Strategy Switching: Smart Fault-tolerance for Resource-constrained Real-time Applications

Lukas Miedema[1], Benjamin Rouxel[1,2] and Clemens Grelck[1]

[1]University of Amsterdam (UvA), Amsterdam, Netherlands
[2]University of Modena and Reggio Emilia (Unimore), Modena, Italy

Abstract

Software-based fault-tolerance is an attractive alternative to hardware-based fault-tolerance, as it allows for the use of cheap Commercial Off The Shelf hardware. However, software-based fault-tolerance comes at a cost, requiring computing the same results multiple times to allow for the detection and mitigation of faults. Resource-constrained real-time applications may not be able to afford this cost. At the same time, the domain of a real-time task may allow it to tolerate a fault, provided it does not occur in consecutive iterations of the task. In this paper, we introduce a new way to deploy fault-tolerance called *strategy switching*. Our method targets Single Event Upsets by running different subsets of tasks under fault-tolerance at different points in time. We do not bound the number of faults in a window, nor does our method assume that tasks under fault-tolerance cannot still fail. Our technique does not require a minimal amount of additional compute resources for fault-tolerance. Instead, our method optimally utilizes any available compute resources for fault-tolerance for resource-constrained real-time applications.

Keywords

Cyber-physical Systems, Resource Constraints, Fault-tolerance, Single Event Upsets, Weakly Hard Real-time

1. Introduction

As transistor density increases and gate voltages decreases, the frequency of *transient faults* or *single event upsets* (SEUs) increases [1]. Hence, the need for fault-tolerance against these types of faults is growing.

Fault-tolerance techniques can either be implemented in hardware or in software. Software-based fault-tolerance is attractive due to its ability to protect workloads on *Commercial Off The Shelf* (COTS) hardware. However, providing general-purpose fault-tolerance against SEUs typically requires redundant execution, often in the form of *Triple Modular Redundancy* [2] (TMR). TMR uses two-out-of-three voting to obtain a majority and mitigate the effects of a SEU. TMR can be implemented at different levels of granularity, e.g. at the compiler level like *SWIFT-R* [3], but also at the OS task level as implemented in *OS Task Level Redundancy* [4]. However, the overhead remains: instrumenting a binary with SWIFT-R increases its execution time by 99 percent. As such, constrained real-time systems may have insufficient processing resources to allow all tasks to run with fault-tolerance. However, for applications structured as a set of periodic tasks, software-based fault-tolerance allows the application of fault-tolerance to only a subset of the task set.

Control tasks may be able to tolerate non-consecutive deadline misses, which has led to the adoption of the *weakly hard model* [5]. A task that is unable to provide a result may not result in catastrophe, provided that in the next period it can provide a result. Per the weakly hard model, each task i has an (m_i, k_i) constraint, indicating that the task must complete at least m_i times successfully out of every k_i times. k_i is said to be the *window size*. We use this (m_i, k_i) constraint with $m_i < k_i$ to deliver more effective fault-tolerance to resource-constrained systems. For example, consider a task set with just two tasks A and B, where only one task can be run under fault-tolerance at a time. Furthermore, both task A and B can tolerate non-consecutive deadline misses. Running task A under fault-tolerance and task B without would leave task B vulnerable to SEUs. However, we could more optimally make use of the scarce fault-tolerance by *switching* between protecting task A and task B in successive iterations of the task set. Tasks under fault-tolerance may still fail (e.g. TMR reaches no majority), and these cases can be detected. When the fault-tolerance technique has failed to protect task A, task A should be protected again in the next iteration of the task set to ensure it does not experience a consecutive fault.

Contribution We propose a new approach for improving fault-tolerance for real-time applications running on resource-constrained systems by *strategy switching*. We minimize the effective unmitigated fault-rate by selecting which tasks are to be run under fault-tolerance. Our approach recognizes that the importance of protecting a task may change over time due to earlier faults or lack thereof, and as such runs different tasks un-

CERCIRAS WS01: 1st Workshop on Connecting Education and Research Communities for an Innovative Resource Aware Society
l.miedema@uva.nl (L. Miedema); benjamin.rouxel@unimore.it (B. Rouxel); c.grelck@uva.nl (C. Grelck)

der fault-tolerance at different times. By exhaustively searching all patterns in which fault-tolerance can be applied, our method optimally utilizes limited available computational resources for fault-tolerance in resource-constrained real-time applications.

Organization In section 2 we introduce our task and fault model. Our method uses a *state machine*, which is introduced in section 3. In section 4, we formalize the construction of the state machine. Then, in section 5, we discuss how our method can lower the consecutive fault rate for an example task set. We explain the flexibility of our method by extending it to alternative task and fault models in section 6. Several pieces of related work exist, which are covered in section 7. The paper is concluded in section 8, and finally in section 9 we discuss various future directions for this technique.

2. System models

Task model We assume a set of periodic tasks $\Gamma = \{\tau_1 ... \tau_n\}$ with a single, global period and deadline D such that the period is equal to or larger than the deadline (no pipelining). Furthermore, we initially assume that each task can afford one (non-consecutive) deadline miss (an (m_i, k_i) constraint of (1,2)), and that each task is equally important. In section 6, we will discuss how some of these assumptions can be relaxed to support a wide variety of task models.

Fault model We use the Poisson distribution as an approximation for the worst-case fault rate of SEUs, which was argued to be a good approximation by Broster et al. [6]. We do not assume universal fault detection: only when the task runs under a fault-tolerance scheme can a fault be detected and mitigated. When a task does not run with fault-tolerance, it is not known whether or not it succeeded. We use the term *catastrophic fault* to describe an unmitigated fault occurring in two consecutive iterations of a task that can tolerate a single unmitigated fault, i.e. the task i has an $(m_i, k_i) = (1,2)$ constraint. Furthermore, a *catastrophic fault* also occurs when a task that cannot tolerate non-consecutive faults experiences an unmitigated fault, i.e. the task has a (1,1) constraint. We do not consider constraints beyond $k_i = 2$ in this paper.

Fault mitigation We assume the presence and implementation of a particular fault-tolerance scheme, and that any task can be run under that scheme. In this paper, we assume that SEUs always go undetected in tasks not under fault-tolerance. Finally, we assume the scheme implements fault-detection and fault-mitigation. Our model allows the fault mitigation to fail (e.g. due to successive SEUs during both replicas of a task under TMR), but assumes that it is known when fault mitigation fails.

Other definitions Given the complexity and number of symbols used in this paper, a table of all symbols and terms has been compiled in Table 1. Each symbol or term used will be defined prior to use, as well as being listed in the table.

3. Strategy Switching State Machine

To both swiftly select a new subset of the task set to run under fault-tolerance while also making optimal decisions, we precompute for each situation the next best subset of tasks to run under fault-tolerance. The result of this is the *strategy state machine*, which is made available to the online component. The strategy state machine is a bipartite state machine, consisting of *strategy* states and *result* states. An example of such a state machine is shown in Figure 1.

The architecture of our strategy switching approach distinguishes between an online part at runtime, as well as an offline part executing ahead-of-time not beholden to any real-time constraints. The offline component prepares the state machine, which is then available for online playback.

Online We introduce a *strategy switching component*, which plays back the strategy state machine, taking transitions based on observed faults as the application runs. At runtime, this component selects a single strategy s ahead of every execution of the task set, which becomes active. The strategy s dictates which tasks run under fault-tolerance (Γ_s), and which ones do not ($\Gamma \setminus \Gamma_s$). Fault-tolerance techniques are typically not a silver bullet solution, and unmitigated faults may still occur in tasks in Γ_s. Furthermore, fault-tolerance techniques can often report the fact that they failed to mitigate a fault (e.g. no consensus in triple modular redundancy). After executing all tasks, the online component uses this information from the execution of the task set to select the matching result r from the state machine. This result reflects the success or fail state, or probability thereof, of each of the tasks. Each possible result r directly maps to its best successor strategy, which is applied to the next iteration of the task set.

Offline The full set of strategies $s \in \mathcal{S}$ is computed ahead of time, as well as the transition relation Δ from any given result $r \in \mathcal{R}$ to the next best successor strategy $\Delta(r) = s$. Strategies which are not schedulable are not

Table 1
Definitions of used symbols and terms

Item	Meaning
Task model	
Γ	Set of all tasks, $\Gamma = \{\tau_1 ... \tau_n\}$
$\tau_i \in \Gamma$	Task $i \in \Gamma$, e.g. τ_A is task A
C_i	*Worst Case Execution Time* (WCET) of task i
D	Global deadline (shared by all tasks)
Fault model	
λ	Fault rate (Poisson)
(m_i, k_i)	Constraint indicating task i has to execute successfully for at least m_i iterations out of every k_i iterations
Unmitigated fault	Fault in a task not mitigated by a fault-tolerance technique
Catastrophic fault	Unmitigated fault that leads to the (m_i, k_i) constraint of the task being violated
States in the state machine	
$\mathcal{S}$	Set of all strategies
$s \in \mathcal{S}$	A strategy
$\Gamma_s \subset \Gamma$	The tasks protected under strategy s
$s_{A,B}$	A strategy protecting task A and B, i.e. $\Gamma_{s_{A,B}} = \{\tau_A, \tau_B\}$
$\mathcal{R}$	Set of all results
$r \in \mathcal{R}$	A result
$r_{A,\overline{B}}$	A result where task A (τ_A) succeeded and task B (τ_B) failed
Transitions in the state machine	
Δ	The transition function for the strategy state machine
$\Delta(s)$	The set of successors of strategy s as per the transition function Δ. Due to the bipartite nature of the state machine, this is always a set of results.
$\Delta(r)$	The successor of result r as per transition function Δ. Always a single element, and due to the bipartite nature of Δ it is always a strategy.
Scoring function	
$\delta(...)$	Scoring function (lower is better), provides steady-state catastrophic fault rate, i.e. the average probability of a catastrophic fault for any iteration of the task set
$\delta(\Delta)$	Scoring function applied to the entire state machine
$\delta(s, \Delta)$	Probability of a catastrophic fault when leaving strategy s considering transition function Δ
$\delta(r, \Delta)$	Probability of a catastrophic fault when leaving result r considering transition function Δ
$\delta(r,s)$	Probability of a catastrophic fault when transitioning from r to s
Probabilities	
p_i	Probability of an unmitigated fault in task τ_i when no fault-tolerance is used
q_i	Probability of an unmitigated fault in task τ_i when fault-tolerance is used

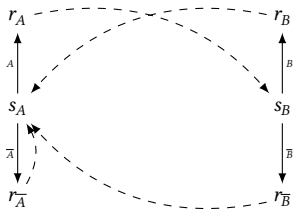


Figure 1: A strategy state machine for $\Gamma = \{\tau_A, \tau_B\}$

included in $\mathcal{S}$. Furthermore, strategies which are dominated by other strategies (i.e. all tasks that are protected by one strategy are also protected by another) are also not considered in $\mathcal{S}$.

We develop an algorithm to compute Δ, such that our choice of Δ provides the lowest steady-state rate of catastrophic faults.

Example State Machine An example of such a state machine is shown in Figure 1, where there are two strategies $\mathcal{S} = \{s_A, s_B\}$. This state machine is not the optimal state machine for this task set, but gives a non-trivial example of what such a state machine may look like. Such a state machine is constructed by the offline component, and made available to the runtime. Each strategy protects only one task (either task A or task B). Finally, fault-tolerance may fail and if it fails this information is available to the strategy switching component. With these assumptions, each strategy has two potential successors: one where the task under protection succeeds and one where it fails (e.g. r_A and $r_{\overline{A}}$ for strategy s_A). Each result state r has just one successor strategy $\Delta(r)$, e.g. $\Delta(r_A) = s_B$ in Figure 1. The following sequence of actions may take place at runtime, as per Figure 1;

1. To start, the runtime initializes by picking a random strategy, say s_B, and applies fault tolerance accordingly. By picking s_B, task τ_B will be executed with fault-tolerance, while task τ_A will not.
2. *The task set is executed (iteration 1).*
3. At the end of the iteration, data from the fault-tolerance applied to task τ_B is used to select a result. If τ_B fails, the result $r_{\overline{B}}$ is selected. However, let us assume τ_B succeeded, and select result r_A accordingly. No information about the success or failure of τ_A is known.
4. r_B links to strategy s_A, which is selected.
5. By switching to s_A, task τ_A will be executed with fault-tolerance, while task τ_B will not.
6. *The task set is executed (iteration 2).*
7. At the end of the iteration, data from the fault-tolerance of task τ_A indicates that task τ_A has failed, letting us select result $r_{\overline{A}}$.
8. $r_{\overline{A}}$ links to strategy s_A, which is selected.

$*$. *The process continues...*

Note that there is no involved selection procedure for the initial strategy (strategy s_B in the example). Our approach is only concerned with obtaining the lowest steady-state fault rate of the application, which is in no way impacted by the choice of initial strategy.

4. Evaluating State Machines

One way to find the optimal state machine transition function Δ for a given $\mathcal{S}$ is to enumerate all possible transition functions, score each transition function, and

select the best one. To do so, we define the state machine scoring function $\delta(\Delta)$. The output of this scoring function is the steady-state rate of catastrophic faults. Specifically, $\delta(\Delta)$ is the weighted probability of a catastrophic fault occurring during any iteration of the task set when applying strategy switching according to the transition function Δ.

To evaluate $\delta(\Delta)$, we compute the steady-state probability distribution of the transition function Δ (e.g. using linear algebra). The steady-state probability distribution provides a probability $P(s|\Delta)$ of finding the state machine in strategy s at an arbitrarily chosen iteration of the task set given Δ. Intuitively, as the number of iterations of the task set approaches infinity, $P(s|\Delta)$ is the fraction of iterations spent with strategy s active. The sum of these fractions over all strategies is 1, i.e $\sum_{s\in\mathcal{S}} P(s|\Delta) = 1$.

Let $\delta(s, \Delta)$ be the probability of a catastrophic fault occurring in strategy s. Together with $P(s|\Delta)$, we can now compute $\delta(\Delta)$:

$$\delta(\Delta) = \sum_{s\in\mathcal{S}} P(s|\Delta) \cdot \delta(s, \Delta)$$

Catastrophic faults, by their definition, only occur when an unmitigated fault occurs in two consecutive iterations of the task set. To compute $\delta(s, \Delta)$, we must not just consider s, but also the successor of s. The successor of s depends on what result r is being chosen, which is given by $P(r|s,\Delta)$. Each r has only one successor strategy $\Delta(r)$. Let $\Delta(r) = s_{to}$. Thus, $\delta(s,\Delta)$ can be expressed in terms of r and $\Delta(r) = s_{to}$ as follows:

$$\delta(s, \Delta) = \sum_{r\in\Delta(s)} P(r|s) \cdot \delta(r, s_to)$$

As s_{to} is a strategy, $\delta(r, s_{to})$ is independent of our choice of Δ. $\delta(r, s_{to})$ is equal to the probability that any task experiences a fault both in r as well as in s_{to}:

$$\begin{aligned}\delta(r, s_{to}) &= 1 - \prod_{\tau_i\in\Gamma} 1 - P(\text{catastrophic fault in } \tau_i|r,s_{to}) \\ &= 1 - \prod_{\tau_i\in\Gamma} 1 - P(\text{fault in } \tau_i|r) \cdot P(\text{fault in } \tau_i|s_{to})\end{aligned}$$

The probabilities $P(\text{fault in } \tau_i|r)$ and $P(\text{fault in } \tau_i|s_{to})$ are computed based on the used fault-tolerance technique, if $\tau_i \in \Gamma_{s_{to}}$, the rate of faults combined with the WCET of τ_i, and information available in the result. For example, a task without fault-tolerance may have a chance of experiencing a fault with $P(\text{fault in } \tau_i) = 1 - e^{-\lambda\cdot C_i}$, where C_i is the WCET of τ_i. Likewise, a task which was run under fault-tolerance according to the previous strategy, and has succeeded according to result r, will have a $P(\text{fault in } \tau_i|r) = 0$. We consider all SEUs to be statistically independent events. To keep independence, use the WCET instead of the average execution time, as the execution times of tasks in the same task set may not be independent. As such, our steady-state rate of catastrophic faults $\delta(\Delta)$ provides an upper bound to the true steady-state fault rate of the application.

Tractability The approach, as presented here, can easily become intractable for even small task sets. Many state space reduction techniques can be realized in the way the state machines are enumerated, which can considerably lower the number of elements. If a candidate state machine Δ contains disconnected sub-graphs, then no unique steady state can be computed, and as such this candidate can be pruned. Furthermore, the candidate state machine may contain strategies that are ignored, i.e. there is no path to that strategy from that same strategy. When this is the case, the steady-state probability $P(s|\Delta)$ is always 0. For such an ignored strategy, the successor strategies of its results do not matter, and its many ways of connecting to successor strategies need not be individually examined. Finally, the domain itself my allow for significant state space reduction. For example, if a precedence relation is added between tasks in the task set over which data is communicated, the failure of a preceding task may imply that the succeeding task is destined to fail. These modifications impact the lattice of strategies, and reduce the size of the schedulable and non-redundant set of strategies $\mathcal{S}$.

For completeness, we discuss the algorithmic complexity of the (naïve) state machine construction algorithm as presented in this paper. While the exact size of the state space depends on a number of factors, such as the number of available strategies $\mathcal{S}$, the upper bound is considerable.

The time complexity of $\delta(\Delta)$ is $\mathcal{O}(|\mathcal{R}| \cdot ss(|\mathcal{S}|))$, where:

- $|\mathcal{R}|$ is the number of results
- $|\mathcal{S}|$ is the number of strategies
- $ss(n)$ provides an upper bound to the computation of the steady state for n strategies

To compute $ss(n)$, the steady-state matrix needs to be computed. For this computation, the LAPACK driver routine DSEGD[1] may be used, with a complexity of $\mathcal{O}(n^3)$, resulting in $ss(n) = n^3$.

The number of strategies is up to all combinations of tasks ($|S| \in \mathcal{O}(|\Gamma|!)$). At the same time, $\delta(\Delta)$ is computed for every possible state machine. As each result can link to any successor strategy, this yields up to $|S|^{|R|}$ state machines. Each strategy has $\leq 2^{|\Gamma_s|}$ results, $2^{|\Gamma_s|} \in \mathcal{O}(2^{|\Gamma|})$ and thus $|R| \in \mathcal{O}(|\Gamma|! \cdot 2^{|\Gamma|})$. Let $n = |\Gamma|$, i.e. n is the number of tasks. Then, the final algorithmic time complexity is

[1] http://www.netlib.org/lapack/lug/node71.html

given:

$$\mathcal{O}(|\mathcal{S}|^{|R|} \cdot |\mathcal{R}| \cdot ss(|\mathcal{S}|)) =$$
$$\mathcal{O}(|\Gamma|!^{|\Gamma|!\cdot 2^{|\Gamma|}} \cdot |\Gamma|! \cdot 2^{|\Gamma|} \cdot |\Gamma|!^3) =$$
$$\mathcal{O}(n!^{n!\cdot 2^n} \cdot n! \cdot 2^n \cdot n!^3) \subset$$
$$\mathcal{O}(n!^{n!\cdot 2^n} \cdot 2^{n^5}) \approx$$
$$\mathcal{O}(n!^{n!\cdot 2^n}) \subset$$
$$\mathcal{O}(n!^{2^{2n}})$$

It must be noted that this is by no means a tight upper bound. In the next section, we will examine a task set with $|\Gamma| = n = 3$. This example requires examination of only 64 candidate state machines, even though such an n value would appear to be completely intractable as per the above formulation.

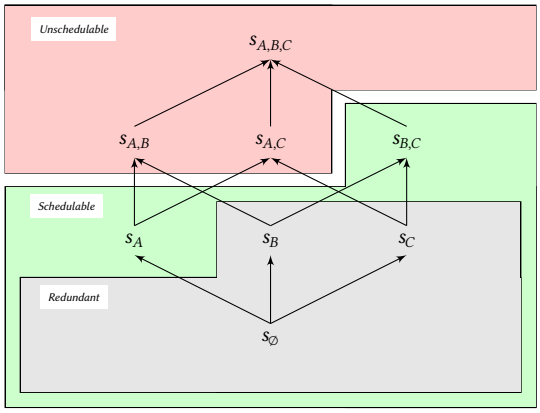


Figure 2: Lattice of strategies for $\Gamma = \{\tau_A, \tau_B, \tau_C\}$

5. Example

Let us consider an example task set $\Gamma = \{\tau_A, \tau_B, \tau_C\}$, with WCET values $C_A = 20$, $C_B = 10$ and $C_C = 10$. We set $\lambda = 10^{-3}$ for this example. For brevity, we abbreviate the probability of no fault in task τ_i when not using fault-tolerance as p_i:

$$P(\text{no fault in } \tau_i|\text{no FT}) = e^{-\lambda \cdot C_i} = e^{-\frac{C_i}{10^3}}$$
$$= p_i$$

As fault-tolerance technique, we use *Triple Modular Redundancy* (TMR). We require a majority for TMR to succeed, and assume there is no other way TMR can fail (e.g. assume no unmitigated faults during voting). A majority for TMR requires two or more copies of the task to be in agreement. Like p_i, we abbreviate the probability of no fault in task τ_i when using fault-tolerance as q_i:

$$P(\text{no fault in } \tau_i|\text{TMR}) = p_i^3 + \binom{3}{2} p_i^2 \cdot (1 - p_i)$$
$$= q_i$$

With the WCET for each task available, we can then compute the p_i and q_i values for all tasks:

$$\begin{array}{lll} \tau_A: & p_A = e^{-20\lambda} = 0.9802 & q_A = p_A^3 + 3(1-p_A)p_A^2 = 0.9988 \\ \tau_B: & p_B = e^{-10\lambda} = 0.9901 & q_B = p_B^3 + 3(1-p_B)p_B^2 = 0.9997 \\ \tau_C: & p_C = e^{-10\lambda} = 0.9901 & q_C = p_C^3 + 3(1-p_C)p_C^2 = 0.9997 \end{array}$$

Enumerating strategies For our chosen task set Γ, there are eight possible subsets and as such eight possible strategies:

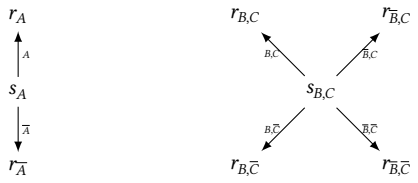
$$\begin{array}{lll} s_\emptyset: & \Gamma_{s_\emptyset} = \{\} & \Gamma \setminus \Gamma_{s_\emptyset} = \{\tau_A, \tau_B, \tau_C\} \\ s_A: & \Gamma_{s_A} = \{\tau_A\} & \Gamma \setminus \Gamma_{s_A} = \{\tau_B, \tau_C\} \\ s_B: & \Gamma_{s_B} = \{\tau_B\} & \Gamma \setminus \Gamma_{s_B} = \{\tau_A, \tau_C\} \\ s_C: & \Gamma_{s_C} = \{\tau_C\} & \Gamma \setminus \Gamma_{s_C} = \{\tau_A, \tau_B\} \\ s_{A,B}: & \Gamma_{s_{A,B}} = \{\tau_A, \tau_B\} & \Gamma \setminus \Gamma_{s_{A,B}} = \{\tau_C\} \\ s_{A,C}: & \Gamma_{s_{A,C}} = \{\tau_A, \tau_C\} & \Gamma \setminus \Gamma_{s_{A,C}} = \{\tau_B\} \\ s_{B,C}: & \Gamma_{s_{B,C}} = \{\tau_B, \tau_C\} & \Gamma \setminus \Gamma_{s_{B,C}} = \{\tau_A\} \\ s_{A,B,C}: & \Gamma_{s_{A,B,C}} = \{\tau_A, \tau_B, \tau_C\} & \Gamma \setminus \Gamma_{s_{A,B,C}} = \{\} \end{array}$$

Not all of these may be schedulable. If $s_{A,B,C}$ was schedulable there would be no reason to use strategy switching. Likewise, if only $s_\emptyset$ was schedulable, there is no way to deploy limited fault-tolerance to this task set. For the example, we assume that only strategies $s_\emptyset$, s_A, s_B, s_C and $s_{B,C}$ are determined to be schedulable by a scheduler. The strategies form a lattice with $s_\emptyset$ as the greatest lower bound, and $s_{A,B,C}$ as the least upper bound. The lattice is shown in Figure 2. Figure 2 also reveals redundant strategies, e.g. there's no reason to choose s_B when $s_{B,C}$ is schedulable. As such, there is no need to consider them. Let $\mathcal{S} = \{s_A, s_{B,C}\}$.

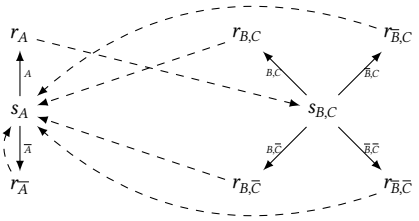
TMR can fail to reach a consensus, and then this information is available to the strategy selection component in the form of a result. As such, in this example each strategy s has $2^{|\Gamma_s|}$ possible results. For s_A this is r_A and $r_{\overline{A}}$, and for $s_{B,C}$ this is $r_{B,C}$, $r_{\overline{B},C}$, $r_{B,\overline{C}}$ and $r_{\overline{B},\overline{C}}$. These results are shown in Figure 3a, which shows the strategy state machine without successor relations for each result.

Evaluating transitions functions There are $|\mathcal{S}|^{|\mathcal{R}|} = 2^6 = 64$ possible transition functions Δ, as all of the six results needs to be linked to one of the two strategies. We will not enumerate all of them, but two examples are shown in Figure 3b and Figure 3c.

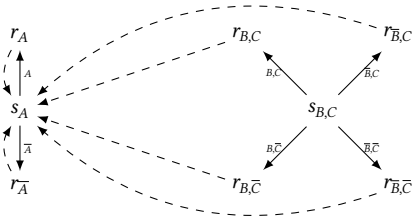
In this example, we will show how to evaluate $\delta(\Delta)$ for the state machine in Figure 3b. Let this be Δ_{3b}.



(a) Partial state machine without successor relations for the results



(b) Switching state machine choosing between s_A and $s_{B,C}$ based on the result



(c) Degenerate state machine always choosing s_A

Figure 3: Example state machines

1. Computing the steady-state probability of Δ_{3b}. The result states are "urgent" states, in which no time can pass. In other words, the result states themselves are not relevant for the steady-state (i.e. the fraction of iterations of the task set spent in state $r \in \mathcal{R}$ is 0). As such, we remove these states for the steady-state computation, linking each strategy to multiple successor strategies. The resulting state machine is a *Discrete-Time Markov Chain*, where each time step is an iteration of the task set Γ. The steady-state can be computed using linear algebra. Let T be the transition matrix for Δ_{3b} with all results removed:

$$T = \begin{bmatrix} P(r_{\overline{A}}|s_A) & P(r_A|s_A) \\ P(r_{B,C} \cup r_{\overline{B},C} \cup r_{B,\overline{C}} \cup r_{\overline{B},\overline{C}}|s_{B,C}) & 0 \end{bmatrix} = \begin{bmatrix} 1-q_A & q_A \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0.0012 & 0.9988 \\ 1 & 0 \end{bmatrix}$$

We can compute the steady-state vector v by solving $v = vT$.

$$v = vT \approx \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \end{pmatrix}$$

Note that a unique steady-state vector v need not exist if there are two or more parts of the state machine that are disconnected. For example, all results of s_A may link back to s_A, while all results of $s_{B,C}$ link back to $s_{B,C}$. In such a case, the steady-state is dependent on the initial state. However, we can safely disregard state machines that depend on the initial condition, as other state machines with identical steady-state behavior must exist as well. If staying in s_A provides the lowest $\delta(\Delta)$ value, then a state machine like shown in Figure 3c would yield the exact same $\delta(\Delta)$ value as a state machine where s_A and $s_{B,C}$ is disconnected with s_A as the initial state.

2. Compute, for each result → strategy transition, the $\delta(r, s_{\text{to}})$ catastrophic fault probability.

$$\begin{aligned}
\delta(r_A, s_{B,C}) &= 1- \quad (1-(0)\cdot(1-p_A))\cdot \\
&\qquad (1-(1-p_B)\cdot(1-q_B))\cdot \\
&\qquad (1-(1-p_C)\cdot(1-p_C)) \\
&= \quad 5.87\cdot 10^{-6} \\
\delta(r_{\overline{A}}, s_A) &= 1- \quad (1-(1)\cdot(1-q_A))\cdot \\
&\qquad (1-(1-p_B)\cdot(1-p_B))\cdot \\
&\qquad (1-(1-p_C)\cdot(1-p_C)) \\
&= \quad 1358\cdot 10^{-6} \\
\delta(r_{B,C}, s_A) &= 1- \quad (1-(1-p_A)\cdot(1-q_A))\cdot \\
&\qquad (1-(0)\cdot(1-p_B))\cdot \\
&\qquad (1-(0)\cdot(1-p_C)) \\
&= \quad 22.98\cdot 10^{-6} \\
\delta(r_{\overline{B},C}, s_A) &= 1- \quad (1-(1-p_A)\cdot(1-q_A))\cdot \\
&\qquad (1-(1)\cdot(1-p_B))\cdot \\
&\qquad (1-(0)\cdot(1-p_C)) \\
&= \quad 9973\cdot 10^{-6} \\
\delta(r_{B,\overline{C}}, s_A) &= 1- \quad (1-(1-p_A)\cdot(1-q_A))\cdot \\
&\qquad (1-(0)\cdot(1-p_B))\cdot \\
&\qquad (1-(1)\cdot(1-p_C)) \\
&= \quad 9973\cdot 10^{-6} \\
\delta(r_{\overline{B},\overline{A}}, s_A) &= 1- \quad (1-(1-p_A)\cdot(1-q_A))\cdot \\
&\qquad (1-(1)\cdot(1-p_B))\cdot \\
&\qquad (1-(1)\cdot(1-p_C)) \\
&= \quad 19923\cdot 10^{-6}
\end{aligned}$$

3. Using the calculated $\delta(r, s_{\text{to}})$ values, compute a $\delta(s \in \mathcal{S})$ value for each strategy.

$$\delta(s_A) = P(r_A|s_A)\cdot\delta(r_A, s_{B,C}) +$$

$$
\begin{aligned}
& P(r_{\overline{A}}|s_A) \cdot \delta(r_{\overline{A}}, s_A) = \\
& 7.6077 \cdot 10^{-6} \\
\delta(s_{B,C}) = & P(r_{B,C}|s_{B,C}) \cdot \delta(r_{B,C}, s_A) + \\
& P(r_{\overline{B},C}|s_{B,C}) \cdot \delta(r_{\overline{B},C}, s_{B,C}) + \\
& P(r_{B,\overline{C}}|s_{B,C}) \cdot \delta(r_{B,\overline{C}}, s_{B,C}) + \\
& P(r_{\overline{B},\overline{C}}|s_{B,C}) \cdot \delta(r_{\overline{B},\overline{C}}, s_{B,C}) = \\
& 29.69985 \cdot 10^{-6}
\end{aligned}
$$

$\delta(s)$ is the probability that a catastrophic fault occurs by selecting strategy s. The bad score of $s_{B,C}$ is not surprising: the Δ_{3b} state machine is not particularly clever as it chooses to switch to strategy s_A even with the knowledge that τ_B or τ_C has failed.

4. Finally, compute $\delta(\Delta_{3b})$ by taking the weighted average of all $\delta(s)$ values by multiplying $\delta(s)$ for each strategy by the steady-state probability $P(s|\Delta_{3b})$ of being in that strategy.

$$
\begin{aligned}
\delta(\Delta_{3b}) &= P(s_A) \cdot \delta(s_A) + P(s_{B,C}) \cdot \delta(s_{B,C}) \\
&= 1.814 \cdot 10^{-5}
\end{aligned}
$$

This process is repeated for all 64 possible transition functions. For brevity, we will not show the evaluation of every single transition function here. Instead, Table 2 shows a subset of all possible Δ values. The columns headed with a result show to which successor strategy that result maps. For example, for the first evaluated state machine $\Delta_1(r_A) = s_A$. The best state machine is also revealed, listed as Δ_{58}. Finally, the last item in the table is $\Delta_\emptyset$, added as a reference. $\Delta_\emptyset$ is the state machine with a single strategy $s_\emptyset$ which protects no tasks, i.e. the behavior obtained when not using any form of fault-tolerance. The best state machine is a 38.3-fold improvement over this default. That is, the best state machine offers a 38.3 times lower rate of catastrophic failure when compared to $\Delta_\emptyset$. Finally, the table also shows two state machines which perform no strategy switching, but still use fault-tolerance. These are Δ_1 and Δ_{64}, staying in s_A and $s_{B,C}$ respectively. The strategy switching solution Δ_{58} offers a 12.9 times lower rate of catastrophic faults when compared to the best of these static solutions.

Table 2
Shortened table of all possible Δ transition functions for Figure 3a

Δ	r_A	$r_{\overline{A}}$	$r_{B,C}$	$r_{\overline{B},C}$	$r_{B,\overline{C}}$	$r_{\overline{B},\overline{C}}$	$\delta(\Delta) \cdot 10^5$
Δ_1	s_A	s_A	s_A	s_A	s_A	s_A	19.935[5]
Δ_2	s_{BC}	s_A	s_A	s_A	s_A	s_A	1.8142[1]
Δ_3	s_A	s_{BC}	s_A	s_A	s_A	s_A	22.055
...							
Δ_{56}	s_{BC}	s_{BC}	s_{BC}	s_A	s_{BC}	s_{BC}	39.489
Δ_{57}	s_A	s_A	s_A	s_{BC}	s_{BC}	s_{BC}	19.935
Δ_{58}	s_{BC}	s_A	s_A	s_{BC}	s_{BC}	s_{BC}	1.5406[2]
Δ_{59}	s_A	s_{BC}	s_A	s_{BC}	s_{BC}	s_{BC}	22.054
Δ_{60}	s_{BC}	s_{BC}	s_A	s_{BC}	s_{BC}	s_{BC}	2.6115
Δ_{61}	s_A	s_A	s_{BC}	s_{BC}	s_{BC}	s_{BC}	*n/a*[3]
Δ_{62}	s_{BC}	s_A	s_{BC}	s_{BC}	s_{BC}	s_{BC}	39.226
Δ_{63}	s_A	s_{BC}	s_{BC}	s_{BC}	s_{BC}	s_{BC}	39.226
Δ_{64}	s_{BC}	s_{BC}	s_{BC}	s_{BC}	s_{BC}	s_{BC}	39.226[5]
$\Delta_\emptyset$	*n/a*						59.010[4]

[1] Example from Figure 3b ($\Delta_2 = \Delta_{3b}$)
[2] Best (lowest) steady-state fault rate
[3] No unique steady-state exists for this transition function
[4] $\delta(\Delta)$ of the task set without any form of fault-tolerance
[5] Does not strategy switch, i.e. always stays in the same strategy

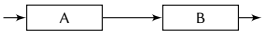


Figure 4: Task set $\Gamma = \{\tau_A, \tau_B\}$ with a precedence relation between τ_A and τ_B

6. Extending the fault and task model

6.1. Adding precedence relations

Directed Acyclic Graph scheduling is an extension to our task model where precedence relations exist between tasks. While this has no impact on the strategy switching algorithm directly as it need not concern itself with scheduling, it may add dependence to the success probability of a task. For this extension, we assume that when a predecessor task fails (produces incorrect data), all successor tasks fail as well due to operating on incorrect data, even when not experiencing a SEU.

To support this assumption, the $\delta(r, s_{\text{to}})$ cost function needs to be modified to consider precedence relations. Specifically, $P(\text{fault in } \tau_i|r)$ and $P(\text{fault in } \tau_i|s_{\text{to}})$ must be replaced to not just consider if τ_i failed in isolation. Let us call this precedence-aware probability the probability of *incorrect output*.

$$
P(\text{incorrect output } \tau_i|x) = P(\text{fault in } \tau_i|x) \cdot \prod_{\tau_j \in pred(\tau_i)} P(\text{fault in } \tau_j|x)
$$

Here, $x \in \mathcal{S} \cup \mathcal{R}$ is either a strategy or a result. $pred(\tau_i)$ is the set of all direct and indirect predecessors of task τ_i. This probability also includes the behavior of any predecessor tasks.

Figure 4 shows an example task set with a precedence relation. Consider the task set in this example with two strategies: $\mathcal{S} = \{s_A, s_B\}$, protecting task τ_A and τ_B

respectively. Each strategy has two outcomes, hence $\mathcal{R} = \{r_A, r_{\overline{A}}, r_B, r_{\overline{B}}\}$. Let us consider $P(\text{fault in } \tau_B | r_{\overline{A}})$: the result $r_{\overline{A}}$ does not directly communicate anything about the state of task B, however due to the precedence relation we know it effectively failed. As such, $P(\text{incorrect output } \tau_B | r_{\overline{A}}) = 1$.

6.2. Selective fault-tolerance and criticality

The task model can be extended to support heterogeneity in the ability of tasks to tolerate non-consecutive faults. Let Γ_N be the set of tasks which cannot tolerate non-consecutive faults. Then, we update $P(\text{catastrophic fault in } \tau_i | r, s_{to})$ to consider non-consecutive faults as catastrophic faults when $\tau_i \in \Gamma_N$.

$$P(\text{catastrophic fault in } \tau_i | r, s_{to} =)$$
$$\begin{cases} P(\text{fault in } \tau_i | r) \cdot P(\text{fault in } \tau_i | s_{to}) & \text{if } \tau_i \notin \Gamma_N \\ P(\text{fault in } \tau_i | r) & \text{else} \end{cases}$$

The goal of the strategy state machine is to minimize the rate of catastrophic faults. This may mean that the rate of catastrophic faults is so low that even a single fault is unlikely to occur across the lifetime of the system. However, when used in soft real-time deployments, a "catastrophic fault" may not be catastrophic while still undesirable. In such a deployment, catastrophic faults may be acceptable. The impact of a task experiencing such a catastrophic fault may not be the same across all tasks. As such, a priority function $p(\tau_i \in \Gamma)$ can be integrated into $\delta(r, s_{to})$.

$$\delta(r, s_{to}) =$$
$$1 - \prod_{\tau_i \in \Gamma} (1 - P(\text{catastrophic fault in } \tau_i | r, s_{to}))^{p(\tau_i)}$$

The $p(\tau_i)$ function assigns relative priority, where higher is better. A task τ_i with $p(\tau_i) = 2$ has twice the importance as a task τ_j with $p(\tau_j) = 1$. Note that this change affects the meaning of the output $\delta(\Delta)$, which is no longer the rate of catastrophic faults. If the impact of a catastrophic fault in a task with $p(\tau_i) = 2$ is equivalent to two catastrophic faults in another task with $p(\tau_j) = 1$, then the final rate $\delta(\Delta)$ can be seen as the rate of catastrophic faults normalized to faults in τ_j.

7. Related work

(m_i, k_i) constraints have been used before in the domain of real-time scheduling. Choi et al. [7] proposed a scheduler, together with an efficient schedulability algorithm for a sporadic task set with tasks under (m_i, k_i) constraints. This scheduler allows for scheduling task sets that would normally not be schedulable, but utilizing their (m_i, k_i) constraints allows them to be scheduled.

Chen et al. [8] proposed a solution that similar to ours. Their method offers fault-tolerance with the goal of reducing the effective fault rate as well as lowering energy consumption. Chen et al. proposes a static scheduling technique called *Static Pattern-Based Reliable Execution*, ensuring each (m_i, k_i) constraint is respected in the presence of transient faults. Furthermore, they propose delaying the execution of their static pattern if no fault is detected at runtime, opportunistically running more unprotected instances of the task with the goal of saving energy. However, if the static pattern is found to be unschedulable as per their schedulability test, their implementation is unable to provide a schedule that minimizes the fault rate for a given resource-constrained real-time system. While their approach offers more flexibility in the task model (specifically the support for (m_i, k_i) constraints with $k_i > 2$), it does not consider that fault mitigation may fail. Our approach optimally lowers the fault rate, regardless of the hardware constrains. Furthermore, our approach recognizes that fault mitigation may fail, and includes this in the calculation for lowering the fault rate.

Gujarati et al. [9] contributed a technique for measuring the fault rate of an application with tasks under (m_i, k_i) constraints. Their technique provides an upper bound for the fault probability per iteration of a *Fault-tolerant Single-Input Single-Output* (FT-SISO) control loop, similar to our $\delta(\Delta)$ output on a task set with precedence relations. Their technique hopes to provide transparency to system designers, allowing analyzing the impact on the reliablity when changing the hardware or software. However, while their approach is aware of (m_i, k_i) constraints, it does not provide schedules that utilize such constraints. Instead, it merely includes them in the reliability calculation.

The domain of strategy switching shares some aspects with *Mixed-Criticality* (MC) systems. In an MC system, the system switches between different levels of criticality depending on the operating conditions of the system. Tasks are assigned a criticality level, and when the system criticality is higher than that of the task, the task is not scheduled to guarantee the successful and timely execution of tasks with a higher criticality level. Pathan [10] combines MC with fault-tolerance against transient faults. As is typical in MC research, as the level of criticality increases, the pessimism increases. Pathan increases the maximum fault rate when switching to a higher level of criticality. In our approach we do not vary the pessimism of any parameter. Instead, we assume the λ parameter provides a suitable upper bound to the fault rate in all conditions. Our approach offers some aspects typically not found in MC systems: while one could argue

that each strategy is really a criticality level, it is a criticality level applied to a subset of the tasks (specifically Γ_s). Finally, the approach by Pathan requires bounding the number of faults that can occur in any window. As such, passing their sufficient schedulability test will (under their fault model) guarantee the system will never experience a fault.

8. Conclusion

In this paper, we have shown how strategy switching can be used to improve fault-tolerance for resource-constrained systems. Our method makes effective use of the ability to vary which tasks receive fault-tolerance. It considers at the start of every iteration of the task set what the best set of tasks is to protect. We have shown how our method computes the optimal strategy state machine for any given task set, minimizing the rate of catastrophic faults. We have also shown the flexibility of our method to be extended to support new task and fault models.

9. Future work

In future work, we hope to improve the tractability of our algorithm by both state space reduction algorithms as well as by using heuristics.

Furthermore, we hope to extend the fault model to distinguish between deadline misses and incorrect results. We also hope to integrate tasks with (m_i, k_i) constraints with $k_i > 2$. Additionally, we hope to integrate the natural ability of tasks to detect faults into our task model, as a SEU may for example lead to a segfault. Finally, we hope to validate our approach using simulation-based analysis.

Acknowledgments

The presentation of this paper was considerably improved in response to comments provided by the anonymous reviewers, and we gratefully acknowledge their insights and assistance.

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 871259 (ADMORPH project). Additionally, this work is partially supported by CERCIRAS COST Action CA19135 funded by COST Association.

References

[1] I. Oz, S. Arslan, A survey on multithreading alternatives for soft error fault tolerance, ACM Computing Surveys (2019).

[2] R. E. Lyons, W. Vanderkulk, The use of triple-modular redundancy to improve computer reliability, IBM journal of research and development 6 (1962) 200–209.

[3] J. Chang, G. A. Reis, D. I. August, Automatic instruction-level software-only recovery, in: International Conference on Dependable Systems and Networks (DSN'06), IEEE, 2006, pp. 83–92.

[4] S. A. Asghari, M. Binesh Marvasti, A. M. Rahmani, Enhancing transient fault tolerance in embedded systems through an OS task level redundancy approach, Future Generation Computer Systems 87 (2018) 58–65. doi:10.1016/j.future.2018.04.049.

[5] G. Bernat, A. Burns, A. Liamosi, Weakly hard real-time systems, IEEE transactions on Computers 50 (2001) 308–321.

[6] I. Broster, A. Burns, G. Rodriguez-Navas, Timing analysis of real-time communication under electromagnetic interference, Real-Time Systems 30 (2005) 55–81.

[7] H. Choi, H. Kim, Q. Zhu, Job-class-level fixed priority scheduling of weakly-hard real-time systems, in: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2019, pp. 241–253. doi:10.1109/RTAS.2019.00028.

[8] K.-H. Chen, B. Bönninghoff, J.-J. Chen, P. Marwedel, Compensate or ignore? Meeting control robustness requirements through adaptive soft-error handling, in: Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems, LCTES 2016, Association for Computing Machinery, New York, NY, USA, 2016, pp. 82–91. doi:10.1145/2907950.2907952.

[9] A. Gujarati, M. Nasri, B. B. Brandenburg, Quantifying the resiliency of fail-operational real-time networked control systems, in: S. Altmeyer (Ed.), 30th Euromicro Conference on Real-Time Systems (ECRTS 2018), volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2018, pp. 16:1–16:24. doi:10.4230/LIPIcs.ECRTS.2018.16.

[10] R. M. Pathan, Fault-tolerant and real-time scheduling for mixed-criticality systems, Real-Time Systems 50 (2014) 509–547.