Analysis and Optimization of Fault-Tolerant Task Scheduling on Multiprocessor Embedded Systems

Jia Huang
fortiss GmbH, Germany
huang@fortiss.org

Jan Olaf Blech
fortiss GmbH, Germany
blech@fortiss.org

Andreas Raabe
fortiss GmbH, Germany
raabe@fortiss.org

Christian Buckl
fortiss GmbH, Germany
buckl@fortiss.org

Alois Knoll
TU München, Germany
knoll@in.tum.de

ABSTRACT

Reliability is a major requirement for most safety-related systems. To meet this requirement, fault-tolerant techniques such as hardware replication and software re-execution are often utilized. In this paper, we tackle the problem of analysis and optimization of fault-tolerant task scheduling for multiprocessor embedded systems. A set of existing fault- and process-models are adopted and a Binary Tree Analysis (BTA) is proposed to compute the system-level reliability in the presence of software/hardware redundancy. The BTA is integrated into a multi-objective evolutionary algorithm via a two-step encoding to perform reliability-aware design optimization. The optimization results contain the mapping of tasks to processing elements, the exact task and message schedule and the fault-tolerance policy assignment. Based on the observation that permanent faults need to be considered together with transient faults to achieve optimal system design, we propose a virtual mapping technique to take both types of faults into account. To the best of our knowledge, this is the first approach in fault-tolerant task scheduling that considers permanent and transient faults in a unified manner. The effectiveness of our approach is illustrated using several case studies.

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; C.3 [**special-purpose and ap plication-based systems**]: Real-time and embedded systems

General Terms

Algorithms, Design, Reliability

Keywords

Embedded Systems, Reliability, Design Optimization

1. INTRODUCTION

With the continuous shrinking of transistor sizes, modern devices are becoming more susceptible to faults [21, 9]. Such faults can be roughly categorized as permanent or transient ones. Permanent faults, as the name suggests, are non-recoverable device defects. They occur relatively rarely but have a large impact on the running system. Transient faults appear for a short time and disappear without damage to the device, e.g., single event upset caused by electromagnetic interference. The occurrence probability of transient faults is usually higher than that of permanent faults. In many safety-related embedded systems, the capability to provide reliable execution even in the presence of both permanent and transient faults is a major requirement.

One traditional way to enhance system reliability is to use *spatial redundancy* (also known as hardware redundancy). For example, in a triple-modular-redundant system, the critical components are replicated three times and the results are voted to produce the output. Hardware replication tolerates both permanent and transient faults and has the advantage of simplified fault detection. However, it comes with high design and production cost. An alternative to handle permanent faults is task migration, i.e. the re-mapping of tasks running on a faulty processor to other non-faulty ones as soon as a defect is detected. Naturally, task migration is only possible if adequate hardware resources are still available. Re-mapping schemes are usually designed carefully to guarantee the feasibility and minimize the migration cost [13, 25].

Temporal redundancy is more cost-efficient to handle transient faults. One possible approach is to schedule critical tasks multiple times and perform voting of the results [24]. Another common technique is to insert checkpoints into the software and rollback the execution from a safe state in case faults are detected [18, 26]. For real-time applications, temporal redundancy must be used with utmost care, since the overhead in time may lead to deadline violations. The schedulability issue in the context of temporal redundancy has therefore become a very important topic [14, 7, 17].

Safety-related systems must tolerate both permanent and transient faults. In most existing work, they are considered separately using dedicated techniques. However, it is particularly important to consider both types of faults in an **unified** manner, in order to achieve the most efficient and reliable design. We explain this point using the following example. Consider the system in Figure 1 consisting of two

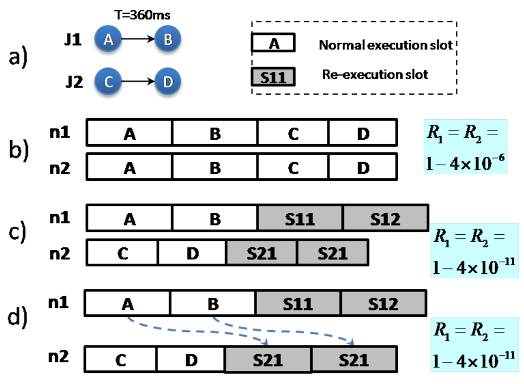


Figure 1: Motivating Example

jobs J_1 and J_2 to be executed on two processors n_1 and n_2. It has the requirement to tolerate a single defect on any of the two processors. A straitforward design considering only the permanent faults could be to use hardware redundancy as shown in Figure 1b. However, such a setup has very limited capability of tolerating transient faults. Assume transient fault probability is 1×10^{-5} for one iteration of each task and the period of both jobs are $360ms$, the failure rate per hour of both J_1 and J_2 can be easily computed as 4×10^{-6}. Using the same amount of resources, the software re-execution technique can achieve much better tolerance to transient faults. In Figure 1c, two re-execution slots (or slack slots) are scheduled on each processor, which can be used to re-execute any previous task that is misbehaving due to transient faults. Using the analysis presented later in section 4, the failure probability of both applications is computed to 4×10^{-11}. However, the schedule in Figure 1c is not capable of tolerating permanent faults on n_1, since the slack slots S_{21} and S_{22} are not large enough to accommodate A and B. Actually, when we schedule the re-execution slots to tolerate transient faults, we can keep the requirement from permanent faults in mind and intentionally increase the sizes of slot S_{21} and S_{22} to fit task A and B (Figure 1d). In this way, permanent defects can also be handled since migration of task A and B are now possible. The schedule 1d therefore has the same tolerance to permanent faults as schedule 1b and achieves much higher tolerance to transient faults.

The focus of this paper is on analysis and optimization of real-time systems that tolerate both permanent and transient faults. We consider the combined utilization of hardware replication and software re-execution techniques and advocate the consideration of both types of faults in a unified manner. The main contributions of this paper are: 1) A binary tree based approach that enables probabilistic analysis of system reliability in the presence of spatial and temporal redundancy; 2) an approach that integrates the analysis with a Multi-Objective Evolutionary Algorithm (MOEA) via an efficient two-step encoding; 3) a virtual mapping technique to consider permanent faults together with transient faults. To the best of our knowledge, this is the first approach that considers permanent and transient faults in a unified manner in fault-tolerant task scheduling.

The remainder of the paper is organized as follows. An overview of existing work is presented in Section 2. The system models used in this paper are introduced in Section 3. Section 4 to 6 describe the proposed reliability analysis and design optimization approach. Section 7 presents the virtual mapping technique. Experimental results are presented in Section 8. Section 9 concludes this paper.

2. RELATED WORK

Reliability-aware design consists of two major tasks: modeling/analyzing of reliability and integration of the approach into the design process. An overview can be found in [3].

Reliability analysis is typically performed in a hierarchical manner, from individual component models up to the system-level model. Recent work [23] proposes a framework that integrates device, component and system level models. *Glaß et al* present in [6] a symbolic approach for reliability analysis focusing on permanent faults. The system behavior under influence of faults is described using the so-called structure function. Then, the system-level reliability can be evaluated based on component level reliability models. The analysis is integrated into a MOEA based optimization framework in [5]. Spatial redundancy is considered and represented by binding the same task to multiple Processing Elements (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 [19]. Compared to the work mentioned above, our reliability analysis considers transient faults and also handles software fault-tolerance techniques.

Several existing approaches focus on transient faults. In [24], the authors present an approach to handle transient faults by selectively inserting task re-executions. *Izosimov et al* [11] study the design optimization of fault-tolerance systems using both spatial and temporal redundancy. The case for combined utilization of check pointing and hardware replication is considered by *Pop et al* [18]. In [20], the authors propose a hybrid scheduling approach for mixed hard and soft real-time tasks. 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. The fault model in [11, 18, 20] assumes a total number of faults that may occur in any component of the system. The distinct failure probabilities of different hardware components are not taken into account. The work [10] describes a more accurate probabilistic analysis of system reliability based on the amount of temporal redundancy and component failure rate. It is also integrated into a tabu-search based optimization procedure. Other work also studies the tradeoff between reliability and other design objectives, such as energy [28] and cost [18]. Our paper tackles a similar problem as the work mentioned above [11, 18, 20, 28] and adopts a similar fault- and process-model. We extend these approaches and propose a probabilistic analysis that computes the system reliability in presence of both spatial and temporal redundancy. Additionally, we introduce an approach based on evolutionary algorithms that allows us to consider multiple optimization objectives, e.g. reliability, schedule length and resource utilization. Another major advantage of our approach is that permanent faults can be taken into account efficiently using the proposed virtual mapping technique.

3. SYSTEM MODEL

We consider an application as the functionality of the system as a whole. The application $\mathcal{A}$ consists of a set of independent *jobs*, each given as a directed acyclic graph. For a

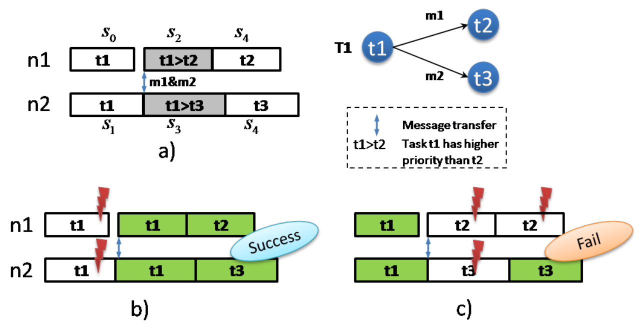


Figure 2: Example Fault Scenario

job $\mathcal{J} = (\mathcal{T}, \mathcal{E})$, the vertices $\mathcal{T} = \{t_0, t_1, ..., t_m\}$ represent a set of *tasks* to be executed and the edges $\mathcal{E} = \{e_0, e_1, ..., e_l\}$ capture data dependencies between tasks. We assume that the set of jobs in $\mathcal{A}$ share the same period. If jobs originally have different periods, they are first transformed into larger graphs representing a hyper-period (Least Common Multiple of all periods) of the application.

Timing predictability is highly desirable for safety-related applications. In this paper, we target on synthesizing *static time-triggered schedules*. Such a schedule $\mathcal{S}$ is composed of a set of non-overlapping slots $\{s_0, s_1, ..., s_n\}$, each of which is a four-tuple $s = (b, f, p, T)$, where b is the start time of the slot, f is the finish time, p is the processor on which the slot is allocated and T is a set of tasks assigned to s. 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. The length of each slot must be no smaller than the Worst Case Execution Time (WCET) of any task possibly assigned to it. The time needed for recovery from faults is also considered in slack slots. The actual utilization of scheduling slots depends on how the scheduler responds to the occurred faults. We adopt a *static priority* based approach within a slot, i.e. the task $t \in s.T$ [1] that has the highest priority among the pending tasks acquires the slot. A similar scheduling setup is also used in the ARINC 653 standard. In our case, a task will be skipped in all subsequent slots once it is executed successfully and the results are made available. Figure 2a depicts an example schedule consisting of 6 slots. Figure 2b and 2c demonstrate two example execution sequences in the presence of faults. In Figure 2b the slot s_2 is used for re-execution of t_1 since both s_0 and s_1 fail. In Figure 2c s_2 executes t_2 since s_0 succeeded with t_1. We use the same assumption as in [11, 10, 27, 20, 28] that transient faults are detected using sanity checks when a task is completed. The timing overhead of fault detection is assumed to be contained in the WCETs of tasks. The effect of simplified fault detection using spatial redundancy and voting is currently ignored, but will be considered in further work.

In multiprocessor systems, if two communicating tasks are mapped to different processors, a message must be scheduled for data transfer, e.g. message m_1 and m_2 in Figure 2a. The latency of message transfer is also considered but not depicted in the figure for clarity. In this work, we consider systems with reliable time-triggered on-chip communication, e.g., the GENESYS architecture [4]. The message schedule $\mathcal{M}$ is described as a set of message slots $\{m_0, m_1, ..., m_k\}$. Each message slot is a four-tuple $m = (b, f, t_{src}, t_{tgt})$, where b is the start of the message, f is the finish time, t_{src} is the source task of the message and t_{tgt} is the sink.

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

4. RELIABILITY ANALYSIS

To analyze the system reliability, we need to investigate how the system behaves in case of faults. We describe the faults occurring in a system by a *fault scenario*:

Definition 1 (Fault Scenario). *A fault scenario is a vector $\boldsymbol{x} = \{x_0, x_1, ..., x_n\}$, which contains for each scheduling slot s_i a variable $x_i \in \{1, 0, NA\}$. It encodes the execution result of s_i: x_i is 1 if the slot executes some task successfully and 0 if the execution fails. x_i is NA if the slot s_i is not used, i.e. each task in $s_i.T$ is either not ready or finished earlier and no task is currently executed in s_i.*

For the given job J, a fault scenario $\mathbf{x}$ is *tolerable* by a schedule $\mathcal{S}$ if J is still executed correctly in presence of faults specified in $\mathbf{x}$. The entire set of fault scenarios that are tolerable by schedule $\mathcal{S}$ is called the *working set* of J, denoted as $W(\mathcal{S}, J)$. The overall probability that J is correct can be obtained by summarizing the occurrence probability of all fault scenarios in the working set:

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

Before presenting the calculation of the working set, we first introduce some intermediate notations. Let $S(t_j)$ represent the set of slots to which task t_j is assigned, i.e. $S(t_j) = \{s \in \mathcal{S} | \wedge t_j \in s.T\}$. The boolean *request* variable $r_{i,j}$ evaluates to *true* if the task t_j *requests* to execute in slot s_i and *false* otherwise. The boolean *utilization* variable $u_{i,j}$ is *true* if the slot s_i is actually used to execute task t_j and *false* otherwise. For the case of static priority scheduling, $u_{i,j}$ computes to:

$$u_{i,j} = r_{i,j} \wedge (\bigwedge_{\substack{t_l \in s_i.T \wedge \\ priority(t_l) > priority(t_j)}} \neg r_{i,l}) \tag{2}$$

that is, s_i is utilized by task t_j only if t_j has the highest priority among all tasks requesting the slot. An execution request is sent only if the following conditions are fulfilled:

$$r_{i,j} = \; isReady \; \wedge notPrev \; \wedge \; notOther \tag{3}$$

The first term *isReady* requires the task t_j to be ready, i.e. all predecessor tasks have been finished successfully. The following terms check the necessity of executing t_j. The term *notPrev* is computed as:

$$notPrev = \bigwedge_{\substack{s_k \in S(t_j) \wedge s_k.p = s_i.p \\ \wedge s_k.f \leq s_i.b}} \neg(u_{k,j} \wedge x_k = 1) \tag{4}$$

It is true if t_j has not been finished in previous slots on the same processor. The term *notOther* checks if the task has been executed successfully on other processors and a message is sent properly:

$$notOther = \bigwedge_{\substack{s_k \in S(t_j) \wedge s_k.p \neq s_i.p \\ \wedge s_k.f \leq s_i.b}} \neg((u_{k,j} \wedge x_k = 1) \wedge$$
$$(\exists m \in \mathcal{M} : m.t_{src} = t_j \wedge m.f \leq s_i.b \wedge m.b \geq s_k.f))$$

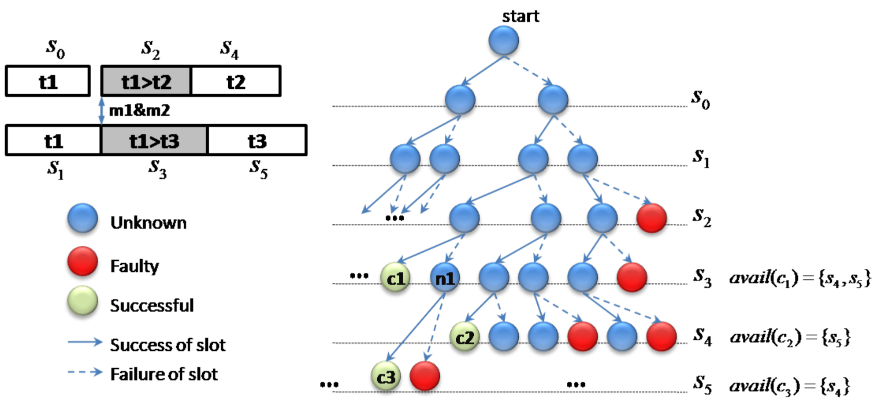


Figure 3: An Example of Binary Tree Analysis

The values of variables $r_{i,j}$ and $u_{i,j}$ can be calculated in an iterative manner. Starting from the earliest scheduling slot, we iteratively consider each $s \in \mathcal{S}$. For a specific slot, we compute the variables from the task with highest priority to the task with lowest priority.

A task is successful if at least one instance of it is executed without faults. Hence we have:

$$success(t_j, \mathbf{x}) = \bigvee_{s_k \in S(t_j)} (u_{k,j} \wedge x_k = 1)$$

For a given schedule, we can construct a function $\varphi_J : \{0,1\}^{|\mathbf{x}|} \rightarrow \{0,1\}$, which takes a fault scenario $\mathbf{x}$ and returns 1 if the job J is still correct under impact of $\mathbf{x}$ and 0 otherwise. Since the entire job is correct only if all of its tasks are correct, the function is given as:

$$\varphi_J(\mathbf{x}) = \bigwedge_{t_j \in \mathcal{T}} success(t_j, \mathbf{x}) \tag{5}$$

With the help of function φ, the working set $W(\mathcal{S}, J) = \{\mathbf{x} | \varphi_J(\mathbf{x}) = 1\}$ can be obtained by a Binary Tree Analysis (BTA). The procedure is demonstrated using an example shown in Figure 3. We consider the scheduling slots according to the order of occurrence, i.e. the slots with earlier starting time are selected first (e.g., from S_0 to S_5 in Figure 3). Slots with equal start time can be considered in arbitrary order. The ith level in the tree is associated with the ith slot and the edges leaving a node in the ith level represent the execution result of that slot. Left branches (solid lines in Figure 3) represent the case that the slot executes some task correctly. Right branches (dashed lines in Figure 3) represent a slot with failed execution. Note that a slot might be unused when all tasks in $s.T$ are either not ready or finished earlier. In this case we skip this level and spawn children in the next level, e.g. node n_1 in Figure 3. By constructing the tree in this way, each node will have a unique path to the start node representing a unique fault scenario. A node at depth m represents a fault scenario in which the first m variables are determined and the rest are considered to be NA. The total depth D of the tree equals the number of scheduling slots: $D = |\mathbf{x}| = |\mathcal{S}|$.

Each node in the tree is associated with its own request/utilization variables. For a specific node n, we compute those variables using (2) to (3) based on the values of request/utilization variables associated with the nodes on the path from n to the start node. This procedure actually computes which task is going to be executed in a slot based on the execution results of previous slots.

Algorithm 1 analysis(n): binary tree analysis with starting node n.

```
computeRUVariables(n);
l ← createLeftBranch(n)
if checkLeftBranch()=successful then
    addToWorkingSet(l)
else
    analysis(l)
end if
r ← createRightBranch(n)
if checkRightBranch()≠ faulty then
    analysis(r)
end if
```

Algorithm 2 BinaryTreeAnalysis(S): top-level routine of BTA for schedule S

```
setScheduleToBeAnalyzed(S);
n0 ← createStartNode();
analysis(n0);
```

With the request/utilization variables, a fault scenario $\mathbf{x}$ can be evaluated using (5), and the corresponding node is assigned to one of the states: *unknown*, *faulty* or *successful*. A node is *faulty* iff, given the current faults specified in $\mathbf{x}$, there exists no possibility to execute the job successfully in the remaining slots. A node is *successful* iff the entire job is already finished using the successful slots specified in $\mathbf{x}$, i.e. the remaining slots are not needed. The *faulty* and *successful* nodes will not spawn further branches. If a node is neither identified as *faulty* nor *successful*, the analysis continues with its children. The tree analysis is complete if all nodes at the maximum depth D was visited or no more *unknown* node exists. Afterwards, the set of *successful* nodes is used as the working set. The analysis process above can be implemented recursively as outlined in Algorithms 1 and 2.

The occurrence probability of a *successful* node $\mathbf{x}$ can be computed as:

$$Pr(\mathbf{x}) = \prod_{x_i \in \mathbf{x} \wedge x_i = 1} Pr(s_i) \cdot \prod_{x_i \in \mathbf{x} \wedge x_i = 0} (1 - Pr(s_i)) \tag{6}$$

where $Pr(s_i)$ is the success probability of the task executed in slot s_i. The probability $Pr(s_i)$ can be computed based on the reliability function of the hardware platform and we assume that it is given a priori [27]. The system reliability can then be obtained using (1).

4.1 Complexity Issues

The complexity of processing a node during BTA is linear with respect to the number of tasks assigned to the corresponding slot (variables r and u need to be computed for each task). However, this number is typically very small and does not grow significantly when the system becomes more complex. We therefore assume the complexity of visiting a node to be constant. In this case, the complexity of the entire analysis is determined by the number of nodes visited. The worst case scenario occurs when all the nodes in depth smaller than $|\mathcal{S}|$ are in the *unknown* state. The complexity is in $\mathcal{O}(2^{|\mathcal{S}|+1})$ in this case.

As the analysis has a worst case exponential complexity, it is important to find approximations that improve the scalability. An observation from equation (6) is that the fault scenarios that specify more faulty slots have much lower occurrence probability, because the failure rate of a task is typically very low. Moreover, a fault scenario that specifies more faults is more likely to be a *faulty* node. Hence, an approximation of the system reliability would be to visit only nodes with at most d faulty slots and to assume all nodes specifying more than d faults as non-tolerable. Since the reliability is obtained using (1), ignoring possibly tolerable nodes is a safe underestimation of system reliability. From the tree point of view, this corresponds to eliminating all nodes with more than d right branches on their paths to the start node.

With the above estimation, the total amount of visited nodes can be computed as follows. We divide the tree into two parts. For the first d levels of the tree, all nodes should be visited, i.e. in total $2^{d+1} - 1$ nodes. For the rest, recall that the set of nodes in level l is a complete enumeration of all possible assignments of the first l variables in a fault scenario. Hence, the number of assignments with maximum d zeros is $\sum_{x=0}^{d} \binom{l}{x}$. The total amount of nodes is then:

$$T(|\mathcal{S}|) = 2^{d+1} - 1 + \sum_{l=d}^{|\mathcal{S}|} \sum_{x=0}^{d} \binom{l}{x} \qquad (7)$$

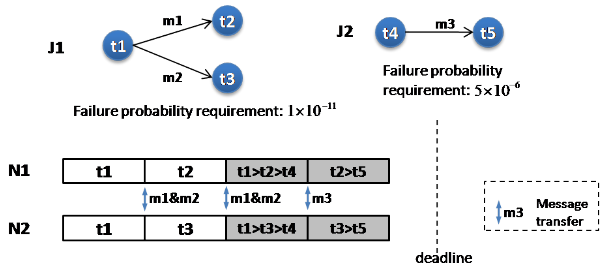
By applying a simple upper bound for the sum of binomial coefficients $\sum_{x=0}^{d} \binom{l}{x} \leq (l+1)^d$, the complexity of the algorithm computes to:

$$\mathcal{O}(T(|\mathcal{S}|)) = \mathcal{O}(\sum_{l=d}^{|\mathcal{S}|} \sum_{x=0}^{d} \binom{l}{x}) \subseteq \mathcal{O}(\sum_{l=d}^{|\mathcal{S}|} (l+1)^d) \qquad (8)$$

The expression above can be further overestimated as:

$$\mathcal{O}(T(|\mathcal{S}|)) \subseteq \mathcal{O}(|\mathcal{S}| \cdot (|\mathcal{S}|+1)^d) = \mathcal{O}(|\mathcal{S}|^{d+1}) \qquad (9)$$

As it can be seen, the complexity of BTA is reduced to be polynomial in $|\mathcal{S}|$ by bounding the maximum number of faults by a constant d. The above analysis shows the worst-case complexity of BTA. During our experiments, we observe that the portion of terminating nodes (mostly *faulty* nodes) increases significantly with higher d and the actual number of visited nodes is much smaller. As an example of runtime, the average execution time of BTA on the *mpeg2* application ($|\mathcal{S}| \approx 35$, measured on a 3GHz CPU) is $754ms$ for $d = 3$ and $3405ms$ for $d = 5$. Thus the runtime of BTA is acceptable for an offline optimization process. To obtain high-quality schedules, we should focus on increasing the coverage of fault scenarios with high probabilities instead of tolerating rare cases. Hence, the BTA is in most cases used with small d, e.g., $d = 3$.



J1
t1
m1
t2
m2
t3
Failure probability requirement: 1×10^{-11}
J2
t4
m3
t5
Failure probability requirement: 5×10^{-6}
N1
t1
t2
t1>t2>t4
t2>t5
m1&m2
m1&m2
m3
N2
t1
t3
t1>t3>t4
t3>t5
deadline
m3
Message transfer

Figure 4: Example of Static Priority Slack Sharing

5. STATIC PRIORITY SLACK SHARING FOR MULTIPLE JOBS

Many multiprocessor systems are designed for co-hosting multiple functionalities concurrently. In particular, there is an increasing trend towards implementing jobs with mixed criticality on a single shared computing platform [2]. It is likely that jobs with different criticality have highly distinct reliability requirements. For highly critical tasks, a significant amount of temporal redundancy is needed to meet their reliability requirements. However, the probability that the software slack is actually used is typically very low. In this case, implementing each job in a step-wise manner without a global view may result in sub-optimal system design (see example below). To cope with this problem, we propose a Static Priority Slack Sharing (SPSS) scheme. The idea is to introduce global re-execution slots and enable sharing of those slots among multiple jobs using a *job-level* static priority approach based on the criticality.

Figure 4 shows an example schedule of two jobs using the SPSS technique. A high-criticality job J_1 and a low-criticality job J_2 are allocated on two processors. Four global slack slots are scheduled, in which J_1 is assigned a higher priority. In this case, scheduling of J_2 in slack slots will have no influence on the execution of J_1. Assume the failure rate of each task is 10^{-5} and the period is $360ms$, we find that the reliability requirement of J_1 is met. For the low priority task J_2, a re-execution slot is granted only if J_1 finishes successfully without using that slot. Due to the fact that the task failure rate is low, such a setup already fulfills the requirement of J_2. Thus, two processors are sufficient to execute both jobs. Without using SPSS, a third processor would be necessary for J_2, since the remaining resources on N_1 and N_2 are not enough.

The BTA is able to analyze a global schedule of multiple jobs and compute the reliability of each job. For this, it has to be modified to consider a node as *unknown* unless the results of all jobs are available (either *successful* or *faulty*). As discussed in section 4.1, the complexity grows rapidly with the total number of scheduling slots. To cope with this problem, we present an extended approach that computes the reliability of each job iteratively, a sketch of which is shown in Algorithm 3.

In the extended algorithm, we iterate over each job from the one with highest priority to the one with lowest priority. For a specific job J, we perform the BTA and obtain the set of *successful* nodes (working set). For each node $\widetilde{n} \in W(\mathcal{S}, J)$, there is an *availability scenario* $\widetilde{g}$ associated. It denotes which shared re-execution slots are used and which are not. Let S_G denote the set of shared re-execution slots in schedule $\mathcal{S}$, $\widetilde{g}$ is a subset of S_G that computes to:

$$\widetilde{g} = \{s_i \in S_G | \widetilde{\mathbf{x}}.x_i = NA\} \tag{10}$$

Where $\widetilde{\mathbf{x}}.x_i$ refers to the value of variable x_i in fault scenario $\widetilde{\mathbf{x}}$ associated with node $\widetilde{n}$. An example is given in Figure 3. The availability is $\{S_4, S_5\}$ for the *successful* node c_1, $\{S_5\}$ for node c_2 and $\{S_4\}$ for node c_3. Note that multiple *successful* nodes may result in the same availability scenario. Hence, the occurrence probability of a specific availability scenario g is:

$$Pr(g) = \sum_{\widetilde{n} \in W(\mathcal{S},J) \wedge \widetilde{g} = g} Pr(\widetilde{\mathbf{x}}) \tag{11}$$

For the analysis of next job J', we iterate over each availability scenario (line 4 in Algorithm 3). For a specific availability scenario g, the remaining slack slots are combined with the slots dedicated for J' to obtain the total schedule $\hat{S}$ (line 5 in Algorithm 3). The $\hat{S}$ is then used for the BTA of J' (line 6). In a certain availability scenario g, the occurrence probability of a fault scenario is

$$Pr(\mathbf{x}, g) = Pr(g)Pr(\mathbf{x}|\hat{S}) \tag{12}$$

Where $Pr(\mathbf{x}|\hat{S})$ is the occurrence probability of $\mathbf{x}$ using the schedule $\hat{S}$ associated with g. The probabilities of tolerable fault scenarios found with each availability scenario are summarized using equation (1) to obtain the system reliability. The BTA of J' computes again the availability scenario for further jobs (line 7 and 9 in Algorithm 3).

Algorithm 3 IterativeTreeAnalysis(): iterative tree analysis for multiple tasks. AS_{old}: the set of availability scenarios from previous job. AS_{new}: the set of availability scenarios for next job. $S(J)$: the set of slots dedicated for job J. The function *combine* computes the overall occurrence probabilities of availability scenarios using (11).

```
1:  $AS_{old} \leftarrow$ initAvailability();
2:  $AS_{new} \leftarrow$ initAvailability();
3:  for all $J \in \mathcal{A}$ with decreasing priority do
4:      for all $a \in AS_{old}$ do
5:          $\hat{S} = S(J) \cup a$
6:          avail $\leftarrow$ BinaryTreeAnalysis($\hat{S}$)
7:          combine($AS_{new}$,avail)
8:      end for
9:      $AS_{old} \leftarrow AS_{new}$
10: end for
```

Complexity. Let $|\mathcal{A}|$ be the number of jobs and $S(J)$ be the set of scheduling slots dedicated to job J, the total number of slots of schedule $\mathcal{S}$ can be represented as $|\mathcal{S}| = \sum_{J \in \mathcal{A}} |S(J)| + |S_G|$. Consider the case that we assume maximum d faults in each job, the maximum number of faults in the entire system is $|\mathcal{A}|\, d$ and complexity of the analysis is in $\mathcal{O}(|\mathcal{S}|^{|\mathcal{A}|d+1})$ according to equation 9. Using the iterative approach, the worst-case complexity of BTA for a single job J is in $\mathcal{O}((|S(J)| + |S_G|)^{d+1})$. The BTA needs to be done for each availability scenario. Assume J is the xth job that we consider, then the previous jobs may encounter up to $(x-1)d$ faults. Those faults may consume shared slack slots and thus result in different availability scenarios. In the worst case, each combination of faults has a different availability scenario, so the total number of BTAs to be performed is $\sum_{i=0}^{(x-1)d} \binom{|S_G|}{i} \leq (|S_G| + 1)^{(x-1)d}$. It can be easily seen that the complexity is significantly reduced. For further complexity reduction, we can apply the same idea as in section 4.1 on availability scenarios by considering the availability scenarios with occurrence probabilities lower than a threshold value as faulty. This is obviously also a safe underestimation of reliability.

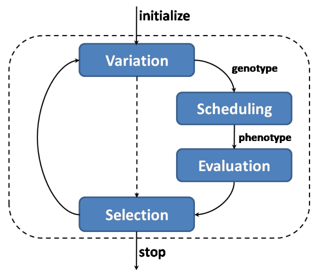


Figure 5: Workflow of EA-based Optimization

6. OPTIMIZATION PROCEDURE

After reliability analysis, the next step is to find the optimal task schedule. Our approach is based on the Multi-Objective Evolutionary Algorithm (MOEA). The EA is performed in two main steps: production of new solutions by varying existing solutions and selection of good solutions based on their fitness (Figure 5). In order utilized MOEA, the schedules need to be encoded as chromosome. However, a direct encoding of a schedule described in Section 3 needs a very large chromosome, which results in a huge search space and low optimization efficiency. To cope with this problem, we utilize a two-step encoding process inspired from [15]. The main idea is, instead of encoding the entire schedule, we only put partial information, namely the mapping and fault-tolerance policy, into the chromosome. A scheduler is integrated to transform the chromosome to an optimized schedule. The resulting schedule is then used for fitness evaluation, e.g. the reliability analysis.

Using this approach, the chromosome contains one gene per task. Each gene is a pair $g = (i, L)$, where i is the integer index of the task and L is a list of integer values denoting the PEs task i is mapped to. An example is shown in Figure 6. 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 1 in Figure 6). The operators that we apply on the chromosome are described in details in [8].

Reconstruction of the schedule from the chromosome is the same as scheduling the task executions with known mapping and fault-tolerance policy. In principle, any existing scheduler for this purpose can be used. The scheduling procedure that we propose consists of three main steps. First, for each mapping entry of a task t, we instantiate a schedul-

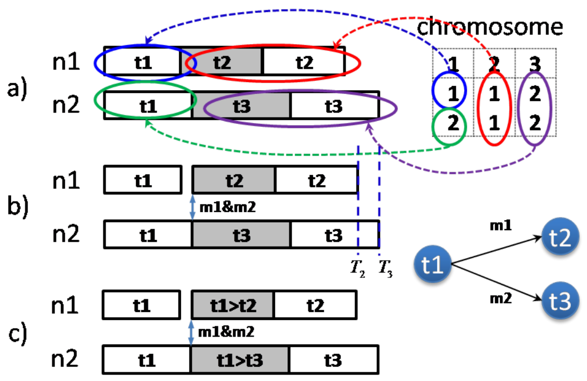


Figure 6: Encoding and Reconstruction of Schedule

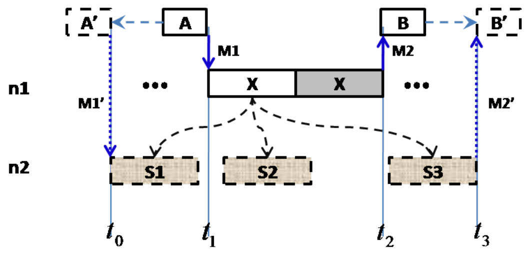


Figure 7: Influence of Data Dependency on Task Migration

ing slot with length equal to the execution time of t. The set of slots is scheduled using a list scheduler. The priority is computed based on two criteria: 1) a task belonging to a job with earlier deadline has higher priority (job-level EDF); 2) for tasks in the same job, the one that has a longer critical path to the sink is assigned a higher priority. Using such an approach, data dependencies are automatically regarded. Second, bus scheduling is performed for each message (Figure 6b). In this paper, we adopt the *transparent recovery* approach [12], which requires that a fault occurring on one PE is masked to other PEs. This has several advantages such as fault-containment and improved traceability. According to transparent recovery, the message should be scheduled after possible re-executions so that faults occurring at the sending side are not visible to the receiver, e.g., if the task t_2 in Figure 6 wants to sent a message to other tasks, the message should be placed at time T_2. Tasks may be postponed due to dependency on messages. In a third step, we perform slack sharing (Figure 6c) using a greedy approach. A slot is shared with all tasks that 1) may become ready before the start time of this slot; 2) has a execution time not greater than the slot size.

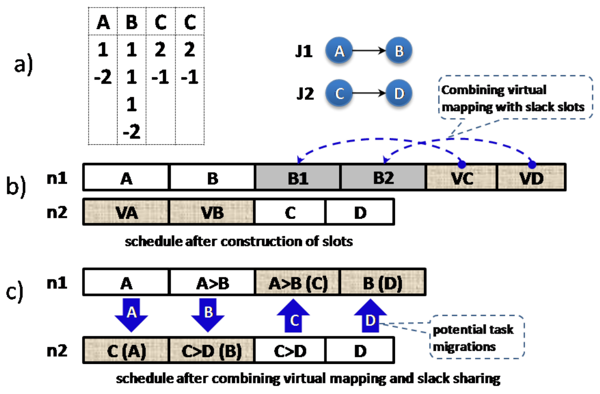
An advantage of the two-step encoding is that additional application constraints can be realized as constraints on the chromosome. E.g, the safety standard IEC 61508 requires each device certifiable to Safety Integrity Level (SIL) 4 to implement at least hardware fault tolerance of one [1]. This can be achieved by adding a constraint on the chromosome enforcing that the mapping list of each task must have at least two unequal entries, meaning that each task has at least one spatial replica. Another example would be separation constraints, e.g. two critical tasks are required to be strictly isolated in space. This in turn requires that the mapping entries of the two tasks do not collide.

7. TOLERATING PERMANENT FAULTS USING VIRTUAL MAPPING

Many safety-related applications require to tolerate certain permanent faults. In section 1 we made the case for a unified consideration of permanent and transient faults. Nevertheless, the analysis and optimization approach presented so far focuses only on transient faults. In this section, we present an extension integrating the consideration of permanent faults. Recall that, to tolerate a permanent defect of some processor p, we need to guarantee that each task mapped to p either has another running instance (spatial replication) or can be migrated to a slack slot on another processor. Thus, a straitforward way is to ensure that each task has at least one replica by adding constraints on the chromosomes. However, such an approach has the drawback that spatial redundancy is much less efficient in terms of contribution to transient fault tolerance and also comes with high hardware cost.

A more cost-efficient alternative to handle permanent faults is task migration. To design such a system, one of the most important goals is to minimize the overhead of migration. The ideal case is that the system recovers from faults with only minor re-configuration. Since attaining the optimal task migration decision is a highly complex task, recent work [13] proposes to compute the task re-mappings statically offline and store them in tables. The pre-computed configurations are then applied at runtime if a processor fails. We adopt a similar approach and target at synthesizing a static schedule that can be adapted with minor changes to handle failure of processors. In case of static time-triggered scheduling, we observe that the migration cost is highly influenced by the data dependencies. Consider the example depicted in Figure 7, where we are going to migrate the task X to one of the possible locations S_1 to S_3. The task A and B are communicating with X via messages. If X is re-mapped to S_1, which is earlier than the original message M_1, the predecessor task A and the message M_1 need to be shifted forward due to data dependency. In consequence, other tasks communicating with A need further adaptation and overall migration cost could be high. A similar situation occurs if X is migrated to S_3, which is later than the original message M_2. In this case the successor tasks need to be shifted backwards. Instead, if X is moved to S_2, the rest of the schedule does not have to change. We learn from this example that, while building a schedule, it is very important to keep track of the migration locations and try to put them into the optimal locations. For the given example, we should try to schedule a slack slot between t_1 and t_2.

To solve this problem, we propose a virtual mapping technique. The idea of virtual mapping is to trace potential places for task migrations already at the time when the schedule is constructed from the chromosome. A virtual mapping of task t to p is represented in our encoding scheme using a negative integer $-p$, which implies that p is the target of migration of task t. For example, the chromosome shown in Figure 8a specifies two entries 1 and -2 for task A, which means A is executed on processor n_1 during normal execution and it should be migrated to n_2 if n_1 fails. When constructing the schedule, we instantiate for a virtual mapping also a slot of the size equal to the execution time of A (slot VA in Figure 8b). This slot is the place where A will be migrated to. Note that the virtual mapping slots are also



a)
A B C C
1 1 2 2
-2 1 -1 -1
1
-2
J1 A B
J2 C D
Combining virtual mapping with slack slots
b)
n1 A B B1 B2 VC VD
n2 VA VB C D
schedule after construction of slots
c)
n1 A A>B A>B (C) B (D)
n2 C (A) C>D (B) C>D D
potential task migrations
schedule after combining virtual mapping and slack sharing

Figure 8: Example of Virtual Mapping

scheduled using the same heuristic presented in Section 6 so that data dependencies are also regarded. This is essential to achieve a low-overhead task migration as shown in Figure 7. Nevertheless, during normal execution, this slot is not left empty but used as a slack slot for other tasks mapped onto the same processor. For example, in Figure 8c, the slot VA is actually used for task C. This technique reclaims the time reserved for task migration and uses it to improve the transient fault tolerance in normal execution. The efficiency of resource utilization is therefore improved. Note that virtual mapping slots may be combined with other slack slots scheduled on the same processor to reduce the length of the schedule. For example the slot VC is combined with B_1 and slot VD is combined with B_2. The combination is only possible if two rules are obeyed: 1) the normal slack slot is no smaller than the virtual mapping slot; 2) no data dependency is violated. These two rules guarantee that the task migration is still valid after combination. Afterwards, the corresponding slack slots are marked as new migration targets (Figure 8c).

There are two main advantages of using virtual mapping. The first is easy implementation, since the optimization process remains unchanged and no further objective is necessary. Tolerance of permanent faults is achieved by adding simple constraints to the chromosome. For example, if it is required to tolerate a defect of processor p, we just need to add the constraint saying that tasks that are mapped only to p must have a virtual mapping. The second advantage is low migration effort. Using the proposed approach, the places for task migrations to handle certain hardware defects are already found and scheduled statically. To carry out the task migration, the scheduling slots do not need to change. Only a simple update of the priority table of virtual mapping slots needs to be done, e.g., during normal execution, task A can already be mapped to VA and be set to lowest priority. When migration is needed, we just set A to the highest priority in the slot. Since the binary of task A is already loaded to the target of migration, timely recovery is expected.

8. EXPERIMENTAL RESULTS

We implemented the analysis and optimization algorithm in JAVA with the help of the opt4j library [16]. The MOEA has 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

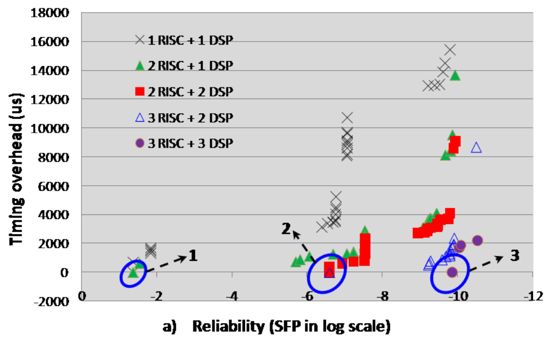


Figure 9: Pareto Optimal Solutions under Different Platform Configurations

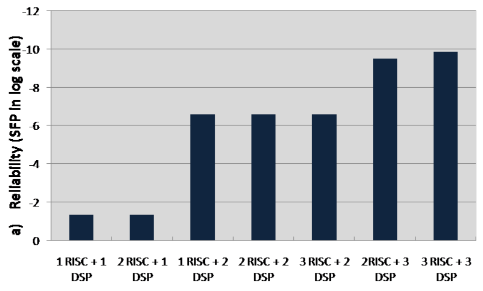


Figure 10: Achievable Reliability Comparison

failure probability of each task on a certain PE is randomly generated between 1×10^{-5} and 1×10^{-7}. 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 (SFP) per hour in logarithmic scale in all experiments, i.e. the lower the value, the higher the reliability. We start our experiment from the single job case and apply the proposed design optimization flow to an *mpeg2* decoder example [22].

An important step during embedded system design is design space exploration. The designer needs to make key design decisions such as the amount and the type of PEs needed, considering various application requirements. To illustrate this step, we run the optimization procedure with several platform configurations consisting of 2 to 6 PEs. Only transient faults are considered for the moment. The MOEA is configured with two objectives. The first one is timing overhead. It is defined as:

$$penalty(S) = \begin{cases} -1 & \text{iff } l \leq d \\ l - d & \text{otherwise} \end{cases} \qquad (13)$$

where l is the finish time of the job in schedule S and d is the deadline. The idea is that, if the deadline is met, we set the penalty to a constant -1 and if not, we set the penalty to the difference between the finish time and the deadline. In this way the optimizer will prefer solutions that meet the timing constraints and optimize other objectives. The second objective is reliability using the SFP as a metric. Figure 9 shows the Pareto optimal solutions found by the optimization. It can be seen that the Pareto fronts obtained with more PEs dominate those obtained with less PEs in most cases, i.e. with more hardware resources, the application can be finished with shorter time and higher reliability. This is due to the increased opportunity for spatial redundancy.

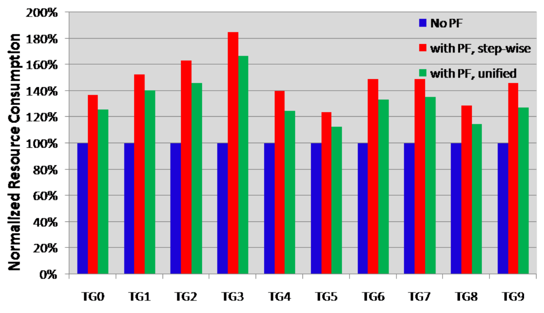


Figure 11: Performance Comparison of Step-wise and Unified Approaches

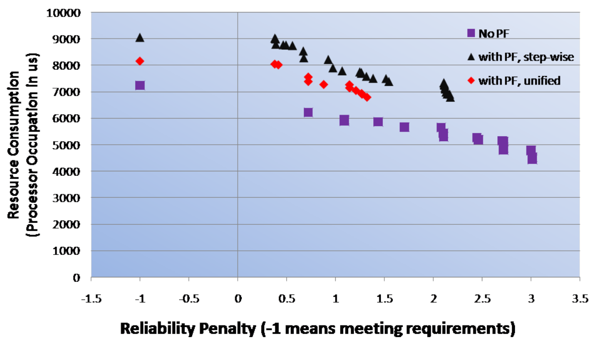


Figure 12: Example of Pareto Optimal Results

For each platform, we are interested in the solution that achieves maximum reliability while meeting the deadline. These solutions are marked with 1 to 3 in Figure 9. As it can be seen, the $2RISC+2DSP$ platform is the minimal one to achieve SFP of 10^{-6} and the $3RISC+3DSP$ platform is necessary to achieve SFP of 10^{-9}. An important observation from Figure 9 is that, if the $2RISC+1DSP$ platform is used, several solutions with SFP around 10^{-6} are very close to meeting the deadline. The same is observed for the platform $3RISC+2DSP$, where several solutions are close to achieving SFP of 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 test two additional platforms with *1 RISC + 2 DSP* and *2 RISC + 3 DSP* (the DSP is faster for the *mpeg2* application). Figure 10 shows the best solution under deadline constraint for each platform. Clearly, the new platforms with *1 RISC + 2 DSP* and *2 RISC + 3 DSP* are the most cost-efficient solutions to achieve SFP of 10^{-6} and 10^{-9}, respectively.

In the next step, the consideration of permanent faults is added and two approaches are compared:

- The step-wise approach in which permanent faults are handled first using spatial replications and then, on top of that, transient faults are handled using temporal and spatial redundancy.

- The proposed unified approach, in which permanent faults and transient faults are considered together using the virtual mapping technique.

As a reference we also compare them with the case in which only transient faults are considered (No-PF). We are interested in how much overhead is needed to fulfill the additional requirement concerning permanent faults. Three optimization objectives are considered, namely schedule length, reliability and resource utilization. For the first two objectives, the same technique as in equation 13 is applied, i.e. the penalty is -1 if the timing/reliability requirements are fulfilled and a positive value otherwise. The resource utilization is the total processor time a schedule occupies. Clearly, all objectives need to be minimized. We assume that it is required to tolerate a single defect on any of the processors. We use a set of Task Graphs (TG) consisting of 5-20 tasks generated synthetically using TGFF. The execution time of each task on the RISC/DSP are generated randomly between 100 to 1000. Figure 11 compares the solution that meets both timing and reliability requirements with minimum resources. The resource consumption is normalized with respect to the reference (No-PF). For the step-wise approach, 47% more resources are needed on average to handle the permanent faults. The unified approach reduces the resource overhead to 33%, i.e. 14% resource saving is achieved. Figure 12 gives a closer view of the Pareto optimal results for one example TG. As it can be seen, the solutions found using the unified approach dominate those found by the step-wise approach. For some jobs, e.g. $TG3$, the additional resources needed to tolerate permanent faults is large. The reason is, those jobs exhibit limited parallelism and the optimizer tends to schedule a large part of the job onto the same processor, so that transient faults can be handled efficiently using re-execution slots. In this case, a large part of the job needs to be replicated/migrated if a defect occurs. As the opposite case, the *mpeg2* application is easy parallelizable and has a relatively tight deadline, which guides the optimizer to a distributed implementation even if permanent faults are not considered. In this case, the additional resources needed are marginal (2% using the unified approach), since only some minor modifications are needed to guarantee feasibility of task migrations.

We proceed with experiments with multiple jobs running concurrently. The focus of this set of experiments is on evaluation of the slack sharing schemes. We compare three configurations: in the first one, no slack sharing is enabled (NSS), i.e. each task has its dedicated replicas and slack slots; in the second one, intra-job slack sharing is used (INTRA), i.e. job-level global slack slots are scheduled and shared amongst all tasks belonging to the same job; and in the third configuration, the proposed SPSS scheme is used (INTER), i.e. global slack slots are shared with all jobs using static priority based approach. We generate 10 random applications with 2 to 3 jobs running concurrently. Figure 13 compares the solution that fulfills deadline and reliability requirements of all jobs with minimum resource consumption. The resource consumption is normalized with respect to the NSS approach. As it can be seen, significant resource saving can be achieved using slack sharing. On average, $INTRA$ and $INTER$ saves 12% and 20% resources, respectively.

9. CONCLUSION

This work considers the reliability-aware task scheduling problem for real-time embedded systems. One main contribution is the proposed BTA approach that enables probabilistic analysis of system reliability in the presence of spatial and temporal redundancy. We also propose an extended version of BTA to handle the case with global slack slots shared by multiple tasks. The analysis is integrated with

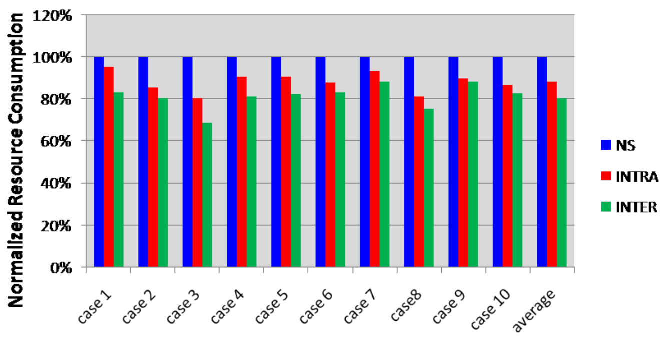


Figure 13: Comparison of Slack Sharing Schemes

an MOEA based optimization process to synthesize legal schedules under timing, reliability and resource constraints. Another contribution of this paper is the virtual mapping technique that enables consideration of permanent and transient faults together in a unified manner. Experimental results verify the effectiveness of our approach. In the next step, we are interested in developing fast heuristics for fault-tolerant task scheduling in order to enhance the scalability of the approach. Another extension of the current work could be to consider the impact of fault-tolerance mechanisms on energy consumption.

Acknowledgment

The authors would like to thank Paul Pop for his helpful suggestions. This work has been supported in part by the European research project ACROSS under the Grant Agreement ARTEMIS-2009-1-100208.

10. REFERENCES

[1] Architectural Requirements. IEC61508-2, chapter 7.4.3.1.1, Tab.2 and 3.

[2] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *RTAS*, 2010.

[3] A. Birolini. Reliability engineering - theory and practice. *Springer, Berlin, Heidelberg*, 2004.

[4] GENESYS Platform. http://www.genesys-platform.eu/.

[5] M. Glaß, M. Lukasiewycz, F. Reimann, C. Haubelt, and J. Teich. Symbolic Reliability Analysis and Optimization of ECU Networks. In *DATE*, 2008.

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

[7] C.-C. Han, K. Shin, and J. Wu. A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults. *IEEE Trans. Comp.*, 2003.

[8] J. Huang, J. O. Blech, A. Raabe, C. Buckl, and A. Knoll. Reliability-aware design optimization for multiprocessor embedded systems. In *Euromicro Conference on Digital System Design (DSD)*, 2011.

[9] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh. Measurement and modeling of computer reliability as affected by system activity. *ACM Trans. Comput. Syst.*, 4, 1986.

[10] 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.

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

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

[13] C. Lee, H. Kim, H.-w. Park, S. Kim, H. Oh, and S. Ha. A task remapping technique for reliable multi-core embedded systems. In *CODES+ISSS*, 2010.

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

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

[16] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich. Opt4J - A Modular Framework for Meta-heuristic Optimization. In *GECCO*, Dublin, Ireland, 2011.

[17] 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.

[18] 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 VLSI*, 2009.

[19] 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.

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

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

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

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

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

[25] C. Yang and A. Orailoglu. Predictable execution adaptivity through embedding dynamic reconfigurability into static mpsoc schedules. In *CODES+ISSS*, 2007.

[26] Y. Zhang and K. Chakrabarty. A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems. *IEEE Trans. CAD of Integrated Circuits and Systems*, 2006.

[27] B. Zhao, H. Aydin, and D. Zhu. Enhanced reliability-aware power management through shared recovery technique. In *ICCAD*, 2009.

[28] D. Zhu and H. Aydin. Reliability-aware energy management for periodic real-time tasks. *IEEE Transactions on Computers*, 99:1382–1397, 2009.