Modeling, measurement and computer power*

by G. ESTRIN, R. R. MUNTZ and R. C. UZGALIS

University of California
Los Angeles, California

INTRODUCTION

Since the early 1960s the literature reveals increasing concern with effectiveness of information processing systems and our ability to predict influences of system parameters. A recent survey paper discusses methods of performance evaluation related to three practical goals: selection of the best among several existing systems; design of a not-yet existing system; and analysis of an existing accessible system. The classification of goals is useful, but we can point to neither the models nor the measures nor the measurement tools to allow reliable judgments with respect to those three important goals at this time.

We choose to discuss three issues which do not fall cleanly into Lucas' categories but which are certain to influence our ability to evaluate computer systems in the 1970s. The three issues are: effectiveness of models of computer systems; requirements to be met by measurement experiments; and application of modeling and measurement to the user interface with computer systems.

The first section provides a context for the other sections by reviewing parameters which make computing systems more or less powerful. The second section gives a critique of the state of modeling. The third section characterizes measurement tools. The fourth section discusses the role of measurement at the user interface.

COMPUTER POWER

We consider a computer system to be composed of: a centralized hardware configuration; a set of terminals for entry and exit of user programs and data; an operating system; public programs and data bases; user programs and data; and users and user protocol for entry and exit.

There is no accepted measure for global power or performance of computer systems. There is even no accepted measure for computer cost. Only when a subsystem or subfunction is isolated does it become possible to determine key parameters. However, it is useful to hypothesize such measures and consider influences on them.

Let us, therefore, define a conceptual measure which we call computer system power, $P$, as a multivariate polynomial function whose coefficients are significance weights. We would, of course, like to have a set of orthogonal functions whose independent variables correspond to measurable parameters but that state of happiness is not apparently within reach. In an attempt to exemplify our philosophy, the authors discuss a set of variables which should influence $P$ keeping in mind that derivation of a figure of merit would require dividing $P$ by some measure of cost.

We intuitively expect computer system power to increase if the:

- execution time of any CPU instruction is decreased
- access time of any memory subsystem is decreased
- transfer rate to or from any memory subsystem is increased
- transmission rate of any bus structure is increased
- transfer rate to and from any input or output device is increased
- delay in resource availability is decreased
- error recovery time is decreased
- number of useful public programs is increased
- performance of any public program is increased
- access time to any public data base is decreased
- arrival, execution, and departure rates of user programs are increased
- execution time or resource requirement of any user program is decreased

* This research was supported by the National Science Foundation, Grant No. GJ 809.
number of effective users increases
amount of protocol for any user decreases

In a deeper, even more qualitative sense, we expect a computer system to be more powerful if the following conditions hold:

- system manager has a model permitting adaptation to changing load
- errors and system imbalances are reported to maintainers and developers
- program documentation and measurements permit modification with few side effects
- average number of user runs before correct execution is decreased
- the quality of any user program increases in the sense that there is more effective use of a source language on a given computer system.

Although the above observations are useful in stating expected events of concern they ignore interactions between such events and give no indication of weighted importance of the individual events. We further characterize our systems by the following simple remarks.

If the time required for every physical transition to reach its new stable state were halved, we would expect throughput of the system to double. If only some of the events were reduced in transition time, we could no longer guarantee that there would be a reduction in computation time because the scheduling of events is a function of starting and stopping times of concurrent processes. Anti-intuitive anomalies\textsuperscript{23,3} are disturbing but do not keep us from conjecturing that they occur only infrequently. If we neglect anomalies, then we cannot expect change in execution time of any one instruction or any one routine or any one compiler to produce a decimal order of magnitude change in a sensibly weighted function of the above parameters. Given reasonable measurement tools and design of measurement experiments we conjecture that somewhere between 10 percent and 50 percent improvement in performance can be accomplished for most systems by changes in assignment and sequencing of resources. Although these percentages do not seem dramatic in their impact, the absolute number of dollars or number of computer hours which would become available is far from negligible.

In contrast with the heuristic probing and tuning of a given system, much greater impact is possible at the user interface with a computer system and by advances in models, particularly validated models of our computer systems. For example, we would guess that there are more than 10 attempts to run a program during its development before it runs once "correctly." For complex programs the ratio of number-of-correct-runs to number-of-runs can approach zero. Hence, if the user interface can be altered so as to increase the probability of a correct run, large benefits may result.

The effect of model development is a more sophisticated and qualitative issue. It is self evident that to the extent that we can predict behavior of even a subsystem through modeling, we can hope to isolate important parameters and the way they affect performance. In fact, only through modeling efforts can we generalize experimental results at one center to apply to many others. Furthermore, it has been recognized that simulation is the most widely used tool in evaluation of systems. If simulation depends upon precise imitation of a computer system, its development cost is generally prohibitive and it is fraught with all the unreliability associated with one-shot development. Effective simulation depends upon validated approximate models of systems and of user programs. Creation of such strong models is the most difficult of our tasks. However, the very process of validating or invalidating simplifying assumptions used in models can lead to new algorithms and improved models. Margolin, Parmelee and Sehatzoff\textsuperscript{39} very competently demonstrate this effect in their recent study of free-storage management algorithms.

In this section we have taken cognizance of the fact that there is no simple (or even complex) formula for computer performance. The reader's attention has been focussed on the last five in the list of factors affecting computer performance because they offer so much more return. The following sections review work in analytic modeling, measurement, and the user interface.

CRITIQUE OF ANALYTIC MODELING

Any system design, any measurement project or any resource allocation strategy is based on some conception of the environment in which it operates. That conception is a model. It is beneficial to have such models explicitly stated so that they can be explored, tested, criticized and revised. Even better, though not often achieved to the extent desired, is a formal analysis of the models.

Models and methods of analysis vary greatly. Our concern here is with probabilistic models of systems and processes and also with discrete graph models of programs. The goals of these analyses are both insight and quantitative results to influence the design of
systems, resource allocation strategies and possibly the design of languages.

While most will argue that the goals of such analyses are inherently worthwhile and must be pursued, there is widespread dissatisfaction with the current state of the field. Basically, there are three major areas of dissatisfaction. First, the models are generally oversimplified in order to make them mathematically tractable. This obviously makes the results questionable and brings us to the second major failing which is that analytic results are often not validated by measurement or simulation. Moreover, in cases where system evaluation studies are carried out, the existing models do not seem powerful enough to provide a uniform basis for measurements. The third major criticism is that most of the literature on analytic modeling is a collection of analyses of specialized models. This points up the lack of very general powerful results which would allow analysis to become an engineering tool. As it is now, each new situation almost always requires a separate analysis by an expert.

While the above are substantial criticisms, this is not to say that analysis has not had its impact. We can cite, for example, the working set model of program behavior, the work on stack algorithms, studies of time-sharing and multiprogramming system resource allocation and analyses of I/O scheduling, the work on data transmission systems and on networks and the work on graph models of programs.

**Promising areas of research**

**Multiple resource models**

Much analytic work has dealt with single resource models. The reason for this is clearly that most of the analytic tools which are available apply to single resource environments. The computer system analyst is typically not a mathematician developing new tools but is generally engaged in applying existing tools. Nevertheless, computer systems are multiple resource systems and we must learn to analyze such systems.

Some recent studies of multiple resource models of computer systems have been made using results by Gordon and Newell. The general model considered by Gordon and Newell is one in which customers (or jobs) require only one resource at a time, but move from one resource to another. An example is illustrated in Figure 1 for three resources.

The nodes in this figure represent resources and the arcs represent possible transitions from one resource to another. When a customer has finished at resource $i$ he moves to (requires) resource $j$ next with probability $P_{ij}$. The arcs are labeled with these probabilities. The service time at each resource is assumed to be exponentially distributed. This is a closed system meaning that the number of customers in the system remains fixed. Gordon and Newell have found expressions for the equilibrium distribution of customers in service or queued at each resource. This allows one, for example, to calculate the utilization of the various resources.

Moore and Buzen have applied this model to multiprogramming systems. Moore measured the MTS system to obtain the transition probabilities and mean service times of the resources and then used the model to estimate system parameters such as resource utilizations. The relatively close agreement to measured system parameters leads one to believe that the model can be used to predict the effect of some changes in system configuration. In using the model in this way, one must be careful that the proposed changes do not significantly affect the basic behavior of the customers. Buzen used the same model to gain insight into resource allocation in multiple resource models of computer systems. His studies include the investigation of buffering and the effects of paging algorithms. Both Moore and Buzen have used the model to try to give a meaningful formal definition to the term "bottleneck." It is of interest that they arrive at different definitions of a bottleneck. The reader is referred to the references for details.
While the studies mentioned above are clearly advances in the study of computer system models there are numerous open questions. For example, the model does not allow the representation of the simultaneous use of several resources such as memory and CPU. Also there is no means for representing the synchronization of events such as a process doing buffered I/O. Another limitation is that the customers in the system are assumed to have the same statistical behavior, i.e., the transition probabilities and service time distribution are the same for all customers.

**Bounds and approximations**

Every evaluation technique makes use of approximations. These approximations may arise, for example: in estimating the system parameters, user and program behavior; or in simplifying the model of the system itself. There is clearly a tradeoff between types of approximations. By simplifying a model one might be able to handle more general types of user and program behavior. Much of the analytic work has been concerned with exact mathematical solutions to models which are themselves gross approximations.

An area which is beginning to be explored is that of approximate solutions to more general models. For example, Gaver has used the diffusion approximation for the analysis of heavily loaded resources in queueing studies. The basic technique is to consider that the work arrival process is not the arrival of discrete customers requiring service but rather a work arrival flow. This work arrival flow is a continuous process with the same mean and variance as the original process. Another example of the use of approximations is the work by Kimbleton and Moore on the analysis of systems with a limiting resource.

It is clear that the use of any approximation requires validation of the results. This may take the form of comparing results with measurements of an actual system, simulation, or obtaining bounds on the error in results. Bounds may also be applied in a different manner. Much has been written on the analysis of time-sharing scheduling algorithms and their effects on response times. Kleinrock, Muntz and Hsu have reported on results which in effect demonstrate the bounds on response time characteristics for any CPU scheduling algorithm which does not make use of a priori knowledge of customers service times. The importance of the bounds is that one can see the limits of the variation in response characteristics that are possible by varying the scheduling algorithm and the extent to which these limits have been approached.

**Program behavior**

A major problem that must be dealt with in any evaluation effort concerned with computer systems is program behavior. Even when using approaches such as benchmarking or trace-driven modeling there is the problem of selection of programs which are in some sense representative of the total population of programs that will be run on the system.

Studies of memory management in particular have had to explicitly include models of program behavior. The early work in this area stressed very general but powerful aspects of program behavior such as "locality" and "working set." More recent work deals with more explicit models of the generation of reference strings which assume more about program behavior but correspondingly allow for more detailed analysis. It is hoped that these models will permit more detailed studies of multiprogramming and procedure sharing.

It is interesting to note that the bulk of this work has been directed toward finding models which can represent the universe of possible programs. More particularly, the goals of this research have been to isolate parameters characterizing program behavior to which memory management is sensitive and to compare the effectiveness of various memory management strategies. This approach is in line with a common theme which runs through most of the work on resource allocation strategies in computer systems. That is, we see allocation strategies attempting to work well over the total population of programs possibly utilizing measurements of recent past history of the process to predict the near future. Outside of work arising from graph models of parallel programs very little has been done to utilize a priori information about a process. Many systems do make a priori distinctions between batch and interactive processes. It seems reasonable though that much more information may be available which would be useful in allocating resources. For example, it has been suggested that the time-slice and paging algorithm parameters be tailored to the process. Use of a priori information assumes that the process is available for analysis prior to execution. This is a valid assumption for production jobs, system processes, and to some degree for all jobs at compile time. Since these processes consume a significant portion of the system resources, gains in efficiency in managing such processes might result in major gains in total efficiency. There are many open problems associated with this approach:

1. Is there a conflict with a program design goal of program modularity? How is information about separately compiled procedures to be combined?
2. Should processes be permitted to advise the system as to their resource needs? How does the system protect itself against false information?

3. How to manage resources effectively for processes which provide a priori information, and also for processes without associated a priori information?

4. What kind of a priori information is actually useful to management of a system: how costly is it to obtain and utilize effectively?

5. How predictable are the resource requirements of processes?

While this approach has received only some slight mention in the literature, it appears to be a fertile area for research.

Graph models of programs provide an abstraction of program structure governing flow of control and demand for resources. They permit a representation fitting somewhere between the full detail of actual programs and parametric or stochastic representations of them. Most work using graph models has been concerned with concurrent processing. However, the graph model analyses explicitly reveal sets of independent tasks which become candidates for alternate sequencing in sequential systems.

Ideally, we search for models of systems and program behavior which provide principles guiding synthesis of configurations along with well founded resource management strategies. Measurement must validate effectiveness of such strategies. The diversity of computations further demands that measured parameters be provided to operating systems and users in order to permit adaptation to dynamic variations in system behavior and to unavoidable anomalies in systems and languages.

Studies during the latter half of the '60s showed how little attention had been given to measurability in the man-made universe of computer systems. The next section characterizes some of the problems in measurement.

**MEASUREMENT OF INFORMATION PROCESSING SYSTEMS**

Tools for measurement of computer systems must satisfy all of the following requirements: detection of prescribed events; recording of detected events; retrieval of accumulated records; data reduction; and display.

We comment on each in turn.

**Detection**

We start by rejecting the absurdity of observing all of the states of a system under observation since it would imply detecting the state of every input, every memory element and every output every time there was a change, along with the time at which the change occurred. Hence, any set of measurement tools must include means of selecting a subset of system states.

Hardware measurement tools provide a prescribed number of sensing probes which may be physically placed on selected register or bus points in a machine under observation. Measurement system registers along with programmed comparators and basic logical operations permit further filtering by allowing detection of a subset of the events sensed by the probes. Even with such filtering the rate of change of detected states may be excessive. If the response time of hardware measurement elements is insufficient, basic circuit changes would be required to make the measurement feasible. If bandwidth is insufficient, it is sometimes possible to introduce a sampling signal and thereby further reduce the number of detected events. In the absence of interaction with software monitor programs, a hardware monitor is clearly limited in its utility. To be convinced of this, one need only consider the kind of program status information change which is observable by probes only when it appears in the form of an operand during the course of computation. Hardware detection can have the virtue of introducing no artifact into the measured system and of being able to detect events whose states are not accessible to measurement programs. Sampled detection may be made more effective by allowing interference with the observed process. If a sampling signal enforces a proper interruption, observed data may be sequentially sensed by detection circuits. The recently reported “Neurotron” monitor is the most interesting implemented hardware monitor, and its design shows the foresight of enabling interaction with software monitor programs.

Software measurement tools consist of programs which detect selected events by virtue of their insertion at state-change points in the sequential computational process. This detection process introduces artifact in execution time, in space required for measurement program storage, and sometimes (e.g., synchronization with asynchronous cyclic processes) in qualitative side effects on existing computational processes. In a sampling mode, measurement programs can have their in-line artifact reduced by disturbing the flow of computation only at a sampling time. At a sampling time, measurement programs may be brought in to check as large a set of memory states as is needed and
then control is returned to the observed system. In the absence of hardware support, a software monitor is limited to observation of those system states which have affected memory contents. In one case, careful analysis of measurement of PL/I functions of an IBM 360/91 revealed anomalies in recorded system states which can best be characterized as artifact introduced by OS/360 when it inserts code associated with I/O interrupts into the code being measured.

It has become clear that we are not faced with mutually exclusive alternatives of hardware detection tools or software detection tools. Rather how much of each; how they are integrated; and how they are made available to experimenters. A paper by Nemeth and Rovner presents a pleasing example of the power of combined hardware and software in the hands of a user. They point out that facilities introduced for hardware debugging are often the kind useful in later program measurements.

**Recording**

If an event of interest has been detected, its occurrence must affect memory contents. Such action may be as simple as incrementing a counter or as complex as storing a lot of state information for later analysis. In the case of nondisturbing hardware measurements, external storage must be provided and the transfer rate must be able to keep up with the rate of change-of-state information observed by a set of probes and associated circuits. In the case of software measurements, sufficient memory space must either be provided to record all relevant state information, or else preprocessing reduction programs must be called in to reduce storage requirements.

**Retrieval**

In the construction of any large system, both a data gathering and retrieval system must be incorporated into the basic design. Failure to do so will limit the amount of instrumentation available later when efficiency questions arise. For example, in a large programming system which has transient program segments, data gathering is easily inserted into any program segment; however, unless a standard data storing program is available, the data gathered cannot be easily retrieved. The IBM PL/I F-level compiler is an example of a programming system broken into transient program segments. It fails to have a data storing program with adequate bandwidth to support meaningful measurement activity.

**Data Reduction**

The amount and kind of data reduction is determined by the goal of the measurement experiment and limitations of measurement tool capabilities in detection, recording and preparation for retrieval. For example, assume that we want to obtain a history of utilization of routines in order to decide which should be kept in primary storage and which should be kept in backup storage. Assume, further, that every time a routine is called: the name of the called routine, the time of day, and the name of the user is recorded. It would not be very meaningful to generate only a history showing the times at which each routine in the system was used by each user. Data reduction would be required to determine, for example, the total number of such uses, an ordering of routines by number of uses, a determination of the number of routines involved in, say 50 percent, of the uses and their names, etc.

**Display**

The goal of the data reduction process is defined by the specified form of display or feedback to the experimenter. If measurement is being made for feedback to an operating system for use in resource allocation, parameter values must be delivered to memory cells to be accessed by the operating system. If measurement is made for accounting purposes or, more generally, to provide the user with feedback about his quality of use and system response, results should be merged into user and system manager files. If measurement is made for operator control and management, simple alphanumeric displays are common. For experimental analysis of system behavior, CRT displays, graphs and computer printout are generally required.

**Measurement Methodology**

The complexity of computer systems dictates special care in the planning of measurement experiments. If the results of experiments are not reproducible, they are of little value. If any assumptions made are not recorded and validated, the results cannot be generalized and applied at another time or place. A large body of statistical theory is available providing methods for abstracting properties out of individual data points, but applicability must be carefully checked. We have little hope of adhering to principles if we do not have a measurement language to prescribe measurement experiments as sequences of commented operations which are appropriately integrated with observed data. The latter step needs wait upon creative development of
measurement tools and their test in meaningful experiments. Measurement capability must be explicitly included during periods of system design and must be available for inclusion in user program design. Digital computer systems and programs are properly characterized as complexes of very elementary functions. Full systems or programs, therefore, generally require partition in order to manage the synthesis process. Each partition introduces possible measures of validity of output, of performance and of cost. Means for measurement should be checked at that point in design and a value judgment made if excessive artifact would be introduced by the measurement process.

If a system contains the structure and primitive operations satisfying the five requirements discussed in this section, it carries the tools for adaptability. We conjecture that much more than the 10 percent to 50 percent improvement alluded to in the Introduction becomes attainable—particularly when measurement tools can influence user behavior.

**COMPUTER POWER AND USER INTERFACE**

In the seventies some stronger effort must be directed toward increasing computer power by a reduction of the complexity of the user interface.

Operating systems and **Higher Level Languages** are tools designed to give the user control of the portion of a computer system he needs. With the exception of work done at SDC, little reported effort has been devoted to the human engineering aspects of these tools. During the last decade, while hardware made a dramatic increase in power, the management tasks required of the operating system increased from trivial to highly complex. At the same time, the users were required to supply (in unnatural form like JCL) more of the parameters which would allow effective management decisions to be made by the operating system. These user-supplied parameters have increased the burden of complexity at the user interface—and reduced the amount of useful work a user can accomplish in a given period of time.

For example, much of the attraction of APL/360 is its simplification of the operating system interface along with the addition of immediate execution of its concise powerful primitives. A batch oriented FORTRAN user perceives this as a tremendous increase in his computer power. A more sophisticated user might see APL as a powerful desk calculator which provides immediate access to functions similar to functions he already commands, less accessibly, in other languages.

Another user interface problem exists at the level of higher level languages. As more advanced hardware becomes available to the user, he seeks to solve more complex problems. When a problem grows beyond a manageable point, the user segments the problem into pieces plus associated linkages. In doing so, however, he introduces a new set of communication problems; a change in one program which affects an interface can now wreak havoc in another “completed” portion of the problem solution. Higher level languages have been lax in the types of program interconnections (and interactions) allowed.

An example of the problems of creating a large programming system are reported by Belady and Lehman using data from the development of OS/360. While this study concerned programs written in assembly language for the IBM 360, the properties which produce the error rates and modification ratios reported in their paper are characteristics of all large programming systems today.

Several techniques for improving the probabilities that a program can be made error free are available in the literature. One of the earliest is Dijkstra’s “Notes on Structured Programming,” and also “THE Programming System.” His system breaks the problem into shells of “pearls” or “primitive” operations. Each shell is built using the “primitives” of the next lower level. This system attempts to minimize interactions, forces the programmer to produce generalized functions which can be tested, and allows easy instrumentation because of the segregation of functions.

Some disadvantages of such a hierarchical scheme make its practical application difficult. Such a scheme increases initial development time because it forces developers to completely understand the structure of the system being built and to redefine the proper function for each hierarchy. Transitions between levels may be costly. Functions at the lowest level are the most general and, therefore, the most frequently used. Small inefficiencies in these functions, or in the method of traversing levels of the structural hierarchy, magnify costs dramatically and force the user away from centralized functions. This defeats the original purpose of the organization.

Another disadvantage of a hierarchical scheme is that while instrumentation of the system is easy, interpretation of the measurements is generally not. Measurement results could change drastically if the organization of the program were modified. Therefore, it is hard to tell how much of what goes on is due to the structural hierarchy and how much is due to the intrinsic properties of the program. Such knowledge points a way toward improvement.
### Compile-Time

<table>
<thead>
<tr>
<th>Number of Error Occurrences</th>
<th>Error Type*</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>263</td>
<td>IEM0227</td>
<td>NO FILE/STRING SPECIFIED. SYSIN/SYSPRINT HAS BEEN ASSUMED.</td>
</tr>
<tr>
<td>87</td>
<td>IEM0182</td>
<td>TEXT BEGINNING **** SKIPPED IN OR FOLLOWING STMT NUMBER.</td>
</tr>
<tr>
<td>74</td>
<td>IEM0725</td>
<td>STATEMENT NUMBER xxxx HAS BEEN DELETED DUE TO A SEVERE ERROR NOTED ELSEWHERE.</td>
</tr>
<tr>
<td>63</td>
<td>IEM0152</td>
<td>TEXT BEGINNING **** IN STATEMENT NUMBER xxxx HAS BEEN DELETED.</td>
</tr>
<tr>
<td>46</td>
<td>IEM1790</td>
<td>DATA CONVERSION WILL BE DONE BY SUBROUTINE CALLS.</td>
</tr>
<tr>
<td>39</td>
<td>IEM0185</td>
<td>OPTION IN GET/PUT IS INVALID AND HAS BEEN DELETED.</td>
</tr>
<tr>
<td>27</td>
<td>IEM0677</td>
<td>ILLEGAL PARENTHESIZED LIST IN STATEMENT NUMBER xxxx FOLLOWS AN IDENTIFIER WHICH IS NOT A FUNCTION OR ARRAY.</td>
</tr>
<tr>
<td>27</td>
<td>IEM0109</td>
<td>TEXT BEGINNING **** IN OR FOLLOWING STATEMENT NUMBER xxxx HAS BEEN DELETED.</td>
</tr>
<tr>
<td>25</td>
<td>IEM0096</td>
<td>SEMICOLON NOT FOUND WHEN EXPECTED IN STATEMENT xxxx. ONE HAS BEEN INSERTED.</td>
</tr>
<tr>
<td>23</td>
<td>IEM0673</td>
<td>INVALID USE OF FUNCTION NAME ON LEFT HAND SIDE OF EQUAL SYMBOL OR IN REPLY, KEYTO OR STRING OPTION.</td>
</tr>
</tbody>
</table>

### Execution-Time

<table>
<thead>
<tr>
<th>Number of Error Occurrences</th>
<th>Error Type*</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>IHE0804</td>
<td>ADDRESSING INTERRUPT.</td>
</tr>
<tr>
<td>12</td>
<td>IHE320</td>
<td>FIXED OVERFLOW.</td>
</tr>
<tr>
<td>8</td>
<td>IHE140</td>
<td>FILE name—END OF FILE ENCOUNTERED.</td>
</tr>
<tr>
<td>7</td>
<td>IHE004</td>
<td>ERROR IN CONVERSION FROM CHARACTER STRING TO ARITHMETIC.</td>
</tr>
</tbody>
</table>


Figure 2a—Most frequent PL/I errors

---

Figure 2b—Average persistence sorted by average persistence
Present studies of forced program structure and program proofs of correctness may begin to provide models on which HLL designers may base their proposals. However, any major changes should be designed to improve the user's system so that each program submittal can be a learning experience. In this way a programming system can be called upon to point out unusual events; draw the programmer's attention toward possible errors; and yet, not produce volumes of output which would be costly to print and which a programmer would refuse to read.

Work at Cornell toward producing compilers which correct human failings rather than punish them has culminated in a highly functional, rapid compiling, and very permissive PL/I student-oriented compiler called PL/C. This compiler does spelling correction of keywords, automatic insertion of missing punctuation, etc. In addition automatic collection of some statistics is done at execution time. For example, each label causes a count to be maintained of the number of times execution passed through that labeled statement. Compilers such as these increase computer power by reducing the complexity of the user interface.

Implementations of HLLs could further help a programmer by giving an optional cross reference listing showing locations where a variable is changed, could be changed, or just referenced. Items could be flagged if they were never referenced or set; only referenced; or only set. In the first two cases spelling correction might be applicable. Statements which use expensive library subroutines or other costly language features could be flagged. Measurement nodes could be easy to insert and operate. These should, in turn, produce meaningful data which relate directly to questions the programmer wanted to ask.

But such discussions have only beat around the bush itself. The real problem, the bush, is the higher level language. The real questions are: What features are error prone? What features of the language allow automatic validity checking of what is written? How can these properties be identified and measured? How can the knowledge of these things be used to reduce complexity of the user interface so that the user perceives an increase in his computer power? Which language constructs are seldom used, adding unnecessary complexity and unreliability?

Efforts to measure the human engineering aspects of computer language use and to provide feedback
into the design stages of higher level languages and the control and command languages of the operating system may provide major increases in computer power by:

- increasing the number of users who can bring problems to the machine
- decreasing the number of problem submissions necessary to bring a job to completion

Work is in progress at SDC, moving toward a man-machine symbiosis. These little publicized approaches to measurement of human problem solving and computer languages have just begun to scratch the surface of this very important area. Work at UCLA has attempted to identify properties of PL/I which are prone to human error. As a first approximation, error rates and persistence curves of various errors identified in students' use of the IBM PL/I F-level compiler is presented in Figure 2. Corresponding results for errors found by Cornell's PL/C compiler are presented in Figure 3. Figure 2a shows a table of the number of occurrences of the most frequent PL/I error types recorded during compilation and during execution times. Figure 2b displays the persistence of errors by PL/I type during the student runs. The vertical coordinate is the error type ordered by the magnitude of PERSISTENCE RATIO. The hori-
zontal coordinate is the PERSISTENCE RATIO and was calculated as an average of (number of sequential trials during which the particular error persisted) divided by the total number of trials. If an error type did not occur in at least 5 problem assignments it was arbitrarily deleted to keep the displayed range of values reasonable. Figures 3a and 3b display the same properties for assignments using PL/C. A total of 128 problem assignments completed by 28 students are included in the statistics. Follow-up work is intended to lead more deeply into language design and hopefully into new techniques for automatically localizing errors in a program.

The basic technique for doing this is to allow the programmer to specify more information than the HLL processor needs to compile the program. An example would be identifiers of the form “CONSTANT” in a PL/I data attribute syntax. CONSTANTs as opposed to variables, would only be set by an initial attribute and would be illegal as a left-hand side of an assignment or as an attribute of a pseudo-variable. In addition, the program could be considered as having this attribute. At several points in a program (e.g., block exit time) these constants would be checked to see if their value had changed. If any had, a warning would be printed; the correct value restored; and the program would continue. Such a new PL/I data type allows automatic checking for consistency to localize errors and yet is almost painless for a programmer to use. When a program is debugged, it is easy to turn off this kind of checking for the sake of more efficient performance. In a hardware environment like the MULTICS GE 645, these errors can be detected dynamically when illegal accesses occur.

Debugging should be designed into the language and taught as part of the language, because the majority of the time a programmer deals with a language, he is also dealing with an incorrect program. Subscript checking, trace information, validity checking at periodic intervals, time and count information, formatted displays of all program information and selective store and fetch recording are the kinds of things which should be available to the HLL programmer.

In addition, measurement tools should be immediately accessible to any user without burdening others, so that if questions of efficiency are raised they can be answered simply and quickly. Some of the measurement tools which seem important are: (1) flow charts or tables as optional output which would stress intermodule dependencies; (2) time and count control statements which could be output, reset and inactivated under program control, and would create output automatically if the program terminated abnormally or without expressly outputting the data gathered; (3) program size should be easily accessible by the program dynamically and summaries of available space should be available at program termination.

In order to increase the complexity of the programming problems which users can handle, languages must be allowed to accommodate personal ways of expressing concepts. To do this, at the very minimum, new data types should be available for the programmer to define as well as operators which use these data types. This begins to syntactically approach the Dijkstra concepts and to allow easier application of hierarchically structured programs. Hopefully these approaches will increase the user’s computer power by making the development of his programs easier.

The programming system itself should be restructured so that more information is available to a HLL processor. Figure 4a shows the diagram of information flow in a usual batch oriented system. The source code is compiled; the resulting object code is passed to the accretion step where library or previously compiled programs are added to it; and the resulting load module
is passed to the execution phase; finally, output from execution is passed back to the user. To provide more automated feedback, Figure 4b shows information flow in a system where statistical summaries of one execution are available to the next compilation. In addition, the accretion step spends much more of its time checking linkage conventions and the validity of argument-parameter choices. This programming system has an edit/compile time library which is designed to help make changes easy. For example, it keeps “COMMON” declarations uniform (read EXTERNAL if you are a PL/I programmer) and it also uses information from the compiler to point the user at syntax errors and information from the execution phase to point the user at semantic errors.

Such modifications can reduce errors and speed the development of programs by improving communication between what are now considered separate program steps. However, the most important changes, across all the proposed modifications, are those changes which will allow the programmer to receive only those pieces of information relevant to the level at which he is programming (i.e., making changes). This would provide dynamic help; help where the programming language acts as an extension of the users’ mind to assist in problem solving and optimization.

It is important to view these changes which move toward dynamic assistance in terms of costs. Each change must cost something in execution time overhead. Some of the more powerful features like selective fetch and store monitoring must be expensive. However, if these features were found valuable, then modification to hardware might diminish costs dramatically. Integrating these techniques into HLLs must be inherently costly because implementation and testing of human interaction with these diagnostic features are difficult to execute in any controlled way—much work must rest on subjective evaluation of users’ behavior. Integration of aids into HLL translators must be initially done without those very aids which are deemed necessary to help programmers modify programs. Therefore, any change is fraught with risks caused by lack of checks in current systems. Obviously bootstrapping is called for and we can expect many passes before achieving effective tools.

The development of richer higher level languages on the one hand, and the development of debugging services and error correcting compilers on the other, exert forces in the direction of increasing performance at the user interface. With appropriate use of models and measurement much more improvement may be obtained.

SUMMARY

Computer systems are different from other systems by virtue of the dynamic fashion in which our comprehension of their behavior may be built into their operation. If validated models are developed, they may then be built into the system itself to serve adaptive resource allocation algorithms. If measurement tools are effectively integrated, they may be made available to the user to improve the quality of his use of programming languages. If the user is, in fact, a team developing programming systems, the modeling and measurement facilities may serve to make much more complex programs possible because a model of programs being built is, itself, generally too complex for a group of unaided humans to manage in an error free way. In the above paper we have sought to open up these questions.

We have not had much experience with effective modeling and measurement. There is an immense amount of data to be observed in a computer system. Cost-effectiveness of performance measurement must be considered. As one of our reviewers put it, “This reviewer has seen some measurement studies lead to system improvements which will pay off sometime in 2018.” Hopefully the 1970s will see more effective modeling and measurement introduced into the design process and selectively carried into developed systems to help both internal process management and the enrichment of external use through the user interface.

REFERENCES

1 R A ASCHENBRENNER L AMIOT
N K NATARAJAN
The neurotron monitor system

2 L A BELADY
A study of replacement algorithms for a virtual storage computer
IBM Systems Journal Vol 5 No 2 pp 78-101 1966

3 L A BELADY R A NELSON G S SHEDLER
An anomaly in the space-time characteristics of certain programs running in paging machines
Communications of the ACM Vol 12 No 6 pp 349-353 June 1969

4 L A BELADY M M LEHMAN
Programming system dynamics or the meta-dynamics of systems in maintenance and growth
IBM Report No L.C 3546 September 1971

5 D P BOVET G ESTRIN
A dynamic memory allocation in computer systems
IEEE Transactions on Computers Vol C-19 No 5 pp 403-411 May 1970
35 L KLEINROCK
Analytic and simulation methods in computer network design
AFIPS Conference Proceedings Vol 36 pp 509-579 1970
Spring Joint Computer Conference Atlantic City New Jersey May 1970

36 L KLEINROCK R R MUNTZ J HSU
Tight bounds on the average response time for time-shared computer systems
Proceedings of the IFIP Congress 71 TA-2 pp 50-58
Ljubljana Yugoslavia August 1971

37 D E KNUTH
An empirical study of FORTRAN programs
Computer Science Department Report No CS-186 Stanford University 1971

38 H C LUCAS JR
Performance evaluation and monitoring
ACM Computing Surveys Vol 3 No 3 pp 79-91 September 1971

39 B H MARGOLIN P P PARMELEE M CHATZOFF
Analysis of free-storage algorithms
IBM Systems Journal Vol 10 No 4 pp 283-304 1971

40 D F MARTIN
The automatic assignment and sequencing of computations on parallel processor systems

41 R MATTSON J GECSEI D SLUTZ I TRAIGER
Evaluation techniques for storage hierarchies
IBM Systems Journal Vol 9 No 2 pp 78-117 1970

42 J M McKinney
A survey of analytical time-sharing models
Computing Surveys Vol 1 No 2 pp 105-116 June 1969

43 C G MOORE
Network models for large-scale time-sharing systems
PhD Dissertation University of Michigan April 1971

44 H MORGAN R A WAGNER
PL/C—The design of a high performance compiler for PL/I
AFIPS Conference Proceedings Vol 38 pp 503-510 1971
Spring Joint Computer Conference Atlantic City New Jersey May 1971

45 A G NEMETH P D ROVNER
User program measurement in a time-shared environment
Communication of the ACM Vol 14 No 10 pp 661-666 October 1971

46 J RODRIGUEZ-ROSELL
Experimental data on how program behavior affects the choice of scheduler parameters
Proceedings of the Third Symposium on Operating Systems Principles pp 156-163 Stanford University October 1971

47 E C RUSSELL G ESTRIN
Measurement based automatic analysis of FORTRAN programs
AFIPS Conference Proceedings Vol 34 pp 723-732 1969
Spring Joint Computer Conference Boston Massachusetts May 1969

48 H SACKMAN
Man-computer problem solving
Auerbach Publishers Inc 1970

49 T J TEOREY T B PINKERTON
A comparative analysis of disk scheduling policies
Proceedings of the Third Symposium on Operating Systems Principles pp 114-121 Stanford University October 1971

50 R C UZGALIS J ALLEN
360 model 91 execution times of selected PL/I statements
Modeling and Measurement Note No 7A September 1971
360 assembly language source code for selected PL/I statements with model 91 execution times
Modeling and Measurement Note No 7B September 1971
Computer Science Department University of California Los Angeles

51 S A VOLANSKY
Graph model analysis and implementation of computational sequences