Static Mapping of Mixed-Critical Applications for Fault-Tolerant MPSoCs

Shin-haeng Kang*, Hoeseok Yang**, Sungchan Kim†, Iuliana Bacivarov‡, Soonhoi Ha*, and Lothar Thiele‡
*School of EECS, Seoul National University, Seoul, Korea, {shkang,sha}@iris.snu.ac.kr
**Department of ECE, Ajou University, Suwon, Korea, hyang@ajou.ac.kr
†Division of CSE, Chonbuk National University, Jeonju, Korea, sungchan.kim@chonbuk.ac.kr
‡TIK Laboratory, ETH Zurich, Zurich, Switzerland, {firstname.lastname}@tik.ee.ethz.ch

ABSTRACT

This paper presents a static mapping optimization technique for fault-tolerant mixed-criticality MPSoCs. The uncertainties imposed by system hardening and mixed criticality algorithms, such as dynamic task dropping, make the worst-case response time analysis difficult for such systems. We tackle this challenge and propose a worst-case analysis framework that considers both reliability and mixed-criticality concerns. On top of that, we build up a design space exploration engine that optimizes fault-tolerant mixed-criticality MPSoCs and provides worst-case guarantees. We study the mapping optimization considering judicious task dropping, that may impose a certain service degradation. Extensive experiments with real-life and synthetic benchmarks confirm the effectiveness of the proposed technique.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*

General Terms

Design, Performance, Reliability

Keywords

Mixed-Criticality, Replication, Re-execution, Task Dropping, Worst-Case Response Time

1. INTRODUCTION

Today's multi-processor systems-on-chip (MPSoCs) come with a great performance potential, but also with strong requirements in terms of reliability, due to the increased power density from scaling in manufacturing technology that accelerates the temperature- and current-dependent faults [1]. These architectures enable the parallel execution of several applications that have different requirements in terms of timing and reliability, resulting in so-called *mixed-criticality systems*. An automotive or avionic system is a typical example, where control, signal processing, and multimedia applications co-exist with different safety and reliability requirement levels.

DAC '14, June 01 - 05 2014, San Francisco, CA, USA
 ACM 978-1-4503-2730-5/14/06
http://dx.doi.org/10.1145/2593069.2593221 ...$15.00.

Table 1: Comparison of scheduling/analysis techniques in previous fault-tolerant mapping techniques.

	Mixed-Criticality	Scheduling	Analysis
[2]	none	static	makespan
[3]	FI/FD/FT	static	makespan
[4]	none	dynamic	simulation
[5]	FI/FT	dynamic	probabilistic
[6]	failure probability	dynamic	worst-case

FT: Fault-Tolerance, FD: Fault-Detection, FI: Fault-Ignorance.

This paper addresses the fault-tolerant mapping problem of mixed-criticality MPSoCs, in case of transient faults. We consider conventional hardening techniques such as re-execution and replication, that come with respective overheads in terms of time and resources, additional to the classical task-to-processor mapping optimization trade-offs. Several research efforts have been focusing on this fault-tolerant mapping optimization problem [2]–[6], as summarized in Table 1. However, when considering mixed-criticality applications, the support for multi-tasking and scheduling analysis is fairly limited. Mixed-criticality systems have in place different scheduling algorithms to certify that high-criticality tasks keep providing the intended service. Dropping low-criticality tasks during execution to ensure that high-criticality tasks keep deadlines is a typical approach [7], that we also consider in this paper. To the best of our knowledge, the dynamic behavior of such mixed-criticality systems, with task dropping, has not been considered in the context of worst-case response time (WCRT) guarantees and mapping optimization so far.

Figure 1 is explaining by a motivational example why the mixed-criticality scheduling is more efficient. Let us consider three task graphs with two criticality levels, *high* and *low* (Figure 1(a)). Figure 1(b) illustrates a possible mapping and scheduling solution, in the case where no fault occurs and all three applications are schedulable within the imposed deadline. Let us consider that tasks A and B in the high-criticality graph are hardened by *re-execution* and *replication*, respectively. If a fault occurs at A, an additional execution is triggered and the high-critical task E will violate the deadline in Figure 1(c). In this case, previous approaches would be allocating more resources to accommodate tasks E or G. But if one considers the possibility of discarding

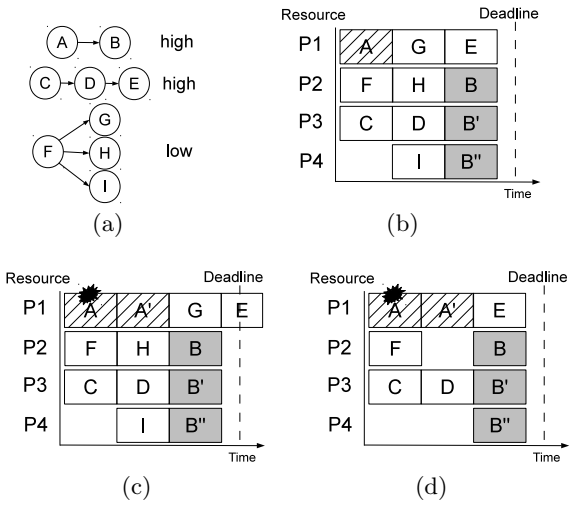

Figure 1: A motivational example: (a) task graphs with mixed criticality, (b) a mapping/scheduling candidate that satisfies the deadline constraint in normal situations, (c) the same configuration fails to meet the deadline constraint due to the re-execution of A, and (d) the constraint is respected by discarding the low-criticality tasks, G, H, and I.

low-criticality tasks during mixed-criticality scheduling, a still valid mapping solution can be found. For instance, in Figure 1(d), tasks G, H, and I with low-criticality, are not triggered, allowing the high-criticality task E to be scheduled on time.[1]

The main challenge in adopting this mixed-criticality scheduling is the analysis and guarantees on the WCRT. Previous approaches (see Table 1) use ordinary scheduling policies and base their analysis on either statically determined makespan [2], [3] or do not guarantee the WCRT [4], [5] relying on simulation or probabilistic analysis. The static scheduling may simplify the optimization complexity but it is inefficient in terms of resource usage [8], and too rigid to be reactive to dynamic system mode changes. At compile time, a static schedule should be synthesized for each possible fault scenario. For instance, in [2], 19 different schedules had to be pre-calculated at compile time for an application with five tasks. Scheduling uncertainties caused by re-execution and passive replication were analyzed in [6] with respect to WCRT, but the effects of task dropping have not been considered.

In this paper we are tackling this challenge and adopt static hardening-mapping/dynamic scheduling policy. That is, once the hardening and mapping decisions of all processing elements (PEs) are done, the tasks mapped on each PE are locally scheduled according to the scheduling policy of that PE. Our proposed WCRT analysis technique, that is designed as a general wrapper of an existing analysis framework such as [9], is safely analyzing the system in the presence of uncertainties caused by hardening and run-time task droppings.

Leveraging on this analysis method, we propose a generic mapping optimization framework for fault-tolerant mixed-criticality MPSoCs, where different hardening techniques can be explored and optimized with WCRT guarantees considering a mixed-criticality scheduling. We judiciously decide the droppable task set during optimization in order to maintain the quality of service as high as possible. The effectiveness of the proposed technique is demonstrated by extensive experiments with real-life and synthetic examples.

[1]For brevity, fault-detection and voting overheads are not illustrated in this example.

2. PRELIMINARY

This section presents the models we use for applications and architecture, as well as considered hardening techniques and mixed criticality concerns.

2.1 System Model

We assume a general MPSoC architecture $\mathcal{A} := (P, nw)$ that consists of a set of (heterogeneous) processors P connected through an on-chip communication fabric nw, such as a shared bus, crossbar switch, or a network-on-chip. In this work, we assume that faults in the communication links are transparent at system level, as typically they are protected by low-level error-resilient techniques [10]. Each processor $p \in P$ is characterized by its $type_p$, leakage power $stat_p$, dynamic power dyn_p, and a constant fault rate per time unit λ_p, as also assumed in [11], [12]. The maximum bandwidth provided by the communication fabric nw is bw_{nw}.

Multiple applications with different levels of criticality are sharing the system, each of them being described as a task graph such as a Kahn process network (KPN). The application set is defined as $\mathcal{T}$, whose elements are the task graphs $t := (V_t, E_t, pr_t, f_t, sv_t) \in \mathcal{T}$. Each task graph t consists of a set of tasks V_t, a set of channels E_t, and an invocation period $pr_t \in \mathbb{Z}$. System designers distinguish between 'droppable' and 'non-droppable' tasks. Non-droppable tasks must always be schedulable even in case of faults, whereas droppable tasks might be dropped by the scheduler. To specify this distinction in the model, non-droppable tasks have a reliability constraint $f_t \in (0, 1]$. The reliability constraint f_t denotes the number of maximum allowable failures per unit time. The lower the reliability constraint f_t is, the higher the criticality level is. An instance of the task graph is released every pr_t time units, and the probability of *unsafe execution* should be smaller than f_t. Each droppable task graph t is associated with the relative importance of the service sv_t, while setting f_t to -1. When a certain set of task graphs is dropped, the quality of service is defined as the sum of sv of alive (non-dropped) task graphs. This value should be set by the system designer as a part of system specification. For non-droppable tasks, sv values are set to ∞, as they are not allowed to be dropped.

Each task $v \in V_t$ of the task graph t is characterized by $(bcet_v, wcet_v, ve_v, dt_v)$, i.e., best-case execution time $bcet$, worst-case execution time $wcet$, voting overhead ve, and detection overhead dt. The voting overhead is related to replication, while detection overhead includes fault detection, storing/restoring the context, and rolling-back for re-execution. These will be explained in detail in the following subsection. A channel $e := (src_e, dst_e) \in E_t$, where $src_e, dst_e \in V_t$, represents a data dependency from task src_e to dst_e, and each transmission causes a data transfer of size s_e.

2.2 Hardening Techniques

Choosing an appropriate hardening technique for a task comes with a trade-off between resource usage and time,

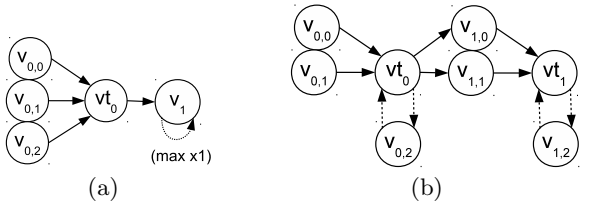

Figure 2: Hardening examples: (a) active replication of v_0 and re-execution of v_1, and (b) passive replication of v_0 and v_1.

and it has been extensively studied [2], [3]. Recently it has been even extended to consider mixed-criticality concerns [6]. This subsection briefly summarizes the hardening techniques we consider, and the related trade-offs. More precise formulation of determining hardening techniques is summarized in [6].

Re-execution. The re-execution scheme assumes that a fault is locally detected at the end of the task execution. Therefore, other than the overhead of re-executing the task, the detection imposes additional overhead. All the stateful variables are rolled-back to the initial state and the same task instance is executed again. In Figure 2(a) v_1 of a simple producer(v_0)-consumer(v_1) application is hardened by re-execution. In this case, the task graph topology remains unchanged, but the task $wcet_v$ is modified to

$$wcet'_v = (wcet_v + dt_v) \times (k+1) \quad (1)$$

with k being the maximum number of re-executions.

Active replication. Active replication uses multiple instances of the hardened task mapped on different processing elements. The quality of replication is usually larger than two, to enable majority voting. For a task that is only duplicated, only detection is possible which is the use case of [5]. Contrary to re-execution, replication will modify the task graph topology. In *active* replication, the replicated tasks are always executed at runtime. Task v_0 is actively triplicated in Figure 2(a).

Passive replication. In passive replication, not all cloned tasks are proactively instantiated, but only on request of the voter. This is particularly beneficial when the system is to be optimized to minimize the average utilization or the average power dissipation. In Figure 2(b), $v_{*,0}$ and $v_{*,1}$ are actively duplicated. When a faulty situation is detected by the voter, a third replica $v_{*,2}$ is instantiated to break the tie (highlighted with dashed arrows in the figure).

2.3 Problem Definition

Given the architecture $\mathcal{A}$ and the application set $\mathcal{T}$, the problem is defined to determine a hardening technique that results in a modified application $\mathcal{T}'$, and for each v such that $v \in V_t$ and $t \in \mathcal{T}'$ to determine a mapping map, where $map : V \rightarrow P$ is mapping tasks to processors, with $V = \bigcup_{t \in \mathcal{T}'} V_t$. In addition, the dropped task set $\mathcal{T}_d (\subset \mathcal{T})$, s.t. $\forall t \in \mathcal{T}_d, sv_t \neq \infty$, needs to be decided. The proposed technique is not specific to a certain objective, thus any formulatable objectives can be minimized/maximized. In this work, we minimize the expected power consumption, considering all possible cases, i.e., $minimize\ \{\sum_{p \in P} (stat_p + dyn_p \cdot u_p)\}$, with u_p the average utilization of processor p. Further, the quality of service after the task dropping can also be considered as a secondary objective: $maximize \sum_{t \in \mathcal{T} \setminus \mathcal{T}_d} sv_t$. Due to space limitation, we do not present the formulations of reliability and architectural mapping constraints, which can be found in [6].

2.4 Mixed-Criticality

From the definition of mixed criticality systems, a task is characterized by different values of $wcet$, in nominal and critical modes [7]. In order to certify the schedulability (or the continuation of service) of higher critical tasks in such systems, many scheduling policies such as [7], [13] have been proposed. In this work, we interpret the variability in $wcet$ as coming from the re-execution scheme, as shown in (1). That is, the nominal execution time ($k = 0$) is the case when there is no fault, while the critical case includes the re-execution and variations in $wcet'$. This is a well-known source of $wcet$ variation, considered in fault-tolerant system designs like [2], [4] and mixed-criticality systems like [5].

3. SCHEDULING ANALYSIS

The hardening techniques introduced above impose uncertainties in the scheduling, thus making it hard to analyze the WCRT. First, in passive replication, the passively replicated tasks are only invoked when the voter detects different values from actively replicated tasks. It cannot be naively assumed that invoking all replicas all the time results in a worst-case response time due to the well-known *scheduling anomaly* [14], which is a situation where a local worst-case execution time does not contribute to the global worst-case. Second, re-execution may alter the execution time of the application too. A task no longer releases a single job in a period, but may release several jobs, as the re-execution scheme allows. Similarly, it cannot be assumed that releasing as many jobs as possible results in a worst-case response time.

These two issues can easily be addressed by modifying the best/worst-case execution time ($bcet$ and $wcet$) of the tasks. If a task is passively replicated, its original $bcet$ is replaced with 0 to model the case when it is not invoked (no fault). Similarly, the re-execution can also be handled with a new value for $wcet$ as shown in Eq. (1).

Another uncertainty originates from task dropping. Whenever a fault occurs, the system enters the *critical* state, where non-droppable tasks should seamlessly continue their execution (i.e., being schedulable), while droppable tasks might be dropped to save space for the high-criticality ones. In other words, 'droppable tasks' become a candidate to be dropped when the system enters the critical state. The decision on which tasks of these candidates will actually be dropped during the critical state is an optimization goal.

We assume that passive replication and re-execution of any task trigger the critical state, while in case of active replication, faults are transparently tolerated and do not introduce any additional timing overhead or system state transition. To be more specific, as soon as a task v exceeds its nominal $wcet_v + dt_v$, the scheduler starts dropping low-criticality tasks. The system goes back to the normal state at the end of the hyperperiod, restoring all the dropped tasks.

Task dropping could also be handled by setting $bcet$ to zero, which would enable it to be statically taken into consideration during analysis. This is a naive approach, however, which would result in very pessimistic results as shown in Section 5.2. As a matter of fact, once the system switches

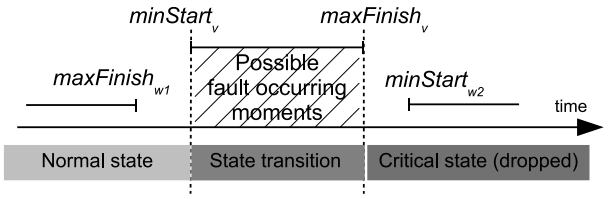

Figure 3: Analysis modes: when v is triggering the system state change, all tasks finished earlier than $minStart_v$ (such as $w1$) are assumed to be operating in the normal state without re-executions. All droppable tasks starting later than $maxFinish_v$ (such as $w2$) will never release their jobs.

to the critical state, the dropped tasks are no longer conditionally invoked but actually detached from the scheduler. In order to consider this fact and reducing the pessimism in the estimates, we conduct an individual scheduling analysis for each possible state transition. The proposed scheduling analysis is presented in Algorithm 1. First, we analyze the system without considering any fault, i.e., neither passive replica or re-executed jobs are released. The preprocessing for this configuration is shown in *lines 2-6*. The normal state analysis is conducted for all tasks in *lines 7-9*, where the minimum starting time ($minStart$) and maximum completion time ($maxFinish$) are calculated for all tasks in the 'normal state'. This information is exploited later on to distinguish between 'criticality states'.

Then, all possible state transitions are investigated in the *for-loop* starting at *line 10*. That is, for the re-executable and passively replicated task v that experiences a fault *for the first time* in the hyperperiod, all other tasks are considered as operating in either normal or critical state, as follows:

- **normal state:** If task w completes earlier than $minStart_v$ ($maxFinish_w < minStart_v$), it is considered to be operating in the 'normal state' (i.e., neither dropping nor re-execution happens for w). Therefore, the $[bcet, wcet]$ bounds are preserved for w (*lines 14-17*). This case is illustrated in task $w1$ in Figure 3, where $w1$ already completes its execution before the first fault occurring moment ($minStart_v$).
- **critical state:** Otherwise, task w possibly goes into the critical state. If w cannot be dropped, its $[bcet, wcet]$ bounds are adjusted according to (1) (*line 26*). Then, for tasks that can be dropped, we distinguish between two cases:
  - If task w starts later than $maxFinish_v$ ($minStart_w > maxFinish_v$), this task is certainly dropped and never executed as the transition to the critical state has already been completed. Thus, it does not appear in the schedule, and it is modeled with $[0, 0]$ bound (*lines 20-21*). Task $w2$ in Figure 3 exemplifies this case. It is noteworthy that it is assumed that the task dropping can be accomplished quick enough before the triggering task completes its original execution. Thus, from the moment of $maxFinish_v$ on, all droppable tasks completely disappear in the system.
  - Otherwise, both possibilities are open, i.e., it can either be dropped (as implied by a *bcet* of zero) or normally executed (*line 23*) as implied in *transition mode* in Figure 3.

With the modified $[bcet, wcet]$ bounds, we perform the schedulability test again at *line 30*. If the result exceeds the current maximum response time, the return value is updated (*lines 31-32*). Then, the loop continues for all tasks $v \in V$ that may trigger system state transitions.

Algorithm 1 Given the architecture $\mathcal{A}$, applications $\mathcal{T}$, hardening decisions, mapping, and dropped application set $\mathcal{T}_d$, return the maximum completion time of $v_{in} \in V = \bigcup_{t \in \mathcal{T}'} V_t$

```
1: ret ← 0                                              ▷ Initialize return value
2: for all v ∈ V do                                     ▷ Pre-processing for normal state
3:     if v is passively replicated then
4:         [bcet_v, wcet_v] ← [0, 0]
5:     end if
6: end for
7: for all v ∈ V do                                     ▷ normal state analysis
8:     [minStart_v, maxFinish_v] = sched(v, map)
9: end for
10: for all v ∈ V s.t. re-executable or passively replicated do
11:                                  ▷ For each task v that may trigger state changes
12:     for all w ∈ V such that w ≠ v do                ▷ For all other tasks
13:         if maxFinish_w ≤ minStart_v then            ▷ normal state
14:             [bcet'_w, wcet'_w] ← [bcet, wcet]
15:             if w is passively replicated then
16:                 [bcet'_w, wcet'_w] ← [0, 0]
17:             end if
18:         else                                        ▷ critical state
19:             if w ∈ V_t s.t. t ∈ T_d then            ▷ For droppable tasks
20:                 if minStart_w > maxFinish_v then    ▷ Dropped
21:                     [bcet'_w, wcet'_w] ← [0, 0]
22:                 else                                ▷ Either executed or dropped
23:                     [bcet'_w, wcet'_w] ← [0, wcet_w]
24:                 end if
25:             else                                    ▷ Non-droppable tasks
26:                 [bcet'_w, wcet'_w] ← [bcet_w, Eq.(1)]
27:             end if
28:         end if
29:     end for
30:     temp = sched(v_in, map) with [bcet', wcet']s    ▷ Re-analyze
31:     if temp > ret then                              ▷ and update if necessary
32:         ret ← temp
33:     end if
34: end for
35: return ret
```

It is worth mentioning that *sched* function in Algorithm 1 is not specific to a certain analysis method, thus any other schedulability analysis can be alternatively used as a back-end as long as it can derive the worst-case/best-case completion/starting time of tasks. While our implementation uses [9], other WCRT estimate techniques such as [15]–[17] can replace it once assisted with a proper best-case starting time estimation. The time complexity of the proposed analysis is $O(|V|^2 + |V|C)$, where C is the time complexity of the *sched* function. For instance, the time complexity of [9] is $O(|V|^3)$, so the overall time complexity of the proposed algorithm is $O(|V|^4)$.

4. DESIGN SPACE EXPLORATION

Due to the well-known complexity of the mapping optimization problem, we use a genetic algorithm (GA) to explore the solution space. The structure of a chromosome in the proposed GA is illustrated in Figure 4, consisting of three sections that represent the allocation of processors, selection of (non-)droppable applications during the critical mode, and binding/hardening information. For the al-

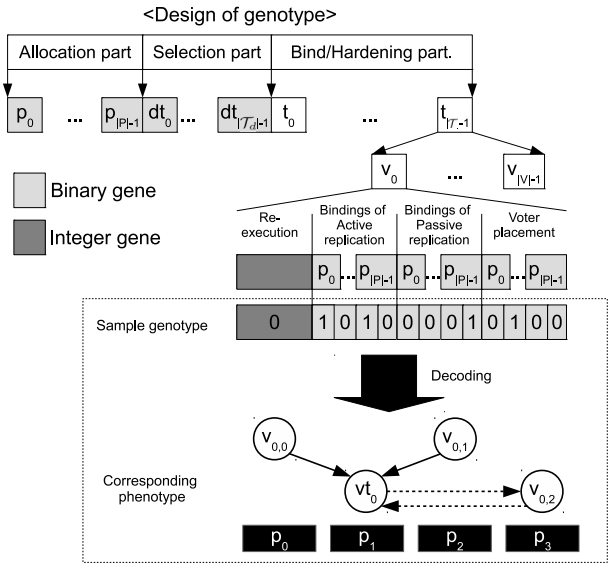

Figure 4: Design of the genotype, and translation of a genotype into a phenotype.

location information, a binary variable is allocated for each processor in the target architecture. Processors that are allocated set on this variable to 1 (otherwise off). For the selection of non-droppable applications, a binary variable set to 1 indicates that the application will not be dropped regardless of mode changes. Otherwise, the application will be dropped when entering the critical mode. The mapping and hardening configurations are determined simultaneously for each task. So, for each task, the degree of re-execution, mappings for active and passive replicas, and mapping of the voter are configured as described in Figure 4.

Before evaluating the fitness of a candidate solution, its feasibility is checked. Infeasibility may come from an abnormal mapping or hardening decision. In such a case, we repair the candidate according to a randomized heuristic that is designed depending on the violation. For example, the reliability constraint is violated, random hardening techniques among active/passive replication and re-execution are applied until the solution meets the constraint. If the candidate map tasks on unallocated processors (a so-called invalid mapping), those illegally mapped tasks will be reassigned to a randomly chosen processor among valid ones. In case of other violated constraints, we penalize the solution with an exceedingly bad fitness value in order to guide the algorithm towards feasible solutions. Once their feasibility has been decided, candidates are quantitatively evaluated as described in Section 2.3. Evaluations of solutions are independent, so they are implemented in a parallel way, using multiple threads to boost the optimization speed.

During GA, new offsprings are iteratively generated by crossovers and mutations of the current population of candidate mappings. The population is maintained constant by discarding individuals with lower fitness values. We use an open-source framework OPT4J [18] as GA engine and SPEA-II [19] as population selector. The population size, number of parent individuals, and number of offsprings are all set to 100 in the experiments. The optimization procedure is set to terminate after $5,000$ generations.

5. EXPERIMENTS

Table 2: WCRT [ms] of two critical applications in the *Cruise* example, for three sample mappings.

	Mapping 1		Mapping 2		Mapping 3	
Adhoc	661	462	819	723	771	525
WC-Sim	661	521	649	568	678	480
Proposed	**666**	**552**	**842**	**815**	**810**	**563**
Naive	796	641	1035	981	1007	915

We show the effectiveness of the proposed technique with two synthetic examples that are randomly generated and three real-life applications. We use a cruise control application *Cruise* from [20], to which we add three synthetic applications, to increase the benchmark complexity. We also use two control benchmarks, "medium/large distributed non-preemptive real-time CORBA application" (*DT-med/large*) inspired from [21], and to which we add complexity and uncertainty by multiplying the invocation period and execution time of the original tasks by 20 times. A more detailed description of the benchmarks can be found in [6].

5.1 Scheduling Analysis

To illustrate the safety of WCRT estimates calculated with Algorithm 1, three different mappings of the *Cruise* benchmark are analyzed and compared in Table 2. First, we built up an artificial scheduling trace that estimates the worst-case in an 'ad-hoc' manner. That is, we assumed that the system enters the critical state at the beginning of the hyperperiod, all re-executable tasks being (maximally) re-executed with $wcet'$ from (1) and all droppable tasks being dropped from the beginning. The WCRT observed in these traces is recorded as *Adhoc* in Table 2. Then, for each sample mapping, we conducted Monte-Carlo simulations that repeat on 10,000 different failure profiles, and recorded the maximum WCRTs in *WC-Sim*. The last comparison is made against *Naive*, where the WCRT is naively estimated by just assuming zero *bcet* to have (0, wcet) range of execution time for all droppable tasks, as explained in Section 3.

The proposed analysis always upper-bounds the simulation and ad-hoc worst-case results. Note that the *Adhoc* approach gives worse results than the simulation in some cases, which confirms that simulation coverage is not enough for WCRT analysis. Even though *Naive* also safely bounds the WCRT, it is too pessimistic compared to the proposed analysis. The pessimism comes from not considering the chronological information of changes in the system status. That is, even though re-execution and task dropping never happen until the fault occurring moment, this hint is not considered in *Naive*, but only in the proposed technique. The WCRT estimates of the proposed technique are not tight, but only represent a safe upper bound. However, this is related to the implementation of the underlying schedulability analysis method, and it is beyond the scope of this paper.

5.2 Effect of Task Dropping

In order to illustrate how mixed-criticality scheduling with task dropping improves the optimality of the system, we have compared the optimized power consumptions with and without task droppings in three benchmarks *DT-med*, *DT-large*, and *Cruise*. They spend 14.66%, 16.16%, and 18.52% more power without task droppings, respectively.

Further, we keep track of all solutions during design space exploration and calculate the ratio of solutions that are in-

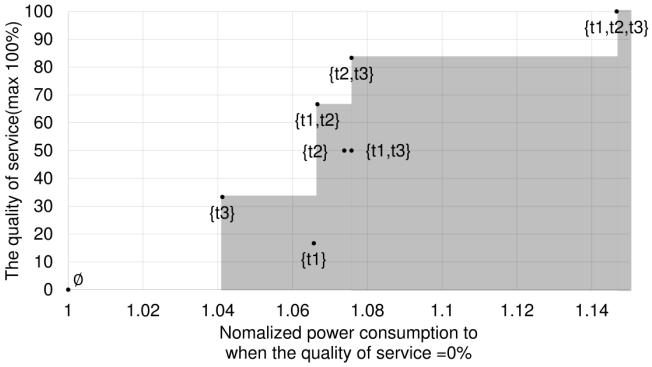

Figure 5: Co-optimization of service and power consumption for *DT-med* benchmark.

feasible without task dropping, but that become feasible if dropping is enabled. The ratios are 0.02% in *Synth-1*, 0.685% in *Synth-2*, 29.00% in *DT-med*, 22.49% in *DT-large*, and 99.98% in *Cruise*. Generally, it is observed that task dropping is particularly helpful when the deadline is close to the scheduling make-span. The more tasks hardened by re-execution, the bigger the ratio is likely to be. For example, 87.03%, 98.66%, and 83.23% of applied hardening techniques are re-executions for *DT-med*, *DT-large*, and *Cruise*, while only 44.29% is for *Synth-1*. During experiments, it is also observed that this ratio increases as the design space exploration converges to optimum.

5.3 Service-Optimality Trade-Off

The gain of task droppings, explained above, comes at the cost of service degradation. We judiciously decide the dropped task set $\mathcal{T}_d$, to maximize the quality of service after task dropping. As explained in Section 2.3, the mapping is optimized with respect to two objectives, one being the minimization of the average power consumption, i.e., $minimize\ \{\sum_{p\in P}(stat_p + dyn_p \cdot u_p)\}$, while the second objective is the maximization of the quality of service after task dropping, i.e., $maximize\ \sum_{t\in\mathcal{T}\setminus\mathcal{T}_d} sv_t$.

The Pareto-front of power-service pairs as output of the optimization is shown in Figure 5 for *DT-med*. When all task are dropped, ϕ, the power optimality is the best as expected. In contrast, the case of no task dropping, $\{t1, t2, t3\}$, shows the maximum quality of service. In total, five Pareto-optimal design points have been obtained, as an outcome of the exploration of the trade-off between service and power efficiency. The proposed technique enables the automatic and systematic exploration of this complex decision problem, and without proper quantification of the effect of the task dropping, it is not trivial to make the decision.

6. CONCLUSION

In this paper, we propose a static mapping optimization technique with worst-case guarantees for mixed-critical applications executing on fault tolerant MPSoCs. In addition to the classical hardening techniques by re-execution and replication, a mixed-criticality scheduling with task dropping is proposed that certifies that high-criticality applications provide their service, and their worst-case response times are guaranteed. Experimental results prove that the proposed analysis technique is able to safely bound the worst-case response times during mixed-criticality scheduling, and can be used to explore the service-optimality trade-off during mapping optimization.

7. ACKNOWLEDGEMENTS

This work was supported by Bio-Mimetic Robot Research Center funded by Defense Acquisition Program Administration (UD130070ID), by the MSIP, Korea, under the ITRC support program supervised by the NIPA (NIPA-2013-H0301-13-1011), by EU FP7 projects EURETILE and CERTAINTY under grant numbers 247846 and 288175, and by Basic Science Research Program through the NRF funded by the Ministry of Education, Science and Technology (NRF-2013R1A1A1012715).

References

[1] Y. Xiang *et al.*, "System-level reliability modeling for mpsocs," in *CODES+ISSS*, Scottsdale, AZ, USA, 2010, pp. 297 –306.

[2] P. Pop *et al.*, "Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication," *Very Large Scale Integration Systems, IEEE Transactions on*, vol. 17, no. 3, pp. 389 –402, 2009.

[3] C. Bolchini *et al.*, "Reliability-driven system-level synthesis for mixed-critical embedded systems," *Computers, IEEE Transactions on*, vol. 62, no. 12, pp. 2489–2502, 2013.

[4] P. v. Stralen *et al.*, "A safe approach towards early design space exploration of fault-tolerant multimedia mpsocs," in *CODES+ISSS*, Tampere, Finland: ACM, 2012, pp. 393–402.

[5] P. Axer *et al.*, "Reliability analysis for mpsocs with mixed-critical, hard real-time constraints," in *CODES+ISSS*, Taipei, Taiwan, 2011, pp. 149 –158.

[6] S.-H. Kang *et al.*, "Reliability-aware mapping optimization of multi-core systems with mixed-criticality," in *DATE*, Dresden, Germany, 2014.

[7] S. Baruah *et al.*, "Towards the design of certifiable mixed-criticality systems," in *RTAS*, IEEE, Stockholm, Sweden, 2010, pp. 13–22.

[8] V. Izosimov *et al.*, "Synthesis of fault-tolerant schedules with transparency/performance trade-offs for distributed embedded systems," in *DATE*, Munich, Germany, 2006, pp. 706–711.

[9] J. Kim *et al.*, "A novel analytical method for worst case response time estimation of distributed embedded systems," in *DAC*, ACM, Austin, TX, USA, 2013, p. 129.

[10] H. Kopetz *et al.*, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[11] A. Sanyal *et al.*, "An improved soft-error rate measurement technique," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 4, pp. 596–600, 2009.

[12] P. Shivakumar *et al.*, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *DSN*, Bethesda, MD, USA, 2002, pp. 389–398.

[13] D. de Niz *et al.*, "On the scheduling of mixed-criticality real-time task sets," in *RTSS*, Washington, DC, USA, 2009, pp. 291–300.

[14] R. Wilhelm *et al.*, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 7, pp. 966–978, 2009.

[15] E. Wandeler *et al.*, "System Architecture Evaluation Using Modular Performance Analysis - A Case Study," in *Proc. ISoLA*, Paphos, Cyprus, 2004.

[16] R. Henia *et al.*, "System level performance analysis–the symta/s approach," *IEE Proceedings-Computers and Digital Techniques*, vol. 152, no. 2, pp. 148–166, 2005.

[17] A. Brekling *et al.*, "Models and formal verification of multiprocessor system-on-chips," *The Journal of Logic and Algebraic Programming*, vol. 77, no. 1, pp. 1–19, 2008.

[18] M. Lukasiewycz *et al.*, "Opt4J - A Modular Framework for Meta-heuristic Optimization," in *GECCO*, Dublin, Ireland, 2011, pp. 1723–1730.

[19] E. Zitzler *et al.*, "Spea2: improving the strength pareto evolutionary algorithm," Tech. Rep., 2001.

[20] N. Kandasamy *et al.*, "Dependable communication synthesis for distributed embedded systems," in *Computer Safety, Reliability, and Security*, Springer, 2003, pp. 275–288.

[21] G. Madl *et al.*, "Tutorial for the open-source dream tool," *Univ. California, Irvine, CA, CECS Tech. Rep*, 2006.