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

Lukas MiedemaORCID (✉)[1] and Clemens GrelckORCID[1]

Faculty of Science, University of Amsterdam, Netherlands
{l.miedema, c.grelck}@uva.nl

**Abstract.** The probability of data corruption as a result of *single event upsets* (SEUs) increases as transistor sizes decrease. Software-based fault-tolerance can help offer protection against SEUs on *Commercial off The Shelf* (COTS) hardware. However, such fault tolerance relies on replication, for which there may be insufficient resources in resource-constrained environments. Systems in the weakly-hard real-time domain can tolerate some faults as a product of their domain. Combining both the need for fault-tolerance and the intrinsic ability to tolerate faults, we propose a new approach for applying fault-tolerance named *strategy switching*. Strategy switching minimizes the effective unmitigated fault-rate by switching which tasks are to be run under a fault-tolerance scheme at runtime. Our method does not require bounding the number of faults for a given number of consecutive iterations.
We show how our method improves the steady-state fault rate by analytically computing the rate for our test set of generated DAGs and comparing this against a static application of fault-tolerance. Finally, we validate our method using UPPAAL.

**Keywords:** cyber-physical systems · resource constraint · weakly-hard real-time · fault-tolerance · single event upset · adaptivity

## 1 Introduction

As transistor density increases and gate voltages decreases, the frequency of *transient faults* or *single event upsets* (SEUs) increases. As such, fault-tolerance techniques are becoming a requirement in computer systems [12]. Fault-tolerance techniques can either be implemented in hardware or in software. Hardware-based techniques are (partially) implemented in silicon, and as such can offer transparent fault-tolerance with minimal overhead. However, an implementation in silicon may not be feasible, e.g. due to the cost of manufacturing special-purpose microprocessors. Software-based fault-tolerance offers an attractive alternative due to its ability to protect workloads on *Commercial Off The Shelf* (COTS) hardware. Code is protected by executing it multiple times (redundant execution). The process of managing replication, determining consensus between the replicas, as well as any mitigation mechanism is entirely done in software.

Redundant execution often takes shape *N-Modular Modular Redundancy* [11] (NMR). NMR uses two-out-of-N voting on the output of some unit of code to obtain a majority and mitigate the effects of a SEU. At least three replicas are needed to obtain a majority in case of fault, which is known as *Triple Modular Redundancy* (TMR). Higher levels of $N$ offer robustness against multiple faults.

Modular redundancy can be implemented at different levels of granularity, e.g. by replicating individual instructions threefold like *SWIFT-R* [6], but also at the OS task level [1]. Regardless, significant overhead remains: instrumenting a binary with SWIFT-Rs technique increases its execution time by 99 percent [6]. As such, constrained real-time systems may have insufficient processing resources to complete protection with fault-tolerance. For applications consisting of multiple components or *tasks*, software-based fault-tolerance allows for protecting a subset of all tasks. This holds even when multiple tasks time-share the same processor, as the application of fault-tolerance is independent of the processor.

Control tasks may be able to tolerate non-consecutive deadline misses, which has led to the adoption of the *weakly hard model* [3]. 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. We use this $(m_i, k_i)$ constraint with $m_i < k_i$ to deliver more effective fault-tolerance to resource-constrained systems.

*Contribution* We propose *strategy switching*, a new approach for improving fault-tolerance for resource-constrained real-time applications. We minimize the effective unmitigated fault rate by selecting which tasks are to be run under the protection of a fault-tolerance mechanism. Our approach uses weakly-hard $(m_i,k_i) = (1,2)$ constraints on tasks to improve the effective fault rate by varying which tasks are protected at runtime. Strategy switching does not require a fault model within which the number of faults are bound, but instead minimizes the effective fault rate regardless of whether complete fault prevention is feasible. Finally, we offer an analytical solution to computing the effective fault rate when using strategy switching, and validate this solution using UPPAAL [2].

*Organization* In section 2 we introduce our task and fault model. Strategies are organized in a *state machine*, which is discussed in section 3. Our state machine construction algorithm is detailed in section 4. To evaluate the effect of our solution on the steady-state fault rate, we propose an analytical technique for obtaining said rate in section 5. We evaluate our strategy switching technique in section 6. Validation of our analytical method is done using UPPAAL in section 7. Related work is discussed in section 8, after which the paper is concluded in section 9. Finally in section 10 we discuss various future directions and propose improvements for our strategy switching technique.

## 2 System models

*Task model* We assume the application is structured as a set of periodic real-time tasks $\Gamma = \{\tau_1 ... \tau_n\}$ with a single, global period and deadline $D$ such that the

Table 1: Definitions of symbols and terms used in the task model

| **Item** | **Meaning** |
|---|---|
| Task model | |
| $\Gamma$ | Set of all tasks, $\Gamma = \{\tau_1 ... \tau_n\}$ |
| $\tau_i \in \Gamma$ | Task $i \in \Gamma$, e.g. $\tau_A$ is task A |
| $E$ | Set of all precedence relations, $E = \{(\tau_i, \tau_j), ...\}$ |
| $C_i$ | *Worst Case Execution Time* (WCET) of task i |
| $D$ | Global deadline (shared by all tasks) |
| Fault model | |
| $\lambda$ | Fault rate (Poisson distribution) |
| $(m_i, k_i)$ | 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 |

period is equal to or larger than the deadline (no pipelining). For each task $\tau_i$, a worst-case execution time $C_i$ is known. We also assume the effects of applying fault-tolerance to a task is known: fault-tolerance may create replicas that have to be scheduled, or the tasks' own $C_i$ value may increase as a result of redundancy. Furthermore, we support non-cyclic precedence relations $E = \{(\tau_i, \tau_j)\}$ for any task $\tau_i$ and $\tau_j$ where $\tau_i$ must precede $\tau_j$ creating a *Directed Cyclic Graph* (DAG) of tasks. Our technique requires the presence of a (global, offline) scheduler that can schedule the task set efficiently across the processors. The scheduler must be able to deal with fault-tolerance applied to any subset of the task set and yield a schedule. As the use of fault-tolerance increases the utilization of the system, the scheduler must be able to identify whether a particular subset of tasks under fault-tolerance is *schedulable* – i.e. able to meet the real-time deadline. Finally, every time the subset of tasks under fault-tolerance changes constitutes a real-time *mode switch*, as the schedule effectively changes from that moment on. As such, we require a middleware capable of making such a switch at the end of every period (when no tasks are running).

*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. [4]. 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 unknown whether or not it succeeded. Successor tasks rely on data produced by their predecessors, as such we consider faults to cascade across precedence relations. 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. 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 protected by a fault-tolerance mechanism. Fault-tolerance implemented using replication may fail (no consensus between the replicas). As such, we assume fault mitigation may fail, and 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 is compiled in Table 1. Each symbol or term used will be defined prior to use, as well as being listed in the table.

## 3 A State Machine of Strategies

To swiftly select a new subset of the task set to protect with fault-tolerance, we precompute the best subset of tasks to protect next for each situation. The response to such a situation is identified by a *strategy*, dictating which tasks to run with fault-tolerance. Exactly one strategy is active at any moment in time, and switching between strategies is facilitated through a *strategy state machine*.

Table 2: Definitions of symbols and terms used in the strategy state machine

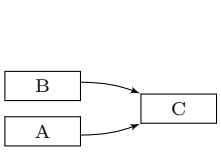
| **Item** | **Meaning** |
|---|---|
| States in the state machine | |
| $\mathcal{S}$ | Set of all strategies |
| $s \in \mathcal{S}$ | A strategy |
| $\Gamma_s \subseteq \Gamma$ | The tasks protected under strategy $s$ |
| $s_{A,B}$ | A strategy protecting task A and B, i.e. $\Gamma_{s_{A,B}} = \{\tau_A, \tau_B\}$ |
| $\mathcal{R}$ | Set of all results |
| $r \in \mathcal{R}$ | A result |
| $r_{A,\overline{B}}$ | A result where task A ($\tau_A$) succeeded and task B ($\tau_B$) failed |
| Transitions in the state machine | |
| $\Delta$ | The transition function for the strategy state machine |
| $\Delta(s)$ | The set of successors of strategy $s$ as per the transition function $\Delta$. 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 $\Delta$. Always a single element, and due to the bipartite nature of $\Delta$ it is always a strategy. |

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. The strategy state

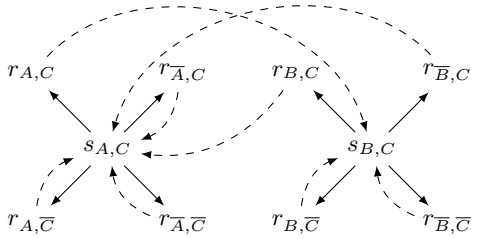
machine is a bipartite state machine, consisting of *strategy* states and *result* states. Figure 1b shows such a state machine. All symbols used to define the strategy state machine is given in Table 2.

*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 to protect with a fault-tolerance scheme ($\Gamma_s$), and which ones not ($\Gamma \setminus \Gamma_s$). Fault-tolerance techniques are typically not a silver bullet solution, and unmitigated faults may still occur in tasks in $\Gamma_s$. Furthermore, these techniques can often report the fact that they failed to mitigate a fault (e.g. no consensus in N-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 $\Delta$ from any given result $r \in \mathcal{R}$ to the best successor strategy $\Delta(r) = s$. Strategies which are not schedulable are pruned from $\mathcal{S}$. Furthermore, strategies which are dominated by other strategies (i.e. there is another strategy that protects a superset of tasks) are also not considered in $\mathcal{S}$.




(a) Task set $\Gamma = \{\tau_A, \tau_B, \tau_C\}$ with a precedence relation $E = \{(\tau_A, \tau_C), (\tau_B, \tau_C)\}$




(b) A strategy state machine for $\Gamma = \{\tau_A, \tau_B, \tau_C\}$

Fig. 1: Example task set with a corresponding example state machine

## 4 Strategy State Machine construction

Building the strategy state machine is a two-step process: (1) enumerating valid strategies and results, (2) determine the best successor strategy for each result. The state machine construction process is guaranteed to produce a state machine as long as the task set without any tasks under fault-tolerance is schedulable. However, in the case that no strategy applying fault-tolerance is schedulable, the strategy state machine becomes *degenerate*. Such a state machine consists

exclusively of the empty strategy, i.e. $\mathcal{S} = \{s_\emptyset\}$ with $\Gamma_{s_\emptyset} = \emptyset$. Given that there is only one strategy protecting nothing, no meaningful switching can occur as there is no other strategy to switch to. Under a degenerate strategy state machine, the application behaves as if it runs without fault-tolerance or strategy switching.

### 4.1 Enumerating strategies and results

The set of all strategies $\mathcal{S}$ and set of all results $\mathcal{R}$ can be constructed by considering every subset $\Gamma_s \subseteq \Gamma$ and applying fault protection accordingly.

The scheduler is used to mark subsets as either schedulable or unschedulable, depending on its ability to produce a schedule that meets the deadline with that subset running with fault protection. Marking each subset in effect forms an annotated lattice over the subset relation. An example of such a lattice is shown in Figure 2. The shown annotation could be the result of a high worst-case execution time $C_X$ when compared to $C_Y$ and $C_Z$. As such, the extra compute needed for running $\tau_A$ under fault-tolerance is much larger than doing the same for $\tau_Y$ or $\tau_Z$. This in turn makes the strategy protecting both $\tau_Y$ and $\tau_Z$ ($s_{Y,Z}$) schedulable, while protecting any task together with $\tau_X$ makes the strategy unschedulable (i.e. $s_{X,Y}$ and $s_{X,Z}$).

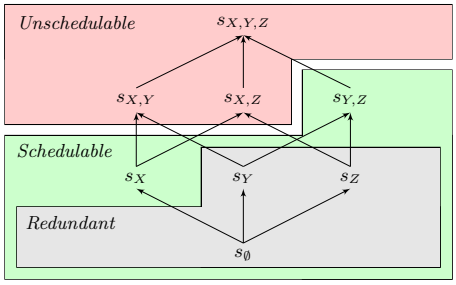



Fig. 2: Example schedulability lattice of strategies for some $\Gamma = \{\tau_X, \tau_Y, \tau_Z\}$

Some strategies protect a subset of tasks also protected by another schedulable strategy (e.g. $s_Y$ protects a subset of $s_{X,Y}$). In Figure 2, this is $s_Y$, $s_Z$ and $s_\emptyset$. These strategies are annotated as *redundant* and as such are discarded together with the unschedulable strategies. Each strategy $s \in \mathcal{S}$ has one result $r \in \mathcal{R}$ for each possible outcome. As success or failure is only known for tasks in $\tau_s$, a result $r$ is constructed for every combination of outcomes for tasks in $\tau_s$.

### 4.2 Strategy linking

Each result $r$ is linked to a successor strategy by a process called *linking*. When transitioning from a result $r$ to a strategy $s$, there is knowledge about two consecutive iterations of the task set. This is used to compute the expected number

of catastrophic faults as shown in Theorem 4.1 by means of $\delta(s,r)$. Then, our algorithm selects the best successor $s$ by minimizing $\delta(s,r)$ per Definition 4.1.

**Theorem 4.1.** *Expected number of catastrophic faults given $r$ was reached and $s$ will be activated*

$$\delta(s,r) = \sum_{\tau_i \in \Gamma} P(\textit{transitive fault in } \tau_i|r) \cdot P(\textit{transitive fault in } \tau_i|s)$$

**Definition 4.1.** *Determining a successor strategy $\Delta(r) \in \mathcal{S}$*

$$\Delta(r) = \arg\min_{s \in \mathcal{S}} \delta(s,r)$$

The fault probability $P(\text{transitive fault in } \tau_i|r)$ and $P(\text{transitive fault in } \tau_i|s)$ can be derived from the tasks execution time, the fault rate $\lambda$, the fault-tolerance scheme (NMR), and the result $r$ and strategy $s$. Given the Poisson distribution and the WCET $C_i$, the chance of a fault in $\tau_i$ is given as $p_i$ in Definition 4.2, while the chance of a fault under NMR is given as $q_i$ in Definition 4.3.

**Definition 4.2.** *Chance of a fault in any invocation of task $\tau_i$ when no fault tolerance ("NOFT") mechanism is applied*

$$P(\textit{fault in } \tau_i|NOFT) = e^{-\lambda \cdot C_i} = p_i$$

**Definition 4.3.** *Chance of a fault in any invocation of task $\tau_i$ when NMR is used*

$$P(\textit{fault in } \tau_i|NMR) = p_i^3 + \binom{3}{2} p_i^2 \cdot (1 - p_i) = q_i$$

The protection status of a task (either "NOFT" or "NMR") can be read from the strategy, as shown in Definition 4.4.

**Definition 4.4.** *Chance of a fault in any invocation of task $\tau_i$ under strategy $s$*

$$P(\textit{fault in } \tau_i|s) = \begin{cases} \textit{if } \tau_i \textit{ protected by } s & = P(\textit{fault in } \tau_i|NMR) \\ \textit{otherwise} & = P(\textit{fault in } \tau_i|NOFT) \end{cases}$$

We assume faults propagate along the DAG as invalid output is sent to successor tasks. Definition 4.5 defines the probability of a transitive fault, where the fault can either originate from itself or from a predecessor.

**Definition 4.5.** *Chance of a transitive fault in any invocation of task $\tau_i$ under strategy $s$*

$$\begin{aligned} P(\textit{transitive fault in } \tau_i|s) = \; & P(\textit{fault in } \tau_i|s) + (1 - P(\textit{fault in } \tau_i|s)) \\ & \cdot \left(1 - \prod_{\tau_j \in pred(\tau_i)} 1 - P(\textit{fault in } \tau_j|s)\right) \end{aligned}$$

The same idea of Definition 4.4 is used to define the chance of success given a result, which is given in Definition 4.6. The same mechanism for handling precedence relations as seen in Definition 4.5 can be applied using $P(\text{fault in } \tau_i|r)$ to derive $P(\text{transitive fault in } \tau_i|r)$, which we will omit for brevity.

**Definition 4.6.** *Chance of a fault in any invocation of task $\tau_i$ when result $r$ of strategy $s_r$ was reached*

$$P(\textit{fault in } \tau_i|r) = \begin{cases} \textit{if } \tau_i \textit{ succeeded per } r & = 1 \\ \textit{if } \tau_i \textit{ failed per } r & = 0 \\ \textit{otherwise} & = P(\textit{fault in } \tau_i|s_r) \end{cases}$$

### 4.3 State machine construction algorithm

We show the entire strategy state machine construction process in Algorithm 1.

① All strategies are enumerated and the scheduler is used to determine for each strategy its schedulability status. The lattice relation, as shown in Figure 2, is used to significantly reduce the number of times the scheduler needs to be invoked. When a strategy is found to be unschedulable, all strategies protecting *more* tasks are immediately marked as unschedulable. Likewise, when an unschedulable strategy is encountered, all strategies protecting *fewer* tasks are marked as schedulable.
② The resulting $\mathcal{S}$ contains all strategies, and is pruned of unschedulable strategies and strategies that protect a subset of tasks than other schedulable strategies.
③ Set $\mathcal{S}$ is further pruned, removing all strategies that provide equal or worse protection when compared to some other strategy in $\mathcal{S}$.
④ Results are constructed and their successor $\Delta(r) \in \mathcal{S}$ is determined for each of the remaining strategies. Each result associated with strategy $s$ is identified with a bitmask $o$ over $\Gamma_s$ such that index $l$ in the bitmasks identifies whether task $\tau_l$ at index $l$ in $\Gamma_s$ is protected. The results are added to $\mathcal{R}$.

### 4.4 Algorithmic complexity

The approach, as presented here, can easily become intractable for even small task sets due to the explosion of $\mathcal{S}$ and $\mathcal{R}$. In section 10, we discuss ways to lower the algorithmic complexity. For completeness, we discuss the algorithmic complexity of the (naïve) state machine construction algorithm as presented.

The number of strategies is up to all combinations of tasks, i.e. $|S| \in \mathcal{O}(|\Gamma|!)$. Each strategy has $\leq 2^{|\Gamma_s|}$ results, $2^{|\Gamma_s|} \in \mathcal{O}(2^{|\Gamma|})$ and thus $|\mathcal{R}| \in \mathcal{O}(|\Gamma|! \cdot 2^{|\Gamma|})$. Let $n = |\Gamma|$, i.e. $n$ is the number of tasks. Then, the final algorithmic time complexity is given in Theorem 4.2.

**Algorithm 1** Complete state machine construction algorithm

① Collect all strategies and their schedulability status
$\mathcal{S} \leftarrow \emptyset$
**for all** $\Gamma_j \in$ subsets of $\Gamma$ **do**
  **if** $\exists s \in \mathcal{S} : \Gamma_s = \Gamma_j$ **then** ▷ Skip if a strategy for subset $\Gamma_j$ already exists
    **continue**
  **end if**
  $\Gamma_s \leftarrow \{\ \text{NMR}(\tau_i)\ :\ \tau_i \in \Gamma_j\ \} \cup (\Gamma \setminus \Gamma_j)$ ▷ Set of all tasks with NMR applied
  **if** schedulable($\Gamma_s$) **then**
    $s_j \leftarrow$ **new** Strategy($\Gamma_s$, schedulable = **true**) ▷ Strategy is schedulable
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{s_j\}$
    **for all** $\Gamma_k \in$ subsets of $\Gamma$ **where** $\Gamma_j \subset \Gamma_k$ **do** ▷ Propagate down the lattice
      $s_k \leftarrow$ **new** Strategy($\Gamma_k$, schedulable = **true**)
      $\mathcal{S} \leftarrow \mathcal{S} \cup \{s_k\}$
    **end for**
  **else**
    $s_j \leftarrow$ **new** Strategy($\Gamma_s$, schedulable = **false**) ▷ Strategy is unschedulable
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{s_j\}$
    **for all** $\Gamma_k \in$ subsets of $\Gamma$ **where** $\Gamma_k \subset \Gamma_j$ **do** ▷ Propagate up the lattice
      $s_k \leftarrow$ **new** Strategy($\Gamma_k$, schedulable = **false**)
      $\mathcal{S} \leftarrow \mathcal{S} \cup \{s_k\}$
    **end for**
  **end if**
**end for**

② Prune strategies based on schedulability and redundancy
$\mathcal{S} \leftarrow \{s : s \in \mathcal{S}, s$ **is** schedulable$\}$ ▷ Using information in $s \in \mathcal{S}$
$\mathcal{S} \leftarrow \{s_i : s_i \in \mathcal{S}, \neg\exists s_j \in \mathcal{S} : \Gamma_{s_i} \subset \Gamma_{s_j}\}$ ▷ Remove redundant strategies

③ Prune strategies based on fault-tolerance quality
**for all** $s_i \in \mathcal{S}$ **do**
  **for all** $s_j \in \mathcal{S} \setminus s_i$ **do**
    **if** $\forall \tau_k \in \Gamma : P(\text{transitive fault in } \tau_k | s_i) \leq P(\text{transitive fault in } \tau_k | s_j)$ **then**
      $\mathcal{S} \leftarrow \mathcal{S} \setminus s_j$ ▷ $s_j$ is equal or worse for all tasks than $s_i$
    **end if**
  **end for**
**end for**

④ Create and link results
$\mathcal{R} \leftarrow \emptyset$
$\Delta \leftarrow \emptyset$
**for all** $s_i \in \mathcal{S}$ **do**
  **for all** $o_j \in$ all combinations of success and failure for $\Gamma_s$ **do** ▷ $o_j$ is a bitmask
    $r_j \leftarrow$ **new** Result($o_j$)
    $s_{\text{next}} \leftarrow s \in \mathcal{S} : \min_{s_k \in \mathcal{S}} \delta(s_k, r_j) = \delta(s, r_j)$ ▷ Linking per Definition 4.1
    $\Delta(r_j) \leftarrow s_{\text{next}}$
    $\Delta(s_i, o_j) \leftarrow r_j$
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{r_j\}$
  **end for**
**end for**

**Theorem 4.2.** *Time-complexity for the construction of the strategy state machine for $|\Gamma| = n$ tasks*

$$\mathcal{O}(|\mathcal{R}| + |\mathcal{S}|) \approx \mathcal{O}(|\mathcal{R}|) \in \mathcal{O}(|\Gamma|! \cdot 2^{|\Gamma|}) = \mathcal{O}(n! \cdot 2^n)$$

Note that this is a high upper bound which is unlikely to be hit by an arbitrary task graph. Redundant strategies are identified and pruned before the results are enumerated as seen in Algorithm 1, reducing the number of results significantly. In further work, we intend to improve the tractability by improving the strategy enumeration process itself.

## 5 Evaluating State Machines

To evaluate and compare our algorithm against a static (non-switching) solution, a way of determining the effective fault rate is necessary. We define the effective fault rate as $\delta(\Delta)$. $\delta(\Delta)$ is the expected number of catastrophic faults for any arbitrary period of the task set managed according to state machine $\Delta$. $\delta(\Delta)$ can be obtained analytically converting it to a Discrete-Time Markov Chain.

### 5.1 Discrete-Time Markov Chain evaluation

The expected number of catastrophic faults in $\Delta$ is the weighted average of the $\delta(r,s) = \delta(r,\Delta(r))$ function from Theorem 4.1. The weight of result $r$ can be derived from its strategy $s$, as shown in Theorem 5.1.

**Theorem 5.1.** *Probability of selecting result $r$ given strategy $s$*

$$P(r|s) = \prod_{\tau_i \in \Gamma_s} \begin{cases} \text{if } \tau_i \text{ fails in } r & = P(\text{transitive fault in } \tau_i|s) \\ \text{otherwise} & = 1 - P(\text{transitive fault in } \tau_i|s) \end{cases}$$

**Theorem 5.2.** *Expected number of catastrophic faults per period of the task set*

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

$P(s|\Delta)$ provides the steady-state probability of finding the strategy state machine in strategy $s$. $P(s|\Delta)$ can be computed by converting the state machine into a *Discrete-Time Markov Chain*. Conversion is applied as follows:

1. Result states are removed, and all incoming edges are transferred directly to the successor of each result state.
2. A transition matrix $T_\Delta$ is created from $\Delta$.
3. The steady-state vector $S\vec{S}_\Delta$ of $T_\Delta$ is computed (e.g. using linear algebra).
4. $S\vec{S}_\Delta(s \in \mathcal{S}) = P(s|\Delta)$.

## 6 Evaluation

To evaluate our technique, we generate a set of task graphs and compute a strategy state machine for each graph across a variety of scenarios.

### 6.1 Dataset

We generate 500 task graphs with between 2 and 20 tasks using *Task Graphs For Free* (TGFF) [9]. Each task graph is statically scheduled once with Forward List Scheduling targeting a 4 core platform, without any fault-tolerance. These schedules are used to determine each task graphs' base makespan. We set the fault rate to $\lambda = 10^{-5}s^{-1}$ and determine fault probabilities using the Poisson distribution, i.e. $P(\text{fault in } \tau_i | \text{ NOFT}) = e^{\lambda \cdot C_i}$. As fault-tolerance technique we use TMR at the task level. The scheduler is tasked with scheduling the three replicas and voter according to their precedence relations.

To simulate resource-constrained scenarios, we obtain results for various deadlines. We define the deadline as a multiple of the base makespan. A multiple of 1 (i.e. deadline = 1×makespan) leaves only strategies that can place task replicas in existing gaps in the schedule, while a multiple of 3 leaves enough time available for every task to run three times.

### 6.2 Results

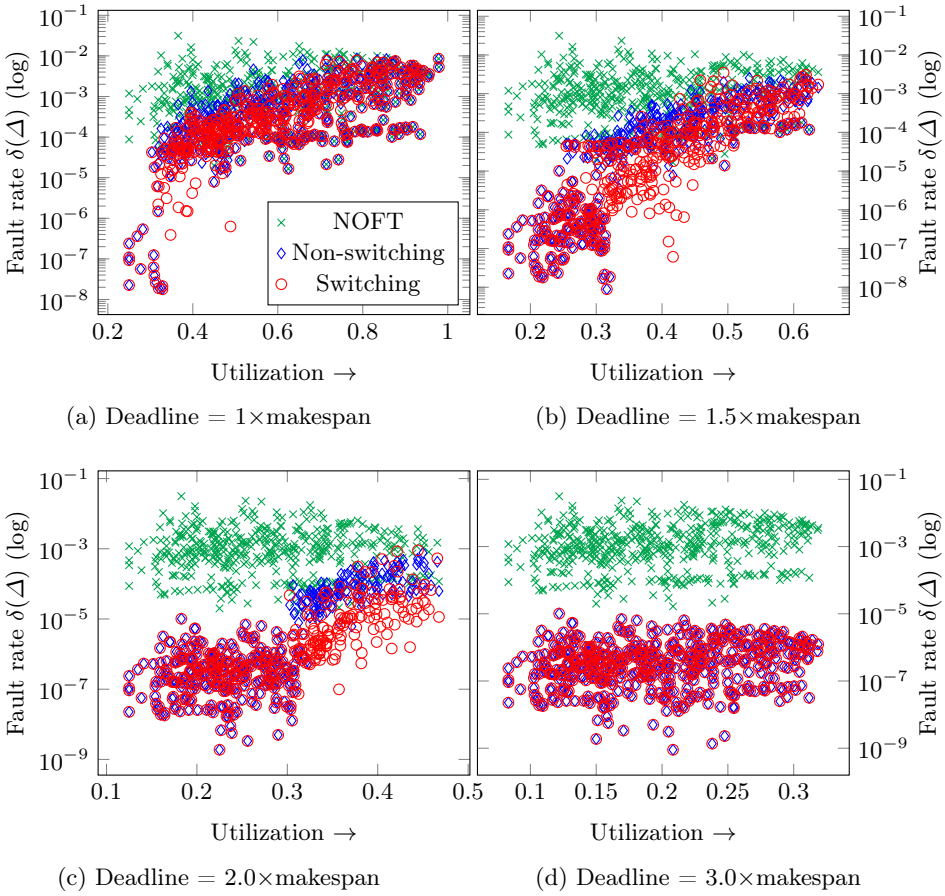
Figure 3 shows the results of the 500 task graphs with four different deadlines. For each plot, the x-axis represents utilization without fault-tolerance, while the y-axis presents the steady-state fault rate $\delta(\Delta)$ (lower is better).

The higher the utilization, the fewer unused resources there are to use for placing replicas. The effect of this is visible – the higher the utilization, the higher the fault rate in both the switching and non-switching case. Each plot compares our strategy-switching solution to a non-switching one where only a single strategy is selected, as well as one without fault-tolerance ("NOFT").

For the extremely constrained scenario of deadline = 1×makespan (Figure 3a), our strategy-switching solution manages to hit a lower fault-rate than the non-switching solution in most cases. Our solution offers a 17.82% lower steady-state fault rate on average (±37.34% std. dev) when compared to the non-switching approach. At a more relaxed deadline = 1.2×makespan, these figures stay about the same (17.92% improvement with ±44.06% std. dev). But relaxing the deadline to 1.4×makespan yields a fault rate reduction of 24.79% (±50.29% std. dev). When the ratio matches the extra resource requirement of TMR ($\leq 3$), all solutions perform identical as seen in Figure 3d.

We analyze this behavior at the hand of Figure 4, where the relative improvement for the first 6 DAGs is shown across multiple makespan/deadline ratios. The strategy switching solution provides a reduction in fault-rate for intermediate makespan/deadline ratios.

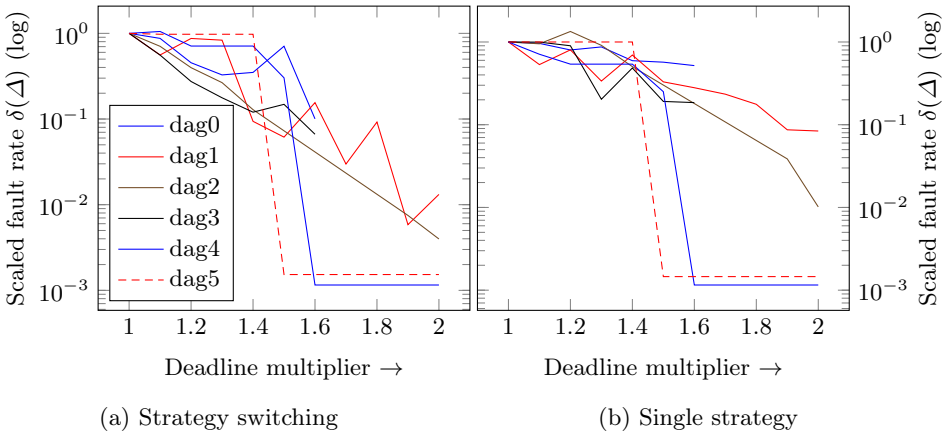
With more relaxed deadlines, the number of possible strategies starts to overtake the tractability of our algorithm: at 1.2×, for 3 of the 500 task graphs




(a) Deadline = 1×makespan (b) Deadline = 1.5×makespan

(c) Deadline = 2.0×makespan (d) Deadline = 3.0×makespan

Fig. 3: Utilization vs. steady-state fault rate for a non-switching solution and our switching solution across various levels of resource constrainity. Lower fault rate is better.




(a) Strategy switching (b) Single strategy

Fig. 4: Relative improvement in fault rate when the makespan multiplier increases for the first 6 DAGs

a state machine cannot be constructed within 30 minutes on our hardware. This grows to 20 of the 500 graphs for a multiplier of 1.4×. However, this does not take away from the validity of the state machine for DAGs where it is feasible.

Our approach does not always yield an improvement. For 1× multiplier, strategy switching offered an improvement when compared to a static solution in 272 of the 500 cases. In 68 of the missing cases, our strategy-switching approach performs *worse* than the non-switching solution. Our greedy $\Delta(r)$ successor determination method as presented in Definition 4.1 is naive and susceptible to make locally-optimal decisions that are detrimental to the total fault-rate. Such cases can easily be avoided however: as we propose an analytical method for computing the fault rate in this paper, the algorithm can easily be extended to verify that the resulting state machine outperforms a static solution. When this is not the case, it can fall back to the static solution. In future work, we intent on creating a better linking algorithm that could also deliver in an improvement in these cases, and not have to fall back to a static non-switching solution.

## 7 Validation using UPPAAL

To improve confidence in our analytical evaluation, we model the online part of the strategy switching using UPPAAL [2]. Online strategy switching is combined with UPPAAL processes for tasks, edges, processors, and a variety of monitoring models. These processes in effect build a complete runtime simulator with which we can study long-running behavior of a weakly-hard real-time application when experiencing a given incidence rate of faults. UPPAAL is used in *Stochastic Model Checking* (SMC) mode [5], which lets us estimate the expected number of catastrophic faults for a large number of periods of a particular task set. For validation to succeed, this estimation must match the analytical solution obtained per Section 5. As each model is specific to one task set and its strategy state machine, we develop a generator that produces a UPPAAL model automatically from a task set and strategy state machine.

### 7.1 UPPAAL processes

Systems models constructed in UPPAAL [2] are composed of a set of concurrently-running UPPAAL *processes* derived from *process templates*. We define 12 templates for validation, in which three categories can be identified:

i) *Our contribution*: a strategy state machine process, plus a set of result matcher processes identifying when particular results is reached
ii) *Hardware & application*: task, edge, NMR voter and processor templates
iii) *Monitoring*: task and edge monitoring templates to propagate faults and register the actual catastrophic fault rate

For brevity we limit discussion to one template: the task process template. This template is shown in Figure 5. A task process is created for each combination of a task $\tau_i$ and strategy $s_j$. It is parameterized with task index $i$, strategy id $j$, the release time $r_i$, its WCET $C_i$, the assigned core $P_i$.

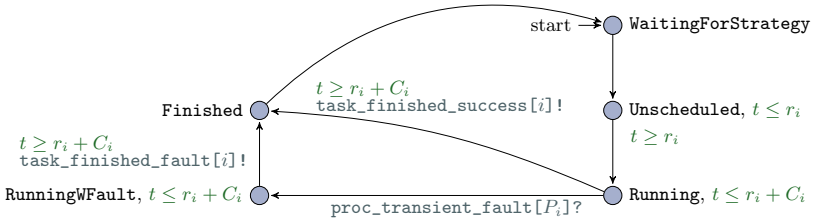



Fig. 5: Simplified UPPAAL process template for a task

When the strategy $s_j$ is activated, the task process moves to the `Unscheduled` location. This location is left when clock $t$ reaches $t = r_i$. In the `Running` state, it may encounter a fault signaled via the `proc_transient_fault[`$P_i$`]`. The fault is captured by moving to `RunningWFault`. The process finishes by either emitting a `task_finished_success[`$i$`]` or `task_finished_fault[`$i$`]` depending on its fault status, which is relayed to a voter process (omitted for brevity).

### 7.2 Validating results

The UPPAAL model is periodic, and counts the number of catastrophic faults encountered by means of the monitoring processes. To determine the global steady-state fault rate $\delta(\Delta)$, we query the number of catastrophic faults for a large number of tasks set iterations. Then, validation succeeds when $\delta(\Delta) \approx \frac{\#\text{ catastrophic faults}}{\#\text{ iterations}}$. We set the confidence interval for this experiment to 95%.

We apply the UPPAAL model transformation to the first 10 task graphs with deadline=1×makespan, the validation results of which can be seen in Table 3. The raw data is shown in the "# **faults**" column, and is obtained using the query `E[<=`$t$`;128](max:catastrophic_faults)`. The formula estimates the number of catastrophic faults seen until time $t$. 128 such simulations are conducted to gain confidence in the stability of the value and get the bounds of the 95% confidence interval (shown with ±). In the formula, parameter $t$ is set to the period × 4096 to simulate 4096 consecutive iterations of the task set. As such, $\delta(\Delta)$ is approximated by dividing the output of the formula by 4096.

Two values (dag2 and dag8) are absent: the UPPAAL query did not return a result in 24 hours. The extremely low fault-rate of dag7 makes seeing a single fault in $4096 \cdot 128$ iterations is 0.029. The remaining values present in Table 3 are within their 95% confidence interval, giving good confidence in the accuracy of our analytical method and therefore our results.

## 8 Related work

The $(m_i, k_i)$ constraints have been used before to improve the efficacy of fault-tolerance in real-time scheduling. [8] proposed a scheduler and an efficient schedulability algorithm for a sporadic task set with tasks under $(m_i, k_i)$ constraints. Their scheduler allows for scheduling task sets that would normally not be schedulable, yet utilizing their $(m_i, k_i)$ constraints allows them to be scheduled.

Table 3: Numerical evaluation using UPPAAL

| **DAG** | **# tasks** | $\lvert\mathcal{S}\rvert$ | **# faults** | $\delta(\Delta)$ **(numerical)** | $\delta(\Delta)$ **(analytical)** |
|---|---|---|---|---|---|
| dag0 | 19 | 1 | $18.9766 \pm 1.543$ | $(4.633 \pm 0.377) \cdot 10^{-3}$ | $4.34 \cdot 10^{-3}$ |
| dag1 | 14 | 17 | $4.32812 \pm 0.574086$ | $(1.057 \pm 0.141) \cdot 10^{-3}$ | $1.01 \cdot 10^{-3}$ |
| dag2 | 21 | 21 | Did Not Finish | | $3.14 \cdot 10^{-3}$ |
| dag3 | 20 | 1 | $21.9219 \pm 2.09571$ | $(5.352 \pm 0.512) \cdot 10^{-3}$ | $5.25 \cdot 10^{-3}$ |
| dag4 | 4 | 1 | $0.453125 \pm 0.113753$ | $(1.106 \pm 0.278) \cdot 10^{-4}$ | $1.09 \cdot 10^{-4}$ |
| dag5 | 5 | 4 | $0.40625 \pm 0.112856$ | $(9.918 \pm 2.755) \cdot 10^{-5}$ | $9.36 \cdot 10^{-5}$ |
| dag6 | 10 | 10 | $1.64844 \pm 0.272017$ | $(4.025 \pm 0.664) \cdot 10^{-4}$ | $3.78 \cdot 10^{-4}$ |
| dag7 | 4 | 1 | $0 \pm 0$ | $(0 \pm 0) \cdot 10^{0}$ | $5.65 \cdot 10^{-8}$ |
| dag8 | 12 | 26 | Did Not Finish | | $3.67 \cdot 10^{-4}$ |
| dag9 | 4 | 2 | $0.148438 \pm 0.0661748$ | $(3.624 \pm 1.616) \cdot 10^{-5}$ | $4.46 \cdot 10^{-5}$ |

Chen et al. [7] proposed a solution that is 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. propose 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.

[10] offers 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 $\delta(\Delta)$ function. Their technique hopes to provide transparency to system designers, allowing analyzing the impact on the reliability 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 them. 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 [13] 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 $\lambda$ 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 each strategy appears as is own a criticality level, it is a level applied to a subset of the tasks (specifically $\Gamma_s$). Finally, [13] 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.

## 9 Conclusion

In this paper, we introduced *strategy switching*, a technique to improve fault-tolerance for resource-constrained systems. By switching the subset of the set of tasks that receives fault-tolerance, we are able to reduce the effective fault-rate for resource-constrained weakly-hard real-time systems. We contribute a comprehensive algorithm for constructing the strategy state machine, as well as an evaluation of our technique across 500 DAGs. In our evaluation, we saw an improvement in the majority of cases when resource constraints are significant. Furthermore, we contribute an analytical technique for analyzing the strategy state machine, and use UPPAAL to validate our technique.

## 10 Future work

We hope to address the issue of tractability in future work, as well as lower the steady-state fault rate of applications by means of an improved linking algorithm. Finally, we hope lift the limitation of $(m_i, k_i)$ where $k_i \leq 2$ in a future paper.

Tractability may be improved by utilizing symmetry in the strategy set $\mathcal{S}$, as well as leveraging heuristic-driven strategy enumeration techniques. Furthermore, we intent to lower the steady-state fault rate of our strategy switching solution by developing a new divide-and-conquer linking algorithm. When not encountering any faults, the lowest steady-state fault rate is achieved by switching between two strategies or remaining in one strategy. This is a logical consequence of the $k_i = 2$ limitation, as it limits the effects of an unmitigated faults to two iterations (two strategies). By identifying these pairs and devising a merge operation, we hope to construct a high-quality composite strategy state machine.

Finally, we aim to support tasks with $k_i > 2$ constraints. The past $k - 1$ successes or fails of a task is needed in $\delta(s,r)$ to compute the expected number of catastrophic faults when evaluating successor strategy $s$. As such, a result should be allocated for each combination of previous fails/successes. Naively allocating these results is trivially intractable. Instead, we hope to create an efficient result enumeration and linking algorithm that can operate at runtime.

## Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 871259 (ADMORPH project). We thank the reviewers for their suggestions on improving this paper.

# Bibliography


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

[2] Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL—a tool suite for automatic verification of real-time systems. Springer (1995)

[3] Bernat, G., Burns, A., Liamosi, A.: Weakly hard real-time systems. IEEE transactions on Computers **50**(4) (2001)

[4] Broster, I., Burns, A., Rodriguez-Navas, G.: Timing analysis of real-time communication under electromagnetic interference. Real-Time Systems **30**(1-2) (2005)

[5] Bulychev, P., et al.: UPPAAL-SMC: Statistical model checking for priced timed automata. arXiv e-prints (2012)

[6] Chang, J., Reis, G.A., August, D.I.: Automatic instruction-level software-only recovery. IEEE (2006)

[7] Chen, K.H., Bönninghoff, B., Chen, J.J., Marwedel, P.: Compensate or ignore? Meeting control robustness requirements through adaptive soft-error handling. LCTES 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2907950.2907952

[8] Choi, H., Kim, H., Zhu, Q.: Job-class-level fixed priority scheduling of weakly-hard real-time systems (2019). https://doi.org/10.1109/RTAS.2019.00028

[9] Dick, R., Rhodes, D., Wolf, W.: TGFF: Task graphs for free (1998). https://doi.org/10.1109/HSC.1998.666245

[10] Gujarati, A., Nasri, M., Brandenburg, B.B.: Quantifying the resiliency of fail-operational real-time networked control systems. Leibniz International Proceedings in Informatics (LIPIcs), vol. 106. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.ECRTS.2018.16

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

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

[13] Pathan, R.M.: Fault-tolerant and real-time scheduling for mixed-criticality systems. Real-Time Systems **50**(4) (2014)