Performance Modeling

Holger Pirk

Slides as of

Get the slides online

https://scientia.doc.ic.ac.uk/api/resources//file/PerformanceModeling.pdf

Motivation

Where we stand

- We can (empirically) determine performance metrics of hardware & software systems if we have access to
  - hardware to run it on
  - the code
  - the input
- **What happens if we lack one of these?**
  - We need to model it!
- Why would that happen, you ask?

Why would we need analytical performance modeling

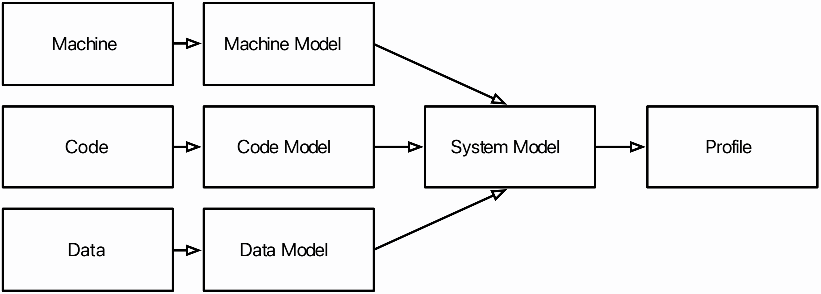
- When we want to know performance "on the cheap" (i.e. without running)
  - For charging before execution
  - For provisioning systems
  - Other reasons?

When and why would we need analytical performance modeling

- For runtime optimization and tuning of course!
  - Think JiT compilers,
  - databases,
  - machine learning toolkits,
  - etc.

Performance Modeling

System (Model) Aspects



Alright, let's model something!

Before we start. . .

Operating assumptions

- We make simplifying assumptions about the input
  - We assume a known distribution (usually uniform without correlation)
- We do not model system noise
  - Could be caused by scheduling, other processes, external factors, ...
- In this lecture, we assume single-threaded, deterministic code
  - Modeling contention in parallel systems is an open research topic

Performance modeling approaches

- Two approaches:

Numerical/Experimental Model

- A series of datapoints describing the observed behavior of the system
- Useful to describe system behaviour for humans
- Predictive power depends on interpretation (example is coming up)

Analytical Model

- A formal characterization of the relationship between parameters and performance metrics
- Often difficult to interpret for humans (moderately useful to describe system)
- Prediction is performed by evaluating the model

Numerical models

Numerical models step 1: gathering data

What we want

Parameter	Metric
0	1
1	0
2	3
3	2
4	2
5	4
0.5	0
1.5	1
2.5	3.2
3.5	1.9
4.5	3
5.5	6

But how do we get pristine results like this?

Numerical models step 1: gathering data

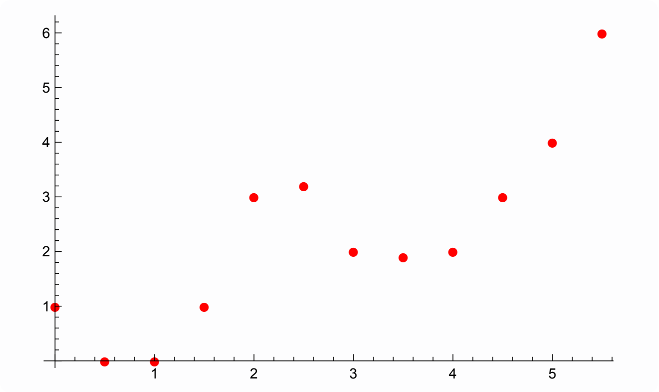
- Through *Microbenchmarking*
- "*Microbenchmarks* are small, specially designed programs used to test some specific portion of a system"

Numerical models step 1: gathering data

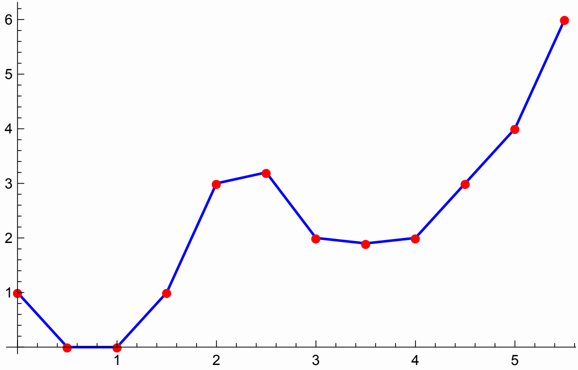
A Memory Subsystem Microbenchmark

```
extern int* input;
extern size_t N;      // some large constant
extern size_t stride; // the parameter of our experiment
int sum = 0;
for(size_t i = 0; i < N; i += stride) {
  sum += input[stride];
}
```

Numerical models step 2: interpret



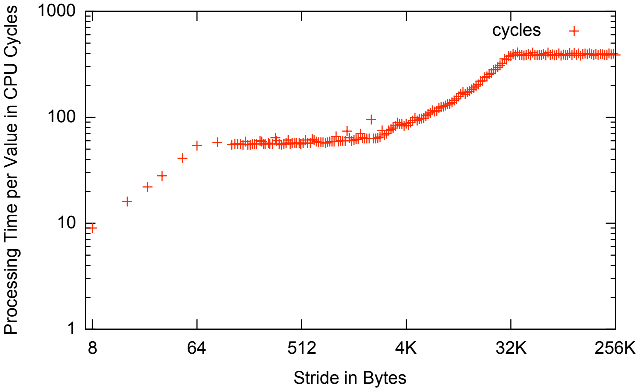
Numerical models step 2: interpret



- Prediction through, for example, interpolation

Example

Numerical models step 2: interpret



Numerical models pro/cons

- Advantages

Numerical models pro/cons

- Advantages
  - Easy to get (if the system is available)

Numerical models pro/cons

- Advantages
  - Easy to get (if the system is available)
  - Based on ground truth

Numerical models pro/cons

- Advantages
  - Easy to get (if the system is available)
  - Based on ground truth
  - Relatively easy to interpret

Numerical models pro/cons

- Advantages
  - Easy to get (if the system is available)
  - Based on ground truth
  - Relatively easy to interpret
- Disadvantages

Numerical models pro/cons

- Advantages
  - Easy to get (if the system is available)
  - Based on ground truth
  - Relatively easy to interpret
- Disadvantages
  - Generalize poorly (i.e., cannot easily be applied to new environments)

Numerical models pro/cons

- Advantages
  - Easy to get (if the system is available)
  - Based on ground truth
  - Relatively easy to interpret
- Disadvantages
  - Generalize poorly (i.e., cannot easily be applied to new environments)
  - Massive amounts of experimental data needed for high-dimensional system parameter spaces

Numerical models pro/cons

- Advantages
  - Easy to get (if the system is available)
  - Based on ground truth
  - Relatively easy to interpret
- Disadvantages
  - Generalize poorly (i.e., cannot easily be applied to new environments)
  - Massive amounts of experimental data needed for high-dimensional system parameter spaces
  - Limited accuracy for/confidence in prediction (data may be missing, inaccurate, . . .)

Numerical models pro/cons

- Advantages
  - Easy to get (if the system is available)
  - Based on ground truth
  - Relatively easy to interpret
- Disadvantages
  - Generalize poorly (i.e., cannot easily be applied to new environments)
  - Massive amounts of experimental data needed for high-dimensional system parameter spaces
  - Limited accuracy for/confidence in prediction (data may be missing, inaccurate, ...)
  - Limited interpretability: contributing factors are (at best) implicit

Numerical models pro/cons

- Advantages
  - Easy to get (if the system is available)
  - Based on ground truth
  - Relatively easy to interpret
- Disadvantages
  - Generalize poorly (i.e., cannot easily be applied to new environments)
  - Massive amounts of experimental data needed for high-dimensional system parameter spaces
  - Limited accuracy for/confidence in prediction (data may be missing, inaccurate, ...)
  - Limited interpretability: contributing factors are (at best) implicit
  - Limited insight: how does the system actually work?

The alternative:

Analytical models

The alternative: analytical models

$$DA'\_total(R_1, R_2) = \sum_{j=\text{abs}\left|h_{R_1} - h_{R_2}\right|+1}^{\max\left(h_{R_1}, h_{R_2}\right)-1} \begin{cases} DA(R_1, j) + DA(R_2, j'), \text{ if } h_{R_1} > h_{R_2} \\ DA(R_1, j') + DA(R_2, j), \text{ if } h_{R_1} < h_{R_2} \end{cases}$$

$$+ \sum_{j=1}^{\text{abs}\left|h_{R_1} - h_{R_2}\right|} \begin{cases} DA(R_1, j), \text{ if } h_{R_1} > h_{R_2} \\ 2 \cdot DA(R_2, j), \text{ if } h_{R_1} < h_{R_2} \end{cases} \qquad (12)$$

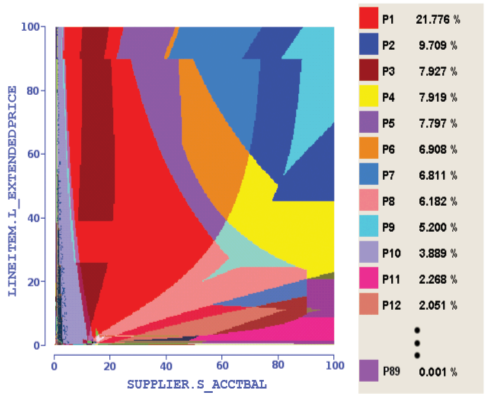
A Model for an R-Tree
[Theodoridis et al.: Cost Models for Join Queries in Spatial Databases]

Analytical models

- Analytical model development is more an art than a craft
- Requires detailed understanding of the system
  - The parameters
  - The effects
- Requires extensive validation
  - Results always questionable
- Often end up **very** complicated to deal with edge cases

Analytical models are often complex

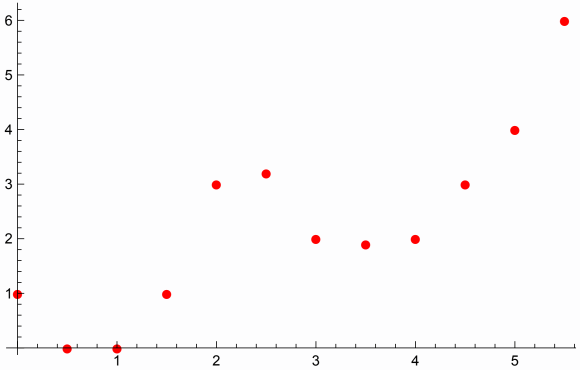
Example: Database plan selection



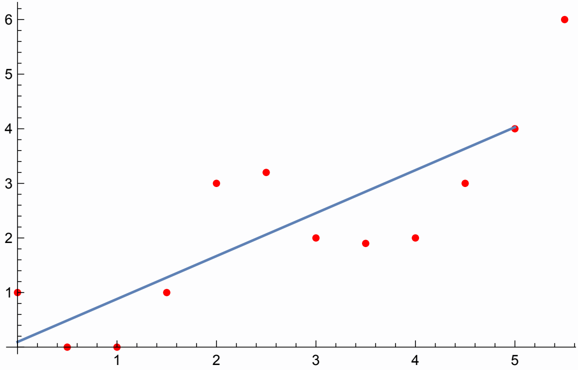
Okay, so how do we get analytical models?

Model Fitting

Turning empirical models into analytical ones...



Turning empirical models into analytical ones...



- Through some form of regression

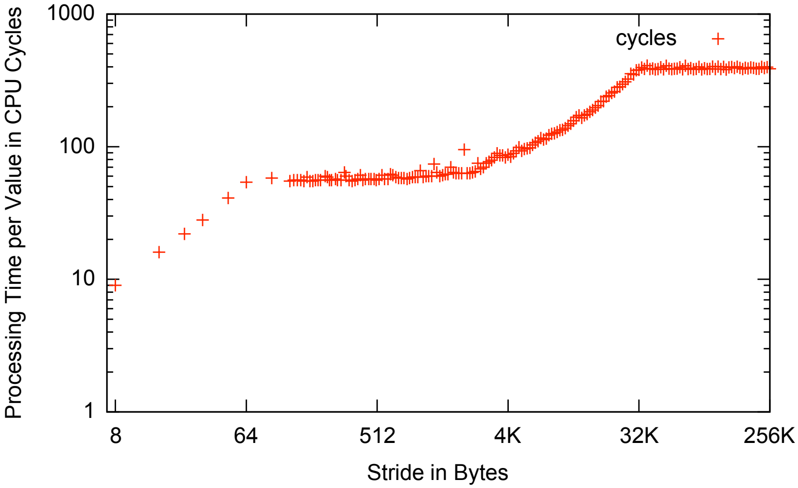
How is this different from numerical modeling?

Admittedly, the line is blurry!

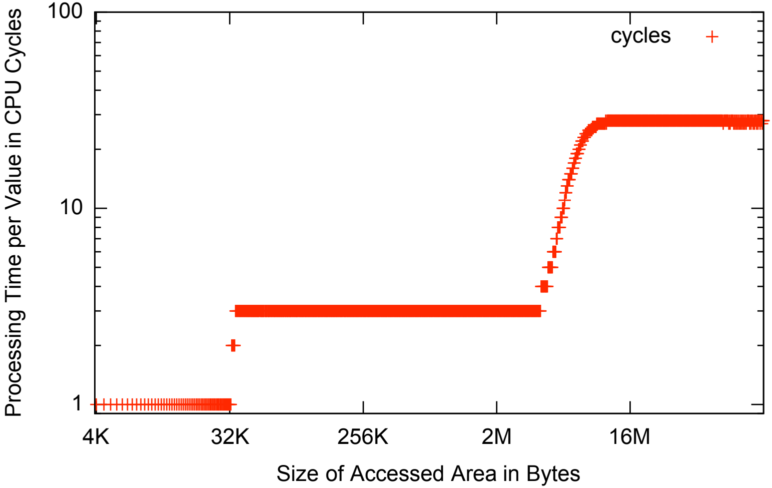
I have decided that interpolation is numerical while regression is analytical

But some things really cannot be done using numerical modeling?

How do you model that. . .



. . . or that?



cycles
100
10
1
4K
32K
256K
2M
16M
Size of Accessed Area in Bytes
Processing Time per Value in CPU Cycles

. . . or that?

For completeness, here is the code

```
extern int* input;
extern size_t N;    // some large constant
extern size_t size; // the parameter of our experiment
int sum = 0;
for(size_t i = 0; i < N; i ++) {
  sum += input[i % size]; // in reality you would use a
                           // bitmask rather than modulo which is expensive
}
```

We need to apply AI

We need to apply AI

Actual Intelligence

We need to apply AI

Actual Intelligence (and some simplifying assumptions)

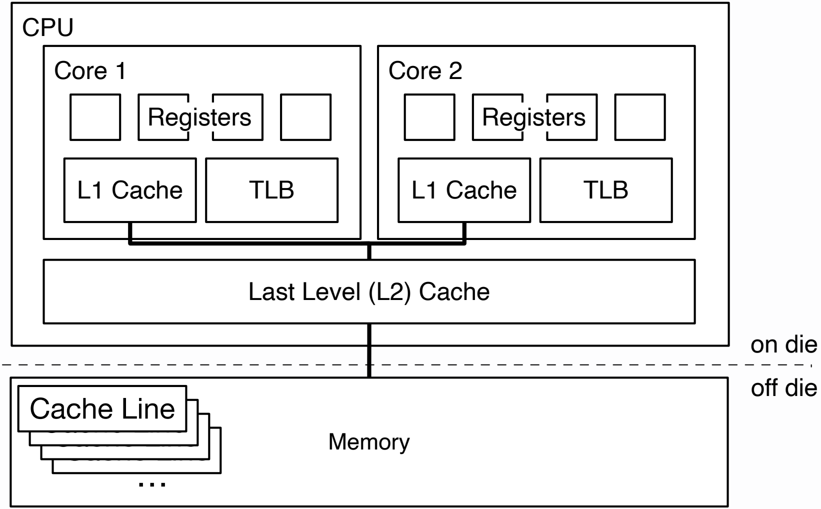
Analytical model ingredients

- A *Characteristic Equation* (with parameters) – An equation that describes the behavior of the target metric of your experiment or system in dependence of a varied parameter
  - In our examples: *stride* and *data size*
- Values for system parameters
  - In our examples: *access latency*, *access granularity (block size)* and *capacity* of the caches

Analytical modeling example: memory access

As seen in [Manegold et al., Generic database cost models for hierarchical memory systems]

What do we know about the system we are trying to model



System parameters

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

A characteristic, non-linear equation

T_{Mem} average time for a memory access

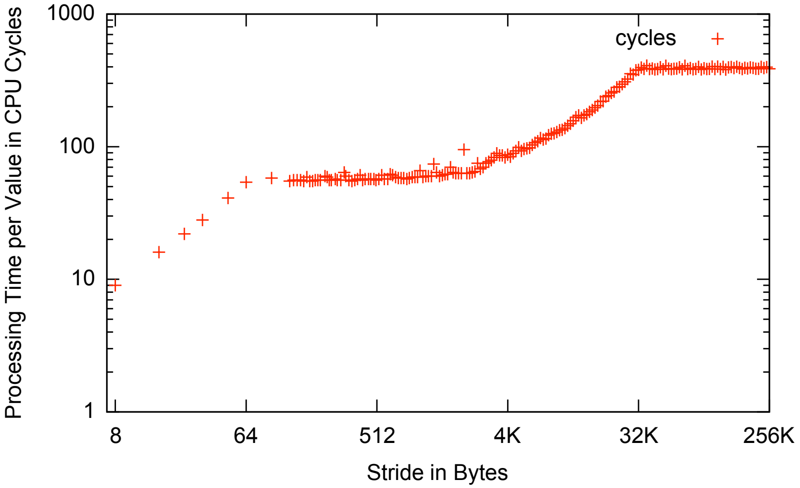
$$
\begin{aligned}
s =& \quad stride \\
T_{Mem} =& \quad l_3 \cdot min\left(1, \frac{s}{B_3}\right) + l_2 \cdot min\left(1, \frac{s}{B_2}\right) \\
& + l_1 \cdot min\left(1, \frac{s}{B_1}\right) + l_0 \cdot min\left(1, \frac{s}{B_0}\right)
\end{aligned}
$$

A characteristic, non-linear equation

T_{Mem} average time for a memory access

$$T_{Mem} = \begin{cases} \mathsf{l0} & \mathsf{size} < \mathsf{C1} \\ \mathsf{l0} + \mathsf{l1} & \mathsf{size} < \mathsf{C2} \\ \mathsf{l0} + \mathsf{l1} + \mathsf{l3} & \mathsf{size} < \mathsf{C3} \\ \mathsf{l0} + \mathsf{l1} + \mathsf{l2} + \mathsf{l3} & \mathsf{Otherwise} \end{cases}$$

Fitting the characteristic equation



Demo Time!

Demo Time!

`https://www.wolframcloud.com/obj/pirk0/Published/CPUModel.nb`

System parameters determined through fitting characteristic equation

Variable	Value
B_0:	1 word (64 bit)
l_0:	1 cycle
C_0:	1 word
B_1:	8 words
l_1	3 cycles
C_1:	4096 words
B_2:	8 words
l_2	55 cycles
C_2:	786432 words
B_3:	512 words
l_3:	1 cycle
C_3:	131072 words

A note

- Some of these can be read from documentation
- However, self tuning systems
    - require less work/expertise
    - are more resilient
    - scale forward (i.e., work on future architectures)
    - and are sometimes more accurate...

Predicting Memory Access Costs

Modeling Memory Access

Let's model this

```
extern int* input1; // uniform random data
extern int* input2; // random data
int sum = 0;
for(size_t i = 0; i < inputSize; i++) {
  sum += input2[input1[i]];
}
```

Parameters

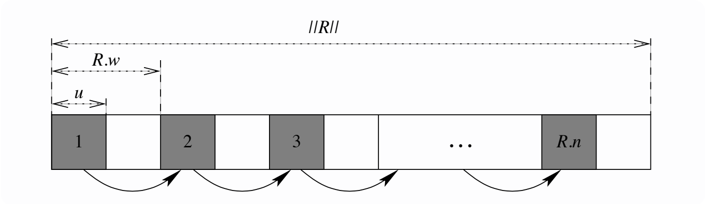
Memory Regions

- it's length ($R.n$), i.e., the number of stored tuples and
- it's width ($R.w$), the size of a tuple in processor words (we will assume a processor with 64bit words).
- The size of the region ($\|R\|$) is defined as the product of length and width.

Access Patterns

- u the number of words read in each access

Modeling sequential access



Estimating the number of cache misses – not examinable

If $R.w - u < B$

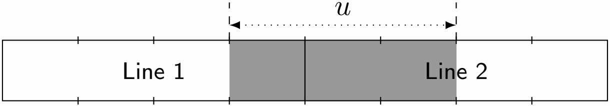
$$M_i^s\left(s\_trav\right) = \frac{R.w \cdot R.n}{B_i}$$

If $R.w - u > B$

$$M_i^s\left(s\_trav\right) = R.n \cdot \left\lceil \frac{u}{B_i} \right\rceil$$

Estimating the number of cache misses – not examinable

Extra cache misses due to misalignment

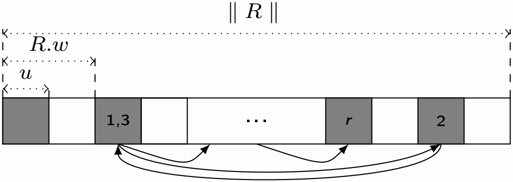


Estimating the number of cache misses – not examinable

If $R.w - u > B$

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

Modeling random access (with repetitive access to elements)



- Additional parameter r, number of accesses

Modeling complex patterns

$\mathcal{P}_1 \oplus \mathcal{P}_2$ the sequential execution of the access patterns $\mathcal{P}_1$ and $\mathcal{P}_2$

$\mathcal{P}_1 \odot \mathcal{P}_2$ the concurrent execution of access patterns.

Example

Code

```
extern int* input1; // uniform random data, 1024 value
extern int* input2; // random data, 64 values
int sum = 0;
for(size_t i = 0; i < inputSize; i++) {
  sum += input2[input1[i]];
}
```

Access pattern description

$$s\_trav(R.w = 1, u = 1, R.n = 1024)$$
$$\odot rr\_acc(R.w = 1, u = 1, R.n = 64, r = 1024)$$

Example

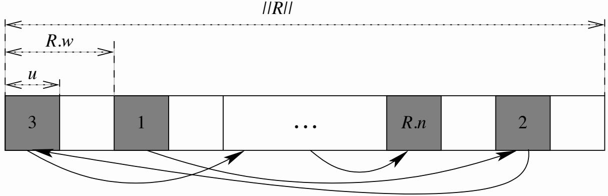
Let's model this

```
extern struct{int a; int b; int c;}* input1; // uniform random data, 1024 value
extern int* input2; // random data, 64 values
int sum = 0;
for(size_t i = 0; i < inputSize; i++) {
  sum += input2[input1[i].a];
}
```

Access pattern description

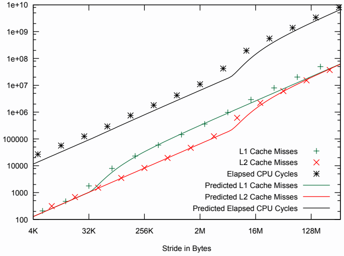
$$s\_trav(R.w = 3, u = 1, R.n = 1024)$$
$$\odot rr\_acc(R.w = 1, u = 1, R.n = 64, r = 1024)$$

Modeling random access without repetitive access



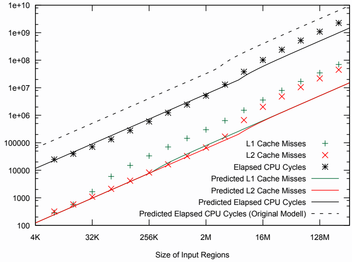
Results

Hash Join Build



$$s\_trav() \odot r\_trav()$$

Hash Join Probe



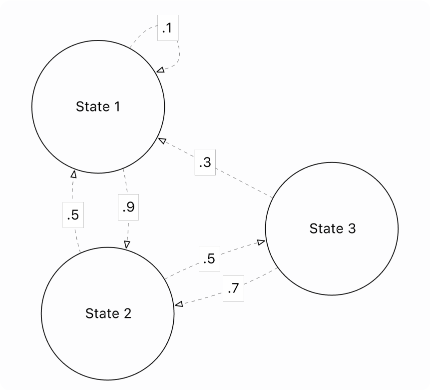
$$\oplus s\_trav() \odot rr\_acc() \odot s\_trav()$$

Modeling stateful systems

Modeling dynamic effects using stochastic methods

- Some effects/components have dynamic state
- State can influence behavior and performance
- Analytical models are, by definition, stateless
- Stochastical models/processes can form the bridge between the two
    - Many exist: random walks, gaussian processes, levy-processes...
    - and most importantly: **Markov Processes/Chains**

(Discrete) Markov chains



- Basically a finite-state machine with transition probabilities
- They have "the Markov property": the next state is only dependent on the previous state and a random variable

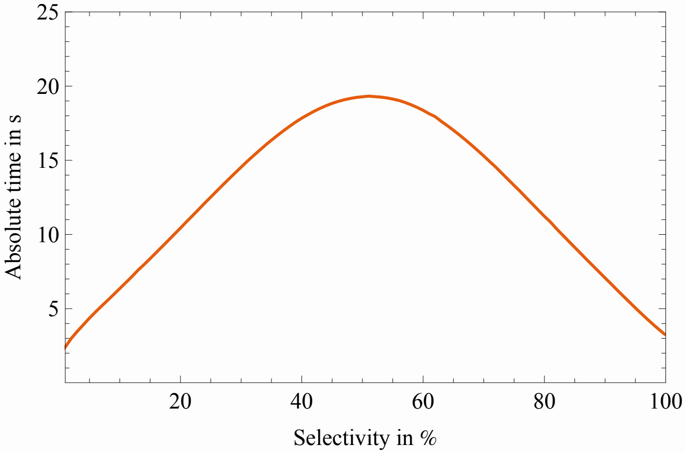
Example: Modeling code

Modeling code

A simple loop

```
extern int* input; // uniform random ints between 0 and 100
int sum = 0;
for(size_t i = 0; i < inputSize; i++) {
  if(input[i] > s) {
    sum += input[i];
  }
}
```

Modeling code



Example: Modeling branch prediction

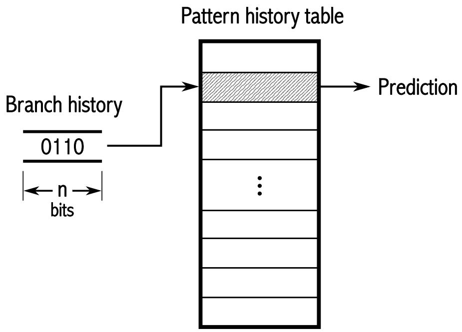
Modeling code

A simple loop

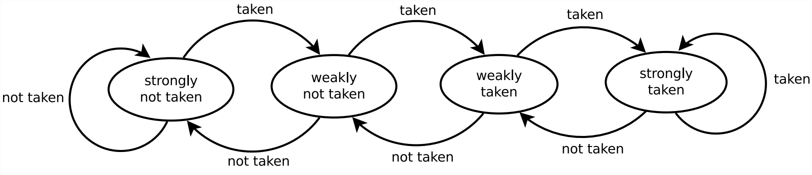
```
extern int* input; // uniform random data
int sum = 0;
for(size_t i = 0; i < inputSize; i++) {
  if(input[i] > 20) {
    sum += input[i];
  }
}
```

What is the branch misprediction rate?

Branch predictors

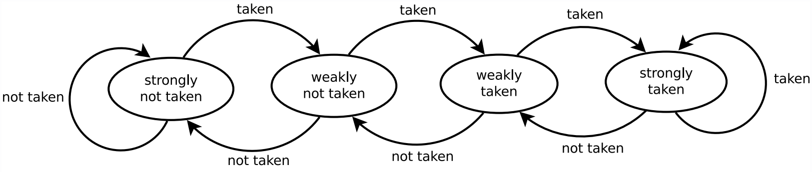


Let's think about this in Markov terms



- Implemented in a *saturated counter*

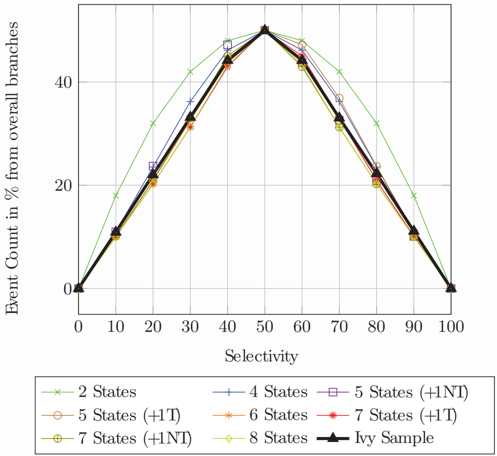
The solution



- Probability of the branch predictor predicting taken:
  - The probability of it being in one of the states on the right
  - We can calculate the probability of it being in any state as the **stationary distribution**
- Branch misprediction rate:
  - $(P(pred\_taken) * P(act\_not\_taken)) + (P(pred\_not\_taken) * P(act\_taken))$

Validation

Comparing stationary distribution with performance counter



Performance modeling recipe

Modeling the entire system

- Is usually infeasible due to scale and noise
- We need to apply modeling with care
- Step 1: identify parts of the code that matter for performance using a profiler
  - Hot code sections (vtune calls this "bottlenecks")
- Step 2: re-create their relevant behavior in a controlled environment
- Step 3: Model
- Step 4: Validate
- We will practise this in the next interactive session

Provide feedback, please!