A PROTOTYPE PARALLEL COMPUTER ARCHITECTURE FOR ADVANCED AVIONICS APPLICATIONS

Mark S. Anastas and Russell F. Vaughan

The Boeing Aerospace Company
P.O. Box 3999, Seattle, Wa. 98124

ABSTRACT

The availability of inexpensive, low power, light weight, radiation hardened, single-chip microprocessors makes the development of tightly coupled multi-microprocessor avionics feasible from the stand point of component cost. Such systems have significant advantages over conventional single processor approaches for many aerospace applications. The advantages include the ability to use a plurality of low technology devices to obtain high throughput, fault tolerance and modular expandability.

There have been, however, a number of technical problems that had to be solved in order to make such systems effective. A research activity was initiated at the Boeing Aerospace Company to address these problems. This has resulted in a new parallel computer architecture called Parallel Transition Machines [3,11]. Computers characterized by this architecture are easily programmed [2,9] and extensible to large numbers of tightly coupled processors. This paper describes an eight processor prototype machine which has verified the cost effectiveness of such machines. This prototype design and its performance characteristics are described in this paper.

DEFINITION OF TRANSITION MACHINES

The Transition Machine architecture is comprised of a conventional tightly coupled multiprocessor whose operation is controlled by an external device, the System Controller (SC). The SC is a programmable device utilized to dispatch activities to the processors. It is thus similar in function to a multiprocessor executive, but has the advantage of extremely low overhead. The significance of this advantage is determined using a parameterized model of multiprocessor throughput performance described in detail in [10].

The Transition Machine executes unique program structures. This non-von Neumann structure has been studied extensively as an abstract conceptual model. This model of parallel computation is set forth by Keller in reference [5]. The conceptual model is "Named transition systems". It is accepted as the basis for the architectural model implemented in Transition Machines. Named transition systems provide a model of the run time structure of parallel computer programs; it is a form into which any "structured" program can be translated. An example implementation of this automatic conversion process is described in a companion paper [9]. The Transition Machine provides a processing environment in which transition systems can be directly executed.

Computation Model Description

The transition system model is defined as $(Q, \rightarrow)$, where Q is the set of possible system states and $\rightarrow$ is the set of transitions between states. A named transition system is a triple $(Q, \rightarrow, \Sigma)$. The components correspond respectively to the set of all possible system states (q1, q2, q3...), a set of all transitions between states ($\rightarrow 1$, $\rightarrow 2$, $\rightarrow 3$...), and a set of names ($\sigma 1$, $\sigma 2$, $\sigma 3$...) associated with groups of individually programmed transitions between states. Each of these groups can be associated with the execution of a single program, encompassing the computer states that could be realized at entry and exit to the program.

Execution of the program effects a transition between states. Since there is a one-to-one correspondence between the indicies on sigma and the names themselves, the indices can be used to indicate the names: I is defined as the set $\{i\}$ which implies Σ. The index $i \in I$ is associated with a group of system transitions described by the statement:

$$\text{WHEN } Ri(d) \text{ DO } d' = \psi i(d)$$

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, ψi,

d, the set of all data items in the system,

$R_i(d)$, the subset of satisfied propositions on the data set, d which are essential to defining the appropriateness of transitioning as determined by performing the data transformation $\psi_i(d)$, and

$\psi_i(d)$, the programmed functional data transformation associated with the group of system transitions indicated by i which operates on the data set, d and results in a revised data set d'.

The set $\{i\}$ represents program segments (including the enabling predicate) that can be written by a programmer (or obtained by a translation from HOL source programs[9]) to effect transformations ψ_i on the data set d when the appropriate set of conditions R_i is satisfied on that data set. The ψ_i are the individual program tasks which constitute a computer program. In a parallel computation step, multiple sets of conditions R_i may be satisfied simultaneously such that multiple tasks can be executed in parallel. The R_i are enabling predicates that indicate the requisite status of the data set which properly enables the execution of the task which performs the transformation, ψ_i.

Each of the enabling predicates R_i is made up of a set $\{R_{ij}\}$, $j \in J$ of unary predicates where J is the total number of unary predicates required by the entire algorithm. A unary predicate may be defined as a single proposition on the data set and thus represents a single data condition whose value may be specified in or by a binary indication, e.g. true or false. Propositions which are examples of unary predicates on the data elements $e_j \in d$ are the following: 1) the data element e_j is available/not available for use in subsequent computations, 2) the data element e_j satisfies/does-not-satisfy a specified relation to some constant or other data element e_j (for example, $e_{j1} < e_{j2}$), and 3) the data element e_j can/cannot be updated.

Architecture Requirements

The computation structure of this conceptual model is logically divided into two aspects, an execution control specification (i.e. the WHEN statement) and a computation specification (i.e. the DO statement). The architectural implementation of this model is analogously comprised of two components as shown in figure 1. The computation component is comprised of a more or less conventional memory/processor system. The control component is a device unique to the Transition Machine architecture, the implementation of which is described briefly further on and in more detail in references [3,11].

There are three basic functions to be performed by the control mechanism which

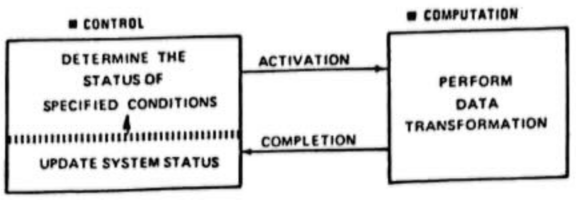
CONTROL
COMPUTATION
DETERMINE THE STATUS OF SPECIFIED CONDITIONS
UPDATE SYSTEM STATUS
ACTIVATION
COMPLETION
PERFORM DATA TRANSFORMATION

Figure 1: Transition Machine Architecture

are to: 1) determine the set of eligible tasks, 2) dispatch the eligible tasks to available processing elements, and 3) update the state on completion of the task. A set of data structures have been defined which provides the information to support these functions and which are particularly amenable to implementation in hardware.

The eligibility of any particular task, i can be determined by selecting the subset of global unary predicates that are pertinent to initiating execution of the transformation $d' = \psi_i(d)$, and by determining the status of those pertinent unary predicates. Thus the control over transitioning can be implemented by:

1. maintaining a variable system status vector S whose components are binary status indications, one for each unary predicate in the global set and
2. maintaining for each task, i a relevance vector, R_i of fixed indications for designating which of the global set of unary predicates are relevant for enabling a particular program task.

Once the sense of the indications in R_i and S have been defined, there exists a logical vector algebraic operation, "." which can be defined such that $E_i = R_i . S$, where E_i is a binary status indication of the eligibility of the task, i. The set of vectors, $\{R_i\}$ can be arranged as a matrix, R, the ith row of which is equivalent to R_i. The vector algebraic operation, "." can then be extended to matrices such that:

$$E = R . S, \text{ where}$$

E is a vector indicating the eligibility status of every defined task in the system.

There is now a prescription for determining task eligibilities (E vector) based on the system status (S vector) and the tasks' conditional data requirements (R matrix). Once the task has been detected as being eligible, it must be dispatched to an available processing element. This can be effected in a number of ways. The approach taken here is to maintain three access control descriptors for each task. These three descriptors specify the data items that can be read, the data items that can be written and the execution bounds for the code associated with the task.

The third and final requirement of the control mechanism is to update the status vector on completion of the task. There are several possible implications on the status of each data condition at the completion of a task. They are as follows:

1. The data condition remains unaffected by the task running to completion;
2. The data condition is satisfied whenever the task runs to completion;
3. The data condition is not satisfied whenever the task runs to completion; and
4. The data condition is data value-dependent, and is determined dynamically by the execution of the task.

The implications to system status at completion of the task is readily accommodated by three data constructs. Two of these are fixed vectors (T and F) together indicating which of the four possible dispositions listed above is to be realized for each data condition, completed task pair. A third vector, V, is set dynamically (4 above) by the task based upon the relationship of data variables computed by the task. For example, the task responsible for computing either of the variables A or B would have to return the status of the condition "A > B" upon completion. Execution of a task which is responsible for computing the value of a variable A would always result in the condition "A available" being set true, and would therefore not be required to return a status value in V.

It is clear that the updated status vector S can be computed as a function of T, F, and V. The class of functions can be restricted by overlapping the defined sense of the fixed vectors T and F so as to provide a mask against unauthorized dynamic changes to S through the V vector.

It should be noted that there are also implied modifications to the system status at entry to a task; These modifications prohibit the same transition from being attempted in multiple processors by denying subsequent update access to d' when the transformation $d' = \psi i(d)$ is initiated. A single vector construct (A), can be used to implement this function. This vector is used to reset a subset of the data conditions required by the program task, effectively disabling the task once it has been activated.

PROTOTYPE SYSTEM CONFIGURATION

In the taxonomy of parallel computer architectures, the prototype system is a multiple instruction multiple data (MIMD) machine. A block diagram of the prototype system is shown in figure 2. It incorporates eight Texas Instruments SBP 9900 microprocessors, four four-ported

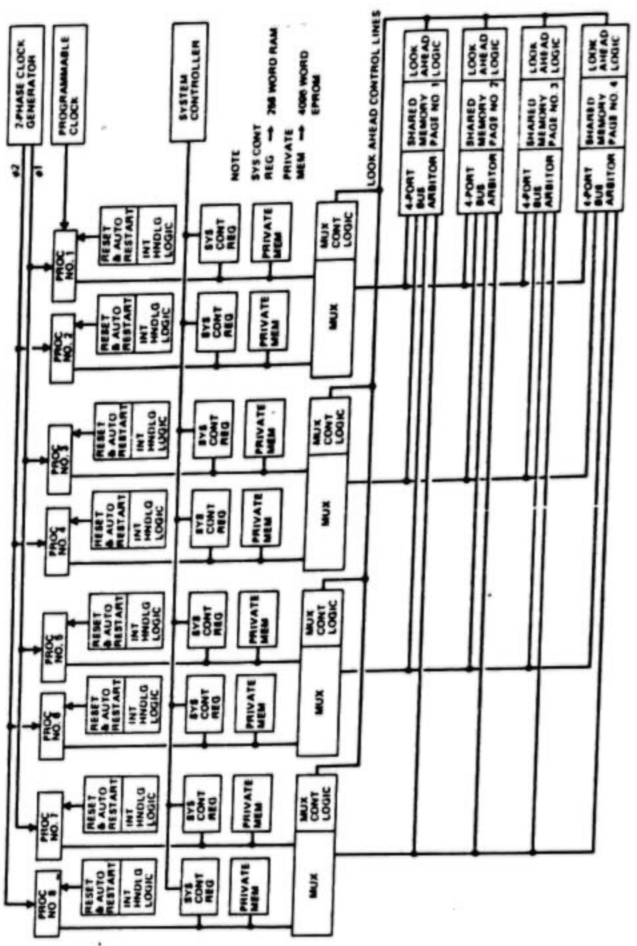

Figure 2: Research Model Block Diagram

common memory modules, shared by all processors and sixteen private memories. The sixteen private memories are divided into a 256 word RAM and a 4096 word EPROM associated with each of the eight processors. In addition, there is an RS 232 serial I/O interface port, an external real-time clock, and an Intecolor 8001 color graphics terminal.

There is additionally a single SC, which provides hardware support for the task dispatch executive control function. The operations performed by the SC are as described above.

Architectural Design Features

A number of design features have been incorporated into this system to ensure that the overhead associated with coordinating the plurality of processing units does not become excessive. There are three sources of overhead that have been addressed specifically by the hardware configuration, memory contention, executive control, and executive lockout.

Memory contention overhead has been virtually eliminated in this system using a combination of time-phased access, multiple independent memory modules, multiple independent address and data busses, interleaved addressing, and high speed memory. The executive control and executive lockout overhead contributions are minimized via the use of high speed logic in the programmable SC.

The prototype incorporates several peripheral features in addition to the rack of cards containing the actual prototype hardware. This total configuration includes support equipment which is not an integral part of the operational configuration, but has nonetheless been essential in the development of the system.

Support Components

The Boeing Aerospace Company's Microprocessor Design Support Center (MDSC) has been used to support development activities associated with the prototype. The MDSC is hosted on a VAX 11/780 running under the VMS operating system. This facility provides disk storage, interactive program edit facilities, cross assembly/compilation capabilities, prints out program listings, writes object programs on the cassette tapes, and downloads object code into the research model.

The support software system used to develop the demonstration and performance monitoring software is described in detail in [9]. This system is comprised of a "WHEN BLOCK" compiler, a cross assembler, and a relocating linking loader. All of these components are hosted on the MDSC VAX 11/780. There is additionally a monitor/debugger program which is resident in a private EPROM associated with each processor. It provides the basic support required to develop and execute software on the research model.

System Controller Module

The SC is implemented on a single wire-wrapped module which interfaces with each of the eight processors through the dual ported private RAM associated with each processor. A block diagram of the SC is shown in figure 3. It is constructed from MSI and SSI components and contains a thirty two column by thirty two row control memory which can be loaded by the processor. At any one time the SC can therefore control up to thirty two different tasks based on the status of thirty two data conditions.

To effect system operation, the processor loads the private RAM interface words with the appropriate values and makes a transition request via a set of control signals. The SC then polls the individual processors until it finds a transition request outstanding, at which time it does a direct memory access (DMA) operation across the interface to determine the desired operation and returns (also via DMA) an activity disposition to the processor.

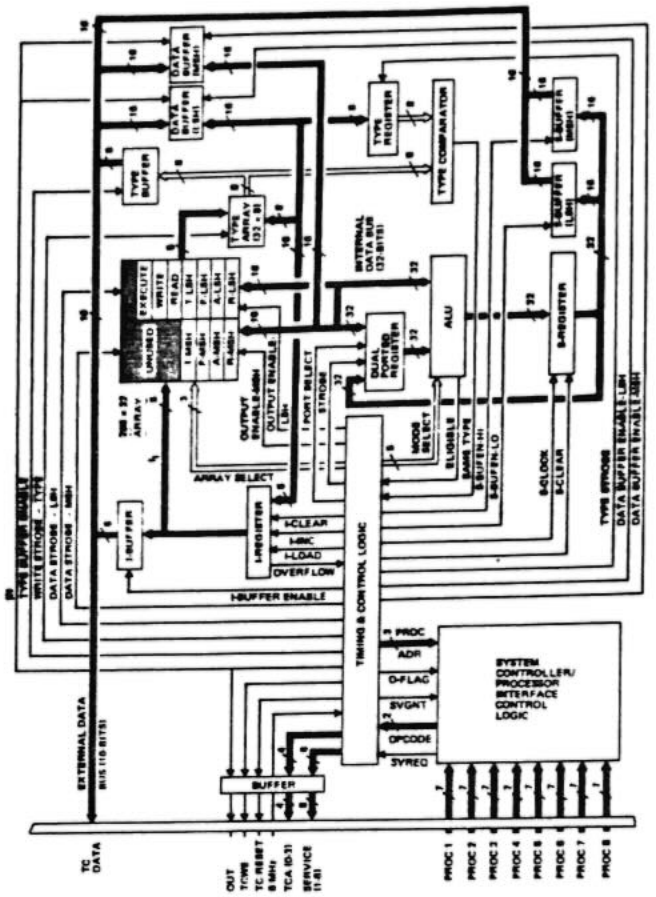

Figure 3: System Controller Block Diagram

The processors and SC operate on an asynchronous request/response basis. Whenever a processor is in need of an activity, it initiates a SC entry transition request. On completion of an application task, the processor returns the variable update vector (V) and initiates an exit transition request which updates the system status vector appropriate for the completed task. The number of active processors can therefore be changed and the system will automatically reconfigure itself with tasks being dispatched to only the active processors.

The processing rate of the SC can be changed under software control. There are two speeds available. The average time required for an entry/exit transition pair is 11.5 microseconds at the high speed setting. The slow speed setting results in an average entry/exit transition time of 115 microseconds. The two modes of operation are used to allow accurate performance data gathering over a wider range of application program execution times.

Dual Processor Modules

There are four wire-wrapped processor modules each of which contains two TI 9900

16-bit microprocessors. (A more detailed description of the SBP 9900 microprocessor can be found in reference [8].) These two processors are multiplexed onto a common bus via time-phase multiplexing. The common bus allows each of the processors to access up to 32K, 16 bit words of shared memory and the RS 232 interface port. Each processor has 4096 words of private EPROM and 256 words of private RAM which are accessible only to that processor. (These private memory addresses are subtracted from the 32K address space of common memory.) The EPROM contains a copy of the monitor/debugger program which is used to support software development on the machine. The RAM is used as a scratch pad area for temporary storage and also provides the interface means between the processors and the SC. The RAM is dual-ported, one port is attached to the processor and the other to the SC. The two devices use a protocol to communicate with each other via DMA to a set of dedicated locations in this private RAM. The EPROM and RAM are both physically located on the dual processor module.

As mentioned above the two processors' address and data busses are connected to single busses for access to common memory. This interconnection approach takes advantage of a TI 9900 memory access idiosyncrasy of initiating memory accesses only on even processor clock cycles. By skewing the system clock pulses for one of the two processors by one cycle, the two processors can access common memory without conflicts. This approach was described in reference [6] for a number of other currently available microprocessors.

Accordingly, there are four phase-one processors and four phase-two processors in the eight processor prototype system. The phase-one processors will never contend for memory with the phase-two processors and vice versa. Memory access conflicts among processors of a given phase can occur, conflict resolution logic having been included on the memory modules for this case.

Shared Memory Modules

There are four identical wire-wrapped memory modules. Together these modules provide 32K words of common memory with an address interleaving scheme which spreads any four sequential addresses across all four memory modules. Each memory module has four access ports, one for each dual processor module. This configuration allows four memory requests to be serviced simultaneously if the four requests are made to different memory modules. Access arbitration logic is used to resolve simultaneous access conflicts among processors to the same memory module. The arbitration logic services the processor with the lowest processor number first when two processors contend for the same memory module.

The interleaved memory has the advantage of randomizing processor memory accesses across the four memory modules. The programmer therefore need not be concerned with explicit distribution of programs and data that may be accessed concurrently among different memory modules. All programs and data are distributed ipso facto among all memory modules.

From a memory contention point of view the resulting system can be be modeled as a four processor by four memory system, with each processor distributing its accesses randomly across the four memory modules. Since for a typical instruction mix, the TI 9900 processor initiates a memory access on about one memory cycle in three, a very small amount of memory contention overhead is encountered. Actual memory contention measurements are described further on.

Peripheral Interfaces

There are only two peripherals in the prototype system, an Intecolor 8001 color terminal and a real time clock. The Intecolor terminal is used as the system console to enter operator commands and also as a display device on which to display system performance data. The Intecolor terminal is interfaced to the multiprocessor system through a serial RS232 interface port which is accessible to all eight processors. The real time clock is used to generate processor interrupts which are used to establish monitoring intervals for gathering the throughput performance data. The real time clock is connected to processor number one, and therefore processor number one is essential to gathering performance data.

DEMONSTRATION SOFTWARE CONFIGURATION

The demonstration software is comprised of two major components, a set of programs to demonstrate qualitatively how software executes in parallel on a multiprocessor system and a set of programs that demonstrates quantitatively the throughput performance obtained from the eight processor research model.

Oualitative Demonstration

In the example program execution demonstration, a parallel flow diagram for the example program being executed is displayed on the Intecolor terminal. The display is comprised of a set of boxes associated with the the application program tasks and a set of lines interconnecting the boxes associated with data conditions used to control the execution order of the tasks. A color coding scheme is used to show the activation status of each task and data condition in the system.

Each task (box on the display) can be in one of four possible states; inactive, eligible, active, or complete. Each data condition (line on the display) has two possible states, true or false. Each of these states is indicated by a unique color on the display.

The demonstration has been set up to allow the operator to execute the example programs with any number of the eight processors active and to display the execution sequence that actually occured during the execution. The display has been designed to scale up the execution time of the tasks so that the viewers can observe details of the processing that has occured. The ability to freeze the display at any point is also provided.

A histogram plot of the number of active and number of eligible tasks in the system during each time interval is also generated. This time-line provides a graphic representation of the processing speed up and processor utilization as a function of the number of active processors.

There are two example programs that have been developed. The first is an Euler coordinate rotation calculation which is used in many aerospace/avionics applications to perform coordinate reference conversions. In this example, the nine element cordinate rotation matrix is computed from the three Euler rotation angles received as input. The second example program demonstrates the application of the Transition Machine architecture to non-scientific applications. A string of up to 12 decimal digits are input by the operator are converted to the text string representation of the number. For example, the digit string 75924 is converted to the text string "seventy five thousand nine hundred twenty four". In this application each of the number groups (billions, millions, thousands, and ones) are processed in parallel. Additional parallelism is realized within each group. The individual strings generated by each of the four groups appear on the screen when the task that generated the string completes.

In developing programs for a parallel machine, the objective is to minimize the calculation duplications and at the same time maximize the number of calculations that can be performed in parallel. A data flow analysis was done manually to meet this objective for the two examples described above. An automated software development support system is described in [9] which automates this data flow analysis of conventional HOL structures and generates an associated set of asynchronous tasks which can be executed in parallel by the Transition Machine.

Quantitative Measurements

The second part of the demonstration is designed to show quantitatively how the throughput performance of the system varies for different application task configurations. A set of synthetic tasks has been developed whose execution duration can be varied and interconnectivity can be modified to simulate any particular task configuration. These tasks are comprised of a delay loop which has been instrumented to calculate various system performance parameters during their execution. The execution time of the synthetic tasks can be varied from a minimum of 306 microseconds to a maximum of 109 milliseconds in 43 microsecond units by varying their loop iteration counts.

There is a copy of the synthetic application task both in common RAM main memory and in the private EPROM of each processor. The operator may select any one of a number of memory configurations. The amount of inter-processor memory contention that is experienced during the program execution will vary appropriately. By locating both the programs and the processor´s workspaces in common memory, memory contention will be maximized. Alternatively, by locating both program and workspace storage in private memory, memory contention will be minimized.

There are four system performance parameters measured; the average number of processors (or portion thereof) executing application tasks, locked out of the SC, being serviced by the SC, and experiencing memory contention. This data could be used to determine the processor utilization and response time for the selected task configuration and number of active processors. It alternatively can be used to determine the critical system design parameters which limit attainable throughput.

The synthetic task configuration utilized in the baseline throughput performance demonstration is comprised of nine independent jobs. Each job is comprised of three tasks which activate each other in sequence. This configuration ensures that there are always nine concurrent tasks so that processor performance degradation will not be due to insufficient jobs.

MEASURED PERFORMANCE DATA

Figure 4 shows the average number of processors executing the application tasks as a function of the number of active processors. The family of curves corresponds to synthetic task execution times varied from 306 microseconds to 109 milliseconds. The number of processors executing application programs represents the useful computing capabilities obtained from the system. As shown in the plot, the maximum number of processors in application programs grows substantially as the average task execution time is increased. The measurements plotted in figure 4 are shown in table 1. This table includes measurements of the number of processors in each of the other processor states as well.

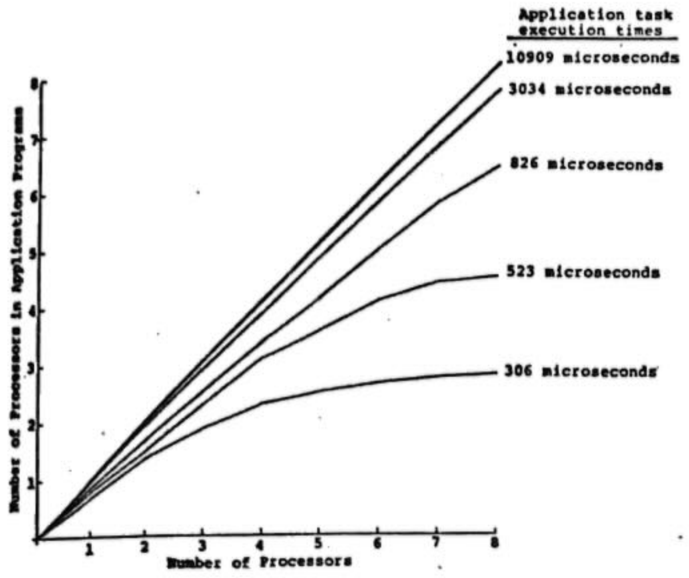

Figure 4: Number of Processors in Application Programs

TABLE 1: Throughput Performance Data

	1	2	3	4	5	6	7	8
MC	0.0	0.0	0.0	0.0	0.01	0.01	0.01	0.02
LT	0.11	0.19	0.53	0.98	1.70	2.45	3.33	4.25
OT	0.24	0.48	0.66	0.81	0.88	0.95	0.98	1.00
AT	0.66	1.33	1.81	2.22	2.41	2.59	2.68	2.74
TASK TIME = 306 microseconds								
MC	0.0	0.0	0.0	0.0	0.01	0.01	0.02	0.02
LT	0.07	0.23	0.29	0.36	0.77	1.15	1.77	2.64
OT	0.16	0.31	0.48	0.64	0.75	0.86	0.92	0.94
AT	0.77	1.46	2.23	3.00	3.48	3.98	4.29	4.39
TASK TIME = 523 microseconds								
MC	0.0	0.0	0.0	0.0	0.01	0.01	0.02	0.03
LT	0.06	0.11	0.20	0.24	0.40	0.47	0.58	0.90
OT	0.11	0.23	0.34	0.45	0.55	0.66	0.77	0.85
AT	0.82	1.66	2.46	3.30	4.04	4.85	5.64	6.23
TASK TIME = 826 microseconds								
MC	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.01
LT	0.01	0.03	0.06	0.09	0.10	0.12	0.14	0.15
OT	0.04	0.07	0.11	0.14	0.18	0.21	0.25	0.28
AT	0.95	1.90	2.84	3.77	4.72	5.66	6.61	7.56
TASK TIME = 3034 microseconds								
MC	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
LT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01
OT	0.0	0.0	0.0	0.0	0.01	0.01	0.01	0.01
AT	1.0	2.0	3.0	3.99	4.99	5.99	6.99	7.99
TASK TIME = 10909 microseconds								

Figure 5 shows a plot of the measured number of processors in each state as the number of active processors is varied from one to eight. This data was generated with the synthetic task configuration described above with the average task execution time of 306 microseconds. All active processors were executing a single re-entrant copy of the synthetic task resident in common memory.

The primary cause of the diminishing return in throughput capabilities as processors are added to the system is the increasing number of processors locked out of the SC. Once a processor is being serviced continuously by the SC, no additional processing capability is obtained by adding processors to the system.

This performance data agrees with the predictions of an analytic model of the sources of overhead in tightly coupled multiprocessor systems[10]. According to the model, the parameter which determines the point where the performance bend over occurs is:

$$\rho = A/o.$$

Here A is equal to the average application task execution time and o is the sum of the processing times of the critical portions in the control mechanism required to service the application task requests. As the number of active processors is increased, the number of processors in application tasks will asymptotically approach:

$$\tau * \rho.$$

Where τ is equal to the number of requests that can be serviced concurrently by the control mechanism.

The SC in the research model can service only one processor request at a time (τ=1) and can operate at two different

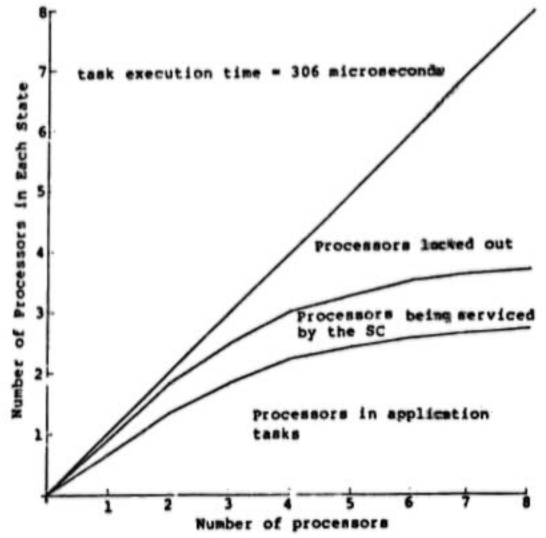

Figure 5: Number of Processors in Each State

speeds. At the high speed setting, the average entry/exit transition pair (O) takes 11.5 microseconds and at the low speed setting it takes ten times longer or 115 microseconds. The performance data shown in the plot of figure 4 was generated with the SC operating at the low speed setting and as shown in the plot, the number of processors in application programs approached T*A/O, as predicted by the model.

To generate the performance plots corresponding to the SC operating at its rated speed, the average application task execution time can simply be multiplied by a factor of 10. In other words, decreasing O by a factor of 10 is equivalent to increasing A by the same factor. This means that for the minimum application task execution time (306 microseconds) and the SC operating in the high speed mode, there will be 7.56 of the eight processors executing application programs on the average. The slower speed option was implemented only for demonstration purposes in order to show the performance bend over phenomena.

This performance data can be compared to the performance data obtained for an implementation of a software executive program for the research model[1]. The software executive program has an average entry/exit processing time of about 1 millisecond as compared to the 11.5 microsecond processing time of the SC. The added efficiency of the SC over its software counterpart allows a much larger number of processors to be effectively combined.

It should be noted that there is virtually no overhead associated with memory contention in the eight processor research model. Memory contention overhead for various processor/memory interconnection schemes has been modeled in a number of papers [4,7,10]. These papers indicate that memory contention in tightly coupled multiprocessor systems need not be a major source of overhead.

There are seven architectural features which have been utilized to limit the amount of memory contention experienced. In summary, the architectural design features described previously have virtually eliminated memory contention as a source of overhead in the research model. The performance degradation experienced as additional processors are added to the system can therefore be attributed to overhead sources other than common memory access contention.

CONCLUSION

The performance data that has been obtained demonstrates the practicality of the Transition Machine architecture and verifies the accuracy of an analytic model [10] which predicts the performance of tightly coupled multiprocessor systems. Together, these demonstrated features lend credence to claims for effectively combining large numbers of microprocessors in tightly coupled configurations using Transition Machine concepts.

The efficiency of the Transition Machine architecture and its amenability to conventional higher order languages programming provide low cost, easily programmed systems which can be utilized to obtain high throughput, fault tolerance and modular expandability. Many advanced aerospace/avionics applications which have processing and environmental requirements that have not been solveable using currently available single processor approaches may now be addressed.

ACKNOWLEDGMENTS

The authors would like to express their sincere appreciation to Phil Corelli who was responsible for the detailed hardware design of the prototype and to Steve Alston who was responsible for hardware integration of the machine. Their expertise was invaluable throughout the development of the system.

REFERENCES

1. Anastas, M.S. "A Multi-Tasking Executive for a Microprocessor-Based Multiprocessor.", M.S. thesis, University of Washington, 1982.
2. ________ and R.F. Vaughan. "Direct Architectural Implementation of a Requirements Oriented Computing Structure." Proc. of MICRO-12. Nov. 1979, pp 93-100
3. ________ and R.F. Vaughan. "Parallel Transition Machines.", Proc. of 1979 Int. Conf. on Parallel Processing. Aug. 1979, pp 76-85
4. Chang, D.Y.; D.K. Kuck; and D.H. Lawrie. "On the Effective Bandwidth of Parallel Memories.", IEEE Trans. Comp. Vol. C-26, No. 5, May 1977, pp 480-490
5. Keller, R.M. "Parallel Program Schemata and Maximal Parallelism." Jour. ACM. Vol. 20, No. 3, July 1973, pp 514-537 and Vol. 20, No. 4, Oct. 1973, pp 696-710
6. Loewer, R. "The Z-80 in Parallel." Byte. July 1978
7. Patel, J.H. "Performance of Processor-Memory Interconnections for Multiprocessors." IEEE Trans. on Computers. Vol. c-30, No. 10, Oct. 1981, pp 771-780
8. Texas Instruments. 9900 Family Systems design and Data Book. Texas Instruments. Huston, Texas. 1978
9. Vaughan, R.F. and M.S. Anastas. "A Software Development Support System for Advanced Avionics Applications Incorporating a Parallel Machine Architecture." Proc. of NAECON ´82. May 1982
10. ________ and M.S. Anastas. "An Analysis of Multiprocessor Throughput Performance in the Limit." Journal of Digital Systems. Vol. 4, No. 2, Summer 1980, pp 153-175
11. ________ and M.S. Anastas. "Microprocessor Based Transition Machines." Proc. of COMPCON Fall 1979. Sept. 1979, pp 327-333