Chapter 1

Instructor: Josep Torrellas
CS433
Course Goals

- Introduce you to design principles, analysis techniques and design options in computer architecture
  - Instruction set design
  - Memory-hierarchy design
  - Pipelining
  - I/O

- The use of cost/performance as a basis for making decisions about computer architecture
- Computer architecture is exciting
- Get you to ask interesting questions about computer architecture
Background

• Assume you have taken:
  – A basic computer organization course
  – A logic design course
  – Assembly language programming

• Assume that you know
  – What an instruction set looks like
  – How to program in C
The Parts of a Computer

- Integer datapath
  - Registers

- Control unit
  - Registers

- FP datapath
  - Registers

- I/O system
  - Disks
  - Graphics Display
  - Network

- Memory system
  - Cache
  - TLB
  - Main Memory
Why Study Computer Architecture?
Performance?

- What do we mean when we say computer A is faster than computer B

- Response time
  - Time from start to completion of an event
  - Execution time
  - Latency

- Throughput
  - Amount of work done per unit time
  - Bandwidth
Performance

- Program **execution time** is the measure of performance (seconds/program)
- Definition of execution time
  - Wall clock time, elapsed time as seen by user includes everything (disk, OS overhead, competition with other jobs).
  - CPU time: time CPU computing on your program, excluding I/O wait time
    - 1. User CPU time
    - 2. System CPU time
- System Performance: elapsed time of an unloaded system
- CPU Performance: user CPU time
  - 90.7s: user CPU time
  - 12.9s: system CPU time
  - 2:39: elapsed time
  - 65%: CPU/elapsed
Evaluating Performance

• Use real programs
  – CAD, text processing, business applications, scientific applications
  – input, output, options
  – May not know what programs users will run

• Kernels:
  – Small key pieces (inner loops) of scientific programs where program spends most of its time
  – e.g. Livermore loops, LINPACK
  – Amenable to hand analysis

• Toy Benchmarks
  – e.g. Quicksort, Puzzle
  – Easy to type, predictable results, may use to check correctness of machine but not as performance benchmark.
Summarizing Performance

- Model a real job mix with a smaller set of representative programs
- Total execution time is the ultimate measure of performance
- Weight benchmarks according to time spent in a real job mix
- How do you summarize performance?
- A single-number performance summary for the programs expressed in units of time should be directly proportional to (weighted) execution time
- A single-number performance summary for the programs expressed as a rate should be inversely proportional to (weighted) execution time
Summarizing Performance

• Given n programs,
  – Average of execution time: arithmetic mean
    \[(1/n)*(\text{Time}_1 + \text{Time}_2 + \ldots + \text{Time}_n)\]
  – If performance is expressed as a rate: the average that tracks execution time: harmonic mean
    \[\frac{n}{(1/\text{rate}_1 + 1/\text{rate}_2 + \ldots + 1/\text{rate}_n)}\]
    where \(\text{rate}_j = f (1/\text{Time}_j)\)
Summarizing Performance

- **Weighted arithmetic mean**
  \[(\text{Time}_1 \times \text{Weight}_1 + \text{Time}_2 \times \text{Weight}_2 + \ldots + \text{Time}_n \times \text{Weight}_n)\]
  where \(\text{Weight}_1 + \text{Weight}_2 + \ldots + \text{Weight}_n = 1\),
  and \(\text{Weight}_j > 0\)

- **Weighted harmonic mean**
  \[1/(\text{Weight}_1/\text{Rate}_1 + \text{Weight}_2/\text{Rate}_2 + \ldots + \text{Weight}_n/\text{Rate}_n)\]
  For example, \(\text{Rate}_j\) is the MIPS rate of machine \(j\).
Geometric Mean for Normalized Execution Time

• Normalize execution time of a program $j$ to the execution time in a reference machine
  $$\Rightarrow \text{Execution time ratio}_j$$

• Geometric mean:
  $$\text{n th root of } \{(\text{Execution time ratio}_1)*…*(\text{Execution time ratio}_n)\}$$
Make the Common Case Fast

- Very important, sort of obvious but often overlooked
- Common case is made slower to make a less common case faster
- The frequent case is often simpler and can be done faster (e.g. addition rarely overflows)
- Not following this principle can increase design time.
- Complex problems should be handled in software
- Hardware should provide fast primitives, not complete solutions
Amdahl’s Law

- Performance gain from improvement of some portion of a computer
- Original Observation: Speedup from parallel processing is limited by the fraction that cannot be parallelized.
- In general,
  \[
  \text{Speedup} = \frac{\text{ET without enh}}{\text{ET with enh}} = \frac{\text{ET}_{\text{old}}}{\text{ET}_{\text{new}}}
  \]
  where enh =>$\rightarrow$ enhancement
  \[
  \text{ET}_{\text{new}} = \text{ET}_{\text{old}} \times \left\{ (1 - \text{fraction\_enh}) + \text{fraction\_enh} / \text{Speedup\_enh} \right\}
  \]
  \[
  \text{Speedup} = \frac{1}{\left\{ (1 - \text{fraction\_enh}) + \text{fraction\_enh} / \text{Speedup\_enh} \right\}}
  \]
Application of Amdahl’s Law

• Parallel application that is 90% parallel, what is the speedup for application on 10, 100 and 1000 processors.
  – 10% I/O and initialization: \( s \)
  – 90% parallel: \( p \)
    – \( S_p = \frac{1}{(s + p/P)} = \frac{1}{(0.1 + 0.9/P)} \)
    – \( S_{10} = \frac{1}{(0.1 + 0.9/10)} = \frac{1}{0.19} = 5.26 \)
    – \( S_{100} = \frac{1}{(0.1 + 0.9/100)} = \frac{1}{0.109} = 9.1 \)
    – \( S_{1000} = 9.9 \)

  – Diminishing returns in performance
Increasing Parallelism

- 9% I/O initialization: s

- 91% parallel: p
  - \( S_p = \frac{1}{s + \frac{p}{P}} = \frac{1}{0.09 + \frac{0.91}{P}} \)
  - \( S_{100} = \frac{1}{0.09 + \frac{0.91}{100}} = \frac{1}{0.099} = 10.09 \)
Using Amdahl’s Law

- Making cost performance trade-offs
  - Application spends 50% time in CPU and 50% of time waiting for I/O
  - Cost of CPU = 1/3, cost of I/O = 2/3
  - New CPU increases CPU performance 5 times and CPU cost 5 times
  - Is using a new CPU a good idea from a cost/performance standpoint.
    - Speedup = 1/ (0.5 + 0.5/5) = 1/0.6 = 1.67
    - Cost increase = 2/3*1 + 1/3*5 = 2.33

- Spend resources proportionately to where time is spent
- How much should we increase CPU speed for equal speedup and cost increase?
CPU Performance

CPU time = CPU clock cycles per program * Clock cycle time

CPI = \[
\frac{\text{CPU clock cycles per program}}{\text{Instruction count}}
\]

CPU time = Instruction count*CPI*Clock cycle time

\[
\frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}
\]

Compiler  ISA  Organization  Technology
CPU Performance

• Another way of looking at CPU time

\[
\text{CPU time} = (CPI_1*I_1 + CPI_2*I_2 + \ldots + CPI_n*I_n)*\text{Clock cycle time}
\]

• CPI is now

\[
CPI = CPI_1*(I_1/\text{Instruction count}) + CPI_2*(I_2/\text{Instruction count}) + \ldots + CPI_n*(I_n/\text{Instruction count})
\]

Frequency of \( I_j \) = \( I_j/\text{Instruction count} \)
CPU Performance Example

-- Instruction frequencies for a load/store machine

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Frequency</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>25%</td>
<td>2</td>
</tr>
<tr>
<td>Stores</td>
<td>15%</td>
<td>2</td>
</tr>
<tr>
<td>Branches</td>
<td>20%</td>
<td>2</td>
</tr>
<tr>
<td>ALU</td>
<td>40%</td>
<td>1</td>
</tr>
</tbody>
</table>

-- All conditional branches in this machine use simple tests of equality with zero
    BEQZ, BNEZ
-- Consider adding complex comparisons to conditional branches
-- 25% of branches can use complex scheme--> no need preceding ALU instruction
-- The CPU cycle time of original machine is 10% faster
-- Will this increase CPU performance?
CPU Performance Example

- **Old CPU performance**
  - \( \text{CPI}_{\text{old}} = 0.25 \times 2 + 0.15 \times 2 + 0.2 \times 2 + 0.4 \times 1 = 1.6 \)
  - \( \text{CPU time}_{\text{old}} = 1.6 \times \text{IC}_{\text{old}} \times \text{CCT}_{\text{old}} \)

- **New CPU Performance**
  - \( \text{CPI}_{\text{new}} = \frac{0.25 \times 2 + 0.15 \times 2 + 0.2 \times 2 + (0.4 - 0.25 \times 0.2) \times 1}{1 - 0.25 \times 0.2} = 1.63 \)
  - \( \text{IC}_{\text{new}} = 0.95 \times \text{IC}_{\text{old}} \)
  - \( \text{CCT}_{\text{new}} = 1.1 \times \text{CCT}_{\text{old}} \)
  - \( \text{CPU time}_{\text{new}} = 1.63 \times (0.95 \times \text{IC}_{\text{old}}) \times (1.1 \times \text{CCT}_{\text{old}}) = 1.71 \times \text{IC}_{\text{old}} \times \text{CCT}_{\text{old}} \)
Locality of Reference

- Fundamental observation about programs
- Possible to predict with high accuracy what a program will reference next based on what it has referenced in the recent past
- Temporal locality: recently referenced locations are likely to be referenced again (e.g. code loops, stack accesses).
- Spatial locality: nearby locations reference together (e.g. array access, code access)
- Memory hierarchy used to exploit locality
Memory Hierarchy

• Basic principle of hardware design: smaller is faster
  – Small memories have less signal propagation delay and decoding
  – Small memories can use more power per cell for speed
  – Small fast memories cost more
• Use memory hierarchy to cost/performance of computer
  – CPU registers
  – Cache
  – Main memory
  – Disk memory
denser, cheaper

smaller, faster

CPU
Registers → Cache ← Memory → Disk

23
MIPS

• An indirect measure of performance

\[
\text{million instr/sec} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} = \frac{\text{Instr. Count}}{\# \text{cycles} \times \text{sec/cycle} \times 10^6} = \frac{\text{clock rate}}{\text{CPI} \times 10^6}
\]

– Execution time = instruction count / MIPS / 10^6

• Problems with MIPS

– Difficult to compare different ISA
– No indication of program or program input
– MIPS can vary inversely to performance (?)

• Native MIPS
Megaflops

- \( \text{MFLOPS} = \frac{\text{# of floating point operations in program}}{\text{Execution time} \times 1,000,000} \)

- Does not measure integer performance
- Assumes that same number and type of operations are executed on all machines
- Changes with mixture of fast and slow operations (type of float pt. operations)
- A function of instruction mix (% of float pt. operations)
Figure 1.15 This 300 mm wafer contains 280 full Sandy Bridge dies, each 20.7 by 10.5 mm in a 32 nm process. (Sandy Bridge is Intel’s successor to Nehalem used in the Core i7.) At 216 mm2, the formula for dies per wafer estimates 282. (Courtesy Intel.)
Die

Figure 1.13 Photograph of an Intel Core i7 microprocessor die, which is evaluated in Chapters 2 through 5. The dimensions are 18.9 mm by 13.6 mm (257 mm²) in a 45 nm process. (Courtesy Intel.)
Die Floorplan

Figure 1.14 Floorplan of Core i7 die in Figure 1.13 on left with close-up of floorplan of second core on right.