An approach to job pricing in a multi-programming environment

by CHARLES B. KREITZBERG and JESSE H. WEBB

Educational Testing Service
Princeton, New Jersey

INTRODUCTION

Computers are amazingly fast, amazingly accurate, and amazingly expensive. This last attribute, expense, is one which must be considered by those who would utilize the speed and accuracy of computers. In order to equitably distribute the expense of computing among the various users, it is essential that the computer installation management be able to accurately assess the costs of processing a specific job. Knowing job costs is also important for efficiency studies, hardware planning, and workload evaluation as well as for billing purposes.

For a second generation computer installation, job billing was a relatively simple task; since in this environment, any job that was in execution in the machine had the total machine assigned to it for the entire period of execution. As a result, the billing algorithm could be based simply upon the elapsed time for the job and the cost of the machine being used. In most cases, the cost for a job was given simply as the product of the run time and the rate per unit time. While this algorithm was a very simple one, it nevertheless was an equitable one and in most cases a reproducible one.

Because of the fact that in a second generation computer only one job could be resident and in execution at one time, the very fast CPUs were often under utilized. As the CPUs were designed to be even faster, the degree of under utilization of them increased dramatically. Consequently, a major goal of third generation operating systems was to optimize the utilization of the CPU by allowing multiple jobs to be resident concurrently so that when any one job was in a wait state, the CPU could then be allocated to some other job that could make use of it. While multi-programming enabled a higher utilization of the CPU, it also introduced new problems in job billing. No longer was the old simple algorithm sufficient to equitably charge for the running of jobs. The two major reasons for this are:

- The sharing of resources by the resident jobs, and
- The variation in elapsed time from run to run of a given job.

Unlike the second generation computer a given job no longer has all of the resources that are available on the computer allocated to it. In a multi-programming computer, a job will be allocated only those resources that it requests in order to run. Additional resources, that are available on the computer, can be allocated to other jobs. Therefore, it is evident that the rate per unit time cannot be a constant for all jobs, as it was for second generation computer billing, but must in some sense be dependent upon the extent to which resources are allocated to the jobs.

The second item, and perhaps the most well-known, that influences the design of a billing algorithm for a third generation computer is the variation that is often experienced in the elapsed time from run to run of a given job. The elapsed time for any given job is no longer a function only of that job, but is also a function of the job mix. In other words, the elapsed time for a job will vary depending upon the kinds and numbers of different jobs which are resident with it when it is run.

In order to demonstrate the magnitude of variation that can be experienced with subsequent runs of a given job, one job was run five different times in various job mixes. The elapsed time varied from 288 seconds to 1,022 seconds. This is not an unusual case, but represents exactly what can happen to the elapsed time when running jobs in a multi-programming environment. The effect, of course, is exaggerated as the degree of multi-programming increases.

Not only can this variation in run time cause a difference in the cost of a job from one run to another,
but it also can cause an inequity in the cost of different jobs; the variation in run time can effectively cause one job to be more expensive than another even though the amount of work being done is less.

Objectives

We have isolated several important criteria to be met by a multi-programming billing algorithm. Briefly, these criteria are as follows.

- Reproducibility—As our previous discussion has indicated, the billing on elapsed time does not provide for reproducibility of charges. Any algorithm that is designed to be used in a multi-programming environment should have as a characteristic, the ability to produce reproducible charges for any given job regardless of when or how it is run, or what jobs it is sharing with the computer.

- Equitability—Any billing algorithm designed for use in a multi-programming environment must produce equitable costs. The cost of a given job must be a function only of the work that the job does, and of the amount of resources that it uses. Jobs which use more resources or do more work must pay more money. The billing algorithm must accommodate this fact.

- Cost Recovery—in many computer operations it is necessary to recover the cost of the operation from the users of the hardware. The billing algorithm developed for a multi-programming environment must enable the recovery of costs to be achieved.

- Auditability—A multi-programming billing algorithm must produce auditible costs. This is particularly true when billing outside users for the use of computer hardware. The charges to the client must be auditible.

- Encourage Efficient Use of the Hardware—Since one goal in a design of the third generation hardware was to optimize the use of that hardware, a billing algorithm that is designed for use in a multi-jobbing environment should be such that it encourages the efficient use of the hardware.

- Allow for Cost Estimating—The implementation of potential computer applications is often decided upon by making cost estimates of the expense of running the proposed application. Consequently, it is important that the billing algorithm used to charge customers for the use of the hardware also enables potential customers to estimate beforehand, the expense that they will incur when running their application on the computer hardware.

We distinguish between job cost and job price: job cost is the amount which it costs the installation to process a given job; job price is the amount that a user of the computer facility pays for having his job processed. Ideally, the job price will be based on the job cost but this may not always be the case. In many organizations, notably universities, the computer charges are absorbed by the institution as overhead; in these installations the job price is effectively zero—the job costs are not. In other organizations, such as service bureaus, the job price may be adjusted to attract clients and may not accurately reflect the job cost. In either case, however, it is important that the installation management know how much it costs to process a specific job.¹²

The development of the job billing algorithm (JBA) discussed in this paper will proceed as follows: first, we will discuss the "traditional" costing formula used in second generation computer systems:

\[
\text{cost} = (\text{program run time}) \times (\text{rate per unit time})
\]

and we shall demonstrate its inadequacy in a multi-jobbing environment. Second, we shall develop a cost formula in which a job is considered to run on a dedicated computer (which is, in fact, a subset of the multi-programming computer) in a time interval developed from the active time of the program.

DEVELOPMENT OF THE JOB PRICING ALGORITHM

In order to recover the cost of a sharable facility over a group of users, the price, \( P \), of performing some operation requiring usage \( t \) is:

\[
P = (C) \left( \frac{t_k}{\sum t_i} \right)
\]

where: \( C \) is the total cost of the facility

\( \sum t_i \) is the total usage experienced

\( t_k \) is the amount of use required for the operation

Consider the billing technique which was used by many computer installations running a single thread (one program at a time) system. Let \( \$m \) be the cost per unit time of the computer configuration. Then, if a program began execution at time \( t_1 \) and terminated execution at time \( t_2 \), the cost of running the program was computed by:

\[
P = \$m (t_2 - t_1)
\]

As the utilization of the computer increased the cost per unit time decreased.

The cost figure produced by (2) is in many ways a
very satisfying one. It is simple to compute, it is reproducible since a program normally requires a fixed time for its execution, it is equitable since a "large" job will cost more than a "small" job (where size is measured by the amount of time that the computers resources are held by the job). Unfortunately for the user, however, the cost produced by (2) charges for all the resources of the computer system even if they are unused.

This "inflated" charge is a result of the fact that, in a single thread environment, all resources of the computer system are allocated to a program being processed even if that program has no need of them. The effect of this is that the most efficient program in a single thread environment is the program which executes in the least amount of time; that is, programmers attempt to minimize the quantity \((t_2 - t_1)\); this quantity, called the \textit{wall clock time} (WCT) of the program, determines the program's cost.

Since the rate of the computer is constant, the only way to minimize the cost for a given program is to reduce its WCT; in effect, make it run faster. Hence, many of the techniques which were utilized during the second generation were designed to minimize the time that a program remained resident in the computer.

The purpose of running in a multi-thread environment, one in which more than the one program is resident concurrently, is to maximize the utilization of the computer's resources thus reducing the unit cost.

In a multi-thread processing system, the cost formula given by (2) is no longer useful because:

1. It is unreasonable to charge the user for the entire computer since the unused resources are available to other programs.
2. The wall clock time of a program is no longer a constant quantity but becomes a function of the operating environment and job mix.

For these reasons we must abandon (2) as a reasonable costing formula. Many pricing algorithms are in use; however, none is as "nice" as (2). If possible, we should like to retain formula (2) for its simplicity and intuitive appeal.\textsuperscript{3} This may be done if we can find more consistent definitions to replace \(m\) (rate) and WCT (elapsed time).

\textit{Computed elapsed time}

A computer program is a realization of some process on a particular hardware configuration. That is, a program uses some subset of the available resources and "tailors" them to perform a specific task. The program is loaded into the computer's memory at time \(t_1\) and

\[
WCT = t_4 - t_1 = \Sigma t_i + \Sigma t_v, \tag{3}
\]

where:

- each \(t_c\) is a compute interval
- each \(t_v\) is a voluntary wait interval.

graphically, the situation is represented as in Figure 1. The solid line represents periods of compute and the broken line indicates intervals of input/output activity. Since \(\Sigma t_c\) is based on the number of instructions executed which is constant and \(\Sigma t_v\) is based on the speed of input/output which is also constant (except for a few special cases), WCT is itself constant for a given program and data in a single thread environment. The ideal case in this type of system is one in which the overlap is so good that the program obtains the \(i+1\)st record as soon as it has finished processing the \(i\)th

Figure 1—States of a program in a single thread environment terminates at time \(t_2\). During the period of residency, the program may be in either one of two states: \textit{active} or \textit{blocked}. A program is \textit{active} when it is executing or when it awaiting the completion of some external event. A program is \textit{blocked} when it is waiting for some resource which is unavailable. These categories are exhaustive; if a program is not active and is not waiting for something to happen then it is not doing anything at all. The two categories are not, however, mutually exclusive since a program may be processing but also awaiting the completion of an event (for example an input/output operation) indeed, it is this condition which we attempt to maximize via channel overlap. Therefore, we define \textit{voluntary wait} as that interval during which a program has ceased computing and is awaiting the completion of an external event. We define \textit{involuntary wait} as the interval during which a program is blocked; a condition caused by contention.

In general, voluntary wait results from initiation of an input/output operation and in a single thread system we have:

\[\text{WCT} = t_4 - t_1 = \Sigma t_c + \Sigma t_v, \tag{3}\]

where:

- each \(t_c\) is a compute interval
- each \(t_v\) is a voluntary wait interval.

Figure 2—A program with maximum overlap
From the collection of the Computer History Museum (www.computerhistory.org)

record. Graphically, this situation is shown in Figure 2 and we can derive the lower bound on WCT as:

$$WCT \geq \sum t_i$$  \hspace{1cm} (4)$$

and, of course:

$$WCT = \sum t_i$$ as $$\sum t_i \to 0$$  \hspace{1cm} (5)$$

In a multi-thread environment, we know that:

$$WCT = \sum t_i + \sum t_i$$ \hspace{1cm} (6)$$

where: $$t_i$$ is an interval of involuntary wait.

But, from the above discussion we know that $$\sum t_i + \sum t_i$$ is a constant for a given program, hence, the inconsistency in the WCT must come from $$t_i$$. This is precisely what our intuition tells us; that the residency time of a job will increase with the workload on the computer. Graphically, a program running in a multi-thread environment might appear as in Figure 3.

During the interval that a program is in involuntary wait, it is performing no actions (in fact, some programmers refer to a program in this state as "asleep"). As a consequence, we may "remove" the segments of time that the program is asleep from the graph for time does not exist to a program activity in involuntary wait. This permits us to construct a series of time sequences for the various programs resident in the computer; counting clock ticks only when the program is active. When we do this a graph such as Figure 4 becomes continuous with respect to the program (Figure 5).

Of course, the units on the x-axis in Figure 5 no longer represents real-time, they represent, instead, the active time of the program. We shall call the computed time interval computed elapsed time (CET) defined as:

$$\text{CET} = \sum t_i = WCT - \sum t_i$$ \hspace{1cm} (7)$$

and as $$t_i \to 0$$, CET $$\to$$ WCT so that we have the relationship:

$$WCT \geq \text{CET}$$ \hspace{1cm} (8)$$

The quantity $$WCT-CET$$ represents the interference from other jobs in the system and may be used as a measure of the degree of multi-programming.

Unfortunately, the CET suffers from the same deficiency as the WCT—it is not reproducible. The reason for this is that on a movable head direct access storage device contention exists for the head and the time for an access varies with the job mix. However, the CET may be estimated from its parameters. Recall that $$\text{CET} = \sum t_i + \sum t_i$$ The quantity $$\Sigma t_i$$ is computed from the number of instructions executed by the program and is an extremely stable parameter. The quantity $$\Sigma t_i$$ is based upon the number and type of accesses and is estimated as:

$$\Sigma t_i = a(i) n_i$$ \hspace{1cm} (9)$$

where $$a(i)$$ is a function which estimates the access time to the $$i$$th file and $$n_i$$ is the number of access to that file. The amount of time which a program waits for an input or output operation depends upon a number of factors. The time required to read a record is based upon the transfer rate of the input/output device, the number of bytes transferred, the latency time associated with the device (such as disk rotation, tape inter-record gap time, and disk arm movement). For example, a tape access on a device with a transfer rate of $$R_T$$ and a start-stop time of $$S_T$$ would require:

$$a(T) = S_T + R_T b$$ \hspace{1cm} (10)$$

seconds to transfer a record of $$b$$ bytes. Hence, for a file of $$n$$ records, we have a total input/output time of:

$$\sum_{i=1}^{n} (S_T + R_T b_i) = nS_T + R_T \Sigma b_i$$ \hspace{1cm} (11)$$

where $$\Sigma b_i$$ is the total number of bytes transferred. In practice $$\Sigma b_i = nB$$ where $$n$$ is the number of records and $$B$$ is the average blocksize. The term $$S_T$$ is, nominally, the start-stop time of the device. However, this term is also used to apply a correction to the theoretical record time. The reason is that while the CET will never be greater than the I/O time plus the CPU time, overlap may cause it to be less. This problem is mitigated by the fact that at most computer shops (certainly at ETS) almost all programs are written in high-level computer languages and, as a result, the job mix is homogeneous. A measure of overlap may be obtained by fitting various curves to historical data and choosing
the one which provides the best fit. In other words, pick the constants which provide the best estimate of the WCT.

It is important to remember that the CET function produces a time as its result. We are using program parameters such as accesses, CPU cycles, and tape mounts only because they enable us to predict the CET with a high degree of accuracy.

The original billing formula (2) which we wished to adapt to a multi-thread environment utilized a time multiplied by a dollar rate per unit time. The CET estimating function has provided us with a pseudo run time; we must now develop an appropriate dollar rate function.

In order to develop a charging rate function we consider the set of resources assigned to a program. In a multi-programming environment, the computer's resources are assigned to various programs at a given time. The resources are partitioned into subset computers each assigned to a program. The configuration of the subset computers is dynamic; therefore, the cost of a job is:

\[
\text{cost} = \sum_{i=1}^{n} \text{CET}_i \cdot r_i
\]  

where \( i \) is the allocation interval; that is, the interval between changes in the job's resources held by the job. \( \text{CET}_i \) is the CET accumulated during the \( i \)th interval. \( r_i \) is the rate charged for the subset computer with the configuration held by the program during interval \( i \).

The allocation interval for OS/360 is a job step.

The rate function

Some of the attributes which the charging rate function should have are:

- the rate for a subset computer should reflect the "size" of the subset computer allocated; a "large" computer should cost more than a "small" computer.
- the rate for a subset computer should include a correction factor based upon the probability that a job will be available to utilize the remaining resources.
- the sum of the charges over a given period must equal the monies to be recovered, for that period.

With these points in mind, we may create a rate function.

The elements of the resource pool may be classified as sharable resources and nonsharable resources. Tape drives, core memory, and unit record equipment are examples of nonsharable resources; disk units are an example of a sharable resource. While these categories are not always exact they are useful since we assume that allocation of a nonsharable resource is more significant than allocation of a sharable resource. At Educational Testing Service, it was determined that the most used nonsharable resources are core storage and tape drives. Therefore, it was logical to partition the computer into subset computers based upon the program's requirement for core and tapes. Tapes are allocated in increments of one; core is allocated in 2K blocks. Hence, there are (# tapes * available core/2,000) possible partitions.

For any given design partition, we would like to develop a rate which is proportional to the load which allocation places upon the resources pool. A single job may sometimes disable the entire computer. If, for example, a single program is using all of the available core storage then the unused devices are not available to any other program and should be charged for. On the other hand, if a single job is using all available tapes, other jobs may still be processed and the charge should be proportionately less.

The design proportion is the mechanism by which the total machine is effectively partitioned into sub-machines based upon the resources allocated to the submachines. A design proportion can be then assigned to any job based upon the resources it requires. The design proportion should have at least the following properties.

- The design proportion should range between the limits 0 and 1.
- The design proportion should reflect the proportion of the total resources that are allocated to the job.
- The design proportion should reflect, in some fashion, the proportion of the total resources that are not allocated to the job, but which the job prevents other jobs from using.

The design proportion proposed for the billing algorithm is based upon the probability that when the job is resident, some other job can still be run. The definition of this parameter is as stated below.

The design proportion of a job is equal to the probability that when the job is resident, another job will be encountered such that there are insufficient resources remaining to run it.

Since OS/360 allocates core in 2K blocks, the number of ways that programs can be written to occupy available core is equal to:

\[
N = \frac{C}{2}
\]  

where,
\[ N = \text{Number of ways that programs can be written} \]

\[ C = \text{Core available in Kilo-bytes} \]

In addition, if there are \( T \) tapes available on the hardware configuration then there are \( T + 1 \) different ways that programs can be written to utilize tapes. Therefore, the total number of ways that programs can be written to utilize core and tapes is given by the following equation,

\[ N = (C/2)(T+1) \]  (14)

where,

\[ N = \text{Total number of ways that programs can be written} \]

\[ C = \text{Core available in Kilo-bytes} \]

\[ T = \text{Number of tape drives available} \]

The design proportion for a given job can be alternatively defined as 1 minus “the probability that another job can be written to fit in the remaining resources.” This is shown as follows.

\[ D_p = 1.0 - \frac{[(C_A-C_U)/2](T_A-T_U+1)}{[(C_A)/2](T_A+1)} \]  (15)

where,

\[ D_p = \text{Design proportion for the job} \]

\[ C_A = \text{Core available in Kilo-bytes} \]

\[ C_U = \text{Core used by job} \]

\[ T_A = \text{Tape drives available on the computer} \]

\[ T_U = \text{Tape drives used by the job} \]

It is important to note that the sum of the design proportions of all jobs resident at one time can be greater than 1.0. For example, consider the following two jobs resident in a 10K, four tape machine.

Job #1: 6K; 1 Tape \( D_p = 17/25 \)
Job #2: 4K; 3 Tapes \( D_p = 19/25 \)

The sum of their design proportion is 36/25. This seems odd at first since the design proportion of a 10K; four tape job is 1.0. However, this can be shown to be a necessary and desirable property of the design proportion. To show that this is the case, it is necessary to consider the amount of work done and the total cost of the work for two or more jobs that use the total machine compared to the cost of the same amount of work done by a single job that uses the total machine. This analysis will not be covered here.

The design proportion function as defined herein is a theoretical function. It is based solely upon the theoretical possibility of finding jobs to occupy available resources. Clearly, the theoretical probability and the actual probability may be somewhat different. Consequently, a design proportion could be designed based upon the actual probabilities experienced in a particular installation. Such a probability function would change as the nature of the program library changed. The design proportion function described above would change only as the configuration of the hardware changed. Either technique is acceptable and the design proportion has the desired properties. That is, the design proportion increases as the resources used by the various jobs increase. However, it also reflects the resources that are denied to other jobs because of some one jobs’ residency. Consider the fact that when all of the core is used by a job, the tape drives are denied to other jobs. The design proportion in this case is 1.0 reflecting the fact that the job in effect has tied up all available resources even though they are not all used by the job itself.

While the design proportion function is simple, it has many desirable properties:

- It is continuous with respect to OS/360 allocation; all allocation partitions are available.
- It always moves in the right direction, that is, increasing the core requirement or tape requirement of the program, results in an increased proportion.
- It results in a proportion which may be multiplied by the rate for the total configuration to produce a dollar cost for the subset computer.
- It is simple to compute.

If it were determined that the required recovery could be obtained if the rate for the computer were set at $35 per CET minute, the price of a step is determined by the equation:

\[ P_{\text{step}} = ((35.0)D_p(\text{core, tapes})) \text{ (CET/60)} \]  (16)

and the price of a job (with \( n \) steps) is:

\[ P_{\text{job}} = \sum_{i=1}^{n} P_{\text{step}} \]  (17)

We have come full circle and returned to our “second generation” billing formula:

\[ \text{cost} = \text{rate} \cdot \text{time} \]

The key points in the development were:

- A multi-tasking computer system may be considered to be a collection of parallel processors by altering the time reference.
- The variation in time of a program run in a multi-
programmed environment is due to involuntary wait time.
- The computed elapsed time may be multiplied by a rate assigned to the subset computer and an equitable and reproducible cost developed.

IMPLEMENTATION OF THE JOB PRICING ALGORITHM

The Job Pricing Algorithm (JPA) is implemented under OS/360 Release 19.6. No changes to the operating system were required; a relatively minor modification was made to HASP in order to write accounting records to disk for inclusion in the accounting system. The basis of the JPA is the IBM machine accounting facility known as Systems Management Facility (SMF).4

Billing under the JPA involves four steps:

1. Collect the job activity records at execution time. The records are produced by SMF and HASP and are written to a disk data set—SYS1.MANX.
2. Daily, the SYS1.MANX records are consolidated into job description records and converted to a fixed format.
3. The output from step (2) is used as input to a daily billing program which computes a cost for the jobs and prepares a detailed report of the day’s activity by account number.
4. Monthly, the input to the daily program is consolidated and used as input to a monthly billing program which interfaces with the ETS accounting system.

The raw SMF data which is produced as a result of job execution contains much valuable information about system performance and computer workload which is of interest to computer center management.

One useful characteristic of the JPA is that costs are predictable. This enables a programmer or systems analyst to determine, in advance, the costs of running a particular job and, more importantly, to design his program in the most economical manner possible. In order to facilitate this process, a terminal oriented, interactive, cost estimating program has been developed. This program is written in BASIC and enables the programmer to input various parameters of his program (such as filesize, CPU requirements, blocking factors, memory requirements) and the cost estimate program produces the cost of the program being developed. Parameters may then be selectively altered and the effects determined.

CONCLUSION

The approach to user billing described in this paper has proved useful to management as well as users. Many improvements are possible especially in the area of more accurate CET estimation. Hopefully, designers of operating systems will, in the future, include sophisticated statistics gathering routines as part of their product thus providing reliable, accurate data for accounting.

APPENDIX

A method of deriving CET parameters

Let the wall clock time (W) be estimated as follows,

\[ W' = A_T X_T + A_D X_D + A_M X_M + C \] (1)

where,

\[ X_T = \text{# of tape accesses} \]
\[ X_D = \text{# of disk accesses} \]
\[ X_M = \text{# of tape mounts} \]
\[ C = \text{CPU time} \]
\[ A_T, A_D, A_M = \text{Coefficients to be determined} \]

We wish to determine the coefficients \( A_T, A_D, \) and \( A_M \) that will maximize the correlation between \( W' \), the computed elapsed time, and \( W \), the actual elapsed time. Define the error \( e \) as,

\[ e = (W - W') \] (2)

The correlation coefficient, \( r \), can be written as,

\[ r^2 = 1 - \left( \frac{\sigma_e^2}{\sigma_w^2} \right) \] (4)

Then, in order to maximize \( r^2 \), it is sufficient to minimize \( \sigma_e \), since \( \sigma_w^2 \) is a constant over a given sample.

\[ \sigma_e^2 = \Sigma (e - \bar{e})^2 = \Sigma e^2 - 2 \Sigma e \bar{e} + \Sigma \bar{e}^2 \] (5)

Since

\[ \Sigma \bar{e}^2 = n \bar{e}^2 \] (6)

we have,

\[ \sigma_e^2 = \Sigma e^2 - \frac{2}{n} (\Sigma e)^2 + (\Sigma e/n)^2 \] (7)

\[ \sigma_e^2 = \Sigma e^2 - n^{-1} (\Sigma e)^2 \] (8)

Finally, we have,

\[ \sigma_e^2 = \Sigma [(W_i - C_i) - A_T X_T - A_D X_D - A_M X_M]^2 - n^{-1} [\Sigma (W_i - C_i)^2] \]
\[-AT \Sigma X_T - AD \Sigma X_D - AM \Sigma X_M \] 

\[\frac{\partial \sigma^2}{\partial A_T} = -2\Sigma [(W_i - C_i) - A_T X_T] \]

\[-AD X_D - AM X_M] X_T + \frac{2}{n} \left[ \Sigma (W_i - C_i) \right] \]

\[-AT \Sigma X_T - AD \Sigma X_D - AM \Sigma X_M] \Sigma X_T \]

(9)

\[\frac{\partial \sigma^2}{\partial A_D} = -2\Sigma [(W_i - C_i) - A_T X_T] \]

\[-AD X_D - AM X_M] X_D + \frac{2}{n} \left[ \Sigma (W_i - C_i) \right] \]

\[-AT \Sigma X_T - AD \Sigma X_D - AM \Sigma X_M] \Sigma X_D \]

(10)

\[\frac{\partial \sigma^2}{\partial A_M} = -2\Sigma [(W_i - C_i) - A_T X_T] \]

\[-AD X_D - AM X_M] X_M + \frac{2}{n} \left[ \Sigma (W_i - C_i) \right] \]

\[-AT \Sigma X_T - AD \Sigma X_D - AM \Sigma X_M] \Sigma X_M \]

(11)

\[A_T \left[ \frac{\Sigma X_T X_M - \Sigma X_T \Sigma X_M}{n} \right] + A_D \left[ \frac{\Sigma X_D X_M - \Sigma X_D \Sigma X_M}{n} \right] + AM \left[ \Sigma X_M^2 - \frac{(\Sigma X_M)^2}{n} \right] = \Sigma (W_i - C_i) X_M - \frac{\Sigma (W_i - C_i) \Sigma X_M}{n} \]

(12)

Since all the partials must vanish, we have,

\[AT \left[ \frac{\Sigma X_T^2 - (\Sigma X_T)^2}{n} \right] + AD \left[ \frac{\Sigma X_T X_D - \Sigma X_D \Sigma X_T}{n} \right] \]

\[+ AM \left[ \frac{\Sigma X_T X_M - \Sigma X_T \Sigma X_M}{n} \right] \]

\[= \Sigma (W_i - C_i) X_T - \frac{\Sigma (W_i - C_i) \Sigma X_T}{n} \]

(13)

\[A_T \left[ \frac{\Sigma X_T X_D - \Sigma X_T \Sigma X_D}{n} \right] + AD \left[ \frac{\Sigma X_D^2 - (\Sigma X_D)^2}{n} \right] \]

\[+ AM \left[ \frac{\Sigma X_D X_M - \Sigma X_D \Sigma X_M}{n} \right] \]

\[= \Sigma (W_i - C_i) X_D - \frac{\Sigma (W_i - C_i) \Sigma X_D}{n} \]

(14)

Solving the simultaneous equations (13), (14), and (15) for $A_T$, $A_D$, and $A_M$ should give values for the parameters that will maximize the correlation between the computed elapsed time and the actual elapsed time.

The technique was applied to a sample month of data which was composed of 19401 job steps. The coefficients determined were,

- $A_T = 0.0251$ seconds
- $A_D = 0.0474$ seconds
- $A_M = 81.2$ seconds

When these coefficients were used in Equation (1) to determine the computed elapsed time, the correlation coefficient between the computed time and actual time over the 19401 steps was 0.825. When other coefficients were used, i.e. $A_T = 0.015$, $A_D = 0.10$, and $A_M = 60.0$, the correlation was only 0.71.

Note: Card read, card punch, and print time constants were not computed in this fashion simply because there is insufficient data on job steps that use these devices as dedicated devices. However, as data become available in the future, the method could be applied to obtain good access times.

REFERENCES

1 L L SELWIN

*Computer resource accounting in a time sharing environment*

Proceedings of the Fall Joint Computer Conference 1970

2 C R SYMONS

*A cost accounting formula for multi-programming computers*

The Computer Journal Vol 14 No 1 1971

3 J T HOOTMAN

*The pricing dilemma*

Datamation Vol 15 No 8 1969

4 IBM Corp

*IBM System/360 operating system: System management facilities*

Manual GC28-6712 1971