Reliability Analysis for MPSoCs with Mixed-Critical, Hard Real-Time Constraints

Philip Axer, Maurice Sebastian, Rolf Ernst
Institut für Datentechnik und Kommunikationsnetze
Technische Universität Braunschweig, Germany
{axer, sebastian, ernst}@ida.ing.tu-bs.de

ABSTRACT

Methods such as rollback and modular redundancy are efficient to correct transient errors. In hard real-time systems, however, correction has a strong impact on response times, also on tasks that were not directly affected by errors. Due to deadline misses, these tasks eventually fail to provide correct service. In this paper we present a reliability analysis for periodic task sets and static priorities that includes realistic detection and roll-back scenarios and covers a hyperperiod instead of just a critical instant and therefore leads to much higher accuracy than previous approaches. The approach is compared with Monte-Carlo simulation to demonstrate the accuracy and with previous approaches covering critical instants to evaluate the improvements.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: fault tolerance; C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

General Terms

Real-Time, Fault Tolerance, Embedded Systems

1. INTRODUCTION

Driven by tight power, performance and cost constraints, embedded system designers are adopting the trend towards multi-processor systems-on-chips (MPSoC). MPSoCs are being or will be used in all embedded application domains like telecommunication (e.g. smart phones), transportation (e.g. car electronics) and industrial automation (e.g. process control). In all domains, it is appealing to use the vast amount of available resources on MPSoC platforms for efficient consolidation of functionalities. Thus, applications of different nature such as control tasks (e.g. anti-lock braking system - ABS), data-oriented processing (e.g. video-based collision detection) and best-effort applications (e.g. entertainment) can be combined on the same MPSoC platform. These applications have *mixed-critical* requirements. Some applications may have hard real-time constraints or are safety-critical whereas others are non-critical at all (e.g. best-effort entertainment)

Since it is predicted that the error-rate will increase caused by continuous transistor shrinking [3], the effect of errors will become even more important than it is today. In the best case this only impairs the consumer experience, whereas in safety-critical applications we might face high casualties in case of failure.

Conservative approaches to increase reliability through fault-tolerance are dual modular redundancy (DMR) or triple modular redundancy (TMR) on system-level. Here, the entire system such as the MPSoC itself is replicated, which always impose a large overhead. In case all functions are of high criticality, system-level redundancy is inevitable, but for mixed-critical systems all non-critical functions are also replicated, causing unnecessary overhead.

Appealing mechanisms to circumvent this problem are spatial and temporal redundancy at the task- or function-level to eliminate unwanted overhead. In this case, only critical functions are replicated and non-critical functions remain untouched. This is especially interesting in MPSoCs because they inherently offer spatial redundancy in the form of cores at very low cost.

Of particular interest are real-time applications. Here, the correctness of behavior depends not only on the logical result, but also on the instant in time at which the result is produced [11]. Guaranteeing real-time properties is especially challenging for computer systems which are working under the influence of errors and error-recovery must not jeopardize response times of critical tasks. Redundancy at the function-level aggravates the situation: Due to the complex interaction of multi-core scheduling, error detection and recovery, it is challenging to give timing guarantees especially in the presence of errors. Without appropriate reliability analysis which also considers real-time properties it is not possible to use a fine grained redundancy approach for real-time applications at all, simply because it would be impossible to estimate the relevant key properties.

Obviously, it is not possible to guarantee the total absence of failures during mission time, even by using a very high degree of redundancy. A common approach is to estimate the mean time to failure (MTTF) and design fault-tolerance mechanisms in a way that the MTTF is sufficiently large and the overhead as low as possible.

Certification agencies have introduced safety standards for design, implementation and maintenance of safety-critical systems. For instance the IEC 61508 [9] is a generic safety standard with various derivatives e.g. ISO 26262 for the automotive domain. It specifies a risk analysis which maps safety functions to Safety Integrity Levels (SIL 1 - SIL 4) depending on their risk (criticality). Each safety integrity level is associated with a bound for the "*probability of dangerous failure per hour*". During the design process, it has to be ensured that the probability of failures per hour which will be observed in the field must not exceed the bound specified by the SIL.

Given that functions on a MPSoC are mostly implemented as a mapped task graph it seems reasonable to anticipate this metric on a task-level and assign criticality constraints to tasks. Critical tasks have to sustain significantly longer than non-critical tasks, thus we consider task-level redundancy, checkpointing and rollback as standard approaches to detect and recover from errors.

The first question throughout this paper is how to model checkpointing and fine-grained task redundancy under the effects of errors and which system architectures support such assumptions. The second question is what are the effects of errors (e.g. re-executions) on other tasks in the system and what is the likelihood that a task misses a deadline due to transient load caused by recovery of erroneous tasks. This will lead to a task-wise reliability analysis which allows to validate criticality constraints (such as SIL) for tasks individually. The goal is to derive the reliability as tight as possible to reduce over-provisioning to a minimum. This allows design and certification of mixed-critical applications running on the same hardware without massive over-provisioning.

The rest of the paper is structured as follows: Firstly, we will discuss related work. In Section 3 we will discuss the general system architecture as well as error and application model. In Section 4 we will present the foundation of the formal reliability analysis. Here we describe a practical method to obtain a task-wise reliability metric. Experiments and comparison with simulative approaches are summarized in Section 5. Finally, we conclude our work in Section 6.

2. RELATED WORK

Related work in the area of fault-tolerant system analysis focus either on logical or on temporal correctness of systems or components. Whenever formal techniques are introduced to compute the reliability with respect to timing constraints, the probability of logical failures is neglected. Similarly considering fault tolerance mechanisms with respect to logical correctness does not take timing effects into account. In this section we will focus on the research of analyzing timing failures.

Baruah et al. describe a method in [2] to account for mixed criticality in real-time systems. They assume that task characteristics such as the worst-case execution time (WCET) are annotated with a certain confidence (criticality level). The response time analysis for a particular task is then carried out in consideration of the interference based on the criticality level. However, fault-tolerance mechanisms and reliability in particular are not considered.

Izosimov et al. [10, 13] suggest another analysis methodology for fault-tolerant real-time systems. In this work re-execution and replication are treated as design alternatives. They assume static execution order scheduling within a MPSoC. Based on this assumption they derived the maximum number of tolerable errors that can be corrected in time. Furthermore, they optimized this number by combining re-execution and replication corresponding to a heuristic optimization algorithm. The drawback is the assumptions of static execution order scheduling which limits the applicability in real-life systems.

In [5, 6] Burns et al. presented an extension of the worst-case response time analysis to incorporate re-execution. This method gives upper reliability bounds extrapolated from the critical instant, which is the worst-case activation scenario where all tasks are released simultaneously. Also in [14] Punnekkat et al. extended this work to derive an optimal checkpointing strategy. In [4] an analysis methodology was introduced focusing on re-transmission of data frames on CAN buses. However, the presented approach is based on pessimistic assumptions. On the one hand only the worst-case scenario (critical instant) is considered, i.e. the situation in which a task suffers from as much workload as possible, where other activations may accommodate significantly more errors. Additionally, it does not explicitly cover error detection mechanisms as an integral part which may delay response times.

We extend the work presented in [15]. Here a similar reliability analysis for message channels of a single CAN bus, considering periodic task-sets scheduled by non-preemptive fixed priority scheduler. To bound the reliability as tight as possible a job-wise analysis approach is introduced, where each job in the hyperperiod, which is the time in periodic systems after which the activation pattern starts to repeat itself, is analyzed. In a second step, the results are composed to a task reliability function. In this paper this approach will be extended to cover multi-core SoCs using redundancy in time and space to detect and correct errors.

3. PRELIMINARIES

3.1 Error Model

Problematic for hard real-time systems are soft errors, due to their strong impact on timing and logical correctness which is discussed in detail in the following sections. The handling of permanent errors is considered as an orthogonal problem which is tackled with other approaches. Permanent errors are covered by e.g. [8]. For the proposed analysis we restrict the scope to soft errors.

Soft errors caused by radiation or variability will eventually manifest as logical bit flips in architectural registers. The occurrence of errors on a core is modeled using Poisson processes with a given error-rate λ_i per core. This models error-events on core components such as ALU and FPU. Specifying per-core error rates accounts for architectures in which some cores may be inherently more reliable than others. Error rates can be reduced by low-level hardware mechanisms (e.g. [1]) or by using more reliable but low-performance process parameters.

For the Poisson model, the following equations give the probability for correct execution of the i-th processor during the time interval Δt and the converse probability that at least one error occurred.

$$P(\textit{no error in time } \Delta t) = e^{-\lambda_i \Delta t} \quad (1)$$

$$P(\textit{errors in time } \Delta t) = 1 - e^{-\lambda_i \Delta t} \quad (2)$$

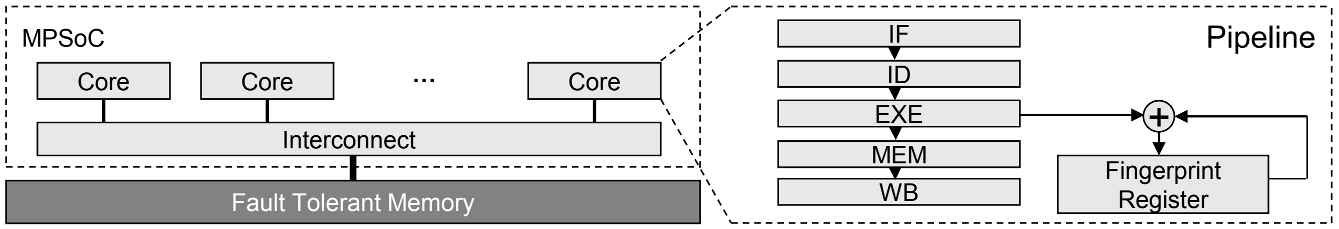

Figure 1: System architecture: multiprocessor system with fault-tolerant memory attached. All cores include a fingerprinting mechanism that track the execution stream in the pipeline.

3.2 System Model

We consider a multi-processor systems-on-chips as shown in Figure 1 consisting of a set of processors $\mathcal{P}$. The MPSoC executes an application consisting of tasks as described later. Processors communicate over a local interconnect which is reliable, meaning it is capable to detect and correct errors on its own. We also assume that the time required for detection and recovery for interconnect-errors is negligible short. These are valid assumptions for a crossbar switch or a bus, where Error Detection or Error Correction Codes (EDC, ECC) can be used. Also, to keep the presented analysis clear, we assume that there is no additional communication overhead which needs to be considered, instead all overhead is implicitly attributed to task properties as later discussed in Section 3.3.

Once affected by a soft error, a non-fault-tolerant processor may fail in two ways: fail-silent or byzantine. A fail-silent failure will result in a "hanging" execution of the current task, but the processor still responds to interrupts. After byzantine failure a task might generate erroneous output, suggesting correct operation. We assume that in both cases, the affected processor will not interfere with operations carried out on other processors.

We assume that in regular, predefined intervals, the application triggers the creation of a checkpoint. The checkpoint includes all recently changed memory regions of the tasks address space, as well as the current register contents. When a checkpoint is established, the processor writes the relevant memory content to a fault-tolerant memory region as depicted in Figure 1. This can be protected main memory (e.g. by using an ECC) as well as dedicated checkpoint memory, e.g. as used in the SafetyNet [17] approach. Furthermore, we assume that the creation of a checkpoint is an atomic transaction and the hardware circuitry (e.g. DMA controller) takes care that the checkpointing process itself is fault tolerant. Once an error in the processor is detected, the most recent checkpoint is restored, and task execution is resumed from this point. Generally, the imposed assumptions are compatible with most of the checkpointing rollback approaches which are summarized in [18].

Until now, we only have discussed recovery mechanisms. To detect errors, we employ a fingerprinting mechanism as presented by Smolens et al. [16] and LaFrieda et al. [12]. The general concept is to calculate a checksum of the execution stream inside the pipeline as shown in Figure 1. In this particular example we only include the execution stage, but it is also possible to include other pipeline stages.

Error detection using fingerprinting works as follows: For some given input in form of an algorithm (program) and some data, the fingerprint is determined by the instruction and data sequence imposed by the program. As long as program and data are the same, the produced fingerprint will always be the same, unless there are errors. By memorizing this fingerprint it is possible to compare it with the fingerprint of a reference execution on another or even the same processor in order to decide on the logical correctness. In case fingerprints don't match, an error must have happened, given that the input data and the programs were the same. This approach has the advantage that state comparison is very inexpensive in terms of time and bandwidth. Additionally, the fingerprint has an arbitrary good error detection coverage (depending on the hash algorithm).

3.3 Application Model

We model an application as a set Γ of independent tasks τ_i, running on a set of processors $\mathcal{P}$. The task set consists of fault-tolerant (ft-tasks) and non-fault-tolerant tasks. A non-fault-tolerant task τ_i is defined by the following tuple: $\tau_i = (T, C, D, p)$. Tasks are periodically activated every T time units, execute for at most time C and must have finished execution before their deadline $D \leq T$. Tasks are scheduled by a preemptive scheduler using a fixed priority p. The job $\tau_{i,j}$ is the j-th incarnation of task τ_i.

Ft-tasks are extensions of regular tasks which are replicated among several cores to increase reliability. As a detection and recovery technique, fine grained redundancy, checkpoint and compare mechanism are used as described in section 3.2. To reflect the redundant processor mapping a task can be mapped to multiple processors denoted by the mapping-set $\mathcal{M} \in \mathcal{P}^\beta$ which maps a task to an arbitrary set of processors, where β denotes degree of redundancy (e.g. $\beta = 2$ corresponds to task-level DMR). For ft-tasks the priority attribute is implicitly assumed to be a tuple of priorities, one for each core in $\mathcal{M}$. Note, that a task can be mapped twice on the same processor, in this case redundancy in time is be used. In most cases β will equal two, otherwise a more efficient majority voting can be employed.

Checkpointing and redundancy imposes additional execution time overhead described by parameters n, t_{cov} and t_{rov}. For a ft-task, the execution time C is divided into n checkpoints of arbitrary length so that $C = \sum t_{cp_i}$. At the beginning of each execution interval, a checkpoint is established causing a *creation overhead* of t_{cov}. At the end of each execution segment, fingerprints of all redundant execution streams are compared. In case of an error, all redundant copies re-execute the recent execution interval with an additional *recovery overhead* of t_{rov}.

Each instance of a ft-task is modeled as a set of primitive (non-ft) tasks with precedence constraints, as shown in Figure 2. The precedence constraints reflect the error detection mechanism: To begin a new checkpoint segment, the correctness of the preceding checkpoint has to be validated by comparing fingerprints of all segments in a *checkpoint group*. A checkpoint group $CG_{i,j,l}$ is the set of primitive jobs in the l-th stage of the precedence graph belonging to job $\tau_{i,j}$.

In the error-free case, there are exactly n_i checkpoint groups, one for each checkpoint. In case that errors have corrupted the execution, the total number of checkpoint groups for the instance of task τ_i is increased by the number of erroneous checkpoint groups. Naturally, the amount of erroneous checkpoint groups is not known a priori, hence we don't know the actual phenotype of the precedence model describing a job $\tau_{i,j}$. But the effects of a given amount of errors on a ft-task can be modeled by using the appropriate precedence model.

Since we focus on hard real-time systems, we are particularly interested if tasks adhere to their deadline constraint under error scenarios. The response time t_r of a job is the difference between its finishing time and its release time, which is the time when the task becomes ready. In that way the response time covers the entire time an event has been queued and the time required for processing. A job has missed its deadline if the response time is greater than the deadline: $t_r > D$. Otherwise the job meets its deadline. The response time t_r of a ft-job $\tau_{i,j}$ is defined as the latency from the activation of the virtual job $T_{scatter}$ until finishing of T_{gather} as shown in Figure 2.

3.4 Error Handling

An error can affect a task in a number of ways, eventually causing a task to logically fail. Unprotected tasks are assumed to fail on the first error, which is a conservative assumption based on the possible failure modes of the processor. For tasks which are replicated, error events can occur during the following execution phases:

1. checkpoint creation
2. regular execution
3. recovery process

For the first case we assume that an error during checkpoint creation will be detected and corrected on-the-fly by the checkpointing subsystem as discussed in section 3.2. For the second case the error will be detected after all segments in $CG_{i,j,l}$ have finished. It might happen that due to errors a task gets stuck in a loop and will not yield processing-time properly. We assume that the operating system implements a budgeting mechanism which enforces maximal execution times e.g. by using a watchdog. This kind of budgeting is for instance used in PharOS [7]. Then the recovery process

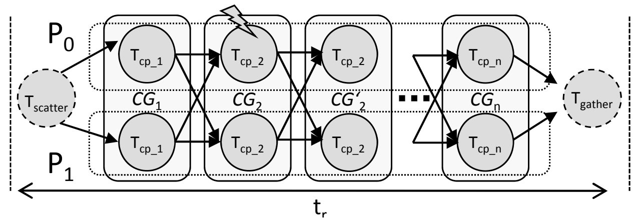

Figure 2: Equivalent precedence graph of a fault-tolerant task τ_i with an error in CG_2.

is triggered with the result that the recent checkpoint group is re-executed. The third scenario is treated like the second, assuming that an error during the recovery process will not corrupt the recent checkpoint. By these means it is possible to detect errors at the end of each task to avoid error propagation in the system (domino effect). Errors will have no effect on tasks when a core is idling.

Figure 3 shows an example Gantt chart as a possible execution trace of a system with two cores and tasks $T_1 - T_4$, priorities are assigned according to the occurrence in the chart, e.g. T_1 has a higher priority than T_2 and so on. In this example T_2 is a fault-tolerant task ($n = 2$, $\beta = 2$) which is replicated on both cores. The segmentation of the fault-tolerant task T_2 into checkpoint groups is highlighted by $CG_{2,1,1} \ldots CG_{2,2,2}$. Important events have been numbered in the order of their occurrence: (1) All tasks activate simultaneously. Both replicas of T_2 start creating a checkpoint. (2) An error event e_1 affects task T_1, causing this task to fail. Further activations of T_1 may still dissipate processor bandwidth but have no further value for the service which T_1 delivers. (3) Fingerprints of all primitive jobs in $CG_{2,1,1}$ are available and match, a checkpoint is established. (4) An error event affects one primitive job of T_2 in $CG_{2,1,2}$. (5) Comparison of fingerprints in $CG_{2,1,2}$ does not match, recovery initiated. A re-execution $CG'_{2,1,2}$ follows. (6) Fingerprint of $CG'_{2,1,2}$ after re-execution matches. First activation of T_2 successfully completed and deadline met. (7) Second activation of T_2 arrives, checkpoint is established. (8) Fingerprints in $CG_{2,2,1}$ available and match, a checkpoint is established. (9) Fingerprints in $CG_{2,2,2}$ available and match. Second activation of T_2 successfully completed and deadline met.

The recovery action delays the response time of T_2 and T_4. In this particular example all jobs still meet their deadlines, although T_1 failed its service. In case more than one error affect T_2, then T_4 and T_2 will both miss their deadlines. The goal of a reliability analysis is to derive probabilistic values for these kind of scheduling conditions under which jobs miss deadlines due to errors.

3.5 Reliability Metric

The lifetime of a task is modeled with a random variable L that represents the time the task is running until the first failure occurs. The distribution function of L is given by F_L. Throughout this paper, we will use the common notion of reliability $\mathcal{R}$ as a metric which is the complementary of F_L. Practically $\mathcal{R}(t)$ denotes the probability that a task is still operating without failures in the interval $[0, t]$ [11].

$$\begin{aligned}\mathcal{R}(t) &= P(\text{no failure in interval } [0,t]) \\ &= 1 - F_L(t) \qquad (3)\end{aligned}$$

Definition 1. (Failure):
A failure results from either timely or logical incorrectness.

As highlighted in the previous section, a logical incorrectness originates from an unrecoverable error, whereas a timing incorrectness arise from additional workload as the result of the recovery process which may interfere with lower priority tasks and violate the real-time constraint $t_r \leq D$. The goal of the formal analysis is to derive $\mathcal{R}_i(t)$ for each task from which we can easily calculate other common metrics

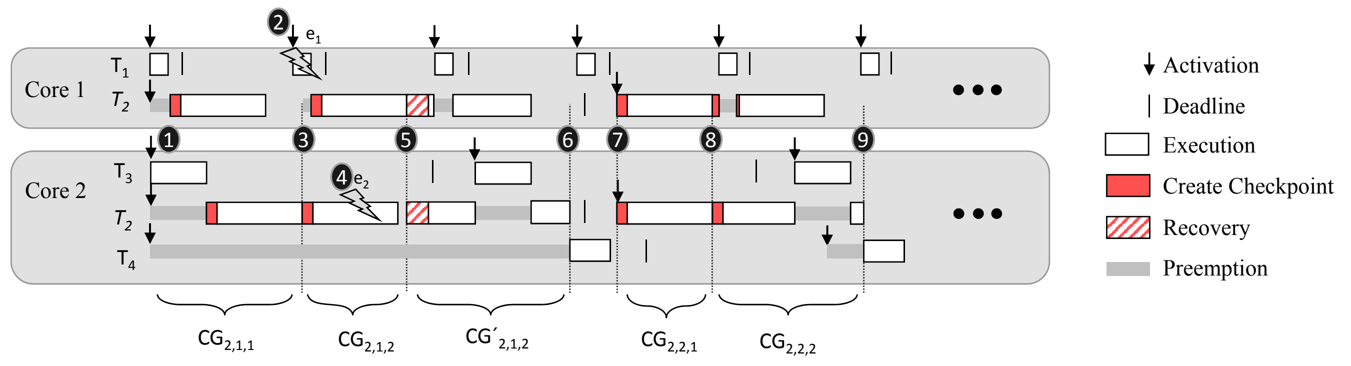

Figure 3: Illustrative example: Task T_2 is redundantly mapped to both cores and split in $n = 2$ checkpoints.

such as the MTTF:

$$MTTF_i = \int_0^\infty \mathcal{R}_i(t)\, dt \tag{4}$$

4. FORMAL ANALYSIS

In this section we will present an algorithm which allows accurate computation for $\mathcal{R}_i(t)$ for arbitrary values of t. Since determining the reliability is an average case problem, we reject the common worst-case approach as used by [5] for our analysis. Instead of considering a task-wise approach, we consider each job individually. By this, a more precise result can be obtained by taking the interference characteristic of each job into account. For this approach we divide the problem into timing-induced errors and logical errors. First, we want to evaluate timing constraint violations caused by recovery and re-execution.

In a periodic task-set, the activation pattern will appear repetitively after the hyperperiod of length $lcm(T_1, \ldots, T_n)$. Thus, it is sufficient to consider all activations in a hyperperiod and extrapolate from these results. For the formal analysis, we will not focus on the calculation of the response times for individual jobs. Methods for response time calculations are well known [19] and can be adapted to the task model that is used in this approach. Thus, given a task set, it is possible to calculate response times of all jobs in a hyperperiod. Now we will introduce some important definitions:

Definition 2. (Success): The fact that the j-th activation of task τ_i is logically correct and meets its deadline will be referred to as $S_{i,j}$.

Definition 3. (Success Probability): The success probability $P[S_{i,j}]$ is the probability that job $\tau_{i,j}$ succeeds.

Although, the actual response time calculation is an orthogonal problem it is necessary to discuss some properties: It is important to note that it is of no relevance for the response time of a ft-task which erroneous primitive job in a checkpoint group *exactly* caused additional delay. The ultimate effect of errors in $CG_{i,j,l}$ is always the same, regardless of the actual faulty sub-job. This can be seen in Figure 3, if e_2 would have hit the corresponding job on core 2, the outcome would have been the same.

For the analysis, it is important to know which jobs interfere with another job. The set of tasks and thus the set of jobs can be obtained by the *timing dependency graph* which indicates the functional and non-functional dependencies between the tasks in the example from Figure 3. The nodes of the timing dependency graph correspond to the tasks in the system and the directed edges represent functional and non-functional dependencies between tasks. Functional dependencies are given through the task graph and non-functional dependencies arise from the local scheduling on a processor. A particular job $\tau_{i,j}$ may be delayed by jobs of higher priority than $\tau_{i,j}$ which are released prior to $\tau_{i,j}$ running on the same resource as $\tau_{i,j}$ (non-functional dependency). Due to precedence constraints (functional dependency), a job $\tau_{i,j}$ can also be delayed by jobs running on another core. We are not interested in the primitive jobs which may delay a given job, but instead in the corresponding checkpoint groups which may delay a job.

Definition 4. (Interference Set): The interference set $\xi_{i,j}$ is the set of all checkpoint groups CG which potentially delay job $\tau_{i,j}$.

Note that if $\tau_{i,j}$ itself is a ft-task its own $CG_{i,j,l}$ are always in $\xi_{i,j}$. Errors in non-ft tasks have no timing effect on other tasks and therefore do not need to be considered in the interference set. The cardinality of $\xi_{i,j}$ is arbitrarily large. Practically we limit the set to a given number of checkpoint groups which have been released time d before the release of $\tau_{i,j}$, thus for large d we gain arbitrarily good estimates of $\xi_{i,j}$. By this, we exploit that jobs which are released sufficient time before $\tau_{i,j}$ will not interfere with $\tau_{i,j}$.

Choosing a too small value for d may lead to overestimation, leading to non-conservative results due to additional interference which is not covered throughout analysis. By analyzing for increasing values of d the result will converge towards the actual reliability.

Based on the given definitions, we can express the reliability of a task τ_i as the the probability that all jobs $\tau_{i,1} \ldots \tau_{i,j}$ have succeeded their execution that have been released in the interval $[0, t]$:

$$\mathcal{R}_i(t) = P[S_{i,1} \wedge \ldots \wedge S_{i,j}] \tag{5}$$

4.1 Feasible Error Scenarios

The algorithm we propose consists of two independent steps that will be carried out for each task: In the first step all *feasible scenarios* for each job of a the task throughout the hyperperiod are enumerated. A feasible scenario is a specific error constellation in which no deadline miss

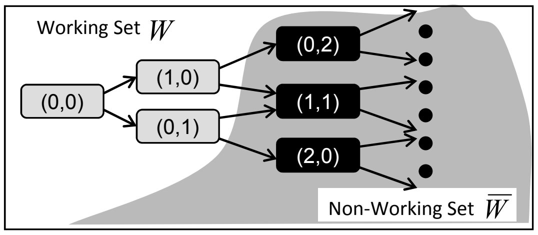
Working Set $\mathcal{W}$
(0,0)
(1,0)
(0,1)
(0,2)
(1,1)
(2,0)
Non-Working Set $\overline{\mathcal{W}}$

Figure 4: Working-set for job $\tau_{4,1}$

(failure) for the task would occurs. In a second step, we transform these scenarios into probabilities through which it is possible to derive the characteristic reliability function $\mathcal{R}_i(t)$.

Coming back to the example in Figure 3, we focus on the scenario enumeration for job $\tau_{4,1}$. The response time of job $\tau_{4,1}$ depends on the re-execution pattern of checkpoint groups which are in the interference set of $\tau_{4,1}$ which is $\xi_{4,1} = \{CG_{2,1,1}, CG_{2,1,2}\}$. For the particular situation depicted, $CG_{2,1,2}$ handles one error (re-execution denoted as $CG'_{2,1,2}$) without causing $\tau_{4,1}$ to miss its deadline. Further analysis would reveal that a scenario where $CG_{2,1,1}$ is exposed to one error would also lead to a feasible schedule, whereas an error in both $CG_{2,1,1}$ and $CG_{2,1,2}$ will cause $\tau_{4,1}$ to miss its deadline. To formalize this concept we will introduce the *error scenarios.*

Definition 5. (error scenario): An error scenario $s^{i,j,k}$: $\xi_{i,j} \to \mathbb{N}_0$ is a function which specifies the amount of re-executions for each checkpoint group $CG \in \xi_{i,j}$.

This function can conveniently be expressed as a tuple, where the n-th component in the tuple denotes the amount of errors for the n-th checkpoint group in the (ordered) interference set $\xi_{i,j}$. The example in Figure 3 shows the error scenario $s^{4,1,1}$ for which the interference set is $\xi_{4,1} = \{CG_{2,1,1}, CG_{2,1,2}\}$. Now, we can simply specifiy the scenario as $s^{4,1,1} = \{CG_{2,1,1} = 0,\ CG_{2,1,2} = 1\}$ or more conveniently $s^{4,1,1} = (0, 1)$.

For each error scenario it is possible to construct an equivalent precedence model (see Section 3.3). This model can then be fed into a response time analysis to verify deadline constraints. Thus, each scenario can either lead to a feasible schedule or to an infeasible schedule, depending on the response time analysis of the error scenario model. We can summarize that $\tau_{4,1}$ will meet the deadline for the scenarios $\{(0, 0), (1, 0), (0, 1)\}$. Figure 4 shows exactly these feasible scenarios which we will call working set.

Definition 6. (Working Set):
The working set $\mathcal{W}_{i,j}$ of a job $\tau_{i,j}$ is the set of error scenarios $s^{i,j,k}$ for which job $\tau_{i,j}$ is guaranteed to meet its timing constraint.

Originating from the error-free case as the root node, we can construct an error graph as shown in Figure 4. In this graph, edges are error events whereas nodes are error scenarios. The graph can be built by using depth first search: For each node we perform a response time analysis and evaluate the real-time constraint $t_r < D$. Each node is then colored with respect to the schedulability of the error scenario which it presents. In case the node is schedulable, we recursively evaluate all successor nodes in the same way until we found all working scenarios. All nodes which are marked *feasible* form the working set $\mathcal{W}_{i,j}$.

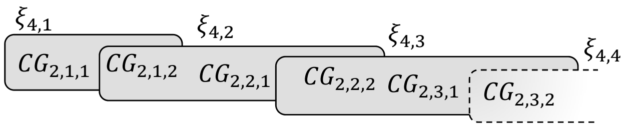

Figure 5: Illustrative example of some interference sets for task τ_4 which are mutually overlapping, causing stochastic dependence for scenarios.

Practically, the graph can be arbitrarily large but it is sufficient to enumerate only a sub-graph. By doing so, we obtain a subset of the real working set which is conservative because all other nodes will be considered as non-working. A pessimistic approach which only considers the worst-case activation would only yield to one particular $\mathcal{W}_i$ which resembles the situation with the least amount of working scenarios.

The success probability of job $\tau_{i,j}$ can be expressed as the probability that one scenario of all working scenarios in $\mathcal{W}_{i,j}$ will actually occur:

$$P[S_{i,j}] = P[s^{i,j,1} \vee \ldots \vee s^{i,j,k}],\ s^{i,j,1}, \ldots, s^{i,j,k} \in \mathcal{W}_{i,j} \quad (6)$$

4

4.2 Probability Computation

Once we have obtained all working sets for all jobs in the hyperperiod we can derive success probabilities. It is not sufficient to calculate scenario occurrence probabilities based on individual working sets alone. The scenarios from successive jobs of the same task are not mutually independent. The reason for this is that consecutive interference sets include common checkpoint groups: $\xi_{i,j} \cap \xi_{i,j+1} \neq \emptyset$. This is also depicted in Figure 5. Note that the interference sets are artificially truncated for the sake of simplicity.

This problem becomes obvious in the following simple example: We assume that the working sets for $\tau_{4,1}$ and $\tau_{4,2}$ have been determined independently in a previous step. Now, since $\xi_{4,1}$ and $\xi_{4,2}$ overlap, job $\tau_{4,2}$ will only see manifestations of $CG_{2,1,2}$ that led to a feasible schedule for job $\tau_{4,1}$, namely zero or one repetitions of $CG_{2,1,2}$ (cf. Figure 4). To express these mutual dependencies, we introduce conditional success probabilities.

Definition 7. (Conditional Success Probability): The conditional success probability is the probability that job $\tau_{i,j}$ meets its deadline given that previous jobs of τ_i have already met their deadlines.

$$P[S_{i,j} | S_{i,j-1} \wedge \ldots \wedge S_{i,1}] \quad (7)$$

By using conditional probabilities, it is possible to express the reliability function (cf. Equation 5) by simple multiplication:

$$\begin{aligned} \mathcal{R}_i(t) = \quad & P[S_{i,1}] \cdot P[S_{i,2} | S_{i,1}] \cdot \ldots \\ & \cdot P[S_{i,j} | S_{i,j-1} \wedge \ldots \wedge S_{i,1}] \end{aligned} \quad (8)$$

The remaining challenge is to derive the conditional success probabilities used in equation 8. We can be sure that job $\tau_{i,j}$ will meet its deadline if the actual error scenario is in $\mathcal{W}_{i,j}$. Beyond that, it is also necessary that previously activated jobs of task τ_i have successfully terminated. Because all scenarios in $\mathcal{W}_{i,j}$ are mutually exclusive, equation 7 can be written as a sum of conditional scenario probabilities:

$$P[S_{i,j}|S_{i,j-1} \wedge \ldots \wedge S_{i,1}] = \sum_{s \in \mathcal{W}_{i,j}} P\left[s|S_{j-1} \wedge \ldots \wedge S_1\right] \quad (9)$$

Remember, we are only interested in the time until the first failure. We process jobs in the order of their activation. Hence, when we look at an arbitrary success $S_{i,j}$ this always includes the fact that all jobs released before $\tau_{i,j}$ have succeeded. From the perspective of $S_{i,j}$, the history of working scenarios forms a complete probability space. Also, when we process job $\tau_{i,j}$ to calculate $P[S_{i,j}|S_{i,j-1}, \ldots, S_{i,1}]$, we know the conditional probability of the predecessor job $P[S_{i,j-1}|S_{i,j-2}, \ldots, S_{i,1}]$ and the probabilities of all scenarios in the working set of $\tau_{i,j-1}$, because they have alreay been evaluated. But as already mentioned in Section 4.1, sometimes scenarios of successive working sets are mutually exclusive. To formalize this we introduce scenario consistency.

Definition 8. (Consistency):
Two error scenarios $s_a = s^{i,j,a}$ and $s_b = s^{i,j-1,b}$ are said to be consistent if there is no $CG \in \xi_{i,j} \cap \xi_{i,j-1}$ so that $s_a(CG) \neq s_b(CG)$.

Practically, this means that consistent error scenarios may contain the same checkpoint group CG in the interference set and if so, the amount of repetitions $s_a(CG)$ and $s_b(CG)$ must be the same, thus s_a and s_b are not contradictory. If two scenarios s_a and s_b are consistent, we can reduce one scenario s_a and remove those checkpoint groups which are already contained in s_b:

Definition 9. (Reduced Scenario):
Given two consistent scenarios s_a and s_b, the reduced scenario $\tilde{s}_a$ is defined as $\tilde{s}_a : \{\xi_a \setminus \xi_b\} \to \mathbb{N}_0$, where $\tilde{s}_a(CG) = s_a(CG) = s_b(CG) \; \forall \; CG \in \{\xi_a \setminus \xi_b\}$

For the example from Figure 5, a reduced scenario $\tilde{s}^{4,2,l}$ from $\mathcal{W}_{4,2}$ would not contain checkpoint group $CG_{2,1,2}$ since this is specified by the predecessor scenario. By application of consistency and reduced scenarios we can calculate the conditional scenario probability as follows:

$$P[s_{i,j,a}|s_{i,j-1,b}] = \begin{cases} P[\tilde{s}_{i,j,a}] & \textit{if } s_{i,j,a}, s_{i,j-1,b} \textit{ consistent} \\ 0 & \textit{otherwise} \end{cases} \quad (10)$$

Then we can put all building blocks together to obtain the final conditional success probability, based on the the scenarios from the previous working set $\mathcal{W}_{i,j}$ which has already been processed in the previous step.

$$\begin{aligned} &P[s_{i,j,k}|S_{i,j-1} \wedge \ldots \wedge S_{i,1}] \\ &= \sum_{s \in \mathcal{W}_{i,j-1}} P[s_{i,j,k}|s] \cdot \tilde{P}[s|S_{i,j-2} \wedge \ldots \wedge S_{i,1}] \end{aligned} \quad (11)$$

Where $\tilde{P}$ is the normalized probability of the scenario s. The probabilites are normalized because all scenarios in $\mathcal{W}_{i,j-1}$ form a total probability space:

$$\sum_{s \in \mathcal{W}_{i,j-1}} \tilde{P}[s|S_{i,j-2} \wedge \ldots \wedge S_{i,1}] = 1 \quad (12)$$

The absolute probability for the entire scenario $P[s]$ is a simple multiplication, since the checkpoint groups in a scenario are independent.

Table 1: Example task-set used for the experiments. All times given in [ms].

Task	T	C	D
T0	300	60	300
T1	250	50	250
T2	100	10	100
T3	300	50	300
T4	600	40	500

$$P[s] = \prod_{CG \in \xi_s} P[R = s(CG)] \quad (13)$$

Equation 13 only considers timing errors and is only valid for ft-tasks. Logical errors which can affect non ft-tasks are currently not considered. In order to integrate the fact that non ft-tasks will not only fail in case of a deadline violation, but also on logical errors, it is necessary to extend the equation by a term (cf. Equation 2) which reflects that the job under analysis will not be hit by an error itself.

$$P_{\textit{non-ft}}[s] = (1 - e^{-\lambda_i C_i}) \cdot \prod_{CG \in \xi_s} P[R = s(CG)] \quad (14)$$

Finally, it is trivial to calculate the probability for certain re-execution patterns by applying Equation 1 and 2 with appropriate values of Δt.

The probability for *exactly* $s(CG_{i,j,l}) = R$ repetitions of a checkpoint group $CG_{i,j,l}$ for a given scenario s can be calculated as follows, where t_e is the execution time of the primitiv jobs in $CG_{i,j,l}$. For the sake of simplicity we assume the error rates for all cores are the same $\lambda = \lambda_p \forall p \in \mathcal{P}$.

$$P[R = 0] = \left(1 - e^{-\lambda(t_{cov} + t_e)}\right)^\beta \quad (15)$$

$$\begin{aligned} P[R = 1] &= (1 - P[R = 0]) \\ &\quad \cdot \left(1 - e^{-\lambda(t_{rov} + t_e)}\right)^\beta \end{aligned} \quad (16)$$

$$\begin{aligned} P[R = r, r > 0] &= (1 - P[R = 0]) \\ &\quad \cdot \left(1 - \left(1 - e^{-\lambda(t_{rov} + t_e)}\right)^\beta\right)^{r-1} \\ &\quad \cdot \left(1 - e^{-\lambda(t_{rov} + t_e)}\right)^\beta \end{aligned} \quad (17)$$

For ft-tasks, the system will try to recover from errors until one successful re-execution took place. Thus, for the probability calculation it is known that all erroneous re-executions are always followed by one correct re-execution.

By inserting the conditional success probabilities in Equation 8, it is possible to obtain $\mathcal{R}_i(t)$. Assuming we have job success probabilities for all jobs in the interval $[0, t_{hyper}]$, with $t_{hyper} = lcm[T_1, \ldots, T_n]$, we can compute the reliability $\mathcal{R}(t_{hyper})$ through Equation 8. From this we can compute the reliability for a given number of A hyperperiods:

$$\mathcal{R}(A \cdot t_{hyper}) = (\mathcal{R}(t_{hyper}))^A \quad (18)$$

5. EXPERIMENTS

For the evaluation of the presented formal analysis we use Monte-Carlo simulation as a reference. We assume that

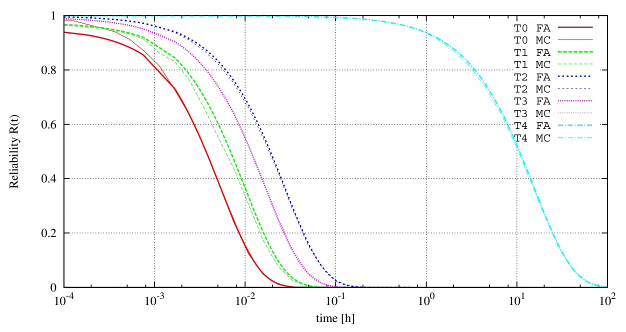

Figure 6: Comparison of formal analysis (FA) accuracy with Monte-Carlo (MC)) reference simulation. $\lambda_{1,2} = 1/sec$, 10.000 samples

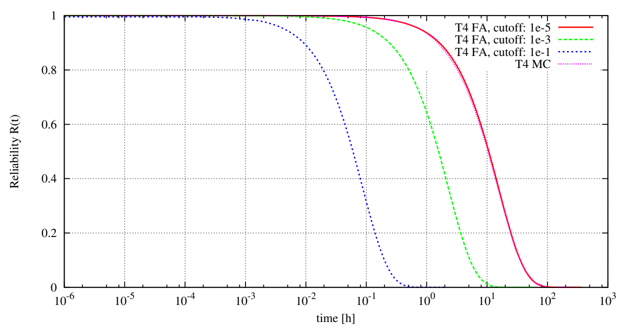

Figure 7: Effect of different working set truncations $\lambda_{1,2} = 1/sec$, 10.000 Monte-Carlo samples

tasks are perfectly synchronized, that means all tasks activate simultaneously at time $t = 0$ and there is no drift of the activation pattern. Under these assumptions, it is possible to calculate the exact response times for each scenario, because there are no uncertainties in scheduling.

For following experiments we use the task set as shown in Table 1. We chose a two-core multiprocessor system and, unless stated otherwise, tasks are mapped according to $M1$ as shown in Table 2. We assigned $T0$ and $T4$ to be fault-tolerant tasks, that means both tasks execute redundantly on core 1 and core 2. Both tasks create two checkpoints with some additional overhead.

Note that the priority assignment and mapping is not optimal in terms of reliability and response times. Rather, the given task-set includes all effects that need to be considered:

- functional dependencies due to precedence constraints of checkpoint groups
- non-functional dependencies caused by scheduling of higher priority tasks
- priority inversion caused by functional dependencies of checkpointing

By our approach it is now possible to efficiently use one multi-core processor where conventional methods such as DMR would require at least two multi-core processors, and thus cause at least 100% overhead. By following our approach, no additional overhead in terms of silicon was added.

5.1 Monte-Carlo-Simulation

For the Monte-Carlo-Simulation, we have implemented a simple static priority preemptive scheduler that will schedule a given task set in the same way, as an operating system scheduler would do. However, we don't schedule real

Table 2: Mapping (M1) of task-set used throughout evaluation.

Task	priority core 1	priority core 2	n	t_{cov}	t_{rov}
T0 (ft)	3	2	2	10	40
T1	4	-	-	-	-
T2	2	-	-	-	-
T3	-	1	-	-	-
T4 (ft)	1	3	2	10	40

tasks but only abstract tokens which represent tasks. For each core in the system, we instantiate one scheduler which is attached to an event-generator. The event-generator will produce error events with an exponentially distributed inter-event time with an average of $1/\lambda_i$. Each time an error event is produced, the scheduler behaves exactly as described in Section 3. The response time of each job is monitored, and failure events caused either by logical incorrectness of regular tasks or timing violations of all tasks are recorded. The reliability function $\mathcal{R}_i(t)$ is then the inverse of the cumulative distribution function of the failure events. By recording a sufficient large amount of failure samples it is possible to approximate the exact reliability with respect to our system and task model.

The comparison of Monte-Carlo approach (MC) and our presented formal analysis (FA) is shown in Figure 6. The graph shows the reliability function $\mathcal{R}(t)$ versus time, which gives the probability that a task is still functioning after time t. The simulation has been carried out for unrealistic error rates to produce a reasonable number of MC simulation runs in acceptable time. For realistic error rates, Monte-Carlo simulation would not be operational due to excessive run times. The reason is that the time per Monte-Carlo run grows with reduced error rates. Unfortunately, error effects are coupled due to their influence on timing. Coupling makes advanced MC techniques such as importance sampling complicated or requires approximations, such as adapting event frequencies, which changes MC results. We use MC simulation only to demonstrate that our approach has very little pessimism. Already for these high error rate, more than 2 hrs computation time were needed. Since the accuracy of our approach is generally independent of the error rate, we may conclude that the accuracy observed in the experiments also holds for realistic error rates.

The reliability analysis was carried out with a search depth value of $d = 12$, higher values increase analysis time, because more scenarios have to be considered and the accuracy improvement is not noticeable. As mentioned in Section 4, it is not possible to list *all* feasible scenarios in the working set for a practical implementation. Thus, the working set is artificially truncated, then potentially working scenarios are considered as non-working which is a pessimistic assumption leading to a conservative result. For this experiment, the working set was truncated when the occurrence probability of a scenario was below a cut-off probability of $1e-12$. This produces equivalent results compared to the

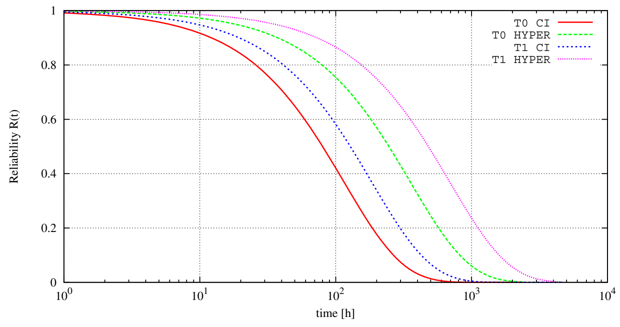

Figure 8: Comparison of hyperperiod-based formal analysis (HYPER) with critical instant based formal analysis (CI). $\lambda_{1,2} = 1/\textit{week}$

Monte-Carlo approach. There is a pessimistic derivation of the formal analysis for very small times $t < t_{hyper}$. The reason for this is that the Monte-Carlo simulator will start executing tasks at time $t = 0$ and jobs in the first hyperperiod have no interference from a previous hyperperiod which could cause deadline violations. Note that the Monte-Carlo result and the formal analysis result converge quickly (see $T0$ in Figure 6).

However, with 17 sec run time, our analysis framework was significantly faster than the processing of 10.000 MC runs which took about 2 hrs. For our approach the run time only depends on the accuracy that shall be achieved and the number of task activations in a hyperperiod.

Figure 7 shows the effect of the working set truncation on the reliability results of task $T4$ from the example task set. We tag all nodes in the working set as not-working which have a smaller occurrence probability than the cutoff probability. This also shows the pessimism which is induced when too small working-sets are considered but also that the curves approach the Monte-Carlo result rapidly.

5.2 Hyperperiod vs. Worst-Case

The presented approach takes advantage of all relative activation phasings which occur during the hyperperiod. Not all activations suffer the same amount of interference compared to the critical instant. Similar to the method presented in [4], we also investigated the approach to consider only the worst-case situation, that is we consider only the activation which causes the worst response time. Figure 8 shows the results of both, the worst-case-based method and the hyperperiod based approach.

Here we evaluated two tasks ($T0$, $T1$) of the example task set. The reliability for the critical instant based method was obtained by taking the worst (smallest) working set of each task. Then the probability calculation was carried out based on the worst working set $\mathcal{W}_i$. The worst-case success probability (WCSP) is obtained by calculating the success probability of the task τ_i for the critical instant:

$$WCSP = \sum_{s \in \mathcal{W}_i} P[s] \tag{19}$$

Then the reliability after one hyperperiod can be extrapolated over all activations in one hyperperiod.

$$\mathcal{R}_i(t_{hyper}) = WCSP^{(t_{hyper}/T_i)} \tag{20}$$

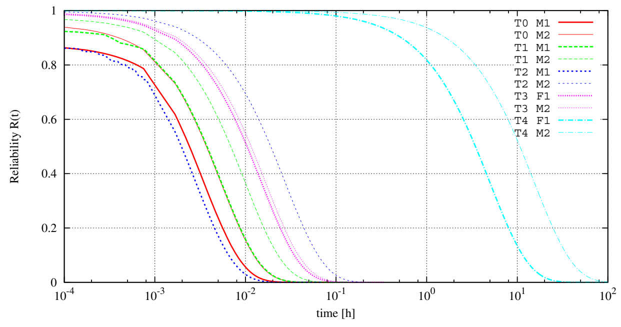

Figure 9: Design space exploration: Reliability of mapping M1 and M2. $\lambda_{1,2} = 1/\textit{sec}$

The comparison shows an improvement for the simple task set of nearly one order of magnitude, but it can be estimated that the difference will be even more drastical for task sets with broader period ranges compared to the simple example presented here, because the effect of relative phasing has more impact.

5.3 Design Space Exploration

By using the presented analysis approach, it is now possible to evaluate alternative mappings of a given task set with respect to the final reliability. This includes different degrees of redundancy, priority assignment or alternative checkpointing implementations which have different overhead trade-offs (e.g. lower checkpoint creation overhead but higher recovery overhead). Investigating the reasons for the results shown in Figure 6, we might find that the priority assignment was not cleverly chosen and we would eventually come up with a different mapping such as $M2$, shown in Table 3. The priorities were chosen in a way that there is no priority inversion among the cores. Also, because $T0$ has a smaller deadline than $T4$, it will be assigned a higher priority. Similarly, we investigate the effect of reducing the amount of checkpoint to $n = 1$.

The resulting reliability functions for mapping $M1$ and $M2$ are shown in Figure 9. As we can see, all reliabilities are improved by the alternative mapping. Notably, the reliability for task $T2$ was increased by nearly a decade and also task $T4$ benefits significantly. However, the reliability of task $T0$, which is a fault-tolerant task, is disappointingly low compared to the other ft-task. If we inspect the size of the working sets for all activations of task $T0$, we find that for all activations there is only *one* working scenario: The error-free scenario. Clearly, task $T0$ has not enough slack time for error recovery. Thus, in order to increase the reliability of $T0$ there are the following options: decrease the execution time, decrease the time for checkpointing, de-

Table 3: Alternative Mapping (M2).

Task	priority core 1	priority core 2	n	t_{cov}	t_{rov}
T0 (ft)	1	1	1	10	40
T1	4	-	-	-	-
T2	2	-	-	-	-
T3	-	2	-	-	-
T4 (ft)	3	3	1	10	40

crease the time for recovery, increase the processor speed, or relax the deadline constraint.

6. CONCLUSION

Upcoming technologies will lead to increased error rates. Even if errors can be corrected, the increased time for handling the recovery process may lead to deadline misses causing a timing-failure in real-time systems. In this paper we have presented a novel algorithm through which we can analyze reliability constraints for a task-set of which some tasks are protected by fine-grained spatial redundancy, checkpointing and rollback mechanisms. This enables legitimate use of such methods on MPSoC platforms.

Contrary to other approaches, we consider a representative hyperperiod and not the worst-case condition because it allows tighter reliability guarantees. The comparison of our algorithm with a reference Monte-Carlo simulation shows very good accuracy with the benefit of significantly shorter analysis time for realistic parameters compared to simulation.

The presented analysis enables the possibility to trade-off different design decisions e.g. priority assignment, amount of checkpoints or degree of redundancy and points out the relevance and potential within resilient system's design.

However, there are some limitations that need to be tackled in future work: Especially, the effects on scheduling of common cause errors need further investigation (e.g. errors on the clock tree). Also MPSoC design usually includes heavy use of shared resources (e.g. common system interconnect and main memory) which is currently not modeled sufficiently. This still leaves room for further research.

7. ACKNOWLEDGMENTS

This work was jointly funded by Advanced Research & Technology for Embedded Intelligence and Systems Joint Undertaking (ARTEMIS JU), the Federal Ministry of Education and Research (BMBF) and the German Research Foundation (DFG) within the project 'RECOMP', support code 01IS10001A.

Also, this work was partly supported by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500). spp1500.itec.kit.edu

8. REFERENCES

[1] T. Austin, D. Blaauw, T. Mudge, and K. Flautner. Making typical silicon matter with razor. *IEEE Computer*, 37(3):57–65, 2004.

[2] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Proc. of Real-Time and Embedded Technology and Applications Symp.*, pages 13–22. IEEE, 2010.

[3] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[4] I. Broster, A. Burns, and G. Rodríguez-Navas. Probabilistic analysis of CAN with faults. In *Proc. of Real-Time Systems Symposium*, pages 269–278. IEEE, 2002.

[5] A. Burns, R. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. In *Proc. of Euromicro Workshop Real-Time Systems*, pages 29–33, 1996.

[6] A. Burns, S. Punnekkat, L. Strigini, and D. R. Wright. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Proc. of Dependable Computing for Critical Applications*, pages 361–378, 1999.

[7] D. Chabrol, C. Aussagues, and V. David. A spatial and temporal partitioning approach for dependable automotive systems. In *Proc. of Emerging Technologies & Factory Automation*, pages 1–8, 2009.

[8] M. Glass, M. Lukasiewycz, F. Reimann, C. Haubelt, and J. Teich. Symbolic reliability analysis and optimization of ECU networks. In *Proc. of Design, Automation and Test in Europe*, pages 158–163, 2008.

[9] International Electrotechnical Commission (IEC). Functional safety of electrical / electronic / programmable electronic safety-related systems, 1998.

[10] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Synthesis of fault-tolerant embedded systems with checkpointing and replication. In *Proc. of Int. Workshop Electronic Design, Test and Applications*, 2006.

[11] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[12] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proc. of Int. Conf. Dependable Systems and Networks*, pages 317–326, 2007.

[13] 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. on VLSI*, 17(3):389–402, 2009.

[14] S. Punnekkat and A. Burns. Analysis of checkpointing for schedulability of real-time systems. In *Proc. of Int. Workshop Real-Time Computing Systems and Applications*, pages 198–205, 1997.

[15] M. Sebastian and R. Ernst. Reliability Analysis of Single Bus Communication with Real-Time Requirements. In *Proc. of Pacific Rim Int. Symp. Dependable Computing*, pages 3–10, 2009.

[16] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatryk. Fingerprinting: bounding soft-error-detection latency and bandwidth. *IEEE Micro*, 24(6):22–29, 2004.

[17] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proc. of Int. Computer Architecture Symp.*, pages 123–134, 2002.

[18] R. Teodorescu, J. Nakano, and J. Torrellas. Swich: A prototype for efficient cache-level checkpointing and rollback. *IEEE Micro*, 26(5):28–40, 2006.

[19] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.