Analysis and optimization of fault-tolerant embedded systems with hardened processors

Izosimov, Viacheslav; Polian, Ilia; Pop, Paul

Published in:
Analysis and optimization of fault-tolerant embedded systems with hardened processors

Publication date:
2009

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Analysis and Optimization of Fault-Tolerant Embedded Systems with Hardened Processors
Viacheslav Izosimov 1, Iilia Polian 2, Paul Pop 3, Petru Eles 1, Zebo Peng 1

1 {viaiz | petel | zebpe}@ida.liu.se Dept. of Computer and Inform. Science Linköping University SE-581 83 Linköping, Sweden
2 polian@informatik.uni-freiburg.de Institute for Computer Science Albert-Ludwigs-University of Freiburg D-79110 Freiburg im Breisgau, Germany
3 Paul.Pop@imm.dtu.dk Dept. of Informatics and Math. Modelling Technical University of Denmark DK-2800 Kongens Lyngby, Denmark

Abstract

In this paper we propose an approach to the design optimization of fault-tolerant hard real-time embedded systems, which combines hardware and software fault tolerance techniques. We trade-off between selective hardening in hardware and process re-execution in software to provide the required levels of fault tolerance against transient faults with the lowest-possible system costs. We propose a system failure probability (SFP) analysis that connects the hardening level with the maximum number of re-executions in software. We present design optimization heuristics, to select the fault-tolerant architecture and decide process mapping such that the system cost is minimized, deadlines are satisfied, and the reliability requirements are fulfilled.

1. Introduction

Safety-critical embedded systems have to satisfy cost and performance constraints even in the presence of faults. In this paper we deal with transient and intermittent faults (also known as “soft errors”), which are very common in modern electronic systems. Their number is increasing with smaller transistor sizes and higher frequencies. Transient faults appear for a short time, cause miscalculation in logic, corruption of data, and then disappear without physical damage to the circuit. Causes of transient faults can be electromagnetic interference, radiation, temperature variations, software “bugs”, etc. [8]. Transient faults can be addressed in hardware with hardening techniques, i.e., improving the hardware architecture to reduce the soft error rate, or in software with techniques such as re-execution, replication, or checkpointing.

In the context of fault-tolerant real-time systems, researchers have tried to integrate fault tolerance techniques and task scheduling [3, 11, 24]. A static cyclic scheduling framework for design of fault-tolerant embedded control systems with masking of fault patterns through active replication is proposed in [14]. Girault et al. [5] propose a generic approach to address multiple failures with active replication. Process criticality is used as a metric for selective replication in [20]. Transparent re-execution and constructive mapping and scheduling for fault tolerance have been proposed in [9]. In [8, 7, 15] we have proposed scheduling and fault tolerance policy assignment techniques for distributed real-time systems, such that the required level of fault tolerance is achieved and real-time constraints are satisfied with a limited amount of resources.

The research mentioned above is focused on software fault tolerance techniques. However, with increased error rate due to new technologies and/or in the case of particular harsh conditions (e.g. high radiation), pure software techniques are not sufficient in order to achieve the required level of fault tolerance [13, 16].

Researchers have recently proposed a variety of hardware hardening techniques. Zhang et al. [21] propose an approach to selective hardening of flip-flops, resulting in a small area overhead and significant reduction in the error rate. Mohanram and Touba [12] have studied selective hardening of combinatorial circuits. Zhou et al. [23] have later proposed a “filtering technique” for hardening of combinatorial circuits. Zhou and Mohanram [23] have studied the problem of gate resizing as a technique to reduce the error rate. Garg et al. [4] have connected diodes to the duplicated gates to implement an efficient and fast voting mechanism. Finally, a selective hardening approach to be applied in early design stages has been presented in [6], which is based on the transient fault detection probability analysis.

However, hardening comes with a significant overhead in terms of cost and speed [13, 19]. The factors which affect the cost are the increased silicon area for fault tolerance, additional design effort, lower production quantities, excessive power consumption, and protection mechanisms against radiation such as shields. Hardened processors are also significantly slower than the regular ones. The manufacturers of hardened processors are using technologies few generations back [13, 19], and hardening enlarges the critical path on the circuit e.g. because of voting mechanism [4] and increased silicon area.

In this work, we combine selective hardening with software fault tolerance in order to achieve the lowest-possible system costs while satisfying hard deadlines and fulfilling the reliability requirements. We use process re-execution to tolerate transient faults in software. To ensure that the system architecture meets the reliability requirements, we propose a system failure probability (SFP) analysis. This analysis connects the levels of redundancy (maximum number of re-executions) in software to the levels of redundancy in hardware (hardening levels). We also propose a set of design optimization heuristics in order to decide the hardening levels of computation nodes, the mapping of processes on computation nodes, and the number of re-executions on each computation node. Processes and messages are scheduled using an approach we have presented in [7, 15]. Experimental results show an improvement of up 55% on synthetic applications in terms of the number of schedulable and reliable fault-tolerant solutions with the acceptable cost; and an improvement of 66% for a realistic application in terms of cost.

The next two sections present our application model and fault tolerance techniques, respectively. In Section 4, we outline our problem formulation. Section 5 illustrates hardening/re-execution trade-offs. Our heuristics are discussed in Section 6 and experimental results are presented in Section 7. In Appendix A we present our SFP analysis.

2. Application and System Model

We model an application as a set of directed, acyclic graphs \( G_k(V_k, E_k) \) \( \in \mathcal{A} \). Each node \( P_i \in V_k \) represents one process. An edge \( e_{ij} \in E_k \) from \( P_i \) to \( P_j \) indicates that the output of \( P_i \) is the input of \( P_j \). A process can be activated after all its inputs, required for the execution, have arrived. The process issues its outputs when it terminates. Processes cannot be preempted during their execution.

We consider that the application is running on a set of computation nodes \( \mathcal{N} \) connected to a bus. Processes mapped on different computation nodes communicate with messages sent over the bus. We consider that the worst-case size of messages is given, which implicitly can be translated into the worst-case transmission time on the bus. In this paper we assume that communications are fault tolerant (i.e., we use a communication protocol such as TTP [10]).

1. This work was partially supported by the Swedish Graduate School in Computer Science (CUGS), the ARTESS++ Swedish Graduate School in Real-Time Systems, and by the DFG project RealTest (BE 1176/15-1).
2. We will refer to both transient and intermittent faults as “transient” faults.
Transient faults can affect processes executed on a computation node, which would lead to a process failure. To reduce the probability of process failure, the designer can choose to use a hardened, i.e., a more reliable, version (h-version) of the computation node. Thus, each node $N_j$ is available in several versions, with different hardening levels, denoted with $h$. We denote $N_j^h$ the h-version of node $N_j$, and with $C_j^h$ the cost associated with $N_j^h$. A pair $(P_i, N_j^h)$ indicates that process $P_i$ is mapped to the $h$-version of node $N_j$. The worst-case execution time (WCET) of $P_i$ executed on $N_j^h$ is denoted $t_{i|h}$. The probability of failure of a single execution of process $P_i$ on $N_j^h$ is denoted $P_{i|h}$. WCETs $(t)$ are determined with worst-case analysis tools [2], while process failure probabilities $(p)$ are determined using fault injection tools [1, 18].

In Fig. 1 we have an application $A$ consisting of the process graph $G_1$ with four processes, $P_1, P_2, P_3,$ and $P_4$. The deadline of the application graph $D = 360$ ms. The execution times ($t$) and failure probabilities ($p$) for the processes on different $h$-versions of computation nodes $N_1$ and $N_2$ are shown in the tables. The corresponding costs are associated with these versions (given at the bottom of the tables).

### 3. Fault Tolerance Techniques

As a software fault tolerance mechanism we use process re-execution. We assume that the error detection and fault tolerance mechanisms are themselves fault tolerant. The time needed for detection of faults is accounted for as part of the WCET of the processes. The process re-execution operation requires an additional overhead captured as $\mu$. For example, $\mu = 15$ ms for the application $A$.

Safety-critical embedded systems have to be designed such that they meet a certain reliability goal $\rho = 1 - \gamma$. In this paper we consider that $\gamma$ is the maximum probability of a system failure due to transient faults on any computation node within a time unit, e.g. one hour of functionality. For example, the reliability goal for the application $A$ in Fig. 1 is $1 - 10^{-5}$ within one hour.

With sufficiently hardened nodes, the reliability goal can be achieved without any re-execution at software level, since the probability of the hardware failing is acceptably small. As the level of hardening decreases, the probability of faults being propagated to the software level is increasing. Thus, in order to achieve the reliability goal, a certain number of re-executions have to be introduced at software level.

In Fig. 2, we consider a process $P_1$ executed on a node $N_1$ with three $h$-versions. The worst-case execution scenario is different for the different $h$-versions. In the first version, two re-executions, $k_1 = 2$, have to be introduced into software in order to meet the reliability goal. The faults will be tolerated with re-execution as presented in Fig. 2a. The first execution of $P_1$, denoted $P_{1|1}$, is affected by a fault and is re-executed as $P_{1|2}$, after a worst-case recovery overhead $\mu = 5$ ms. The second execution $P_{1|2}$, in the worst case, also fails and is re-executed. Finally, the third execution $P_{1|3}$ will complete without faults. In the second version with a higher hardening level, only one re-execution, $k_1 = 1$, has to be added into software, which will correspond to the worst-case scenario with one re-execution in Fig. 2b. In the most hardened version, $k_1 = 0$, and process $P_1$ is executed without re-executions at software level.

Note that the worst-case execution time of process $P_1$ has increased with the hardening. Nevertheless, in the example, an increased level of hardening has resulted in smaller worst-case delays (which is not necessarily the case in general). In Appendix A we show how the maximum number of re-executions $k_1$ which have to be introduced at software level on $N_1$ is connected to the reliability goal and the hardening level of the computation nodes.

### 4. Problem Formulation

As an input we get an application $A$, represented as a set of acyclic directed graphs $G_i \in A$. Application $A$ runs on a bus-based architecture as discussed in Section 2. The reliability goal $\rho$, the deadline, and the recovery overhead $\mu$ are given. Given is also a set of available computation nodes each with its available hardened $h$-versions and the corresponding costs. We know the worst-case execution times, and the failure probabilities are obtained with fault injection experiments [1, 18] for each process on each $h$-version of computation node. The maximum transmission time of all messages, if sent over the bus, is given.

As an output, the following has to be produced: (1) a selection of the computation nodes and their hardening level; (2) a mapping of the processes to the nodes of the selected architecture; (3) the maximum number of re-executions on each computation node; and (4) a schedule of the processes and communications.

The selected architecture, the mapping and the schedule should be such that the total cost of the nodes is minimized, all deadlines are satisfied, and the reliability goal $\rho$ is achieved. Achieving the reliability goal implies that hardening levels are selected and the number of re-executions are chosen on each node $N_j$ such that the elaborated schedule, in the worst case, satisfies the deadlines.

### 5. Motivational Examples

The first example, depicted in Fig. 3, shows how hardening can improve schedulability if the error rate is high. In Fig. 3, we consider one process, $P_1$, and one processor, $N_1$, with three $h$-versions, $N_1^1$ without hardening and $N_1^2$ and $N_1^3$ progressively more hardened. The corresponding failure probabilities, the WCET and costs are depicted in the table. We have to meet a deadline of 360 ms and the reliability goal of $1 - 10^{-5}$ within one hour. As shown in Appendix A, the hardening levels are connected to the number of re-executions in software, to satisfy the reliability goal. Thus, using $N_1^1$, we have to introduce 6 re-executions to reach the reliability goal, as depicted in Fig. 3a, which, in the worst case, will miss the deadline of 360 ms. However, with the $h$-version $N_1^2$, the failure probability is reduced by two orders of magnitude, and only two re-executions are needed for satisfying the reliability goal $\rho$. This solution will already meet the deadline as shown in Fig. 3b. In case of the most hardened architecture depicted in Fig. 3c, only one re-execution is needed. However, using $N_1^3$ will cost twice as much as the previous solution with less hardening. Moreover, due to performance degradation, the solution with the maximal hardening

---

**Figure 1. Application Example**

**Figure 2. Re-execution and Hardening**

<table>
<thead>
<tr>
<th>$N_1$</th>
<th>$h=1$</th>
<th>$h=2$</th>
<th>$h=3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$t$</td>
<td>$p$</td>
<td>$t$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>60</td>
<td>$1.2\times10^{-4}$</td>
<td>75</td>
</tr>
<tr>
<td>$P_2$</td>
<td>75</td>
<td>$1.3\times10^{-4}$</td>
<td>90</td>
</tr>
<tr>
<td>$P_3$</td>
<td>60</td>
<td>$1.4\times10^{-4}$</td>
<td>75</td>
</tr>
<tr>
<td>$P_4$</td>
<td>75</td>
<td>$1.6\times10^{-4}$</td>
<td>90</td>
</tr>
</tbody>
</table>

**Cost**

<table>
<thead>
<tr>
<th></th>
<th>$t$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_1$</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>$N_2$</td>
<td>20</td>
<td>40</td>
</tr>
</tbody>
</table>

---

1. This solution will already meet the deadline as shown in Fig. 3b. In case of the most hardened architecture depicted in Fig. 3c, only one re-execution is needed. However, using $N_1^3$ will cost twice as much as the previous solution with less hardening. Moreover, due to performance degradation, the solution with the maximal hardening
Our design strategy is outlined in Fig. 5. As an input we get the application graph \( G \) the set of computation nodes \( N \), deadline \( D \), and the re-execution graph \( \mathcal{G} \) of the selected set of nodes, the number of re-executions to be supported in software, the mapping of the application, and, finally, the static schedule.

The design heuristic explores the set of architectures, and eventually selects that architecture, which minimizes cost, while still meeting the schedulability and reliability requirements of the application. The heuristic starts with the monoprocessor architecture \( n = 1 \), composed of only one (fastest) node (lines 1-2). The mapping, selection of software and hardware redundancy (re-executions and hardening levels) and the schedule are obtained for this architecture (lines 5-9). If the application is unschedulable, the number of computation nodes is directly increased, and the fasted architecture with \( n = n + 1 \) nodes is chosen (line 15). If the application is schedulable on that architecture with \( n \) nodes, i.e., \( SL \leq D \), the cost \( C \) of that architecture is stored as the best-so-far cost \( C_{\text{best}} \). The next fastest architecture with \( n \) nodes (in the case of no hardening) is then selected (line 18). If on that architecture the application is schedulable (after hardening is introduced) and the cost \( C < C_{\text{best}} \) it is stored as the best-so-far. The procedure continues until the architecture with the maximum number of nodes is reached and evaluated.

If the cost of the next selected architecture with the minimum hardening levels is higher than the best-so-far cost \( C_{\text{best}} \), such architecture will be ignored (line 6).

The evaluation of an architecture is done at each iteration step with the MappingAlgorithm function. MappingAlgorithm receives as an input the selected architecture, produces the mapping, and returns the schedule corresponding to that mapping. The cost function used for optimization is also given as a parameter. We use two cost functions: (1) schedule length, which produces the shortest-possible schedule length \( SL \) for the selected architecture for the best-possible mapping (line 7), and (2) architecture cost, in which the mapping algorithm takes an already schedulable application as an input and then optimizes the mapping to improve the cost of the application without impairing the schedulability (line 9). MappingAlgorithm tries a set of possible mappings (as, for example, in Fig. 4), and for each mapping it optimizes the levels of redundancy in software and hardware, which are required to meet the reliability goal \( \rho \). The levels of redundancy are optimized inside the mapping algorithm with the RedundancyOpt heuristic presented in Sect. 6.3, which returns the levels of hardening and the number of re-executions in software. The function dependencies are shown in Fig. 5. The re-executions in software are obtained with ReExecutionOpt heuristic, called inside

DesignStrategy(\( g, n, D, \rho \))
1. \( n = 1 \)
2. \( \mathcal{AR} = \text{SelectArch}(\mathcal{G}, n) \)
3. \( C_{\text{best}} = \text{MAX}\_\text{COST} \)
4. while \( n \leq |\mathcal{G}| \) do
5. SetMinHardening(\( \mathcal{AR} \))
6. if \( C_{\text{best}} > \text{GetCost}(\mathcal{AR}) \) then
7. \( SL = \text{MappingAlgorithm}(\mathcal{G}, \mathcal{AR}, D, \text{ScheduleLength}) \)
8. if \( SL \leq D \) then
9. \( C = \text{MappingAlgorithm}(\mathcal{G}, \mathcal{AR}, D, \text{Cost}) \)
10. if \( C < C_{\text{best}} \) then
11. \( C_{\text{best}} = C \)
12. \( \mathcal{AR}_{\text{best}} = \mathcal{AR} \)
13. end if
14. else
15. \( n = n + 1 \)
16. end if
17. end if
18. \( \mathcal{AR} = \text{SelectNextArch}(\mathcal{G}, n) \)
19. end while
20. return \( \mathcal{AR}_{\text{best}} \)
21. end DesignStrategy

Figure 3. Hardware Recovery vs. Software Recovery (1)

Figure 4. Hardware Recovery vs. Software Recovery (2)

Figure 5. General Design Strategy
RedundancyOpt for each vector of hardening levels. Then the obtained alternative of redundancy levels is evaluated in terms of schedulability by the off-line scheduling algorithm Scheduling, which is shortly described in Sect. 6.4. After completion of RedundancyOpt, Scheduling is called again to determine the schedule for each selected mapping alternative in MappingAlgorithm.

6.1 Illustrative Example
The basic idea behind our design strategy is that the change of the mapping immediately triggers the change of the hardening levels. Thus, there is no need to directly change hardening since it can be guided by the mapping. To illustrate this, let us consider the application \( A \) in Fig. 1 and mapping in Fig. 4a. Processes \( P_1 \) and \( P_2 \) are mapped on \( N_1 \), while processes \( P_3 \) and \( P_4 \) are mapped on \( N_2 \). Both nodes, \( N_1 \) and \( N_2 \), have the second hardening level \( (h = 2) \), \( N_1^2 \) and \( N_2^2 \). With this architecture, according to our SFP calculation, one re-execution is needed on each node in order to meet the reliability goal. As can be seen in Fig. 4a, the deadlines are satisfied in this case. If, however, processes \( P_1 \) and \( P_2 \) are moved (re-mapped) onto node \( N_2 \), resulting in the mapping in Fig. 4e, then using the third hardening level \( (h = 3) \) is the only option to guarantee the timing and reliability requirements, and this alternative will be chosen by our algorithm for the respective mapping. If, for a certain mapping, the application is not schedulable with any available hardening level, for example, the mapping in Fig. 4d, this mapping will be discarded by our algorithm.

6.2 Mapping Optimization
In our design strategy we use the MappingAlgorithm heuristic with two cost functions, schedule length and the cost. We have extended the algorithm from [7, 15] to consider the different hardening and re-execution levels. The mapping heuristic investigates the processes on the critical path. Thus, at each iteration, processes on the critical part are selected for the re-mapping. Processes recently re-mapped are marked as “tabu” (by setting up the “tabu” counter) and are not touched. Processes, which have been waiting for a long time to be re-mapped, are assigned with the waiting priorities and will be re-mapped first. The heuristic changes the mapping of a process if it leads to (1) the solution that is better than the best-so-far (including “tabu” processes), or (2) to the solution that is worse than the best-so-far but is better than the other possible solutions. At every iteration, the waiting counters are increased and the “tabu” counters are decreased. The heuristic stops after a certain number of steps without any improvement.

Moreover, in order to evaluate a particular mapping, for this mapping we have to obtain the hardening levels in hardware and the maximum number of re-executions in software. This is performed in the RedundancyOpt function, presented in the next section.

6.3 Hardening/Re-execution Trade-off
Every time we evaluate a mapping move by the MappingAlgorithm, we run RedundancyOpt to obtain hardening levels in hardware and the number of re-executions in software (the latter obtained with ReExecutionOpt). The heuristic takes as an input the architecture \( \mathcal{R} \) with the minimum hardening levels and the given mapping \( \mathcal{M} \).

At first, the heuristic increases the schedulability of the application by increasing the hardening levels in a greedy fashion, obtaining the number of re-executions for each vector of hardening. The schedulability is evaluated with the Scheduling heuristic. Once a schedulable solution is reached, we iteratively reduce hardening by one level for each node, again, at the same time obtaining the corresponding numbers of re-executions. For example, in Fig. 4a, we can reduce from \( N_1^2 \) to \( N_1^1 \), and from \( N_2^2 \) to \( N_2^1 \). If the application becomes unschedulable, for example, in the case we reduce from \( N_1^1 \), such a solution is not accepted. Among the schedulable hardened alternatives, we choose the one with the lowest cost and continue. The heuristic iterates while improvement is possible, i.e., there is at least one schedulable alternative. In Fig. 4a, the heuristic will stop once \( h \)-versions \( N_1^2 \) to \( N_2^2 \) have been reached, since the solutions with less hardening are not schedulable.

The ReExecutionOpt heuristic is called in every iteration of RedundancyOpt to obtain the number of re-executions in software. The heuristic takes as an input the architecture \( \mathcal{A} \) and mapping \( \mathcal{M} \) and the hardening levels \( \mathcal{H} \). It starts without any re-executions in software and increases the number of re-executions in a greedy fashion. The heuristic uses the SFP analysis and gradually increases the number of re-executions until the reliability goal \( \rho \) is reached. The exploration of the number of re-executions is guided towards the largest increase in the system reliability. For example, if increasing the number of re-executions by one on node \( N_1 \) will increase the system reliability from \( 1 - 10^{-5} \) to \( 1 - 10^{-3} \) and, at the same time, increasing re-executions by one on node \( N_2 \) will increase the system reliability from \( 1 - 10^{-1} \) to \( 1 - 5 \times 10^{-3} \), the heuristic will choose to introduce one more re-execution on node \( N_2 \).

6.4 Scheduling
In this paper we adapt an off-line scheduling strategy, which we have proposed in [7, 15], that uses “recovery slack” in order to accommodate the time needed for re-executions in case of faults. After each process \( P_i \) we assign a slack equal to \( (\tau_{ij} + \mu) \times k_j \), where \( k_j \) is the number of re-executions on the computation node \( N_j \) with hardening \( h \). The slack is shared between processes in order to reduce the time allocated for recovering from faults.

The Scheduling heuristic is used by the RedundancyOpt and mapping optimization heuristics to determine the schedulability of the evaluated solution, and produces the best possible schedule for the final architecture.

7. Experimental Results
For the experiments, we have generated 150 synthetic applications with 20 and 40 processes. The worst-case execution times (WCETs) of processes, considered on the fastest node without any hardening, have been varied between 1 and 20 ms. The recovery overhead \( \mu \) has been randomly generated between 1 and 10% of process WCET.

Regarding the architecture, we consider nodes with five different levels of hardening. The failure probabilities of processes running on different \( h \)-versions of computation nodes have been obtained using fault injection experiments. We have considered three fabrication technologies with the average transient (soft) error rates (SER) per clock cycle at the minimum hardening level of \( 10^{-10} \), \( 10^{-11} \), and \( 10^{-12} \), respectively, where \( 10^{-10} \) corresponds to the technology with the highest level of integration and the smallest transistor sizes.

The hardening performance degradation (HPD) from the minimum to the maximum hardening level has been varied from 5% to 100%, increasing linearly with the hardening level. For a HPD of 5%, the WCET of processes increases with each hardening level with 1, 2, 3, 4, and 5%, respectively; for HPD = 100%, the increases will be 1, 25, 50, 75, and 100% for each level, respectively. Initial processor costs (without hardening) have been generated between 1 and 6 cost units. We have assumed that the hardware cost increases linearly with the hardening level. The system reliability requirements have been varied between \( \rho = 1 - 7.5 \times 10^{-6} \) and \( 1 - 2.5 \times 10^{-5} \) within one hour. The deadlines have been assigned to all the applications independent of the transient error rates and hardening performance degradation of the computation nodes. The experiments have been run on a Pentium 4 2.8 GHz processor with 1Gb memory.
niques. However, as $SER$ is increased to $10^{-11}$, our OPT strategy already outperforms MIN. For $SER = 10^{-10}$, OPT is significantly better than both other strategies since in this case finding a proper trade-off between the levels of redundancy in hardware and the levels of software re-execution becomes more important.

The execution time of our OPT strategy for the examples that have been considered is between 3 minutes and 60 minutes.

We have also run our experiments on a real-life example, a vehicle cruise controller (CC) composed of 32 processes [8]. The CC considers an architecture consisting of three nodes: Electronic Throttle Module (ETM), Anti-lock Braking System (ABS) and Transmission Control Module (TCM). We have set the system reliability requirements to $p = 1 - 1.2 \times 10^{-5}$ within one hour and considered $\mu$ between 1 and 10% of process average-case execution times. The $SER$ for the least hardened versions of modules has been set to $2 \times 10^{-12}$; five $h$-versions have been considered with $HPD = 25\%$ and linear cost functions. We have considered a deadline of 300 ms. We have found that CC is not schedulable if the MIN strategy with the minimum hardening levels has been used. However, CC is schedulable with the MAX and OPT approaches. Moreover, our OPT strategy with the trading-off between hardware and software redundancy levels has produced results 66% better than the MAX in terms of cost.

8. Conclusions

In this paper we have considered hard real-time applications mapped on distributed embedded architectures. We were interested to derive the least costly implementation that meets imposed timing and reliability constraints. We have considered two options for increasing the reliability: hardware redundancy and software re-execution.

We have proposed a design optimization strategy for minimizing of the overall system cost by trading-off between processor hardening and software re-execution. Our experimental results have shown that, by selecting the appropriate level of hardening in hardware and re-executions in software, we can satisfy the reliability and time constraints of the applications while minimizing the cost of the architecture. The optimization relies on a system failure probability analysis, which connects the level of hardening in hardware with the number of re-executions in software.

9. Appendix A

A.1 System Failure Probability (SFP) Analysis

In this appendix we present an analysis that determines the system failure probability, based on the number of re-executions in software and the process failure probabilities on the computation nodes with different hardening levels.

The process failure probability $p_{f0}$ of process $P_i$, executed on computation node $N_{ij}$ with hardening level $h$, is obtained with simulation using fault injection tools such as [1, 18]. Merging of a process $P_i$ on the $h$-version of computation node $N_{ij}$ will be denoted as $M(P_i) = N_{ij}^h$.

In the analysis, first, we calculate the probability $Pr(0; N_{ij})$ of no faults occurring (no faulty processes) during one iteration of the application on the $h$-version of node $N_{ij}$. This is the probability that all processes mapped on $N_{ij}$ will be executed correctly:

$$Pr(0; N_{ij}) = \prod_{P_i \in M(P_i) = N_{ij}^h} (1 - p_{ij})$$

To account for faulty processes and re-executions, we will first refer to $f$-fault scenarios as to combinations with repetitions of $f$ faults on the number $\Pi(N)$ of processes mapped on the computation node $N_{ij}$. Under a combination with repetitions of $n$ on $m$, we will understand the process of selecting $n$ elements from a set of $m$ elements, where each element can be selected more than once and the order of selection does not matter [17].

For example, an application $\lambda$ is composed of processes $P_1$, $P_2$, and $P_3$, which are mapped on node $N_{ij}$. $f = 3$ transient faults may occur, e.g. $f = 3$. Let
us consider one possible fault scenario. Process $P_1$ fails and is re-executed, its re-execution fails but then it is re-executed again without faults. Process $P_2$ fails once and is re-executed without faults. Thus, in this fault scenario, from a set of processes $P_1$, $P_2$, and $P_3$, processes $P_1$ and $P_2$ are selected; moreover, process $P_3$ is selected twice, which corresponds to one repetition.

The probability of recovering from a particular combination of $f$ faults consists of two probabilities, the probability that this combination of $f$ faults has happened and that all the processes, mapped on $N_j$, will eventually be re-executed without faults. The latter probability is, in fact, the no fault probability $Pr(0; N_j)$.

According to the problem formulation, the system non-failure probability is

$$
Pr_S = Pr(0; N_j) \sum_{S^* \in (m^*)} Pr(S^*; m^*)
$$

where $|S^*| = f$, $|m^*| = |S|$, $S \subset N$, $|S| = \Pi_{j=1}^{m} sup(m(a) \in S)$. The combination with repetitions is expressed here with a finite multiset $(S^*, m^*)$ of a multiset $(S, m)$ [17]. Informally, a multiset is simply a set with repetitions. Formally, in the present context, we define each our finite multiset as a function $m: S \rightarrow N$ on set $S$, which includes indices of all processes mapped on $N_j$, to the set $N$ of (positive) natural numbers. For each process $P_j$ with index $a$ in $S$ the number of repetitions is the number $m(a)$, which is less or equal to $f$ faults (expressed as a supremum of function $m(a)$). The number of elements in $S^*$ is $f$, i.e. the number of faulty processes. Thus, if $a$ is repeated $f$ times, $m(a) = f$, i.e., $P_a$ fails $f$ times, $S^*$ will contain only repetitions of $a$ and nothing else.

From (2), the probability that the system recovers from all possible faults is a sum of probabilities of all fault recovery scenarios:

$$
Pr_S = Pr(0; N_j) \sum_{(S^*, m^*) \subset (S, m^*)} \prod_{j=1}^{m} Pr(j; N_j)
$$

Suppose that we consider a situation with maximum $k_j$ re-executions on the $h$-version of the node $N_j$. The node fails if more than $k_j$ faults are occurring. From (3) and (1), we will derive the failure probability of the $h$-version of node $N_j$ with $k_j$ re-executions as

$$
Pr(f > k_j; N_j) = 1 - Pr(0; N_j) - \sum_{k=1}^{k_j} \prod_{j=1}^{m} Pr(f_j; N_j)
$$

where we subtract from the initial failure probability with only hardware redundancy, $1 - Pr(0; N_j)$, the probabilities of all the possible successful recovery scenarios provided with $k_j$ re-executions.

Finally, the probability that the system composed of $n$ computation nodes with $k_j$ re-executions on each node $N_j$ will not recover, in the case more than $k_j$ faults have happened on any computation node $N_j$, can be obtained as follows:

$$
Pr(f > k_j; N_j) = 1 - \prod_{j=1}^{n} \left(1 - \sum_{k=1}^{k_j} \prod_{j=1}^{m} Pr(f_j; N_j)\right)
$$

According to the problem formulation, the system non-failure probability in the time unit $t$ (i.e., one hour) of functionality has to be above the reliability goal $p = 1 - \gamma$, where $\gamma$ is the maximum probability of a system failure due to transient faults within the time unit $t$. Considering the calculations above have been performed for one iteration of the application (i.e., within a period $T$), we obtain the following condition for our system to satisfy the reliability goal

$$
1 - \prod_{j=1}^{n} \left(1 - \sum_{k=1}^{k_j} \prod_{j=1}^{m} Pr(f_j; N_j)\right) > p
$$

### A.2 Computation Example

To illustrate how the formulate (1)-(6) can be used in obtaining the number of re-execution be introduced at software level, we will consider the architecture in Fig. 4a. At first, we compute the probability of no faulty processes for both nodes $N_1$ and $N_2$:

$$
Pr(0; N_1) = \left(1 - (1.2 \cdot 10^{-1}) \cdot (1 - 1.3 \cdot 10^{-8})\right) = 0.99997500015
$$

$$
Pr(0; N_2) = \left(1 - (1.2 \cdot 10^{-1}) \cdot (1 - 1.3 \cdot 10^{-8})\right) = 0.99997500015
$$

The combinations of faults in the re-executions are mutually exclusive.

Symbols [ ] indicate that numbers are rounded up with $10^{-11}$ accuracy, and [ ] indicate that numbers are rounded down with $10^{-11}$ accuracy. It is needed for pessimism of fault-tolerant design.