LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-114

RESEARCH DIRECTIONS IN COMPUTER ARCHITECTURE

Jack B. Dennis
Samuel H. Fuller
William B. Ackerman
Richard J. Swan
Kung-Song Weng

September 1978

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

MIT/LCS/TM-114

RESEARCH DIRECTIONS IN COMPUTER ARCHITECTURE

Jack B. Dennis
Samuel H. Fuller
William B. Ackerman
Richard J. Swan
Kung-Song Weng

September 1978

MIT/LCS/TM-114

RESEARCH DIRECTIONS IN COMPUTER ARCHITECTURE

Jack B. Dennis
Samuel H. Fuller
William B. Ackerman
Richard J. Swan
Kung-Song Weng

(To be published in The Impact of Research on Software Technology. P. Wegner, Ed., MIT Press.)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

12. RESEARCH DIRECTIONS IN COMPUTER ARCHITECTURE[1]

J. B. Dennis
Massachusetts Institute of Technology
S. H. Fuller
Carnegie-Mellon University
W. B. Ackerman
Massachusetts Institute of Technology
R. J. Swan
Carnegie-Mellon University
K.-S. Weng
Massachusetts Institute of Technology

1. Introduction

The "architecture" of a computer system defines the interface that the hardware presents to the software of the system, and determines how this interface is realized by subunits of the computer system. In the early days of the stored program computer, when the simple "von Neumann" form of main memory and "arithmetic logic unit" was unquestioned, knowledge of logic design, the technology of logic and memory devices, elementary machine language programming techniques, and a good measure of common sense were all that was required to be a computer architect. Now, things have changed. Our concept of what we expect a computer system to do for us has reached a high degree of sophistication -- large data bases, multiple concurrent processes, and programming languages that offer recursive programming, abstract data types, protection, and access control. These expectations have been met by elaborate software systems -- operating systems, data management systems, and runtime support for language implementations. The ability of system designers to meet these expectations, and the quality of the facilities they can provide to the application programmer, are critically dependent on properties of mechanisms built into the hardware. Thus it has become essential that contemporary computer architects be aware of how architectural decisions interact with software quality -- how hardware structures can more effectively meet the needs of operating systems and modern concepts of program and data structure.

---

1. This research was supported by the National Science Foundation under contract MCS75-04060 A01, and by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661.

Key Words: computer architecture, computer science research, distributed computer systems, multiprocessor computer systems. supercomputers, computer networks, personal computers

Many developments, such as pipelined processors and caches, and most advances in component technology have improved cost-performance while the architecture seen by the programmer has been little affected. The reduced hardware costs, particularly for memory, have reduced pressures for the utmost compactness and efficiency in programs. This has greatly fostered the use of high level languages without super-optimizing compilers, and has allowed use of relatively inefficient, interactive, interpretive language systems.

The introduction of automatic migration of code and data between levels in the memory hierarchy (originally in the Atlas computer [63]) has greatly aided software development by effectively masking constraints imposed by small primary memories, and eliminates the need for elaborate program overlay mechanisms. Similarly, the introduction of interrupts has allowed programming of real-time tasks without introducing program constraints to ensure regular device polling.

Not all architectural innovations have been of benefit to the programmer. In many cases there has been little concern for how the new features would affect the ease with which correct software may be prepared. In fact, as we shall see, many recent advances in computer architecture have presented the programmer with new and difficult challenges rather than making his task easier. The introduction of large, baroque instruction sets -- while presented as aids to programmers -- may have had an overall negative software impact. The use of high level languages may be inhibited because it is difficult to utilize "special architectural features" from within a standard language.

Radical departures from traditional "von Neumann" architectures have almost always led to severe software problems. We will see below that innovative attempts to produce very high performance systems -- such as multiprocessors and vector and array machines -- have presented the programmer with new and difficult challenges.

With the plummeting cost of hardware and concern for the enormous difficulty of building large software systems, it should be possible somehow to apply this tremendous technological capability to the advantage of the programmer. Computer designers should concentrate their efforts on architectural advances having a positive impact on the software problem. However, this has not always been the direction of past efforts. In commercial computers, the principal response to software requirements has been the development of compatible series of computer models that permit the user to move to more powerful or technologically advanced equipment and be able to run old programs without modification. Although certainly a very significant development, compatible lines of computers do not at all simplify the construction of new software. In fact, innovation in the direction of better support for

programming is inhibited by the conservative attitude engendered by the huge inventory of operational software.

How can computer architecture contribute to lessening the software problem? To answer, we must understand where the difficulties arise in current practice. Probably the most important factor is whether or not a program can be expressed effectively in a high level language. Essentially all research on understanding the software problem from the viewpoint of program structure, specification and correctness assumes programs are written in a completely specified high level language -- usually one in which type correctness may be checked by the translator. Only for those parts of a program that may be expressed within the strict limits of definition of a high level language that supports a suitable programming methodology can the benefits of this research be realized. Yet in large application programs, programmers nearly always find it necessary to step outside the facilities provided by a high level language. Why?

One major reason is to manage the transfer of information among the physical media of computer memory systems. When these transfers must be directed by the programmer, the overall behavior of a large program becomes very difficult to comprehend. This holds even when such transfers are directed using the facilities of a high level language as in Cobol. A second major reason concerns the coordination of concurrent activities, which is supported by *ad hoc* mechanisms built into complex operating systems. The nature of these coordination mechanisms and their effect on program structure is very dependent on properties of the underlying hardware. Finally, a significant portion of the code in many serious computer applications concerns actions to be taken in response to failures of hardware components. Having the right fault tolerance capability built into the hardware could contribute much to the simplification of software.

Subsequent sections examine recent developments in the several areas of computer architecture that have the most significant interaction with software. In Section 2 we remind ourselves of the amazing magnitude of the advance in hardware technology and the possibilities this suggests.

The next two sections review major areas of innovation in computer system organization: hardware support for referencing and controlling access to programs and data, and schemes for exploiting concurrency for performance and for meeting software needs. Many of the new concepts advanced in these areas may be interpreted as introducing mechanisms into computer systems in support of improved ways of structuring large programs.

Instead of developing mechanisms with the hope that they will turn out to provide the right basis for software, it is possible to start from a sound programming methodology (including a language specification) and develop the hardware/software system to support it. This leads to the concept of language based architecture reviewed in Section 5.

Fault tolerance is a very important aspect of computer systems in most applications and is another place in which hardware and software considerations interact strongly. Work in this area is discussed in Section 6. Since correct operation of programs depends on the correctness of the hardware on which they run, ways of ensuring correctness of hardware function is an important but largely neglected area of architectural research. This area is discussed in Section 7.

For completeness, we have included some comments about research issues in computer networks and personal computers -- new and important phenomena in the field of computer systems. We conclude with a summary of the most significant architectural research directions toward improved software support.

2. Technology

Technological advance over the past two decades has been dramatic in reliability, in cost-performance improvement, and in the alternative technologies open to the designer. Semiconductor device technology has doubled the number of active elements per integrated circuit device every one or two years since 1959 [51], and this advance is expected to continue well into the 1980's. One can now buy random access memory with 16K bits per LSI device, and a million bits per device seems possible in the future. We now have microprocessors that embody on one semiconductor chip the complexity of logic that required a roomful of electronics just twenty-five years ago.

The development of mass storage technology has been almost equally impressive. Magnetic storage technology has achieved three orders of magnitude improvement in information density over twenty years [42]. The magnetic bubble and charge coupled device (CCD) technologies provide new cost-effective alternatives in the performance range between the semiconductor and magnetic recording technologies. Research in magnetic, electron beam and laser technology hold the promise of inexpensive and reliable mass memory systems in the future.

All this holds the tantalizing prospect of cheap computing: all the computing power and services of a large contemporary computer system available at your desk; the Library of Congress in your office. The difficulty is, of course, that no one knows how to put this revolutionary technology to work so as to turn our dreams into practical realities. The limitation is not our ability to build hardware, but the problems of building the software systems that deliver computing power and services to the user. The challenge to the computer architect is to exploit this continuing revolution in technology to the benefit of the software designer. It seems that the cost-performance promise of the new technology cannot be fully exploited without a gross departure of computer systems from their classical form. Yet it is not clear how this change can be made without giving the programmer more difficult problems to solve instead of supporting the simpler programming methodologies he would dearly like to have.

3. Memory Structures, Addressing, and Protection

The problem of effectively utilizing memory devices has always been a challenging one. The simple connection of a computer processing unit to a fixed-size random access memory, with each instruction address field giving the full address of a unique word or byte of the memory, has several shortcomings: First, high speed random access memory is expensive; it is not usually feasible to have as much of it as one needs for all data associated with an application. This is true even though memory prices are dropping faster than processor prices, and larger amounts of high speed memory are becoming feasible. Second, even if such a large memory were practical, the instructions would need to have excessively long address fields to directly access the memory. This is the "address space problem". Third, a homogeneous block of directly addressable memory is not the best interface to present to the programmer. It is better for the memory, as seen by the programmer, to have some logical structure. Such structuring serves several purposes: If the form of the memory space reflects the way data is organized in programming languages, the memory system will provide cleaner support for these languages. Furthermore, structuring can support such features as protection and controlled sharing of data.

3.1. Virtual Memory

The usual approach to the solution of the memory space and addressing problems is by methods for transforming the addresses generated by computer instructions to the addresses of locations in the physical memory media. Any such method could be called a "virtual memory" scheme [26], but the term "virtual memory" is commonly taken to imply use of a multi-level memory system.

Such a virtual memory system increases the effective speed of a mass memory device through the use of a faster "buffer" memory that holds the most active data items.

The motivation for multi-level memory systems stems from the enormous range of memory performances and prices. This range has existed since the earliest computers and, despite enormous technological improvements in the past and projected for the future, is likely to persist. Current popular online memory devices range from ECL RAM, with access time of about 10^{-8} sec and cost of about 10¢/bit, to magnetic disks with access time of about .05 sec and cost of about .001¢/bit.

The virtual memory techniques that deal with the memory space and addressing problems provide solutions to the protection problem as well. The operating system needs to be protected from user program errors, and it is useful for program modules to be protected from errors in other modules of the same program or other concurrently running programs. Such a protection mechanism makes it possible to isolate faults and to prevent unplanned interactions between modules. This is important in large software systems because it facilitates testing and maintenance.

The earliest programming techniques for running programs too large for "main memory" were "overlay" schemes in which a program was divided into sections and stored in auxiliary memory such as a drum or disk. Sections were read into main memory for execution as needed. While this alleviated both the memory space problem and the addressing problem (since an instruction only needed to contain enough address information to refer to data in those sections that were also in main memory), it did so at extremely high cost to the programmer. The programmer had to explicitly divide the program into sections and assign a place in memory where each section would reside during execution. Cumbersome means were necessary to allow one section to generate addresses of data in other sections. Furthermore, careful analysis of runtime memory usage was necessary in order to make the overlay operations efficient. Such programs could not be effectively written in high level languages. For modern large software systems to run under such conditions would make them prohibitively costly and complex. Current virtual memory systems provide a solution to the memory space and addressing problems in a manner that is conceptually simple and consistent with clean programming in high level languages.

A basic virtual memory implementation consists of a small fast memory to improve the apparent speed of a slower large auxiliary memory. The virtual address space is the address space of the auxiliary memory, so the computer processing unit generates addresses that refer directly to that memory. A small part of the data is kept in the high speed memory unit, and the computer directly

manipulates only the high speed memory. Data is transferred between the high speed unit and the auxiliary memory by a combination of hardware and software mechanisms to keep the "most active" data in the high speed memory at all times. If the algorithm for replacement of data blocks in the fast memory is properly designed and the fast memory is large enough to hold the "working set" [26] of the computation, then the overall performance will be nearly as good as that of a system whose virtual address space is implemented entirely with the fast memory.

Virtual memory techniques are commonly used at two levels of system organization: in "demand paging" memories and in "cache" memories. Demand paging memories usually involve use of the computer's main memory (magnetic core or MOS RAM) as the high speed element and a magnetic disk or drum as the auxiliary memory.

In a typical simple paging memory system, the computer's high speed (e.g. core) memory is divided into blocks or "pages" of a thousand or so words each. The auxiliary memory (e.g. disk) is divided into a much larger number of pages of the same size. The virtual address space is the size of the disk, so the addresses generated by the computer are long enough to uniquely specify a word on the disk. Copies of some of the pages are kept in core memory, where they are accessible to the processor. Whenever the computer generates a memory address in an instruction, high speed hardware checks whether that address lies in a page that is in core. If so, the address is translated to the appropriate "physical" address and the memory operation takes place. If the indicated page is not in core memory, a "page fault trap" occurs, and software takes over. The operating system decides which page in core to replace, hopefully choosing one that is not likely to be accessed again soon. It writes the contents of that page onto the disk and then reads in the desired page from the disk. If the page being displaced has not been modified by any computer instruction, it does not need to be written back onto the disk.

Efficient demand paged memory systems require a special implementation mechanism in the computer's hardware or microcode. At the very least, an address translation mechanism is required for directing virtual memory references to the appropriate real memory address, and causing a page fault trap if the data is not in the high speed memory. A mechanism to detect writing operations on a page is useful for determining whether the page needs to be rewritten onto the auxiliary memory when it is displaced from main memory.

The first demand paged virtual memory system was implemented on the Atlas machine at Manchester in 1960 [63], and the idea has since been adopted for a number of commercial systems such as the DEC TENEX [10] and IBM 370 [13] systems.

The "cache" mechanisms of high performance computers are a form of virtual memory. Such mechanisms are implemented entirely in hardware or microcode. A cache operates on the same principle as a demand paged memory, using a very high speed memory as the fast component and the computer's main memory as the "auxiliary memory". The virtual memory space is unstructured and is identical to the computer's main memory. Cache systems are thus invisible to the program at the instruction level, and are used simply to increase the computer's speed. The invisibility to the programmer means that a cache can be added to a computer design without changing the instruction set or otherwise impairing compatibility. This is in contrast to the use of high speed "general registers" in a processor, another method of decreasing the time required to access heavily used data items. A cache was first used on the Atlas 2 machine at Cambridge University [69], and was introduced commercially on the IBM 360/85 in 1969 [48].

A very large directly addressable virtual memory may solve the memory size problem but aggravate the address space problem: address fields of instructions need to be very large to generate the necessary addresses. This can be very wasteful when the virtual memory is much larger than the amount of memory that a program actually needs to use at any one time. For example, in the paging memory illustrated above the address space consists of the entire disk. Properly designed memory structures attack both the memory space problem and the addressing problem. The addressing problem can also be alleviated by means that are independent of the virtual memory mechanism, but this often precludes the implementation of a unified addressing and protection mechanism. Many computers (for example, the IBM 360/370) use an address field consisting of a "base register" number and a relatively small displacement. These registers are somewhat similar to the segment base registers to be described below, but differ in that, in computers like the 360/370, the memory space is still treated as an unstructured homogeneous array, and the contents of base registers may be set arbitrarily by any executing program. This destroys their potential use as part of a protection scheme.

3.2. Structured Virtual Memory (Segmentation)

More sophisticated virtual memory systems than the "unstructured" type just described are implemented by some contemporary operating systems. These designs arose from efforts to solve the memory space, addressing, and protection problems together. In such a system, a machine level program has access at any time to only some portion of a large address space. The memory space is viewed as a collection of "segments" or "objects", accessed through "codewords", "descriptors", or "capabilities" [33]. In some systems, these objects may contain capabilities for other objects, thereby permitting tree structures of objects. To

make reference to a location in memory, the program specifies a codeword or descriptor of the segment and the offset within that segment. In more elaborate systems, descriptors or capabilities are grouped into "capability segments" (called "descriptor segments" on Multics or "program reference tables" on the B5000) that define an addressing environment for a program module. Mechanisms are provided to change the addressing environment as control passes from one program module to another.

The Rice University Computer [44] introduced the use of codewords to control access to objects in memory. The Burroughs B5000 [4] introduced the "program reference table", a form of descriptor table, and provided for automatic retrieval of segments from auxiliary memory. An advanced version of this system is the current Burroughs B6700 system [53]. The Multics system [20, 52, 22] was the first to provide dynamic generation of the access tables and automatic access table switching during program module entry and exit. Multics was first implemented using the special features of the General Electric 645 processor and has evolved into a commercial software product supported by the Honeywell 6180. The C.mmp system [70] is an object-oriented system using a naming method in which each capability has a universal meaning wherever it is used throughout the system. That is, capabilities may be transmitted from one process or module to another yet always refer to the same unique object. C.mmp also allows objects to be things other than segments, such as I/O devices, files, and protected entries to programs.

3.3. Protection

Protection mechanisms are used in multi-user computer systems to prevent programs from being interfered with, accidentally or intentionally, by other users' programs [60]. More recently, protection mechanisms of various kinds have been appreciated as aids in the development, operation, and maintenance of reliable software. This is because a major stumbling block in the development of software systems, especially large and complex systems, is the problem of unwanted interactions among modules.

Protection mechanisms arise naturally from mechanisms to solve the addressing problem because data can be protected by guaranteeing that there is no way a program module can generate addresses referring to the data. For example, if program modules are stored on a disk and read into high speed memory one at a time, the size of the entire program is much larger than the set of addresses that the computer instructions can generate. Since the address field of an instruction is presumably only large enough to refer to an address space the size of high speed memory, the other areas of the virtual memory space (i.e. those modules not in high speed memory) are inaccessible.

Memory protection of program modules from each other can be provided by the virtual memory techniques described previously. These generally work by making it impossible for a module to generate addresses in other modules, rather than by checking all references to see whether they are legal. For example, in a segmented system, this is done by protecting the segment descriptors (base addresses) in the capability segment. This is easily accomplished by placing the capability segment in a protected area of memory (that is, in a segment that is not itself listed in the capability segment). When this is done, there is simply no way that a program can refer to memory locations that are not currently in its capability segment. In systems that use capabilities for protection, a capability can be thought of as an unforgeable "ticket" which gives the program holding it the right to access the segment that it represents. Such systems usually have provisions for capabilities to represent all items that need to be protected: segments, I/O devices, data bases, etc. The capabilities have encodings of all the necessary access rights. For example, a segment capability might contain bits telling whether the segment may be modified by the process using that capability. Provision for controlled sharing of objects (e.g. "public" files and programs) is also very important. Pioneering systems of this type were the Burroughs B5000 and Multics.

An operating system supporting protected use of segments, files, or other memory abstractions must have a provision for making it possible for a module to access objects by name. A common way of doing this is through the use of "directories" which contain associations of names and capabilities. These capabilities may provide access to other directories, allowing a tree structure of accessible objects. Once again, the protection mechanism works by making it impossible for the module to "address" objects which it has no right to access. A module accesses an object by specifying its access path (list of names) through the directories. If an object is not in the directory, there is no access path leading to it.

Protection on a more microscopic level than the per-module basis is also frequently useful, though not nearly as common. Individual arrays can be protected from out-of-bounds references by using "descriptors" to refer to them at the hardware level, as was first done on the Rice University Computer [44].

3.4. Research Directions

Historically, addressing and protection features of computer systems arose from the requirements of multiprogrammed and multi-user ("time-sharing") computer systems. The goals were to provide for efficient sharing of processor and memory resources by independently written programs, and to provide for controlled sharing of on-line programs and data. The mechanisms were

developed in terms of machine level concepts of program organization, and little consideration was given to how these mechanisms would relate to or support large programs written in a high level language. It has turned out that, although it is possible to use the addressing facilities of the more advanced systems to implement large application programs having good structure, the modules of these programs cannot be written in a standard high level language. Rather, control of the addressing environment must be accomplished by calls upon system routines whose effects can be understood only through detailed knowledge of the hardware/software mechanisms of the system.

There is now considerable interest in "object-oriented" systems stemming from the ideas developed in the Hydra operating system for C.mmp [70]. It seems that such a system should provide a natural hardware/software base for supporting object-oriented programming languages such as Alphard [71] and CLU [49] that have evolved from studies of program structure and data abstractions. Hopefully, these separate developments can be brought together to produce computer systems that support the cooperative construction of large programs using modern software methodology.

One should not underestimate the difficulty of this task. It is by no means clear that current object-oriented systems provide the correct basis for object-oriented languages, since this was not an objective in the design of these systems. Consistency between system behavior and language semantics in such areas as module interfaces, support for data structures, access control, memory management philosophy, and coordination of concurrent activities must be understood and specified. Achieving the understanding of the requirements for consistency between addressing and access control mechanisms of computer systems and the semantics of quality programming languages is a most important area of system research.

4. Parallelism and Concurrency in Computer Architecture

An important way of improving computer system performance is through various forms of concurrency. A simple example is the overlapping of the execution of one instruction with the fetching of the next. This increases the utilization of both the processor and the memory system.

Within the fundamental speed limitation of any specific implementation technology a variety of architectural schemes have been used to increase overall performance by using multiple hardware units arranged to operate concurrently. The principal schemes are:

(1) Instruction overlap to exploit parallelism contained within a conventional instruction stream.

(2) Multiple data streams, exploiting the parallelism inherent in algorithms expressible in vector or array form. This is done either with an array of processing elements or with one processor that performs the operations in "pipeline" fashion. Such machines are often called "supercomputers".

(3) Multiple instruction streams, each with its own data stream, to exploit parallelism in asynchronous parallel algorithms and parallelism due to independent users. These machines are called multiprocessor systems.

The first technique, overlapped instruction execution, is widely used but the degree of parallelism is limited to the number of concurrently executable instructions that may be identified by "lookahead" in the instruction stream. The advantage of this technique is that it can be completely transparent to the programmer. Manufacturers can offer a family of computers with the same machine language but a range of performance according to the degree of instruction overlap employed in the processor implementation.

Single Instruction, Multiple Data (SIMD) stream computers are suitable for algorithms expressible in vector or array form. This includes some problems having large economic significance, such as seismic data processing and weather forecasting.

The third form of parallelism, with multiple independent instruction and data streams, has the potential of offering performance improvement for a wide range of applications. Small multiprocessor systems, with two to four processors, have been used commercially for some time [32]. Normally these are used to exploit parallelism between independent users in a timesharing facility. C.mmp [70] is a fully operational 16-processor system with a sophisticated, general purpose operating system.

4.1. Single-Sequence High Performance Machines

A number of techniques are used to gain increased performance in conventional single-sequence computers. These include memory interleaving to allow several memory operations to progress simultaneously, overlap of a processor's instruction fetch and execute cycles (fetching the next instruction before the current one completes), instruction lookahead, and use of a cache. Computers with instruction lookahead may fetch several instructions ahead of

the current one, analyze their data dependency, and execute those that do not depend on of results of other incomplete instructions. Cache memories have been described in Section 3.1. Although these techniques increase the complexity of processors, they have been worthwhile because computer system costs have, in the past, been dominated by memory costs.

The most ambitious early effort was the Stretch computer [9], which even attempted to execute instructions whose effect might have to be canceled once the outcome of a test became known. The mainstay of high performance computers has been the CDC 6600 and 7600 machines [66] and IBM machines evolved from the 360/91 and 92 [15].

4.2. Supercomputers

The need to take advantage of concurrency in a computer system in order to improve performance has led to a number of unusual machine architectures called "supercomputers". There are a variety of such architectures. They are generally intended for large numerical problems, and therefore give emphasis to efficient processing of vectors and arrays.

"Vector" or "pipeline" machines such as the CDC STAR [14], CRAY-1 [21], and TI-ASC [68] are basically sequential machines whose instruction sets include instructions that direct the machine to perform certain operations repetitively over an entire sequence of values. This makes it possible to do the arithmetic operations at extremely high speed without the need to perform separate instruction fetches and other bookkeeping tasks for each operation.

An "array" machine such as the Illiac IV [5] has a large number of arithmetic units and can do array operations truly simultaneously. The arithmetic units are all controlled by one instruction processor operating on one instruction stream, and the only autonomy that the arithmetic units have is the choice of executing or not executing (depending on the state of the small local memory) the current instruction.

Existing supercomputer designs have a number of drawbacks and problems that are the subject of intense research. One problem, discussed below, is the difficulty of writing efficient programs. Another problem is that the performance of existing machines on operations other than those for which they were optimized is often not very good. For example, some vector computers perform poorly on simple ("scalar") data. Unless the program is written in such a way that scalar operations are infrequent, the speed of scalar operations will tend to have a dominant effect on the overall speed of the computation, and much of the benefit of rapid vector processing will be lost. A third problem is that

supercomputers do not appear to be well adapted to LSI technology. LSI devices are generally slower than the circuits of which current high performance computers, including supercomputers, are made. The advantage of LSI devices lies rather in their small size and extremely low power consumption. Therefore, a computer that exploits LSI technology must do so by using a large number of these relatively slow circuits, rather than a smaller number of very fast ones. However, this does not seem to be the direction in which supercomputer development is going. Present supercomputers use the fastest logic circuits available.

In the case of array machines, the independent processing elements provide a natural modular decomposition. However, instructions are normally broadcast from a central control element and the consequent need to maintain close synchrony between processing elements may limit the speed and/or number of PE's. Reliability can only be provided via standby spares because vector operations are clearly not very tolerant of a variable number of processing elements. The very narrow range of applicability limits the market for these machines and so they derive little benefit from economies of scale.

An alternate approach to high performance in specific applications is the use of specialized machine architectures. An example is the Goodyear STARAN Associative Processor [7], in which a computation on an array of words is performed as a sequence of operations on bit slices of the array. Another was a proposed bit array processor [67] intended for use in image processing. Such machines do not seem to support conventional high level language concepts and present a difficult programming challenge. Nevertheless, they offer a large performance advantage in the applications for which they are designed. Given the ease with which such machines can be built, and the difficulty of building and programming more powerful general purpose machines, such specialized machines will certainly become increasingly common.

4.3. Programming Supercomputers

Clearly, if the performance potential of an array or vector machine is to be realized, a good way of generating machine level programs that make heavy use of the vector or array operations implemented by the hardware is required. If programs are to be written in a high level language, two approaches are possible: Either a conventional high level language may be extended to include composite operations on vectors or arrays as visible language features, or a language processor may be developed to recognize instances of vector or array operations in ordinary programs. In the latter case experience with "vectorizing" compilers has been disappointing in terms of providing object code that achieves high performance from existing high level language code. In either case, if the special

capability of a vector or array machine is to be exploited, the programmer must write his high level program in such a way that vector or array operations predominate and scalar operations account for a small portion of all computation steps. This presents a real challenge to the programmer. Typically, computational algorithms are drastically revised to make efficient use of vector operations, and problem data are reformatted from their natural aggregate form to permit advantageous use of distributed operations. This observation is supported by experience with APL in which, although aggregate operations on high level data types allow simple expression of algorithms, the resulting programs typically run slowly.

At present, the performance potential of array and vector machines is reached only with the expenditure of considerable programming effort, and this is true even when the application has previously been programmed in a high level language. The programming is done either at a low level or using a language processor extended to support the aggregate operations built into the hardware.

While one cannot rule out the possibility that automatic vectorization of algorithms will become a feasible methodology for using supercomputers, the prospects are uncertain. It seems that the most positive remark that can be made about supercomputers and software technology is that they have in no way contributed to simplifying the construction of software.

4.4. Multiple Sequence Computer Systems

The concept of having several parts of a computer system operating simultaneously by executing independent instruction sequences was put into practice in the early input/output processors, for example the I/O channels of the IBM 709. However, the Burroughs B5000 [4] appears to be the first system to use multiple processors performing computational tasks -- a (shared memory) multiprocessor. This one project introduced an amazing list of innovations: the concept of identical processor and memory modules interconnected by a matrix ("crossbar") switching arrangement; the idea that each processor acts independently and takes tasks from a common scheduling list; the concept that machine code is read-only so several processors can be executing the same procedure simultaneously; and the idea that many programs may run on such a system at once and that any processor may execute any program.

The Burroughs B5000 was one of the early "multiprogramming" systems, in which system programs controlled sharing of system resources by a number of concurrently executing programs. Its development was motivated by the performance gain possible through use of several processing units. This made

sense only in an environment in which many independent programs could be running concurrently, for there was no accepted methodology for writing single programs that could use several processors at once.

In multiprogramming systems, each program may be viewed as defining an independent computational *process* (sequence of instruction executions) which may be run on any available processor in the system. Innovative multiprocessor systems developed after the B5000 include the D825 [1], Multics [52], and C.mmp [70].

It is not necessary to have multiple processors to do multiprocessing, as a single processor may be arranged to switch its attention among many programs (processes). Early systems of this type were the Atlas [63] and Honeywell 800 [11]. This technique is used in all modern operating systems.

4.5. Multiprocessor Systems with Shared Memory

As the cost of processing units has fallen precipitously with the evolution of minicomputers and then microprocessors, the attraction of multiprocessor systems as a research area has grown immensely. Perhaps the best known project is the C.mmp system [70], in which up to 16 processors are connected to 16 memory modules through a 16x16 crossbar type switch. This system has allowed study of how parallel computations may be organized to run effectively and to find where the limitations in system performance lie. A variety of artificial intelligence and numerical tasks have been implemented for it.

Multiprocessor systems with up to, say, eight processors are in moderately wide use to provide reliability, to extend the performance of conventional timesharing and batch facilities and for dedicated control tasks. However, there are very few systems with dedicated processors whose stated goal has been to exploit the obvious potential for high performance. Pluribus [40] is an interesting example of a medium sized multiprocessor (the largest known system has 14 processors) which has been used successfully. It is used for a single, dedicated task -- a node in a packet-switched network. A single processor in this application could not meet the performance and fault-tolerance requirements.

Why have multiprocessors not been more successful as an approach to high speed computation? There seem to be two reasons: programming problems, which will be discussed later; and technical problems that make it difficult to achieve high performance. Multiprocessors pose several challenging efficiency problems: One such problem is context switching, that is, switching a real processor from one process to another. Operating systems must execute many instructions to change contexts, so programs must make such changes relatively

rarely if performance is not to be badly degraded. Another source of inefficiency is in naming. Data items in memory that are accessible to all processes should have the same name (whether a symbolic name or a numerical address) in each process's address space. This can require a complex translation mechanism, often involving multiple table accesses if virtual memory systems are used. The virtual memory system, naming mechanism, context switching mechanism, and operating system software all interact, and the overhead can be quite large even if the hardware and software are designed very carefully. These systems also have a technical problem in the connections between processors and memory units. To operate efficiently, the memory system must be divided according to some interleaving scheme, so different processors can use different memory units simultaneously and each processor will, most of the time, have quick access to whichever memory unit it needs. (If this is not done properly, "memory interference" between processors occurs and may cause serious performance degradation.) Since each processor must be able to communicate with each memory, and the number of memory units must be roughly proportional to the number of processors, the necessary crossbar switch will have a complexity proportional to the square of the number of processors in high performance implementations. However, many interesting compromises in cost and performance are available in switching networks that exchange delay for reduced complexity.

4.6. Distributed Systems

Some of the technical difficulties that arise in multiprocess systems can be alleviated by giving each processor a private memory and letting the processors communicate with each other by some means other than a shared memory. The usual method is to make the communication channels between processors behave like input/output devices, and to let the processors communicate by sending "messages" to each other. These systems are commonly called "distributed systems" or "computer networks". The UCI Distributed Computing System [34] is an example of a distributed system that has been developed. Among those under development are the HXDP network [46], the DCCS network [30], and the "office automation" systems being developed by IBM, Wang Laboratories, Xerox Corporation, and elsewhere. The rationale behind distributed systems is that most applications can be partitioned so that message communication, rather than direct references to a shared memory, provides good performance.

Distributed systems avoid the problems of memory interference and the large crossbar switch between processors and memory, requiring only transmission of "messages" among the processors. Much of the overhead associated with naming and memory mapping is alleviated, but overhead associated with data transmission arises, and it becomes very expensive to send

large blocks of data between processors. Furthermore, distributed systems are not suitable for "object-oriented" languages or program structures. Systems that use memory objects such as segments or data structures, or process synchronization objects such as semaphores, are currently popular, but they are nearly impossible to implement on distributed systems. This is because the objects themselves cannot be transmitted in any meaningful way through a communications link.

Distributed systems provide interesting opportunities for the design of fault-tolerant systems, making them useful for military and business applications where reliability is important. The properties of distributed systems that make this feasible are the "loose" inter-processor coupling and the absence of critical centralized subsystems such as crossbar switches. A degree of fault-tolerance may also be achieved in a multiprocessor system (as in the case of Pluribus [40]), but it is much more difficult to do so. A properly designed fault tolerant distributed system undergoes "graceful degradation" in the presence of component failures: The system continues to function, but with decreased efficiency or performance.

Use of distributed hardware alone does not assure fault tolerance -- very careful software design is required also. The software must be free of critical centralized databases and scheduling or coordinating programs. This is an extremely challenging problem. It is of course also necessary to choose a network topology with multiple paths between any two computers, so that a computer or link failure will not cause the network to become disconnected.

Another advantage of distributed systems is that the individual processors do not need to be powerful or expensive. Even microprocessors can be used to build high performance systems. The availability of low-cost microprocessors makes distributed systems an attractive approach for parallel processing systems.

The Cm* system [64] achieves the appearance to the program of a shared memory multiprocessor, but uses processors each of which has its own local memory. Like a distributed system, this eliminates most of the expense of the processor/memory crossbar switch. In the Cm* system, processors are grouped into clusters, with a "K-map" module on each cluster, which handles memory requests from one processor to the memory of another. The K-map units are further interconnected to handle memory references between processors of different clusters. The overall organization is such that any processor can address the memory of any other processor. Processors communicate simply by reading or writing other processors' memory, and these memory transactions are performed as data transmissions on the interconnecting busses. Hence the system's appearance to the programmer is that of a multiprocessor with shared memory, and programming techniques developed for multiprocessors and "object-oriented" systems can be used. The effect of using a relatively low speed

data bus instead of a central crossbar switch is simply that memory references between different processors are slower than local references.

4.7. Programming for Parallel Processors

Programming for multiprocessors and distributed systems requires radical changes in the way algorithms are presented, and this is probably the most important single reason for the failure of multiprocess systems to become widely accepted.

One such problem is task decomposition. Not all programs decompose well for execution on parallel processors. The speed improvement over a single processor is quite impressive for some programs, and not very good for others. A more serious problem is the ease of task decomposition for programs written in high level language. Nearly all programs that have been written for parallel processors were carefully and expertly coded in a language in which the use of parallel processors was specified explicitly.

The question of whether parallelism should be specified by the programmer or detected by the language translator is similar to the question of finding vector operations in programs to be run on supercomputers. The success of "parallelizing" compilers for translating existing high level language programs has been no better than that of "vectorizing" compilers. On the other hand, it is not clear that programming systems in which the parallelism is visible to the programmer are able to make parallel programming cost-effective in terms of programming effort. Writing correct parallel programs is, with present design methods, notoriously difficult. Furthermore, there is as yet only one well known high level language, Concurrent Pascal [12], that is aimed at the needs of concurrent programming.

The basic concepts involved in programming for multiprocessors are fairly well known. (In fact, they are older than multiprocessors, having been used earlier on "virtual process" systems and timesharing systems.) The principal concepts are process synchronization and mutual exclusion of processes from "critical regions". The most common techniques for implementing these are semaphores [29], monitors [41], and "test-and-set" instructions. These concepts are discussed in the paper in this volume on concurrent programming.

However, there is as yet no consistent or adequate methodology for using these concepts in application software, whether in high level or low level languages. Methods of verifying correctness of parallel programs are still in a primitive state, and verification is even more important in parallel programs than in conventional ones, due to the nondeterminism of the underlying system.

Design of programs for efficient execution on parallel processors also requires careful attention to issues other than analysis of concurrency. In distributed systems, the relatively high cost of transmitting messages between processors (as opposed to communicating through the shared memory) needs to be considered carefully, to minimize the amount of data that must be communicated. In the Cm* system the cost of transmitting data between processors is largely hidden, but it still exists, and performance suffers if programs attempt to send large amounts of data between processors that are distant in the interconnection structure.

In distributed systems a few additional programming problems arise. The difficulties of simulating semaphores and global memory "objects" were mentioned previously. Also, design of programs that use fully distributed control, necessary for fault tolerance, is a challenging task. This is a very active research area, and poses many interesting problems.

4.8. Data-Driven Computers

There is a growing interest in computers in which instructions of a stored program are activated by the arrival of data; these are called data-driven computers. Since many instructions in a data-driven computer may be activated at one time, the possibility of concurrent instruction execution is an inherent property. Indeed, some proposed forms of data-driven computers are claimed to achieve concurrent processing of hundreds or thousands of instructions.

Data-driven computers are a radical departure from the Von Neumann form of stored computer, since the central concept of sequential instruction execution is discarded. An instruction is ready for execution exactly when all the data items it needs arrive (from other instructions) at the holding registers reserved for them.

A variety of hardware arrangements for implementing data-driven instruction execution and reaping the benefits of concurrency have been proposed. One approach depends on partitioning a program into concurrently executable parts, and the dynamic assignment of hardware units to process those parts. Examples of this kind of data-driven machine are the "data flow multiprocessor" of Rumbaugh [59], in which processing units are dynamically assigned to data flow procedure activations, a system being constructed in Toulouse, France by a group under Syre [65], and a machine built at Burroughs by Davis [24] to be part of an envisioned tree-structured hierarchy of machines.

A second approach uses a small hardware unit for each instruction of a stored program, and communication networks through which data values are sent to instructions, and instructions, once activated by arrival of the necessary operand values, are sent to an appropriate processing unit for execution. Work on this form of data-driven architecture is in progress at MIT [27], and closely related work is being done at the University of California at Irvine [2].

Since data-driven computers are a radical departure from conventional architectures, careful design is required to ensure that they support sound programming methods. There is hope that they will, because data-driven computers generally require complete independence of program parts as the criterion for allowing concurrent execution. This requirement of independence of program parts is consistent with efforts to identify program structures that aid reliability, maintainability, and proof of correctness.

4.9. Research Directions

To evaluate the future prospects of the various system organizations, it is necessary to consider their cost/performance, suitability for LSI fabrication, and support of a sound programming methodology. The present forms of single sequence high performance computers and supercomputers do not seem to be the most promising directions. They use expensive high speed logic, and do not have large, repetitive structures, so LSI technology is not so useful.

The various forms of parallel and data-driven processors are much more attractive designs for taking advantage of LSI technology. These systems show promise for future evolution, but much work will be required before this promise can be realized. Multiprocessors with shared memory and systems such as Cm* require careful consideration of the design of hardware structures such as memory busses and switches to maximize efficiency. The "message passing" technique used in distributed systems will require careful hardware design if it is to compete with the memory sharing technique. These systems also need to be designed so that the hardware naturally supports the requirements of programming and operating systems.

The most important research area is the design of programming languages and methodologies for the various system organizations. All of these architectures require careful design of the languages with which they are programmed -- carelessly chosen languages may lead to hopelessly inefficient execution. Development of languages that satisfy the machines' requirements will be a major achievement and will have a profound effect on the future of these systems.

5. Language-Based Architecture

In the last two sections we have seen how the changing nature of desired computer services and demands for high performance and economy have led to major innovations in the structure and organization of computer systems. These innovations have produced practical computer systems of great complexity, and we have seen how this complexity often works to the disadvantage of the programmer: If he wishes to exploit the innovative aspects of the system he can no longer express his computation within the scope of a usual high level language because the innovative aspects of the computer are accessible only through machine level programming.

Many innovations, especially in addressing and protection, were designed to facilitate the construction of large programs in a multi-user operating environment. In their most advanced form (in systems using capability- or object-based addressing/protection schemes), these mechanisms may be viewed as architectural concepts designed to narrow the gap between hardware structure and the requirements of a modern programming methodology.

On one hand a programming methodology must include a language for expressing programs, whereas on the other hand the capability architectures have not been carefully thought through regarding the feasibility of their use in support of a high level language -- existing or to be conceived. Rather, the mechanisms were invented as *ad hoc* solutions to certain system problems concerning the sharing of data objects and program modules, and controlling access to them. In consequence it is not surprising that inconsistencies are discovered when the attempt is made to provide users with a high level programming language in which they can express programs in a form that effectively uses the innovative aspects of the system.

Computer science has also followed another path toward providing the programmer with facilities better matched to modern ideas about program organization. This is through use of software -- elaborate compilers and runtime support (operating systems) -- to transform the bare machine into a more suitable programmer interface. The classic example is PL/I and OS/360 which present to the user such a tremendous range of facilities that his talents are taxed to the utmost by the knowledge required to effectively utilize them in his application. Yet, the OS-PL/I combination, itself consisting of an *ad hoc* assortment of features, does not support any well-defined methodology of program construction.

The complexity of this software illustrates the distance between the current hardware and desired function. This suggests:

(1) the hardware is being used very ineffectively, and

(2) system complexity engenders uncertainty about its correctness and consistency, especially in the face of continual updates and alterations to the system itself.

An alternative is to develop a computer system architecture with a precise and complete statement of the base language that the system will support at the machine level. Such a machine can support many user languages, implemented through translation into the base language, or by execution by an interpreter expressed in the base language. Then hardware, firmware, and system software may be developed with a clear objective: to provide a correct implementation of the specified base language that meets cost/performance criteria. This approach will be called *language-based architecture.*

Some machines have been designed to represent properties of existing languages directly in hardware. An example is the Fortran machine [6]. Also, there have been various proposals for an Algol 60 machine [17, 39], and systems dedicated to support APL are now marketed. But these machines do not represent significant architectural innovation. The languages suggested are standard languages with all their deficiencies, and the machines are designed by putting the standard translation and runtime support routines into microcode or wired logic. These implementations obviously share all the problems that software implementations have -- they simply run faster.

On the other hand, there have been several occasions in computer history when new mechanisms were introduced expressly to support language features. Two stand out as particularly significant.

One of these is the "stack" mechanism provided in the Burroughs B5000 [4] to support procedure activation and termination including the allocation and deallocation of storage locations for local variables. This was motivated by the requirements of "block structured" languages such as Algol 60, which provide for local variable declaration and permit recursive invocation of procedures. The present B6700 [53] provides a more advanced version of this mechanism. Other computers incorporating this sort of stack mechanism include English Electric KDF9 [23] and the Burroughs D825 [1] series.

The other early innovation in support of a language concept is the codeword scheme introduced by Iliffe in the Rice University Computer [44]. Although, as we have seen, Iliffe's idea led to the use of descriptors and capabilities as mechanisms to implement controlled sharing of procedure and data segments, the original motivation was to provide hardware support for an efficient implementation of dynamic data structures (for example, arrays whose bounds vary during program execution). Basically, accessing a memory segment indirectly through a codeword allowed detection of a reference to a word in the

segment for which no storage was allocated. This caused control transfer to software routines which could alter the allocation of memory as required. Since all reference to segments was via codewords, and conventions ensured that the software could locate all codewords, the physical addresses in the codewords could be updated to reflect the changed storage allocation without concern by the application programmer.

Iliffe's idea is a precursor of present thinking about abstract data types in programming. It is unfortunate that this aspect of his work was ignored for so many years. Recently there has been renewed interest in descriptors, as exemplified by the VAX-11 computer [25].

5.1. The Symbol Machine

While important innovations, the Rice University computer and the Burroughs systems are not language-based architectures because neither machine was designed to exactly implement a completely specified base language. Indeed, both projects had reached operation before the concept of giving a precise semantics for a programming language had gained much attention.

One project stands out as both introducing architectural innovation and being a *bona fide* language-based design. This is the SPL language and Symbol machine designed and built by the Fairchild Corporation and installed at Iowa State University for evaluation [57]. The most significant architectural innovation in the Symbol machine is the memory system which presents an unusual interface to the several specialized processing units. The memory system provides direct hardware support for the principal user level data types of the SPL language: character strings, and vectors whose components may be character strings or vectors. Vectors are represented in the memory system by a chain of cells each of which may hold either an arbitrarily long character string, or a pointer to another vector (the address of its initial cell). Any processing unit may request the memory system for access to the contents of a cell, to advance to the next component of a vector, or to scan a character string.

Unfortunately, the Symbol machine implemented the SPL compiler directly in hardware. This is generally not a wise step, since it restricts the system's flexibility and dedicates a large part of the hardware to performance of a function which is used only occasionally.

5.2. Research Directions

Perhaps the primary problem with language-based architectures is the lack of generality of the languages implemented. Usually, it is difficult or impossible for the user of a language-based machine to meet a requirement not supported in the design of the base language. Thus, to be generally applicable, the base language implemented must be complete: it must support the complete needs of a sufficiently broad range of applications that the machine is viable in the marketplace. If large-scale application programs are anticipated, then the language design must include provisions for:

(1) combining independently written program modules,
(2) coordination of concurrent activities (processes),
(3) input/output, and
(4) data base access and manipulation.

So far no such comprehensive and consistent design of a language and corresponding machine has been completed. However, certain current areas of research are closely related to this goal and promise important advances toward establishing the feasibility of ultimately developing generally useful language based architectures. One of these areas is the study of object-oriented computer operating systems and their relation to object oriented languages. This work has been discussed in Section 3. A second clearly important area of research is high level language support for concurrent programming. This research area is discussed in the paper in this volume on concurrent programming.

Finally, the projects to develop data-driven computers discussed at the end of Section 4 aim for general-purpose language based architectures. Data-driven computers differ so radically from the conventional form of stored program computer that conventional approaches to machine language design and code generation simply do not apply. Thus, for a data-driven computer, the relation of the machine architecture to the user high level language must be developed afresh. For this reason, these projects must pursue simultaneous development of a data-driven architecture and a corresponding base language whose characteristics support programs to be executed on the machine.

A natural characteristic of a base language implemented by a data-driven computer is that program modules execute without side effects, and there is no concept of control flow, hence no control transfer statements. To some, this quality seems consistent with the trend toward "clean" language designs and makes data-driven computers attractive as a basis for an advanced programming methodology, whereas to others this indicates a serious limitation in generality of such data-driven machines.

Language-based architecture is an exciting and promising field of research, but it places serious demands on the researcher wishing to make an important contribution. Both computer organization and programming language semantics involve elaborate descriptions and specifications. Ensuring that a computer design and a language specification are compatible and consistent is a formidable challenge. Positive practical results are still some years away; yet the potential gains warrant a substantial research effort.

6. Fault Tolerance

Fault tolerance is the ability of a system to remain in useful operation in the presence of hardware failures. It is an important quality of all computer systems because hardware that is immune to failure is unknown and equipment that is out of service due to breakdown is an investment that is not delivering its value. Yet there are many computer applications in which fault tolerance is of far greater significance. In real-time systems, a failure will often result in a transaction being lost; in an airline reservation system the cost might be just the inconvenience of entering a transaction request again, but in a banking system it might mean failure to enter an item in a personal account -- a more serious consequence. In these systems, of course, a failure that makes the service unavailable for more than a few minutes is catastrophic for the business it supports. Even more serious is the possibility of loss of life for a computer system performing air traffic control or supporting a manned space mission. The most taxing demand for fault tolerance has arisen in unmanned planetary exploration, where the on-board computers must perform for years with no possibility of human access for testing or component replacement.

Here we are concerned with the failure of a computer system caused by failure of hardware. A computer system may also fail -- in the sense of not performing its intended function -- if the intentions are not carried out faithfully in the design and implementation of its hardware and software. Much has been written about software failure, but this seems to be a misuse of the word: programs that do not perform their intended function on perfect hardware have not failed, they are simply incorrect.

Certainly the problems of constructing programs and hardware that are correct are very important, but the principal approaches toward improving confidence in hardware/software designs do not fall within the scope of fault tolerance research. Hardware correctness is aided by improved methods of hardware description, better structuring principles, automated design aids, and design verification and testing techniques. These are discussed in Section 7. Confidence in correctness of software is improved by corresponding techniques as discussed in the chapters of this book on language design and program

specification, verification, and testing. The computer architect can help by providing hardware that permits well-structured programs to run efficiently and by devising approaches to fault tolerance that do not interfere with the use of sound principles of program structure. Nevertheless some interesting work has been done on application of the modular redundancy concept to the problem of software correctness [38, 56].

In discussing fault tolerance, we distinguish between faults and errors. A *fault* is a temporary or permanent change in a physical component that causes it to fail to perform as intended. An *error* is the failure of a component to produce the intended result, and may be caused by a fault or by an incorrect design. Here we only consider errors caused by faults.

The ultimate objective of a fault-tolerant system is to implement *fault masking*, which means that occurrences of faults in the system are unobservable by system users. (The "users" may be programs, external equipment, or humans.) Schemes for implementing fault tolerance differ in the degree to which complete fault masking is achieved. The ideal fault-tolerant system would be one that masks completely all conceivable component failures. Realistic systems fall short of this ideal. Typically a scheme will only ensure masking of one failure at a time; furthermore, in most schemes the occurrence of a failure will result in degradation of some aspect of system performance or capacity even if system function is maintained. In general not even this degree of fault tolerance is obtained. In most schemes, a component failure, even though detected and recovered from, may lead to information loss that is observable by system users. The users must either accept the information loss as an unfortunate compromise, or they must implement additional fault tolerance mechanisms outside the system that was supposed to be fault-tolerant. These external mechanisms, although clearly useful, can never achieve complete fault masking, because they unavoidably make invalid assumptions about continued functioning of the underlying system following a failure.

6.1. State of the Art

To achieve fault tolerance, redundancy is required -- for masking faults and for detecting that faults have occurred. Redundancy in practice takes two principal forms: coding redundancy and modular redundancy. Coding redundancy is the use of extra bits in the representation of information so that a fault that causes an error in information is very likely to be caught by logic that tests whether the coded information is valid. Coding redundancy is mainly used for fault detection, although error correcting codes which permit correct data to be reconstructed from erroneous representations are now in use, for example in large semiconductor memories. Modular redundancy is the use of several copies

of a hardware module in such a way that some module can provide the correct result in event of failure of one module in a group. A popular form is triple modular redundancy in which any two modules whose results agree can outvote a failing module, thereby masking the fault.

Use of multiple copies of hardware that do not augment performance has not been attractive in large computer systems. Here schemes have been devised in which duplicate modules are used operationally, but the system is able to continue operation with degraded performance in event of module failure. However, failures are not masked and restoring system operation following detection of a fault may require human assistance. Another form of redundancy is repetition in time; a computation may be run again if an error is detected during the first run, or, if errors are likely to be transient, a computation may be run twice on the same equipment and the results compared. An example is the instruction retry mechanism used on some commercial machines.

In a system that does not mask all faults an alternative is to provide fault detection mechanisms which will inform the user of any fault. The user is then responsible for implementing any corrective action and resuming operation of application programs from a consistent state. The action taken by the user in response to a detected fault is called *recovery*. The most common recovery procedure is to restore the system to a known state -- a "checkpoint" -- that has been saved for such a contingency. Such a scheme usually does not provide complete fault masking because some faults may occur in parts of the system essential to correct functioning of the recovery procedure, or unmasked faults may occur in the fault detection and reporting mechanism itself.

The difficulty of implementing complete fault masking increases with the generality of system function. In the case of large scale computer systems designed to support general purpose use, little work has been done even to establish whether an ideal fault-tolerant implementation is possible, let alone to devise practical schemes. Efforts to develop fault-tolerant computer systems have been most successful in the case of machines dedicated to a particular application or application area, especially machines intended for space missions or airborne applications.

The ESS (Electronic Switching System) developed by Bell Laboratories [62] for uninterrupted operation of the telephone system uses duplicate processors that check each other continuously, and redundant memory devices and controllers. Failing components are switched out of the system automatically, and then manually removed and repaired.

The experimental STAR (Self Test And Repair) computer built at the Jet Propulsion Laboratory [3] utilized a number of techniques to realize fault tolerance. Extensive coding redundancy was used in memory and processing units. Recovery from transient errors was accomplished by restarting the application program from a point from which a valid prior system state could be restored. If a non-transient failure occurred, the computer "repaired" itself by switching to one of a number of standby units. The modules that performed the checking and reconfiguration used triple modular redundancy. If one failed, it too was replaced by a standby unit.

The fault tolerant multiprocessor developed at the C. S. Draper Laboratory [43] utilized triple modular redundancy and dynamic reconfiguration in the memories, processors, and interconnecting busses.

The onboard computer developed for the OAO (Orbiting Astronomical Observatory) satellite [19] used double, triple, and quadruple redundancy of various processing and memory units, with automatic disconnection of failing components.

In large commercial systems the need to preserve the data base and the need to use standard programming tools for applications are prime design considerations. In this context, the techniques used to realize the required protection from the effects of hardware faults are very different from the techniques used to achieve fault masking in the systems mentioned above. In general, complete fault masking is not realized, and some fault occurrences lead to information loss or to system failure requiring human intervention to resume normal operation.

In a transaction system (an airline reservation system, an inventory control system, an online banking system are examples), messages arrive from remote terminals and are processed using information retrieved from a data base, yielding updated records in the data base and responses transmitted to the terminal. In these systems it is most essential that the system be up and responsive to inquiries at all times. It is not as important that every individual transaction be carried out faultlessly, so long as users can recognize when a transaction has been incorrectly handled and can repeat the query. It is important that consistency of the data base be maintained; in particular, if it ever happens that processing of a transaction is terminated by a fault before completion, the data base must not be left in an inconsistent state.

The approach frequently used in such systems is to use a dual central processor/main memory configuration. If one system fails, operation can be switched to the other, usually in a few seconds or less. The new system recovers as much data as possible from the failed system and carefully checks its validity,

so that there is as little interruption of service as possible, and the data base is restored to a consistent state with as little information loss as possible.

In computer centers that support general purpose computing by many users, the reliability objectives are quite different. These systems must support application programming by users in several popular languages such as Fortran, Cobol, APL, etc. without imposing restrictions. Furthermore, the applications are such that loss of as much as a day's availability of the system is preferable to the expense of having duplicate facilities. The reliability consideration of greatest importance in these circumstances is the integrity of users' data bases - the files of programs and data retained in the computer system (and its tape/disk libraries) on behalf of users. Since the acceptance of online files and user interaction through remote terminals, the standard approach has been to periodically record all online files onto a set of backup tapes which are retained by the computer center until such time as there would be no possibility of their being required to recover information following a system failure. These backup procedures were pioneered in the MIT CTSS system [50], and developed to a very sophisticated form in Multics [52].

In these systems, a user might lose all information he has generated since the most recent dump in the event of a system crash. If users wish to avoid this possibility, they must build into their programs or their method of using the computer their own fault tolerance scheme. An example is the checkpoint/restart procedure used in long computations.

In a computer center employing a multiprocessor machine configuration with duplicate equipment of each sort (processors, memory units, input output controllers, etc.) one might expect that this hardware redundancy could be used to achieve "graceful degradation", without information loss. However, the advantages of resource sharing are such that operation of the process and storage management tasks are not at all independent. Failure of any CPU or memory module (main memory or disk) can leave scheduling or storage allocation data in an inconsistent state, possibly altered in such a way that continued system operation would be disastrous. No schemes for implementing fault masking in this context have been devised. It takes very careful thought and clear goals to achieve fault tolerance in such a general context.

6.2. Research Directions

All of these systems fall short of the ideal. Furthermore, there is little basis in theory for their structure. In each case, *ad hoc* extensions or adaptations of conventional computer structures are devised so that an adequate gain in fault tolerance is obtained. Achieving even this reliability generally demands that the

application programming be done following conventions that interfere with good program structure. Even so, the possibility of information loss, as a consequence of some single component failures, or as a consequence of recovery schemes, is almost always present and must be considered by designers of application software. Essentially no research has been reported concerning ways of achieving the ideal in a general purpose context, or to show that it is impractical to achieve such an ideal.

Much can be learned from further work on modular redundancy, redundant coding schemes, fault masking, detection and recovery techniques and their application. Since the ideal fault tolerant computer is beyond the state of the art, and there are important applications for highly reliable specialized computers, this is a wide open and significant area for future research. A realistic objective for research in fault tolerance is to expand the class of computer systems for which complete masking of single component failures is feasible. The aim should be to understand the cost/reliability trade-offs of various techniques and the types of failures that can be completely masked.

7. Methodology of Hardware Design

The correct operation of software depends on the correctness of the computer system on which it runs. With increasingly complex and elaborate hardware systems, greater confidence is needed that the hardware is performing as intended. Methodology of hardware design is required both to achieve this objective and to provide aids for the design and production of hardware systems.

There are similarities between hardware and software. Both are realizations of algorithms that implement high level behavior by combining lower level operations. Much research effort has been devoted to the problems of formal program specification, correctness and verification, and programming language design based on principles of structured programming. It is surprising that there has been little corresponding development of formal specification and proof techniques for hardware systems and their implementation.

Of course, much attention has been given to the use of computers in the production of new computer systems. Simulation of logic design has become a standard part of system development. The clerical work of translating logic designs into printed circuit layouts and wire lists has been largely taken over by the computer [37]. While these aids help ensure that a detailed design is faithfully implemented (and that all instances of a design are equivalent), they do not give confidence that the design itself is correct.

What is lacking is a methodology that permits an architect to develop a design with confidence that a system constructed according to the design will function as intended. Such a methodology would also be useful for establishing the equivalence and compatibility of a line of systems that implement the same functional specification over a range of cost/performance. In parallel with the corresponding approach to software methodology, there are several key elements:

(1) A formal notation for representing a design; we will call this notation an Architecture Design Language (ADL);

(2) A way of precisely specifying the function to be performed by the system (its behavior); we will call such a formalism a Function Specification Language (FSL); and

(3) Methods for verifying the correctness of a design given in an ADL against the intended function specified in a FSL.

Since designs can be represented at different levels of detail, a design at each level consists of functional specifications of each component and a structural description of how components are organized. An ADL, therefore, intrinsically includes a FSL as part of its definition. Once developed, an ADL and a FSL would not only provide a framework for verifying the correctness of designs but would also serve as precise communication media for independent development of system components.

In contrast to the effort devoted to software specification formalisms and verification techniques, little work has been done for hardware systems except at a low level of design involving microprogramming [47] and logic gate level designs [58]. Much work has been centered around developing description languages for register transfer level (RTL) designs [45, 61, 16, 31, 8]. (The term "RTL design" applies to a way of structuring a hardware design by separating control structures from data operation structures such as registers and functional operations.) While these languages provide features for description of RTL designs, they are inadequate as specification languages: formal semantics of these languages have not been defined, proof techniques have not been developed, and it has not been demonstrated that they are satisfactory for representing designs at higher levels of abstraction.

Yet RTL design languages exhibit some features desirable of an ADL: they are based on a methodology of design which is found applicable to a range of design alternatives including synchronous and asynchronous systems [54]; and considerable progress has been made toward the automatic translation of RTL designs into logic designs. In fact, studies have been done which successfully

used a limited set of hardware modules for direct realization of a design given at the RTL level [18].

New developments of more elaborate computer system structures such as distributed systems and special purpose processors suggest that RTL languages for specification and design may not be adequate. Furthermore, advances in large scale integration also seem to invalidate many assumptions on which logic designs have been based, and different design methods may need to be developed.

Ideally, an ADL can be used as a design tool which allows successive refinement of designs. To this end, an ADL must satisfy two conflicting requirements: at the highest level, it should be independent of technology of hardware components; and at the lowest level, it needs be sufficiently expressive to represent alternative designs exploiting particular choices of hardware devices and interconnection disciplines.

Accompanying the development of FSL's and ADL's, research is required in verification techniques for hardware systems that can eliminate the need for exhaustive simulation. This is particularly important in the case of systems in which there is much concurrency, and correctness of coordination mechanisms is impossible to demonstrate through simulation.

Another area is the specification of performance requirements and evaluation with respect to performance. Little work of general applicability has been done, most evaluations having been done using *ad hoc* techniques devised for the specific system under consideration. Relevant work of a general nature includes methods for the analysis of timed Petri nets [55]. The PMS notation [8] is a means for describing the gross overall structure of a computer system in terms of which some simple performance estimate for the system may be derived from the characteristics of its components. This is a profitable area for research.

8. Computer Networks

An area of great current interest is computer networks, meaning a collection of computers or computer systems together with facilities that permit programs running on distinct computers to interact by messages sent over communication lines. The most familiar example of a computer network is the ARPANET [35] created by the Advanced Research Projects Agency as an experimental system linking computer systems operated by universities and research institutes, and now operated by the Defense Communications Agency. Users of one host computer system may send messages, programs, and files to users of other network hosts, or may log in over the network as a user of another

host provided he has obtained authority to use resources of the other machine and has satisfied security arrangements. An older network of very different character is created by the communication links between computer-based airline reservation systems that permit one airline to confirm flight arrangements on other airlines.

An important aspect of a network is the degree of "coherence" it supports. The utopian ideal is a network of time-shared computer systems so organized that all users on the network feel as though they were using one system and had access to each other's programs and data bases just as they would on a single local system. No such network exists, and the problems standing in the way are formidable. These include the problem of handling queries and updates to a distributed data base in the most general context, and the problem of convenient program exchange among computer installations, neither of which have sufficiently general solutions at present.

A second important aspect of a network is the autonomy of the individual nodes. This autonomy is paramount in the two networks cited above. Each node is safe from disruption by the failure of other nodes and even from deliberate sabotage attempts by users or implementors at other nodes. In general, the nodes use different hardware supporting different operating systems and management philosophies. A consequence of this extensive autonomy is an almost total absence of coherence. Neither network cited above supports compatible exchange of programs or the sharing of distributed data bases except as deliberately provided for by the user community on their own.

In the Airlines Network an adequate level of coherence for the goals of the network is obtained just by standardizing the message formats and protocol for inter-airline inquiries, reservation requests and confirmations. The ARPANET supports a general protocol for establishing communication between programs running on different hosts. In addition, the network provides a general mail service. These facilities are made possible by specific additions to the host operating systems in accordance with conventions adopted by network planners.

It is clear that any increase in the level of coherence provided by a network will be achieved only at some sacrifice of autonomy. Researchers should seek to identify the different levels of coherence that are desirable, and to understand exactly what autonomy must be relinquished to obtain each degree of coherence. One of the most serious factors to be studied is the possible loss of failure tolerance in a network designed to realize the highest degree of coherence.

8.1. Personal Computers

The fast-decreasing cost of computer hardware has brought private personal ownership of computers into widespread reality. The prospect of many individuals having "personal computers", which would make available myriad services ranging from personal accounting and income tax aids to shopping and encyclopedic reference services, has captured the imagination of our science prophets [28]. What are the real prospects, research directions and opportunities presented by this vision?

Let us start by ruling out of the scope of our discussion the fixed program microcomputers which will be appearing in our ovens, washing machines and automobiles, for these are a straightforward extension of current microprocessor technology.

It is clear that before long a computer having the capability of a medium-scale System/360 machine will be available in a tiny package for a price affordable by the average family. The big question is: How will these machines be programmed? Current personal computers are purchased (and tinkered with) by computer hobbyists whose use of them depends on an intimate knowledge of machine language. How will the field evolve from this situation to one in which the naive user may call upon a variety of sophisticated services with only knowledge of the application/problem domain with which the service is concerned? The first steps in the evolution toward the ultimate personal computer are easy to foresee. First, relatively simple machines will be marketed together with applications packages provided by the manufacturer for applications of high interest to the consumer. As in the case of current microprocessors, the machines and their software will be intentionally incompatible across brands so each service will require the machine supplied by its manufacturer. Second, *de facto* interface standards will evolve as customers determine the most popular products based on the quality and breadth of service available, and these standards will become the basis for secondary sources of software packages. Personal computer users will emerge as a kind of computer network in which the principal communication mode is program distribution through the mails.

The tragedy of this evolution is that there will be no convenient and effective way for users to combine services offered by different software sources; this is because the underlying hardware and software standards are unlikely to satisfy even the most basic requirements for modular program construction. Also, there will be no carefully-thought-out general approach to the implementation of applications such as investment or shopping information services which require online access to proprietary data bases. The attainment of these levels of generality in personal computers requires solutions to problems that have not yet

been solved in the context of conventionally operated computer systems. Their solution will require deep understanding of the nature of the problems, and creative proposals for their solution. This is an exciting area for research. Through the *ad hoc* construction of networks and acquisition of personal computers we will soon discover how serious and difficult the problems are.

9. Conclusion

The major impact of recent innovations in computer architecture -- super-computers and microprocessors -- has been a corresponding increase in difficulty of the problems facing the software engineer. The most promising road to better architectural support for software is to narrow the gap between the abstractions called for by a problem solution and the capabilities directly realized by computer hardware. *Ad hoc* extension of past system designs must be replaced by system architectures based on sound principles -- the architect must have the view that his system implements the collection of features used to construct programs and must start with a precise understanding of what programming language and/or methodology his system will support.

In the long term, research in language-based computer system architecture may lead to hardware bases for computer systems that provide users with programming interfaces much simpler than those provided by the large systems of today -- interfaces that are more general, are consistent with good language design ideas, and meet essential requirements for modular program construction. It is not clear that such systems can be designed to run efficiently while remaining within the conventional architectural framework of sequential processing units. It is possible that these future systems will be radically different in structure from current computer systems and many years may pass before they become practically usable.

Recent work on tagged architectures [36] and capability-based systems has suffered from inadequate consideration of program structure and the programming language and methodology to be used in writing applications. The view of capabilities (or codewords as they were originally called) changed from that of a hardware mechanism to support a language concept, as advocated by Iliffe, to a mechanism for access control and information sharing in multiuser operating systems. With the development of object-oriented computer systems, there is new interest in developing corresponding programming methodologies and in relating the functions implemented in these systems to the requirements of object-oriented programming languages.

Closer at hand is the problem of effectively putting many microprocessors into harmonious cooperation and easing the accompanying software burden. Extending the classical multiprocessor-multimemory system structure has at least two drawbacks which must be addressed by any new proposal for research. First, the complexity of the processor/memory switch appears to grow in size as the product of the numbers of processor and memory units so long as uniformly fast response to memory requests is required. Even if this structure were attractive, implementation of the crossbar switch does not lend itself readily to advantageous use of LSI technology. Second, the cost of switching processors among computational processes so as to keep the processors occupied imposes a serious limitation on dividing programs into concurrently executable parts, and a satisfactory programming methodology for such systems remains a challenge to software research. (The development of Concurrent Pascal by Brinch-Hansen is a step in the right direction.) The Cm* project addresses the memory/processor switch problem by grouping processors (with local memory) around K-map units that handle nonlocal memory references with longer response delay but in a hardware structure that does not grow as the square of the processor count. It will be interesting to see how programming issues will be addressed within this system structure.

On the other hand, microprocessor technology offers the opportunity to develop a new programming methodology based on the unique properties of microprocessor hardware. An interesting area for architectural research is loosely coupled (no shared memory) multi-microcomputer systems using a packet communication protocol. The corresponding programming language/methodology would use message passing as an underlying semantic concept.

A successful architectural concept arising from one of the research areas suggested above will almost certainly fail to achieve practical compatibility with classical standards (such as the Fortran and Cobol languages and commercial operating systems) on which much of the existing inventory of software depends. It follows that the new systems will not be employed where the main requirement is to run old applications at existing levels of performance. Rather, the new systems will replace the old when and only when the advantages in performance and programmability outweigh the costs of maintaining the old system.

Of course, microprocessors are very significant in stand-alone applications in everything from cars to process controllers, instruments, and home appliances. However, the software structure is conventional. The role of computer architecture research here is to evolve microcomputer architectures that are better matched to the software structures used in stand-alone applications and that support a corresponding software methodology. The area most in need of creative work is the handling of the interaction of the microcomputer with its

environment and the problems of meeting real-time demands while maintaining an understandable and reliable software structure. In the near term the most important development will be standard software tools usable over a range of microprocessor designs. The architecture research issue is to identify a class of microprocessor architectures that are at once compatible with the standard software tools (including a high level programming language) and yet permit a wide variety of hardware trade-offs as technology evolves.

Supercomputers, in spite of a large effort devoted to the programming of applications, have not lived up to their expectations. Further research on software approaches to obtaining better performance from these machines is not promising. Until general approaches to highly parallel computers that can support spatially distributed computations are available, a good approach to important problems requiring high-speed computation is to build specialized hardware for the problem.

In the immediate future the principal driving force behind new commercial machines is the need to run old software and to support familiar languages. This explains why the IBM System/360 line has been so successful and why the 360 interface has been supported by several other hardware manufacturers. It is unfortunate that maintenance of support for the enormous inventory of commercial software requires machine compatibility at the instruction set level, and thus inhibits architectural innovation. This inherent conservatism focuses the attention of the architect on ways of describing accurately the standard interface that is the target of his system design, means of ensuring that his product exactly supports that interface, methods of predicting and evaluating the cost-performance of a system design, and means for building fault tolerance into systems. Research in these areas will help the development of more efficient and reliable systems to support the existing software base.

A more forward-looking prospect is that the specification of the new language DOD-1 by the Department of Defense will provide computer architects with a standard interface to guide their development of a new generation of computers for defense applications. It is even conceivable, albeit unlikely, that the DOD-1 standard may allow software compatibility without requiring a standard instruction set, permitting architects unusual freedom in designing hardware support.

9.1. Final Words

In reviewing the scope of this chapter one is struck by the great variety of structures and approaches being explored in the study of computer architecture. This is in surprising contrast with the trend evident in programming language

design toward languages based on sound principles of program structure. Computer scientists seem to be approaching a consensus about what characteristics a good programming language should have. It should be possible to provide efficient and reliable hardware for running programs constructed according to these principles. Yet there has been little research effort having this objective.

Current work in software methodology is providing new knowledge about program structure, especially in the area of concurrent programming. As this knowledge is incorporated into the design of programming languages, computer system architects should strive to learn how to apply it toward building hardware structures better suited to the needs of future generations of programmers.

References

1. Anderson, J. P. et al, "D825 - A Multiple Computer System for Command and Control," 1962 FJCC, AFIPS Conf. Proc., Vol. 22, pp 86-96.
2. Arvind and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," Proceedings of IFIP Congress, Vol. 7, 1977, North Holland Publishing Co., pp 849-853.
3. Avizienis, A. et al, "The STAR (Self Testing And Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," IEEE Trans. on Computers, Vol. C-20, No. 11, Nov. 1971, pp 1312-1321.
4. Barton, R. S., "A New Approach to the Functional Design of a Digital Computer," Proc. Western Joint Computer Conference, IRE-AIEE-ACM, 1961, pp 393-396.
5. Barnes, G. H. et al, "The Illiac IV Computer," IEEE Trans. on Computers, Vol. C-17, No. 8, Aug. 1968, pp 746-757.
6. Bashkow, T. R., A. Sasson, and A. Kronfeld, "System Design of a Fortran Machine," IEEE Trans. on Electronic Computers, Vol. EC-16, No. 4, Aug. 1967, pp 485-499.
7. Batcher, K. E., "STARAN Parallel Processor System Hardware," 1974 NCC, AFIPS Conf. Proc., Vol. 43, pp 405-410.
8. Bell, C. G. and A. Newell, *Computer Structure : Readings and Examples*, McGraw-Hill, N.Y., 1971.
9. Blosk, R. T., "The Instruction Unit of the Stretch Computer," Proc. Eastern Joint Computer Conference, IRE-AIEE-ACM, 1960, pp 299-324.
10. Bobrow, D. G. et al, "TENEX, A Paged Time Sharing System for the PDP-10," Comm. ACM, Vol. 15, No. 3, Mar. 1972, pp 135-143.
11. Bouvard, J., "Experience with Multiprogramming on the Honeywell 800-1800," Proceedings of IFIP Congress, Vol. 2, 1965, Spartan Books, Wash., D. C., pp 364-365.

12. Brinch Hansen, P., "The Programming Language Concurrent Pascal," IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, June 1975, pp 199-207.
13. Case, R. P. and A. Padegs, "Architecture of the IBM System/370," Comm. ACM, Vol. 21, No. 1, Jan. 1978, pp 73-96.
14. Control Data Corp., *Control Data STAR-100 Computer System, Hardware Reference Manual*, Publication 60256000, Arden Hills, MN, 1973.
15. Chen, T. C., "The Overlap Design of the IBM System/360 Model 92 Central Processing Unit," 1964 FJCC, AFIPS Conf. Proc., Vol. 26, part II, pp 73-80.
16. Chu, Y., "An Algol-like Computer Design Language," Comm. ACM, Vol. 8, No. 10, Oct. 1965, pp 607-615.
17. Chu, Y., "Architecture of a Hardware Data Interpreter," Proc. Fourth Annual Symposium on Computer Architecture, Computer Architecture News, Vol. 5, No. 7, March 1977, pp 1-9.
18. Clark, W. A. and C. E. Molnar, "The Promise of Macromodular Systems," Proc. Sixth IEEE Computer Society International Conference, San Francisco, Sept. 1972, pp 309-312.
19. Cooper, A. E. and W. T. Chow, "Development of On-board Space Computer Systems," IBM Journal of Research and Development, Vol. 20, No. 1, Jan. 1976, pp 5-19.
20. Corbato, F. J., J. H. Saltzer, and C. T. Clingen, "Multics - The First Seven Years," 1972 SJCC, AFIPS Conf. Proc., Vol. 40, pp 571-583.
21. Cray Research, Inc., *CRAY-1 Computer System Reference Manual*, Publication 2240004, Bloomington, MN, 1976.
22. Daley, R. C. and J. B. Dennis, "Virtual Memory, Processes, and Sharing in Multics," Comm. ACM, Vol. 11, No. 5, May 1968, pp 306-312.
23. Davis, G. M., "The English Electric KDF9 Computer System," Comp. Bull., Dec 1960, pp 119-120.
24. Davis, A. L., "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," Proc. Fifth Annual Symposium on Computer Architecture, SIGARCH Newsletter, Vol. 6, No. 7, April 1978, pp 210-215.
25. Digital Equipment Corp., *VAX-11/780 Architecture Handbook*, Maynard, MA, 1977.
26. Denning, P. J., "Virtual Memory," ACM Computing Surveys, Vol. 2, No. 3, Sept. 1970, pp 153-189.
27. Dennis, J. B. and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," Proc. Second Annual Symposium on Computer Architecture, Computer Architecture News, Vol. 3, No. 4, Jan. 1975, pp 126-132.
28. Dertouzos, M. L., and J. Moses, Ed., *Future Impact of Computers: A Twenty Year View*, MIT Press, Cambridge, MA, forthcoming.

29. Dijkstra, E. W., "Cooperating Sequential Processes," in *Programming Languages*, F. Genuys, Ed., Academic Press, N. Y., 1968, pp 43-112.
30. Donnelley, J. E., "A Distributed Capability Computing System," Proc. Third International Conference on Computer Communication, Aug. 1976, pp 432-440.
31. Duley, J. R. and D. L. Dietmyer, "A Digital System Design Language (DDL)," IEEE Trans. on Computers, Vol. C-17, No. 9, Sept. 1968, pp 850-861.
32. Enslow, P., *Multiprocessors and Parallel Processing*, Wiley, N.Y., 1974.
33. Fabry, R. S., "Capability-Based Addressing," Comm. ACM, Vol. 17, No. 7, July 1974, pp 403-412.
34. Farber, D. J. et al, "The Distributed Computing System," Proc. Seventh IEEE Computer Society Conference, Feb. 1973, pp 31-34.
35. Feinler, E. J., *Arpanet Resource Handbook*, NIC39335, Network Information Center, Stanford Research Institute, Menlo Park, CA, 1976.
36. Feustel, E., "On The Advantage of Tagged Architecture," IEEE Trans. on Computers, Vol. C-22, No. 7, July 1973, pp 644-656.
37. Fitch, A. E., "A User Looks At D.A. - Yesterday, Today, Tomorrow," Proceedings of Sixth Annual Design Automation Workshop, June 1969, pp 371-389.
38. Gannon, J. D. and J. J. Horning, "Language Design for Programming Reliability," IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, June 1975, pp 179-191.
39. Haynes, L. S., "The Architecture of an Algol 60 Computer Implemented with Distributed Processors," Proc. Fourth Annual Symposium on Computer Architecture, Computer Architecture News, Vol. 5, No. 7, March 1977, pp 95-104.
40. Heart, F. E. et al, "A New Minicomputer/Multiprocessor for the ARPA Network," 1973 NCC, AFIPS Conf. Proc., Vol. 42, pp 529-537.
41. Hoare, C. A. R., "Monitors: An Operating System Structuring Concept," Comm. ACM, Vol. 17, No. 10, Oct. 1974, pp 549-557.
42. Hoagland, A. S., "Magnetic Recording Storage," IEEE Trans. on Computers, Vol. C-25, No. 12, Dec. 1976, pp 1283-1288.
43. Hopkins, A. L. Jr. and T. B. Smith III, "The Architectural Elements of a Symmetric Fault-Tolerant Multiprocessor," IEEE Trans. on Computers, Vol. C-24, No. 5, May 1975, pp 498-505.
44. Iliffe, J. K. and J. G. Jodeit, "A Dynamic Storage Allocation Scheme," The Computer Journal, Oct. 1962, pp 200-208.
45. Iverson, K. E., "A Common Language for Hardware, Software, and Applications," 1962 FJCC, AFIPS Conf. Proc., Vol. 22, pp 121-129.
46. Jensen, D., "The Honeywell Experimental Distributed Processor - An Overview," IEEE Computer, Vol. 11, No. 1, Jan. 1978, pp 28-38.

47. Joyner, W. H., G. B. Leeman, and W. C. Carter, "Automated Verification of Microprograms," IBM Research Report RC5941, Dec. 1976.
48. Liptay, J. S., "Structural Aspects of the System/360 Model 85; Part II - the Cache," IBM Systems Journal, Vol. 7, No. 1, 1968, pp 15-21.
49. Liskov, B. H. et al, "Abstraction Mechanisms in CLU," Comm. ACM, Vol. 20, No. 8, Aug. 1977, pp 564-576.
50. M.I.T. Computation Center, *The Compatible Time-Sharing System - A Programmer's Guide*, MIT Press, Cambridge, MA, 1963.
51. Moore, Gordon E., "Progress in Digital Integrated Electronics," International Electronic Device Meeting, IEEE, 1974, pp 11-13.
52. Organick, E. I., *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, MA, 1972.
53. Organick, E. I., *Computer System Organization -- The B5700/B6700 Series*, Academic Press, N.Y., 1973.
54. Patil, S. S., "On Structured Digital Systems," Proceedings of International Symposium on Computer Hardware Design Languages and Their Applications, City University of New York, IEEE, Sept. 1975, pp 1-6.
55. Ramchandani, C., "Analysis of Asynchronous Concurrent System by Timed Petri Nets," Ph.D Thesis, Dept. of Electrical Engineering and Computer Science, MIT, Sept. 1973.
56. Randell, B., "System Structure for Software Fault Tolerance," IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, June 1975, pp 220-232.
57. Richards, H. Jr. and R. J. Zingg, "The Logical Structure of The Memory Resource in The SYMBOL-2R Computer," Symposium on High-Level-Language Computer Architecture, Univ. of Maryland, College Park, Maryland, ACM-IEEE, Nov. 1973, pp 1-10.
58. Roth, J. P., "Systematic Design of Automata," 1965 FJCC, AFIPS Conf. Proc., Vol. 27, part I, pp 1093-1100.
59. Rumbaugh, J. E., "A Parallel Asynchronous Computer Architecture for Data Flow Programs," TR-150, Project MAC, MIT, Cambridge, MA, May 1975.
60. Saltzer, J. H. and M. D. Schroeder, "The Protection of Information in Computer Systems," Proc. IEEE, Vol. 63, No. 9, Sept. 1975, pp 1278-1308.
61. Schlaeppi, H. P., "A Formal Language for Describing Machine Logic, Timing and Sequencing (LOTIS)," IEEE Trans. on Electronic Computers, Vol. EC-13, No. 4, Aug. 1964, pp 439-448.
62. Staehler, R. E. et al, "The 1A Processor," Bell System Technical Journal, Vol. 56, No. 2, Feb. 1977, pp 119-312.
63. Sumner, F. H., G. Haley, and E. C. Y. Chen, "The Central Control Unit of the 'Atlas' Computer," Proceedings of IFIP Congress, 1962, North Holland Publishing Co., pp 657-662.
64. Swan, R. J., S. H. Fuller, and D. P. Siewiorek, "Cm* - A Modular Multi-Microprocessor," 1977 NCC, AFIPS Conf. Proc., Vol. 46, pp 637-644.

65. Syre, J. C., D. Comte, and N. Hifdi, "Pipelining, Parallelism and Asynchronism in the LAU System," Proceedings of the International Conference on Parallel Processing, J. L. Baer, Ed., Aug. 1977, pp 87-92.
66. Thornton, J. E., *Design of a Computer: The Control Data 6600*, Scott, Foresman, Glenview, IL, 1970.
67. Unger, S. H., "A Computer Oriented Toward Spatial Problems," Proc. IRE, Vol. 46, No. 10, Oct. 1958, pp 1744-1750.
68. Watson, W. J. and H. M. Carr, "Operational Experiences with the TI Advanced Scientific Computer," 1974 NCC, AFIPS Conf. Proc., Vol. 43, pp 389-397.
69. Wilkes, M. V., "Slave Memories and Dynamic Storage Allocation," IEEE Trans. on Electronic Computers, Vol. EC-14, No. 2, Apr. 1965, pp 270-271.
70. Wulf, W. A. and C. G. Bell, "C.mmp - A Multi-Mini-Processor," 1972 FJCC, AFIPS Conf. Proc., Vol. 41, part II, pp 765-777.
71. Wulf, W. A., R. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs," IEEE Trans. on Software Engineering, Vol. SE-2, 1976, pp 253-264.