1 Introduction

SPE is about fulfulling **non-functional** requirements. To define an appropriate optimsiation level, define a **target metric** (e.g. *latency*). Either set an **optimisation budget** (e.g. *devtime*) or **optimisation target/threshold**, aka **quality of service (QoS)** objective, (*the statistical props of a metric that hold for the system*). A **Service Level Agreement (SLA)** is a *formal, legal contract specifying QoS objectives as well as penalties for violations*. Requirements must be **SMART**:

- **Specific**: *State what is acceptable in numeric terms.*
- **Measurable**: *Can be quantified and tracked.*
- **Acceptable**: *Rigorous enough to guarantee success.*
- **Realisable**: *Lenient enough to allow implementation.*
- **Thorough**: *Ensure all aspects of a system are specified.*

1.1 Measuring Performance

We have to measure an *actual (possibly prototype) system*. Often costly & based on instrumentation. We can either:

- **Monitor**: constantly required to enforce SLAs. Incur cost & often not continuous.
- **Benchmark**: first get sys into predefined **steady** state, then do ops (**workload**) while measuring perf. Can be outside prod. A workload can be **batch** (e.g. *query set*) for measuring throughput, or **interactive** (uses *driver* that generates reqs) for latency.

Alternatively, **model** perf using **analytical models** or **simulations**. We could also use a hybrid approach.

1.2 Optimisation Opportunities

Parameters (aka **resources**) are the sys and workload characteristics that affect perf:

- **Sys** params: dont change while system runs (e.g. *CPU*, *instruction costs*).
- **Workload** params: change while system runs (e.g. *users*, *available memory*, *clock rate*).

Parameters can be *numeric* / *nominal* (e.g.*runs on battery*, *has GPU*).

1.3 Utilization & Bottlenecks

Utilization is the % of the available resource used to perform the service. A **bottleneck** is the resource with the highest util e.g. *CPU-bound*).

1.4 Code Paths

Systems have **perf dominating** codepaths. Optimising these gives most perf gain for least effort. **Critical path**: sequential part of the code. **Hot path**: path that takes the most time.

2 Profiling

To find optimisation opportunities we can identify the *hot path* or **bottlebeck**. These are funcs of sys behavior, described by **events**, or *change in sys state*. This is usually restricted to a certain granularity. Events can be:

- **Simple / Atomic**: e.g. *sent package*, *executed instruction*, *clock ticked*.
- **Complex**: e.g. *cache line evicted*, *instruction aborted due to misspeculation*.

Events may have a **payload** and **accuracy** (*the degree to which the payload represents reality*).

Event sources have a **generator** (observing sys state) & **consumer** (processing events). Can be *online* (real-time) or *offline* (post-mortem).

2.1 Tracing

Trace: *complete log of all state the sys has been in*, ordered list of events. Accuracy is *inherited* from the events. Collection overhead is high.

Call stack contains *stack frames*, containing vars, return addrs & ptrs to the underlying stack frame. Recording entire call stack is expensive, as CPU must walk stack on every func call/return. For small cheap funcs, this overhead is significant. This causes **perturbation**.

Perturbation: the effect of the measurement process on the sys being measured, distorting results. To prevent, reduce **fidelity** (the amount of detail recorded) with **sampling**. Either sample *regularly* or *randomly*. If we skip events, we can miss a critical event. Fortunately, these run for longer & are more likely to be sampled. **Sampling intervals** is the distance between two samples being taken. This can be:

- **Time Based**: set a hardware timer & sample when fires. Use CPU *reference cycles* as proxy metric. Inaccurate, nondeterministic & noisy; easy to interpret.
- **Event Based**: in terms of occurence of an event. Accurate, deterministic & less noisy; hard to interpret.

Quantisation Errors occurs when interval resolution is limited and time is continuous. They mean that some events may be missed or misattributed.

Indirect Tracing: Some trace events are **dominated** by others (*executed depending on their outcome*). To reduce overhead, trace only dominating events and infer dominated ones.

Profiles represent info on characteristics of a sys in terms of the resources it spends in certian states. Is an *aggregate* of events over a metric (e.g. *total cache misses*). Profiles **lose information**, but can reduce perturbation in real-time.

FLAME GRAPHS: X-axis shows stack profile population, sorted alphabetically (not by time). Y-axis shows stack depth. Each rectangle represents a stack frame, and its width is the number of collected samples. Colours are arbitrary, used to distinguish funcs.

Event sources should be **detailed**, **accurate** and cause *little perturbation*. Either:

- **Software**: using a *library* (manual instr) / logging), using a *compiler* (automatic instr), or with the *OS* (kernel counters).
- **Hardware**: using *performance monitoring counters* (PMCs). Expensive on the CPU but no observable perturbation to software.

2.2 Instrumentation

Instrumentation: augment program with event logging code. High overhead, but no need for hardware support and very flexible. Can be:

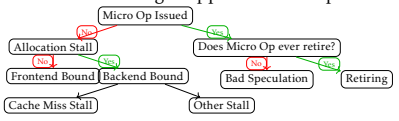
- **Manual** (using libraries to log). Gives fine control, no hardware / compiler support required. But, high overhead for impl & runtime, disabled for release build. Needs recompilation for selective instr.
- **Automatic** (using compiler to insert code). Less control & needs compiler support.
- **Binary Instrumentation** (modifying binary to insert logging code). No source code / recompilation needed, but needs sophisticated tools. **Static** (modify before exec) is simple & portable but inflexible. **Dynamic** (modify during exec) is flexible & works with JIT but complex & platform dependent.

2.3 Performance Counters

Software Perf Counters (OS): kernel-level counters logging events (e.g. *context switches*, *page faults*). Low overhead, but limited accuracy & detail. **Hardware PCs** are special regs that can be configured to count low-level events like *cache misses*, *branch mispredictions*. Buggy, poorly documented & can have bad accuracy.

2.4 Bottleneck Analysis

CPUs can stall on control deps (*branch mispredictions)*, due to a lack of compute resources, & on data access. To analyse, count cycles during which certain things happen. For example:



3 Modelling

If we lack hardware, code or data to perform experiments, use **modelling**. We need to make:

- **Input** assumptions (e.g. data follows distribution).
- **System** assumptions (e.g. dont model system noise).
- **Code** assumptions (e.g. code is deterministic).

3.1 Numerical Modelling

Numerical / Experimental Model: series of datapoints describing the observed sys behaviour are collected, model is fitted to data.

1. **Gather data**: ∀ param setting, run **microbenchmark** (small, specially designed to test specific portion of sys) multiple times to get avg perf metric.
2. **Interpret**: fit model to data (*e.g. interpolation*).

```
extern int* input; extern size_t N;
extern size_t stride; // Experiment param
int sum = 0; for (size_t i=0;i<N;i+=stride)
  sum += input[i]; // Benchmarking mem access
```

These are **easy to get** (if sys available) & based on **ground truth**. But **generalizes poorly** (*cannot be applied to new envs*), requires **massive experimental data**, has **limited accuracy** in prediction (*data may be missing/inaccurate*), **limited interpretability** & **limited insight**.

3.2 Analytical Modelling

Analytical Model: formal characterization relating params & perf metrics. Require **detailed understanding** of sys & **validation** (*results always questionable*). Can be very complex to deal with edge cases. *The line is blurry*, but analytical models can model behaviours that numerical models cannot. These have:

- A **Characteristic Equation** (*with params*): describes target metric behaviour or a sys in dependence of a varied param.
- **Values of sys params** (*e.g. access latency, cache sizes*).

3.3 Modelling Memory

- B_0: Size of a General Purpose Register of the CPU
- l_0: Access latency of level 1 cache
- C_0: Capacity of a General Purpose Register of the CPU
- B_1: Size of a cache line in level 1 cache
- l_1: Access latency of level 2 cache
- C_1: Capacity of level 1 cache
- B_2: Size of a cache line in level 2 cache
- l_2: Access latency of level 3 cache
- C_2: Capacity of 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 times page size

In general, B is the access granularity, l is the miss latency and C is the capacity of a memory level.

With stride s, a characteristic non-linear eq for **mem access time**:

$$T_{\text{mem}} = l_3 \cdot \min(1, \tfrac{s}{B_3}) + l_2 \cdot \min(1, \tfrac{s}{B_2}) + l_1 \cdot \min(1, \tfrac{s}{B_1}) _0 \cdot \min(1, \tfrac{s}{B_0})$$

Alternatively, based on the size of the data accessed, S:

$$T_{\text{mem}} = \begin{cases} l_0 & S < C_1 \\ l_0 + l_1 & S < C_2 \\ l_0 + l_1 + l_2 & S < C_3 \\ l_0 + l_1 + l_2 + l_3 & \text{otherwise} \end{cases}$$

Some parameters can be read from documentation, but some require experimentation to find. Self tuning systems require *less work*, are *more resilient*, *scale forward* (work on future architectures) and can be *more accurate*.

3.4 Access Patterns

```
extern int* input1;//len=1024, uniform rand data
extern int* input2;//len=64, random data
int sum = 0; for (size_t i=0;i<N;i+=stride):
  sum += input2[input1[i]];
```

Can use params to capture behaviour of a mem region:

- $R.n$ the **length** (i.e. *number of stored tuples*).
- $R.w$ the **width** (i.e. *size of each tuple in words*).
- $\|R\|$ the **size** (i.e. $R.n \times R.w$).

u = num words read per access. We also have operators:

- $\mathcal{P}_1 \oplus \mathcal{P}_2$: sequential exec of access patterns $\mathcal{P}_1, \mathcal{P}_2$.
- $\mathcal{P}_1 \odot \mathcal{P}_2$: concurrent (**interleaved**) exec of $\mathcal{P}_1, \mathcal{P}_2$.

For the code above, we can use access patterns `s_trav` and `rr_acc` to model the access pattern as:

$$\texttt{s\_trav}(R.w = 1, u = 1, R, n = 1024) \odot$$

$$\texttt{rr\_acc}(R.w = 1, u = 1, R.n = 64, r = 1024)$$

3.5 Modelling Stateful Systems

Some effects / components have dynamic **state** that influences behaviour & perf. **Markov Chain**: *finite state machine* with *transition probabilities*. **Markov Property** states that the next state depends only on the current state & a random variable, not the history of prev states. To model *branch prediction*:

```
extern int* input; // uniform rand data
int sum = 0; for (size_t i = 0; i < N; i++):
  if (input[i] > 20) sum += input[i];
```

We can model a **saturated counter** with 4 states:

- S_0: Strongly Not Taken
- S_1: Weakly Not Taken
- S_2: Weakly Taken
- S_3: Strongly Taken

The probability of it being in any state is a **stationary distribution**. Here, the branch misprediction rate is $(P(\text{pred\_taken}) \cdot P(\text{act\_not\_taken})) + (P(\text{pred\_not\_taken}) \cdot P(\text{act\_taken}))$.

Modelling can be infeasible (*scale & noise*). We:

1. Identify *bottleneck* using a profiler.
2. Recreate behaviour in a controlled environment (e.g. **microbenchmark**).
3. **Model** & **Validate**.

4 Efficient Code

CPU bound apps are **poorly impl**, operate on **small data**, are **math heavy** or apply **memoriented optimisations**. The target metric is **wall clock time** (*time taken to exec program*). **Stall cycles** = good indicator of CPU efficiency.

- **Control Hazards**: due to data-dependent changes in control flow.
- **Structural Hazards**: due to lack of exec resources.
- **Data Hazards**: due to operands unavailable on time.

CPU efficiency strongly influenced by pipeline design:

- **Speculative**: keeps pipeline full even if no instrs are eligible. *Addresses some control hazards, but occasionally needs to flush pipeline.*
- **Superscalar exec**: diff instrs may be in same stage.
- **Out-of-order exec**: exploits instr independence to execute intrs as soon as operands ready, not following program order. *Addresses data & structural hazards.*
- Executing **statically parallel** (*e.g. SIMD*)? Allows performing the same operation on multiple data points simultaneously. *Addresses data hazards.*

Common techniques include:

- **Partial Evaluation**: precompute bits when inputs known at compile time. E.g. *funcin lining*, *JIT* & *constant folding*.
- **Loop specialization**: loop transformed to handle specific cases more efficiently. Specified for compilers with **metaprogramming**.
- **If-conversion / Predication**: removes ctrl haz by converting cond branches to straight code with predicated instrs. Can lead to worse perf (*when cond is rarely true or when predicated instrs are more exp than original branch*).
- **SIMD Vectorization**: perform same op on multiple data simul, improving perf on data-parallel tasks. Not always beneficial (*if data not well-aligned or overhead of setting up SIMD ops outweighs the perf gains*).

4.1 Data Hazards

Caches have fixed line size (e.g. 64B), & fixed capacity. When **cache miss** occurs, pipeline stalls (**data hazard**). If data previously accessed, **capacity miss**. Else, **compulsory miss**.

- **Fully mem bound** if all stalls are data hazards.
- If memory bus fully utilized, its **bandwidth bound**.
- Otherwise, its **memory latency bound**.

If code latency bound & cache misses compulsory, **prefetch**: speculatively load next line by *recognizing access patterns* (**hardware prefetching**). Breaks for irregular accesses. **Software P.** explicitly prefetches data, optimising irregular access patterns. Done with *CPU instrinsics*.

If code bandwidth bound, increase **cache line util**: $\frac{\text{Data Req by Instrs}}{\text{Loaded in Cache}}$. *Changing data layout* to ensure more data loaded into cache is actually used by instrs.

4.2 Reducing Cache Footprint

> Capacity misses are caused by **thrashes**: when the working set of data exceeds the cache capacity, leading to frequent evictions and reloads of data.

Loop tiling breaks down loops into smaller tiles, allowing for better cache util & reducing cache misses. Processing data in smaller chunks that fit into cache minimizes thrashing. **Multipass Algorithms** are more efficient if they reduce cache footprint.

4.3 False Sharing Hazards

MESI (Modified, Exclusive, Shared, Invalid) is a *cache coherence protocol*, ensures consistency of data across > 1 cores. It defines states of cache lines & transition rules:

- **Modified** and only present in the current cache. Must be written back to mem before eviction.
- **Exclusive**: unmodified and only in current cache.
- **Shared**: unmodified and may be in multiple caches.
- **Invalid** and cannot be used.

False Sharing is a **control hazard** where > 1 threads access different vars on the same cache line, leading to unnecessary cache coherence traffic. To avoid, **pad** data structs.

5 Multicore & Parallelism

- **Data Level Parallelism** (DLP) - *e.g. vector instrs.*
- **Instr Level** (ILP) - *e.g. superscalar out-of-order CPUs.*
- **Task Level** (TLP) - *e.g. many threads/procs.* Became more important after *Dennard* scaling.

> **Parallelism**: do multiple computations simul. **Concurrency**: manage multiple computations sim, may or may not be exec simul. *Parallelism ⊂ concurrency.*

Parallelism is good for *perf*, *cost eff* (shares RAM, disk, etc). *Datacenters optimised for* ***total cost of ownership****, including hardware purchase, power, cooling, security & Mean Time To Failure (MTTF).*

> **Amdahl's Law**: Speedup = $\frac{1}{(1-P)+\frac{P}{N}}$ where P is the prop of the program thats parallelizable, N is num CPUs. As $N \to \infty$, Speedup $\to \frac{1}{1-P}$, so max speedup limited by sequential portion of the program.

5.1 Multithreading

Threads are concurrent, & share mem within a proc. If mapped to seperate cores, **parallelism**. In reality, low level ops are complex to program, and thread man / comm is expensive. Naive multithreading is as follows:

1. Start with a sequential application.
2. Apply existing tools for perf analysis.
3. Use Amdahl's law to identify what to parallelize.
4. Use multithreading to execute in parallel.

This is often wrong as it ignores **critical path** (*Sequence of tasks determining min time for an op.*). Often, the best choice is to parallelize the critical path, even if it has a smaller P, as it can lead to a larger overall speedup.

5.2 Communication, Synchronization

Explicit Sharing: threads use shared mem, need sync to avoid race cond. **Cache coherency** keeps data consistent across cores. For this, need hardware support for **atomic RMW**. Add a **locked** state to cache lines, which allows core to lock a cache line while its being modified, to stop others from accessing it until complete.

The universal atomic instruction is **compare-and-swap (CAS)**: `if (addr == old_val): addr = new_val`. Used to impl sync primitives. Hardware provides more atomic functions for perf, but CAS is enough.

Its possible to write code that produces incorrect states due to concurrency issues. This section of code is a **critical section**. Threads must sync to access critical section. In **exclusive** mode, one thread may access a critical section. In **shared** mode, many threads may access a critical section simultaneously, but only if they are not modifying shared data.

- **User Level Lock**: Use a CAS to signal when a lock is acquired. On lock, loop until CAS is *released* to *acquired*. On unlock, set the lock to *released*. This wastes cycles, can cause **thread starvation** (as no fairness guarantee on acquire order).
- **Kernel Level**: Use a **futex** to avoid busy-waiting. This syscall reschedules thread if blocked trying to acquire. This is expensive if already released (`CAS + syscall` vs `CAS`). But, it keeps fairness/order of blocked threads.
- **Hybrid**: Try to acquire the lock using CAS, fallback to futex if fail. Avoids syscall overhead in common case where the lock is not contended, but avoids starvation otherwise.

False sharing: When many threads access diff vars on **same cache line**. To avoid, **pad** structs (*mem tradeoff*).

5.3 Thread & Task Management

On Demand threading creates a thread for each incoming task onto the system:

```
void process(const vector<size_t>& jobs):
  for (size_t job : jobs):
    thread t([job, &]() { process_job(job); });
    t.detach();
```

Simplest parallel approach, but expensive if `num_threads > num_cores`, especially when holding locks. Could create new threads ∀ batch:

```
void process(const std::vector<size_t>& jobs):
  std::vector<std::thread> threads;
  for (size_t j : jobs) threads.emplace_back
      ([&]() { process_job(j); });
  for (auto &t : threads) t.join();
```

Here, each batch runs its tasks in parallel, but frequent creation / destruction causes overhead.

Instead, create **thread pool**: fixed num of **worker threads** created at startup, tasks added to queue for workers to process. This is **thread dispatching**:

```
size_t i; vector<worker_deque> workers;
void process_burst(const vector<size_t>& jobs):
  for (auto job : jobs):
    workers[i++ % workers.size()].add_job(job);
```

Minimal queue contention, but *event driven*, & producer unaware of imbalance. **Work stealing**: shared job queue that consumers pull from:

```
worker_deque queue;
void process_burst(const vector<size_t>& jobs):
  for (auto job : jobs) queue.add_job(job);
```

This has *optimal consumer balancing*. But, its *event driven* so expensive queue manipulation due to contention.

5.4 Streaming & SEDA

Streaming (*pipelining*). Each *func (stage)* given a thread, each task an input queue. Funcs enqueue res into next func's queue. *Event driven* so exp queue man, *bad locality* for IO data, but good *code & temporary data* **locality**. **Staged Event Driven Architecture (SEDA)**: allows many threads per stage with dynamic load balancing. Each stage has a thread pool, tasks are enqueued to stages based on their type. Better perf under load, but complex to impl and man. Programmer defines stages & tasks, SEDA framework handles rest.

5.5 Multiprocessing

Multiprocessing uses many procs to solve a problem. Same concurrency / parallelism but has comms & sync tradeoffs:

- **No implicitly shared memory** by default.
- Communication is **explicit** and **expensive**.
- Allows multiple tasks that *execute independently* and have simple, explicitly expressed *communication*.

We can assign processes to each group of tasks. Without shared mem, **comm** is key:

- **System overheads**: syscall, scheduling & memcpy.
- **Programmability**: cannot pass complex structs, so need serialization (*expensive & error-prone*).

We could use *explicit shared mem* (support zero-copy, no std interface, local-only), *sockets* (intermediate copies, network stack processing even when local) or *pipes* (intermediate copies, same read-write interface, local-only).

6 Algorithms

- **Blocking** alg use locks to protect shared structs. Simple to implement but cause contention & deadlocks.
- **Non-blocking** alg: *suspension of any thread cannot cause suspension of other threads*. CAS but no locks. **Lock free**: 1 thread guaranteed to progress, **wait free**: all threads will progress.

6.1 The ABA Problem

CAS can fail to detect changes if the value changes from A to B and back to A. Solutions include:

- Using **intermediate nodes** (`shared_ptr`, expensive).
- Using **tagged pointers** (e.g. 128-bit CAS with a version number). Suffers from *wraparound problem*: version number can overflow (less likely than ABA).
- **Defer reclamation** using `read-copy-update (RCU)` or **hazard pointers**. Allows threads to safely access shared data without locks, but very complex.

7 System Interfaces

Abstractions to make *developers productive*, **multiplex** (*unused hardware is a wasted purchase & energy*). Also consider **cache** (*storage $ increases with perf*) and to **avoid comm** (*expensive*).

7.1 OS Interfaces

OS impl protected **objects** inside the *kernel*, that mediate all accesses (e.g. *sockets*, *files*, etc). EXPENSIVE! A syscall:

1. [USER] Store arguments in registers.
2. [USER] Execute `syscall` instruction.
3. [KERNEL] Santize environment.
4. [KERNEL] Save state.
5. [KERNEL] Execute syscall handler.
6. [KERNEL] Restore state.
7. [KERNEL] System return instruction.

Instead, use **library OS**: the OS kernel as a lib to a **single app**. Very adaptable to app needs, but elimates privelege separation overhead.

7.2 Memory & Paging

Fast storage is expensive, so **limit size** by **caching**. However, a **cache miss** needs expensive copy time, so reduce misses by predicting *access patterns*.

7.3 Security & Syscall

Buffered IPC: kernel mediates comm between procs. Simple but expensive. Must maintain *2 copies* of data in userpace & kernel space. However, *non-blocking* if buffer available.

Unbuffered IPC: procs comm directly. More efficient but complex. *Blocking* if other procs not ready to receive/send data.

7.4 CPU Interfaces

The **instruction set architecture (ISA)** also has tradeoffs in:

- *Security / Performance* (privelage levels).
- *Complexity / Performance* (memory consistency models).
- *Adaptability / Performance* (specialized hardware vs volume cost).

7.5 Caching

A CPU chip has many caches which exploit **spatial** & **temporal locality**. ISA extended to control & flush entries for cross core consistency. Software must be optmised to maximise cache hits. We have:

- A **hardware context cache** (L1, L2 ... LLC).
- An **MMU** cache (TLB) to trans virt addr to phys addr.
- MMU also has **partial walk cache (PWC)** which caches intermediate PT entries to speed up addr trans when missing in TLB. Each level in MMU PT has separate PWC, and OS can ctrl each PWC size. All 4 caches can be accessed in parallel.

Must keep mem trans caches (TLB, PWC) consistent on **page downgrade** (e.g. from read-write to read-only). **TLB shootdown**: OS sends an inter-processor interrupt (IPI) to all cores to invalidate TLB entries for a page that has been downgraded. Expensive & scales poorly with core count. To mitigate:

- Avoid downgrading pages.
- Asynchronous downgrade syscalls.
- Hardware acceleration.

7.6 NUMA

CPU connected to mem through *extensible bus*. **Non Uniform Memory Access (NUMA)**: each CPU has its own local mem. Usually paired with **cache coherent interconnect** (e.g. Intel QPI). Accessing own mem always faster, so write code with *good locality*.

7.7 Virtualization

Allows mult virt machines to run on a single machine, reducing cost with higher security than containers / procs. But *VM except* are **expensive**, with long microcodes to switch between guest/host. I.e. *hypercalls*, *trapped instrs*, *trapped mem* & *interrupts*. A VM exit:

1. [HOST KERNEL] Trapped instr / interrupt.
2. [GUEST USER] Santize environment.
3. [GUEST USER] Save state.
4. [GUEST USER] Execute handler.
5. [GUEST USER] Restore state.
6. [GUEST USER] VM resume instruction.

VMs use nested PT, so a PT traversal goes from 4 to 24 mem accesses (as each virt PT itself must be trans). To fix:

- Let hypervisor manage PTs (**shadow paging**). Requires VM-exits to manage guest PTs.
- Use *6* guest-host **page walk caches**.

7.8 IO Interfaces

Describe data going in/out 2 (e.g. *network*, *storage*, *GPUs*, etc). Modern devs accessed as mem using **mem mapped IO (MMIO)**. Dev is assigned a range of phys addr at boot time, corresponding to ≥ 1 PCIE **bus addr ranges (BARs)**. *Hardware uses addr to distinguish between DRAM & BARs.* A **device spec** describes which offsets in the BARs correspond to which dev funcs. A modern dev may have both:

- **Circular Qs** with req/res buffs in *host mem*.
- **BAR** to check/config Q changes w/ **MMIO**.

7.9 Device Interconnect

PCIe is a common interconenct for devs, with $\approx 1\mu s$ latency. Uses **MMIO** and *interrupts* as messages. *dev-to-mem has same perf & locality problems as cpu-to-mem, with same solutions.*

7.10 Device Models

An **interrupt driven model** works well but has context switch overhead:

1. MMIO writes to program device.
2. Dev reads/writes request/response buffs.
3. OS waits for an interrupt.
4. MMIO reads to check result.

A **polling model** has no context switch, and can give an answer faster (*with short polling time*):

1. MMIO writes to program device.
2. Dev reads/writes request/response buffs.
3. OS repeatedly polls MMIO to check result.

A **hybrid model** polls for some time, then tells dev to interrupt if not done. More complex but has better perf.

7.11 Device Virtualization

Accessing a virt dev is a mem trap. By default, **trap & emulate**, where hypervisor unmaps dev **BARs** to trap **MMIO** accesses, and emulates dev operation on every trap. **Extremely expensive**, requiring many VM exits per dev op.

Paravirtualizing the dev maps shared mem between VM & hypervisor. A single trapped MMIO access notifies the hypervisor of new reqs (*doorbell req*). Much faster, but requires changes to guest OS & dev drivers.

Passthrough maps dev **BARs** directly to VM, and configures IOMMU to allow VM to access the dev. Fastest, but eliminates isolation between VMs, causes security issues, & *requires hardware support from IO devices to have a different state per guest VM*.

7.12 OS level interfaces

Traditional **blocking interfaces** (POSIX `read`/`write`) are simple but incur high overhead. Every call requires a hardware transition to kernel mode, ≥ 1 data copy, and up to 2 context switches if thread must be rescheduled while waiting.

Non-blocking int. eliminate context switches & data copies to preserve cache locality.

- **Asynchronous:** App submits a *start-of-op* and continues other work until kernel signals completion (e.g. Linux AIO for storage).
- **Event-based:** App expresses interest in specific `fd`s and only operates after polling the kernel to confirm the op will not block (e.g., Linux `epoll` for networking).

For max efficiency, **direct dev assignment** (*passthrough*) allows user-mode apps to do MMIO directly to a dev's virt funcc. Bypasses OS entirely, eliminating syscalls & intermediate copies. Apps typically use specialized libs like **DPDK** (networking) or **SPDK** (storage) to manage these devs.

8 System Programming Models

Thread per Task is *simple*:

1. [MAIN] Listen for new connections.
2. [MAIN] Accept a new connection.
3. [MAIN] Create thread to handle conn.
4. [WORK] Do all blocking ops (*read req, read-/write disk, write network response*).

Under high concurrency, leads to **resource exhaustion** (too many threads) and **context switching overhead**.

8.1 Worker Pools

Pre-allocate ≥ 1 thread pools, avoiding costs of on-demand thread creation. But, no clients processed when workers blocked; still has context switching overhead when # CPUs > # threads:

1. [MAIN] Listen for new connections.
2. [MAIN] Accept a new connection.
3. [MAIN] Enqueue conn to a worker pool.
4. [WORK] Dequeue conn, do blocking ops.
5. [WORK] Return to pool, wait for next conn.

8.2 Event Based

Typical of IO heavy network servers. **Events** used to concurrently handle multiple ops:

- Preallocate and pin **one thread per core**.
- Use **non-blocking IO** and an **event loop** to handle multiple connections in each thread.
- Keep a **state machine** $\forall$ concurrent contexts.

No thread sched & minimal mem usage (*state machine $\ll$ thread stack*), but more complex.

9 Scale Out

Increasing perf by adding more machines. **Scaling up** adds more rsrcs to a single machine.

9.1 Metrics

Take into account **tenant** (*client*), who has a **service level objective (SLO)** (*what they wants*), caring about *performance / $* & *latency*. The **operator** has a **service level agreement (SLA)** (*what tenant pays for*), caring about *infra cost*. An SLO defines *target metrics* and their *perf*, by studying the app's **critical path**. Derive a metric SLO $\forall$ components.

- **Latency** varies, so define a **latency SLO** as a percentile of the latency distribution (*e.g., 99th percentile latency < 100ms*).
- Ideally, **energy consumption** $\propto$ **util**, but in reality $\exists$ fixed energy cost $\Rightarrow$ hardware isnt energy proportional. Define an **energy SLO** as target energy per req (*e.g., < 1 J/req*).
- We want to *almost max* **util** to min cost, but also need to maintain headroom for traffic spikes to avoid perf degradation. Define a **util SLO** as target avg util (*e.g., 70% average CPU utilization*).

9.2 Tools

Components comm over the network. To max CPU perf, use **asynchronous communication**.

An **elastic system** scales up/down in response to load. Requires **automated scaling** based on monitoring metrics (*e.g., CPU util, req latency*).

9.3 Scale Out Architectures

To scale out, *break app up into concurrent components* by defining async comm APIs and protocols. Then deploy components on different nodes elastically (*different # of procs per component*). Handle failures by monitoring & restarting dead processes. $\exists$ **productivity / perf trade-off** from scaling out. Achieve better performance by breaking up the app, but this increases complexity & reduces productivity. Made worse by unexpected failures and slowdown. Self-managing scale-out sys gives all benefits of consolidation (**lower cost**), but all **disadvantages of self-management** (*monitoring*, *maintenance*, *scheduling*, etc).

9.4 Microservices

An app is a **collection of microservices**: one service (*program*) per task. Sys capacity increased by increasing service instances, connected through a network. *Monitoring required to scale # of inst.*

Good application architecture (**modular**) but sysman is complex.(Must manage many **expensive VMs**: *long boot time*, *duplicate mem*). Additioanlly, complex management in *monitoring*, *VM image distribution*, *life-cycle management*.

Instead use **containers** (*lightweight*, *fast startup*, *shared mem*), but still have to manage container images & life-cycle.

9.5 Serverless

Here, tenant declares **what to run** (*function*), which is simpler to manage & pays by use. Operator handles **how to run it**, by **abstracting** microservice inst, **load balancing** between instances and **auto-scaling** based on load.

Don't set CPU util to 100% to allow time for catch-up on small inefficiencies. Simple for tenant, but operator must man **many short-lived instances** (**standing problem**), hard to **schedule** & **monitor**.

9.6 Function as a Service (FaaS)

Decouples tenant logic from service; tenant provides **business logic** (*function*) and **trigger**. Function is stateless with fine granularity. Operator provides & man **VM**, **containers** & the **runtime**. $\exists$ **limited VM/container/runtime configurations**, which simplifies man for operator (*deep ecosystem integration*, easy to *predict, prefetch & reuse*).

9.7 Communication Mechanisms

HTTP REST + JSON: server API on top of HTTP, many frameworks, easy to debug. Loose principles with headers for caching, statelessness, layering, etc. but **inefficient**.

RPCs define server API as function calls. *Can* be async, many frameworks. Operates on binary data. Simple to understand, with RPC framework generating client/server code stubs. But, **tightly coupled** to the server API, and **difficult to debug**. Both REST & RPC transfer *data buffs* taking many cycles (**standing problem!**).

RDMA moves **between mem of 2 machines** without involving CPU, reducing latency & increasing throughput. Contains *streams* (similar to sockets) and *direct mem access*. Can operate in **reliable / unreliable** modes (*similar to TCP/UDP*). Moves OS network stack into NIC, hard to program & debug, **tightly coupled** to server API. To allow this:

- App preregisters mem buff with NIC using OS.
- OS driver sends virt->phys mem trans to NIC (for the buffer).
- NIC associates translations to each buffer ID + offset.
- Reception of RDMA ops uses the NIC's trans for that buffer.
- Paging app mem out requries invalidating a NIC's trans (expensive!).

The **data center tax** says 30% of CPU cycles are spent on communication.

Reducing util reduces tail latency, and incr util reduces costs.

9.8 Load Balancing

Load Balancing: distributing tasks (*reqs*) over a set of resources. **Service Latency** is defined by **arrival distribution** (*time between arrivals*, typically Poisson), **arrival assignment** (*who & when is assigned to a process*) & **service time distribution** (*time to process a request*, ideally constant but typically exponential).

Queueing theory statistically models cumulative effects of arrival & service. Parameters: is it **single queue** (*all requests go to one q*) or **multiple qs** (*reqs are distributed across many qs*); is it **first-come-first-served (FCFS)** or **processor sharing (PS)** (*reqs are processed in parallel*). Goal is to **minimize tail latency** and **work conservation** (*core is never idle if there's work to do*).

Head of Line (HOL) Blocking: a request at the front of the queue is delayed, causing all subsequent requests to be delayed as well. This can lead to increased latency and reduced utilization.

9.9 Load Proportionality

When using *scale-up* model, monitor app metrics & determine when to scale up/down system. When using *scale-out* model, $\exists$ large waste when multiple nodes are underutil, so exploit elasticity. Simple with microservices, & efficient with serverless / FaaS.