NEW DIRECTIONS IN SOFTWARE 1960-1966

BY

ASCHER OPLER

Reprinted from the PROCEEDINGS OF THE IEEE
VOL. 54, NO. 12, DECEMBER, 1966
pp. 1757-1763
COPYRIGHT © 1966—THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.
PRINTED IN THE U.S.A.

New Directions in Software 1960–1966

ASCHER OPLER

Abstract—**This article reviews selected significant developments in the rise of software to an equal partner to computer hardware. Topics covered include transition problems, hardware–software interdependency, control programs, programming languages and their processors and certain data-file management aspects. The period October 1960 to April 1966 is covered.**

AT THE TIME when the first special computer issue of these PROCEEDINGS appeared (October 1953), the subjects to be reviewed in this article were given scant treatment. Only the paper of Hopper and Mauchly [1] appears relevant. When the second special issue appeared (January 1961), an article by Orchard–Hays [2] covered the "state-of-the-art" as of October 1960. Since that date, there has been a technological revolution resulting in the transformation of the position of software[1] from that of *a useful aid* to programming and operating computers to that of *an equal partner* to hardware in terms of importance in computer technology.

This article will *not* attempt to review and evaluate all developments in this field between October 1960 and April 1966. Instead, the current position of software will be summarized and four significant areas will be covered in some detail. This delimitation is necessary since the scope of work in this field is far beyond inclusion in a summary article. Indeed, since 1960, there have been at least ten major special conferences on aspects of software, the publication of more than one hundred papers and the appearance of at least a dozen books and monographs on software topics. In a report date 1963[3], 52.1 percent of a sample of 549 members of the Association for Computing Machinery listed software development as their primary activity. In 1966, in the United States alone, an estimated 5000 programmers are involved in the production of software.

The article will concentrate primarily on developments in the United States although activity reported in other countries represents a significant contribution. The International Federation for Information Processing has provided means for effective international communication via two major conferences (IFIP Congress—1962 in Munich and IFIP Congress—1965 in New York), a series of special conferences on programming languages and their processors and the work of IFIP committees.

Only general-purpose software for general-purpose computers will be covered. The development of computing systems for *dedicated* purposes (e.g., telecommunication, process control, reservations systems) has stimulated the development of software with the requisite attributes of reliability, response time, and channel capacity. While control programs for such applications have many features in common with control programs for computers used to handle a variety of computational tasks, discussion of their special problems cannot be included here. However, several other papers in this issue consider relevant aspects of such software. Furthermore, time-sharing software is covered elsewhere in this issue.

Manuscript received August 29, 1966; revised October 10, 1966.

The author is with Computer Usage Education, Inc., New York, N. Y.

[1] For this article, software is defined as the complete set of control programs, language processors and utility programs generally supplied to complement computing equipment (hardware). Specific application programs are not considered software for purposes of this article.

SOFTWARE PRODUCTION

The five and one-half year period covered has seen the development of a production process for software fabrication. In a talk presented early in this period, the author [4] pointed out that a software crisis was at hand because:

1) Quantity requirements were rising.
2) Product delivery was lagging.
3) Product quality was unsatisfactory.
4) Product costs were exceeding estimates.
5) The shortage of qualified designers and implementers was growing.
6) Proposed methods for automatic software production were not fulfilling expectations.

Today, software production remains a problem area. The major computer manufacturers have all found it necessary to abandon the practice of assigning the responsibility for producing software systems to conscientious programmers with the requisite technical competence. In re-evaluating the production problem, the management of each computer company has generally come to the same set of conclusions.

1) Production of hardware and software should be carried out under single unified responsibility. Software should no longer be regarded as "sales support" produced by the marketing division.

2) Production of software should be carried out at the same site as production of hardware.

3) Plant production technology should be applied to software. This calls for development of prototypes, specifications, quality control, schedule control, cost control, acceptance testing, etc.

4) Although this technology forms an effective production *framework*, qualified implementers constitute the *core*. To obtain the services of programmers with sufficient technical competence, large-scale recruitment and training should be undertaken.

5) Automatic software production technology should be developed and when perfected, put into operation.

This transition to factory-type production methods is still in progress. Completion is a few years away, but those manufacturers who have installed such methods appear satisfied with all results except costs, which have mounted beyond all projection exceeding, in many ways, hardware production costs. To date, automatic production of software has not been successful in turning out a product of commercially acceptable quality. This lack has been partially compensated for by the application of computers to many other processes involved in software production scheduling, specification, check-out, testing, documentation, etc.

Transition to Next-Generation Computers

The development, marketing, and installation of second-generation computers (transistors and printed circuits) was completed during this period and the introduction of third-generation (microelectronic circuitry) computers initiated. That problems exist in adapting applications programs for a next-generation computer has always been known, but it is only in the past two years that most people have become aware of a) the size of the investment of man-years in the existing pool of programs and, b) the difficulty of executing a smooth, minimum effort transition.

There are a number of known aids to transition between computer generations. For second-to-third generation conversion, the following are available:

1) Simulation of the old computer on the new computer.
2) All-hardware compatibility mode in which special circuitry permits the new computer to perform all functions of the old.
3) Emulation in which certain functions of simulating the old computer are performed by special circuitry while others are performed by software.
4) Recompilation of programs written in those programming languages for which processors are available on both the old and new computers. Where the two versions of the programming language have significant differences, language version to language version translators may be useful.
5) Data conversion aids which assist in reformatting data, changing character sets, labels, etc., as required by the new computer and its software.

The one method that is absent is that of fully automatic translation of any program (existing in either machine code or assembly language) to programs that can execute successfully on the new computer. While considerable effort has been spent in the development of such programs (8, 9, 10), the only translations in successful use today are special cases:

1) Translators operating between two computers with relatively small differences in logical design (e.g., Honeywell 200/IBM 1401).
2) Programs which translate the simple equivalences of a program and flag the difficult portions (11). These are sometimes called "reprogramming aids."

At the present time, the two most promising conversion routes are 1) via emulation and/or all-hardware compatibility, and 2) via use of standard programming languages. However, past difficulties are expediting the current movement toward the *mandatory* use of such languages. Concurrently, language standardization efforts, long in progress, are resulting in standardization at a most propitious time.

Hardware-Software Interdependency

For many years, despite claims to the contrary, there has been little significant interchange between logical designers and designers of software systems. To software system designers, it appeared that hardware designs were facts-of-life to be surmounted by their product. Computer designers blamed software designers for masking the beauty of the logical design with programming systems.

The direction that computer design has taken has forced an end to this isolation. The design of the Burroughs B-5000 (1961) was clearly the product of joint efforts. The computer aided the language processors by providing push-down storage, while the logical design necessitated a specialized control program in order to operate. This pioneering effort pointed out:

1) That software performance could be enhanced by inclusion of appropriate manipulative ability in the order set.
2) That software designers could build processors aimed at taking advantage of such facilities.
3) That an unconventional logical design made a special control program an *integral* part of the whole hardware-software system.
4) That the logical design forced the adaptation of a rigorous set of programming techniques and conventions.

Since that time, increasing interchange—even teamwork—has often occurred. Consolidation of the two developmental efforts under single management has facilitated this interchange. The knowledge that the two groups could really support and encourage radical departures led to a new and better mutual appreciation. Based on the logical equivalence of hardware and software, careful and economic trade-offs have begun to appear. For example, if the cost of main core storage can be drastically reduced, large complex control systems may reside in memory replacing expensive complex circuitry, provided that the control functions can have fast enough response. This illustrates the type of interaction that requires considerable joint analysis to reach proper decisions.

Third-generation computers generally have less control circuitry than earlier computers because of greater reliance on control programs. The recognition of the importance of control programs is borne out in the design of a two-mode computer which prevents "ordinary" programmers from altering control programs stored in memory. In the event

of attempted alteration, an interruption occurs. This leads to a three-level computer viewpoint:

1) Hardware (Hardware Designer)
2) Control Program (Software Designer)
3) User Programs (Applications Programmer).

It is evident that those who utilize any specific level must not be permitted to alter higher level facilities.

Effect of Microprogramming

The implementation of a computer design may be done in (at least) two ways. In the conventional, the logical design is realized in electronic circuitry. In the other, a highly simplified processor is designed and then the full design converted to a step-by-step microprogram which effects execution of each instruction in the order set by a series of microprogrammed instructions.

The latter approach has been known (and occasionally implemented) for many years. The practical difficulties lay a) in the high cost of the micro-memory which stores the microinstructions and b) in the relative slowness of the microprogram (8 to 30 microsteps are generally needed to carry out a single command). The principal advantages lay in a) reduced production cost and b) flexibility of treating the order set.

When microminiaturized circuitry was developed for third-generation computers, operation times of 10–100 nanoseconds were made readily available, but large main memory access time remained in the order of 500–2000 nanoseconds. Designers took a new look at microprogramming and realized that the time required for a storage access would generally overlap the time for 8 to 30 microsteps. Promising new techniques were available for preparing high speed *read-only memories* which could store the microprograms. With these developments, commercially successful microprogrammed computers (IBM System/360 Models 30, 40, 50, 65, 67; Spectra 70, etc.) were designed and produced.

Once the production of microprogrammed computers was commenced, a further area of hardware-software interaction was opened via microprogramming. For example, more than one set of microprograms can be supplied with one computer. A second set might provide for execution of the order set of a different computer—perhaps one of the second generation. Additional microprogram sets might take over certain functions of software systems as simulators, compilers and control programs. Provided that the microsteps remain a small fraction of a main memory access cycle, microprogramming is certain to influence future software design.

Storage Allocation

Perhaps the one area that most reflects current thinking about programming is that of storage allocation. Some years ago, the principal problems concerned relocation of separately assembled programs to form a single executable unit. Emphasis then shifted to problems of overlay in which specified segments stored in an external medium were loaded as explicitly directed, replacing portions of code no longer needed. The next storage allocation problem of concern was that of dynamic relocation in which execution of certain sections of code was momentarily halted, the code shifted to a new position in core storage and then execution allowed to resume. Dynamic relocation proved useful in processing large complex programs on relatively small computers and in multiprogramming situations. Another storage allocation problem of general interest centered on the automatic assignment of sections of code to internal and to external memory and the development of control programs to handle required overlay automatically.

The subject of storage allocation is one that continues to be vigorously developed. Acceptance of time-sharing, multiprogramming, and multiprocessing using a single shared memory has forced an accelerated development of storage allocation technology.

Central to the new technology is a change which can be best grasped by considering storage from the viewpoint of a single user. Instead of *real* memory (limited to magnetic core memory), he addresses a *virtual* memory (exceeding main memory size by orders of magnitude). Consider an oversimplified example: if there are 1000 words of main memory available for the user and he addressed location 13 500, the thirteenth "page" of 1000 words is automatically fetched from external storage, after preserving the current contents of the 1000 words, prior to the accessing of location 500 in *that* page now in main memory. Thus a user has virtual access to large memory without concerning himself with relocation, overlay, etc.

Such systems for "swapping" "pages" of external and internal memory are generally slow (several milliseconds per swap) and complex in their control program design. Considerable attention is being given to a) strategies of swapping which minimize the number of swaps required and b) hardware and software schemes to simplify virtual memory control programs.

Organizationally, virtual memories operate with address-free code stored in external media. Reference tables, stored in main memory, are consulted whenever a storage access is required. A reference to a page not currently in main memory triggers a swap, an update of the memory reference table and a binding of the page to an available main memory area.

In addition to the development of the virtual memory concept, new techniques for sharing common coding have also been developed. The common coding may be an interrupt-resolution routine, a square-root subroutine or a Fortran compiler. In order to be used in a nondestructive way by a number of simultaneous or near-simultaneous users, these common routines must be written in *re-entrant* code (also called *pure procedure*). Such code is written with only invariant instructions and constant data. Code external to the routine is addressed only by indirect methods. All variables, parameters, and work areas are located in

private user space or in common push-down storage. There are as many copies of these elements as there are users.

Software in Computer System Control

A supervisory control program is now an integral part of every third-generation computer. In effect, the hardware requires the control program to enable the computational functions to be carried out.

To illustrate, consider how multiprogramming is implemented via *software* in distinction to implementation in *hardware* (as in the Honeywell-800). The control program includes a number of functional routines to permit multiprogramming. One regulates a circulating queue of actively executing programs, another dispatches input/output tasks as channels and equipment becomes available. The multiprogramming capability is built into the control program as follows: Whenever any interrupt occurs (if none does occur "naturally," an interrupt is forced every few milliseconds via an internal clock), the currently executing program is examined to ascertain whether it is in a *hold* condition pending availability of an I/O device or pending completion of a previously initiated task. If the program is in a *hold* condition, others in the circulating queue are examined to locate the next program not in a *hold* condition. That one is selected to replace the previous (single) active program and execution commences at the point where it was interrupted during its previous running. In this way, the multiprogramming monitor allows each of a group of programs to continue executing while the others wait for channel or device availability.

Another area in which functions formerly assigned to hardware are now handled by software is that of input/output control. When computers interfaced a very small number of prespecified devices, it was reasonable to design logic for reading and writing each device. In the past few years, the number of available devices has multiplied and the rate of new device introduction has exceeded the rate of new computer generation introduction. As a consequence, designers of the input/output portion of a new computer face a *nearly impossible* task—to design a computer capable of interfacing a hundred or more different devices—many not yet developed. The answer clearly lies in the software realm. New computers have a *minimum* of I/O hardware—limited to generalized channel and control unit logic. All of the sophistication and the specialization for different devices is built into the software. One consequence of this trade-off is the requirement for large amounts of main storage to hold all the required programming.

Current software for input/output processing is, in part, an integral portion of the supervisory control program. One portion must monitor the interrupt system to identify interrupts originating outside the central processor or to initiate an I/O interrupt when appropriate. Another portion of the I/O system must establish and maintain a queue of pending I/O requests and must dispatch or stack them according to availability or nonavailability of equipment. The part that analyzes error indications must be able to isolate both the faulty device and the specific fault and then it must branch to an appropriate recovery routine specialized for that device and that error. Other parts of a third-generation I/O system include specialized code for recognizing and treating differences between specific devices, channel and device availability tables, assignment tables and corresponding allocation and assignment routines.

Multiprogramming and input/output control represent but two of the ways in which hardware functions are now being relegated to software. Other obvious areas are time-sharing control, telecommunication, and graphic-display systems.

In addition to functions previously assigned to hardware, control programs have taken on many new functions. Some are refinements of older practices; others are required because of the manner in which we currently view the operational environment. This can perhaps be understood in terms of the two following paragraphs.

In *1960* the computer was viewed as a single entity of considerable capability. Computational tasks at an installation were generally homogeneous (scientific, commercial, etc.) and embodied in single programs. A queue of these programs was organized and processed sequentially by the computer.

In *1966*, the computer is viewed as a collection of resources (CPUs, memory boxes, input/output devices, etc.). Computational tasks at an installation are generally heterogeneous and embodied in a myriad of program modules. A list of all required computational tasks, along with corresponding priorities and needed resources, is fed to the computer. Tasks are analyzed, schedules established and processing commences. According to the supply of useful resources, as many programs (or as many parts of programs) as possible are run concurrently. As tasks end and before new tasks commence, resources are re-allocated to maximize computational work per unit time.

To implement the self-scheduling, self-resource-allocating computer, the control program must contain schedule-planners, schedule-dispatchers resource-control routines, numerous queue-control programs, priority-conflict resolution programs, recording and accounting routines, etc.

These functions can be supplied only at the expense of main memory space. (Ultra-compact systems using overlays from fast magnetic drum or disk have been implemented to save core space at the expense of either response time, available facilities or both.) As the number of devices increases, as the number of programs that can run in parallel increases, so the main memory space allocated to control programs increases more and more. In 1960, medium-size computer sizes were typically 8192 words or 20 000 characters. Today, a typical medium-size computer has 4 to 8 times this capacity, with control programs often occupying 10 000 or more words.

Programming Languages

The history of the development of programming languages during the covered five and one-half years could be the subject of a lengthy monograph. Indeed, concise histories of some of the major languages have appeared [5]. A number of conferences dealing with aspects of programming languages have been held and programming languages committees are to be found in numerous national and international professional societies. Furthermore, in

recent years, a move toward standardization of several languages has resulted in the development of standards for Fortran and Cobol. In the broad scope of this article, only a few generalities can be stated about programming languages; readers with special interest can refer to the voluminous literature. Remarks about processors (compilers) for programming languages will be made in the next section.

Currently, these languages have collectively reached a pinnacle of acceptance. Three of the languages (Algol, Fortran, and Cobol) are most widely used because good processors are available for most computers and the languages themselves are proving useful. Dialects and variants of these languages proliferated during the last five and one-half years, but the introduction of new broad-use languages has diminished. Thus the field seems to be moving toward the establishment of a small number of stable standard languages.

A new programming language of unusually broad scope (Programming Language/I) has been under development by IBM for two years, but no processor has been made available to date. The language is designed to be coalescent with respect to previously developed scientific, commercial, and other programming languages. IBM's heavy backing will undoubtedly make this language an influential one. Many other manufacturers are currently planning also to provide PL/I compilers.

Considerable attention has been given to special-purpose languages. Such languages fall into two categories—those whose semantics require special treatment and those whose syntax and semantics both require special treatment. In the former category are application-oriented languages which use terminology specific to a discipline—thus there are trajectory languages, telecommunication languages, natural language-manipulation languages, picture languages, military languages, algebraic-manipulation languages, simulation languages, etc. Into the second category fall languages with unusual syntactic structure. These include languages for list processing (e.g., LISP, IPL-V,) and languages which are self-extending (e.g., TOOL, SET, SLANG, and X-POP).

Another direction of language development is toward implicit or descriptive languages. In the programming languages mentioned above, the writer uses language in an explicit, prescriptive manner stating the actions to be done and the performance sequence. In an implicit language, the writer describes the initial status and the desired final status of his computation. A processor then develops a program that will meet the requirements. At present, report writing, graphic and input/output languages of this type have been developed. File structure problems are also subject to treatment using such implicit languages.

Processors for Programming Languages

While the publications dealing with programming languages have been voluminous, the areas associated with the compilers that process them have received relatively less attention. There exist a number of publications [6] setting out, in detail, how one processor for a given language was developed for a given computer. In general, such publications appeared several years after the completion of the project and describe useful techniques. While such volumes make interesting reading, the problems relate to other specific designs in only a general way.

During the five and one-half years covered, a vast number of significant contributions to compiler design and to implementation technique have been made but, because of a competitive situation and because of enormous pressure to produce successors, publication of these methods has been limited.

The most influential contribution was that of Irons (7)—namely *syntax direction*. This concept served to relate language specification to processor specification. Iron's original syntax-directed compiler proved impractical, but efficient modifications and variants were soon developed. Although many significant contributions to fundamental design and technique were published, a meaningful gap still exists between processor techniques described in the current literature and processor techniques in use today.

The development of an efficient processor has rarely, if ever, been produced by the application of a *single* design principle. Several single-design processors have been produced but have not been commercially successful.

Most compilers produced today are syntax-oriented in their language decomposition phase and use recursive routines for machine code generation. With usual memory space limitations, most designers must choose between a fast "one-pass" compiler with minimum (primarily local) output code optimization or a slower "multi-pass" compiler with cascading series of global code-optimizing routines. Of course, processors which may operate in large magnetic core memories, can optimize globally in a single pass.

Compilers can be optimized for certain performance requirements only at the expense of others. In the last few years, we have seen very compact processors, very fast ones, very large ones, etc. Designers are now accepting the fact that design objectives must be specialized. The general compiler pleasing all users does not and cannot exist.

Time-sharing and multiprogramming requirements have turned attention to two new types of processors—conversational and re-entrant compilers. The former allows a time-sharing programmer stationed at an attached terminal to develop his program in a conversational style: (1) a statement is typed by the user; (2) it is checked and either accepted or diagnosed as an error; (3) in the latter event, the user, having been notified, may correct it on the spot, and (4) in any event, he prepares the next statement. When the process has been completed, the user directs the system to compile his program and store it for future use. An alternative approach permits *fractional* compilation of each statement as it is accepted, with subsequent binding of the fractions to form a single executable whole.

A re-entrant compiler is one constructed entirely with re-entrant code. For each active concurrent user, a separate

storage area is established for input, work, diagnostics, output. The actual compiler coding, of course, remains invariant. Construction of re-entrant compilers has required the development of new techniques—but many design principles of ordinary processors have been used.

Production of Programming Language Processors

Probably more than one hundred full-scale compilers have been produced in the United States alone during the covered period. Each year, an increasing number are scheduled and announced. Nevertheless, difficulties continue to plague compiler producers.

In general, difficulties may be attributed to failure to:

1) Realize that the task (compiler production) is a complex one requiring control by capable managers.

2) Comprehend the need for trade-offs between a) implementation time, b) implementation man-power, c) compiler performance, d) object program performance, and e) quality of product.

3) Comprehend the systems-integration aspects since compilers are only one element in an ensemble of software elements.

4) Allow sufficient time, manpower and computer time to bring the just completed product to an adequate level of error-free operation.

5) Allocate sufficient computer time for the various phases of compiler production.

6) Use experienced producers of successful compilers to produce subsequent ones. (Compiler techniques are "more 'art' than science" and new implementers must re-discover many of the basic techniques).

7) Separate the concepts of objectives, specification, design, implementation, and evaluation.

8) Resolve the conflict between the choice of a well-known design for a new compiler (with possible loss of efficiency) and the choice of developing a new design (with probable delay in completion) for a new situation.

9) Distrust techniques believed to simplify the task: writing a compiler in its own language, writing in other high-level languages, writing using special compiler-production languages, etc. Although the principle is valid, few compilers produced using these methods have been as generally successful in the eyes of users as those written in assembly language.

Major producers of programming language processors have become aware of the scope and complexity of the technological problems. Currently, more realistic estimates of calendar time, manpower and computer time are being made. Major producers of software (computer manufacturers and software companies specializing in this field) are devoting more attention to software problems; management of software companies is becoming more realistic and tough-minded. In this environment, objectives are established, objectives lead to specifications; designs to meet specifications are evaluated, quality and schedule assurance are applied and the final product is rigorously tested both for reliability and conformity to the specifications.

A host of shortcut methods for expediting software production have been developed. Most have failed to provide the capability necessary to meet performance and other objectives.

This rigorous atmosphere dominates only the realm of computer manufacturers and independent software organizations. At the universities and other research-sponsored centers, vigorous research activity continues in selected software areas. While many significant contributions have originated at such centers, many researchers have failed to grasp the nature of the real software problems.

Data-File Management Software

Although thousands of small programs are being written to perform a single task operating on a small amount of data, the trend toward large integrated groups of programs operating on an integrated data base is quite evident. Furthermore, computation is becoming more and more *distributed* with increasing use of multiple computers, telecommunication network and remote access stations. One of the software facilities designed to support large, integrated applications operating on distributed equipment is data-file management software.

In 1960, the dominant data storage medium was magnetic tape, although rotating magnetic disk systems were in some use and small, slow magnetic drums were beginning to fade from use. Today, while magnetic tape usage has greatly increased, it is declining *relative* to the use of four other devices: fast, high-capacity, magnetic drums; small, demountable magnetic disk units; fixed magnetic disk units; and very high-capacity, moveable, individual magnetic strip, transport units. All of these can access any of a large number of data records in a much shorter average access time than magnetic tape units can.

This capability has given rise to considerable development in techniques (12, 13) to allow managing and efficient accessing of data files stored on such devices. Such systems allow the use of a variety of different access methods, maintenance, and file security methods. These systems often extend their capabilities to files originally described in COBOL or other high-level languages. Data-handling systems of this type rank among the largest and most complex software projects. Principal difficulties are concerned with the sheer size of the data control program routines, and the difficulty of handling small, simple data records when adherence to the rules of the large dominant system are mandatory. The file-access software must cope with a number of problem areas including:

1) Allowing sufficient "device independence" such that drums, stationary or demountable disks and magnetic strip device may be manipulated in a manner that appears identical in effect to the user.

2) Allowing manipulation of files or records within a file or items within a record by a variety of general techniques including addressing by name, by an index code, and by position.

3) Obtaining the optimum redundancy—omitting needless duplication on the one hand, but providing sufficient copying facility to insure ability to preserve files to prevent vital loss in the event of malfunction.

4) Allowing location of records based on the content of an item.

5) Allowing indication of interrelations in complex data bases.

6) Providing the capability of sorting files while resident on the device.

Summary

Because of the level of activity in this field which has burgeoned in the past five and one-half years, it has been possible to give only a cursory summary of some of the more significant developments. In summary, the covered period can be characterized by:

1) Increasing software requirements under pressure of user demand and strong hardware dependence on software.

2) Failure to solve production problems despite major emphasis on production and quality control.

3) Acceptance of the use of programming languages.

4) Attempts to control proliferation of different languages.

5) User acceptance of the dominance of large (software) supervisory systems.

6) Realization of the huge cost of transition to next-generation computers.

7) Unsuccessful attempts to develop completely satisfactory general techniques for automatic software production and general techniques for automatic translation of programs from one computer to another.

In the period that lies ahead, in addition to continued efforts to solve problems mentioned in 2) and 7) above, new items can be expected to preoccupy software specialists including:

1) Allocation and linkage problems associated with time-sharing, multiprogramming, etc.

2) Problems of fractional compiling, conversational computing, and similar man-machine interactive techniques.

3) Problems of simplifying, managing, and maintaining programming systems of enormous size and complexity.

4) Extension of microprogramming to supervisors, compilers, etc.

5) Software for visual, auditory, and other human manipulative interfaces with computing systems.

6) Increased attention to the contributions being made in more fundamental fields such as automata theory, linguistics, information science, etc.

Bibliography

[1] G. M. Hopper and J. W. Mauchly, "Influence of programming techniques on the design of computers," *Proc. IRE*, vol. 41, pp. 1250–1253, October 1961.

[2] W. Orchard-Hays, "The evolution of programming systems," *Proc. IRE* vol. 49, pp. 283–295, January 1961.

[3] F. Coss, "A profile of the programmer," *Commun. ACM*, vol. 6, pp. 592–594, 1963.

[4] A. Opler, "Current problems in automatic programming," in *Proc. WJCC*, pp. 365–370, May 1961.

[5] "Toward better documentation of programming languages," *Commun. of the ACM*, vol. 6, pp. 76–93, 1963.

[6] M. Halstead, *Machine Independent Computer Programming*. Washington, D. C.: Spartán Books, 1962.
B. Randall and L. J. Russell, *Algol 60 Implementation*. London: Academic Press, 1964.
A. P. Ershov, *Alpha Automatic Programming System* (in Russian). Novosibersk, Russia: USSR Academy of Science, Siberian Div., 1965.

[7] E. T. Irons, "A syntax directed compiler for Algol 60," *Commun. ACM*, vol. 4, pp. 51–54, 1961.

[8] J. H. Gunn, "Problems in program interchangeability," in *Symbolic Languages in Data Processing*. New York: Gordon and Beach, 1962, pp. 777–790.

[9] A. Opler, D. Farbman, M. Heit, W. King, E. O'Connor, R. Goldfinger, H. Landow, J. Ogle, and D. Slesinger, "Automatic translation of programs from one computer to another," in *Information Processing 1962*. Amsterdam: North-Holland, 1962, pp. 550–553.

[10] H. Oswald, "Automatic machine language program translation," Rome Air Development Center, Griffiss AFB, N. Y., Tech. Rept. RADC-TR-65-95, May 1965.

[11] D. M. Wilson and D. J. Moss, "CAT:A 7090-3600 computer aided translator," *Commun. ACM*, vol. 8, pp. 777–781, 1965.

[12] C. W. Bachman and S. B. Williams, "The integrated data store," in the *1964 Proc. FJCC*, vol. 26, pt. 1, pp. 411–422, November 1964.

[13] W. A. Clark, "The functional structure of OS/360: III—Data Management," *IBM Sys. J.*, vol. 5, pp. 30–51, 1966.