60017

System Performance Engineering

Imperial College London

Contents

Chapter 1

Introduction

1.1 Logistics

Dr Holger Pirk

Dr Luis Vilanova

First Half

- Hardware efficiency in complex systems
- Scaling up
- *"getting the most bang for buck"*

Second Half

- Scale out Processing

1.1.1 Extra Resources

UNFINISHED!!!

1.2 What is System Performance Engineering

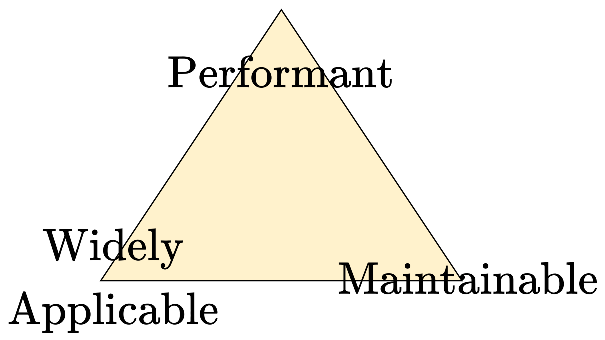
System **Definition 1.2.1**

A collection of components interacting to achieve a greater goal.

- Usually applicable to many domains (e.g a database, operating system, webserver). The goal is domain-agnostic
- Designed to be flexible at runtime (deal with other interacting systems, real conditions) (e.g OS with user input, database with varying query volume and type)
- Operating conditions are unknown at development time (Database does not know schema prior, OS does not know number of users prior, Tensorflow does not know matrix dimensionality prior)

Large & complex systems are typically developed over years by multiple teams.

The challenge with *system performance engineering* is to make systems maintainable, widely applicable and fast.



System Performance Engineering **Definition 1.2.2**

Performance engineering encompasses the techniques applied during a systems development life cycle to ensure the non-functional requirements for performance will be met.

- Functional requirements (correctness, features) are assumed to be met.
-

High Performance Computing **Definition 1.2.3**

High performance programming uses highly distributed & parallel computer systems (e.g supercomputers, clusters) to solve advanced problems.

- Focuses on solving a single computationally difficult problem.
- Workloads are well defined and known at development time.
- Sometimes supported by custom hardware (e.g FPGAs, ASICs, custom CPU extensions)

1.3 Performance Engineering Process

1.3.1 Metrics

A *target metric* is used to quantitatively measure any improvement in *performance* (e.g for use in a *SLA*). The metric needs to be wel defined:

- When measuring starts (e.g when to measure latency from)
- Where measuring is done (is server response time measured on server, on a client, under what conditions?)

Imperials **Example Question 1.3.1**

Provide some example of metrics regarding a database.

Latency	Measuring time to query, planning time, the whole systems response time over a network.
Throughput	Measure the maximum request/second possible (often used to compare web-servers)
Memory Usage	measurable, but must be careful (e.g os interaction)
Scalability	Can define a metric regarding how quickly some metric (e.g throughput) increases with scale (e.g instances of a distributed system)

It is also important to define when a requirement is satisfied.

- Setting an optimisation budget (e.g in developer hours)
- Setting a target or threshold (e.g $x\%$ over baseline implementation)
- Combination of both

1.3.2 Quality of Service (QoS) Objectives

Quality of Service Objectives **Definition 1.3.1**

A set of statistical properties of a metric that must hold for a system.

- Can include preconditions (e.g to define the environment/setup)
- Can be in conflict with functional requirements (e.g framerate vs realism in graphics)

Game On **Example Question 1.3.2**

Give an example of a basic QoS Objective for a game's framerate.

The game's framerate will be on average (over *... preconditions ...*) $60fps$ if run on a GPU rated at $50GFlops$ or higher.

1.3.3 Service Level Agreements

Service Level Agreements (SLAs) **Definition 1.3.2**

Legal contracts specifying *QoS objectives* and penalties for violation.

- Non-functional requirements (not about system correctness)
- Can be legally enforced

Amazon ***Extra Fun!* 1.3.1**

Amazon Web Services (AWS) provides a set of *service level agreements* relating to performance and availability. Violations are resolved by providing customers with service credits. Amazon SLAs

When defining requirements for an SLA:

Specific	State exact acceptance criteria (numerical terms).
Measurable	Ensure the metrics used can actually be measured.
Acceptable	Requirements should be rigorous such that meeting them is a meaningful success.
Realisable	Counter to **Acceptable** - need to be lenient enough to allow implementation.
Thorough	All necessary aspects of the system are specified.

1.4 Performance Evaluation Techniques

1.4.1 Measuring

- Performed on the actual system (can be prototype or production/final).
- Can be difficult and costly (need to mitigate any impact of the measuring system on the system itself).
- As it is on the actual system, it can (if done properly) yield accurate results.

The two main types of measurement are:

Monitoring **Definition 1.4.1**

Measuring in production to get real usage performance metrics.

- Observe the system in its production environment
- Collect usage statistics and analyse data (e.g user's preferred query types/structure, schema designs for databases)
- Can monitor for and report SLA violations.

Benchmarking **Definition 1.4.2**

Measuring system performance in a controlled setting (e.g lab).

- The system to set into a predefined (or steady/hot) state
- Perform some workload while measuring performance metrics.

Benchmarking requires representative workloads in order to get metrics likely to be representative of a production environment.

Batch Workload **Definition 1.4.3**

Program has access to entire batch at start of the benchmark.

- Useful when a throughput metric is being measured
- Simple to generate, and can even be recorded from a production environment.

Interactive Workload **Definition 1.4.4**

A program generates requests to pass to the system being benchmarked.

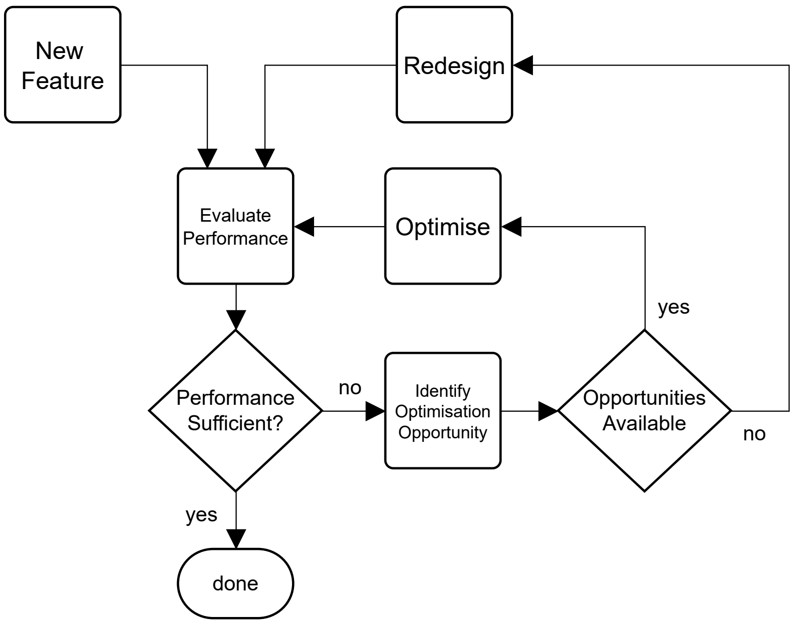
- Useful when a latency metric is being measured.
- Workload generator needs to fast enough to saturate system being benchmarked.
- Often more representative of a production environment (e.g an operating system receives a workload over time)

Hybrid **Definition 1.4.5**

A common setup combining batch and interactive workload strategies (e.g sample random queries from a predefined work set).

In order to get useful results from which

1.5 Optimisation Loop



Performance Parameters **Definition 1.5.1**

System and workload characteristics that affect performance.

System Parameters	Do not change as the system runs (instruction costs, caches)
Workload Parameters	Change as he system runs (available memory, users)
Numeric Parameters	Quantitative (e.g CPU frequency, available memory, number of user)
Nominal Parameters	Qualitative parameters (Runs on battery, has a GPU, runs in a VM)

The term *resource Parameters/Resources* refers to the parameters of the underlying platform (e.g CPU, memory).

Utilisation — **Definition 1.5.2**

The proportion of a resource used by a to perform a service by a system.

- A service has limited resources available (e.g CPU time, memory capacity, network bandwidth etc)
- Total available resources/resource budget available to a service is a parameter

Bottleneck — **Definition 1.5.3**

The resource with the highest utilisation.

- The limiting factor in performance of a system
- given some resource x is the bottleneck, the system is x-bound (e.g CPU-bound).
- Not always a resource, and performance may be bottlenecked by some other factor (e.g latency-bound $\rightarrow$ the system is dominated by waiting for some operation)

It is typically infeasible to identify all bottlenecks for an entire complex software system.
To limit optimisation complexity efforts should be restricted to optimising code paths that have particularly large effect on performance.

Critical Path — **Definition 1.5.4**

The sequence of activities which contribute the larges overall duration.

Hot Path — **Definition 1.5.5**

A code path where most of the execution time is spent (e.g very commonly executed subroutine)

In order to optimise we require:

- Ability to quickly compare alternative designs
- Ability to select a near optimal value for platform parameters

While workload parameters are not typically controllable at this stage, some system parameters are.

Parameter Tuning — **Definition 1.5.6**

Finding the vector within the parameter space that minimises resource usage, or maximises performance.

- Exploring the parameter space is expensive (even with non-linear optimisation)
- Analytical models can be used to accelerate search.
- Tuning needs to consider tradeoffs (e.g much of a cheap resource versus little of an expensive one)

Analytical Performance Model — **Definition 1.5.7**

A model describing the relationship between system parameters and performance metrics.

- Having an accurate analytical model for the system is analogous to *understanding* the performance of the system.
- Models can be stateless (e.g an equation) or stateful (e.g using markov chains).
- Need to model dynamic systems with (ideally small) static models.
- Very fast (faster than search parameter space)
- Allow for *what-if analysis* of system and workload parameters.

Simulation — **Definition 1.5.8**

A single observed run of a stateful model

- Can see all interactions within the system in perfect detail
- Extremely expensive to run (limiting number of simulations and the speed of the optimisation workflow)
- Much more rarely used than other techniques described

Chapter 2

Profiling

Event **Definition 2.0.1**

A change in the state of the system.

- Usually some granularity limit is used (e.g clock tick)
- Optionally has a payload (properties describing the event - e.g cache line evicted → the addresses & data evicted)
- Has an accuracy - degree to which the event represents reality (many events are numeric & come with measurement related error)

Simple/Atomic Event	Executed instruction, clock tick, function called
Complex Event	Cache line evicted, ROB flush due to misspeculation

Event sources have two components:

Generator	Observes changes to system state (online → part of the runtime system)
Consumer	Processes events and converts into meaningful insights (offline or online)

The overhead associated with collecting events can be very high.

Pertubation **Definition 2.0.2**

The effect of analysis on the performance of a system.

- If it is constant/deterministic we can subtract it from measurement to get an accurate result.
- Non-deterministic pertubation negatively affects accuracy as it cannot separate analysis overhead from measured performance

Stacking up! **Example Question 2.0.1**

Instrumentation is added to a program to inspect its stack at regular intervals, and record the stack trace. Assuming it is implemented to ensure the time spent traversing the trace is deterministic and can be removed from any results, why may this instrumentation still result in non-deterministic pertubation?

Data Cache Side Effects
The instrumentation may bring *deep* parts of the stack (*shallow* is *hot* and likely cached) into cache, evicting other lines. Hence the tracing *pollutes* the cache and results in more misses & hence more memory related stalls for the program.

We could also more weakly argue about instruction cache limitations, or potential for associativity conflicts occurring may have been avoided by design in the original program, but due to placement of instrumentation's static data & text have their positions moved.

Fidelity **Definition 2.0.3**

The level of granularity with which events are recorded.

- Perfect fidelity means every event is recorded.
- Lower fidelity generally means less overhead.

Trace **Definition 2.0.4**

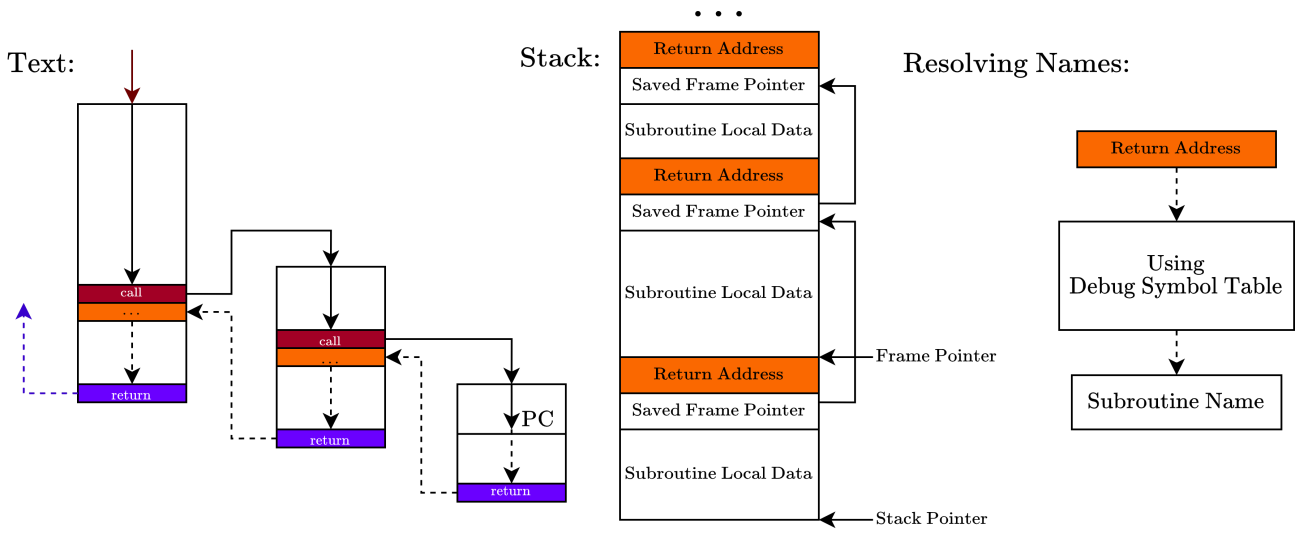
A complete log of the system's state (and hence changes in state/events) for some time period.

- Events usually totally ordered (exceptions being with parallelism - i.e multicore systems)
- Accuracy is inherited from events (e.g accuracy of clock, hardware counters etc)
- Analysing traces is time consuming (solution: Profiling)

Events may be aggregated to reduce the logging overhead.

2.0.1 Call Stack Tracing

A common trace is to capture and inspect the stack of a thread.



- Stack frame layout must conform to a convention that stores frame pointers and return addresses on the stack.
- Debug symbols included in the binary (e.g part of the ELF spec) can be used to convert return addresses scraped from the stack into symbols from the source code (function names).
- The stack may not easily represent the call structure of a program (inlining, tail call elimination, constant evaluation and propagation)

2.1 Sampling

Rather than recording all events, we can reduce overhead by sampling some events.

Performance	Fidelity traded against pertubation for reduced overhead in recording events.
Less Pertubation	Fewer interactions with program → less effect on performance → less pertubation.

Imperfect May skip sampling some events (e.g very small functions), but more expensive functions are more likely to be sampled, and we generally care about these more.

2.1.1 Time-Base Intervals

We can make use of hardware supported timer interrupts to set timers before using an interrupt to trap and allow the sampler to take control.

- Can use CPU *reference cycles* as a proxy metric to time
- Hardware clocks are often poorly defined & vary (i.e variable clock frequencies in modern CPUs, synchronisation of clocks between CPUs)
- Easily available & easy to interpret (ticks $\propto$ time)

2.1.2 Event-Base Intervals

A generalisation of time-based intervals (as a clock tick $\rightarrow$ time is an event)

- Can define in terms of occurrences of an event (e.g function call)
- Depending on design, can be accurate / low noise
- Can be tricky to interpret as a link to some measure proportional to time is needed (typically we care about execution time)

Quantisation Errors **Definition 2.1.1**

The resolution of an interval is limited (e.g by clock), but time is continuous. Mapping events to a discrete measure of time can introduce errors & bias (costs may be attributed to the wrong time & hence wrong state).

2.1.3 Indirect Tracing

Indirect Tracing **Definition 2.1.2**

We can sample from parts of a program, and infer traces from the structure of the program.

For example control flow instructions dominate non-control flow instructions, we could sample from control flow instructions, and then infer event between (non-control flow instructions) (called *basic block counting*), effectively indirectly tracing them.

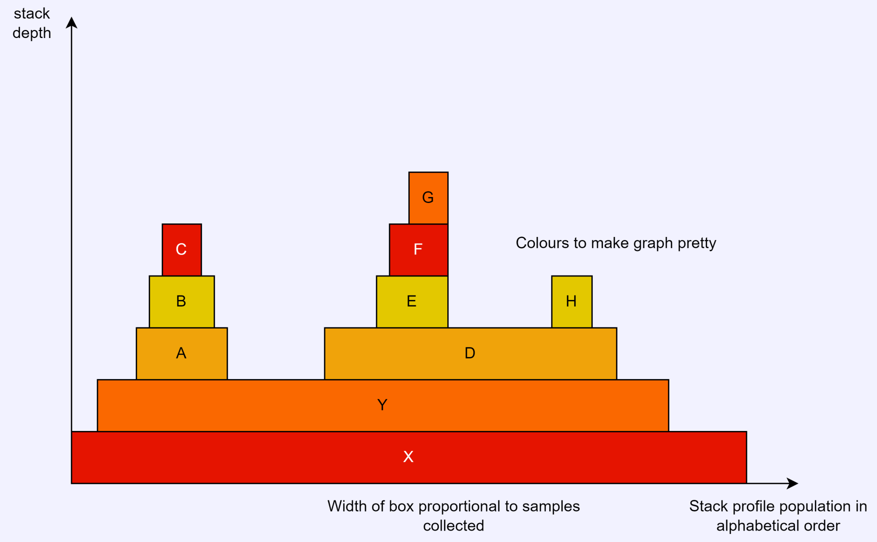
- Can be used to reduce overhead from tracing
- fidelity and accuracy good (depending on the events traced and the indirection used)

Profile **Definition 2.1.3**

A (usually graphical) representation of information relating to the characteristics of a system in terms of resources (quantified) used in certain states.

- An aggregate over a specific metric (e.g a global aggregate if total cache misses, or per event such as cycles per instruction, or cache misses per function)
- Information is lost in aggregation
- Aggregation has lower overhead than recording all events, so can reduce pertubation

Flame Graph **Definition 2.1.4**


stack depth
G
C
F
Colours to make graph pretty
B
E
H
A
D
Y
X
Width of box proportional to samples collected
Stack profile population in alphabetical order

2.2 Recording Events

Instrumentation **Definition 2.2.1**

Adding event logging code to a program.

Easily Applicable	No need to extra hardware or OS support.
Flexible	Can implement any kind of logging required.

High Overhead	
Perturbation	As part of the program can effect performance

2.2.1 Manual Instrumentation

Logging using a library (e.g `printf` logging)

Fine control	Programmer can easily specify exactly where to log what.
Supportless	No need for compiler or hardware support.

High Overhead	
Disable	Need to disable for release builds (recompile without any logging)

2.2.2 Automatic Instrumentation

Compiler supported injection of event recording code into a program.

- Can be done at source level, or within some intermediate form (e.g injecting into some bytecode)
- Can potentially reduce overhead compared with manual instrumentation (compiler instruments at a lower level representation)

2.2.3 Binary Instrumentation

Instrumenting an already compiled binary.

Static	Adding instrumentation directly, overhead can be assessed from the binary.
Dynamic	Adding instrumentation at runtime (works well with JiT)

2.2.4 Kernel Counters/ Software Performance Counters

The kernel already has the tools required to collect many kinds of events. These tend to be higher-level OS interactions, rather than microarchitectural events. For example:

- Network packets sent
- Virtual memory events (e.g page faults)
- Context Switches
- Threads spawned

2.2.5 Emulation

2.2.6 Hardware Counters

Special registers configured to count low-level events as well as intervals (e.g cycles)

- Fixed number can be active at runtime
- Often buggy / unmaintained / inaccurate (and poorly documented) → only the most popular counters are trustworthy

2.3 Perf

UNFINISHED!!!

RTFM ***Extra Fun!* 2.3.1**

Intel 64 and IA-32 Architectures Optimization Reference Manual.

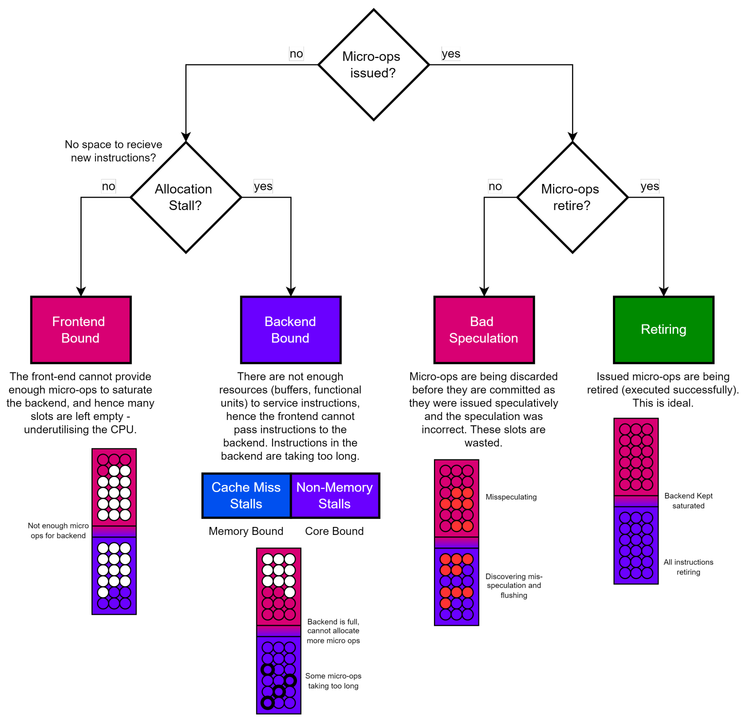
- Microarchitectural features documented.
- Written as an optimisation guide.
- Code example can be found in its github repo

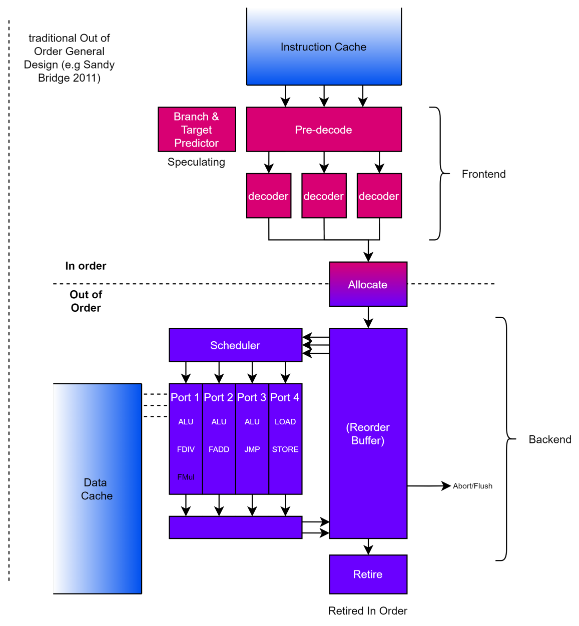
2.4 Microarchitectural bottleneck analysis

Advanced Computer Architecture ***Extra Fun!* 2.4.1**

A basic understanding of computer architecture is reuired (pipelining, in order, caches, out of order, speculation).

The 60001 - Advanced Computer Architecture module by Prof Paul Kelly covers this in great depth.


Micro-ops issued?
no
yes
No space to recieve new instructions?
Allocation Stall?
no
yes
Micro-ops retire?
no
yes
Frontend Bound
Backend Bound
Bad Speculation
Retiring
The front-end cannot provide enough micro-ops to saturate the backend, and hence many slots are left empty - underutilising the CPU.
There are not enough resources (buffers, functional units) to service instructions, hence the frontend cannot pass instructions to the backend. Instructions in the backend are taking too long.
Micro-ops are being discarded before they are committed as they were issued speculatively and the speculation was incorrect. These slots are wasted.
Issued micro-ops are being retired (executed successfully). This is ideal.
Not enough micro ops for backend
Cache Miss Stalls
Non-Memory Stalls
Memory Bound
Core Bound
Backend is full, cannot allocate more micro ops
Some micro-ops taking too long
Misspeculating
Discovering mis-speculation and flushing
Backend Kept saturated
All instructions retiring



See Intel's V-Tune performance metrics definitions for more

2.5 Vtune

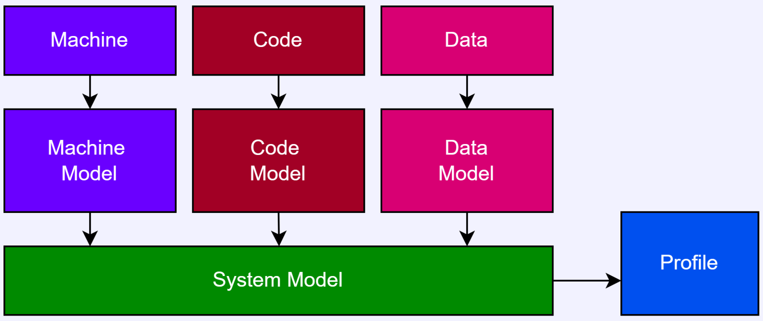
UNFINISHED!!!

Chapter 3

Modelling

3.1 Motivation

System Model **Definition 3.1.1**



A model used to characterise the performance of a system. Used to estimate performance without the overhead of running the system.

During this course we make several simplifying assumptions:

- **Input Data from a Known Distribution**
  Typically uniform. We avoid complications form correlated inputs.
- **No System Noise** Noise caused by the operating system (scheduling, other processes actions) and other factors.
- **Single Threaded & Deterministic Code** Modelling parallel systems requires considering contention and is an open area of research.

3.2 Numerical Models

Numerical Model **Definition 3.2.1**

Empirical measurement of the observed behaviour of the system.

- Describes actual system behaviour (is a measurement)
- Prediction limited → depends on the human interpretation of the data.

Easy To Create	As long as the system is available to run benchmarks on.
True	It is an empirical measurement of the actual system (or some component of it).
Easy to Interpret	We can easily see how performance metrics vary with a given parameter.

Poor Generalisation	We cannot easily apply measurements to new values for parameters (mainly descriptive model / poor prediction)
Costly	For a large number of parameters, high fidelity a large amount of data is required, and hence many runs of benchmarks.
Limited Prediction	We can infer predictions from measurement, but it is difficult to extrapolate with confidence.

Microbenchmark **Definition 3.2.2**

Small programs designed to test a specific portion of a system.

Numerical Models are constructed by:

1. Data (performance metrics) gathered through *microbenchmarks* on a range of parameter values.
2. Data is analysed/interpreted.

3.3 Analytical Models

Analytical Model **Definition 3.3.1**

A formalised relationship between system parameters and performance metrics.

- Hard to interpret (limited descriptive use)
- Evaluating the model (given parameters) predicts performance metrics
- A detailed understanding of the system is required
- Model must be validated using experimental data

Models can even be used inside the system itself. For example using an analytical model to determine which optimiser to use for a query plan

Example Models ***Extra Fun!* 3.3.1**

An example of a cost model for an R-Tree can be found in Cost models for join queries in spatial databases.

The Picasso Database Query Optimizer Visualizer another example of analytical models being used in the context of databases.

3.3.1 Empirical to Analytical

We fit an analytical model via regression on data from a numerical model.

Benchmark

For example we could benchmark the memory system of a machine:

```c
#include <stdint.h>
#include <stddef.h>

extern int32_t* input_data; // An array of data
extern size_t N;            // A large constant
extern size_t size;         // Parameter: size of the region accessed

void benchmark() {
    volatile int32_t sum = 0;
    for (size_t i = 0; i < N; i++)
        // mod is used (expensive), to reduce use size power of 2 and a bitmask
        sum += input_data[i % size];
}
```

System Parameters

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

Model

Given s is the stride parameter in the characteristic equations:

$$T_{Mem} = \sum_{i=0}^{3} l_i \times \min\left(1, \frac{s}{B_i}\right) \quad \text{or we could model as} \quad T_{Mem} = \begin{cases} l_0 & s < C_1 \\ l_0 + l_1 & s < C_2 \\ l_0 + l_1 + l_2 & s < C_3 \\ l_0 + l_1 + l_2 + l_3 & otherwise \end{cases}$$

We can compare the effects of altering system parameters on both models.

Fitting

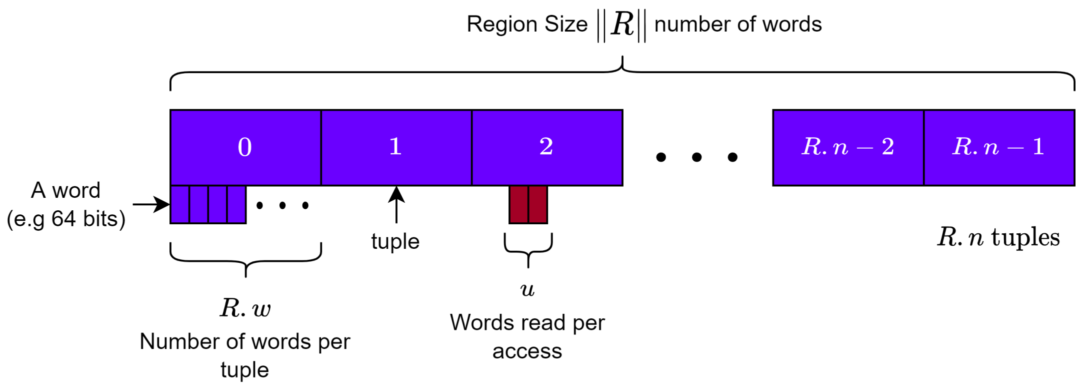
Some parameters can be taken from documentation, others may be derived from experiment.

Ideally the model should automatically self-tune to set parameters. Automating the tuning process has several advantages:

- Less work required by humans.
- Can adapt to changing conditions (e.g if cache size is artificially reduced by contention for shared cache my many threads)
- Can scale forward (if CPU is updated to newer generation, model can be re-tuned to fit changed system parameters)

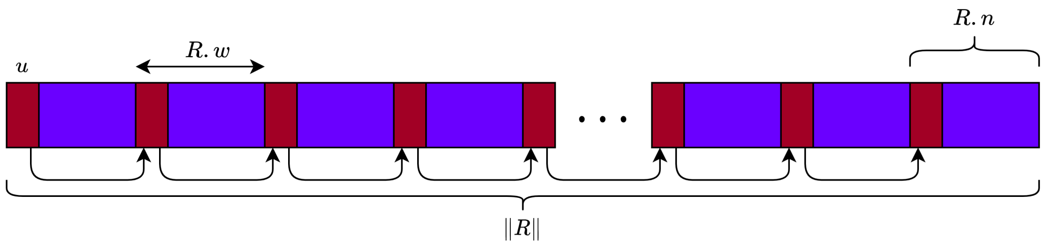
3.4 Memory Access Patterns

3.4.1 Memory Region



The number of words skipped over between each access is $R.w - u$

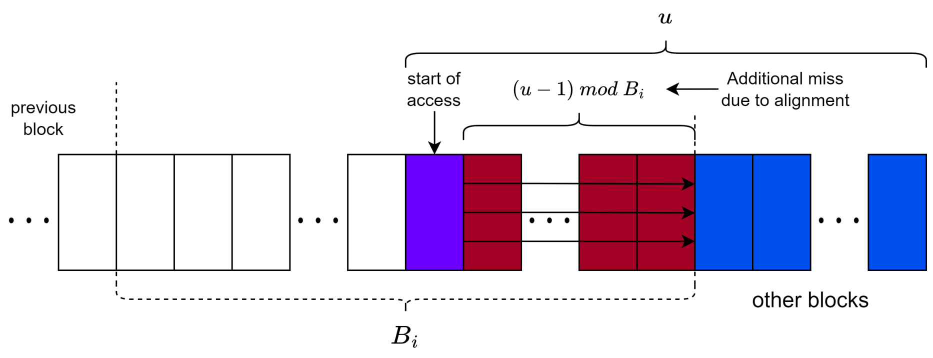
3.4.2 Sequential



Given some block size we can estimate the number of misses with a simple model.

- Assume a cold start (no part of the region in cache already)
- Assume accesses are within blocks, not over boundaries (e.g if $u > 1$ and went over the edge of a cache line)
- *s_trav* means *sequential traversal*

$$M_i^s(s\_trav) = \begin{cases} \frac{\|R\|}{B_i} & R.w - u < B_i \\ R.n \times \left\lceil \frac{u}{B_i} \right\rceil & R.w - u \geq B_i \end{cases}$$

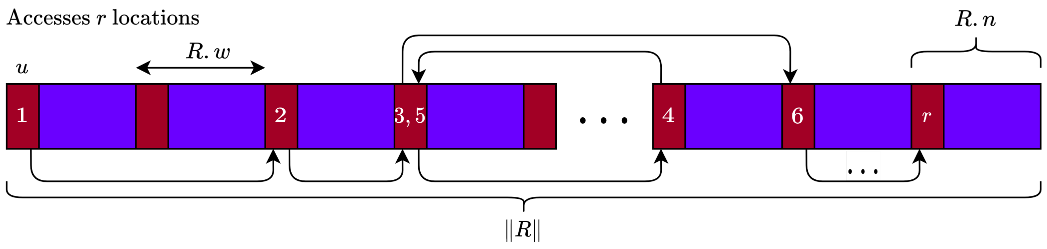


We may want to consider additional misses due to misalignment (u stretching over two Bs), in which case we much change the $R.w - u \geq B_i$ case to:

$$M_i^s(s\_trav) = R.n \times \left(\left\lceil \frac{u}{B_i} \right\rceil + \frac{(u-1) \mod B_i}{B_i} \right)$$

3.4.3 Repetitive Random Access

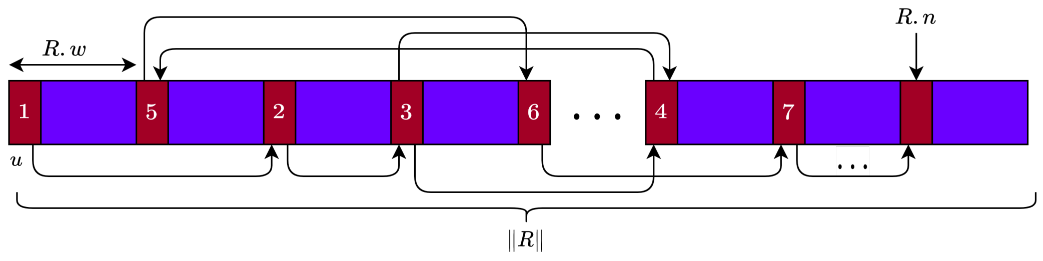
$$M_i^s(rr\_acc) = \text{undefined in lecture}$$



Randomly access r locations (can repeat accesses)

3.4.4 Random Traversal

$$M_i^s(r\_trav) = \text{undefined in lecture}$$



Access all $R.n$ locations in some random order.

3.4.5 Modelling Complex Patterns

$\mathcal{P}_1 \oplus \mathcal{P}_2$	Sequential execution of patterns $\mathcal{P}_1$ then $\mathcal{P}_2$.
$\mathcal{P}_1 \odot \mathcal{P}_2$	Concurrent execution (access patterns interleaved).

We can hence combine access patterns.

Random Arrays **Example Question 3.4.1**

Create an access pattern description for the following program. List any other assumptions.

```c
#include <stdint.h>
#include <stddef.h>

extern int32_t input_data_1[i]; // uniform random data (used for index)
extern int32_t input_data_2[j]; // random data
extern size_t N;               // A large constant

void benchmark() {
    int32_t sum = 0;
    for (size_t i = 0; i < N; i++)
        sum += input_data_2[input_data_1[i]];
}
```

We assume that $N = i$ and all entries of `input_data_1` x are such that $0 \leq x \leq i$ (otherwise array accesses can be out of bounds)

$$s\_trav(R.w = 1, u = 1, R.n = i) \odot rr\_acc(R.w = 1, u = 1, R.n = j, r = i)$$

More complex access patterns can be modelled.

$$\underbrace{s\_trav(\dots) \odot r\_trav(\dots)}_{\text{Hash Join Build}} \oplus \left(\underbrace{\underset{\text{Read Input}}{s\_trav(\dots)} \odot \underset{\text{Access Hashtable}}{rr\_acc(\dots)} \odot \underset{\text{Write Output}}{s\_trav(\dots)}}_{\text{Hash Join Build}} \right)$$

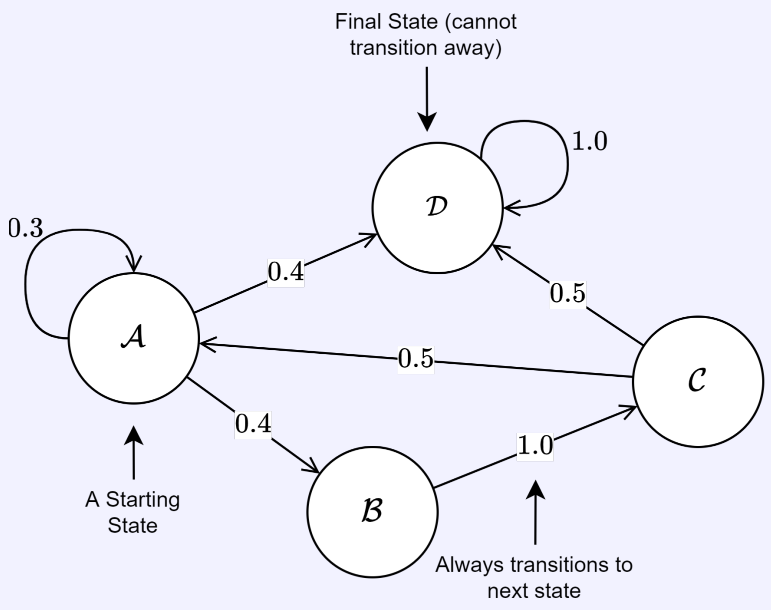
3.5 Modelling State

Analytical models are stateless, but most systems have some kind of dynamic state that can influence behaviour and performance.

Stochastic models can be used to encode state (combine analytical models and dynamic state).

Discrete Time Markov Chains (DTMC) **Definition 3.5.1**

A stochastic model describing a sequence (discrete time steps) of possible states.



- Much like a probabilistic finite state machine → the next state is only dependent on the previous (and random variables for transition) (called the *markov property*).
- Transition probabilities out of a state must sum to 1
- Can be represented as a matrix (directed graph represented as an adjacency matrix)

3.5.1 Simple Branch Direction Prediction

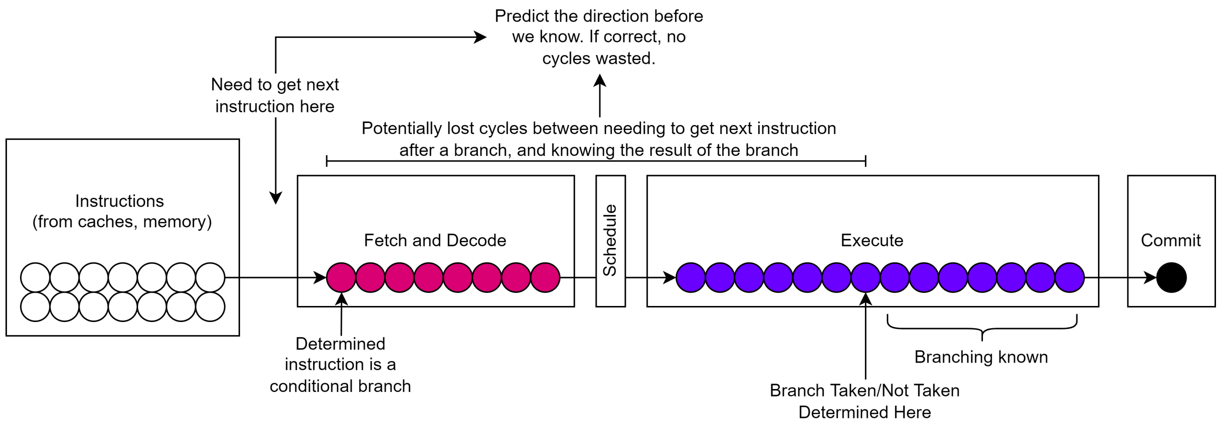
Consider a simple conditional branch, to a known target (instruction). For example a basic loop:

```
int N = 42;

void a() {
    int i = 0;
    while (i < N) {
        i++;
    }
}
```

Compiled for x86 with `-O1` (godbolt) we get the following:

```
a:
    ; Load N to edx and determine if any loop iteration should run (N may be 0)
    mov     edx, DWORD PTR N[rip]
    test    edx, edx

    ; Start looping
    jle     .L1         ; [Conditional branch, known target]
    mov     eax, 0      ; int i = 0
.L3:
    add     eax, 1      ; i++
    cmp     eax, edx    ; Check that i < N by checking if i == N yet
    jne     .L3         ; [Conditional branch, known target]
.L1:
    ret                 ; [Conditional branch, unknown target -> target prediction with RAS]
N:
    .long   42
```

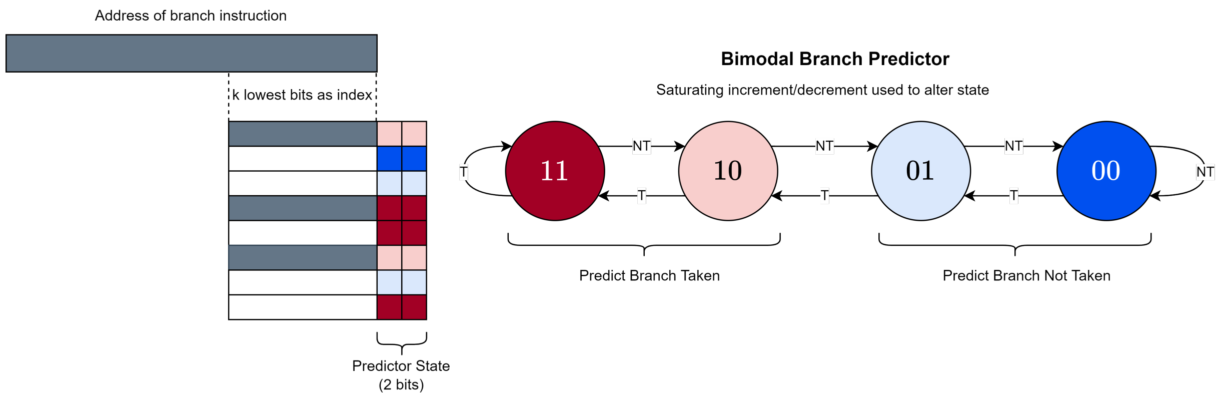
We will focus on the conditional branches with a known target (target prediction is more complex).


Predict the direction before we know. If correct, no cycles wasted.
Need to get next instruction here
Potentially lost cycles between needing to get next instruction after a branch, and knowing the result of the branch
Instructions (from caches, memory)
Fetch and Decode
Schedule
Execute
Commit
Determined instruction is a conditional branch
Branching known
Branch Taken/Not Taken Determined Here

There are several ways to reduce potential penalty from branch instructions:

- Smaller frontend (reduce cycles between fetch and branch determination).
- Early branch resolution.
- Branch delay slot (branch does not take affect until n instructions after) (used in MIPS).
- Branch prediction.

A simple branch prediction scheme is to use a 2-bit branch history table, the result of branch resolution for a given branch instruction changes the state in the table, this state is used to predict other execution of branches at that instruction. We can represent the branch predictor state for a given instruction by a finite state machine.



Hence we can also model the branch predictor state as a *markov chain*.

Stationary Distribution **Definition 3.5.2**

A probability distribution for a specific *markov chain* that remains the same (stationary) as time progresses.

Given the transition matrix $\mathbf{P}$ of a chain, the stationary distribution can be represented by a vector of probabilities for being in any given state π_∞ such that:

$$\mathbf{P}\pi_\infty = \pi_\infty$$

Hence π_∞ is an eigenvector of $\mathbf{P}$ with associated eigenvalue 1.

- It describes the probability of being in each state after infinite steps.

		Actual Takeness	
		Taken	**Not Taken**
Prediction	**Taken**	Correct	Mispredict
	Not Taken	Mispredict	Correct

$$P(\text{Mispredict}) = P(\text{Predict Taken}) \times P(\text{Actually Not Taken}) + P(\text{Predict Not Taken}) \times P(\text{Actually Taken})$$

3.6 Modelling an Entire System

Modelling an entire system is typically infeasible (scale and noise). Instead specific components and paths can be modelled.

1. Identify code that matters for performance using a profiler (Hot code sections, called bottlenecks in vtune)
2. Re-create behaviour in a controlled environment (*microbenchmarking*)
3. Create an analytical model of the code path/component.
4. Validate the model with experimental results.

Chapter 4

Efficient Code

4.1 Motivation

4.2 CPU Efficiency

CPU Bound **Definition 4.2.1**

UNFINISHED!!!

Control Hazard **Definition 4.2.2**

UNFINISHED!!!

Structural Hazard **Definition 4.2.3**

UNFINISHED!!!

Data Hazard **Definition 4.2.4**

UNFINISHED!!!

Chapter 5

Parallelism

5.1 Motivation

Denbard/MOSFET Scaling **Definition 5.1.1**

A scaling/power law stating that as transistors get smaller:

- Power density is constant, hence power $\propto$ area.
- Voltage and current decrease with transistor length.

Hence as transistor size decreases they become faster, more energy efficient and cheaper.

- Part of a paper (1974) on MOSFETs (metal-oxide-semiconductor field-effect transistors) co-authored by Robert Dennard
- Scaling becomes limited by transistor leakage, which grows as a proportion of power consumed by the transistor as scale is decreased.
- Leakage converts to heat which must be dissipated to prevent damage to the chip. This presents a limiting factor on scaling.

Slow end of *Dennard scaling* limit single threaded performance improvements $\Rightarrow$ Increase parallelism to attain higher performance.

Ambdahl's Law **Definition 5.1.2**

Where S is speedup, and each p_i, s_i is the speedup for a proportion of the program.

$$S = \left(\frac{p_1}{s_1} + \frac{p_2}{s_2} + \cdots + \frac{p_n}{s_n}\right)^{-1}$$

We can simplify this for the basic case of proportion p of a program perfectly parallelised over n threads:

$$S = \frac{1}{(1-p) + \frac{p}{n}}$$

5.1.1 Types of Parallelism

Data-Level Parallelism **Definition 5.1.3**

Increasing throughput by operating on multiple elements of data in parallel.

Typically this in the form of **S**ingle-**I**nstruction **M**ultiple-**D**ata (**SIMD**) instruction-set extension (instructions that operate a single simple instruction on several (typically adjacent) data elements.

Some such extensions include:

MMX	An early **SIMD** instruction set extension for IA32 developed by Intel for the Pentium P5 microarchitecture (1997), only supports integer arithmetic.
SSE	(**S**treaming **S**IMD **E**xtensions) developed by Intel that support floating point arithmetic.
AVX	(**A**dvanced **V**ector **E**xtensions) developed by Intel and AMD for x86-64 (Sandy bridge & AMD Bulldozer IN 2011).

Another form is **S**ingle-**I**nstruction **M**ultiple-**T**hreads (**SIMT**) where each thread operates on some data, and many threads execute the same instructions in lockstep. This is the programming model used by most GPUs (e.g see Nvidia's CUDA).

Instruction Level Parallelism **Definition 5.1.4**

Systems where multiple instructions can be executed in the same step (e.g instructions per cycle > 1). Many techniques are used to achieve this:

Pipelining	Allows higher throughput by splitting an instruction pipeline into stages that can be executed in parallel.
VLIW & **EPIC**	(**V**ery-**L**ong **I**nstruction **W**ord & **E**xplicitly **P**arallel **I**nstruction **C**omputing) architectures where instructions contain bundles of operations and explicitly determine which operations can be run in parallel (e.g Intel's Itanium/IA64). It is a way for statically scheduled processors to take advantage of ILP, but is not common.
Superscalar	Dynamically scheduled processors that can inspect instruction dependencies at runtime in order to dispatch fetched instructions for execution out-of-order and in parallel. Many other techniques can be used here such as speculative execution and register renaming.

Nearly all modern processors are pipelined & superscalar.

Flynn's Taxonomy ***Extra Fun!*** **5.1.1**

A classification scheme for computer architectures that neatly describes the type of parallelism they use (see wikipedia here).

Task-Level Parallelism **Definition 5.1.5**

Splitting an algorithm into sections that can be run in parallel.

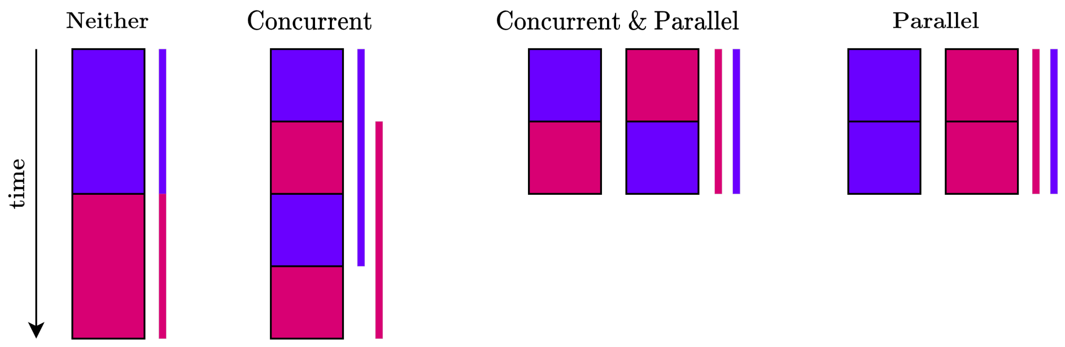
- Parallelism must be extracted by the programmer.
- Tasks run in separate processes or threads, potentially on different cores, or processors.
- Each task being run in parallel can be different (different instructions and either different data or shared data accessed)

Crazy Fast IPC ***Extra Fun!*** **5.1.2**

The L4 microkernel implements *short IPC* where a sending process places the message in registers, and a fast context switch to the receiving thread without clearing registers (explained here).

The L4ka project at Karlsruher University's Operating Systems group has an Itanium implementation that takes advantage of this (and Itanium specifics $\rightarrow$ lots of registers) to complete short IPC in as few as 36 cycles.

5.1.2 Concurrency



Parallel **Definition 5.1.6**

"Decomposing a problem into smaller tasks executed at the same time on different compute resources"

Concurrent **Definition 5.1.7**

"Decomposing a problem into smaller tasks executed during overlapping time periods"

$\mathcal{C}$	$\neg\mathcal{P}$	An OS scheduler interleaving the execution of two threads on a core.
$\mathcal{C}$	$\mathcal{P}$	Threads switching cores (OS scheduler typically tries to avoid this).
$\neg\mathcal{C}$	$\mathcal{P}$	Two threads executed entirely on separate cores.

5.1.3 Cost Efficiency

What is it good for! **Example Question 5.1.1**

What are the main advantages of parallelism?

- Better performance through parallelism
- Better cost-efficiency (can share RAW, disk, network cards between several cores, rather than needing separate machines)

Much of this is workload and algorithm specific $\rightarrow$ a user needs to analyze their specific use case to determine the appropriate hardware.

Datacenters optimise for *total cost of ownership*:

- Entire hardware lifecycle (purchase, expected maintenance, mean time-to-failure).
- Energy consumption & cooling requirements.
- land, building & even security costs.
- Existing hardware compatibility.

CPU provisioning **Example Question 5.1.2**

Given the following specifications:

- 1 Gbps network
- Requests are 1KB
- Each requet takes 50μs to process

$$\underbrace{2^{30}}_{\text{1 Gbps Network}} \div \underbrace{2^{3}}_{8\ bits=1\ byte} \div \underbrace{2^{1}0}_{\text{1 KB request}} = 2^{17} \text{ requests per second}$$

Each core can process a request in 50μs so:

$$(50 \times 10^{-6})^{-1} = (5 \times 10^{-5})^{-1} = 2 \times 10^{4} \text{ requests per second}$$

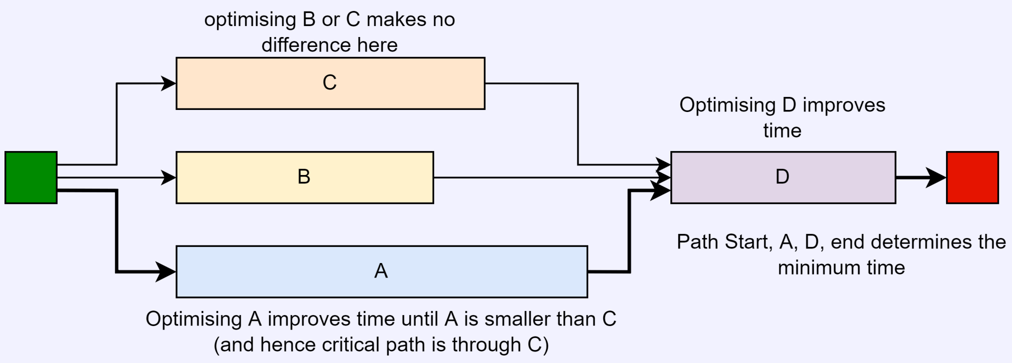
Hence to saturate the flow of incoming requests:

$$cores = \left\lceil \frac{requests/sec}{requests/sec/core} \right\rceil = \left\lceil \frac{2^{17}}{2 \times 10^4} \right\rceil \approx \lceil 6.5 \rceil = 7$$

5.1.4 Critical Path Analysis

Critical Path **Definition 5.1.8**

A sequence of tasks determining the minimum time for an operation.


optimising B or C makes no difference here
C
Optimising D improves time
B
D
Path Start, A, D, end determines the minimum time
A
Optimising A improves time until A is smaller than C (and hence critical path is through C)

- The profiler is unaware of the critical path across threads (open area of research)
- Optimising outside of the critical path will often not improve time.

5.2 Cache Coherency

CPUs contain multiple levels of data caches (L1, L2, L3/LLC) which need to be coherent (same cached location $\Rightarrow$ same value).

A cache coherency protocol determines how a CPU achieves this. Typically each cache line has some state, with the protocol determining how and when cache lines transition state depending on issues reads and writes.

MSI is a simple cache coherency algorithm.

UNFINISHED!!!

In order to add support for atomics and additional **Locked** state is included

5.3 Atomics

Atomic read, modify and write operations require hardware support and are exposed to programmers through compiler supported libraries (e.g stdatomic for C11, C++ atomic and Rust `std::sync::atomic`).

Lock Prefix ***Extra Fun!* 5.3.1**

The IA32 & x86-64 support a `lock` prefix for some instructions to specify they should be run atomically.

```c
#include <stdatomic.h>

void example() {
    int a = 1;
    int b = 4;
    int temp = 3;
    atomic_compare_exchange_strong(&a, &b, temp);
    return;
}
```

```
example:
    mov DWORD PTR [rsp-4], 1
    mov eax, 4
    mov edx, 3
    lock cmpxchg DWORD PTR [rsp-4], edx
    ret
```

Atomic methods often allow the programmer to specify the memory order (order of memory accesses around the atomic access).

```
enum class memory_order : /* unspecified */ {
    relaxed, // relaxed ordering - no guarantee on ordering
    consume, // release-consume ordering
    acquire, // release-acquire ordering
    release, // release-acquire and release-consume ordering
    acq_rel, // both release and acquire
    seq_cst  // sequentially consistent ordering
};
```

These orders are well-documented here on cppreference.

5.4 Synchronisation

5.4.1 Synchronisation Primitives

Mutex/Lock **Definition 5.4.1**

Can only be held by one thread, blocks otherwise. Single thread in critical region.

`std::mutex`

Shared Mutex/RW Lock **Definition 5.4.2**

A mutex with a shared and exclusive lock. Allows any number of *readers* or single *writer* into a critical region.

`std::shared_mutex`

Semaphore **Definition 5.4.3**

Can be incremented/decremented, waits until value is > 0. Allows n threads into a critical region.

`std::counting_semaphore`
`std::binary_semaphore`

Condition Variable **Definition 5.4.4**

Set threads to wait until a condition is signalled as true (can use predicates, or have threads signal to wake up n waiting threads).

`std::condition_variable`

Barrier **Definition 5.4.5**

Forces n threads reaching the barrier to wait, until n have arrived, at which point all are unblocked.

`std::barrier`

5.4.2 Lock Implementation

User-Space

Must make use of atomics to create spinlocks.

- Basic locks just use CAS to wait on a flag.
- Intrinstics such as `_mm_pause()` can be used to wait efficiently (produces no-ops)
- Backoff (fixed & exponential) can be used to decrease contention in access to the lock's flag.
- Other spinlock types such as ticket locks can add guarantees in order of acquisition by threads.

No Kernel No kernel involvement means potentially less overhead when contention is low.

Spinning	Waiting threads consume valuable CPU resources while spinning. This can become a severe issue if the spinning thread has higher priority than the thread holding the lock.
Starvation	For a basic thread implementation there is no ordering on acquisition, so a thread may be constantly skipped/other threads that more recently attempted acquire the lock. (Note: ticket locks guarantee this ordering)

Kernel Level Lock

Lock and unlock via a syscall, with logic implemented in kernel space. If a thread cannot acquire the lock, thens schedule another thread.

- Can implement same CAS based acquisition attempt within the kernel (multiple threads on multiple cores may be servicing a syscall simultaneously)
- In systems with a single hardware thread, mutual exclusion within the kernel can be achieved by disabling interrupts (e.g as in Pintos, not possible on modern systems)

Fairness	Can keep a queue of blocked/waiting threads and wake them in order.

Expensive	If the lock is released/available, then acquisition has the additional overhead of a syscall when compared to a user level implementation.

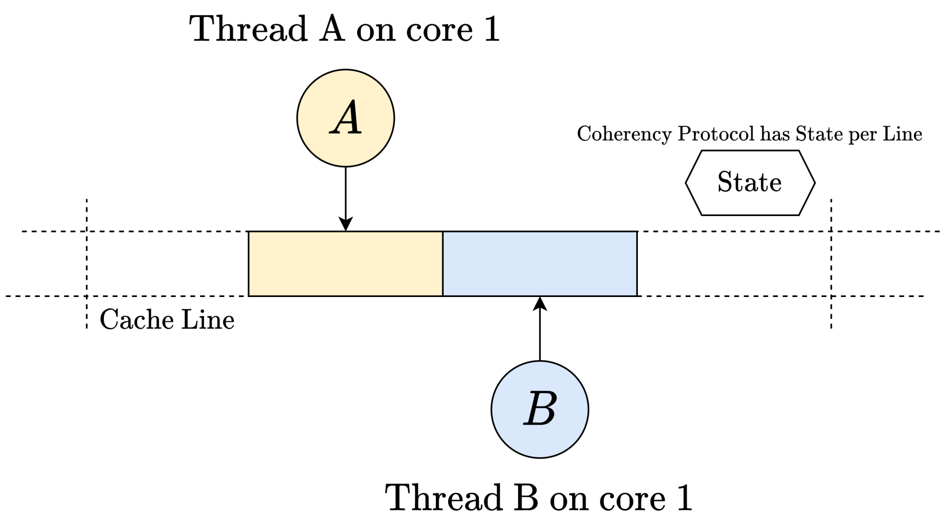
Hybrid Level Lock

A user level lock that bails out to a kernel level list of blocked waiters when contention is high.

- Most popular implementation is the linux *futex* (*fast userspace mutex*).
- `pthread_mutex_lock` uses a futex internally

Adaptable	Can dynamically adapt to spin in userspace, or block from kernel based on contention at runtime.

5.5 False Sharing



Cache coherency protocols store state per line. Multiple threads may not access shared data, but may access locations in that map to the same cache lines and if any threads write the line may be continually invalidated.

- We can use hardware counters (e.g profiling with perf, using the HITM (hit modified) counter on x86-64 CPUs) to find false sharing occuring.

5.6 Distributing Work

We can implement some basic work distribution schemes using a Task.

```
struct Task {
  virtual void run() = 0;
};

// We can make use of std::reference_wrapper to pass references to a polymorphic task type
using TaskRef = std::reference_wrapper<Task>;
using TaskQueue = std::vector<TaskRef>;
```

- Virtual method allows for polymorphism in tasks
- Task can store state, and return values.

We will make use of some example tasks:

```
struct PrintTask : Task {
  int id;

  PrintTask(int id) : id(id) {
    std::cout << "Created PrintTask: "
              << id
              << std::endl;
  }

  void run() override {
    std::cout << "PrintTask: "
              << id
              << std::endl;
  }
};
```

```
struct WaitTask : Task {
  int id;
  std::chrono::milliseconds wait_ms;

  WaitTask(int id, int waitfor)
    : id(id), wait_ms(waitfor) {
    std::cout << "Created WaitTask: "
              << id
              << std::endl;
  }

  void run() override {
    std::cout << "started WaitTask: "
              << id
              << std::endl;
    std::this_thread::sleep_for(wait_ms);
    std::cout << "Ended WaitTask: "
              << id
              << std::endl;
  }
};
```

Runnable through a function for each work distribution:

```
int main() {
  /* Can allocate tasks on stack, in a vector, etc */
  PrintTask a(0); PrintTask b(1); PrintTask c(2); PrintTask d(3);

  /* Create a task Queue to pass to method */
  TaskQueue tasks{a, b, c, d};

  auto start = std::chrono::steady_clock::now();

  /* Run tasks (comment out all but one method) */
  process_sequential   (tasks);
  process_on_demand    (tasks);
  process_fork_and_join(tasks);
  process_dispatch     (tasks);
  process_jobsteal     (tasks);

  auto end = std::chrono::steady_clock::now();

  /* Display time */
  std::cout << "Time: " << (end - start).count() << std::endl;
}
```

5.6.1 Sequential

```cpp
void process_sequential(TaskQueue &tasks) {
  for (auto &task : tasks) {
    task.get().run();
  }
}
```

Simple	No reasoning about concurrency required, easy to implement & understand performance.
Low Overhead	No overhead for using synchronisation primitives, as required by concurrent schemes.

No Parallelism

5.6.2 On Demand

```cpp
// Invariant: tasks' lifetime > threads (threads reference tasks)
void process_on_demand(TaskQueue &tasks) {
  std::vector<std::thread> threads;
  for (auto &task : tasks) {
    std::thread([&]() { task.get().run(); }).detach();
  }
}
```

Simple	Each thread only accesses task once, all synchronisation after is part of the task.

Performance	Spawning threads is expensive, and when there is a large number of threads (e.g when os threads > hardware threads) scheduling and contention on locks protecting shared data accessed by tasks reduce performance.

5.6.3 Fork & Join

For each batch of tasks, spawn new threads.

```cpp
void process_fork_and_join(TaskQueue &tasks) {
  std::vector<std::thread> threads;
  for (auto &task : tasks) {
    threads.emplace_back(std::thread([&]() { task.get().run(); }));
  }
  for (auto &thread : threads) {
    thread.join();
  }
}
```

OpenMP ***Extra Fun!* 5.6.1**

OpenMP is a set of standards for adding high-level parallelism directives into multiple languages (Fortran, C and C++). OpenMP references

```cpp
#pragma omp for
for ( /* ... */ ) {
    /* block executed in parallel  with other iterations */
}
```

5.6.4 Work Dispatching

Each worker has its own queue, tasks are dispatched to each queue.

- Queue is accessed by a main thread and a single worker
- Fixed number of workers

```
struct WorkerQueue {
  WorkerQueue() : finish_(false), worker_([&]() {
    while (!finish_.load()) {
      auto task = get_task();
      if (task.has_value()) task.value().get().run();
    }
  }) {}

  void add_task(TaskRef task) {
    const std::lock_guard<std::mutex> lock(tasks_mutex_);
    tasks_.push_front(task);
  }

  std::optional<TaskRef> get_task() {
    const std::lock_guard<std::mutex> lock(tasks_mutex_);
    if (!tasks_.empty()) {
      auto task = tasks_.back();
      tasks_.pop_back();
      return task;
    } else { return {}; }
  }

  size_t size() const {
    const std::lock_guard<std::mutex> lock(tasks_mutex_);
    return tasks_.size();
  }

  void finish() {
    finish_.store(true);
    worker_.join();
  }

private:
  mutable std::mutex tasks_mutex_;
  std::deque<TaskRef> tasks_;
  std::atomic<bool> finish_;
  std::thread worker_;
};
```

We can then create a pool of `WorkerQueues` and allocate tasks in a round-robin scheme.

```
template <size_t THREADS> struct WorkerPool {
  WorkerPool() : task_number_(0) {}

  void push_task(TaskRef task) {
    size_t index = task_number_.fetch_add(1) % THREADS;
    queues_[index].add_task(task);
  }

  void finish() {
    for (auto &q : queues_)
      q.finish();
  }

private:
  std::array<WorkerQueue, THREADS> queues_;
  std::atomic<size_t> task_number_;
};

void process_dispatch(TaskQueue &tasks) {
  WorkerPool<5> workerPool;
  for (auto t : tasks) {
    workerPool.push_task(t);
  }
  workerPool.finish();
}
```

Improve these examples! ***Extra Fun!*** **5.6.2**

These examples have threads spin until a job is found, to improve efficiency we could suspend threads while no tasks are present.

- Could use a semaphore (`#include <semaphore>` from C++20 (only available in newer versions of GCC)).
- Need to ensure that suspended threads are still ended when `WorkerQueue::finish` is called.
- Could use a condition variable to implement producer-consumer pattern with finish.

Contention Each thread has its own queue, and hence can only contend for access with the thread-/threads adding to the queue, not other workers.

Consumer Balancing Some threads may finish jobs quickly and idle, while others are still completing work, but the tasks may still be allocated to busy threads.

5.6.5 Work Stealing

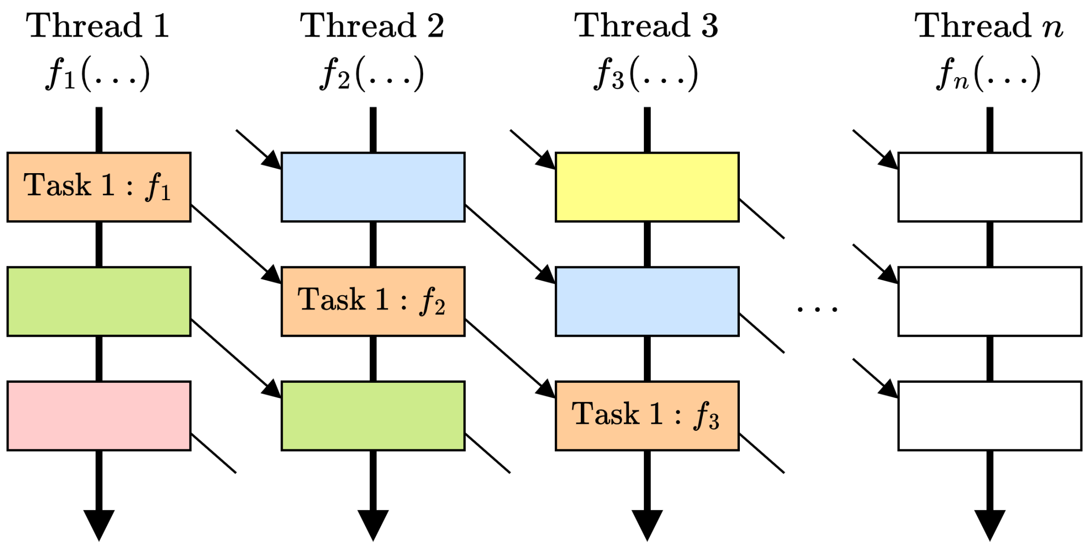
Many threads contend to take tasks from a queue.

```
template <size_t THREADS> struct TaskPool {
  TaskPool() : finish_(false) {
    for (auto &t : threads_)
      t = std::thread([&]() {
        while (!finish_.load()) {
          tasks_mutex_.lock();
          if (!tasks_.empty()) {
            TaskRef task = tasks_.back();
            tasks_.pop_back();
            tasks_mutex_.unlock();
            task.get().run();
          } else {
            tasks_mutex_.unlock();
          }
        }
      });
  }

  void submit_task(TaskRef task) {
    const std::lock_guard<std::mutex> lock(tasks_mutex_);
    tasks_.push_front(task);
  }

  size_t size() const {
    const std::lock_guard<std::mutex> lock(tasks_mutex_);
    return tasks_.size();
  }

  void finish() {
    finish_.store(true);
    for (auto &t : threads_)
      t.join();
  }

private:
  std::mutex tasks_mutex_;
  std::deque<TaskRef> tasks_;
  std::array<std::thread, THREADS> threads_;
  std::atomic<bool> finish_;
};

void process_jobsteal(TaskQueue &tasks) {
  TaskPool<5> taskpool;
  for (auto t : tasks) {
    taskpool.submit_task(t);
  }
  taskpool.finish();
}
```

Consumer Balancing Tasks are *stolen* by idle threads, busy threads are not overloaded with tasks.

Contention All threads waiting for new jobs contend for access to the same queue.

5.6.6 Streaming



Rather than executing tasks in parallel, execute each stage of a task in a pipeline.

- Data needs to be communicated between threads (shared memory between cores)
- Temporary data for each stage is not shared.

I-Cache Locality Running the same code many times for better temporal locality in instruction cache.

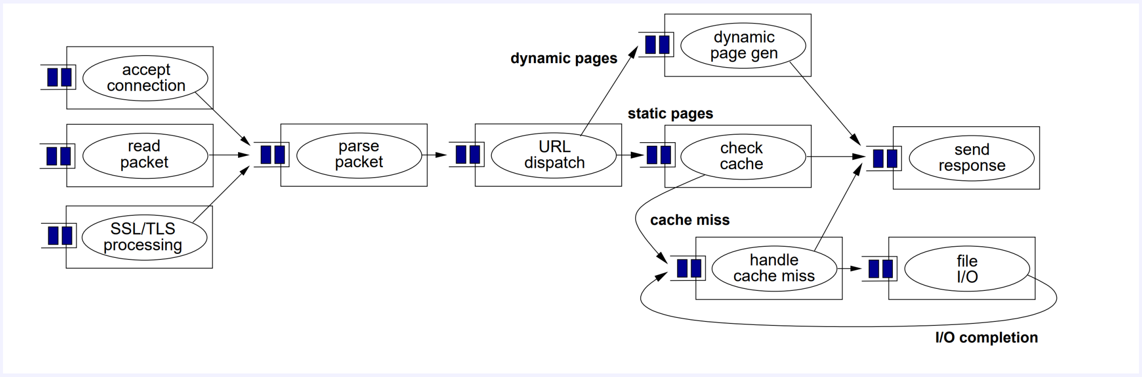
D-Cache Locality	Data must be input and output from stages. Hence we necessarily get slower initial access times due to sharing data & cache coherency protocol.
Pipeline Woes	Get common issues related to unbalanced pipeline stages/pipeline fragmentation.
Flexibility	If the number of tasks increases of decreases, we cannot just spawn or remove a thread (as with work stealing, dispatch etc), we need to add/remove an entire pipeline.

Staged Event Diven Architecture (SEDA) **Definition 5.6.1**

A combination of streaming and worker-pool based systems.

- Each stage has an input queue, as well as its own thread pool.
- When a bottleneck occurs (e.g a queue becomes full) a stage can make a decision (e.g drop a request)
- Flexible/can react to load (i.e stages can spawn or remove threads from their pools)
- Queues mean stages can be inspected to see contents & size to optimise for load.

The paper behind SEDA provides a webserver example:



5.7 Allocators

A single global allocator is a source of contention in a multithreaded system.

- Use a cache of blocks per-thread, this cache can be accessed without contention.
- Does global allocation & free in batches.

5.8 Multiprocessing

Multiprocessing **Definition 5.8.1**

A process can contain several threads (multi threading). Multiprocessing involves using multiple processes.

- Synchronisation achieved with structures shared between processes (e.g a *futex* can be shared between processes)
- There is no shared memory by default (shared pages can be used to allow shared memory between processes), hence processes cannot interfere/are separated
- Communication between processes is explicit and expensive (requires a syscall)

Processes can even be run on separate machines, and explicit communication implemented using a network (e.g Apache Beam, Hadoop).

Multiprocessing is best used when a system executes independent tasks with simple, explicitly expressed communication.

- A common pattern is to assign processes to groups of independent tasks
- UNIX systems allow for easy multiprocessing through pipes (`process1 | process2`)

5.8.1 Communication

Common interprocess-communication mechanisms include:

Explicit Shared Memory

Zero-Copy	Just as with multithreading, large data can be communicated without copying.
No Syscalls	Once shared memory is setup between processes, communication can occur entirely in userspace.

No Interface	There is no standard interface, operating systems (e.g Linux, Windows, Fuschia) have their own shared-memory implementation and interface (syscalls).
Local-Only	Requires the processes to run on the same machine.

Sockets

Uniform Interface	Across operating systems, allowing the programmer to write without considering the system to deploy on, or even if the communicating processes are on the same machine.

Copies	Needs to copy communicated data, even if communication is machine-local.
Overhead	Needs to use the provided network stack.

Pipes

Common Interface	Supported by many unix-style operating systems, and used by many applications.

Copies	Must copy communicated data to a buffer (producer-consumer pattern).
Local-Only	Cannot distribute processes over multiple machines.

5.9 Programming Models

UNFINISHED!!!

Chapter 6

Tools

6.1 Motivation

C++ is a very complex language, that combines:

- High level abstractions (templates, constructors & destructors (to RAII), macros, polymorphism from inheritance/virtuals)
- Fine grained, manual control over memory (can use malloc, new, other allocators, in heap, or even on the stack) and memory access (all the power of C's pointer arithmetic)
- Undefined behaviour, to allow compilers leverage to optimise (e.g signed integer overflow is undefined in the C++ standard, so that compilers can implement it based on the target architecture)

These provided abstractions are powerful, but can be difficult to check correctness for.

What about Rust? ***Extra Fun!* 6.1.1**

Rust was largely invented as an attempt to solve the correctness issues encountered by C++, without the performance tradeoff made by many other languages with more memory safety (e.g swift, go).

The language is split into safe and unsafe, with necessarily unsafe (but high performance) code being written in unsafe to provide safe abstractions. Safe rust has several advantages over C++:

- No undefined behaviour.
- Any program with data races cannot compile (ownership & borrow checker)
- Complete memory safety (no pointers, borrow checker)
- Sanitary macros (very powerful metaprogramming abstraction - comparable with languages such as Scala)
- Traits and generics (much better error messages than with templates, C++ is attempting this with concepts)

For unsafe rust sanitizer-like tools are available using MIRI.

Rust's performance is comparable with C++.

6.2 Correctness

6.2.1 Patterns

Resource Aquisition is Initialisation (RAII) **Definition 6.2.1**

UNFINISHED!!!

6.2.2 Compiler Warnings

Compilers already provide a large number of warnings.

- GCC provided warnings
- Clang provided warnings
- Linters can also be used to catch code quality issues, that could potentially result in correctness issues in future (hard to understand code leads to misunderstandings).

```
# Use all warnings and error on warnings (e.g to fail CI)
g++ ... -Wall -Werror ...
```

It is possible to enable and disable compiler warnings from the code.

```
// Selecting levels for -Wformat
#pragma GCC diagnostic warning "-Wformat"
#pragma GCC diagnostic error "-Wformat"
#pragma GCC diagnostic ignored "-Wformat
```

More can be found in GCC's diagnostic pragmas documentation, they can also be specified as attributes (see attribute specifiers).

6.2.3 Sanitisers

Sanitisers instrument code at compile time to detect errors at runtime.

- Typically take large performance reduction
- Known bugs are present in sanitisers (though these are typically very rare edge cases)
- GCC provides several that can be found here

Address Sanitizer (Asan) Definition 6.2.2

Detects out of bounds, use after free and other memory access related bugs.

```
g++ ... -fsanitize=address
```

Undefined Behaviour Sanitizer (Ubsan) Definition 6.2.3

Checks for a large number of undefined behaviours in C++

```
g++ ... -fsanitize=undefined
```

Thread Sanitizer Definition 6.2.4

A data-race detector, uses a vector clock algorithm to detect data races at runtime.

```
g++ ... -fsanitize=thread
```

Malloc weirdness Example Question 6.2.1

Does the following code segfault? Use Asan to determine if the following code has any memory related bugs.

```
#include <stdlib.h>
#include <iostream>

int main() {
    int* a = (int*) malloc(100);
    a[-1] = 3;

    std::cout << "here"<< std::endl;
}
```

```
g++ memory_example.cpp -o memory_example
./memory_example

# result of code
here
```

There is no segfault (this is due to how malloc chunks work as explained here), however we are accessing outside the bounds of the memory allocated.

When running with asan we see the warning:

```
g++ memory_example.cpp -o memory_example -fsanitize=address
./memory_example

# asan output
==1094==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60b0000000ec at ...
# The error originates in "main"
```

6.2.4 Valgrind

Valgrind provides tools to detect memory issues and data races.

Memcheck

```
valgrind --tool=memcheck program # place args on end
```

- Should include debugging symbols when compiling
- Much slower than sanitizers, but can take any binary.
- Memcheck documented here

Memcheck this! **Example Question 6.2.2**

Use Valgrind Memcheck to find the incorrect index in the previously checked program.

```
# include debug symbols
g++ memory_example.cpp -g -o memory_example
valgrind --tool=memcheck ./memory_example

# valgrind output:
==2375== Invalid write of size 4
==2375==    at 0x1091EB: main (memory_example.cpp:6)
==2375==  Address 0x4db2c7c is 4 bytes before a block of size 100 alloc'd
==2375==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/ ...
==2375==    by 0x1091DE: main (memory_example.cpp:5)
```

Helgrind

```
valgrind --tool=helgrind program # place args on end
```

Data race detection using valgrind.

6.3 Performance

6.3.1 VTune

UNFINISHED!!!

6.3.2 Perf

UNFINISHED!!!

6.3.3 Coz

UNFINISHED!!!

Chapter 7

System Interfaces

7.1 Operating System Interfaces

7.2 CPU Interfaces

7.3 Virtualization

7.4 I/O Interfaces

UNFINISHED!!!

Chapter 8

Scale Out

8.1 Design Considerations

8.2 Microservices

8.3 Serverless

8.4 Function as a Service

FaaS **Definition 8.4.1**

8.5 Communication

Chapter 9

Credit

Content

Based on the System Performance Engineering course taught by Dr Holger Pirk and Dr Luis Vilanova.

These notes were written by Oliver Killane.