FAULT TOLERANCE FOR MULTIPROCESSOR SYSTEMS VIA TIME REDUNDANT TASK SCHEDULING

Hussain Al-Asaad and Alireza Sarvi
Department of Electrical & Computer Engineering
University of California
Davis, CA, U.S.A.

ABSTRACT

Fault tolerance is often considered as a good additional feature for multiprocessor systems but nowadays it is becoming an essential attribute. Fault tolerance can be achieved by the use of dedicated customized hardware that may have the disadvantage of large cost. Another approach to fault tolerance is to exploit existing redundancy in multiprocessor systems via a task scheduling software strategy based on time redundancy. Time redundancy reduces the expense of additional hardware needed to achieve fault tolerance at the expense of additional computation time, which is more affordable. In this paper we present a general-purpose time redundant task-scheduling scheme for real time multiprocessor systems that is capable of tolerating various hardware and software faults. Our experimental simulation results show that our technique is highly effective, feasible, and promising.

Keywords: Fault tolerance, time redundancy, task scheduling, multiprocessor systems.

1 INTRODUCTION

Real-time systems can be classified as hard real-time systems in which the consequences of missing a deadline can be catastrophic and soft real-time systems in which the consequences are relatively tolerable. In hard real time systems it is important that tasks complete within their deadline even in the presence of a failure. Examples of hard real-time systems are control systems in space stations, auto pilot systems, and monitoring systems for patients with critical conditions. In soft real-time systems it is more important to economically detect a fault as soon as possible rather than to mask a fault. Therefore the main priority is given to mechanisms that are able to detect a fault fast and with the least cost. It is assumed that a proper action can be done immediately after the detection of a fault. In the worst case, the proper action may be replacing the faulty unit with a correct one and repairing the faulty unit off-line. Examples of soft real-time systems are all kind of online transactions processing such as airline reservation, banking, and E-commerce applications. A multiprocessor system can be used to process and provide vital on-line information such as weather prediction or stock market data. Although failure of a processor is not often catastrophic, it cannot be tolerated for a long period of time. Long downtime of a web server can cause loss of business income for companies who rely on that server. It is interesting to note that while the cost of the system itself decreases, the cost for maintenance and downtime penalty grows. Thus failure of a single processor does not have destructive effects as long as the failure is detected with a minimum period of time and ideally with the least cost.

In this paper, we target multiprocessor systems where availability is more important than reliability. The advantage of our approach is that there in no need for hardware redundancy and so the cost for detecting faults will be minimum. It also has the least impact on the speed and performance of each processor. The only drawback is time redundancy, which will be analyzed and evaluated via simulation experiments for different scenarios. The following section reviews some well-known fault-tolerant techniques implemented in different architectures. Section 3 summaries fault-tolerant techniques particularly for multiprocessor systems. Section 4 is dedicated to our approach and the results of our simulations. Finally, Section 5 presents a summary of contributions and a discussion of possible future improvements.

2 BACKGROUND

Computers built in late 1950s had a 12-hour mean time to failure. The maintenance staff of a dozen full-time computer engineers could repair the machine in about eight hours. This failure-repair rate cycle provided 60 percent availability [1]. The vacuum tube and related components were the major sources of failures as they had lifetime of a few months. Thus the machines rarely operated for only more than a day without interruption. Scope of computer's applications has been expanded significantly since then. These applications include real-time control of transportation and communication systems, space flights, automated factories, and even monitoring critically ill patients in hospitals. The fault-tolerance requirements for these applications far exceed the established reliability for previous computer generations. In general, in order to detect a fault and handle it properly, designers use one of the following two techniques:

- Self-Testing: The module performs its own operation and some extra work is done to validate its correct functionality. Built-in Self Test (BIST) and error detecting codes for transferring messages are examples of this approach.
- Comparison: Two or more copies of the same module perform the same operation and a comparator examines their results. If there is a disagreement, a proper action is performed.

Self-testing requires additional circuits or a redesign of existing modules. For example, BIST requires extra three hardware components to a digital circuit: a pattern generator, a response analyzer, and a test controller. However, self-testing using error detecting codes requires a redesign of the system.

Error detecting codes are often used for digital communication and memories to ensure correct transmission of data. However, in case of data modification like arithmetic processing, they often do not have the capability to detect errors efficiently. On the other hand as the integrated circuits become more and more complex designing the customized additional circuits such as BIST to detect the errors of these processors would be more time-consuming, costly and sophisticated as well. Ironically the price of the complicated processor does not increase as much as their complexity does. Thus it would be simpler and more economic to use the comparison technique to detect faults.

In custom fault-tolerant design, 30 percent of processor circuits and 30 percent of processor design time are devoted to self-testing [2]. On the other hand the duplication with comparison technique saves that much effort and time and the result is overall reduction of design and circuit cost. Advantages of the duplication and matching technique include (*i*) low design cost with wide applications, (*ii*) easiness to upgrade for higher reliability without massive technical redesign, and (*iii*) high fault coverage.

Redundancy resources already exist in multiprocessor systems and thus in our method we exploit this redundancy in comparing the results of all pairs of processors. Of course a device that has two identical modules to perform the same operation has twice problems as a device with one module. Thus redundancy by itself provides error detection capability but does not improve availability/reliability. Availability/reliability can be improved if we include some kind of recovery or repair facility. This can be achieved by a mechanism that determines which module is faulty, switches the faulty module off-line, and connects the output of the well-functioning unit to the main system output. Fault detection circuits can be built in the two units to determine the faulty unit in case of disagreement. The original idea of duplication with comparison using different techniques for detecting the faulty unit has been employed by some well-known manufactures such as AT&T, Tandem, and Intel [2][3].

The simplicity, low design cost, and low performance impact on the system, makes the comparison technique more attractive. Duplication with comparison is applicable to all areas and all levels of computer design and therefore is widely used. If the duplicated modules have access to the same common source like a bus or memory then some errors can affect both modules at the same time and faults may not be detected because both copies are in agreement. Thus careful physical separation of the modules is necessary to provide confinement areas and be able to detect overlapping failures.

Time redundancy reduces the need of extra hardware at the expense of spending additional time. In many applications like web servers the trend is to use the available high performance PCs than customized computers with additional extra built-in hardware to provide fault-tolerance. Time redundancy methods, which eliminate the need for extra hardware, have received much attention as an alternative practical

solution. Extra hardware impacts design, power consumption, and cost. On the other hand extra time to detect faults can be tolerated when operations are less critical and urgent.

The simplest approach to increase the reliability of a system using time redundancy is to re-execute an operation on the same functional unit and compare the two results. This method detects transient/intermittent faults but not permanent ones. Other techniques that can detect permanent faults include: Alternating Logic, RESO, Re-computing with Swapped Operands (RESWO) and Re-computing via Duplication with Comparison (REDWC). Alternating logic method [4] can be used for self-dual circuits like full adder. In a self dual circuit complementary inputs provides the complementary outputs and so verification of the circuit can be done easily by comparing the outputs when complemented inputs are given at different times. RESO [5] is suitable for logical or arithmetic operations. Two operations are performed on a single module. The first is a regular operation and the second involves applying operands that are shifted one bit, for example, and the result will be shifted again in the reverse order after the operation. If the circuit is fault free then the results are the same. However if there is a faulty bit slice, its effect will be realized in different positions for each operation and therefore the fault can be detected. The need for additional hardware like the shifter is the main problem in RESO. In RESWO [6], two operations are performed. One regular computation and another one where the operands are swapped and the operation is executed on the same module. A faulty bit slice affects different bit position each time and correct functionality of the module can be verified over time. In REDWC [7] an N-bit module is virtually divided into two units. The two units perform a partial computation involving the least significant $N/2$ bits of the operands in a first time slot and then perform another partial computation involving the most significant $N/2$ bits of the operands in a second time slot. In case of addition the carry out of the first computation is used for the second calculation. In each time slot, the results are compared to check for errors.

The above fault tolerant techniques are applicable to any digital system. Specialized fault tolerant techniques for multiprocessors via task scheduling are discussed next.

3 Time Redundant Task Scheduling

Fault tolerance is particularly relevant to multiprocessor systems since the higher number of processors increases the chance of a fault in these systems. Generally, fault tolerance in a real-time multiprocessor system can be achieved by scheduling multiple copies of tasks. Primary/backup (PB) and triple modular redundancy (TMR) [8][9] are the two basic approaches for task scheduling in different processors. In the PB technique, two copies of a task are executed serially. The backup copy is executed only if the correct result is not generated from the primary task. In this method it is assumed that there is a fault detection mechanism to detect a fault in each processor. In TMR, three copies of a task are executed simultaneously and the result is achieved by a majority vote.

Maode [10] proposed a new time redundant task scheduling algorithm. The basic idea of this algorithm is that the scheduler assigns tasks in a chunk of a fixed size. There is a local queue for each processor and one main backup queue that hold all assigned tasks. When a processor finishes all tasks in its assigned chunk, it notifies the scheduler and that chunk will be removed from backup queue. It is assumed that all faults in the system can be detected and isolated by a hardware mechanism. When a fault appears in a processor, the scheduler reassigns its unfinished tasks to another processor. Customized built-in hardware is required in this method to notify the scheduler for any failure.

Agrawal [11] suggested another time redundant task scheduling algorithm called RAFT (recursive algorithm for fault tolerance). In this algorithm, each task is assigned to two processors and the result is compared. If a matching signature is found, then the result is given to the user. Otherwise, the process of assigning the job to another processor continues until a matching pair is found. The restriction of this method is its large time overhead. At any given time at most only half of the processors can be utilized.

The mentioned approaches target hard real time systems in which it is very important that a task is executed within its deadline. The techniques can be applied when reliability and correct execution of each individual task is the main consideration of system designers. Examples include satellite computers where it is not possible to repair the system on-line,

computers that perform critical operations that cannot be interrupted even for the duration of repair like aircraft computer systems, or computers in which the repair is significantly expensive. On the other hand there are some applications like web servers that may allow some downtime for repair and recovery in case of failure but correct functionality of processors should be verified periodically. We target this kind of applications and propose a new fault tolerant scheme that we describe next.

4 The Proposed Scheduling Scheme

Our fault tolerant time redundant task scheduling scheme targets systems where availability is more significant than reliability. The advantage of our approach is that there in no need for hardware redundancy and so the cost for detecting faults will be minimum. It takes advantage of multiple components of the system itself as redundant resources and performs the same task for each two of them periodically. Thus there is no dedicated checker or backup processor. Instead at any given time a single processor serves as a tester for another processor. Each pair of processors will be compared to each other so that multiple faults at the same time can be detected easily and quickly as well. The only drawback of our approach is time redundancy, which is analyzed and evaluated for different scenarios later in this section.

The algorithm could be applied to any multiprocessor system or any network of computers. Web servers are good examples where our method can be applied practically. Similar to the real world, we assume that the required time for each task is different from job to job. One client may request a web page to be displayed and another client may request a transfer of a file or execute an interactive program. Thus the time for each task is considered to be random.

The main issue of multiprocessor scheduling is to determine when and where a given task should be executed. In our algorithm, some tasks serve as tests for pairs of processors. To explain our scheduling scheme, we assume that we have n processors with the names P_1, P_2, …, P_n. Initially, the first available task is selected as a test and is assigned to the processor pair (P_1P_2). Other tasks are assigned to the remaining processors to achieve the best utilization. If there is an agreement in the responses of P_1 and P_2 to the assigned task, then the response is dispatched to the user. Otherwise, an error is detected and the multiprocessor system needs to take an appropriate action such as repair or reconfiguration of the system. If no error is detected from comparing P_1 and P_2, then we need to test the next pair of processors (P_1P_3). To do so, both P_1 and P_3 need to be available. So, P_1 waits until P_3 finishes its current task and then the first available task is selected as a test for the pair (P_1P_3). The process is repeated for the following pair sequence (P_1P_4), …, (P_1P_n), (P_2P_3), …, (P_2P_n), (P_3P_4), …, ($P_{n-1}P_n$). Once the last pair is tested, we cycle back to testing the first pair (P_1P_2) and so on. With this approach, the existence of more than one faulty processor at the same time can be detected easily.

A sample run of our scheduling scheme is illustrated in Figure 1. In this figure, there is a sequence of seven tasks that need to be executed on a multiprocessor system with four processors. Two tasks served as tests for processor pairs: T_1 for (P_1P_2) and T_6 for (P_1P_3). Note that processor P_1 stops for 5 time units waiting for processor P_3 to become available (shown as W in Figure 1). This waiting time is wasted since the order of testing the processor pairs is fixed.

The primary concern of our approach is to use a method that has the minimum impact on the speed and performance of each processor. For this reason

Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7
Task time	25	30	20	15	30	10	15

Processor	Task scheduling						
P_1	T_1			W	T_6		
P_2	T_1			T_5			
P_3	T_2				T_6		
P_4	T_3		T_4			T_7	

Figure 1 An example of our time redundant task scheduling scheme.

we focus only on fault detection. Our method detects all permanent faults and several transient/intermittent faults. Once a fault is detected, a proper recovery mechanism is initiated to take some appropriate action such as re-execution of the client's request, restarting the faulty system, or repairing the faulty system off-line. The scheduler itself can take over and handle the situation when two processors disagree. In this case, the scheduler can assign the failing task to the first available processor and the correct result will be provided to the user by means of majority voting. Fault masking is thus achieved again without extra hardware.

We have coded our scheduling scheme in a C++ program with a user-friendly interface that asks the user to enter the number of processors, the maximum random time for each task, and the maximum number of jobs. The program then concurrently provides the status of each processor in terms of its availability and the remaining time needed to finish its task. At the end, the total amount of the time spent for regular operation (where a task is assigned to a processor as soon as it is available) and the fault tolerant operation is calculated.

Simulation is performed in two modes. First we assume that there is a chunk of tasks in the queue. We evaluate the required time to execute the batch of tasks when the system performs regularly and when our method of fault tolerance is used. The total time spent is the difference between the time the first task is started and the time when the last task is finished and all processors are available. The results of modeling and simulating the system are given in Table 1. From this table we can see that providing fault tolerance with a limited number of processors has a significant impact on processing time. In other words, as the number of processors increases, the time overhead decreases.

When there are only three processors, dedicating a processor for checking anther processor means loosing almost one third of resources. However when the number of processors is greater, the time difference between the two approaches diminishes and fault tolerant scheduling becomes more advantageous. The maximum task time in Table 1 was fixed to 10 so that the impact of other elements on the whole time spent can be examined more clearly.

Consider having 20 tasks in the queue in which one of them requires 20 seconds and all the rest need 1 second to be completed. In this case 20 processors can process these 20 independent tasks in parallel in 20 seconds. Two processors can do the same thing as well. Thus when the range of job times is enlarged, the impact of loosing one processor on the whole process time is negligible. Actually this is the case in web servers in which their process time differs from simple task like submitting a web page to more time-consuming processes such as transferring a large file. In Table 2, the number of tasks is fixed to 15 and the maximum task time is varied from 5 to 40. When the maximum task time increases, the difference between the processing time with fault-tolerant scheduling and

Table 1 Simulation results of several processor-task configurations using a maximum task time of 10 time units.

Number of processors	Number of tasks	Processing time with fault tolerance	Regular processing time	Percentage time overhead
3	10	29	16	81.25
4	10	20	13	53.85
5	10	13	12	8.33
6	10	9	9	0.00
3	20	56	31	80.65
4	20	33	24	37.50
5	20	27	20	35.00
6	20	20	17	17.65
7	20	17	16	6.25
8	20	16	14	14.29
9	20	15	13	15.38
10	20	13	12	8.33
3	40	102	60	70.00
4	40	65	46	41.30
5	40	48	38	26.32
6	40	39	33	18.18
7	40	35	30	16.67
8	40	32	26	23.08
9	40	27	24	12.50
10	40	24	21	14.29
3	60	165	91	81.32
4	60	102	70	45.71
5	60	75	57	31.58
6	60	62	49	26.53
7	60	52	42	23.81
8	60	45	37	21.62
9	60	40	33	21.21
10	60	34	30	13.33

Table 2 Simulation results of several processor-task configurations with the number of tasks fixed to 15.

Number of processors	Maximum task time	Processing time with fault tolerance	Regular processing time	Percentage time overhead
3	5	19	11	72.73
4	5	12	9	33.33
5	5	10	8	25.00
6	5	8	7	14.29
7	5	8	6	33.33
8	5	7	5	40.00
9	5	5	4	25.00
10	5	5	4	25.00
3	10	39	24	62.50
4	10	29	20	45.00
5	10	22	17	29.41
6	10	17	15	13.33
7	10	16	14	14.29
8	10	15	12	25.00
9	10	12	11	9.09
10	10	11	10	10.00
3	20	57	35	62.86
4	20	37	27	37.04
5	20	28	22	27.27
6	20	23	19	21.05
7	20	21	18	16.67
8	20	19	18	5.56
9	20	18	18	0.00
10	20	18	18	0.00
3	40	156	91	71.43
4	40	82	79	3.80
5	40	79	68	16.18
6	40	71	55	29.09
7	40	55	48	14.58
8	40	48	42	14.29
9	40	42	38	10.53
10	40	38	38	0.00

regular processing time decreases. That is especially the case when the number of processors is not very limited.

As the maximum task time increases, the limited number of processors can be exploited more efficiently than the case when there is greater number of processors. As a matter of fact we may reach to a point that increasing the number of processors does not reduce the whole process time anymore. If we know the average process time for each task and average task number we can achieve the optimum number of processors from the output of our program as well.

Table 3 Simulation results with continuous task arrivals after executing 150 time units.

Number of processors	Maximum task time	Mean arrival interval time	Number of executed tasks with fault tolerance	Number of executed tasks without fault tolerance	Percentage performance degradation
4	20	1	49	70	42.86
4	20	2	47	63	34.04
4	20	3	41	50	21.95
4	20	4	37	38	2.70
5	30	1	41	63	53.66
5	30	2	39	53	35.90
5	30	3	38	49	28.95
5	30	4	37	38	2.70

The above results focuses on the behavior of a system that handles a chunk of tasks that already exist in the queue and analyzes the effects of the number of tasks, the maximum random time of each task, and the number of processors on the execution time. Another situation that can be considered is when tasks arrive continuously. To do so, we assume that tasks arrive with some time distance following Poisson distribution. The scheduler statically assigns the arrived tasks to the scheduled pair of processors and the other available processors simultaneously. Table 3 shows typical results when the simulation is run for 150 time units. It is assumed that the tasks arrive continuously and the mean difference between arrival times ($1/\lambda$) is changed from 1 to 4.

The result of simulation shows that as the difference in time between arrivals increases, the performance difference between the system employing fault-tolerance and the one without fault-tolerance decreases. If we have $(1/\lambda) > (T/P)$, where P is the number of processors and T is the maximum task time, then the tasks arrive in such a way that there is almost always an available processor to take care of the task. In this situation, the processors actually wait for the tasks rather than the tasks wait for available processors and hence there is no accumulated tasks in the queue and the difference in tasks processed by the two systems in a fixed period of time is negligible.

5 DISCUSSION

The scheme presented in this paper for achieving fault tolerance in multiprocessor systems efficiently exploits the redundant resources that already exist in the system and therefore the need for dedicated customized hardware is eliminated at the expense of extra time. The analysis of the simulation data shows the feasibility of bringing the promising method to systems where extra time can be tolerated more than extra hardware like web servers. Our simulation program can also be used to determine the optimum number of processors in different scenarios depending on the characteristics of the tasks.

The presented algorithm is designed to detect individual faults as well as multiple faults that may occur at the same time. It is assumed that proper action will be taken after fault detection. The scheduler itself can take over and act in case of failure. Again without any need for extra hardware the scheduler can reassign the task that cause the disagreement between the two processors to a third processor and sent the correct result to the client by majority voting. This approach can be implemented by providing a backup queue in which all assigned tasks are saved.

An important question in the design of a fault-tolerant multiprocessor system is where the scheduler is executed. One possibility is to execute the scheduler on a dedicated processor. This approach can be easily implemented, however, the problem of a fault in the dedicated processor may be catastrophic. Another approach is to run the scheduler on multiple processors in the multiprocessor system. Hence, once a processor becomes available, it determines the scheduling pattern and then the next available processor verifies the selected pattern. Further research is needed to explore the various scenarios discussed above.

One possible improvement to our scheduling scheme is to reduce the waiting time of the processors by allowing a variable order of processor pair comparisons. To do so, we need an array that records which pairs of processors are compared and the scheduler determines the next processor pair to be compared based on availability of processors and the scheduling array. Further research is needed to explore this possibility and to evaluate it via simulation experiments.

REFERENCES

[1] J. Gray and D. Siewiorek, "High-availability computer systems", *IEEE Computer*, Vol. 24, pp. 39-48, September 1991.

[2] D. Siewiorek and R. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, 1982.

[3] D. Siewiorek, "Architecture of fault-tolerant computers: A historical perspective", *Proceeding of the IEEE*, Vol. 79, pp. 1710-1734, December 1991.

[4] D. Reynolds and G. Metze, "Fault detection capabilities of alternating logic", *IEEE Transactions on Computers*, Vol. C-27, pp. 1093-1098, December 1978.

[5] J. H. Patel and L. Y. Fung, "Concurrent error detection in ALUs by recomputing with shifted operands", *IEEE Transactions on Computers*, Vol. C-31, pp. 589-595, July 1982.

[6] B. Johnson, "Fault-tolerant microprocessor-based systems", *IEEE Micro*, Vol. 4, No. 6, pp. 6-21, December 1984.

[7] B. Johnson, J. Aylor, and H. Hana, "Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder", *IEEE Journal of Solid-State Circuits*, Vol. 23, No. 1, pp. 208-215, February 1988.

[8] G. Manimaran and C. Murthy, "A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, pp. 1137-1152, November 1998.

[9] S. Ghosh, R. Melhem, and D. Mosse, "Fault-tolerant scheduling on a hard real-time multiprocessor system*", Proc. International Parallel Processing Symposium*, 1994, pp. 775-782.

[10] M. Maode and H. Babak, "A fault-tolerant strategy for real-time task scheduling on multiprocessor system", *Proc. International Symposium on Parallel Architectures, Algorithms, and Networks*, 1996, pp. 544-546.

[11] P. Agrawal, "Fault tolerance in multiprocessor systems without dedicated redundancy", *IEEE Transactions on Computers*, Vol. 37, pp. 358-362, March 1988.