MICROPROCESSOR-BASED TRANSITION MACHINES

Russell F. Vaughan and Mark S. Anastas

BOEING AEROSPACE COMPANY
P.O. Box 3999
Seattle, Washington 98124

The general objective of the approach recommended in this paper is to reduce the cost of computer systems which require more throughput than can be afforded by single microprocessors. It proposes the construction of systems of multiple microprocessors operating in a "tightly coupled" mode. The approach has been shown to be potentially applicable to systems of up to about 100 microprocessors operating in parallel. The specific characteristics to be described include: A system architecture which supports the separation of control and computation; a separate programmable control device which effectively reduces the multitasking overhead; and a configuration of multi-microprocessors which reduces memory contention.

Introduction

Microprocessors can be shown to provide a basis for a standard building block approach for tightly coupled as well as loosely coupled ADP systems. The more difficult programs associated with the former systems have been shown to derive primarily from the extensive software control overhead required to coordinate such tightly coupled systems.[1,3,18] The computer architecture described in this paper addresses these problems.

The architecture involves two distinct aspects: Data transformations and program control. The control aspect determines which data transformations are to be performed based upon the current state of the system reflected in a system status vector. The control aspect has been allocated to a programmable hardware device called a "System Controller" which provides the equivalent of the multiprocessor executive functions with an overhead which is orders of magnitude less than for a conventional executive. The individual data transformations are performed by application programs executing in microprocessors. The configuration is shown in Figure 1.

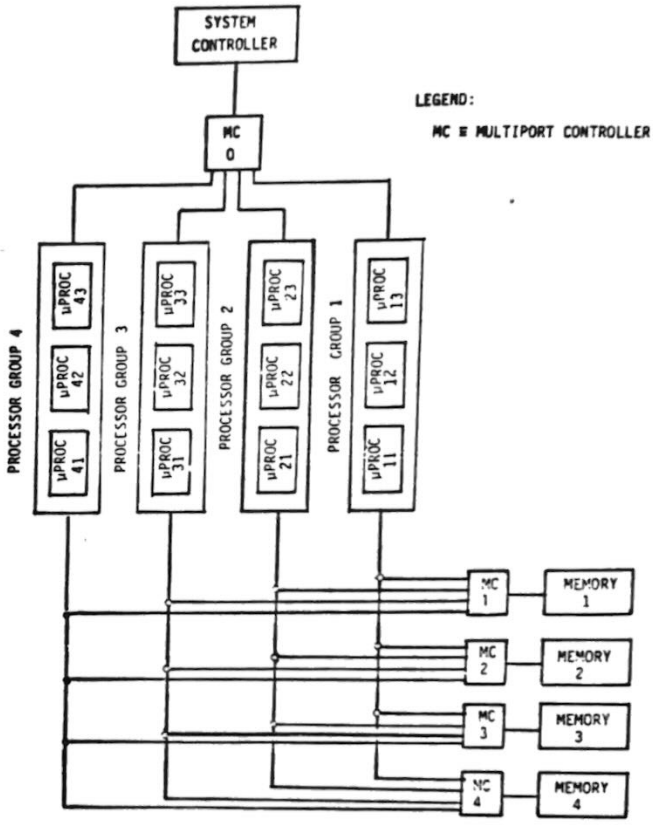

Figure 1: Microprocessors Configured As A Transition Machine

There are four major areas to be described:

1. The abstract model of computer architecture,

2. The System Controller and processor interface operation,

3. The multiprocessor configuration design approach (processor grouping, global memory and bus organization), and

4. The microprocessor firmware specification.

Computer Architecture Model

Multiprocessor Architecture

For tightly coupled multiprocessor systems, common task control queues maintained by the operating system require lockout protection against simultaneous access by more than a single processor. The overhead induced by this queue lockout cannot be bounded in the same sense as

CH1465-4/79/0000-0327$00.75

memory contention.[3,16] Various queueing approaches with their implied critical section lockout characteristics were accommodated by a model described in reference 3 (and summarized further on) to determine best approaches to reducing overhead from this source. It was shown that to obtain a significant improvement, the task queueing and scheduling overhead must be significantly reduced. The problem of task control in the multiprocessor was therefore intensely studied. A new approach consisting of a hardware device which performs the function of a traditional multiprocessor executive was developed. The advantage of this approach is an extreme reduction in overhead. Results of this study are to be described, with design details provided in reference 4.

Hardwiring an executive program is not in general a feasible approach although it has been done for specific systems.[6] It is the dynamic request basis of executive programs that reduces feasibility in the general case for large systems. To avoid these problems, an abstract model of computation has been used as the basis for a hardware/software architecture for which requests are not required. This abstract model is called "Named Transition Systems".[13] The architecture of the machines derived from it are called "Transition Machines".

Application Program Structure

The application program structure which runs on Transition Machines has been shown to exhibit considerable software development and maintenance advantages over conventional structures.[2] The structure involves a single construct for every software procedure, characterized as follows:

when (a set of conditions are met),
do (a data transformation)

The statement that "a set of conditions are met" is an assertion of propositions on the system data base which define the appropriateness of the data transformation. The types of propositions on the data base that can effect the eligibility of application programs are:

1. A data element is available/not available for use in subsequent computations,

2. A data element satisfies/does-not-satisfy a specified relation to some constant or other data element (for example: a < b), and

3. A data element can/cannot be updated.

Maintaining the status of these conditions and performing the operations required to determine the eligibility of individual programs based on it have been allocated to a separate control device in Transition Machines.

The "data transformation" can be implemented as a sequential set of programmed operations on data, performed by microprocessors sharing a common memory.

It can be shown that any program can be written (or alternatively translated) into this When Block format. This is based upon the proof of Böhm and Jacopini[8] that any program can be written as a "structured" program.

Abstract Architecture Model

Figure 2 shows a general characterization of this architecture; it is comprised of two major components as was the computation structure described above. The control component maintains

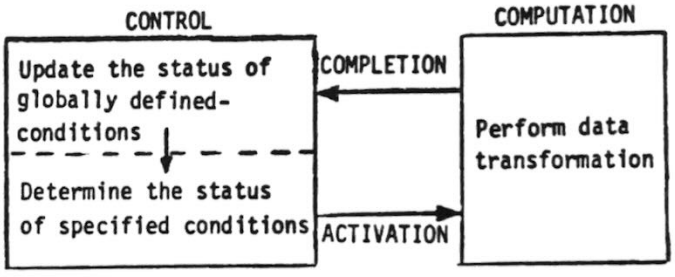

Figure 2: Components of Transition Machine Architecture

status indications for all of the relevant data base conditions in the system. It also contains indicators associated with each application program specifying the subset of the global set of conditions required to enable the specific program and indicators specifying the modification to the global set of conditions implied on completion of the program. The computation component executes the code associated with the data transformation aspect of each program, and returns data-value-dependent status conditions to the control component.

The operation of these two components is as follows: The control component first determines an eligible application program by examining the current status of the global set of conditions specified by each requirements indicator associated with the program. An eligible subsystem is identified to the computation component which then executes the specified sequential arithmetic operations associated with the application program and returns status indications specifying the conditions that have been modified dynamically by the program. The control component updates the global condition status indications associated with having completed the program. The cycle is then repeated until the system runs to completion.

The dynamic status update is a requirement imposed by having no data base operations performed directly by the control component as part of the eligibility determination. The control component must therefore separately maintain status indications of relevant conditions current in the data base. This is accommodated by incorporating an update to the global set of condition status indications on completion of each program. There are several possible modifications to the condition status indications that may be implied on completion of an application program. They are as follows:

1. The condition remains unaffected by the program running to completion.

2. The condition is satisfied whenever the program runs to completion.

3. The condition is negated whenever the program runs to completion, and

4. The condition status is determined dynamically by the execution of the program.

The first three updates are of an implicit nature, i.e., on completion of the associated application program the status of each of the conditions can be statically determined independent of the data values. The fourth update must be returned by the computation component as the result of an explicit operation on data at completion of the data transformation. This allows incorporation of condition updates determined dynamically during the data transformation.

Control Component Design

To implement the control component as a device, referred to as the System Controller, a set of data constructs has been defined on which logic operations can be performed to efficiently implement the required control functions. These constructs include the following:

S = A single globally defined system status vector which contains one binary status indicator for each data base condition which is relevant to the eligibility of at least one application program.

R = A relevance vector for each program which contains one binary status indicator for each data base condition maintained in the system status vector, indicating whether or not the condition is required to enable the associated program. (The set of globally defined conditions maintained by the System Controller is defined as the union of all conditions required to enable the individual application programs which comprise the system.)

E = An eligibility vector which contains one binary status indicator for each application program, representing the eligibility or non-eligibility of the program.

Figure 3 shows the dimensional relationships of these constructs. The R vectors are arranged

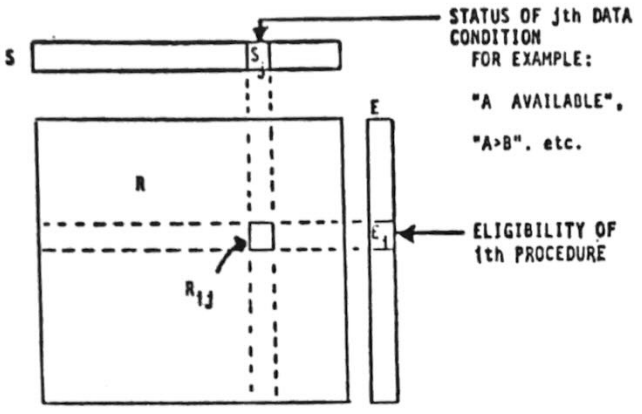

Figure 3: Dimensional Relationships of S, E, and R

in a matrix form as shown and the E vector can then be generated by forming the logical "dot" product of the R matrix and the S vector:

$$E = R \cdot S$$

The details of this and other operations are described elsewhere.[4] It is this basic matrix operation that the System Controller performs to determine the eligibility of application programs.

To incorporate the update to the S vector on completion of a program, the following additional constructs are defined:

U = An update vector of fixed indications for each application program, designating the modification implied to each data base condition at completion of the program. Elements of this vector take on one of four possible update values: set true, set false, not changed, or determined variably during program execution.

D = A single variable (dynamically updated) vector returned by the computation component to the System Controller on completion of a program. This vector contains one binary status indicator for each data base condition indicating the status of conditions which are to be determined dynamically during the execution of the program.

The U vectors can be arranged in a matrix form, where each row of the matrix is associated with an application program and each column is associated with a data condition. As described previously, elements of this matrix have one of four possible values. Two binary indicators (T and F) for each element (or equivalently, two separate matrices) are therefore sufficient to support this representation. Table I shows one possible sense assignment for T and F. The updated status vector can be computed as a fixed

TABLE I: Update Matrix Definitions

T_j	F_j	$S_{j\ NEW}$	IMPLIED MODIFICATION TO S_j
0	0	V_j	SET VARIABLY TRUE OR FALSE
0	1	0	SET FALSE
1	0	1	SET TRUE
1	1	$S_{j\ OLD}$	UNCHANGED

$$S_{j\ NEW} = (T_j \wedge \bar{F}_j) \vee (\bar{F}_j \wedge V_j) \vee (T_j \wedge S_{j\,OLD})$$

logical expression of the current S vector, and the T, F, and D vectors associated with the completed program. By defining the sense of T, F, and the logical update expression appropriately, a mask against unauthorized dynamic changes to S through the D register has been provided.

Finally, to dispatch an application program (the data transformation aspect) to the computation component, three additional constructs associated with each application program are included in the System Controller. These are "read", "write", and "execute" descriptors which allow/restrict the computation component to perform only the specified data transformation on the specified data items. These values are stored in special task interface registers in the processor when the program is assigned to the processor. A multiport controller provides the synchronization required for interfacing the task interface registers of multiple processors to the System Controller.

The overall operational flow of the System Controller in conjunction with the computation component is shown in Figure 4.

Multi-Microprocessor Configuration

To support the development of a multiprocessor configuration of microprocessors, analytical work was performed to aid in defining optimal configurations. A general parameterized model of the major overhead contributions was developed. A processor utilization model and a memory contention model were used to study and determine preferred methods to reduce the effects of processor utilization, memory contention, multitasking control, and processor software lockout associated with multitasking. The relative affects of control program execution overhead, control program lockout and memory contention were determined and are described in reference 3. It was shown that memory contention overhead can be contained within bounds, and that the control program execution overhead is indirectly responsible for the non-linear return of throughput, through the control program lockout as shown in Figure 5.

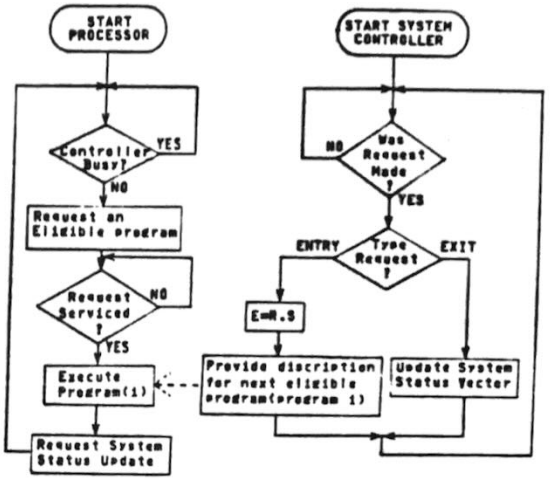

Figure 4: Transition Machine Operational Flow

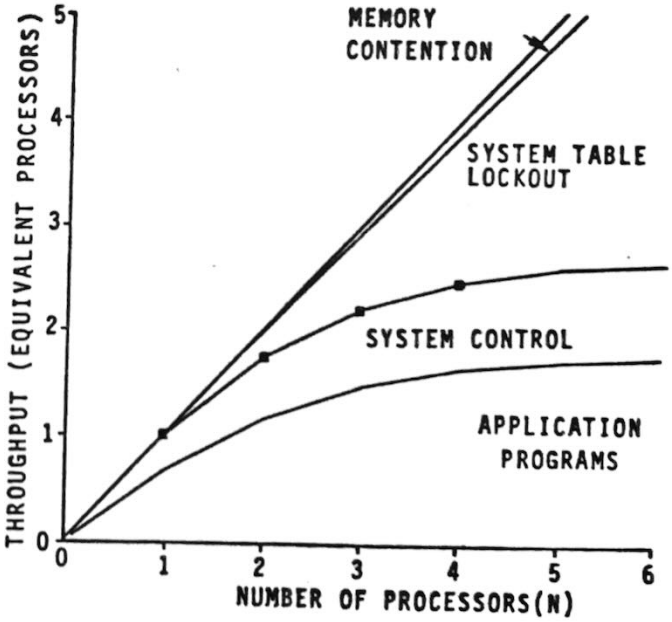

Figure 5: Contributions To Multiprocessor Overhead

Memory Organization

Various methods of connecting multiple processors to a common memory can be employed to reduce memory contention so that it need not contribute appreciable overhead. These methods include the use of multiport/multibus organizations[11,12], address interleaving[6,7,9], fast memories, time-phased processors[15], a large number of independent memory modules (M) relative to the number of processors (N), and combinations of these approaches.

"Square" multiprocessors (equal numbers of processor groups and memory modules) exhibit a probability of memory contention which is very small, and is independent of system size[10]. Arbi-

trarily small memory contention characteristics can be realized in systems which "grow" in congruent "rectangular" form. Figure 6 illustrates the expected memory contention probabilities for various ratios of numbers of processors and memories as systems increase in size proportionately. Figure 7 illustrates the effect of varying

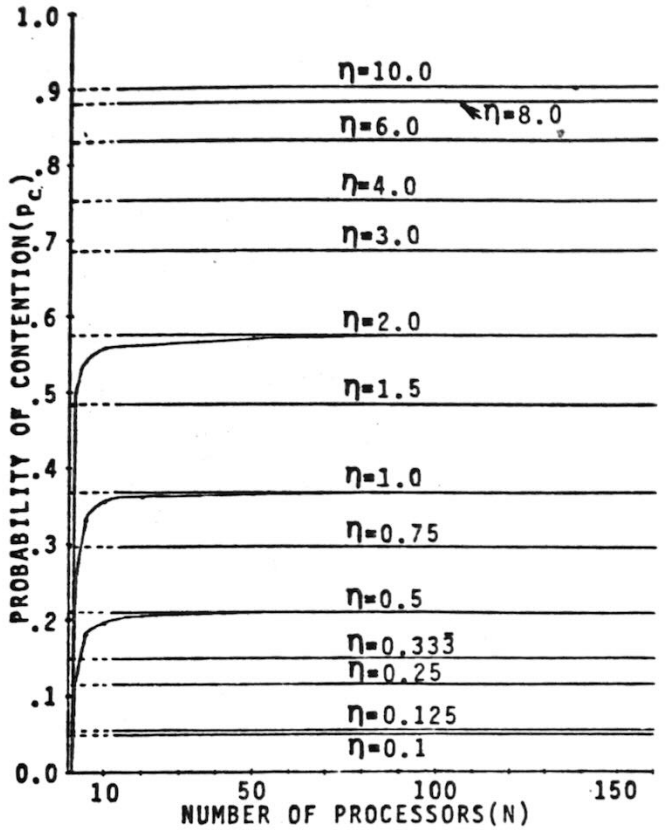

Figure 6: Memory Contention Probability Limits For "Rectangular" Systems (N=η.M)

processor request and memory response logic timing ratios. Both of these figures assume random memory address determination (approximately realized by address interleaving) and that each processor makes a synchronized memory access on every cycle which is a worst case assumption relative to typical off-the-shelf microprocessors. These results are derived in reference 3. For the TI 9900 which is being employed in an eight processor prototype system (and many other commercially available microprocessors) the latter assumption is too austere since typically less than 50% of processor cycles involve memory referencing. To approximately model the TI 9900 situation, it can be assumed that there are on the order of one-half of the processors accessing memory on any cycle. Thus, the effective ratio of number of memory modules over the number of processors in the configuration model can be doubled.

Finally, a configuration is proposed which is characterized by groups of time-phased processors on common buses each of which is interfaced to all memory modules via multiport controllers; relatively fast memories are proposed, as is address interleaving. Figure 1 shows such a configuration for which processors within a group will never contend with each other and the other methods reduce interference between groups to an insignificant level.

Multitasking Overhead Characterization

The ratio, , of average application program execution time over control program execution time was found to be a very significant parameter in throughput considerations. Reference 3 derives the limiting throughput performance formula for a multiprocessor as $T = \rho(1 - p_c)$ where p_c is the probability of memory contention on any memory access. Thus, the limiting throughput for congruent rectangular multiprocessor systems is independent of the number of processors. Adding more processors results in increasingly diminishing returns. In Figure 8 a family of throughput curves are presented representing different values of ρ with memory contention assumed to be zero in each case. Detail design data on the System Con-

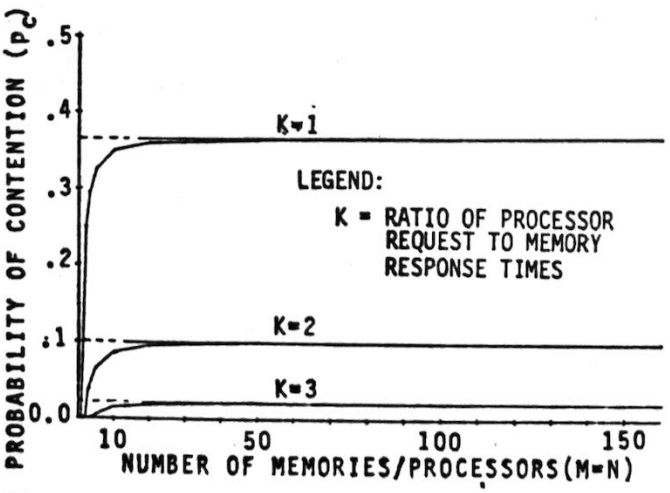

Figure 7: Effect of Varying Processor Request and Memory Response Times on P_c

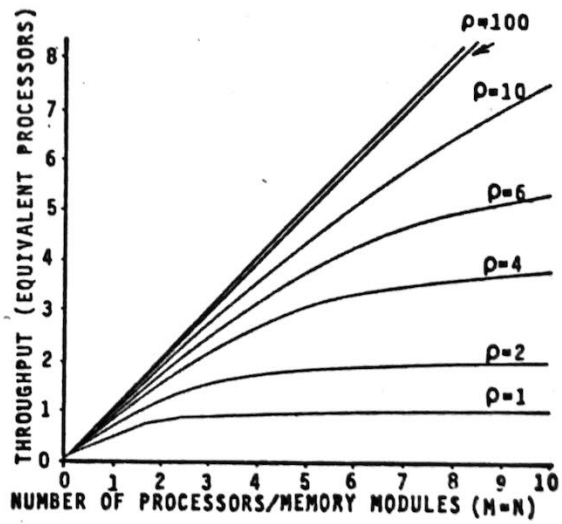

Figure 8: Impact of ρ on Throughput

troller has been obtained which indicates that an average entry/exit transition overhead associated with a single application program task can be reduced using state of the art component technology to something on the order of one microsecond. This constitutes several orders of magnitude overhead reduction when compared with current software executive approaches. This means that even when a very low activation level of software is addressed (i.e., very short average program execution time) a large value of ρ can be expected when microprocessors are involved. The importance of this is amplified by processor utilization considerations.

Processor Utilization

The amount of parallelism available in typical programs has been investigated by Kuck[14]. The result of this investigation has shown that the average number of possible parallel paths in a program is linearly related to the size of the program rather than related as the order of log of the size as was formerly thought to be the case. This conclusion is encouraging relative to using arrays of microprocessors to obtain significant throughput capability. But to exploit this amount of parallelism requires allocation of individual program tasks to processors at a very low level (even below the HOL statement level in some cases). This then becomes a very stringent requirement on the overhead associated with the coordination of parallel processes. This requirement has been met by the low overhead System Controller.

Microprocessor Firmware

The unique computer architecture described previously requires unique capabilities in the microprocessor firmware in several areas:

Special Macro Instructions

The particular System Controller interface requirements suggest bit assignment capabilities based upon the results of logical operators such as greater than, equal, etc. This will accommodate the dynamic condition assignments such as: D(i) = a > b.

The assignment of processor activities from the System Controller is effected by loading task interface registers in the processor with read, write and execute descriptor values. Addressing modes should therefore be microprogrammed to be effected indirectly through these registers.

Interrupt Structure

The approach that is being pursued is to connect interrupts directly to processors as in conventional architectures. The System Controller however will have a "row" in its constructs assigned to interrupts. (Note that the null program associated with the interrupt "row" will be precluded from eligibility because of unrealizable internally specified conditions.) The S vector will have condition indicators associated with specific interrupts whose values indicate whether the interrupt has occurred. The interrupt response processing routines can then be treated as application programs whose eligibility is based on the status of these indicators.

When an interrupt occurs, the processor will first save the current processor state (program counter, processor status word, etc.) so on completion of the interrupt handling, the processor can return to the previous task. After saving the processor state it will wait until the System Controller is ready to service a request, save the current values of the task interface registers and overstore them with values appropriate to the System Controller interrupt row. The (microprogrammed) interrupt procedure in the processor will then load values associated with the specific interrupt into the D register, and initiate an exit transition. This exit transition causes the condition indicators in the S vector associated with the interrupt to be set appropriately. The procedure will then wait until the exit transition has been completed by the System Controller. The processor will then restore the interrupted task interface registers, restore the previous processor state and return to the interrupted activity. The entire procedure is shown in Figure 9.

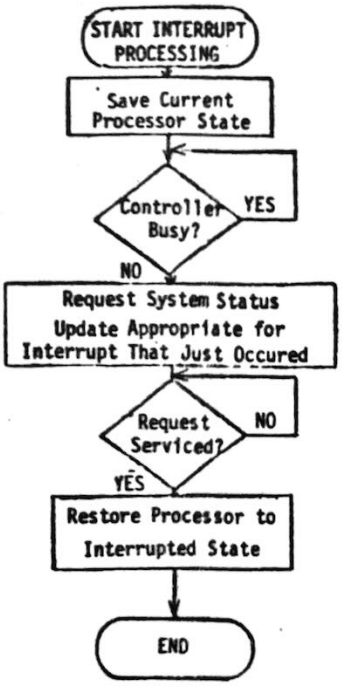

Figure 9: Interrupt Processing Operational Flow

This entire sequence is completely transparent to the interrupted program. As the result of these operations, whatever actions are appropriate in response to the interrupt will be determined eligible by the System Controller when the next processor makes a request for an activity.

Conclusions

Analyses indicate that with the approach to multi-microprocessing essential to Transition Machine architectures, a linear increase in processing capability can be realized as system size is increased up to about 100 processors. This projection applies to highly parallel system applications using a single control device. Multiple control devices could be used to increase efficiency if more processors were required for an application.[4] An eight processor, four memory module, single System Controller prototype system is currently being developed to demonstrate feasibility.

The use of the system control constructs in a top-down software design hierarchy is currently under investigation. The operating system concepts involved in the manipulation of these matrix overlays and main memory program/data allocation is also under investigation. Some of the concepts being considered are discussed in reference 4.

In actuality of course, practical hardware considerations come into play before 100 parallel processors would be obtainable. These may impose more sever limitations. These considerations involve such problems as pin-out which could nonethe-less be addressed by LSI implementations.[17] In effect, by virtually eliminating the multitasking and memory contention overhead, the emphasis in cost reduction for extremely large throughput computers obtained by multiprocessing many smaller computers must now shift. The emphasis must now be focused on the physical characteristics of the components from which these systems are implemented and the software development support tools.

References

1) Adams, G., and Rolander, T., "Design Motivations for Multiple Processor Microcomputer Systems", Computer Design (March 1978)

2) Anastas, M. S., and Vaughan, R. F., "Direct Architectural Implementation of a Requirements-Oriented Computing Structure," (submitted for publication in 1979)

3) Anastas, M. S., and Vaughan, R. F., "Limiting Multiprocessor Performance Analysis", Proceedings of the 1979 International Conference on Parallel Processing, IEEE, Michigan (August 1979)

4) Anastas, M. S., and Vaughan, R. F., "Parallel Transition Machines", Proceedings of the 1979 International Conference on Parallel Processing, IEEE, Michigan (August 1979)

5) Basket, F., and Smith, A. J., "Interference in Multiprocessor Computer Systems with Interleaved Memory", Comm. of ACM 19, 6 (June 1976)

6) Berg, R. O., and Thurber, K. J., "A Hardware Executive Control for the Advanced Avionic Digital Computer System", NAECON '71 Record (1971)

7) Bhandarkar, D. P., "Analysis of Memory Interference in Multiprocessors", IEEE Trans. on Computers C-24, 9 (Sept. 1975)

8) Böhm, C; and Jacopini, G., "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules":, Comm. of ACM 9, 5 (May 1966)

9) Burnett, G. J.; and Coffman, E. G., Jr., "A Combinatorial Problem Related to Interleaved Memory Systems", J. of ACM 2, 1 (Jan. 1973)

10) Chang, D. Y., Kuck, D. J., and Lawrie, D. H., "On the Effective Bandwidth of Parallel Memories", IEEE Transactions on Computers, May 1977

11) Enslow, P. H., Jr., "Multiprocessor Organization - A Survey", ACM Comp. Surveys 9, 1 (March 1977)

12) Hojberg, K. S., "One-Step Programmable Arbiters for Multi-processors", Computer Design (April 1978)

13) Keller, R. M., "Formal Verification of Parallel Programs", Comm. ACM 19, 7 (July 1976)

14) Kuck, D. J., "A Survey of Parallel Machine Organization and Programming", ACM Computing Surveys 9, 1 (March 1977)

15) Loewer, R., "The Z-80 in Parallel", Byte (July 1978)

16) Madnick, S. E., "Multi-Processor Software Lockout", Proc. ACM Nat. Con. (1968)

17) Siewiorek, D. P., "The Use of LSI Modules in Computer Structure: Trends and Limitations", Computer (July 1978)

18) Weissberger, A. J., "Analysis of Multiple - Microprocessor System Architecture", Computer Design (June 1977)