Computer Design – A Quantitative Approach

Shweta Ohri

Jagannath International Management School,

Vasant Kunj,

New Delhi-70

Abstract

In this paper we discuss the quantitative approach of computer architecture. Also, we discuss parallelism, scalability, principle of locality and Amdahl's Law in the context of quantitative measurement. Furthermore, we demystify computer architecture through an emphasis on cost-performance-power trade-offs and good engineering design. We believe that the field has continued to mature and move toward the rigorous quantitative foundation of long-established scientific and engineering disciplines in the context of computer architecture.

Keywords:
Parallelism, Scalability, Principle of Locality, Amdahl's Law

Introduction

Quantitative approach has become the mainstay of computer architecture research. However, the tremendous complexity of computer systems is making them both difficult to reason about and expensive to develop. Detailed software analysis has therefore become essential for evaluating ideas in the computer architecture field. Industry uses quantitative approach extensively during processor and system design because it is the easiest and least expensive way to explore design options. Also, it is even more important in research to evaluate radical new ideas and characterize the nature of the design space.

In the remaining of this paper, there are four more sections. In Section 2, we provide a brief idea about parallelism and scalability. Also, the principle of locality and Amdahl's Law are discussed in section 3 and 4. Furthermore, we discuss the processor performance equation in section 5. Finally, conclusions and references are provided in Section 6.

1. Parallelism

Parallelism allows a set of processor to work helpfully to solve a computational problem. This concept is extensive adequate to include parallel supercomputers that have hundreds or thousands of processors, networks of workstations, multiple-processor workstations, and embedded systems. It is very are interesting because it offers the potential to concentrate computational resources---whether processors, memory, or I/O bandwidth---on important computational problems. In the context of quantitative approach, parallelism has sometimes been viewed as a rare and exotic subarea of computing, interesting but of little relevance to the average programmer. A study of trends in applications, computer architecture, and networking shows that this view is no longer tenable. Parallelism is becoming ubiquitous, and parallel programming is becoming central to the programming enterprise.

Taking advantage of parallelism is one of the most important methods for improving performance. Every chapter in this book has an example of how performance is enhanced through the exploitation of parallelism. We give three brief examples, which are expounded on in later chapters. Our first example is the use of parallelism at the system level. To improve the throughput performance on a typical server benchmark, such as SPEC Web or TPC-C, multiple processors and multiple disks can be used. The workload of handling requests can then be spread among the processors and disks, resulting in improved throughput. Being able to expand memory and the number of processors and disks is called *scalability*, and it is a valuable asset for servers. At the level of an individual processor, taking advantage of parallelism among instructions is critical to achieving high performance. One of the simplest ways to do this is through pipelining. The basic idea behind pipelining is to overlap instruction execution to reduce the total time to complete an instruction sequence.

2. Principle of Locality

Important fundamental observations have come from properties of programs. The most important program property that we regularly exploit is the *principle of locality:* Programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past. The principle of locality also applies to data accesses, though not as strongly as to code accesses. Two different types of locality have been observed. *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.

Computer pioneers correctly predicted that programmers would want unlimited amounts of fast memory. An economical solution to that desire is a *memory hierarchy,* which takes advantage of locality and cost-performance of memory technologies. The *principle of locality,* presented in the first chapter, says that most programs do not access all code or data uniformly. Locality occurs in time (temporal *locality*) and in space (*spatial locality*). This principle, plus the guideline that smaller hardware can be made faster, led to hierarchies based on memories of different speeds and sizes. Figure 5.1 shows a multilevel memory hierarchy, including typical sizes and speeds of access. Since fast memory is expensive, a memory hierarchy is organized into several levels—each smaller, faster, and more expensive per byte than the next lower level. The goal is to provide a memory system with cost per byte almost as low as the cheapest level of memory and speed almost as fast as the fastest level. Note that each level maps addresses from a slower, larger memory to a smaller but faster memory higher in the hierarchy. As part of address mapping, the memory hierarchy is given the responsibility of address checking; hence, protection schemes for scrutinizing addresses are also part of the memory hierarchy. The importance of the memory hierarchy has increased with advances in performance of processors. Figure 5.2 plots processor performance projections against the historical performance improvement in time to access main memory. Clearly, computer architects must try to close the processor-memory gap. The increasing size and thus importance of this gap led to the migration of the basics of memory hierarchy into undergraduate courses in computer architecture, and even to courses in operating systems and compilers. Thus, we'll start with a quick review of caches. The bulk of the chapter, however, describes more advanced innovations that address the processor-memory performance gap. When a word is not found in the cache, the word must be fetched from the memory and placed in the cache before continuing. Multiple words, called a *block* (or *line*), are moved for efficiency reasons.

3. Amdahl's Law

The performance gain that can be obtained by improving some portion of a computer can be calculated using Amdahl's Law. Amdahl's Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used. Amdahl's Law defines the *speedup* that can be gained by using a particular feature. What is speedup? Suppose that we can make an enhancement to a computer that will improve performance when it is used. Speedup is the ratio

$$Speedup = \frac{Performance\ for\ entire\ task\ using\ the\ enhancement\ when\ possible}{Performance\ for\ entire\ task\ without\ using\ the\ enhancement}$$

Alternatively,

$$Speedup = \frac{Execution\ time\ for\ entire\ task\ without\ using\ the\ enhancement}{Execution\ time\ for\ entire\ task\ using\ the\ enhancement\ when\ possible}$$

Speedup tells us how much faster a task will run using the computer with the enhancement as opposed to the original computer. Amdahl's Law gives us a quick way to find the speedup from some enhancement, which depends on two factors:

4.1 *The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement*—For example, if 20 seconds of the execution time of a program that takes 60 seconds in total can use an enhancement, the fraction is 20/60. This value, which we will call Fraction enhanced, is always less than or equal to 1.

4.2 *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*— This value is the time of the original mode over the time of the enhanced mode. If the enhanced mode takes, say, 2 seconds for a portion of the program, while it is 5 seconds in the original mode, the improvement is 5/2. We will call this value, which is always greater than 1, Speedup enhanced. 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:

$$Execution\ time_{new} = Execution\ time_{old} \times \left((1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}\right)$$

The overall speedup is the ratio of the execution times:

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

4. The Processor Performance Equation

Essentially all computers are constructed using a clock running at a constant rate. These 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:

$$CPU\ time = CPU\ clock\ cycles\ for\ a\ program\ \times Clock\ cycle\ time$$

$$CPU\ time = \frac{CPU\ clock\ cycles\ for\ a\ program}{Clcok\ rate}$$

In addition to the number of clock cycles needed to execute a program, 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). Because it is easier to work with, and because we will deal with simple processors in this chapter, we use CPI. Designers sometimes also use *instructions per clock* (IPC), which is the inverse of CPI. CPI is computed as

$$CPI = \frac{CPU\ clock\ cycles\ for\ a\ program}{Instruction\ count}$$

This processor figure of merit provides insight into different styles of instruction sets and implementations.

By transposing instruction count in the above formula, clock cycles can be defined as

$$IC\ \times CPI$$

This allows us to use CPI in the execution time formula:

$$CPU\ time = Instruction\ count\ \times Cycles\ per\ instruction\ \times Clock\ cycly\ time$$

Expanding the first formula into the units of measurement shows how the pieces fit together:

$$\frac{Instructions}{Program} \times \frac{Clock\ cycles}{Instruction} \times \frac{Seconds}{Clock\ cycles} = \frac{Seconds}{Program} = CPU\ time$$

As this formula demonstrates, processor performance is dependent upon three characteristics: clock cycle (or rate), clock cycles per instruction, and instruction count. Furthermore, CPU time is *equally* dependent on these three characteristics:

- A 10% improvement in any one of them leads to a 10% improvement in CPU time.

Unfortunately, it is difficult to change one parameter in complete isolation from others because the basic technologies involved in changing each characteristic are interdependent:

- *Clock cycle time*—Hardware technology and organization
- *CPI*—Organization and instruction set architecture

- *Instruction count*—Instruction set architecture and compiler technology

Luckily, many potential performance improvement techniques primarily improve one component of processor performance with small or predictable impacts on the other two. Sometimes it is useful in designing the processor to calculate the number of total processor clock cycles as

$$CPU\ clock\ cycles = \sum_{i=1}^{n} IC_i \times CPI_i$$

Where, IC_i represents number of times instruction i is executed in a program and CPI_i *represents* the average number of clocks per instruction for instruction i. This form can be used to express CPU time as

$$CPU = \left(\sum_{i=1}^{n} IC_i \times CPI_i\right) \times Clock\ cycle\ time$$

5. Conclusions

In this paper we discussed parallelism, scalability, principle of locality and Amdahl's Law. Computer technology has made incredible progress in the roughly 60 years since the first general-purpose electronic computer was created. Today, less than $500 will purchase a personal computer that has more performance, more main memory, and more disk storage than a computer bought in 1985 for 1 million dollars. This rapid improvement has come both from advances in the technology used to build computers and from innovation in computer design.

Although technological improvements have been fairly steady, progress arising from better computer architectures has been much less consistent. During the first 25 years of electronic computers, both forces made a major contribution, delivering performance improvement of about 25% per year. The late 1970s saw the emergence of the microprocessor. The ability of the microprocessor to ride the improvements in integrated circuit technology led to a higher rate of improvement— roughly 35% growth per year in performance. Therefore, we can conclude that the quantitative analysis of computer design is mandatory to assess the performance.

.

References

[1] Patterson, D. A. and Hennessy L. J. (2007), "Computer Architecture – A Quantitative Approach", Mc Graw Hill.

[2] McMahon, F. M. [1986]. "The Livermore FORTRAN kernels: A computer test of numerical performance range," Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, Univ. of California, Livermore (December).

[3] Nairy, C., and D. Soltis [2003]. "Itanium 2 processor microarchitecture," IEEE Micro 23:2 (March–April), 44–55.

[4] Mead, C., and L. Conway [1980]. Introduction to VLSI Systems, Addison-Wesley, Reading, Mass.

[5] Mellor-Crummey, J. M., and M. L. Scott [1991]. "Algorithms for scalable synchronization on shared-memory multiprocessors," ACM Trans. on Computer Systems 9:1 (February), 21–65.

[6] Menabrea, L. F. [1842]. "Sketch of the analytical engine invented by Charles Babbage," Bibiothèque Universelle de Genève (October).

[7] Lam, M. S., E. E. Rothberg, and M. E. Wolf [1991]. "The cache performance and optimizations of blocked algorithms," Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, Santa Clara, Calif., April 8–11. SIGPLAN Notices 26:4 (April), 63–74.

[8] Jordan, H. F. [1983]. "Performance measurements on HEP—a pipelined MIMD computer," *Proc. 10th Int'l Symposium on Computer Architecture* (June), Stockholm, 207–212.

[9] Jordan, K. E. [1987]. "Performance comparison of large-scale scientific processors: Scalar mainframes, mainframes with vector facilities, and supercomputers," *Computer* 20:3 (March), 10–23.

[10] Jouppi, N. P. [1990]. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *Proc. 17th Annual Int'l Symposium on Computer Architecture*, 364–73.

[11] Jouppi, N. P. [1998]. "Retrospective: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *25 Years of the Int'l Symposia on Computer Architecture (Selected Papers)*, ACM, 71–73.

[12] Jouppi, N. P., and D. W. Wall [1989]. "Available instruction-level parallelism for superscalar and superpipelined processors," *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems,* IEEE/ACM (April), Boston,
272–282.

[13] Jouppi, N. P., and S. J. E. Wilton [1994]. "Trade-offs in two-level on-chip caching," *Proc. 21st Annual Int'l Symposium on Computer Architecture,* Chicago, April 18– 21, 34–45.

[14] Kaeli, D. R., and P. G. Emma [1991]. "Branch history table prediction of moving target branches due to subroutine returns," *Proc. 18th Int'l Symposium on Computer Architecture (ISCA),* Toronto, May, 34–42.

[15] Kahan, J. [1990]. "On the advantage of the 8087's stack," unpublished course notes, Computer Science Division, University of California at Berkeley.

[16] Kahan, W. [1968]. "7094-II system support for numerical analysis," *SHARE Secretarial Distribution* SSD-159.

[17] Kahaner, D. K. [1988]. "Benchmarks for 'real' programs," *SIAM News* (November).

[18] Kahn, R. E. [1972]. "Resource-sharing computer communication networks," *Proc. IEEE* 60:11 (November), 1397–1407.

[19] Kane, G. [1986]. *MIPS R2000 RISC Architecture,* Prentice Hall, Englewood Cliffs, N.J.

[20] Kane, G. [1996]. *PA-RISC 2.0 Architecture*, Prentice Hall PTR, Upper Saddle River, N.J.

[21] Kane, G., and J. Heinrich [1992]. *MIPS RISC Architecture,* Prentice Hall, Englewood Cliffs, N.J.

[22] Katz, R. H., D. A. Patterson, and G. A. Gibson [1989]. "Disk system architectures for high performance computing," *Proc. IEEE* 77:12 (December), 1842–1858.

[23] Keckler, S. W., and W. J. Dally [1992]. "Processor coupling: Integrating compile time and runtime scheduling for parallelism," *Proc. 19th Annual Int'l Symposium on Computer Architecture* (May), 202–213.

[24] Lam, M. S., and R. P. Wilson [1992]. "Limits of control flow on parallelism," Proc. 19th Symposium on Computer Architecture (May), Gold Coast, Australia, 46–57.

[25] Lambright, D. [2000]. "Experiences in measuring the reliability of a cache-based
storage system," Proc. of First Workshop on Industrial Experiences with Systems Software (WIESS 2000), collocated with the 4th Symposium on Operating Systems Design and Implementation (OSDI), San Diego, Calif. (October 22).

[26] Lamport, L. [1979]. “How to make a multiprocessor computer that correctly executes multiprocess programs,” IEEE Trans. on Computers C-28:9 (September), 241–248.

[27] Laprie, J.-C. [1985]. “Dependable computing and fault tolerance: Concepts and terminology,” Fifteenth Annual Int’l Symposium on Fault-Tolerant Computing FTCS 15. Digest of Papers. Ann Arbor, Mich. (June 19–21), 2–11.

[28] Larson, E. R. [1973]. “Findings of fact, conclusions of law, and order for judgment,” File No. 4-67, Civ. 138, Honeywell v. Sperry-Rand and Illinois Scientific Development, U.S. District Court for the State of Minnesota, Fourth Division (October 19).

[29] Laudon, J., A. Gupta, and M. Horowitz [1994]. “Interleaving: A multithreading technique targeting multiprocessors and workstations,” Proc. Sixth Int’l Conf. on Architectural Support for Programming Languages and Operating Systems (October), Boston, 308–318.