PARALLEL TRANSITION MACHINES

Mark S. Anastas and Russell F. Vaughan
Boeing Aerospace Company
P.O. Box 3999
Seattle, Washington 98124

Abstract -- The architecture for a general purpose parallel computer called a Parallel Transition Machine is derived from an abstract theoretical model of parallel computation. Its development is pursued to a design description involving hardware block diagrams. Applicability across a broad spectrum of computational requirements is suggested by the modular extendability of the hardware units. It is apparent that computational problems can be broken down in a design hierarchy where each level executes in a virtual Transition Machine dynamically assignable to a free hardware unit.

Introduction

There are completely general models of parallel computation [8], but there are currently no machine architectures suitable for efficient execution of parallel programs generated in accordance with one of these models. The parallel computer architectures which do exist are special purpose devices appropriate only within a restricted domain of any general parallel computational model. For example, the restricted homogeneous parallelism afforded by an associative or an array processor can only perform identical operations on multiple data sets concurrently.

This paper describes a family of computational machine architectures which implements a general model of parallelism. The conceptual model of parallel computation described by Keller [8] under the nomenclature of transition systems has been accepted here as the basis for a more detailed model of a machine architecture. This Parallel Transition Machine model provides a machine architecture in which transition systems can be executed. The details of this architecture are defined to a level where development can proceed. A prototype system is currently under development; some of the detail design issues addressed by this development are discussed in reference [3].

The architecture can be characterized as a multiprocessor with a separate System Controller as shown in Figure 1. Interrupts, I/O controllers, and other special purpose processors can be integrated into this model of Transition Machines, and introduce no significant developmental problems. The System Controller is in essence a functional equivalent (implemented in hardware) of a multiprocessor executive for transition systems. The operation of the System Controller is effected by a series of logical operations on fixed data constructs descriptive of the conditions under which the various computations become eligible. It effects the system transitions by performing matrix operations on a system status vector to obtain a procedure eligibility vector. The procedure eligibility vector provides a basis for task assignments to the processors; completion of the assignments results in a modified system status vector.

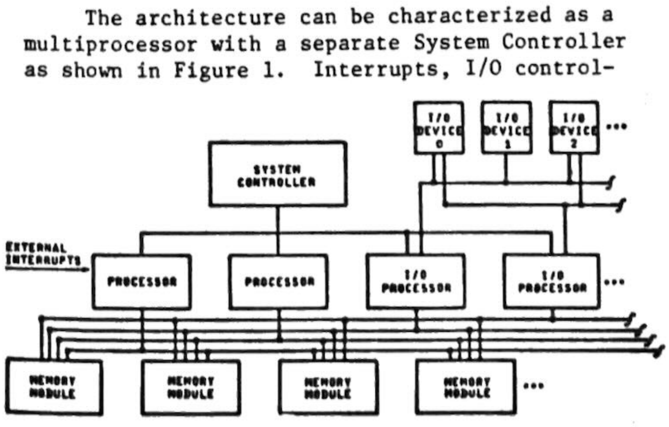

FIGURE 1: PARALLEL TRANSITION MACHINE

The development cost of System Controllers is small relative to the cost of the multiprocessors which are controlled by them. System transitions can be effected in a fraction of the time that is currently required for straightforward software multiprogramming executives. This supports an approximately linear extendibility of throughput in large array multiprocessors. To implement the equivalent system control matrix operations in a software multiprocessing executive would be infeasible due to the high overhead as shown in reference [3].

The application of Parallel Transition Machines to large systems is extremely promising, and the feasibility of configuring arrays of coordinated microprocessors seems evident [14]. But large systems introduce commensurate challenges; for example, an operating system and linkage editor of considerable complexity are required to implement the overlaying of partitioned matrices.

Parallel Transition Machines also require programming structures which are not traditional. These non-traditional structures provide the advantages of highly structured programs which result in enhanced software productivity [1]. It is possible, however, to develop only the translator for a suitable existing compiler so that traditional program structures could be translated to run on Parallel Transition Machines.

Abstract Parallel Computation Model

Parallel Transition Machines are based on a particular abstract model of parallel computation, selected because of its generality. It is transition systems $(Q,\rightarrow)$, where Q is the set of possible system states and $\rightarrow$ is the set of transitions between states as described by Keller. A named transition system is a triple $(Q, \rightarrow, \Sigma)$. The components correspond respectively to the set of possible system states $(q_1, q_2, q_3 \ldots)$, a set of transitions between states $(\rightarrow_1, \rightarrow_2, \rightarrow_3 \ldots)$, and a set of names $(\sigma_1, \sigma_2, \sigma_3 \ldots)$ associated with groups of individually programmed transitions between states [8]. Since there is a one-to-one correspondence between the indices on sigma and the names themselves, the indices will be used to indicate the names: i implies σ_i, and $I \equiv \{i\}$ implies Σ. The index $i \varepsilon I$ is associated with a group of system transitions described by the statement:

when $R_i(\xi)$ do $\xi' = \psi_i(\xi)$

The symbols in this statement are defined as follows:

i	=	the index of the group of transitions whose common feature is that they all result in the data transformation indicated by the function ψ_i.
ξ	=	the set of all data items in the system.
$R_i(\xi)$	=	the subset of satisfied propositions on the data set, ξ which are essential to defining the appropriateness, and therefore constitute the enabling predicate, for transitioning as determined by performing the data transformation $\psi_i(\xi)$.
$\psi_i(\xi)$	=	the programmed functional data transformation, associated with the group of system transitions indicated by i, which operates on the data set, ξ and results in a revised data set ξ'.

The group i can be associated with a procedure (including preamble) that can be written by a programmer to effect the data transformation, ψ_i on the data set ξ when the appropriate set of conditions R_i is satisfied on that data set. (Although obviously not the intent in Keller's work, it has been demonstrated that program requirements can be implemented to advantage in this manner [1].) In a parallel computation step, multiple sets of conditions, R_i can be satisfied simultaneously such that multiple transitions can proceed in parallel. The R_i are enabling predicates that indicate the requisite status of propositions on the data set ξ which properly enable the function ψ_i. Relevant propositions that have been defined on data elements $e_k \varepsilon \xi$ are the following:

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

2. the data element e_k satisfies/does-not satisfy a specified condition relative to some constant or other data element $e_{k'}$, (for example, $e_k < e_{k'}$), and

3. the data element e_k can/cannot be updated.

This paper deals exclusively with valid parallel programs; these programs will not exhibit race conditions, and therefore procedures which read and write the same data element will have a predetermined execution order specified by their respective enabling predicates. The properties of determinacy, commutativity and persistence are described by Keller [9]. These and other properties of valid parallel programs are also discussed in references [5], [6], [8], and [11].

Transition Machine Model

As an organizational basis for implementing the architectural model of parallel computation, we have defined a set of constructs and the logical matrix operations on these constructs which effect the system control functions for a Parallel Transition Machine. The constructs and logic are exemplified in Figure 2.

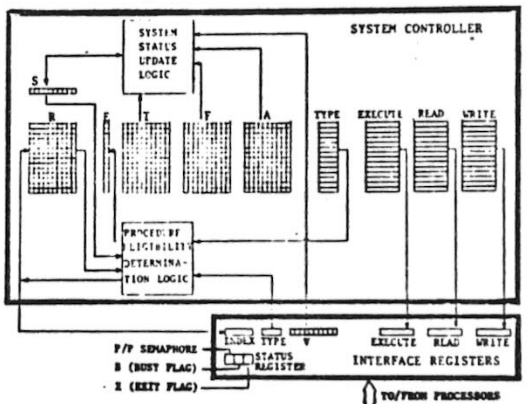

FIGURE 2: PARALLEL TRANSITION MACHINE CONTROL CONSTRUCTS AND LOGIC

Eligibility Determination

Definition 1. The system status vector, S is a set of binary status indications for a set of propositions concerning the data set ξ such that for every possible proposition on the set there is an associated status indication, S_j in S if and only if the proposition on the data set is relevant to enabling some procedure in the system. (In hierarchical implementations discussed further on, conditions which are relevant to every procedure at a given level will also be excluded.) $S_j = 1$ if the associated proposition on the data set is met, $S_j = 0$ otherwise.

For convenience we will use the phrase "data condition" in referring to a "proposition on the data set" throughout the remainder of this paper.

Definition 2. The system eligibility vector, E is a set of binary status indications for the set of predicates R_i, such that for each predicate R_i there is an associated status indication, E_i in E indicating whether R_i is currently satisfied, enabling the associated procedure. $E_i = 1$ indicates the associated predicate is satisfied; $E_i = 0$ otherwise.

Definition 3. A data condition associated with S_j is relevant to enabling procedure i if and only if the data condition whose status is indicated by S_j is included in the predicate R_i.

Proposition 1. The predicate, R_i can be represented as a set of binary relevance indications associated (and in conjunction) with each of the data conditions whose status is maintained in S. This proposition follows directly from the previous definitions.

Definition 4. The relevance matrix, R is comprised of binary relevance indications, r_{ij} indicating the relevance of a data condition j to enabling procedure i. Relevance is indicated by $r_{ij} = 0$, irrelevance by $r_{ij} = 1$.

Definition 5. The logical dot product, of a matrix M (with dimension IxJ) and a vector, W (a vector of dimension J) is defined as the vector, $P = M \cdot W$, with dimension I, where

$$P_i = \bigwedge_{j=1}^{J} M_{ij} \vee W_j$$

In this equation (and throughout this paper) the following symbol definitions apply:

$$\bigwedge_{n=1}^{N} x_n \equiv x_1 \wedge x_2 \wedge \cdot \cdot \wedge x_n.$$

$\wedge \equiv$ logical "AND",
$\vee \equiv$ logical "OR".

Proposition 2. The system eligibility vector, E can be computed appropriate to a given state of the system by generating the logical data product of the relevance matrix, R and the system status vector, S.

Proof:

From definition 5 is follows that:

$$\left[R \cdot S\right]_i = \bigwedge_{j=1}^{J} r_{ij} \vee S_j$$

From definitions 4 and 1 it follows that $r_{ij} \vee S_j = 1$ if and only if data condition j is either met or irrelevant to enabling procedure i. Then by proposition 1 it follows that $\left[R \cdot S\right]_i = 1$ if and only if all data conditions of the predicate R_i are satisfied. Thus, $\left[R \cdot S\right]_i = E_i$ by definition 2, and it is proved that $E = R \cdot S$ as proposed.

There is now a prescription for determining procedure eligibilities based on system status and the procedures' data conditional requirements. What remains to be shown is the computation of the new system status vector appropriate to having completed a given procedure.

System Status Update

Since Keller did not address actual implementations of named transition systems, it was not incumbent upon his work to address representation and the associated maintenance of state. In this paper however, we posit that there are J data conditions (propositions concerning the data set) whose status (true or false) will provide sufficient information concerning the state of the system to effect any and all of the named transitions defined for the system. But the status indications for these data conditions maintained in the S vector are not a part of the data set ξ associated with the transformations $\psi_i(\xi)$. Therefore, they must be updated separately in order that changes of state be reflected in the system status vector.

There are several possible implications on the status of a data condition at the completion of a procedure which implements the data transformation. They are as follows:

1. The data condition's status remains unaffected by the procedure running to completion.

2. The data condition's status is satisfied whenever the procedure runs to completion.

3. The data condition's status is negated whenever the procedure runs to completion.

4. The data condition's status is determined dynamically during the execution of the procedure.

The fixed constructs which are implemented to effect system status modifications are described below:

Definition 6. The jth element, t_{ij} of the true condition vector, T_i is a binary status indication associated with procedure i and the data condition, j such that $t_{ij} = 1$ implies the data condition j is either satisfied or unchanged by the completion of procedure i.

Definition 7. The jth element, f_{ij} of the false condition vector, F_i is a binary status indication associated with the procedure i and the data condition, j. The element $f_{ij} = 1$ implies the data condition j is either negated or unchanged by the completion of procedure i.

Definition 8. The variable condition update vector, V is a set of binary status indications which can be set dynamically by a procedure running in a sequential processor. The component V_j is set to 1 by the procedure to indicate that data condition j is satisfied or V_j is set to 0 to indicate data condition j is not satisfied. For elements in S that are not to be dynamically updated, the associated element in the V vector can be set to either 0 or 1.

Proposition 3. The four possible implications on change in system status following completion of procedure i can be computed according to the formula:

$$S_{new} = (S_{old} \wedge T_i) \vee (T_i \wedge \overline{F}_i) \vee (\overline{F}_i \wedge V_i)$$

where the bar indicates the logical NOT operation.

Proof: The proof follows directly from the definitions of the associated vectors as shown in Table I.

TABLE I SYSTEM STATUS UPDATE POSSIBILITIES

t_{ij}	f_{ij}	S_{jNEW}	Implications to System Status
1	1	S_{jOLD}	unchanged
1	0	1	set true
0	1	0	set false
0	0	V_j	set variably

It should be noted that there are many forms which definitions 6 through 8 and proposition 3 could have taken. The expression which we have used has the advantage of restricting the range of V such that a procedure can dynamically modify only conditions for which it is authorized.

Proposition 4. The rangeof V is restricted such that V can modify only a specific subset of the data conditions, j. This subset is determined by T_i and F_i for procedure i such that S_j is determined by V_j if and only if $t_{ij} = 0$ and $f_{ij} = 0$.

Proof: The implied new values of S_j for the various values of t_{ij} and f_{ij} from proposition 3 are shown in Table I from which the proposition follows directly.

It should be noted that there are also implied modifications to system status at entry to a procedure; these modifications are to prohibit the same transition from being attempted in other processors by denying subsequent update access to relevant portions of ξ when $\xi' = \psi_i(\xi)$ has been initiated.

In order to accommodate exclusive data access, another construct must be added to negate availability of data which is to be updated by a currently activated procedure. The update is required to insure that read/write conflicts do not arise between procedures whose execution is not "indivisible". A discussion of the concept and definition of indivisibility can be found in references [9] and [11]. To implement this update, a vector A_i has been defined which is associated with each procedure, i to specify the status update implied on entry to that procedure.

Definition 9. The vector A_i is a set of binary status conditions a_{ij}, where the index j is associated with the data conditions whose status is maintained in S. $a_{ij} = 1$ if and only if the jth data condition is a mutually exclusive data availability condition required at entry to procedure i; $a_{ij} = 0$ otherwise.

Proposition 5. Modifying the system status vector according to the formula $S_{NEW} = S_{OLD} \wedge \overline{A}_i$ prior to entry is sufficient to effect contemporaneous access protection for procedure i.

The proof of this proposition follows immediately from definitions 1, 4 and 9 and proposition 2 if there are no procedures activated prior to activating procedure i which are affected by or affect these mutually exclusive data availability conditions. If such procedures are currently active, procedure i would not have become eligible. (Refer to Keller [9] for definitions of commutativity and persistence as they relate to valid parallel programs.)

Proposition 6. If $\overline{A}_i$ is identical to the ith row in R for all i, then all procedures with any entry conditions in common must execute sequentially.

The proof of this proposition follows as a special case of proposition 5.

Proposition 7. Modifying the system status vector according to the formula $S_{NEW} = S_{OLD} \vee A_i$ restores S to its original value.

Proof: The proof of this proposition follows directly from definition 9 and proposition 5 if there are no changes to S between entry and exit of the ith procedure. When there are other procedures initiated or terminated in the interval, the proof holds because no procedures can proceed in parallel if they are affected by or affect the same data availability condition covered by A_i. (Refer to Keller [9] for definitions of commutativity and persistence.) Therefore, for every condition for which $a_{ij} = 0$ there will have been no intermediate change to S_j and the proof is completed.

Proposition 8. The change in system status following completion of procedure i can be computed according to the formula:

$$S_{NEW} = ((S_{OLD} \vee A_i) \wedge T_i) \vee (T_i \wedge \overline{F}_i) \vee (\overline{F}_i \wedge V_i)$$

The proof follows directly from the proofs of propositions 3 and 7.

It has been shown in reference [14] that interrupts can be integrated into the model in a near conventional manner. Externally activated procedures are defined for them which can never become eligible based upon their R_i vector, but which have an associated system status update identical to internally activated procedures, when they exit. This updated system status will then activate appropriate interrupt processing procedures.

Procedure Activation

The emphasis of the preceding definitions and propositions has been to create a basis for determining the eligibility of the individual data transformations which comprise the computation, as well as to maintain a current system status vector. In effect we have a sufficient basis for the determination of the "When $R_i(\xi)$". This does not however include a sufficient basis for the activation of the data transformations, i.e., the "DO $\xi' = \psi_i(\xi)$". As a basis for this activation procedure, it will suffice to maintain a triple of descriptors for each procedure ($READ_i$, $WRITE_i$ and $EXECUTE_i$) in the System Controller. These descriptors designate respectively the elements of the data base, ξ which are to be read, the elements of the data base, ξ' which are to be written, and the starting address of the executable procedure, ψ_i which implements the data transformation. The appropriate descriptor triple can be transferred to the interface registers to effect activation of the procedure in the requesting processor as shown in Figure 2.

The addressing structure of the application programs which implement the transformations is shown in Figure 3. The EXECUTE register value

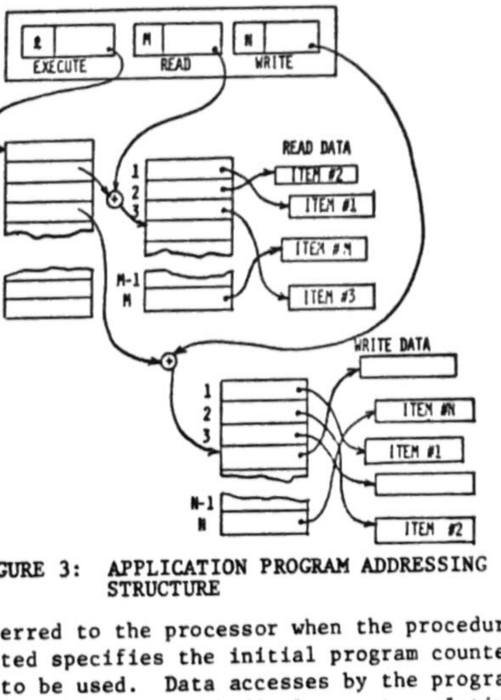

FIGURE 3: APPLICATION PROGRAM ADDRESSING STRUCTURE

transferred to the processor when the procedure is activated specifies the initial program counter value to be used. Data accesses by the program must be implemented with displacements relative to pointer packets whose starting addresses are indicated by either the READ or WRITE descriptor register value. This displacement specifies a particular descriptor value which in turn points to the data item being referenced. This scheme accommodates unique arguments to re-enterable programs as well as providing a basis for containment in multilevel secure systems [15].

Processor Type Accommodations

The architecture model has been extended to include heterogeneous processor types. This is effected by maintaining a processor type designation for each procedure, $TYPE_i$. The procedure eligibility determination includes an evaluation of the equivalence between the type of the requesting processor and the type designation of the eligible procedure. Thus, if the defined procedure requires an I/O activity, an I/O controller would be specified as a requirement for the procedure. Having incorporated this approach into the model allows procedures to specify a special processor type such as floating point processors, vector instruction set processor, byte or word oriented processor, or just a specific processor model if several are multiprocessed in the same configuration.

The data construct, TYPE in Figure 2 is defined to accommodate this capability. In addition, each processor must have its own type identification available to the eligibility determining logic in the interface registers.

System Controller Hardware Organization

The System Controller is the device that is designed to contain the fixed data constructs for each procedure, performs the logic to determine procedure eligibility and system status updates, and assigns activities to processors. The device described in this section is a specific design based on the architecture model which has just been described. This design is applicable to either single or multiple processor systems. Figure 4 is a functional block diagram of this device. The content and function of the major blocks in the diagram are described below:

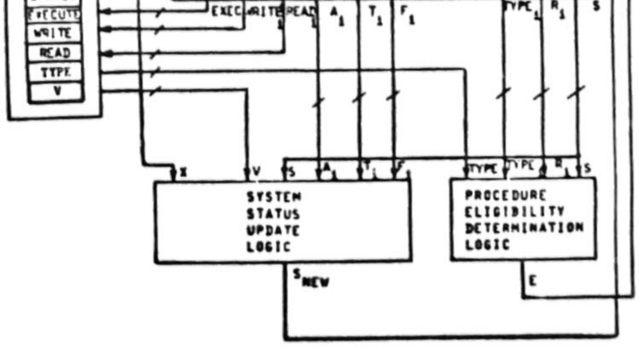

FIGURE 4: FUNCTIONAL BLOCK DIAGRAM OF THE SYSTEM CONTROLLER

Processor/System Controller Interface

The interface block contains all data and control registers accessible to the processors. The structure and use of these registers are as follows:

STATUS a 3-bit read/write register whose bits are labeled P/P, B, and X, and which contain the following synchronization, protocol and mode request information.

P/P is a 1-bit binary semaphore used to prevent multiple processors from accessing the System Controller interface registers simultaneously. The P/P semaphore is set when a processor is accessing the System Controller and it is reset when no processor is currently accessing the System Controller.

B is used to prevent the processors from accessing the System Controller while it is busy servicing a request. When a processor makes a request to the System Controller, it waits until B is reset, sets X to the appropriate value, and sets B true. This activates the System Controller which resets B when the request has been serviced.

X is used to notify the System Controller of the type of service being requested. X is set (true) by the processors when the service requested is the result of a procedure exiting and it is reset when an activity is requested. X is only required in multiple processor implementations.

TYPE is a register used to contain the processor type identification. The System Controller uses this register to determine the next eligible procedure whose identification is to be loaded into INDEX. TYPE contains the processor category appropriate to the processor making the request. The System Controller returns the index of the next eligible procedure, whose type matches the value in the TYPE register.

INDEX is a register used to contain the identification of either the assigned procedure or the procedure currently being exited. As the fulfillment of processor activity requests, the System Controller loads INDEX with the index of the next eligible procedure whose type matches the value contained in the TYPE register, or INDEX is loaded with a 0 if no procedures of the appropriate processor type are eligible. When a procedure exits, the System Controller assumes INDEX contains the associated procedure index.

EXECUTE contains the entry point of the procedure whose index is contained in INDEX. (Refer to Figure 3.) EXECUTE is loaded by the System Controller as the result of the activity request. EXECUTE is unused when an exit is requested.

READ contains an indirect pointer to the global data item(s) accessible to the associated procedure in a read capacity. (Refer to Figure 3.) READ is loaded by the System Controller as the result of the activity request. READ is unused when an exit is requested.

WRITE contains an indirect pointer to the global data item(s) accessible to the associated procedure in a write capacity. (Refer to Figure 3.) WRITE is loaded by the System Controller as the result of the activity request. WRITE is unused when an exit is requested.

V contains the variable status update vector loaded by the processors upon exit from a procedure. This vector allows a procedure to return variable data condition status to the system status vector. Notice that his allows the task to modify only selected data elements since any attempt to modify unauthorized data will be masked out by the T and F vectors, stored internally to the System Controller.

By allowing the processors to access only the data and status registers defined above, all system control logic is localized to the System Controller. This also prevents the processors from accessing unauthorized programs, data, or control information, providing a natural basis for implementing secure systems. The security aspects of Transition Machines are discussed in reference [15].

Data Constructs

The data block is comprised of memory modules which contain the data structures required to control the system transitions, as shown in Figure 2. These include the EXECUTE, READ, WRITE, and TYPE arrays and the T,F,A, and R matrices as defined previously. The data block receives one input, the data select bus which addresses each of the EXECUTE, READ, WRITE, TYPE, T,F,R, and A constructs concurrently, causing the element indexed in each of these memories to be output on its associated data bus.

It is assumed that there is a load capability which will allow the programmer to change the content of these memory modules. The content must necessarily change during program development or in real-time in large systems where the matrices will be overlayed dynamically during the execution of the system (analogous to overlaying the task control blocks by a conventional operating system). For dedicated special purpose applications, however, these constructs could be fixed and put into read-only memory. The load procedures are system-dependent and are therefore not a subject of this paper. An implementation applicable to large general-purpose systems is discussed further on and is the subject of continuing research. The sizing of these memory modules is addressed further on in this paper also.

Fixed Transition Logic

The fixed transition logic block contains the combinational logic necessary to update the system status vector and to determine procedure eligibility. This block requires the T,F,V, and A vectors for the procedure currently being assigned or exited in order to generate the new system status vector, S. The logic expression for the new S vector generated as the result of either an activation or procedure exit request are shown in Figure 5. This diagram uses the

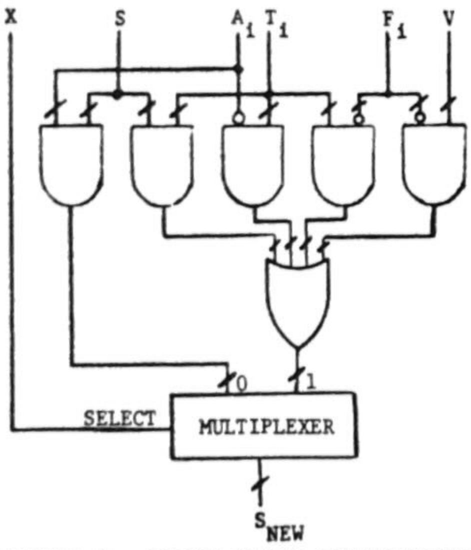

FIGURE 5: SYSTEM STATUS UPDATE LOGIC

convention defined in Figure 6 of using a "/" on a line to indicate multiple lines treated identically.

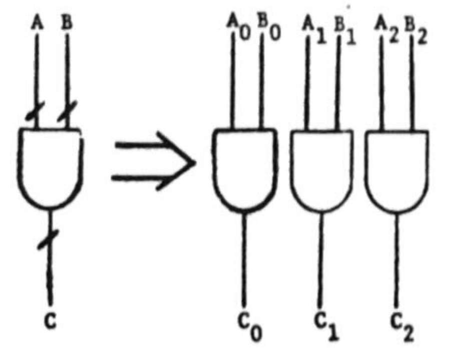

FIGURE 6: TREATMENT OF MULTIPLE LINE GATE INPUTS

The combinational logic also combines the current S vector with successive rows from the R matrix and compares the successive elements of the type array with the TYPE register to determine the eligibility of each procedure. A single output bit, E_i is provided as an input to the synchronization and control logic. The logic to obtain E_i is shown in Figure 7.

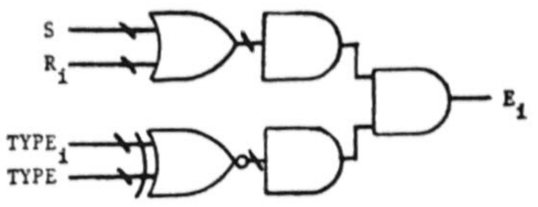

FIGURE 7: PROCEDURE ELIGIBILITY DETERMINATION LOGIC

The System Controller design described here assumes the eligibility vector (E) is computed one element at a time. This need not be the case. An associative memory can be used to generate the entire E vector in parallel which will result in much faster transition speeds but requires more hardware support.

Synchronization and Control Logic

The synchronization and control logic synchronizes the operation of all the components of the System Controller. The control logic operates as shown in Figure 8. The System Controller waits

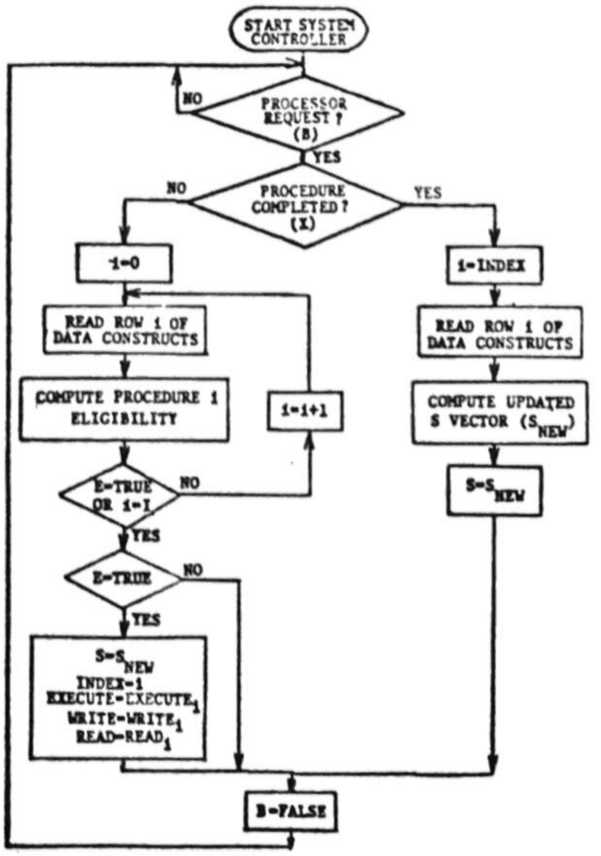

FIGURE 8: SYNCHRONIZATION AND CONTROL LOGIC

in an idle state until its execution is initiated by a processor request (i.e., B set). If X is set, the System Controller initiates a system status update and if X is not set, a procedure eligibility determination and assignment is initiated.

When a procedure exit request is initiated, the non-zero procedure index provided by the processor is used as the address selection value on the data select bus. This in turn causes the appropriate array elements to be output on each of the memory data buses. The fixed transition logic then updates the system status vector to effect the update implied by the procedure exit. The B indicator in the STATUS register is then reset to indicate the request has been serviced.

The procedure activation request causes successive rows of the R matrix and the TYPE array to be output on their respective data buses. As each row is output, the fixed transition logic generates the next element of E. INDEX is loaded with either the procedure index for the first eligible procedure or with a zero signifying that no procedure is currently eligible. If INDEX is non-zero, the EXECUTE, READ, and WRITE pointers associated with the indexed procedure are transferred to their respective interface registers. The fixed transition logic then generates the new S vector appropriate to the protection of the assigned procedure. At this point, a complete entry transition has been effected. The System Controller busy indicator, B is then reset to allow the processors access to the interface registers again.

The detailed processor logic and System Controller interface and internal logic are provided in the syntactical expressions of Figure 9 and 10. These figures have didactic value as indicative of a source language structure applicable to application programs to be run on Transition Machines.

Performance Characteristics

A throughput performance model was developed which predicted the throughput capabilities of parallel transition Machines [13]. This model was made general enough to include multiprocessors with software executive control mechanisms. The three major contributions to system overhead that were examined are memory contention, the overhead associated with the central control mechanism, and control mechanism lockout experienced while waiting for a request to be serviced. Memory contention contributions were assumed to arise even from processors which are determining their next application program assignment (i.e., currently executing the executive program in the case of a conventional system or waiting for the System Controller to service the outstanding procedure entry request in Transition Machines). Since Parallel Transition Machines can be designed to be exempt from this contribution to memory contention, measured performance should be better than predicted by the model in this regard.

The significant performance parameter was shown to be $\rho \equiv A/\emptyset$, where $\emptyset$ is the characteristic System Controller overhead per application procedure (eligibility determination and system status update), and A is the characteristic application procedure execution time requirement. The value of ρ determines the number of processors that can be effectively combined in a tightly coupled mode of operation as described in the reference.

```
When Exit false (Entry)
 When test and set of P/P semaphore true
  When B false (System Controller not busy)
   Begin:  Assignment Processing
    Store TYPE
    Store X false (entry)
    Set B true (activate System Controller)
    When B false (System Controller not busy)
     If INDEX≠0 then
        Load INDEX
        Load READ
        Load WRITE
        Load EXECUTE
        Clear P/P semaphore
        Transfer Control to EXECUTE
        Set Exit true
     Else
        Clear P/P semaphore
   End:  Assignment Processing
When Exit true
 When test and set of P/P semaphore true
  When B false (System Controller not busy)
   Begin:  Exit Processing
    Store INDEX
    Store V
    Set X true (exit)
    Set B true (activate System Controller)
    Set Exit false
    Clear P/P semaphore
   End:  Exit Processing
```

FIGURE 9: PROCESSOR CONTROL LOGIC

```
When B true
 Begin:  System Controller logic
  If X true (exit) then
   i = INDEX
   $S_{NEW} = ((S_{OLD} \vee A_i) \wedge T_i) \vee (\overline{F}_i \wedge V) \vee (T_i \wedge \overline{F}_i))$
  Else (entry)
   Clear i
   While $E_i = 0$ and $i < I$
    Increment i
   $E_i = \bigwedge_{j=1}^{J} (s_j \vee r_{ij}) \wedge (\text{TYPE} \odot \text{TYPE}_i)$
   If $E_i = 0$ then
    INDEX = 0
   Else
    INDEX = i
    $\text{READ} = \text{READ}_i$
    $\text{WRITE} = \text{WRITE}_i$
    $\text{EXECUTE} = \text{EXECUTE}_i$
    $S_{NEW} = (S_{OLD} \wedge \overline{A}_i)$
  Set B false
End:  System Controller Logic
```

FIGURE 10: SYSTEM CONTROLLER LOGIC

The overhead, $\emptyset$ is very dependent upon the component technology used in the development of the System Controller, A is dependent upon the speed of the individual processors. Current component technology will support $\emptyset \leq 1$ microsecond. With microprocessors, $A \geq 100$ microseconds is very conservative. This yields $\rho = 100$, which indicates according to the model that on the order of 100 microprocessors could be combined in a tightly coupled mode of operation controlled by a single System Controller with a proportionate throughput capability.

Lockout is the primary reason for the flattening of the throughput curve as a function of the number of processors. To avoid this problem in Parallel Transition Machines, multiple system Controllers (including separate interface registers) can be incorporated so that if a processor is locked out of one System Controller it can attempt to acquire another, etc. Thus, in a batch type system, many disjoint computations could be running contemporaneously across all processors.

A given computation will be characterized by some maximum and average numbers of concurrent execution paths. (See for example Kuck [10].) The average concurrency will determine the processor utilization realizable during the execution of a given computation. Thus, in general where many processors are included in a configuration, there is a requirement for concurrency of active computations in order to achieve efficient utilization of processors. This applies particularly where a large system is being used for many small computations.

The upper limit on throughput in Parallel Transition Machines is thus determined by other than system control considerations. Physical module interconnection schemes now become the limiting factors. High speed buses and processor/memory groupings [12] are likely approaches to extending these limits.

System Controller Memory Sizing

In what has been presented so far, there has been the implicit assumption that all of the constructs associated with relevant data conditions and programmed procedures for an entire system can be accommodated in the data constructs memory modules in the System Controller. In a prototype system currently in the development stages, these constructs are contained in a single module and are allocated dimensions of 32 conditions by 32 procedures. Analytic studies and simulations have indicated that as a rule of thumb there are one and one-half times as many conditions required to control a tightly coupled network of procedures as the number of procedures involved. This would indicate that optimum memory utilization would be more probable for System Controllers characterized by such dimensional ratios.

How large these memories should be made to support general applications is a key issue. There is no real problem in sizing these arbitrarily large, but application systems have a way of outgrowing single physical modules. Methods have therefore been investigated which extend the logical capacity of the System Controller in modular incremental units to justify the development of a standard System Controller applicable across a broad spectrum of system sizes.

The most direct method of extending capacity is to directly increase the dimensions of the memory in the System Controller. The design described in the previous section can be expanded to accommodate more data conditions (i.e., horizontal expansion of the arrays) or more procedures (i.e., vertical expansion of the arrays) by cascading multiple sets of the standard component blocks which are then connected to a common processor/System Controller interface. This facilitates the construction of an arbitrarily large System Controller by the interconnection of many standard System Controller components each of fixed size. This cascading of component System Controllers is shown in Figure 11 for horizontal expansion. A

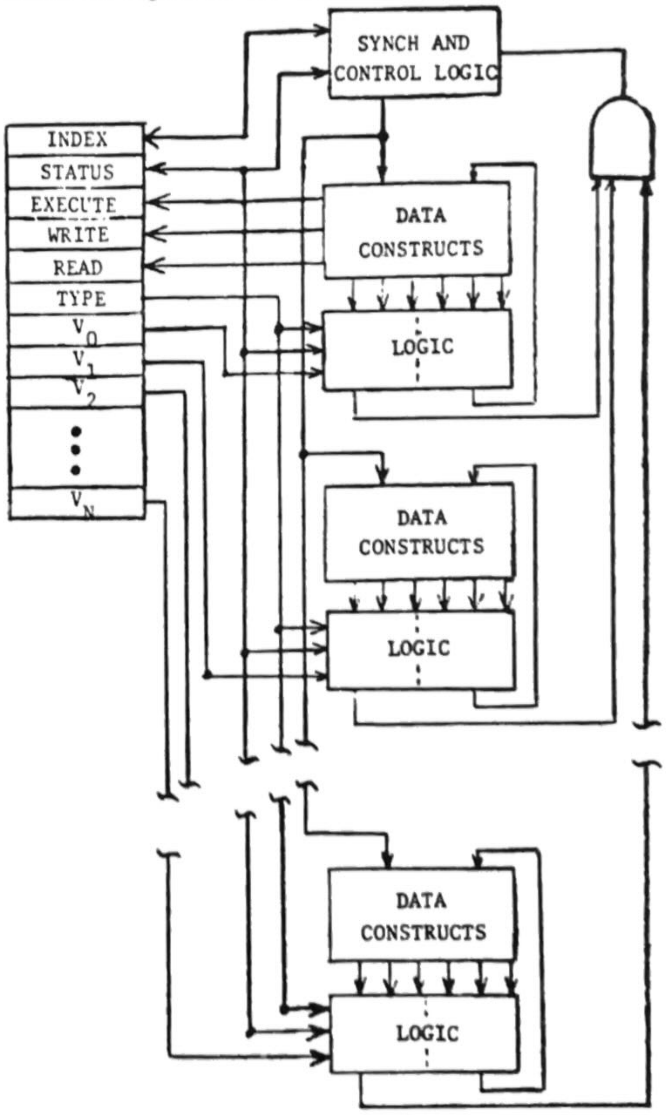

FIGURE 11: MODULAR HORIZONTAL EXPANSION OF THE SYSTEM CONTROLLER

similar approach is applicable to vertical expansion, where a master System Controller is used to multiplex between vertical segments, each of which is controlled by its own System Controller. These methods of modular expansion do not increase worst case transition times and can be applied in a structured physical hierarchy.

Virtual Transition Machines

Memory size problems are not new to computing, and the resolution of former problems encountered with insufficient main memory can be applied directly to the System Controller memories. The concept of virtual memory and commensurable virtual machines is particularly germain to Transition Machines. The segmentation and paging of the data constructs required by Transition Machines can be an integral part of a hierarchical application program design approach. The approach encompasses the design and implementation of systems as a complete transition system at each level in a hierarchy. This approach incorporates the capability of associating an indentured R matrix with a row in a higher level matrix. Conditions appropriate to each matrix level include only those which are relevant to the procedures (or immediate further indentured matrices) at this level. This top-down recursion can proceed in the extreme all the way down to where the procedures become the instruction set of processors or even indivisible operators. This instruction set can then be restricted to exclude branching type instructions. In fact, even going up one level, the Parallel Transition Machines can be used to implement a completely general system in conventional processors without requiring a branch type (GOTO) instruction in the application domain of the processors. The implications to software productivity are discussed in reference [1].

To implement the logical extension of the data constructs by partitioning the R, T, F and A matrices, additional data constructs and algorithms will be required to effect the dynamic real-time loading and overlaying of procedures. The required constructs are the following:

1. System Controller identification

2. Relevance matrix identification

3. Indication of whether a procedures or an indentured matrix is associated with each row in an R matrix.

4. Controller active indication for each System Controller

5. Logical parent identification (relevance matrix identification of parent)

6. Physical parent identification (System Controller identification of parent)

These data constructs are felt to be sufficient to implement a global operating system which results in a virtual implementation of the total system relevance matrix. Such an operating system is the subject of current and anticipated future research.

Conclusions

It has been demonstrated that even though there are no completely general parallel computation architectures commercially available, such computers are nonetheless realizable. Parallel Transition Machines which meet these specifications are defined to a level where credibility is established. A prototype of such a machine is currently in the developmental stages at the Boeing Aerospace Company. A design has been presented in this paper which is extremely flexible to meeting a wide range of implementation variations. Such machines appear to have considerable advantages over current machine architectures in several areas. These area include: Multiprocessing throughput, software productivity, and ADP security.

It is left to the future to develop the compilers, linkage editors and overlaying operating systems appropriate to the application of such computers.

References

[1] Anastas, M. S. and Vaughan, R. V., "Direct Architectural Implementation of a Requirements-Oriented Computer Structure", (submitted for publication in 1979)

[2] Anderson, G. A., and Jensen, E. D., "Computer Interconnection Structures: Taxonomy, Characteristics and Examples", ACM Comp. Surveys 7,4 (Dec. 1975), pp 197-213

[3] Bell, C. G. and Newell, A., Computer Structures: Readings and Examples, McGraw-Hill, N.Y. (1971)

[4] Conway, M., "A Multiprocessor System Design", Proc. AFIPS 1963 Fall Jt. Computer Conf., Spartan Books, Baltimore, Md. (1963) pp 139-146

[5] Dijkstra, E. W., "Cooperating Sequential Processes", In Programming Languages, F. Genuys (Ed.), Academic Press (1968) pp 43-112

[6] Habermann, A. N., "Synchronization of Communicating Processes", Comm. ACM 15,3 (Mar. 1972) pp 171-184

[7] Jensen, E. D., Thurber, K. J., and Schneider, G. M., "A Review of Systematic Methods in Distributed Processor Interconnection", Proc. IEEE Int. Conf. on Communications (1976)

[8] Keller, R. M., "Formal Verification of Parallel Programs", Comm. ACM 18,7 (July 1976), pp 371-384

[9] Keller, R. M., "Parallel Program Schemata and Maximal Parallelism", J.ACM 20,3 (July 1973), pp 514-537

[10] Kuck, D. J., "A Survey of Parallel Machine Organization and Programming", ACM Comp. Surveys 9,1 (Mar. 1977) pp 29-59

[11] Lipton, R. J., "Reduction: A Method of Proving Properties of Parallel Programs", Comm. ACM 18,12 (Dec. 1975) pp 717-721

[12] Swan, R. J. Bechtolsheim, A., Lai, K. and Ousterhout, J. R., "The Implementation of the Cm* Multi-Microprocessor", AFIPS Conf. Proc. Vol. 46, Nat. Computer Conf. (1977) pp 645-655

[13] Vaughan, R. F. and Anastas, M. S., "Limiting Multiprocessor Performance Analysis", Proc. 1979 Int. Conf. on Parallel Processing, (Aug. 1979)

[14] Vaughan, R. F. and Anastas, M. S., "Microprocessor-Based Transition Machines", Proc. COMPCON FALL '79 (Sept. 1979)

[15] Vaughan, R. F. and Anastas, M. S., "Preliminary Analyses to Obtain an Expanded Model and Preferred Implementation of Verifiably Secure ADP System", Boeing Doc. D180-25090-1 (1979)