A Myopic View of Computer-Based System Design

J. W. Layland and C. C. Klimasauskas
Communications Systems Research Section

This article presents one approach to the economical allocation of resources in a complex logical system. The main goal is the understanding of systems that may involve specialized digital hardware, a computer with software, and possibly a specialized microcode within the computer processor. These components are treated as uniformly characterizable options in the design of the system that will ultimately be used.

I. Overview

In this article we present our approach to the economical allocation of resources in a complex logical system. A computer is one example of such a system, but our main goal will be the understanding of systems that may involve specialized digital hardware, a computer with software, and possibly specialized microcode for the extension of the capabilities of the computer itself. It is our intention to consider these components as uniformly characterizable options in the design of the system we will ultimately use.

The system to be designed must be represented in some unambiguous fashion in terms of a set of primitive operations. The logical n-input NAND gate is sufficient to build all systems, and in fact, is one of the alternate bases for the complexity work of Savage (Ref. 1). However, to provide a foothold for resource-allocation trade-offs, we must represent our system with a cascade of higher-level primitives, and evaluate the effect of implementing the higher-level primitives with resources of varying cost and performance. For concreteness of example, we will take our highest-level primitives to be the arithmetic and logical operators of a typical medium-scale computer, and consider several purely computational tasks as our system to be implemented.

Each primitive operator on the highest level must be implemented in some low-level operator, with the lowest-level logical operators being gates. Each can be constructed directly in the lowest level (built-in hardware) or

constructed from some intermediate-level primitive (programmed from primitives of a lower level). For all its apparent mystique, microprogramming is nothing more nor less than the programming of the machine-language-level primitive operators in terms of some lower-level set of primitives. The microprogram memory is usually higher speed than the main memory of the computer, and it is frequently read-only. Our options thus include direct construction of the primitive operators in hardware and programmed implementation in memory of various speeds and costs. The relative benefit of each type of implementation for the various primitives depends strongly upon how other primitives are implemented.

If the number of options being considered can be restricted to a relatively small number, as would occur in evaluating off-the-shelf options for a commercial computer, then an ad hoc graphical comparison can be used to evaluate them. Examples of this nature will be developed in a later section. For most real-life problems, the number of options involved will be too large to handle manually, so we expect at some future date to formulate the allocation techniques we are developing into a form suitable for solution by nonlinear mathematical programming methods.

II. Motivation

We can find motivation for our investigations in almost all aspects of computer-oriented activities today. High central processing unit (CPU) utilization is regarded as the ideal in most general purpose computer installations, and it is sought after by increasing memory to allow for multiprogramming, requiring an expanded operating system, and additional resources of other types, perhaps including a faster CPU. Is this economically sensible? To answer that, we need to know the true shape of the performance vs cost curve for computer processors. "Grosch's Law," which contends that performance is proportional to the square of the cost, is approximately valid for the IBM System/360 (Ref. 2, pg. 525), but thus may be evidence of an IBM marketing strategy and not a technical phenomenon. Minicomputers appear to violate that "law," either because of real wiggles in the performance vs complexity curves, or because they are more rapidly making use of developments in component technology. We should know which in order to plan intelligently for the future.

High utilization alone does not make for efficient use of resources. For almost any computer extant we could synthesize a job or job-stream that would almost totally occupy the CPU time and available memory space and yet very poorly utilize these resources. As an easy example, perform error-correcting code analysis or decoding on a processor with very high-speed (an expensive) floating point arithmetic operators, and without effective bit-manipulating operators. Here, a significant part of the CPU is left idle, even though the CPU itself is not. This same CPU-part will also be (mostly) idle whenever our computer is doing text editing, program compilation or assembly, and many other jobs that computers do.

TYMNET, the communications network of the TYMESHARE corporation, utilizes Varian 620i minicomputers for message concentrators and terminal controllers (Ref. 3). Data on the low-speed asynchronous communication lines to terminals are sampled on a bit-by-bit basis, and packed by software into characters (typically 8-bits). The hardware needed to interface the computer to the communication circuitry is minimized this way, but the computer moves many (18-bit) words about to enter each bit of data. Was this the economical choice? Or could they have more economically added character-assembly buffers to the communications interface hardware? Well-formulated questions such as this one can be answered directly in terms of dollars.

In the Deep Space Stations, minicomputers perform many tasks with responsibility shared with external hardware. In handling down-link telemetry, for example, computers do low-speed bit synchronization, control high-speed bit synchronizers, control block decoders, and perform (via microprogram) sequential decoding. In each case a choice must be made to partition between external equipment and software. We wish to develop criteria to guide that trade-off, and will describe some initial steps toward that goal in this article.

III. System Representation and Optimization

The first step toward an optimizing solution of a problem is the complete and unambiguous description of that problem. This is no less true in the allocation of computational resources. Most, if not all, of the computer-based systems of greatest interest are finite state machines (FSMs); or, more precisely, are *collections* of FSMs. Examples range from an Antenna Position Control Subsystem to a communications switching computer, to the operating system of a general purpose computer, to a dedicated on-line reservation system. No programming language in general use today admits a concise description of an FSM, although both high-level algebraic languages and machine-assembly languages have been

used to implement these systems. Direct FSM representation has long been used in the processing of languages (Ref. 4), and has been recently applied to communications processing (Ref. 5). There is for now, however, no clear standard language that could be used to describe both hardware and software implementation of general FSMs.

Suppose that a system under development has been described as an FSM, and that steps toward its implementation are underway. In this process, the system would be broken down into simpler, more primitive machines. These second-level machines are themselves further segmented, until a stage is reached where the direct implementation of the lowest-level machines in hardware, or in any of the myriad of programming languages, is a semitrivial exercise. In short, the system is subjected to a top-down design (Ref. 6) — not in terms of a subroutine hierarchy for an anticipated programmed implementation, but in terms of an implementation-independent FSM representation.

Given a complete description of the system as reduced to this primitive level, it should be a straightforward task to determine how total system performance depends upon the performance of each of the lowest-level submachines. And we can determine the performance and cost of each of these lowest-level machines for each of the three likely implementations: hardware, software executed from computer main memory, and software executed from high-speed memory (microcode). The implementation pattern over all primitive submachines that yields lowest system cost can be determined from these inputs by a straightforward but possibly long computational process. What will happen in this process is that those primitive submachines that most affect total system performance will be implemented in the fastest, most expensive fashion, and those primitives that little affect total performance will be implemented in the slowest, cheapest fashion.

Once the implementation mode of the lowest-level machines has been determined, those machines on the next-to-lowest level must be examined. Any of these for which the constituent submachines are implemented in hardware should probably also be implemented in hardware or microcode, but this can be ascertained by the same sort of computations that established the implementation mode of the lowest-level submachines. This type of examination should be carried through all levels of the system representation, assigning an implementation to each as it progresses.

The degree of optimality of the completed system depends upon the way in which it is modularized, or broken into its constituent submachines. Modularization is a difficult and, even if well understood, not a well documented process (Refs. 6 and 7). From the designer's viewpoint, the best criterion appears to be understandability. This may or may not be a good criterion from the standpoint of system performance, but we expect that performance is not critically dependent upon the specific modularization selected. A poor modularization will at worst prevent the suboptimized restricted problem from closely approaching the true optimum. However, refusal to modularize the system will present the optimizer with a massive, unmanageable problem.

The process we have just described is similar to the tuning process that is frequently applied to working software modules. In tuning software, the software module is instrumented to determine where it spends the majority of its time, and that region subjected to an intense optimization effort. Tuning has been defended on the basis that it is difficult to know before software is implemented where the most critical areas are. We believe that those critical areas will be quite obvious during the course of design using an implementation-independent representation, and that the additional flexibility that exists before implementation will yield a much superior result from the attempted optimization.

Virtual memory systems (Ref. 8) and systems using a high-speed buffer or cache (Ref. 9) attempt to perform automatically an optimization process similar to what we have described. In each, the main memory is a relatively slow, relatively inexpensive memory. Data are moved from main memory to the expensive high-performance memory in blocks, so that most accesses to memory can be satisfied by accesses to the fast unit, only rarely requiring that data be moved between levels of memory. In theory, at least, the more critical parts of the software reside permanently in the fast memory, while the remainder is only transiently in the fast memory, and hence executes more slowly due to the time required to access it in main memory.

We do not have a well-defined FSM representation technique, so for a concrete example of the evaluation and trade-off methods just discussed, we turn in the next section to an area where we have a complete representation — working software modules — and evaluate implementation alternatives for the "primitive submachines", represented by the machine instructions.

IV. Computer Processor Evaluation/Design

A method of computer performance evaluation known as the "instruction mix" method (Ref. 10) compares computer processors on the basis of the execution times for the instructions that are used most heavily in the jobs that the computer is to perform. This is not a widely accepted method because it ignores all of the subtleties of input/output queuing, memory usage, and all features of the vendor-supplied software that interface the typical user to his machine. These objections can mostly be waived if the job to be done is in process control where the computers are small and their operating system minimal or nonexistent. The user's problem-solving machine is then built almost directly from the machine instructions.

The user's system must in any case be built from the base computer's instruction set, whether it is built directly overall, or partially indirectly via instructions interpreted by an operating system. The computer instructions form the lowest-level primitive operator that is readily visible. We can determine the relative importance of the various instructions with respect to any specific task by tracing the execution of working software that performs this task. A companion article by Klimasauskas describes the interpretive tracing program that we used to gather data (Ref. 11). The program runs on the Xerox Sigma 5, so all data are specifically relevant to that computer and its software.

Execution data have been gathered on the instruction usage of three different types of software: the FORTRAN IV compiler, the on-line text EDITor, and user programs written in FORTRAN. The FORTRAN IV compiler was probed initially to determine how sensitive its instruction usage was to the statement types being compiled. Sensitivity was found to be very small – a few percent – which meant that we could realistically identify the instruction usage of a "typical" compilation. Figure 1 is a bar graph of the selected "typical" compilation data. Each column of Fig. 1 corresponds to one of the Sigma instructions, and the mnemonic (Ref. 12) for the associated instruction is written vertically beneath the column. Height of the column of asterisks is proportional to the percentage usage of that instruction on a logarithmic scale. The character "=" on the bottom line designates unused instructions, and "#" designates usage below 1%. By way of example, load immediate (LI) is the first instruction on the graph, and it accounts for about 2.4% of the total instruction executions.

EDIT proved to be a less stable subject, perhaps due to the greater diversity of services provided to the user. The "typical" instruction usage depends upon the user statistics, which we have neither the facility nor the real need to measure. Our need for exemplary material is satisfied by considering the extremes of EDIT instruction usage that are shown in Fig. 2a and 2b.

A set of user-written programs could obviously show a great variability. To avoid either a massive statistics-gathering project or extensive arguments relative to what constitutes a typical user job or mix of jobs, we arbitrarily selected matrix inversion as being one identifiable user task that occurs frequently enough to be worthy of investigation. We traced the inversion of matrices of varying dimensions from 2×2 to 100×100. The dependence of the total floating-point instruction usage relative to total instructions is shown in Fig. 3. The detailed instruction usage summary graph appears as Fig. 4 for the inversion of the largest matrix.

It is a relatively easy matter to go from the statistics of the sort shown in Figs. 1 to 4, plus published execution times of the Sigma 5 instructions (Ref. 12) to an effective execution time for that computer relative to the task to which the statistics apply. This effective execution time is proportional to the cost of performing that particular task on that computer. We can evaluate processor options, such as floating-point hardware vs software floating-point simulators, using this cost-indicator. With slightly more effort we can determine an effective execution time for other processors by first determining what instructions or sequences of instructions on the processor being considered perform the same functions for the program as each of the Sigma instructions used. The individual instruction execution times so devised then determine the effective execution time for the processor. This process has been carried out for the Digital Equipment Corporation PDP-11, and on several IBM processors. The result of this is shown in Table 1.

The processor itself is only one of the cost-contributing elements of a computer. Both main memory and input/output equipment usually cost far more than the CPU. The performance of the CPU, however, determines how much time the entire system is occupied by a specific task, except when that task is I/O-bound, or executing at a rate constrained by one or more pieces of external equipment. We shall assume in the following that the tasks of interest are not I/O-bound, and in fact, that cost involved in the external equipment is not of interest. Memory cost is of interest, and since we do not have a

good characterization for memory requirements, memory size is treated as a free parameter. Our specific comparisons will be between the PDP-11/20 and the Sigma 5 with various processor options. We will assume a fixed cost, say 1 unit per 8×10^3 bytes, for the main memory of these computers, and normalize all other item costs to this figure. (We believe that the difference in memory prices established by the manufacturers is an indicator of overall technological advance during the period between the design of the computers.) If we assume that the relative costs of CPU and memory depends predominantly on CPU architecture and complexity, and little upon the technology-base for construction and the manufacturer's marketing strategy, then normalization to a nominal memory cost allows direct cost-wise comparison of the CPU complexities. The relative cost figures for the components of each of the computers have been derived from the manufacturers advertised prices, and are only approximate. The modifications to our work to perform true cost evaluation of various computers from quoted prices and to include peripheral equipment costs would be routine.

We assume that the computers of interest will be uniprogrammed. Multiprogramming has no advantage unless at least some of the tasks to be performed are I/O-bound, and for this article at least, we wish to avoid the queuing intricacies that arise there. For each task, the computer is totally occupied by it from start to finish. The cost of performing that task is simply the cost of the computer configuration, times the execution time of the task, divided by the total lifetime of the equipment. Lifetime is assumed to be the same for all equipment. Since we are comparing units rather than estimating exact costs, we can normalize execution times by the execution time of one of the configurations. The effective execution time for the task and processor is then used for the time the task occupies the processor. Since the configuration cost is a straight-line function of the memory size allocated, the task execution cost is also a straight-line function of memory size. Table 2 lists the configurations considered, together with the relevant cost parameters. Figure 5 shows the relative costs of executing the matrix inversion task for these configurations, and Fig. 6 shows the relative cost of executing the FORTRAN compiler. Minimum cost for all configurations occurs at zero memory, but this is illusory because no work could be accomplished without some minimum memory allocated. For large memory sizes, the cost of the processor is significantly below that of the memory. Hence, more complex processors with faster execution times become increasingly economical with increasing memory size.

V. A Computer Design Exercise

It is a conceptually easy step to go from evaluation of computer processors to investigating the design of one. In designing a computer, we must be aware that it is not the computer per se that we are interested in, but it is in fact the economical implementation and operation of the users machine that is to be built using the computer and possibly other components. Let us suppose that the machine under design is to perform a known specific task and that we have reduced that task to an intermediate-level machine representation called the Sigma Instruction Set. In this form, the remaining system design is equivalent to the design of a Sigma-like computer that has been optimized for the specific task at hand.

There are many options available in this design. As one extreme, we could build a separate hard-wired machine to perform the operations of each of the Sigma instructions. As the other extreme we could build a very simple machine and make it interpret the Sigma instructions. If the simple machine program (the control program) were stored in a fast read-only memory, this would be a typical microprogramming situation. The control program could be stored in read-only memory, or in read-write memory of varying speeds and costs. In this environment, optimization consists of determining which parts of the control program belong in which type of memory.

A wide range of intermediate configurations is possible. A machine might be able to directly execute a basic subset of the Sigma instructions, but trap to an interpreting control program for the more complex instructions. The obvious basic subset would include a LOad, STore, Shift-by-one, and the ADD, SUBtract, logical AND, logical OR, and logical Exclusive OR available in a single medium-scale integration (MSI) logical array. This base machine could have the indexing and indirect addressing operations of the Sigma, but it need not as these could be made available in the same fashion as the more complex instructions. The set of primitive operators of the system has in this case been implemented in so simple a fashion that we believe little or no manipulation of its cost and performance is possible. Optimization is to be performed by varying the memory type to which each segment of the control program is to be allocated. But we also have the option of implementing some of the other instructions directly in hardware, or adding non-Sigma instructions that would make interpretation of the more complex Sigma instructions easier and faster.

We can write the control program to interpret the complex Sigma instructions without knowing what type of memory it will be executed from. The control program does depend upon instructions that are added to the base machine, but mostly these represent replacement of parts of the control program by hardware features. The greatest effort is thus involved with developing the Sigma instructions from the basic machine. In today's market, there are essentially three memory technologies that could readily be considered: core ($\sim$ 1 μs), MOS semiconductor (0.5 μs), and Bipolar semiconductor (0.1 to 0.3 μs). The cost of these on a per-bit basis is monotone-increasing with speed. In implementing 100 instructions, we have 3^{100} options out of which to select the optimum. We would not wish to compute through this space many times because of iterations in the base machine structure, nor would we be able to compute through it even once without some sensitive heuristic to reduce its effective dimensionality.

Let us label the instruction set as $\{I_i\}_{i=1}^{n}$, and denote as p_i the usage probability for instruction I_i within some user's task of interest. The program to implement I_i sequences through s_i steps in the control memory. This parameter s_i depends only upon the operation performed by I_i, and not upon the implementation chosen. Let the control memory segment for I_i be implemented from components with step-time t_i, where t_i in the above paragraph can have any one of three values. The cost-per-step is known to be a function of t_i; call it $c(t_i)$. The total cost of the submachine that implements instruction I_i is thus $s_i \cdot c(t_i)$. The net execution time for I_i is $max\ \{1, s_i \cdot t_i\}$, where the 1 represents the time necessary to fetch an I_i from the computer's main memory.

For convenience, we will let both the step-time and the per-unit cost of the main memory be unity, and normalize other items appropriately. Let B be the normalized cost of the base machine, and M_u be the number of units (and cost) of the main memory assigned to the end user. Then the total cost of our processor is

$$B + M_u + \sum_{i=1}^{n} s_i \cdot c(t_i)$$

The effective execution time is

$$\sum_{i=1}^{n} p_i \cdot max\ \{s_i \cdot t_i, 1\}$$

The net normalized cost of a job is proportional to the product of these two

$$C_{eff} = \left(\sum_{i=1}^{n} p_i \cdot max\ \{s_i \cdot t_i, 1\}\right)\left(B + M_u + \sum_{i=1}^{n} s_i \cdot c(t_i)\right) \tag{1}$$

The memory allocation, that is, the choice of t_i, which minimizes C_{eff} is the sought-for optimum. The allocation, of course, depends on the assumed cost function c.

At this point, let us approximate our real problem with an easily solvable one. Assume first that the unity minimum on the execution time of I_i can be ignored. This means either that the instructions are all longer than that minimum, or that we have interposed a cache or buffer memory between main memory and CPU. Assume second that there is a continuum of memory speeds available, instead of the above three, and approximate the cost function by $c(t) \approx t^{-\beta}$ for some power β. (This is not unlike current pricing with $\beta \approx 1.5$.) By optimizing in this fashion, we risk finding a t_i either above or below the accessible range; both extremes suggest that reorganization of the base machine is necessary. Differentiating C_{eff} with respect to each of the t_i's produces

$$\frac{\partial}{\partial t_j} C_{eff} = 0, \qquad \text{so}$$

$$\frac{p_j s_j t_j}{\sum_{i=1}^{n} p_i s_i t_i} = \beta \frac{s_j t_j^{-\beta}}{B + M_u + \sum_{i=1}^{n} s_i t_i^{-\beta}} \tag{2}$$

We can derive several conclusions by manipulating Eq. (2). By summing over j, we see that the optimum occurs when

$$\sum_{i=1}^{n} s_i t_i^{-\beta} = \frac{1}{\beta - 1}(B + M_u) \tag{3}$$

which means that the total machine control cost is proportional to total costs of base machine and main memory. If Eq. (3) is inserted into Eq. (2), we see that the optimum occurs when

$$\frac{p_j s_j t_j}{\sum_{i=1}^{n} p_i s_i t_i} = \frac{s_j t_j^{-\beta}}{\sum_{i=1}^{n} s_i t_i^{-\beta}} \tag{4}$$

or in words: when the fraction of total time spent in I_j is equal to the fraction of total instruction-building costs expended on I_j. We can in fact reduce Eqs. (3) and (4) to show that

$$t_j = \left(\frac{(\beta - 1) \sum_{i=1}^{n} s_i \left(\frac{p_i}{p_j} \right)^{\beta/(\beta+1)}}{B + M_u} \right)^{1/\beta} \quad (5)$$

Establishing allocations from Eq. (5) is far easier than numerically minimizing Eq. (1), but we must still interpret these results in terms of the real-world problem. If any of the t_j are significantly outside of the range covered by current technology, we should revise the overall design by adding or deleting instructions from the base machine, then recheck the allocation of control memory. It may also happen that the overall performance of the resultant machine is inadequate to fulfill some external requirement, or is too fast and is always waiting for something to do. We can accommodate these factors within Eq. (5) by pretending that M_u is larger or smaller than anticipated. We should also, at this point, return to the basic design and question whether M_u was properly estimated in the first place, since the selected figure produced unrealistic results when applied to the processor design. Once all t_j are within the technically feasible range, they can be approximated by the nearest available step-time, and the result used as a starting value for the numerical minimization of Eq. (1).

The initial steps of this process have been performed for a fully microprogrammed implementation of the Sigma instructions used by the FORTRAN compiler. The base machine was assumed to have logic for segmenting the Sigma instructions into their constituent fields, basic arithmetic/logical instructions, plus a shift by 2^n instruction class, and test/add/shift combination instructions for facilitating multiplies and divides. Figure 7 shows the pertinent parameters that result from the application of Eq. (5) to this task. Most of the control-program memory times were close together and within the feasible range. However, the multiply and divide operations had relatively long memory times, indicating that, for this job at least, they were constrained more by memory cost than execution time. Most likely, the combination instructions added to speed up the multiply/divide operations should be deleted from the base machine, and replaced by a conditional jump, which saves storage at the expense of time. Neither this iteration, nor the next step of assigning allowed memory times to the instruction control memory have yet been performed.

VI. Where We Are Now

It should be clear at this point that we could continue almost forever with design examples of processor and memory structures. The design exercise initiated in Section V could of itself consume several months of effort. We are not at the moment prepared to expend that effort on that particular example.

The viewpoint we have developed for the efficiency characterization of computer processors is a slight refinement of the instruction-mix method (Ref. 10) for computer performance evaluation. It is a useful tool for the evaluation of computer processor options, and a tractable measuring device for an end-use-oriented optimization of processor design.

Although the work presented here has been specifically processor oriented, the general results obtained have turned out to be functions of the memory allocated to the user's process: the optimized processor cost is proportional to the user's memory cost. Thus, a way is needed to characterize the memory required in the performance of user tasks. Given that characterization, we should be able to optimize operations with the user's memory in much the same way as we have attempted here for the control memory.

We are also as yet unable to say anything about economy-of-scale in computer systems. Again, we need a characterization of memory requirements, and possibly other parameters to permit us to estimate the overall performance vs overall cost relationship. In addition, we may need to investigate the behavior of I/O boundedness; queuing; and multiprogramming as they relate to the synchronization of real-time events, such as those that exist within a DSN tracking station, with computational events within a controlling computer.

Finally, we believe that future progress along the path initiated here depends upon the development of a system description technique, such as the Finite State Machine representation discussed in Section II, that will allow a significant portion of the system design process to be performed without prior commitment to hardware, firmware, or software for implementation. Both computer assembly languages and current high-level algebraic languages correspond to implementation languages, rather than description languages, when applied to systems instead of to calculation problems.

References

1. Savage, J. E., "A Collection of Results on Computational Complexity", in *Supporting Research and Advanced Development*, Space Programs Summary 37-65, Vol. II, pp. 42-47, Jet Propulsion Laboratory, Pasadena, Calif., Sept. 30, 1970.

2. Bell, C. G., and Newell, A., *Computer Structures; Readings and Examples*, Part 6, Section 3, McGraw-Hill Book Company, New York, N.Y., 1971.

3. Beere, M. P., and Sullivan, N. C., "Tymnet – A Serendipitous Evolution," *IEEE Transactions on Communications Technology*, Vol. COM-20, N.3, p. 511-515, June 1972.

4. Gries, D., *Compiler Construction for Digital Computers* (especially Chap. 3) John Wiley & Sons, New York, N.Y., 1971. (NB: this reference is a modern entry to a wealth of past literature, part of which is specifically referenced in paragraph 3.6.)

5. Birke, D. M., "State Transition Programming Techniques and Their Use in Producing Teleprocessing Device Control Programs," *IEEE Transactions on Communications Technology*, Vol. COM-20, No. 3, p. 569-575, June 1972.

6. *Software Engineering*, Edited by P. Naur and B. Randell, NATO Science Committee Conference Report, Jan. 1969.

7. Parnas, D. L., *On the Criterion to be used in Decomposing Systems into Modules*, Carnegie-Mellon University Report – CMU-CS-71-101, Aug. 1971.

8. Denning, P., "Virtual Memory," *Computing Surveys*, Vol. 2, No. 3, pp. 153-189, Sept. 1970.

9. Conti, C. J., "Concepts for Buffer Storage," *IEEE Computer Group News*, pp. 9-13, Mar. 1969.

10. Lucas, H. C., "Performance Evaluation and Monitoring," *Computing Surveys*, Vol. 3, No. 3, pp. 79-91, Sept. 1971.

11. Klimasauskas, C. C., "An Execution Analyzer for the Sigma 5 Computer," *The Deep Space Network Progress Report*, Volume XII, pp. 176-188, Jet Propulsion Laboratory, Pasadena, Calif., Dec. 15, 1972.

12. *XDS Sigma 5 Computer Reference Manual*, Xerox Publication #90-09-59D, Feb. 1970.

Table 1. Computer configuration speed comparison

Computer item	FORTRAN compiling time	Matrix inverse time
Sigma 5 with floating hardware	1.0	1.0
Basic Sigma 5 CPU	1.0	9.7
Basic PDP-11/20 CPU	2.7	16.0
PDP-11/20 with floating hardware	2.7	2.9
IBM 360/44	2.0	1.7
370/135	1.6	1.7
360/65	0.66	0.54
370/155	0.40	0.34
360/85	0.15	0.09

Table 2. Computer configuration cost comparison

Computer item	Normalized cost
8×10^3 bytes of storage	1.0
Basic PDP-11/20 CPU	1.6
PDP-11/20 with floating-point hardware	4.3
Basic Sigma 5 CPU	7.8
Sigma 5 with floating-point hardware	10.5

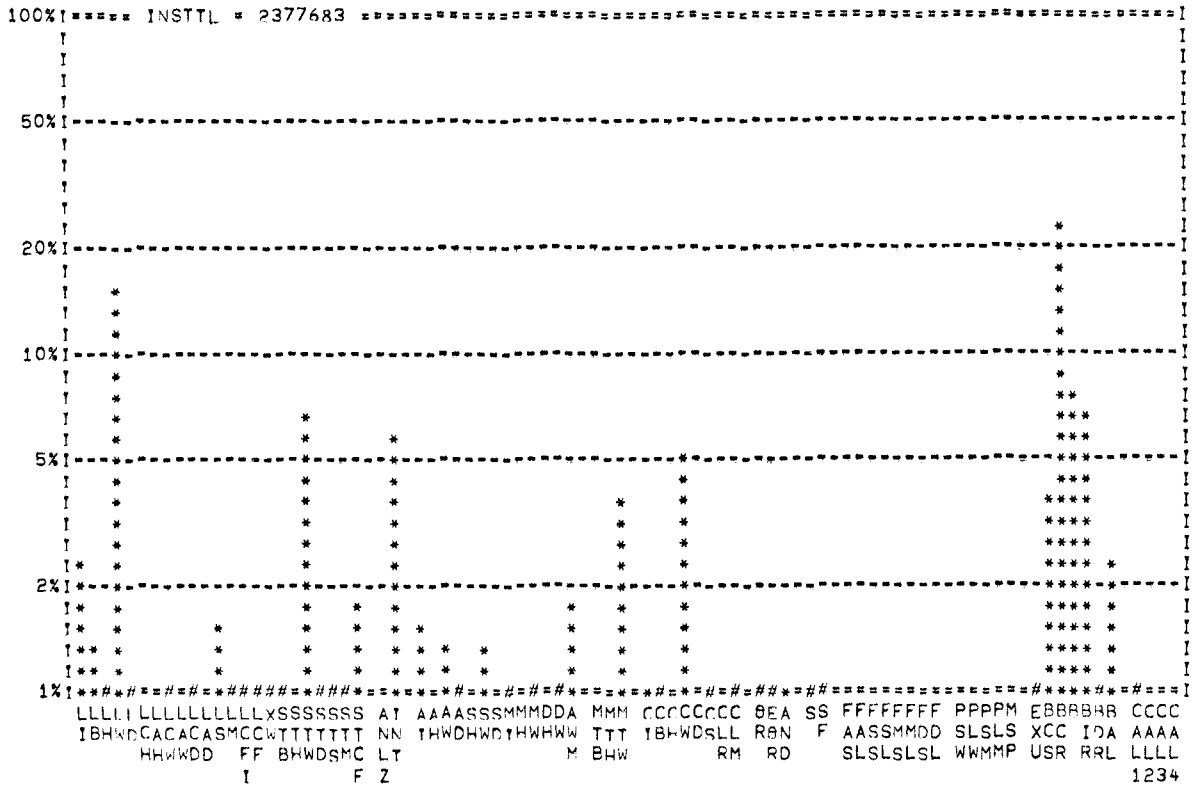

Fig. 1. Instruction usage of typical compilation

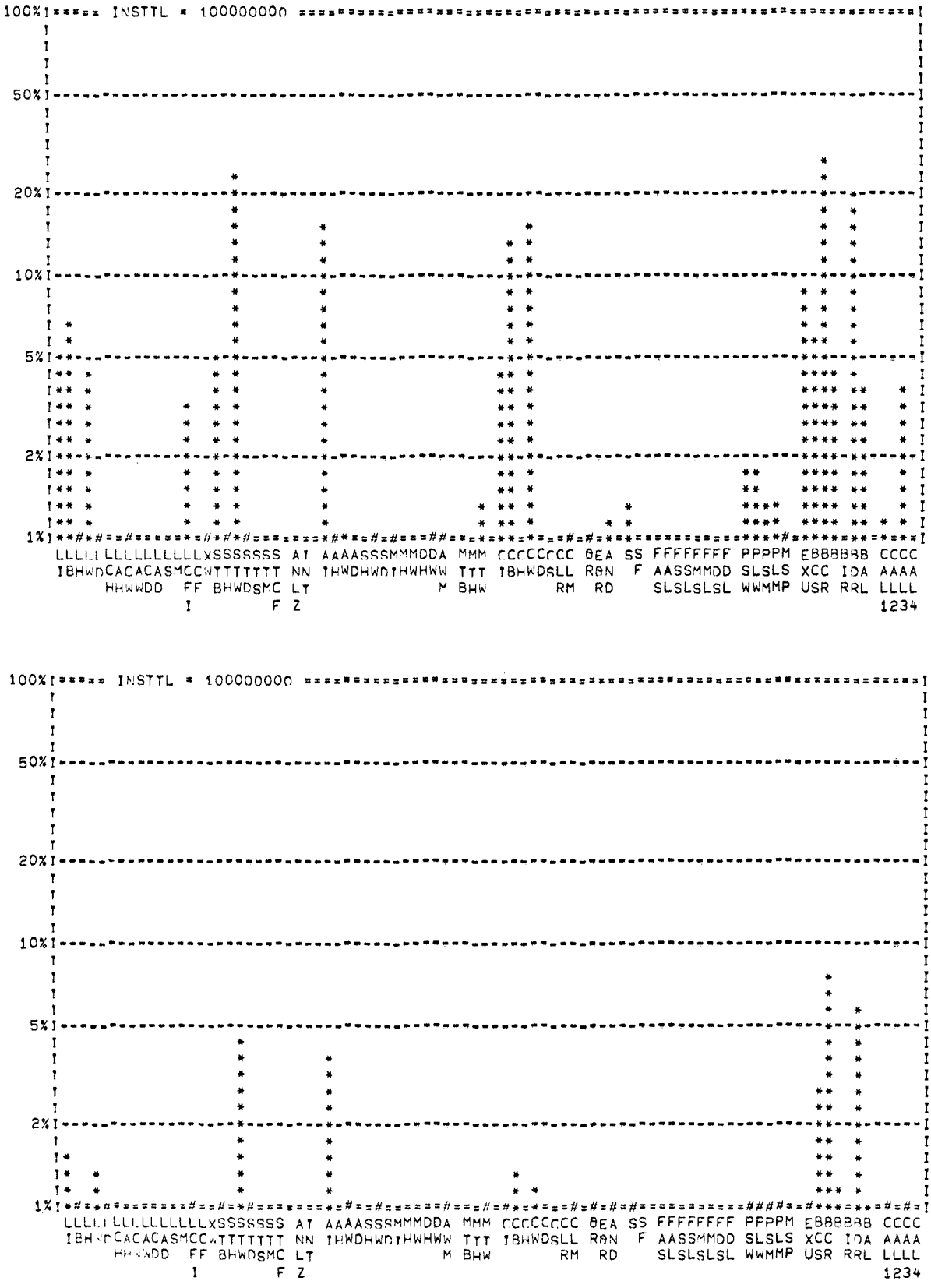

Fig. 2. Instruction usage of extremes of EDIT operation

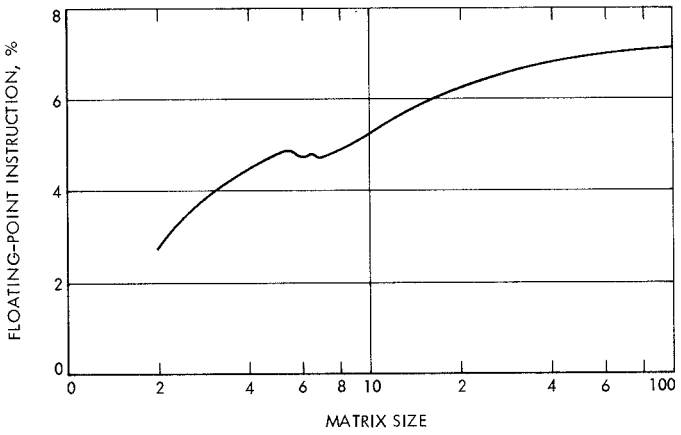

Fig. 3. Usage of floating point instructions in matrix inversion

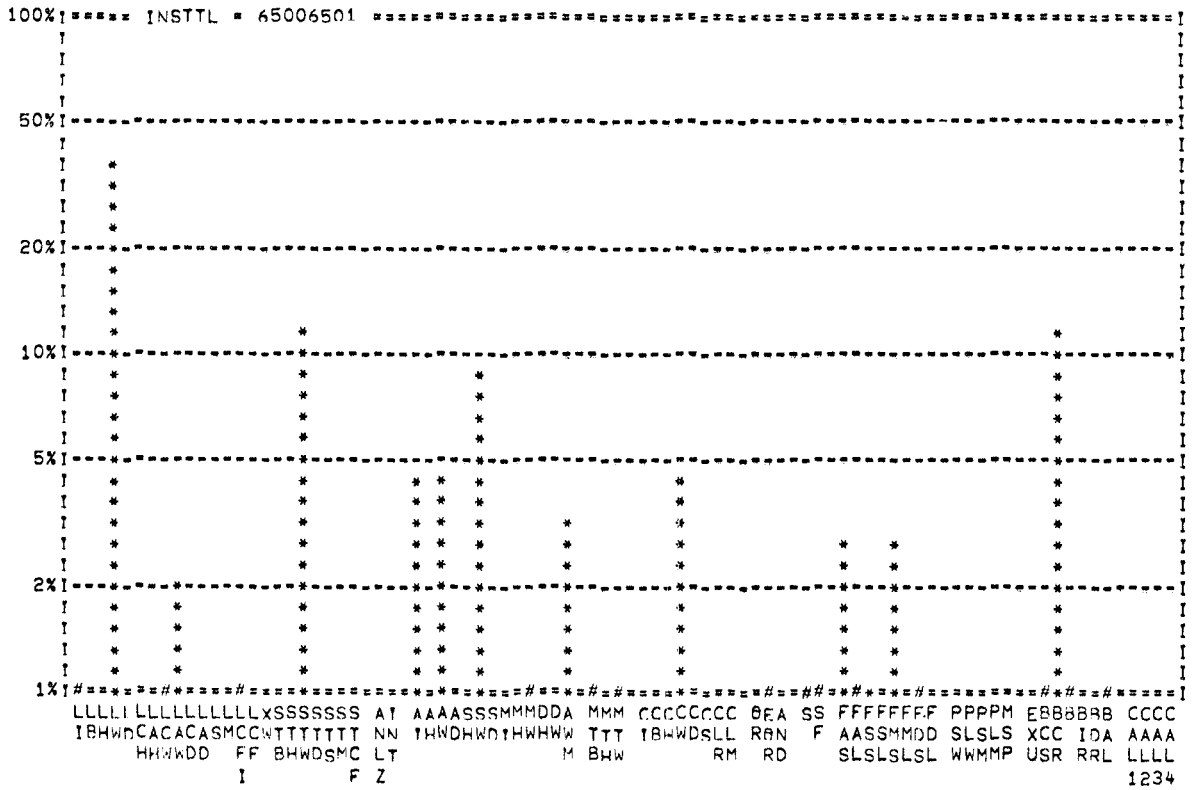

Fig. 4. Instruction usage of matrix inversion, $n = 100$

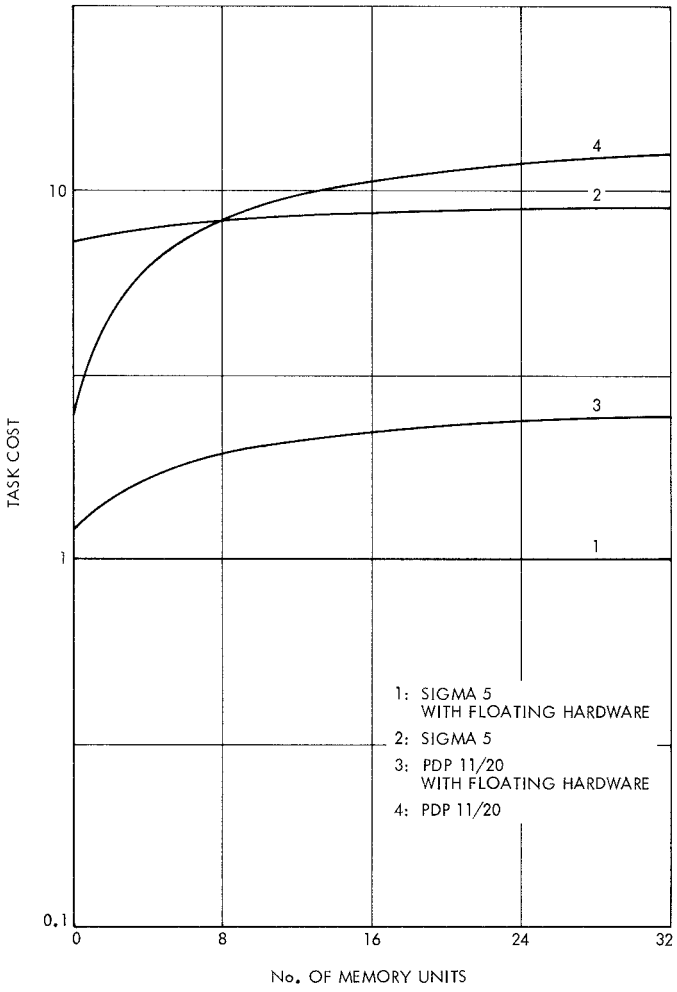

Fig. 5. Relative cost of executing matrix inversion

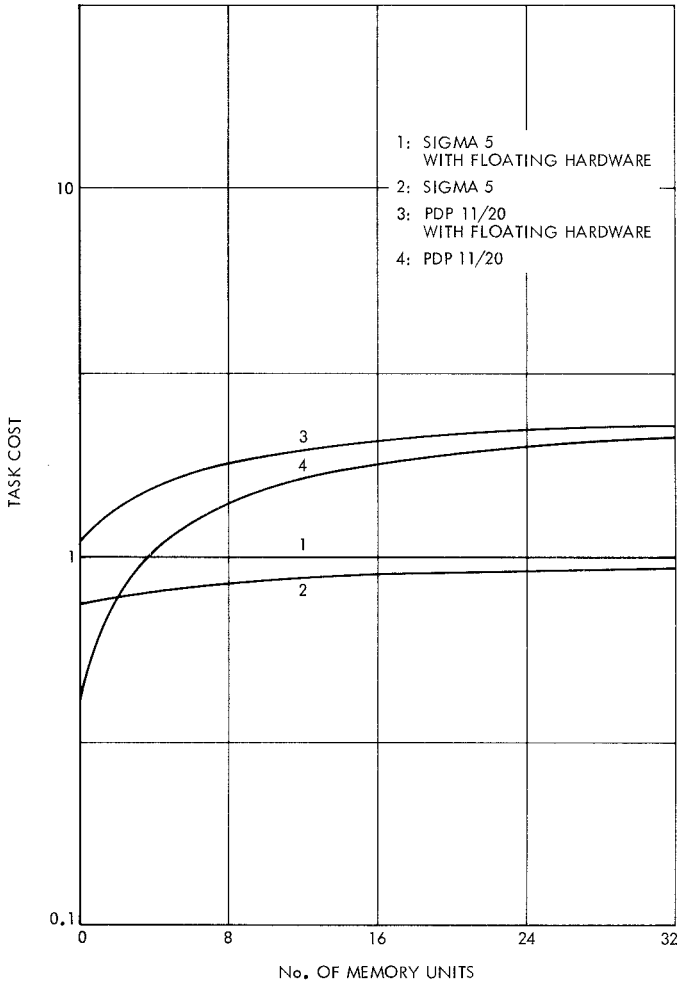

Fig. 6. Relative cost of FORTRAN compilation

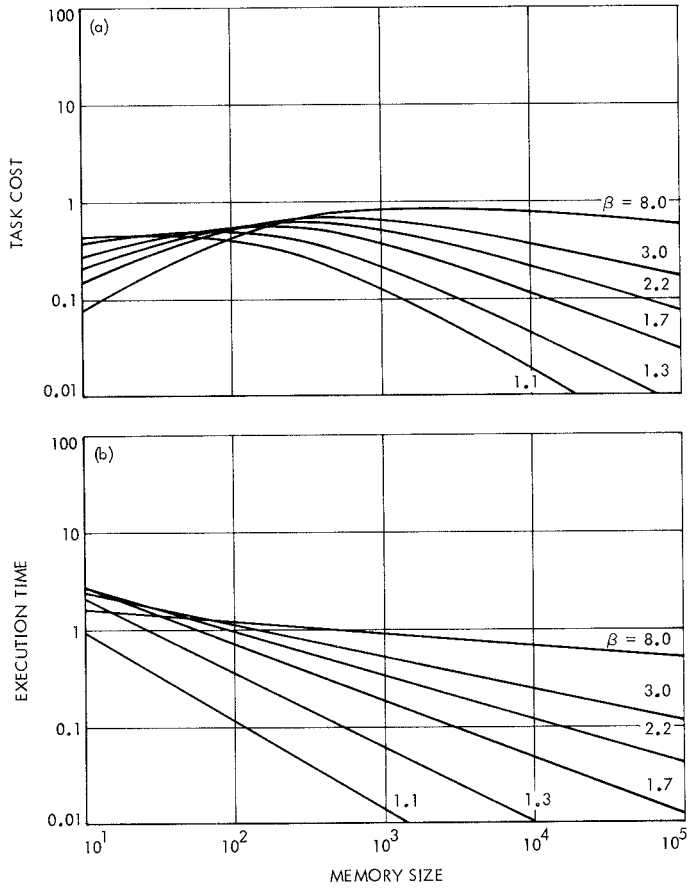

Fig. 7. Performance parameters of hypothetical machine: (a) C = 1: control memory of same speed as main; (b) T = 1: control memory of same speed as main