and server can only forward 50 req/s

Running load test

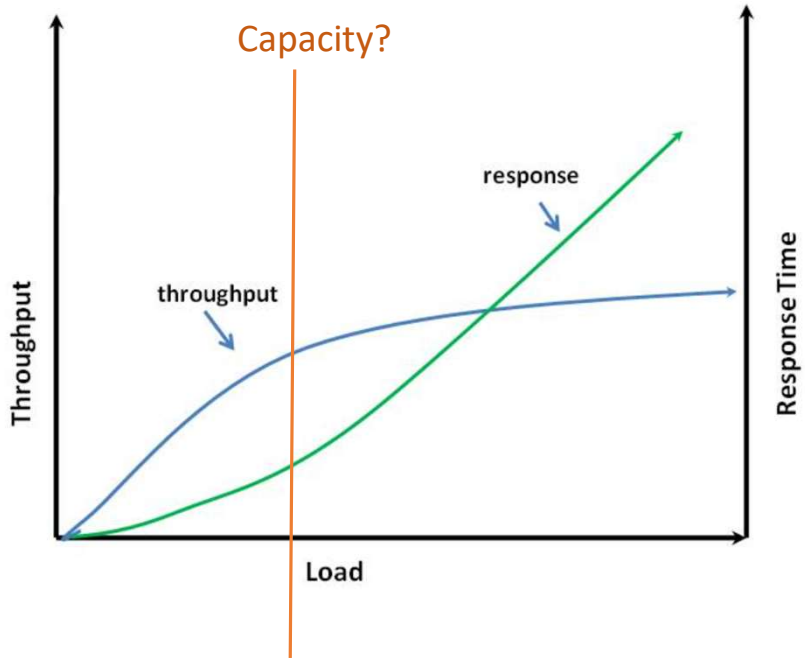
- The system under test (the entire system or one component) is sent a large number of requests from a load generator
  - Load generators can be software or hardware components that generate a large amount of synthetic traffic
- The input load is varied across multiple levels, and the output performance metrics of the system are measured for each input load
- How would you write a simple load generator?
  - A multi-threaded application, running on a large multicore machine, with each thread sending a continuous stream of requests?

What to expect in a load test?

- Consider web application with max throughput (capacity) 100 req/s
- Suppose incoming traffic into system is only 10 req/s
  - The system throughput is 10 req/s, no performance bottleneck
  - No component overwhelmed, quick (low) response time
- As incoming load approaches capacity, e.g., 90 req/s
  - All incoming requests are served, throughput is 90 req/s
  - Bottleneck component starts to get busy, queue builds up, responses take longer
- If incoming load exceeds system capacity, e.g., 200 req/s
  - Bottleneck component fully saturated, max throughput (capacity) achieved, throughput flattens (cannot increase any more)
  - Response times are higher due to queueing delays, continue to increase with increasing load

Identifying system capacity

- System developers perform load tests of their systems to generate graphs of performance metrics vs input load
- Graphs of throughput and latency help to identify system capacity
- As input load approaches capacity, throughput flattens out and latency increases sharply
- Ideal operating point of a system: just below max capacity or saturation
  - Close to max throughput, not too long response times, no errors



How to improve system capacity?

- When system is running at capacity or saturation, some hardware resource at bottleneck is fully (100%) utilized
  - E.g., all CPU cores are fully busy with no spare CPU cycles
  - E.g., hard disk is running at full capacity performing reads/writes
- How to improve capacity?
  - Increase hardware resources at bottleneck component
  - Or, optimize system to use hardware resources more efficiently
- Sometimes, bottleneck due to software issues only (no hardware resource is fully utilized)
  - E.g., maximum number of file descriptors opened by process, cannot open any more
  - E.g., threads wasting time waiting for locks, even though CPU is free
  - E.g., not enough threads to do the work, even though cores are free
  - Such issues can be fixed by rewriting code or tuning system parameters

Monitoring usage of hardware resources

- At saturation, performance of bottleneck component cannot increase further because some hardware resource is fully utilized
- Monitor usage of all hardware resources using various tools:
  - Tools to monitor CPU utilization, what fraction of CPU cycles in each CPU core are fully utilized and by which processes, e.g., “top” in Linux.
  - Tools to monitor memory usage, what fraction of main memory is free and what fraction is used by user/kernel, e.g., “free” in Linux
  - Tools to monitor memory bandwidth usage, how much of the memory bus bandwidth is utilized by ongoing memory accesses, for local and NUMA memory
  - Tools to monitor utilization of various I/O devices like disk and network card, and rate of data transfer to/from device, e.g., “iostat” in Linux
- Once we identify which hardware resource is saturated, identify why the hardware resource is being used so heavily: profiling tools

Profiling tools

- Performance profiling software (e.g., perf, oprofile) help us identify the root cause of performance issues
- Profilers monitor the execution of a program and help to:
  - Count various hardware events (e.g., cache misses) and software events (e.g., page faults) occurring in the system
  - Attribute events to parts of program code responsible for events
  - Understand how CPU time is spent in executing various user/system functions
  - Understand how various hardware resources are utilized
- By analyzing profiler output, we can identify
  - Parts of code that are performing inefficiently
  - Hardware or software events that contribute to poor performance
- Profiling is a starting point for performance optimization

Optimizing CPU performance

- If all CPU cores are saturated by application, identify which parts of the code are leading to high CPU usage via profiling, and optimize
  - If user functions/libraries using more CPU than required, optimize/rewrite the code to be more efficient, or use high-performance libraries
- If excessive CPU usage by kernel code, optimize where possible, e.g.,
  - High CPU usage due to frequent interrupt handling with high speed network card → move to a more optimized device driver which generates fewer interrupts or split interrupt processing to multiple CPU cores
  - Upgrade to better file systems to reduce file I/O overhead
  - Tune CPU scheduler parameters to minimize context switching overhead

Using multiple cores better

- Having multiple threads helps utilize multiple cores better, in most cases
- However, threads cannot execute in parallel all the times
  - Example: only one thread at a time can execute critical section
- Amdahl's law: estimate performance gains due to parallelism
  - Let T1 = time required to perform a task on one CPU core
  - Let Tp = time required to perform task when running in parallel on "p" cores
  - Let α = fraction of task that can be parallelized
  - We have $Tp = (\alpha * T1 / p) + (1- \alpha) * T1$
  - Speedup due to using multiple cores = T1/Tp (ideally p if $\alpha=1$)
  - For large values of p, speedup approx. $1/(1- \alpha)$
  - If α is small, speedup is small, poor multicore scalability
- Minimize code that cannot be parallelized, for better multicore scaling

Optimizing memory/cache usage

- If memory usage too high, system performance may be sub-optimal due to thrashing (too many page faults, excessive swapping to disk)
  - Reduce memory foot-print of application, so avoid swapping
  - Improve locality of reference within program, so that working set size (amount of memory being actively used) is low
  - Using huge pages (larger page size) improves TLB hit rates
- If poor cache hit rates and high memory bandwidth utilization (even though memory is free) CPU wasted in waiting for memory access
  - Write code in a way that optimizes cache usage, and reduces need to fetch data from main memory

Recap: CPU caches

- CPU fetches instructions/data from memory of process
  - Faster memory access implies faster application performance
- First step in a memory access: check CPU caches if data is present
  - CPU caches store recently accessed memory in 64 byte cache lines
  - Uses locality of reference to avoid expensive main memory access
- Multiple levels of cache, some private, some common across cores
  - Memory location is cached in the private cache of one core C0, another core C1 also wishes to access the same memory contents → cache line is shared across cores via cache coherence mechanism
  - Cache coherence protocol ensures consistent view of memory across cores
  - But cache coherence mechanisms add overhead to memory access

Optimizing cache usage (1)

- Align data structures to cache lines using language library primitives or compiler hints
- Store frequently accessed variables together in the same 64 byte cache line
- Write code with lower working set size (frequently accessed code sections or data structures) that fits in CPU caches
- Write code to increase locality of reference (access data that is already in cache as far as possible)
  - Example: access matrix along rows rather than along columns
  - Example: merge two for-loops that loop over same array

Optimizing cache usage (2)

- When accessing data from multiple cores, avoid cross-core cache coherence traffic to make cache access faster
- Threads of program running on separate cores should access data in separate cache lines as far as possible
  - True sharing: two threads read same memory address from separate cores
  - False sharing: two threads read separate memory addresses, but both locations are on the same cache line
  - Both cause cache line to bounce across cores, false sharing to be avoided
- Avoid shared data and lock contention between threads as far as possible
  - Shared lock variable accessed from multiple cores, cache line bounces across cores
  - Use newer techniques (optimized locks, lockfree data structures)

More on optimizing memory

- DRAM allows random access of memory (jump to any address), but sequential access of memory is better for performance
  - CPU prefetcher predicts which memory will be accessed next (estimates stride length of access) and fetches it into cache
- Pre-allocation of memory is better than dynamic allocation via malloc
  - General purpose malloc that does variable sized allocation can be slow
- Custom memory allocators better than general purpose allocators
  - Slab allocators are better when malloc is in a few fixed sizes
  - Store data in memory-mapped anonymous pages instead of heap
- Avoid copying memory contents unnecessarily
  - Memory mapping a file avoids copying file data from kernel memory to user buffers

Other optimizations (beyond CPU, memory)

- Compiler optimization turned on, to enable generation of optimized binary application code
  - Advanced techniques used to generate efficient machine code in compilers
- Some parts of application code can be offloaded to hardware accelerators to run quicker
  - Graphics Processing Units (GPUs) are used to run video processing and rendering algorithms efficiently
- When I/O is bottleneck, consider caching result of I/O in storage that can be accessed faster, for future use
- If nothing else works, add more hardware resources to increase performance and handle incoming load
  - Vertical scaling: add more hardware resources to the bottleneck machine
  - Horizontal scaling: add additional machines to handle extra load

Software bottlenecks

- Sometimes, system performance bottleneck is not hardware
  - Adding more resources will not help fix the problem
- How to identify such cases?
  - Throughput flattens out even when no resource is fully utilized
  - Performance does not increase by adding more resources
- Software bottlenecks harder to identify, fixed by carefully tuning system parameters
- Example: process hits maximum limit on file descriptors, so cannot handle new requests even though there are enough resources
- Other parameters: number of threads, queue sizes

Optimum number of threads in thread pool

- Consider a multithreaded server with a thread pool of workers
  - Master thread places new clients/requests in shared queue
  - Worker threads in thread pool retrieve requests one by one and process
  - Worker thread can block multiple times to service one client
  - Too low number of worker threads cannot efficiently use CPU cores
- Min no. of threads in thread pool to fully utilize a single CPU?
  - Suppose each worker thread performs 0.01 seconds of computation on CPU to process client request (service demand) and 1 second waiting for I/O before running on CPU again (turnaround time)
  - Optimum number of threads = turnaround time / service demand = 100
  - What if fewer than 100 threads? 50 threads will lead to 50% CPU utili, but still throughput flattens out

Blocking vs event-driven I/O

- So far, we have assumed thread blocks while handling I/O request, so multiple threads needed to handle multiple requests concurrently, and fully utilize CPU
- Alternate way of handling concurrent requests – asynchronous event-based I/O APIs
  - E.g., make a system call that starts the I/O and returns immediately, notifies later when request is complete
  - E.g., if each request takes 0.01 sec and event-driven APIs used, then a single thread can handle 100 req/s on a single core
- Event-driven APIs very popular for networking applications
  - One way to build a networking app is assign a request/socket to each thread, let thread block on reads for that socket
  - Another way is to handle all sockets in a single thread via event-driven APIs like epoll

Sizing buffers, queues in system

- Many buffers in a system, e.g., buffer to hold requests between threads in a pipeline, buffer to hold incoming data from I/O devices
  - Essential to handle bursts of data coming into system
  - If buffer size too low, threads/processes may not have enough work
- Little's law (famous queueing theory formula): N = R * W, where
  - N = expected number of requests being served in the system
  - R = rate of arrival of requests
  - W = average time spent by a request in the system (time to process request + time spent waiting in various queues)
- Very general principle, useful in many scenarios
  - Requests arrive at 100 req/s, each worker thread takes 2 seconds for processing a request, then we need a buffer of at least 200 requests between master and worker
  - What if buffer size < 200? Some requests have to be dropped while all threads are busy, threads may not have work when they are free

Throughput vs response time

- Response time of the system is sum of time spent at each component
- Time at each component is service demand (processing time) + queueing delay (waiting time)
- Throughput of the entire system is determined by service demand of slowest component (bottleneck)
- Service demands of non-bottleneck components impact response time
- What if you optimize and reduce service demand of a non-bottleneck component? Throughput may not change, but response time will improve
- Most performance engineering starts with optimizing the bottleneck
  - Once the slowest component is optimized, bottleneck shifts elsewhere!
  - Performance engineering is an iterative process

The performance engineering workflow

- Build a multi-tier system based on functional requirements
- Perform a load test, identify capacity and bottleneck component
  - If hardware bottleneck, try to optimize hardware resource usage
  - If software bottleneck, tune system parameters (threads, buffers, …)
- If bottleneck capacity improves, overall system performance is better, bottleneck may shift elsewhere
- Repeat the process until satisfied with overall system performance
- Other ways to improve system performance: scaling (adding more replicas of a component), caching, …
- What beyond performance engineering? Reliability, fault tolerance