Modeling and Analysis

An Introduction to System Performance Evaluation Methodology

HISASHI KOBAYASHI
IBM Corporation

ISBN 0-201-14457-3
ABCDEFGHIJK-HA-798

Foreword

The field of systems programming primarily grew out of the efforts of many programmers and managers whose creative energy went into producing practical, utilitarian systems programs needed by the rapidly growing computer industry. Programming was practiced as an art where each programmer invented his own solutions to problems with little guidance beyond that provided by his immediate associates. In 1968, the late Ascher Opler, then at IBM, recognized that it was necessary to bring programming knowledge together in a form that would be accessible to all systems programmers. Surveying the state of the art, he decided that enough useful material existed to justify a significant publication effort. On his recommendation, IBM decided to sponsor The Systems Programming Series as a long term project to collect, organize, and publish principles and techniques that would have lasting value throughout the industry.

The Series consists of an open-ended collection of text-reference books. The contents of each book represent the individual author's view of the subject area and do not necessarily reflect the views of the IBM Corporation. Each is organized for course use but is detailed enough for reference. Further, the Series is organized in three levels: broad introductory material in the foundation volumes, more specialized material in the software volumes, and very specialized theory in the computer science volumes. As such, the Series meets the needs of the novice, the experienced programmer, and the computer scientist.

The Editorial Board

Contents

CHAPTER 3
BASIC QUEUEING ANALYSIS

CHAPTER 4
THE SIMULATION METHOD

CHAPTER 5
DATA ANALYSIS

1

Computer Performance Evaluation: An Introduction

1.1 THE EVOLUTION OF COMPUTERS

The evolution of computers from their infancy three decades ago to their sophistication and pervasiveness in our society has been startling; it is certainly one of the fastest technological developments in human history. In this introductory section, we glance at major developments in the history of digital computers. It is customary to divide the era of computers into "generations" (Denning, 1971; Rosen, 1969). The first-generation machines (from approximately 1940 to 1950), represented by the ENIAC, used vacuum tubes for arithmetic operations; the time per operation ranged from 0.1 to 1.0 msec. Main-memory components consisted of electrostatic tubes and delay lines. These were augmented by auxiliary memory such as paper tapes, punched cards, and delay lines.

The second-generation (1950–1964) computer systems, such as the IBM 7040 and 7094, adopted transistor technology for logical operations: Their time per operation ranged from 1 to 10 microseconds. Magnetic drums and magnetic core appeared as main memory with access time also ranging from 1 to 10 μsec. Magnetic tapes, disks, and drums became available as auxiliary memory. The development of the first software systems—assemblers, relocatable loaders, and FORTRAN—made significant impacts on the use of computers. Floating-point arithmetic, interrupt facilities, and special-purpose I/O equipments were developed, and software services, such as subroutine libraries, batch monitors, and I/O control routines, enhanced the efficiency of digital computers.

With the introduction of the IBM System/360 and CDC 6600, the era of third-generation computer systems began. Integrated circuits were used in the CPU: These achieved speeds on the order of 0.1 to 1.0 μsec per operation. Storage capacity also expanded, in both main memory and auxiliary memory. But at least an equally important advance in the third-generation computing systems was made in software, that is, in the introduction of powerful and sophisticated *operating systems*. An operating system is a collection of computer system software that is responsible for allocating and controlling the use of the hardware, program, and data resources. The operating systems were designed to affect the capability of allowing several programs (or tasks) to run simultaneously by sharing resources. The sharing is, of course, motivated by the effective use of expensive computer resources, thereby increasing the system's productivity and users' satisfaction. The operating systems were also developed to relieve users and programmers of the detailed and tedious tasks involved with the system operation, such as converting data to the formats required by the hardware of the various devices. An operating system called OS/360 was designed to serve all the IBM System/360 models for a variety of applications.

In the late 1960s and 1970s, the third-generation computer systems moved into what we may call the "late third" generation or what some people designate as the "fourth" generation. These systems are characterized by faster machines utilizing more advanced LSI (large-scale integration) technology and semiconductor memory. The advances in storage technology are most noteworthy in this phase of computer system evolution. Additional memory devices, including flexible media and magnetic recording, have been developed over a broad range of accessing speeds and costs-per-unit capacity. An optimum combination of these technologies from the viewpoint of performance-cost tradeoffs has resulted in storage structures of many levels, collectively referred to as *storage hierarchies* (or *memory hierarchies*). A typical storage hierarchy consists of *cache* (buffer storage interposed between the processing unit and main storage), main storage (also called the backing store), drum, disk, and tape storage, and possibly including on-line archival mass storage. The memory requirements of programs have often outpaced the growth of storage capacity, placing on users the great burden of allocating storage space within the storage hierarchy. Thus, highly automated procedures have been devised for the allocation of storage spaces to the individual tasks and for the transfer of pieces of the programs and data from one level to another in the hierarchy. Progress in conventional storage devices, both semiconductor and magnetic types, continues to make them even faster and cheaper. In the future, such devices as charge-coupled

devices, magnetic bubbles, beam-addressed optical storage, and holographic storage will further enlarge the list of available technologies.

Dynamic address translation, based on paging or segmentation (or combinations thereof), came into vogue in the mid-1960s as a major advancement in memory management. A storage hierarchy supported by such a memory management policy has come to be known as a *virtual storage* or *virtual memory*. The MULTICS (Multiplexed Information and Computing Service) system jointly developed by MIT and GE, and the IBM System/370 virtual-storage operating systems represent systems that employ the virtual storage concept. The architectural concept of virtual storage has been extended further into the notion of *virtual machines*, which can accommodate simultaneously several operating systems by dynamically sharing the resources of a single *real* machine. CP-67 (which was developed for the System/360 Model 67) and its successor, VM-370, are examples of operating systems that support multiple virtual machines.

Also in the late 1960s and early 1970s a number of *multiprocessing* systems came into existence. The wider use of multiprocessing was spurred on by economies of scale in the production of processors, by the flexibility to obtain the desired processing power through the addition of processing units, and by the prospect of uninterrupted (although degraded) performance in the event of failure. Parallel computers, such as the IBM 2938 array processor, the ILLIAC IV, the CDC STAR-100, and the Texas Instruments ASC (Advanced Scientific Computer), have evolved as alternative architecture for faster computation in specific environments.

Recent advances in computer architecture and hardware/software technologies have not only introduced a number of new computer applications, but have also impacted the ways in which information processing takes place. On-line teleprocessing and interactive use of a system with a large *data base* is now widespread. As the emphasis in the information processing industry shifts from the conventional mathematical computations to information management, the data base has become increasingly central to an overall system design.

The growth of computer applications and the effort for improved human/machine interface are stimulating the development of new peripheral devices and data entry technologies. The fast growth of information processing and management has also led to the development of computer-communication networks, as represented by the ARPANET (Advanced Research Projects Agency Network) and its descendants. We can regard *networking* as an extension of the resource-sharing concept exploited in the multiprogrammed and time-shared systems: Resources of host computers (their computing powers, data bases, and functions) at

geographically distributed locations are accessible to a program or user through terrestrial, radio, and satellite links. The bursty nature of message traffic from computers or user terminals has required new ways of sharing or *multiplexing* transmission links. The combination of time-division multiplexing with packet-switching techniques is a notable example. In parallel with the continuing growth of large-scale general-purpose computing systems, *minicomputers* and, more recently, *microcomputers* (or *microprocessors*) have emerged, creating new applications of information management and processing.

1.2 THE ROLE OF PERFORMANCE EVALUATION AND PREDICTION

The issue of performance evaluation and prediction has concerned users throughout the history of computer evolution. In fact, as in any other technological development, the issue is most acute when the technology is young; the persistent pursuit of products with improved cost-performance characteristics then constantly leads to designs with untried and uncertain features.

The need for computer performance evaluation and prediction exists from the initial conception of a system's architectural design to its daily operation after installation. In the early planning phase of a new computer system product, the manufacturer usually must make two types of *prediction*. The first type is to forecast the nature of applications and the levels of "system workloads" of these applications. Here the term *workload* means, informally, the amount of service requirements placed on the system. We shall elaborate more on workload characterization in Section 1.4. The second type of prediction is concerned with the choice between architectural design alternatives, based on hardware and software technologies that will be available in the development period of the planned system. Here the criterion of selection is what we call "cost-performance tradeoff." The accuracy of such prediction rests to a considerable extent on our capability of mapping the performance characteristics of the system components into the overall system-level performance characteristics. Such translation procedures are by no means straightforward or well established.

Once the architectural decisions have been made and the system design and implementation started, the scope of performance prediction and evaluation becomes more specific. What is the best choice of machine organization? What is the operating system to support and what are the functions it should provide? The interactions among the operating system components—algorithms for *job scheduling*, *processor scheduling*, and

storage management—must be understood, and their effects on the performance must be predicted. The techniques used for performance evaluation and prediction during the design and implementation phases range from simple hand calculation to quite elaborate simulation. Comparing the predicted performance with the actual achieved performance often reveals major defects in the design or errors in the system programming. It is now a widely accepted belief that the performance prediction and evaluation process should be an integral part of the development efforts throughout the design and implementation activities.

After a new product is developed, the computer manufacturer must be ready to predict the performance for specific applications and requirements of potential buyers. The manufacturer must propose an optimal combination and organization of its hardware and software products to offer the best solution to a customer's requirements. This activity is often referred to as the *configuration process*. Although the performance prediction and evaluation tools and methodologies that are utilized during the system development phase can be used for the purpose of configuration, there is an additional factor required in this effort. The projected user environment must be translated into a set of quantitative parameters that can be used as inputs to a performance prediction model. This is the workload characterization problem again, but more precision is called for in this instance.

When a product system is installed at a customer's site, the computer vendor or service company must see that the system realizes its full potential and meets the promised performance level. Such *system tuning* activities were (and quite frequently still are) traditionally based on intuition and experience. The complexity and sophistication of contemporary large-scale computers are such that the globally optimum and stable operating point can no longer be easily found by mere intuition or trial-and-error procedures. The system tuning requires a clear understanding of the complex interactions among the individual system components. A systematic procedure of performing this task is yet to be developed, and awaits our continuing research and development efforts.

1.3 PERFORMANCE MEASURES

In the previous section, we frequently used the term performance without precisely defining it. In this section, we clarify the term so that we have a well-understood common ground on which to develop the discussions of the following chapters.

When we say that "the performance of this computer is great," it means perhaps that the quality of service delivered by the system exceeds

our expectation. But "the measure of service quality" and "the extent of expectation" vary depending on the individuals involved, be they system designers, installation managers, or terminal users. If we attempt to measure the quality of computer performance in the broadest context, we must consider such issues as user response (as well as the system response), ease of use, reliability, user's productivity, and the like as integral parts of the system's performance. Such discussions, however, fall within the realm of *nonquantitative* sciences that involve social and behavioral sciences. Despite our full awareness that performance analysts cannot avoid what are ultimately behavioral questions, the scope of this text is quite limited: We discuss the computer performance only in terms of *clearly measurable* quantities. This is done with the same spirit with which we conventionally define, for instance, the signal-to-noise ratio and the probability of decoding errors as measures of performance of communication systems.

There is certainly more than one choice for the measure of performance. We can classify performance measures into two categories: user-oriented measures and system-oriented measures. The user-oriented measures include such quantities as the *turnaround time* in a batch-system environment and the *response time* in a real-time and/or interactive-system environment. The turnaround time of a job is the length of time that elapses from the submission of the job until the availability of its processed result. Similarly, in an interactive environment, the response time of a request represents the interval that elapses from the arrival of the request until its completion at the system. There are several variants of the response time measure in common use, due to differing definitions of the moment of the request arrival and the moment of completion. For instance, we may define the arrival time as the moment when the user pushes the RETURN key at his or her terminal; the completion time may be the moment when the first line from the system output is typed out at the terminal. When terms such as response time are used in making performance comparisons, they should be accompanied by unambiguous definitions.

In interactive systems, we sometimes use the term *system reaction time*, which is the interval of time that elapses from the moment an input arrives in the system until it receives its first *time slice* of service. It measures how effective a scheduler is in dispatching service to a newly arrived input. Turnaround time, response time, and reaction time are all considered *random variables;* hence, we can talk about their *distributions, expected values, variances,* and the like.*

* See Section 2.4.1 for the formal definitions of random variables and these related terms.

Usually we categorize jobs or requests in several different priority classes and assign to the individual job the *priority* value of its class. Many factors may determine the assignment of a priority to a job: the job's urgency, its importance, and its resource-demand characteristics. We often define and compute the turnaround time or response time separately for different job classes.

The system-oriented (or installation-oriented) measures are typically *throughput* and *utilization*. Throughput is defined as the average number of jobs processed per unit time. It measures the degree of productivity that the system can provide. If jobs arrive at a system according to some arrival mechanism that is *independent* of the state of the system, throughput is equivalent to the average arrival rate, provided that the system can complete the jobs without creating an ever-increasing backlog. But in this case throughput is not an adequate measure of performance; rather, it is a measure of system *workload*.

The notion of throughput makes sense when either (1) there is always some work awaiting the system's service, or (2) the job arrival mechanism depends on the system state. Case (1), in the context of queueing theory, means that the system is unstable in the sense that the queue, or backlog, will grow without bound. In practice, however, we may define and measure throughput over a finite interval in which the input queue is never empty. Throughput thus defined is a proper indicator of a system's *capacity*. Case (2) applies when we assume a finite number of job generation sources. Suppose that, in an interactive system, there is a finite number N of terminal users actually logged on. Assume further that a terminal is *blocked* while its request is in the system, either waiting for or receiving service. If there are n jobs in the system, only the remaining $N-n$ terminals are eligible for generating requests. Thus, the effective arrival rate is a (linearly) decreasing function of the system state, n.* We can envision a similar situation in a batch-system environment: There may be a sufficiently large number of users to keep the system continually busy. In reality, however, as the system congestion level increases, a user may be discouraged from submitting a new job. Again, the job arrival rate will be some decreasing function of the number of outstanding jobs. This *negative feedback* loop inherent in the job generation mechanism makes the system always stable.

The *utilization* of a resource is the fraction of time that the particular resource is busy. The CPU utilization is the most popular measure of system usage, although it is not necessarily the most important in complex systems. When the CPU is not idle, it may be in either of two busy states:

* See Sections 3.9 and 3.10 for a full description of such models.

the *problem program state* (or simply the *problem state*) and the *supervisory program state* (or the *supervisor state*). The former represents the portion of time when the CPU is actually executing the programs written or called by the users; the latter is the time consumed in executing such operating system components as the scheduler and various interrupt-handling routines. The distinction is commonly assumed to be synonymous with that of "useful work" versus "overhead." Yet it must be noted that much of the supervisor-state operation provides necessary and useful service for the user programs; hence the "overhead" categorization may be misleading.

If we assume a system with a single CPU, and if the CPU utilization figure excludes the supervisor state, then we find the following simple relationship between throughput λ (jobs per second) and the CPU utilization ρ_{CPU}:

$$\rho_{CPU} = \lambda \bar{S}_{CPU}, \tag{1.1}$$

where $\bar{S}_{CPU}$ (seconds per job) represents the average CPU time required to process a job.

The mean response time, which we denote by $\bar{T}$, is found to have the simple relation with throughput

$$\lambda \bar{T} = \bar{n}, \tag{1.2}$$

in which $\bar{n}$ represents the average number of jobs (waiting or being served) in the system. Both Formulas (1.1) and (1.2) are special cases of the formula $L = \lambda W$ that appear frequently in a more general context. (See Section 3.6 for details of this formula.)

1.4 WORKLOAD CHARACTERIZATION AND PERFORMANCE EVALUATION TECHNIQUES

As we stated earlier, by the term workload we mean the amount of *service demands* imposed on the system by a set of jobs in a given application. The job arrival rate, or more generally the job arrival mechanism, is certainly one factor that determines the system's workload. For example, the Poisson arrival assumption, which we shall discuss in Section 3.3, can be regarded as a part of the *workload model*. We remarked in the previous section that the job arrival process should depend on the degree of the system's congestion. Thus, the system's workload and the system's performance are not independent of each other.

The job arrival model is only a part of the workload characterization: We must also represent the work demands brought in by the individual

jobs. Since a computer is an entity consisting of multiple resource components, the work demands of a job must be represented by at least (1) the CPU work demand, (2) memory space demand, (3) I/O (input/output) work demand, and (4) demands on software components. Thus, we must translate a given application environment into a set of work demands seen by system resource components. This translation procedure was a relatively simple matter in the early days when the CPU was the major critical resource in the system and each job was sequentially executed in isolation. In the following discussion we first review the conventional methods of workload characterization and performance evaluation: specifically, instruction mixes, kernel programs, and benchmarks.

Instruction mixes

One popular method of representing the workloads on the CPU is the *instruction mix*. By an instruction mix we mean the *distribution of relative frequencies* $\{f_i\}$ of instruction types (for the given instruction set) observed in a typical application environment of the system in question. An instruction mix is obtained by running a set of representative programs and counting the number of occurrences of individual instruction types—Add/Subtract, Multiply, Divide, Load, Store, Shift, branch operations, and the like. Since the instruction execution times t_i of each instruction type i are known for a given CPU, we can calculate τ, the average execution time per operation,

$$\tau = \sum_i f_i t_i, \tag{1.3}$$

where the sum is taken over all distinct instruction types i. If we choose the microsecond (μsec) as the unit of time $\{t_i\}$, then the quantity $1/\tau$ represents the speed of the CPU measured in MIPS (Million Instructions Per Second).

The instruction mix provides valuable information for the design and implementation of a processor: When the design of the CPU adopts microprogramming, the instruction mix gives a good indication whether to emphasize the performance efficiency or minimize the space occupancy of a particular instruction. There are, however, a number of shortcomings and limitations of the instruction mix. First of all, there is a question of the "representativeness" of the chosen instruction mix. Since it is expensive in machine time to produce a new instruction mix figure, we often rely on the relative frequencies $\{f_i\}$ measured by others. Even if we have program tracing facilities and can afford the machine time, the matter of selecting a set of representative programs is not trivial.

Second, the instruction mix $\{f_i\}$ is the first-order statistic; it completely lacks information on serial dependency and instruction overlap. In

a system with buffer store (cache) or pipelined CPU, the effective execution speed depends on the sequence pattern of the instruction and data references, not just the relative frequency. Furthermore, because the instruction mix does not include demand for other system resources, such as I/O devices, it cannot be used for overall system performance evaluation.

Kernel programs

The second method used to represent loads on a CPU is the *kernel program method.* A kernel program is a small program segment that represents the inner loop of a frequently used program. For instance, in scientific applications, a matrix inversion routine and a differential equation solution program may be selected as kernel programs. A payroll program is an example of a business application. A kernel program is useful in the early stages of a CPU design for the evaluation of different instruction sets and different architectural ideas. This is because the kernel preserves such important information as the sequence of instructions and the relative positions of branch instructions. Kernels also allow evaluation of some software components. For instance, the quality of a compiler can be evaluated by programming and compiling the kernel and examining the object code.

Although kernel programs contain significantly richer information than instruction mixes, there are still a number of limitations on their use. First, a performance analysis based on kernels requires coding the kernel program for the machine in question, using the specific instruction set. Thus, any such study is usually done using only a small number of kernel programs. Then the question of "representativeness" will be a more critical issue here than with the instruction mix method, in which measurement over a wide range of programs is relatively easy, although expensive in terms of computer time. Second, although a kernel program allows the performance evaluation of CPU/main-memory usage, it does not normally include adequate information on I/O operations. Thus, kernel programs are not adequate for the evaluation of multiprogramming systems.

Benchmarks

A *benchmark* is a complete program that is written in a high-level language and is considered to be representative of a given class of application programs. Sort programs and file updating programs are popular examples of benchmarks. By running mixed benchmarks on the system under evaluation, the performance of the entire system in realistic circumstances can be estimated. The quality of compilers can also be