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

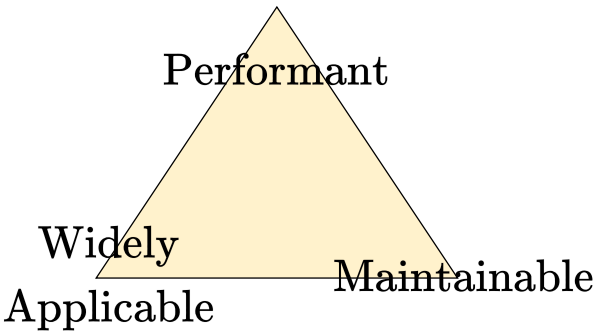
System **Definition 1.2.1**

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* . . .) $60 fps$ if run on a GPU rated at $50 GFlops$ 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.

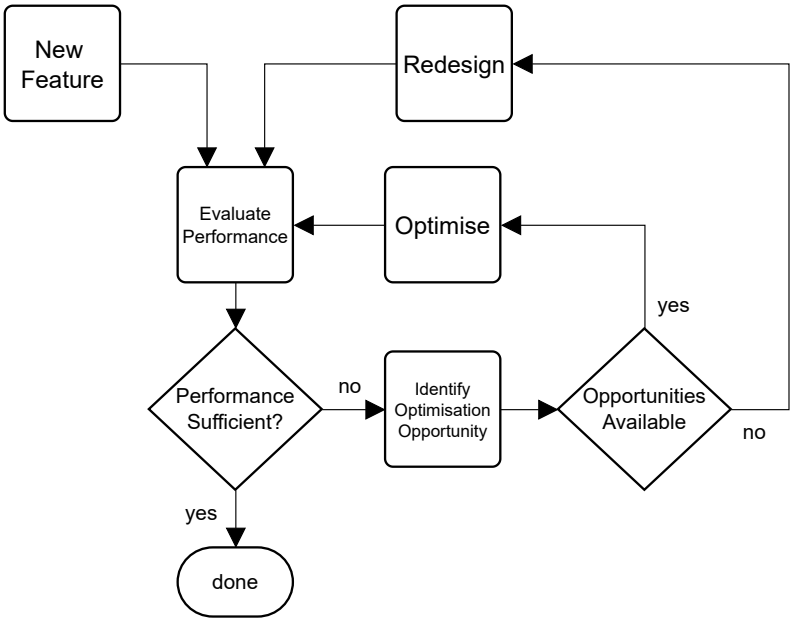
- Useful when a latency metric is being measured.
- Workload generator needs to fast enough to saturate system being benchmarked.
- Often more representative of a production environment (e.g an operating system receives a workload over time)

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**

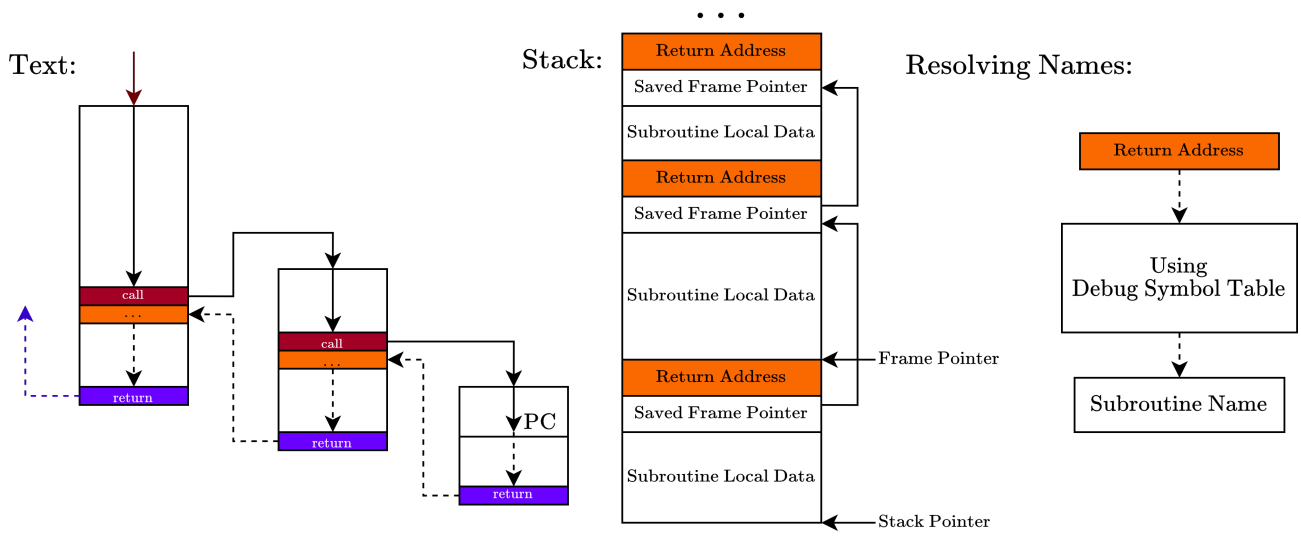
A complete log of the system's state (and hence changes in state/events) for some time period.

- 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 Less Pertubation	Fidelity traded against pertubation for reduced overhead in recording events. 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.

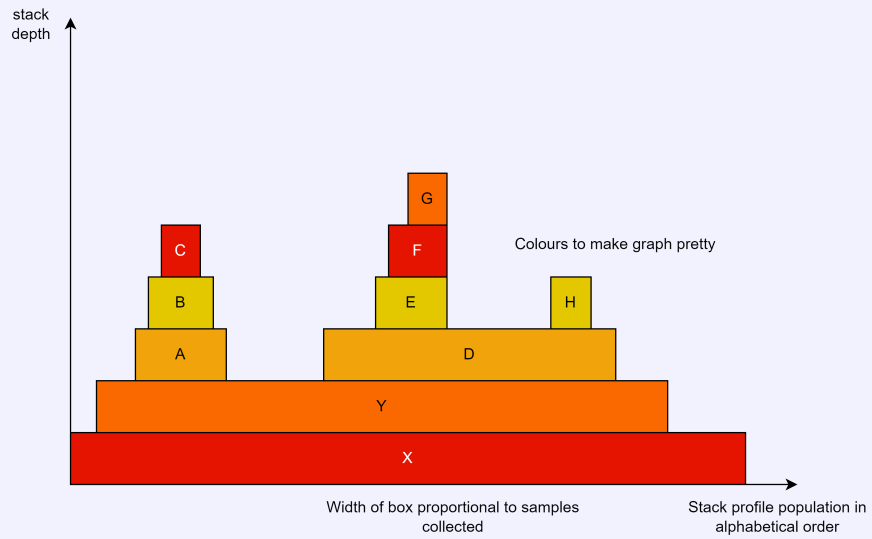
- Can be used to reduce overhead from tracing
- fidelity and accuracy good (depending on the events traced and the indirection used)

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**



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.

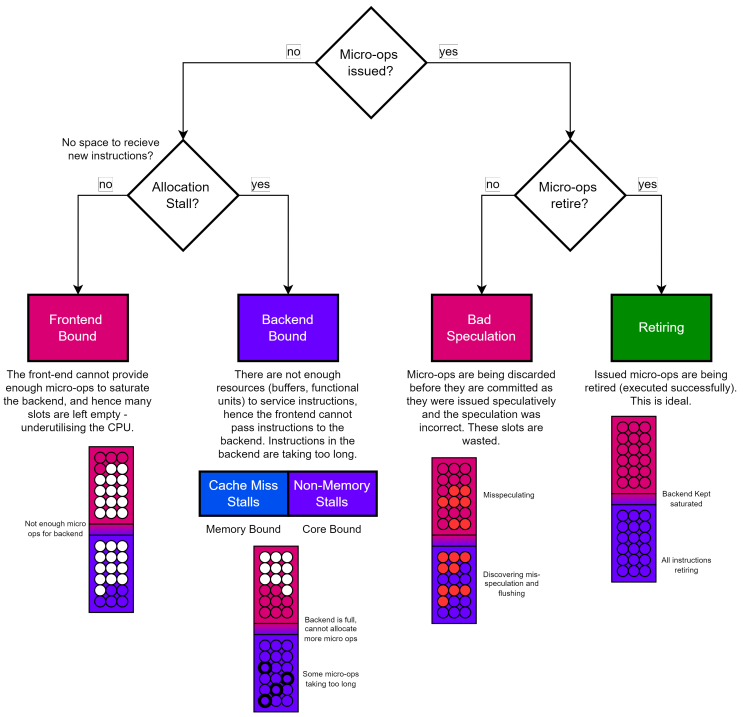
- Microarchitectural features documented.
- Written as an optimisation guide.
- Code example can be found in its github repo

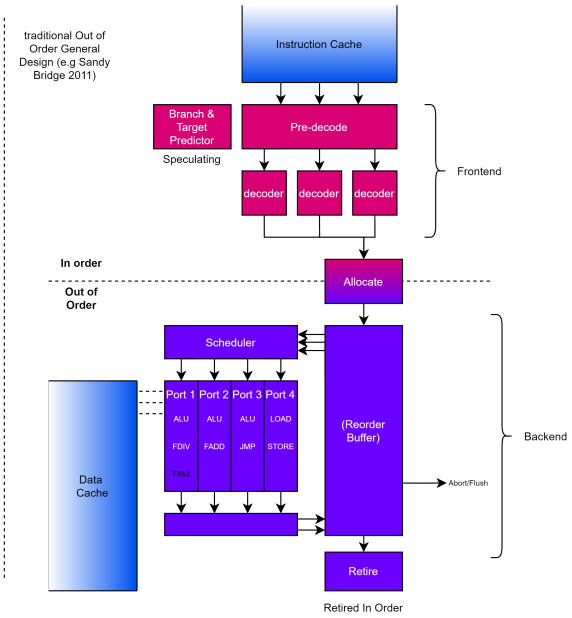
2.4 Microarchitectural bottleneck analysis

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.





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

2.5 Vtune

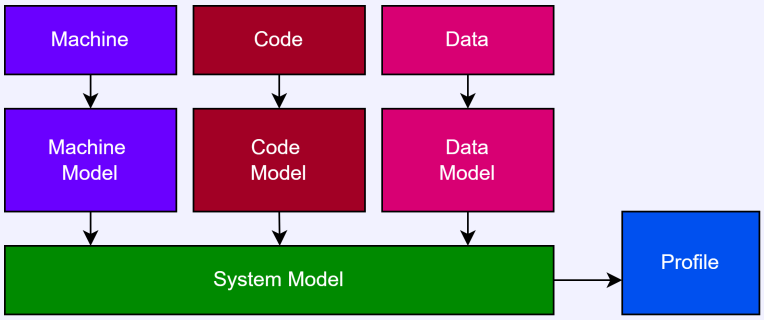
UNFINISHED!!!

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:

```
#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

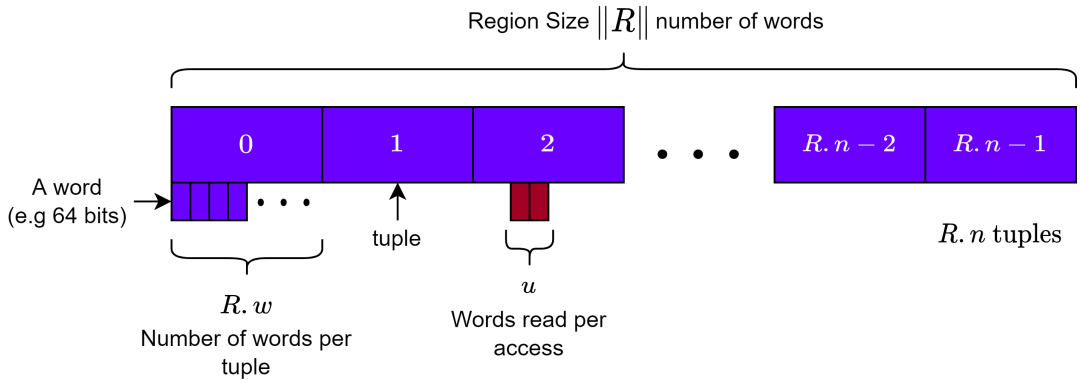
Some parameters can be taken from documentation, others may be derived from experiment.

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)

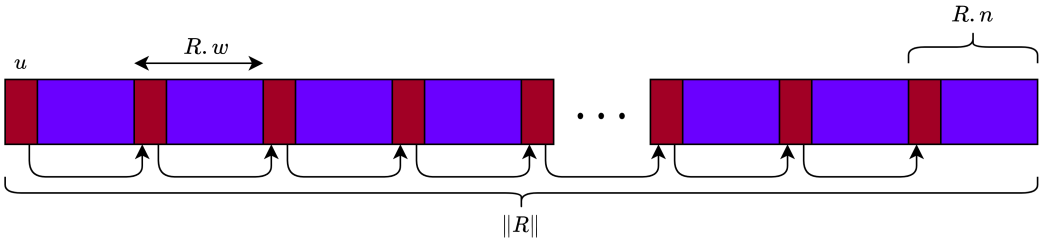
3.4 Memory Access Patterns

3.4.1 Memory Region



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

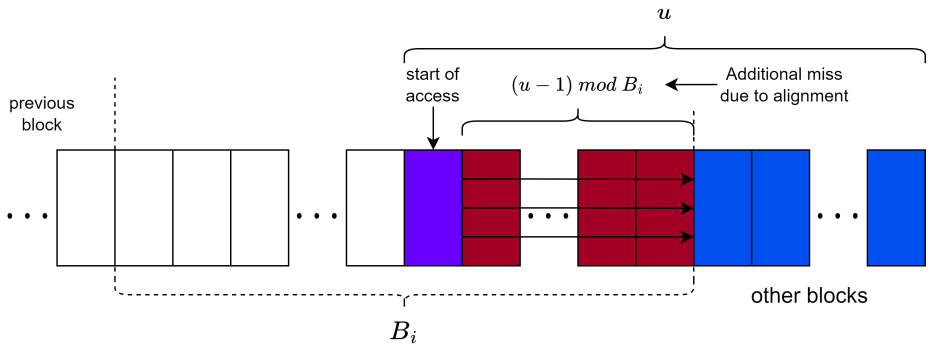
3.4.2 Sequential



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*

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

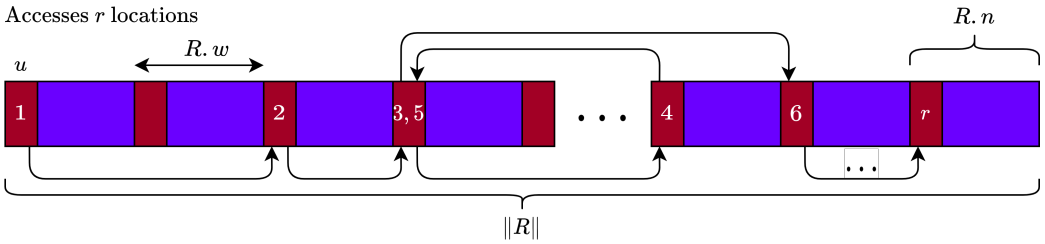


We may want to consider additional misses due to misalignment (u stretching over two Bs), in which case we much change the $R.w - u \geq B_i$ case to:

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

3.4.3 Repetitive Random Access

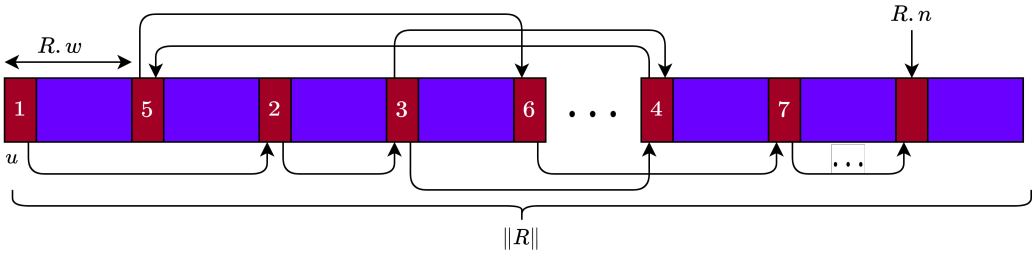
$$M_i^s(rr\_acc) = \text{undefined in lecture}$$



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.

```
#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.

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

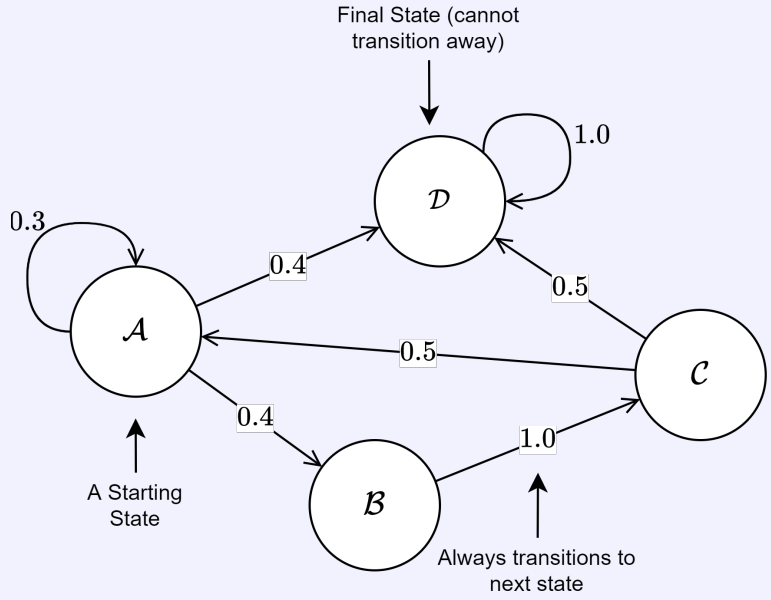
3.5 Modelling State

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

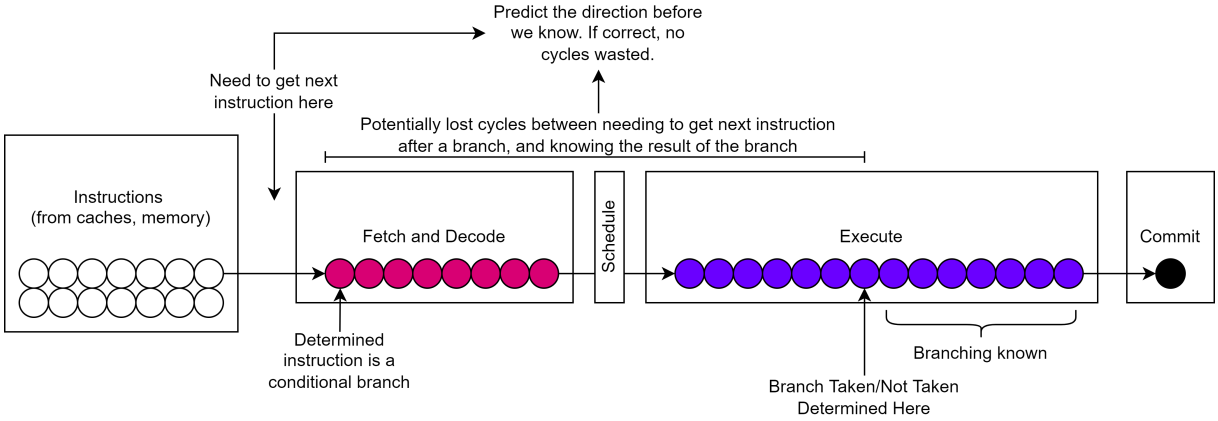
Consider a simple conditional branch, to a known target (instruction). For example a basic loop:

```
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
```

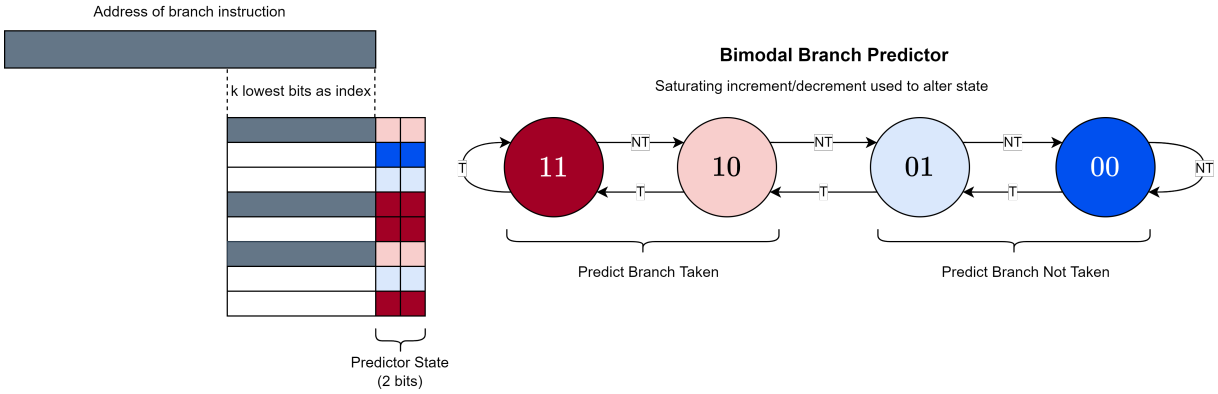
We will focus on the conditional branches with a known target (target prediction is more complex).



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 Diustribution **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

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.