GLOBAL JOURNAL OF ENGINEERING RESEARCH VOL. 6 NO.1 2007: 31 - 38
COPYRIGHT© BACHUDO SCIENCE CO. LTD PRINTED IN NIGERIA. ISSN 1596 – 292X

DESIGN AND PROPERTIES OF A FAULT TOLERANT MULTIPROCESSOR SYSTEM

M. E. EKPENYONG, N. M. UMOH and E. E. EKONG

(Received 15 May, 2006. Revision Accepted 4 July, 2006)

ABSTRACT

The main aim of this paper is to report on the design of a Multiprocessor System, which consists of a number of identical processors connected to a common store This system will continue to function after a hardware fault due to the malfunction of a single processor In this design, both the hardware and software aspects of the reliability problem have been considered. Solutions have also been proffered to other system tolerance problems arising from other kinds of failures by proposing additional properties to the Dual-Bus Multiprocessor Organization A Scheduling State Transition diagram is constructed and a software design for our FTMS is also presented. Both the hardware and software aspect of this design were tested Test result showed that a complete breakdown or failure of systems could be temporarily masked

KEYWORDS: Design Diversity, Standby Spares, Exception Handling, Communication Path, Synchronization Scheme.

INTRODUCTION

Systems are made up of subsystems, which work together to achieve system objectives The cost of system failure is enormous It therefore becomes imperative that we design and build systems that will continue to be operational despite the malfunctioning of a subsystem In this paper, we are considering a multiprocessor system, in which the operation of the system will switch over to a standby spare due to the failure of the default. The objective of this hardware management or short term scheduling is to allocate physical resources to processes as soon as they become available. The design aims to maintain good utilization of processors or equipment and to simulate a virtual machine for each process and a set of primitives, which enables concurrent processes to achieve mutual exclusion of critical regions and communicate with one another

The aim for tolerating a single processor failure is to allocate the primary and backup copies of a task to two different processors such that the backup copy subsequently executes if and only if the primary copy fails to complete due to processor failures. Not all backups need to execute, even in the presence of a single processor failure Since only tasks allocated to the failed processor are affected and need their backup copies to be executed, certain backup copies can be scheduled to overlap with one another

Fault Tolerance - Update

Laprie et. al. (1995) define Fault Tolerance as how to provide, by redundancy, service, complying with the specification in spite of faults having occurred or occurring. They argue that fault tolerance is accomplished using redundancy. This argument is good for errors, which are not caused by design faults. Design- diversity has increased the pressure on the specification of multiple variants of the same equivalent specification to aid programmers in creating variant algorithms for necessary redundancy.

Gray (1991) estimates that 60-90% of current failures are Software failures which implies that a larger focus on software reliability and fault tolerance is necessary in order to ensure a Fault Tolerant System. In his discourse on Software fault tolerance, he described the nature of the Software problem, Current methodologies for solving the problems, and offered some thoughts on future research directions.

DeVale et al(1999) and Knight et al(1986) in their researches held the view that Software errors may be correlated in N-Version Software systems The researches by Knight(1986) are case studies, and may not provide an in-depth account on Software Systems to enable us draw a conclusive result

Storey (1996) in his paper remarked that it is important to understand the nature of the problem that Software Fault Tolerance sets out to solve and concluded that "Software Faults are all design faults "

The traditional Hardware Fault Tolerance was designed to conquer manufacturing faults environmental and other secondary faults. Design diversity was not an applied concept to Hardware fault tolerance solution At this point, N-way redundant systems solved many simple errors by replicating the same hardware. DeVale(1999) has provided evidence on Design diversity and independent failure modes, which have been shown to be a very difficult problem

The difference between Fault Tolerance and Exceptional Handling is that Exceptional Handling diverges from the specification while Fault Tolerance attempts to provide acquiescent services with the specification after detecting a fault

The techniques of Fault Tolerance fall into two categories Software defence and Protective redundancy For systems, which must be available without interruption, the system must dynamically reconfigure itself (Sommerville, 1995) The replacement must substitute the faulty component without stopping the system. Both hardware and software fault tolerances are beginning to face new class of problems of dealing with design faults. Inacio (1998) is of the opinion that Hardware designers will soon face how to create a microprocessor that effectively uses one billion transistors, as part of this task, building a correct microprocessor becomes more of a challenge

Fault-tolerance is considered in the design of real-time scheduling algorithms to make systems more reliable [Liberato et. al. (1999), Alvarez et. al. (1999)]. Liberato et. al. (2000) proposed a feasibility-check algorithm for fault-tolerant scheduling. The well known Rate-Monotonic First-Fit assignment algorithm was extended in Alan et. al. (1999) A delayed scheduling algorithm using passive replica was developed in Ahn et. al (1997) Caccamo et. al. (1998) presented a scheduling algorithm for hybrid task sets consisting of hard periodic tasks However, both of the above algorithms assume that the undelying system is homogeneous. Qin *et al* (2003) investigated an efficient off-line scheduling algorithm for real time tasks with precedence constraints in a heterogeneous environment. The paper provides more features and capabilities than existing algorithms that schedule only independent tasks in real time heterogeneous systems. The proposed algorithm in addition,

M. E. Ekpenyong, Dept. of Maths, Stats. and Computer Science, University of Uyo, P M B. 1017, Uyo, Akwa Ibom State, Nigeria
N. M. Umoh, Dept. of Maths, Stats. and Computer Science, University of Uyo, P M B 1017, Uyo, Akwa Ibom State, Nigeria
E. E. Ekong, Dept. of Maths, Stats. and Computer Science, University of Uyo, P M B 1017 Uyo, Akwa Ibom State, Nigeria

takes heterogeneities of computation, communication and reliability into account, thereby improving the reliability

Short-Term Scheduling

This aims to explain how concurrent processes are scheduled on a computer with one or more identical processors connected on a single internal store as shown in Fig 1

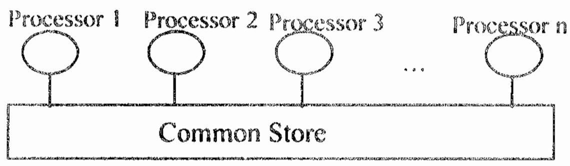

Fig.1: Identical Processors connected to a common store

The number of concurrent processes can exceed the number of processors, but the store is assumed to be large enough to satisfy all concurrent processes at any given time

HARDWARE REQUIREMENT

Let us start by posing two questions that will guide us in our design

Q1. If a processor fails during computation (after allocation of operations), how would we reschedule the process handled by the failed processor? Assuming no process shares the result of computation.

Q2. If another processor requires the result of a failed processor before it starts or before it completes its operation, how would we handle such a situation to avert error?

For the first question, it is assumed that each sub operation has equal priority, where time becomes a scheduling factor

In the second question, the starting operation that will hand-over its result has the highest priority

Before answering these questions in detail, let us review the various architectures of multiprocessor systems

Multiprocessor System Organization

A Multiprocessor system is an interconnection of two or more CPUs sharing common memory and Input-Output equipment. The term "Processor" in Multiprocessor can mean either a Central Processor Unit (CPU) or transport Processor – Input/Output Processor (IOP). However, a system with a single CPU and one or more IOPs is not usually included in a multiprocessor definition unless the IOP has computational facilities comparable to the CPU.

Four interconnection schemes exist: Multiport Memory, Crossbar Switch, Time-Shared Common Bus and Dual-Bus Structure. We shall briefly discuss these organizations to enable us choose a suitable design

A Multiport Memory System employs separate buses between each memory module and each CPU or IOP. This is shown in Fig 2 for four CPUs and four memory modules (mm). Each memory bus is connected to its processor bus

In this organization, memory access conflicts are resolved by assigning fixed priorities to each memory port. The priority for memory access associated with each processor may be established by the physical port position that its bus occupies in each module. Thus, CPU4 would have priority over CPU3 and CPU3 will have priority over CPU2 and with CPU1 having the least priority. This design is not comfortable for our case since there is no common memory

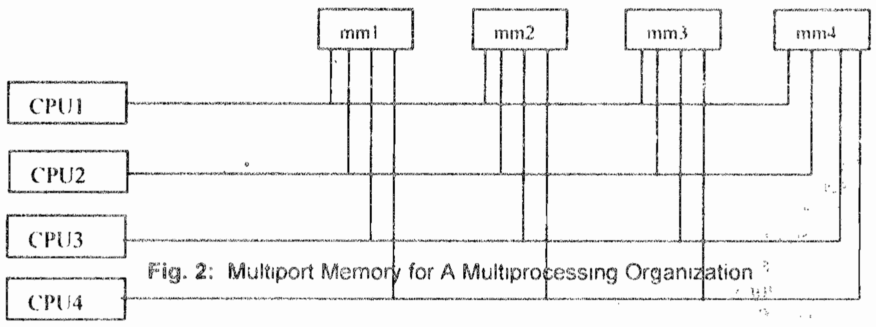

Fig. 2: Multiport Memory for A Multiprocessing Organization

The Crossbar Switch Organization consists of a number of cross points, placed at intersections between processor buses and memory module paths. Fig. 3 shows a crossbar switch interconnection between four CPUs and four memory modules

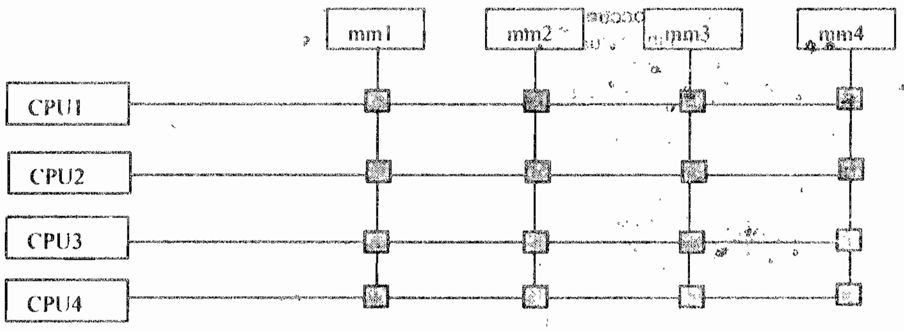

Fig. 3: Crossbar Switch for a Multiprocessor Organization

The organization in fig. 3 also resolves multiple requests for access to the same memory module on a predetermined priority basis. This organization supports simultaneous transfers from all memory modules. However, the hardware required to implement the switch can be quite large and complex

Common Bus Multiprocessor System consists of a number of processors connected through a common path to a memory unit. A Time-Shared Common Bus for four processors is shown in Fig. 4. Only one processor can communicate with the memory at any given time.

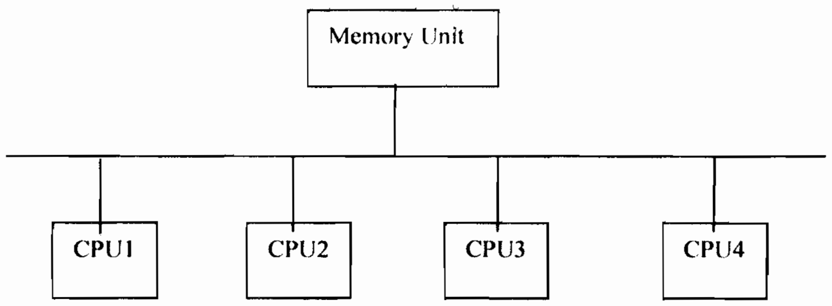

Fig. 4: Time-Shared Common Bus Multiprocessor Organization

Here, the processor that is in control of the bus at the time conducts transfer operations. Any other processor wishing to initiate a transfer must first determine the availability status of the bus. The system may exhibit memory access conflicts since one common bus is shared by all processors. Memory contention must be resolved with a bus controller that establishes priorities among the requesting units.

The design in Fig. 4 can either be tightly coupled or loosely coupled.

A question now arises:

Q3. Assuming a processor while in operation collapses or fails, how would we recover to a great extent, its status?

The Time-Shared Common Bus Multiprocessor Organization fails in this question because, if a recovery is to be made after the failure of a processor (during computation), it is only the assignment status that will be recovered (that defined in the process description table before processing started). Since computation was not complete, temporal results could have been lost. However, it is not a good practice for processors in this design to communicate the common memory so often to store temporal results, since contention may result and communication speed can be unnecessarily slowed.

A single common bus system is restricted to one transfer at a time. This implies that when one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus. The processors in the system can be kept busy more than often through the implementation of a Dual-Bus Structure as shown in Fig. 5. Here, we have a number of local buses each connected to its own local memory and to one or more processors.

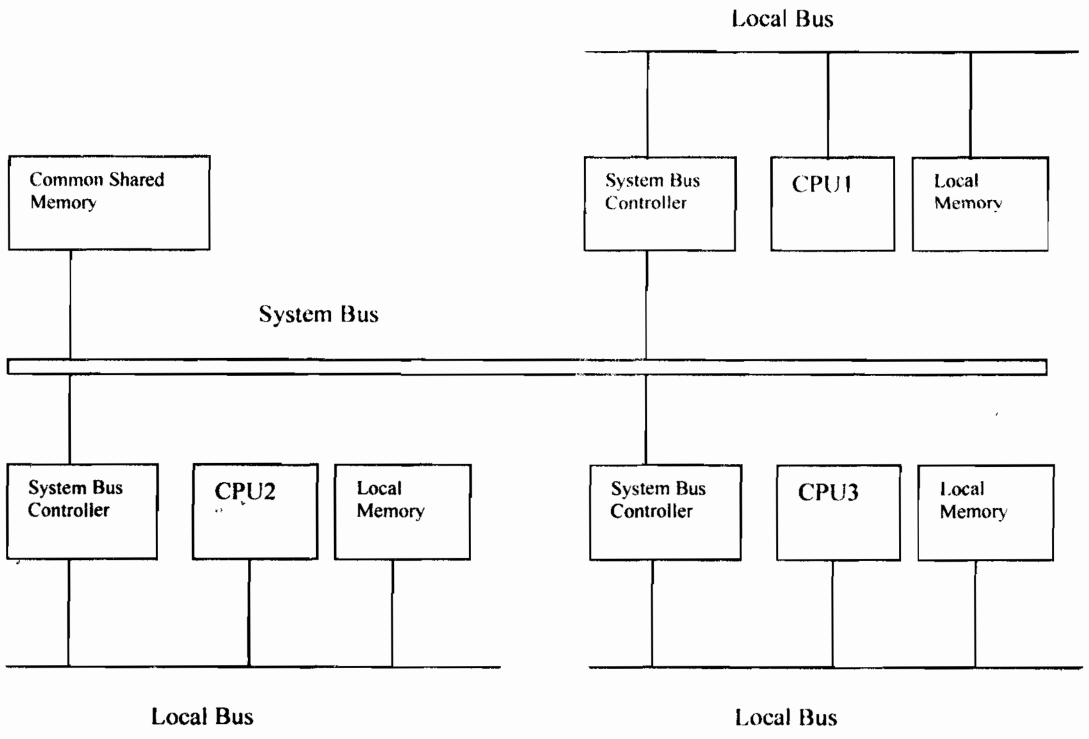

Fig. 5: Dual-Bus Multiprocessor Organization

This design eases the problems of the previous designs. Memory contention is reduced by assigning the longer running processes to specific processors by copying their instructions and data to the local memory of those processors, while communication speed can be kept high by exchanging information through the common store (Mano, 1982).

The answer to Q3 is now feasible, since each processor can store temporarily, the results of its computations in its local memory and if it incidentally fails, its local memory can be recovered or retrieved and handed over to a free processor to complete. A recovery program resident in the common shared memory can accomplish this. The organization in Fig. 5 will be used for our design of a fault tolerant multiprocessor system.

SOFTWARE REQUIREMENT

Fault-Tolerant Software is Software that continues to give correct outputs despite occasional failures in hardware or in parts of the software.

Fault-Tolerant Software design is based on the concepts of Software redundancy. A software module (procedure, function) is redundant if it performs a function identical to some other software module. Design of a Fault-Tolerant system is carried out by carefully placing redundant software modules at critical points in the system so that a failure by a primary module can be recovered and corrected by a redundant standby spare or backup module. We shall in the next section (Software aspect) develop a scheduling algorithm that schedules real-time jobs with dependent tasks at compile time

COMPONENTS AND METHODS

HARDWARE ASPECT

Our Fault Tolerant Multiprocessor System is that which despite a malfunction of a single processor continues to function

To ensure an efficient Fault Tolerant Multiprocessor System, Anderson & Lee(1981) suggested that we also guard against other kinds of failures such as: intermittent hardware faults by the provision of multiple hardware modules (processes with some private memory, main memory modules; buses; input/output access logic).

From these suggestions we here suggest the following components and have come up with a modified design of the Dual-Bus Multiprocessor Organization:

i. Bus guardians in case a bus line fails;
ii. Free processors and memory modules pool to serve as standby spares in case a processor or a local memory fails.
iii. A control switch to (switch) control to the guardians or spares.

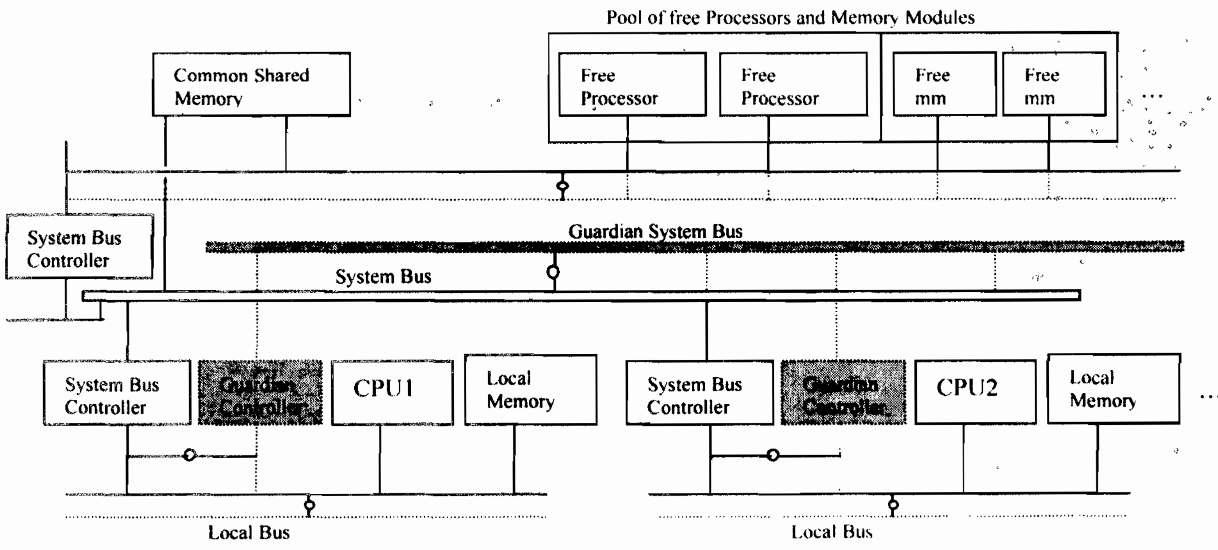

Fig. 6: Construction of a fault tolerant multiprocessor system using suggested components. The dotted lines and shaded units represent guardians.

DISCUSSION

With our current design, guardians can take over data transmission whenever the main bus line fails. A control switch represented by the symbol (-0-) is used to disable a failed line and enable a guardian. With this technique, faults can be masked temporarily. The switch becomes enabled after rectification of the fault.

Also, with a backup of the Common Shared Memory to a free Memory Module in the pool (periodic auto-backup), this design will mask the Common Shared Memory, even when it fails. The system can be programmed to switch control and use the free Memory Module as the Common Shared Memory.

Our current design is a dynamic system, and we must ensure proper communication amongst processors. To obtain this, a communication path needs be established through common input-output channels. The most common procedure is to set aside a portion of the memory, which is accessible to all processors. The primary use of the common memory is to act as a message centre, similar to a mailbox, where each processor can leave messages for other processors and pick up those, which are intended for it.

The sending processor structures a request, a message, or a procedure, and places it in the mailbox. The status bits in common memory are used to indicate the condition of the mailbox, whether it has meaningful information and for which processor it is intended. The receiving processor can check the mailbox periodically to determine whether there are valid messages for it. The response time of the procedures can be significant since a processor will recognize requests only when it does its next polling of messages. A better procedure is for the sending processor to alert the receiving processor directly by means of an interrupt signal. This can be accomplished through a software initiated inter-processor interrupt and can be done by an instruction in the program of one processor which when executed, produces an external hardware interrupt signal in a second processor. This alerts the interrupted processor of the fact that a new message was inserted in memory by the interrupting processor.

To ensure mutual exclusion in our current design, a sequential switching circuit, called arbiter is needed to guaranty that at any instant in time, either the central processor or a peripheral device, but not both, can access the internal store to read or write a single word. If processors try to overlap their access, the arbiter enables one of them to proceed and delay the rest for a few micro seconds it takes to access the store.

So, the machine instructions load and store are implemented as critical regions

```
var store: shared array index of words;
address: index; register: word;
region store do register: = store (address);
region store do store(address): = register;
```

Apart from the above hardware scheduling of access to single store words, there should be another arbiter to which all processors are connected. This arbiter with two machine operations (enter and leave region) implements the hardware aspect of critical regions performed on process descriptions and queues.

CONSTRUCTION OF THE STATE TRANSITION DIAGRAM

Since we are considering the design at the lowest level of programming, i.e. at the descriptive level or architectural level, we here, construct a State diagram that will maintain a good utilization of the equipment and ensure Fault Tolerance. We have assumed three processors CPU1, CPU2 and CPU3 with a common shared memory (though each processor has its own local memory). CPU3 requires the result of CPU2 to complete processing.

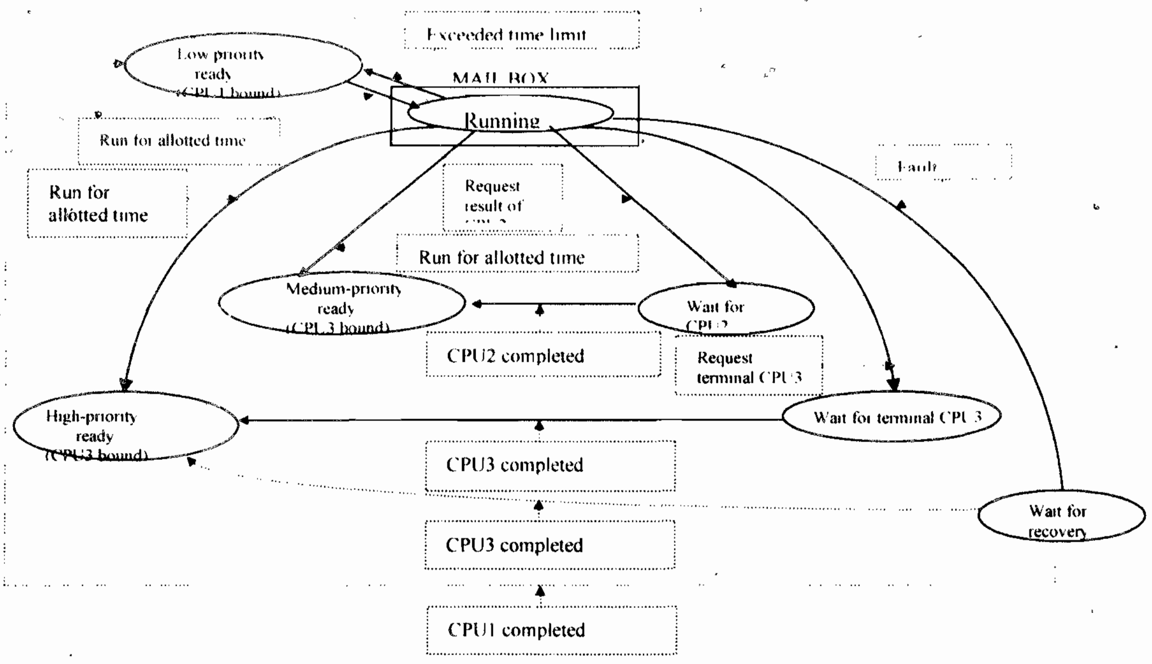

Fig. 7: Scheduling State Transitions for our FTMS Design in Fig. 6.

We have subdivided the ***ready*** and ***wait*** to further control the allocation of processes. The running state indicates the message centre which coordinates the control operations (evaluate messages and results). This scheduling diagram is a simplification of a desired policy that might be used in a timesharing (common store) multiprocessor system. The scheduling policy is to select a process from the high-priority ready list; if there is none, a process is selected from the medium priority ready list. A low priority process is only run if there are no high priority processes.

A fault may result from a hardware or software malfunction and a wait for recovery may mean a change to standby spare(s) or reallocation of job(s) when a failure occurs.

The dotted arc and line from the wait for recovery state represent recovery transitions. A recovery transition is a process where the system uses the spare(s) to finish uncompleted task(s).

SOFTWARE ASPECT

Inacio (1998) has discussed the current methods for software fault tolerance. These methods include Recovery blocks, N-Version programming, and Self-checking software.

The Recovery block and the Self-checking Software schemes will introduce complications to our design. The Recovery block method increases the pressure on the requirement to be specific enough to create different multiple alternatives that are functionally the same and the cost in time of trying multiple alternatives may be too expensive, especially for real time systems. This method requires that each module build a specific adjudicator and the decider. Self checking software is not discussed rigorously in literatures, but it is rather a more ad hoc method used in some important systems. These Software have been implemented in some extremely reliable and safety-critical systems already deployed in our society. They include the Lucent ESS-5 phone switch and the Airbus A-340 airplanes (Lyu,1995). Another hindrance of Self checking Software is that the code coverage for fault toleran system is unknown

We recommend the N-Version programming method for our design. This method employs a majority-voting scheme The reason for this recommendation is that, we can use redundant processors to generate copies of identical inputs. If one or more of the inputs have errors in them, we take the input computed by the majority of the processes and discard the other inputs.

The number of faults that can be tolerated without computing incorrect results can be defined as a function of the number of redundant processes N_p.

Number of faults tolerated $\leq$ floor$((N_p - 1)/2)$; $N_p \geq 3$.

Given 6 redundant processes, each executing identical program and using exactly identical data, then the correct answer will be obtained even if floor((6-1)/2) = 2 processors fail

Suppose $P_1, P_2, \ldots, P_n$ are processes computing a single result for an algorithm. Let $R_1, R_2, \ldots, R_n$ be the values or results returned by $P_1, P_2, \ldots, P_n$, respectively Then, if the results are binaries and let * denote a logical **and**, and + a logical **or**, we have:

Majority result = $(P_1 * P_2 + P_1 * P_3 + . + P_1 * P_n) + (P_2 * P_3 + P_2 * P_4 + \; + P_2 * P_n) + \; + P_{n-1} * P_n$

The N – Version Programming Method

While the recovery block method requires that each module builds a specific adjudicator, the N-version method uses a single decider, which we shall refer to here as a driver This concept attempts to parallel the traditional hardware fault tolerance concept of N-Way redundant hardware. Here each module is made up with N-way different implementations. We shall apply the scheme proposed by Chen and Avizienis (1978) in the implementation of the software aspect of the reliability problem in multiprocessor systems.

The scheme states as follows.

a. Let a program driver invoke each of the versions (program versions)
b. Let the driver wait for the processors to complete their evaluation or execution, and
c. Let the driver compare and act upon the N sets of results

To ensure the implementation of the above scheme, mechanisms are required to synchronize the actions of the driver and the versions and to communicate outputs from the versions to the driver. The scheme also requires that each version be executed atomically and have access to the same input space.

A synchronization mechanism can be implemented based on the use of ***wait*** and ***send*** primitives. The versions ***wait*** and do not commence processing until a ***send*** is executed by the driver. Similarly, the driver waits until ***send*** responses are received from all N versions to indicate that their outputs are complete. The voting check on the received outputs can then be evaluated. There is need for multiprocessor systems to contain a ***complex synchronization scheme***, to ensure that the executions of the versions on multiprocessors do not get out of step.

The hardware requirement to implement this design successfully has been met by our design. Each version could be executed in parallel on independent processors. The driver program should be resident in the common shared memory to control communication, such that after the computation by each independent processor, results are routed to the shared memory for evaluation, where the mailbox serves as a message sender and receiver

Each version during execution must have access to an identical set of input values by being given access right to a read-only, shared, global structure.

Only limited experiments with N-Version Programming have been undertaken. The simple reported experiments (Avizienis & Chen, 1977; Chen & Avizienis, 1978) consisting of running sets of student programs as 3-Version programs on an IBM 360/91, were of a mixed nature The scheme worked for some sets of programs and unfortunately, other combinations of programs failed to provide the required tolerance and in one case, a version caused the Operating system to abort the execution of the 3-Version program of which it was a part. This notwithstanding, N-Version programming remains (at the present stage of the investigation) an interesting and potentially effective approach to Fault Tolerance.

Software Design (Algorithm).

We here present an algorithm that schedules real time jobs with dependent tasks at compile time The objectives of the algorithm are

1 Total schedule length reduced so that more tasks can be completed before their dead lines,
2. Permanent failures in one processor being tolerated.
3. The system reliability being enhanced by reducing the overall reliability cost of the schedule.

Before developing the FTMS algorithm, let us consider some definitions of notation.

NOTATIONS	**DEFINITIONS**
D(x)	The set of predecessors of task x. D(x) = {x \(xi,x) $\in$ E}
S(x)	The set of successors of task x, S(x) = {xi\(x,xi) $\in$ E}
F(x)	The set of feasible processors of which x^B can allocated
B(x)	A set of predecessors of x's backup copy
XQi	The queue in which all tasks are scheduled to pi,s(xq+1)=∞, and f(x0) = 0.
Xqi'(x)	The queue in which all tasks are scheduled to pi, and cannot overlap with the backup copy of task x, where s(xq+1)=∞, and f(x0) = 0
EATi(x,xj)	The earliest available time for the primary or backup copy of task x if message e sent from $v_j \in D(x)$ represents the only precedence constraint.
$EATi^P(x)$	MIN($xj^P \in D(x^P)$) {EATi(x^P,xj^P)}
$EATi^B(x)$	MIN($xj \in D(x^B)$) {EATi(x^B,xj)}
$ESTi^P(x)$	Earliest start time for the primary copy of x on processor pi.
$ESTi^B(x)$	Earliest start time for the backup copy of x on processor pi.
$EST^P(x)$	MIN(pi$\in$P) {ESTi(x^P)}
$EST^B(x)$	MIN(pi$\in$P) {EST(x^B)}

FTMS Algorithm

```
1.  sort tasks by deadlines in accending order, subject to precedence constraints, and generate an ordered list OL;
2.  for each task x in OL, observing the order, schedule primary copy $x^P$ do
    2.1  $s(xp) \leftarrow \infty$ ; $rc \leftarrow \infty$; Xqi = NULL;
    2.2  for each processor pi do /*determine whether task x should be allocated to pi*/
    /*calculate $ESTi^P(x)$, where Xqi = {x1,x2,...,xq} is the queue where all tasks are scheduled to pi, $s(xq+1) = \infty$,
    and f(x0) = 0*/
```

```
        2.2.1   for (j=0 to q+1) do /*compute the EST of x on pi*/
                if s(xj+1) – MAX{f(xj),EATi^P(x)} ≥ ci(x) then /*check if unoccupied
                                                    time intervals interspersed*/
        ESTi^P(x) = MAX{f(xj), EATi^P(x);
end for

        2.2.2   if x^P starts executing at ESTi^P(x) and can complete before d(x) then /*
                                                              determine ESTi*/
                determine the reliability cost of x^P on pi;
        if ((rci<rc) or (rci=rc and ESTi^P(v)<s(x^P))) then /*find maximum rc*/
           s(x^P) ← ESTi^P(x); p← pi; rc← rci; /*assign start time and rc*/
        end for

    2.3 if no proper processor is available for x^P, then return(FAIL);
    2.4 assign p to x, where rc of xp on p is minimal. XQi ← XQi+x^P;
    2.5 update messages information;
end for

3.      for each task x in OL, schedule backup copy x^B  do
        3.1     s(xB) ← ∞; rc← ∞;
/*determine whether the backup of task x should be allocated to pi*/
        3.2     for each feasible processor pi ∈ F(x), subject to proposition 2 and Theorem 2 (Qin(2003)) do
        3.2.1   for (xj∈XQi) do  /*identify already scheduled backup copies on pi that
                           can overlap with x^B */
        if (xj is a primary copy) or ((xj is a backup copy) and (p(xj)=p(x))) then
                /*subject to proposition 1 (Qin (2003)*/
                copy xj into task queue XQi'(x);
        3.2.2   determine whether x^P is a strong primary copy;
        3.2.3   for (all xj in task queue Xqi'(x)) do  /*check if the unoccupied time intervals, interspersed by currently
        scheduled tasks, and time*/
              if s(xj+1)-MAX{f(xi), EATi^B(v)}  /*slots occupied by  backup*/
          ESTi^B(x) = MAX {f(xi),EATi^B(x)} /*copies that can overlap with x^B, can accommodate x^B */
end for
        3.2.4   if x starts executing at ESTi^B(x) and can complete before d(x) then
                               /*determine ESTi based on equation (13), Qin (2003)*/
                        determine rci of x^P or pi;
                if ((rci<rc) or (rci=rc and ESTi^B(x) < s(x^B))) then  /*find the
                                                               minimum rc*/
                                s(x^B) ← ESTi^B(x); p← pi; rc← rci;
                        end if
3.3     if no proper processor is available for x^B, then return (FAIL);
3.4     find and assign p ∈ F(x) to x, where the rc of xB on p is minimal;
        XQi ← XQi+x^B;
3.5     update messages information;
3.6     for each task xj ∈ B(x) do /*avoid messages redundancy*/
                xj sends message to x^B if possible (based on Theorem 1 and
                Expression(1), Qin(2003))
3.7     for each task xj ∈ s(x) do  /*avoid messages redundancy*/
                if P(x^P) ≠ P(xj^P) or x^P is not a strong primary copy then /*based on
                                                        Theorem (3), Qin(2003)*/
                  X^B sends message to xjp if possible;
        end for
        return(SUCCESS);
```

CONCLUSION

The Design of FTMS is at its infancy and lacks sufficient literature. This paper has offered (i) a descriptive design of a FTMS (ii) a State Transition diagram (scheduling diagram) for the designed FMTS (iii) an efficient software algorithm for FTMS design and (iv) has provided a step towards further researches in this area. The design has been found to be efficient for multiprocessor systems. Finally, Fault Tolerance remains the most appropriate approach to masking system faults.

REFERENCES

Ahn, K., Kim, J & Hong, S. "Fault-Tolerant Real-Time Scheduling using Passive Replicas." In Proc of the 1997 Pacific Rim International Symposium on Fault Tolerant Systems, Taipei, Taiwan, December 15-16, 1997

Alan, A., Mancini, L V., Federico Rossini, "Fault-Tolerant Rate-Monotonic First-Fit Sheduling in Hard-Real-Time

Systems," IEEE Trans. Parallel and Distributed Systems, 10(9), pp. 934-945, 1999.

Alvarez, P. M. and Mossé, D. "A Responsive Approach for Scheduling Fault Recovery in Real-Time Systems," Proceedings of fifth IEEE Real-Time Technology and Applications Symposium, canada, pp. 1-10, June 1999.

Anderson, T. and Lee, P. A. 1981. Fault Tolerance, Principles And Practice. Prentice-Hall International, Inc., London.

Avizienis, A. and Chen, L. "On The Implementation Of N-Version programming For Software Fault-Tolerance during Program Execution," Proceedings COMPSAC 77, Chicago (IL), pp. 149-155. November, 1977.

Caccamo, M. & Buttazo, G. "Optimal Scheduling for Fault-Tolerant and Firm Real-Time Systems," 5th International Conference on Real-Time Computing Systems and Applications, Hiroshima, Japan, October 27-29, 1998.

Chen L. and A. Avizienis, "*N-Version Programming: A Fault-Tolerant Approach To reliability Of Software Operation,*" Digest Of Papers FTCS-8: Eight Annual International Conference On Fault –Tolerant Computing, Toulouse, pp. 3-9 (June 1978).

DeVale and Koopman., "An N-Version Approach To Measuring Operating System Robustness," in FTCS-29, 1999.

Gray, J. & Siewiorec, D. P., "High-Availability Computer Systems," IEEE Computer, 24(9): 39-48, September, 1991.

Inacio, C. "Software Fault Tolerance," Carnegie mellon University, 18-849b, Dependable Embeded Systems., Spring, 1998. www ece cmu.edu/~koopman/des sgg/sw fault-tolerance

Knight, J C. and Leeveson, N. G., "An Experimental Evaluation Of The Assumption Of Independence In Multi-Version programming", IEEE Transactions On Software Engineering, Vol. SE-12, No. 1. (January, 1986), pp. 96-109.

Laprie, J C., Arlat, J., Beounes, C. & Kanoun, K "Architectural issues in Software Fault-tolerance," in "Software Fault-tolerance", Lyu, M. R. Ed., Wiley and sons, 1995. pp. 47-80.

Liberato, F.; Lauzac, S.; Melhem, R. & Mossé, D. "Fault Tolerant Real-Time Global Scheduling on Multiprocessors," Proc. of Euromicro Workshop in Real-Time Systems. 1999.

Liberato, F.; Melhem, R. & Mossé, D. "Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard real-Time Systems," IEEE Transactions on Computers, Vol. 49. No. 9, September 2000.

Lyu, M. R., 1995, ed., Software Fault Tolerance. John Wiley and Sons, Inc., Chichester, England.

Mano, M. M. 1982. Computer system Architecture, Second Edition Prentice-Hall, inc., Englewood Cliffs, N. J., U.S.A. pp 454-459

Qin, X., Jiang, H. and Swanson, D. R. "An Efficient Fault-Tolerant Scheduling Algorithm for Real-Time Tasks with Precedence Constraints in Heterogeneous Systems Proceedings of 30th International Conference on Parallel Processing. hhtp://citeseer.nj.nec. com/518704.html

Sommerville, L. 1995. Software Engineering., Fifth Edition., Addison Wesley, Harlow, England.

Storey, N. 1996. Safety-Critical Computer Systems. Addison-Wesley. Harlow, England.