Reflections on the design of a CAI operating system

by E. N. ADAMS

IBM Watson Research Center
Yorktown Heights, New York

INTRODUCTION

A time shared general purpose computer programmed to operate in conversational mode may be used as a vehicle for computer aided instruction (CAI). Although at present CAI is a relatively expensive medium the outlook is that as the efficiency of time sharing system increases, the cost per terminal of such a CAI system will decrease and CAI will become practical for more and more educational applications. As a result R&D activity in CAI has increased markedly in the last few years.

Current R&D in CAI is being addressed to a number of important problem areas: system design and terminal capability, programming languages and procedures, pedagogical techniques in relation to various subject matter areas, problems of operational use. Most current CAI work is not especially concerned with cost effectiveness and operating efficiency, since in an R&D phase one is more concerned with feasibility and effectiveness than immediate practicability and efficiency. Nevertheless, we think a discussion of cost effectiveness and operating efficiency is worthwhile as an avenue of insight into the wise use of CAI.

Cost effectiveness of a CAI program is perhaps best approached conceptually in terms of the strategy of the instructional program and how well it makes use of both human time and system capabilities. In particular, since an information system is a high-cost medium one expects that where good cost effectiveness is achieved it will be through a well conceived exploitation of the system capabilities which are unique to a computer based system:

- Means of input and display which permit flexible man-machine communication,
- Capability to process and respond to messages written in natural language,
- Capability to rapidly evaluate complex mathematical functions,
- Capability to record, analyze, and summarize student performance data, and
- Capability to administer programs of instruction in which flow of control is contingent on a variety of program parameters and indices of performance.

We cannot discuss operational efficiency of a CAI system in such general terms, but instead must consider how the physical configuration and the programming system conform to the pattern of instructional use of terminals and processor: an optimal system design for a program which primarily exploits the display capability would be quite different from that for a program which primarily exploits the processing power.

We have looked at a number of CAI programs prepared for use with the IBM 1440, and IBM 7010-1440, and the IBM 1500 to see what generalizations about system design they might suggest. These programs were all designed to be used with a terminal which was either (a) a typewriter, (b) a CRT-keyboard station with light pen input coordinate sensing capability, or (c) either (a) or (b) supplemented by an A/V device which could display prestored output in the form of still pictures of audio records. (The authors of these programs did not have available, for example, a "Sketchpad-like" capability for graphics manipulation, a large simulator, or any other technical feature using truly large processing capability). We think that as a group these programs may be representative of a broad class of CAI programs which can be executed on a system of modest cost.

The instructional programs we have examined exhibit several basically different structures:

- A "scrambled book" structure, in which the
A program consists of small presentation-and-test units ordered in a defined sequence, but with an occasional "test and branch" unit to transfer control from one sequence to another as appropriate on the basis of student achievement.

- A "drill and practice" structure, in which the program consists of modules, each module containing a pool of similar exercises. Items within a given module need not be presented in a prescribed order but may be drawn from the pool in a random way. Thus a student does not "go through" a module, but "practices within it" until he attains to some statistical criterion of mastery. Within such a module the program is conveniently structured into a "text block" and a "control block." Drill and practice programs present a wide spectrum of system requirements in the areas of terminal characteristics, system response rate, processor load.

- A "diagnosis and decision" structure, in which the student is required to carry out a complex task consisting of a number of subtasks that are logically related and may have to be carried out either wholly or partly in a prescribed order. Such a program typically does not consist of a sequence through which the student must proceed, but of a network of program blocks and of logical trees. Examples of such programs are chemical analysis or medical diagnosis.

- A "simulated laboratory" structure, in which the student "experiments" with a mathematical model of a process or processes. Such a program may contain simulator blocks, request handling blocks, and tutorial blocks. It may make use of auxiliary equipment used either on or off line, and the student may exercise considerable initiative as to what part of the program has control at any moment. Business games represent a widely familiar example of simulation program.

The quantitative technical requirements of these programs vary widely, so we are limited to qualitative conclusions in regard to what the system program ought to do. Combining these with some frequently expressed requirements of program authors and student users, we compile the following list (non-exhaustive) of system features which are important for design of the operating system.

- **System response** usually should be "very fast," in the sense that the student should be working much more than he is waiting. Some users specify a system response time of a small fraction of a second; in many cases we think it may suffice to have the machine reply to a student in a tenth of the time it took him to frame a message.

- **Source language** must provide a number of functional requirements, some of which will be listed below. The most difficult questions regarding source language involve how "easy" it is for various classes of users to avail themselves of various features.

- For the **source language compiler**, capability to compile programs entered either "off line" or "on line," very fast compile service where small emendations are made to a program.

- **Transaction recording**: capability to mark and specify content of transaction records which may include time data, contents of storage, text of messages; simple procedures to request automatic retrieval, analysis, and summary of data in records.

- **Recovery from malfunction**: in addition to features generally desired on other time shared systems, special provisions to recover recent contents of working storage after system malfunction; also, especially in system configurations having remote terminals, a number of features to inform users of system or terminal status in event of malfunction.

Of the performance requirements referred to above, one which is troublesome to realize quite generally is that of fast response times. CAI programs are commonly very long; the stored program to process a single student message may consist of hundreds to thousands of instructions of machine code. Moreover because each of the many users of a program will need his own working storage, a single tutorial program may require a very large amount of machine storage, usually peripheral storage. Thus a central problem in a design of the operating system is how to provide adequate working storage for an instructional program and how to quickly get into core all the necessary program and information from peripheral storage at program execution time.

In the disc oriented systems we have worked with, the preferred location of all programs not being currently executed is on disc. The core memory is partitioned into an area for the monitor program and areas for programs to be executed under the monitor. A "course" program to supervise a given student is normally brought into core only when his terminal indicates to the system that it has completed sending in a message; at that time an appropriate part of the course is copied into a core area allotted to his program.

Core allocation tradeoffs to speed up terminal service are involved at several levels in our system designs. These tradeoffs are primarily concerned with making best use of available core storage by a proper
allocation:
  - Among monitor, interpreter, and user programs.
  - For a given user, among course program, working storage, and routines needed to execute "functions."
  - Among competing user programs.

To facilitate making the tradeoffs between monitor, interpreter, and user programs we made a distinction between two kinds of routines which constitute our source language: common routines, (which are "operation codes"), and infrequently used or very lengthy routines, (which are "functions"). The two kinds of routine are compiled and executed in different manners. The "operation codes" are either compiled directly into machine language or translated at execute time by an interpreter which is always resident in core; in either case, the machine code necessary for execution is all in core at execution time. The "functions" on the other hand are left untranslated in the compiled program and are executed interpretively by special routines which are fetched at execution time, either from disc or, in special circumstances, from elsewhere in core. By making a routine a "function" rather than an "operation code" we trade off the extra delay incurred when it is necessary to fetch the routine from disc against the core space necessary to store and interpret it.

In allocating core storage for a given user the "page" size of program is the natural parameter. In determining the page size one should consider not only the total number of instructions in the user program but also the number of these which must actually be executed in processing a typical student response. Thus it might be that to deal with the expected range of replies to a particular question the program would have to contain 1000 instructions, and yet that any of the replies most commonly given by students would be completely processed by the first 100 of these instructions. In all these common cases then, the remaining 900 instructions are not required in core. From such examples we are led to consider choosing the page size as small as possible, so long as the most commonly used answer processing strategies can be brought into core on one or two pages, even though an occasional slow system response will be experienced by the terminal user who has made an uncommon answer.

A given user program will suffer incremental delays in execution (which contribute to the response delay) because of every reference to disc, whether for contents of previous working storage areas, for routines to interpret "functions," or for more program to execute, so in principle one should tradeoff all three of these delays in making core allocation. However, in our system designs we have thus far always taken the approach of providing only a small amount of storage (of the order of 1000 bytes) for bit switches, numbers, buffers, etc., but making it all available in core at execution time. Thus our chief tradeoffs involved the "page size" of program brought in, against the average probability that the program to interpret a desired function would be in core, and both of these against the number of programs using core at one time.

It may seem obvious that if the processor must commonly make several references to disc in the course of processing a single student message there would be definite advantage in processing several terminal programs simultaneously, since during the many milliseconds the disc arm is in motion seeking more program to service one terminal, the processor could be processing program for another terminal. Clearly such multiprogramming would permit the system to avoid the situation that all users experience extremely long response delays whenever a single "problem" user requires an extremely long time to have his terminal serviced. However, when core is divided among many programs one pays a price for frequent interruption, competition of programs for disc arms, etc., and as the available core is divided among more and more user programs, and as the average amount of core available to each one goes down correspondingly, a point is reached beyond which the average user will get slower terminal service. This point is reached roughly speaking when every user requires on average multiple fetches of program from disc to service a single request. Thus the optimum number of programs to share core is a function of the amount of core to be shared; if there is very little core available altogether, multiprogramming may actually result in slower terminal service on average.

The programs we have seen represent such a great variability of structure, that no single storage allocation could approach being optimal for all of them at once; the best allocation would clearly change from user to user, hence, from hour to hour as the courses loaded on the system change. It would be very desirable, therefore, for the monitor to provide means for the system operator to vary at will (a) the maximum number of user programs which will be simultaneously serviced, (b) the number (and if desired the identity) of functions which will be resident in core, and (c) the "page size" of the executable program.

We may indicate the magnitudes involved here by means of an example. Our 7010 programs require that there be stored somewhere from 900 to 30,000
characters of compiled machine code to process the answers to a single question. This code is on disc and is brought in at a page at a time when an answer is being processed, our page being 900 characters at present. Many programs use several functions for each 900 characters of compiled code, each 7010 function itself consisting of 900 to 3,600 characters of code. Clearly in executing programs which use so many functions, more function code comes into core than program code. In seeking how to balance time spent in fetching functions with time spent in fetching new programs we concluded that for many of our programs on the 7010 10-20,000 characters of core for “recently used functions” would suffice to adequately limit the disc arm movements involved in the necessary fetches. The page size for correspondingly good program access rate would range between 1,800 and 4,500 characters. With the amount of core space we have for working storage we would have room for 2 to 4 user programs after optimizing the size of the page and the core block reserved for functions. We are still making system measurements to improve our understanding of these tradeoffs.

We have not time on this program to explore any of the complex questions involved in source language specifications. We will merely enumerate some of the functional capabilities the programming system must provide:

a. A convenient means of programming basic “question-and-answer” sequences is very important for all programs. A “question-answer” sequence commonly begins with a question, (i.e., a request for student input) which is followed by a series of program blocks, each consisting of a single strategy for reacting to the student’s answer. An individual strategy typically begins with a test of the student answer and a branch on the outcome; if the test “succeeds” an associated sequence of statements is executed; if the test “fails” control passes immediately to the point in the program where the next strategy begins.

b. Capability to edit, search, transform, and extract from student messages in a variety of ways. Such capabilities are essential so that the student can communicate in an other than monosyllabic way and in regard to a variety of tasks without cumbersome formatting and coding procedures.

c. Working storage which can be addressed by the instructional programmer in which he can store various kinds of information: decimal numbers, clock readings to tenths (preferably hundredths) of a second, address locations, alphanumeric messages, logical conditions expressing “states” which have occurred in execution of the program.

d. Means for the instructional program to test and branch on a variety of conditions involving stored information: bit switch values, numerical inequalities, alphabetical order.

e. Capability for the instructional program to do arithmetical operations on stored data.

f. Capability to execute as part of a source language program special routines written in lower level language and to add new routines to the source language at will.

g. Capability to efficiently search, mark, and modify complex text structures.

Our system users have had much experience with the entry of new program material, both on-line or by cards punched off-line. A number of users find that for the entry of original program material card entry is preferable. However, there is consensus that the combination of on-line entry and immediate instruction-by-instruction compiling (as in the IBM 1440 and 1500 Coursewriter systems) is very valuable when an author is revising a program, since a large number of minor program corrections must eventually be made, and one wishes to shorten the turn around time for proof and revision. For other problems of program preparation on-line compiling is not a panacea.

The process of on-line revision of a program commonly results in its being located in an undesirable pattern on disc (one for which response times will be long), and leaves the author without a simple documentation of the program. Thus it seems that an author must normally end up making complete recompiles of a program after significant revisions are complete and before normal use.

Logging and analysis of transaction data is an especially important function of a CAI system. There are several different kinds of data request which seem to be important, corresponding to different functions for which a CAI system may be used:

- Program requests for purposes of program control, e.g., to affect a student’s progress through the next units of program.

- Teacher requests concerned with class supervision; these may involve brief summaries of individual performance statistics and a report of the distribution of the class members within the program at a definite point of time.

- Teacher requests concerned with student counseling or evaluation; these may involve a detailed record of performance statistics for an individual over a period of time.
• Author requests aimed at item evaluation and improvement; these will involve actual texts of student messages and distributional data concerning the processing of messages by various blocks of program.
• Educational research requests; these may involve any of various kinds of data, and may be organized by item or by students, possibly with reports of correlations and summaries of correlation between particular data sets.

For purposes of designing the logging and analysis system the most important difference between the various kinds of data requests is not in the nature of the data needed, but in the time at which requests are made and serviced in relation to the time that the data are taken. Roughly speaking there are three kinds of requests:
• Requests to be processed as data is taken, such as those in current “course registers.”
• Requests which must be processed very quickly, such as weekly status reports.
• Requests which may be processed on a convenience basis as the necessary files are made up, such as large data compilations for authors or researchers.

Two kinds of files are required to service these requests: files which are always current in the system and files which are made up from transaction log tapes. Data summary request procedures will be different for the two kinds of files: the procedure for querying current files is via on-line author service programs or utilities, and request must be serviced when the CAI system is operating; the procedure for querying tape files is an “off-line” request probably via cards to a program which sorts and searches tape files.

The file preparation and sort programs to make these data summaries convenient to get will be fairly elaborate. For greatest convenience of use the programs should provide for accepting requests having a fairly flexible request format and should produce data summaries which also provide for a range of formats. Unfortunately, our understanding of the design of these programs is still quite limited, and our present programs leave much to be desired.

The problems of avoiding and recovering from malfunctions are big ones for all time shared systems, so it would be redundant to discuss most of them here. Probably the single most crucial recovery problem for CAI is that of reconstituting a memory area which a program was using for working storage at the time of malfunction. The situation of the student whose “terminal record area” is lost is not entirely analogous to that of the user who loses his work area while doing computation. This latter person normally has many of the results he had previously obtained in the form of a printout at his terminal, so need only repeat certain portions of the work. The student, whose records are lost, by contrast, has no knowledge of or access to the memory areas which control his program, so has no choice but to repeat in detail a stale conversation.

We consider that the aversive affect of having to repeat a long conversation is very great, so we feel it is essential to keep exact copies of the vital terminal status records somewhere in the system and to update them after each interchange between student and machine. Then when a system malfunction does occur it will be possible to restart the program at a point in time no more than a minute or so before the malfunction.

While “cleaness” of recovery is the most crucial reliability requirement for CAI, rapidity of recovery is also very important. It is worthwhile spending considerable programming effort to arrange that minor malfunctions can be detected and corrected without operator intervention since then recovery can commonly take place in seconds and only one or a few users would ordinarily be aware that a malfunction occurred. Most malfunctions requiring operator intervention can still be recovered from in about one minute without serious inconvenience to users, if operator procedures are good and the operators have suitable utilities available for the purpose. However, malfunctions which require all users to reinitiate their programs are relatively much more annoying to the users, who may lose four or five minutes on account of the malfunction. We feel that a few of these per day, perhaps as many as one each hour, can be tolerated by the students if recovery is “clean.”

A certain number of malfunctions are going to occur which cannot be so quickly recovered from, especially in a project where there is experimental hardware or an active program to develop new features in the source language. In our 7010 operation we have experienced “big” troubles causing time losses of the order of an hour or more in a week about once in two months. Since experience indicates that addition of new “functions” to the source language will be a continuing need we should assume such troubles will occur on any system.

It is most desirable that whenever any observable malfunction occurs, the system gives the user information as to whether the delay will be a big one as soon as feasible. If, as is more than likely the case, the user is scheduled into a definite time slot of a half-hour or an hour, he would often prefer to just leave
if the delay is going to be an extended one. For a user at a remote station it is desirable to have a status query capability in the system, so that the user can easily learn the status of the system. On the rare but big malfunction which takes more than a few minutes for recovery, an “answering service” for status queries from terminals is also valuable.

In conclusion I repeat our views of requirements have been developed in a certain framework of computer size, terminal capability, student population, and role of the computer in instruction; we are aware of a wide range of potential applications of CAI to which our experience is only partly applicable.