Towards Fault-Tolerant Embedded Systems with Imperfect Fault Detection

Jia Huang
fortiss GmbH, Germany
huang@fortiss.org

Kai Huang
fortiss GmbH, Germany
khuang@fortiss.org

Andreas Raabe
fortiss GmbH, Germany
raabe@fortiss.org

Christian Buckl
fortiss GmbH, Germany
buckl@fortiss.org

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

ABSTRACT

Many state-of-the-art approaches on fault-tolerant system design make the simplifying assumption that all faults are detected within a certain time interval. However, based on a detailed experimental analysis, we observe that perfect fault detection is not only an impractical assumption but even if implementable also a suboptimal design decision. This paper presents an approach that takes imperfect fault detection into account. Novel analysis and optimization techniques are developed, which distinguish detectable and undetectable faults in the overall workflow. Besides synthesizing the task schedules, our approach also decides which of the available fault detectors is selected for each task instance. Experimental results show that our approach finds solutions with several orders of magnitude higher reliability than current approaches.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Reliability

Keywords

Embedded Systems, Reliability, Design Optimization

1. INTRODUCTION

To meet the reliability requirements of safety-critical embedded systems, fault-tolerance techniques such as active redundancy are widely adopted. Active redundancy can be implemented in both the space and the time domains. In the space domain, critical components can be replicated into multiple copies to enhance the error resilience. In the time domain, software tasks can be selectively re-executed. Fault-tolerant system design using active redundancy is a very challenging task that involves solving two major problems, namely finding the optimal utilization of temporal and/or spatial redundancy and the scheduling of tasks (including replicas) under timing constraints. Over the past decades, a lot of research efforts have been devoted to this field. A review of related work is presented in Appendix A.

To cope with the high problem complexity, many state-of-the-art studies make simplifying assumption on the fault models and modes. Perfect fail-silent behavior is one assumption that is often used in literature. It is assumed that all faults are detected within a certain time interval and the fault-detection overhead is contained in the tasks' Worst-Case Execution Times (WCETs), e.g., in fault-tolerant task scheduling [8, 15, 4, 17, 6, 5], in reliability-aware energy management [16, 20, 22] and in error-aware system design [9, 10]. With this assumption, each task will produce either a correct output or no output at all. Although fail-silence is a highly desirable property, it is difficult to implement in practice. The prerequisite is the existence of a perfect fault detector that achieves 100% coverage under the given fault hypothesis.

The simplifying assumption of perfect fault detection is problematic. On the one hand, a perfect detector might not exist or is difficult to implement, making the algorithms developed under this assumption less useful in practice. On the other hand, even if implementable, perfect detectors typically come with high resource and timing overheads. In recent work [12, 18] 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 [12]). Hence, approaches under this assumption are very pessimistic, as the most expensive fault detector is selected for every task.

This problem can be viewed from a slightly different angle: choosing to implement the perfect fault detector is not only an **assumption** but also an important **design decision**. While making this assumption, all design alternatives with partial fault detectors are ignored without any justification. For example, when active redundancy is concerned, no analysis is performed to find out if it is more efficient to spend the available resources on applying better fault detection or a higher number of replications. Actually, our experimental results show that the answer is highly application and architecture dependent. Detailed discussions are presented in Section 3 and Appendix B.

In this work, we put special emphasis on the effect of imperfect fault detection and present the first approach (to the best of our knowledge) to synthesize fault-tolerant schedules with reliability guarantee using imperfect fault detectors. Besides computing the task schedule and utilizing redundancy, our approach also decides which of the available fault detectors should be selected for each task. The main contributions of this papers are: 1) an experimental analysis on the impact of imperfect fault detection on system-level reliability; 2) a reliability analysis approach that computes the probabilities of both detectable and undetectable faults in the presence of redundancy; 3) an Multi-Objective Evolutionary Algorithm (MOEA) based approach for reliability-aware design optimization.

The remainder of the paper is organized as follows. The system models is first introduced in Section 2. The next Section discusses a motivating example. The main contribution of this work, namely the reliability analysis and optimization approaches are presented in Section 4 and Section 5, respectively. Experimental results are provided in Section 6. Section 7 concludes the paper.

2. PRELIMINARIES

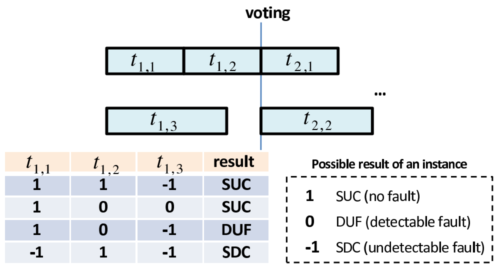
2.1 Fault Model and Fault Tolerance

This paper focuses on tolerating transient faults and adopts the classical fault model that occurrence of transient faults follows a Poisson distribution with a constant failure rate λ. The consideration of permanent faults could be added, e.g. using the technique proposed in [5]. A task may be replicated into multiple copies (or instances) to implement temporal/spatial redundancy. The set of N replicas for a task t_i is denoted as $R(t_i) = \{t_{i,1}, ..., t_{i,N}\}$. For a specific instance, software fault detectors can be implemented. A software fault detector typically transforms the original program into an instrumented version, adding the capability to detect transient faults that occur at runtime of the program. The arithmetic codes [18] and critical variable technique [13] are examples of this kind. We assume that each task instance tries to implement the fail-silent behavior, i.e., as long as the fault detector[1] reports a fault, this specific task instance will not produce any output. This behavior is desirable since the correct outputs from other instances will not be polluted.

As discussed in section 1, fault detectors are typically imperfect in reality. We characterize a fault detector implementation as a pair $d = \{c, o\}$, where c is the fault detection *coverage* in percentage and o is the timing overhead for fault detection. The overhead is defined in percentage with respect to the stand-alone WCET of the task. In this way, a task t_i that implements the fault detector indexed k has the WCET $w_i(1 + o_k)$. We assume that a library of implementable fault detectors are available at design time for each task (denoted as D_i for task t_i).

We assume a voting mechanism with majority voting is

[1]For simplicity, the term fault detector used in the rest of the paper is meant to be software fault detectors unless mentioned otherwise.



$t_{1,1}$	$t_{1,2}$	$t_{1,3}$	result
1	1	-1	SUC
1	0	0	SUC
1	0	-1	DUF
-1	1	-1	SDC

Figure 1: Example Fault Scenario

implemented if redundancy is available. The voter collects results from all instances and produces a single output for the successor tasks. The qualitative execution results of these replicas (i.e. if they deliver a correct output or not) are described by a *fault scenario*. A fault scenario is a vector $x = \{x_1, ..., x_N\}$, which contains a variable $x_l \in \{1, 0, -1\}$ for each task instance, where x_l is 1 if $t_{i,l}$ produces a correct output (no fault occurs); x_l is 0 if $t_{i,l}$ fails silent (a fault occurs and is detected) and x_l is -1 if $t_{i,l}$ produces an incorrect output (i.e., a fault occurs and is not detected). The voter generates an output if and only if a dominating result (or a majority) is found.

The overall execution of a task, considering all its instances, could result in the following 3 scenarios: 1) the task executes successfully (SUC): it experiences no fault or only some faults that are later masked 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 (see Section 3).

Figure 1 depicts an example of the voting scenario. If the fault scenario is $x = \{1, 1, -1\}$, the incorrect output of $t_{1,3}$ is masked and the overall result is SUC. In the scenario $x = \{1, 0, 0\}$, 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 $x = \{1, 0, -1\}$, a correct and an incorrect output are sent to the voter. However, the voter cannot identify the correct input since no majority is found. In this case, the voter will generate no output and the overall result is DUF. In the last example scenario $x = \{-1, 1, -1\}$, two incorrect outputs are sent to the voter. Note that the fault scenarios model only the qualitative 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[2].

2.2 System models

We consider applications modeled as directed acyclic Task Graphs (TGs). The vertices $\mathcal{T} = \{t_0, t_1, ..., t_m\}$ of a TG represent a set of *tasks* to be executed and the edges capture data dependencies. The stand-alone WCET of

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

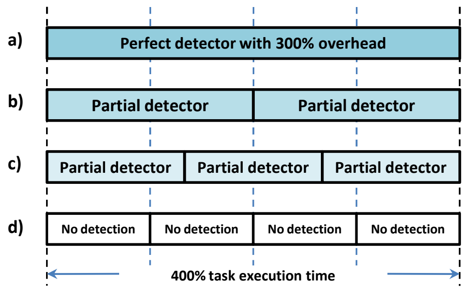


Figure 2: Example Scenario

the task t_i on processor p_j without any fault detection is denoted using $w_{i,j}$. For a instance $t_{i,l} \in R(t_i)$, the processor that will execute $t_{i,l}$ is denoted by $node(t_{i,l})$ and the fault detector ID it implements is denoted by $det(t_{i,l})$. The execution time and fault detection coverage of this instance are therefore $w_i^l = w_{i,node(t_{i,l})}(1 + o_{det(t_{i,l})})$ and $c_{det(t_{i,l})}$, respectively. According to the Poisson fault model, the following formulas could be used to compute the probabilities that an instance is executed successfully without transient faults (denoted by SUC) or experiences detectable/undetectable faults (denoted by DUF/SDC):

$$P_{SUC}(t_{i,l}) = e^{-\lambda_{node(t_{i,l})} w_i^l}$$

$$P_{DUF}(t_{i,l}) = (1 - e^{-\lambda_{node(t_{i,l})} w_i^l}) c_{det(t_{i,l})}$$

$$P_{SDC}(t_{i,l}) = (1 - e^{-\lambda_{node(t_{i,l})} w_i^l})(1 - c_{det(t_{i,l})})$$

Our target architectures are heterogeneous multiprocessor platforms with time-triggered communication, e.g., the GENESYS [3] architecture. The communication between tasks is implemented with messages. The communication can be protected with dedicated techniques (e.g., error correction code) and is therefore assumed as reliable.

3. MOTIVATING EXAMPLE

To understand the impact of imperfect fault detection on the system reliability, we carried out a set of experiments considering two scenarios. In the first one, we fix the amount of redundancy and analyze the influence of detection coverage on the system-level reliability. In the second one, we do it vice-versa, i.e., varying the number of replications with fixed fault detector. The detailed experimental data and discussion is presented in Appendix B. In general, we observe that the selection of fault detector and the utilization of redundancy show a tradeoff. In particular, when the system features only limited amount of resources or the application has tight timing constraints, inappropriate selection of fault detector might disallow certain options for redundancy due to the timing overhead. We explain this issue using the following example.

Consider a simple task running on a single processor system. Similar as the experiments in Appendix B, we reuse the result of [18] and assume that the rate of undetectable faults decreases exponentially with linear fault detection effort. It is further assumed that the perfect fault detection (100% coverage) incurs 300% timing overhead (typical value in [18]). Figure 2a depicts the schedule using the perfect detector. By spending all resources on fault detection, SDCs are completely eliminated. Figure 2b is another possible schedule, in which the task is replicated twice

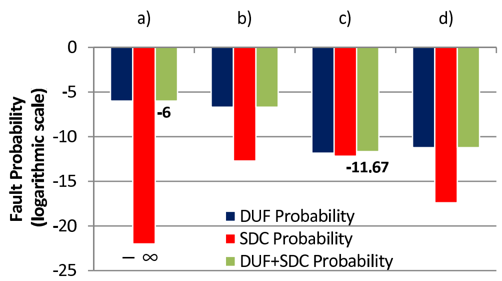


Figure 3: Reliability of the Example Schedules

and the remaining time (200% task execution time in this case) is used to implement two partial fault detectors (each 90% coverage using the 100% detection effort). Figures 2c and 2d show two similar schedules with higher number of replications. When multiple replicas of the same task are available, the results from different instances can be compared to detect or even mask the occurred faults. Figure 3 compares the probability of DUF and SDC for each schedule. For schedule a, although SDCs are avoided completely, the DUF probability is very high, since any transient fault occurring on the single task instance results in a DUF. With imperfect fault detectors (schedule b to d), SDC will not totally disappear but the probability of DUF can be significantly reduced. If both types of faults are considered together, the overall failure probability ($DUF + SDC$) of schedule c is almost six orders of magnitude lower than that of schedule a.

The selection of the best schedule depends on the reliability goal of the application. Many systems have specific requirements concerning DUF and/or SDC. For example, the IBM Power 4 processor-based systems target 10-25 years Mean Time Between Failures (MTBF) for DUF and 1000 years MTBF for SDC [2]. The schedule using perfect fault detectors may not meet the requirements of all applications. Moreover, the criticality of a certain type of faults is application-specific. For systems that require fail-operational behavior, DUFs and SDCs could be equally bad and schedule c is clearly a much better design choice. For other systems, SDCs might be more critical and schedule a or d are more preferable.

From the analysis above, it can be seen that the selection of appropriate fault detectors is critical. The decision has to be made jointly with other design parameters, e.g., task mapping and utilization of redundancy. However, the existing work assuming perfect fault detection prohibits the exploration of design alternatives using partial fault detectors. To tackle this problem, we need 1) a way to evaluate the system quality regarding both DUF and SDC; and 2) an optimization approach for reliability-aware design space exploration. The next two sections present our approach on these issues.

4. RELIABILITY ANALYSIS

Using the voting setup introduced in Section 2, the schedule generated by our algorithm falls into the category of *strict schedules* [4, 1]. Strict schedules obey the rule that if a task t has a data dependency on task t', all replicas of t' should be completed before any replica of t starts. With this restriction, all tasks use exclusively the voter output and the tasks of a TG can be considered independently in the reliability analysis.

For a task t_i, a fault scenario x is *tolerable* if the voter can produce a correct output in the presence of the faults specified in x. This condition can be computed by the following binary function *tolerable()*, which evaluates to *true* if the correct outputs are able to dominate.

$$tolerable(x) = ((\sum_{t_{i,l} \in R(t_i)} x_l) > 0) \tag{1}$$

Where $R(t_i)$ denotes the set of replicas of task t_i and $x_l \in \{1, 0, -1\}$ is the execution result of task $t_{i,l}$. Similarly, the fault scenario x is *silent* if the voter cannot distinguish a dominating result and x is *faulty* if the incorrect results are majority.

$$silent(x) = ((\sum_{t_{i,l} \in R(t_i)} x_l) = 0) \tag{2}$$

$$faulty(x) = ((\sum_{t_{i,l} \in R(t_i)} x_l) < 0) \tag{3}$$

The probability that a task is executed successfully can be computed by summarizing the occurrence probability of all tolerable fault scenarios:

$$P_{SUC}(t_i) = (\sum_{\forall x: tolerable(x)=true} P(t_i, x)) \tag{4}$$

where $Pr(t_i, x)$ is the probability that the fault scenario x happens. As x specifies the qualitative execution result ($SUC/DUF/SDC$) of each instance of task t_i, the probability $Pr(t_i, x)$ can be computed as a product of occurrence probability of each task instance:

$$P(t_i, x) = \prod_{\substack{t_{i,l} \in R(t_i) \\ \wedge x_l = 1}} P_{SUC}(t_{i,l}) \prod_{\substack{t_{i,l} \in R(t_i) \\ \wedge x_l = 0}} P_{DUF}(t_{i,l}) \prod_{\substack{t_{i,l} \in R(t_i) \\ \wedge x_l = -1}} P_{SDC}(t_{i,l})$$

The instance-level probabilities $P_{SUC}(t_{i,l})$, $P_{DUF}(t_{i,l})$ and $P_{SDC}(t_{i,l})$ are computed from the fault model introduced in Section 2.2. In a similar way as in equation 4, the probability that a task results in a fail-silence ($P_{DUF}(t_i)$) or it produces a faulty output ($P_{SDC}(t_i)$) can be computed.

The complete set of tolerable (or silent or faulty) scenarios can be obtained by systematically enumerating all fault scenarios. Since each task instance has three possible results (1,0, or -1), the overall number of combinations is 3^N, where N is the number of replicas. Although this enumeration has exponential complexity, it is still acceptable in practice since the number of replicas for a task is typically very small, e.g., more than 3 replicas for a task is rarely used in practice. The above step is performed for all tasks in the application so that the task-level probabilities P_{SUC}, P_{DUF} and P_{SDC} are obtained. Then, we can proceed with analyzing the reliability of the entire application. Naturally, an application consisting of tasks $\mathcal{T}$ is successful (i.e. SUC) only if all of its tasks are successful:

$$P_{SUC}(\mathcal{T}) = (\prod_{t_i \in \mathcal{T}} P_{SUC}(t_i)) \tag{5}$$

The application is silent (i.e. DUF) if at least one of its tasks is silent, because if any task fails to produce an output, the successor tasks cannot proceed due to data dependency and the entire application has to start over. This probability is denoted by $P_{DUF}(\mathcal{T})$. The application is faulty (i.e. SDC, the corresponding probability is denoted by $P_{SDC}(\mathcal{T})$), if none of its tasks is silent and at least one of its tasks is faulty. Assume t_0 is the first task in $\mathcal{T}$, the application is faulty if t_0 is faulty and the remaining tasks are non-silent (denoted by $P_{\overline{DUF}}(\mathcal{T}\backslash t_0)$), or t_0 is successful and the remaining tasks are faulty.

$$P_{SDC}(\mathcal{T}) = P_{SDC}(t_0) P_{\overline{DUF}}(\mathcal{T}\backslash t_0) + P_{SUC}(t_0) P_{SDC}(\mathcal{T}\backslash t_0)$$

Since $P_{\overline{DUF}}(\mathcal{T}\backslash t_0)$ is the sum of $P_{SUC}(\mathcal{T}\backslash t_0)$ and $P_{SDC}(\mathcal{T}\backslash t_0)$, the above formula can be rewritten as:

$$\begin{aligned} P_{SDC}(\mathcal{T}) = {} & P_{SDC}(t_0) P_{SUC}(\mathcal{T}\backslash t_0) + \\ & (P_{SDC}(t_0) + P_{SUC}(t_0)) P_{SDC}(\mathcal{T}\backslash t_0) \end{aligned} \tag{6}$$

As can be seen, $P_{SDC}(\mathcal{T}\backslash t_0)$ is the only unknown term. Hence, the SDC probability can be computed in a recursively manner. The complexity is linear with the number of tasks. The DUF probability can then be computed by:

$$P_{DUF}(\mathcal{T}) = 1 - P_{SUC}(\mathcal{T}) - P_{SDC}(\mathcal{T}) \tag{7}$$

5. OPTIMIZATION PROCEDURE

After having the reliability analysis, the next step is to develop an optimization approach to search for high-quality designs. We identify two major scenarios that the designers may encounter. In the first one, the system is intended to execute a single application, so the design goal is to maximize the reliability while meeting the deadline. We show that this problem can be transformed into a deadline assignment problem that can be solved using Integer Linear Programming. Appendix C details the transformation and ILP formulation. In the second scenario, multiple applications may be executed on the same platform. We add an additional optimization objective that the resource consumption is to be minimized so that more space can be reserved for future applications. A Multi-Objective Evolutionary Algorithm (MOEA) based optimization procedure is presented for this case.

To use MOEA for optimization, the solutions (in our case the task schedules) need to be encoded into chromosome. The proposed encoding scheme maintains a gene (i, M) for each task, where i is the integer index of the task and M is a list of mapping entries. Each mapping entry encodes an instance of the task and is represented 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 4 illustrates an example, in which task 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. Since we are targeting on generating strict schedules [4, 1], the reconstruction of the schedule from the chromosome can be done using a simple greedy heuristic. We consider all tasks in the TG 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.

We consider three optimization objectives. The first two are the reliability objectives, one for DUF and one for SDC. The metric is Failure In Time (FIT). One unit FIT specifies one failure in a billion hours. The conversion from failure probabilities computed in section 4 to FIT is

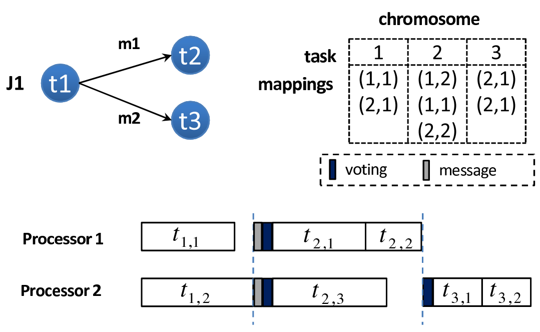


Figure 4: Example of Encoding Scheme

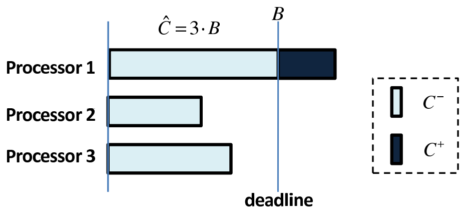


Figure 5: The Resource Consumption Objective

straightforward. In the third objective, we intend to encode the design goal of minimizing resource consumption while meeting the deadline. The resource consumption (denoted by C) is defined as the overall processor time that a schedule occupies. 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 5 depicts an example. The objective function is defined as follows:

$$penalty = \begin{cases} C & \text{iff } C^+ = 0 \\ \hat{C} + C^+ & \text{otherwise} \end{cases} \quad (8)$$

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.

6. EXPERIMENTS

We implement the analysis and optimization algorithms in JAVA using the opt4j library [11]. We assume that the target platform consists of two types of Processing Elements (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}. We again use the exponential model in [18], i.e., the undetectable faults reduce exponentially with linear fault detection effort. Random fault detectors are generated following this law.

The proposed approach is applied on an mpeg2 decoder example taken from [19]. We compare the performance of two approaches: 1) the proposed approach that explores the optimal utilization of fault detectors (*ExploreDetector*); 2) existing approaches that utilize the perfect fault detector[3]

[3]For better visualization in the logarithmic scale, the results we presents are using the detector with 99.9% coverage. If the perfect fault detector is used, the probability of SDC can be reduced to 0, but the probability of DUF remains almost the same.

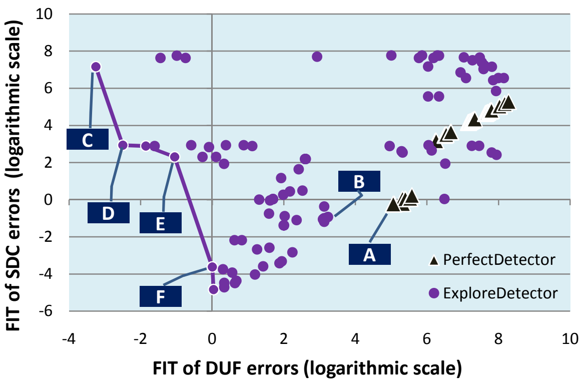


Figure 6: 2D Projection of Optimization Results

for all tasks ($PerfectDetector$). We use the MOEA optimizer to compute the Pareto optimal results considering the three objectives introduced in section 5. Figure 6 shows the results using an example platform that consists of 2 $RISC$s and 2 DSPs. 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 Figure 6 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 Figure 6. 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 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. It is also worth noticing that, since most of the solutions found by $PerfectDetector$ implement 2 replicas, the curve formed by those solutions has similar shape as the curve in Figure 7b.

The optimization results from MOEA can also be viewed from different angles. Extended discussion of this case study is presented in Appendix D. To go one step further, our approach can be used to perform reliability-aware design space exploration (DSE), e.g., to find out the amount and

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 1: Comparing Representative Implementation Alternatives

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

Table 2: Execution Time of Optimization Approach

type of PEs necessary to meet certain reliability goal. The DSE flow is also illustrated in Appendix D.

We measure the execution time (in seconds) of the our approach on a Windows machine with 3GHz 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[4] 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

In reliability-aware system design, many existing studies adopt the assumption that fault detection is always perfect to simplify the problem. We observe that this assumption causes several practical issues and may exclude the optimal design alternative. In this paper, we present an approach to synthesizing fault-tolerant design with reliability guarantee applying imperfect fault detectors. The proposed analysis and optimization techniques can be used for reliability-aware DSE. Experimental results verify that our approach finds solutions with several orders of magnitude higher reliability compared to current approaches.

Acknowledgement

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

8. REFERENCES

[1] A. Benoit, L.-C. Canon, E. Jeannot, and Y. Robert. Reliability of task graph schedules with transient and fail-stop failures: complexity and algorithms. *Journal of Scheduling*, 2011.

[2] D.C.Bossen. Cmos soft errors and server design. In *Reliability Physics Tutorial Notes, Reliability Fundamentals, pp. 121.07.1*, 2002.

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

[4] A. Girault and H. Kalla. A novel bicriteria scheduling heuristics providing a guaranteed global system failure rate. *IEEE Transactions on Dependable and Secure Computing*, 2009.

[5] J. Huang, J. O. Blech, A. Raabe, C. Buckl, and A. Knoll. Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems. In *CODES+ISSS*, Oct 2011.

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

[7] V. Izosimov, I. Polian, P. Pop, P. Eles, and Z. Peng. Analysis and optimization of fault-tolerant embedded systems with hardened processors. In *Design, Automation and Test in Europe (DATE)*, 2009.

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

[9] J. Lee, I. Shin, and A. Easwaran. Online robust optimization framework for qos guarantees in distributed soft real-time systems. In *EMSOFT*, 2010.

[10] A. Lifa, P. Eles, Z. Peng, and V. Izosimov. Hardware/software optimization of error detection implementation for real-time embedded systems. In *CODES+ISSS*, 2010.

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

[12] G. Lyle, S. Chen, K. Pattabiraman, Z. Kalbarczyk, and R. Iyer. An end-to-end approach for the automatic derivation of application-aware error detectors. In *DSN*, 2010.

[13] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer. Automated derivation of application-aware error detectors using static analysis: The trusted illiac approach. *Dependable and Secure Computing, IEEE Transactions on*, 8(1):44 –57, 2011.

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

[15] P. Pop, V. Izosimov, P. Eles, and Z. Peng. Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication. *IEEE Trans. VLSI*, 2009.

[16] P. Pop, K. H. Poulsen, V. Izosimov, and P. Eles. Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems. In *CODES+ISSS*, 2007.

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

[18] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer. Software-implemented hardware error detection: Costs and gains. In *Third International Conference on Dependability*, 2010.

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

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

[21] D. Zhu and H. Aydin. Energy management for real-time embedded systems with reliability requirements. In *ICCAD*, 2006.

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

[4]generated using TGFF `http://ziyang.eecs.umich.edu/~dickrp/tgff/`

APPENDIX

A. RELATED WORK

In the past decades, much research effort has been devoted to fault-tolerant system design considering transient faults. *Girault et al* [4] combine task scheduling with active spatial redundancy and present a bicriteria heuristic algorithm. Beside scheduling parameters, their algorithm also determines the number of replications that are needed to achieve certain reliability goal. *Izosimov et al* [8] study the design 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 [15] *Pop et al* study a similar problem and consider in addition the utilization of check-pointing and roll-back technique. The authors in [17] utilize a hybrid scheduling approach to handle mixed hard and soft real-time tasks. The aforementioned work [8, 15, 17] 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. In the follow-up work [7], a more accurate probabilistic analysis is presented. Nevertheless, this analysis considered only temporal redundancy. *Huang et al* further extend the approach and propose a binary tree based approach for probabilistic reliability analysis considering both spatial/temporal redundancy and shared re-execution slack.

Other work also studies the tradeoff between reliability and other design objectives, such as energy [22] and cost [14]. In [16] 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 [21]. They study static scheduling approaches for energy minimization under reliability constraints [22]. 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.

In all the work mentioned above, perfect fault detection is assumed. The assumption is that, all faults can be detected when a task is completed and timing overhead of fault detection is contained in the WCETs of tasks. In this paper, we show that certain configuration using imperfect fault detectors combined with replication can outperform those approaches that assumes perfect fault detection.

B. EXPERIMENTAL ANALYSIS ON THE IMPACT OF IMPERFECT FAULT DETECTION ON SYSTEM RELIABILITY

Figure 7 summarizes the results of the first simulation. We increase the fault detection coverage from 1% to 100% with a step width of 1% while fixing the number of replications. Figure 7a shows the case that a single instance is scheduled. As expected, the probability of SDC decreases linearly with the detection coverage, since all detected faults are converted to $DUFs$. In Figure 7b, two replicas are scheduled. The probabilities of both SDC and DUF decrease with higher coverage. The reason is as follows: In general, adding redundant components is a recovery technique that migrates faults from the DUF to the DTF class and implementing fault detectors is a detection technique that migrates faults from the SDC to the DUF class. However, if used together, the effects of redundancy and that of fault detection become *correlated.* As an example, assume the first instance generates a correct output whereas the second one encounters a fault. If the fault is undetected, the second one will produces a faulty output. Since the voter cannot distinguish which of the two outputs is correct, the system results in DUF. As the counterpart, if the fault is detected, the faulty instance can fail-silent and the only (and correct) output from the first instance is taken, resulting in a successful scenario. Hence, besides converting $SDCs$ to $DUFs$, fault detection can also convert $DUFs$ to $DTFs$ if voting is available. For this reason, probabilities of both DUF and SDC decease.

If three replicas are utilized (Figure 7c), the DUF probability first increases and then decreases with higher coverage, whereas the probability of SDC decreases constantly. The reason is that, the effect of SDC-to-DUF dominates when the coverage is still low (upper part of the Figure 7c), and the effect of DUF-to-DTF dominates when the coverage is relatively high. An observation from this set of simulations is that, higher fault detection coverage reduces the amount of SDCs but not necessarily reduces the amount of DUFs.

In the second simulation, we increase the number of replications while fixing the detector implementation. We reuse the result of [18] and assume that the rate of undetectable faults decreases exponentially with linear fault detection effort. We further assume the perfect fault detection (100% coverage) incurs 300% timing overhead (typical value in [18]). Several fault detectors with timing overhead ranging from 0% to 300% (corresponds to detection coverage from 0% to 100%) are tested. Figure 8 summarizes the results[5]. As can be seen, when the detection coverage is low, the probability curve shows a zigzag behavior with increasing number of replications (e.g., curve a). This is because the task instances themselves have only poor fault detection and the system relies mainly on the voter to discover the faults. On the one hand, the voter detects a fault when the number of correct and incorrect results breaks even. Hence, when we increment the number of replicas from an odd number and make it even (e.g., from 1 to 2), the fault detection capability of the voter is enhanced, resulting in a reduction of undetected faults (SDC probability drops). On the other hand, the voter recovers a fault when correct results are the majority. Hence, when a new instance is added to an even number of replicas, the amount of recoverable faults increases (DUF probability drops).

As the counterpart, if the task instances have already good fault detectors (e.g., in the case of curve d), the system reliability will be improved more smoothly by inserting more redundancy, i.e., both $DUFs$ and $SDCs$ can be eliminated at the same time. In other words, the effect of active redundancy could be amplified by good fault detection.

C. ILP BASED OPTIMIZATION FOR SINGLE-OBJECTIVE CASE

[5]The figure excludes the case of 0% and 300% by intension, because some of the probabilities are 0 and hard to be visualized in logarithmic scale.

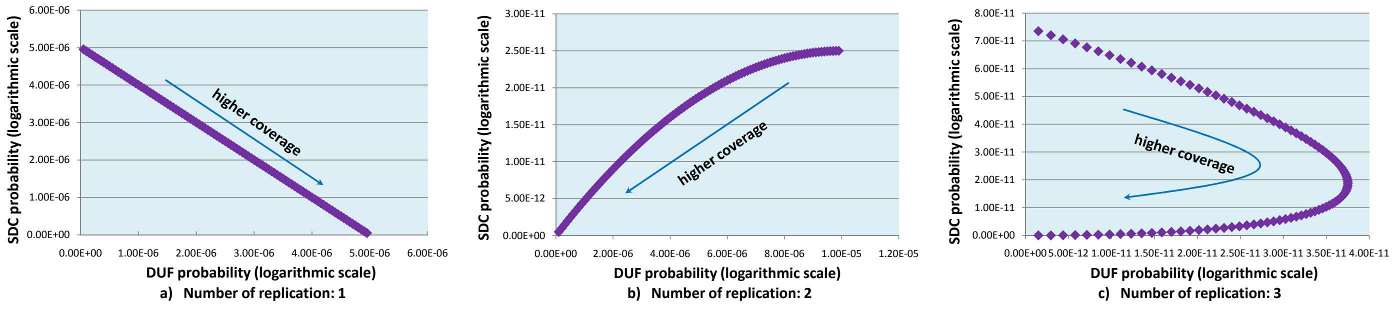


Figure 7: Effect of Fault Detection with Fixed Replication

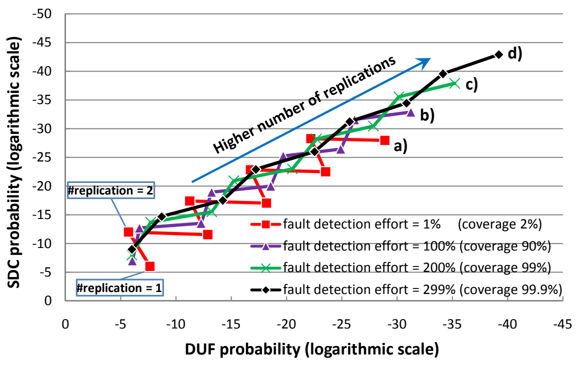


Figure 8: Effect of Replication with Fixed Fault Detection Coverage

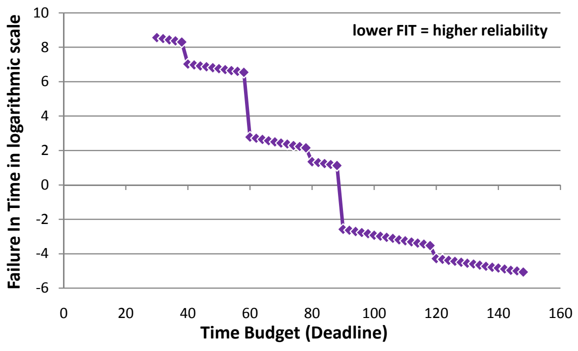


Figure 9: Example of Reliability Function

As mentioned in Section 5, this section presents an ILP based solution to handle the design scenario of maximizing the reliability of a single application. A real-time application typically has an end-to-end deadline that represents the time budget B for the entire application. The total budget can be distributed to individual tasks so that each task t_i has a local deadline b_i. The maximum reliability that can be archived by a task is constrained by the available local time budget. To describe this relationship, we define a Reliability Function (RF) $U_i(b)$, which is a monotonic function that models the achievable reliability of task t_i with given time budget b. Figure 9 depicts an example RF. The metric for reliability is Failure In Time (FIT). To capture both $DUFs$ and $SDCs$, we define the value of $U_i(b)$ to be a weighted sum of the FIT of both fault classes, i.e.:

$$U_i(b_i) = \alpha FIT_{DUF}(b_i) + \beta FIT_{SDC}(b_i) \qquad (9)$$

The weighting factors represent the criticality of the type of fault for the application. The RF for a task t_i can be obtained as follows. The possible time budget b_i assigned to t_i is lower-bounded by its execution time and upper-bounded by the available system slack time, i.e., $b_i \in [C_i, C_i + B - \sum_{\forall j} C_j]$. We sample this range with a fixed step width. For each sample value b, we investigate all design alternatives that fit into b, i.e., we try different numbers of replications and all implementable fault detectors[6]. For each design, the DUF and SDC probabilities are analyzed using equation 1-3 and the reliability is evaluated by equation 9. We assign the value of the $U(b)$ to be highest achievable reliability under the budget constraint.

After having the reliability function for all tasks, we can now compute the system reliability. The system-level SDC probability can be computed using equation 6. Since the success probabilities $P_{SUC}(\mathcal{T}\backslash t_0)$ and $P_{SUC}(t_0)$ are typically very close to 1, we approximate equation 6 as follows:

$$P_{SDC}(\mathcal{T}) < P_{SDC}(t_0) + P_{SDC}(\mathcal{T}\backslash t_0) < ...$$
$$= \sum_i P_{SDC}(t_i)$$

As can be seen, the system-level SDC probability can be overestimated by summarizing the SDC probabilities of all tasks. It can easily be verified that the system FIT can also be computed in an additive manner from the tasks' FITs. Similar approximation exists for the DUF probability. Let $\vec{\mathbf{b}}$ be a vector that contains the timing budget for each task. The system reliability can be approximated as $U_{sys}(\vec{\mathbf{b}}) = \sum_{t_i \in \mathcal{T}} U_i(b_i)$. The optimization problem becomes a deadline assignment problem stated as follows:

$$\begin{aligned} &Minimize : U_{sys}(\vec{\mathbf{b}}) = \sum_{t_i \in \mathcal{T}} U_i(b_i), \\ &Subject\ to : \sum_{b_i \in \vec{\mathbf{b}}} b_i \le B \end{aligned} \qquad (10)$$

[6]This procedure is durable since the number of alternatives is very limited. On the one hand, the number of replications for a single task is typically very small. On the other hand, since fault coverage increases monotonically with detection effort, we can simply choose the best detector that fits into the budget. When the complexity is still too high, methods like Monte Carlo simulation can be used to approximate the RF.

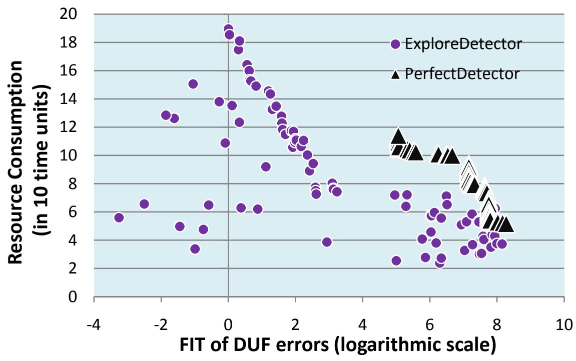


Figure 10: 2D Projection of Optimization Results: FIT of DUF vs Resource Consumption

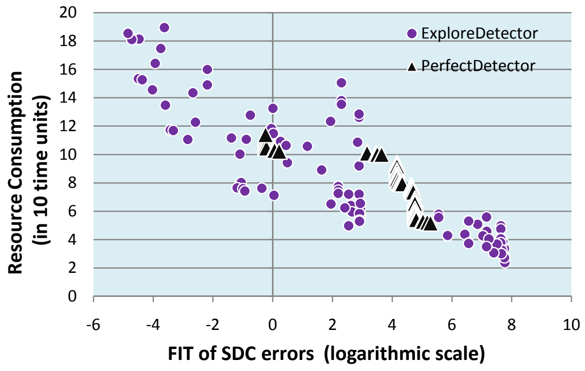


Figure 11: 2D Projection of Optimization Results: FIT of SDC vs Resource Consumption

By restricting the local time budget of each task to be a set of discrete values (as what is done to sample the RF), the above problem can be transformed into an integer linear programming problem and solved using standard solvers. Assume that M samples in the RF are considered for each local deadline value, i.e. $b_i \in \{b_{i,1}, ..., b_{i,M}\}$. We define a set of binary variables to describe the assignment of b_i:

$$x_{i,m} = \begin{cases} 1 & \text{iff } b_i \text{ is assigned to the } m\text{th sample } b_{i,m} \\ 0 & \text{otherwise} \end{cases}$$

Obviously, b_i can only be assigned to exactly one sampling value:

$$\sum_{m\in[1,M]} x_{i,m} = 1, \ \forall t_i \in \mathcal{T}.$$

The actual value of b_i can then be denoted as:

$$b_i = \sum_{m\in[1,M]} x_{i,m} b_{i,m}.$$

The actual reliability of the task i is:

$$u_i = \sum_{m\in[1,M]} x_{i,m} U_i(b_{i,m}).$$

The ILP problem can be stated as:

$$\begin{aligned} &Minimize: \sum_{t_i\in\mathcal{T}} u_i, \\ &Subject\ to: \sum_{t_i\in\mathcal{T}} b_i \le B \end{aligned} \tag{11}$$

The ILP formulation consists of $M\,|\mathcal{T}|$ binary variables (the x variables) and $2\,|\mathcal{T}|$ integer variables (for the b and u variables).

D. EXTENDED EXPERIMENTAL RESULTS

As mentioned in Section 6, this section presents extended experimental results. In Figure 6, the performance of two approaches is compared. For the $PerfectDetector$ approach, the FIT of SDC can be kept relatively low due to good detection coverage. However, the FIT of DUF is always beyond 10^5. Using the $ExploreDetector$ approach, we can obtain a wider spectrum of solutions, from the one that achieves very low FIT of DUF (C in Figure 6) to the one that achieves very low FIT of SDC (F in Figure 6). The designers can select the best implementation according to the application requirements.

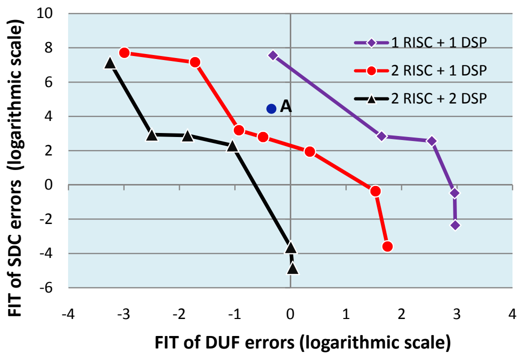


Figure 12: Comparing Results of three Architectures

The optimization results from MOEA can also be viewed from different angles. In Figure 10, the results are projected to in a 2D plane considering the FIT of DUF and resource consumption. Similarly, Figure 11 focuses on the FIT of SDC and resource consumption. Clearly, for both cases, the solutions found by $ExploreDetector$ have better quality than those found by $PerfectDetector$. Concerning SDC, the performance of $PerfectDetector$ is relatively close to that of $ExploreDetector$. Nevertheless, the performance gap is significant considering DUF. In this sense, the main drawback of the $PerfectDetector$ approach is that the design objective is biased. It fails to take application-specific reliability requirements into account. Instead, the focus is always on reducing the SDCs. For many applications (e.g. those requires fail-operational behavior), this is certainly a suboptimal approach.

The propose approach can be used to perform reliability-aware design space exploration. To show the DSE flow, we apply the approach on several platforms consisting of 2 to 4 processors. In Figure 12, we compare the maximum achievable reliability using different architectures. Clearly, the solutions found using a larger architecture dominate those obtained using a smaller architecture, due to the possibility of implementing more replications and/or better detectors. From these results, the designer may choose the best platform that meets the application requirement. For example, if the reliability goal is point A in Figure 12, the $2RISC + 1DSP$ platform is the cheapest one adhering to the requirement.