Design and Engineering of Computer Systems

Lecture 33:
Performance profiling and optimization

Mythili Vutukuru
IIT Bombay
Performance engineering

• Story so far this week:
  • Understanding performance: parameters, metrics
  • How to run a load test, types of load tests
  • Back-of-the-envelope calculations for sanity checking results
  • Tune system parameters to optimize hardware resource usage

• What after load testing and performance measurement?
  • If system performance can handle expected incoming load, do nothing, else optimize
  • Monitor utilization of hardware resources to identify bottleneck resource
  • Profiling tools help to identify root cause of high resource utilization
  • Apply various techniques to fix root cause and optimize performance
  • Note that improving performance of one component may shift bottleneck elsewhere; cannot fully eliminate all performance bottlenecks
  • Performance engineering is iterative process until performance matches expected incoming load to a system
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.
Performance profiling

- Performance profiling software (e.g., perf, oprofile, PCM tools) 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.
Profiling: Statistics of events

• Profiling software collects statistics of various hardware and software events in the system when an application is executing
  • Counts per-thread, per-process, per-CPU, or system-wide

• Hardware events (typically exposed via special CPU registers)
  • Number of CPU cycles (default event)
  • Instructions executed per CPU cycle (indicates CPU efficiency)
  • Cache misses for various levels of CPU caches
  • TLB misses, other events like cache pre-fetch misses, ...

• Software events (typically maintained by OS)
  • Page faults, context switches, ...
Profiling: Attributing events to specific code

• In addition to counting events, profilers can also attribute event to specific portions of the code, by inspecting PC when event occurred
  • Not possible to do this every time event occurs, too much overhead
  • Such information can only be exposed on some subset of events

• Sampling: for some subset of events, information about the code responsible for causing the event is stored
  • By sampling PC value periodically, we can find out which parts of code is consuming what fraction of CPU cycles
  • By sampling PC value during cache misses, we can know which parts of code are responsible for poor cache performance

• Profilers translate from PC to function name for easy readability

• End result of profiling: identify which parts of application code to optimize
Performance optimization techniques (1)

- If all CPU cores are saturated by application, identify which parts of the code are leading to high CPU usage via profiling, and then 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 (e.g., NAPI drivers), or split interrupt processing to multiple CPU cores (e.g., using RSS)
  - Upgrade to better file systems to reduce file I/O overhead
  - Tune CPU scheduler parameters to minimize context switching overhead
  - Better memory allocators to avoid dynamic memory allocation overhead
Performance optimization techniques (2)

• If memory usage too high, system performance may be sub-optimal due to thrashing (too many page faults, excessive swapping to disk)
  • Reduce size of in-memory data structures in application
  • Improve locality of reference within program, so that working set size (amount of memory being actively used) is low

• If poor cache hit rates and high memory bandwidth utilization, CPU cycles wasted in waiting for memory access
  • Design frequently used data structures to fit within CPU caches
  • Write application code in a way that maximizes locality of reference (spatial, temporal) and improves cache hit rates, TLB hit rates
  • Sequential access of memory and compiler hints to improve performance of hardware pre-fetchers
Performance optimization techniques (3)

- 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
Summary

• In this lecture
  • How to use profilers to understand performance bottlenecks
  • Techniques to optimize performance

• Programming exercise: install a profiler (e.g., perf). Use it to profile a simple program you have written which has a lot of CPU computation. Study the profiler’s output to see if you can identify the bottleneck.