高级计算机系统结构

沈海华
shenhh@ucas.ac.cn

第九讲 Performance Engineering

- What is performance engineering?
  - Set of activities, practices and tools to ensure a system is well-designed, well-implemented and meets its non-functional requirements
- If you don't measure it, it doesn't work
- Even if it works it needs to be efficient
  - Performance → Efficiency

Performance Engineering Steps

- Measurement
  - Identify critical metrics of interest (e.g., CPI, QPS, IPS/W)
  - Select and setup monitoring and profiling tools to evaluate them
  - Measure their impact to performance
- Analysis
  - Interpret the results to get some meaningful insight on improvements
  - Often includes statistical analysis
- Improvements
  - Identify optimizations to tackle performance/efficiency issues
- Repeat
  - **Iterative process**: repeat steps 1 – 3

Metrics

- Performance
  - IPC (or CPI): instructions per cycle (or cycles per instruction)
  - IPS: instructions committed per second
  - QPS: queries per second, higher level metric
  - Execution time: more robust/universal metric
  - AMAT: average memory access latency
  - Fairness, priorities, …
- Power
  - Watts
  - Instructions/W, mem access/W, QPS/W, QPS/(W*$), etc.

Performance Metrics

- *Latency or Execution Time or Response Time*
  - Wall-clock time to complete a task
- *Bandwidth or Throughput or Execution Rate*
  - Number of tasks completed per unit of time
  - Metric is independent of exact number of tasks executed

Plane	DC to Paris	Speed	Passengers	Throughput (pmph)
Boeing 747	6.5 hours	610 mph	470	286,700
BAD/Sud Concorde	3 hours	1350 mph	132	178,200

- *Quality of Service*
  - Variability in latency or throughput delivered

Performance Metrics (2)

- *App-specific metrics:*
  - Queries per second (QPS)
  - Request round-trip latency
- *Other performance-related metrics:*
  - Fairness (across processes, threads, requests)
  - Priorities (critical jobs should execute before secondary apps)
  - Progress (no starvation even for low-priority jobs)

Reminder: Program Execution Time

- Latency metric: program execution time in seconds

$$CPUtime = \frac{Seconds}{Program} = \frac{Cycles}{Program} \cdot \frac{Seconds}{Cycle}$$

$$= \frac{Instructions}{Program} \cdot \frac{Cycles}{Instruction} \cdot \frac{Seconds}{Cycle}$$

$$= IC \cdot CPI \cdot CCT$$

- Your system architecture can affect all of them
  - CPI: memory latency, IO latency, …
  - CCT: cache org., power budget, …
  - IC: OS overhead, compiler choice …

Independent?

Power Metrics

- Watts: simply the power consumption of a system (can be measured at different granularities)
- Power on its own is rarely of interest
- We are mostly interested in power-performance, or power-cost metric
- IPS/W or QPS/W: performance metric for a specific power budget
- IPS/(W*$) or QPS/(W*$): performance for a specific power & cost budget

Performance Engineering Steps

- Measurement
  - ~~Identify critical metrics of interest (e.g., CPI, QPS, IPS/W)~~
  - Select and setup monitoring and profiling tools to evaluate them
  - Measure their impact to performance
- Analysis
  - Interpret the results to get some meaningful insight on improvements
  - Often includes statistical analysis
- Improvements
  - Identify optimizations to tackle performance/efficiency issues
- Repeat
  - **Iterative process**: repeat steps 1 – 3

Evaluation choices

- Real experiments
  - Use one/more machines to evaluate a program or system optimization
- Experiments using analytical models
  - Use an analytical model that captures the critical features of the system or program
  - Requires validation of the model
- Experiments using a simulator
  - Use a tool that closely approximates the execution on a real machine
  - Ideally: small deviation between real and simulated systems

How to measure in a real system?

- Obvious: Time!!
  - What's the performance of the workload (whatever performance mean for that app)
- Breakdown of execution time:
  - Based on the code (profiling which functions take the most time)
  - Based on events (using performance counters)
- Performance counters: finer grain info on where time goes
  - Hardware exposes very rich info on microarchitectural events → performance/efficiency insight
- How to gain visibility to performance counters? → Profiling/ Monitoring tools

Performance Counters

- Depend on the underlying architecture
- Don't trust all of them ☺
- Include:

INST_RETIRED: instructions retired

CPU_CLK_UNHALTED: clock cycles when thread is not halted

LLC_MISSES: number of LLC misses

LLC_REFS: number of LLC accesses

BR_MISS_PRED_RETIRED: number of mispredicted branches retired

MEM_INST_RETIRED: number of memory instructions retired

L2_RQSTS/L2_DATA_RQSTS: L2 instrs fetch hits/all L2 requests

L1D_PREFETCH: L1 D-cache hardware prefetch misses

IO_TRANSACTIONS: number of I/O transactions

…

Profiling Tools

- Perf
- PAPI
- Oprofile
- Libpfm
- Many more…

Perf

- Perf is a simple interface to collect and analyze performance counters (shipped with Linux)
- Per-CPU, per-thread and per-workload counters

perf stat –v ls →

```
Performance counter stats for 'ls':

        6.844663 task-clock                #    0.371 CPUs utilized
             263 context-switches          #    0.038 M/sec
               0 CPU-migrations            #    0.000 M/sec
             281 page-faults               #    0.041 M/sec
       3,574,541 cycles                    #    0.522 GHz
               0 stalled-cycles-frontend   #    0.00% frontend cycles idle
               0 stalled-cycles-backend    #    0.00% backend  cycles idle [37.43%]
       4,628,962 instructions              #    1.29  insns per cycle        [37.43%]
         819,708 branches                  #  119.759 M/sec                  [37.43%]
      0.018427044 seconds time elapsed
```

- Provides the simple counters easily, can be tedious for more specialized

PAPI

- More detailed interface to get access to performance counters
- Based on *libpfm4*
- Provides a high-level (simple measurements) and low-level interface (fully programmable, visibility to the supported counters of a specific ISA)
- Event-driven: can start, stop and read specific events → counters
- Can also correlate performance issues with counter values

http://icl.cs.utk.edu/papi/software/index.html

Using Counters in Perf Engineering

- One use case: based on perf counters from one system can we predict the performance in other systems?
- Why? → insight on desired arch features, benefit from switching, needed app changes → faster and cheaper than exhaustive profiling

- Step 1: collect perf counters from many apps offline → determine the important to performance
- Step 2: run target app in one architecture + periodically collect perf counters (capture execution phases)
- Step 3: input profiling data to performance prediction scheme → determine impact from new architecture

Performance Models

- Simplest model??

$$CPUtime = IC \cdot CPI \cdot CCT$$

- Extend:
  - Memory latency (hit/miss rates, miss penalties, access latencies)
  - Cache latency (hit/miss rates, etc.)
  - Branch mispredictions (penalties, etc.)
  - …
  - Very detailed models that capture many aspects of the architecture
  - Where does this stop? Is more detailed always better?

Power Models

- Tools that are based on power models (e.g., CACTI) → provide estimations of power consumption for architecture components, or full system power consumption

- Many papers on full system or detailed microarchitectural models to estimate power consumption
  - Wide spectrum of complexities
  - Wide spectrum of accuracies

- What is the catch with all these models?

Cacti

- **Cacti** (Wilton et al, 1996, Thoziyoor et al, 2007) 是 DEC公司开发的专门用来评估 Cache 的延时、功耗，面积的工具。只要使用者输入 Cache 的大小、相联度、块大小等基本的结构信息和工艺参数，**Cacti** 就可以得出该 Cache 的动态功耗和静态功耗、所占面积、访问 Cache的延迟等结构设计者关心的物理信息。

Analytical Models

- Advantages?
  - Decouples drawing conclusions about a system or application from having to do real experiments
  - Easier/Faster to evaluate many "what-if" scenarios
  - What else?
- Disadvantages?
  - Models have limitations: you cannot capture every aspect of a system in a model and still keep it concise and simple to use
  - Models **require** validation: otherwise garbage in-garbage out
  - Models can be tied to the system on which they were trained – training vs. testing input considerations

Simulation Techniques

- Many options:
  - Full system vs. user-level
  - Functional vs. timing
  - Emulation vs. instrumentation
  - Trace-driven vs execution-driven
  - Event-driven vs cycle-driven
- What does this mean?

Full System vs User Level

- Full system simulators:
  - Include privileged modes and emulate peripheral devices to support an OS
  - Can faithfully run multithreaded and multiprogrammed workloads and I/O-bound (storage + network) apps
  - Examples: gem5, Flexus, MARSS
  - Disadvantages: slower, harder to setup, harder to develop
- User-level simulators:
  - Only capture the user-level part of the application
  - Much easier to develop → no need for device timing models, large disk images or booting an OS
  - Much easier to scale → no OS supports 1000 cores currently
  - Examples: Graphite, Snipper, zsim, CMPSim, HORNET
  - Disadvantages: cannot faithfully capture complex workloads, apps that spend a lot of time in the OS, apps where threads>cores and interactive workloads

Functional vs. Timing

- Functional simulators:
  - Only simulate functionality
  - No timing aspects → no performance output
  - Good for ISA evaluation
  - Not appropriate for performance optimizations
  - Examples: Simics
- Timing simulators:
  - Also simulates timing of execution → performance output
  - Appropriate for a larger spectrum of studies
  - More complex, timing can introduce inaccuracies
  - Examples: Graphite, Snipper, zsim, gem5, Flexus, …

Emulation vs. Instrumentation

- Emulation-based simulators:
  - Decodes instructions and invokes functional and timing models
  - Used when the simulated ISA is different from the host's
  - Examples: gem5, Flexus, MARSS
  - Speed: 200-500KIPS
- Instrumentation-based simulators:
  - Adds instrumentation calls to the simulated binary
  - Calls the timing models before each basic block/mem operation
  - Leverages the host's ISA for functional simulation (Dynamic Binary Translation – DBT)
  - Faster than emulation-based (10-100MIPS)
  - Examples: CMPSim, Graphite, zsim

Trace/Event- vs. Execution-driven

- Trace/Event-driven simulators:
  - Based on pre-recorded streams of instructions for fixed input
  - Faster
  - Less flexible, depends on completeness, level of detail, distortion of traces
  - Examples: CMPSim, …
- Execution-driven simulators:
  - Allow dynamic change of instructions to be executed based on different inputs
  - More flexible, accurate, extensible
  - More complex to design
  - Examples: Graphite, gem5, Flexus, zsim, …

The real system has randomness… does your simulator?

- Same starting state, no interrupts, deterministic event ordering? You have a problem
  - e.g., your benchmark executes +/-10% of instructions in the real system depending on e.g., starting machine state
  - Your baseline design happens to hit the -10%
  - Your 5% IPC-improved design happens to hit the +10%...
  - Often worse in parallel benchmarks
- Add some randomness, even artificially (+/-2 cycles on memory accesses) [Alameldeen and Wood, IEEE Micro 06]
  - May not model the real randomness, but often good enough

Simulation

- Advantages:
  - Easier to evaluate "what-if" scenarios than in a real system (especially hardware changes)
  - Cheaper than buying the real systems
  - What else?
- Disadvantages:
  - Simulators are not perfectly accurate → Need **detailed validation**
  - Typically much slower than a real system (100KIPS-100MIPS)
  - Many tradeoffs in simulation design, e.g., full system vs. user-level → pros and cons of each

Performance Engineering Steps

- ~~Measurement~~
  - ~~Identify critical metrics of interest (e.g., CPI, QPS, IPS/W)~~
  - ~~Select and setup monitoring and profiling tools to evaluate them~~
  - ~~Measure their impact to performance~~
- Analysis
  - Interpret the results to get some meaningful insight on improvements
  - Often includes statistical analysis
- Improvements
  - Identify optimizations to tackle performance/efficiency issues
- Repeat
  - **Iterative process**: repeat steps 1 – 3

Stats for Architects

- Basic principles
- How to use in real systems to interpret your results

Basic Stats

- Distribution: the set of outcomes of a system with the probability of each

$$\sum_{x \in R} f(x) = 1 \qquad \int_{-\infty}^{\infty} f(x)dx = 1$$

- f(x): probability distribution function (PDF)

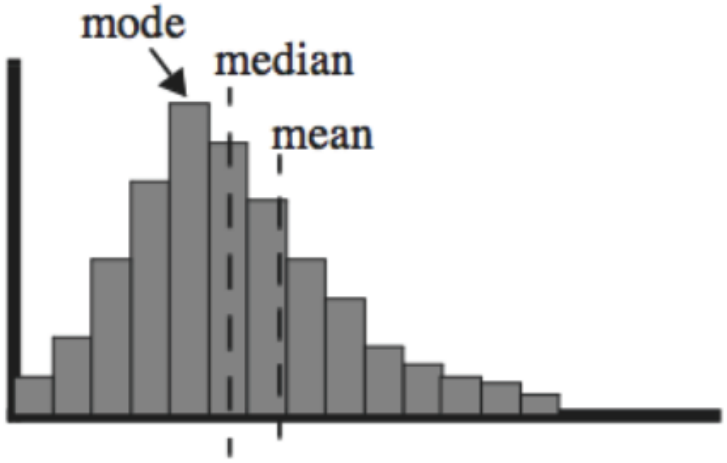
$$\text{Prob}\{a \le x \le b\} = \int_{a}^{b} f(x)dx$$

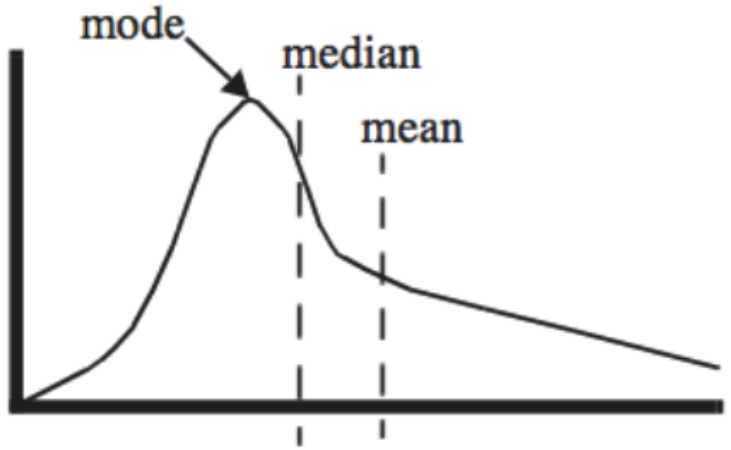
- E[x]: expected value

$$E[X] \equiv \sum_{x \in R} xf(x) \qquad E[X] \equiv \int_{-\infty}^{\infty} xf(x)dx$$

“Averages”

- Mean: average of values in the frequency distribution
- Median: the value in the midpoint of the frequency distribution
- Mode: most frequently occurring data in a dataset





“Averages”

- **Mean(平均数):**

数学定义： 一组数据的总和除以这组数据个数所得到

应用：用于线性正态分布

- **Geometry Mean(几何平均数):**

数学定义：n个变量值连乘积的n次方根

应用：用于对数正态分布

- **Median(中位数)：**

数学定义：将一组数据按大小顺序排列，处在最中间位置的一个数，左右各有50%数据

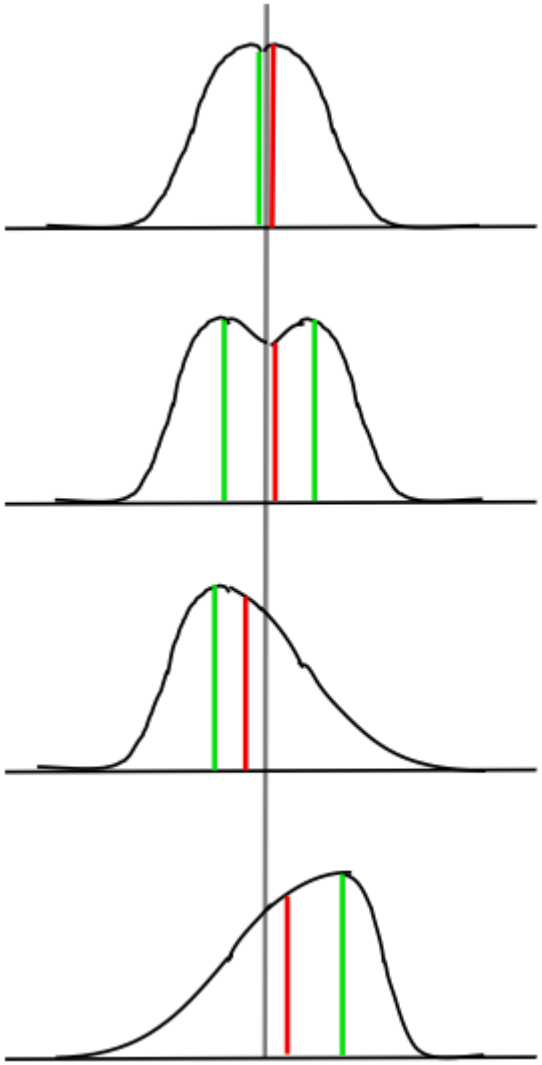
应用：Median不受数据形态分布的影响，不需要是正态分布，可以减少异常值对整个数据的影响，是用的比较多的一个统计值。

- **Mode(众数）**

数学定义：在一组数据中出现次数最多的数叫做这组数据的众数

应用：数据分布最高峰所在的位置

不同分布下的“Averages”示意图



MEAN = MEDIAN = MODE

MEAN = MEDIAN

MEAN > MEDIAN > MODE

MEAN < MEDIAN < MODE

CDFs

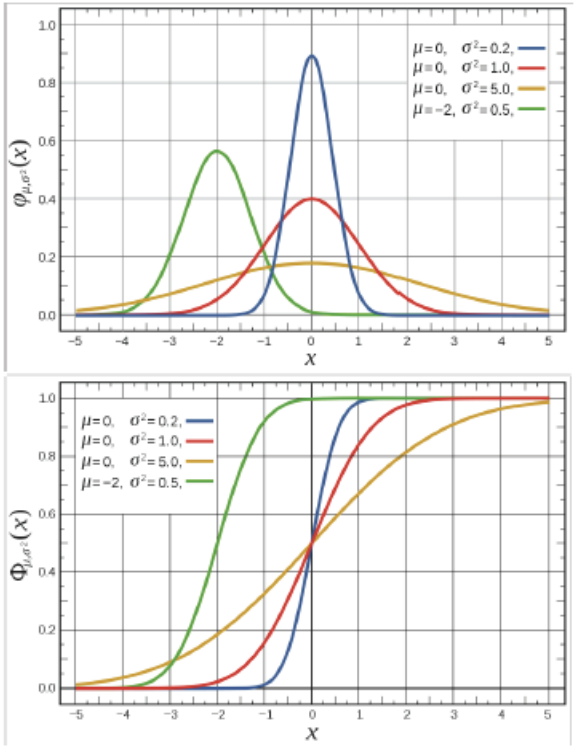
- Cumulative Distribution Function (CDF): Cumulative sum of probabilities up to a certain point, for a given PDF

$$c(a) \equiv Prob[x \le a] \equiv \int_{-\infty}^{a} f(x)dx \qquad c(+\infty) \equiv 1 \qquad c(-\infty) \equiv 0 \qquad 0 \le c(x) \le 1$$

$$Prob[a \le x \le b] \equiv \int_{a}^{b} f(x)dx = c(b) - c(a)$$

$$\downarrow$$

$$Prob[x > a] = 1 - c(a)$$



Sampling

- Empirical distribution: A collection of samples
- Sample mean: $\bar{x} \equiv \sum \frac{1}{n} x_i$
- Variance: $Var[\bar{x}] = n \cdot \left(\frac{1}{n}\right)^2 Var[X] = \frac{Var[X]}{n}$

  or: $Variance[X] = E[X^2] - \mu^2 = E[X^2] - (E[X])^2 = \frac{1}{n}\sum_i x_i^2 - \left(\frac{1}{n}\sum_i x_i\right)^2$
- Variance decreases linearly with n
- Std decreases with the sqrt of n
- More samples → higher accuracy of sample mean

方差和标准差

- 方差(Variance)是实际值与期望值之差的平方平均数
- 标准差(Standard deviation)是方差的平方根
- 方差和标准差是测算离散趋势最重要、最常用的指标，可以有效放大样本波动。
- 样本方差或样本标准差越大，样本数据的波动就越大。

为什么同时需要方差和标准差？

- 标准差和均值的量纲（单位）是一致的，在描述一个波动范围时标准差比方差更方便。
  举例：一个班男生的平均身高是170cm,标准差是10cm,那么方差就是100cm。可以进行的比较简便的描述是本班男生身高分布是170±10cm，方差就无法做到这点。
- 思考：样本量与方差、标准差的关系？

Central Limit Theorem

- **Theorem:** If $\bar{x}$ is the sample mean of a random variable, X, with unknown distribution with mean μ and variance σ^2 then the distribution of $\bar{x}$ approaches a distribution that is N(μ, σ/√n) as n becomes large

- Translation: As long as we have enough iid samples, we can ignore the distribution of X and the distribution of $\bar{x}$ will be close to normal

中心极限定理

- 设从均值为μ、方差为σ^2（有限）的任意一个总体中抽取样本量为n的样本，当n充分大时，样本均值的抽样分布近似服从均值为μ、方差为σ^2/n的正态分布。

- 中心极限定理（central limit theorem）是概率论中讨论随机变量序列部分和分布渐近于正态分布的一类定理。这组定理是数理统计学和误差分析的理论基础，指出了大量随机变量累积分布函数逐点收敛到正态分布的累积分布函数的条件。

Confidence Intervals

- 置信区间又称估计区间，是用来估计参数的取值范围的。
- 置信区间展现的是这个参数的真实值有一定概率落在测量结果的周围的程度。置信区间给出的是被测量参数的测量值的可信程度，而前面所要求的“一定概率”则被称为置信度（或者置信水平）。

- Goal: How accurate is our sample mean?

- Assuming $\bar{x}$ is normally distributed (based on C.L.T.), to be 90% sure our sample mean is within k units from the real mean, we need:

$$Prob[\mu - k < \bar{x} < \mu + k] = 0.9$$

Confidence Intervals Practically

- We take N samples from a population (e.g., run a benchmark N times) and want to approximate a *parameter* about the whole population (e.g., the *true mean time* of all runs) with those samples (e.g., the *sample mean time* of the N runs)
  - Can we compute the actual error between both?
- An X% confidence interval is the range of values that is X% likely to contain the true value across the whole population
  - Multiple ways to estimate
  - Typically, assume Gaussian distribution, compute sample mean and std, and use inverse CDF to compute (symmetric) range
  - In most real-world systems, increasing N makes interval smaller
    - Infinite-variance distributions exist, in paper…

计算置信区间

(1)知道样本均值(M)和标准差(ST)时：

置信区间下限：a=M - n*ST;　置信区间上限：a=M + n*ST;

当求取90% 置信区间时 n=1.645

当求取95% 置信区间时 n=1.96

当求取99% 置信区间时 n=2.576

（2）通过利用蒙特卡洛（Monte Carlo）方法获得估计值分布时：

先对所有估计值样本进行排序，置信区间下限：a为排序后第lower%百分位值;　置信区间上限：b为排序后第upper%百分位值.

当求取90% 置信区间时 lower=5　upper=95;

当求取95% 置信区间时 lower=2.5　upper=97.5

当求取99% 置信区间时 lower=0.5　upper=99.5

当样本足够大时，（1）和（2）获取的结果基本相等。

Confidence Intervals Practically

- What does that mean for you, when reporting results??
- Do not simply report an average across runs!
- What is the 90-95th confidence interval for that number?
  - If it's ±μ there's a problem
- Over which sample (N (?) runs of a benchmark) were these numbers collected?

Means

- Arithmetic: $$amean = \frac{1}{N}\sum_{i=1}^{N} x_i$$
- Harmonic: $$hmean = N \Big/ \sum_{i=1}^{N} \frac{1}{x_i}$$
- Geometric: $$gmean = \left(\prod_{i=1}^{N} x_i\right)^{1/N}$$
- For positive differing quantities, amean > gmean > hmean
- Rules of thumb: amean for absolutes, hmean for rates (speeds), gmean for ratios
- In practice, use first principles as much as possible to derive aggregate metrics
  - Weighting or other means can be useful
  - And be honest… (Q: most/least used means in papers?)

Statistically Significant Experiments

- Most fields: "Our experiment shows the vaccine is effective in 85%(+/-2%) of subjects…"
- Computer architects (often): "We ran each experiment once, here are the bars"
  - "What are the confidence intervals?" Common responses:
    - Madness is doing the same thing twice and expecting a different result!
    - Simulations take a long time! Better to simulate for 5x longer…
    - Confidence what?
- The Java tribe: "We ran each benchmark 10 times and report the best execution times"
  - The other 9 are to warm the JVM up…

Observational Error

- Most experiments (and definitely computers) are subject to variability
- Two types of observational error:
  - Systematic: Always occurs in the same way
    - Performance counter bugs, instrumentation overhead, room temperature & turbo, simulator bugs…
  - Random: Due to natural system variability and non-determinism
    - Initial machine state, VM mappings, ASLR, interrupts, benchmarks that use randomized algorithms, …
    - In parallel systems, *amplified* by lock acquisition order, barrier synchronization, etc.
- Avoiding systematic error:
  - Detect them… good luck
  - Either redesign experiment or estimate impact and adjust measurement
- Reducing random error: Make your confidence intervals small

Summarizing Performance

- Ideal world:
  - Ideal chip manufacturer: Compared to our old chip, our new one improves performance of benchmark 1 by 10%, benchmark 2 by 50%, benchmark 3 by -10%, etc.
  - Ideal customer 1: I mostly run (something similar to) benchmark 2, let's upgrade
  - Ideal customer 2: I'm half ~1, half ~3, not for me…
- Real word:
  - Customer: I don't know what I run, just give me a number!
  - Chip manufacturer: OK, here's the mean improvement…

Sampling and cold-start effects

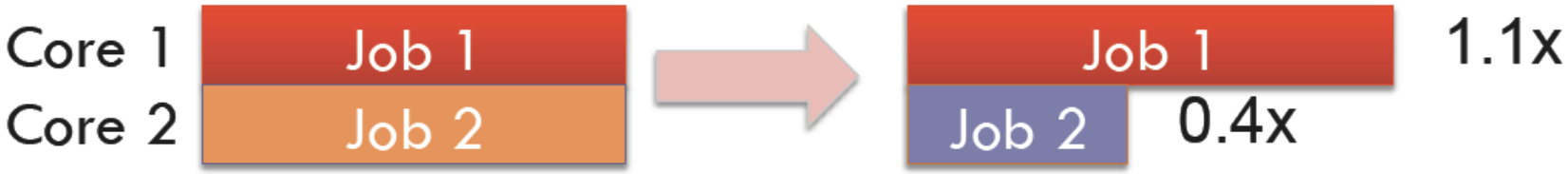
- Often, can only run short benchmarks (~100M instrs)
  - But want to estimate performance of much longer runs!
- Problem 1: Choose statistically significant portions of the program. Options:
  - Analyze the workload beforehand, pick samples [SimPoints, Sherwood et al, ISCA/SIGMETRICS 03]
  - Periodic or randomized sampling, and treat it as a sampling problem [SMARTS, Wunderlich et al, ISCA 03]
- Problem 2: Microarchitectural state (caches, predictors, etc) not warmed up!
  - Functional-only or detailed (timing) warming

Scalability

- Speedup(N) = Time on 1 processor/Time on N processors
  - What's the best we can do? Linear?
  - Often sublinear…
- Strong scaling: Speedup on 1…N processors with fixed *total* problem size
- Weak scaling: Speedup on 1…N processors with fixed *per-processor* problem size

Multi-programmed setups

- Parallel processors execute multiple jobs…
- How to compute performance improvement of this?



- Options (assuming work == instructions):
  - Variable-work methodology: Measure time to finish N instructions
    - Issues?
  - Fixed-work methodology: Measure time to finish N instructions for each program, then average
    - Terminate/keep running/rewind programs as they finish?
    - Issues?

Summary

- Performance engineering is CRITICAL to ensure something works as expected
- Many different ways to evaluate a system/app (real experiments, simulation, analytical models)
- Many different tools to profile the way the system/app behaves (perf. counters → perf, papi, etc.)
- End game: analyze and interpret the output data correctly → statistically important results, statistically sound analysis, report reasonable metrics

致谢：

本讲内容参考了M.I.T. Daniel Sanchez教授的课程讲义，特此感谢。