1) System Performance Engineering (SPE)

We need to build fast, maintainable, widely applicable systems. 2 = possible, all 3 = very hard. Need to strike a balance. **Process**: produce a solution, make it work, and then make it fast.

PE encompasses techniques applied during sys dev life cycle to ensure the non-functional requirements for performance will be met.

1.1) When to stop optimizing:

1) Define target metrics: throughput, latency, scalability, mem use, energy consumption, emissions, elasticity (how well it scales up / down / respond to changes), efficiency.

2) Decide when these targets are met:

Easy: Set an optimization budget: usually dev time. Harder: Set an opt target that is **agreed with the client:** real time reqs/threshes that must be met.

Soft Reqs: Meeting it is ideal but not obligated.

Hard Reqs: Crucial to meet.

Quality of Service Objectives (QoS):

Statistical properties of a metric that will hold for the system. Can have pre-conds: *"framerate of the game is at least on average 60 FPS on a GPU with at least 50 GFlops."*

Service Level Agreements (SLA):

Formal legal contracts specifying QoS objectives, and **penalties for violations**. *Trading orders will take max. 1ms response time. Else, the user is eligible for 10% credit towards fees.*

1.2) SMART Requirements:

S - Specific: State exact reqs in numeric terms

M - Measurable: Stated reqs must be measurable

A - Acceptable: Reqs must be rigorous enough to guarantee system success in reality

R - Realisable: lenient enough for implementation

T - Thorough: Reqs must ensure that all necessary aspects of the system are specified

1) Ways of **Measuring**:

Monitoring (testing the system over a long time):
- Performed on the prototype or final actual system.
- If done properly, it's accurate. Done using **instrumentation:** (adding benchmarking code into the source code). Can be costly and difficult.

Benchmarking (controlled settings: e.g in a lab):
- The system is put in a predefined steady/hot state
- We perform important operations (perhaps the most common or expensive operations – known as the benchmark **workload**) & take perf metrics.
- Data generator: code that gives the sys benchmarking data
- Query set: the operations we perform on the data.
- **Benchmark Workloads:**
  - Can be **batched** (a set of queries):
    - Prog has access to entire batch from start.
    - Simpler as generator perf doesn't matter.
    - Good when measuring throughput
  - **Interactive** (a prog generates requests):
    - Generator (often randomly) gens work bit by bit
    - Useful when metric is latency
    - Gen speed must >= speed of sys under eval

Interpreting Results: Need to aggregate (avg) **many** runs and report variance. Combat Noise.

1.3) The Optimization Loop

New Feature -> Eval Perf -> good enough = done
'->not good enough: Idnt Opt Oprt

Identify Opt Opportunity: if available then Eval Perf If not available, then redesign -> New Feature.

1) Identifying Optimization Opportunities:

To find out where to opt, need examine Sys Params

Sys Params: Constant as Sys runs (caches, CPU)

Workload Params: Change as sys runs (users, available mem)

Numeric Params: CPU freq, avail mem, users, mem throughput…

Nominal Params: Runs on Battery? has GPU? is VM?

Utilisation: % of a **specific resource** used in performing a service. Total amount or budget of resources available to a service is a parameter.

Bottleneck: resource with the highest utilisation

"x"-bound: indicates x is the bottleneck of the sys. e.g: CPU bound or Latency Bound (most of the time the system is waiting for operations to complete)

Performance Dominating Code Paths (PDCP): To lower optimization complexity restrict effort to these paths. **Critical Path:** sequential part of code. **Hot Path:** Path that takes the most time.

After identifying PDCPs and Sys Params, we know what code to optimize, and what to optimize it for!

2) Methods of Optimization:

1. Quickly compare different designs (development)
2. Quickly select close-to optimal vals for a platform (Parameter Tuning):

- We can't control workload params, but can control sys params.
- **Parameter Tuning:** Finding the vector in param space that either minimises resource consumption or maximises a perf metric.
- **Analytical Performance Models:** Formal characterization of the relationship between sys params and performance metrics. **Complicated**. Need to model dynamic sys with small static model. Models: Stateless (Char Eqs) / Stateful (Markov) Allows fast, "what-if" analysis; simplifies tuning. If you can build a good analytical model, then you've understood the system.

Example: An I/O bound application, that needs to read 40MB per request is limited to 10 requests per second when running on a hardware platform with a single disk providing 400MB/s bandwidth.

Simulation Example: PE-Grep

Simulation: Single observed run of stateful model. Expensive to calculate. PE-Grep ended up outperforming GREP as PE-Grep worked on a specific string & was able to compare 64 bits at a time (rather than 8 bits - SIMD; working on 64 bits as a single unit of work) - efficient in most CPUs. Traded generality (& some maintainability) for perf.

2) Performance Tracing and Profiling

To identify optimization opportunities, remember:
- We need to identify the hotpath
- … and the bottleneck, in terms of CPU, Mem… etc
- Both are functions of the system behaviour, so we need to describe that.

Events: Describe System behaviour.
- Any changes in system state. We restrict it to a certain granularity.
- **Simple/Atomic Events:** sent package, executed instruction, loaded addr from mem… etc
- **Complex Events:** cache line evicted from L1 to L2; instruction aborted due to mis-speculation
- Events have opt *payload* (data describing event)
- Accuracy: degree at which event's value represents the ground truth

Event Sources: they produce events. Two parts:
- Generator: observes changes to sys state to then generate events. (Online, part of runtime env/sys)
- Consumer: processes events. On/offline.
  - Requirements: Detailed, accurate, low perturb.
  - We collect events from Software (Libraries for Manual Instrumentation/Logging), Compilers (Automatic), OS (Kernel Counters).
  - Hardware Performance Counters
- Emulators (run software in a simulated CPU giving us perfect control over timing. V low perturbation, but very intensive to run systems in Emulators).

Events are fed into tracers/profilers.

2.1) Event Tracing

Event Trace: a complete log of every state the system has been in during the period of interest.
- Formed of events, usually totally ordered.
- Accuracy is "inherited" from events.
- Event collection overhead might be high

Example**: Call Stack Tracing** (store vals of data when jumping and linking -> fun calls) Problems:
- Recording entire call stack expensive
- The stack needs to be walked, ptrs need chasing
- Call stacks are deep. All Frame Pointers must be written to memory.
- For small funcs, processing this is even more expensive than the funcs themselves.

Perturbation: Degree at which performance of a system changes when it is being watched.
- Non-deterministic. Negatively affects accuracy.
- To reduce perturbation, **reduce fidelity.**

Fidelity: degree of exactness with which something is copied or reproduced.
- **Perfect Fidelity:** every-thing recorded.
- **Reduced:** Not everything recorded.
- Reducing fidelity is done **via sampling**.

Call Stack Sampling:
- Skip some events. We might not sample some small funcs, but expensive ones get sampled more.
- Low perturbation, and good performance.
- Fidelity trades against perturbation and perf.

Intervals: distance between two samples being taken. Interval = 1: same as event-tracing.

Time Based Intervals:
- Set hardware recurrent timer. Sample when it 0s.
- Use CPU reference cycles as a proxy metric
- Inaccurate, non-deterministic and noisy (computer clocks are poorly defined)
- Clockrate varies, clocks vary across CPUs.
- Easy to interpret (time is inv proportional to perf)

Event Based Intervals:
- Generalises Time-Based (as CPU time is discrete)
- Define intervals in terms of the occurrence of an event, e.g: sample every fifth function call
- Accurate, deterministic semantics with low noise
- Tricky to interpret (we're interested in time taken)

Quantization Errors: Due to limited internal res, data is sometimes off by multiple clock cycles.
- Time is continuous in reality but not on computers.
- Causes Quantization Error / Bias - costs attributed to the wrong state.

Indirect Tracing: • Some trace events are "dominated" by others (executed depending on their outcome). Intervals defined by the exec flow.
- For example, control-flow instructions (if, else, for, while) dominate non-control-flow instructions
- Can massively reduce overhead - with some CF Analysis we avoid sampling specific instructions based on what executed earlier.
- Can trace the Control Flow instrs, and outcome, build the trace based on our program!
- High fidelity & accuracy usually good (depending on the event and the indirection)

Summary: • Tracing collects (subsets of) events.
- Perturbation is an issue but can be avoided.
- Analysing traces is extremely tedious; lots of data, little structure. Hard to work with. • Profiling superior

2.2) Profiling (e.g: Flame Graphs)

Profile: Characterization of a system in terms of the resources it spends in certain states.
- A (possibly global) aggregate over the events of a specific metric e.g: tot cache misses, tot CPU cycles
- Or broken down by some other event: cycles per instruction (CPI), cache misses by line of code
- Info is lost, we don't profile everything ofc.
- Produces Post-mortem for ease of interpretation
- Realtime to reduce perturbation (aggregation is cheaper than dumping - we aggregate many events into unifying bit of data and record that)

Flame graphs: • y-axis: stack depth.
- x-axis shows stack profile pop, sorted alphabetically. Each rectangle is a stack frame.
- box width is proportional to num samples

2.3) Instrumentation
- Augmenting program with event logging code
- No need for any hardware support, very flexible
- Overhead is high, Perturbation is high

1) **Manual Instrumentation**: printf logging (or using a logging library).
+Fine control over instrumentation.
+Needs no support from hardware or compiler.
-High overhead for implementation & runtime
- Usually disabled for release build.

-Needs recompilation for selective enabling

2) **Automatic**: Compiler supported injection of event recording code into a program.
- Source-to-source rewriting is possible

-Less control, need for compiler support
+Can be done quickly, things are done consistently.

3) **Binary**: Instrumenting already compiled binary.
- Static - Adding instrumentation directly, overhead can be assessed from the binary.
- Dynamic: No recompilation. Performed at runtime. Works with JiT-compiled code.

Note: Sometimes, functions we may expect to show up in our profile don't. This is because our time quanta is too big – the func is too fast!
We need higher fidelity/lower overhead!

Software Performance Counter (OS):
- Tracks Network Packages sent, VMEM Operations
- Not fast enough if we want to write code that is efficient at the microarchitectural level… Solution:

Hardware Support:
- Special regs can be configd to cnt low-lvl events.
- Fixed number can be active at runtime

Defines collected events as well as intervals.
- Unfortunately: Often buggy or unmaintained, poorly docum, poor accu. Common ones are ok…

Performance Counters help us make useful profiles for Microarchitectural Bottleneck Analysis:

2.4) Microarchitectural Bottleneck Analysis:

Pipelined Execution: exec aiming to engage all the CPU pipeline via assembly line type workflow.

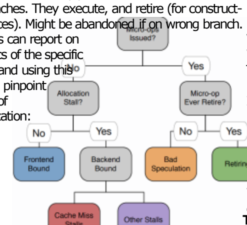
Control Hazard / Pipeline Bubble: We can't populate the pipeline if we don't know what instructions are next (e.g: if we have JMP). Results in a stall. Solution: **Branch Prediction.**

Branch Prediction: guess next instr & continue loading the pipeline and executing.

Resource Stall/Structural: stalls arising from waiting for some computation to complete – e.g: for an ALU result.

Data/Memory Stalls are common esp. if we get L1/L2 misses. CPUs often stall on Data Access. We want to find if specific micro arch issues, e.g. Data Stalls/ALU Stalls/Branch Mispredictions or Cont Flow deps. are optimizable causes of slowdown.
- Instructions enter pipeline and interact with ALUs, FUs, caches. They execute, and retire (for constructing traces). Might be abandoned if on wrong branch.

Profilers can report on statistics of the specific things and using this we can pinpoint points of optimization:



Frontend bound – no instructions issued, but resources were avail; nothing entered the pipeline. Due to CF dependencies - jump couldn't be speculated.

Cache Miss Stalls: detected due to a lack of resources, or other types of stalls (usually the ALU).

Instruction Issued; didn't retire: misspeculation.

Retiring: What should happen; the good state.

3) Analytical Performance Modelling
- We could empirically find hard & software sys metrics if we have access to hardware, code & input
- Modelling is useful if we don't have access to this:
  - It also lets us cheaply find the performance
  - Allows us to charge $ for software service exec.
  - Allows runtime opt & tuning: JiT comps, dbs, ML
- We make a model of the machine, code, and data, and feed it into a SysModel that we can profile
- Assumptions: we make simplifying assumptions about the input: (e.g: assume known distribution).

We don't model system noise (due to scheduling, other processes, ext factors).
- Assume single-threaded, deterministic code.

3.1) Numerical / Experimental Models

1) For every relevant parameter in the system, get a set of metric data points for each relevant point in that parameter's parameter space. That data point is the metric we try to predict.

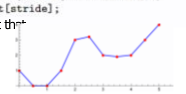
1.1) Done by running an experiment that isolates that performance characteristic: microbenchmarking

Microbenchmark: a small, specially designed program used to test a small part of the system.

Memory System Microbenchmark Example:

```
extern int* input;
extern size_t N;        // some large constant
extern size_t stride; // the parameter of our
                      // experiment
int sum = 0;
for(size_t i = 0; i < N; i += stride) {
  sum += input[stride];
```

2) We interpret that data, e.g: by interpolating between the points to get predictive vals in between measurements.



+Easy to get (if sys available) +Based on ground truth +Easy to interpret.

-Generalizes poorly. –Lots of experimental data needed for high dimensional system param spaces.
-Limited accuracy or prediction confidence (especially if data is missing, or inaccurate).
-Limited intepretability (contributing factors hard to see), -Limited insight (how does sys *actually* work?)

3.2) Analytical Models
- Very complicated (overkill, unless rly precision)
- Development very hard. Requires detailed sys understanding, extensive validation, dealing with tricky edge cases, because adding more and more optimizations to a complex system in specific edge case makes those systems even more complex.

Building Analytical Models:
- Done by turning numerical models to analytical models - ie: we take the data point graph we had before and perform some kind of regression.
- Regression can't model especially complex curves.

Analytical Models are defined by:
- **Characteristic Equations** (with parameters): An equation that describes the behaviour of the target metric of your experiment or system in dependence of a varied parameter.

In our microbenchmark examp: stride and data size
- **Values for system parameters:** in our examp: access latency, access granularity (block size), cache, capacity. Broader examples:

Variable	Description
B_0:	Size of a General Purpose Register of the CPU
l_0:	Access Latency of the Level 1 Cache
C_0:	Capacity of a General Purpose Register of the CPU
B_1:	Size of a cache line of the Level 1 cache
l_1	Access Latency of the Level 2 Cache
C_1:	Capacity of the Level 1 Cache
B_2:	Size of a cache line of the Level 2 cache
l_2	Access Latency of the main memory
C_2:	Capacity of the Level 2 Cache
B_3:	Size of a Memory Page
l_3:	Lookup time in the Page Table
C_3:	Number of Memory Pages in the TLB tims Page size

Self Tuning Systems:
- Fit to sys params based on how sys works rn
- Require less work/expertise are more resilient (some other user / unpredictable workload may share the device)
- scale forward (i.e., work on future architectures)
- sometimes more accurate

Worked Example: (s = stride)

With the same code as in the microbenchmark:

$$T_{mem} = l_3 * \min(1, s/B_3) + l_2 * \min(1, s/B_2) + l_1 * \min(1, s/B_1) + l_0 * \min(1, s/B_0)$$

If we changed experiment for sum+=input[i%size]

$$T_{Mem} = \begin{cases} l0 & size < C1 \\ l0 + l1 & size < C2 \\ l0 + l1 + l3 & size < C3 \\ l0 + l1 + l2 + l3 & \text{Otherwise} \end{cases}$$

We can find the sys params by fitting to our char eq, tho it's often in the documentation too.

Things to consider with regards to CPU efficiency

1. Is the system executing instructions speculatively?
2. Is the system executing instructions dynamically parallel; e.g: superscalar execution.This is not multi-core parallelism!
3. Is the system executing instructions out-of-order?
4. Is the system executing instructions statically parallel? Statically bundled packages (SIMD or VLIW)

Speculative Execution keeps the pipeline filled. Addresses some control hazards. Requires predictable code, else may have to flush the pipeline.

A more complex example:

```
extern int* input1; // uniform random data, 1024 values
extern int* input2; // random data, 64 values
int sum = 0;
for(size_t i = 0; i < inputSize; i++) {
  sum += input2[input1[i]];
```

This has a mem access of a mem access; with two outside inputs dictating that

Parameters:

1) **Memory Region Parameters:**
- length (R.n), i.e., the number of stored tuples
- width (R.w), the size of a tuple in processor words (we will assume a processor with 64 bit words).
- The size of the region (||R||) is defined as the product of length and width.

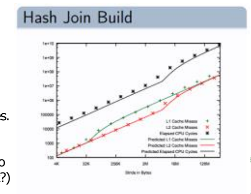
2) Access Patterns: *u* ts the number of words read in each access. *r* is the number of accesses – we add this as an extra param for the random acc case.

Modelling Random Access w/repetitive elem access):
- P1⊕P2 the seq exec of the access patterns P1 and P2
- P1⊙P2 the concurrent execution of access patterns.

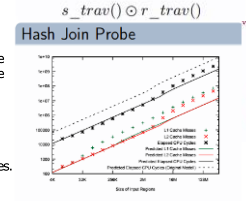
1) For above code: *s_trav(R.w = 1, u = 1, R.n = 1024)* We have a stride (u) of 1 byte at a time, a word size of int (1 byte). For the seq case, R.n = 1024, because we have 1024 tuples in input1

2) ⊙rr_trav(R.w = 1, u = 1, R.n = 64, r = 1024) For the random access case, r = 1024, because we access for each tuple in input1, and R.n = 64 as input 2 has 64 tuples.

Modelling random access without repetitive access is useful too, e.g for a hash join (which builds a new hashmap based off reading the old one and probes the new one based on values of the second hashmap):



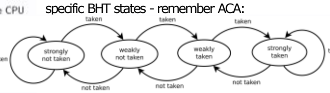
$s\_trav() \odot r\_trav()$



$\oplus s\_trav() \odot rr\_acc() \odot s\_trav()$

3.3) Stateful Systems
- Some effects/components have dynamic state
- State can influence behaviour and performance
- Stochastic models/processes have state unlike anal
- Markov Processes/Chains (FSMs with transit probabs)
- Next State only depend on curr state & random var

Crucial for Branch Pred: as it is random based on specific BHT states - remember ACA:



- Probability of the branch predictor predicting is prob of it being in one of the states on the right.
- We can calculate the probability of it being in any state as the Markov End State (stationary dist)
- Branch misprediction rate: (P(pred_taken) ∗ P(act_not_taken)) + (P(pred_not_taken) ∗ P(act_taken))

4) Efficient Code – hardware lvl optimization

Balanced System: A system without a bottleneck, where all resources are equally utilised. Either nothing or everything is bottlenecking the system.

Perfect Balance is ideal; in practice unachievable. The fundamental tradeoff in our computer systems is between **CPU** and **Bandwidth Efficiency:** we distinguish between balanced, compute-bound, latency-bound and bandwidth-bound code.

4.1) CPU Efficiency: CPU bound apps typically:
- poorly implemented, or work on tiny cache-resident data sets, or maths (especially flop) heavy
- **Target metric**: Wallclock time is the best metric.
- CPI not too useful as instrs are param themselves
- Stall cycles are a decent indicator caused by hazards: Control, Structural hazards, Data hazard

In the following section, we'll discuss factors that should be abused by the programs we write in order to have high performance code.

4.1.1) Out of Order Execution
- Kicks in after decoding decode-phase
- Instructions with no unsatisfied dependencies move onto execution
- e.g: in a*b+d*e+b*b, b*b is scheduled before d*e, if b is L1 resident, and a,d,e aren't.
- Addresses data and structural hazards; not control

4.1.2) SIMD: Single Instruction Multiple Data
- Performing same operation across lots of data.
- Uses specialised vector registers.
- Instrs like: MMX, SSE, SSE2, AVX512.

4.1.3) VLIW: Very Long Instruction Words
- Single instr, with different operations on multiple or the same data item
- Compiler controlled superscalar execution

4.2) Tricks to Exploit these Factors:

1) **Write Runtime Predictable Code:**
- Do not evaluate code in the critical path whenever possible - evaluate code as early as possible

Partial Evaluation: Make Compiler do the work: *Treat programs eval as a multi-phased process*
- In every phase, you know more about the result -we converge to what we want
- you can evaluate more of it/create a more specialised program/better program.
- the fix point is obviously the actual result
- Examples: Function Inlining, JiT compilation, Constant Expression Eval, Lifting Expensive ops.
- Lifting: e.g: **Loop Invariant Code Motion**.

4.2.1) Loop Specialisation: Add code for edge cases of a loop, or split a loop into many different possible code blocks which do the operation in different ways (specific ones faster than others).
- Makes the code more complex, but faster. Remember our tradeoffs!
- The compiler however can do this for us!

Metaprogramming helps guide the compiler to generate all this code for us: c++ templates (can input values like 1 / 2 to turn runtime vars into compile time vars; can use a hashmap to codify special cases), pragma…

```
template <int scale> void scaleVectorPE(int* input, size_t inputSize)
  for(size_t i = 0; i < inputSize; i++)
    input[i] *= scale;

void scaleVector(int* input, size_t inputSize, int scale) {
  map<int, function<void(int*, size_t)>> const specialCases //
                                                            //
      {{0, scaleVectorPE<0>},
       {1, scaleVectorPE<1>},
       {2, scaleVectorPE<2>},
       {4, scaleVectorPE<4>}};
  if(specialCases.count(scale))
    specialCases.at(scale)(input, inputSize);
  else
    for(size_t i = 0; i < inputSize; i++)
      input[i] *= scale;
```

- Function inlining collapses JMPs into branches

4.2.2) Predication / if conversion means we don't have to speculatively execute:

```
for(size_t i = 0; i<inputSize; i++) {
  if (input[i] < high) output[outI++] = input[i]; }
```

Becomes:

```
for(size_t i = 0; i < inputSize; i++) {
  output[outI] = input[i]; outI += (input[i]<high);}
```

- This is more work overall, but means no speculative exec. In cases we strongly take one of two branches, this predicated code performs worse.
- Predication turns control deps into data deps.
- Work per iteration is now 100% predictable.

4.2.3) SIMD Vectorization: We can make num iterations predictable, speeds things up too.
- Compilers try to autovectorize. But they fail in more complex cases. Directives help them out: `_mm256_<intrin_op>_<suffix>(<data type> <parameter1>, <data type> <parameter2>, <data type> <parameter3>)`
- Example:

union v8f { float floats[8]; __mm256 simdVector;};

We can vectorize a normal loop that would add an array of arrays of 8 floats with SIMD like so to do it way quicker:

```
float sum = 0;
for(size_t i = 0; i < bounds1 / 8; i++) {
  v8f values{.simdVector = _mm256_i32gather_ps(
              input2, ((__m256i*)input1)[i], sizeof(int))};
for(size_t i = 0; i < 8; i++)
  sum += values.floats[i];
```

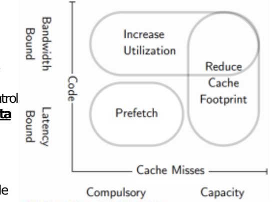
- Bottom line: use intrinsics, keep data in vec registers, check with Godbolt

4.2.4) Avoiding Data Hazards
- Caches have a fixed cache line size (e.g: 64 byte)
- Caches have a fixed capacity (e.g: 32 Kbyte)
- Most caches evict LRU. **Cache Miss**: data accessed by instruction but not in cache. **Data Hazards:** Pipeline stall cycles due to cache misses.

Classifying Cache Misses:
- if accessed before, we call the respective miss a **cache miss**. • If not, it is a **compulsory miss**.

Classifying Code: • Code is **fully memory bound** if all stall cycles are due to data hazards.
- If the memory bus is fully utilized, we call it **memory bandwidth bound.**
- If the Memory Bus is not fully utilized we call it **memory latency bound**.

Memory Bandwidth Bound - we aren't sending enough data Memory Latency Bound - our data doesn't arrive quick enough




```
#include <stdlib.h>
struct tuple { int x; int y; int z;}
int sumIt(tuple* input, long size) {
  int sum = 0;
  for(size_t i = 0; i < size; i++)
    sum += input[i].x;
  return sum;
}
```

-Runs into datasize/cachelinesize many compulsory hazards on an in order processor. Latency bound. These datahazards can't be avoided, but we can do:

4.2.5) Asynchronous Processing
- Caches can speculatively load the next cache line
- Done by pattern recog (adj cache lines, strides)…
- Doesn't work so well for random mem access.
- **Software Prefetching:**

We can provide prefetching hints to the compiler: __builtin_prefetch(&input2[input[i+16].x]) (prefetches values of up to 16 indexes ahead). Num indexes: need a sweetspot. Should benchmark to find the ideal.

If we're still memory bound after soft prefetching:

Cache Size Utilization: = data requested by instrs / data loaded in cache
- We can increase this changing the data layout in memory. Array of Structs vs Struct of Arrays!
- We should try our best to read things stored nearby in main memory first, increasing the chance we read things from nearby cache lines.

Dealing with Capacity Misses:
- Happens due to cache thrashing – our cache is too small; we end up evicting things that we need later!
- We can reduce cache thrashing with loop tiling:
- Imagine a simple loop which we multiply numbers from two different arrays together. We can instead work in tile sized chunks (tile is cache line length):

```
for(size_t tileI = 0; tileI < xrange; tileI += tileSize) {
  for(size_t tileJ = 0; tileJ < yrange; tileJ += tileSize) {
    for(size_t inTileI = 0; inTileI < tileSize; inTileI++) {
      auto i = tileI*tileSize+inTileI;
      for(size_t inTileI = 0; inTileI < tileSize; inTileI++) {
        auto j = tileJ*tileSize+inTileJ;
        sum += input0[i] * input1[j];
```

- Not always simple – sometimes we need extra work, not just reordering. Note that code has a typo; the second loop's var should be inTileJ.
- **Radix Sorting:** Produces partitions of the whole, sorts in those partitions by performing a number of partition steps - like quicksort, and merges them at the end. The access pattern of these values is always sequential - one by one - good in cache.
- **Conclusion: Measures:** If bandwidth bound, increase cache-line utilization: If latency bound, prefetch. If cap bound, reduce footprint/hot dataset.

4.2.6) False Sharing
- For cache coherence across cores we use MESI. (4 cache line states: modif., exclu., shared, invalid)

False Sharing: slows down perf on distrib caches.
- Two cores have a copy of a cache line.
- Whenever they modify it, they communicate the change to the other core.
- However, we be looking at separate bytes in the cache line, and yet communicating changes the other cores don't even need. **False Sharing!**
- Commonly occurs with counter variables.
- Fixed by padding cache lines - make sure cores only access their own cache lines - by making variables not TOO close to each other in memory.

5) Multicore and Parallelism

Forms of parallelism:
- Data level: vector instrs (MMX SSE, AVX)
- Instr level: (superscaler out of order processor)
- Task level: (GPUs, P-threads; by programmers)

Parallelism: Physically doing many tasks at once. AKA: True Concurrency.

Concurrency: Executing different tasks in overlapping time periods. Pintos type concurrency. One core of execution.

If we have multiple cores & interleave work, its both

Amdahl's Law:

Speedup(p, s) = 1 / ((1-p) + p/s)

Where: s = num threads, p = parallelisable part of the code, as a decimal proportion.

Increasing cores: better performance; cheaper to run; but there is diminishing returns.

Data Centers can be seen as a computer/system to optimize! Thus, we optimize for Total cost of Ownership. Factors: hardware costs & mean time to failure. Energy consumption, rent, cooling, security.

5.1) Multithreading

Threads run concurrently & share memory. If mapped to separate cores; parallelism.

5.1.1) Performance Analysis Steps

1. Start with Sequential Application.
2. Apply tools for performance analysis:
`perf record <program> <arguments>`
`perf report --stdio --percent-limit=1`
3. Use Amdahl's Law to identify what to parallelize, and then use multiple threads.
4. Once a function is fully parallelized, rerun the perf analyser.

Critical Path: "sequence of tasks deter-mining min time for an application"

Critical Path of example app:
Base app: a→c Removing a(): b→c
After removing b(): a→c Removing c(): a

The profiler however is unaware of critical paths across threads.

The point here: optimizing a gives us **at most** a 6.5% performance boost, as we have to wait for the thread that goes in b to join as well. We'd have to optimize a AND b, or we could optimize c instead and hit 2 birds.

5.1.2) Explicit Sharing & Atomics:
- Cache Coherency (CC) keeps data in caches consistent across cores. MSI Protocol: Modified, Shared, Invalid. Per cache line state machine.
- **Data Races** stem from the non atomicity of RW ops, and lack of cache coherency.
- Coherency isn't enough; we need RMW.
- C++ provides **atomics!**
- We have increment, decrement, Compare and Swap, linked-load and linked-store.
- CC states: Modified, Shared, Invalid, **Locked**

CAS(addr, old_val, new_val):

Atomically performs:

```
if (*addr==old_val): *addr == new_val;
```

Example atomic increment with CAS:

```
void atomic_inc(uint64_t* addr) {
  bool swapped = false;
  while (not swapped) {
    auto old = *addr;
    swapped = CAS(addr, old, old+1);
  }
}
```

C++ has:

```
atomic_{load, store},
atomic_compare_exchange_{weak, strong},
atomic_fetch_{add, sub, or, xor, and},
std::atomic<Type>, std::atomic_flag.
```

5.1.3) Critical Sections
- Sections of code in which the operations should be executed atomically (ie: only one thread should be allowed in at once)
- We can use CAS explicitly with a Boolean to check if the CS has been entered, or a lock that's implemented with CAS to guard critical sections. Lock before and after the CS.
- Semaphores (N inside critical sec), Lock / Mutex (1 thread), Condition Vars (one or more threads wait for a condition to be true), Barriers (wait until N threads are waiting inside barrier, then unlock all). Shared_Lock (aka Read/Write Lock – multiple threads in shared mode, 1 in exclusive mode.
- All of them are std::<name of thing>

5.1.4) Lock Implementations

1) User Level Locks:

```
#include <atomic> class Lock {
private: std::atomic<bool> locked;
public: Lock() : locked(false) {}
  void lock() {
    bool expected = false;
    while (!locked.compare_exchange_
           weak(expected, true)) {}
      expected = false;
    }
  }
  void unlock() {locked.store(false);}
};
```

Easy. But we busy wait; expensive. Threads can starve too…

2) Kernel Level Locks:

Algorithm
- Make a system call to acquire lock
- Logic is implemented in kernel level, similar to user level. Re-schedule thread if blocked and trying to acquire.
- System call to release lock; kernel wakes blocked threads up.

Observations
- Expensive if already released (CAS vs syscall + CAS).
- Keeps fairness between threads.

Futexes:
```
class futex {
public: futex(uint32_t init_value):
        counter(init_value) {}
  void acquire() {
    int i = 0;
    for (; i < 100; i++) {
      int cVal = counter.load();
      if (cVal > 0) {
        if (counter.cmp_ex_str(cVal,cVal-1) {
          return;  } } }
    int res = syscall(SYS_futex,
      reinterpret_cast<uint32_t*>(&counter),
      FUTEX_WAIT, 0, nullptr, nullptr, 0);
    if (res != 0) exit(res);  }
  void release() {
    counter++;
    int res = syscall(SYS_futex,
      reinterpret_cast<uint32_t*>(&counter),
      FUTEX_WAKE, 1, nullptr, nullptr, 0);
    if (res != 0) exit(res);  }
  private: std::atomic<uint32_t> counter;
};
```
Best of both worlds!
• **False sharing** can be found on perf: HITM metric. (e.g: we had a miss when we should've had a hit).We get the address where this happened with perf c2c which tells us where to optimize.
• Fix with padding or mem alignment of structs, etc.

5.1.5) Thread and Task Management Models:
1) Single server generating & processing job bursts. Simple, efficient, but no parallelism.
2) **Dispatch** a new thread per job. Simplest parallel approach but inefficient if #threads > #cores, especially with locks.
3) **Threads for batched instruction processing using a threadpool**. Each batch runs in parallel. Efficient, unless we have task imbalance (some threads get way more work than others).
4) **Work Stealing Threadpools (Consumers):** We have a shared job queue that the server adds to as it receives requests. Consumers pull from this queue. This is optimal consumer balancing, but we can have contention on the queue.
5) **Streaming**: Pipeline with concurrent stages:
• We have a thread per function (task queued as input; the thread processes this; and then enqueues a task with the output for the next stage)
• Very similar to hardware pipelining!
• Good for locality of code, and temporary data, bad locality for input/output data, and we have event-driven per function threads.

Analysing/Optim. Perf of Streaming Systems
• We can detect issues by comparing throughput to a threshold we expect to be above.
• We can check for bottlenecks by examining which queues have lots of items in them.
• If a queue is always empty, maybe we have too many per function threads.
• Slow stage, increase per function threads.
• If too many idle threads, decrease threads.
• We need to make stages equally costly.

SEDA: queues, modular stages, th pools per stage
• **Staged Event Driven Architecture**
• Designed for mass concurrency and dynamic ctrl.
• Bottleneck are explicit, fails on full queue
• Observable load: can peek at all queue lengths and contents. Can react to load by adding / removing threads, batching queues, queue length..
• Very flexible for tuning, used in practice
• Stages across threads hurts performance; communication required vs cache locality

5.2) Multiprocessing
• Scalable Distributed Pipelines use processes for their concurrency; deployed across machines. IPC happens across the network.
• Need to min comm overheads; max resource util:
• Without shared mem, communication is harder:
• overheads: syscalls, scheduling, memcpy
• programmability: passing complex data hard: need to serialize and deserialize... !!
• No perfect solution:
Explicit Shared Memory: has futex, no need for copying; but no std comm interface (local only)
Sockets: Immediate copies, but requires network stack processing, even locally
Pipes: Imm copies, same R/W interface; only local

6) Tools, Patterns & Algorithms
• **Correctness:** -GCC –Wall - Valgrind, memcheck (valgrind <program>); -GCC –fsanitize=address **(ASAN)**, race detection: -valgrind –tool=helgrind <program>, tsan too.
• **Performance: Basic Counting:** perf stat <program>, **Profiling:** perf record program... + perf report, with the callgraph: add –g after record/report. Perf list: (lists all hardware and software counters); perf help. **coz:** can give optimization hints on parallel programs. Works by injecting time delays to make it as if everything else is faster, and sees the result of the "speedup". Not particularly reliable.

Higher Scale Memory Allocations:
• Default malloc doesn't scale with threads.
• We need to use global structures to track alloc/free. But they suffer contention across threads.
• The better solutions are hierarchical and batched.
• We behave the same globally, but per thread we
• maintain a local cache, and alloc/free in batches.
• Examples: Tcmalloc, Jemalloc.

Patterns: RAII!!
We can add cleanup code to destructors; like closing fds. We can simplify locks/scoped locks with RAII:

Algorithms: Non-blocking:
• Non-blocking algos use RMW; very complex. We don't suspend any threads.
• Non-block lock free: system always makes prog
• NB wait free: All threads always make progress
```
void push(Node* node) {
  do { node->next = head; } while (not CAS(&head, node->next, node)); }
Node* pop() {
  Node* current = head;
  while (current) {
    if (CAS(&head, current, current->next)) break;
    current = head;
  }
  return current;
}
```
This runs into the **ABA** problem - thread A goes into pop, thread B goes into pop twice and then pushes, and thread A sees the same value as before at the CAS - and assumes that the state was the same when it wasn't actually...

T1: pop	T2: pop1, pop2, delete2, push1
current = 1, next = 2	
	current = 1, next = 2 CAS(head, 1, 2)
	current = 2, next = 3 CAS(head, 2, 3)
	delete 2
	head = 3, 1->next = 3, CAS(head, 3, 1)
CAS(head, 1, 2)	

solutions:
• intermediate nodes (expensive): std::shared_ptr
• tagged ptrs to wrap around: e.g: 128 bit CAS
• Data reclamation (read-copy-update atomic)

7) System Interfaces
• Cost calculated on time in a system, and energy.
• Von Neumann Bottleneck - fetching operands is actually more costly than computing them, and yet communication is essential for compute.
• Everything is a trade off; perf, cost, productivity.

Keys to an Efficient System:
• **Abstract** complexity (e.g: fds vs device drivers) to make programming with your system quicker
• **Multiplex** – don't keep around unused hardware; it's a waste of money and energy.
• **Caching.** Storage costs increase with perf.
• **Avoid Communication** where possible.

7.1) OS Interfaces
• Abstraction is everywhere: CPU/threads -> sched; Mem/VMem ->Paging; Security/Privilege ->Syscalls
• Protected objects are implemented inside the kernel; accesses like sockets, files, mem allocation.

7.1.1) The Anatomy and Cost of System Calls:
Anatomy: store args in regs; 2) do syscall (enter kernel mode) 3) Sanitize environment 4) save state 5) execute handler 6) restore state 7) system return (reenter user mode)
Cost: step 2, 3, 4, 6 & 7 are all expensive due to added code and flushing the pipeline compared to a typical function call: • Empty function: 5.5±0.04ns • Empty syscall: 62.07±0.11ns • Overhead: 11x
• About 4x DRAM accesses
Unikernel & Library OS: has a simple OS kernel as a **library** to an app. Adaptable to app needs, (micro&exokernels), eliminates app./libOS privilege separation (hence no need for syscalls). e.g: LKL, OSv. Run as process, container or VM (unikernel)
• Resolve privilege by making just use Function Calls isolating each process with a copy of the OS in **user mode** (rather than running one OS in Kernel).
• We still need a real OS on the hardware for communication, but we rely on syscalls much less

7.1.2) Caching and Storage
• Fast storage is very expensive. Limit the size of it, and use caching, in hardware and software.
• Cache misses require expensive (time, energy) copy op. Reduce misses (predict patterns)

7.1.3) Copying & IPC
Buffered IPC: We have a buffer in the kernel which we send/write and read from either end. A process writes to the kernel; and another asks it for data. Async. Non Blocking. 2 Copies of Data.
Non Buffered: A process tells the kernel it wants to communicate & gets blocked until another tells it wants to read. We directly write from one process to the other end. Blocking but 1 copy of data.

7.2) CPU Interfaces 7.2.1) ISAs
• Contract between the software and hardware.
• ISA defines tradeoffs: security vs perf (privilege), complexity vs perf (e.g: x86 vs arm mem consistency), adaptability vs perf (specialised HW v volume cost)
ISAs are an abstraction that hide microarchitecture, but architecture aware opts remove the abstraction

7.2.2) Caching in the CPU
• CPUs are riddled with cache (L1/L2/L3, TLB (mappings from vaddr to paddr), **PWC**). Caches exploit spatial & temporal locality.
• ISA is extended to control & flush entries; CC.
• Software needs to be optimized for cache hits.
Partial Walk Caches: cache partial PT traversals up to a specific level (prefixes of our path), we update it as we walk the PT. Speeds up page access. The TLB stores the longest prefix match, and if it doesn't hit we check the smaller prefixes of different granularities (PWC1: prefix size 1; PWC2..)
• We check TLB&PWCs at the same time, which reduces our memory accesses.
• Exploits locality at multiple granularities, we know addresses in the same GB for example will have a similar mem addr for many of the first bits.
When updating TLBs; we need consistency across many cores:
• We need to update on page downgrade (turning a page from RW to Read Only), or removing a page.
• We dont need an update when bringing into memory; other cores will page fault & simply fetch.
• TLB shootdown: upon downgrade; flush pipeline, save context, call **victim set** (the other cores with the page), notify them, then invalidate TLB, wait for the other cores to do so too, and then restore context, flush pipeline, resume.
• Solutions: Avoid downgrades / asynch downgrade syscall. Add more hardware for comm. **Intel RAR.**

CPU Access to Memory:
• A bus connects all memory and CPUs. Problems: Signal Prop time, Protocol overheads, congestion
• **NUMA:** Non Uniform Memory Access (AKA due to propagation time, accessing some memory is quicker for some cores than other cores).
• NUMA node: a CPU & close memory connection w/ controllers. Memory closer to CPU quicker. Locality is essential: taskset, set_mempolicy, pthread_setaffinity_np.

7.2.3) Machine Virtualization
• Used a lot in cloud/enterprise systems, by reducing cost by consolidating into 1 phys machine, and is more secure than containers & processes.
• **Exceptions are very expensive**: lots of code to switch between guest/host, and need save many and restore many registers. Hypercalls, trapped instrs (cupid, rdtsc), trapp mem, interrupts cause it.
• Anatomy of a VM exit: same as syscall but we have a trapped instr at start, and then resume VM at end. Enter & leave guest user/host kernel mode
• **To speed things up**: since every privilege transition is a context switch, keep multiple execution contexts (& switch between; SMT). Extend SMT to separate VMT/hypervisor contexts.
Memory Translation is expensive in VMs:
• VMs typically use nested page tables – PT traversal goes from 4x to 24x memory accesses
• Solutions – Let Hypervisor manage PTs (shadow paging); avoid PT changes. Use expensive VM-exits to manage guest PTs. Use guest and host 6x PWCs.
• AKA, use lots of energy&hardware for speed.

7.3) IO Interfaces 7.3.1) Device as Memory
• Using MMIO, devices are accessed as if they're just memory. We give the device range of p_addrs on boot. Corresponds to multiple PCIe BARs
• Every address is **routed** in machine: we distinguish between Host Mem and device BARs.
• We split mem between host mem and MMIO. We use Memory Controllers and IO controllers for this.
• Circular queues of fixed size (set on boot) as a producer / consumer queue (CPU pushes reqs; IO dev consumes to respond) buffers (host memory).
• In the device, BAR to check/configure queue changes (MMIO) by storing the head/tail of that devices work queue.

7.3.2) Device Interconnect
CPU-to-Device: A generic interconnect.
• Connects all devices and CPUs. Has usual issues: Signal propagation & Protocol ovrhds & Congestion
• Locality is key.
Example: PCIe. • About 1 µsec round-trip latency.
• MMIO and interrupts as messages.
• Placement/routing is key • Remote access bad (delay, bandwidth, congestion)
Device-to-Memory: Has usual & locality issues.
• Same locality solns as before: eg: set_mempolicy
Multi-homed Devices: connect the same device to many PCIe ports. Assign addrs to specific PCIe ports. Device local to many NUMA nodes. Expensive

7.3.3) Device and Driver Models
1) Interrupt-driven model
1. MMIO writes to program device
2. Device reads req buffer(s) / writes resp buffer(s)
3. Wait for interrupt 4. MMIO reads to check result
2) Polling model 1. MMIO writes to prog device
2. Device reads req buffer(s) / writes resp buffer(s)
3. Poll for result using MMIO reads
Faster than interrupt if polling time is short
3) Hybrid model – Keep track of inter-arrival and polling latency over time – Dynamically switch between modes – Example: Linux NAPI (interrupt or polling for network devices, based on load).
For newer models, the biggest issue is the interconnect overheads. It's just too slow! Solution:
• Add cache coherency to device interconnect
• CPU polling loop stays in local cache
• Device invalidates polled line to announce, e.g when a network packet arrives.

7.3.4) Device Virtualization Models
Trap and emulate
• Hvisor unmaps devic BARs to trap MMIO accesses
• Hypervisor emulates device op on every trap
• Very expensive to access virtualized devices
• Trivial VM-exit is 450 nsec
• Every operation is a few MMIO accesses
Para-virtualized
• Map shared memory between VM and hypervisor
• A single trapped MMIO access to signal new operations (doorbell)
Passthrough
• Map device BARs into the VM + configure IOMMU to translate guest PAs
• IOMMU/IOTLB translates accesses from devices (similar to MMU/TLB for CPUs)
• Device sharing needs new hardware support ("virtual device functions")
• The device needs one set of BARs and "device state" for each guest VM

7.4) OS Lvl IO Interfaces: Blocking Interfaces:
• Traditional POSIX calls: Pipes, Unix domain sockets, network sockets, open, close, read, write.
• Simple to understand; expensive. Rely on syscalls

7.4.1) Non Blocking Interfaces:
• Avoids the overheads from context switching (destroys cache) & long data copying (destroys too)
Asynchronous: We dispatch the operation in the background; the kernel signals when it's done.
• Designed for storage IO.
• We submit list of ops: io_submit(aio_context_t ctx_id, long nr, struct iocb** iocbpp)
• Wait for events to complete: io_getevents(aio_context ctx_id, long min_nr, long nr, struct io_event *events, struct timespec *timeout)
Event Based: only do the operation when it won't block. Designed for network IO, easy to understand.
• Express interest and availability of operations: e.g. read from fd1, write from fd2. We do something else then poll for available ops (events); and operate on an FD if told it won't block.
• Add {fd, operation} into polling set – operation can be read, write, etc...: epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)
• Wait for {fd, operation} in polling set to be rdy epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout)
• Must use non-blocking I/O on file descriptors – fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, 0) | O_NONBLOCK. • Example: while (missing_read) { auto n = read(fd, ...); if (n<0 and errno==EWOULDBLOCK) { epoll_ctl(fd, need_read, ...); break; } missing_read -= n; }
• Linux began with select() and poll() but had the **Thundering Herd** problem; when an event is available the kernel wakes multiple threads but only 1 actually does it. This spikes CPU usage for just one useful op. epoll solves it with a single epoll channel; multiple channels with EPOLLEXCLUSIVE (put the fd on multiple channels).

7.4.2)Direct Device Assignment
• Doing IO thru OS is performance expensive. (switching on syscalls, data copies, its general purp)
Device **passthrough** with user mode apps faster:
• Application does MMIO directly to virtual functions.
• No syscalls, potentially no copies (app sets up receive buffers before hand)
• Can use ad-hoc libraries and protocols.
IO_uring: a hybrid between doing stuff through kernel and in user mode, covers storage & network
• Can wrap many syscalls and make them asynch.
• Req/resp queue shared between user&kernel
• App manips queue in user mode. App notifies kernel that new ops are in the queue (doorbell)
• Performance improvements when the app:
• polls for results -> less context switches
• pushes req batches -> less contention & syscalls
• uses kernel thread to poll doorbell; no syscalls
• pre-registers receive buffers -> zero cop
• Kernel puts data in free user buf; signal w/uring
• App directly accesses buffer with received data

8) System Programming Models
Once again Performance vs Productivity trade off:
• Concurrency e.g: DDA improves performance, but has overheads and is difficult to work with.
• We think about servers, distributed systems and large-scale compute / cloud; usual for this module

8.1) One Thread Per Task • Typical of old network servers. Easy to program and understand.
Typical Sequence of Operations:
1. Main thread listens for new connections
2. Main thread accepts new connection
3. Main thread creates a thread for that connection
4. New thread does the blocking operations; read/write network request, read/write from disk.
• Scales poorly; thread creation is expensive (time & memory). Thread are c-switchd 2x when blocking

8.2) Pre-allocated Worker Pools
• Avoids cost of on-demand thread creation
• No threads processed if workers are blocked
• Still has context switching overheads
Typical Sequence of Operations:
Same as above, but we allocate a free thread from the pool for the connection, and then release it back into the pool when the connection terminates.

8.3) Event Based – used for I/O Hvy Servers
Use events to concurrently handle multiple ops
• Pre-allocate and pin one thread per core
• Use non-blocking I/O
• Keep a "state machine" for each concurr. context
Observations: no thread scheduling: one thread per core, minimal memory usage: state machine is much smaller than a thread, very complicated to program: logic broken into myriad of small sequens.
e.g: Linux epoll; in Meta's **Memcached Servers:**
In-mem k/v store • Can handle thousands of clients
• Only does network I/O. Uses **small** 456B on heap
State Machine: *accept, read, compute, write, read...*
Observations: Calls to read/write incur copies. Bad locality at high load (application and network stack data and locks go across cores
IX: Another Example: Unikernel-type design
• Shared-nothing architecture: Per-core data strctrs and NIC queue. NIC steers (by hashing on data) packets to qs (and therefore cores). No lck contenti.
• End-to-end processing: maximizes cache locality
• Minimum system overheads – VM with NIC passthrough, zero copy: NIC writes into user buffers, adaptive syscall batching that trades-off latency vs. throughput
Observations: excellent throughput & latency on simple cases. Bad latency on more complex cases.

8.4) Language and Runtime Support
• We can't have both performance and productivity.
• But languages and runtime can abstract and simplify concurrency.
Callback Based I/O:
• Non-blocking logic on main thread: busy with predictably short operations
• IO (longer tasks) handed off to worker thread pool
• Typical seq: 1) Receive and process request – 2) Push I/O request to worker queue with associated callback 3) Worker pops I/O request 4) Worker pushes response when done 5) Main thread pops any response and executes associated callback
Promises and Futures: async prod/consumer API
• Producer – Start producer: std::async(...) – std::promise<Type>::set_value() – std::promise<Type>::set_exception ()
• Asynchronous consumer – Wait for result: std::future<Type>::get().
Execute code when future is ready std::future<Type>::then(...)
Coroutines: Cooperative, non-preemptive concurrency on **a single thread.**
• **Single-thread concurrency**: multiple asynchronous tasks on a thread
• **Cooperative**: language defines task boundaries and scheduling points
• **Non-preemptive**: task runs until it calls task scheduler (cooperates)
co_await, co_yield, co_return
Reads like sequential code:
```
auto data = co_await read_data(...);
auto newdata = co_await process_async(data);
auto error = co_await write_data(newdata);
if (error) report();
```
Stops us waiting for I/O. **Concurrency**, not Parallelism

9) Scale Out
• So far we've scaled up. But past a point that gets too expensive; scale out into many machines. Cloud!

9.1) Metrics, SLOs and Important Metrics:
• Tenant cares about **Service Level Objectives (SLOs):** application latency, throughput, etc...
• Performance per dollar is important.
• Operator cares about SLAs; and infrastruct. cost
Defining SLOs:
• Set target metrics and their performance
• Example: correl. b/w revenue & end-end latency
• Study structure and perf of the app's critical path
• Derive metric SLO for every component
• e.g: for Deathstar Media server they identified the two **critical components**; opt that. Best perf / $.
• In our SLOs we should define our absolute worse case (e.g: latency 500ms; even though average is; for example 300ms); because even infrequent worst case overheads are commonly encountered when systems are in mass use! A google query might contact 1000 servers!
• Thus select SLOs based on percentiles and num. servers used. Use Binomial Dist.

Energy Proportionality: energy has a very high minimum cost (keeping things on). Not linear. Best usage is when hardware is off; or at max util.
Utilization and SLOs:
• To max util: increase app load; co-locate apps
• reduces cost (purchase+energy); but can easily violate SLO
• As utilisation increases we get huge performance degradation: no leeway to hide hiccups (e.g., cache miss, interrupt, ...) & performance interference
Going from SLOs to SLAs:
SLOs derived from app arch & business model.
SLAs are a contract from tenant to operator. Better margins mean app is more reliable and simple, but means the tenant pay more. They're coupled with the tenant and operator's business model.
Asynchronicity: Many components comm over network. Need to optimize CPU perf, sys interfaces and comm (no blocking of threads waiting for comm). Scale out aims to do this in a distrib system
Elasticity: Load varies all the time. We need elastic sys that can adapt to change: e.g: add/re CPUs acc to load (allows operator to lease hware elsewhere)

9.2) Building Scale-out Architectures
1) The simplest recipe to scale out:
Break app into concurrent components.
• Define asynch communication APIs and protocols.
2) Deploy on different nodes
• Elastically deploy vari. processes for components
3) Handle failures; monitor & restart dead processes
Simple -> high productivity; poor perf!
Self Managed: Positives of consolidation (cost), negs of self-manage (monitoring, maint, schedu).
2) **Micro-services:** Apps made of micro-services:
• One service / program for each small task
• Incr capacity by incr service instances (processes)
• Connect everything through the network
UNIX ethos on the cloud: each instance is on a diff vm. e.g: u_auth, file name lookup, cache...
System Perspective: • Good app architecture; as very modular and services evolve independently.
• System management is complex; VMs are expensive (long boot time; duplicate mem), monitoring, VM image distribution, life-cycles...
• Containers vs VMs:
• Isolate tenants w VMs; isol services w containers
• Containers cheaper setup&boot; less duplic mem
• Kubernetes manages containers on VMs
3) **Serverless**: The Tenant declares **what to run:**
• Microservice is executed on certain events: e.g: certain files are written/net addresses accessed
• Simpler to manage; pay per use (sub-second)
The operator handles how to run it:
• **Abstract** m-s instance (IP+port, REST endp, ...)
• **Load-balance** between micro-service instances
• **Auto-scale** (start/stop) m-s instncs based on load
Example: **Kubernetes auto-scaling:**
• Helps build **elastic** apps.
Tenant: describes what they want:
type: Resource **Operator**: monitor, load-balance, life-cycle service instances
name: cpu • Example: HTTP request
target:
type: Utilization
averageUtilization: 60 **System Perspective:**
• Simple for tenant: they just declare the service within a container image.
• Operator manages per tenant VMs & per service containers. They provide the service runtime and monitor reqs / manage instances according to load.
• Dispatch to hot instances
The Standing Problem:
• Starting and stopping serverless instances is expensive: container creation expensive (OS-level virtualization; must setup name spaces for the file sys, setup users, PIDs, network...). Boot takes time.
4) Function-as-a-Service (FaaS):
• Decouples tenant logic from service!
• Tenant only provides business logic, and **trigger:**
A stateless function. SLA usually says function can be killed after some time:100ms->~4s
• Operator provides and manages the rest: VMs, Containers and Runtime.
• Tenants use separate VMs for security. One container+runtime per func for reproducibility.
• Set of pre-def runtime w/versions (python3, NodeJS)
System Perspective:
Very limited set of VM/container/runtime configs
• Provided by the operator → deep integr w/ecosys
• Easier to predict, prefetch, and reuse
• Tenant can be bound to reused inst. at runtime
• Ongoing research to speed up instantiations and understanding relationships. Privacy perf trade.
Cutting Edge Methods for Serverless Boot Time:
• Serverless instance classes, on temporal locality
Hot: instance is actively used, not a problem
Warm: exists on sy; not been recently used –
Cold: no such serverless instance on the system
Prefetch I-cache and TLB entries from memory
• Excellent for warm starts (speedup: 1.2x)
Prefetch image contents from storage
• Excellent for cold starts (large speedup: 3.7x) –

9.3) Communication Mechanisms:
From Application to Network transport, need define:
• Message Semantics (retrieve image)
• Application API (bitstream get_image(name))
• Transport Protocol (TCP/IP)
• msg format (struct get {int size;char[] name;};)
• Produce code to go b/w app API and transport
Need to consider low-level raw bit manip code, encryption compression. Canned solutions exist.
1) Web Based Technologies:
HTTP REST + JSON most common:
• REST: representational state transfer. Self describes resources and operations
• It's a Server API on top of HTTP. Text based (easy debug), lots of frameworks (Rails, Flask).
• Defines loose principles: HTTP cmds and paths associated with server ops (get users: "GET /users")
Headers for caching, statelessness, layering.
2) Remote Procedure Calls:
Server API defined as Function Calls.
• Can be asynch. Binary based; more efficient to transfer and compress. Frameworks: gRPC.
Easy to use model. Framework actually does the heavy work by filling out the functions.
• Service developer defines RPCs as funcs.
• RPC dev generates client/server code stubs;
bridges application functions to/from network msgs.
• Client only deals with calling stubs, server with function call handlers, and actually invokes the code
Example:RPC API: defined as a function signature:
service Greeter {rpc SayHi (HiRequest) returns (HiReply) {}} message HiRequest { string name = 1; } message HiReply { string message = 1; } RPC compiler generates client/server code (stubs) Client: calls RPC-generated stubs – auto reply = greeter.SayHi("hi!");
The Standing Problem once again: we spend a lot of time on data buffer transport alone.
3) Remote DMA: Stream: send / receive like sockets at user-level directly from program to NIC. Kernel no longer mediates access to network data.
• Memory: r/w. RDMA read from remote buff into local mem addr. To handle memory access: App pre-regs mem buffer using OS. OS driver does vmem/paddr translation. NIC associates translations to buff_id and offset, we use the NIC's trans for that buff. Page faults: make NIC invalidate trans (expensive)!Two modes: un/reliable ala: TCP vs. UDP. Observations: We just bypassed OS again (directly use hardware): move OS net stack to NIC
• Less CPU load on both sides • No CPU load on r/w
Hard to program; need exch buff IDs ahead of time
The Data Center Tax: Systems are getting larger with more complex communication. Complex data management is now taking up to 30% of CPU. No solution; RDMA or accel engines help mitigate it.

9.4) Taming Latency and Cost at Scale
Perf Goal: **Min Tail Latency** (worst case latency)
Tradeoff: reduce utilization to reduce tail latency, but this increases costs. Can also improve **loading.**

9.5) Load Balancing:
"Distributing a set of tasks (requests), over a set of resources (cores, machines)" Multiple factors:
• Time and information: static vs. dynamic
• Scale: cross-core vs. cross-machine
Queueing Theory: 3 param define service latency
• Arrival Distribution: Poisson (time b/w arrivals)
• Arrival Assignment (who/when assigned process)
• Service Time Dist (time to process a request): ideally constant, realistically exponential, bimodal (heterogeneous). • We can statistically model cumulative effects of arrival and service.
• Params: single queue vs multi queue; FCFS vs PS (processor sharing)
• Goals: Low tail latency, cores are never idle when there is work to do. **Head-of-Line (HOL) blocking: bad** • Reduces util; found in switching nets; if first packet isn't ready then all behind blocked **Examples Scale Up Sys / Research:**
1) IX (Presented at OSDI'14 conf):
• Massive improvement: ↓tail latency ↑throughput
• Multi queue system: 32x queues/G/1 cpu/FCFS.
• Perfect on constant/exponential service times
• System designed to "narrow" the exp
2) ZygOS (SOSP' 17): Almost single queue sys
• Work conserving; avoids HOL blocking
• Supports large service dispersion gracefully
3) R2P2: Towards single queue for all instances. Global queue when establishing conn; regular client / server conn after. Doesn't rebalance long-lived conns. Reduces tail lat thanks to overloaded nodes.
4) Load-proportionality: scale up: PEGASUS:
• Monitor app metrics; use DVFS to speed core up/down based on power.
Summary: Load Proportionality; Scale Out:
• Waste if nodes underutiled. Exploit elasticity to add/rem service instances according to load.
• Micro-service arch makes this simple (self-contained indep services) • Serverless and FaaS make it efficient (faster boot, reduced sys image size) • FaaS adaptable with short lived funcs.

10) Specialised Devices
• Unlike general purp CPUs; specialised do one task but do it quickly & with low energy.
• Past examples: FPUs, I/O management processors. Today: DMA engines – break down large data transfer into smaller ones; embedded on most devices; available on some CPUs.
• Metrics: Energy (E), Time (T), E*T, $E*T^2$
• For HPC: T, or $E*T^2$ are useful metrics
• Mobile phones: E, or E*T
General Purpose CPUs: High freq, out-of-order speculative super-scalar pipeline, large cache hierarchy but with much slower memory.
• And app w/large working set and random access
• Goal: make a CPU that runs same prog cheaper
Sources of energy consumption:
• Being general purpose rather than specialised (e.g: software prog vs hardware impl)
• **Transferring Data** (Dist,bus complexity,hops)

10.1) Types of Specialised Device
1) **Acceleration**: CPUs waste time/energy in complex algorithms. Lots of redundant data mov to/from regs; lots of untapped parallelism. Specialised Hardware for Specific Operations:
• Avoids temporary data movement: (data directly flows through compute logic)
• Exploits fine grained parallelism **in hardware**
2) **Integration**: Delay and energy wasted moving data back and forth from regs, cache, DRAM, swapspace... Instead, spend transistors to integrate devices into the chip directly! Chip interconnect -> PCIe; shorter dist too
3) **Move computation into the data**: we need external mem, but accessing this is expensive. Data doesn't fit inside the CPU: we need external mem, but accessing this is expensive.
• Avoid moving data by instead computing where it is stored! Doable as mem device will have a lot more internal parallelism/bandwidth.

10.2) Examples of Specialised Device
FPGA/ASIC: way more efficient, harder to prog
• Hardware is fully parallel
• FPGAs: reprogrammable (hard), slower than ASIC
• ASIC: logic designed for 1 purpose; most efficient
More programmable but less efficient than these:
• We can add accelerators to each component:
1) PIM (processing in memory) -> Mem:
• Use properties of new mats to exec simple arith: Massive parallelism (each stored bit is a parallel circuit), no extra logic. Done with new tech/mats.
2) Near Memory Processing (NMP) -> Mem
• Memory has parallelism at many levels (channel, rank, bank, can put logic in any one of them)
• Has more internal thruput (c,r,b, used parallel)
• Needs less external thruput (e.g. reduce/filter op)
3) GPUs (remember ACA):
• Huge Mem Bandwidth • Hugely parallel
• Program -> thread blocks -> warps -> threads
• Multiple threads scheduled by hardware (using thread blocks), cheap context switches when waiting for memory (asynch conc) across warps
• Very efficient for MatMul for ML.
4) Near-data processing (NDP) -> Storage
• Similar to NMP; limited PCIe bandwidth, massive channel bandwidth
• Moves compute into the SSD: possible replicate across channels. Done with FPGAs: Samsung CSD, general purpose cores (NGD Newport)
• Useful for database filtering & text search
5) Smart NIC (Network Interface Card)
• Compute between network & PCIe/CPU
• This gives additional parallelism (NIC+CPU)
• Network-specialised accelerators (crypto, FPGA)
• Can reduce data traffic (NIC-CPU)
• Can be transparent to CPU applications
• Useful for security analysis and detection (IDS) data caching (for hot data or requests).
• **Example:** AWS Nitro: deployed on most AWS nodes. Nitro replicates HW/MMIO.
• VM has same performance as non-virtualized (native IO) as a result despite virtualization
• Thus AWS tenants don't pay virtualization costs, and AWS operators can change their impl at any time.
6) Smart Switch (Network/Cloud level accel):
• Makes switch logic more flexible (generalized typical ops) Match/action pipeline:Match headers / contents → IP==1.1.*.* and port==1234
Act on matches → IP := 2.2.2.2
• Used for impl new protocols, caching, other app specific logic in the network
Example: Barefoot Tofino:
• Generic streaming design on programmable hardware pipeline with new prog language (P4)
• Can go as far as implementing application collectives (e.g., map/reduce)