Gaines, B. R. (1973). Computer technology and its utilization: today and tomorrow. In Proceedings of National Engineering Laboratory Conference on Small Computer Applications in Industry, East Kilbride, UK.

COMPUTER TECHNOLOGY AND ITS UTILIZATION

TODAY AND TOMORROW

B.R. Gaines

Department of Electrical Engineering Science
University of Essex

Sections

1. Introduction

2. Historic Perspective

3. The Inter-relationship of Machine, Problem and System Designer

   3.1 The Designer-Problem Relationship

   3.2 The Machine-Problem Relationship

   3.3 The Machine-Designer Relationship

   3.4 Computer-aided Design of Software

4. Summary and Conclusions

Paper presented at:

National Engineering Laboratory Conference
"Small Computer Applications in Industry"

Glasgow 27-29 March 1973

COMPUTER TECHNOLOGY AND ITS UTILIZATION: TODAY AND TOMORROW

B.R. Gaines*

1. Introduction

The pace of advance in computer technology is so rapid that a major problem for industry is to maintain a comparable rate of advance in computer applications. If reliability and power continue to increase, and price, size and power consumption continue to fall, there is a strong incentive to sit on the fence and wait for a stable technology. However, will it come? - when? - and what will it be? In the meantime what strategies can be adopted to take advantages of minicomputer technology without suffering from the inherent obsolescence generated by rapid change.

It would be absurd to pretend that there are definite answers to these questions. However, an assessment of probabilities is possible through an analysis of the case histories of current technologies and of projected future machines. This paper presents a number of salient factors relating to computer technology and its applications. It is not a guide to machine selection but does attempt to raise those questions that should be asked in evaluating computers and proposed computer applications. In particular, the three-part relationship between system designer, problem and machine is examined in some detail to give the hardware developments an overall applications perspective.

2. Historic Perspective

A brief review of the historic development of computers shows the effects of two major factors: (a) technical improvement; (b) application development, that is 'hardware technology' and 'software technology' respectively. The first is readily understood as progress in components leading to increasing reliability and decreasing cost (although the discontinuities in potential applications as these variables go through critical values should not be underestimated). The factor of 'application development', however, is one which is less readily comprehended because, even though 'knowhow' plays a role in all industries, it has a unique dominance in computer applications resulting from extreme versatility and lack of problem-orientation of the computer itself. A technology which is in itself so universally applicable gives no clearly defined indication of the way in which it should be used to solve a particular problem or implement a particular system.

Two examples will serve to illustrate the importance of applications knowhow:

(1) Everyone has been taken by surprise by Winograd's results on natural

*Department of Electrical Engineering Science, University of Essex.

language conversational interaction which demonstrate the potential for a virtually man-to-man relationship with the computer, given a real topic to discuss and suitable software technology with which to encode algorithms supporting discussion. However, no new hardware is involved and these programs could have been run over a decade ago.

(2) Even more remarkable in demonstrating that the 'limitations' of computer may often be overcome dramatically by means other than hardware development is the story of the Cooley-Tukey 'Fast Fourier Transformation'. Here we have a situation where midicomputers were in use for routine spectral analysis of transducer records when suddenly, and literally 'at the stroke of a pen', their throughput was multiplied by 1,000 or more. The consequences of the availability of low-cost, high-speed spectral analysis are still not fully worked through and are affecting the basic measurement technology of many industries. However, the lessons of the Fast Fourier Transform (or rather the Slow Fourier Transform accepted and used for so many years) are important in demonstrating that in our analysis of the hardware technology we must not forget the importance of the designer-computer relationship - limitations of computer size or speed may be inherent in the accepted 'solution' rather than the original problem and the computer user should be encouraged to make the imaginative lateral leap which brings the impossible within grasp without any breakthrough in hardware technology.

Having emphasized the importance of software technology and applications knowhow, one may note the progress in the hardware itself. The 1940's were periods of experimental development of temperamental machines whose brief periods of meaningingful activity were garnered for military use. In the 1950's the available up-time although expensive and intermittent was sufficient to feed into scientific research and some brave (or foolhardy) commercial projects. In the early 1960's machine were still in use with meantimes between failures (MTBFs) of 30 minutes or so, but by the late 60's silicon semiconductors had taken the MTBFs of the computer itself to some 1000's of hours.

In the 70's we are seeing an increasing scale of monolithic integration reduce the size, power consumption and cost of machines. By the end of the decade it is safe to predict that the equivalent of the £2000 naked mini of today will be a set of chips costing less than £100, drawing its power from a small cell. Most importantly these chips will make available sensible 'computers' with minimal memory at costs of less than £20. However, the availability of low-cost active logic, as opposed to passive memory, is also beginning to have its affect on machine power, and the £2,000 for the present-day 8K-byte mini with minimal arithmetic capability will then buy a machine with some 100K-bytes of memory and processing power exceeding the larger present-day maxi-computers, such as the IBM 360 and PDP10.

With this reduction in the cost/size/power-consumption/processing-limitations, etc., of the computer itself the pressure of technical development will be applied increasingly to computer peripherals and the later 70's and early 80's will be seen retrospectively as the time of transition to total system development. Computers will be integrated into other systems and the CPU/peripheral distinction will vanish. Electro-mechanical peripherals, particularly bulk storage, will be replaced by all solid-state electronic or electromagnetic systems wherever motive force is not inherently required. By the late 80's the technical problems of man-computer

communication will have been solved by conventional means such as speech communication, to be replaced in the 90's by direct sensory prostheses.

To go further takes us into science fiction regions that will only serve to weaken the argument. However, we have gone far enough to indicate that there is no end in sight to the development of computer technology. He who waits for the bandwagon to stop rolling will only see it disappearing over ever more distant horizons. The important lesson of history is to jump on at the right time - for any application there are critical values of performance parameters below which a system cannot function, but above which further improvements, for that application, make little difference. This applies to all parameters, including cost, size, reliability, and processing power. Real-time applications requiring on-demand access to the computer could not succeed whilst MTBFs were below 100 hours but improvements above 1000 hours have a diminishing effect. Many instrumentation applications did not make sense until a reasonable size processor could be purchased for a few thousand pounds, but further decreases in possible processor cost would create an imbalance in relation to the cost of the associated peripheral systems (whose costs are not decreasing at such a fast rate) and the most competetive product will be one that uses processors of increased capability rather than decreased cost. Similarly commercial office applications have constraints not only of reliability and cost but also of size/power-consumption since special environments are difficult to accommodate, but once a certain level of environmental tolerance is reached further stages are progressively less useful.

Timing is of the essence when dealing with a rapidly developing technology and the vital question is generally not whether to utilize computers but when? The question of timing is not only a predictive one but also a normative one - the first user at a technically feasible time forces others to become users. In the computer field he also introduces a major planning problem of gearing a continuing development into a technology unique for its rapidity of change. In the following section we consider some aspects of this problem.

3. The Inter-relationship of Machine, Problem and System Designer

The diagram of Figure 1 presets the basic paradigm for a computer applications situation. Note that it expresses a three-part relationship between computer, problem and designer. Many problems, both of computer design and application, arise from neglect of one component in this relationship. A machine is not necessarily the best for an application even if it has an excellent architecture suited to the relevant problem class - to actually apply it to that problem class the system designer needs to be able to understand it (at one level optimality often leads to complexity, whilst at another machines such as hybrid analog and DDA are not widely understood) and he also needs adequate developmental tools, the availability of which may be more relevant than any other considerations.

Similarly the concept of 'system analysis' as a relationship only between the systems designer and the problem independent of computer technology is a dangerous one if pursued too far - most human activities are self-optimizing and goal-directed and the current operation of a non-computer-based system will have evolved to make best use of the available technology. The system analyst must attempt to evaluate ends not means, and needs to take great care to avoid stating a partial problem within too limited a frame of reference.

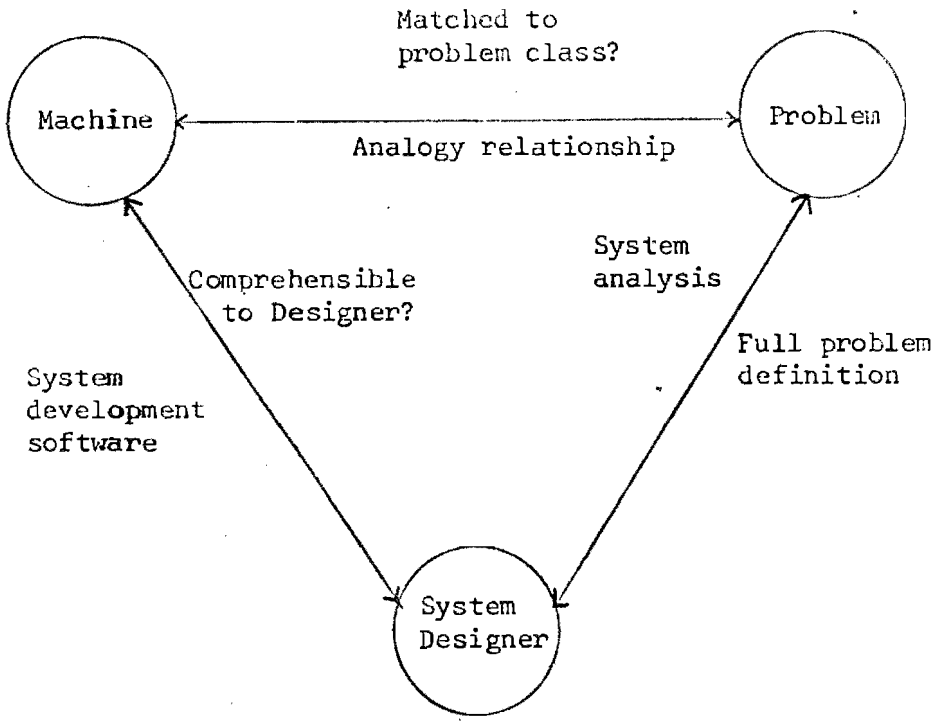

Figure 1. The Three-Part Relationship in Computer Applications

3.1 The Designer-Problem Relationship

Taking the links between each pair of elements in Figure 1 one by one: that between the system designer and the problem appears the least relevant to the theme of this paper since the computer is not in the path. However the designer/problem relationship is clearly of major importance - it is a truism that a vaguely stated, time-variant 'problem' is not suitable for computer solution. Such a problem is probably not suitable for any form of 'solution', but a human organization can at least cope with it and contain it (and probably enjoy it!).

It is generally emphasized that it is necessary to have a full and definitive statement of a problem in order to compose algorithms (essentially mechanical problem-solving processes guaranteed to succeed within a given frame of reference) for its solution which can be implemented on a computer. It is less widely realized that the 'language' in which a problem is framed can profoundly affect the ease with which the algorithms can be generated and implemented and may also determine the suitability of a given machine organization for use in connection with the problem. The link between problem and machine for purposes of implementation passes through the system designer - there is one linguistic transformation for the problem to be defined by him and a further linguistic transformation for the solution to be programmed by him (where 'him' encompasses a multitude of hierarchical structures!). It is quite possible (and indeed quite expected, the more complex 'him' becomes) for the machine-problem relationship to be simple and healthy, but for the problem/system designer transformation to be such a weird imposition that the programmer/machine re-transformation is dominated by factors unrelated to the original

problem - the final result being a complex and abstruse linguistic exercise crushed under its own documentation.

Even though from a hardware technology viewpoint we would concentrate on the relationship between machine and problem and its suitability to the problem area, in practical applications the route through the designer may totally dominate the feasibility and success of a project.

Other papers in this symposium will comment on this aspect of applications definition and programming - we will only note that in relation to machine architecture it makes the path in Figure 1 between machine and designer of at least equal importance to that between machine and problem. Much work on machine architecture is open to the criticism that it pays too little attention to the problems and viewpoint of the system designer/programmer, and much systems software is primarily concerned with hiding or transforming aspects of the machine which are a source of difficulty to the programmer.

3.2 The Machine-Problem Relationship

The link between machine and problem is that about which we generally think when considering the technical suitability of machines to problem, or application areas. At the simplest level, since we know that a small Turing machine is a universal computer, the question of suitability is one only of storage capacity and speed - how much memory do we need (the Turing machine tape length) and how much computation must we do in unit time (the Turing machine symbol processing rate).

In practice the factors of storage space and speed loom large in all computer selection processes, and most other aspects of computer archictecture relevant to the machine-problem link stem from them. The encoding of an instruction set into an order-code is an information-theoretic problem of bit-utilization (from our current point-of-view). The more frequently-used instructions should be encoded into fewer bits than the less frequently used instructions. The problem of speed is slightly more complex - since the computer operates serially, and assuming that each serial operation takes about the same time, the less computer operations required to effect each operation in the application the faster the computation will proceed. The limit is when each operation in the application corresponds to one operation in the computer. There is then an identity between the computer instruction-set and the terminology in which the application is described - we say the computer provides an 'analogue' of the problem.

There are several points of interest arising from the preceeding paragraph:

(1) Optimized information-theoretic encodings may become very complex and generate a fairly involved encoding process (typically decisions as to whether to use a short address mode, a long address mode or an index register mode) which is a load upon the program writer or compiler, or worse still upon the loader or run-time system - e.g. the whereabouts of other program segments may not be known until load time and the whereabouts of data may vary at run-time.

In theory a compile-time problem is acceptable because it does not effect the system operation. In practice, since the provision of compilers is a major bottleneck in computer development and application, any problems in writing them are serious. In any event the

sensible compiler writer will turn the problem into one at run-time by not using any of the clever, economical and fast, but not universally applicable, instruction types.

(2) The 'analogue' computer as described is clearly excellent in providing not only speed but also a one-to-one correspondence between the application-description and the computer program. The electronic analogue computer has exactly this property in relation to linear differential equation simulation. It is this property of problem-analogy which the special-purpose high-level language (such as APT for machine tool work) attempts to provide - a 'virtual machine' is defined whose instructions are analogies of the operations required in a class of applications, and the compiler transforms a real-machine into this virtual machine.

(3) Both space and time efficiency assume knowledge of the class of problems that a machine will have to solve. This is, with current techniques, a concept which is very difficult to define and utilize in any meaningful way. It is this, more than any other factor, which has given the general-purpose digital computer its market impact in even highly specialised areas such as process control. We do not need extreme generality and it brings with a host of problems - a specialized instrument designed for a specific application virtually defines the way in which it is to be used, and its capabilities and limitations are intrinsic and non-variable. A digital computer is at the opposite extreme, giving no indication of how it is to be used and with capabilities and limitations dependent on how we use it. However, the special-purpose instrument is dying a natural death under the influence of mass-production and mass-marketing requirements (even the venerable oscilloscope has so many programming switches and dials now that some form of 'translater' could help in setting it up!), and the special-purpose computer seems never able to get off the ground.

The type of machine architecture which comes closest to giving the best of all worlds (or at least an effective compromise) is the multi-level programmable machine, in which a high-speed, general purpose 'kernel' is 'microprogrammed' to interpret the instruction set of a (virtual or real - according to whether you are looking down or up!) machine which may have many special-purpose, problem-orientated facilities. Such micro-programmability gives an added feeling of security in that not only can the machine do anything that may crop up, but it can also be fine-tuned to do them fast enough.

3.3 The Machine-Designer Relationship

The third link in Figure 1 is that between the machine and the system-designer/programmer. Without too much melodrama we may note that the designer has to impose his will on the computer, and that he is essentially moulding an amorphous object with few characteristics into a highly specific tool. Unfortunately the computer is unintelligent in that it does not have the capability of self-organization to achieve specified goals. The designer not only has to impose his will, but he also has to know both it and the machine sufficiently well to impress upon the machine every detail of his requirements under all possible conditions. In this section we are primarily concerned with the designer's knowledge of the machine and with the mechanisms by which he ensures that it does as he wishes.

There is a curious anomaly in machine design in that at the extreme 'mini' end of the range, the programs which can be fitted onto the computer are so small that the programmer cannot possibly have much to remember and it is both fair and necessary to make the machine structure complex and give him scope for detailed, bit-by-bit,program optimization to conserve space and squeeze as much as possible out of his 'mini'. As the machine grows larger, however, the programs grow and the programmer has less memory available (in his head) for complex order-codes and clever programming tricks, and less time available to utilize either of them for optimization. Thus as the machine becomes bigger it should also become simpler.

The paradox is less apparent if we consider that simplicity for the programmer is often brought at the cost of complexity of hardware. We give him floating-point numbers as unitary data types, simplifying programming at the cost of an additional arithmetic unit. Thus our progress from the simple Turing machine to the multi-data type, segmented, wonder work-horse of today is characterized by increased hardware complexity leading, hopefully, to increased ease of programming - provided we do not pass some of the complexity onto the programmer.

A simple calculation serves to illustrate the extreme approaches. Wonder machine X is advertized as having 4096 fantastic instructions, each one highly problem-orientated and optimized to have just those effects most often wanted - each instruction has a distinctive name, and a week's course on each will leave the programmer knowing at least the name and, possibly, what the instruction does. Machine Y has the same number of instructions but they may be decomposed into:

16 operations with 8 address modes on 4 data types and
8 data lengths.

The programmer for Y has 16+8+4 things to learn - 28 facts to remember, not 4096.

Such 'co-ordinate system' instruction sets have been used in part in most machines because they arise naturally in encoding an instruction into bit-fields. However, they are rarely carried to the logical extreme, and one finds that certain operations can only be used with certain data types and certain address modes, or that the machine is two-address for integer arithmetic but one-address for floating-point arithmetic. This makes even the most elementary operations, such as moves, fraught with complications since the address modes may vary with data-type and for some modes there may not even be a 'move' instruction! Not only is it desirable that there should be uniformity and homogenity in dealing with all operations, modes and data types, but also that this should be extendible to other operations and data types if the configuration is expanded. Here an adequate 'extra-code' or emulator-trap system is necessary, established as an integral part of the basic machine order-code (so that the programmer sees the same class of instructions for both new and old data types).

The advantage of a co-ordinate system approach to instruction-set design is that a large number of different instructions are generated as the product of much smaller sets of operations, etc. Some of the possibilities may not be particularly useful or meaningful, but the wastage can be made negligible. The advantages to the programmer and compiler writer are a coupling of power with simplicity that makes for a very close programmer-machine relationship. There are associated advantages in the possibilities for true 'high-level' 'assembly' languages for the machine.

The problem of program development itself is not the theme of this paper and will be discussed elsewhere. However, the delineation between hardware and software design in computer systems is becoming increasingly blurred and it is legitimate at this point to look briefly at software development for its similarities to the hardware development problem.

3.4 Computer-aided Design of Software

The fabrication of a system by 'programming' rather than by more conventional and concrete engineering techniques gives the system a unique flexibility and capability for modification both during development and in the field. However, the shift of implementation technology requires a concomitant shift in design and development technology: the development of software equivelants to the drawing office, prototype production, exploratory test-beds, etc. This is all the more important because many, perhaps the majority, of minicomputer buyers are 'first-time' users without previous experience of computers and without computer-based support facilities. In time the ready availability of the machines which is causing this flood into new territory will also have the compensatory effect of saturating most potential applications areas - however, for several years at least, the growth of applications is likely to exceed the growth of the necessary back up technology.

It is not trivial to note that systems heavily dependent on computer software development are those most ripe for the application of 'computer aided design'. This may be obvious but the fact that editors, assemblers and compilers are CAD tools for software development itself tends to be overlooked, in that they are seen as an essential part of the computer system, not as tools largely independent of the computer which should be tailored to the requirements of the system designer. It is true that most mini-computer manufacturers supply the design tools for their computer software in such a form that some configurations of their own computer can actually implement them. However, this is technically irrelevant, and introduces some practical confusion in that is is highly unlikely that the minicomputer configuration required for the final system is able to support an adequate software generation system - the two specifications are generally far apart.

Cross-assemblers and compilers enabling the programs for one machine to be created on another have been in use for some time. However they represent only one comparatively minor step in the design process: the translation from program specification to machine-acceptable code. On one side of them are the text creation systems such as interactive editors, and on the other side are the code testing systems such as emulators. Both editors and emulators are systems which cannot be handled well under batch facilities and require a good interactive system to support them. This, in itself, is a positive reason for having design software on a minicomputer: it is easy to provide an interactive editor on a suitable minicomputer configuration, and the machine can generally be programmed as an efficient emulator of itself under a symbolic debug. However, the editor requires backup store, preferably drum or disc, and the debug requires additional mainframe memory.

The logical conclusion is that for minicomputer software development an interactive system, not necessarily related to the target system, should be set up with suitable computer-aided design tools for minicomputer software development: a powerful interactive editor, an assembler (pref-

erably a meta-assembler) and an interactive emulator.

A major difficulty in implementing this conclusion arises through the costs involved in the developmental system. Although good program development resources can amplify the efforts of the programmer by a factor of five or more, the capital cost has in the past generally been too high for many potential users. However, with the advent of low-cost backup memories such as flexible discs, and low-cost high-speed displays, this situation should change over the next few years.

4. Summary and Conclusions

There is clearly far more to be discussed and this paper has raised questions rather than resolved them. However, the technical details of individual computer designs are irrelevant to most applications until a full account is taken of the associated factors outlined in this paper. For the future we can look forward to increasing computing power at decreasing cost, size, and power-consumption. We can also expect increasing attention to be paid (both in machine development and marketing) to the programmer/machine link, by extending the simplicity and power of instruction sets and by providing adequate software development resources.

Gradually the system will build up inertia both through the existence of successful applications packages and through the availability of good development software and trained staff for certain machines. This will make it difficult to market new machines which are incompatible with previous software, and manufacturers will be forced to exploit new technology and machine architectures through configurations which are able to appear both as previous generation machines and as the new machines - that is through variable microprogram machines.