Performance Metrics and Measurement

Wei Wang

Optional Readings from Textbooks

- “Computer Organization and Design,” Chapters 1.6 to 1.10.
- “Computer Architecture: A Quantitative Approach,” Chapter 1.8 “Measuring, Reporting and Summarizing Performance.”

Road Map

- Performance Metrics and Measurement
- The CPU Performance Equation
- Amdahl’s Law
- Benchmarks
- Simulators

Performance Metrics and Measurement

Performance Metrics

- Execution time is often what we target
- Throughput (tasks/sec) vs. latency (sec/task)
- How do we decide the tasks? Benchmarks
  - Processor design is a typical engineering process, no one design works the best for all use cases
  - Therefore, a processor design is typically optimized for a special set of use cases.
  - Benchmarks represent the applications for the target use cases.
  - Types of benchmarks,

    - Representative programs (SPEC, SYSMARK, etc)
    - Kernels: Code fragments from real programs (Linpack)
    - Toy Programs: Sieve, Quicksort
    - Synthetic Programs: Just a representative instruction mix (Whetsone, Dhrystone)

Measuring Performance

- Average Execution Time of all applications:

$$\frac{1}{n}\sum_{i=0}^{n} time_i$$

- This is arithmetic mean
  - This should be used when measuring performance in execution times.

Measuring Performance cont'd

- Weighted Execution Time:

$$\frac{1}{n}\sum_{i=0}^{n} weight_i \times time_i$$

- Weighted average is useful when different types of applications have different importance.

Measuring Performance cont'd

- Normalized performance
  - Execution times are normalized to the performance of a reference system.

$$\sqrt[n]{\prod_{i=0}^{n} \frac{ref\_time_i}{time_i}}$$

- Geometric mean is better here (arithmetic mean can vary depending on the reference system).
- Usually measures performance gains/losses over the reference system

Harmonic Mean

- 30 mph for the first 10 miles
- 90 mph for the next 10 miles
- Average speed? (30+90)/2 = 60mph
- WRONG! Average speed = total distance / total time
  - 20/(10/30+10/90) = 45mph

Harmonic Mean (cont’d)

- The same idea applies when compute the average rate of computer operations.
- Consider n applications, each perform O_n operations in t_i time, $1<i<n$
- Then the average operation rate (number of operations per unit time) is

$$Rate_{avg} = \frac{\sum O_i}{\sum t_i}$$

CPI and IPC

- CPI: Cycles per instruction
  - A common processor performance metric for execution times
- IPC: Instructions per cycle
  - A common processor performance metric for execution rates.

MIPS

- Millions of Instructions Per Second (MIPS)
  - Not the MIPS ISA!
- `MIPS`
  `= instruction count/(execution time x 10`^{6}`)`
  `= clock rate/(CPI x 10`^{6}`)`
- Problems
  - ISAs are not equivalent, e.g. RISC vs. CISC
    - 1 CISC instruction may equal many RISC!
  - Programs use different instruction mixes
  - May be ok when comparing same benchmarks, same ISA, same compiler, same OS

MFLOPS

- Millions of FLoating-point Operations Per Second (MFLOPS)
- Can be mis-leading either,
  - FP-intensive apps needed
  - Traditionally, FP ops were slow, integer operations can be ignored
  - BUT today, memory operations are usually the slowest!
- "Peak MFLOPS" is a common marketing fallacy
  - Basically, it just says #FP-pipes X Clock Rate
  - Peak performance is not sustainable, hard to achieve with real applications.

Processor Frequency

- Is this a metric? Maybe as good as the others...
- One number, no benchmarks, what can be better?
- Many designs are frequency driven.
  - Common before 2004.
  - Nowadays, power consumption is also important.

CPU Performance Equation

CPU Performance Equation

- Execution_Time = seconds/program

$$\frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

Affected by

Algorithm;
ISA;
Compiler;

Compiler (scheduling);
Micro-Architecture;

Technology (transistor size);
Physical Chip Design;
Circuit Design;

Common Architecture Tricks

- Instructions/Program (Path-length) is constant
  - Same benchmark, same compiler
  - Ok usually, but for some ideas compiler may change
- Seconds/Cycle (Cycle-time) is constant
  - “My tweak won’t impact cycle-time”
  - Often a bad assumption
- Just focus on Cycles/Instruction (CPI or IPC)
  - Most academic architecture studies do just this!

Bottom-line of Performance Metrics

- Two quotes from “Computer Organization and Design,”
  - “Execution time is the only valid and unimpeachable measure of performance.”
  - “Similarly, any measure that summarizes performance should reflect execution time.”

Amdahl’s Law

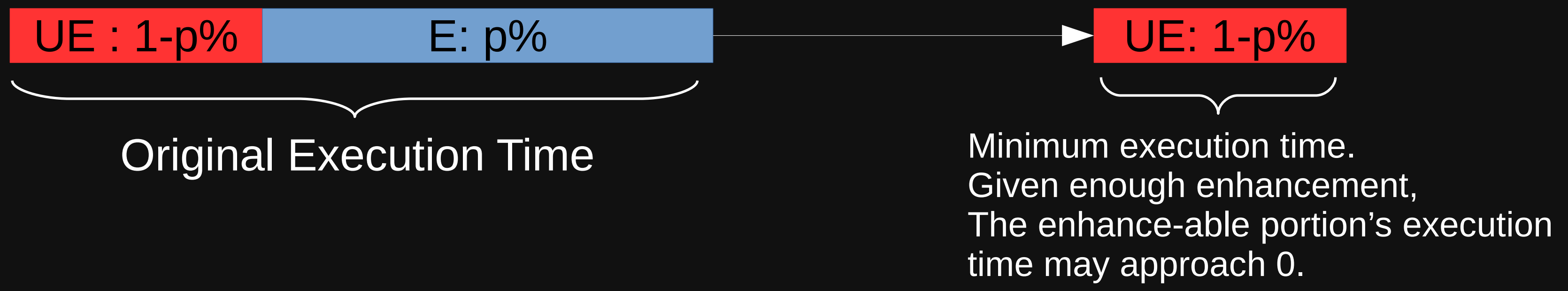
The Basic Speedup Definition for Enhanced Execution

- Speedup of a enhanced execution of an application is defined as

$$Speedup = \frac{\text{Original Execution Time}}{\text{Enhanced Execution Time}}$$

Amdahl's Law

- Intuition of Amdahl's law:
  - If a group of applications are *p%* enhance-able (E) and *(1-p%)* un-enhance-able (UE),
    - the minimum total execution time of these application is the execution times of the un-enhance-able portion.
    - The maximum speedup is bounded by the un-enhance-able portion.



Amdahl's Law: Equation for Speedup

- The equation of Amdahl's Law:

$$Speedup = \frac{\text{Original Exec Time}}{\text{Enhanced Exec Time}} = \frac{1 - p\% + p\%}{(1 - p\%) + \frac{p\%}{s}} = \frac{1}{(1 - p\%) + \frac{p\%}{s}}$$

  - *Speedup* is the overall speedup of all applications after enhancement
  - *p%* is the percentage of the enhance-able applications
  - *s* is the speedup of the enhanced applications

Amdahl's Law: The Limit on Speedup

- If the enhance-able applications' speedup approaches infinity:

$$\lim_{(s \to \infty)} Speedup = \frac{1}{1 - p\% + \frac{p\%}{s}} = \frac{1}{1 - p\%}$$

  - i.e., the maximum speedup is bounded by the execution time of the unenhance-able applications

Amdahl's Law Examples

$$p = Fraction_{enhance} = 95\%,\ s = Speedup_{enhance} = 1.1x$$

$$Speedup_{overall} = \frac{1}{(100\% - 95\%) + \frac{95\%}{1.1}} = 1.094$$

$$p = Fraction_{enhance} = 5\%,\ s = Speedup_{enhance} = 10.0x$$

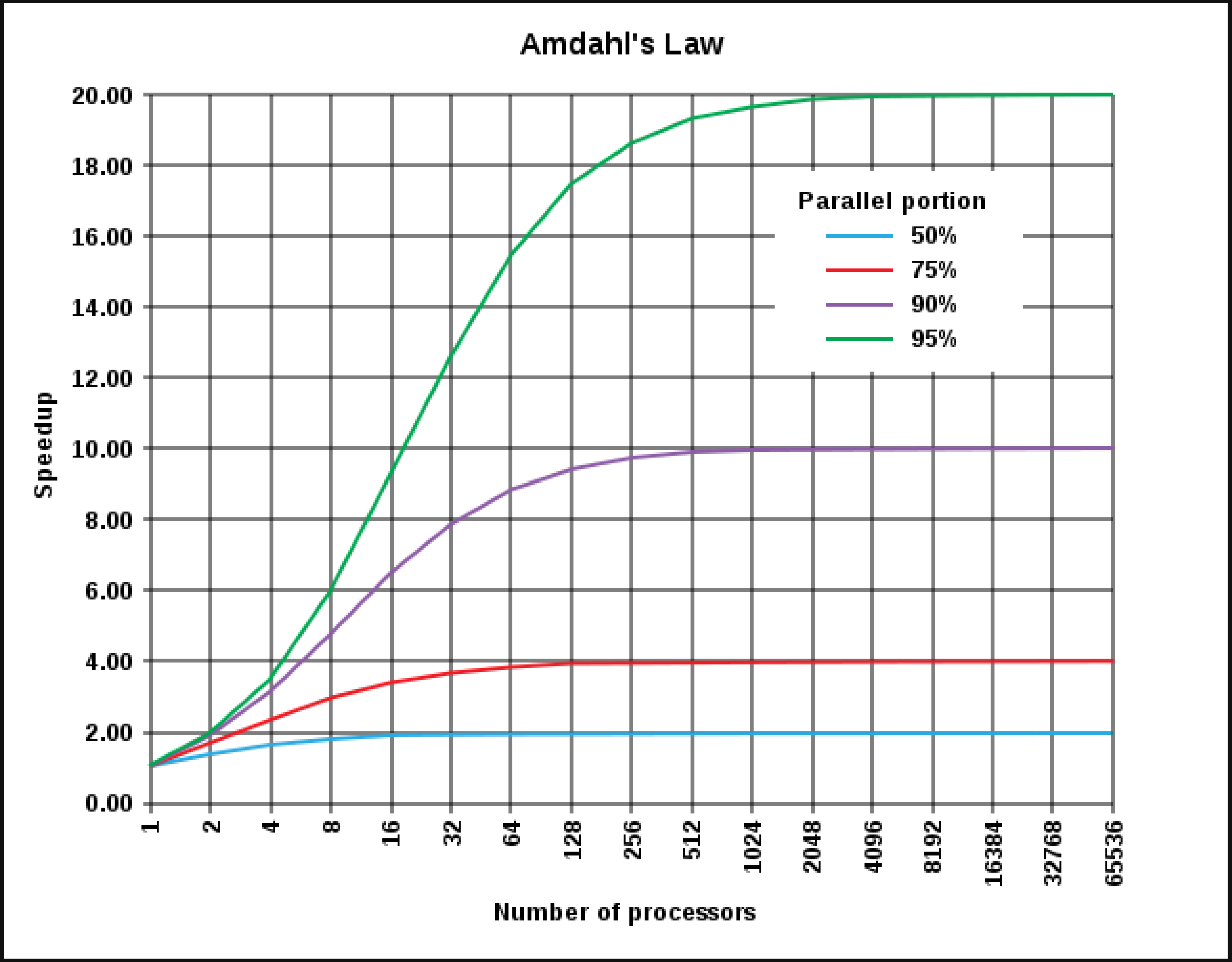
$$Speedup_{overall} = \frac{1}{(100\% - 5\%) + \frac{5\%}{10.0}} = 1.047$$

Better to improve the common cases.

$$p = Fraction_{enhance} = 5\%,\ s = Speedup_{enhance} = \infty$$

$$Speedup_{overall} = \frac{1}{(100\% - 5\%)} = 1.052$$

Visualization of Amdahl's Law with Parallelization as the Enhancement



Benchmarks

Benchmarks

- Benchmarks are typical applications that represent a common use case.
  - Representativity and completeness are two main requirements.
- Common benchmark suites
  - *SPEC CPU* (int and float) represents common desktop and server applications
  - *Rodinia* represents common general purpose GPU applications.
  - *NAS Parallel Benchmark* Suite, represents scientific applications.
  - *TPC* benchmark suite, represents online transactional database applications.
  - *Yahoo! Cloud Serving Benchmark*, represents NoSQL database applications.
  - *Cloud Suite*, represents cloud applications.

Benchmark Suite Example: SPEC CPU INT 2006

Benchmark	Language	Descriptions
400.perlbench	C	Perl Interpreter
401.bzip2	C	Compression
403.gcc	C	Compiler
429.mcf	C	Vehicle scheduling, route planning
445.gobmk	C	The game of Go
456.hmmer	C	Protein sequence analysis
458.sjeng	C	A chess program
462.libquantum	C	Simulates a quantum computer
464.h264ref	C	A reference implementation of H.264/AVC
471.omnetpp	C++	OMNet++ discrete event simulator
473.astar	C++	Pathfinding library for 2D maps
483.xalancbmk	C++	XML translation

Benchmark Suite Example: SPEC CPU INT 2017

Benchmark	Language	Description
600.perlbench_s	C	Perl interpreter
602.gcc_s	C	GNU C compiler
605.mcf_s	C	Route planning
620.omnetpp_s	C++	Discrete Event simulation - computer network
623.xalancbmk_s	C++	XML to HTML conversion via XSLT
625.x264_s	C	Video compression
631.deepsjeng_s	C++	Artificial Intelligence: alpha-beta tree search (Chess)
641.leela_s	C++	Artificial Intelligence: Monte Carlo tree search (Go)
648.exchange2_s	Fortran	Artificial Intelligence: recursive solution generator (Sudoku)
657.xz_s	C	General data compression

Benchmark Suite Example: SPEC CPU FP 2006

Benchmark	Language	Descriptions
410.bwaves	Fortran	Fluid Dynamics
416.gamess	Fortran	Quantum Chemistry.
433.milc	C	Physics / Quantum Chromodynamics
434.zeusmp	Fortran	Physics / CFD
435.gromacs	C, Fortran	Biochemistry / Molecular Dynamics
436.cactusADM	C, Fortran	Physics / General Relativity
437.leslie3d	Fortran	Fluid Dynamics
444.namd	C++	Biology / Molecular Dynamics
447.dealII	C++	Finite Element Analysis
450.soplex	C++	Linear Programming, Optimization
453.povray	C++	Image Ray-tracing

Benchmark Suite Example: SPEC CPU FP 2006 cont'd

Benchmark	Language	Descriptions
454.calculix	C, Fortran	Structural Mechanics
459.GemsFDTD	Fortran	Computational Electromagnetics
465.tonto	Fortran	Quantum Chemistry
470.lbm	C	Fluid Dynamics
481.wrf	C, Fortran	Weather
482.sphinx3	C	Speech recognition

Benchmark Suite Example: SPEC CPU FP 2017

Benchmark	Language	Description
603.bwaves_s	Fortran	Explosion modeling
607.cactuBSSN_s	C++, C, Fortran	Physics: relativity
619.lbm_s	C	Fluid dynamics
621.wrf_s	Fortran, C	Weather forecasting
627.cam4_s	Fortran, C	Atmosphere modeling
628.pop2_s	Fortran, C	Wide-scale ocean modeling (climate level)
638.imagick_s	C	Image manipulation
644.nab_s	C	Molecular dynamics
649.fotonik3d_s	Fortran	Computational Electromagnetics
654.roms_s	Fortran	Regional ocean modeling

The Challenge to Design Benchmarks

- Applications and use cases are constantly evolving.
  - We still do have good (or widely accepted) Machine Learning benchmarks yet.
- Too many potential applications, hard to be both representative and complete
  - When Cloud Suite first came out, people were susceptible to it, as its memory behavior contradicted common believes.

Simulators

Simulators

- What is a simulator?
  - A simulator is a software written to model (simulate) the operations of real hardware devices.
- Why use simulator?
  - Building real chips is expensive.
  - In architecture design phase, there are multiple prototype designs that need to be tried out (with benchmarks).
  - Software simulators are much easier to modify and cheaper to experiment with.

Types of Simulators

- Full system simulator
  - Simulate most if not all hardware components, including the processor and memory.
  - Typically can run a full OS on it.
  - E.g., GEM5, Simics
- Mirco-architecture Simulator
  - Simulate a processor or its interval components
  - May be cycle-accurate in that the simulator faithful reproduce the processor operations cycle-by-cycle. Cycle-accurate provides details interval insights about the processor.
  - Usually can run a simple program with simple or no OS system calls
  - E.g., SimpleScalar CPU simulator, CMP$Sim cache simulator
- Instruction Set Simulator
  - Simulate whole instruction set execution
  - Usually can run a simple program with simple or no OS system calls
  - E.g., SPIM simulator simulates MIPS ISA.
- Special Metric Simulators
  - There are some simulators that are used to determine rare hardware design metrics, such as power consumptions and chip area size.
  - E.g., McPAT for power, area and timing simulation.

Disadvantages of Simulators

- Slow!
  - It may take months to simulate just one second (one billion cycles) of execution
- In-accurate
  - There may be bugs or incorrect assumptions in the simulator.
  - To reduce execution time, some simulators simplify the components it considers unimportant.
  - Cycles are easy to account, but energy usage and chip area sizes are hard to determine accurately with simulator.
  - No one believes other people’s simulator results…
- Simulators are typically used in the early step in chip design. Real prototype chips are still indispensable in the design flow.

Acknowledgment

- These slides are partially based on the lecture notes from Dr. David Brooks.