Des Autom Embed Syst (2012) 16:189–220
DOI 10.1007/s10617-013-9105-6

A framework for reliability-aware design exploration on MPSoC based systems

Jia Huang · Andreas Raabe · Kai Huang · Christian Buckl · Alois Knoll

Received: 27 June 2012 / Accepted: 13 February 2013 / Published online: 23 April 2013

Abstract Applying system-level fault-tolerant techniques such as active redundancy is a promising way to enhance the system reliability for safety-related applications. Embedded system design using active redundancy is a challenging task that involves solving two major problems, namely finding the optimal redundancy configuration and mapping/scheduling of the application (including the redundant components) to the platform under timing and reliability constraints. This paper presents a framework for automatic synthesis of fault-tolerant designs on multiprocessor platforms. The core of the framework consists of: (1) a reliability analysis, that computes the system-level reliability in the presence spatial and temporal redundancy, and (2) an optimization approach for reliability-aware design space exploration. The proposed approach considers both transient and permanent faults and is among the first to support system design using imperfect fault detectors. The framework takes an application model, a platform model and a set of application requirements as input, and generates the recommended design parameters, including task-to-processor binding, task schedule and the selection/placement of redundancy. The effectiveness of our approach is illustrated using several case studies.

Keywords Reliability · Fault-tolerance · Design exploration · Real-time systems

J. Huang · A. Raabe · C. Buckl
fortiss GmbH, Guerickestr. 25, 80805 Munich, Germany

J. Huang
e-mail: huang@fortiss.org

A. Raabe
e-mail: raabe@fortiss.org

C. Buckl
e-mail: buckl@fortiss.org

K. Huang (✉) · A. Knoll
TU München, Boltzmannstraße 3, 85748 Garching bei Munich, Germany
e-mail: kai.huang@in.tum.de

A. Knoll
e-mail: knoll@in.tum.de

1 Introduction

Reliability and safety are becoming one of the most important concerns in today's embedded system design. However, as technology scales, modern devices are becoming more susceptible to faults [6]. The hardware faults can be permanent (hard errors), transient (soft errors), or intermittent [46]. Permanent faults cause non-recoverable device defects once they manifest, threatening the system's lifetime. Permanent faults are expected to increase significantly in deep-submicron era [45], due to increased power density and other scaling effects. Transient faults typically arise due to cosmic particles striking the circuit. They do not fundamentally damage the device but may corrupt the application execution. The soft error rate is also expected to increase for each technology generation [6]. Intermittent faults represents malfunction of the device that appear and disappear repeatedly.[1] Despite the efforts made in the hardware community to enhance the hardware reliability, there is an increasing need to use system-level fault-tolerant mechanisms to mitigate the impact of such faults.

System-level fault-tolerance typically involves active redundancy. By replicating certain components in the system, certain faults could be tolerated due to the availability of redundant components. Redundancy can be implemented in both the space and the time domains [9]. In the space domain, critical components can be replicated into multiple copies (also called hardware redundancy). Hardware replication tolerates both transient and permanent faults. For example, a triple-modular-redundant system replicates the critical components three times and votes the results to produce an output. Hardware replication has less timing overhead since the replicas can typically run in parallel. However, extra hardware comes with high design and production cost. In the time domain, software tasks can be selectively replicated (also called software redundancy) [49]. Software replication is more cost-efficient to tolerate transient faults but comes with an overhead in time [20]. For real-time applications, temporal redundancy must be used with utmost care to guarantee schedulability of the tasks (including replicas).

The configuration of fault-tolerant mechanisms, including selection and placement of redundancy, is a critical design decision. First of all, the designer has to reason about the timing and reliability properties of the system in the presence of redundancy to check if all requirements are met. Second, since redundancy comes with high overhead, the optimality of the design is an important concern. Last but not least, the placement of redundancy is tightly coupled with many other design parameters. For example, the amount of redundancy highly influences the schedulability of the application [20, 22]. In particular, for system design on Multiprocessor System-On-Chip (MPSoC) platforms, the selection/placement of redundancy has to be considered jointly with the classical task mapping and scheduling problem.

Over the past decades, a lot of research efforts have been devoted to the field of fault-tolerant system design using active redundancy. However, the existing work still has some limitations. In general, most of the limitations are caused by the simplifying assumptions made to reduce the problem complexity. For example, a lot of work considers either transient fault [12] or permanent fault [14]. Also, only certain class of redundancy are considered and the concurrent usage of temporal/spatial redundancy are not well studied (see Sect. 2 for

[1] Our approach focuses on fine-grained analysis of system reliability. E.g., the proposed analysis can compute the probability that the application is executed correctly for one iteration. In this context, the impact of intermittent faults is similar to that of permanent faults. Hence, we do not consider intermittent faults separately in the scope of this paper.

details). Moreover, fault detection is often assumed to be perfect, i.e., the system always detects the fault if one occurs [55]. Although studying simplified version of the entire problem forms important steps, those simplifications limit the practical use of the approaches.

This paper presents a new approach for the design of reliable real-time systems on MPSoCs. Our approach takes an application model, a platform model and a set of application requirements as input, performs design space exploration and generates a set of recommended design parameters, including task-to-processor binding, tasks schedule and the configuration of fault-tolerant mechanisms. The main contributions of our approach are: (1) A binary tree based approach for probabilistic analysis of system reliability in the presence of spatial and temporal redundancy. The analysis is very generic and supports many advanced fault-tolerance techniques such as shared recovery slack [20] and imperfect fault detection [17]; (2) An efficient two-step encoding to transform the problem into a Multi-Objective Evolutionary Algorithm (MOEA) instance for reliability-aware design optimization; (3) a framework that integrates analysis and exploration approaches to allow automated synthesis of fault-tolerant system design [16].

The remainder of the paper is organized as follows. Section 2 reviews related work in the field. Some preliminaries including the system models are provided in Sect. 3. Section 4 and Sect. 5 details the analysis and optimization approaches, respectively. Section 6 presents the experimental results. Section 7 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 [5].

Reliability analysis Reliability analysis is typically performed in a hierarchical manner, from individual component model up to the system-level model. For permanent faults, reliability models can be constructed at the component-level by analyzing the physical failure mechanisms, such as electromigration (EM) and time dependent dielectric breakdown (TDDB). Based on large set of experiments, researchers proposed several empirical models. For example, EM and TDDB are usually considered to have lognormal and Weibull distributions, respectively [7, 10]. Recent work [48] proposes a framework that integrates device, component and system level models. For transient faults, one classic model is from Shatz and Wang [44], which assumes that the occurrence of transient faults follows a Poisson law with a constant error rate. The reliability model is also extended to cover the effects of voltage scaling on reliability [52, 54].

Fault-tolerant system design focusing on permanent faults When focusing on permanent faults, the system reliability is often referred to lifetime reliability. The common measure is Mean Time To Failure (MTTF). Popular mechanisms to increase the MTTF of the system include hardware hardening, hardware redundancy and task migration. While hardening is mainly a hardware technology, the last two mechanisms involve interesting scheduling and optimization problems. The work [15] presents a lifetime-aware task mapping approach on chip multiprocessors. They focus on wear-out related permanent faults and take into account temperature-dependent failure mechanisms. The Ant Colony Optimization (ACO) algorithm is used to search for the optimal task mapping schemes. The authors in [18] consider a similar problem. In particular, the aging effect of components in a multiprocessor system is taken into account. Feldmann et al. [8] presents an approach that focuses on analyzing the feasibility of the system up on permanent faults. They define a new metric *k-bindability*, which

specifies the property that a feasible binding of the application to the platform exists even if any k components fail. Quantified Boolean Formulas are used to calculate the k-bindability of a system. Glaß et al. consider a similar problem [14]. They extend the approach in [8] and consider redundant binding of a task to multiple resources for the sake of reliability improvements. The system behavior in the presence of redundancy is described using the so-called *structure function* and represented as Binary Decision Diagrams (BDDs). A path in the BDD towards *true* represents a combination of faults that is tolerable with the current system setup. The system-level reliability can then be evaluated based on component-level reliability models. The analysis is integrated into a MOEA based optimization framework to find the best task bindings [13]. The same authors further consider the automatic insertion of voting components in [41]. Compared to the work mentioned above, our reliability analysis considers transient faults and is also able to handle software fault-tolerance techniques.

In [53] the authors utilize online fault detection and task migration to maximize the expected MTTF. On detecting certain faults, the system is restarted and the tasks are re-allocated to remaining non-faulty components according to a pre-computed plan. The task migration cost is not considered, instead, the focus is on increasing the MTTF as much as possible. The work [30] proposes to extend the MTTF by active allocation of *slack* in the system. For example, some processors can be intentionally replaced by high-performance ones, so that tasks from failed processors have higher possibility to be migrated. When task migration is used, one of the most important issues is to guarantee the application requirements after the recovery while in the mean time reducing the migration cost. Yang et al. [50] propose an approach for generating schedules with predictable response to faults. They partition the initial schedule into several bands, which are designed in a way that the capability of re-mapping tasks is embedded. The work is extended in [51] to minimize the latency of applications. Lee et al. [25] study the problem of static task re-mappings under throughput constraints for streaming applications.

Fault-tolerant scheduling considering transient faults Several existing approaches focus on transient faults. In [49], the authors present an approach to handle transient faults by selectively inserting task re-executions. They focus on using the otherwise wasted resources to enhance the system reliability in a best-effort manner. The work [40] presents an approach for static scheduling with fixed fault-tolerant mechanism assignment. To be more specific, each task is replicated twice so that a single processor failure can be handled. Girault et al. [12] consider fault-tolerant scheduling with active task replications and present a bicriteria heuristic algorithm. They adopt the classic Poisson fault model and assume perfect fail-silent behavior. Besides task scheduling, their algorithm also determines the number of replications that are needed to achieve certain reliability goal. Only spatial redundancy is considered and the replicas of a task are always scheduled on different cores.

Izosimov et al. [20] study the design optimization of fault-tolerance systems using both spatial and temporal redundancy. In particular, the technique of sharing re-execution slack among multiple tasks is proposed to improve the efficiency. A tabu-search based optimization procedure is used to find the best schedule with scheduling length being the optimization goal. In [37] Pop et al. study a similar problem and consider in addition the utilization of check-pointing and roll-back technique. The authors in [42] utilize a hybrid scheduling approach to handle mixed hard and soft real-time tasks. The aforementioned work [20, 37, 42] is based on a simplified fault model. Instead of modeling faults as probabilistic events, they assume that the system may experience at most N faults and those faults may occur in any component of the system. Under this assumption, the authors propose approaches for automatic derivation of the optimal the task-to-PE mapping and fault-tolerance policy

assignment (e.g., the amount of replication and placement of check points). The simplified fault model has the limitation that the distinct failure probabilities of different hardware components are not taken into account. In the follow-up work [19], a more accurate probabilistic analysis is presented. Nevertheless, this analysis considered only temporal redundancy. Our paper tackles a similar problem as the work mentioned above [20, 37, 42, 55]. The proposed probabilistic reliability analysis is more generic and computes the system reliability in presence of both spatial/temporal redundancy and shared re-execution slacks. Additionally, we introduce an approach based on evolutionary algorithms that allows consideration of 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.

Other work also studies the tradeoff between reliability and other design objectives, such as energy [55] and cost [36]. In [38] the authors present a Constraint Logical Programming (CLP) based approach for scheduling and voltage scaling for fault-tolerance systems. Zhu et al. show that voltage scaling has direct and adverse effects on system reliability [54]. They study static scheduling approaches for energy minimization under reliability constraints [55]. The core idea is, instead of using all available slack time for energy management, a portion of the slack is especially reserved to schedule task re-executions, such that the reliability loss can be recuperated.

An important limitation of the work mentioned above [20, 37, 38, 42, 54, 55] is perfect fault detection assumption. To reduce the problem complexity, the authors assume that all transient faults can be detected when a task is completed and timing overhead of fault detection is contained in the WCETs of tasks. However, fault detectors, especially those have high detection coverage, may come with high resource and timing overheads [43]. These resources could potentially be used for other purposes, e.g., to implement more replicas. Seen from another angle, the overhead in fault detection may limit the resource available for active redundancy, resulting in sub-optimal system reliability. Hence, it is important to consider optimization of fault detector implementation in the design flow. The previous work [17] discusses in particular the selection of error detector and the proposed techniques are integrated in our analysis/optimization framework (see Sect. 4.2). Experimental results verify that certain configuration using imperfect fault detectors combined with replication can outperform those approaches that utilize only perfect fault detectors.

In [26], the authors consider another important tradeoff, namely the tradeoff between hardware-implemented and software-implemented fault detection. They propose to selectively implement fault detectors in a FPGA fabric tightly coupled with the processor, so that the fault detector can run in parallel with the original program and the timing overhead of fault detection can be reduced. Given limited FPGA resource, it is critical to decide which fault detector goes to hardware. Optimization techniques are proposed for this purpose. FPGA-accelerated fault detection is currently not considered in our work. To take this issue into account, the problems considered in [26] and [17] have to be combined. Here, the design goal is to decide both *which* fault detector to implement and *where* to implement. Nevertheless, considering the combined problem is out of the scope of this paper.

3 Preliminaries

3.1 Fault model

Following the classic terminology, a **fault** is a physical defect, imperfection, or flaw that occurs within some hardware or software component [39]. A fault may be dormant, which

means the execution of the component is not affected, or it may be activated, which means an **error** is incurred in the component.[2] The error, as the manifestation of a fault, may subsequently cause a **failure**. A failure is an observable event that the system deviates from the specified behavior. Fault-tolerance is the technique to reduce the probability of failure despite the presence of faults. It can be applied at architectural level to reduce the probability of fault-to-error transition [33], or it can be applied on application-level to reduce the error-to-failure transition. Our work focuses on the later case.

We consider software tasks as the basic components. The DSE framework takes the error rates of tasks as input and aims at optimizing the system reliability (i.e., the system-level failure rate). The task-level error rates are obtained from an **fault model**, which describes the mode, distribution and other properties of faults. Fault models are typically proposed by reliability engineers after detailed analysis and modeling of the physical failure mechanisms [48]. The system designer may select the appropriate one for the target application domain. Our DSE approach do not have restriction on the selected fault model, as long as the task error rates can be obtained. This section discusses the fault models that we select for the experiments, concerning both transient and permanent faults.

Transient faults may cause errors in a program. It can either be that the program execution is corrupted (program hanging, segmentation error, etc.) or that the program executes smoothly but delivers an incorrect output. In both cases, the task is considered as faulty. Nevertheless, since transient faults do not fundamentally damage the device, we assume that only the single task during which the faults occur is corrupted. The successor tasks can be executed normally after a recovery process. Moreover, we focus on errors of the application program and consider the kernel software (e.g., OS scheduler, watchdog) to be fault-free.[3]

We adopt the classical Poisson fault model to describe the distribution of transient faults. The Poisson model is well established and used in many related literature [2, 12, 44, 55]. It assumes transient faults to be independent events following a Poisson distribution with a constant failure rate. Under this assumption, the following equations compute the probability that a task is executed correctly and the converse probability that the task experiences transient faults:

$$P(\text{task } t_i \text{ executes correctly on processor } p) = e^{-\lambda_p w_i} \tag{1}$$

$$P(\text{task } t_i \text{ experiences transient faults}) = 1 - e^{-\lambda_p w_i} \tag{2}$$

where λ_p is the failure rate of the processor p and w_i is the Worst-Case Execution Time (WCET) of task t_i. The reliability requirement concerning transient fault is given by the maximally allowable failure probability of the system.

Note that by assuming the Poisson fault model, our approach does not consider Common-Mode Failures (CMF). In reality, CMFs could cause correlation between faults, which violates the independence assumption made in the fault model. However, our approach is not intended to handle CMFs, since, as observed in [24], active redundancy is not a solution to CMF [31]. Instead, CMFs have to be mitigated by dedicated techniques, such as design diversity, architectural-level fault-containment and spatial/temporal separation [32, 34]. In general, taking CMFs into reliability analysis is relatively straightforward, e.g., using techniques mentioned in [31]. The real problem is how to estimate the probability of CMFs,

[2]Dormant faults are not considered in our approach, since they are neither noticeable nor harmful. In other words, we focus on activated faults only. In this case, a fault is equivalent to an error from the designer's viewpoint.

[3]This is because we cannot apply fault-tolerant techniques such as active redundancy on the kernel software.

which can be extremely difficult [31]. For this reason, a lot of research effort has been devoted to CMF avoidance. In our paper, we assume CMF avoidance techniques are systematically applied, allowing us to use the Poisson model to model transient faults.

Concerning permanent faults, we focus on defects of processing elements in the MPSoC platform and assume each individual core to fail independently. This assumption requires core-level fault containment in the underlying platform. Although fault containment is challenging to implement, it is a prerequisite to enable using MPSoCs for safety-related applications.[4] Hence, recent work [15, 18, 25] mostly make the same assumption. Also, research efforts are spent on temporal/spatial separation techniques to implement the desired fault containment property, e.g., the ACROSS architecture [1]. In this context, the entire system is "alive" as long as the remaining working cores can still provide expected service. The component-level reliability is typically given as reliability functions and the design goal is to optimize system MTTF (cf. [14]). In reliability analysis, permanent and transient faults are typically considered separately, since their physical failure mechanisms and impact on the system are significantly different. In our approach, we target on considering both type of faults in a unified manner, since it improves the system performance as shown in [16]. One obvious way to achieve this is to integrate the existing analysis in [14] and consider lifetime reliability as one extra optimization goal. However, additional optimization objectives reduce the optimization efficiency considerably. To overcome this problem, we assume that the reliability requirements concerning permanent faults are given as constraints. For example, the requirement could be that the system must tolerate one permanent fault of any of the processors. We develop an encoding technique that guarantees these constraints during the optimization process (see Sect. 5).

3.2 Fault tolerance mechanisms

As mentioned before, we focus on using active redundancy to enhance the system reliability. Software tasks are replicated into multiple copies (replicas). The replicas can be executed on the same component (temporal redundancy) or distributed to several components (spatial redundancy). The set of N replicas for a task t_i is denoted as $R(t_i) = \{t_{i,1}, \ldots, t_{i,N}\}$. The availability of replicated software tasks allows for implementation of subsequent voting. The voter collects inputs from all replicas of a task, including both temporal and spatial copies, and produces a reliable data for successor tasks. By comparing the redundant results, the voter may detect or correct errors. In this thesis, we consider a **majority** voter, which generates an output if and only if more than half of the inputs have equal value. In this case, the voter can *correct* an error, if only less than half of the replicas are faulty. If no majority is found from the voting inputs, the voter reports an error, which means the error is *detected* but not corrected. In rare cases, more than half of the replicas can fail and send equal but incorrect result to the voter. Since the voter just selects the majority, the error will escape. We assume that the voter features a timer to detect missing inputs, i.e., if one replica encounters a fault and fails to send its result to the voter, the available data gathered from other working replicas will be used for voting.

Another fault-tolerant mechanism that we consider is additional fault detectors embedded into a specific task instance. This could be done in hardware, software or combined [26].

[4] Another possibility is to consider the entire MPSoC as a fault containment unit and apply active redundancy in distributed chips. However, this option comes at a much higher hardware cost. Moreover, it does not well exploit the benefit of the MPSoC platform, e.g., fast on-chip synchronization and communication between different redundant components.

Software-implemented fault detection typically involves transformation of the original program into an instrumented version, adding the capability to detect transient faults that occur at runtime of the program [43]. At the tasks' completion, some check rules are executed to decide if a fault has occurred. The arithmetic codes [43] and critical variable technique [35] are examples of this kind. Hardware techniques typically introduce some monitoring functionality, e.g., fingerprinting mechanism [23], to check if the program is executed as expected.

Fault detectors implemented at individual task can help to improve the system reliability, since, up on detecting a fault, the task can take appropriate actions such as safe shut-down stop error propagation. Reliability analysis becomes more complicated in this case since the fault detection coverage also becomes one factor that influences the system reliability. To simplify the problem, most existing approaches that consider embedded fault detection assume that all faults can be detected using such fault detectors [19, 20, 42, 52, 55]. In other words, the fault detection is considered as *perfect* and all task instances can have a fail-silent behavior. In this case, only correct outputs will be sent to the successor tasks and voting becomes trivial. This assumption reduces the problem complexity significantly but raises some practical concerns (see Sect. 4.2 for details). In this paper, we start with the same assumption and present our tree-based reliability analysis approach. However, this assumption is relaxed and the residual error of fault detection will be taken into account in the second step.

Concerning error-recovery, we assume that the system will roll-back to a safe state and execute the next scheduled task after a failure caused by transient faults. The timing overhead of recovery is considered as constant annotated by the user [20]. To recover from permanent faults, the schedule table on each core is switched to a pre-computed emergency schedule (see Sect. 5.4) and the system restarts from the beginning of the application's period.

3.3 System models

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 job $\mathcal{J} = (\mathcal{T}, \mathcal{E})$, the vertices $\mathcal{T} = \{t_0, t_1, \ldots, t_m\}$ represent a set of *tasks* to be executed and the edges $\mathcal{E} = \{e_0, e_1, \ldots, e_l\}$ 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.

Our target architectures are heterogeneous multiprocessor platforms with time-triggered communication, e.g., the GENESYS [11] architecture. The communication between tasks is implemented with messages. The message schedule $\mathcal{M}$ is described as a set of message slots $\{m_0, m_1, \ldots, m_k\}$. 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. The communication can be protected with dedicated techniques (e.g., error correction code) and is therefore assumed to be error-free.

3.4 Scheduling models

Timing predictability is highly desirable for safety-related applications. In our approach, we target on synthesizing *time-triggered fault tolerant schedules*. We support two scheduling models, namely hierarchical combination of Time-Triggered and Static Priority scheduling (*TT-SP*) and Time-Triggered scheduling with Flexible Slack (*TT-FS*). The *TT-FS* scheme is first proposed in [20] and *TT-SP* is introduced in [16].

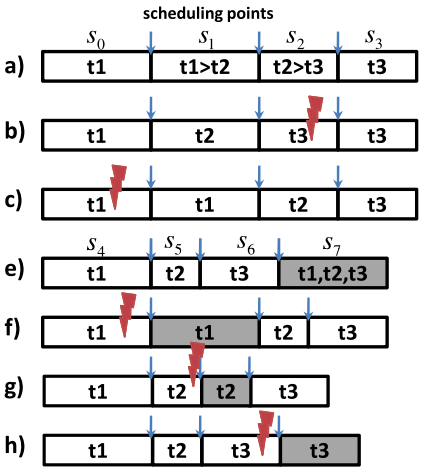


Fig. 1 Example of *TT-SP* and *TT-FS*

TT-SP In the *TT-SP* scheme, the available processing elements are globally arbitrated in time and budgets are statically allocated to tasks. In each time slot, a set of tasks are allocated and ordered using static priorities. At runtime, the *pending* task that has the highest priority acquires the slot for execution. A task is pending if and only if (1) all the required data is available; (2) the execution is necessary, i.e., the task has not been executed successfully in previous slots. Figure 1a shows an example of *TT-SP* schedule. The slot S_1 is allocated with two tasks and t_1 has higher priority. In this case, the re-execution of t_1 will take place in S_1 whenever necessary, e.g., as shown in Fig. 1c where the first instance of t_1 fails. Task t_2 may execute in S_1 only if the high-priority task t_1 finishes before the start of S_1 (Fig. 1b). A *TT-SP* schedule can be described as a set of non-overlapping slots $\mathcal{S} = \{s_0, s_1, \ldots, s_n\}$, each being 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 list of tasks with decreasing priority assigned to s. The size of a time slot is determined by the longest worst-case execution time (WCET) of all tasks assigned to it. An important feature of *TT-SP* is that the start/end time of each slot is fixed and does not have dependency on the faults occurred.

TT-FS In the *TT-FS* scheme, two types of time slots are scheduled, namely normal slots and *slack slots*. The later is intended to be used for re-execution of instances misbehaving due to transient faults. Slack slots are often shared by multiple tasks. Figure 1e shows an example, in which the slack slot S_7 is shared by t_1, t_2 and t_3. The slack slots only reserve time for re-execution but do not have a fixed start time. Instead, they will be utilized whenever necessary. In Fig. 1f, the first instance of t_1 encounters a fault and S_7 is used immediately to re-execute the same task. The normal slots S_5 and S_6 are delayed in this case. Figure 1g and 1h are another two scenarios, in which S_7 is used to re-execute t_2 and t_3, respectively. Naturally, the size of the slack slots must be no smaller than the WCET of any tasks assigned to it. To describe a *TT-FS* schedule, the four-tuple $s = (b, f, p, T)$ needs to be extended with an additional binary variable denoting if the slot is a slack slot or not.

The analysis and optimization techniques presented in this work support both *TT-SP* and *TT-FS*. For the sake of simplicity, we focus mainly on the *TT-SP* scheme for the rest of the paper. Nevertheless, we present the details on how to utilize the same techniques for *TT-FS*. It is up to the designer to choose one of the scheduling schemes.

Implementation Both *TT-SP* and *TT-FS* are distributed scheduling models. They are based on pre-computed schedule tables stored statically at each individual core. Since the system is time-triggered, synchronization between cores is a prerequisite for implementation. The synchronization protocol is platform-specific. For our target ACROSS architecture [1], this is done by synchronizing all application cores with the same network clock. We assume the scheduler is part of the OS that is fault-free.

At runtime, the *TT-SP* scheduler picks the pending task with highest priority for execution. Once one replica of a task is finished successfully, the other replicas of the same task are removed from subsequence time slots for the current iteration to avoid duplicated executions. Another situation that should be avoided is that a task in previous slot becomes "hanging" due to fault and blocks the execution of subsequence tasks. A hardware watchdog can be used for this purpose. The implementation of a TT-FS is a bit more complicated, since the schedule has to be adapted at runtime depending on the faults occurred. In general, once a fault occurs, the scheduler has to make emergency response and try to achieve a correct execution by using the slack slots. Detailed explanation of the implementation with an example is presented in [21].

4 Reliability analysis

In the proposed framework, the reliability analysis focuses on computing the system-level reliability under impact of transient faults. Permanent faults are taken into account using an encoding technique in the optimization procedure (Sect. 5). Computing the system reliability of a given design is a very difficult problem, especially when fault-tolerant mechanisms such as active redundancy are present. Recent work [4] puts special emphasize on the complexity of reliability analysis. The authors distinguish two types of schedules, namely *strict schedules* and *general schedules*. The strict schedules obey a rule that if a task t has a data dependency on task t', all replicas of t' should be completed before any replicas of t start. With this restriction, the execution results (success or faulty) of predecessor tasks will have no influence on the start time of successor tasks. In this way, the tasks can be considered independently in reliability analysis and a closed form formula can be derived [12]:

$$\Pr(\mathcal{S}, J) = \prod_{t \in J} \left(1 - \left(1 - \Pr(t)\right)^{num(t)}\right) \tag{3}$$

where $\Pr(\mathcal{S}, J)$ is the reliability of job J achieved by schedule $\mathcal{S}$, $\Pr(t)$ is the reliability of task t and $num(t)$ is the number of replicas that task t features. As for the general schedules, the author prove that the problem is at least as hard as NP-Complete problems [4].

The reliability analysis for *TT-SP* and *TT-FS* schedules is even more difficult than the ordinary general schedules. First of all, a larger set of fault-tolerant mechanisms are utilized. In the current work [4, 12] the replicas of each task are always mapped to different processors. In other words, only the hardware redundancy is used. For *TT-SP* and *TT-FS*, concurrent hardware and software redundancy has to be considered. Second, shared time slots must be supported. In [4, 12], each slot is used for exactly one task. At the beginning of each slot, we automatically know which task is going to be executed. For *TT-SP* and *TT-FS* schedules, slots can be shared by multiple tasks. The actual utilization of slots depends on the execution history of previous slots. Figure 2 depicts an example, in which the utilization of slack slots S_2 and S_3 depends on the execution result (success or faulty) of previous slots S_0 and S_1. In particular, the execution results on one processor might also influence the execute

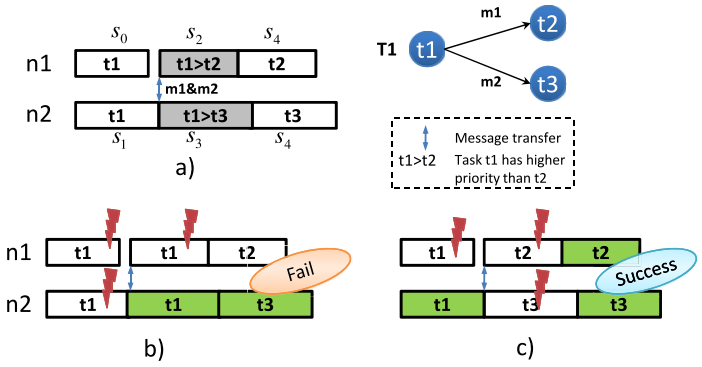


Fig. 2 Example execution scenarios for *TT-SP*

sequence on other processors. In the same figure, if the instance t_1 succeeds in slot S_1 on processor n_2, the message will transfer the correct result to processor n_1 and the slot S_2 can be left for t_2 (see the execution scenario in Fig. 2c). Otherwise, the slot S_2 has to be used for re-executing t_1 (Fig. 2b). The analysis algorithm must also maintain the execution history to select the correct task for the next slot. New analysis techniques are needed to conquer the extra complexity. In principle, to obtain the system-level reliability for *TT-SP* and *TT-FS* schedules, we need to carefully investigate which combinations of faults are tolerable by a certain schedule and which combinations are not. In the next sections, we propose a binary tree based approach.

We describe a combination of faults occurring in a system by a *fault scenario*:

Definition 1 (Fault Scenario) A fault scenario is a vector $\mathbf{x} = \{x_0, x_1, \ldots, x_n\}$, 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 $s_i.T$[5] is either not ready or finished earlier and no task is actually 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:

$$\Pr(\mathcal{S}, J) = \sum_{\mathbf{x} \in W(\mathcal{S}, J)} \Pr(\mathbf{x}) \tag{4}$$

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., $S(t_j) = \{s \in \mathcal{S} | t_j \in s.T\}$. 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

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

scheduling, $u_{i,j}$ computes to:

$$u_{i,j} = r_{i,j} \wedge \left(\bigwedge_{\substack{t_l \in s_i.T \wedge \\ priority(t_l) > priority(t_j)}} \neg r_{i,l} \right) \tag{5}$$

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:

$$r_{i,j} = isReady \wedge notPrev \wedge notOther \tag{6}$$

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

$$notPrev = \bigwedge_{\substack{s_k \in S(t_j) \wedge s_k.p = s_i.p \\ \wedge s_k.f \leq s_i.b}} \neg(u_{k,j} \wedge x_k = 1) \tag{7}$$

It is true if t_j has not been successfully finished on the same processor. The term *notOther* checks if the task has been executed successfully on other processors and a message is scheduled to convey the result to the local processor:

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

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.

As mentioned before, we assume perfect fault detector at this point. Hence, a task is successful if at least one instance of it is executed without faults:

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

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:

$$\varphi_J(\mathbf{x}) = \bigwedge_{t_j \in \mathcal{T}} success(t_j, \mathbf{x}) \tag{8}$$

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 Fig. 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 Fig. 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 Fig. 3) represent the case that the slot executes some

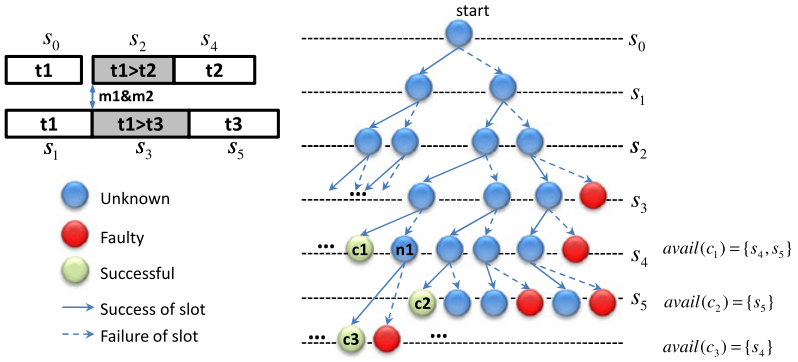


Fig. 3 An example of binary tree analysis

task correctly. Right branches (dashed lines in Fig. 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 (see node n_1 in Fig. 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 (5) to (6) 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.

With the request/utilization variables, a fault scenario $\mathbf{x}$ can be evaluated using (8), 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 are used as the working set. The analysis process above can be implemented recursively as outlined in Algorithms 1.

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

$$\Pr(\mathbf{x}) = \prod_{x_i \in \mathbf{x} \wedge x_i = 1} \Pr(s_i) \cdot \prod_{x_i \in \mathbf{x} \wedge x_i = 0} \big(1 - \Pr(s_i)\big) \tag{9}$$

where $\Pr(s_i)$ is the success probability of the task executed in slot s_i. The task-level error probability $\Pr(s_i)$ are computed using fault model, e.g., the Poisson model described in Sect. 3.3. With the task-level reliability and the working set, we can obtain the system reliability using Eq. (4).

4.1 Complexity and approximation

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

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

```
//compute request and utilization variable using 5 and 6
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
```

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 Eq. (9) 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 (4), ignoring possibly tolerable nodes is a safe underestimation of system reliability (see proof below). From the tree point of view, this corresponds to eliminating all nodes with more than d right branches on their paths to the root node.

With the above estimation, the complexity of BTA is reduced to be polynomial in $|\mathcal{S}|$ (see [16] for the proof). Note that what we analyze in [16] is 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 3 GHz CPU) is 754 ms for $d = 3$ and 3405 ms for $d = 5$. Thus the runtime of BTA is acceptable for an offline optimization process.

Correctness proof of the approximation in BTA We prove the correctness of the approximation in BTA by showing that the approximation is pessimistic underestimation of system reliability. The design decisions made based on the BTA result are therefore safe.

Lemma 1 *Eliminating nodes during binary tree analysis is a safe underestimation of system reliability.*

Proof The system reliability is computed by accumulating the occurrence probability of all nodes in the working set (see Eq. (4)). By eliminating a node during BTA, we consider the node as *faulty* without checking even if it is possibly *successful*. The occurrence probability of this node will not be added to the system reliability. In this case, the computed system

reliability will be less or equal than the real value. Hence, the BTA result is pessimistic and safe. □

Theorem 1 *Visiting only nodes with at most N ($N > 0$) faulty slots during BTA is a safe approximation of system reliability.*

Proof We prove it by induction. *Start node.* The fault scenario represented by a node is determined by the path from the node itself to the start node. Since the start node has an empty path, it represents a dummy fault scenario and will not be considered during BTA. *Nodes in the first level.* For a node in level l, there is exactly l branches along the path to the start node. Since $l \leq N$, the two nodes in the first level are never eliminated by approximation. If they represent tolerable fault scenarios, the according probabilities will be accumulated.

Induction. Assume the BTA is visiting a node A in level L. The number of faults specified by scenario A (denoted by $faults(A)$) must be less than or equal to N, otherwise it is already eliminated. The two child nodes of A is expanded only if A is an *unknown* node, i.e., this branch terminates if A is identified as a *successful* or a *faulty* node (cf. Algorithm 1). The branch to the left child B specifies a successful execution of the slot associated with the current level. Hence, $faults(B) = faults(A) \leq N$ and B will not be eliminated. No approximation is made at this point. The branch to right child C specifies a new fault occurring at the current level and $faults(C) = faults(A) + 1$. If $faults(C) \leq N$, the BTA continues normally with C and no approximation is performed. If $faults(C) > N$, node C is directly considered as *faulty* and the BTA terminates at this branch even though the children of C could possibly be successful. Nevertheless, eliminating possible *successful* node is a safe underestimation of the system reliability according to Lemma 1. Hence, for either of the children of the current node A, the approximation, if it takes place, is safe. □

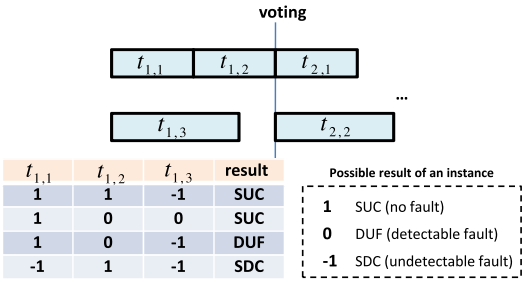
4.2 Designing fault-tolerant systems using imperfect fault detectors

Up to now, we present the analysis approach under the assumption that all transient faults are detected at the end of the task execution. This assumption simplifies the problem but is problematic in practice. On the one hand, a perfect detector might not exist or is difficult to implement, making the algorithms developed under this assumption less useful. On the other hand, even if implementable, perfect detectors typically come with high resource and timing overheads. In recent work [29, 43] it has been shown that the time needed for high-coverage fault detection may become much longer than the execution time of the task itself (e.g., the timing overhead could be 400 % using techniques proposed in [29]).

The problem can be considered from a different angle. In the fault-tolerance community, researchers have develop various fault detection techniques that achieves certain fault detection coverage and comes with certain overhead [29, 43]. It is critical to select which fault detector is to be implemented for each task. By making the assumption of perfect fault detection, a *biased design decision* is made, which selects the most expensive fault detector for every task. Other design alternatives with partial fault detectors are ignored. Moreover, since fault detection causes timing overhead, selecting better fault detector reduces the opportunity for spatial/temporal replication. Clearly, a tradeoff analysis is needed to find the optimal setup.

Taking imperfect fault detection into account, the execution of a task instance may result in three scenarios: (1) it executes successfully, (2) a transient fault occurs and is detected and (3) a transient fault occurs and is not detected. Detectable and undetectable faults have

Fig. 4 Voting scenario analysis



different influence on the system reliability. If a fault is detected, fail-silent behavior can be achieved. In this case, the faulty instance produces no output and the correct outputs from other instances will not be affected. As the counterpart, an undetected fault leads to the case that a wrong output is delivered to the subsequent tasks without warning. If a subsequent task receives different inputs from the replicated tasks, it has no knowledge at this point about which of the input is correct. In practice, voting is typically used to handle this dilemma. Since faults are considered as rare events, the majority among the set of inputs is considered to be correct.

We assume a majority voting mechanism is implemented for each task that has replicas available. The voter generates an output if and only if a dominating result (i.e., a majority) is found. The overall execution of a task, considering all its instances, could again result in the following 3 scenarios: (1) the task executes successfully (*SUC*): it experiences no fault or only some faults that are later corrected by the voter; (2) Detected Unrecoverable Faults (DUF): the voter fails to find a dominating result and thus produces no output; and (3) Silent Data Corruption (*SDC*): multiple faults occur and the incorrect outputs mask the correct one. Both *DUF* and *SDC* are unwanted behavior that negatively influences the system reliability.

The BTA approach has to be extended to consider imperfect fault detection. On the one hand, the *fault scenarios* has to be extended to model three states.[6] That is, the variable x_l that describe the execution result of slot l has three possible values: x_l is 1 if the slot executes some task correctly; x_l is 0 if the slot fails silent (a fault occurs and is detected) and x_l is -1 if the slot produces an incorrect output (i.e., a fault occurs and is not detected). On the other hand, voting has to be supported. Since the voter has to collect the inputs from all replicas to generate an output, the schedules that the framework generates are always *strict schedules* [4, 12]. In this case, the tasks in a job can be considered independently in the reliability analysis. The details of the extended analysis is presented in [17].

Figure 4 depicts an example of the voting scenario analysis of the task t_1, which features three replicas. If the fault scenario is $\mathbf{x} = \{1, 1, -1\}$,[7] the incorrect output of $t_{1,3}$ is masked and the overall result is *SUC*. In the scenario $\mathbf{x} = \{1, 0, 0\}$, both $t_{1,2}$ and $t_{1,3}$ produce no result, and the only output from $t_{1,1}$ will be taken. Hence, the overall result is also *SUC*. In the scenario $\mathbf{x} = \{1, 0, -1\}$, a correct and an incorrect output are sent to the voter. The voter cannot identify the correct input since no majority is found. In this case, no output is generated and the overall result is *DUF*. In the last example scenario $\mathbf{x} = \{-1, 1, -1\}$, two incorrect outputs are sent to the voter. Note that the fault scenarios model only the qualitative

[6] In this case, the tree that is built is not anymore a binary tree. However, we just keep the name BTA for simplicity.

[7] Since tasks are considered independently, only three slots are relevant for analyzing t_1.

result (0, 1, or −1), but the voting is performed based on the real value of the tasks' outputs. Hence, if two outputs are incorrect, two cases might happen: (1) the two incorrect outputs are equal and mask the single correct one, resulting a *SDC*; (2) the two incorrect outputs are unequal and the voter does not see a dominating value, resulting in a *DUF*. To stay on the safe side, we have to assume the first case (*SDC*), because the probabilities of the two cases are very difficult to be quantified, even if possible.[8]

The BTA approach is used to analyze the voting scenario for each task. Afterwards, the system-level reliability, considering both *DUF* and *SDC*, is computed [17]. One thing worth mentioning is that, since reliability analysis of strict schedules is much easier, the extended BTA has linear complexity in the number of tasks.

5 Optimization procedure

Guided by analysis results, we considered how to find the optimal task schedule. We adopt the Multi-Objective Evolutionary Algorithm (MOEA) as the optimization engine. To use MOEA, the candidate solutions must be encoded into a special data structure called *chromosome*. The set of chromosome maintained by the optimizer is called the population. In each iteration, the optimizer selects a subset of solutions from the population, which are used as parents to produce offspring (new solutions). This procedure is done by applying crossover and/or mutation operators. The new solutions are evaluated by fitness functions and high quality solutions will replace low quality ones in the population. This process repeats until a candidate with sufficient quality is found or a maximum number of iterations is reached. Figure 5 shows an overview of MOEA.

To describe a *TT-SP* or *TT-FS* schedule, the information about each slot is needed, including the start/finish time and task assignments. A direct encoding of such a schedule generates a very large chromosome, resulting in a huge search space and low optimization efficiency. To cope with this problem, we utilize a two-step encoding process inspired from [27]. The main idea is, instead of encoding the entire schedule, we only put partial information, namely the mapping and redundancy configuration, into the chromosome. A scheduler is used to rebuild the schedule from the chromosome, which is then used for fitness evaluation, e.g., reliability analysis. For the rest of this section, we present the encoding techniques for *TT-SP*

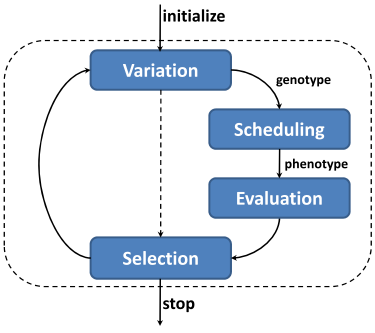


Fig. 5 Workflow of EA-based optimization

[8]The probabilities are highly influenced by the application characteristic, the output data type, common caused errors, etc.

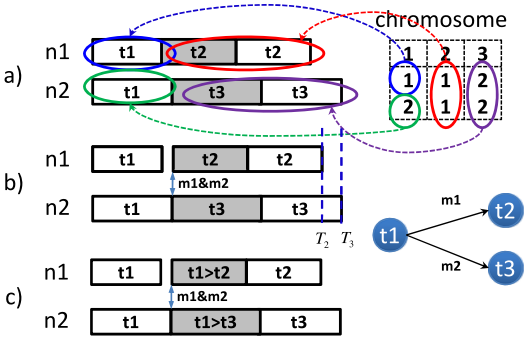


Fig. 6 Encoding and reconstruction of schedule

schedules. Section 5.2 discusses necessary changes in the approach to support *TT-FS* schedules.

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 Fig. 6. Multiple mappings of the same task onto the same PE are interpreted as re-execution slots (task 2 and 3 in Fig. 6); multiple mappings of the same task onto different PEs are interpreted as spatial replications (task 1 in Fig. 6).

5.1 Schedule reconstruction

Reconstruction of the schedule from the chromosome is the same problem as scheduling the tasks with known mapping and redundancy configuration. The reconstructed schedule is sent to the BTA to evaluate the reliability for current solution. The selection of scheduler is a user decision and has no influence on the correctness of analysis. For example, the user may implement a scheduler that only generates strict schedules or another one that also generates general schedules. The BTA is generic and supports both types.

The scheduling procedure that we propose consists of three main steps. First, for each mapping entry of a task t, we instantiate a scheduling 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 (Fig. 6b). In this paper, we adopt the *transparent recovery* approach [22], which requires that a fault occurring on one PE is masked to other PEs. This approach 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 sender are not visible to the receiver, e.g., if the task t_2 in Fig. 6 sends a message to other tasks, the message should be placed at time T_2. Tasks may be postponed due to dependency on messages. In the last step, we perform slack sharing (Fig. 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 an execution time not greater than the slot size.

An advantage of the two-step encoding is that many application specific constraints can be easily translated into rules on the chromosome. An example would be separation constraints, e.g., two critical tasks are required to be strictly isolated in space. This can be guaranteed by adding a constraint that the mapping entries of the two tasks do not collide.

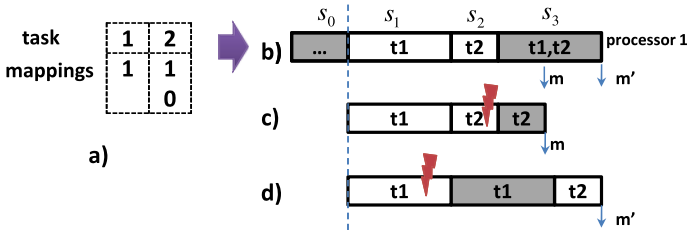


Fig. 7 Example of message placement for *TT-FS* schedules

5.2 Encoding of *TT-FS* schedules

The encoding scheme needs slight modifications to handle *TT-FS* schedules. We use the integer 0 to denote slack slots and integers larger than 0 to denote regular slots. Figure 7 shows an example. Two tasks t_1 and t_2 are allocated on processor 1 and one slack slot is scheduled. To reconstruct the schedule from chromosome, the same list scheduler described in Sect. 5.1 can be used to generate an initial schedule. The slack slots are placed right after the regular slots of the same tasks, e.g., S_3 is located directly after S_2. We introduce a greedy slack sharing approach that works as follows. First, the schedule is divided into several segments. The segments are separated by one or several consecutive slack slots. Then each slack slot is allowed to be shared by all tasks in the segment just before itself. As an example, the regular slot S_1 and S_2 in Fig. 7 belong to the segment separated by slack slots S_0 and S_3. Hence S_3 is shared by task t_1 and t_2. The size of slack slots is set to the largest execution time of all tasks sharing the slot.

For the *TT-FS* Schedules, special care needs to be taken on the message placement step. This is because normal slots might be delayed due to out-of-order execution of slack slots. The message scheduling has to take this issue into account and make sure that faults occurred on one processor are masked to other processors even if the task is delayed. This can be explained using an example. Assume t_2 in Fig. 7b is going to send some message to other processors and we want to mask a single fault that occurs on processor 1. Figure 7c shows the execution scenario that t_2 encounters a fault. Based on the idea of transparent recovery, the message m originated from t_2 should be placed no sooner than the second instance of t_2 to guarantee a correct output message. However, the message should be scheduled at an even later time (m' in the figure), since the worst-case scenario happens when t_1 encounters a fault as shown in Fig. 7d. In other words, if the message is placed at m, a single fault on t_1 cannot be tolerated and the system reliability decreases. Thus, our scheduler always analyzes the worst-case scenario and places the messages accordingly.

5.3 Encoding of design using imperfect fault detectors

To take imperfect fault detection into account, the coding of the problem needs to be extended. For each tasks instance, an additional attribute is needed to model the selection of the fault detector. Each task instance is now encoded as a pair (p, d), where p is the processor that will execute the instance and d is the index of the fault detector to implement. Figure 8 illustrates an example, in which tasks 1 and 3 are replicated 2 times and task 2 is replicated 3 times. The lower part of the figure depicts the corresponding schedule that the chromosome represents. The reconstruction of the schedule from the chromosome can be done using a simple greedy heuristic. The resulted schedules are always strict schedules

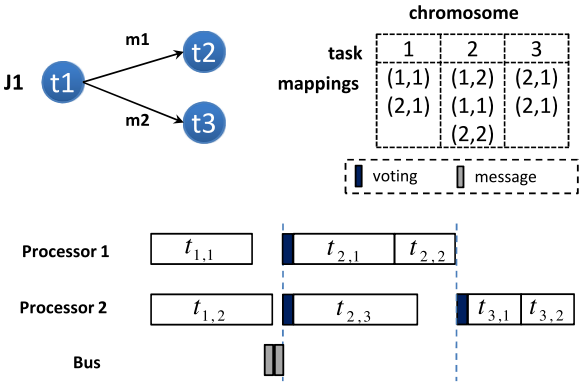


Fig. 8 Example of encoding scheme

[4, 12] according to the proposed voting setup.[9] We consider all tasks in the application in topological order. For each task, the replicas specified in the chromosome are instantiated and scheduled greedily at the earliest possible time. Output messages are scheduled at the end of execution. If the current task has data dependency on previous tasks, a voter is inserted. The failure rate of the voter is added to the failure rate of the current task. Other relevant details of the optimization approach are presented in [17].

5.4 Tolerating permanent faults using virtual mapping

The analysis and optimization approach presented so far focuses only on transient faults. However, many safety-related applications also have requirement on permanent fault tolerance. 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 straightforward way to tolerate a single permanent fault is to ensure that each task at least one spatial replication. This can be done by adding a constraint on the chromosomes. The drawback is however low resource efficiency and high hardware cost due to large amount of spatial redundancy.

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 [25] proposes to compute the task re-mappings statically offline and store them in tables. The pre-computed configurations are applied at runtime if a permanent fault is detected. We adopt a similar approach and synthesize static schedules 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 Fig. 10, where we are going to migrate the task X to one of the possible locations S_1 to S_3. The tasks 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

[9] If two tasks have data dependency, all instances of the source task send the output to the voter (therefore execute before the voting) and all instances of the target task read the voter result as input (therefore execute after the voting). This guarantees the schedule to be strict.

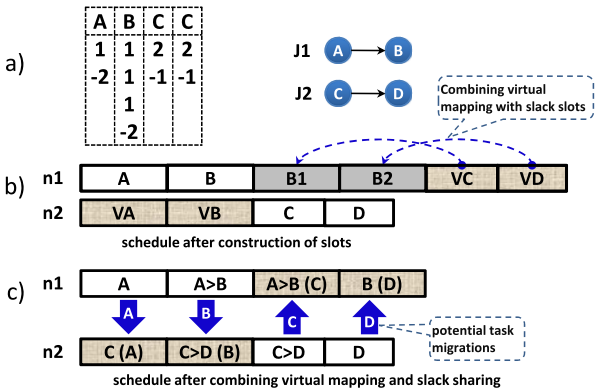


Fig. 9 Example of virtual mapping

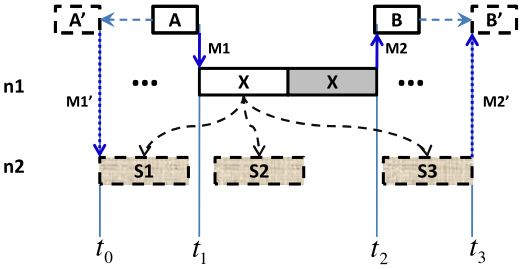


Fig. 10 Influence of data dependency on task migration

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. An indication from this example is that, while building schedules to tolerate transient faults, we should keep permanent faults in mind and try to make such low-overhead migrations feasible. 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 Fig. 9a 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 Fig. 9b). This slot is the place where A will be migrated to. Note that the virtual mapping slots are also scheduled using the same heuristic presented in Sect. 5 so that data dependencies are also regarded. This is essential to achieve a low-overhead task migration as shown in Fig. 10. 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 Fig. 9c, 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 (Fig. 9c).

Note that we assume a use scenario that task migration is only considered as an emergency response of the system when permanent faults occur. The goal is to guarantees continuous service of the system, possibly with degraded quality due to lack of resources, before a maintenance (e.g., replace the failed hardware) can be carried out. In this case, we do not consider the reliability concerning transient faults after the migration.

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 to allow other tasks to acquire the slot in normal execution. 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 can be achieved.

6 Experimental results

We implemented the analysis and optimization algorithm in JAVA using the opt4j library [28]. 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 1×10^{-5} and 1×10^{-7} (the failure probabilities are in the typical value range for soft error rates [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 (SFP) per hour in logarithmic scale in the experiments, i.e., the lower the value, the higher the reliability is. As the applications we use two sets of Task Graphs (TGs). The first is a set of random TGs with 5 to 15 tasks generated synthetically using TGFF.[10] The execution time of each task on the RISC/DSP is generated randomly between 100 and 1000. The second is an *mpeg2* decoder example from [47] that consists of 13 tasks.

The goal of the first set of experiments is to evaluate the accuracy and runtime of the reliability analysis. Several instances of the BTA with different approximation factors are evaluated. The approximation is achieved by bounding the maximum considered faults (MF) as introduced in Sect. 4.1. Figure 11 presents the results averaged over 100 test runs. The tests are carried out on random schedules from 10 TGs. As it can be seen, the execution time increases rapidly as MF becomes larger. The run time of BTA with $MF = 5$ is around $16x$ higher than the case with $MF = 3$. As for the accuracy, all analysis with MF larger than 1 bounds the average relative error to less than 10 %. The BTA with $MF = 3$ achieves a very good tradeoff between runtime (around 3 seconds) and accuracy (99.3 %) and becomes

[10] TGFF http://ziyang.eecs.umich.edu/~dickrp/tgff/.

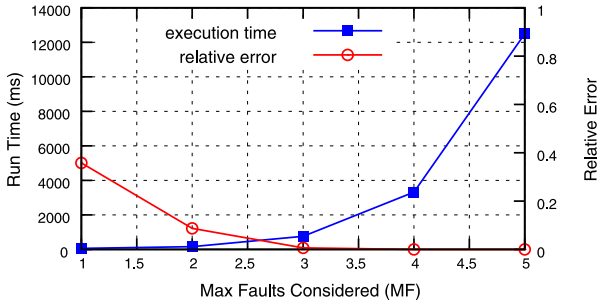


Fig. 11 Evaluation of BTA with approximation

a good choice in practice. Actually, the BTA should be used in most cases with relatively small *MF*. Since the occurrence probabilities of transient faults are typically very low (e.g., at the magnitude of 10^{-5} [3]), fault scenarios with a large number of faults happen very rarely. The scheduler should focus on covering all the fault scenarios with high probabilities instead of tolerating rare cases. For example, if a single-fault scenario with probability 10^{-5} is not tolerable, it becomes the system bottleneck. Even if all other fault scenarios can be tolerated, the maximum achievable reliability is limited to $1–10^{-5}$.

Note that using a lower *MF* value results in more pessimistic result and higher underestimation of system reliability, since a larger number of nodes are eliminated and considered as non-tolerable (see Sect. 4). The solution found using a coarse analysis is safe due to pessimism. However, it may happen that no feasible solution is found even if some exists. The approximation factor should be selected in a way that the accuracy of analysis is sufficient to reach the same accuracy of applications' reliability goal. The accuracy of the analysis can be roughly estimated as follows. Assume the transient fault probability of tasks are at the magnitude of 10^{-5}, the occurrence probabilities of single-fault scenarios are at the magnitude of 10^{-5} and the probability of 2-fault scenarios are at the magnitude of 10^{-10}. If we consider $MF = 2$, the analysis will drop all fault scenario with more than 2 faults, and the reliability is determined by the portion of tolerable single-fault and 2-fault scenarios. Since the BTA accumulates the occurrence probability of tolerable scenario to obtain the system-level reliability, the accuracy of analysis is also 10^{-10}. Hence if the reliability goal is at a higher accuracy, e.g., 10^{-11}, a high *MF* should be considered. Also, using different approximation factors in the design process might be helpful. E.g., the coarse analysis can be used to obtain some fast results and more accurate analysis can be used for final decision making and verification.

6.1 Design space exploration case study

An important step during embedded system design is design space exploration. The designer has to address problems such as what is the amount and the type of PEs needed to meet all application requirements. To illustrate this step, we consider the *mpeg2* application and run the optimization procedure with several platform configurations consisting of 2 to 6 PEs. The execution time of tasks is specified according to [47]. The deadline of the application is set to two times the critical path of the TG to allow some slack for reliability improvement. 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:

$$penalty(S) = \begin{cases} -1 & \text{iff } l \leq d \\ l - d & \text{otherwise} \end{cases} \tag{10}$$

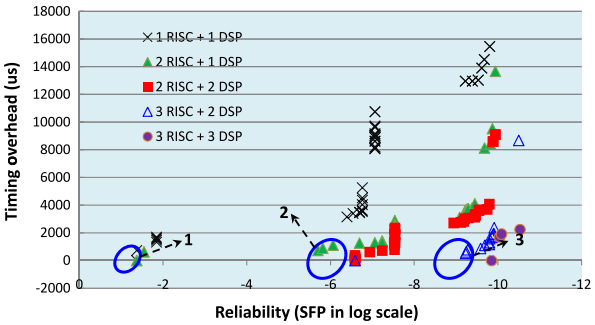


Fig. 12 Pareto optimal solutions under different platform configurations

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 12 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.

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 Fig. 12. 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}. Those results are useful for the designer to make decisions in the early design phase. In [16], more details of this case study is presented.

6.2 Comparison of *TT-SP* and *TT-FS*

Qualitative comparison The *TT-SP* and *TT-FS* schemes can be compared in several aspects:

- Resource efficiency. Both *TT-SP* and *TT-FS* allow sharing of time slots by multiple tasks and are therefore more efficient than traditional techniques with dedicated redundancy for each task, e.g., [12]. For example, the schedule in Fig. 1a and 1e are able to tolerate a single fault of any of the tasks t_1, t_2, t_3. Without using the slot-sharing technique, we would have to replicate all three tasks once, in order to achieve the same level of fault-tolerance. However, much more resources are needed in this case. If we compare *TT-SP* and *TT-FS*, the later achieves generally higher resource efficiency. This is because only the slack slots are shared and must have their sizes set to the largest WCET of all tasks assigned to them. Again in same example, the schedule in Fig. 1a has the length $|t_1| + \max(|t_1| + |t_2|) + \max(|t_2| + |t_3|) + |t_3|$, whereas the schedule in Fig. 1e has the length $|t_2| + |t_2| + |t_3| + \max(|t_1| + |t_2| + |t_3|)$. It can be easily verified that the length of Fig. 1e is no larger than that of Fig. 1a. Nevertheless, the *TT-SP* scheme can also outperform *TT-FS* in certain circumstances (see the discussion of Fig. 14).
- Predictability. The *TT-SP* scheme exhibits higher predictability in sense that the start time of all slots are fixed and known. In Fig. 1, the scheduling points, where the runtime scheduler has to be called, are marked. As it can be seen, the scheduling points of *TT-SP* is fixed

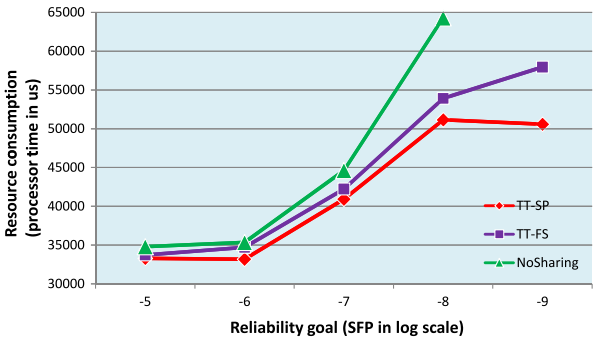


Fig. 13 Comparing *TT-SP* and *TT-FS* using *mpeg2*

in time and do not depends on the occurred faults. In contrast, the scheduling points vary for different fault scenario in case of *TT-FS*.

– Scheduler Complexity and Overhead. The implementation of a *TT-SP* scheduler is straightforward. At each scheduling point, we just pick the pending task that has the highest priority for execution. The implementation is more complicated for *TT-FS*. Depending on the faults that occurred previously, the schedule has to be adapted at runtime and the new scheduling points need to be determined. In Fig. 1f for example, the slot S_5 and S_6 have to be delayed due to the re-execution of t_1 using S_7. In this case, the complexity in scheduler implementation is higher, resulting in generally higher scheduling overhead.

Experimental comparison To evaluate the performance of *TT-SP* and *TT-FS*, we extend the optimizer with an additional optimization objective, namely resource consumption. We vary the reliability goal of the *mpeg2* application from SFP 10^{-5} up to SFP 10^{-9} and check the minimum amount of resources to achieve the desired reliability level. The timing constraints remain the same. The $2RISC + 2DSP$ platform is considered as the target architecture. For the fitness of timing and reliability, the same technique as in Eq. (10) is applied, i.e., the penalty is set to -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.

Three approaches are compared, namely the *TT-SP* scheme, the *TT-FS* scheme and the traditional approach without slack sharing (*NoSharing*). *NoSharing* is a similar approach as the existing work [12]. However, the result is obtained using our optimization framework with three objectives instead of the bicriteria heuristic proposed in [12]. Figure 13 compares the minimum resources needed to meet both timing and reliability requirements. Clearly, *TT-SP* and *TT-FS* out-perform *NoSharing* significantly by allowing recovery slack to be shared by multiple tasks. As the reliability requirement becomes higher, more redundancy needs to be added and the benefit of slack sharing also increases. When the SFP goal is 10^{-8}, 25.7 % more resources are consumed by *NoSharing*. Moreover, *NoSharing* fails to provide any feasible solution[11] when the SFP goal is set to 10^{-9}. What is a bit unexpected is that *TT-SP* exhibits better performance than *TT-FS* for the *mpeg2* application. After detailed analysis of the schedules, we find out that this is because the tasks of *mpeg2* have a large variation on failure probabilities. This implies that more replicas should be scheduled for tasks with high failure rates. However, the slack slots in *TT-FS* implicitly treat all tasks in the same way. We explain this issue using a simple example depicted in Fig. 14.

[11] When the approach fails to find a solution, the corresponding point is missing in the figure.

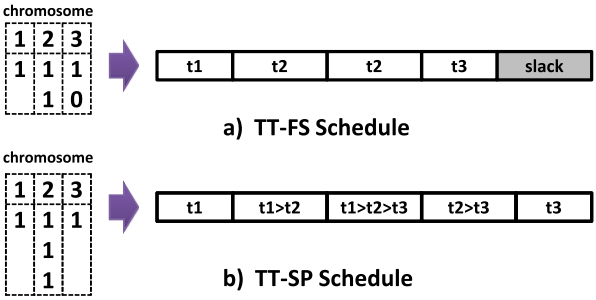


Fig. 14 An example for comparing *TT-SP* and *TT-FS*

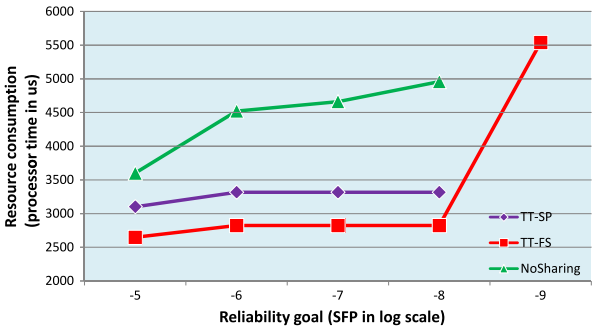


Fig. 15 Comparing *TT-SP* and *TT-FS* using random TG

Consider that three tasks t_1 to t_3 are allocated on processor 1 and task t_2 has a higher failure rate. *TT-FS* allocates one shared slack slot for all three tasks and an extra replica for t_2 (Fig. 14a). In this way, the schedule tolerates any single fault on t_1 to t_3 and also two consecutive faults on t_2. Since t_2 has the largest execution time, the size of the slack slot is set to $|t_2|$. The overall length of schedule is then $|t_1| + 3 * |t_2| + |t_3|$. Figure 14b shows the optimal *TT-SP* schedule that utilizes the same amount of resources. Three replicas are allocated for t_2 and then shared with t_1 and t_3. Thanks to the sharing of three slots, the schedule in Fig. 14b tolerates a larger set of fault scenarios (actually, it tolerates two faults on any of the tasks) and therefore achieves higher system-level reliability. In other word, the *TT-SP* schedule shows better performance in this scenario.

We did the same experiment on a randomly generated Task Graph (TG), whose tasks have small variation on failure rates. Figure 15 summarizes the results. As it can be seen, the *TT-FS* approach consumes less resources to reach the same reliability level. For this example, the resource saving archived by slack sharing is larger than the case of *mpeg2*, e.g., 75.7 % for SFP 10^{-8}. This is because the deadline of the TG is relatively loose and the possibility to use re-execution slots is higher.

6.3 The case with permanent faults

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.

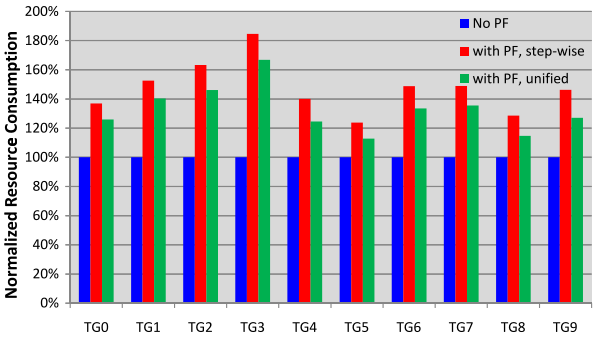


Fig. 16 Performance comparison of step-wise and unified approaches

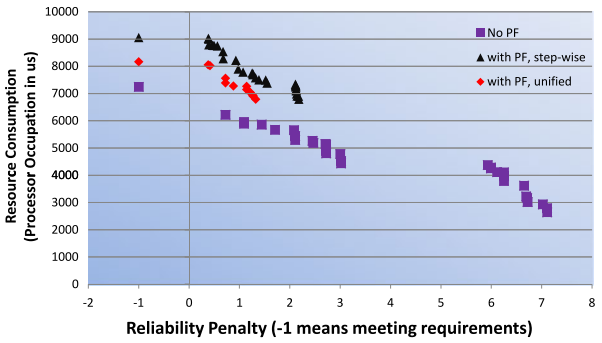


Fig. 17 Example of Pareto optimal results

– 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 a case in which only requirements on transient fault tolerance are added (No-PF). We are interested in how much overhead is needed to fulfill the additional requirement concerning permanent faults. We again use three optimization objectives, namely schedule length, reliability and resource utilization. We assume that it is required to tolerate a single defect on any of the processors. Figure 16 compares the solution that meets both timing and reliability requirements with minimum resources tested on 10 random TGs. 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 17 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 are 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.

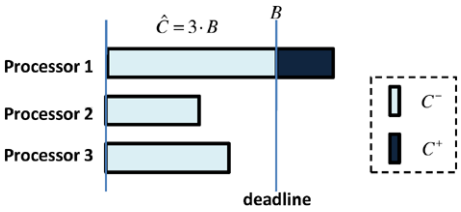


Fig. 18 The resource consumption objective

6.4 The case with imperfect fault detection

We use the same mpeg2 decoder example to evaluate the extended approach that considers imperfect fault detection. Two approaches are compared: (1) the proposed approach that explores the optimal utilization of fault detectors (*ExploreDetector*); (2) existing approaches that utilize the perfect fault detector only. We use the MOEA optimizer to compute the Pareto optimal results considering three objectives. The first two are reliability objectives, one for the amount of detectable faults (*DUF*) and one for the amount of undetectable faults (*SDC*). The metric that we use in this experiment is Failure In Time (FIT). One unit FIT specifies one failure in a billion hours. The conversion from failure probabilities computed in Sect. 4 to FIT is straightforward. In the third objective, we intend to encode the design goal of minimizing resource consumption while meeting the deadline. The objective function is defined as follows. Let B be the deadline of the application and N be the number of processors available in the execution platform. The available time budget within the deadline is $\hat{C} = NB$. For a given schedule S, we use C^- to denote the fraction of resource consumption that is within the deadline and C^+ to denote the part above the deadline. Figure 18 depicts an example. The objective function is defined as:

$$penalty = \begin{cases} C & \text{iff } C^+ = 0 \\ \hat{C} + C^+ & \text{otherwise} \end{cases} \tag{11}$$

By constructing the objective function as above, each schedule that violates the deadline ($C^+ > 0$) has a higher penalty value than any schedule that meets the deadline. For two schedules that meet the deadline, the one that has less resource consumption will be preferred. Clearly, all three objectives are to be minimized.

Figure 19 shows the results using an example platform that consists of 2 *RISC*s and 2 *DSP*s. The dots in the figure show the results projected into a 2D plane, with the vertical axis being the FIT of *SDC* and the horizontal axis being the FIT of *DUF*. The Pareto front considering only the two reliability objectives is marked using a solid line. The triangle symbols in Fig. 19 show the results of the *PerfectDetector* approach. Clearly, the solutions found by *ExploreDetector* is of much higher quality than those found by *PerfectDetector*. The gap in terms of FIT is several orders of magnitude in this experiment. Moreover, the *ExploreDetector* approach provides a much wider spectrum of solutions, allowing the designers to carefully evaluate the tradeoff between the two classes of faults and select the implementation that fits the application requirements.

We mark some representative implementation alternatives in Fig. 19. *A* is the best solution in terms of reliability found by the *PerfectDetector* approach; *B* is a solution found by *ExploreDetector* which is close to and dominates *A*; *C* to *F* belong to the Pareto optimal solutions found by *ExploreDetector*. Table 1 compares these implementations in several aspects, e.g., the average number of replications for each task, the average fault detection coverage over all task instances and the resource consumption. It can be seen that implementation *A* has the lowest number of replications, since a lot of resources are already consumed

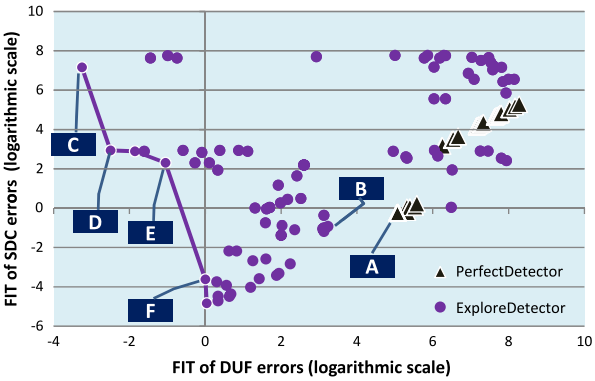


Fig. 19 2D Projection of Optimization Results

Table 1 Comparing representative implementation alternatives

Solution	DUF FIT (log.)	SDC FIT (log.)	Avg. rep.	Avg. cov. (%)	Resource (time unit)
A	5.06	−0.23	3.25	99.9	114.0
B	3.24	−0.93	3.67	63.0	74.2
C	−3.25	7.15	3.50	84.4	55.9
D	−2.49	2.93	3.83	74.9	65.5
E	−1.04	2.30	3.92	83.0	150.6
F	0	−3.62	3.92	89.3	189.5

Table 2 Execution time of optimization approach

Application (num. tasks)	200 round	500 round	1000 round	1500 round
mpeg2 (13 tasks)	29.0	76.4	120.8	198.3
TG1 (50 tasks)	78.3	179.0	395.1	583.0
TG2 (100 tasks)	195.9	442.2	777.0	1692.0

by fault detection. The solution *B* has higher quality than *A* concerning all three objectives. Using fault detectors with average coverage of 63 %, it achieves much higher reliability than *A* and saves 35 % resources. The implementation *F* achieves higher reliability than *A* as well. By spending 65 % more resources, it reduces the FIT of *DUF* by 5 orders of magnitude and the FIT of *SDC* by more than 3 orders of magnitude. In [17], extended discussion of this case study is presented.

To evaluate the scalability of the approach, we measure the run time (in seconds) of the flow on a Windows machine with 3 GHz CPU. The MOEA is configured to run for 200, 500, 1000 and 1500 rounds. Table 2 presents the results. For a small TG (e.g., mpeg2), the analysis and optimization procedure takes only a few minutes to execute for 1500 iterations. As expected, the execution time grows linearly with the number of iterations. It is also worth mentioning that the execution time also increases roughly linearly with the size of TG. This is because the reliability analysis, as most computational intensive operation, has

linear complexity in the number of tasks. For a syntactic TG[12] with 100 tasks, the 1000-iteration EA takes about 13 minutes. In general, the runtime is acceptable for an off-line design space exploration procedure.

7 Conclusion and outlook

The paper presents a framework for reliability-aware mapping and scheduling of real-time tasks onto MPSoC based platforms. Spatial and temporal redundancy is utilized to meet the reliability goal and we focus on finding the optimal configuration of redundancy. A reliability analysis is proposed to evaluate the system-level reliability in the presence of active redundancy. Guided by the analysis results, an optimization approach is used for automated design space exploration. The proposed approach takes application-specific constraints into account and provides recommended design alternatives, including key design parameters such as task-to-processor mapping, task/message scheduling, and which/how fault-tolerant techniques should be utilized. We also integrate consideration of permanent faults and supports fault-tolerant system design using imperfect fault detectors.

For the next step, we are interested in integrating the framework into a model-based development environment. The optimization process will then take a set of input models and produce a set of transformed models that fulfill certain reliability requirements. Moreover, heuristics algorithms with higher scalability are desirable for fault-tolerant task mapping and scheduling. Another extension of the current work could be to consider the impact of fault-tolerance mechanisms on energy consumption.

Acknowledgements This work has been supported in part by the European research project ACROSS under the Grant Agreement ARTEMIS-2009-1-100208 and the German BMBF projects ECU (grant number: 13N11936) and Car2X(grant number: 13N11933).

References

1. ACROSS Project. http://www.across-project.eu/
2. Axer P, Sebastian M, Ernst R (2011) Reliability analysis for mpsocs with mixed-critical, hard real-time constraints. In: International conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 149–158
3. Baumann R (2002) The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction. In: International electron devices meeting (IEDM)
4. Benoit A, Canon LC, Jeannot E, Robert Y (2011) Reliability of task graph schedules with transient and fail-stop failures: complexity and algorithms. J Sched 15(5):615–627
5. Birolini A (2004) Reliability engineering—theory and practice. Springer, Berlin
6. Borkar S (2005) Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. Micro, IEEE 25(6):10–16
7. Degraeve R, Groeseneken G, Bellens R, Depas M, Maes H (1995) A consistent model for the thickness dependence of intrinsic breakdown in ultra-thin oxides. In: Electron devices meeting
8. Feldmann R, Haubelt C, Monien B, Teich J (2003) Fault tolerance analysis of distributed reconfigurable systems using sat-based techniques. In: International conference on field programmable logic and applications
9. Fohler G (1997) Adaptive fault-tolerance with statically scheduled real-time systems. In: Proceedings ninth euromicro workshop on real-time systems, 1997
10. Gall M, Capasso C, Jawarani D, Hernandez R, Kawasaki H, Ho PS (2001) Statistical analysis of early failures in electromigration. J Appl Phys 8(2):732–740

[12]Generated using TGFF http://ziyang.eecs.umich.edu/~dickrp/tgff/.

11. GENESYS: http://www.genesys-platform.eu/
12. Girault A, Kalla H (2009) A novel bicriteria scheduling heuristics providing a guaranteed global system failure rate. IEEE Trans Dependable Secure Comput 6(4):241–254
13. Glaß M, Lukasiewycz M, Reimann F, Haubelt C, Teich J (2008) Symbolic Reliability Analysis and optimization of ECU Networks. In: Design, automation and test in Europe (DATE), pp 158–163
14. Glaß M, Lukasiewycz M, Streichert T, Haubelt C, Teich J (2007) Reliability-aware system synthesis. In: Design, automation and test in Europe (DATE), pp 409–414
15. Hartman AS, Thomas DE, Meyer BH (2010) A case for lifetime-aware task mapping in embedded chip multiprocessors. In: International conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 145–154
16. Huang J, Blech JO, Raabe A, Buckl C, Knoll A (2011) Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems. In: International conference on hardware-software codesign and system synthesis (CODES+ISSS), Taipei, Taiwan, pp 247–256
17. Huang J, Huang K, Raabe A, Buckl C, Knoll A (2012) Towards fault-tolerant embedded systems with imperfect fault detection. In: 49th design automation conference (DAC), San Francisco, CA, pp 188–196
18. Huang L, Yuan F, Xu Q (2009) Lifetime reliability-aware task allocation and scheduling for mpsoc platforms. In: Proceedings of the conference on design, automation and test in Europe (DATE), pp 51–56
19. Izosimov V, Polian I, Pop P, Eles P, Peng Z (2009) Analysis and optimization of fault-tolerant embedded systems with hardened processors. In: Design, automation and test in Europe (DATE), pp 682–687
20. Izosimov V, Pop P, Eles P, Peng Z (2005) Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems. In: Design, automation and test in Europe (DATE), pp 864–869
21. Izosimov V, Pop P, Eles P, Peng Z (2006) Synthesis of fault-tolerant schedules with transparency/performance trade-offs for distributed embedded systems. In: Design, automation and test in Europe (DATE), pp 706–711
22. Kandasamy N, Hayes J, Murray B (2003) Transparent recovery from intermittent faults in time-triggered distributed systems. IEEE Trans Comput 52(2):113–125
23. LaFrieda C, Ipek E, Martinez J, Manohar R (2007) Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In: International conference on dependable systems and networks (DSN), pp 317–326
24. Lala J, Harper R (1994) Architectural principles for safety-critical real-time applications. Proc IEEE 82(1):25–40
25. Lee C, Kim H, Park Hw, Kim S, Oh H, Ha S (2010) A task remapping technique for reliable multi-core embedded systems. In: International conference on Hardware/Software codesign and system synthesis (CODES+ISSS), pp 307–316
26. Lifa A, Eles P, Peng Z, Izosimov V (2010) Hardware/software optimization of error detection implementation for real-time embedded systems. In: International conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 41–50
27. Lukasiewycz M, Glaß M, Haubelt C, Teich J (2007) Sat-decoding in evolutionary algorithms for discrete constrained optimization problems. In: IEEE Congress on evolutionary computation
28. Lukasiewycz M, Glaß M, Reimann F, Teich J (2011) Opt4j: a modular framework for meta-heuristic optimization. In: Proceedings of the 13th annual conference on genetic and evolutionary computation (GECCO), pp 1723–1730
29. Lyle G, Chen S, Pattabiraman K, Kalbarczyk Z, Iyer R (2010) An end-to-end approach for the automatic derivation of application-aware error detectors. In: IEEE/IFIP international conference on dependable systems networks (DSN), pp 584–589
30. Meyer BH, Hartman AS, Thomas DE (2010) Cost-effective slack allocation for lifetime improvement in noc-based mpsocs. In: Proceedings of the conference on design, automation and test in Europe (DATE), pp 1596–1601
31. Mitra S, Saxena N, McCluskey E (2000) Common-mode failures in redundant vlsi systems: a survey. IEEE Trans Reliab 49(3):285–295
32. Mitra S, Saxena N, McCluskey E (2002) A design diversity metric and analysis of redundant systems. IEEE Trans Comput 51(5):498–510
33. Mukherjee SS, Weaver C, Emer J, Reinhardt SK, Austin T (2003) A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: Proceedings of IEEE/ACM international symposium on microarchitecture (MICRO)
34. Obermaisser R, Hoftberger O (2011) Fault containment in a reconfigurable multi-processor system-on-a-chip. In: Industrial electronics (ISIE), 2011 IEEE international symposium on
35. Pattabiraman K, Kalbarczyk Z, Iyer R (2011) Automated derivation of application-aware error detectors using static analysis: the trusted illiac approach. IEEE Trans Dependable Secure Comput 8(1):44–57

36. Pinello C, Carloni LP, Sangiovanni-Vincentelli AL (2004) Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications. In: Design, automation and test in Europe (DATE), pp 1164–1169
37. Pop P, Izosimov V, Eles P, Peng Z (2009) Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication. IEEE Trans Very Large Scale Integr (VLSI) Syst 17(3):389–402
38. Pop P, Poulsen KH, Izosimov V, Eles P (2007) Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems. In: International conference on Hardware/Software codesign and system synthesis (CODES+ISSS), pp 233–238
39. Pradhan DK (1996) Fault-tolerant computer system design
40. Qin X, Jiang H (2006) A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems. Parallel Comput 32(5–6):331–356
41. Reimann F, Glaß M, Lukasiewycz M, Haubelt C, Keinert J, Teich J (2008) Symbolic voter placement for dependability-aware system synthesis. In: International conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 237–242
42. Saraswat PK, Pop P, Madsen J (2010) Task mapping and bandwidth reservation for mixed hard/soft fault-tolerant embedded systems. In: IEEE real-time and embedded technology and applications symposium (RTAS), pp 89–98
43. Schiffel U, Schmitt A, Süßkraut M, Fetzer C (2010) Software-implemented hardware error detection: costs and gains. In: Third international conference on dependability
44. Shatz SM, Wang JP (1989) Models and algorithms for reliability-oriented task-allocation in redundant distributed-computer systems. IEEE Trans Reliab 38:16–27
45. Srinivasan J, Adve S, Bose P, Rivers J (2004) The impact of technology scaling on lifetime reliability. In: IEEE/IFIP international conference on dependable systems networks (DSN), pp 177–186
46. Storey N (1996) Safety-critical computer systems. Addison Wesley/Longman, Reading
47. Thiele L, Bacivarov I, Haid W, Huang K (2007) Mapping applications to tiled multiprocessor embedded systems. In: International conference on application of concurrency to system design (ACSD), pp 29–40
48. Xiang Y, Chantem T, Dick RP, Hu XS, Shang L (2010) System-level reliability modeling for mpsocs. In: International conference on Hardware/Software codesign and system synthesis (CODES+ISSS), pp 297–306
49. Xie Y, Li L, Kandemir M, Vijaykrishnan N, Irwin M (2004) Reliability-aware co-synthesis for embedded systems. In: IEEE international conference on application-specific systems, architectures and processors (ASAP), pp 41–50
50. Yang C, Orailoglu A (2007) Predictable execution adaptivity through embedding dynamic reconfigurability into static mpsoc schedules. In: International conference on Hardware/Software codesign and system synthesis (CODES+ISSS), pp 15–20
51. Yang C, Orailoglu A (2009) Towards no-cost adaptive mpsoc static schedules through exploitation of logical-to-physical core mapping latitude. In: Design, automation and test in Europe (DATE), pp 63–68
52. Zhao B, Aydin H, Zhu D (2009) Enhanced reliability-aware power management through shared recovery technique. In: International conference on computer-aided design (ICCAD), pp 63–70
53. Zhu C, Gu ZP, Dick RP, Shang L (2007) Reliable multiprocessor system-on-chip synthesis. In: International conference on Hardware/Software codesign and system synthesis (CODES+ISSS), pp 239–244
54. Zhu D, Aydin H (2006) Energy management for real-time embedded systems with reliability requirements. In: IEEE/ACM international conference on computer-aided design (ICCAD), pp 528–534
55. Zhu D, Aydin H (2009) Reliability-aware energy management for periodic real-time tasks. IEEE Trans Comput 99:1382–1397