Using Massive Time Redundancy to Achieve Node-level Transient Fault Tolerance

J. Aidemark, J. Karlsson
Laboratory for Dependable Computing
Chalmers University of Technology
S-412 96 Göteborg, Sweden
{aidemark, johan}@ce.chalmers.se

1. Introduction

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

For a cost-effective implementation of a fault-tolerant distributed real-time system, it is necessary to design the computer nodes in the system in such way that they exhibit well-behaved failure semantics. This is achieved by including error detection and fault-tolerance mechanisms at the node-level. Nodes that exhibit only[1] fail-stop/(crash) failures or omission failures [1] can be considered to have well-behaved failure semantics.

Fail-stop semantics implies that the node produces either correct results (at the right time) or produces no results at all, i.e. it stops producing results when affected by a fault. An omission failure occurs if the node fails to produce a particular result, but continues to produce correct (and timely) results after the omission failure occurred. These failures can be handled effectively by fairly simple distributed redundancy management protocols, see e.g. the MARS system [2].

Previous research has shown that transient faults are common in digital systems [3]. Transient faults can be caused by particle radiation, e.g. in aircraft at high altitude by high-energy neutrons, or in spacecraft by heavy-ions. Power fluctuations and electromagnetic interference are other causes of transient faults in computer systems.

In this paper we describe an implementation of a small real-time kernel that achieves transient fault tolerance at the node-level. The objective 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 should fulfil fail-stop or omission failure semantics. These properties are achieved by combining hardware and software error detection mechanisms (EDMs) with massive time redundancy. The kernel uses fixed-priority pre-emptive scheduling [4].

We consider real-time systems with hard deadlines where fault-tolerance is achieved by executing each critical task on two nodes with fail-stop and omission failure semantics. When such a system is used in an environment where transients are common, such as a car or a satellite, the time it takes to recover from a transient fault has a significant impact on the overall system reliability. The longer it takes to recover from a transient fault, the higher the probability that the system crashes because another transient fault affects the remaining non-faulty node.

Many existing systems, e.g. MARS [2], do not provide node-level transient fault tolerance, which means that the recovery of all transient faults are handled via the distributed redundancy management protocol. In this case, the recovery time is in the order of seconds or minutes. The advantage of handling transient faults at the node-level is that the average recovery time for transients can be reduced by several orders of magnitude, which improves the overall reliability.

Node level transient fault tolerance also improves the system resiliency to correlated node failures, which occur when a single disturbance causes transient faults in several nodes simultaneously.

[1] That is, the probability for other types of failures is assumed to be negligible.

2. Related work

Error detection by comparing the result from two replicated executions of critical tasks was used in the MARS system [2].

Recovery from transients faults can be achieved by using a checkpointing scheme [5]. Checkpointing is done by saving the state of the processor to stable storage in regular intervals or when certain data is updated. When a fault occur, the system is restored to the last checkpoint and the execution can proceed.

Fault-tolerant scheduling of real-time tasks in a uniprocessor environment is studied in [6]. This work is based on the assumption that transient faults are always detected, and that recovery can be achieved by a retry/ re-execution of the affected task. The aim is to allocate adequate time for the recovery action (i.e. the retry/ re-execution). In [6] time is reserved prior to execution using a technique called overloading. Overloading reserve less time than needed to re-execute all tasks based on the condition that at most one fault can occur during a specific time interval. An approach for dynamically allocating time for re-execution upon a recovery request is presented in [7].

Techniques such as robust data structures, assertions, plausibility tests and checking execution time of system routines were used in [2] to detect transient faults during execution of operating system code.

3. Basic ideas

In massive time redundancy, a task is executed at least three times to produce three or more copies of a result. A majority vote is performed on the copies to mask any faults (cf. massive redundancy in hardware, e.g. a TMR system).

In our real-time kernel, each critical task is always executed twice during normal operation. We use the term *task replica*, or just *replica*, to denote a particular instance of the task execution. The results of the two replicas are compared to detect errors. If the two results do not match, a third replica of the task is executed. The results of the three replicas are then checked by a majority vote. If none of the results match, no result is delivered, which leads to a omission failure. If two results match, they are accepted as a valid result of the task.

Errors can also be detected by hardware and software EDMs such as illegal op-code detection or variable constraint checks. In this case, the affected replica is immediately terminated, and a new replica is started.

We assume that there is enough slack (unused cpu time) available to allow at least one task to execute three replicas, without causing any other task to miss a deadline. However, if two or more task are affected by near-coincident transient faults, it may not be enough slack available to allow all of them to execute three replicas, without causing other tasks to miss their deadlines.

After an error is detected, the kernel checks the deadline of the task to determine if it is possible to execute an additional task replica before the deadline. If not, no result is delivered and an omission failure occurs. If time is available, a new replica is started. The task result is delivered only when two matching results have been produced before the deadline.

We assume a simple task model where the input is read in the beginning of the task and the result is delivered at the end of the task. We also assume that the input to a task is not updated until the task result is delivered.

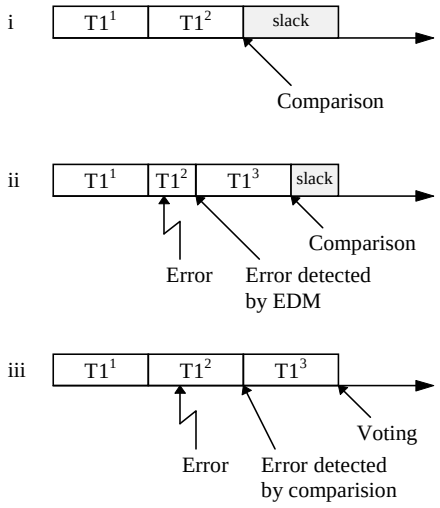

Figure 1: Critical task with replicated executions and fault handling.

Three time diagrams of the execution of task replicas is shown in Figure 1. The time for the execution of the kernel code is not included in the diagrams. The figure shows three different scenarios: (i) Fault free operation, where two replicas $T1^1$ and $T1^2$ of task T1 is executed and their results are compared to detect errors. The

results match and can be delivered directly, not using any of the available slack. (ii) The error is detected by an EDM during execution of the second replica $T1^2$. The affected replica $T1^2$ is terminated and a new replica $T1^3$ is immediately started. $T1^3$ will use time reclaimed from the removed replica $T1^2$ as well as time from the available slack After two replicas have been executed, a comparison is made to confirm that the results match before the task result is delivered. (iii) The error is detected by the comparison. Thus, the results from the two replicas $T1^1$ and $T1^2$ are dissimilar. It is not possible to decide which of the two replicas that produced the correct result so a third replica of the task is started, called $T1^3$, using the available slack. The results of the three executions are check by a majority vote. As previously described, no result is delivered if none of the results match. If two results match, they are accepted as a valid task result.

4. Implementation and Validation

We are implementing a small real-time kernel that achieves transient fault tolerance by using the ideas described above. The kernel will be implemented for the Thor microprocessor developed by Saab Ericsson Space AB.

Fault injection will be performed to estimate the probability of violating the fail-stop or omission failure semantics. We will use two fault injection tools, FIMBUL [8] and MEFISTO-C [9]. FIMBUL inject faults in the physical unit of Thor using the scan-chains. Faults can be injected in the registers and in the cache. MEFISTO-C inject faults in the VHDL model of the Thor processor. Here, faults can be injected in any state element (i.e. a flip-flop, latch or register) of the processor.

5. Summary and Discussion

This paper describes an implementation of a real-time kernel that achieves transient fault tolerance by using massive time redundancy. The real-time kernel is being implemented for the Thor microprocessor, and will be validated by using fault injection to estimate the probability of violating the fail-stop or omission failure semantics. Although the estimated coverage results will be specific for the chosen microprocessor, the experiments will give valuable insight into the usefulness of the proposed technique. In the current version, the kernel only handles transient faults that occur during execution of application tasks.

Future work include the development of techniques that ensure fail-stop semantics when transient faults occur during the execution of the kernel code. We will also investigate different policies for handling near-coincident transient faults that affect more than one task. Another interesting issue is to study error containment between tasks and between tasks and the kernel.

6. References

[1] F. Cristian, "Understanding Fault-Tolerant Distributed Systems", *Communications of ACM*, Vol.34, No.2, 1991, pp. 56-78.

[2] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger, "Tolerating transient faults in MARS" *20th Int. Symp. on Fault-Tolerant Computing* (FTCS-20), IEEE, Newcastle Upon Tyne, U.K., June 1990. pp. 466-473.

[3] 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*, Aug 1986, pp. 214-237.

[4] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings, "Fixed Priority Pre-emptive Scheduling: An Historical Perspective", *Real-Time Systems*, vol. 8, no. 2/3, 1995, pp. 129-154.

[5] D. K. Pradhan, *Fault-Tolerant Computer System Design*, Prentice Hall, USA, 1996, pp. 160-192.

[6] S. Ghosh, "Guaranteeing Fault Tolerance through Scheduling in Real-Time Systems", *Ph.D. Thesis, Dept. Computer Science. University of Pittsburgh*, USA, 1996.

[7] P. Mejia-Alvarez, and D. Mossé, "A responsiveness approach for scheduling fault recovery in real-time systems" *Real-Time Technology and Applications Symposium*, IEEE, Vancouver, Canada, June 1999. pp. 4-13.

[8] P. Folkesson, S. Svensson, and J. Karlsson, "A comparison of simulation based and scan chain implemented fault injection", *28th Int. Symp. on Fault-Tolerant Computing* (FTCS-28), IEEE, Munich, Germany, June, 1998. pp. 284-293.

[9] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models: the MEFISTO tool", *24th Int. Symp. on Fault-Tolerant Computing* (FTCS-24), IEEE, Austin, TX, USA, June 1994. pp. 66-75.