Reliability-Aware Design Optimization for Multiprocessor Embedded Systems

Jia Huang, Jan Olaf Blech, Andreas Raabe, Christian Buckl
fortiss GmbH
Guerickestr. 25, 80805 Munich, Germany
{*huang,blech,raabe,buckl*}*@fortiss.org*

Alois Knoll
Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
knoll@in.tum.de

***Abstract*—This paper presents an approach for the reliability-aware design optimization of real-time systems on multi-processor platforms. The optimization is based on an extension of well accepted fault- and process-models. We combine utilization of hardware replication and software re-execution techniques to tolerate transient faults. A System Fault Tree (SFT) analysis is proposed, which computes the system-level reliability in presence of the hardware and software redundancy based on component failure probabilities. We integrate the SFT analysis with a Multi-Objective Evolutionary Algorithm (MOEA) based optimization process to perform efficient reliability-aware design space exploration. The solution resulting from our optimization contains the mapping of tasks to processing elements (PEs), the exact task and message schedule and the fault-tolerance policy assignment. The effectiveness of the approach is illustrated using several case studies.**

I. Introduction

With the continuous shrinking of transistor sizes, modern devices are becoming more susceptible to faults. Such faults can be permanent ones, e.g., device defects or transient ones, e.g., single event upsets (SEUs) caused by electromagnetic interference. Several research studies have shown that transient faults occur much more frequently than permanent faults, and their number is increasing with new technology generations [1]. Consequently, in many embedded systems, especially those for safety-related applications, the capability to provide reliable execution even in the presence of transient faults becomes a major requirement.

One traditional way to enhance reliability of systems is to use hardware redundancy (also known as *spatial redundancy*). For example, in a triple-modular-redundancy (TMR) system, the critical components are replicated three times and the results from the three redundant components are voted (usually based on the majority criterion) to produce the output. Such a system can then detect and correct a single fault in any of the three replicas. However, hardware replication often incurs high design and production cost. As the counterpart, the idea of software redundancy (or temporal redundancy) is proposed to reduce the cost. One possible approach is to schedule critical tasks multiple times and perform voting of the results [2]. Another common technique is to insert checkpoints into the software and re-execute the task in case faults are detected [3], [4]. For real-time applications, software redundancy must be used with utmost care, since the overhead in time may lead to deadline violations. The schedulability issue in the context of software redundancy has therefore become a very important topic [5], [6], [7].

In many utilization scenarios, the optimal implementation which respects all design constraints can only be achieved with simultaneous application of multiple fault-tolerance techniques. For example, in [8], the authors show that a schedulable and cost-efficient solution can be achieved by combined utilization of spatial and temporal redundancy. The combination of software redundancy and hardware hardening techniques is considered in [9]. Also, the safety standards sometimes have special requirements or recommendations of the fault-tolerance techniques to be applied. For example, [10] requires a device certifiable to Safety Integrity Level (SIL) 4 to implement at least hardware fault tolerance of one. This means, pure software techniques even with a large amount of redundancy are not sufficient to achieve the desired level of reliability for SIL4. On the other hand, a pure hardware-based solution might become prohibitively expensive. Therefore, integrating several fault-tolerance techniques and analyzing the overall system is a critical task. Modern Multiprocessor System-on-Chip (MPSoC) platforms provide adequate hardware resources and flexibility to well explore the tradeoff between different fault-tolerance techniques.

The focus of this paper is on the reliability-aware design optimization for real-time embedded systems. We consider the combined utilization of hardware replication and software re-execution techniques to tolerate transient faults. Our work is based on well accepted fault- and process models, e.g., these models have been used in [9], [11]. The design optimization is performed using Multi-Objective Evolutionary Algorithms (MOEA), the result of which contains the mapping of tasks to Processing Elements (PEs), the exact task and message schedule and the amount of the hardware/software redundancy to be applied. We illustrate the effectiveness of the approach using several case studies.

The main contribution of this paper is: 1) a system fault tree analysis that computes the system-level failure probability in the presence of hardware/software redundancy; 2) an approach that integrates the proposed reliability analysis into a MOEA based optimization framework; 3) an inter-job

slack sharing scheme for further reliability enhancement.

The rest of the paper is organized as follows. Section II provides a related work review. The system models used in this paper are introduced in Section III. Section IV and V present the proposed reliability analysis and design optimization approaches. Experimental results are presented in Section VI. Section VII concludes this paper.

II. Related Work

Reliability-aware design consists of two major tasks: modeling/analyzing reliability and integration of reliability into the design process. An overview can be found in [12].

Reliability analysis is typically performed in a hierarchical manner, from the reliability model of individual components up to the system-level model. In [13], the authors present a symbolic approach for reliability analysis focusing on permanent faults. The so called structure function is introduced and represented as a Binary Decision Diagrams (BDDs). It describes the system behavior under the influence of faults. Then, based on the reliability model of the components and the structure function, system-level reliability is evaluated. The work [9] describes a system failure probability analysis that determines the system reliability based on the amount of software redundancy and component failure rate. It is also integrated into a tabu-search based optimization procedure. Recent work [14] proposes a modeling framework that integrates device, component and system level models.

In [2], the authors describe an approach that enhances the reliability by selectively inserting task re-executions. The reliability analysis introduced in [13] is integrated into a MOEA based optimization framework in [15]. The spatial redundancy is considered and represented by binding the same task to multiple PEs. No software fault-tolerance technique is considered since only permanent faults are regarded. The same authors further consider the automatic insertion of voting components in [16]. There are also studies that consider the tradeoff between reliability and other design objectives, such as energy [11] and cost [3].

Our work is most closely related to [8], [3], [17]. In [8] the authors study the design optimization of fault-tolerance systems using both hardware and software redundancy. The case for combined utilization of check pointing and hardware replication is considered in [3]. In [17], the authors propose a hybrid scheduling approach for mixed hard and soft real-time tasks. In the work mentioned above, the optimization framework automatically determines the task-to-PE mapping and fault-tolerance policy assignment, e.g., the amount of replication and placement of check points. We took their fault- and process-model as a starting point for our work and adapted it to our needs. In their fault model, a total number of faults that may occur in any components of the system is assumed. However, in many systems, the component failure probability can be highly distinct depending on the type of hardware and such an assumption might be inaccurate. We therefore propose an accurate probabilistic analysis for the system-level reliability. In addition to this, we introduce an approach based on Evolutionary Algorithms that allows us to consider multiple optimization objectives, e.g., reliability, schedule length and resource utilization.

III. System Models

We consider an application A as a set of independent periodic *jobs* running simultaneously in the system. A job $J \in A$ is a directed acyclic graph, whose vertices V represent a set of tasks to be executed and the edges E capture data dependencies between tasks. For each edge, a message is associated to represent the data transfer. We assume that the set of jobs in A share the same period. If jobs have originally different periods, they are first transformed into larger graphs representing a hyper-period (LCM of all periods) of the application. We use T to denote the set of all tasks in application A and $T(n)$ to represent the set of tasks mapped to processor n.

Our target architecture is a Multiprocessor System-on-Chip (MPSoC) with time-triggered on-chip communication. The GENESYS [1] platform is one example of such an architecture. The set of available Processing Elements (PEs) is denoted using N. The communication bus is arbitrated using TDMA. If the two communicating tasks are mapped to the same PE, data transfer can be realized via local memory and no bus slot is needed. Otherwise, a dedicated time slot needs to be reserved. The message transfer is currently assumed to be fault-free. The consideration of message fault and optimization of the fault handling techniques are addressed in future work.

Timing predictability is highly desirable for safety-related applications. In this paper, we target on synthesizing *static time-triggered schedules*. Such a schedule S is a set of sub-schedules, each describing the scheduling information for a specific PE. There must be exactly one sub-schedule S_n for each PE $n \in N$. A sub-schedule consists of a set of scheduling slots. A scheduling slot is a 3-tuple $s = (t_s, t_f, T)$, where t_s is the start time of the slot, t_f is the finish time and T is a set of tasks that may execute in the slot. A slot can be a normal task execution slot or a re-execution slot (also called slack slot). The later is meant to be shared by multiple tasks and used for re-execution of instances misbehaving due to transient faults. Figure 1a depicts an example schedule that has three slots for processor n_1 and n_2 each.

The actual utilization of scheduling slots depends on how the scheduler responds to the occurred faults. In this paper, we consider a static non-preemptive scheduling approach. In this case, the task $t \in s.T$ [2] that has the highest priority among all pending tasks acquires the slot. Figure

[1] http://www.genesys-platform.eu/

[2] The notation $s.X$ denotes the element X in the tuple s in the entire paper.

1 demonstrates two example execution scenarios. In 1b, the slot S_2 is used to re-execute t_1, since both S_0 and S_1 fail. In 1c, the same slot is used for t_2, since t_1 is already finished with S_0. We use the same assumption as in [8], [9], [17], [11] that the transient faults are detected using sanity checks at the completion of a tasks' execution. The timing overhead of fault detection is assumed to be contained in the Worst-Case Execution Time (WCET) of tasks.

The combination of faults that occur is described by a fault scenario F, which is a set of partial fault scenarios, one for each PE. A partial fault scenario F_n for PE n is a vector of integers of length $|T(n)|$, specifying the number of faults happening to each of the tasks mapped to n. A partial fault scenario with x faults in total compares to a selection of x tasks out of $T(n)$, where each task can be selected multiple times and the order does not matter [9], [18]. For a specific PE n, the maximum number of tolerable faults depends on the amount of redundancy. We define the fault-tolerance capability $max(S, n)$ as the number of re-execution slots scheduled on n plus the number of tasks which are replicated at least once in other PEs. Obviously, a fault-scenario is not expected to be tolerable if the amount of faults specified for any processor n is higher than $max(S, n)$. To analyze the system-level reliability, we are interested in identifying the set of fault scenarios that can be tolerated using the current schedule. To do this it is safe to investigate only the finite set of fault-scenarios:

$$\hat{F}(S) = \{F | \forall n \in N \ : \sum_{t \in T(n)} F_n(t) \leq max(S, n))\}.$$

IV. System Fault Tree Analysis

This section presents the System Fault Tree (SFT) analysis that computes the system failure probability (SFP) based on the component failure rate and the amount of software/hardware redundancy. In general, to compute the failure probability of a specific job J, we need to identify the complete set of tolerable fault scenarios that does not lead to a failure of J. Such a set is called the *working set* of J with schedule S, denoted by $W(J, S)$.

We define a function σ, which takes an application A, a schedule S and a fault scenario $F \in \hat{F}(S)$ and returns a set of booleans, representing the execution result (success or failure) of each job $J \in A$ under the impact of faults specified in F. A job succeeds if at least one instance of each of its tasks is executed without fault before the deadline $D(T)$. Evaluation of σ compares to a symbolic execution of the schedule according to the fault scenario, and then identify the result for each job from the trace. Figure 1 shows the execution traces of two example fault scenarios. As it can be seen, although F_1 and F_2 specify both three faults, F_1 can be tolerated with the current schedule whereas F_2 leads to a failure of J_1 (since t_2 is not executed successfully). Note that some fault scenarios may not be applicable according to the scheduling policy. Consider the setup in Figure 1

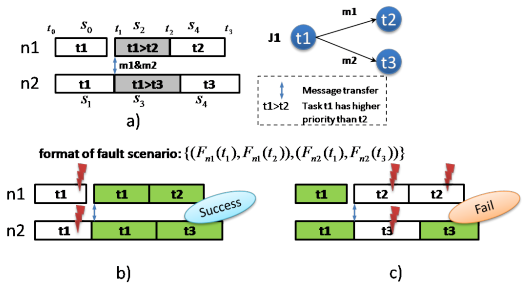

Figure 1. Example Fault Scenario

for example, the fault scenario $F = \{(2, 0), (0, 1)\}$ is non-applicable, because it specifies two faults in t_1 on n_1, but the second execution of t_1 on n_1 will not occur since a replica of p_1 is already executed successfully on n_2 and a message is transfered to n_1. The set of non-applicable fault scenarios must be excluded from the working set.

Computing the working set can be done via fault tree analysis. Figure 2 illustrates the analysis for the schedule depicted in Figure 1. From the none-fault case $\{(0, 0)(0, 0)\}$, we iteratively increase the number of faults and check if the new fault scenario is still tolerable. The procedure builds a tree structure representing the possible fault scenarios. The depth of the tree is restricted by the fault-tolerance capabilities $max(S, n)$ of the PEs. The failure or non-applicable nodes of the tree will not spawn further branches.

After obtaining the working set, the success probability of a job (denoted by $Pr(J, S)$) can be computed by summarizing the occurrence probability of the set of tolerable fault scenarios $Pr(F)$:

$$Pr(J, S) = \sum_{F \in W(J,S)} Pr(F) \tag{1}$$

Let $Pr(t, n)$ denote the success probability of task t on node n, the occurrence probability of fault scenario F is:

$$Pr(F) = \prod_{n \in N} \Big(\prod_{t \in T(n)} (1 - Pr(t, n))^{F_n(t)} \cdot \prod_{\substack{t \in Succ \\ (F,S,n)}} Pr(t, n)) \tag{2}$$

Where $T(n)$ is the set of tasks mapped on n, $F_n(t)$ is the number of faults on task t specified in F_n and $Succ(F, S, n)$ is the set of successfully finished tasks on n. $Succ(F, S, n)$ can be obtained from the trace of evaluating $\sigma(A, S, F)$.

The number of nodes visited during the fault-tree analysis increases exponentially with the depth. Let $|F| = \sum_{n \in N} |F_n|$ be the total number of entries in the fault scenario, we need to visit $\binom{|F|+d-1}{d}$ nodes at depth d in the worst case. This implies that the computational complexity also increases exponentially. Since the stand-alone failure rates of the tasks are typically very low, the nodes located deeper in the fault-tree have much lower occurrence probability. Moreover, the portion of non-tolerable fault-scenarios will also increase significantly as the depth increases. In Figure 2 for example,

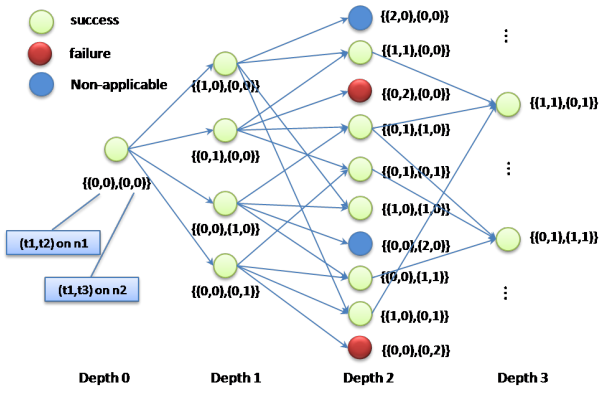

Figure 2. Example Fault Tree Analysis

only 2 out of 20 fault scenarios are tolerable at depth 3. Thus, in many circumstances, a safe underestimation of system reliability would be to consider only a bounded depth during analysis. This compares to assuming a maximum number of faults that may occur anywhere in one period of execution.

A. *Inter-Job Slack Sharing*

The above analysis is able to handle the case for shared re-execution slots for tasks within the same job (intra-job slack sharing). In this paper, we propose in addition an inter-job slack sharing scheme. This scheme is motivated by the emerging needs to cope with mixed-criticality jobs, i.e. applications with highly distinct reliability requirements running in the same platform. For high criticality jobs, significant amount of software redundancy is needed to meet the high reliability requirements. However, the probability that the software slack is actually used is typically very low. In this case, reusing the slack time for low criticality jobs using the static-priority approach may lead to significant saving of hardware resources. We demonstrate this point using an example shown in Figure 3.

Assume two jobs J_1 and J_2 having the same period and deadline is to be scheduled. J_1 is a high-criticality task requiring failure probability lower than $1 \cdot 10^{-11}$ and J_2 is a low-criticality task with the requirement $5 \cdot 10^{-6}$. Simple analysis shows that three re-execute slots are needed to fulfill the requirement of J_1, if it is scheduled on a single processor (Figure 3a). However, the deadline is violated in this case. Figure 3b and 3c present two feasible schedules that meet both deadline and reliability requirements of J_1 on two processors via combined utilization of redundancy in space and time domain. For both cases, the low-criticality job can not be scheduled using the remaining resources on processor N_1 and N_2, hence, a third processor N_3 is needed for J_2. However, using the proposed inter-job slack-sharing approach, the two jobs can be scheduled as shown in Figure 3d with all requirements met. The low-criticality job J_2 is assigned to low priority on the shared re-execution slack, i.e. the low-criticality job J_2 will use the re-execution slot reserved for high-criticality task J_1, only if

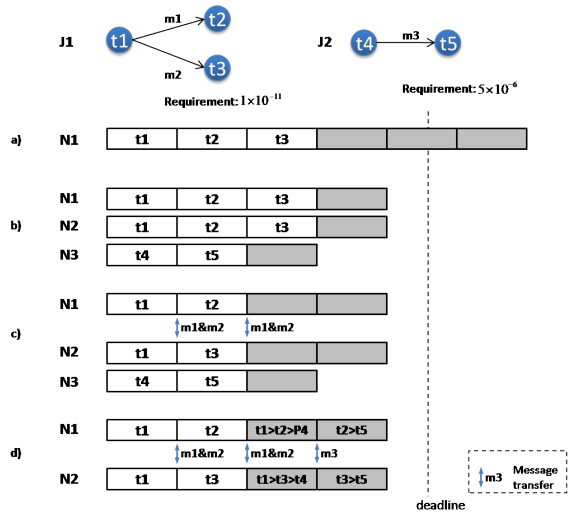

Figure 3. Motivating Example of Inter-Job Slack Sharing

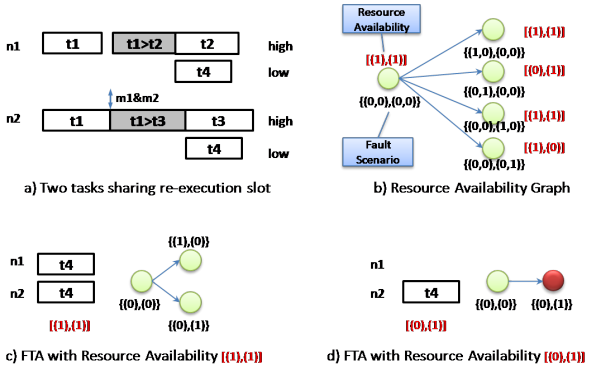

Figure 4. SFT Analysis under Resource Availability Constraints

J_1 successfully finishes without using that slot. Due to the fact that the failure rate of J_1 is low, such a setup already fulfills the requirement of J_2. In this case, two processors are sufficient for executing J_1 and J_2.

The example shows the benefit of the slack-sharing. Another important observation is that, when slack sharing is used, the system-level optimal allocation strategy might be different from the job-level optimum. For example, the two schemes in Figure 3b and Figure 3c have the same resource consumption for J_1(in total 8 execution slots on two processors). However, the further task J_2 is schedulable on two processors only using the scheme shown in Figure 3c. This implies the need for a system-level optimization strategy.

The SFT analysis needs to be extended to cope with the inter-job slack sharing scheme. The extended procedure is explained as follows using an example (Figure 4). A sketch of the algorithm is also provided in Algorithm 1. Assume that a low priority job consisting of a single task t_4 shares the last slots of job J_1 in Figure 1. We first do the SFT for the high priority job J_1. Besides determining the failure probability, the SFT also gathers information about which slots are still left for the low priority job (Figure 4b). As

it can be seen, when no fault occurs or only a single fault occurs on t_1, both re-execution slots are available for t_4 (availability scenario $[(1),(1)]$). When a fault occurs on t_2 or t_3, only one slot is left for t_4 (availability scenario $[(0),(1)]$ or $[(1),(0)]$). A partial schedule of t_4 can be built based on the resource availability and the corresponding SFT can be performed (Figure 4c and 4d). In a certain availability scenario, the occurrence probability of a fault scenario is computed as:

$$Pr(F, RA) = Pr(RA) \cdot Pr(F|RA) \tag{3}$$

Where $Pr(RA)$ is the probability of the availability scenario computed from SFT of high priority tasks and $Pr(F|RA)$ is the occurrence probability of F in the partial schedule associated with RA. One possible way to reduce the complexity of the extended SFT analysis is to ignore resource availability scenarios with occurrence probability lower than a threshold value. This is obviously again a safe underestimation of reliability.

Algorithm 1 IterativeTreeAnalysis(): iterative SFT for multiple jobs using the inter-job slack sharing scheme. AS_{old}: the set of availability scenarios from previous job. AS_{new}: the set of availability scenarios for next job. The function *buildPartialSchedule* constructs the partial schedule for a job based on the availability of shared re-execution slots. Since different fault scenarios may result in the same availability scenario, a function *combine* is used to compute the overall occurrence probabilities.

```
1:  AS_old ← initAvailability();
2:  AS_new ← initAvailability();
3:  for all J ∈ 𝒜 with decreasing priority do
4:    for all a ∈ AS_old do
5:      S' = buildPartialSchedule(S,a)
6:      avail ← SFTAnalysis(S',J)
7:      combine(AS_new,avail)
8:    end for
9:    AS_old ← AS_new
10: end for
```

V. Optimization Heuristic

Our optimization strategy is based on the Multi-Objective Evolutionary Algorithm (MOEA). The MOEA optimizer works as follows. The algorithm maintains a set of candidate solutions called the population. During each iteration, the optimizer selects a subset of solutions as parents, which are manipulated using crossover and mutation operators to produce offspring. The new solutions are evaluated using the fitness function and high quality solutions will replace low quality ones in the population. This process can be repeated until a candidate with sufficient quality is found or a maximum number of iterations is reached. The efficiency of EA-based optimization is heavily influenced by the length

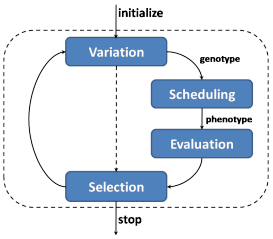

Figure 5. Workflow of EA-based Optimization

of the chromosome, since it determines the overall search space. However, a direct encoding of the schedule as discussed in Section III results in a very large chromosome. To cope with this problem, we utilize a two-step optimization process as shown in Figure 5 inspired from [19]. The main idea is that, instead of encoding the entire schedule, we only put partial information, namely the mapping and fault-tolerance policy, into the chromosomes. A scheduler is integrated into the general variation-selection process to transform the chromosome to an optimized schedule. The resulting schedule is then used for fitness evaluation, e.g., the SFT analysis.

Using the above approach, we encode the mapping and the fault-tolerance policy information as shown in Figure 6. The chromosome contains one gene for each task $t \in T$. Each gene is a pair $g = (i, j)$, where i is the integer index of the task and j is a list of integer indicating the set of PEs onto which task i is mapped. Multiple mappings of the same task onto the same PE are interpreted as re-execution slots (task 2 and 3 in Figure 6); multiple mappings of the same task onto different PEs are interpreted as spatial replications (task 4 in Figure 6). Reconstruction of the schedule from the chromosome is the same as scheduling the task executions with known mapping and fault-tolerance policy. A notable advantage of the two-step approach is that the scheduling algorithm is orthogonal to our encoding scheme in the sense that any existing scheduling algorithms for this purpose can be used. Moreover, as long as the scheduler is implemented correctly, the variation of the chromosome will always produce valid solutions.

The scheduling procedure we propose consists of three main steps. In the first step, a list scheduler is used to greedily schedule the tasks to the beginning of the scheduling period. The list scheduler determines the order of tasks based on their priorities. In our implementation, we use two criteria: tasks belonging to a job with shorter deadline have higher priority (job-level EDF); for tasks within the same job, the one that has longer path to the sink node has higher priority. An example is shown in Figure 6a. Note that data dependencies between tasks are automatically respected by the list scheduler. In the second step, *bus scheduling* is performed. For each task, which expects input from some predecessors mapped to other PEs, a non-overlapping bus slot is reserved for the message transfer. In this paper, we

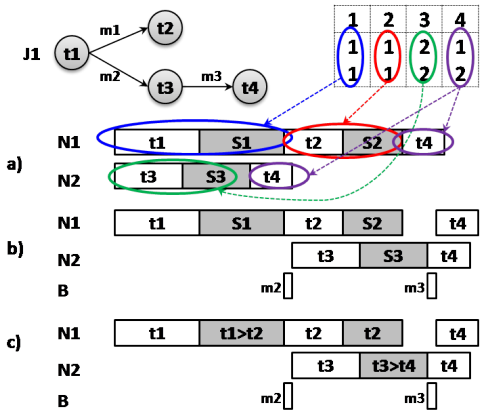

Figure 6. Computing the Schedule from Chromosome

consider the *transparent recovery* mechanism [20], where a fault happening on one PE is masked to other PEs. It has several advantages such as fault-containment and improved traceability. Transparent recovery requires that the message is scheduled in a way that the faults occurring to the sender are not visible to the receiver. For example, consider Figure 6b, the message m_2 can not be scheduled directly after execution of t_1, instead, it should be placed after the possible re-execution of t_1. Naturally, the receiver task t_3 must then be delayed to the end of the message transfer. It is a known limitation that transparent recovery may increase the overall schedule length.

In the third step, we do optimization of the schedule using the slack sharing scheme (Figure 6c). As mentioned before, the idea is to re-claim the unused slack slots. Note that the slack slots can be used to re-execute a task only if its size is larger than the WCET of that task. This implies that a larger slack slot is possible to be utilized for re-execution of a larger set of tasks, and therefore introduces higher benefit for system reliability, and vice verse. Hence, we need to determine properly the size of each slack slot. To reduce the complexity, we do not introduce the optimization of slack sizes explicitly, instead, we fix the size of the slack slots scheduled for a task t to the WCET of t. In Figure 6b for example, the chromosome specifies one slack slot for t_1 and one for t_2. The slot S_1 scheduled after t_1 is set to the WCET of t_1 and is sharable with t_2. In contrast, the slack slot S_2 of smaller size is not sharable with t_1. Due to the penalty in reliability, adding one slack slot for t_1 will be preferred by the optimizer. By mutating the number of slack slots scheduled for each task, we implicitly optimize the size of re-execution slack. Both intra-job and inter-job slack sharing schemes can be integrated in this step.

A. Crossover and Mutation

In order to improve the performance of the EA, we have implemented some crossover and mutation operators that add problem-specific knowledge to the optimization. We present those operators in the following:

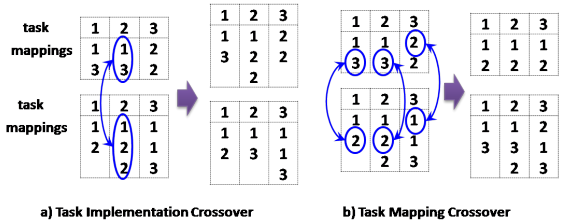

Figure 7. Task Implementation Crossover Example

Task Implementation Crossover: This operator randomly selects a set of tasks and swaps the entire implementation of these tasks between two chromosomes, including the amount of spatial/temporal redundancy and mapping. The rest of the chromosome remains unchanged. Figure 7a shows an example in which task 2 is selected for crossover.

Task Mapping Crossover: This operator performs crossover for the implementation of each task separately. Given two chromosomes, the mapping entries for a chosen task are randomly swapped. Figure 7b shows an example in which 3 mapping entries are selected for crossover in total.

Increment Redundancy: This mutation operator inserts a new mapping entry for a randomly selected task. Insertion of the new mapping x to task t might result in: 1) a slack slot, if the chromosome already contains a mapping of t to x, or 2) a spatial replication, if t has not been mapped to x.

Decrement Redundancy: The counterpart of *Increment Redundancy*, removes one mapping entry from a random task. There must be at least one mapping for each task.

Re-Mapping: This mutation operator randomly changes selected mapping entries. The result might be: 1) re-mapping of the tasks to other PEs or 2) transformation of a slack slot to a spatial replication or vice verse.

VI. Experiments

For the first experiment we applied the proposed design optimization flow to an MPEG2 decoder example [21]. The analysis and optimization algorithm are implemented in JAVA and run on a Windows machine with 4GHz CPU and 4GB memory. The MOEA is configured with a population of 100 implementations and runs for 300 generations. We assume that the target platform consists of two types of PEs, namely a RISC processor and a DSP. The failure probability of each task on a certain PE is randomly generated between $1 \cdot 10^{-5}$ and $1 \cdot 10^{-6}$. We restrict each task to have at most 2 spatial replicas and 2 re-execution slots. For the metric of reliability, we use the system failure probability per hour in logarithmic scale in all experiments, i.e. the lower the value is, the higher the reliability is.

An important step during embedded system design is design space exploration. For example, the designers may need to determine the amount and the type of PEs that are necessary to fulfill the application requirements. To illustrate this step, we first run the optimization procedure with five platform configurations consisting of 2 to 6 PEs. Figure 8

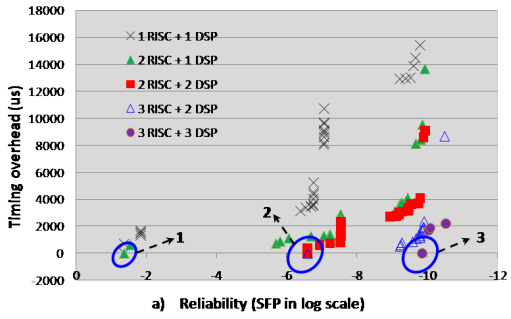

Figure 8. Optimal Solutions under Different Platform Configurations

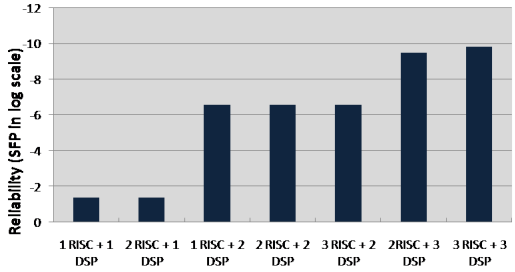

Figure 9. Achievable Reliability under Different Platform Configurations

shows the Pareto front results from the optimization. The horizontal axis shows the reliability and the vertical axis is the overhead of schedule length with respect to the deadline, i.e., overhead 0 implies meeting the deadline and a positive value means a deadline violation. It can be seen that, the Pareto fronts obtained with more PEs dominate in most cases the Pareto fronts obtained with less PEs, i.e. the application is finished with shorter time and higher reliability. This is due to the increased opportunity for spatial redundancy.

For each platform, we are interested in the solutions that achieve maximum reliability with deadline constraints fulfilled. These solutions are marked with 1 to 3 in Figure 8. As it can be seen, the $2RISC + 2DSP$ platform is the minimal one to achieve System Failure Probability (SFP) of $1 \cdot 10^{-6}$ and the $3RISC + 3DSP$ platform is necessary to achieve SFP of $1 \cdot 10^{-9}$. An important observation from Figure 8 is that, for the $2RISC + 1DSP$ platform, several solutions with SFP around $1 \cdot 10^{-6}$ are very close to meeting the deadline. The same is observed for the platform $3RISC+2DSP$, where several solutions are close to achieve SFP of $1 \cdot 10^{-9}$. This implies that, if some PEs can be replaced by faster ones, using 3 or 5 PEs might already be sufficient and become more cost-efficient design choices. We therefore tested two additional platforms with *1 RISC + 2 DSP* and *2 RISC + 3 DSP* (the DSP is faster for the mpeg2 application). Figure 9 compares for each platform the maximum reliability achieved under deadline constraints. Clearly, the new platforms with *1 RISC + 2 DSP* and *2 RISC + 3 DSP* are the most cost-efficient solution to achieve SFP of $1 \cdot 10^{-6}$ and $1 \cdot 10^{-9}$, respectively.

For the second experiment we use a set of synthetic task graphs to evaluate the slack sharing schemes. Each of the TG consists of 5 to 15 tasks and we consider random use cases in which 2 to 3 TGs run simultaneously. The platform contains two RISCs and two DSPs and the execution time of tasks are generated randomly between 100 to 1000. We consider three optimization objectives, namely schedule length, reliability and resource utilization. For the objective of schedule length, the penalty value is calculated as:

$$penalty(S) = \begin{cases} -1 & \text{iff } l \leq d \\ l - d & \text{otherwise} \end{cases}$$

where l is the actual schedule length and d is the deadline of the task. The idea is, if the deadline is met, we set the penalty to -1 and if the deadline is violated we set the penalty to the gap between the actual length and the deadline. In this way the optimization will lead to solutions that meet the deadline and optimize other objectives. The same is done for the reliability objective, i.e. the penalty is -1 if the reliability requirements are fulfilled and a positive value otherwise. The resource utilization is the absolute processor time occupation. Using the above setup, all three objectives need to be minimized. Three configurations are compared: NSS, for which no slack sharing is enabled; INTRA, which only uses intra-job slack sharing and INTER, which uses both intra and inter job slack sharing.

Figure 10 shows the two-dimensional projection of the Pareto optimal solutions for one example use case. Similar results are also obtained for other use cases. The horizontal axis is the reliability penalty and the vertical axis is the resource utilization. As it can be seen, significant resource saving comparing with NSS is achieved using the intra-job slack sharing scheme. By enabling the INTER scheme, further saving in resource consumption is observed. In the right part of the curve (reliability penalty larger than 4), the performance of INTRA and INTER is very close to each other. The reason is, for those solutions, the reliability is actually very low which suggests that the available re-execution slots are very limited, resulting in limited opportunity for further improvement using inter-job slack sharing. In contrast, in the left part of the curve, INTER shows notable benefit. In particular, considering the minimum resources that are needed to fulfill the reliability requirement (solution 1 to 3 marked in Figure 10), INTRA and INTER saves 14% and 20% total resource consumption, respectively. Figure 11 compares the solution that has the minimum resource consumption and fulfills all deadline and reliability requirements. The resource consumption is normalized with respect to the NSS approach. On average, $INTRA$ and $INTER$ saves 12% and 20% resources, respectively.

VII. Conclusion

This work considers the reliability-aware Design Space Exploration (DSE) problem for real-time embedded systems. The main contribution is a SFT analysis that provides

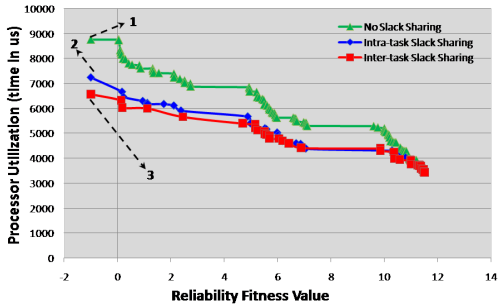

Figure 10. Optimization Result: Reliability vs Utilization

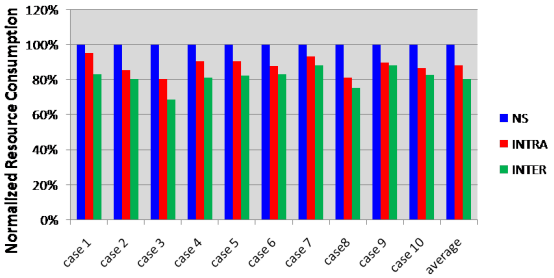

Figure 11. Comparison of Slack Sharing Schemes

probabilistic information about system reliability and an approach that integrates SFT into an evolutionary algorithm based optimization process. We have proposed a two-step approach for efficient encoding of the chromosome and a set of problem-specific operators for manipulation of the chromosome. We have also proposed and evaluated an inter-job slack sharing scheme for further reliability enhancement. Next step will be the integration of DSE procedure into a model-based development framework. The optimization process will then take a set of input models and produce a set of transformed models that fulfill certain reliability requirements. Another direction of future work is to consider other non-functional properties besides reliability.

ACKNOWLEDGMENT

This work has been supported in part by the European research project ACROSS under the Grant Agreement ARTEMIS-2009-1-100208.

REFERENCES

[1] J. Sosnowski, "Transient fault tolerance in digital systems," *IEEE Micro*, February 1994.

[2] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M. Irwin, "Reliability-aware co-synthesis for embedded systems," in *ASAP*, 2004.

[3] P. Pop, V. Izosimov, P. Eles, and Z. Peng, "Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication," *IEEE Transactions on Very Large Scale Integration Systems*, 2009.

[4] Y. Zhang and K. Chakrabarty, "A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2006.

[5] F. Liberato, R. Melhem, and D. Mosse, "Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems," *IEEE Transactions on Computers*, 2000.

[6] C.-C. Han, K. Shin, and J. Wu, "A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults," *IEEE Transactions on Computers*, 2003.

[7] C. Pinello, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, "Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications," in *DATE*, 2004.

[8] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems," in *DATE*, 2005.

[9] V. Izosimov, I. Polian, P. Pop, P. Eles, and Z. Peng, "Analysis and optimization of fault-tolerant embedded systems with hardened processors," in *DATE*, 2009.

[10] Architectural Requirments, "IEC61508-2, chapter 7.4.3.1.1, Tab.2 and 3."

[11] D. Zhu and H. Aydin, "Reliability-aware energy management for periodic real-time tasks," *IEEE Trans. Computers*, 2009.

[12] A. Birolini, "Reliability engineering - theory and practice," *Springer, 4th edition, Berlin, Heidelberg*, 2004.

[13] M. Glaß, M. Lukasiewycz, T. Streichert, C. Haubelt, and J. Teich, "Reliability-Aware System Synthesis," in *DATE*, 2007.

[14] Y. Xiang, T. Chantem, R. P. Dick, X. S. Hu, and L. Shang, "System-level reliability modeling for mpsocs," in *CODES+ISSS*, 2010.

[15] M. Glaß, M. Lukasiewycz, F. Reimann, C. Haubelt, and J. Teich, "Symbolic Reliability Analysis and Optimization of ECU Networks," in *DATE*.

[16] F. Reimann, M. Glaß, M. Lukasiewycz, C. Haubelt, J. Keinert, and J. Teich, "Symbolic Voter Placement for Dependability-Aware System Synthesis," in *CODES+ISSS*, 2008.

[17] P. K. Saraswat, P. Pop, and J. Madsen, "Task mapping and bandwidth reservation for mixed hard/soft fault-tolerant embedded systems," in *RTAS*, 2010.

[18] A. Björner and R. P. Stanley, *A Combinatorial Miscellany*.

[19] M. Lukasiewycz, M. Glaß, C. Haubelt, and J. Teich, "Sat-decoding in evolutionary algorithms for discrete constrained optimization problems," in *CEC*, 2007.

[20] N. Kandasamy, J. Hayes, and B. Murray, "Transparent recovery from intermittent faults in time-triggered distributed systems," *IEEE Transactions on Computers*, 2003.

[21] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping applications to tiled multiprocessor embedded systems," in *ACSD*, 2007.