Experimental Evaluation of Time-redundant Execution for a Brake-by-wire Application

Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson
Department of Computer Engineering
Chalmers University of Technology
S-412 96 Göteborg, Sweden
{aidemark, vinter, peterf, johan}@ce.chalmers.se

Abstract

This paper presents an experimental evaluation of a brake-by-wire application that tolerates transient faults by temporal error masking. A specially designed real-time kernel that masks errors by triple time-redundant execution and voting executes the application on a fail-stop computer node. The objective is to reduce the number of node failures by masking errors at the computer node level. The real-time kernel always executes the application twice to detect errors, and ensures that a fail-stop failure occurs if there is not enough CPU-time available for a third execution and voting. Fault injection experiments show that temporal error masking reduced the number of fail-stop failures by 42% compared to executing the brake-by-wire task without time redundancy.

1. Introduction

Distributed real-time systems are increasingly being used to control critical functions in automotive and aerospace applications, such as fly-by-wire, brake-by-wire and steer-by-wire systems. These systems must be fault-tolerant to be safe and reliable.

Previous research has shown that transient faults are common in digital systems [1]. These faults can be caused by power fluctuations, electromagnetic interference or by particle radiation. Radiation-induced transient faults are mainly a problem in space and at high altitude, however they may also occur at ground level [2]. In addition, as the computer industry strives to reduce both the geometry and the power supply of components, the risk of environmentally induced faults increases.

A cost-effective technique for handling transient faults is to use time-redundancy [3, 4, 5, 6]. The declining prices of high-performance microprocessors and micro-controllers make time redundancy increasingly attractive for achieving fault-tolerance in real-time systems.

We have developed a real-time kernel that mask errors by triple time redundant execution and majority voting. The kernel executes all critical tasks twice and compares the results to detect errors. A third execution is started if an error is detected by the comparison or a CPU-exception. This allows the kernel to mask errors by conducting a majority vote on three results. We call this technique temporal error masking.

The real-time kernel uses fixed priority scheduling to control temporal error masking. Before starting additional executions when an error is detected, the kernel checks whether it is feasible to re-execute the task and meet the deadline. The output of a task is delivered only when two matching results have been produced.

The objective of temporal error masking is to tolerate transient faults at the node-level whenever possible. For permanent faults and transient faults that cannot be handled at the node level, the node must fulfill fail-stop or omission failure semantics [7]. These properties are achieved by combining hardware and software error detection mechanisms with temporal error masking.

In this paper, we present an experimental evaluation of a brake-by-wire application executed by the real-time kernel. We consider a distributed control system, where a brake-by-wire task for each wheel is executed on two fail-stop computer nodes operating in active redundancy. Such a system can tolerate single node failures without degrading the ability of the system to brake the vehicle. If two nodes that control the braking of one wheel both fail, then the system can still brake the vehicle using the other wheels. However, the efficiency of the brakes will be substantially degraded. Therefore, it is important to minimize the probability of double node failures.

Such double node failures can occur as a result of a single external disturbance that cause transient faults in more than one node, or near-coincident transient faults. Temporal error masking improves the systems ability to cope with such transient faults without degradation.

We conducted fault injection experiments to validate the kernel and assess the effectiveness of the temporal error masking. Transient bit-flips were injected into the internal registers and flip-flops of the CPU that executed the kernel. We have previously shown that such bit-flips

in an engine controller may cause permanent failures, such as locking the engine at full speed [8].

The objective of the experiments was to estimate the error coverage with respect to errors that occur during the execution of the brake-by-wire application task. No faults were injected during execution of the kernel code, since no mechanisms for detection and recovery of such faults are included in the current version of the kernel.

The next section presents related work. The principles of temporal error masking are presented in Section 3, while Section 4 presents the brake-by-wire application. Section 5 describes the experimental set-up and Section 6 reports the results of the fault injection experiments. Finally, the conclusions of this study are given in Section 7.

2. Related work

Time redundancy is a well-known technique for achieving fault tolerance [3]. A majority of studies on time redundancy focuses on detecting errors, i.e. to execute an operation twice and compare the results. This is done on different levels such as on the instruction level [4], procedure level [5] or at the task level [6]. Execution of diverse software versions also allows detection of software errors [9]. Recent studies on double execution and comparison utilize new technologies in processors to reduce the time overhead [10].

Although most studies on time redundancy focus on detecting errors, some studies have addressed the use of triplicated execution and voting to mask errors on a single node. Triplicated execution is evaluated in [11], where each software module and the voting mechanisms are executed three times to mask errors.

Another approach using time redundancy is *roll back recovery* [12]. In roll back recovery, the error is detected by various error detection mechanisms and additional time is used to re-execute the failed operation. *Retry* is the simplest scheme, where the failed operation is just repeated. Another scheme is to use *checkpointing*, i.e. the state of the processor is saved at regular intervals or when certain data is updated. If an error is detected, the system is restored to the last checkpoint and the operation is repeated.

The temporal aspect of fault tolerance is addressed in fault-tolerant scheduling. In this scheduling theory, time for recovery is included in the schedule while still guaranteeing that all tasks meet their deadlines. Fixed priority scheduling is extended to include time for re-execution of failed tasks in [13]. However, it is not necessary to reserve time for re-executing all tasks in advance, if one assumes that there is an upper bound on the number of tasks that can be affected by transient faults during a specific time interval. Using several time redundant copies of a task and taking a vote on their output to tolerate transient faults in a static scheduling system is proposed in [14]. However, when time redundancy is used in a static scheduled system, extra time must be pre-scheduled for re-execution of each task.

Our real-time kernel takes advantage of the flexibility in fixed priority scheduling to control triplicated executions of tasks to mask errors. The kernel has been used for executing a brake-by-wire application allowing the temporal error masking technique to be evaluated using fault injection.

3. Temporal error masking

Our real-time kernel allows each critical task to be executed twice during normal operation. The term *copy* is used to denote a particular instance of the task execution. The results of the two copies are compared to detect errors. If the results match, a third copy does not have to be executed and the time can be used by other tasks. If the two results do not match, a third copy of the task is executed. The results of the three copies are then checked by a majority vote. If the majority voter detects two matching results, they are accepted as a valid result of the task, otherwise no result is delivered, which leads to an omission failure. Errors can also be detected by hardware and software EDMs. In this case, the affected copy is immediately terminated and a new copy is started, see Figure 1.

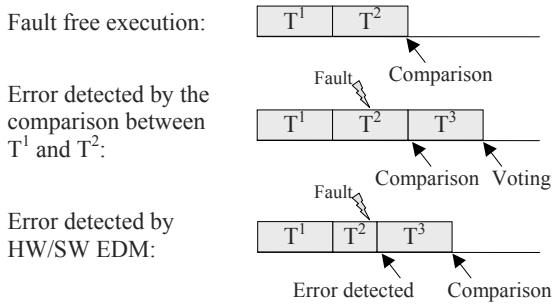

Figure 1. Error Detection and Recovery

After an error is detected, the kernel checks the deadline of the task to determine whether it is possible to execute an additional task copy before the deadline. If not, no result is delivered and an omission failure occurs. The output from the non-faulty node must then be used. If time is available, a new copy is started. The task result is delivered only when two matching results have been produced before the deadline.

We assume that enough slack is available in the task schedule to allow at least one task to execute three copies without causing any other task to miss a deadline. However, if several tasks are affected by near-coincident transient faults, sufficient slack may not be available to allow all of them to execute three copies without causing

other tasks to miss their deadlines.

The dynamic behavior of the technique enlarges the output jitter. However, this may be solved by adding a separate task responsible for delivering the results of the tasks. This extra task executes with an offset in time from the periodic task performing the actual computation [15].

3.1. Policies when recovery is not possible

Depending on the requirements of the application, different policies on how to deal with unrecoverable errors may be chosen. For example, a feedback control system is able to withstand a certain delay in the delivery of the control signals without losing the stability of the system [16]. In addition, some systems may even tolerate value failures due to the inertia of the controlled object. In [8] faults were injected in an engine control algorithm and almost 90% of the value failures produced were minor failures, i.e. failures with no noticeable impact on the engine. Therefore, if an error cannot be recovered: (a) an omission failure may occur, (b) instead of an omission, the previous result may be delivered or (c) if a HW exception occurs in the first or second execution, it may be assumed that the fault affects only that execution. The result from the non-affected execution can thus be delivered. To inform the receiver of the uncertainty of the result, quality information can be included in the message. A first-class result is a verified result, i.e. the result has been produced twice and the two results match, while a second-class result is a result from an execution that is assumed to be unaffected by the fault. A third-class message contains a previous result.

4. Brake-by-wire application

Brake-by-wire systems are expected to replace hydraulic brake systems in future road vehicles. In a brake-by-wire system, the driver's brake intention is transmitted electronically from the brake pedal to electro-hydraulic or electro-mechanic brake actuators positioned on each wheel.

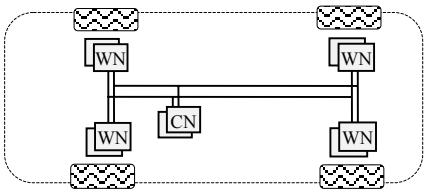

Figure 2. Brake-by-wire system

Some advantages of brake-by-wire are simplified assembling and service of the brake system. There are also environmental advantages as no hydraulics system is used. In addition, the brake-by-wire approach simplifies adaptation of assistance systems, such as ESP (electronic stability program). A distributed architecture for such a system can be implemented according to Figure 2, where the driver's brake intervention is sent to a central node (CN). The central node handles the all-embracing control, distributing the correct brake force to each wheel node (WN). The individual wheel nodes control that the requested brake force is applied to the respective wheel. Note that hardware replicated wheel nodes may not be required, since it is possible to brake a car using only three wheel nodes. On the other hand, if the wheel nodes are integrated in an ESP system, each wheel node is critical.

A MATLAB/Simulink brake-by-wire model provided by Volvo Technological Development is used as the target application. We used the Real-Time Workshop Ada Coder, which is an extension of Simulink to generate the Ada software code used in the experiments. Figure 3 shows the top-level view of the model.

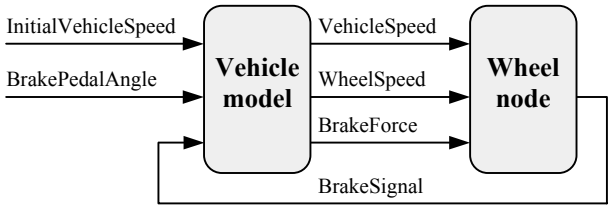

Figure 3. Brake-by-wire model

The model consists of two parts, one part modeling the vehicle and the other part modeling a wheel node. The input to the vehicle model is an initial speed value and the brake pedal angle. In this study, the vehicle model is initiated with a speed of 15 km/h and the brake pedal is activated after 15 ms. The vehicle model uses the brake pedal angle to calculate a brake force, which is delivered to the wheel node. The wheel node calculates the force to be applied on the brake discs. Here, the calculated force is returned to the vehicle model (*BrakeSignal*). The vehicle model calculates the speed reduction caused by the friction force obtained when the brake pad is pressed against the brake disc and then sends new information about the vehicle speed and the speed of the wheel to the wheel node (*VehicleSpeed, WheelSpeed*). The wheel node uses the speed of the vehicle and the speed of the wheel to calculate the wheel slip, i.e. the speed difference between the vehicle and the wheel, reducing the brake force if a specified slip level is exceeded. The brake force is otherwise increased. This allows the brake force to be adjusted for optimized braking performance.

5. Experimental set-up

Figure 4 shows the experimental set-up used to evaluate the time redundant execution of the brake-by-wire application.

A Unix workstation is hosting a Thor microprocessor board [17] used as the target system for our experiments as well as the GOOFI fault injection tool [8]. Our real-

time kernel is running on the microprocessor board, handling the execution of a task containing the code generated for the wheel node. The code generated for the vehicle model is executed on the Unix workstation and is not a target for fault injection. The GOOFI tool performs the fault injection experiments and forward the communication data between the vehicle model and the wheel node. 125 loop iterations (corresponding to 125 ms) of the brake-by-wire model are executed and data is exchanged between the vehicle model and the wheel node every fifth loop.

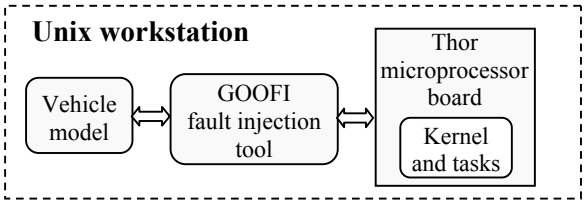

Figure 4. Experimental set-up

5.1. The Thor microprocessor board

The target system for our experiments is a microprocessor board featuring a 32-bit Thor RISC microprocessor, developed by SAAB Ericsson Space AB, and 512 KB RAM. The CPU includes a direct mapped write-back data cache of 128 bytes as well as several internal EDMs. The EDMs can be divided into run-time checks, control flow checking and main memory error checking. Only the run-time checks were activated in this study. The run-time checks include mechanisms commonly found in other microprocessors as well as other checks such as constraint checks of array indices or loop variables.

5.2. The real-time kernel

A small real-time kernel supporting time-redundant execution of tasks was developed to perform the experimental evaluation of the time redundancy technique. Tasks are executed in a periodic receive-compute-send loop. The input data are received in the beginning of the loop from input devices or other tasks. The input data are then processed and the results are sent to actuators or to other tasks in the system in the end of the loop.

Figure 5 shows how the execution of tasks is handled by the kernel. Task A in Figure 5 uses code generated for the wheel node (see Section 4) to calculate the brake force and is considered critical. The brake force calculation returns the output of the computation and a checksum calculated on all output values and all state variables. The checksum produced by each copy is compared to detect errors, see Section 3. Task B (not included in the brake-by-wire model) is non-critical and calculates a slip value to be used, e.g. for icy road warning systems. The two tasks are not connected and they do not use any common variables.

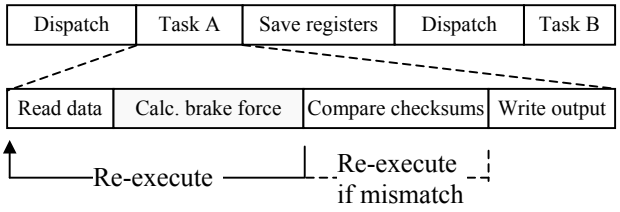

Figure 5. Execution of a critical task

5.3. Fault injection environment

The GOOFI fault injection tool was configured to use Scan-Chain Implemented Fault Injection (SCIFI) to inject faults in the Thor microprocessor. Faults were injected via the internal scan chains into the internal state elements of Thor. The scan chains were also used for observing the internal state of the microprocessor before and after a fault was injected.

Fault model: Transients are modeled by single bit-flips. The single-bit-flip model has become a de-facto standard for modeling the effects of transient faults in fault injection experiments, although it is not a perfect representation of all transient faults.

Fault injection locations: Faults were injected into the registers and data cache of the Thor microprocessor via scan-chains. The scan-chains cover 2250 fault locations of a total of 4400 state elements in Thor. The fault injection locations were selected randomly using uniform sampling among the 2250 state elements.

Points in time for fault injection: The time redundancy technique is evaluated with respect to errors affecting the application tasks. Thus, faults were only injected during the execution of the brake force calculation, see Figure 5. The points in time at which faults are injected were selected randomly in this interval using uniform sampling.

5.4. Definitions

The results of each fault is classified according to the consequences with respect to the required failure semantics, see Figure 6. The requirement is fulfilled if the computer in spite of the fault delivers a correct result, or if a fail-stop or omission failure occurs. The requirement is violated if a value failure or a timing failure occurs.

A value failure occurs when the computer produces an erroneous output value, i.e. an error is not detected or an error is detected but the recovery action fails.

A timing failure occurs when the output from the computer arrives after the deadline. An omission failure occurs when an error is detected but there was not enough time to produce two identical results before the deadline.

Timing failures and omission failures are not considered in this evaluation of the kernel (no deadline is defined and tasks are not pre-empted). Instead, the fault tolerance latency [18], i.e. the total time for error detection and error recovery, is measured.

A correct result is produced if the fault leads to a latent or overwritten error. A correct result is also produced when an error is detected and recovery is successful. Fail-stop failures occur when an error is detected but no recovery could be made, e.g. for CPU exceptions triggered during execution of the kernel code.

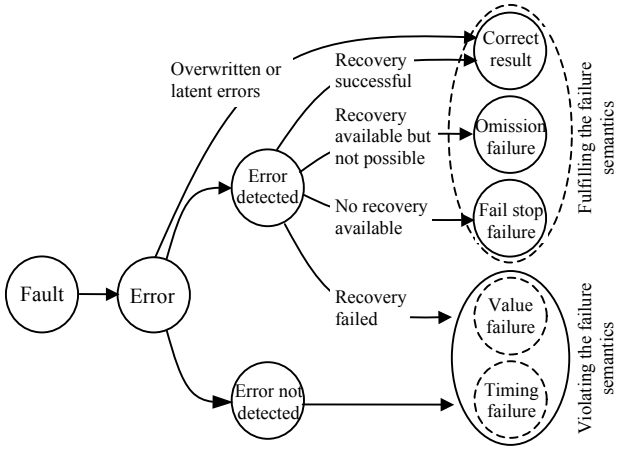

Figure 6. Error propagation and effects

6. Results

Table 1 shows a summary of the results of the fault injection experiments. Two versions of the kernel were evaluated. One without support for time redundancy, called Kernel, and one that implements time redundancy, called FT-Kernel.

Table 1. Result of the fault injection

	Kernel		FT-kernel	
	% (95% conf)	#	% (95 % conf)	#
Correct result	80.9% (± 0.98%)	4962	88.8% (± 0.75%)	6017
Fail-stop failure	17.4% (± 0.95%)	1068	10.1% (± 0.72%)	681
Value failures	1.73% (± 0.33%)	106	1.15% (± 0.25%)	78
Injected faults	6136		6776	

The percentage of correct results produced increases from 81% using Kernel to 89% using FT-kernel, while the fail-stop failures decrease from 17% to 10%.

The 10% fail-stop failures observed for the FT-kernel are caused by faults injected in the cache. The corresponding errors remained latent until activated by kernel code, for which no recovery mechanisms are available.

The value failures decrease from 1.7% to 1.1%. The 1.1% value failures stem either from the case when errors are not detected (0.7%) or when errors are detected but the recovery fails (0.4%).

We have made a preliminary investigation of the causes for the 0.7% non-detected errors. We conjecture that most of these are control flow errors (0.6%), i.e. there is an unexpected jump to an incorrect location in the program code, although further investigation of the error propagation is required to verify this. Another reason for the non-detected errors is errors in the output variable from the second task copy (which is used as the final output) after the checksum has been calculated (0.07%). The reason for the remaining non-detected errors (0.03%) has not yet been identified.

The 0.4% recovery failures are due to errors affecting the input variables in the second and third task copy, causing the majority voter to deliver a faulty output due to two equal but faulty results. The cache is flushed, i.e. all updated rows in the cache are written to memory, between each task copy to avoid faults injected in the cache-lines causing two copies to use the same faulty values. This works for constants since they are not updated. However, input variables are updated in the first task copy (all copies use the same input variables) causing potentially faulty values to be written to main memory and be used by two subsequent task copies.

Table 2 presents the percentage of errors detected by the Thor run-time checks and the double execution mechanisms.

Table 2. Percentage of errors detected by the error detection mechanisms for the FT-kernel.

	%	(95 % conf)	#
All errors detected by Thor run-time checks	14.74 %	(± 0.84 %)	999
Errors detected by double execution	3.31 %	(± 0.43 %)	224

The percentage of errors detected by all Thor run-time checks decreases from 17.4% for the Kernel, to 14.7% in the case of the FT-kernel since more errors were overwritten during execution of the FT-kernel. Time redundant execution of critical tasks detects 3.3% of the errors. Figure 7 shows a histogram of the fault tolerance latency for faults detected by the run-time checks. The fault tolerance latency for faults detected by the double execution is the time for executing the task again and a small amount of time for performing the majority vote (aprox. 105 us).

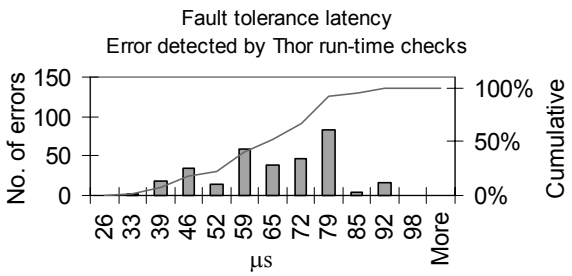

Figure 7. Fault tolerance latency

7. Conclusion

The experimental evaluation of the FT-kernel clearly demonstrates the effectiveness of our temporal error masking technique. The percentage of correct results increased from 81% to 89% using temporal error masking, while fail-stop failures decreased from 17% to 10% and value failures decreased from 1.7% to 1.1%. Nevertheless, the error handling mechanisms in the FT-kernel need to be improved as the percentage of value failures is unacceptably high.

By analyzing the fault injection data, we investigated the causes for the value failures. Most value failures were assumed to be caused by control-flow errors (0.6% of all errors). It is likely that these errors would have been detected by the control flow checking mechanism available in the Thor processor. (The current implementation of the FT-kernel does not support the use of control flow checking.) The other main reason for value failures (0.4% of the errors) was errors that affected the common input to the second and third execution causing those two executions to produce identical but incorrect checksums. These value failures can be avoided by protecting the input data with error detecting and correcting codes. They can also be avoided by using triplication and majority voting.

Future work will focus on extending the FT-kernel with additional error detection and recovery mechanisms to further reduce the probability of value failures. We will investigate the techniques discussed above to improve the handling of errors affecting the application tasks. We will also consider mechanisms to handle faults that affect the execution of the kernel code. These mechanisms will be validated by injecting faults during the execution of the kernel code. We will also validate the FT-kernel with respect to transient faults in the main memory.

8. Acknowledgements

This work was supported by ARTES and the Swedish Foundation for Strategic Research (SSF). We would like to thank Stefan Asserhäll and Torbjörn Hult at Saab Ericsson Space for their technical assistance with the Thor processor. We thank Jerker Lennevi and Henrik Lönn at Volvo Technological Development for providing and supporting the brake-by-wire model. Special thanks go to Philip Koopman for many valuable suggestions and for proposing the term temporal error masking. We also thank the anonymous reviewers for their constructive criticism.

9. References

[1] R.K. Iyer, D.J. Rossetti, and M.C. Hsueh, “Measurement and Modeling of Computer Reliability as Affected by System Activity”, *ACM Trans. on Computer Systems*, 4(3), 1986, pp. 214-37.

[2] E. Normand, “Single Event Upset at Ground Level”, *IEEE Trans. on Nuclear Science*, 43(6, pt.1), 1996, pp. 2742-50.

[3] Johnson B.W., *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wessley, 1989.

[4] N. Oh, P.P. Shirvani and E.J. McCluskey, "Error Detection by Duplicated Instructions In Super-scalar Processors," *IEEE Trans. on Reliability*, Sep. 2001.

[5] Oh, N., and E.J. McCluskey, “Procedure Call Duplication: Minimization of Energy Consumption with Constrained Error Detection Latency” *in Proc. IEEE Int’l Symp. on Defect and Fault Tolerance in VLSI Systems*, 2001, pp. 182 –187.

[6] A. Damm, “The Effectiveness of Software Error-Detection Mechanisms in Real-Time Operating Systems”, *in FTCS Digest of Papers. 16th Annual Int’l Symp. on Fault-Tolerant Computing Systems*, Washington, DC, USA, 1986, pp. 171-176.

[7] F. Cristian, “Understanding Fault-Tolerant Distributed Systems”, *Comm. of the ACM*, 34(2), 1991, pp. 56-78.

[8] J. Vinter, J. Aidemark, P. Folkesson, and J. Karlsson, “Reducing Critical Failures for Control Algorithms Using Executable Assertions and Best Effort Recovery”, in *Proc. Int’l. Conf. on Dependable Systems and Networks*. Göteborg, Sweden, 2001, pp 347 -356.

[9] T. Lovric, "Dynamic Double Virtual Duplex System: A Cost-Efficient Approach to Fault-Tolerance", *in Dependable Computing for Critical Applications 5*, IEEE Computer Society, 1998, pp 57-74.

[10] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors", *in Proc Int’l Conf. on Dependable Systems and Networks,* Madison, WI, USA, 1999, pp 84-91.

[11] Schuette, M.A., Shen J.P., Siewiorek D.P., and Zhu Y.X., “Experimental Evaluation of Two Concurrent Error Detection Schemes”, in *FTCS Digest of Papers. 16th Annual Int’l Symp. on Fault-Tolerant Computing Systems,* Washington, DC, USA, 1986, pp. 138-143.

[12] Pradhan D.K., *Fault-Tolerant Computer System Design*, Upper Saddle River, New Jersey, Prentice Hall PTR, 1996.

[13] A. Burns, S. Punnekkat, L. Strigini, and D.R. Wright, “Probabilistic Scheduling Guarantees for Fault-Tolerant Real-Time Systems”, *in Dependable Computing for Critical Applications 7*, Piscataway, NJ, USA, 1999, pp. 361-378.

[14] G. Fohler, “Adaptive Fault-Tolerance with Statically Scheduled Real-Time Systems”, *in Proc. Ninth Euromicro Workshop on Real Time Systems*, Los Alamitos, CA, USA, 1997, pp 161-167.

[15] I. Bate, and A. Burns, “Schedulability Analysis of Fixed Priority Real-Time Systems with Offsets”, *in Proc. Ninth Euromicro Workshop on Real Time Systems*, Toledo, Spain, 1997, pp 153-160.

[16] K.G. Shin, and H. Kim, “Derivation and Application of Hard Deadlines for Real-Time Control Systems”, *IEEE Trans. on Systems, Man and Cybernetics*, 22(6), 1992 pp. 1403-13.

[17] Saab Ericsson Space AB, Microprocessor Thor, Product Information, 1993.

[18] H. Kim, and K.G. Shin, “Evaluation of Fault Tolerance Latency from Real-Time Application's Perspectives”, *IEEE Trans. on Computers*, 49(1), 2000, pp. 55-64.