Lecture - 4

Measurement

Dr. Soner Onder
CS 4431
Michigan Technological University

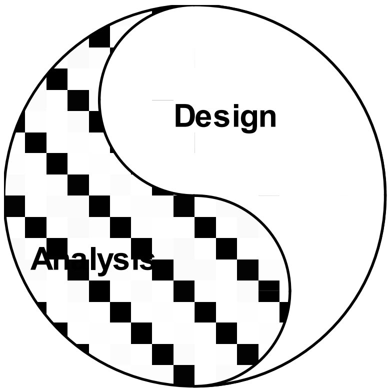
Acknowledgements

- David Patterson
- Dr. Roger Kieckhafer

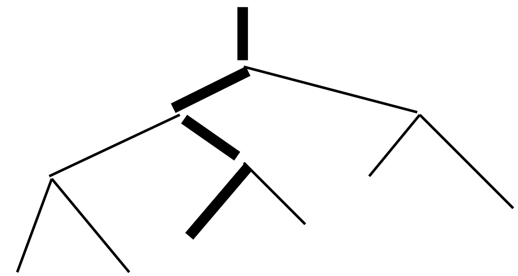
Computer Architecture is Design and Analysis

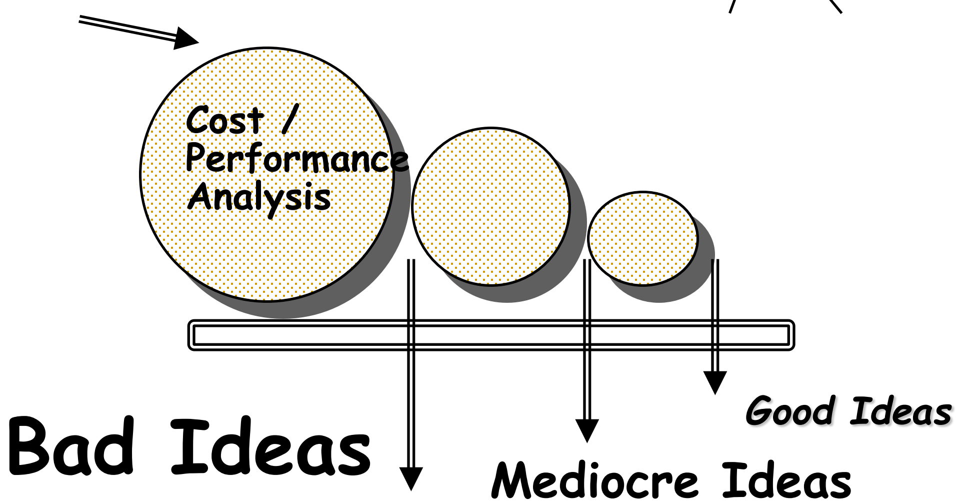
Architecture is an iterative process:

- Searching the space of possible designs
- At all levels of computer systems



Creativity





What Computer Architecture brings to Table

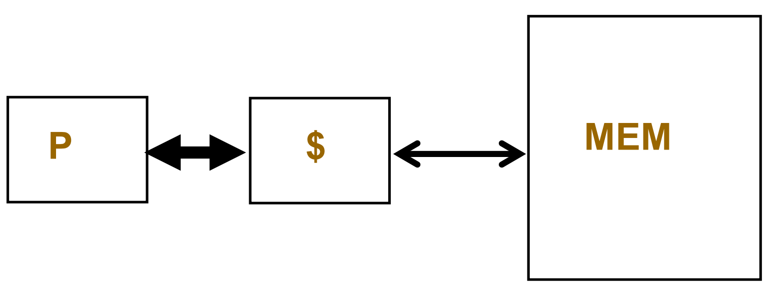
- Other fields often borrow ideas from architecture
- Quantitative Principles of Design
  1. Take Advantage of Parallelism
  2. Principle of Locality
  3. Focus on the Common Case
  4. Amdahl's Law
  5. The Processor Performance Equation
- Careful, quantitative comparisons
  - Define, quantity, and summarize relative performance
  - Define and quantity relative cost
  - Define and quantity dependability
  - Define and quantity power
- Culture of anticipating and exploiting advances in technology
- Culture of well-defined interfaces that are carefully implemented and thoroughly checked

1) Taking Advantage of Parallelism

- Increasing throughput of server computer via multiple processors or multiple disks
- Detailed HW design
  - Carry lookahead adders uses parallelism to speed up computing sums from linear to logarithmic in number of bits per operand
  - Multiple memory banks searched in parallel in set-associative caches
- Pipelining: overlap instruction execution to reduce the total time to complete an instruction sequence.
  - Not every instruction depends on immediate predecessor $\Rightarrow$ executing instructions completely/partially in parallel possible
  - Classic 5-stage pipeline:
    1) Instruction Fetch (Ifetch),
    2) Register Read (Reg),
    3) Execute (ALU),
    4) Data Memory Access (Dmem),
    5) Register Write (Reg)

2) The Principle of Locality

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straight-line code, array access)
- Last 30 years, HW relied on locality for memory perf.



3) Focus on the Common Case

- Common sense guides computer design
  - Since its engineering, common sense is valuable
- In making a design trade-off, favor the frequent case over the infrequent case
  - E.g., Instruction fetch and decode unit used more frequently than multiplier, so optimize it 1st
  - E.g., If database server has 50 disks / processor, storage dependability dominates system dependability, so optimize it 1st
- Frequent case is often simpler and can be done faster than the infrequent case
  - E.g., overflow is rare when adding 2 numbers, so improve performance by optimizing more common case of no overflow
  - May slow down overflow, but overall performance improved by optimizing for the normal case
- What is frequent case and how much performance improved by making case faster => Amdahl's Law

4) Amdahl's Law

$$ExTime_{new} = ExTime_{old} \times \left[(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right]$$

$$Speedup_{overall} = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

Best you could ever hope to do:

$$Speedup_{maximum} = \frac{1}{(1 - Fraction_{enhanced})}$$

Amdahl's Law example

- New CPU 10X faster
- I/O bound server, so 60% time waiting for I/O

$$\text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \text{Fraction}_{\text{enhanced}}\right) + \dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

$$= \frac{1}{\left(1 - 0.4\right) + \dfrac{0.4}{10}} = \frac{1}{0.64} = 1.56$$

- Apparently, its human nature to be attracted by 10X faster, vs. keeping in perspective its just 1.6X faster

5) Processor performance equation

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	Inst Count	CPI	Clock Rate
Program	X		
Compiler	X	(X)	
Inst. Set.	X	X	
Organization	X		X
Technology			X

Definition: Performance

- Performance is in units of things per sec
  - bigger is better
- If we are primarily concerned with response time

$$\text{performance}(x) = \frac{1}{\text{execution\_time}(x)}$$

" X is n times faster than Y" means

$$n = \frac{\text{Performance}(X)}{\text{Performance}(Y)} = \frac{\text{Execution\_time}(Y)}{\text{Execution\_time}(X)}$$

Performance: What to measure

- Usually rely on benchmarks vs. real workloads
- To increase predictability, collections of benchmark applications, called *benchmark suites*, are popular
- SPECCPU: popular desktop benchmark suite
  - CPU only, split between integer and floating point programs
  - SPECint2000 has 12 integer, SPECfp2000 has 14 integer pgms
  - SPECSFS (NFS file server) and SPECWeb (WebServer) added as server benchmarks
- Transaction Processing Council measures server performance and cost-performance for databases
  - TPC-C Complex query for Online Transaction Processing
  - TPC-H models ad hoc decision support
  - TPC-W a transactional web benchmark
  - TPC-App application server and web services benchmark

Relative Performance Metrics

- Given two design options (X and Y)
- Execution Time:
  - T_X ≡ Execution time of a workload run on option X
  - T_Y ≡ Execution time of a workload run on option Y
- Performance:
  - $Perf_x \equiv 1/T_X$
  - $Perf_Y \equiv 1/T_Y$
- Speedup of X over Y ($S_{x/y}$):

$$S_{X/Y} \equiv \frac{Perf_X}{Perf_Y} = \frac{T_Y}{T_X}$$

Relative Performance Metrics

- Percent Improvement in Performance
- "X is n% faster than Y" means:
  - $$n \equiv 100\left[\frac{Perf_X - Perf_Y}{Perf_Y}\right] = 100\left[\frac{Perf_X}{Perf_Y} - \frac{Perf_Y}{Perf_Y}\right]$$
  - $$n = 100\left[S_{X/Y} - 1\right]$$
- Example:
  - Y takes 15 seconds to complete a task,
  - X takes 10 seconds to complete the same task.

Revisiting Amdahl's Law

- Amdahl's Law States:
  - $$T_E = T_0\left(1 - F_E\right) + \left(\frac{T_0\, F_E}{S_E}\right) = T_0\left[\left(1 - F_E\right) + \left(\frac{F_E}{S_E}\right)\right]$$
- Plugging into the definition of Speedup yields:
  - $$S_{E/0} = \frac{T_0}{T_E} = \frac{1}{\left(1 - F_E\right) + \left(\frac{F_E}{S_E}\right)}$$
- Note: If $\mathbf{F_E = 1.0}$, Then:
  - $$T_E = \left(\frac{T_0}{S_E}\right) \Rightarrow S_{E/0} = S_E$$

Revisiting Amdahl's Law

- Example: An enhancement (E) improves the speed of Floating Point (FLP) instructions by a factor of 2.
  - $S_E = 2$
  - $$S_{E/0} = \frac{1}{(1 - F_E) + \left(\frac{F_E}{2}\right)}$$
- Where: F_E = the fraction of FLP instructions in Program
  - e.g. General Pgm: If $F_E = 0.1$, Then $S_{E/0}$ =
  - e.g. Scientific Pgm: If $F_E = 0.9$, Then $S_{E/0}$ =

Execution Time

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

- Execution Time for a CPU
  - T(N)
  - N
  - CPI
  - $\tau = 1 / f$ (clock period = 1 / frequency)
- $T(N)$ =

Cycles Per Instruction

- Calculating CPI:
  - F_k ≡ *Fraction of instructions of type* k, k $\in \{1 \ldots m\}$
  - CPI_k ≡ *CPI for instruction type k*
  - CPI ≡ *Mean CPI for entire program*

$$CPI = \sum_{k=1}^{m} F_k \times CPI_k$$

- Conclusion: Invest Resources where time is Spent!
  - Focus on instruction types for which (F_k x CPI_k) is largest

Example: Calculating CPI

- Typical Instruction Mix
  - **Type (k)** | F_k | CPI_k | F_k*CPI_k | **(% Time)**

Type (k)	F_k	CPI_k	F_k*CPI_k	(% Time)
ALU	50%	1	.5	(33%)
Load	20%	2	.4	(27%)
Store	10%	2	.2	(13%)
Branch	20%	2	.4	(27%)

  - **CPI = 1.5**

Example

- Suppose we have the following measurements:
  - Frequency of FP operations: 25 %
  - Average CPI of FP operations: 4.0
  - Average CPI of other instructions: 1.33
  - Frequency of FPSRQ: 2 %
  - CPI of FPSQR: 20
- Assume two design alternatives: Decrease the CPI of FPSQR to 2, or decrease the average CPI of all FP operations to 2.5. Use processor performance equation to calculate.
  - Observe that the clock speed and instruction count remain identical.
  - Find original CPI first:
    - CPI = 4 x 0.25 + 1.33 x 0.75 = 2.0
    - $CPI_{new\ sqrt} = CPi_{original} – 0.02 \times (CPI_{old\ FPSQRT} – CPI_{of\ new\ FPSQRT})$
      - = 2.0 – 0.02 * (20 -2) = 1.64
    - $CPI_{new\ FP}$ = (0.75 x 1.33) + (0.25 x 2.5) = 1.62
    - Speed-up = $CPI_{original} / CPI_{new\ FP}$
      - = 2.0 / 1.62 = 1.23 = 1.23

Throughput

- Execution Time is relative
  - Depends on number of operations executed
  - Interested how much work was done in that time
- *W(N)* = Mean operations executed per unit time
  - $$W(N) \equiv \frac{N}{T(N)}$$
- Typical Units
  - MIPS = Millions of Instructions per Secon
  - MFLOPS = Millions of Floating Point Ops per Second
- Both MIPS & MFLOPS need time measured in μsec

Throughput

- Marketing Hype vs. Actual Performance:
  - MIPS:
    - Machines with different instruction sets ?
    - Programs with different instruction mixes ?
      - Dynamic frequency of instructions
    - Uncorrelated with performance
  - MFLOPs:
    - Machine dependent
    - Often not where time is spent

Programs to Evaluate Processor Performance

- Toy Benchmarks
  - 10-100 line program
  - e.g.: sieve, puzzle, quicksort
- Synthetic Benchmarks
  - Attempt to match average frequencies of real workloads
  - e.g., Whetstone, dhrystone
- Kernels
  - Time critical excerpts from Real programs
  - e.g., gcc, spice

Common Benchmarking Mistakes

- Only average behavior represented in test workload
- Skewness of device demands ignored
- Loading level controlled inappropriately
- Caching effects ignored
- Buffer sizes not appropriate
- Inaccuracies due to sampling ignored

Common Benchmarking Mistakes

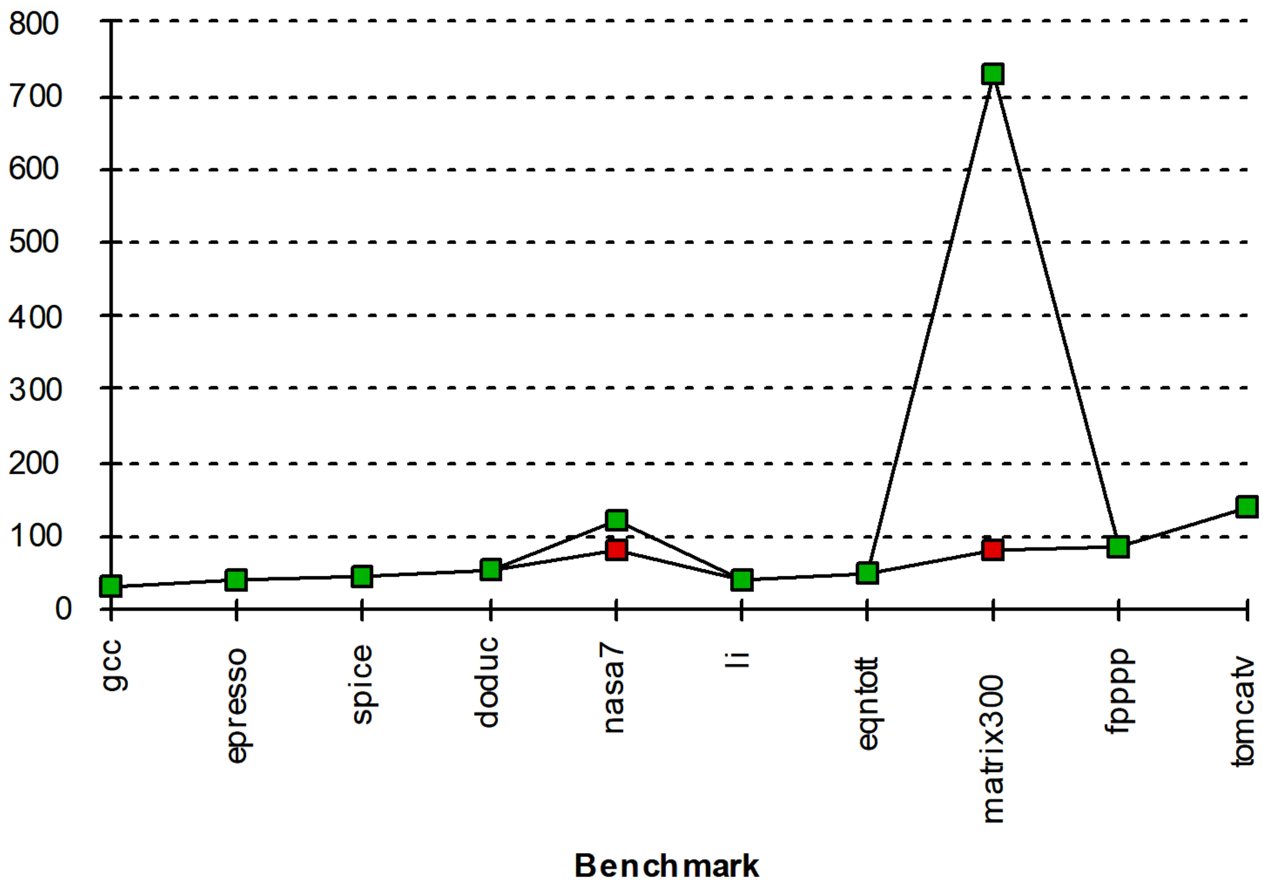
- Ignoring monitoring overhead
- Not validating measurements
- Not ensuring same initial conditions
- Not measuring transient (cold start) performance
- Using device utilizations for performance comparisons
- Collecting too much data but doing too little analysis

Revisiting SPEC: System Perf. Evaluation Cooperative

- First Round 1989
  - 10 programs yielding a single number
- Second Round 1992
  - SpecInt92 (6 integer programs) and
  - SpecFP92 (14 floating point programs)
    - Compiler Flags unlimited.
- Third Round 1995
  - Single flag setting for all programs;
  - new set of programs
  - “benchmarks useful for 3 years”

SPEC First Round

- One program: 99% of time in single line of code
- New front-end compiler could improve dramatically



How to Summarize Performance

- Arithmetic mean (weighted arithmetic mean)
  - tracks execution time:
- Harmonic mean (weighted harmonic mean) of rates
- Normalized execution time
  - Relative to some baseline system
  - Handy for scaling performance.

Summary

- Relative Performance Metrics
  - Speedup:

$$S_{X/Y} \equiv \frac{Perf_X}{Perf_Y} = \frac{T_Y}{T_X}$$

  - Percent Improvement:

$$n = 100\,[S_{X/Y} - 1]$$

- Amdahl's Law
  - Limited impact of enhancements

$$T_E = T_0\left[(1 - F_E) + \left(\frac{F_E}{S_E}\right)\right]$$

$$S_{E/0} = \frac{T_0}{T_E} = \frac{1}{(1 - F_E) + \left(\frac{F_E}{S_E}\right)}$$

Summary

- Absolute Performance Metrics
  - Exec time: $$T(N) = N \times CPI \times \tau$$
  - CPI: $$CPI = \sum_{k=1}^{m} F_k \times CPI_k$$
  - Throughput $$W(N) \equiv \frac{N}{T(N)}$$
    - Could be any granularity
    - If measured in Instructions: $$W(N) = \frac{1}{CPI \times \tau} = \frac{f}{CPI}$$


Photo Courtesy Michigan Tech Archives