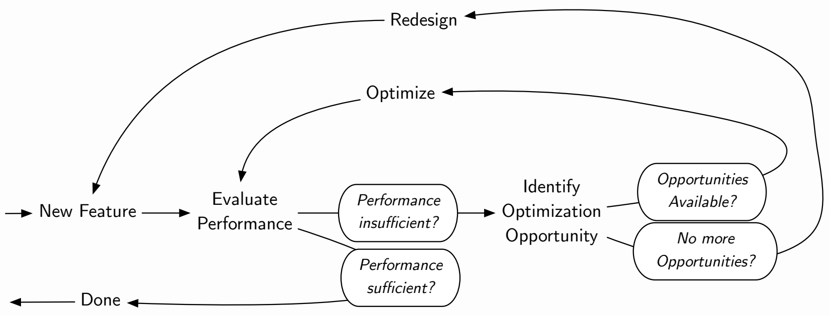
Performance Tracing & Profiling

Holger Pirk

Slides as of 14/01/22 11:57:44

Recall:

The Optimization Loop



So, how do we identify optimization opportunities?

How to identify optimization opportunities

- We identify the hot path (the code that takes the most time)
- We identify the bottleneck
  - in terms of CPU, Memory, Network, ...
- Both are functions of the system behavior

So, how do we describe system behavior?

What are events?

- Definition: *Any change of the system state*
- Usually restricted to a certain granularity
  - Simple/atomic events
    - sent package, executed instruction, loaded address from memory
    - clock has ticked
  - Complex events
    - cache line evicted from L1 to L2 cache, instruction aborted due to misspeculation
- Events have an optional *payload*
- An event has an *accuracy*: the degree to which its value represents reality

What can you do with events?

Where they come from

- *Event Sources* are have two components
  - The *generator* observes the changes to the system state
    - Usually online, i.e., part of the runtime environment/system
  - The *consumer* processes the events
    - Can be offline or online

Where they go

- Tracing
- Profiling

Trace

- Definition: *A complete log of every state the system has ever been in (in the period of interest)*
  - Comprised of events
  - Events are ordered (usually totally ordered)
- Accuracy is "inherited" from the vents
- Event collection overhead may be high

Example: Call stack tracing

Call stack tracing

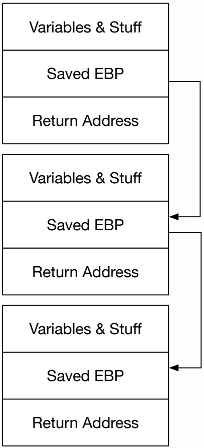
A typical call stack

Address
0x231fa90
0x7828b72
0x8913ee1

- So, what does a stack look like in reality?

A call stack

Illustration



Call stack tracing

Problems

- Recording the entire call stack is quite expensive
  - The stack needs to be walked, pointers need chasing
  - Call stacks can be deep
  - All frame pointers have to be written to memory
- In particular for small/cheap functions, call stack processing can be way more expensive than the function itself

We call this problem…

Perturbation

Definition: Perturbation

The degree to which the performance of a system changes when it is being analyzed.

- Perturbation negatively affects accuracy if it is non-deterministic
- A bit like quantum theory
  - You influence the state of the system just by looking at it

How do we reduce perturbation?

How to reduce perturbation

- We reduce *fidelity*
- Fidelity (Oxford Dictionary): *the degree of exactness with which something is copied or reproduced*
- Perfect fidelity, i.e., every event is recorded
- Reduced fidelity, i.e., not every event is recorded

How?

Sampling

- Idea: do not collect all events to reduce perturbation
- Option 1: Sample in regular intervals
- Option 2: Sample in random intervals

Example: Call stack sampling

- Idea: Skip some events
  - there is a chance you will not sample a function
  - Fortunately, more expensive functions will be sampled more often
- But:
  - good performance
  - even more important: less perturbation
  - fidelity can be traded against performance/fidelity

What is an interval?

Sampling Intervals

- The distance of two samples being taken
  - Obviously, interval size 1 makes sampling equal to event-tracing
- Two options for specification: time-based and event-based

Time-based Intervals

- Idea: set a (hardware) recurrant timer and sample whenever it runs out
- We use CPU *reference cycles* as a proxy metric
- Inaccurate, non-deterministic and noisy (computer clocks are poorly defined)
  - Clock rate varies, clocks may not be exactly synchronized among CPUs, etc.
- Easy to interpret (since time is inversely proportional to performance)

Event-based Intervals

- Generalization of Time-based intervals (since computer time is discrete)
- Define an interval in terms of the occurence of an event
- Example: sample every fifth function call
- Accurate, deterministic semantics and low noise
- Tricky to interpret (in the end, we are interested in time)

Quantization errors

- Interval resolution is limited (usually to single clock cycles but sometimes more)
- Time is (practically) continuous
- This introduces "quantization errors/biases"
    - E.g., costs being attributed to the wrong state

Here is an interesting instance of event-based intervals:

Indirect Tracing

- Idea: trace events that dominate others
  - Think of it as intervals defined by the execution flow
  - For example, control-flow instructions (`if`, `else`, `for`, `while`) dominate non-control-flow instructions
  - can be used to reduce overhead
  - Fidelity and accuracy depends on the event and the indirection

Wrapup: Tracing

- Tracing collects (subsets of) events
- Perturbation is a problem but can be worked around
- **But:** Analyzing traces is extremely tedious
    - Lots of data, little structure, lots of cognitive overhead

The solution:

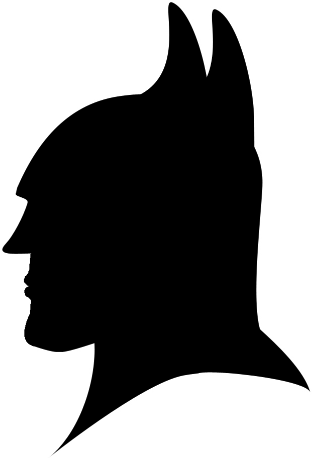
Profiling

Definition: Profile

An outline of something, especially a person's face, as seen from one side.

Profiling

A profile



Profiling

Definition: Profile

A graphical or other representation of information relating to particular characteristics of something, recorded in quantified form

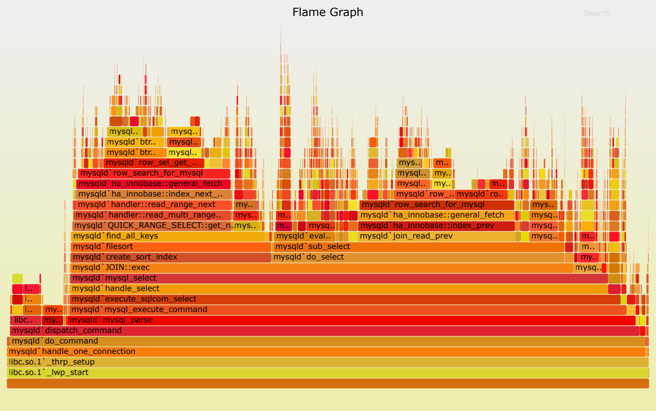
In our context

A characterization of a system in terms of the resources it spends in certain states.

Profile

- An aggregate over the events of a specific metric
  - This can be a global aggregate
    - Total cache misses, total CPU cycles
  - Or broken down by some other event
    - Cycles per instruction (CPI), cache misses by line of code
- Caution: Information is lost
- Why?
  - Post-mortem for ease of interpretation
  - Realtime to reduce perturbation (assuming aggregation is cheaper than dumping)

Flame Graphs



https://queue.acm.org/detail.cfm?id=2927301

Flame Graphs

- X-axis shows the stack profile population, sorted alphabetically (not by time),
- Y-axis shows stack depth
- Each rectangle represents a stack frame
- Width of a box is preoportional to the number of collected samples
- Colors are usually not significant

Okay, now that we know what to do with events. . .

. . . let us talk about specific ways to collect events

Requirements for event sources

- Detailed
  - As much information as we need
- Accurate
  - The measurements should closely describe the real-world
- Little perturbation

Where to get events?

- Software
  - Library: Manual Instrumentation/Logging
  - Compiler: Automatic Instrumentation
  - OS: Kernel Counters
- Hardware:
  - Performance counter
- Emulator:
  - a funky hybrid, minimal perturbation but usually not scalable

Instrumentation

- Augmenting program with event logging code
- Advantages
  - No need for any hardware support
  - **very** flexible
- Disadvantages
  - Overhead is high
  - Perturbation is high

Instrumentation

- Three approaches
    - Manual Instrumentation
    - Automatic source-level instrumentation
    - Automatic binary instrumentation
        - Static (compile-time) or
        - Dynamic (runtime)
        - As usual, there are hybrids

Manual

- basically `printf` logging (or using a logging library)
- Advantages
  - Fine control over instrumentation
  - Needs no support from hardware or compiler
- Disadvantages
  - high overhead for implementation & runtime
  - usually disabled for release build
    - needs recompilation for selective enabling

Automatic

- Usually compiler-supported
- Source-to-source rewriting is possible
- Disadvantages
  - Less control
  - Need for compiler support
- Advantages
  - Let's discuss this!

Binary Instrumentation

- Static
  - No magic, simple, portable
  - Instrumentation overhead can be assessed from binary
- Dynamic
  - No recompilation
  - Can be performed on running process
  - Works with JiT-compiled code

- http://llvm.org/docs/XRay.html

LLVM-XRay

```
curl --compressed https://www.gutenberg.org/cache/epub/2229/pg2229.txt | iconv -c -f UTF8 -t
↪  ASCII | tr -d '\r' > faust.txt
for i in {1..1000}; do cat faust.txt >> faust1000.txt; done
clang++ -g -O0 -fxray-instrument -fxray-instruction-threshold=1 ~/pegrep.cpp
XRAY_OPTIONS="patch_premain=true xray_mode=xray-basic verbosity=1" ./a.out faust1000.txt

llvm-xray convert -f yaml -symbolize -instr_map ./a.out xray-log.a.out.* | less
llvm-xray account -sort=count -sortorder=dsc -instr_map ./a.out xray-log.a.out.*
```

LLVM-XRay

Explanation

The logging functions by default prune records that are less than 5 microseconds equivalent in walltime deduced from the cycle counter deltas. This allows XRay to retain only records that have a measurable impact in walltime.

We want higher fidelity/lower overhead!

The solution: Hardware Support!

Software Performance Counters (OS)

- Network Packages sent
- Virtual Memory Operations
- ...
- Let's say We want to write code that is efficient at the microarchitectural level

Software is good, Hardware is better!

Hardware Performance Counters

- Special registers that can be configured to count low-level events
  - Fixed number can be active at runtime
- Can be used to define collected events as well as intervals
- Unfortunately:
  - Often buggy or unmaintained
  - Sometimes poorly documented
  - Accuracy can be poor
    - The common ones are usually okay

Examples

Try this

```
hlgr@sprite17:~$ perf list pmu | egrep "^  [^ ]" | less | wc
    802     1009    45255
```

Examples

And this

```
hlgr@sprite17:~$ perf list pmu | tail +53 | head -n 20
cache:
  l1d.replacement
       [L1D data line replacements]
  l1d_pend_miss.fb_full
       [Number of times a request needed a FB entry but there was no entry
        available for it. That is the FB unavailability was dominant reason
        for blocking the request. A request includes cacheable/uncacheable
        demands that is load, store or SW prefetch]
  l1d_pend_miss.pending
       [L1D miss outstandings duration in cycles]
  l1d_pend_miss.pending_cycles
       [Cycles with L1D load Misses outstanding]
  l1d_pend_miss.pending_cycles_any
       [Cycles with L1D load Misses outstanding from any thread on physical
        core]
  l2_lines_in.all
       [L2 cache lines filling L2]
  l2_lines_out.non_silent
       [Counts the number of lines that are evicted by L2 cache when triggered
        by an L2 cache fill. Those lines are in Modified state. Modified lines
```

But most importantly

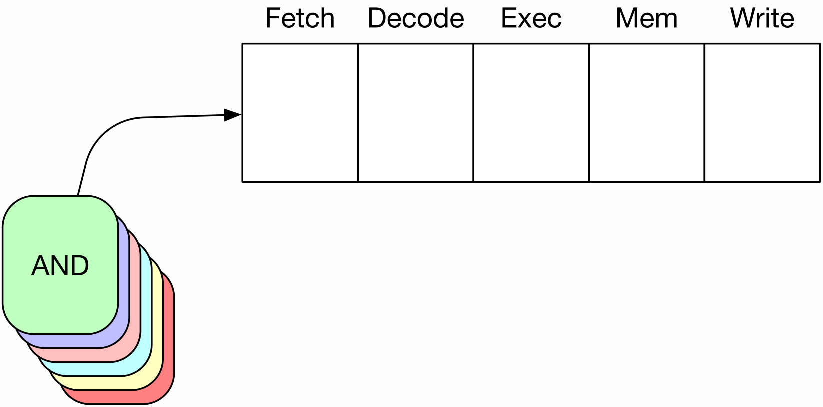
RTFM!

i.e., the "Intel 64 and IA-32 Architectures Optimization Reference Manual"

How does a CPU work?

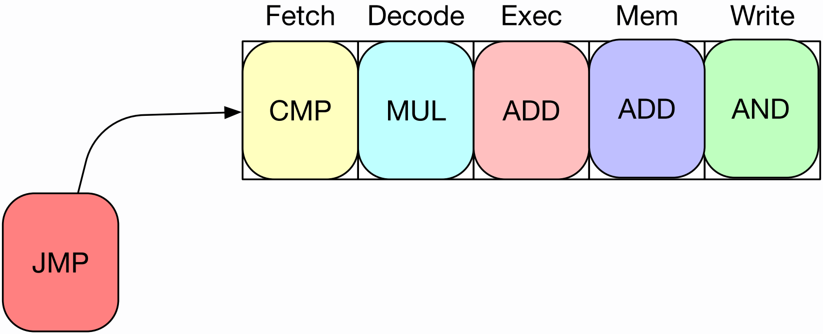
How does pipelined execution work?

An Empty CPU Pipeline



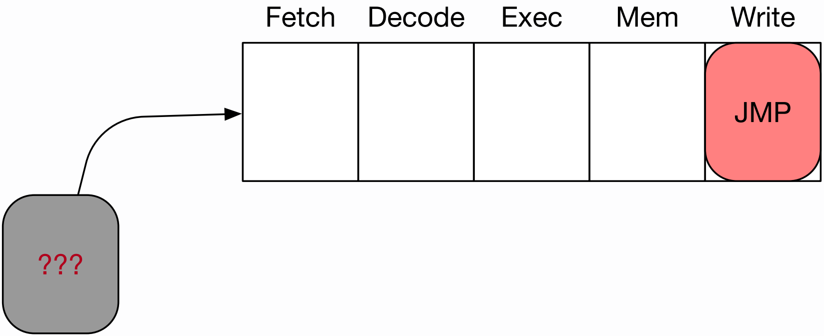
How does pipelined execution work?

A Filled CPU Pipeline



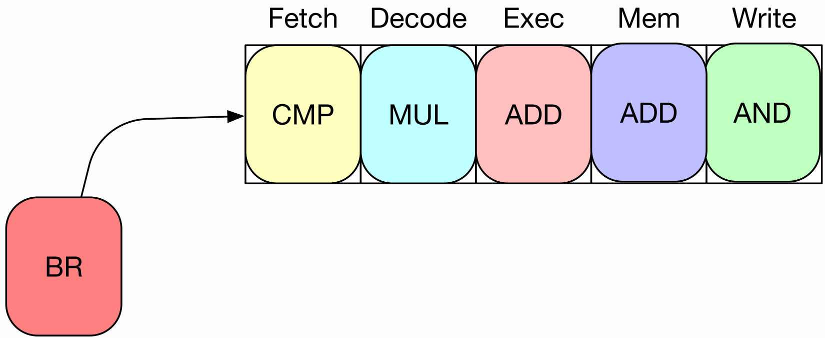
How does pipelined execution work?

A control hazard/pipeline bubble



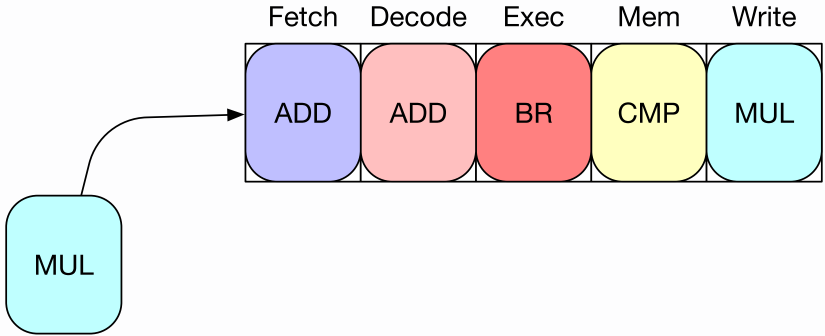
How does Branch Prediction work?

A branch to be predicted (a.k.a., speculated upon)



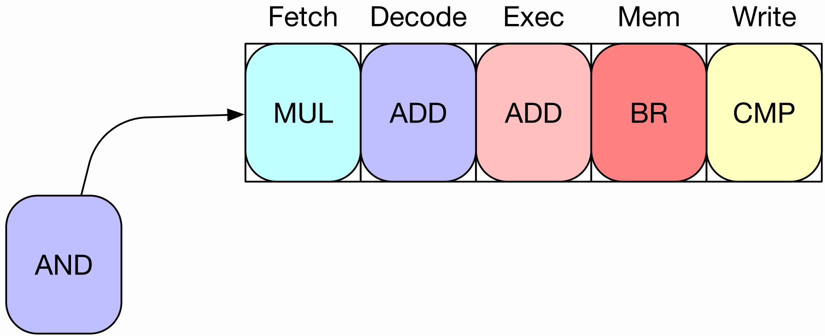
How does Branch Prediction work?

A speculatively executed branch



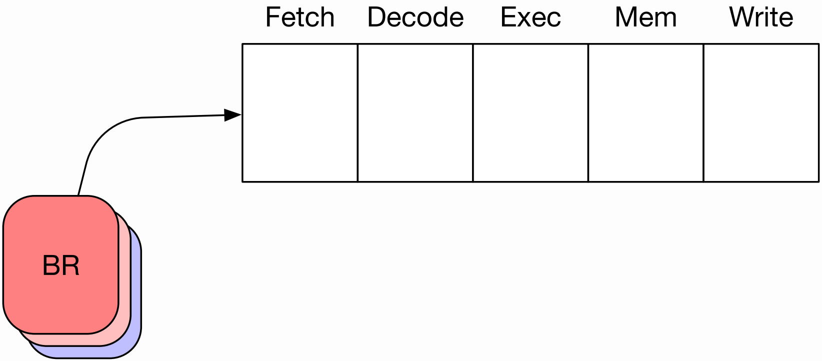
How does Branch Prediction work?

Time of prediction resolution



How does Branch Prediction work?

Rollback upon misprediction

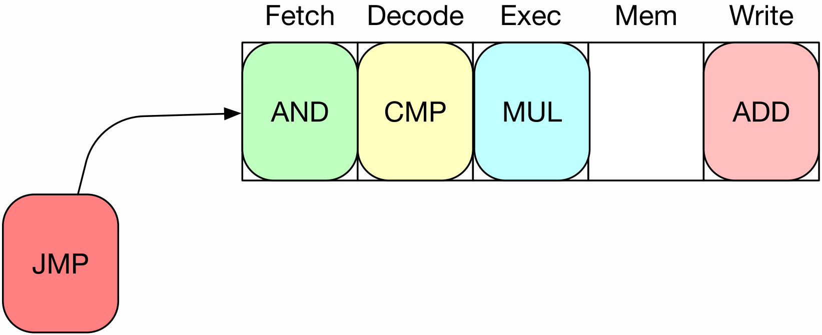


How does pipelined execution work?

- Bottom line:
  - CPUs can stall on control dependencies

Resource Stalls

An ALU Stall

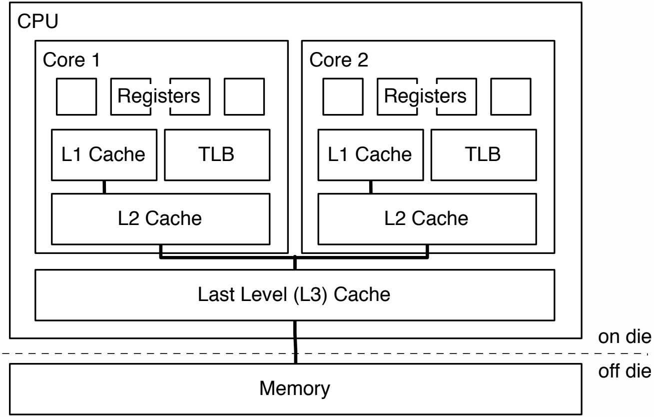


Resource Stalls

- Bottom line:
  - CPUs can stall due to lack of compute resources

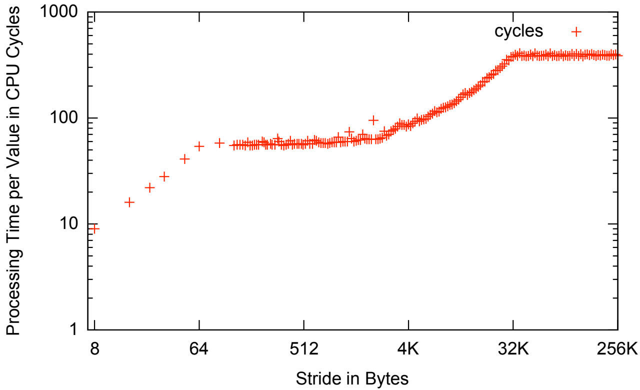
How does the memory subsystem work?

A CPU



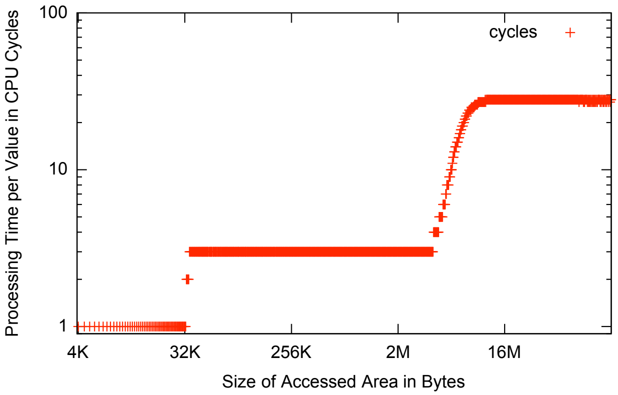
How does the memory subsystem work?

Memory Access Latencies (depending on locality)



How does the memory subsystem work?

Memory Access Latencies (depending on data size)



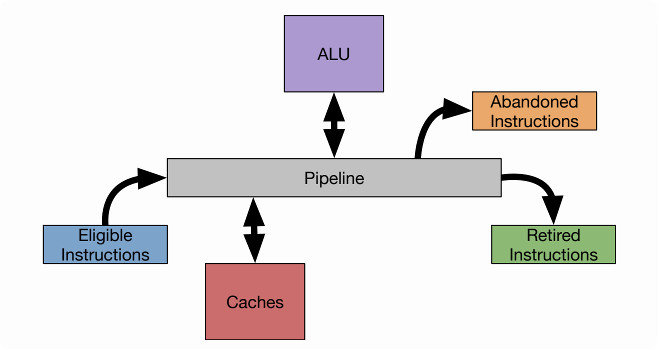
How does the memory subsystem work?

- Bottom line:
  - CPUs can stall on data access

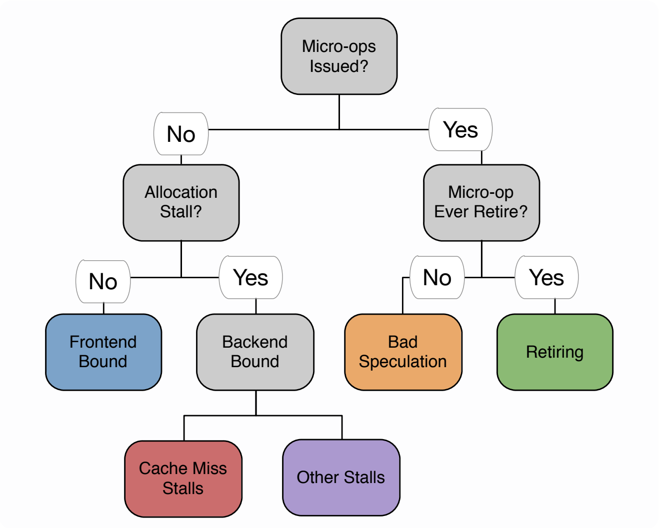
Bottleneck analysis

- Let's find some microarchitectural bottlenecks
  - Data Stalls
  - ALU Stalls
  - Branch Mispredictions
  - Control-flow dependencies

Bottleneck analysis



Bottleneck analysis



Provide feedback, please!

https://co339.pages.doc.ic.ac.uk/feedback/profiling

Get the slides online

https://co339.pages.doc.ic.ac.uk/decks/Profiling.pdf