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

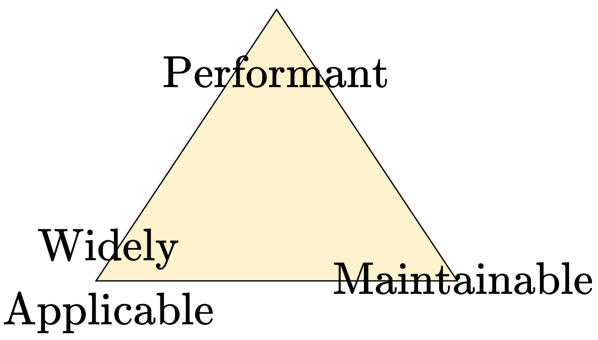
System **Definition 1.2.1**

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.

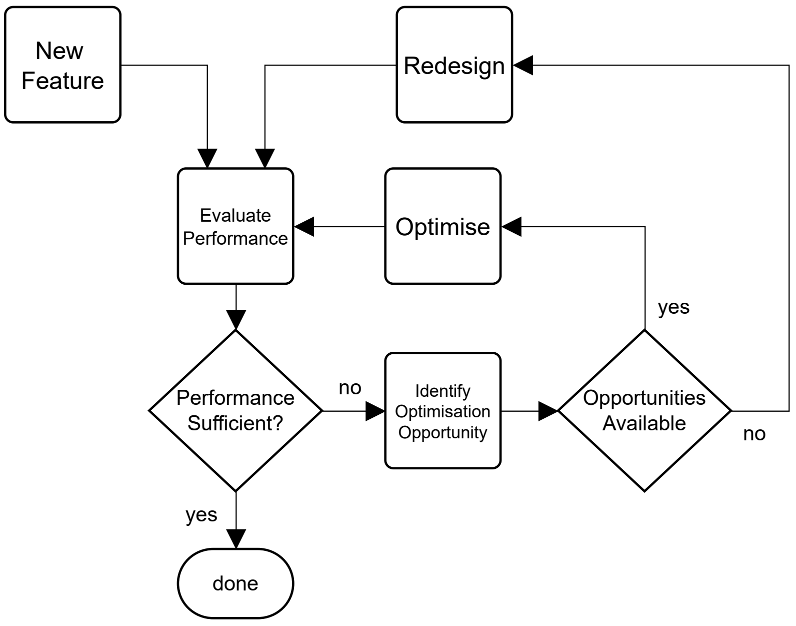
- Useful when a latency metric is being measured.
- Workload generator needs to fast enough to saturate system being benchmarked.
- Often more representative of a production environment (e.g an operating system receives a workload over time)

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**

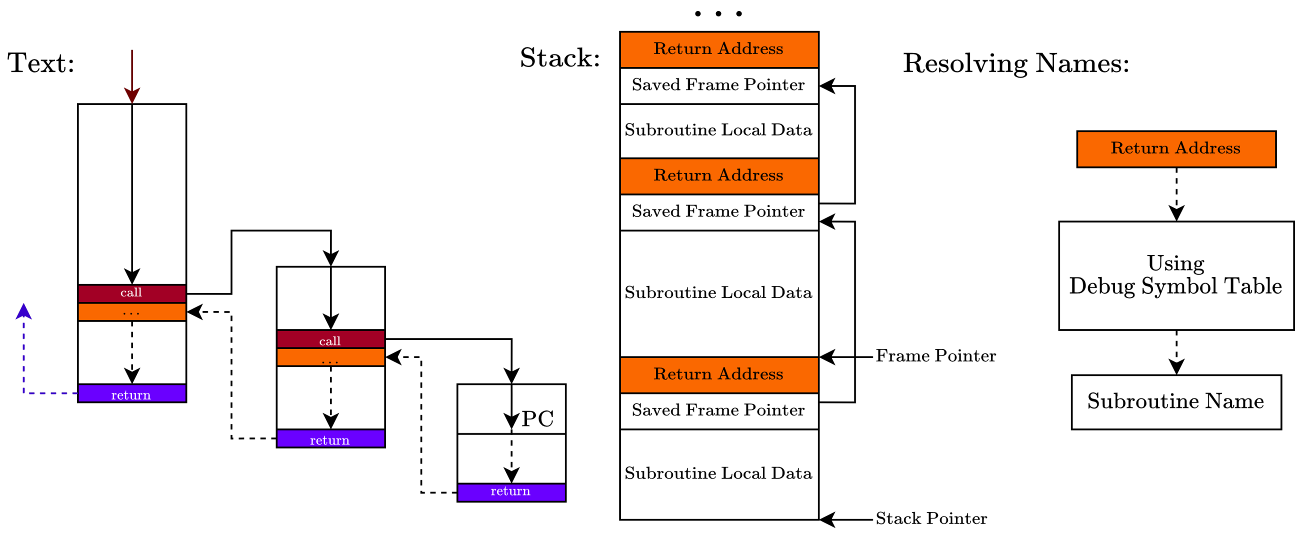
A complete log of the system's state (and hence changes in state/events) for some time period.

- 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), effectively indirectly tracing them.

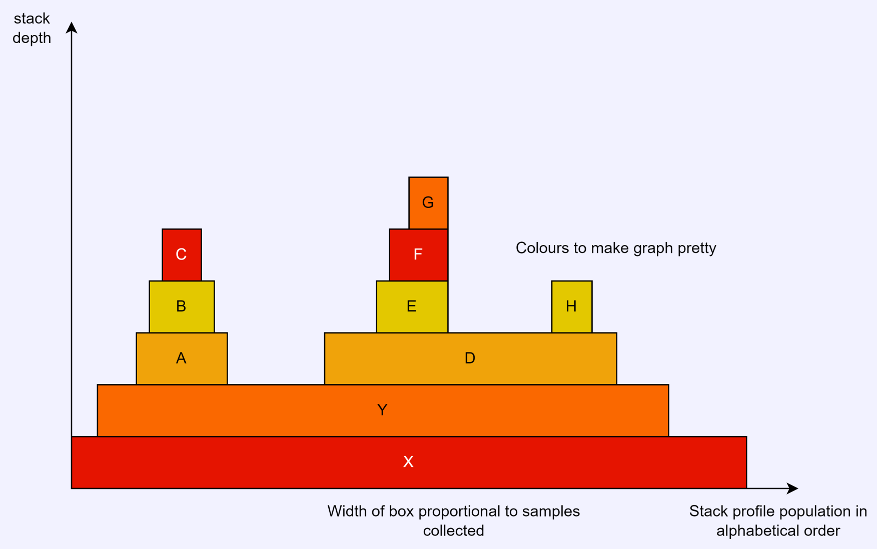
- Can be used to reduce overhead from tracing
- fidelity and accuracy good (depending on the events traced and the indirection used)

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) $\to$ 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.

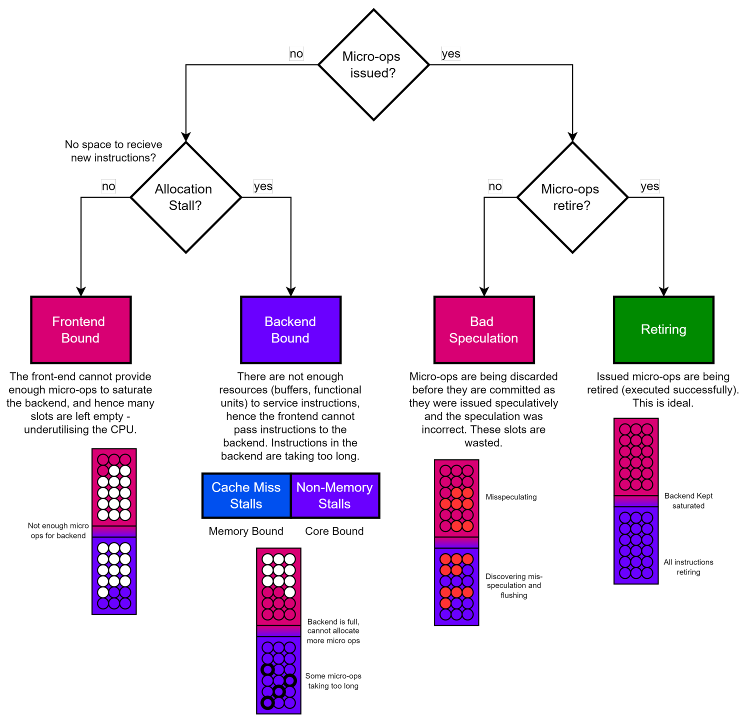
- Microarchitectural features documented.
- Written as an optimisation guide.
- Code example can be found in its github repo

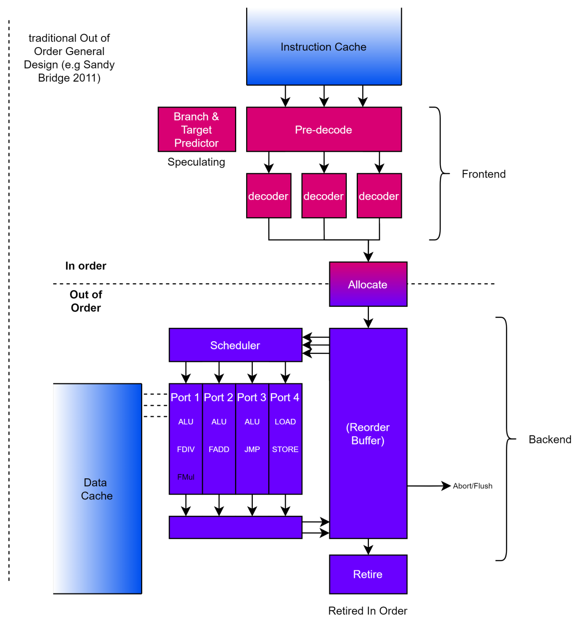
2.4 Microarchitectural bottleneck analysis

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



2.5 Vtune

UNFINISHED!!!

Chapter 3

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.