Summer 1980

Multiprocessor/Multicomputer Systems and Optimal Loading Techniques

Francis D. Adams

*University of Central Florida*
MULTIPROCESSOR/MULTICOMPUTER SYSTEMS AND
OPTIMAL LOADING TECHNIQUES

BY

FRANCIS (FRANK) D. ADAMS
B.S.E.E., University of Miami, 1974

RESEARCH REPORT

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
in the Graduate Studies Program of the College of Engineering
at the University of Central Florida; Orlando, Florida

Summer Quarter
1980
MULTIPROCESSOR/MULTICOMPUTER SYSTEMS AND
OPTIMAL LOADING TECHNIQUES

FRANCIS (FRANK) D. ADAMS

ABSTRACT

This report reviews the subject of multiprocessor/multicomputer systems and optimal loading techniques.

This report covers:

1. The interrelationship of Multiprocessor/Multicomputer (Multiple Instruction stream Multiple Data Stream, MIMD) systems and other architectures by presenting a categorization of computer architectures.

2. Comparison of Multiprocessor/Multicomputer (MIMD), versus Parallel Processor (Single Instruction stream Multiple Data stream, SIMD) systems.

3. Multiprocessor/Multicomputer problems, pitfalls and new goals.

4. Investigation of loading techniques by reviewing particular MIMD executive designs.

Approved by: Director of Research Report
# TABLE OF CONTENTS

## Part I. ARCHITECTURE

Categorization of Computer Architectures .......... 1
  Uniprocessor (SISD) and Pipelined Unit
    Processor (MISD) Performance .................. 2
  Parallel (Array and Associate Array) Processor
    (SIMD) Performance ........................... 6
  Multiprocessor/Multicomputer (MIMD) Performance .. 7
  Multiprocessor/Multicomputer (MIMD) versus Parallel
    Processor (SIMD) Systems ................. 11

## Part II. OPTIMAL LOADING TECHNIQUES FOR MULTIPROCESSOR/MULTICOMPUTER SYSTEMS .......... 13

  Multiprocessor/Multicomputer Problems, Pitfalls
    and New Goals. .................................. 13
  Problems ........................................... 13
    Overhead problems. ............................. 13
    Design restructuring problems ................. 14
  Pitfalls ............................................ 15
  New Goals .......................................... 16
  A Look Toward Optimal Loading Techniques .......... 18
    Operating Systems ................................ 21
      Operating system functions ..................... 21
    Firmware ......................................... 22
    Modularity ....................................... 24
    Design Approach ................................. 25
      General structure of the executive ............. 26
      The locator. .................................... 27
      The parts list .................................. 28
      Executive table structure ..................... 28
      Interpreter (processor) control ............... 29
      System loader .................................. 31
  Conclusion ........................................ 33

REFERENCES CITED ..................................... 34
PART I

ARCHITECTURE

Categorization of Computer Architectures

Flynn (1972) has described computer architectures by means of properties of data and procedure (instruction) streams. Flynn allows for single and multiple streams of both data and instructions. This has led to four categories of computer architectures. Some properties of different type processors can be derived by considering the concept of a stream. The four categories of computer architectures, based upon their stream properties are given in figure 1 (Thurber 1976). These are:

a. Single Instruction stream Single Data stream, SISD: unit processor

b. Multiple Instruction stream Single Data stream, MISD: Pipelined Unit Processor

c. Single Instruction stream Multiple Data stream, SIMD: Parallel Processor or Associative Processor, and

d. Multiple Instruction stream Multiple Data stream, MIMD: Multiprocessor or Multicomputer.

The performance features of the following figure 1 architectures are summarized, in-part from the work of Turn (1974), in the following paragraphs for comparison.
SD: Single Data Stream | MD: Multiple Data Stream
---|---
SI: Single Instruction Stream | Unit Processor | Parallel Processor and Associative Processor
MI: Multiple Instruction Stream | Pipeline Processor | Multiprocessor/Multicomputer

Fig. 1. Classification of generic processor architectures.

Uniprocessor (SISD) and Pipelined Unit Processor (MISD) Performance

The Single Instruction stream Single Data stream (SISD) and Multiple Instruction stream Single Data stream (MISD) categories of computer architecture are related in terms of the uniprocessor or unit processor and are combined here in summary.

The basic processor operation cyclically performs the following steps:

a. Fetches from memory and decodes an instruction.
b. Generates address of operand (if needed).
c. Fetches the operand from memory (if needed).
d. Executes the instruction.
e. Generates the address of the next instruction.

A major improvement in processor operation has been the introduction of concurrency into the basic computation cycle. This concurrency has been accomplished by special "look-ahead" circuits which simultaneously fetch several required instructions and data sets. Branching situations are handled by fetching the next instructions from all locations identified by the conditional branch. These concurrency techniques require increases in the effective memory-data-
transfer rates so that the required number of instructions and data words can be fetched while the immediate instruction is being executed. Hierarchical memory systems with fast block-transfer capabilities transfer the data/instructions into a set of interleaved fast-access memories. These fast-access memories can fetch and transfer several words simultaneously to the processor, thus, in effect, permitting the required data rates.

Computer process performance (i.e., computing speed) is determined by the effective memory-cycle time and the time to perform operations in the processor. An instruction execution time, $t_{ei}$, can involve $P+1$ memory-cycle times, $t_{mc}$, one to fetch the instruction and $P$ to fetch operands or store results, and $t_i$, the time to perform the operation in the processor for the $i^{th}$ category of instructions:

$$t_{ei} = (P+1)k_m t_{mc} + t_i ; P = 0, 1, 2, ...$$

The multiplier $k_m$ varies between 1 and 0 to indicate the degree of memory "transparency" (i.e., the look-ahead features on an interleaved memory in the control unit). Ideally, $k_m = 0$, then the memory is totally transparent to the processor. High-speed uniprocessors are designed in this direction.

The effect of Pipeline uniprocessor organization is to mask the memory accesses ($k_m = 0$) and to break the arithmetic operations into sequences of segments, permitting more concurrency in instruction execution, hence eliminating instruction fetching for long data streams.

With multiple decoded instructions and associated data in register stacks, multiple execution units become very attractive
(wherever the program structure allows concurrent operation). In the CDC 7600, for example, nine execution units (Bonseigneur 1969) can operate independently, each specialized for a particular task (e.g., fixed add, multiply, divide, shifting, normalizing, branching, and Boolean operations). With LSI technology it will become increasingly attractive not only to implement faster algorithms in hardware but also to possibly include other elementary functions (e.g., square root, logarithmic, trigonometric, and exponential functions) as separate execution units. The result will be a much faster execution of the algorithms as well as performance gains through concurrent instruction set up, whenever allowed by the program structure. However, this capability will be acquired at the expense of additional hardware units and more complex control units, which will be necessary for "farming out" instructions to the execution units.

Multiple, specialized execution units have a different use in the "pipeline" (MISD) structure. The computation sequence applied to a large number, N, of data words is segmented into the M tasks performed by M execution units, the pipeline segments. Ideally, the tasks are balanced so that each task requires approximately the same execution time, $t_s$. The pipeline is operated by passing the partially processed data words from one execution unit to the next every $t_s$ time interval. The time to obtain the first result (the latency time) is $Mt_s$ time units. Thereafter, however, the result is produced every $t_s$ time unit, representing a speed gain of a factor of M (i.e., each result via an SISD structure could require $Mt_{ei}$ time units). Also associated with the pipeline execution is a setup time, $T_o$, used
to structure the execution units for the immediate task. The time to process N words through the pipeline is $T_o + M t_s + (N-1) t_s$; hence the effective computing time per word, $T_e$, is

$$T_e = \frac{T_o + M t_s + (N-1) t_s}{N}$$

As $N$ increases, $T_e$ approaches $t_s$. For small values of $N$, however, the performance may be degraded.

The largest pipeline processor, CDC STAR-100, obtains a speed increase of a factor of 80 in an addition operation (memory-to-memory); e.g., a single addition requires 1.76 usec (including 1.28 usec memory-cycle time and 0.48 usec addition time), whereas pipeline additions require 0.04 usec per data word. The pipeline latency is 1.76 usec, and segment time $t_s = 0.04$ usec. The setup time, $T_o$, is 1-1.5 usec (Turn 1974).

Implementation of the execution and control unit can be simplified by using programable read-only memories (microprogramming) to store not only the constants required for the algorithms, but also the bit patterns for activating control signals. Benefits of microprogramming (firmware) are further discussed in Part II of this report.

Before continuing with the next category of computer architecture, it is important to note that uniprocessors of the category just discussed are typically what make up the individual processors of a Multicomputer system. So, indeed, we have just looked at a component part of MIMD systems.
Parallel (Array and Associative Array) Processor (SIMD) Performance

An array processor (parallel processor) is characterized by a Single Instruction stream and Multiple Data streams (SIMD). The principle features are:

a. A large array (typically 64 or more) of processing elements (PEs) controlled by a single control unit so that a single instruction controls the operation of all the PEs.

b. Private memory unit and addressing circuits for each PE. The PE memories can communicate with a large common memory.

c. Some control is available to the individual PE; i.e., the PE can be enabled or disabled during the execution of particular instructions, as determined by the results of local tests.

d. Memory addresses and data (common to all PEs) are broadcast from the central control.

e. Processing elements in the array with connections to their nearest neighbors for data transfer.

Effective use of an array processor requires that all processing elements operate simultaneously most of the time. If simultaneous operation is achieved, the speed gain of an array processor over that of a uniprocessor will approach N for N processing elements. More likely, however, only a fraction, $k(0<k<1)$, of the processor will be operating simultaneously. Given a particular application with a known average value of $k$, the array processor speed $S_{ap}(N)$ can be expressed as

$$S_{ap}(N) = kNS_{pe}$$
where $S_{pe}$ denotes the processing speed of the PEs. $S_{pe}$ has the same units, instructions per second, as $\frac{1}{t_{ei}}$ for the uniprocessors. The units of MIPS (Millions of Instructions Per Second) have been cited in system comparisons (Turn 1974 and Thurber 1976).

Performance comparisons with SISD or MISD can be made by comparing $S_{ap}$ with $\frac{1}{t_{ei}}$ or $\frac{1}{T_e}$, respectively. An SIMD array processor will surpass the computational rate (MIPS) of an SISD machine when $S_{pe} \approx \frac{1}{t_{ei}}$, since $kN$ is designed to be greater than one.

The PEs may range in complexity from a simple bit-serial uniprocessor to high-performance pipeline architecture. However, because of the large number of PEs used and the large speed gain achieved through array processor architecture, the PEs probably will not incorporate the look-ahead features that have provided considerable performance gain in uniprocessors. Hence there will be very little "transparency" of PE memory. Moreover, it is not likely that the memory itself will incorporate sophisticated architectural features (i.e., interleaving, buffering, multiword data paths).

An associative processor is actually a special array processor featuring an associative-memory in which every word in the associative memory has its own processing unit. The array-processing structure of the associative processor is potentially capable of providing greater speed than the uniprocessor architecture for special purpose applications.

Multiprocessor/Multicomputer (MIMD) Performance

Multiprocessors have multiple streams for both instructions
and data (MIMD): the processors typically share a common memory. Several modes of operation are possible, namely, the processing of (a) separate, independent computing tasks, (b) different parts of the same computing task, or (c) a mixture of tasks provided by the operating system running in a multiprogrammed mode.

The three principal advantages of multiprocessor architecture are:

a. Extended computing capability because of more than one processor in the system.

b. Flexibility in processing tasks with widely varying resource requirements.

c. Redundancy in processors, which increases system reliability and availability for critical tasks. If a processor fails, the system can continue with the remaining processor(s) in a computationally degraded form.

A multiprocessor system permits the processors to be "reconfigured" into highly reliable forms, using some processors as a redundant backup to processors performing critical tasks; during less critical computations all processors may perform distinct computing tasks. One processor in a multiprocessor system can function as the master controller. It contains the operating systems for allocating and scheduling computing tasks and memory resources to the remaining processors. Other multiprocessors operating concepts permit each processor to become the master controller as it completes its previous task and then to assign itself a new task. We will look closer at these concepts in Part II of this report.
Efficient utilization of the multiprocessor system requires a steady stream of tasks to the processors. The processor(s) that operates as master controller is thus largely occupied with system overhead task of allocating, scheduling, processing interrupts and assigning priorities. This overhead increases with $N$, the number of processors, until the entire activity of the controlling processor(s) is devoted to control tasks. The performance of the multiprocessor system could be expressed as:

$$S_{mp}(N) = N(1-k_N)S_P$$

where $k_N$ represents the fraction of the control-processor(s) activity devoted to system overhead tasks, and $S_P$ is the speed of the component processors. The dependence of $k_N$ on $N$ is difficult to assess, $k_2 = 0.3$ and $k_4 = 0.5$ have been used (Turn 1974) to forecast projected performance with the above expression. However, these forecasts have not been validated.

A multicomputer system (referred to as a "federated" or "polymorphic" computer system) is also categorized as a Multiple Instruction stream, Multiple Data stream (MIMD) system. The individual processors in the system are, typically, complete uniprocessors, as discussed earlier in this report, with their own independent memory and I/O systems. Communications between the processors take place via communication links (data buses) or through a common data base in a mass-memory system.

The component processors of a multicomputer system normally process independent tasks. However, they may share tasks when these
tasks exceed the capabilities of individual processors. They may also take over critical tasks of one of the processors in the event of a failure in order to increase the viability of the system.

The advent of minicomputers has focused considerable attention on multicomputer systems. Various processing and control tasks can be assigned to dedicated minicomputers.

The processors in a multicomputer system may be independent units with their own particular processing rates. The performance of the entire system is of interest when the processors participate in common or related computations. There will be a certain amount of degradation in performance (even beyond that of multiprocessor systems) because the interaction takes place through more remote means—mass memory or communication system. The degree of degradation is highly dependent on the specifics of the inter-communication system and the nature of the computing tasks. This degradation can be identified with a multiplicative factor, \( k_i \), having a value less than 1, such that the gross speed of the multicomputer system, \( S_{mc}(N) \), is given by:

\[
S_{mc}(N) = \sum_{i=1}^{N} k_i S_i
\]

where \( S_i \) is the speed of an individual processor of the multicomputer system. (Many different component processors may be part of the system). The individual processors can be assumed to be uniprocessors of the type described earlier.

\( S_p \) and \( S_i \) of the preceding expressions for multiprocessor/multicomputer (MIMD) systems have units of instructions per second. Performance comparisons with SISD, MISD and SIMD can be made by
comparing $S_{mp} (N)$ or $S_{mc} (N)$ with $\frac{1}{E_{ei}}$, $\frac{1}{T_{e}}$ or $S_{ap} (N)$, respectively.

A computer network such as the ARPANET (Turn 1974) is a multiprocessor system with processors of various architectures at remote locations (including transcontinental distances). The processors are time-shared by users at each processor location as well as anywhere else in the network. Accordingly, the specific processing capabilities and data bases at all locations are available to all users. ARPANET consists of about 25 nodes, each with its own computer and communications interface processors (IMPs). The ILLIAC IV, (McIntyre 1970) array processor is one node of this network.

**Multiprocessor/Multicomputer (MIMD) Versus Parallel Processor (SIMD) Systems**

In an attempt to alleviate possible confusion between multiprocessor/multicomputer (MIMD) systems and parallel processors (SIMD) the following discussions are presented.

To date uniprocessors have been the cheapest problem solution in most environments. However, there are high performance environments in which they fail. In some cases, general purpose computers are used in a multiprocessor/multicomputer (MIMD) mode. It may be difficult to go to a general MIMD mode because of the attendant executive overhead and memory access conflicts. Because of the unsuitability of many conventional machines, special purpose parallel processor (SIMD) computers (designed with the problem requirements in mind) are often used. So, the key distinction is in application; and parallel processors are special purpose rather than general purpose. Special purpose machines by definition are useless outside
of a range of problems, but there are special purpose application environments in which they are workable (Thurber 1976).

In cases of general purpose application where the processors are used in a MIMD mode, an additional distinction is that the processors have to intercommunicate, or the system is not a MIMD system. The sequel will examine some problems with MIMD systems.

First, there is the problem of communications between processors, especially those of a computer network (remote processors). This is a significant element contributing to multicomputer performance. However, this in itself is an area in which a study towards optimal techniques could be endless, therefore other than this mention, the problem associated with communications will not be considered in this report. For further details in this area, the works of Chen (1974), Thurber (1974) and Akkoyunlu, Bernstein and Shautz (1974) are ideal sources. An interesting observation is that as of 1970, solving this problem optimally was unachievable (Howard 1970).

Other problems associated with MIMD systems are enumerated in the following section.
PART II

OPTIMAL LOADING TECHNIQUES FOR MULTIPROCESSOR/MULTICOMPUTER SYSTEMS

Multiprocessor/Multicomputer Problems, Pitfalls and New Goals

Problems

Problems occur as we expand a computer system by adding processors in an MIMD configuration. These problems can be generalized into two problem areas. These are the problems of overhead and the problems of design restructuring.

Overhead Problems

In a general environment overhead causes the addition of one processor resource to yield less than one processor unit of throughput. This phenomena is due to scheduling overhead, memory access conflict, and I/O conflicts. To think that it only requires a few percent of the capability of a CPU for overhead functions when another CPU is added to a multiprocessor/multicomputer configuration is unrealistic (Thurber 1976). Because the multiprocessor system is based on sharing tasks, memory and I/O, conflicts are to be expected. Partial alleviation of contention problems can be accomplished by providing each processor with a buffer memory and by giving the controlling processor the task of keeping the buffers loaded. The individual processors can use a certain amount of look-ahead within their buffers and can request instructions and data not in the buffer. A closer look at means of alleviating these problems is included in Part II, further
Design Restructuring Problems

Assuming that the processor conflict problem can be solved, the designer needs to consider other possible disadvantages of multiprocessor/multicomputer systems; e.g., incremental cost of augmentation. To add a CPU to a system could cost a large number of dollars compared to adding another simple processing element (PE) to an array (parallel) processor (Thurber 1976). There are many possible "hidden" costs to add one processor, such as complete restructuring of the application programs, redistributing them over the set of processors and the design cost of the new or modified Operating System to handle the new configuration.

Effective MIMD systems requires minimizing the above problems. Hence, the loading techniques for multiprocessor/multicomputer (MIMD) systems which minimize the above problems are defined as Optimal.

My initial research effort in this area was to contact all major computer manufacturers in an attempt to acquire information on available optimal loaders or loading techniques which have been implemented for use with the multiprocessor/multicomputer configurations offered by these manufacturers. The result of this correspondence has uncovered that no optimal loaders/techniques are currently available and that a user must currently implement his own loading techniques. Therefore, since no information was acquired on existing available optimal loaders and loading techniques, my study turned to what research has been done and reported in this area.

This study included an exhaustive review of literature for
those papers and reports which have formed the bulk of the reported research and results to date. This task was unexpectedly difficult since few papers and reports directly addressing this topic were uncovered. This literature review included a number of automated literature searches, several major libraries and the National Technical Information Services. The results are included in the list of references of this report.

Pitfalls

Although it is not surprising that multiprocessors/multi-computers have not been used except on a highly specialized basis, it is depressing. The thought that some attempts to use MIMD configurations may have failed due to the lack of industry to provide adequate techniques for the systems they manufacture is even more depressing.

Pitfalls or reasons why multiprocessors/multi-computers (and implementation of optimal loading techniques) have not materialized as uncovered by this study from Bell and Strecker (1976) may be:

a. The basic nature of engineering is to be conservative. This is a classical deadlock situation: We cannot learn how to program multiprocessors until such systems exist; a system cannot be built before programs are ready.

b. The market doesn't demand them. Another deadlock: how can the market demand them, since the market doesn't even know that such a structure could exist?

c. We can always build a better single, special processor. This design philosophy stems from local optimization of the designed
object, and ignores global costs of spares, training, reliability and the ability of the user to dynamically adjust a configuration to his load.

d. There are more available designs for new processors than we can build already.

e. Planning and technology are asynchronous. Within the computer industry, not all products are planned and built at a particular time, hence, it is difficult to get the one right time when a multiprocessor would be better than an existing Uniprocessor together with one or two additional new processors.

f. Incremental market demands require specific new machines. By having more products, a company can better track competitors by specific unprocessors.

Which boils down to the basic theory of supply and demand, that is, at present there is no real demand for multiprocessor/multicomputer systems implementation in a broad enough scope to necessitate their supply with total software which includes such things as optimal loading techniques.

New Goals

The evolution of computer systems is driven by two technologies: hardware and software. But there is a fundamental conflict: whereas hardware advances reduce costs of newer machine configurations, software improvements are generally more effective on existing machine configurations (which tend to overlook MIMD configurations).

Formal studies of computer architecture (Dietz and Szewerenko 1979) attempt to resolve this fundamental conflict. The idea is to
provide interfaces in a computer system that foster compatibility between these technologies, which naturally evolve at different rates. Thus, system developers can reimplement only part of the total computer system at a time (incremental reimplemention). With this current concept "computer architecture" means "the structure of the computer system a programmer needs to know in order to write any time independent, machine language program that will run correctly on the computer system." By this definition, two computer systems with the same "architecture" will run the same software. This concept implies a new machine can take advantage of the most recent hardware technology and still use the software base of an earlier machine. The goal then of such implementation is to provide a faster cheaper machine that runs the old available software without change.

Attempts toward this current goal (Burr and Gordon 1977) via reimplentations of an architecture usually invalidate much of the software. However, there have been cases where reimplementation of architectures (specifically the military-qualified Norden PD P-11/34M) has captured all of the software operating system and diagnostic software, but MIMD configurations and the problems associated with MIMD were overlooked.

During proceedings of the symposium on Computer Architecture, Bell and Strecker (1976) of Digital Equipment Corporation expressed a hope that convincing arguments will be forthcoming about the effectiveness of multiprocessors in order to establish these structures on an applied basis.
One argument is as follows. Based on the performance expressions for the various computer architectures categorized in Part I, MIMD structures can and will prove their effectiveness, however, current performance goals overlook MIMD configurations and associated problems. For example, performance goals for military computer systems (Burr et al. 1979) are influenced by requirements and available technology, more so by available technology (i.e., the old available software). It's time to let requirements lead the way influencing such performance goals, and this may very well be done only by giving up the "force fit" of available software at the machine level.

Even though any new commercial architectural entries must take into account the financing or acquisition of their software base to become competitive in the marketplace, higher level and application software should be the principal objective of compatibility.

Performance goals for such systems as a Military Computer Family (Stone 1979) should require MIMD configurations. This may not be possible using an available software base. It's time to move to a new foundation for meeting such requirements. This foundation being new concepts in the executive design. The reports referenced in the following section of this report are ideal examples in this direction. An executive design for multiprocessor (MIMD) systems should include the goals delineated in the following section.

**A Look Toward Optimal Loading Techniques**

Loading techniques for multiprocessor/multicomputer systems are dependent on the executive designs for the particular systems.
After our brief look at the problems in implementing a multiprocessor/multicomputer architecture, it should be apparent that the majority of these problems can only be solved by a strong foundation in the executive design. It is the intent of this section of this report to provide a look toward optimal loading techniques by looking at particular proposed multiprocessor (MIMD) executive designs. Only by practice and viewing the attempts of others can an individual develop his own techniques.

Primary goals for executive structures per studies done for the U.S. Air Force from Zucker (1972) and Kilbride, Iwasa and Scheid (1972) are as follows:

a. The executive must be designed to operate within the framework of the multiprocessor configuration shown in figure 2. The design must not presume specific processors or other hardware devices, nor may it presume that all processors are identical in their characteristics. The switch matrix may either afford complete interconnection among processors, memories and other subsystems, or not. This last requirement arises from the observation that, even in a configuration which affords each processor access to every memory module and subsystem, hardware failures may disable one or more access paths between processor and device.

b. The executive is required to manage the assignment of processors to simultaneous operation on the total data processing workload in such a manner as to insure maximal utilization of processors and other resources while maintaining appropriate work sequencing, (i.e., the most urgent portions of the workload are process-
Fig. 2. Multiprocessor configuration
ed first, whenever possible). The executive must also provide protection against mutual interference among the operations performed in the several processors.

c. System tasks such as the system loader and the communication task are required of the executive to initiate the system and to allow for system communication with the external environment.

These goals address all of the problems associated with MIMD systems noted earlier in this report and a look at techniques for obtaining these goals certainly encompasses a look toward optimal loading techniques.

Operating Systems

An operating system is the computer system software that assists hardware in implementing various executive (supervisory and control) functions to aid in the execution of user tasks. Hence, executive is the generic name for the collection of procedures included in the operating system that provide the basic control and monitor functions.

Operating System Functions

An operating system is required to perform eight important functions for the users and for the system:

1. Creating and removing tasks
2. Controlling progress of a task
3. Recognizing, responding to and reporting failures of errors.
4. Allocating hardware resources among tasks
5. Providing access and links to software resources
6. Providing for intertask communication
7. Allowing for sharing of code and data between tasks
8. Allowing for multiprocessing efficiency where the work is distributed over a number of processors.

Firmware

A computer traditionally interprets and executes a sequence of machine codes that is its order set and the only code it recognizes. When each individual operation is performed, transfers of information occur among functional components (e.g., registers, memory, adder, etc.) of the computer. The communication between functional components is caused by the set of primitive machine operations called micro-operations that are ways of opening and closing gates and circuits between registers and the basic logic elements within the computer. In conventional computers, the order set of the machine is a totally defined code interpreted by a wired-in set of circuits within the control unit of the computer. Thus, a large, inflexible and complex control unit is developed.

Microprogramming is an orderly way of designing control sequences to execute machine instructions that uses programming techniques such as the sharing of common sequences among different machine instructions (subroutines) to provide simplicity as well as flexibility.

The microprogrammable computer allows for a soft control of micro-operations. A microinstruction specifies one or more micro-operations that could be performed in a fixed time interval. The microinstruction is stored in a fast memory called a microprogram
memory. The microprogram is a set of microinstructions used to interpret machine code.

Therefore, one avenue to obtain a strong foundation in the executive design of a multiprocessor configuration is an approach in which the processors are microprogrammable. They have a flexible control unit and they are specialized by replaceable microprograms for the various roles they must perform. Firmware is the word used for microprograms that will reside within a control memory of the processor's control unit. Hence, firmware specializes the instruction design for a specific purpose.

Microprogramming has been used primarily for emulation. Emulation is a firmware interpretation of machine instructions and data structures of one machine by another. A major expense of replacing one computing system with another arises in the rewriting of existing programs due to differences between systems. Since the cost involved in conversion is great, emulation saves money and reduces difficulties. A customer can have his current machine emulated so his old code can be run. He can also use the new more general computer to run new programs of varies structures and specific applications, written to exploit its features.

However, firmware can go beyond emulation. It may be used to create tailored instructions for special programs. If a computer is used consistently to do searches but has no search instructions, a search instruction should be built into the firmware where a software subroutine call may have previously existed. When new devices are added, new firmware accommodations may be supplied (i.e., error recov-
ery procedures, formatting, or enhanced interrupt capabilities). Conversely, instructions not used by the system may be removed or replaced by more useful ones.

Implementors of control programs and operating systems can develop their own order sets and data structures in firmware. Built-in queue management, memory allocation, table referencing, sorting and pointer handling are part of the basic needs of the executive system. Thus, special firmware can be developed for efficiency, where efficiency is needed.

Modularity

The Air Force Avionics Laboratory (AFAL) Wright-Patterson Air Force Base, Ohio has contracted two independent companies to develop proposed executive structures with the preceding goals. Before looking at some of the specifics of these design approaches, a key consideration common in these design approaches is a modularization concept oriented toward tailorability.

Modularity is a building block technique which is a natural and necessary extension of the modular hardware architecture. Such architecture provides a method for accommodating changing computing activities as well as new configurations of hardware modules.

An engineer designing a large system will not originally design every component that is used in his system. He will use subsystems and components previously designed by others, and only logical interconnections must originate from him. If the components and subsystems he uses are designed to interface with each other, then the overall design will be easier. When interfaces between components
are needed, they may be included in the overall design.

Similarly, the software designer should not have to design every system as if he lives in a vacuum. Each new system should not be started as if no similar system had ever been designed. Components should be used from a running software system to develop another system which may be similar to it in some areas of design. Software modularity allows for the use of components of software for developing a program.

The system software is divided into functional modules that can be linked into a system after each module has been individually validated. All processors of the system are independent of each other and may select any executive module or any application task for execution.

The result is a set of tailored hardware and software modules that may be appropriately interconnected to form a variety of systems. The concept of modularity will help reduce the cost of software development and the time required to write and test all programs.

A brief description of some of the executive modules for a proposed executive structure follows. Modules of the executive include such functions as scheduling tasks, resource allocation, error recovery, error detection, reconfiguration and file handling. These may be selected as needed using a set of selection parameters, from a library of functions, at mission (system requirement) definition time.

Design Approach

This section of this report summarizes the results of a study
done by the Advanced Development Organization of Burroughs Defense, Space and Special Systems Group for the Air Force Systems Command, Wright-Patterson Air Force Base (Zucker 1972). This work provides a description of the executive structure to accomplish Multiprocessor executive functions in an aerospace environment. The software approach avoids the conventional difficulties that normally occur in the development of large monolithic operating systems. The resulting system is flexible, since it must accommodate both broad computing activities and varying computing hardware.

The result of the study is a description of an executive structure, together with a set of executive modules, that may be tailored to perform the requested executive functions for a specified multiprocessor configuration and for a given set of applications.

This approach to the development of the multiprocessor executive program is a building block technique. The operating system is a set of modules any one of which may be obtained and run by any processor whenever it is needed. These modules are basically a set of independent "S" instructions which may be used by any task on any defined configuration of firmware and hardware. This method of system development produces a distributed executive that is both simple and flexible in structure.

General Structure of the Executive

The three sectors making up the Burrough's executive are the Locator, the parts list, and the modules. The modules are both programs and data (tables) normally thought of as the operating system. Each module is selected for use through the Locator. The Locator is
microcode that is included in all the microprogrammable processors. Each module is selected through the Locator using a program parameter that will identify the desired module. The Locator, using the parts list, locates the module requested and then executes it.

The parts list is a table of all modules associated with an individual mission. Each entry contains the location of each module. A bit is used to lock the module when the need arises. (No two processors should write the same module into core memory at the same time, but once in core any number of processors may use it.)

The Locator

The microcode common to all parts of the system must include the Locator and Allocator. This is the section of microcode that allows a task access to the system modules available in the parts list.

When a task is assembled, a table is developed for it that includes the names of all the executive modules needed by it. This becomes the source table for this task. Each programmed task must have such a source table.

Each time an executive module is to be accessed, the entry in the source table corresponding to the desired module is requested. The combined source tables for every program run on the system are used to develop the system parts list for a specific mission.

Each source table entry becomes an index into the parts list as well as a pointer back to the processor's memory. The address in memory initially is the address of the Allocator. When an executive module is referenced via the source table, control is transferred to
the Allocator. The Allocator finds space for the module, then changes
the address in the source table entry to point to this space. When
the module has been placed, it may then be executed. Once a module
has been placed, it remains there until overlayed with another module
by the Allocator. When a module is referenced, the source table points
directly to the address of the module.

The Locator must save the position of the task instruction
calling for the module. It can do this in the part of the work area
reserved for this purpose. The Locator then uses the index supplied
by the task to select the proper entry in the source table. The source
table entry is used either to go to the Allocator to find the module
or to locate and execute the required module. Before executing the
required function, the last used counter is updated. The global last
used counter (LUC) updates a count by one each time an executive module
is used. The latest count is kept with the source table entry for
the module last executed. When the memory must be overlayed, the last
used executive modules may be distinguished from those not used recent-
ly that are better candidates for overlay. When the executive module
has been completed, the earlier task is continued from the point of
the executive call.

The Parts List

The parts list is a set of entries that locate and describe
the modules of the executive. When a module is not present in the
processor's memory, it must be allocated space there and copied from
main memory using the parts list.

Executive Table Structure
The concept of executive construction for this approach developed into a "table driven" executive. All modules of the system executive for such a Multiprocessor will be independent of each other. The parts binding the system together will be the system tables. All information about system status, system and job configuration, task status, task characteristics, etc. is encoded into a series of tables. Each interpreter (processor) picks up a table entry to determine its next course of action. Table updating is done with care so that two interpreters do not try to update the same entry at the same time.

Switching among tasks and components is directed by the tables' contents. A common set of executive tables for all interpreters forces cooperation among interpreters.

**Interpreter (Processor) Control**

In this modular multiprocessing system, each interpreter schedules itself and performs its own executive functions. A single interpreter may be dedicated to a particular problem for the duration of a mission. However, any interpreter may select this dedicated task for as long as it needs to be run. The scheduling and other control modules as well as the system tables reside in main memory, and any interpreter may use them to schedule itself.

Any interpreter may perform any function by simply loading the function module. Whenever an interpreter is available, it searches the task table for a ready-to-run task. This task may be an I/O task, a processing task, or a task that combines both processing and I/O functions.

In previous systems, input output functions have been perform-
ed in hardware modules different from the data processing functions. The I/O and the central processor modules were separate, dedicated hardware devices. In this multiprocessor system, I/O control and the data processing functions are all performed by interpreters.

When an interpreter is started, it gets the scheduling module and uses it to select a task that is ready for processing. The task entry selected is checked to be sure that all the resources it needs are available to it. When the task is ready, it is read into the memory of the interpreter; and the task is begun. During the running of a task, the interpreter must report the time periodically to the interpreter table in order to inform the system that neither the interpreter nor the task running is malfunctioning. The task being run may also use executive modules to achieve common functions such as I/O operations, bounds checking, sin/cos calculations, etc. When a task is completed or its must stop because of a new requirement presently unavailable, the end task function is used. This function must deallocate resources and update the system tables for a completed task. If a task is to be suspended for awhile, the state of the task and its resources will be preserved in its work area so that any interpreter may resume running the task when it becomes ready to run again.

The executive module that is used after a task has been ended checks the interpreter table. This is the module that is used to keep tabs on the system. Each interpreter checks to determine that all other interpreters have been obediently checking into the table. If all is well, the interpreter can go on to schedule the next available
task in the task table. However, if an interpreter has not reported in as it was supposed to do, the checking interpreter must report the fact to the operator and rescue the task from the bad interpreter. The task is then backed up to a restart point, the task table is updated to reflect the tasks ready-to-run status, and the interpreter returns to the scheduling module.

**System Loader**

There are some functions of an executive system that cannot be classified as modules, i.e. any executive functions that are not available to an interpreter (processor) via the common Locator module. These are dedicated system functions that must be performed independently of the tasks and the interpreters in the system. These tasks are initiated by the scheduling module when needed, and some are run on a cyclic or continuous basis (e.g., communicator and I/O process). These executive system tasks include a communicator, utility packages and a system loader. The system task of primary interest here is the system loader.

A system generation program provides a means for describing the machine configurations and system features desired for the needs of a specific mission. The output generated is a tape, disk or card file containing a directory, the parts list, modules and system tasks. At system start-up time, a single interpreter must be chosen to bootstrap a loader into microprogram memory and then to load the executive into the system. When the tables have been initialized, the other interpreters in the system can select a task for running, using the scheduling module.
The loader must either read the executive from a given storage device into the mass storage area of the system or into core memory. The executive loader must be informed if a cold start, cool start, or warm start is necessary. A cold start assumes that the mass storage directory has been destroyed and a new one must be initiated. A cool start assumes that the directory is intact and available for use. A warm start indicates that the executive in mass storage as well as the directory is intact. Only the core memory must be initialized.

After the parts of the executive that must be in core are put into core, (parts list, tables, and schedules), the initializer is used to update, organize and initialize the system. The initializer performs the following functions:

a. Updates and initializes tables
b. Organizes and classifies core storage
c. Checks and sets I/O status
d. Initiates system tasks (e.g., communicator, I/O pack, confidence programs, and diagnostics).

When the system is ready to start running, all interpreters must be informed of this status. The bootstrap interpreter broadcasts an interrupt for all interpreters. When the interpreters assigned to user tasks are made aware that the system is ready for use, they load the scheduler into their memories and start searching for a task to perform.

The first system task will be the communicator. Through this task, information will be received from external sources that will
initiate the user tasks in the interpreters.

**Conclusions**

The approach summarized in the preceding sections has executive design goals in common with an approach developed in parallel for the Air Force by System Development Corporation (Kilbride, Iwasa and Scheid 1972) and yet the proposed implementations concentrate on different areas. Some of these differences include:

a. Burrough's approach includes a firmware concept in which the processors (interpreters) are microprogrammable. All modules of the system executive are independent and the parts binding the system together will be the system tables.

b. System Development Corporation's Approach concentrates on communicating between application tasks and between executive modules not residing in the same processor. A special executive module, the Monitor, routes messages and transforms them into control table settings.

The approaches are referenced to amplify the existence of the problems associated with MIMD systems rather than to present firm solutions. These approaches, although never totally implemented, illustrate that the development of such new concepts in executive design can be the foundation for minimizing the overhead and design restructuring problems, as we look toward optimal loading techniques for MIMD systems.
REFERENCES CITED


