An approach to the simulation of a time–sharing system

by NORMAN R. NIELSEN
Stanford University
Stanford, California

INTRODUCTION

The past several years have seen the development of time-sharing capability on a variety of second generation computers.[1,2,3,4,5,6] For the most part, though, these projects should really be termed experimental efforts. Because the systems were providing a broad or general purpose service, it was not possible to take advantage of the many shortcuts employed by the more specific purpose systems such as those for airline reservations.[7] No assumptions could be made about the size, running time, or bug-free condition of the programs to be executed.

Accordingly, the general purpose systems had to be equipped to cope with the full range of time-sharing problems. File and memory protection schemes became crucial. As programs became larger and longer running, memory space became a scarce commodity, necessitating the consideration of swapping programs in and out of high speed memory from secondary storage, relocating programs, and dealing with core fragmentation problems. System monitors, command languages, editors, and conversational compilers (assemblers) had to be developed. System bugs and malfunctions were frequent. These failures often destroyed files or caused other frustrating delays. Main memory and secondary storage limitations were often restrictive. High system overhead and relatively slow swapping rates often entailed poor response times.

Despite such dire conditions, these systems were very well received. The users felt that the ready access to a large machine and the opportunity for economical man-machine interaction more than offset the aforementioned difficulties. Little wonder, then, that there was widespread interest in time-sharing and a belief that this technique would play a large role in future computing.

It was at this point that the first announcements of the new third generation computers were made. There was every indication that this equipment would permit the development of vastly superior time-sharing systems, for the new hardware would enable many of the limitations suffered by previous systems to be surmounted at a reasonable cost. Not only was the circuitry to be more reliable due to the new solid logic or integrated circuit construction, but memories were to be larger, processor speeds to be faster, and random access storage devices to have greater capacities and higher transfer rates. Many of the former special hardware additions were to be standard features.

Further, certain systems were being marketed especially for time-sharing use, and manufacturers were promising to deliver time-sharing software packages. No longer would an installation have to mount a substantial development effort in order to be able to offer time-sharing capability on its own equipment. Not surprisingly, a tremendous interest developed in these new systems.

The nature of the problem

After the initial excitement subsided, some disturbing aspects became apparent. Despite the promise of these proposed systems, there was little information available on performance or operating characteristics. In terms of equipment acquisition and system operation, this posed significant problems for computation center managers. In particular, consider the difficulties faced in trying to determine the effects of hardware configuration, software modification, time-sharing algorithms, and user behavior upon system performance.

The new hardware was to be quite modular in design, so that one configured a system from a set of standard components rather than ordering a standard time-sharing machine. Hence, a variety of performance information was vital. What was the best configuration to serve a particular set of needs? What was an optimal configuration given an equipment acquisition budget of a particular size? What was the minimum or cheapest

From the collection of the Computer History Museum (www.computerhistory.org)

system which would perform adequately in a particular environment? What level of performance could be expected from a specific configuration? What would it cost to provide time-sharing service to 100 users? Despite the importance of this line of questioning, answers were for the most part unobtainable.

A similar situation existed with respect to software. Since an installation might wish to modify the manufacturer's software in order to serve a particular environment more effectively (e.g., incorporating new services or features), it was important to be able to determine the effects of these changes. Would system performance be improved or degraded? Would reprogramming to reduce the overhead associated with certain system operations significantly improve performance, or would it only serve to expose some other problem area? Again, answers were not available.

Associated with every time-sharing system is a set of algorithms for secondary storage allocation, job scheduling, etc. One of the empirically determined facts about time-sharing is that nearly every individual has his own ideas as to what constitutes an appropriate algorithm for each of the various functions. Yet, short of intuition (which is often unreliable in complex situations) there were no procedures by which the merit of suggested alternative algorithms could be judged.

Another problem area concerned the user job stream. A system might have certain characteristics which would significantly impact the processing of particular types of work. Turning the situation around, particular jobs might have characteristics which would affect the total performance of a system. This latter case was of particular interest since Scherr's[8] work on the Project MAC time-sharing system had indicated that users would indeed change the characteristics of their jobs in response to poorer service or higher usage charges. Again, appropriate information about the new systems was not available.

The need to simulate

There was thus a need to develop some type of tool for the analysis of these new time-sharing systems, so that the aforementioned problem areas could be investigated and satisfactory information could be provided. One possible approach would be analytical. For example, Scherr,[8] Schrage,[9] and Smith[10] have successfully developed mathematical models of time-sharing systems or scheduling algorithms. These efforts have not, however, exhibited the flexibility that would be necessary in order to investigate the great variety of topics outlined above. When one is pursuing variations of a particular question, an analytical approach all too often leads to a dilemma. If the new problem is faithfully represented, the model becomes insolvable; if a solvable formulation is developed, the model does not adequately address the problem area being investigated.

Another possible approach would be simulation. Appropriately used, this technique can offer the required flexibility and can serve as an effective tool for system analysis. The simulation work of Scherr[8] and Fine and McIsaac[11] is illustrative. It is not surprising, then, that there has been a substantial interest in simulating the behavior of the new time-sharing systems.

However, as is now being realized, it is not a straightforward matter to simulate one of these new systems. Despite the successful application of simulation techniques to older systems, this approach has not been very effective in analyzing the new systems. A primary reason for this failure has been complexity. Very few persons realized the complexity embodied in some of the standard systems in the third generation lines, much less in the time-sharing systems. Evidence now abounds in the form of software delays, reduced capabilities, performance problems, etc.

The effect of this complexity upon attempted simulations has also been striking. Programming often became long and involved; development time and cost climbed while execution efficiency declined. At times it was necessary to make departures from the system being modeled in order to obtain a reasonable execution speed or a feasible implementation with a particular language. On the other hand, the ability of a simulation model to perform the necessary analysis role has adequately been demonstrated. Thus, the major problem lay in coping with the added complexity in an appropriate fashion.

An attempt was made by the author to develop a generalized time-sharing system simulation which would avoid the trap of complexity without at the same time defeating its own purpose. A reasonably successful model was developed, implemented, and subsequently applied in a study of the IBM 360/67 time-sharing system on order by the Stanford Computation Center. The results of that study have been reported elsewhere[12] and will not be repeated here. Suffice it to say that through the use of the simulation model it was possible to pinpoint a number of problem areas arising from the design of the 360/67 time-sharing system. Further experimentation indicated system modifications which would relieve or eliminate many of those problems. Because of the effectiveness of that simulation, it is appropriate to consider the approach which was taken in its design and construction.

What to simulate

Despite is obviousness, the question of what to simulate was an important one. The four problem areas which were discussed previously (hardware configura-

From the collection of the Computer History Museum (www.computerhistory.org)

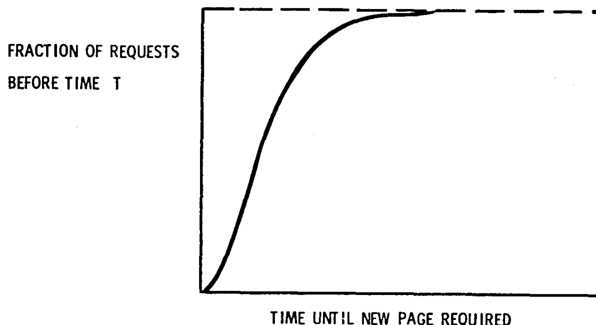
tion, software modification, time-sharing algorithms, and user behavior) were taken to be the goals toward which the simulation should be directed. A careful study of these questions led to the selection of a "software" rather than a "hardware" related level of detail. This permitted attention to be focused at a single level and enabled the resulting model to be much more efficient.

The software operating systems are concerned to a significant degree with the allocation of memory and secondary storage space and with the transmission of data between these two media. Since many of the new systems are paging systems* and since a core image swapping system** is but a special case of a paged system (page size = core size), the page was taken as the basic unit of space allocation. In addition, since the time taken by the system to perform its various functions can normally be measured adequately in tenths of milliseconds, this measure was selected as the basic unit of time.

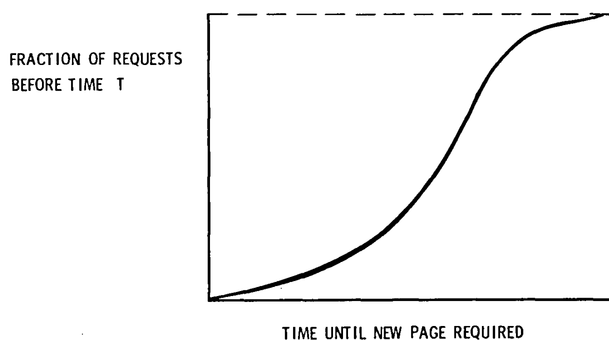
This restriction of the focus of the simulation model to a rather homogeneous level of detail achieved two aims. Not only did it make the model simpler, smaller, and faster, but it also eliminated the need for the much longer simulated time-span which would be required to study the grosser aspects of a system. The effects of these other areas, however, need not be ignored. For example, the more detailed hardware concerns about dynamic address relocation or data channel—CPU memory reference interference can be addressed in a separate simulation or other study concentrating at this level. The results of these other investigations can then be taken into account by inputting an appropriate set of parameters into the main simulation. A similar situation holds for those areas of much grosser detail. Such questions as the long term storage requirements (e.g., from day to day) for program or data files can be handled by a simple pencil and paper study, with the results again being treated as a set of input parameters.

Representation of jobs

The selection of a scheme for representing simulated user jobs can have a significant impact upon the success of a simulation. This is particularly true when such a scheme must be able to reflect behavior realistically in a paging environment. Inasmuch as many of the models of swapping systems use a single distribution to determine the amount of execution time that is to elapse between I/O operations, an obvious approach would be to follow the same line of thinking. A study could be made of the paging behavior of a job and a set of curves developed. Each distribution would indicate the execution time between requests for an additional page given that a certain cumulative execution time had elapsed since the beginning of the time slice or execution period currently allotted to the program. Alternatively, the distribution of execution time between new page requests could be plotted given that a certain percentage of the program was already in memory. Figure 1 illustrates two such distributions for a situation in which paging activity decreases as the percentage of the program already in memory increases.



25% IN MEMORY
Figure 1a



75% IN MEMORY
Figure 1b
Distributions of paging activity

Despite the simplicity of this approach, it has one serious defect. It is possible to determine only that *a* page is required not *which* page is required. Since the particular page cannot be identified, neither can its

*A paging system is one in which memory is allocated in interchangeable sections (pages) to portions of several programs at any given time. Sections of a program being returned to memory for a period of execution will be relocated dynamically into whatever page locations are then available.

**A swapping system is one in which memory is allocated contiguously to one or more programs. A program being returned to memory for a period of execution will always be placed in the same locations that it had previously occupied.

From the collection of the Computer History Museum (www.computerhistory.org)

location. Consequently, some assumptions must be made as to the location of each required page—e.g., on a queue in memory waiting to be written out (from the last time slice), on a drum (in any one of a number of rotational positions), on a disk, etc. One way of making these assumptions is to use another distribution which indicates the probability of finding a needed page in each of the possible locations. Note, though, that this is often the very information that is to be obtained from the simulation. The use of such input data makes it impossible to determine the effect of different core management or paging algorithms, thereby rendering the model ineffective for investigating some of the more important problem areas.

A more accurate technique, utilizing the same basic approach, would be to associate with each page a distribution which would indicate the probability of the next reference or transfer being to any particular page. This is illustrated in Figure 2. The master distribution of execution times would then be altered to reflect execution time between references or transfers to a different page rather than to a page which had not been previously used during that time slice. In fact, since detailed information would be kept on each page, it would be preferable to associate a second distribution with each page indicating the possible execution times before a reference or transfer would be made outside that page (see Figure 3).

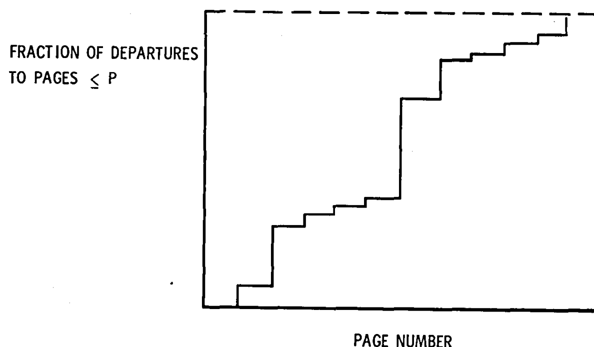


Figure 2—Distribution of page references from page X

Such a scheme, though, is undesirable from both a theoretical and a practical point of view. From a theoretical standpoint a single pair of distributions for each page is simply not able to reflect adequately the behavior of a job. For example, a page might make frequent reference to one set of pages during the initialization phase of a job but to a different set during the result printout. A set of pairs of distributions would thus be required, each pair reflecting the job's behavior at a different stage in its life.

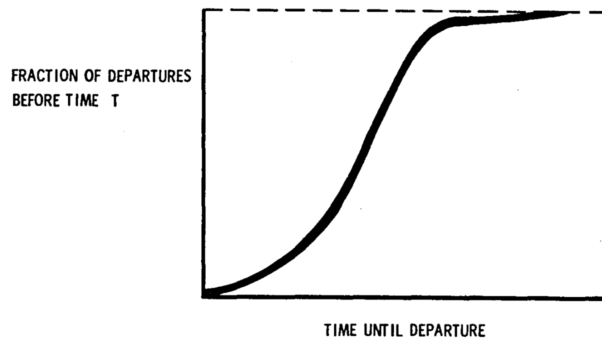


Figure 3—Distribution of execution time for page X

Even if an appropriate rearrangement of the program would enable a single pair of distributions to suffice, the scheme would not be practical for use in an actual simulation study. When one considers the fact that a random number would have to be read or calculated and a table look-up operation performed each time the simulated program referenced or transferred to a different page, the problem becomes apparent. In addition, the storage requirements for the distributions would likely entail very large memory capacities, frequent overlays, or a very compact internal representation (which in turn would increase the number of instructions required for the table look-up).

What was needed was a technique which would reduce the memory and CPU cycle requirement but which would still retain the essential detail about a program's operation. The approach finally selected was a fairly deterministic one. Each stage of a program is represented as a chain or sequence of instructions which describes the specific behavior of that program during that stage. This description indicates a series of specific page references and I/O operations interspersed with execution times. In this way random number generations, table look-up operations, and table storage requirements are substantially reduced.

On the other hand, this procedure raises the new problem of storing these detailed job behavior descriptions. This problem can be significantly reduced by permitting the reuse of instruction sequences. That is, if there are repetitive aspects to a job, it is possible to cycle back and reuse the instruction sequence again and again, thereby eliminating the need to duplicate descriptions several times. For example, consider the case of a compilation task. During the syntax checking and table building phase a description of the process-

From the collection of the Computer History Museum (www.computerhistory.org)

ing of two statements might be constructed and repeated fifty times in order to reflect the processing of a one hundred statement source file. A description of the code building phase could then be substituted and repeated several times to reflect the execution of the next stage of the compiler, etc. Additional space and execution reduction techniques are discussed at a later point in this section.

The use of a deterministic description involves the loss of a certain amount of authenticity. This is especially true when a particular sequence is repeated many times. However, if the sequences are appropriately constructed, this should not materially affect the accuracy of the simulation. As a further safeguard, the instruction sequences are developed in a two step process.

First, a master or prototype description of each different type of job that might be processed by the simulated system is developed and input to the model as data. Thus, there might be prototypes for compilation, file maintenance, list processing, etc. Then during the actual runs, whenever a job requires another instruction sequence for its next stage, a specific sequence for that particular job will be constructed from the appropriate part of the master prototype. The construction process itself involves a certain amount of randomness, so the procedure is actually semi-deterministic in nature. Thus, each simulated program resembles the appropriate prototype job, but none is a carbon copy of it. Consequently, if several programs of the same type are running on the system, the effects of the determinism are even further obscured. The aforementioned procedures will be more meaningful after a consideration of the way in which job behavior can be described at both the specific and prototype levels.

The instructions which comprise the specific descriptions are each composed of four fields as follows:

A	F	C	T

A—the page on which the operation is to be performed

F—indicator flags which may affect the operation

C—the op code designating the operation to be performed

T—the raw execution time scheduled to elapse prior to performance of the operation.

A limited amount of experimentation revealed that, by appropriate use of flags, a total of seven different operations would suffice to describe the execution behavior of a job. An eighth operation (C=5) was subsequently added to improve the execution efficiency of the simulation. The function of each of these operations is as follows:

C=0: After executing for time T perform an end of time slice for the current job. This instruction never appears in the job's sequence; rather it is dynamically set up by the simulation as the next operation whenever a time slice end is required.

C=1: After time T obtain page A. That is, if page A is not in memory, set up the necessary queue entries to have it read into memory and place the task in page wait status. If page A is already in memory, return to the execution of the job (for time T of the next instruction). This operation provides the basic means for bringing pages into memory.

C=2: After time T obtain page A as for the previous operation but then increment A by 1. When A refers to the end of a program segment (16 consecutive pages, the last of which is evenly divisible by 16), reduce A by a value which is a function of the flags which are then set in the instruction. This operation can be used to reflect the building of tables and files, since each time the sequence is repeated (up to a limit) the next higher page of the file will be referenced.

C=3: After time T write pages A through A+X out on device Y, where X and Y are specified by the status word representing page A. This operation can also be used to represent the reading of information into parts of pages. That is, the particular pages must be brought into memory before the I/O can begin, since these pages will either contain information to be written out or information that will not be overwritten and hence must be preserved.

C=4: After time T read pages A through A+X (in their entirety) from device Y. This operation differs from the previous one in that any set of available pages can be assigned to receive the input since the contents will be completely overwritten. The previous copies of these pages will then be deleted from their secondary storage locations.

C=5: After time T obtain X consecutive pages beginning with page A, where X is a function of the flags which are set in the instruction. This operation is merely a shortcut to avoid using several of the basic operations (C=1). The pages will be read on different drum revolutions, overhead will be charged as if

From the collection of the Computer History Museum (www.computerhistory.org)

there were several separate page fetches, etc. The intent is thus to improve the performance of the simulation without affecting the simulated system.

C=6: Release the job's virtual memory (the logical memory area used by the program as opposed to physical memory) from page A through the end of the program segment. In addition to freeing up virtual memory for reassignment to this or other jobs for different programs, the operation marks the core and/or secondary storage locations of these pages available for use.

C=7: After time T provide a terminal interaction of length A for the job.

With a little practice it becomes quite easy to develop job descriptions in terms of these operations. Table I illustrates part of an instruction sequence describing (in a trivial fashion) the first phase of a compilation. The first instruction indicates that five milliseconds of computation take place using the pages which have already been brought into memory for the job. Then a terminal interaction of fifteen seconds duration occurs. After the terminal input of a new source statement, the first five pages of the compiler are called (pages 37-41). The first two pages of the symbol table (pages 12-13) are next called after four milliseconds of execution. A sixth page of the compiler (page 42) is then called following ten milliseconds of execution. One millisecond later the source statement is placed in the second page of the source file (page 5), and after another millisecond the condensed source statement is placed in the third page of the condensed file (page 22). The last instruction indicates that one millisecond of execution occurs before an entry is made in the control table (page 45).

Despite the ease with which instructions can be developed, it is quite another matter to write compact descriptions which at the same time will be both accurate and efficient. Although a program making use of the semi-deterministic descriptions will execute much faster than one using a multitude of distributions, this still does not guarantee that the program will have a reasonable running time. For example, consider the case of a two page program which executes one instruction from the first page, one instruction from the second page, another from the first page, etc. The program itself would not encounter any performance problems during execution, but a simulation having to track each reference would be extremely slow. Hence, a few other helpful features were included in the design of the basic operations.

Table I Sample Instruction Sequence

A	F	C	T	Comment
15	0	7	50	Terminal interaction
37	3	5	0	First 5 pages of compiler
12	0	5	40	First 2 pages of symbol table
42	0	1	100	Sixth page of compiler
5	0	1	10	Second page of source file
22	0	1	10	Third page of condensed source file
45	0	1	10	Control table

For instance, in the two page program example given above, it is not necessary to track each transfer between the pages so long as both pages can be retained in the simulated system's memory during the time slice. Accordingly, the shortcut instruction (C=5) could be used to bring in the pages, and a rather long execution time could be specified to take place prior to the performance of the next operation. In this way the paging behavior of the program could be reflected without becoming bogged down at the level of individual instructions.

This does not, however, completely solve the problem. Should the execution time be sufficiently long, one or more time slice ends may occur before the next operation is performed. At the beginning of a new time slice, no operation is performed; rather, execution proceeds using whatever pages are returned to memory by the paging algorithm. In the event that this algorithm does not return all of the pages actually intended to be in memory (in this case two), the shortcut procedure will produce erroneous results. Consequently, provision was made for as many of the previous operations in the sequence as desired to be performed at the beginning of each time slice, so that the actual paging activity could be reflected. The reexecution of these instructions is handled in such a fashion that timing considerations are not distorted.

The ability to remove instructions from a sequence is another efficiency aid. For example, it might be that every repetition of a sequence would be identical with the exception of the first, wherein a read instruction is required to load a program from disk storage. In order to avoid the necessity of constructing a separate sequence specifically to perform the read from disk, a flag is used to indicate that the instruction is to be deleted from the sequence immediately after performance of its specified operation.

The master sequences contain prototypes for all of the specific instructions except time slice end (C=0). At the time that a specific sequence is constructed, the F, C, and T fields of the prototypes are carried over directly to the corresponding fields of the instructions being developed. The A fields which refer to logical pages of the prototype job are translated to reference

From the collection of the Computer History Museum (www.computerhistory.org)

the corresponding logical pages of the specific job. The A field of instruction type 7 represents a terminal "think" time, and it is selected according to a probability distribution (see below).

In addition to the prototype instructions, the master sequences contain six different types of control instructions which guide the construction of the specific sequences for the individual tasks. Some of these control instructions can be used very effectively to reduce job description storage requirements; others can be used to introduce some variation into the sequences which are constructed. The format for these additional instructions remains the same, but the A and T fields have quite varied interpretations. The function of each of these control operations is as follows:

- C′=0: Set the terminal user headscratch parameter to be A,T. Until this parameter is reset, each terminal interaction instruction that is constructed for a job will be given an interaction time equal to the T value of this instruction plus a random increment drawn from the uniform distribution O,A. This interaction time covers the transmission time for output to the terminal, the headscratch or think time of the programmer, and the typing or input time for the response.
- C′=1: Set the loops parameter of the job to be T plus a random increment drawn from the uniform distribution O,A. This parameter specifies how many times the instruction sequence is to be repeated before creation of a new sequence.
- C′=2: Terminate construction of the current instruction chain. T indicates the location in the master sequence where construction of the job's next sequence should begin after the now completed one has been executed the appropriate number of times.
- C′=3: Transfer the sequence construction process to a new location in the master sequence in the manner indicated by the value of A:
  - A=1: Save the index to the current location plus one in the master sequence for a later return; transfer to the instruction at location T.
  - A=2: Transfer to the instruction at the location which was previously saved.
  - A=3: Store the index to the current location plus one in location T minus one; transfer to location T.
  - A=4: Transfer to the location specified in location T.
  - A=5: Transfer to location T.
  - A=6: Compute a random number R, and compare it with the A fields of the succeeding instructions. When R exceeds A, transfer to the location specified by the associated T field.
- C′=4: Set the status word for page A. Depending upon the flags, the status word will be set to indicate that the page should be considered as changed, unchanged, or shared page T during any given time slice. A page is said to be changed if at least one word is stored in it during a time slice, so that the page's secondary storage location (from which it was read) does not contain a correct copy at time slice end. A shared page is one which several jobs make use of in common, so that only one copy of that page need exist in the system at any one time.
- C′=5: Reserve or release I/O device, A, which is dedicated for use by a particular job (rather than a particular job type). A flag determines whether an allocation or a release is to be made. In the event that a requested device is not available, a wait (dummy terminal interaction) will be inserted into the instruction sequence and a subsequent attempt made to allocate the device.

The use of the status setting control instruction (C′=4) implies that, for the life of an instruction sequence, any particular page will always be changed (unchanged) during each time slice regardless of when the time slice ends might occur. Although this situation is unlikely to be strictly true in practice, the careful design of the master sequences can make it a reasonable approximation. To try to determine from a distribution at the end of every time slice whether or not each page was changed leads toward the same problems faced previously with regard to the determination of page references from a distribution. Hence, the decision to fix the alterability of pages for the duration of a sequence.

The concurrent existence of several jobs of the same type presents no particular problem with respect to paging, since the logical page numbers of the master sequences are translated to unique job page numbers during the construction of each job's instruction sequences. Device references, however, undergo no such translation, so that every job of the same type will reference the same physical devices. In the case of the direct access devices which are prevalent on the new time-sharing systems, this may be quite realistic. In the case of sequential devices such as tape drives this poses a problem, for in general such devices must be assigned exclusively to a single job.

From the collection of the Computer History Museum (www.computerhistory.org)

For this reason the notion of device pools was developed, permitting a number of similar devices such as tape drives to be assigned to a pool covering certain logical numbers. The reserve instruction then causes the assignment of one of these devices to the job, and all subsequent references to this logical unit by that job will be translater to refer to the proper physical unit.

Despite these various shortcuts and aids, great care must still be taken in developing a set of prototype job descriptions. It is, unfortunately, quite easy to develop rather large and inefficient descriptions. On the other hand, quite compact descriptions are possible. For example, the thirty-five job prototypes used in the analysis of Stanford's proposed system required less than 1150 words of 7090 memory. This was not without cost as it took three man-months to put these descriptions together. Nevertheless, given the availability of such a set of descriptions, it is possible to shift job mixes in seconds merely by adjusting the parameters which govern the assignment of jobs to terminals. Further details about the instruction language as well as sample job descriptions may be found in Nielsen.[13]

Output data

Although the development of appropriate performance data for the system being analyzed is a standard problem of simulation, the much greater complexity of the new time-sharing systems has not been without effect. Not only is there a greater variety of possible performance measures, but there is also a vital need to obtain a comprehensive set of measures which might indicate why a particular set of performance figures resulted from a run. That is, given that a system's performance is unsatisfactory during a particular run, some type of information about the conditions underlying those results must be given to the investigator in order that he may develop new ideas for modifications which might improve the situation. Accordingly, it was decided to develop a rather large selection of information during each run. Depending upon the wishes of the investigator, some twenty-five to fifty pages of output can be obtained.

With respect to the simulated work load, distributions can be obtained which indicate the responses received by different job classes (priorities) and by different job types. With respect to the central hardware, statistics can be obtained which indicate CPU utilization (for job execution, overhead, etc.), paging activity, and memory shortages. With respect to secondary storage devices and other I/O units, distributions can be accumulated which indicate the queue sizes and waiting times for both read requests and write requests to each device. Distributions of available space on these devices as well as peripheral equipment utilization statistics can also be obtained.

Further, since it is not possible at the time of development to foresee every piece of information that might later be required from the model, it is necessary to make provision in the design for the satisfaction of unanticipated data demands. The approach selected for this situation was the maintenance of several files into which data could be placed, either on an occurrence basis or on a sampled basis, for later analysis. This mechanism provides a great deal of flexibility at very little cost, particularly since a standard data writing routine enables the collection of new types of information to be readily instituted and the efficient buffering and overlapping of output data to be incorporated. The use of such a routine is also illustrative of the next problem area to be considered.

Modularity of the simulation

The complexity of the new time-sharing systems and the problems already encountered by investigators trying to simulate them tend to make execution efficiency a prominent characteristic of any model. Despite the need for speed, though, it may well be advantageous to compromise this goal to a certain degree. Inevitably it becomes necessary to alter or modify a simulation model after it has been constructed and used. Hence, this factor should be considered during the design phase, so that subsequent changes can be made quickly and easily. More than one model has lost much of its effectiveness when desired modifications could not be implemented within a feasible time span.

Consequently, a fairly modular design was selected for the simulation. To the extent possible each of the major algorithms and functions was isolated from the rest of the model. Thus, for example, the job scheduling algorithm, the data output routine, and the device managment algorithm were each developed as a separate section of the model. Although this produced several obvious execution inefficiencies, the ease with which subsequent changes were incorporated into the simulation clearly demonstrated that the price was very cheap indeed.

Language selection

The choice of a language in which to implement a simulation can often have a significant impact upon the effectiveness of that simulation. Even if implementations were to be based upon a common model, the use of different languages would almost of necessity involve design modifications of varying degrees. In some instances these will be of major import. Five major areas in which language choice is felt are: compilation speed, execution speed, programming difficulty, memory utilization, and compiler availability.

From the collection of the Computer History Museum (www.computerhistory.org)

For the case in point, the most important consideration was execution speed. Memory utilization took on a similar importance in view of the model's scope and complexity. Programming difficulty and compilation speed were also important, but only insofar as it was possible to work without undue restrictions. Since the resulting program was to be made available to other investigators, a greater than normal weight was placed upon language availability on various computers.

An assembly language would probably have best met the execution speed goal. However, these languages are machine dependent, and their ease of programming leaves much to be desired. General purpose languages such as Fortran offer a much better compromise. Not only do some compilers have optimizing passes which are capable of emitting quite efficient code, but compilation speeds are often very reasonable in relation to other alternatives. Although some languages do have extensions for simulation work (e.g., Algol-Simula,[14] CORC-CLP[15]), these systems are not in widespread use.

Interpretive simulation languages such as GPSS[16] probably have the fastest "compilation" speeds, but execution efficiency is seriously hampered. A language such as Simscript[17] appears to be more appropriate. Not only does it offer greater flexibility, but the modularity which it imposes upon a program reduces compilation (although not loading) requirements. On the other hand this modularity extracts a price at run time. In contrast to other languages, Simscript offers ways to utilize memory more efficiently. Again, though, a substantial price is extracted in terms of execution efficiency. Some of these problems have been corrected in more recently developed Simscript compilers, but these processors are not as yet in general use.

Accordingly, despite the multitude of simulation languages available and despite the programming advantages of these languages, the simulation was implemented in Fortran IV. This is not to say that the use of Fortran was without a price, for it entailed considerable extra designing, programming, and debugging work. From an effort or convenience point of view, a high level simulation language would have been preferable. Even more general features such as the report generator of Simscript or the automatic data collection of GPSS would have been of assistance. However, these factors were not the primary criteria for language selection in this particular instance.

The fact that Fortran did not have explicit capabilities for simulation was in some ways advantageous. Granted, no timing mechanism is automatically provided and no ready list handling or queueing procedures are available, thereby forcing the programmer to develop his own routines. However, this does afford the opportunity to tailor these routines to fit the needs of the particular simulation, enabling significant gains in program efficiency to be made. For example, the timing mechanism in the time-sharing simulation was designed to operate from two queues in order to take advantage of certain characteristics of the model. The various queues of pages available for assignment or awaiting transfer to or from secondary storage were also treated in specialized fashions. A dynamic memory allocation scheme was employed to supply entries for the multitude of queues contained in the model. A description of the various special design features may be found in Nielsen.[12]

As an indication of the effectiveness of the design and implementation decisions discussed in this and previous sections, the simulation was able to reflect the time-sharing behavior of an IBM 360/67 at an execution ratio ranging between 1 to 2 and 1 to 3 the right way while executing on a 360/50 with a 256K byte memory. That is, one minute of 360/50 time was required to simulate between two and three minutes of 360/67 operation. This was much better than had been anticipated for a simulation at this level.

Parameter inputs

Having designed and implemented a simulation model, the next step was to construct a set of input data for an actual application. The problems associated with the development of appropriate parameter values are similar to the ones faced by investigators who simulated some of the existing time-sharing systems and will not be elaborated upon here. However, the word existing does indicate one pitfall.

For example, at the time that information about the 360/67 time-sharing system was desired, that system did not exist. Consequently, it was necessary to simulate the *intended* rather than the actual design of that system. Further, as on any project of such magnitude, implementation does not always proceed along the design path. Thus, not only must the design and its changes be monitored, but so must the implementation.

CONCLUSION

The simulation of the new large-scale third generation time-sharing systems is not just a move along the continuum of time-sharing simulations. Rather, there has been a rather sharp increase in the complexity of the new systems which has necessitated a correspondingly sharp change in the manner in which they are simulated. The approach illustrated in this report is not the only one nor is it in all likelihood the best one.

From the collection of the Computer History Museum (www.computerhistory.org)

Despite the advantages, the use of semi-deterministic job descriptions is not an easy task, and the development of specialized list structures and processing routines is not without hazard. However, this approach has been employed quite successfully in the study of one of the new time-sharing systems, and it may help others to refine their own ideas about simulating these systems if not actually to develop their own simulation model.

REFERENCES

1 F J CORBATO M M DAGGETT R C DALEY
An experimental time-sharing system
Proc 1962 SJCC vol 21 Spartan Books Washington DC pp 335-344

2 T E KURTZ K M LOCHNER JR
Supervisory systems for the Dartmouth time-sharing system
Computers and Automation vol 14 no 10 pp 25-27 55 October 1965

3 J McCARTHY S BOILEN E FREDKIN J C R LICKLIDER
A time-sharing system for a small computer
Proc 1963 SJCC vol 23 pp 51-57

4 J H MORRISSEY
The QUIKTRAN system
Datamation vol 11 no 2 pp 42-46 February 1965

5 J I SCHWARTZ E G COFFMAN C WIESSMAN
A general purpose time-sharing system
Proc 1964 FJCC vol 26 Spartan Books Washington DC pp 397-411

6 J C SHAW
JOSS: A designer's view of an experimental on-line computing system
Proc 1964 FJCC vol 26 Spartan Books Washington DC pp 455-464

7 *A survey of airline reservation systems*
Datamation vol 8 no 6 pp 53-55 June 1962

8 A L SCHERR
An analysis of time-shared computer systems
MAC-TR-18 Massachusetts Institute of Technology Project MAC June 1965

9 L E SCHRAGE
Some queuing models for a time-shared facility
Unpublished doctoral dissertation Cornell University February 1966

10 J L SMITH
An analysis of time-sharing computer systems using Markov models
Proc 1966 SJCC vol 28 Spartan Books Washington DC pp 87-95

11 G H FINE P V McISAAC
Simulation of a time-sharing system
Management Science vol 12 no 6 pp BI80-194 February 1966

12 N R NIELSEN
The simulation of time-sharing systems
Comm of ACM vol 10 no 7 pp 397-412 July 1967

13 N R NIELSEN
The analysis of general purpose computer time-sharing systems
Document 40-10-1 Stanford University Computation Center December 1966

14 O DAHL K NYGAARD
SIMULA—an ALGOL-based simulation language
Comm of ACM vol 9 no 9 pp 671-678 September 1966

15 W E WALKER J J DELFAUSSE
The Cornell list processor
Cornell University 1964

16 IBM CORPORATION
General purpose systems simulator III user's manual
Form H20-0163

17 H M MARKOWITZ B HAUSNER H W KARR
SIMSCRIPT: A simulation programming language
Prentice-Hall 1963

From the collection of the Computer History Museum (www.computerhistory.org)