On-line Testing and Recovery in TMR Systems for Real-Time Applications

Shu-Yi Yu and Edward J. McCluskey

Center for Reliable Computing – Stanford University, Stanford, California
{syu, ejm}@crc.stanford.edu

Abstract

Triple Modular Redundancy (TMR) is known to improve reliability in real-time computing systems for short mission times. However, TMR-based systems are not effective for longer missions. For failures caused by transient faults, we have designed a new recovery scheme for TMR systems so that they can be used for long mission applications. The scheme can effectively recover computing systems from single transient faults without introducing any re-computation delay; hence, it is suitable for real-time applications. A robot controller is used as a case study. Theoretical analysis and implementation results show that, with very small hardware overhead for recovery logic, the proposed scheme can significantly improve system reliability and lengthen its lifetime. A new state restoration scheme for recovery is presented and shown to have lower overhead than a conventional restoration scheme.

1. Introduction

Many real-time computing systems are used under hazardous or remote circumstances, such as in nuclear power plants [1], aircraft, and spacecraft [2]. In these environments, computing systems are highly susceptible to errors due to radiation. Maintenance and repair is usually very expensive and time-consuming for these applications. In addition, timing is a critical issue because each transaction has to meet strict deadlines. Hence, fault tolerance with minimum time overhead becomes a requirement in real-time systems.

One approach to provide fault tolerance in real-time systems is through redundancy, such as N-modular redundancy (NMR) or duplex pairs. An NMR system replicates a computing resource into N modules running in parallel and uses voters to mask errors at outputs. Therefore, NMR can successfully tolerate faults as long as they happen in no more than $\lfloor N/2 \rfloor$ modules. A duplex-pair system uses two pairs of duplicated modules, which are four in total, for fault tolerance [3]. A module is duplicated for error detection purpose, and the duplex is used in pairs so that when an error is detected in one of the duplex modules, the output from the other would be selected as the output and the system could still continue producing correct data. Although NMR and duplex pairs can increase reliability of systems, they are only effective for short mission times unless repair is possible. Several techniques have been developed to lengthen mission times in NMR systems. For permanent faults, self-purging redundancy [4] and NMR/Simplex [5-6] have been proposed to lengthen system lifetimes. Permanent fault repair has also been implemented in duplex pair systems, such as Stratus computers [7]. For transient faults, the Draper Laboratory has built a real-time computing system with quadruple processors with transient fault recovery mechanism to improve reliability [8-10]. A quadruple system is good for preventing all single point failures including Byzantine failures[1] in distributed systems. Another possible solution is to use a duplex pair system that can easily be recovered from single failures by copying data from a correct duplex subsystem to the corrupt one. However, for a non-distributed computing system with no Byzantine failures, a TMR system, which has lower area overhead than a quadruple system and a duplex pair system, can be a better choice. Since TMR is extensively used in many applications [11], there is a need of designing suitable transient error recovery schemes for TMR systems.

Transient error recovery is achieved by restoring correct states to continue operation. One way to restore correct states is through re-computation, which is called *rollback recovery* [12]. In rollback recovery, system operation is backed up to some point in its processing, which is called a *checkpoint*. When an error occurs and is detected during operation, the system will restore its state back to the previous checkpoint and re-compute. Re-computation has been used in TMR systems for recovery [13]. However, re-computation has time overhead. Another way to obtain correct states is called *roll-forward recovery*, which copies correct states from a fault free redundant module or a spare to the faulty module to avoid re-computation. Roll-forward recovery schemes have been used in duplex systems with spares [14-15] and quadraplex systems [9]. However, duplex systems with spares do not provide sufficient redundancy and roll-forward in these systems still has potential to cause re-computation even for single faults. A quadraplex system provides enough redundancy so that no single fault will require re-computation. But the cost of a quadraplex system is high. In addition, while states in the faulty module are being restored, other fault-free modules are still operational [9]. This can be used in a quadraplex system but may not be suitable in a TMR system. The

[1] A Byzantine failure is a failure where a faulty processor continues execution and gives misleading information [11].

reason is that in a quadraplex system with a faulty module being restored, three modules are left for execution; but in a TMR system when a faulty module is being restored, only two modules are operational and cannot provide error masking.

We present here a new approach to provide recovery for transient errors in TMR systems. Our recovery scheme uses a roll-forward approach to avoid re-computation and to meet strict transaction deadlines. By exploiting redundancy in TMR, no re-computation is needed for single module failures. Hence, it is very suitable for real-time applications. The three modules are synchronized during recovery, so error masking is always preserved during computation.

This paper is organized as follows. Section 2 defines the problem and describes our on-line testing and recovery scheme. Section 3 shows a reliability analysis of the roll-forward recovery system. We calculate reliability of a TMR system with the recovery scheme, and compare it with TMR systems without recovery. Section 4 describes the implementation of the system. Hardware overhead introduced by the recovery scheme is shown in this section. Section 5 concludes this paper.

2. On-line Testing and Roll-Forward Recovery

2.1 Problem Definition

Consider a computing module that has combinational circuitry and storage elements. The computing process in the computing module is a real-time task and each transaction has to be finished before its deadline. TMR is used to provide error masking for the module. During the module's operation, transient faults may occur in the computing module. An error may be latched into storage elements and the system may end up being in an incorrect state producing erroneous outputs in successive cycles. Our purpose is to correct the erroneous data residing in storage elements through recovery, so that the module lifetime is not stopped by a transient fault. In addition, the performance impact caused by the recovery process has to be minimized, so that transactions' deadlines could still be met.

Our scheme is suitable for recovery from single transient faults in computing modules. The scheme can be used in conjunction with permanent fault repair schemes such as self-purging redundancy [4] to repair systems from permanent faults.

2.2 On-line Testing for TMR

In this section, we describe on-line testing used for TMR systems, and describe different forms of TMR in which we implement our roll-forward recovery scheme.

Two forms of TMR are discussed in this section. One is *hardware redundancy*, and the other is *multi-threading* [16-17]. In hardware redundancy, the hardware is replicated with two copies. A voter is used to determine the final output among the three modules. To add an on-line testing capability, voted outputs are compared with module outputs. On-line testing for TMR is shown in Fig. 1. The outputs of each module, noted as Out0, Out1, Out2, are compared with the voted output. If they are different, error signals, ERR0, ERR1, ERR2, are generated so that the faulty module is identified. Disagreement detectors have been used in previous schemes such as hybrid redundancy [18] to remove a faulty module from the system.

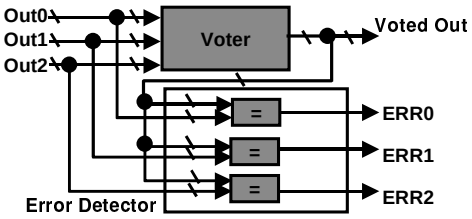

Figure 1. Voter/error detector for on-line testing in TMR.

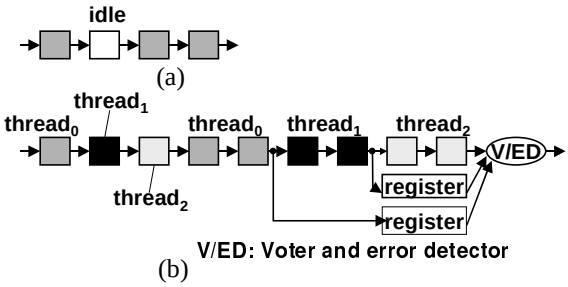

Figure 2. Multi-threading: (a) one-threaded computation, (b) three threads for fault tolerance.

Multi-threading is illustrated in Fig. 2. Figure 2 (a) shows an algorithm executed in multiple stages reusing hardware. An example of hardware reuse is arithmetic logic units (ALUs) in microprocessors. Different arithmetic operations may use the same ALU at different times for computation. For algorithms executed in multiple stages, some resources may be under-utilized (idle) because of data dependencies and memory latency. To obtain fault tolerance by multi-threading, three copies of the same algorithm are executed as different threads in the hardware. As shown in Fig. 2 (b), the replicated instructions of redundant threads use hardware resources which are originally idle. The final output is obtained by voting among the results from the three threads. When transient faults are present, the redundant threads provide a means for fault masking. To add on-line testing capability, the voter/error detector shown in Fig. 1 is used.

Multi-threading is not suitable for systems in which no resource is idle at any time. In these systems, we cannot take advantage of idle resources for redundant computation. Therefore, another form of TMR needs to be used.

Hardware redundancy has small time overhead. In addition to transient faults, it is capable of tolerating permanent faults. However, it is expensive in terms of hardware overhead. Redundancy obtained by multi-

threading is as effective as hardware redundancy for tolerating transient faults, where multi-threading might have smaller hardware overhead since hardware is reused. However, computation latency in multi-threading may be increased. The amount of time overhead in multi-threading depends on the resource usage and data dependency, and it varies among applications. In a computation that has high hardware utilization, a recovery scheme with multi-threading is not necessarily suitable to replace rollback recovery, since a three-threaded design may cause delay longer than simple re-computation after each checkpoint at which there is an error.

Since our recovery method is not restricted to a particular redundancy technique, throughout this chapter we will use the term *copy* to describe both original and replicated modules or threads. For hardware redundancy, a copy means a replicated hardware *module*, and for multi-threading, a copy means a replicated computation *thread*. In the following discussion, we use the notation *HR* for hardware redundancy, and *MT* for multi-threading.

2.3 Recovery Scheme

To determine if there is need for recovery, three computation copies are compared and the results are examined at pre-scheduled computation points. We define these computation points as *checkpoints*. At checkpoints, if comparators detect disagreement among three copies, a recovery process will be initiated. Inputs are buffered and will be processed when the recovery process finishes.

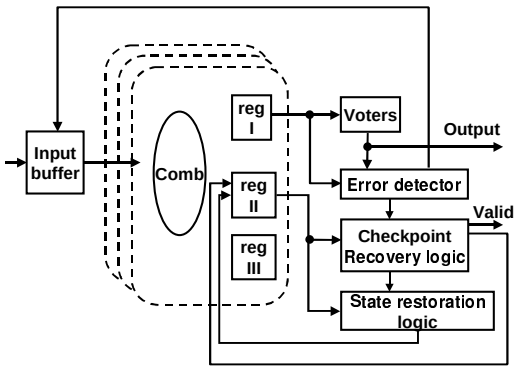

Figure 3. The block diagram of a TMR system with the roll-forward recovery scheme.

Figure 3 shows a block diagram of a TMR system with recovery mechanism. The system consists of three hardware copies or three computation threads, an input buffer, a voter for the output, an error detector, a checkpoint logic/recovery control block, and a state restoration block. In each copy, registers can be divided into three different sets. Register I is the set of registers that contain output data. As shown in the diagram, their outputs are connected to the voter. Register II is the set that contains data that are still needed for computation after checkpoints. Examples for Register II are program counters in microcontrollers and microprocessors. Therefore, data stored in Register II need to be recovered at a checkpoint when a fault happens. As shown in the diagram, these registers are connected to the state restoration logic block and the checkpoint/recovery block. Register III is the set of registers that store intermediate results that are not needed after the checkpoints. They do not need to be recovered when faults are present. Examples for Register III are registers that store intermediate results for subfunction calls. After the subfunction finishes and returns to a main function, these data are not needed anymore. In some applications, Register I and Register II may have a portion that overlaps. That is, some registers that contribute to the output also need to be recovered during recovery.

There are two major reasons to initiate recovery processes at preset checkpoints instead of as soon as a fault happens. One reason is that with pre-scheduled checkpoints and recovery processes, a fault can be recovered after real-time transactions are finished, hence, transactions will not be interrupted or delayed by the recovery process. The other reason is in some applications, checkpoints can be inserted in a way that the amount of internal data or states which need to be recovered is minimized. For example, during a computation task there can be many intermediate values that only contribute to the current task and are not needed for the next task. If we insert checkpoints at completion of tasks, there is no need to recover registers that store these intermediate values since they are not needed anymore. Certainly, if a recovery process is not initiated immediately when a fault happens, extra faults might happen before the recovery process is actually initiated. Because of multiple faults, the system might not be recovered successfully. Longer delay for recovery processes increases the probability of multiple faults. However, a recovery process that can be initiated at any clock cycle is too expensive in terms of both hardware and time overhead. Therefore, recovery processes are scheduled at checkpoints.

During the system's normal operation, all copies execute their functions normally. System outputs are obtained by voting. At checkpoints, the checkpoint/recovery controller checks the error detection result. Figure 4 shows the logic of checkpoint and recovery. Checkpoints can be inserted by adding an additional *checkpoint state* in the finite state machine or by counting cycles to a pre-defined *checkpoint period*. As shown in Fig. 4 (a), if a checkpoint state is used, when the current state in the finite state machine is the checkpoint state, a checkpoint process is initiated. If a counter is used for checkpoints, the counter value is compared with the checkpoint period, and when they are the same, a checkpoint process is initiated. In a

checkpoint process, error indication signals, ERR0, ERR1, and ERR2 from comparators, are examined to see if there is need for recovery. The logic that initiates a recovery process is shown in Fig. 4 (b). If errors are detected, then a recovery process is initiated. If no errors are detected, the normal process will continue until the next checkpoint. During the recovery process, a signal will be raised to indicate that outputs are invalid, and operation is temporarily stopped. The state restoration block provides correct data to recover states stored in Register II. State restoration will be discussed in detail in the following paragraphs. Incoming data are buffered and will be processed after recovery is completed. When the restoration process is finished, normal operation will continue.

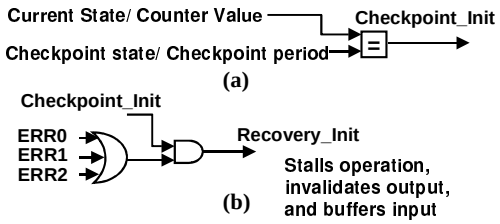

Figure 4. Logic used in the recovery scheme: (a) checkpoint logic, and (b) recovery logic.

The state restoration process is done by restoring internal states in the faulty copy from correct copies. Two schemes for obtaining correct states are described here. They are illustrated in Fig 5. One scheme, the *voting scheme*, is to obtain correct states by voting among the three copies. In [9], voters are used to obtain correct states for recovery. The voted results are loaded into all three copies. As shown in Fig. 5 (a), multiplexers (MUXs) are inserted between registers and signals that were originally connected to the registers. During normal function, original signals are selected to be loaded into the registers. During recovery, voted signals are selected to loaded into the registers. We present a new scheme, the *direct-load scheme*, which obtains correct states by loading states directly from one of the correct copies into the faulty one. As shown in Fig. 5 (b), MUXs are added at inputs of Register II. During normal operation, original signals are selected to be loaded into the registers. When recovery is initiated, the data in fault-free copies are loaded into the faulty registers. If a copy is fault-free, then its register value is held and the content is not changed.

To compare these two schemes, the fault scenarios can be categorized as follows:

(1) Only one copy is faulty: In this case, both schemes can remove incorrect states from the faulty copy.

(2) Multiple copies are faulty, but each faulty copy has incorrect data located at different registers: In this case, since each faulty copy has incorrect data at different registers, the majority voter will always provide correct data for recovery. Then the data are loaded into all three copies to recover faults. Hence, the voting scheme can potentially recover incorrect states successfully in the presence of multiple faults. However, for the direct-load scheme, it could not identify and correct the faulty copy due to multiple faults.

(3) Multiple copies are faulty, but faults are located in the same registers on different copies: In this case, since there are multiple faults, the direct-load scheme cannot recover them correctly. As for the voting scheme, since faulty copies have the same registers that contain incorrect data, the majority voting process is not able to give correct data for these registers. Hence, both schemes will not work in this situation.

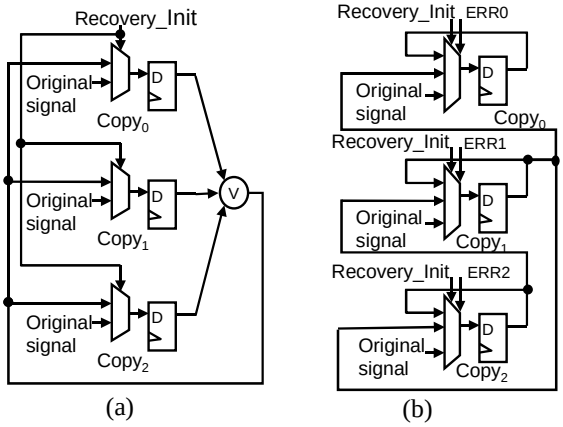

Figure 5. The data restoring process: (a) the voting scheme and (b) the direct-load scheme.

From the discussion of the above fault scenarios, we see that the voting scheme can recover more faults than the direct-load scheme. The direct-load scheme needs larger MUXs to select signals, and the voting scheme needs hardware for voters. We implement these two different schemes in the recovery technique to compare their hardware overhead. To compare reliability between the two schemes, a detailed analysis is required, and it is beyond the scope of this paper. The analysis would require the information of data dependency so that we can calculate the probability of the above three situations.

Figure 6 shows a fault scenario for recovery. As shown in the figure, when a single fault occurs, erroneous outputs are masked by voters. Error detection results are checked at checkpoints by extra hardware as shown in Fig. 4. When a faulty copy is identified, the corrupted data is recovered from correct data in fault-free copies. After recovery normal operation is continued again. As shown in the figure, for single copy failures, the fault-free copies provide sufficient information for recovery. Therefore, no rollback is needed. However, failures in common logic such as voters and recovery logic cannot be recovered. To further improve reliability, a fault tolerance mechanism is added to common logic.

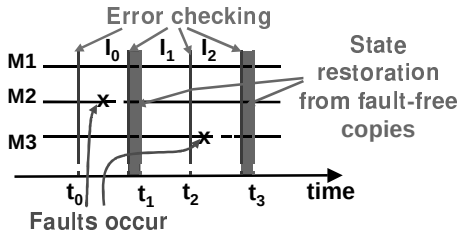

Figure 6. A fault scenario for recovery.

3. Reliability Analysis

Reliability analysis of the roll-forward recovery scheme is shown in this section. A Markov chain is used to model a TMR system with the recovery scheme. To show the improvement, the reliability of the system is calculated and compared with a simplex system, a TMR system, and a TMR-Simplex system. A detailed derivation of the analysis is in the appendix.

We choose a robot controller as our case study to calculate reliability because a robot controller is usually used in real-time applications and is very often life-critical so that it needs fault tolerance mechanisms [19]. A second-order linear algorithm is chosen as the control algorithm for the robot controller [20]. This work is continuation of our ROAR project [21]. In our previous work [17], we have implemented the robot controller in a reconfigurable platform, the WILDFORCE® board [22], and have added fault-masking capability in it via TMR. We have demonstrated that a reconfigurable platform is suitable for implementing robot controllers in terms of both reliability and performance.

Figure 7 shows a block diagram of a robot system [20]. The robot controller collects data of current robot position and its trajectory to calculate needed force. The interface between the robot and the controller includes sensors and A/D, D/A converters. The calculated value is converted to an analog signal, and that signal drives the motor to move the robot.

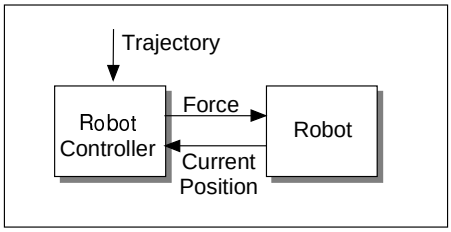

Figure 7. Block diagram of a robot system.

In our analysis, the hardware area and timing data of the robot controller are used to calculate its reliability. The failure probability of the recovery and checkpoint logic is taken into consideration. The robot controller executes the control algorithm, and each computation of the algorithm takes 7 clock cycles. For TMR in hardware redundancy, the whole computation takes 7 clock cycles, which is the same as the simplex design because delay of the voter is small and is not part of the critical path. The three-threaded controller takes 11 clock cycles to complete three computations with the same computation resource as a single-threaded design. Checkpoints are set at the completion of each computation. The recovery process takes one clock cycle. In a larger design with a larger area and more registers to restore when faults are present, more clock cycles will be needed for recovery. The hardware area data is obtained from synthesis using the Synplify® software. We chose the Xilinx Virtex XCV1000 as the target library. The reason we choose an FPGA as the synthesis target for our analysis is that the controller design is going to be implemented on a Virtex FPGA chip in the ROAR project. The unit of synthesized area is CLB (Configurable Logic Block) slices. One CLB slice contains two registers and two LUTs (lookup tables.) We approximate the number of CLB slices as the maximum number of LUTs/2 and regs/2. A detailed description of implementation and synthesis is in Sec 4. The block area and timing information used for reliability calculations is listed in the appendix A.3. Only the voting scheme for state restoration is used in the calculation. The reason is that the voting scheme has larger area in state restoration logic and has larger overhead (which will be shown in Sec. 4). We use the larger area to compute the worst case for reliability for comparison with systems that do not have any recovery mechanism.

Figure 8 shows the reliability of the system with the time axis normalized to mean time to failure (MTTF) of the simplex design. The figure shows that our recovery scheme has much better reliability than Simplex, TMR w/o recovery, and TMR-Simplex systems. Both hardware redundancy and multi-threaded TMR have shown the improvement. In addition, in the hardware-redundant design, the average lifetime, which is the area under the reliability curve, is 9.6 times the lifetime of the TMR design w/o any recovery. However, multi-threaded TMR has slightly lower reliability than hardware TMR, and its lifetime is 8.4 times the lifetime of the TMR design w/o any recovery. This is because in a multi-threaded TMR system, the execution time between checkpoints is longer than that in a hardware TMR system, therefore each thread is more vulnerable to errors between two checkpoints. Moreover, the multi-threading design has larger common logic, such as the thread schedular, that is

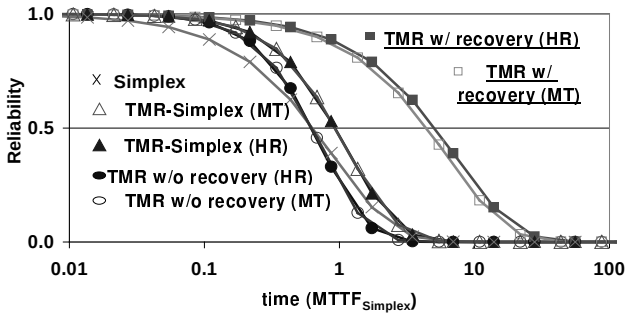

Figure 8. Reliability of the robot controller with different fault tolerance techniques.

shared by the three threads and is not protected by multi-threading. The shared logic can be triplicated to further improve reliability for multi-threading designs.

4. Implementation

In order to estimate the overhead of the roll-forward recovery scheme, the scheme was implemented in the robot controller. The details of the implementation and the synthesized results are described in this section.

We want to investigate the cost of the recovery scheme using different redundancy techniques and different state restoring methods, as mentioned in Sec. 2. We also want to compare its cost with a TMR system without any recovery mechanism. Therefore, the robot system is implemented with different designs as follows:

(1) A simplex system.
(2) A TMR-HR system without any recovery scheme. It contains three copies of the controller and a voter for output results.
(3) A TMR-MT system without any recovery scheme. It contains a controller with three threads of computation and a voting mechanism.
(4) A TMR-HR system with the recovery scheme, and state restoration obtained through voting. The hardware contains three copies of the robot controller, each with extra multiplexers with registers for recovery. It also contains logic for checkpoint control and a voting mechanism for both output and internal states.
(5) A TMR-MT system with the recovery scheme, and state restoration obtained through voting. The hardware contains a controller with three computation threads. The registers that need to be recovered are connected with multiplexers at their inputs. The system also contains logic for checkpoint control and voting mechanism for both output and internal states.
(6) A TMR-HR system with the recovery scheme, and state restoration obtained through the direct-load scheme. The hardware is similar to that in (4), except that it does not have voters for state restoration.
(7) A TMR-MT system with the recovery scheme, and state restoration obtained through the direct-load scheme. The hardware is similar to that in (5), except that it does not have voters for state restoration.

To evaluate the cost when the system is implemented in both reconfigurable hardware and ASICs, the designs are synthesized with target libraries including Xilinx Virtex XCV1000 FPGA, and LSI lcbg10p library, using Synplify® and Synopsys® tools, respectively. Each design is divided into multiple function blocks, and each block is synthesized with timing optimization. The results are listed in Tables 1 and 2. The numbers in parentheses are the area normalized with the result obtained from the simplex design. Results of non-combinational logic (registers) and combinational logic (LUTs) are listed.

From Tables 1 and 2, several observations can be made:

Table 1. Area synthesized using the XCV1000 library.

	TMR Type	Recovery	State Restore	Regs	LUTs
Simp	N/A	No	N/A	402 (1.00)	654 (1.00)
TMR	HR	No	N/A	1206 (3.00)	1979 (3.03)
		Yes	Voting	1242 (3.09)	2323 (3.55)
			Direct-load	1242 (3.09)	2262 (3.45)
	MT	No	N/A	924 (2.30)	1514 (2.31)
		Yes	Voting	958 (2.38)	1819 (2.78)
			Direct-load	958 (2.38)	1906 (2.76)

Table 2. Area synthesized using the LSI lcbg10p library.

	TMR Type	Recovery	State Restore	Non-comb	Comb
Simp	N/A	No	N/A	11619 (1.00)	24404 (1.00)
TMR	HR	No	N/A	34857 (3.00)	73399 (3.01)
		Yes	Voting	35636 (3.06)	76575 (3.14)
			Direct-load	35636 (3.06)	76143 (3.13)
	MT	No	N/A	27273 (2.35)	31432 (1.29)
		Yes	Voting	28002 (2.41)	34740 (1.42)
			Direct-load	27996 (2.41)	34266 (1.40)

(1) Recovery overhead: Recovery overhead, which is the difference between the area of TMR w/o any recovery and TMR w/ recovery, is 6%~9% in sequential logic and 11%~47% in combinational logic, with respect to simplex area. It is small compared to the overhead introduced by TMR (300% in hardware redundancy and 121%~241% in multi-threading).
(2) Combinational (LUTs) vs. non-combinational (registers) overhead: The non-combinational logic overhead of designs with recovery mechanism does not differ much from that of TMR designs without recovery. The reason is that the major overhead for recovery in non-combinational logic is the input buffers, and they are not many compared with registers in a simplex design. The combinational logic overhead varies a lot among different designs and different libraries. This will be discussed in the paragraphs (3) through (5).
(3) Comparison between different libraries: For the same designs, the sequential logic overhead does not differ

much among different libraries (maximum 5%), but the combinational logic varies more (maximum 136%). The combinational overhead obtained from LSI lcbg10p library is smaller than that from XCV1000 FPGA library. The reason is that in XCV1000, most combinational logic has to be synthesized into 4-input LUTs. Hence, the cost for a two-input logic block or a three-input logic block costs as much as a four-input logic block in the FPGA. Multi-threaded designs contain many MUXs in order to connect replicated registers to the same computation hardware. In addition, in the designs with a recovery mechanism, the voters create many three-input logic blocks, and two-input multipliexers are added for the registers that need to be restored. Therefore, in these designs, synthesized results with the FPGA library give more combinational overhead than those with LSI lcbg10p library. The lack of flexibility in logic block mapping in FPGAs induces larger combinational logic overhead. Nevertheless, by giving away the logic block mapping flexibility and logic mapping efficiency, the FPGA provides the flexibility of reconfiguration so that we can exploit it to further repair a design with permanent fault [23].

(4) Hardware redundancy vs. multi-threading: In our case, all multi-threaded designs, with or without recovery, require less hardware overhead than any of the hardware-redundant designs. The reduced sequential overhead is from the shared registers in common logic blocks. The reduced combinational overhead is from the shared computation resources in multi-threaded designs. However, the number of clock cycles of computation in multi-threaded designs is more than those of hardware redundant designs. This shows that in applications where hardware resources are limited but performance requirements are not so strict, multi-threading can be an appealing approach. However, this is not necessarily true for all applications. For some designs that do not have idle stages, multi-threading is not an efficient scheme to implement TMR and can introduce more overhead than hardware redundancy.

(5) State restoration techniques: The direct-load method has smaller combinational overhead than the voted method for state restoration. Their non-combinational overhead is the same. The reduced combinational overhead comes from not using voters in the direct-load method.

Summarizing, we found that the recovery logic introduces small overhead compared with hardware of TMR in both hardware redundancy and multi-threading. Among state restoration schemes, our direct-load scheme has less overhead than the voting scheme. The overhead is less in ASICs than in FPGAs, but with the design implemented in FPGAs we could obtain reconfigurability and more flexibility.

5. Conclusion

In this paper, we have presented a new roll-forward recovery scheme that improves reliability of TMR systems. Unlike other roll-forward recovery schemes, our scheme exploits redundancy in TMR so that no re-computation is needed for single-failure recovery. Therefore, it is very suitable for real-time applications. Reliability analysis of a case study has shown that reliability of a TMR design is significantly improved by adding the recovery scheme. We presented a new state restoration scheme, the direct-load scheme, which is capable of recovering faults in a single faulty module. Synthesis results show that the scheme has smaller overhead than the voting scheme, a conventional state restoration approach. However, it is less capable of recovering multiple faults than the voting scheme. We have applied the scheme to two forms of TMR: hardware redundancy and multi-threading. Synthesis area data show that the recovery mechanism introduces small hardware overhead in both forms of TMR.

Acknowledgements

The authors would like to thank Dr. Nirmal Saxena, Dr. Subhasish Mitra, Dr. Santiago Fernandez-Gomez, and Dr. Wei-Je Huang for their useful feedback and suggestions. This work was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-97-C-0024.

References

[1] Kim, J. H., J. K. Lee, H. S. Eom, and J. W. Choe, "An Underwater Mobile Robot System for Reactor Vessel Inspection in Nuclear Power Plants," *Proc. 6th Int'l Symp. on Robots and Manufacturing*, pp. 351-356, 1996.

[2] Wu, E. C., J. C. Hwang, and J. T. Chladek, "Fault Tolerant Joint Development for the Space Shuttle Remote Manipulator System: Analysis and Experiment," *IEEE Trans. on Robots and Manufacturing—Recent Trends in Research, Education, and Applications*, Vol. 9, No. 5, pp. 675-680, 1993.

[3] Bartlett, J. F., Tandem Computers Inc., Cupertino, CA, "A Nonstop Operating System," *Proc. of the 11th Hawaii Int'l Conf. on System Sciences*, pp. 103-117, 1978.

[4] Losq, J. "A Highly efficient Redundancy Scheme: Self-Purging Redundancy," *IEEE Trans. Comp.* C-25, pp. 569-578, 1976.

[5] Mathur, F. P., "Reliability Estimation Procedures and CARE: The Computer Aided Reliability Estimation Program," *Jet Propulsion Laboratory Quarterly Tech. Review 1*, Oct 1971.

[6] Mathur F. P. and P. DeSousa, "Reliability Modeling and Analysis of General Modular Redundant Systems," *IEEE Trans. Rel.*, R-24, No. 5, pp. 296-299, 1975.

[7] Webber, S. and J. Beirne, "The Stratus Architecture," *Digest of Papers. Fault-Tolerant Computing: Twenty-First International Symposium*, pp. 79-85, 1991.

[8] Lala, J. H., L. S. Alger, R. J. Gauthier, M. J. Gauthier, and M. J. Dzwonczyk, "A Fault Tolerant Processor to Meet Rigorous Failure," *Proc. of IEEE/AIAA 7th Digital Avionics Systems Conf.*, pp. 555-562, 1986.

[9] Adams, S. J., "Hardware Assisted Recovery from Transient Errors in Redundant Processing Systems," *FTCS 19th Digest of Papers*, pp. 512-519, 1989.
[10] Adams S. J. and T. Sims, "A Tagged Memory Technique for Recovery from Transient Errors in Fault Tolerant Systems", *Real-Time Systems Symposium*, pp. 312-321, 1990.
[11] Siewiorek, D. P. and R. S. Swarz, *Reliable Computer Systems – Design and Evaluation*, 3rd Ed, Digital Press, 2000.
[12] Chandy, K. M. *et al.*, "Rollback and Recovery Strategies for Computer Programs," *IEEE Trans. on Computers*, Vol. C-21, No. 6, pp. 546-556, 1972.
[13] Chande, P. K., A. K. Romani, and P. C. Sharma, "Modular TMR Multiprocessor System," *IEEE Trans. on Industrial Electronics*, Vol. 36, No. 1, pp. 34-41, 1989.
[14] Pradhan, D. K., and N. Vaidya, "Roll-Forward Checkpointing Scheme: Concurrent Retry with Nondedicated Spares," *IEEE Workshop on Fault Tolerant Parallel and Distributed Systems*, pp. 166-174, 1992.
[15] Xu, J. and B. Randell, "Roll-Forward Error Recovery in Embedded Real-Time Systems," *Proc. of 1996 Int'l Conf. on Parallel and Distributed systems*, pp. 414-421, 1996.
[16] Saxena, N. R. and E.J. McCluskey, "Dependable Adaptive Computing Systems," *IEEE Systems, Man, and Cybernetics Conf.*, pp. 2172-2177, Oct. 11-14, 1998.
[17] Yu, S.-Y., N. Saxena, and E. J. McCluskey, "An ACS Robot Control Algorithm with Fault Tolerant Capabilities," *IEEE Symp. FCCM*, pp. 175-184, 2000.
[18] Siewiorek, D. P. and E. J. McCluskey, "Switch Complexity in Systems with Hybrid Redundancy," *IEEE Trans. Comp.*, C-22, pp. 276-282, 1973.
[19] Hamilton, D. L., J. K. Bennett, and I. D. Walker, "Parallel Fault-Tolerant Robot Control," *Proceedings of SPIE*, Vol. 1829, pp. 251-261, 1992.
[20] Craig, J. J., *Introduction to Robots: Mechanics and Control,* Addison-Wesley Publishing Company, 2nd edition, 1989.
[21] crc.stanford.edu/projects/roar/roarSummary.html, 2001.
[22] Annapolis Micro Sys. Inc., www.annapmicro.com, 2001.
[23] Huang, W.-J. and E. J. McCluskey, "Column-Based Precompiled Configuration Techniques for FPGA Fault Tolerance," to appear in *FCCM'01*, 2001.
[24] Buchner, S., M. Baze, D. Brown, D. McMorrow, and J. Melinger, "Comparison of Error Rates in Combinational and Sequential Logic," *IEEE Trans. on Nuclear Science*, Vol. 44, No. 6, pp. 2209-2216, 1997.
[25] Trivedi, K. S., *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, Prentice-Hall, Inc., 1982.

Appendix A: Reliability Analysis

A.1 A TMR System with the Recovery Scheme

Reliability analysis for a TMR system with our roll-forward recovery technique is described in this section. Our scheme is designed for recovery of transient faults, therefore, only transient faults are considered in the analysis.

All modules are assumed to be identical and faults are independent of one another. The system will only produce a correct output if two or more of the copies produce the same result and common logic blocks such as the checkpoint controller, the output voter, and the error detector function correctly. Otherwise the system is considered as failed. In addition to the targeted module for error recovery, the reliability of common logic blocks is taken into consideration.

To obtain a lower bound on reliability, we assume that once a transient fault occurs in any component (such as a gate) of a logic block, the logic block fails to produce correct outputs until the error is recovered or is overwritten by further data. The reason is that although a transient fault may only appear in a logic block for a very short period of time, the erroneous data corrupt by the fault can be written into a register and stays. Hence, the faulty data appear as permanent faults until they are recovered or overwritten. We also assume that an incorrect internal signal will always produce erroneous outputs at its following checkpoint and will be detected if the error detector is fault-free.

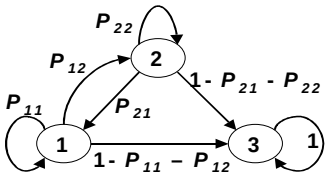

Figure A.1. A state diagram of a TMR system with the roll-forward recovery scheme.

We define a *checkpoint period* as the period between two checkpoints. Figure A.1 shows the Markov state diagram of a TMR system with the roll-forward recovery scheme. State transitions occur at the beginning of every checkpoint period. Assume that normal system operation time and recovery time between checkpoints are constants, so that transition probabilities are constants. Definition of each state is stated as follows:

State 1: At the beginning of a checkpoint period, the system is functioning and all of the three copies are fault-free.

State 2: At the beginning of a checkpoint period, the system is functioning and two of the three copies are fault-free. This happens when one copy becomes faulty during the previous checkpoint period, but a fault re-occurs in the faulty copy during the recovery process. An example of errors that will corrupt computation for the next checkpoint is a bit-flip in a register which belongs to Register II in Fig. 3, during a recovery process. Since faults in Register II will damage subsequent operation, the copy fails to produce correct output at the next checkpoint. The effect of State 2 cannot be neglected when recovery processes are long or recovery logic is complicated.

State 3: The system fails. It happens when faults occur to any of the logic that is shared by the triplicated copies during normal operation or recovery process, or when two or more copies are faulty.

Assume that transient errors occur independently in each unit area in unit time. The system has a single clock

source, and the clock period is constant. We define the transient error rate per unit area per clock cycle as p. Research has shown that transient error rate is different for combinational and sequential circuits [24]. In our analysis, we assume that storage elements are distributed uniformly among combinational circuits; hence, failure rate can be assumed as a constant per unit area per unit time. This can be close to reality if synthesized circuit is homogeneous so that the density of storage elements is approximately uniform. The probability that a logic block with area A functions in a c clock-cycle duration is $(1 - p)^{A \times c}$.

To illustrate all area parameters that will be used in the following discussion, referring to Fig. 3 again for the recovery block diagram. The following symbols denote area of each block: A_M for a module, $A_{REG\text{-}II}$ for registers and related logic in Register II; A_{INP} for input buffer; A_{VOTE} for voters, A_{ECR} for ECR blocks; and $A_{RESTORE}$ for restoration logic. For a multi-threaded design, another parameter A_{SCHE} needs to be included for the area of a thread scheduler, which is shared by three threads.

The following notation is used for timing parameters:

c_{normal}: clock cycle counts for normal operation between two checkpoints.

c_{recov}: clock cycle counts for a recovery process.

$c_{checkpoint}$: clock cycle counts in a checkpoint period. Therefore,

$$c_{checkpoint} = \begin{cases} c_{normal} + c_{recov}, & \text{when there is recovery,} \\ c_{normal}, & \text{when there is no recovery.} \end{cases}$$

To express transition probabilities, the probability of each component being fault-free between two checkpoints are denoted as follows:

P_M: probability that a copy (hardware module or thread) is fault-free during normal operation and error detection between two checkpoints. For a copy with effective area A_M, $P_M = (1-p)^{A_M \times c_{normal}}$.

P_{COMM}: probability that common logic, which is shared by the three copies, is fault-free during normal operation. The duration is c_{normal}. For hardware redundancy, the common logic includes the voter, input buffer, and error detection/checkpoint logic/recovery control. The state restoration is not included because it will only be used during recovery.

$$P_{COMM} = (1-p)^{(A_{INP} + A_{VOTE} + A_{ECR}) \times c_{normal}}.$$

For multi-threading, it also includes the multi-thread scheduler, hence,

$$P_{COMM} = (1-p)^{(A_{INP} + A_{VOTE} + A_{ECR} + A_{SCHE}) \times c_{normal}}$$

P_{REC}: probability that the common logic is fault-free during recovery, c_{recov}, given that an error occurs during normal operation and the recovery control process needs to be initiated. The related logic includes the input buffer, error detection/checkpoint logic/recovery control, and state restoration logic. Therefore,

$$P_{REC} = (1-p)^{(A_{INP} + A_{ECR} + A_{RESTORE}) \times c_{recov}}.$$

$P_{M\text{-}REC}$: probability that no fault occurs to a copy during recovery, c_{recov}, to cause incorrect output for the next checkpoint. The logic that its faults will cause failure to a copy includes all the registers that are accessed during a recovery process, and all related logic such as multiplexers connected to these registers. Thus,

$$P_{M-REC} = (1-p)^{A_{REG-II} \times c_{recov}}.$$

$P_{M\text{-}REC}$ cannot be neglected if the time needed for recovery is not very small compared with checkpoint periods.

The symbol Pij means the probability of a transition from State i to State j. Pij's are described as follows:

P_{11}: probability that the system stays in State 1. This happens when no faults occur to the hardware or when one copy is faulty but is correctly recovered. Hence,

P_{11} = Prob(no faults occur during normal function) + Prob(only one copy is faulty during normal function, and no faults occur during recovery process)

$= (P_M^3 \times P_{COMM}) + (3 \times P_M^2 \times (1\text{-}P_M) \times P_{COMM} \times P_{REC} \times P_{M\text{-}REC}^3)$.

P_{12}: probability of transition from State 1 to State 2, that is, when faults occur in one module during normal function and error detection, and fault occurs to essential logic in the faulty copy during recovery. No other faults can happen to the common logic or other two fault-free copies, otherwise the system will go to State 3. Hence,

P_{12} = Prob(only one module fails during normal function and the recover process is initiated) × Prob(only fault occurs to essential logic in the faulty module, given that the recovery process is initiated)

$= (3 \times P_M^2 \times (1\text{-}P_M) \times P_{COMM}) \times (P_{REC} \times P_{M\text{-}REC}^2 \times (1\text{-}P_{M\text{-}REC}))$.

P_{21}: Probability of transition from State 2 to State 1. This happens when no extra faults occur to the original fault-free copies, and the original faulty copy is recovered successfully. Hence,

P_{21} = Prob(no faults happen to the two fault-free modules and common logic during normal function and the recovery process is initiated) × Prob(no faults happen in the recovery process)

$= (P_M^2 \times P_{COMM}) \times (P_{REC} \times P_{M\text{-}REC}^3)$.

P_{22}: Probability that the system stays in State 2. This happens when no extra faults occur to the original fault-free hardware, and faults re-occur to the original faulty copy. Hence,

P_{22} = Prob(no faults happen to the two fault-free modules and common logic during normal function and the recovery process is initiated) × Prob(only fault occurs to essential logic in the faulty module, given that the recovery process is initiated)

$= (P_M^2 \times P_{COMM}) \times (P_{REC} \times P_{M\text{-}REC}^2 \times (1\text{-}P_{M\text{-}REC}))$.

Reliability of the system is obtained by calculating the probability of the system staying in the functioning

state, State 1 and State 2, at checkpoint n. Let $D_i[n]$ be the probability that the system stays in State i at checkpoint n, where $1 \le i \le 3$ and $0 \le n$. Assume that the system is initially fault-free, which means the system stays at State 1 at checkpoint 0, so that

$D_1[0] = 1$, $D_2[0] = 0$, and $D_3[0] = 1 - D_1[0] - D_2[0] = 0$.

The transitions are expressed as recursive equations,

$D_1[n+1] = P_{11} \times D_1[n] + P_{21} \times D_2[n]$, and

$D_2[n+1] = P_{12} \times D_1[n] + P_{22} \times D_2[n]$.

For a system with fixed checkpoint periods, we use $R[n]$ to denote the reliability at the nth checkpoint. The reliability at checkpoint n is calculated as,

$R[n]$ = Prob (system in State1 at checkpoint n) + Prob (system in State2 at checkpoint n)

$$= \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} D_1[n] \\ D_2[n] \end{bmatrix} = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} P_{11} & P_{21} \\ P_{12} & P_{22} \end{bmatrix} \begin{bmatrix} D_1[n-1] \\ D_2[n-1] \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} P_{11} & P_{21} \\ P_{12} & P_{22} \end{bmatrix}^n \begin{bmatrix} D_1[0] \\ D_2[0] \end{bmatrix} = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} P_{11} & P_{21} \\ P_{12} & P_{22} \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix}. \quad \text{(eq 1)}$$

A.2 Other Systems

In this section, the reliabilities of a simplex, a TMR without recovery, and a TMR-Simplex without recovery, are computed. These systems do not need checkpoints. However, for ease of comparison, we insert hypothetical checkpoints into computation processes. In this case, checkpoint periods are c_{normal} clock cycles. In Markov analysis a state transition happens at every checkpoint, that is, a state transition happens every c_{normal} clock cycles. Continuous time reliability analysis of these systems can be found in [25].

(1) Simplex:

A simplex system only contains a single copy of the computation, and no voting or error detection mechanism is added. P_M has the same definition as that in Sec. A.1.

$R[n]$ = Prob(the system is functioning from the beginning to checkpoint n) = P_M^n.

(2) TMR system without recovery:

In a TMR system without recovery, the original computing system is triplicated, and a voter is used to vote for output results. The state diagram is the same as Fig A.1. P_M has the same definition as that in Sec. A.1. We denote the probability that the output voter with area A_{VOTE} is fault-free during a checkpoint period as P_{VOTE}, and $P_{VOTE} = (1-p)^{A_{VOTE} \times c_{normal}}$.

Probabilities of transitions between states are:

$P_{11} = P_M^3 \times P_{VOTE}$.

$P_{12} = 3 \times P_M^2 \times (1-P_M) \times P_{VOTE}$.

$P_{21} = 0$, because of no recovery.

$P_{22} = P_M^2 \times P_{VOTE}$.

Reliability equation is the same as (eq 1), but with transition probabilities listed above.

(3) TMR-Simplex system without recovery:

A TMR-Simplex system without recovery switches from TMR to Simplex when faults are detected. The state diagram is the same as Fig. A.1. However, for TMR-Simplex we give states different definition:

State 1: This is the TMR state. All three copies and the voter/switch are fault-free.

State 2: This is the Simplex state. The simplex copy and the switch are fault-free.

State 3: The system fails. It happens when the voter or switch is faulty, or when two or more copies are faulty.

P_M has the same definition as that in (2). We denote the probability that the switch with area A_{SWITCH} is fault-free during a checkpoint period as P_{SWITCH}, and

$$P_{SWITCH} = (1-p)^{A_{SWITCH} \times c_{normal}}.$$

We denote the probability that the error detector with area A_{ED} is fault-free during a checkpoint period as P_{ED}. The error detector is used to detect the faulty copy to change TMR into Simplex configuration.

$$P_{ED} = (1-p)^{A_{ED} \times c_{normal}}.$$

Probabilities of transitions between states are:

$P_{11} = P_M^3 \times P_{VOTE} \times P_{ED} \times P_{SWITCH}$.

$P_{12} = 3 \times P_M^2 \times (1-P_M) \times P_{VOTE} \times P_{ED} \times P_{SWITCH}$.

$P_{21} = 0$.

$P_{22} = P_M \times P_{SWITCH}$.

Reliability equation is the same as (eq 1), but with transition probabilities listed above.

A.3 Implementation Data for Reliability Calculation

Table A.1. Area data of the robot controller.

Item	CLB slices
Module (A_M) – Simplex, TMR w/o recovery (HR)	327
Module (A_M) – TMR w/ recovery (HR)	364
Single thread scheduler	4
Module (A_M) –TMR w/o recovery (MT)[2]	323
Module (A_M) – TMR w/o recovery (MT)	360
Input buffer (A_{INP})	16
Voter (A_{VOTE})	8
ECR block (A_{ECR})	17
Three-thread scheduler (A_{VOTE})	6
State restoration ($A_{RESTORE}$)	32
Register II (A_{REG-II})	42
Error detector (A_{ED}) – TMR-Simplex	11
Switch (A_{SWITCH}) – TMR-Simplex	16

Table A.2. Clock cycles for the robot controller.

Item	Clock cycles
Task duration (c_{normal}) – HR	7
Task duration (c_{normal}) – MT	11
Recovery (c_{recov})	1

[2] The area is obtained by subtracting the single-thread scheduler area from the simplex module area.