The Note of CSE 120

Dongjing Wang

Created date: 03/31/2022

Contents

Moore's Law refers to Gordon Moore's perception that the number of transistors on a microchip doubles every two years, though the cost of computers is halved. Moore's Law states that we can expect the speed and capability of our computers to increase every couple of years, and we will pay less for them. Another tenet of Moore's Law asserts that this growth is exponential.

- **Pipelining**: overlap steps in execution; watch out for dependencies
- **Parallelism**: execute independent tasks in parallel
- **Prediction**: better to ask for forgiveness than permission...
- **Caching**: keep close a copy of frequently used information
- **Indirection**: go through a translation step to allow intervention
- **Amortization**: coarse-grain actions to amortize start/end overheads
- **Redundancy**: extra information or resources to recover from errors
- **Specialization**: trim overheads of general-purpose systems
- **Focus on the common case**: optimize only the critical aspects of the system

2 Lecture II

2.1 Performance

2.1.1 Performance Basics: Latency

- How long it takes to do X?
- Latency: the Base Units

- Clock period: duration of a clock cycle
  E.g., 250ps = 0.25ns = 250×10^{-12}s
  - This is the basic unit of time in all computers

- Clock frequency (rate): cycles per second
  E.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

2.1.2 Performance Basics: Throughput

The rate that a unit of work is completed or started by a mechanism
- How much X work is done in a given time?

2.1.3 Performance: Latency vs Throughput

-Latency improves by speeding up a core?
-Throughput improves by adding more cores?

2.1.4 Execution (or CPU) Time

- Execution time improved by
  -Reducing number of clock cycles
  -Or Increasing clock rate

- These two goals are not always compatible
  -Must often trade off clock rate against cycle count

- Execution Time
  -Execution Time = Cycles *x* Period =“cycles per program” / “clock rate”

2.1.5 Performance Definition

- For a machine A running a program P:
  - Perf(A,P) = 1/(Time(A,P))
- For two computer A and B, on the same program P:
  - IF Perf(A,P) > Perf(B,P) **THEN** Time(A,P) < Time(B,P)

2.1.6 Speedup or "relative performance"

- Speedup = Time(OLD) / Time(NEW)

2.1.7 Relative Performance

- Latency:
  - Time(A,P) = 1 hour, Time(B,P) = 2 hour
  - A is 2 times faster
- Bandwidth/Throughput:
  E.g., A fab can start one new chip per minute and takes 1 month to complete
  - Latency is 1 month
  - Throughput is 1 chip/minute
- More common architecture names:
  - Instructions per cycle
  - Nanoseconds (or picoseconds) cycle time
  - Frequency (inverse of cycle time)

2.2 Different Laws

2.2.1 Bell's Law

- Smaller form factors are initially slower, but more common and cheaper. Eventually they become the dominant form.

2.2.2 Iron Law

- Also known as the "CPU performance equation"
- Execution Time =“Instruction Count” × “Cycles Per Instruction” × “Clock Period”
- [(Implementation) (architecture) (realization)]

2.2.3 Amdahl's Law

- Partial Speedup
  - Performance gain limited by the fraction of improved
  - Speedup = $\frac{\text{CPUTime}_{\text{old}}}{\text{CPUTime}_{\text{new}}} = \frac{\text{CPUTime}_{\text{old}}}{\text{CPUTime}_{\text{old}}\left[(1-f_x)+\frac{f_x}{S_x}\right]} = \frac{1}{(1-f_x)+\frac{f_x}{S_x}}$
- Make Common Case Efficient
  - Lesson's from Amdahl's law
    - Make common cases fast: as $f_x \to 1$, Speedup$\to S_x$
    - But don't over-optimize common case: as $S_x \to \infty$, Speedup$\to$ 1 / (1-f_x)
      - Speedup is limited by the fraction of the code accelerated
      - Uncommon case will eventually become the common one
  - Amdahl's law applies to cost, power consumption, energy...
- Multiple Fractions
  - Do not memorize equation (many uses cases will not have what you need)
  - E.g: 2 partial speedup fractions

2.2.4 little's Law

Parallelism (or occupancy) = Throughput × Latency

- Little's law or flow balance formula
  - The average number of transactions in a stable system is equal to their average arrival rate, multiplied by their average time in the system
- Captures the relation between
  - Latency – how long it takes to do one thing
  - Throughput – the rate which things get done

	Computer A	Computer B	Computer C
Program P1(secs)	1	10	20
Program P2(secs)	1000	100	20
Program P3(secs)	1001	110	40

Table 1: Data Anlysis

  - Parallelism – how many things get done at a time

- Implications
  - Typically, want to improve one thing while another is constant
  - Example: want to improve throughput but latency is fixed
    - Implies must improve parallelism

- Example
  - Suppose you can make a sandwich in 2 minutes
    - Latencysandwich = 2min
    - Throughputsandwich = (1 sandwich / 2min) = 0.5 sandwiches / minute = 30 sandwiches / hour
  - Suppose that you need 30 minutes to each a sanwidth
    - How many tables do you need? (tables == parallelism eating)
    - Latencysandwich = 30min
    - Throughputsandwich = 0.5 sandwich / minute
    - Parallelism = 30 * 0.5 = 15 tables is the minimum
    - What if you have 5 tables?
      - 5 = 30 * x -> If you do more than 0.16 (5/30) sandwich per minute, you can not sell more

- Data Anlysis
  People have different ways to analyze the data in Table 1
  - Idea 1: (not great but some people use it)
    - (The way to analysis in **Table 2**). Bad idea because programs can execute for very different times. If a program runs for 1h, and the rest for few seconds. The small become irrelevant.

	Computer A	Computer B	Computer C
Program P1(sec)	1	10	20
Program P2(sec)	1000	100	20
Program P3(sec)	1001	110	40

Arithmetic Mean	~667	~73.3	~26.6
Speedup vs. A	1	667/73 = ~9.1	667/26 = ~25.01

Table 2: Idea 1

	Computer A	Computer B	Computer C
Program P1(sec)	1/1 = 1	10 / 1 = 10	20 / 1 = 20
Program P2(sec)	1000/1000 = 1	100/1000 = 0.1	20/1000 = 0.2
Program P3(sec)	1001/1001 = 1	110/1001 = ~0.11	40/1001 = ~0.04

Arithmetic Mean	~667	~3.4	~6.68
Speedup vs. A	1	1/3.4 = ~0.3	1/6.68 = ~0.15

Table 3: Idea 2

- Idea 2: Normalize against A, then "fake unit times" (A bit better but not great))
  - (The way to analysis in **Table 3**). A bit better but not great.
- Idea 3: 1st speedup, then average (look similiar idea, no?)
  - (The way to analysis in **Table 4**). What??? idea 2 and idea 3 "look reasonable" but they are very different?

- Not let's use geo mean[1] and try again. (Idea 1 is simple and inaccurate, so we skip that

[1]The example of Geometric Mean: $\sqrt[3]{(50 \times 75 \times 100)} \approx 72.1$. The example of Arithmetic Mean: $(50+75+100)/3 = 75$.

	Computer A	Computer B	Computer C
Program P1(sec)	1/1 = 1	1 / 10 = 0.1	1 / 20 = 0.05
Program P2(sec)	1000/1000 = 1	1000/100 = 10	1000/20 = 50
Program P3(sec)	1001/1001 = 1	1001/110 = ~9.1	1001/40 = ~25

Speedup vs. A	1	~6.4	~25.01

Table 4: Idea 3

	Computer A	Computer B	Computer C
Program P1(sec)	1/1 = 1	10/1 = 10	20/1 = 20
Program P2(sec)	1000/1000 = 1	100/1000 = 0.1	20/1000 = 0.02
Program P3(sec)	1001/1001 = 1	110/1001 = ~0.11	40/1001 = ~0.04

Geo Mean	1	~0.47	~0.252
Geo Speedup	1	1/0.47 = ~2.1	1/0.252 = ~3.97
Avg(Idea 2)	1	1/3.4 = ~0.3	1/6.68 = ~0.15

Table 5: Idea 2 + Geo mean

	Computer A	Computer B	Computer C
Program P1(sec)	1/1 = 1	1/10 = 0.1	1/20 = 0.05
Program P2(sec)	1000/1000 = 1	1000/100 = 10	1000/20 = 50
Program P3(sec)	1001/1001 = 1	1001/110 = ~9.1	1001/40 = ~25

Idea 3 + Geo	1	~2.1	~3.97
Idea 2 + Geo	1	1/0.47 = ~2.1	1/0.252 = ~3.97

Table 6: Idea 3 + Geo mean

one).

- Idea 2 + Geo Mean
- Idea 3 + Geo Mean
  - With Geo Mean, Idea2 or Idea3 yield the same result. Geo is more consistent, base independent and recommended.
- Summary
  - Arith < Geo < Harmonic
  - Geo and Harmonic must be normalized or weird
  - Geo is the most fair as it is base independent

2.3 CPI & IPC

2.3.1 Example

In the case of **Table 5**, CPI = 50% × 1 + 15% × 2 + 20% × 2 + 15% × 1 = 1.35, IPC = 1/1.35 ≈ 0.741.

ALU	50%	1
Branches	15%	2
Loads	20%	2
Stores	15%	1

Table 7: An example

2.3.2 Two Common Usages

- Instruction CPI
  - Each instruction has a fix CPI
  - E.g: load takes 7 cycles, adds 5 cycles
    - CPI_load = 7
    - CPI_add = 5
- Average CPI (most common meaning)
  - Depends on the program or instruction
  - Lower CPI means better performance
    - $\text{CPI} = \sum_{i=1}^{n} \frac{\text{IC}_i}{\text{IC}} \times \text{CPI}_i$
- IPC (Instructions Per Cycle) is the inverse of CPI

2.4 Benchmark

- Programs used to measure performance: Supposedly typical of actual workload
- Benchmark suite: collection of benchmarks
  - Plus datasets, metrics, and rules for evaluation
  - Plus a way to summarize performance in one number
- Examples
  - SPEC CPU2006 (integer and FP benchmarks)
  - TPC-H and TCP-W (database benchmarks)

  - EEMBC (embedded benchmarks)

- Warning
  - Different benchmarks focus on different workloads
  - All benchmarks have shortcomings
  - Your design will be as good as the benchmarks you use
- Instruction count and mix is very important
- Performance is best determined by running real applications
  - M1 may be faster with chess games than M2, but M2 may be faster with vim
    * I do not play chess in my computer, so M2 is better for me
    * If you do not use vim (poor you), M1 may be better for you
- "Standard" sets of benchmarks tend to be domain specific
  - SPECint and SPECfp for single core performance
  - SPECrate, parsec, splash2... for multicore
  - Rodinia for GPUs
  - TCP for databases
  - Dhrystone and EMMBC for embedded cores
  - ...

3 Lecture III

3.1 ISA

3.1.1 Time Table

- Accumulator
  - One implicit register (<1960)
  - Good: Simple hardware
  - Bad: Lots of memory traffic
- Stack - LIFO (1960-1970, hp48, and Java Virtual Machine!)

CISC	RISC
Memory-to-Memory load/store incorporated in instr.	Register-to-Register Separate load/store instructions
HW difficult to implement	HW easy to implement
Multi-cycle complex instructions	Simple (single clock) instructions
Small code size	Large code size
High CPI	Low CPI
Low clock frequency	High clock frequency
Variable length instructions	Same length instructions
Complex instruction decode	Simple instruction decode

Table 8: CISC vs. RISC

  - Smaller is faster (100-1000x)
  - Shorter names (fewer bits to encode)

- Registers (1960 - now)
  - Most popular architecture (RISC and CISC)

3.1.2 Different ways

- Stack Architecture
- Accumulator Architecture
- Register Memory Architecture (CISC)
- Load-Store Architecture (RISC)

3.1.3 RISC vs. CISC (Complex/Reduced Instruction Set)

- Hot debate in the 80s
- CISC (Complex Instruction Set Computer)
- RISC (Reduce Instruction Set Computer)
- Multiple comparisons (In Table 8)

3.2 Generic RISC

3.2.1 Instruction Types

- Arithmetic and logical: add, and, or, sub, xor, ...
- Data movement: loads/stores, register-to-register
- Control
  - compares
  - conditional branches
  - unconditional jumps/syscall
  - procedure call/return
- Floating point
- Miscellanea and extensions (Vector, string, BCD)

3.2.2 Control Instructions

- Branches(conditional)
  - branch <cond><target>
  - typical conditions are eq and ne
    - bne r1,r2,target
    - beq r3,r7,target
    - beq target
  - Branches vs Jumps
    - Branches, conditional control flow transfer
    - Jumps, unconditional control flow transfer
  - Jumps (unconditional)
    - jump <target>
  - Function calls and returns
    - Typically implemented with jump instructions
  - Targets are often PC-relative
    - fewer bits (target is typically close by)

        * position independence
    - Jumpl
        * Jump and link

3.2.3 ISA vs ABI

- ISA: Instruction Set Architecture
    - Instructions, encoding, instructions meaning
- ABI: Application Binary Interface
    - Conventions that the “compiler” or “app writer” has for a given ISA
        * E.g.
            · When doing a fcall, save to stack regs from x18 to x27
            · Callee or caller saves
    - In this class, you need to worry about ISA not ABI