SOFTWARE AND ITS IMPACT: A QUANTITATIVE ASSESSMENT

B. W. Boehm

December 1972

P-4947

SOFTWARE AND ITS IMPACT: A QUANTITATIVE ASSESSMENT *

> You software guys are too much like the weavers in the story about the Emperor and his new clothes. When I go out to check on a software development the answers I get sound like, 'We're fantastically busy weaving this magic cloth. Just wait a while and it'll look terrific.' But there's nothing I can see or touch, no numbers I can relate to, no way to pick up signals that things aren't really all that great. And there are too many people I know who have come out at the end wearing a bunch of expensive rags or nothing at all.
>
> --An Air Force decisionmaker

INTRODUCTION

Recently the Air Force Systems Command completed a study, "Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980s," or CCIP-85 for short. The study projected future Air Force command and control information processing requirements and likely future information processing capabilities into the 1980s, and developed an Air Force R&D plan to correct the mismatches found between likely capabilities and needs.

Although many of the CCIP-85 conclusions are specific to the Air Force, there are a number of points which hold at least as well elsewhere. This article summarizes those transferable facts and conclusions.

---

*To be published in *Datamation*. The views in this article do not necessarily reflect those of the United States Air Force.

Basically, the study showed that for almost all applications, software (as opposed to computer hardware, displays, architecture, etc.) was "the tall pole in the tent"--the major source of difficult future problems and operational performance penalties. However, we found it difficult to convince people outside the software business of this. This was primarily because of the scarcity of solid quantitative data to demonstrate the impact of software on operational performance or to provide perspective on R&D priorities.

The study did find and develop some data which helped illuminate the problems and convince people that the problems were significant. Surprisingly, though, we found that these data are almost unknown even to software practitioners. (You can test this assertion via the "Software Quiz" shown in the Appendix.) The main purpose of this article is to make these scanty but important data and their implications better known, and to convince people to collect more of it.

Before reading further, though, please turn to the Appendix and try the Software Quiz. It's intended to help you better appreciate the software issues which the article goes on to discuss.

SOFTWARE IS BIG BUSINESS

One convincing impact of software is directly on the pocketbook. For the Air Force, the estimated dollars for FY 1972 are in Fig. 5; an annual expenditure on software of between $1 billion and $1.5 billion, about three times the annual expenditure on computer hardware and about 4 to 5 percent of the total Air Force budget. Similar figures hold elsewhere. The recent World Wide Military Command and Control System (WWMCCS) computer procurement was estimated to involve expenditures of $50 to $100 million for hardware and $722 million for software.[1] A recent estimate for NASA was an annual expenditure of $100 million for hardware, and $200 million for software--about 6 percent of the annual NASA budget.

---

[1] *Datamation*, March 1, 1971, p. 41.

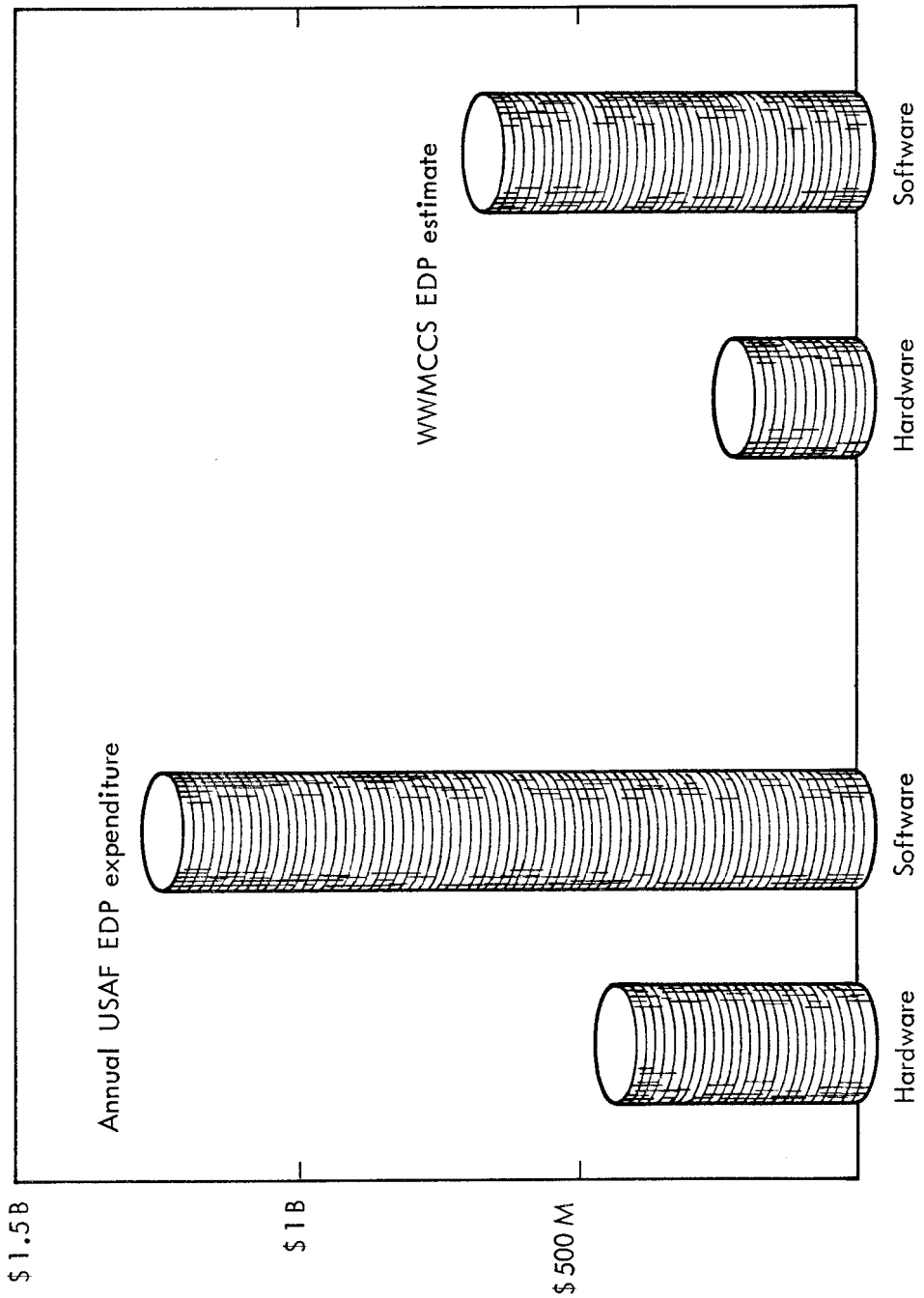

Fig. 5—USAF software is big business

For some individual projects, here are some overall software costs:

IBM OS/360	$ 200,000,000[2]
SAGE	250,000,000[3]
Manned Space Program, 1960-70	1,000,000,000[3]

Overall software costs in the U.S. are probably over $10 billion per year, over one percent of the gross national product.

If the software-hardware cost ratio appears lopsided now, consider what will happen in the years ahead, as hardware gets cheaper and software (people) costs go up and up. Figure 6 shows the estimate for software expenditures in the Air Force going to over 90 percent of total ADP system costs by 1985; this trend is probably characteristic of other organizations, also.

One would expect that current information-processing research and development projects would be strongly oriented toward where the future problems are. However, according to recent Congressional testimony by Dr. Ruth Davis of NBS on federally-funded computing R&D projects:

> ...21 percent of the projects were concerned with hardware design, 40 percent were concerned with the needs of special interest communities such as natural sciences, engineering, social and behavioral sciences, humanities, and real-time systems, 14 percent were in the long-range payoff areas of metatheory, while only 9 percent were oriented to the highly agonizing software problems identified by most customers as their major concern.[4]

One result of the CCIP-85 Study has been to begin to reorient Air Force information processing R&D much more toward software. Similar R&D trends are evident at DOD's Advanced Research Projects Agency (ARPA), National Science Foundation, and the National Bureau of Standards. But much remains to be done.

---

[2] Alexander, T., "Computers Can't Solve Everything," *Fortune*, May 1969.

[3] Boehm, B. W., "System Design," in *Planning Community Information Utilities*, (eds.) H. Sackman and B. W. Boehm, AFIPS Press, 1972.

[4] "Government Bureau Takes on Role of Public Protector Against Computer Misuse," *ACM Communications*, November 1972, p. 1018.

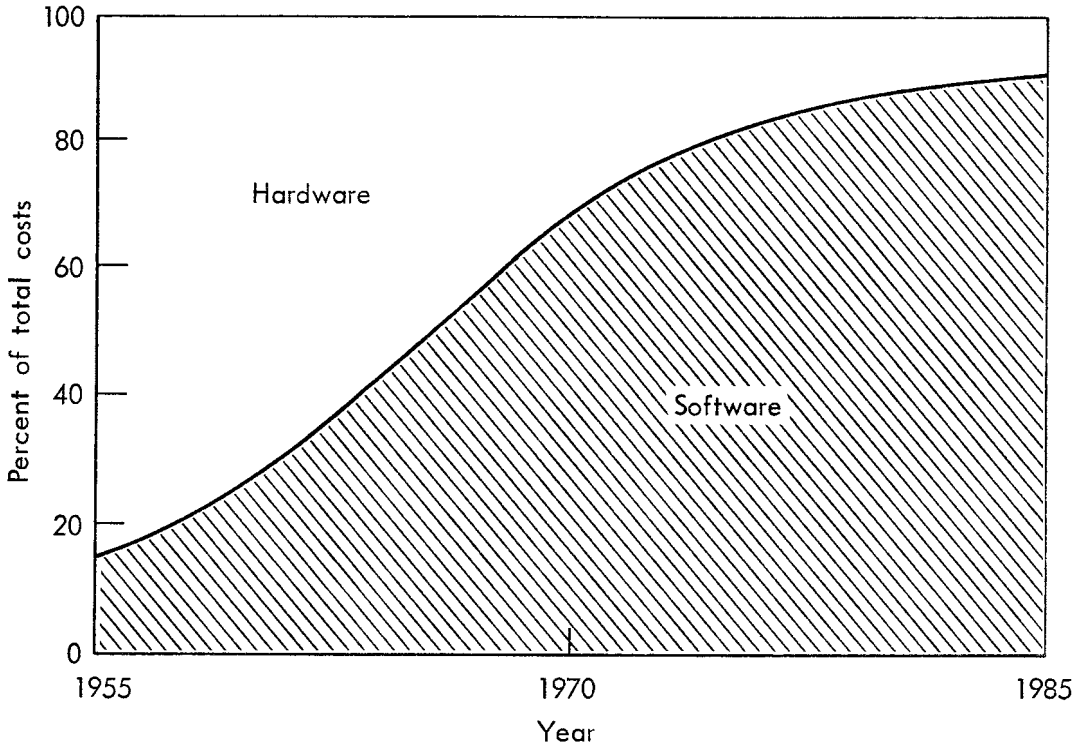

Fig. 6—Hardware/software cost trends

INDIRECT COSTS DUE TO SOFTWARE ARE EVEN BIGGER

Big as the direct costs of software are, the indirect costs are even bigger, because software generally is on the critical path in overall system development. That is, any slippages in the software schedule translate directly into slippages in the overall delivery schedule of the system.

Let's see what this meant in a recent software development for a large defense system. It was planned to have an operational lifetime of seven years and a total cost of about $1.4 billion--or about $200 million a year worth of capability. However, a six-month software delay caused a six-month delay in making the system available to the user, who thus lost about $100 million worth of needed capability--about 50 times the direct cost of $2 million for the additional software effort. Moreover, in order to keep the software from causing further delays, several important functions were not provided in the initial delivery to the user.

Again, similar situations develop in domestic applications. IBM's OS/360 software was over a year late.[2] The U.S. air traffic control system currently operates much more expensively and less effectively because of slippages of years in software (and also hardware, in this case) development, which have escalated direct software costs to over $100 million.[5] Often, organizations compensate for software development slippages by switching to a new system before the software is adequately tested, leading to such social costs as undelivered welfare checks to families with dependent children, bad credit reports, and even people losing their lives because of errors in medical software.

GETTING SOFTWARE OFF THE CRITICAL PATH

Once software starts slipping along the critical path, there are several more or less unattractive options. One option

---

[5]Hirsch, P., "What's Wrong With the Air Traffic Control System," *Datamation*, August 1972, pp. 48-53.

is to add more people in hopes that a human wave of programmers will quickly subdue the problem. However, Brooks' excellent article[6] effectively shows that software is virtually incompressible with respect to elapsed time, and that such measures more often make things worse rather than better. Some other unhappy options are to skimp on testing, integration, or documentation. These usually cost much more in the long run. Another is just to scrap the new system and make do with the old one. Generally, the most attractive option is to reduce the system to an austere but expandable initial capability.

For the future, however, several opportunities exist for reducing software delays and getting software off the critical path. These fall into three main categories:

1. Increasing each individual's software productivity
2. Improving project organization and management
3. Initiating software development earlier in the system development cycle.

INCREASING SOFTWARE PRODUCTIVITY: DEFINITIONS

Figure 7 shows a simplistic view of likely future trends in software productvity. It is probably realistic in maintaining at least a factor-of-10 spread between the 10th and 90th percentiles of software productivity, but it begs a few important questions.

One is, "What is software?" Even the courts and the Internal Revenue Service have not been able to define its metes and bounds precisely. The figures above include computer program documentation, but exclude operating procedures and broad system analysis. Clearly, a different definition would affect software productivity figures significantly.

---

[6]Brooks, F., "Why Is The Software Late?" *Data Management*, August 1971.

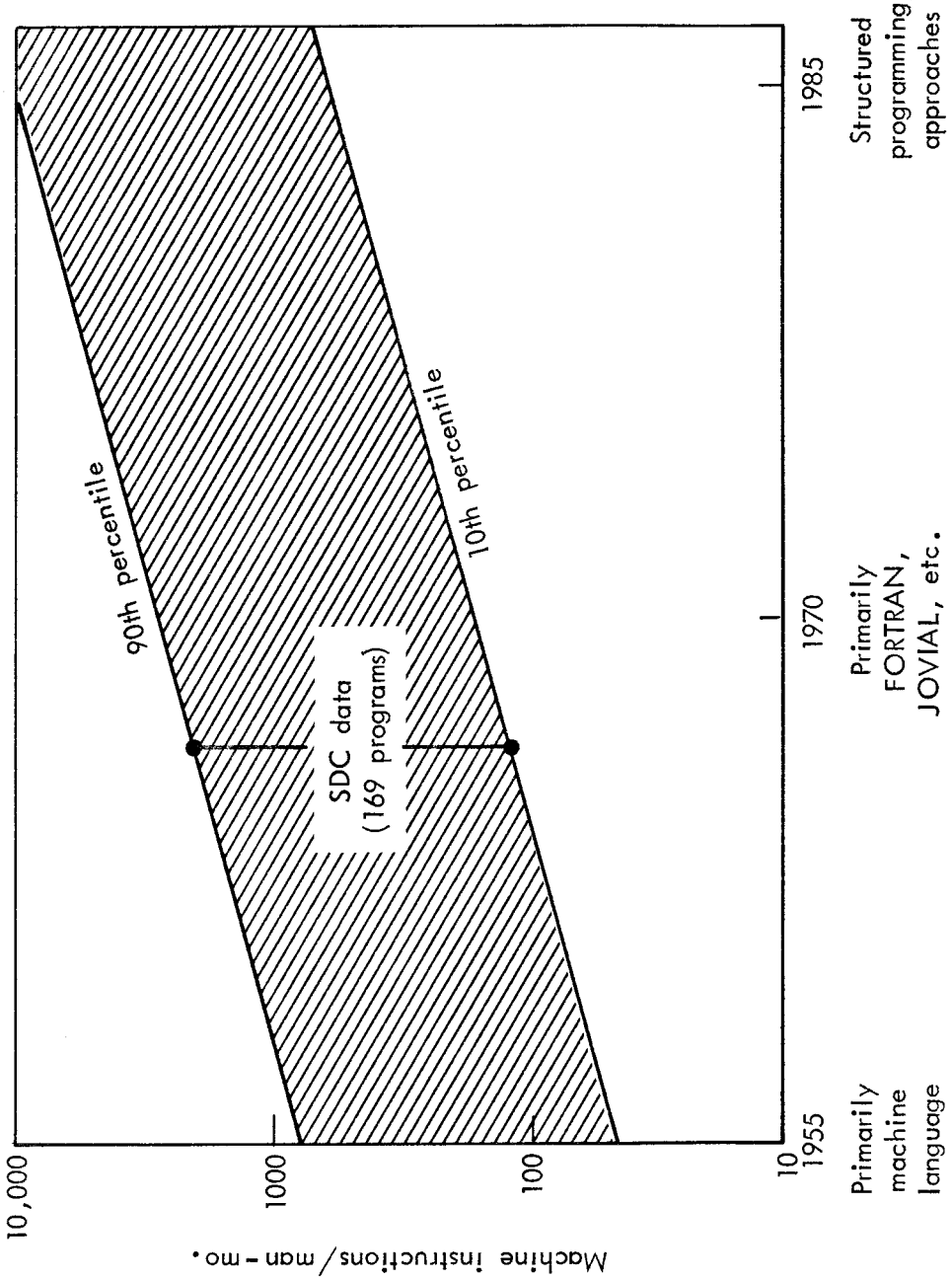

Fig. 7— Technology forecast: software productivity

Another important question is, "What constitutes software production?" As early as the mid-1950s there were general-purpose trajectory analysis systems with which an analyst could put together a modular, 10,000-word applications program in about 10 minutes. Was this "software production?" With time, more and more such general-purpose packages as ICES (MIT's Integrated Civil Engineering System), Programming-by-Questionnaire, RPG, MARK IV, and SCERT, have made the creation of significant software capabilities so easy that they tend to be eliminated from the category of "software productivity," which continues to refer to those portions of the software directly resulting from handwritten strings of assembly or FORTRAN-level language statements. Figure 8 is an attempt to characterize this trend in terms of a "50 percent automation date:" the year in which most of the incoming problems in an area could be "programmed" in less than an hour by a user knowledgeable in his field, with one day of specialized training.

Thus, if we want to speak objectively about software productivity, we are faced with the dilemma of:

> either redefining it in terms of source instructions rather than object instructions--thereby further debasing the unit of production (which isn't completely objective even using object instructions as a base);
>
> or, continuing to narrow the range of definition of "software productivity" to the more and more difficult programs which can't be put together more or less automatically.

The eventual result of ARPA's major "automatic programming" effort will be to narrow this latter range even further.[7]

---

[7] Balzer, Robert M., *Automatic Programming*, Institute Technical Memorandum, University of Southern California, Information Sciences Institute, September 1972.

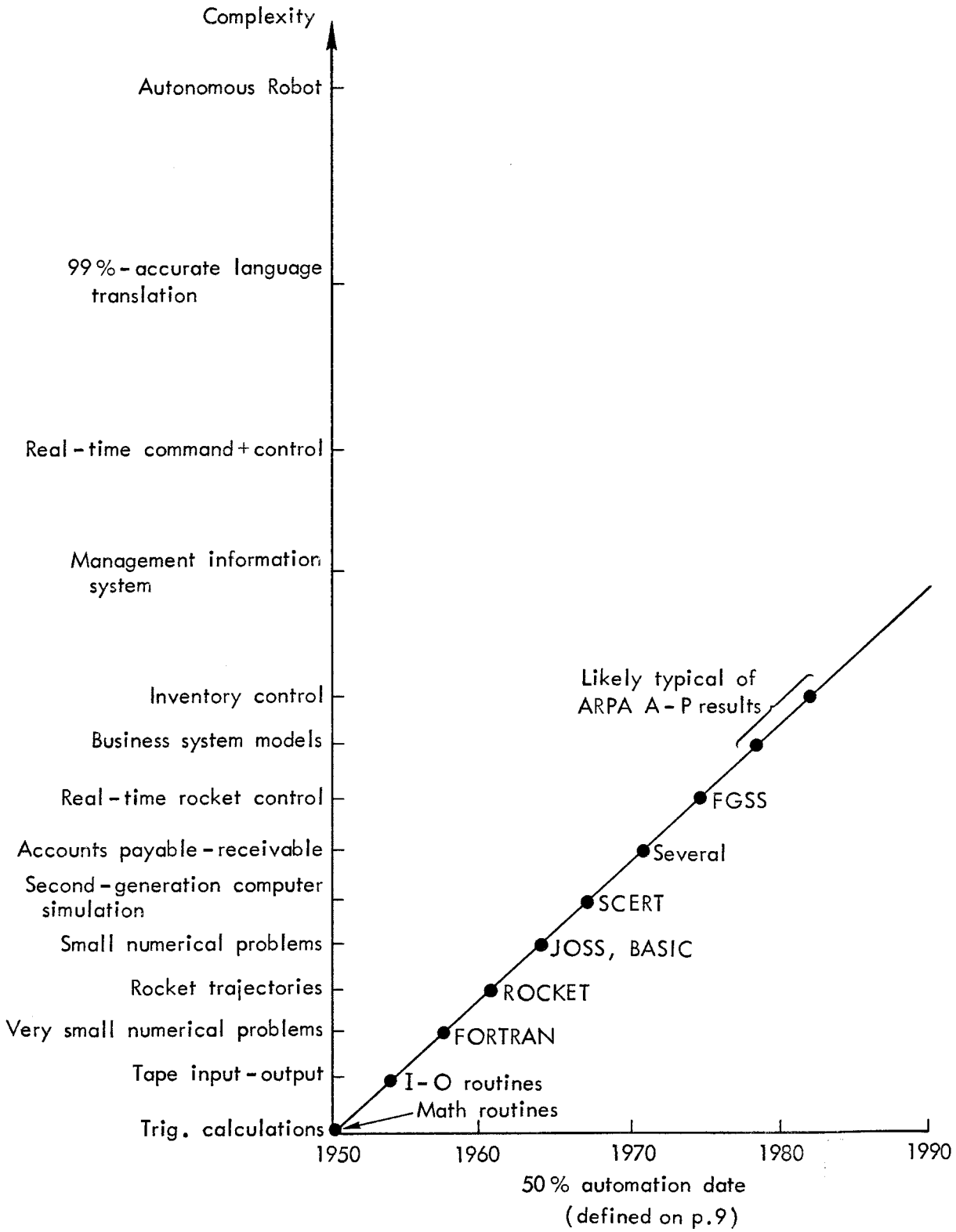

Fig. 8—Growth of automatic programming

INCREASING SOFTWARE PRODUCTIVITY: FACTORS

However, the fact remains that software needs to be constructed, that various factors significantly influence the speed and effectiveness of producing it, and that we have at least some measure of control over these factors. Thus, the more we know about those factors, the more our decisions will lead to improved rather than degraded software productivity. What are the important factors?

One is computer system response time. Studies by Sackman and others[8] comparing batch versus on-line programming have shown median improvements of 20 percent in programming efficiency using on-line systems.

However, in these same studies, variations between individuals accounted for differences in productivity of factors up to 26:1. Clearly, selecting the right people provides more leverage than anything else in improving software productivity. But this isn't so easy. Reinstedt[9] and others have shown that none of the selection tests developed so far have an operationally-dependable correlation with programmer performance. Weinberg, in his excellent book,[10] illustrates the complexity of the issue by citing two programmer attributes for each letter of the alphabet (from age and agility through zygosity and zodiacal sign), each of which might be a plausible determinant of programmer performance. Still, the potential payoffs are so large that further work in the areas of personnel selection, training, and evaluation should be closely followed. For example, the Berger Test of Programming Proficiency has proved fairly reliable in assessing the programming capability of experienced programmers.

---

[8] Sackman, H., *Man-Computer Problem Solving*, Auerbach Publishers, Inc., 1970.

[9] Reinstedt, R. N., "Results of a Programmer Performance Prediction Study," *IEEE Trans. Engineering Management*, December 1967, pp. 183-187.

[10] Weinberg, G., *The Psychology of Computer Programming*, Van Nostrand Reinhold, 1971.

Other factors such as programming languages have made significant differences in software productivity. Rubey's PL/I study showed differences of up to 2:1 in development time for the same program written in two different languages. In a related effort, Kosy obtained a 3.5:1 productivity improvement over one of the Rubey examples by using ECSS, a special-purpose language for simulating computer systems.

Weinberg has also shown[10,11] that the choice of software development criteria exerts a significant influence on software productivity. In one set of experiments, programmers were given the same program specification, but were told either (Group P) to finish the job as promptly as possible, or (Group E) to produce as efficient a program as possible. The results were that Group E finished the job with an average of over twice as many runs to completion, but with programs running an average of 6 times faster.

Another important factor is the software learning curve. The table below shows the estimated and actual programming effort involved in producing three successive FORTRAN compilers by the same group.[12]

Compiler Effort No.	Man Months	
	Estimated	Actual
1	36	72
2	24	36
3	12	14

Clearly, software estimation accuracy has a learning curve, also.

---

[11] Weinberg, G. F., "The Psychology of Improved Programming Performance," *Datamation*, November 1972.

[12] McClure, R. M., "Projection vs Performance in Software Production," in *Software Engineering*, (eds.) P. Naur, and B. Randell, NATO, January 1969.

But other factors in the programming environment make at least as large a contribution on any given project. The most exhaustive quantitative analysis done so far on the factors influencing software development was an SDC study done for the Air Force Electronic Systems Division in 1965,[13] which collected data on nearly 100 factors over 169 software projects and performed extensive statistical analysis on the results. The best fit to the data involved 13 factors, including stability of program design, percent mathematical instructions, number of subprograms, concurrent hardware development, and number of man-trips--but even that estimate had a standard deviation (62 man-months) larger than the mean (40 man-months).

INCREASING SOFTWARE PRODUCTIVITY: PRESCRIPTIONS

Does all this complexity mean that the prospect of increasing software productivity is hopeless? Not at all. In fact, some of the data provide good clues toward avenues of improvement. For example, if you accurately answered question 1 on the Software Quiz, you can see that only 15 percent of a typical software effort goes into coding. Clearly, then, there is more potential payoff in improving the efficiency of your analysis and validation efforts than in speeding up your coding.

Significant opportunities exist for doing this. The main one comes when each of us as individual programmers becomes aware of where his time is really going, and begins to design, develop and use thoughtful test plans for the software he produces, beginning in the earliest analysis phases. Suppose that by doing so, we could save an average of one man-day per man-month of testing effort. This would save about 2.5 percent of our total expenditure on software. Gilchrist

---

[13] Nelson, E. A., *Management Handbook for the Estimation of Computer Programming Costs*, SDC, TM-3224, October 31, 1966.

and Weber[14] estimate about 360,000 software practitioners in the U.S.; even at a somewhat conservative total cost quotation of $30,000 per man-year, this is about $10.8 billion annually spent on software, yielding a testing savings of about $270 million per year.

Another opportunity lies in the area of programming languages. Except for a few experiments such as Floyd's "Verifying Compiler," programming languages have been designed for people to express programs with a minimum of redundancy, which tends to expedite the coding process, but makes the testing phase more difficult. Appropriate additional redundancy in a program language, requiring a programmer to specify such items as allowable limits on variables, inadmissible states and relations between variables,[15] would allow a compiler or operating system to provide much more help in diagnosing programming errors and reducing the time-consuming validation phase. For example, of the 93 errors detected during execution in Rubey's PL/I study, 52 could have been caught during compilation with a validation-oriented programming language containing features such as those above.

Another avenue to reducing the validation effort lies in providing tools and techniques which get validation done more efficiently during the analysis phase. This is the approach taken in structured programming. This term has been used to describe a variety of on-line programming tool boxes, programming systems, and innovative structurings of the software production effort. An example of the first is the Flexible Guidance Software System, currently being developed for the Air Force Space and Missile Systems Organization. The second is exemplified by the Technische Hogeschool Eindhoven (THE)[16]

---

[14] Gilchrist, B., and K. E. Weber, "Employment of Trained Computer Personnel--A Quantitative Survey," *Proceedings, 1972 SJCC*, p. 641-648.

[15] Kosy, D. W., *Approaches to Improved Program Validation Through Programming Language Design*, The Rand Corporation, P-4865, July 1972.

[16] Dijkstra, E. W., "The Structure of the 'THE' Multiprogramming System," *ACM Comm*, May 1968.

and automated engineering design (AED) systems, while innovative structuring may be seen in experiments such as the IBM chief programmer team (CPT) effort.[17] Although they are somewhat different, each concept represents an attempt to bring to software production a "top-down" approach and to minimize logical errors and inconsistencies through structural simplification of the development process. In the case of the THE system, this is reinforced by requiring system coding free of discontinuous program control ("GO-TO FREE"). In the chief programmer approach, it is accomplished by choosing a single individual to do the majority of actual design and programming and tailoring a support staff around his function and talents.

As yet, none of the systems or concepts described has been rigorously tested. Initial indications are, however, that the structured approach can shorten the software development process significantly, at least for some classes of programs and programmers. In one case, the use of AED reduced the man-effort of a small system from an envisioned six man-months to three man-weeks. A major experiment using the CPT concept (on an 83,000-instruction system for the New York *Times*) cut expected project costs by 50 percent and reduced development time to 25 percent of the initial estimate.

The validation statistics on this project were particularly impressive. After only a week's worth of system integration, the software went through five weeks of acceptance testing by *Times* personnel. Only 21 errors were found, all of which were fixed in one day. Since then during over a year's worth of operational experience, only 25 additional errors have been found in the 83,000-instruction package.[18]

At this point, it's still not clear to what extent this remarkable performance was a function of using remarkably skilled programming talent, and to what extent the performance

---

[17] Baker, F. T., "Chief Programmer Team," *IBM Systems Journal*, Vol. II, No. 1, 1972, pp. 56-73.

[18] Baker, F. T., "System Quality Through Structured Programming," *Proceedings, 1972 FJCC*, pp. 339-344.

gains could be matched by making a typical programming team into a Chief Programmer Team. Yet the potential gains were so large that further research, experimentation and training in structured programming concepts was one of the top priority recommendations of the CCIP-85 Study.

IMPROVING SOFTWARE MANAGEMENT

Even though an individual's software productivity is important, the CCIP-85 Study found that the problems of software productivity on medium or large projects are largely problems of management: of thorough organization, good contingency planning, thoughtful establishment of measurable project milestones, continuous monitoring on whether the milestones are properly passed, and prompt investigation and corrective action in case they are not. In the software management area, one of the major difficulties is the transfer of experience from one project to the next. For example, many of the lessons learned as far back as SAGE are often ignored in today's software developments, although they were published over 10 years ago in Hosier's excellent 1961 article on the value of milestones, test plans, precise interface specifications, integrated measurement capabilities, formatted debugging aids, early prototypes, concurrent system development and performance analysis, etc.[19]

Beyond this, it is difficult to say anything concise about software management that doesn't sound like motherhood. Therefore, this article will simply cite some good references in which the subject is explored in some detail.[20,21,22]

---

[19] Hosier, W. A., "Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming," *IRE Transactions on Engineering Management*, Vol. EM-8, June 1961, pp. 99-115.

[20] Naur, P. and B. Randell (eds.) *Software Engineering*, NATO Science Committee, January 1969.

[21] Buxton, J. N. and B. Randell, (eds.) *Software Engineering Techniques*, NATO Science Committee, April 1970.

[22] Weinwurm, G., (ed.) *On the Management of Computer Programming*, Auerbach, 1970.

GETTING AN EARLIER START: THE SOFTWARE-FIRST MACHINE

Even if software productivity never gets tremendously efficient, many of the most serious software agonies would be alleviated if we could get software off the critical path within an overall system development. In looking at the current typical history of a large software project (Fig. 9) you can see that the year (or often more) spent on hardware procurement pushes software farther out onto the critical path, since often the software effort has to wait at least until the hardware source selection is completed.

One of the concepts developed in the CCIP-85 Study for getting software more off the critical path was that of a "Software-First Machine." This is a highly generalized computer, capable of simulating the behavior of a wide range of hardware configurations. Figure 10 provides a rough plan of such a "software first" machine. It would have the capability of configuring and exercising, through its microprogrammed control, a range of computers, and could also simultaneously provide some additional hardware aids to developing and testing software.

Suppose a large organization such as the Air Force owned such a machine. The following events could then take place: a contractor who is trying to develop software for an airborne computer could start with a need for a machine which is basically the IBM 4PI, but with a faster memory and different interrupt structure. This software contractor could develop, exercise, store, and recall his software based on the microprogrammed model of the machine. When it turned out that this architecture was hampering the software developers, they could do some hardware/software tradeoffs rather easily by changing the microprogrammed machine representation; and when they were finished or essentially finished with the software development, they would have detailed design specifications for the hardware that could be produced through competitive procurement in industry.

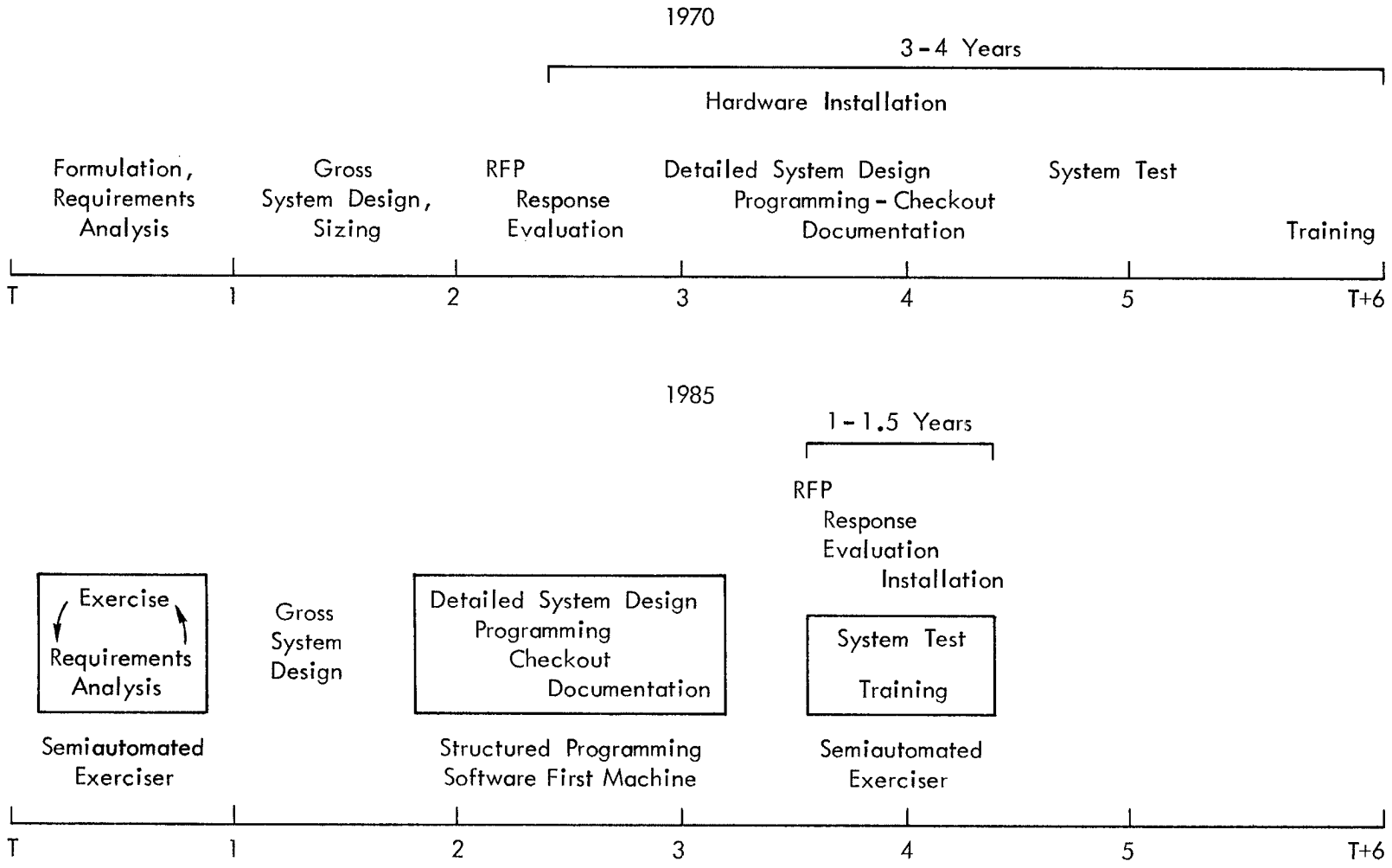

Fig. 9—The software development cycle

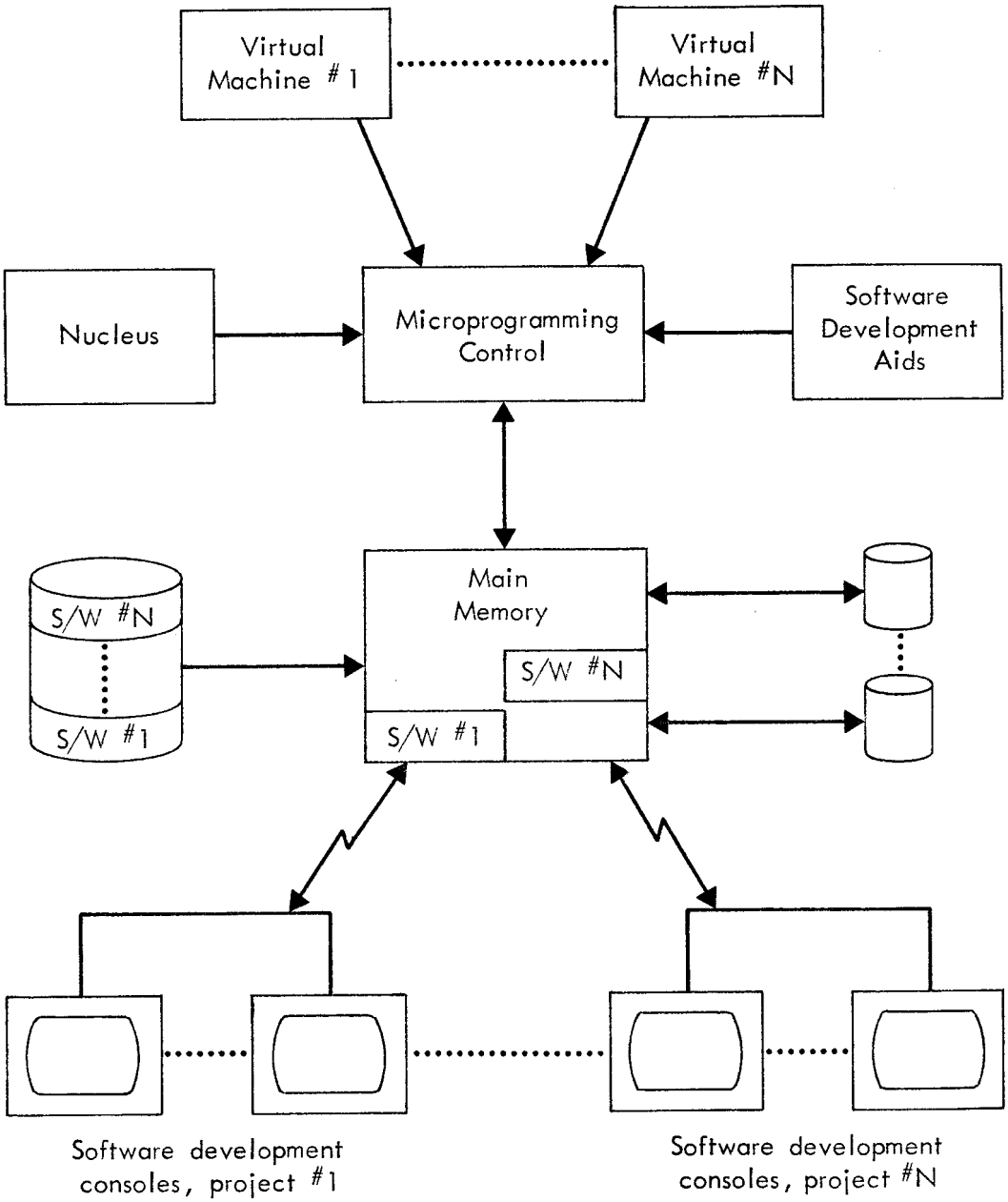

Fig. 10—Software-first machine concept

Similarly, another contractor could be developing software for interface message processors for communications systems, based on variants of the Honeywell DDP 516; another could be improving a real-time data processing capability based on an upgrade of a CDC 3800 computer on another virtual machine.

The software-first machine could be of considerable value in shortening the time from conception to implementation of an integrated hardware/software system. In the usual procurement process (Fig. 9), the hardware is chosen first, and software development must await delivery of the hardware. With the software-first machine, software development can avoid this wait, as hardware procurement can be done during the system test phase; the necessary hardware fabrication will start from a detailed design and, with future fabrication technology, should not introduce delays. This saving translates also into increased system operating life, as the hardware installed in the field is based on more up-to-date technology.

However, the software-first machine concept has some potential drawbacks. For example, it might produce a "centrifugal tendency" in hardware development. Allowing designers to tailor hardware to software might result in the proliferation of a variety of similar although critically different computers, each used for a special purpose.

A final question concerning the software-first machine remains moot: Can it be built, at any rate, at a "reasonable" cost? Architectures such as the CDC STAR, Illiac IV, and Goodyear STARAN IV would be virtually impossible to accommodate in a single machine. Thus, it is more likely that various subsets of the Software-First Machine characteristics will be developed for various ranges of applications.

One such variant is underway already. One Air Force organization, wishing to upgrade without a simultaneous hardware and software discontinuity, acquired some Meta 4 microprogrammed machines which will originally be installed to emulate the

existing second-generation hardware. Once the new hardware is in operation, they will proceed to upgrade the system software using a different microprogrammed base. In this way they can upgrade the system with a considerably reduced risk of system down time.

Another existing approach is that of the microprogrammed Burroughs B1700, which provides a number of the above characteristics plus capabilities to support "direct" execution of higher-level language programs.

OTHER HARDWARE-SOFTWARE TRADEOFFS

In addition, there are numerous other ways in which cheaper hardware can be traded off to save on more expensive software development costs. A most significant one stems from the striking difference between "folklore" and "experience" in the hardware-software curves shown in Fig. 2B of the Software Quiz. This tradeoff opportunity involves buying enough hardware capacity to keep away from the steep rise in software costs occurring at about the 85 percent saturation point of CPU and memory capacity.

Thus, suppose that one has sized a data-processing task and determined that a computer of one-unit capacity (with respect to central processing unit speed and size) is required. Figure 11 shows how the total data-processing system cost varies with the amount of excess CPU capacity procured for various estimates of the ratio of ideal software-to-hardware costs for the system.* The calculations are based on the previous curve of programming costs and two models of hardware cost: the linear model assumes that cost increases linearly with increases in CPU capacity; the "Grosch's Law" model assumes that cost increases as the square root of CPU capacity. Sharpe's data[23] indicates that most applications fall somewhere between these models.

---

*"Ideal software" costs are those that would be incurred without any consideration of straining hardware capacity.

[23]Sharpe, W. F., *The Economics of Computers*, Columbia University Press, 1969.

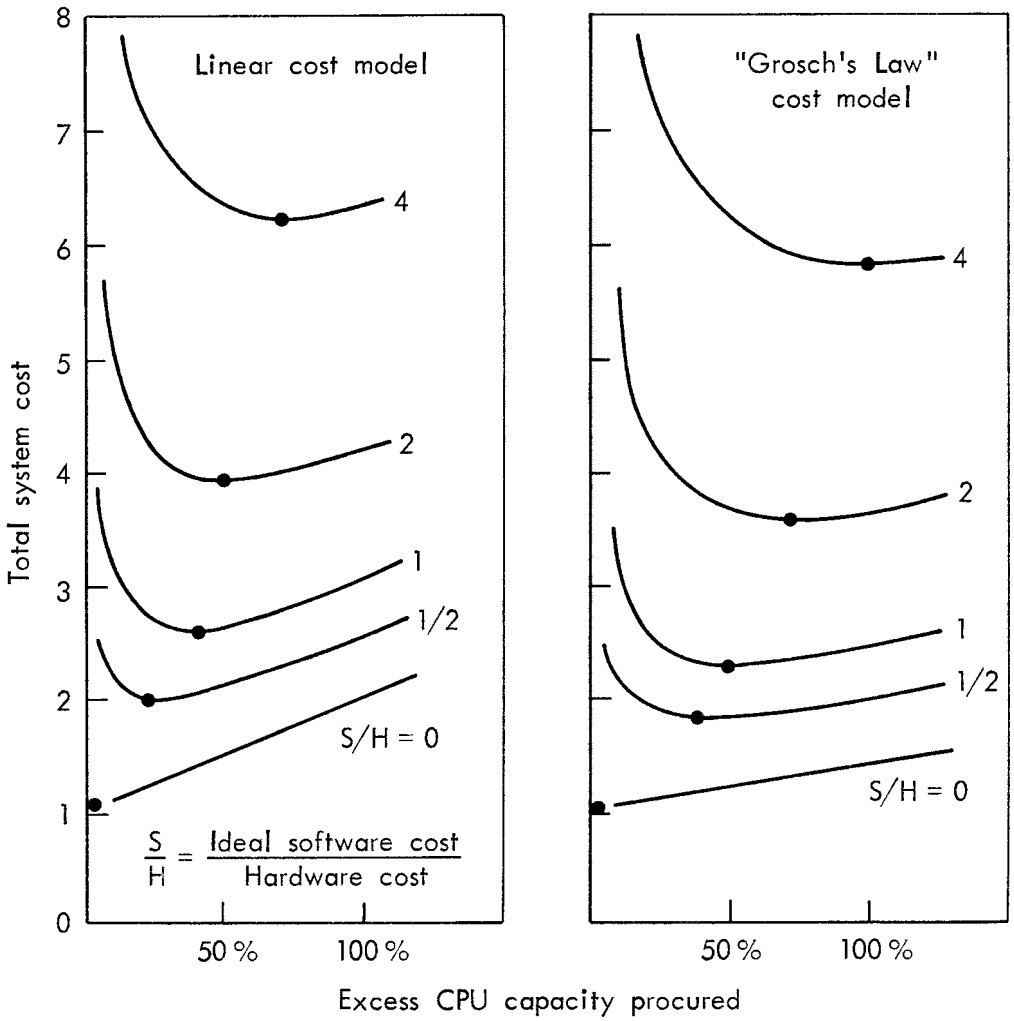

Fig. 11—Hardware-software system costs

It should be remembered that the curves are based on imprecise observations; they clearly cannot be used in "cookbook" fashion by system designers. But even their general trends make the following points quite evident:

(1) Overall system cost is generally minimized by procuring computer hardware with at least 50 percent to 100 percent more capacity than is absolutely necessary.

(2) The more the ratio of software-to-hardware cost increases (as it will markedly during the seventies), the more excess computing capacity one should procure to minimize the total cost.

(3) It is far more risky to err by procuring a computer that is too small than one that is too large. This is especially important, since one's initial sizing of the data-processing job often tends to underestimate its magnitude.

Of course, buying extra hardware does not eliminate the need for good software engineering thereafter. Careful configuration control must be maintained to realize properly the benefits of having extra hardware capability, as there are always strong Parkinsonian tendencies to absorb excess capacity with marginally useful tasks.

SOFTWARE RESPONSIVENESS

Another difficulty with software is its frequent unresponsiveness to the actual needs of the organization it was developed for. For example, the hospital information system field has several current examples of "wallflower" systems which were developed without adequately consulting and analyzing the information requirements of doctors, nurses, and hospital administrators. After trying to live with these systems for a while, several hospital administrators have

reluctantly but firmly phased them out with such comments as, "We know that computers are supposed to be the way to go for the future, but this system just doesn't provide us any help," or, "Usage of the system began at a very low level--and dropped off from there."

The main difficulties stem from a lack of easily transferable procedures to aid in the software requirements analysis process. This process bears on all-too-striking resemblance to the class of folktales in which a genie comes up to a man and tells him he has three wishes and can ask for anything in the world. Typically, he spends his first two wishes asking for something like a golden castle and a princess, and then when he discovers the operations, maintenance, and compatibility implications of his new acquisitions, he is happy to spend the third wish getting back to where he started.

Similarly, the computer is a sort of genie which says, "I'll give you any processed information you want. All you need to do is ask--by writing the software to process it." Often, though, we go the man in the folktale one better by canvassing a number of users (or non-users) and putting their combined wish lists into a software requirements analysis. But our technology base for assessing the operations, maintenance, and compatibility implications of the resulting software system is just as inadequate. Thus, large airline reservations software developments (Univac/United, Burroughs/TWA) have reached the point that the customer preferred to wish them out of existence rather than continue them--but only after the investment of tens of millions of dollars. In other cases, where no alternative was available, software rewrites of up to 67 percent (and in one very large system, 95 percent) have taken place--<u>after</u> the "final" software package had been delivered--in order to meet the user's operational needs.

Considering the major needs for better requirements analysis techniques, the relative lack of available techniques, and the added fact (from Fig. 1B of the Software Quiz) that about 35 percent of the total software effort goes into analysis and design, it is not surprising that the top-priority R&D recommendation made by the CCIP-85 Study was for better techniques for performing and validating information system requirements analyses, and for generating and verifying the resulting information system designs.

The recent *Datamation* articles on automated system design[24,25] indicated some promising initial developments in this area such as Teichroew's ISDOS project, FOREM, and IBM's TAG (Time-Automated Grid) system. Other significant aids are being developed in the area of special languages and packages such as SCERT, CASE, CSS, SAM, and ECSS to accelerate the process of design verification by simulating information-processing systems. Also, ARPA's major research effort in automatic programming is focused strongly on automating the analysis and design processes.[7]

SOFTWARE RELIABILITY AND CERTIFICATION

Another major area in which the CCIP-85 Study identified a serious mismatch between future needs and likely software capabilities was in the area of software certification: of providing guarantees that the software will do what it is supposed to do.*

---

[24] Teichroew, D., and H. Sayari, "Automation of System Building," *Datamation*, August 14, 1971, pp. 25-30.

[25] Head, R. V., "Automated System Analysis," *Datamation*, August 14, 1971, pp. 23-24.

*Other significant problem or opportunity areas identified by CCIP-85 included (in order) data security, airborne computing power, multisource data fusion, data communications, source data automation, image processing, performance analysis, parallel processing, and software transferability.

This is a significant concern right now, but it becomes even more pressing when one extrapolates current trends toward more complex software tasks and toward more and more automated aids to decisionmaking. Just consider the trends implicit in the results of the recent AFIPS/Time Survey[26] which indicated that currently 30 percent of the labor force must deal with computers in their daily work, but only 15 percent of the labor force is required to have any understanding of computers. Extrapolating this trend into the 1980s, as is done in Fig. 12, indicates that perhaps 40 percent of the labor force will be trusting implicitly in the results produced by computer software.

SOFTWARE RELIABILITY: PROBLEM SYMPTOMS

Will software be deserving of such trust? Not on its past record. For example, some of the most thoroughly tested software in the world is that of the Apollo manned spaceflight efforts. Yet on Apollo 8, an unforeseen sequence of astronaut actions destroyed the contents of a word in the computer's erasable memory--fortunately, not a critical error in this case. And on Apollo 11, the data flow from the rendezvous radar was not diverted during the critical lunar landing sequence, causing a computer overload that required Astronaut Armstrong to divert his attention from the process of landing the spacecraft--fortunately again, without serious consequences. And during the ten-day flight of Apollo 14, there were 18 discrepancies found in the software--again fortunately, without serious consequences.

---

[26] *A National Survey of the Public's Attitudes Toward Computers*, AFIPS and Time, Inc., November 1971.

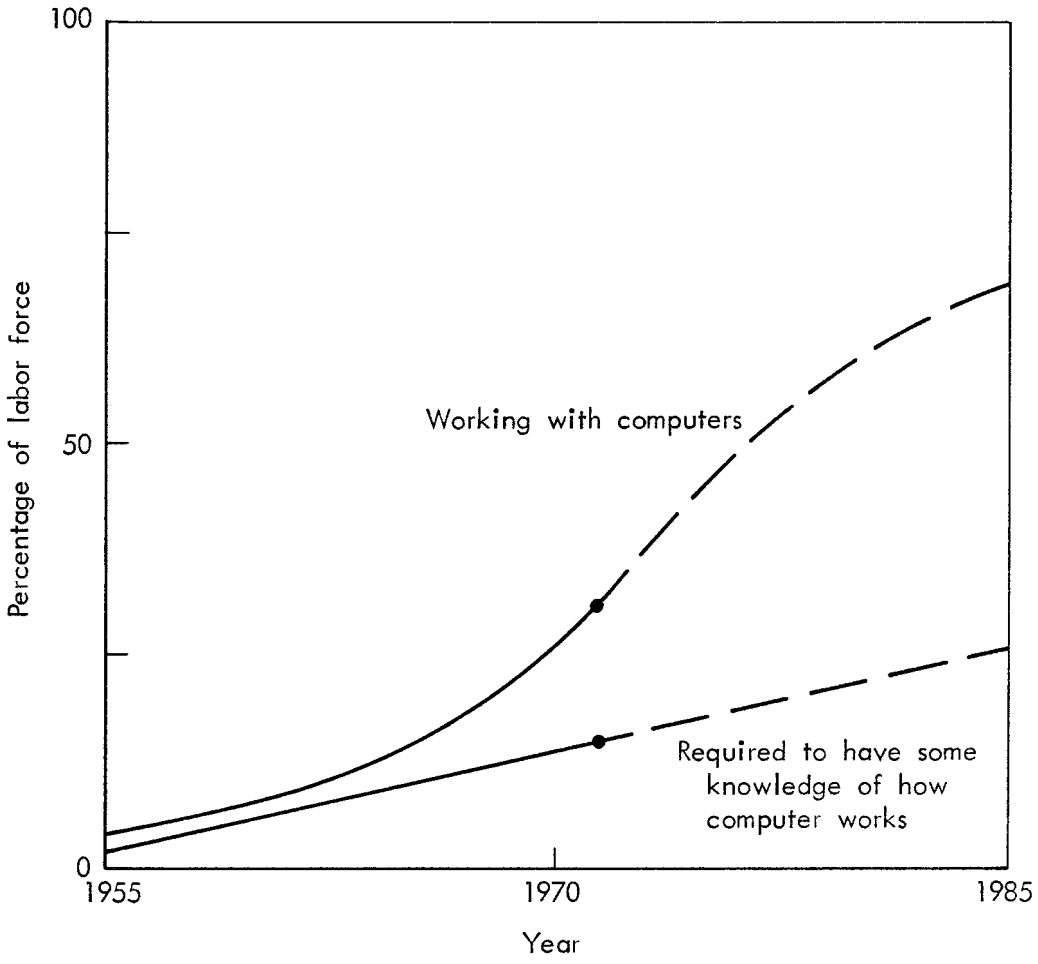

Fig. 12—Growth of trust in computers and software

Other space missions haven't been so fortunate. Recently a software error aboard a French meteorological satellite caused it to "emergency destruct" 72 out of 141 high-altitude weather balloons, instead of interrogating them. An early U.S. Mariner interplanetary mission was lost due to a software error. And the Soviet Union has had missions fail because of software errors.

Down on Earth, software reliability isn't any better. Each new release of OS/360 contains roughly 1000 new software errors. On one large real-time system containing about 2,700,000 instructions and undergoing continuous modifications, an average of one software error per day is discovered. Errors in medical software have caused people to lose their lives. And software errors cause a constant stream of social dislocations due to false arrests, incorrect bank balances or credit records, lost travel reservations, or long-delayed payments to needy families or small businesses. Also, lack of certification capabilities makes it virtually impossible to provide strong guarantees on the security or privacy of sensitive or personal information.

SOFTWARE RELIABILITY: TECHNICAL PROBLEMS

As the examples above should indicate, software certification is not easy. Ideally, it means checking all possible logical paths through a program; there may be a great many of these. For example, Fig. 13 shows a rather simple program flowchart. Before looking at the accompanying text, try to estimate how many different possible paths through the flowchart exist.

HOW MANY DIFFERENT PATHS THROUGH THIS FLOWCHART?

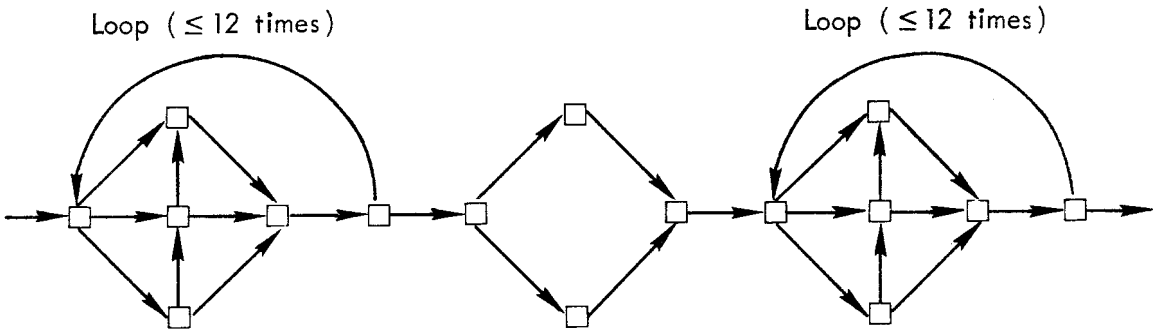

Fig. 13

Even through this simple flowchart, the number of different paths is about ten to the twentieth. If one had a computer that could check out one path per nanosecond (10^{-9} sec), and had started to check out the program at the beginning of the Christian era (1 A.D.), the job would be about half done at the present time.

So how does one certify a complex computer program that has incredibly more possible paths than this simple example? Fortunately, almost all of the probability mass in most programs goes into a relatively small number of paths that can be checked out.

But the unchecked paths still have some probability of occurring. And, furthermore, each time the software is modified, some portion of the testing must be repeated.

Figure 14 shows that, even for small software modifications, one should not expect error-free performance thereafter. The data indicate that small modifications have a better chance of working successfully than do large ones. However, even after a small modification the chance of a successful first

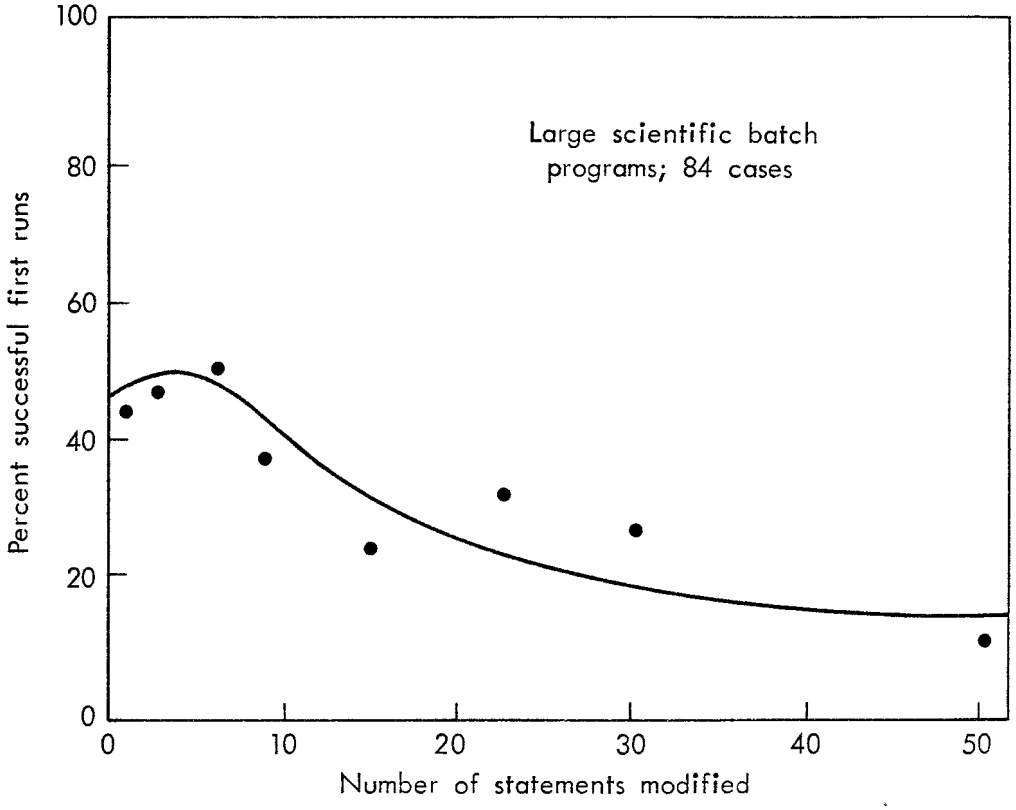

Fig. 14—Reliability of software modifications

run is, at best, about 50 percent. In fact, there seems to be a sort of complacency factor operating that makes a successful first run less probable on modifications involving a single statement than on those involving approximately five statements--at least for this sample.

At this point, it's not clear how representative this sample is of other situations. One roughly comparable data point is in Fig. 3B of the Software Quiz, in which only 7 percent of the errors detected were those made in trying to correct previous errors. The difference in error rates is best explained by both the criticality of the application and the fact that the modifications were being made in a software validation rather than a software maintenance environment.

In another analysis of software error data performed for CCIP-85 by McGonagle,[27] 19 percent of the errors resulted from "unexpected side effects to changes." Other sources of errors detected over three years of the development cycle of a 24,000-instruction command and control program are shown in the table below. These data are of particular interest because they provide insights into the causes of software errors as well as their variation with type of program.

DISTRIBUTION OF SOFTWARE ERROR CAUSES

Causes of Error	Hardware Diagnostics (%)	Executive (%)	User Programs (%)	Total (%)
Unexpected side effects to changes	5	25	10	19
Logical flaws in the design				
Original design	5	10	2	8
Changes	5	15	8	12
Inconsistencies between design and implementation	5	30	10	22
Clerical errors	40	20	50	28
Inconsistencies in hardware	40	--	20	11
	100	100	100	100
Total errors detected, 3 yr sample	36	108	18	162
Number of instructions	4K	10K	10K	24K

[27] McGonagle, J. D., *A Study of a Software Development Project*, James P. Anderson, and Co., September 21, 1971.

SOFTWARE CERTIFICATION TECHNOLOGY

Against the formidable software certification requirements indicated above, the achievements of current technology leave a great deal to be desired. One organization paid $750,000 to test an 8,000-instruction program, and even then couldn't be guaranteed that the software was perfect, because testing can only determine the presence of errors, not their absence. The largest program that has been mathematically proved correct was a 433-statement Algol program to perform error-bounded arithmetic; the proof required 46 pages of mathematical reasoning.[28]

However, there are several encouraging trends. One is the impressive reduction of errors achieved in the structured programming activities discussed earlier in this article. Another is the potential contribution of appropriately redundant programming languages, also discussed earlier. A third trend is the likely development of significant automated aids to the program-proving process, currently an extremely tedious manual process. Another is the evolutionary development and dissemination of better software test procedures and techniques and the trend toward capitalizing on economies of scale in validating similar software items, as in the DOD COBOL Compiler Validation System. But even with these trends, it will take a great deal of time, effort, and research support to achieve commonly usable solutions to such issues as the time and cost of analytic proof procedures, the level of expertise required to use them, the difficulty of providing a valid program specification to serve as a certification standard, and the extent to which one can get software efficiency and validability in the same package.

WHERE'S THE SOFTWARE ENGINEERING DATA BASE?

One of the major problems the CCIP-85 Study found was the dearth of hard data available on software efforts which would allow us to analyze the nature of software problems, to convince people unfamiliar with software that the problems

---

[28] Good, D. I., and R. L. London, "Computer Interval Arithmetic: Definition and Proof of Correct Implementation," *ACM Journal*, October 1970, pp. 603-612.

were significant, or to get clues on how best to improve the situation. Not having such a data base forces us to rely on intuition when making crucial decisions on software, and I expect, for many readers, your success on the Software Quiz was sufficiently poor to convince you that software phenomena often tend to be counterintuitive. Given the magnitude of the risks of basing major software decisions on fallible intuition, and the opportunities for ensuring more responsive software by providing designers with usage data, it is surprising how little effort has gone into endeavors to collect and analyze such data. Only after a decade of R&D on heuristic compilers, optimizing compilers, self-compiling compilers and the like, has there been an R&D effort to develop a usage-measuring compiler. Similar usage-measuring tools could be developed for keeping track of error rates and other software phenomena.

One of the reasons progress has been slow is that it's just plain difficult to collect good software data--as we found on three contract efforts to do so for the CCIP-85 Study. These difficulties included:

- Deciding which of the thousands of possibilities to measure;
- Establishing standard definitions for "error," "test phase," etc.;
- Establishing what had been the development performance criteria;
- Assessing subjective inputs such as "degree of difficulty," "programmer expertise," etc.;
- Assessing the accuracy of post facto data;
- Reconciling sets of data collected in differently defined categories.

Clearly, more work on these factors is necessary to insure that future software data collection efforts produce at least roughly comparable results. However, because the data collection

problem is difficult doesn't mean we should avoid it. Until we establish a firm data base, the phrase "software engineering" will be largely a contradiction in terms. And the software components of what is now called "computer science" will remain far from Lord Kelvin's standard:

> When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of <u>science</u>.

But, in closing, I'd like to suggest that people should collect data on their software efforts because it's really in their direct best interest. Currently, the general unavailability of such software data means that whoever first provides system designers with quantitative software characteristics will find that the resulting system design tends to be oriented around his characteristics.

For example, part of the initial design sizing of the ARPA Network was based on two statistical samples of user response, on Rand's JOSS system and on MIT's Project MAC. This was not because these were thought to be particularly representative of future Network users; rather, they were simply the only relevant data the ARPA working group could find.

Another example involves the small CCIP-85 study contracts to gather quantitative software data. Since their completion, several local software designers and managers have expressed a marked interest in the data. Simply having a set of well-defined distributions of program and data module sizes is useful for designers of compilers and operating systems, and chronological distributions of software

errors are useful for software management perspective. Knuth's FORTRAN data, excerpted in Fig. 4B of the Software Quiz, have also attracted considerable designer interest.

Thus, if you're among the first to measure and disseminate your own software usage characteristics, you're more likely to get next-generation software that's more responsive to your needs. Also, in the process, there's a good chance that you'll pick up some additional clues which begin to help you produce software better and faster right away.

Appendix

A SOFTWARE QUIZ

Very little in the way of quantitative data has been collected about software. But there is some which deserves to be better known than it is. Because otherwise, we have nothing but our intuition to guide us in making critical decisions about software, and often our intuition can be quite fallible. The four questions below give you a chance to test how infallible your software intuition is. Answers follow each question and its related figure.

1A. Where Does the Software Effort Go?

If you're involved in planning, staffing, scheduling or integrating a large software effort, you should have a good idea of how much of the effort will be spent on Analysis and Design (after the functional specification for the system has been completed), on Coding and Auditing (including desk checking and software module unit testing), and on Checkout and Test. See how well you do in estimating the effort on a percentage basis for the three phases. The results for such different large systems as SAGE, OS/360, and the Gemini space shots have been strikingly similar.

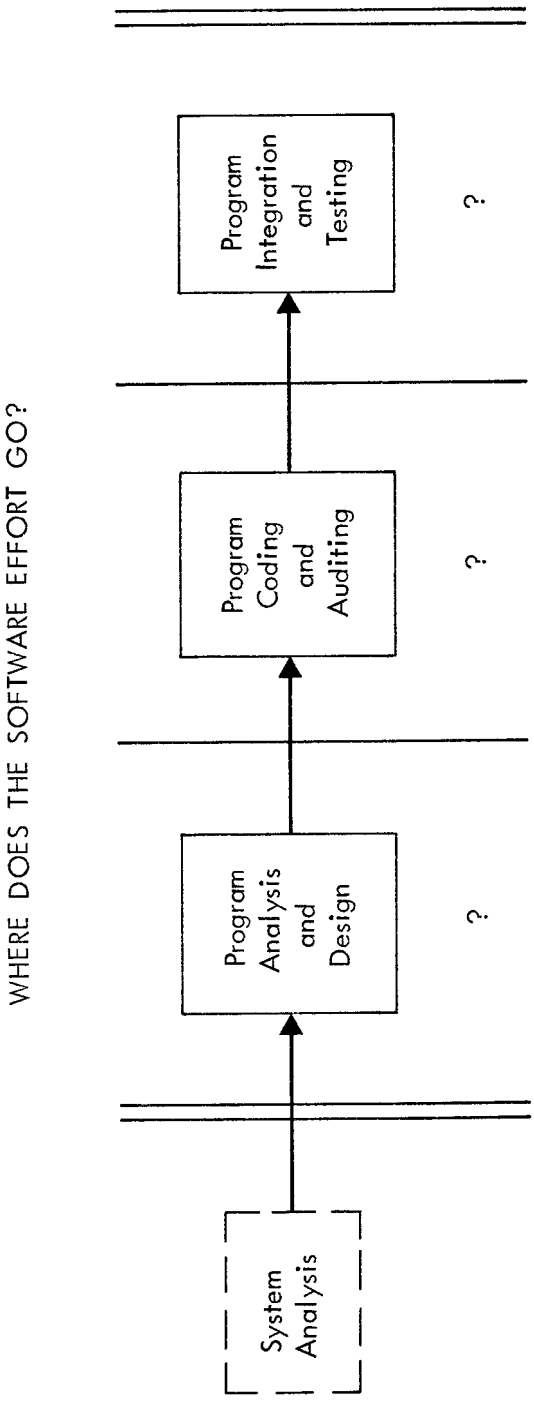
WHERE DOES THE SOFTWARE EFFORT GO?
System Analysis
Program Analysis and Design
Program Coding and Auditing
Program Integration and Testing
?
?
?
WHAT PERCENTAGE OF THE EFFORT IS SPENT ON EACH?

Fig. 1A

1B. Where Does the Software Effort Go?

How close did you come to that large 45-50 percent for checkout? Whatever you estimated, it was probably better than the planning done on one recent multimillion dollar, multiyear (non-defense) software project by a major software contractor which allowed two weeks for acceptance testing and six weeks for operational testing, preceded by a two man-month test plan effort. Fortunately, this project was scrapped in midstream before the testing inadequacies could show up. But similar schedules have been established for other projects, generally leading to expensive slippages in phasing over to new systems, and prematurely delivered, bug-ridden software.

Another major mismatch appears when you compare the relative amount of effort that goes into the three phases with the relative magnitude of R&D expenditures on techniques to improve effectiveness in each of the phases. Relatively little R&D support has been going toward improving software analysis, design, and validation capabilities.

The difference in the later TRW data probably reflects another insight: that more thorough analysis and design more than pays for itself in reduced testing costs.

(Refs.: Boehm, B. W., "Some Information Processing Implications of Air Force Space Missions: 1970-1980," *Astronautics and Aeronautics*, January 1971.

Wolverton, R., *The Cost of Developing Large-Scale Software*, TRW Paper, March 1972.)

WHERE DOES THE SOFTWARE EFFORT GO?

	Analysis and Design	Coding and Auditing	Checkout and Test
SAGE	39%	14%	47%
NTDS	30	20	50
GEMINI	36	17	47
SATURN V	32	24	44
OS/360	33	17	50
TRW Survey	46	20	34

Fig. 1B

2A. How Do Hardware Constraints Affect Software Productivity?

Another useful factor to know in planning software development is the extent to which hardware constraints affect software productivity. As you approach complete utilization of hardware speed and memory capacity, what happens to your software costs? Do they stay relatively constant or do they begin to bulge upward somewhat? The data here represent 34 software projects at North American Rockwell's Autonetics Division with some corroborative data points determined at Mitre.

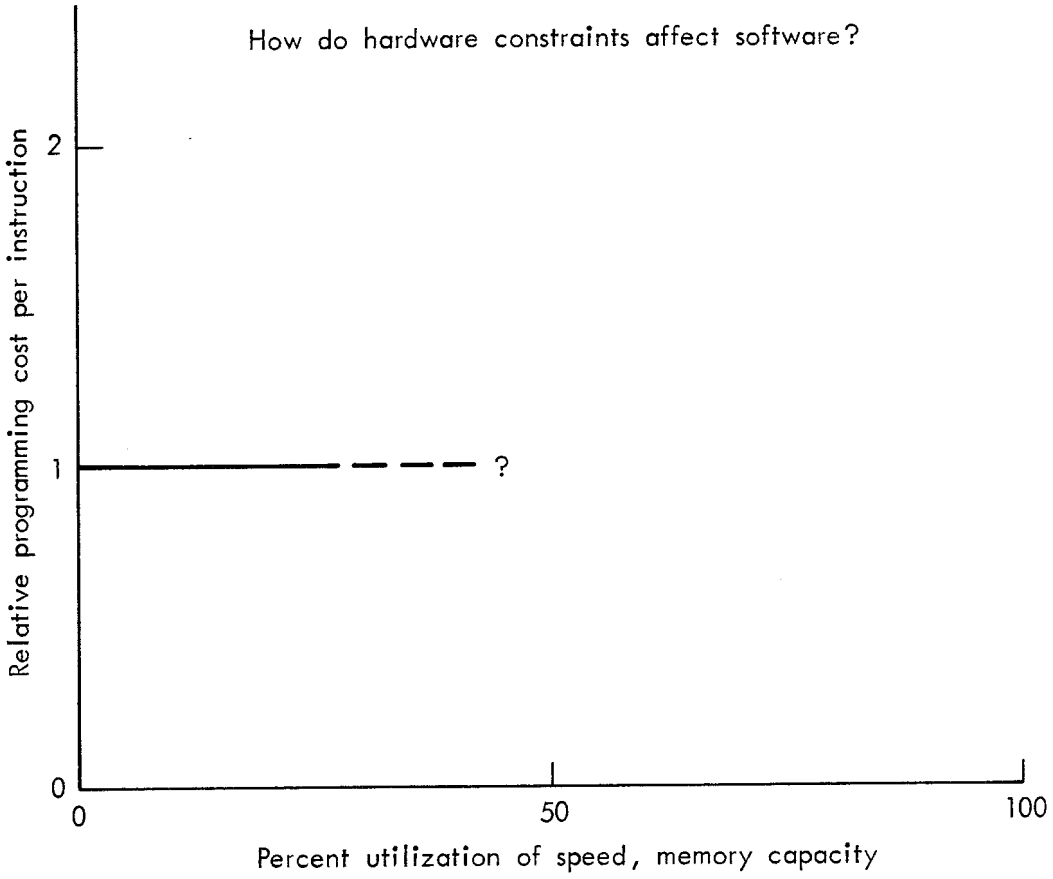
How do hardware constraints affect software?
Relative programming cost per instruction
2
1
0
?
0
50
100
Percent utilization of speed, memory capacity

Fig. 2A

2B. How Do Hardware Costs Affect Software?

Hopefully, your estimate was closer to the "experience" curve than the "folklore" one. Yet, particularly in hardware procurements, people make decisions as if the folklore curve were true. Typically, after a software job is sized, hardware is procured with only about 15 percent extra capability over that determined by the sizing, presenting the software developers with an 85 percent saturated machine just to begin with. How uneconomic this is will be explained by Fig. 11 in the text.

Those data also make an attractive case for virtual memory systems as ways to reduce software costs by eliminating memory constraints. However, the strength of this case is reduced to the extent that virtual memory system inefficiencies tighten speed constraints.

(Ref.: Williman, A. O., and C. O'Donnell, "Through the Central 'Multiprocessor' Avionics Enters the Computer Era," *Astronautics and Aeronautics*, July 1970.

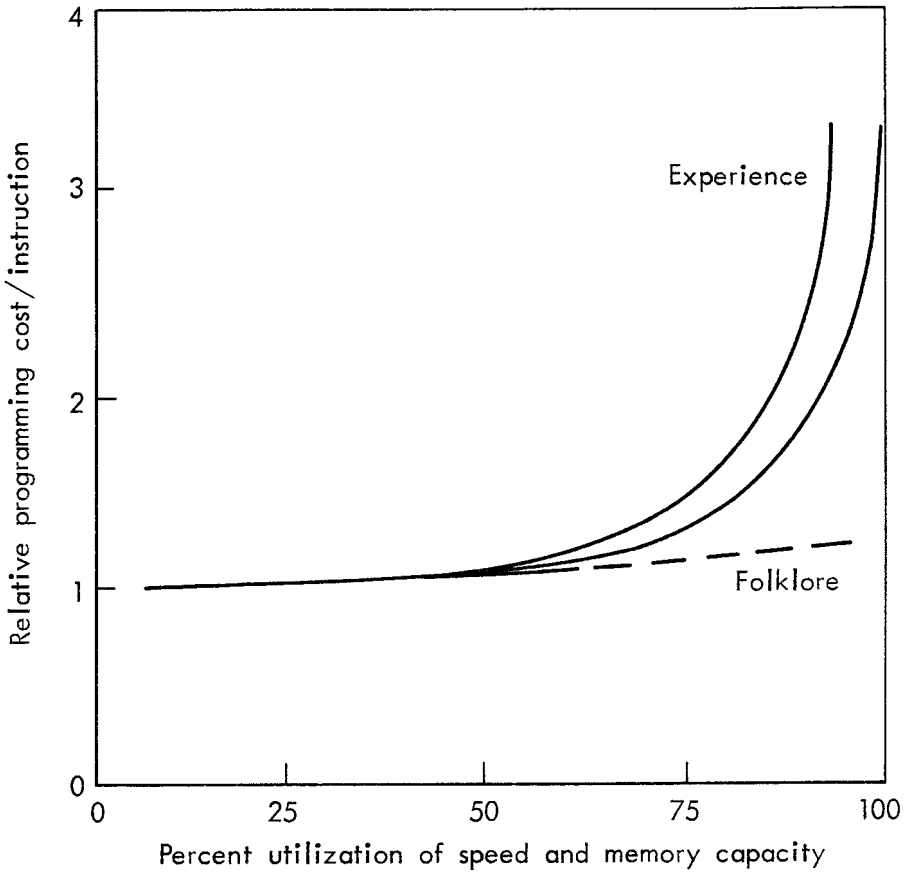

Fig. 2B — How do hardware costs affect software?

3A. Where are Software Errors Made?

If you're setting test plan schedules and priorities, designing diagnostic aids for compilers and operating systems, or contemplating new language features (e.g., GOTO-free) to eliminate sources of software errors, it would be very useful to know how such errors are distributed over the various software functions. See how well you do in estimating the distribution of errors for typical batch programs and for the final validation of a critical real-time program.

WHERE ARE SOFTWARE ERRORS MADE?
(What percentage in each category)

	Batch (all errors)	Real-time (final validation phase only)
Computation and assignment	?	?
Sequencing and control	?	?
Input-output	?	?
Declarations	?	?
Punctuation	?	Not available
Correction to errors	Not available	?
Total	100 %	100 %

Fig. 3A

3B. Where Are Software Errors Made?

Several points seem fairly clear from the data. One is that GOTO-free programming is not a panacea for software errors, as it will eliminate only some fraction of sequence and control errors. However, as Column 4 shows, the sequence and control errors are the most important ones to eliminate, as they currently tend to persist until the later, more difficult stages of validation on critical real-time programs. Another point is that language features can make a difference, as seen by comparing error sources and totals in PL/I with the other languages (FORTRAN, COBOL, and JOVIAL), although in this case an additional factor of less programmer familiarity with PL/I also influences the results.

(Refs.: Rubey, R.J., et al, *Comparative Evaluation of PL/I*, United States Air Force Report, ESD-TR-68-150, April 1968.

Rubey, R. J., *Study of Software Quantitative Aspects*, United States Air Force Report, CS-7150-R0840, October 1971.)

WHERE ARE SOFTWARE ERRORS MADE?

	7 batch programs (all errors)		Benchmark space booster control (all errors)	On-board space booster control (final validation phase only)
	PL/I	2 COBOL 2 JOVIAL 3 FORTRAN		
Computation and assignment	9 %	25 %	28 %	20 %
Sequencing and control	20	17	27	51
Input-output	8	8	7	6
Declarations	32	35	38	16
Punctuation	31	15	n. a.	n. a.
Corrections to errors	n. a.	n. a.	n. a.	7
Total (%)	100 %	100 %	100 %	100 %
Errors (No.)	214	140	313	87

Fig. 3B

4A. How Do Compilers Spend Their Time?

Recently, Donald Knuth and others at Stanford performed a study on the distribution of complexity of FORTRAN statements. Try to estimate what percentage of their sample of 250,000 FORTRAN statements were of the simple form A=B, how many had two operands on the right-hand side, etc. If you're a compiler designer, this should be very important, because it would tell you how to optimize your compiler--whether it should do simple things well or whether it should do complex things well. Here the results refer to aerospace application programs at Lockheed; however, a sample of Stanford student programs showed roughly similar results.

WHERE DO COMPILERS SPEND THEIR TIME?
(Knuth study: 440 Lockheed programs: 250,000 statements)

Number of operands		%
1	(A = B)	?
2	(A = B ⊕ C)	?
3	(A = B ⊕ C ⊕ D)	?
> 3		?

Fig. 4A

4B. How Do Compilers Spend Their Time?

It's evident from the data that most FORTRAN statements used in practice are quite simple in form. For example, 68 percent of these 250,000 statements were of the simple form A=B. When Knuth saw this and similar distributions on the dimensionality of arrays (58 percent unindexed, 30.5 percent with one index), the length of DO loops (39 percent with just one statement), and the nesting of DO loops (53.5 percent of depth 1, 23 percent of depth 2), here was his reaction:

> The author once found.....great significance in the fact that a certain complicated method was able to translate the statement
>
> C(I*N+J):=((A+X)*Y)+2.768((L-M)*(-K))/Z
>
> into only 19 machine instructions compared with the 21 instructions obtained by a previously published method....The fact that arithmetic expressions usually have an average length of only two operands, in practice, would have been a great shock to the author at that time.

Thus, evidence indicates that batch compilers generally do very simple things and one should really be optimizing batch compilers to do simple things. This could be similarly the case with compilers and interpreters for on-line systems; however, nobody has collected the data for those.

(Ref.: Knuth, D.E., "An Empirical Study of FORTRAN Programs," *Software Practice and Experience*, Vol. 1, 1971, p. 105.)

NUMBER OF OPERANDS IN FORTRAN STATEMENTS
(Knuth study: 440 Lockheed Programs, 250,000 statements)

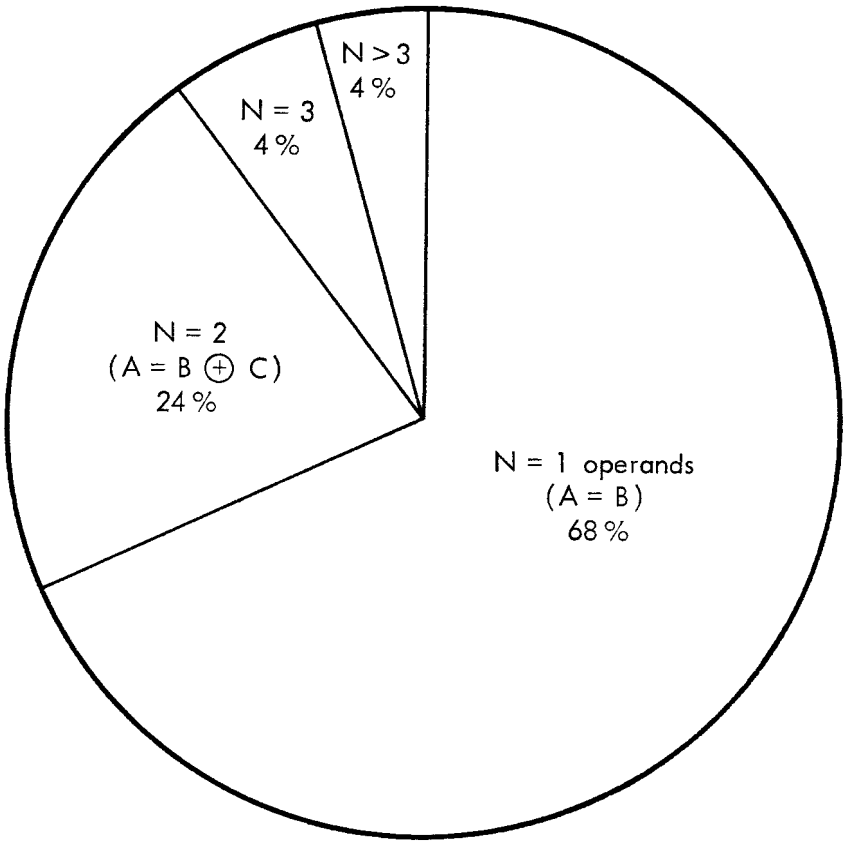

Fig. 4B

ACKNOWLEDGEMENTS

Hundreds of people provided useful inputs to CCIP-85 and this extension of it; I regret my inability to properly individualize and acknowledge their valuable contributions. Among those providing exceptionally valuable stimulation and information were Generals L. Paschall, K. Chapman, and R. Lukeman; Colonels G. Fernandez and R. Hansen; Lieutenant Colonel A. Haile, and Captain B. Engelbach of the United States Air Force; R. Rubey of Logicon; R. Wolverton and W. Hetrick of TRW; J. Aron of IBM; A. Williams of NAR/Autonetics; D. McGonagle of Anderson, Inc.; R. Hatter of Lulejian Associates; W. Ware of Rand; and B. Sine. Most valuable of all have been the never-ending discussions with John Farquhar and particularly Don Kosy of Rand.