Introduction

Maintainable - Widely Applicable - Fast

QoS Objectives: Statistical properties of a metric that shall hold for system SLAs: Legal contracts specfiying QoS objectives and penalties for violations

Specific, Measurable, Acceptable, Realisable, Thorough

Benchmarking: Get sys into predefined state, perform series of operations while measuring relevant perf metrics

Batch benchmarking: Program has entire batch from the start, no generator perf (throughput)

Interactive benchmarking: Generator works piece by piece (randomly), gen at least as fast as sys under eval (latency)

Sys params - do not change while sys runs (caches, etc)

Workload params - might change (users, avail. mem.)

Util - percentage of resource that is used to perform service. Bottleneck - resource with highest util

Critical path - Sequential part of code. Hot path - Takes most time

Parameter tuning - vector in param space that mins resource consumption or maxes perf metric.

Analytical models - (stateful or stateless) allow what-if analysis. Much faster than above (no real-time tests/feedback).

Simulation - single observed run of stateful model

Non temporal write - writes straight to mem (no cache)

Perf Tracing & Profiling

Events - change of sys state

Simple - exec instr., load addr., clock tick, etc.

Complex - cache line eviction, misspeculated instr. aborted

Event payload - pre-aggregation (additional data about event)

Event generator - observes changes, usually part of runtime (online).

Consumer - processes events (off(process later)/online)

Tracing - Complete log of every state the system has been in (totally ordered events). Call stack tracing expensive (walk stack, chase pointers, stack might be more than L1 cache line)

Pertubation - Degree perf of a sys changes when analysed. Reduce fidelity (how many events recorded) by sampling. Small functions may not be sampled, but expensive ones will be more often.

Time based sampling - hardware timer, inaccurate, non-deterministic, noisy, easy to interpret.

Event based - Quantization errors/biases (coarser granuality of system, as events are continuous (not discrete) time)

Indirect tracing - Trace dominating events (control flow instr. dominate non-control flow)

Profiling - Characterisation of sys in terms of resources it spends in certain states.

Profile - aggregate over events of a specific metric (info is lost)

Event sources (gen and consumer) - must be detailed, accurate, little pertubation.

Instrumentation - augment program with event logging code (flexible, high overhead). Manual - write to logfile. Auto - compiler supported, less effort and control. Binary - inserts probes into binary/bytecode (JIT)

Pipeline bubble - pipeline stall due to hazard

Microarchitecture usage - percentage of pipeline slots (max rate μarch can accept μops) used when profiling.

Retiring - % of μops successfully exec and retired.

Frontend bound - % of pipeline slots where no μops where delivered. Instr. cache miss, branch mispredict. Lots of branching.

Backend bound - mem bound: waiting on data fetch (cache miss). core bound: data dependencies, exec unit contention (FLops), resource saturation, wrong prefetching.

Bad spec - % of μops not retired.

Allocation stall - no execution units avail

Modelling - Assume input follows uniform without correlation, no sys noise.

Numerical Model - Describe observed behaviour (interpolation). Easy to get if system available, generalises poorly (limited accuracy/interpretability)

Analytical Model - Formal characterisation of the relationship between params and perf metrics. Need to define system params and varied params.

Access Pattern Algebra:

R.n - length (number of stored tuples)

R.w - width (size of tuple in words (8B)) (stride)

$||R||$ - size of region (R.n * R.w) (size of array)

u - number of words read in each access

Random access (rep. access): r - number of accesses

$P1 \bigoplus P2$: seq access - entire p1 then entire p2.

$P1 \bigodot P2$: interleaved access of p1 and p2.

s_trav(R.w = stride size, u = words read in each access, R.n = length of array)

$\bigodot$ rr_acc(R.w = stride size (words in datatype), u, R.n = length of array, r = number of accesses for this array)

r_trav (no rep. access): Useful for data intensive applications.

Hash table write - r_trav. Hash table read - rr_acc. RW input/output - s_trav.

Heap traversal - s_trav(R.w=1, u=1, R.n=vec1Size)$\bigodot$rr_acc(R.w=1, u=1, R.n=vec2Size, r = vec1Size*log(vec2Size)). vec1 could be 1 int.

Pairs in two arrays - n(n + 1)/2

Limitations - Random may be sequential repeated. Cache/reg locality. Repeated identical access (can unroll).

Stateful modelling - discrete Markov chains.

Branch misprediction rate - P(pred_taken) * P(act_not_taken) + (P(pred_not_taken) * P(act_taken)

Efficient Code

Control hazard - data-dependent changes in control flow (branches)

Structural hazard - lack of execution resources

Data hazard - operands (i.e. data) not available on time

False sharing - cache lines invalid ping-pong.

Superscalar - more than one instr. in pipeline stage (batches)

Factors - speculation, superscaler, out-of-order, SIMD/VLIW

Out-of-order addresses data / structural, leaves control. Speculative addresses some control.

VLIW - single instr, different ops on multiple/same data (compiler controlled superscaler)

Runtime predictable: no code eval in critical path, eval early

Partial eval - JIT, function inlining, template metaprogramming, const eval.

Const. eval - can't do if static var set somewhere else

Use bit-shifts where possible for special cases (metaprogramming)

Predicate instructions - branch free code, execute anyway and set back to original if wrong.

Use compiler intrinsics to use SIMD/VLIW. Keep values in vector registers (reg to reg transfers expensive).

Capacity miss - value accessed before.

Compulsory miss - not accessed before.

Mem bound - only data hazard stalls.

Mem bus fully utilised - memory bandwidth bound, otherwise memory latency bound.

Software prefetching - intrinsic. Cache line util - struct of arrays. Capacity misses (thrashing) - tile data. False sharing - pad cache line.

Multicore & Parallelism

Data Level Parallelism - Vector instr.

Instruction Level - Superscaler OOO Processor.

Task Level - GPUs, POSIX threads (programmer manually extracts).

Parallelism - Multiple cores. Concurrency - Interleaved execution. Parallelism $\implies$ Concurrency

Amdahl's Law - Speedup(Frac of program optimised [p], Improvement [s]) = 1/((1-p) + p/s)

Increasing core count has diminishing returns (yield of cores, silicon)

Threads share memory within a process

MSI cache coherency - modified, shared, invalid, (locked). Coherency not sufficient - load / store race condition, need atomics.

CAS - if (*address == old_val) then *address = new_val

Critical sections - atomicity between instr., guard related sections w lock.

Related CS - must enter/exit using same atomic var. Unrelated - should use different atomic var. Concurrency error - Thread rescheduled inside CS. Parallelism error - cache coherency not caught up.

Semaphore - n threads inside critical section. Shared lock - multiple threads in shared mode, one thread in exclusive mode. Condition var - one or mode threads wait until notified a condition is true. Barrier - wait until n threads are waiting in barrier, then unlock

User-level lock - CAS signal when acquired, otherwise loop (cycle waste or thread starved).

Kernel-level lock - Sys call to block / wake up thread.

Hybrid - Loop user-level for x, then block at kernel level (futex).

False sharing - parallel access to different vars on same cache line (coherency ping-pong).

On Demand - threads created when task arrives, destroy on complete.

Fork/Join - Ideal for recursive algos, not suitable for I/O bound tasks. Overhead for dividing and combining. Split until tasks are async.

Work Dispatching - tasks from central queue. Idle threads when queue empty. Thundering herd. $\iff$ CPUs

Work Stealing - every thread has its own task queue. Steal from other queues when idle. Reduced contention as threads use their own queue.

Thread streaming - Job queue for each function.

SEDA - Reactive thread pools for event-driven stages. Modular and scalable, but stages across threads bad (communication vs cache locality).

Multi-processing - explicit (expensive) communication (serialise/deserialise, memcopy overheads, sockets use network stack, pipes/shared-mem are local-only)

Tools, Patterns, Algos

Non-blocking: Thread suspension doesn't affect others (uses atomics)

Non-blocking + lock-free: At least one thread is guaranteed to progress

Non-blocking + wait-free: Every thread is.

Lock-free stack: atomics, CAS and retry mechanisms.

ABA problem: CAS works because A is the same in cache when stack changed lower down. Fix with node version tagging (CAS checks context), hazard-pointers or epoch-based reclamation (once node removed, not recycled until certain no threads are accessing).

System Interfaces

System calls are expensive - switch to kernel mode (save and sanitise) and back.

Buffered IPC - Buffer where messages stored and retrieved. Bulk transfer and async, but data could be stale and buffer could over/underflow.

Unbuffered IPC - Messaged directly to receiver address space. Sender blocked until receives ack. Receiver the bottleneck, both processes need to be scheduled to run.

Partial Walk Caches - Contain info about what subset of page tables the page is in - no full walk. PWC for each subset (3,2,1 or 2,1 or 1)

Memory translation consistency - translation caches consistent on page downgrade (read-write to read-only, unmap, TLB shootdown). Avoid with async syscalls and HW acceleration (only invalidate the relevant cores).

CPUs connected through extensible bus: Pin threads to CPUs (pthread_setaffinity) and access close mem (set_mempolicy on NUMA systems)

VM syscalls are trapped and emulated. Can have SMT seperating VM/hypervisor contexts.

VM Mem Translation: Nested page tables for each virtual PT (GVA - GPT - GPA - HPT - HPA). Shadow PT (ro) (needs VM exit for updates to shadow PT, syncs on page faults, but GVA - SPT (hypervisor) - HPA). Or use PWC.

MMIO - Device has range of physical addresses at boot time. Device instr looks like mem access.

Circular queues - Continuous data transfers (wrap-around). Also req/resp buffers.

Base address reg (BAR) - address ranges for devices

I/O Controller - connects devices and CPUs.

Interrupt-driven model: MMIO writes to program device, which r/w req/resp buffers. Waits for interrupt and MMIO reads result.

Polling model: Same as above, but polls res. Faster than interrupt if poll short. Can use hybrid.

PCIe - interconnect between MMIO and device (accessed with translated PCIe transactions). Can add control reg state into CPU cache and invalidate (coherency) for speed.

VM MMIO - expensive with VM exit, multiple MMIO accesses with PT translation.

Para-virtualisation - OS aware of virtualisation, make hypervisor calls rather than MMIO (no VM exits). Don't need to emulate (pretend).

Passthrough - MMIO addresses from host in guest, less exits, device per VM.

Pre-fork/thread to be ready to accept calls, or async for non-blocking.

Data copies: need to move data from kernel to user space on receive (degrades cache locality). Fix: zero-copy accesses kernel buffer, DMA accesses mem from devices.

Async (storage): req kernel for storage data, continue, block until data available.

Event-based (network): copy to user space on receive. only works when kernel signals the operation will not block (FD, buffer, etc).

epoll - stores (fd, operation) in polling set, uses non-blocking I/O. epoll_wait returns events ready for I/O.

Direct device assignment - VM passthrough.

Direct Mem Access (DMA) - hardware to mem without CPU intervention.

Programming Models

One thread per task - high overhead for creation/context-switch

Worker pools - new clients wait, switch overhead

Event-based - good for heavy I/O, pre-allocate and pin thread to core with non-blocking I/O. Keep a state-machine for each concurrent context (much smaller than thread stack)

memcached - network I/O, giant hash table in RAM (kv store) that uses epoll with state machine (still has copies and bad locality at high load)

ix - shares nothing (own hardware resources / data structs). focus on cache locality (NUMA)

Scale-Out

SLO - what tenant wants (latency, throughput)

Tail latency is very frequent with scale-out (1 - (P(not happening))^n)

Max util - no leeway to hide hiccups

Container creation is expensive (namespaces, users, PID, network), init takes time, use FaaS.

RPC - binary-based, client has calling stubs, server handles function call

RDMA - less cycles on data buffer transport (allocate/free, send/recv). Stream (send/recv similar to sockets) and memory (read/write)./ Reliable/unreliable like TCP/UDP. OS network stack in NIC, but need to exchange buffer IDs ahead of time. OS can't enforce things, needs pinning mem.

Service latency - defined by arrival dist. (time between arrivals (poisson), arrival assignment, service time dist. (time to process, exp. realistically)

Aim for low tail latency and max util. Processor sharing (round robin).

Head of line blocking - first job blocks (takes ages)

R2P2 - receiver-driven, lands in initial central queue then direct connection through response path.

Specialised Devices

ASIC - single purpose. FPGA - reprogrammable (slowly) ASIC.

NMP - process near mem (each channel, rank, bank). NDP - process in disk, SSD channel bandwidth much better than PCIe. Smart NIC - cache data, security analysis. Smart switch - Match/action on headers/contents. GPU/TPU. RDMA - straight into mem over network. Multicore SIMD. NVLink - fatter PCIe.