Parallelism and Performance

Intensive Computation

Annalisa Massini
2017/2018

References

Computer Architecture - A Quantitative Approach

Hennessy Patterson

Chapter 1 - Fundamentals of Quantitative Design and Analysis

Section 1.8 - Measuring, Reporting, and Summarizing Performance

Section 1.9 - Quantitative Principles of Computer Design

Measuring Performance

- When we say one computer is faster than another we can mean different things.

- The computer user is interested in reducing *response time* - the time between the start and the completion of an event - also referred to as *execution time*

- The operator of a warehouse-scale computer may be interested in increasing *throughput* - the total amount of work done in a given time

Measuring Performance

- In comparing design alternatives, we often want to relate the performance of two different computers: X and Y

- When we say “X is faster than Y” we mean that the response time or execution time is lower on X than on Y for the given task
- In particular, “X is *n* times faster than Y” will mean:

$$\frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Measuring Performance

- Since *execution time is the reciprocal of performance*, the following relationship holds:

$$n = \frac{\text{Execution time Y}}{\text{Execution time X}} = \frac{\text{Performance X}}{\text{Performance Y}}$$

- The phrase *the throughput of X is 1.3 times higher than Y* signifies that the number of tasks completed per unit time on computer X is 1.3 times the number completed on Y

Measuring Performance

- Unfortunately, time is not always the metric quoted in comparing the performance of computers
- But (for Hennessy and Patterson) ***the only consistent and reliable measure of performance is the execution time of real programs***
- All proposed alternatives to time as the metric or to real programs as the items measured have eventually led to misleading claims or even mistakes in computer design

Measuring Performance

- Even execution time can be defined in different ways depending on what we count

- The most straightforward definition of time is called

  wall-clock time, *response time*, or *elapsed time*

  which is the latency to complete a task, including *disk accesses, memory accesses, input/output activities, operating system overhead everything*

Measuring Performance

- With multiprogramming, the processor works on another program while waiting for I/O and may not necessarily minimize the elapsed time of one program
- Hence, we need a term to consider this activity

- ***CPU time*** recognizes this distinction and means the time the processor is computing, ***not*** including the time waiting for I/O or running other programs

- Clearly, the response time seen by the user is the elapsed time of the program, not the CPU time

Measuring Performance

- **Benchmarks** can be used to measure performance
- The best choice of benchmarks is real applications
- Attempts at running programs much simpler than a real application have led to performance pitfalls
- Examples include:
  - *Kernels*, which are small, key pieces of real applications
  - *Toy programs*, which are 100-line programs (such as quicksort)
  - *Synthetic benchmarks*, which are fake programs invented to try to match the profile and behavior of real applications (as Dhrystone)
- All three are discredited today (compiler writer and architect can conspire to make the computer appear faster than on real applications)

Taking advantage of parallelism

- In the design and analysis of computers, we need
  - Principles and guidelines
  - Observations about design
  - Equations to evaluate alternatives

- Taking advantage of parallelism is one of the most important methods for improving performance
  - Parallelism at the system level – scalability
  - Parallelism at the level of an individual processor - parallelism among instructions
  - Parallelism at the level of digital design - memories and ALUs

Taking advantage of parallelism

- Fundamental observations come from properties of programs

- The most important program property that we regularly exploit is the ***principle of locality***
  - ***Temporal locality*** states that recently accessed items are likely to be accessed in the near future
  - ***Spatial locality*** says that items whose addresses are near one another tend to be referenced close together in time

Taking advantage of parallelism

- An important and pervasive principle of computer design is to focus on the ***common case***:
  - In making a design trade-off, **favor the frequent case** over the infrequent case
- This principle applies when determining how to spend resources, since the impact of the **improvement is higher** if the occurrence is **frequent**

- In applying this simple principle, we have to decide what the frequent case is and how much performance can be improved by making that case faster

Amdahl's Law

- The performance gain obtained by improving some portion of a computer can be calculated using **Amdahl's law**
- Amdahl's law:
  - states that the ***performance improvement is limited*** by the ***fraction of the time*** the faster mode can be used
  - defines the ***speedup*** that can be ***gained by using a particular*** **feature**

Speedup =

(**Performance** for entire task **using** the enhancement when possible)***/***
(**Performance** for entire task **without using** the enhancement)

Speedup =

(**Execution time** for entire task **without using** the enhancement)***/***
(**Execution time** for entire task **using** the enhancement when possible)

Amdahl's law

- Amdahl's law gives us a quick way to find the speedup from some enhancement, which depends on two factors:

1) The **fraction of the computation time** in the original computer that can be converted to take advantage of the enhancement, that is

   Fraction$_{\textbf{enhanced}}$ = time with enhancement/total time

   Example:

   - A program that takes 60 seconds in total
   - 20 seconds of the execution time can use an enhancement
   - The fraction is: 20/60
   - **This value is always less than or equal to 1**

Amdahl's law

- Amdahl's law gives us a quick way to find the speedup from some enhancement, which depends on two factors:

2) The improvement gained by the enhanced execution mode, that is, how much faster the task would run if the enhanced mode were used for the entire program:

   Speedup$_{enhanced}$ = original mode time / enhanced mode time

   Example:

   - A portion of the program in the original mode is 5 seconds
   - In the enhanced mode takes 2 seconds
   - The improvement is 5/2
   - **This value is always greater than 1**

Amdahl's law

- The **execution time** using the original computer with the enhanced mode will be the time spent using the unenhanced portion of the computer plus the time spent using the enhancement:

$$\text{Execution time}_{new} = \text{Execution time}_{old} \times \left((1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}} \right)$$

- The overall **speedup** is the ratio of the execution times:

$$\text{Speedup}_{overall} = \frac{\text{Execution time}_{old}}{\text{Execution time}_{new}} = \frac{1}{(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}$$

Example

- We want to enhance the processor used for Web serving
- The new processor is **10 times faster** on computation in the Web serving application than the original processor
- Assume that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time
- ***What is the overall speedup gained by incorporating the enhancement?***

Example

- We want to enhance the processor used for Web serving
- The new processor is **10 times faster** on computation in the Web serving application than the original processor
- Assume that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time
- ***What is the overall speedup gained by incorporating the enhancement?***

$$\text{Fraction}_{\text{enhanced}} = 0.4 \qquad \text{Speedup}_{\text{enhanced}} = 10$$

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}} = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = \frac{1}{0.64} = 1.6$$

Amdahl's law

- Amdahl's law can serve as a guide to:
  - how much an enhancement will improve performance
  - how to distribute resources to improve cost-performance
- The goal is to spend resources proportional to where time is spent
- Amdahl's law is useful
  - for ***comparing the overall system performance of two alternatives***
  - to compare two processor design alternatives

Example

- A common transformation in graphics processors is ***square root***
- Implementations of floating-point square root (FPSQR) vary significantly in performance among processors for graphics

- Suppose
  - **FPSQR is responsible for 20% of the execution time** of a critical graphics benchmark and
  - **FP instructions are responsible for half of the execution time** for the application

Example

- Two proposals:
  - To enhance the FPSQR hardware and speed up this operation by a factor of 10
  - To try to make all FP instructions in the graphics processor run faster by a factor of 1.6
- Compare these two design alternatives

Example

- We can compare these two alternatives by comparing the speedups

$$\text{Speedup}_{\text{FPSQR}} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

Example

- We can compare these two alternatives by comparing the speedups

$$\text{Speedup}_{\text{FPSQR}} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1-0.5)+\frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

Example

- We can compare these two alternatives by comparing the speedups

$$\text{Speedup}_{\text{FPSQR}} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1-0.5)+\frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

- Improving the performance of the FP operations overall is slightly better because of the higher frequency

Processor Performance Equation

- All computers are constructed using a clock running at a constant rate
- Discrete time events are called *ticks, clock ticks, clock periods, clocks, cycles, or clock cycles*
- Computer designers refer to the time of a clock period by its duration (e.g., 1 ns) or by its rate (e.g., 1 GHz)

- CPU time for a program can then be expressed two ways:
  - **CPU time = CPU clock cycles for a program × Clock cycle time**

or

  - **CPU time = CPU clock cycles for a program / Clock rate**

Processor Performance Equation

- We can also count the number of instructions executed - the ***instruction path length*** or ***instruction count (IC)***
- If we know the **number of clock cycles** and the **instruction count**, we can calculate the average number of ***clock cycles per instruction (CPI):***

CPI = CPU clock cycles for a program / IC

- From this formula we obtain
  - **CPU clock cycles for a program = CPI x IC**

Processor Performance Equation

- This allows us to use CPI in the execution time formula and obtain the **performance equation**:
  - **CPU time = IC × CPI × Clock cycle time**
- In fact using the units of measurement we have:

$$\text{IC} \times \text{CPI} \times \text{Clock cycle time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instructions}} \times \frac{\text{Seconds}}{\text{Clock cycles}} =$$

$$= \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

- Observe that **processor performance** is ***equally* dependent** upon **clock cycle** (or rate), **clock cycles per instruction**, and **instruction count**

Processor Performance Equation

- It is useful to calculate the number of total processor clock cycles as

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

- where
  - IC_i is the number of times instruction i is executed in a program
  - CPI_i is the average number of clocks per instruction for instr. i

Processor Performance Equation

- This expression can be used to express CPU time as

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}$$

- and the overall CPI as

$$\text{CPI} = \frac{\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i}{\text{Instruction count}} = \sum_{i=1}^{n} \frac{\text{IC}_i}{\text{Instruction count}} \times \text{CPI}_i$$

Example

- Suppose we have made the following measurements in the previous example :
  - Frequency of FP operations = 25%
  - Average CPI of FP operations = 4.0
  - Average CPI of other instructions = 1.33
  - Frequency of FPSQR = 2%
  - CPI of FPSQR = 20
- Assume that the two design alternatives are:
  - to decrease the CPI of FPSQR to 2 or
  - to decrease the average CPI of all FP operations to 2.5.
- Compare these two design alternatives using the processor performance equation

Example

- Observe that only the CPI changes
- The clock rate and instruction count remain identical
- We start by finding the original CPI with no enhancement:

$$\text{CPI}_{\text{original}} = \sum_{i=1}^{n} \text{CPI}_i \times \frac{\text{IC}_i}{\text{Instruction count}} =$$

$$= (4 \times 25\%) + (1.33 \times 75\%) = 2.0$$

- We can compute the CPI for the enhanced FPSR by subtracting the cycles saved from the original CPI:

$$\text{CPI}_{\text{new FPSR}} = \text{CPI}_{\text{original}} - 2\% \times (\text{CPI}_{\text{old FPSR}} - \text{CPI}_{\text{new FPSR only}}) =$$

$$= 2 - 2\% \times (20 - 2) = 1.64$$

Example

- We can compute the CPI for the enhancement of all FP instructions (the same way or) by summing the FP and non-FP CPIs:

$$CPI_{newFP} = (2.5 \times 25\%) + (1.33 \times 75\%) = 1.625$$

- Since the CPI of the overall FP enhancement is slightly lower, its performance will be marginally better
- The speedup for the overall FP enhancement is

$$Speedup_{newFP} = \frac{CPU\,time_{original}}{CPU\,time_{newFP}} = \frac{IC \times Clock\,cycle \times CPI_{original}}{IC \times Clock\,cycle \times CPI_{newFP}} =$$

$$= \frac{CPI_{original}}{CPI_{newFP}} = \frac{2.0}{1.625} = 1.23$$

Conclusions

- It is often **easier** to use the processor performance equation than Amdahl's law
- In fact,
  - It is often possible to **measure the constituent parts of the processor performance equation**
  - It may be **difficult** to measure things such as the **fraction of execution time** for which a set of instructions is responsible
  - In practice, this would probably be computed by summing the product of the instruction count and the CPI for each of the instructions in the set
- Hence, the *starting point is often individual instruction count and CPI measurements* → **performance equation**