Building Reliable Embedded Systems with Unreliable Components

Invited Paper

Zebo Peng
Embedded Systems Laboratory (ESLAB)
Linköping University
Sweden
e-mail: Zebo.Peng@liu.se

***Abstract*—This paper deals with the design of embedded systems for safety-critical applications, where both fault-tolerance and real-time requirements should be taken into account at the same time. With silicon technology scaling, integrated circuits are implemented with smaller transistors, operate at higher clock frequency, and run at lower voltage levels. As a result, they are subject to more faults, in particular, transient faults. Additionally, in nano-scale technology, physics-based random variations play an important role in many device performance metrics, and have led to many new defects. We are therefore facing the challenge of how to build reliable and predictable embedded systems for safety-critical applications with unreliable components. This paper describes several key challenges and presents several emerging solutions to the design and optimization of such systems. In particular, it discusses the advantages of using time-redundancy based fault-tolerance techniques that are triggered by fault occurrences to handle transient faults and the hardware/software trade-offs related to fault detection and fault tolerance.**

I. INTRODUCTION

We are entering the era of pervasive embedded computing with massive amounts of electronics and software controlling virtually all devices and systems in our everyday life. New functions and features are introduced to embedded computer systems on a daily basis, which has led to the creation of many new gadgets and the addition of powerful functionality to existing systems. For example, modern automobiles already contain a large amount of electronics and software for vehicle control, safety, and driver support, while new features such as automatic intelligent parking assist, blind-spot information system, and navigation computers with real-time traffic updates are being introduced [1]. Many of such embedded systems implement safety-critical applications with high demands on fault tolerance and strict timing constraints (e.g., X-by-wire) or contain functions that require very high computation capacity with reliability and guaranteed QoS (e.g., image and speech recognition, intelligent navigation, and high-end multimedia).

At the same time, with continuous silicon technology scaling, integrated circuits are implemented with smaller transistors, operate at higher clock frequency, and run at lower voltage levels. Therefore, they are subject to more and more faults, especially transient faults, also called soft errors. Transient faults are caused by cosmic radiation, alpha-particles, electromagnetic interference, static electrical discharges, ground loops, power supply fluctuations, humidity, temperature variation, pressure, vibration, loose connections, noise, etc. Recent studies have indicated that the rate of transient faults is increasing rapidly in modern electronic systems [2]. Previously, transient faults had mainly large impacts on the memory and latches of a digital system, while their impacts on the logic parts are becoming also significant with new CMOS technologies.

Although transient faults do not lead to permanent damage of the circuits, they often cause errors that can have catastrophic consequences in many safety-critical applications. Additionally, in deep submicron technology used nowadays to implement advanced embedded systems, physics-based random variations play a more and more important role in many device performance metrics [3]. It has recently been pointed out that this rise in the inherent systematic and random variations will have effects that are far-reaching in every aspect of design, manufacturing, test, and overall reliability [4].

We are therefore facing the challenge of how to build reliable and predictable embedded systems with unreliable components where the reliability problem stems from transient faults as well as their interaction with random process variation [5]. This challenge has to be addressed at multiple levels of abstraction, including circuit, architecture, software, and system levels. Moreover, cross-layer optimization, in particular, hardware/software trade-off, is needed in order to generate high quality solutions.

II. TRANSIENT FAULT TOLERANCE TECHNIQUES

Traditionally, research on fault tolerance has mainly dealt with permanent faults. While several techniques proposed to tolerate permanent faults may be used also for tolerating transient faults, they are only efficient if the number of transient faults is not large.

A common strategy for fault tolerance is to use different forms of *redundancy* with respect to hardware, software, information, or time. A well-known example of *hardware-redundancy* techniques is the triple modular redundancy

Copyright © 2010 by Institute of Electronics, Silesian University of Technology

(TMR), where three copies of hardware are implemented and a voting mechanism is used to produce a single output. There are many variations in implementing the basic TMR concept. For example, in the MARS approach, each fault-tolerant component is composed of three computation units, two main units and one shadow unit [6]. Once a transient fault is detected, the faulty component must restart while the system is operating with the non-faulty component. This architecture can tolerate one permanent fault and one transient fault at a time, or two transient faults.

Another example of hardware redundancy is the XBW architecture [7], where hardware duplication is combined with double process execution. Four process replicas are therefore run in total. Such an architecture can tolerate either two transient faults or one transient fault with one permanent fault. Other hardware replication examples can often be found in avionics. For example, an airborne architecture with seven hardware replicas, which can tolerate up to three transient faults, has been studied in [8], based on the flight control system of the JAS 39 Gripen aircraft. Such solutions, based on a large degree of hardware redundancy, are very costly and can only be used if the amount of resources is virtually unlimited.

An example of *software redundancy* is N-version programming (NVP), also known as multi-version programming, where multiple functionally equivalent programs are independently developed from the same specification [9]. If a specification is implemented with three different software versions, together with a voting mechanism, it will be able to tolerate transient faults occurring in one of the three software processes or one permanent fault if the three software processes run on three different hardware units. An important additional advantage of NVP is that it addresses also the problem of software bug introduced in the software development process. In the case of 3-version programming, the system will produce correct results if only one version of the program has bugs. However, the overheads in terms of both additional development efforts and run-time redundancy are very large for NVP.

In the case of *information redundancy*, error detection and correction codes are used by adding some redundancy (i.e., some extra data) to the functional data. While information redundancy schemes are efficient in detecting and correcting faults that occur when data are transmitted over unreliable communication channels or stored in unreliable memories, they cannot be used to address faults that cause miscalculations in logic. Similar to hardware and software redundancy, information redundancy incurs also a large overhead in terms of extra hardware, which will be present in the system when implemented.

Finally, *time redundancy* (TR) is a technique that intends to reduce the amount of extra hardware at the expense of additional time. When implemented in software, after a transient fault is detected, a fault tolerance mechanism will be invoked to deal with it. The simplest fault tolerance technique to recover from fault occurrences is re-execution, where a process is executed again if affected by faults [10].

An example of software re-execution is illustrated in Fig. 1, where we have process P_1 and $k = 2$ transient faults that may happen. In the worst-case fault scenario depicted in Fig. 1, the first fault happens during P_1's first execution, denoted as $P_{1/1}$, and is detected by an error detection mechanism (Note that we assume here that error detection is carried out during the software execution, and the error detection overhead is therefore considered as part of the process execution time; we will address this issue in Section III). After a recovery overhead (used to restore all initial inputs of the process) of $\mu = 5$ *ms*, depicted with a light gray rectangle, P_1 will be executed again. Its second execution $P_{1/2}$ in the worst-case could also experience a fault. Finally, the third execution $P_{1/3}$ of P_1 will take place without fault, as illustrated in Fig. 1 [11].

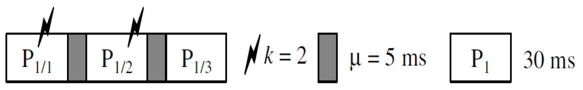


Fig. 1. Time redundancy by software re-execution

If the real-time OS kernel used to activate re-execution is itself fault-tolerant, and the error detection and recovery mechanisms work, we have a reliable TR implementation that handles *k* faults by software re-execution. In the case of a distributed embedded system where several processors are connected via a bus, which is very common in many modern embedded systems, we assume that some fault-tolerance communication protocols are used, such TTP [12], to make sure that the communication is also fault tolerant.

The TR scheme based on software re-execution is a very efficient fault-tolerance technique for handling transient faults, since the re-execution mechanism will only be invoked when a fault actually occurs. When no fault happens, only the original version of the software process will be executed, and there is almost no overhead (except the fault detection one). For example, for process P_1 in Fig. 1, only its first execution $P_{1/1}$ will be performed, as in the case of normal execution. This scheme is therefore called *on-demand redundancy*, in contract to the *always-present redundancy* based of hardware/software/information redundancies. In the case of always-present redundancy, large hardware overhead is always needed. For example, three times of hardware resources plus a voting mechanism are physically implemented with the TMR scheme.

While it has been noted that the rate of transient faults is increasing rapidly and will become large in the future, still, transient faults occur with a relatively low frequency with the current technologies. Additionally, many transient faults will not be propagated to the software level or will not have an impact on the execution of a given software process. For example, if a transient fault hits the multiplier of the CPU, but the current software process doesn't involve any multiplication instruction, this fault will not have any impact on the computation results of the process, and doesn't need to be tolerated. This means that software re-execution will only be needed rarely, and the average time penalty we have to pay for using a TR strategy at the software level is very low, while it reduces very much the amount of extra hardware.

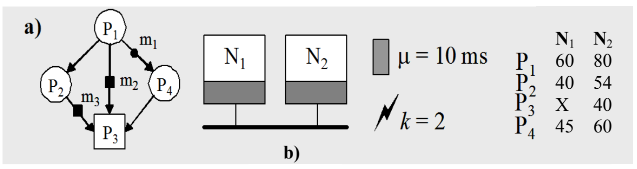
However, even if the transient fault rate is currently low on average, many transient faults can happen in a burst during a short period of time. This will not be a problem for general purpose computing, since only certain processes will run a bit slower due to many re-executions, when a burst of transient faults hits the system. The *average* performance reduction is still very small. However, for embedded systems that have strict timing constraints, software re-execution may lead to solutions where the timing constraints are not satisfied. Therefore, we need to design embedded systems for safety-critical applications by considering both fault-tolerance and real-time requirements at the same time.

III. Design optimization with fault-tolerance and real-time requirements

When a fault-tolerance mechanism, such as software re-execution, is deployed in an embedded system, it must be taken into account in the global design-space exploration process in order to generate solutions that fulfill all the real-time constraints and meet the reliability objective at the same time. Optimization should also be performed to minimize the implementation cost since most of these embedded systems are utilized in products in very competitive market sectors such as the automotive industry. This global optimization problem consists of several components, and we will discuss a few important ones as follows.

A. Design Optimization Problems

Typically, the functionality of an embedded system is given in term of a set of directed and acyclic graphs, where a node denotes a software process, and an edge the data-dependency relation between two processes. Each process can be associated with a deadline, and we assume, in this paper, that the deadlines are hard, i.e., they must always be met even in the presence of multiple transient faults. Fig. 2(a) illustrates a simple example of such functionality (application) with only one directed graph consisting of four processes (P_1, P_2, P_3, and P_4) and five edges.



	N_1	N_2
P_1	60	80
P_2	40	54
P_3	X	40
P_4	45	60

Fig. 2. Application and architecture example

The given functionality is to be executed on a hardware architecture in a periodic manner with a given period. And we assume that a distributed hardware architecture consists of a set of nodes, which share a broadcast communication channel (e.g., a bus), is used. The communication channel is statically scheduled such that one node at a time has access to it, according to a given schedule determined off-line. The architecture can be either given or to be designed in the design-space exploration process. In the latter case, we have the architecture selection task, which determines the number of nodes, selects the nodes (e.g., from a set of available processors with different performances and costs), and decides on the parameters and protocols for the communication channel. Fig. 2(b) illustrates such an architecture with two nodes (N_1 and N_2), connected by a bus.

We should also design a software architecture that runs on the CPU in each node, and has a real-time kernel as its main component. Process activation on a node and message transmission via the communication channel will be done based on the schedule tables that are stored locally in each node and in the communication channel controller.

Given the functionality and hardware/software architecture, a key design problem is to determine the mapping of software processes to hardware nodes such that transient faults are tolerated and all deadlines of the application are met. Such mapping decision can be made to optimize the completion time of the processes (i.e., the performance of the real-time application), which can lead to a cheaper hardware solution, when there are slacks between the completion times and the deadlines of the processes and therefore a slower hardware architecture can be utilized instead [11]. To perform this optimization, the worst-case execution times of the processes when mapped on different hardware nodes should be given. For the above example, the table at the right hand corner of Fig. 2 depicts these process execution times. For example, the worst-case execution time of P_1 is 60 *ms* when mapped on node N_1, and 80 *ms* when mapped on node N_2. The "X" symbol in the table is used to indicate mapping restriction. For example, process P_3 is not allowed to be mapped on node N_1. This restriction can be used by the designer to denote that a decision has already been made, for some reason, that a given process (e.g., P_3) should always be mapped on a given node (e.g., N_2).

The mapping decision has a very large impact on the traffic over the commutation channel. If one process sends a large amount of data to another process, and the two processes are mapped on different nodes, this communication is performed by message passing over the communication channel, and should be considered during the design process. On the other hand, if these two processes are mapped on the same node, the data can be sent via the shared memory, and the communication time can be accounted as part of the process execution time. For the example given in Fig. 2, if P_1 and P_2 are mapped on N_1, while P_3, and P_4 on N_2, there are three messages (denoted as m_1, m_2, and m_3) that need to be explicitly considered and scheduled on the communication channel. Message m_1, for example, is used to send data from process P_1 mapped on node N_1 to process P_4 mapped on N_2. The communication from P_1 to P_2, on the other hand, doesn't need to be explicitly considered, since P_1 and P_2 are mapped on the same node, N_1.

The mapping decision can also take into account other aspects, such as the degree of transparency, which denotes how much impact recovering from a transient fault at one hardware node will have on the schedules of other nodes. A fully transparent system means that the occurrence of faults at any node will not impact the executions on any other nodes. The transparent property has the advantages of fault containment and improved debugability, and it needs less memory to store the fault-tolerant schedules [10]. In general, a designer would like to introduce as much transparency as possible. However, transparency will incur additional delays that can lead to the violation of some real-time deadlines. A fine-grained approach to transparency has been proposed in [13], where the designer specifies a desired degree of transparency by declaring certain processes and messages as frozen. A frozen process or message will have a fixed start time (for a process, more precisely, the *first execution* of it will have a fixed start time), regardless of the occurrence of faults in the rest of the system. In Fig. 2, the frozen processes and messages are depicted with squares, while the regular ones are drawn as circles.

The debugability of the application is improved by transparency because it is easier to observe the behavior of frozen processes and messages in alternative schedules that correspond to different fault scenarios. An optimization algorithm has been developed to map processes to nodes such that not only the application is schedulable (i.e., all processes will meet their deadlines) but also the transparency properties imposed by the designer are satisfied [13].

After the mapping task is performed, processes mapped on the same hardware node should be scheduled. The scheduling decision should be taken to make sure that re-execution slack is introduced in the schedule for the re-executed processes. For this decision, we need to know how many re-executions are needed in different fault contexts. It is usually assumed that at most k transient faults may occur anywhere in the system during one operation period, and the scheduling algorithm should make sure that in the worst-case fault scenario, the execution of all processes and their needed re-executions will be completed before their respective deadlines. The basic idea to reduce the amount of re-execution slack is to share a slack by several processes. Note that for two processes to share the same slack, they need to be mapped on the same node. This means that the mapping decision has a large impact on the scheduling results. Therefore, the mapping task and scheduling task should be performed together to generate a globally optimal solution. Note also that the messages sent over the communication channel should also be scheduled. And again, to achieve global optimization, process scheduling and message scheduling should be considered at the same time.

The maximal number of transient faults to tolerate during an operation period can be either given by the designers based on statistics data (for the example in Fig. 2, k is given as 2) or derived by a system failure probability (SFP) analysis technique. An SFP analysis technique is used to ensure that the final design meets the given reliability requirements, by connecting the levels of fault tolerance in software to the levels of transient-fault rate in hardware [14].

B. *Fault detection optimization*

As stated before, software re-execution is an efficient time-redundancy technique to handle transient faults, since re-execution is only performed if there is actually a fault. However, a fault detection mechanism is needed to decide if there is a fault in the first place. This mechanism will always be present in the implementation, and it is a major source of hardware overhead. Therefore optimization of the fault detection mechanism is very important, and the optimization result has a very large impact on the efficiency of the overall fault tolerance implementation.

An application-aware error detection technique to identify critical variables in a program, which exhibit high sensitivity to random data errors, has been proposed in [15]. The backward program slice for each acyclic control path is extracted for the identified critical variables, and each slice is optimized at compile time, resulting in a series of checking expressions. These will be inserted in the original code, immediately after the computation of a critical variable. Finally, the original program is instrumented with instructions to keep track of the control paths followed at run-time and with checking instructions that would choose the corresponding checking expression, and then compare the results obtained [15]. This technique has two main sources of performance overhead: path tracking and variable checking. In the context of transient faults, both of them can be implemented either in software, potentially incurring high performance overheads, or in hardware, which can lead to costs sometimes exceeding the amount of available resources [16].

An optimization technique to make trade-offs between hardware/software implementations of the above application-aware error detection scheme has been developed and reported in [16]. It minimizes the global worst-case execution length of the software processes, while meeting the imposed hardware cost constraints and tolerating multiple transient faults. The reported work demonstrates that it is possible to reduce the worst-case schedule length by more than a half with only as few as 15% hardware fractions available [16].

IV. HARDWARE/SOFTWARE TRADE-OFFS

A different approach to address the problem of transient faults is to improve the hardware technology and/or architecture to reduce the fault rate, and, hence, the number of faults propagated to the software level [17]. Researchers have proposed a variety of hardware hardening techniques. For example, Zhang et al. have proposed an approach to harden flip-flops, resulting in a small area overhead and significant reduction in the transient fault rate [18]. Mohanram and Touba have studied hardening of combinatorial circuits [19]. Finally, a hardening approach to be applied in early design stages has been presented in [17], which is based on transient fault detection probability analysis.

Hardware hardening comes, however, with a significant overhead in terms of cost and speed [20]. The main factors which affect the cost are the increased silicon area, additional design effort, lower production quantities, excessive power consumption, and protection mechanisms against radiation (such as shields). Hardened circuits are also significantly slower than the regular ones. Manufacturers of hardened circuits are often forced to use technologies a few generations back [20]. Hardening also enlarges the critical path of the circuit, because of a voting mechanism [21] and increased silicon area. Therefore careful trade-offs must be made in respect to how much hardware hardening should be done vs. how many software re-execution should be implemented, in order to meet time and cost constraints within the given resources, and at the same time, deliver the reliability goals.

The input to the design-space exploration process with such trade-offs consists of the application (captured as a set of acyclic directed graph) mapped on a bus-based architecture (as discussed in Section III.A), a given reliability goal, and the recovery overhead μ. For each node in the hardware architecture, a set of available hardware implementations with different hardening level and their corresponding costs is also given. It is assumed that the worst-case execution times and the failure probabilities for each process on each hardening version of the nodes are known. And the maximum transmission time of messages if sent over the bus is given. A design-space exploration algorithm should select an implementation from the available hardening alternatives for each node, determine the mapping of the processes on the nodes with the selected implementation, and schedule the processes together with their required re-executions and the needed communications (it is assumed that communication are fault-free due to the use of fault-tolerance protocol).

An algorithm to perform this design-space exploration so as to minimize the total cost of the hardware nodes, while achieving the reliability goal and meeting all timing constraints given with the application, is presented in [14]. The algorithm is based on a sequence of iterative design optimization heuristics, and utilizes a system failure probability analysis approach, which connects the level of hardening in hardware with the number of re-executions in software [14].

V. Conclusions

Transient-fault or soft-error rates have been increasing rapidly due to the continuous device scaling, increased clock frequency, high temperature, voltage scaling, and process variation. How to handle transient faults in the context when an embedded system is used for safety-critical applications, which have a high reliability requirement, is therefore becoming a critical issue. This paper discusses the advantages of using the *time-redundancy* strategy at the software level to handle these transient faults. It presents also several design optimization problems with respect to time-redundancy by software re-execution and several emerging solutions to solve them.

Many issues discussed in this paper, including transient faults and their tolerance, error detection, software re-execution, and hardware hardening, are not new, taken individually. However, the interplay of these issues and their increased impacts have led to great challenges to the research community. In particular, there are still many open problems in how to develop efficient global optimization techniques to consider both fault-tolerance and real-time requirements at the same time, to make hardware/software trade-offs for systems affected by transient faults and large process variation, and to build reliable and predictable embedded systems with unreliable components.

Acknowledgments

I thank Petru Eles, Viacheslav Izosimov, Adrian Lifa, and Paul Pop for the cooperation and help they have provided. Many ideas presented in this paper are inspired by the results of several joint research projects with their participations.

References

[1] B. Bouyssounouse and J. Sifakis (Eds.), *Embedded Systems Design: The ARTIST Roadmap for Research and Development*, Springer, 2005.

[2] A. Maheshwari, W. Burleson, and R. Tessier, "Trading Off Transient Fault Tolerance and Power Consumption in Deep Submicron (DSM) VLSI Circuits," *IEEE Trans. on VLSI Systems*, 12(3), 2004, pp. 299-311.

[3] S. Borkar, "Designing Reliable Systems from Unreliable Components: the Challenges of Transistor Variability and Degradation," *IEEE Micro*, Nov.- Dec. 2005, pp. 10-16.

[4] T. M. Mak and S. Nassif, "Guest Editors' Introduction: Process Variation and Stochastic Design and Test," *IEEE Design & Test of Computers*, Vol. 23, No. 6, 2006.

[5] Y. Cao, P. Bose, and J. Tschanz, "Guest Editors' Introduction: Reliability Challenges in Nano-CMOS Design," *IEEE Design & Test of Computers*, Vol. 26, No. 6, 2009, pp. 6-7.

[6] H. Kopetz, et al., "Distributed Fault-Tolerant Real-Time Systems: The MARS Approach", *IEEE Micro*, 9(1), 1989, pp.25-40.

[7] V. Claesson, S. Poledna, and J. Soderberg, "The XBW Model for Dependable Real-Time Systems", *Proc. Intl. Conf. on Parallel and Distributed Systems*, 1998, pp. 130-138.

[8] K. Alstrom and J. Torin, "Future Architecture for Flight Control Systems", *Proc. 20th Conf. on Digital Avionics Systems*, 1B5/1-1B5/10, 2001.

[9] L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation, Fault-Tolerant Computing," *Proc. International Symposium on Fault-Tolerant Computing*, 1995.

[10] N. Kandasamy, J. P. Hayes, and B. T. Murray, "Transparent Recovery from Intermittent Faults in Time-Triggered Distributed Systems," *IEEE Trans. on Computers*, 52(2), 2003, pp.113-125.

[11] V. Izosimov, P. Pop, P. Eles and Z. Peng, "Design Optimization of Time- and Cost-Constrained Fault-Tolerant Distributed Embedded Systems," *Proc. Design Automation and Test in Europe Conf. (DATE'05),* 2005, pp. 864-869.

[12] H. Kopetz and G. Bauer, "The Time-Triggered Architecture," *Proc. of IEEE*, 91(1), 2003, pp. 112–126.

[13] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Synthesis of Fault-Tolerant Schedules with Transparency/Performance Trade-offs for Distributed Embedded Systems," *Proc. Design, Automation and Test in Europe Conf. (DATE'06)*, 2006, 706-711.

[14] V. Izosimov, I. Polian, P. Pop, P. Eles, and Z. Peng, "Analysis and Optimization of Fault-Tolerant Embedded Systems with Hardened Processors," *Proc. Design, Automation and Test in Europe Conf (DATE'09)*, 2009, pp. 682-687.

[15] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "Automated Derivation of Application-Aware Error Detectors using Static Analysis: The Trusted Illiac approach," *IEEE Trans. Dependable and Secure Computing*, 2009.

[16] A. Lifa, P. Eles, Z. Peng, and V. Izosimov, "Hardware/Software Optimization of Error Detection Implementation for Real-Time Embedded Systems," *Proc. Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2010),* 2010.

[17] J. P. Hayes, I. Polian, and B. Becker, "An Analysis Framework for Transient-Error Tolerance," *Proc. IEEE VLSI Test Symp.,* 2007, pp. 249-255.

[18] M. Zhang, et al., "Sequential Element Design With Built-In Soft Error Resilience", *IEEE Trans. on VLSI Systems*, 14(12), 2006, pp. 1368-1378.

[19] K. Mohanram and N. A. Touba, "Cost-Effective Approach for Reducing Soft Error Failure Rate in Logic Circuits," *Proc. Intl. Test Conf. (ITC)*, 2003, pp. 893-901.

[20] P. Patel-Predd, "Update: Transistors in Space", *IEEE Spectrum*, 45(8), 2008, pp. 17.

[21] R. Garg, N. Jayakumar, S. P. Khatri, and G. Choi, "A Design Approach for Radiation-Hard Digital Electronics," *Proc. Design Automation Conf. (DAC)*, 2006, pp.773-778.