Low-cost Checkpointing in Automotive Safety-Relevant Systems

Carles Hernandez, Jaume Abella
Barcelona Supercomputing Center (BSC-CNS)

***Abstract*—The use of checkpointing and roll-back recovery (CRR) schemes is common practice to increase the likelihood of a task completing with the correct result despite the presence of faults. However, the use of CRR mechanisms is challenging in the severely constrained design space of safety-relevant embedded systems, such as those controlling critical functions in the automotive domain. CRR schemes introduce non-negligible time and memory overheads that may jeopardize the feasibility of their implementation. In this paper we propose a low-cost checkpointing mechanism suitable for safety-relevant embedded systems deploying light-lockstep architectures. The proposed checkpointing mechanism increases the reliability of the system while keeping timing and memory overhead low enough.**

I. Introduction

The need for increased functionality has forced designers of safety-relevant embedded systems to include more computation and memory resources within a single chip. The integration of a higher number of transistors in the same chip cannot be achieved without using smaller technology nodes, that suffer increased process variability. Small imperfections in transistors and wires introduced in the manufacturing process become more significant as they are a significant fraction of the feature size. Additionally, as the number of transistors per unit area increases, lower voltages are required to keep power density below a certain threshold, thus making signals within the chip to be more sensitive to electromagnetic disturbances [3].

Hardware faults may make programs in charge of some safety-related functions lead to unexpected behaviour. Whenever this occurs, hardware/software means must take care of detecting those errors and recovering the system to a safe state. Redundant execution is a common mean for error detection in safety-relevant applications usually running on top of light-lockstep architectures [9], [13], [14]. Additionally, error handling methods like recovery through repetition are state-of-practice mechanisms and considered in certification standards (e.g., in ISO26262 [15]). These reasons make checkpointing and rollback recovery (CRR) schemes convenient for safety-relevant systems. CRR schemes are widely used in the high-performance arena as a way to improve reliability at low cost. Traditional CRR schemes store the processor's state and data memory into a safe storage to build a recovery point. A safe recovery point is ensured as checkpointed data is compared by means of redundant execution. However, regular checkpointing mechanisms as those used in the high-performance community cannot be directly used in the context of safety-relevant systems.

CRR schemes increase the likelihood of a task completing on time with the correct result despite the presence of faults. As a drawback, checkpointing increases task's execution time and the time overhead due to checkpointing must be considered in the task scheduling feasability analysis. Furthermore, checkpointing mechanisms designed for embedded systems require low and bounded memory overhead. The straightforward approach to reduce memory overhead is to let the user control the points where to inject checkpoints selecting those points where data requirements are minimum. For example, in the automotive domain applications are developed according to the principles defined in the AUTOSAR standard [2]. AUTOSAR divides applications into atomic units called *runnables*, which are the smallest unit of execution. Runnables communicate through shared memory and message passing, and have clear input/output interfaces. Those runnables are grouped into tasks and/or software components. In the case of software components, they communicate only through message passing. Thus, runnable, task and software component boundaries are suitable points to checkpoint execution due to the clear interface among them, which limits the amount of data that needs to be checkpointed. On the other hand, the size of application components (runnables, tasks and software components) can be arbitrarily large, thus not allowing to find acceptable bounds for the time elapsed between checkpoints, and so not having bounds for the time-to-error-detection. In the extreme case we can find traditional retry-based recovery schemes that do not save any checkpoint and thus, need restarting the complete application on an error detection.

In this paper we (1) review appropriate CRR mechanisms from the perspective of safety-relevant systems, (2) characterize their impact in the timing of tasks, and (3) propose a cost-effective error recovery mechanism suitable for systems deploying light-lockstep architectures. The proposed recovery mechanism combines coarse-grain full checkpointing (memory and registers) with fine-grain light checkpointing (registers only) for fast error detection and recovery with limited timing and memory overheads. Results show that coarse-grain full checkpointing scheduling guarantees are only slightly worse than an ideal checkpointing scheme under very severe failure rates while achieving outstanding reliability values for the failure rates targeted by ISO26262 [15]. Additionally, low detection latency bounds are also attained at very low cost with the use of light fine-grain checkpointing.

II. Cost-Effective Coarse-Grain Error Recovery in Automotive Safety-Relevant Systems

A large number of mechanisms for error recovery exists in the literature. Next we introduce CRR mechanisms for error recovery in the context of automotive systems, and how they relate to error detection by means of lockstep execution, which is the main error detection mechanism for critical automotive systems [9], [13], [14].

A. Checkpointing and Rollback Recovery

The most trivial recovery mechanism is the one that, on the occurrence of an error, sets the system to a fail-safe state, resets the system to reach an error-free processor state and retries the execution of the application from the beginning. However, this trivial approach that perfectly fits with ISO26262 recovery mechanism requirements [15] has several limitations. On one hand, error detection may occur long after error occurrence, thus leading to long time wasted performing useless computation.

978-3-9815370-4-8/DATE15/©2015 EDAA

While this is not a challenge for functional correctness, it is detrimental for timing correctness since processor usage increases, leading to missing the scheduled deadlines so that the transfer to a fail-safe state occurs too late or the unavailability of the system lasts too long. On the other hand, to cope with the demand for increased functionality, it is desirable to allow multiple safety-relevant applications execute in the same system (e.g., on a multicore processor). If multiple such applications with mixed criticalities run concurrently, resetting the system may not be acceptable. For instance, an ASIL B[1] or ASIL C application must not be allowed to trigger a system reset if an ASIL D application is also being run, as this would violate the required isolation across different integrity levels making an ASIL D application depend on lower integrity ones.

Therefore, fine-grain mechanisms are needed to recover from errors selectively for each application, and to perform such recovery short after the error occurs. CRR has been shown to be a very popular solution in the high-performance domain [22], [27]. CRR requires saving snapshots of the state of the application periodically and, on an error, rollback the state to the latest error-free snapshot. To perform a checkpoint at a given point in the execution of a program the state of the processor has to be saved in a memory component with enough storage capacity. The straightforward approach to perform a checkpoint is to suspend the execution of the application while the contents of processor's memory and registers (so its architectural state) are written in memory. Recovery is carried out by reloading the original application binary file and restoring the processor state that was checkpointed.

The size of a checkpoint depends on the approach used. There are two main different checkpointing approaches: full and incremental checkpointing. Full checkpointing requires saving all application's architectural state (its memory space and registers contents), so it is very costly in the general case as the data used at a given point in time can be huge. On the contrary, incremental checkpointing, like the *undo log based checkpointing technique* [7], log all the memory writes performed by the program after the last checkpoint. Memory transactions are intercepted by code instrumentation strategies that allow to populate the log buffer. To avoid an unacceptable overhead of searching for duplicate entries every write transaction generates a new entry in the log. This makes undo log checkpointing techniques to offer very poor memory guarantees as the size of the log cannot be upperbounded.

Although more efficient techniques have been devised to identify the faulty module so that rollback is not needed, they have been proven only on some particular blocks smaller than full cores [26] and so, they cannot be straightforwardly adopted in the context of safety-related lockstep-based systems.

B. Reducing Memory Overhead

The best way to limit the memory overhead required to save applications's data is to control the exact point where checkpoints are performed so that they occur when the data volume to be checkpointed is known and low. We propose to do this by taking into account the mechanisms used for communication across components in the context of automotive systems. For example, in the context of AUTOSAR, applications are divided into atomic units called *runnables*. Runnables communicate among them through shared memory and message passing. Runnables

[1]ISO26262 standard defines four different Automotive Safety Integrity Levels (ASIL), from A to D, being D the highest one, and so the one where the strictest validation and verification means are required.

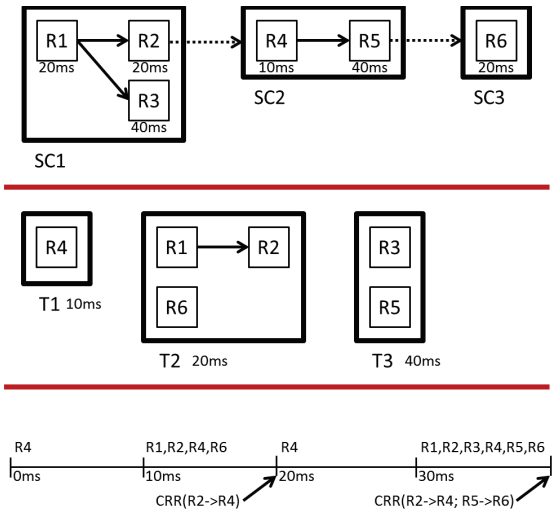

Fig. 1. Example of an AUTOSAR application divided into software components, tasks and runnables. A potential schedule is shown at the bottom.

can be grouped in two different ways: (1) runnables running with the same period (e.g., every 10ms) can be grouped in *tasks*. Also, runnables performing a particular functionality can be grouped into *software components*. Therefore, each runnable belongs to a task and to a software component simultaneously. Tasks, like runnables, communicate among them through shared memory and message passing. Software components, instead, only communicate among them through message passing. Thus, the user can identify the data shared across runnables, tasks and software components and, based on their respective execution periods determine when checkpoints need to occur so that the amount of data checkpointed is affordable.

We illustrate this with the example in Figure 1. In this figure we see 6 runnables (from R1 to R6), organized into 3 software components (SC1, SC2 and SC3) and 3 tasks (T1, T2 and T3). Next to each runnable it is indicated its execution frequency. Straight lines indicate communication through shared memory whereas dashed lines indicate communication through message passing. We also show a potential schedule of the runnables in time. In this example, the user could set up a checkpoint every 20ms right after R6, thus saving R2 output data sent to R4 and, once every two checkpoints, R5 output data sent to R6. An analogous analysis can be done for the avionics domain for the functions in a software partition as defined in IMA [1].

C. CRR Schemes in the Context of Light-Lockstep Architectures

The main principle behind lockstep execution is the replication of computation in different pieces of hardware comparing their outputs for error detection. Although comparison can occur at different granularities (pipeline stage, instruction, off-core activity, system level, etc.), light-lockstep systems where error detection occurs at off-core level has been shown to be a very convenient solution [11].

In light-lockstep architectures, error detection is performed by comparing the transactions that are propagated to the on-chip bus to find any possible mismatch. If a CRR mechanism is used, whenever an error is detected by means of the parallel redundant execution, the system is rollback to a previous safe checkpoint. The lockstep architecture ensures stored data to be safe as all data is exposed to the on-chip bus when stored in memory. In a lockstep system, whenever a checkpoint is performed the possible errors latent in the processor that may be

stored in the architectural registers or operative system kernel registers, are automatically exposed and therefore, detected by the lockstep architecture. Hence, CRR schemes running in a lockstep architecture have an upperbounded error detection latency determined by the checkpoint interval, as this is the maximum latency needed to detect latent errors in the processor. In this context, having frequent checkpoints is desirable to maintain error detection latency low. Note that having a low error detection latency is crucial to increase the reliability of the system. On one hand, assuming a given fault probability, a deadline will be satisfactorily reached with a given probability as well. On the other hand, when errors are caused by permanent faults the system needs to transition to a fail-safe state. To allow the system distinguish between transient and permanent faults, fault diagnosis is required. Fault diagnosis is performed according to error persistence [8].

III. Error Detection and Recovery Latency Analysis

In this section we show that a relevant number of errors remain dormant in the processor for a long time. This makes error detection latency to be determined by the checkpointing frequency. Given that low-cost checkpoints are only possible at certain points in the execution of an application, error detection latency bounds may be too high to guarantee the timing correctness of the system. Therefore, if low-cost light lockstep is delivered for error detection, further means must be provided to timely detect errors.

A. Fault Injection Experiments

We have characterized the timing behaviour of errors in the processor by injecting both single bit upsets (SBU) and permanent faults[2]. Transient and permanent faults can be injected in any processor component. However, in our experiments we have injected faults only in the register file as faults in other components are usually quickly exposed to the shared bus and thus, effectively detected by the lockstep core, or reach a register in few cycles. For instance, the instruction cache (IL1) is only written with data received through the bus, so the core can only corrupt the IL1 requesting wrong addresses, which would be detected by the lockstep core. Similarly, the data cache can be written with new data fetched from the bus, where errors would be detected analogously to those of the IL1, or by store operations. In our particular architecture we consider write-through caches, as in many processors used in safety-relevant applications, so write operations are immediately propagated to the shared bus, thus allowing again the lockstep to immediately detect errors.

We use the EEMBC Autobench benchmark suite [24], which is a well-known suite reflecting the current real-world demand of some automotive embedded systems. Benchmarks are executed on a SystemC processor model resembling the AURIX processor [14]. The simulation tool used is an enhanced version of the Soclib simulator which is a widely known open-source tool for virtual prototyping [19].

Given N_{reg} registers, each benchmark is executed $N_{reg} \cdot 100$ times, so 100 times per register. On an execution the value of the particular register under consideration is poisoned with a fault at a cycle chosen randomly across the number of cycles of a fault-free execution. Permanent faults remain in the register after they show up in a given cycle while transient faults disappear after the erroneus bit register is overwritten. To compute error detection latency we measure the number of cycles elapsed since the fault is injected until it is detected in the shared bus. Faults undetected by the end of the execution are ignored since the program finishes execution and they have not reached any observable device (memory or I/O). Analogously, faults that disappear without propagating [12], [18] become also irrelevant (e.g., because the register holding the fault is overwritten before the fault propagates).

[2]Although multiple-bit upsets (MBU) have been regarded as frequent enough not to be dismissed [21], they would also be detected by lockstep processors analogously to SBU.

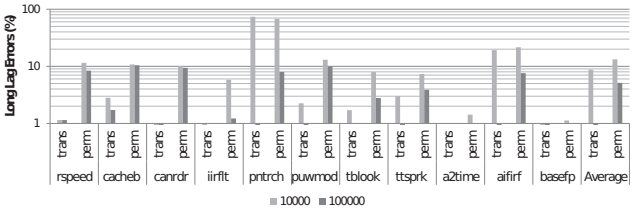

Fig. 2. Percentage of Long Lag Errors due to permanent and transient faults.

Figure 2 shows the fraction of permanent and transient errors with detection latencies above 10,000 and 100,000 cycles. A significant fraction of the total faults for the majority of applications has very long detection latencies. On average, around 5% and 1% of the faults have detection latencies above 100,000 cycles for permanent and transient faults, respectively. This confirms that the presence of long-lag errors cannot be neglected.

B. Lightly Verbose Lockstep Operation

Light lockstep architectures can also provide error detection timing guarantees if they operate in **LI**ght **VE**rbose (LiVe) mode as proposed in [11]. *LiVe* operation relies on sending register file contents through the network periodically so that error detection features of lockstep can detect any discrepancy across values in the different cores.

LiVe defines a *Maximum Detection Interval* (or MDI for short). Such MDI is the maximum time it can elapse since a register holds wrong data until such value is sent to the network. Enforcing the MDI not to be exceeded requires each register to be sent to the network every MDI cycles at most.

Register communications are performed by means of non-blocking write operations to a non-cacheable destination address not mapped into any device. By doing so, the register value is exposed into the network and when the core executing the trailing thread sends this register, the checker component attached to the on-chip bus snoops the value for error detection. Note that *LiVe* has no functional impact due to the extra write operations.

Overall, the maximum latency elapsed since an error occurs until it is detected is determined by (i) how long an error takes to reach a register (typically very few cycles), how much time is elapsed since a register holds a wrong value until it is sent through the network (MDI cycles at most), and how long it takes since a value is sent through the network until it reaches the trailing core and the error is eventually notified (again, typically few cycles). Thus, maximum error detection latency mostly depends on MDI [11].

IV. Cost-Effective Fine-Grain Checkpointing in Light-Lockstep Architectures

The low-cost checkpointing (**LC**) mechanism proposed in this paper is based on the fact that increasing checkpoints frequency

is not a valid mean to improve error detection latency. On the contrary, in this paper we consider that checkpoints are generally only doable at certain points during the execution of a task in order to avoid the need for storing the whole shared memory space (or at least a non-negligible fraction of it). The exact point when checkpoints are cheap depends on the concrete application. For instance applications with reduced intertask dependences may allow checkpointing at several points. However, as a rule of thumb, checkpointing costs are minimized at entity (task, runnable, software component) boundaries. At those points only the messages or shared variables containing the data to be exchanged amongst entities have to be stored in memory. In any case, checkpointing frequency cannot be unlimitedly increased as checkpointing timing overheads also increase with checkpointing frequency. For the sake of commodity, we abuse the term *task* and use it to refer to entities in the rest of the paper.

The proposed **LC** scheme combines the use of both coarse-grain *Full* and fine-grain *Light* checkpoints. *Light* checkpoints (only registers) are performed using the light lockstep system operating in *LiVe* mode as explained in Section III-B. This allows us to reduce error detection latency bounds of those errors that remain dormant in the cores for a long period. By triggering *LiVe* within a *Full* checkpointing interval (I_F) the maximum detection interval is set to the *Light* checkpointing interval (I_L) without incurring in any memory overhead. Note that *LiVe* does not store any data in memory. Triggering *LiVe* several times within I_F incurs in the overhead of periodically sending the contents of the registers to a given memory address. However, the time overhead of *Light* checkpointing is much smaller than the overhead of performing a task *Full* checkpoint, and upperbounded [11].

While *Light* checkpointing provides nearly-immediate error detection, *Full* checkpointing is still needed given that, on an error detection, the snapshot of the memory and registers captured by *Full* checkpointing needs to be restored.

Figure 3 shows timing diagrams of a regular checkpointing mechanism and the proposed **LC** scheme. In a regular checkpointing mechanism (Fig 3 left) the maximum error detection latency (Δ_e) is set by the checkpointing interval (I_F). On the contrary, with our **LC** proposal Δ_e is reduced by inserting *Light* checkpoints between two consecutive *Full* checkpoints. The number of *Light* checkpoints in a given I_F reduces average and maximum detection latency.

Regular CRR mechanisms have an overhead that depends on the cost of saving the checkpoint (C_F), the time required for recovery (R), and the number of faults in a given time window. In our case we also have to include in the equation the cost of performing *LiVe* checkpoints (C_L) in a period T[3]. Assuming that faults follow a *Poisson* distribution with a fault rate λ, the average number of faults in a period T is equal to λT. The total overhead of our **LC** proposal in a period T is as follows:

$$O(T) = C_F + \lambda T(R + T) + C_L \tag{1}$$

The overhead is, therefore, the cost of a *Full* checkpoint (C_F), of all *Light* checkpoints (C_L), and the time required to recover and reexecute ($R + T$) multiplied by the expected number of errors in the period (λT).

In a regular CRR scheme, the total cost of recovery in case of a dormant error is equal to $R+T$. When combining *Full* and *Light* checkpoints the cost of recovery is reduced as, on average, errors are detected in $T/2$. The overhead per unit time is a convex function that can be computed dividing the overhead, shown in Equation 1, by T. The minimum of this function represents the checkpointing interval [5] that minimizes the overhead of the checkpointing mechanism for a given λ. In a regular CRR mechanism the optimal checkpointing interval is $I_F = \sqrt{C/\lambda}$ while when using *Full* + *Light*, assuming the overhead of *Light* checkpoints is negligible, the optimal checkpointing is given by $I_F = \sqrt{2C/\lambda}$. In other words, when our checkpointing mechanism is deployed a given target reliability can be achieved with a reduced checkpointing frequency.

However, in a realistic scenario memory overhead costs cannot be neglected. Under this premise the checkpointing frequency is determined by the timing characteristics of the application. We define the following procedure to determine the intervals for *Full* and *Light* checkpoints.

- Given an application we identify the point at which the cost of *Full* checkpointing is low. Once these points are identified the interval for *Full* checkpointing I_F is known.
- With the given I_F we compute the actual P_{τ_i} and the error detection latency bounds. If these values are acceptable the process finishes. If P_{τ_i} is below the target probability or the detection latency is above the acceptability threshold we insert *Light* checkpoints until the target probability is reached and/or the detection latency is below the acceptability threshold . The resulting I_L is the interval we use for *Light* checkpoints.

V. Light-Checkpointing Timing Analysis

In this section we characterize the timing behaviour of entities in the presence of faults with the proposed **LC** mechanism to derive expressions to compute P_{τ_i}. In particular we show the expressions for one task running in lockstep and several tasks in the lockstep. Expressions in this section are given for faults following a *Poisson* distribution. Analogous expressions can be derived for other fault models like those in [17].

A. One task in Lockstep

With the proposed checkpointing scheme the worst-case overhead recovery of a task running in lockstep in isolation is determined by the cost of placing *Full* checkpoints (C_F), the costs of setting *Light* checkpoints (C_L), the time to log shared data (C_S), and the time to recover the system (R). Let us define W_i to be the worst-case execution time of a task τ_i. In the presence of k fault ocurrences the worst-case execution time can computed as:

$$Wk_i = W_i + C_F + C_L + k \cdot R \tag{2}$$

Given a deadline (D) for task completion we have to guarantee that $Wk_i \leq D$. Provided that the number of *Full* checkpoints is computed according to the procedure explained in Section IV, we can solve Equation 2 to compute the maximum number of faults (k_{max}) that can be tolerated within the provided deadline [7]. Finally, the probability of a task completing on time is equal to the probability (P) that no more than k_{max} faults occur within a time window D. As faults are assumed to follow a *Poisson* distribution $P(k < (k_{max} + 1))$ is given by:

$$P_{\tau_i} = \sum_{0}^{k_{max}} \frac{e^{-\lambda D}(\lambda D)^n}{n!} \tag{3}$$

[3]Note that C_L stands for the cost of *all LiVe* checkpoints between consecutive *Full* checkpoints.

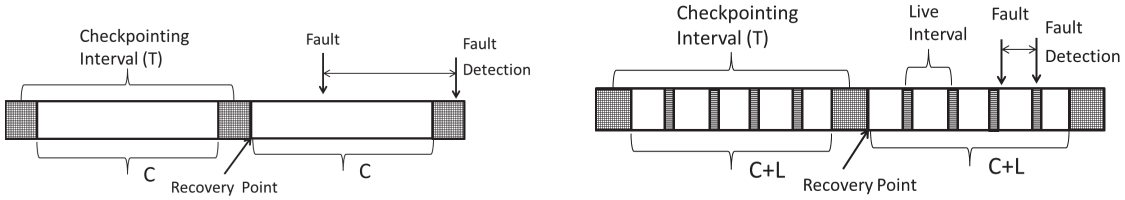

Fig. 3. Comparing traditional CRR schemes and the proposed **LC**.

B. Multiple tasks in Lockstep

For the case of several tasks running concurrently in lockstep we consider a standard fixed priority scheme with a finite number (N) of tasks with minimum interarrival rate T, a worst-case execution time Wc, deadline D, and priority P. We consider that tasks with shorter deadlines have higher priority, although our analysis can be easily extended to different hypotheses. All tasks are deemed to start at $T = 0$. The overhead of the operating system (OS) to schedule tasks is neglected for the sake of clarity. Let F_x be the extra computation required by task x if an error is detected during its execution. The value of F_x is the time required for the task full re-execution or for the partial re-execution of tasks in case checkpoints are inserted within task boundaries. When considering multiple fault ocurrences characterized by their maximum interarrival rate T_f, the worst-case response time R_i for a task i can be computed as [4]:

$$R_i = Wc_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil Wc_j + \left\lceil \frac{R_i + \Delta_e}{T_f} \right\rceil \max_{x \in hep(i)} F_x \quad (4)$$

Δ_e stands for the time an error remains dormant in the system (from occurrence until detection). As explained before, for those errors that remain long in the processor, the detection latency Δ_e is determined by the CRR scheme, i.e. errors only show up when the appropriate mechanism is triggered. In a coarse-grain checkpointing scheme $\Delta_e = I_F$ while when using a combination of coarse-grain and fine grain checkpointing we have $\Delta_e = I_L = I_F/l =$ being l the number of intermediate *Light* checkpoints within the *Full* checkpointing interval. Note that Δ_e cannot be arbitrarily reduced as the cost of *Light* checkpoints must be considered as well.

The immediate benefit of using the proposed checkpointing mechanism is the improvement on the schedulability of the system for a given fault arrival interval. Sensitivity analysis can be applied to Equation 4 to find the minimum fault interarrival interval (T_F) under which the system remains to be schedulable. Once T_F is known, the probability that during lifetime (L) no two faults arrive within a given time window (w) lower than T_F is computed using the following Equation [4]:

$$P(w \geq T_F) = e^{-\lambda L}\{1 + \lambda L + \sum_{n=2}^{\infty} \frac{(\lambda L - (n-1)\lambda T_F)^n}{n!}\} \quad (5)$$

VI. Evaluation

In this section we evaluate the proposed **LC** checkpointing mechanism using a real automotive application. The application we use is an Engine Management System (EMS). An EMS is a typical automotive embedded real-time system devoted to control the amount of fuel and the injection time to achieve smooth revolutions of the engine. The EMS application consists of eleven cyclic tasks with periods of 1, 4, 5, 8, 16, 20, 32, 64, 96, 128 and 1024 ms. As the amount of data shared across tasks is very low, we identify task boundaries as suitable points to introduce *Full* checkpoints. Further details on the application can be found in [23]. For fault injection, which is performed analytically (not through Monte-Carlo experiments), faults are considered to follow a *Poisson* distribution with rate λ. Figure 4 shows the probability of the system to remain schedulable at different failure rates when coarse-grain checkpoints are placed at 16, 32 and 128 ms. Obviously, shorter checkpointing intervals increase the probability of the system to remain schedulable. However, even for high failure rates the probability of scheduling with coarse-grain checkpointing is only 9% worse than an ideal checkpointing scheme[4]. For the hardware failure rates targeted by ISO26262, that are in the order of $10^{-11}s^{-1}$ for ASIL-D applications, the reliability achieved using the coarse-grain checkpointing scheme suffices.

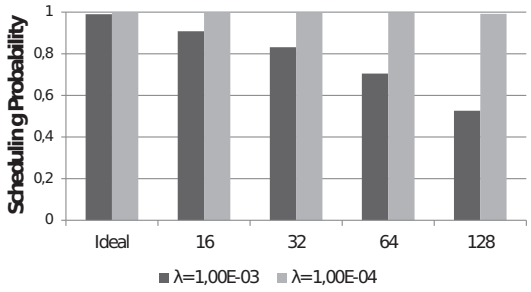

Fig. 4. Probabilistic scheduling guarantees for different failure rates and checkpointing intervals (L=3600s).

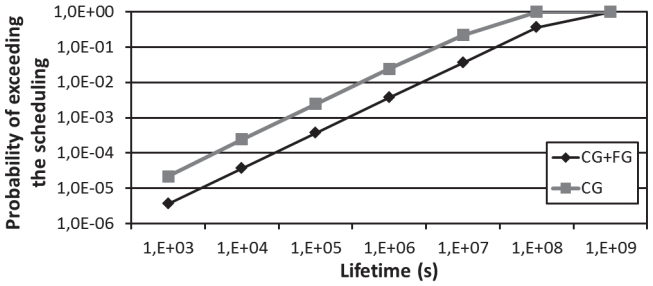

Fig. 5. Scheduling guarantees achieved with Coarse-grain (CG) and Coarse-grain plus Fine-grain (CG+FG) checkpointing ($\lambda = 10^{-7}$).

As stated before the main drawback of using a coarse-grain checkpointing scheme is the side effect this has on the maximum detection latency. In this regard, our proposal makes use of a low-cost fine-grain checkpointing scheme on top of the coarse-grain checkpointing mechanism to allow reduced detection latencies. Figure 5 shows how the improvement in the detection latency bounds increases the reliability of the checkpointing algorithm. In particular, it shows the probability of a catastrophic scenario where an ASIL-D function does not complete in time, thus jeopardizing safety. As shown, our approach reduces such probability by one order of magnitude.

[4]We consider as ideal checkpointing scheme the one having infinite frequency and zero overhead costs

Further, note that, although $\lambda = 10^{-7}$, so 4 orders of magnitude higher than for ASIL-D, it resembles the case of having 10,000 cars running such an application. Moreover, another important benefit of having reduced latency bounds is the improvement on the worst-case detection time that allows the system to transition to a fail-safe state in the presence of permanent faults as checkpointing cannot recover from those faults.

VII. Related Work

Safety-relevant systems pose a number of constraints in terms of coverage, certifiability and cost that limit the error detection and correction solutions that can be adopted in these domains. Typically, industry in these domains relies parity and error correction codes for memory devices [6] and lockstep for cores as this allows to deal with transient and permanent faults simultaneously [20], [13], [9], [14]. In this context, safety implications of faults with triple modular redundancy (TMR) have been investigated [25]. However, TMR is more expensive than dual modular redundancy needed for lockstep execution, and thus to the best of our knowledge it has not been used in the automotive domain.

Lockstep mechanisms have been regarded as effective to attain functional correctness and light versions have been deployed in processors used for safety-relevant systems [14], [9], [28]. Light lockstep, however, fails to attain any kind of timing guarantees, which are mandatory for those systems. Recently, *LiVe* has been proposed to close the gap by performing frequent checks of the processor state at low cost to expose any error with a low and upperbounded latency [11]. However, recovery mechanisms matching those error detection ones have been neglected so far for multicores, as multicores have not been widely deployed in safety-relevant systems yet and have been considered only from a general-purpose perspective [10]. Only some approaches perform fault-tolerant scheduling of tasks in non-lockstep systems for error detection and recovery [16], [29]. In this paper we review the use of some error recovery mechanisms in the context of safety-relevant multicores and show how they fail the intent. Instead, we show how by smartly combining light lockstep processors implementing *LiVe* and conventional error recovery mechanisms, timing guarantees can be attained also for error recovery.

VIII. Conclusions

The need for increased performance for safety-related systems leads to the use of multicore processors running mixed-criticality workloads simultaneously, where each application, if critical, can be run in lockstep mode. While recovery in single-core systems can be performed by means of resetting the system in the context of the automotive domain, this solution is no longer valid with mixed-criticality workloads as other critical functions may be running in the system. Moreover, restarting tasks also incurs a significant performance loss that may lead to deadline misses. In this paper we propose a cost-effective combination of coarse-grain and fine-grain checkpointing that enables the scheduling of several applications by guaranteeing quick task's recovery on an error for those hardware failure rates targeted in the ISO26262 automotive safety standard.

Acknowledgements

The research leading to these results has received funding from the ARTEMIS Joint Undertaking VeTeSS project under grant agreement number 295311. This work has also been funded by the Ministry of Science and Technology of Spain under contract TIN2012-34557 and HiPEAC. Jaume Abella is partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717.

References

[1] Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. *ARP4761*, 2001.
[2] AUTOSAR. AUTomotive Open System ARchitecture, 2012. http://www.autosar.org.
[3] C. Bolchini et al. High-reliability fault tolerant digital systems in nanometric technologies: Characterization and design methodologies. In *DFT*, 2012.
[4] A. Burns, S. Punnekkat, L. Strigini, and D. R. Wright. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Dependable Computing for Critical Applications 7, 1999*, pages 361–378, Nov 1999.
[5] K. M. Chandy. A survey of analytic models of rollback and recovery strategies. *IEEE Transactions on Software Engineering*, 1975.
[6] C.L. Chen and M.Y. Hsiao. Error-correcting codes for semiconductor memory applications: A state of the art review. *IBM Journal of R&D*, 28(2):124–134, 1984.
[7] N. Chen, Y. Yu, and S. Ren. Checkpoint interval and system's overall quality for message logging-based rollback and recovery in distributed and embedded computing. In *ICESS*, 2009.
[8] J. Espinosa, D. de Andres, J.C. Ruiz, and P. Gil. The challenge of detection and diagnosis of fugacious hardware faults in VLSI designs. In *Dependable Computing*, volume 7869 of *Lecture Notes in Computer Science*, pages 76–87. Springer Berlin Heidelberg, 2013.
[9] Freescale Semiconductor. *Qorivva MPC5643L Microcontroller Data Sheet*, 2013.
[10] D. Gizopoulos, M. Psarakis, S.V. Adve, P. Ramachandran, S.K.S. Hari, D. Sorin, A Meixner, A Biswas, and X. Vera. Architectures for online error detection and recovery in multicore processors. In *DATE*, 2011.
[11] C. Hernandez and J. Abella. Live: Timely error detection in light-lockstep safety critical systems. In *DAC*, 2014.
[12] D. Holcomb, W. Li, and S.A. Seshia. Design as you see FIT: System-level soft error analysis of sequential circuits. In *DATE*, 2009.
[13] IBM. *PowerPC 750GX Lockstep Facility. Application note*, 2008.
[14] Infineon. AURIX - TriCore datasheet. highly integrated and performance optimized 32-bit microcontrollers for automotive and industrial applications.
[15] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.
[16] V. Izosimov. *Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems*. PhD thesis, Linkoping University, 2006.
[17] S.-W. Kwak, B.-J. Choi, and B.-K. Kim. An optimal checkpointing-strategy for real-time control systems under transient faults. *IEEE Transactions on Reliability*, 50(3):293–301, Sep 2001.
[18] J. Liang, J. Han, and F. Lombardi. Analysis of error masking and restoring properties of sequential circuits. *IEEE Trans. Comput.*, 62(9), 2013.
[19] LiP6. SoCLib. www.soclib.fr/trac/dev.
[20] R.E. Lyons and W. Vanderkulk. The use of triple modular redundancy to improve computer reliability. *IBM Journal of R&D*, 6(2):200–209, 1962.
[21] M. Maniatakos, M.K. Michael, and Y. Makris. Investigating the limits of AVF analysis in the presence of multiple bit errors. In *IOLTS*, 2013.
[22] P.J. Meaney, S.B. Swaney, P.N. Sanda, and L. Spainhower. IBM z990 soft error detection and recovery. *Device and Materials Reliability, IEEE Transactions on*, 5(3), 2005.
[23] M. Panic, S. Kehr E. Quinones, B. Boeddeker, J. Abella, and F.J. Cazorla. Runpar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores. In *CODES+ISSS*, 2014.
[24] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.
[25] S. Resch, A. Steininger, and C. Scherrer. Software composability and mixed criticality for triple modular redundant architectures. In *SAFECOMP*, 2013.
[26] P. Reviriego, C.J. Bleakley, and J.A Maestro. Diverse double modular redundancy: A new direction for soft-error detection and correction. *Design Test, IEEE*, 30(2), 2013.
[27] C. Rusu, C. Grecu, and L. Anghel. Coordinated versus uncoordinated checkpoint recovery for network-on-chip based systems. In *Workshop on Electronic Design, Test and Applications*, 2008.
[28] STMicroelectronics. 32-bit power architecture microcontroller for automotive SIL3/ASILD chassis and safety applications.
[29] Y. Zhang and K. Chakrabarty. Energy-aware adaptive checkpointing in embedded real-time systems. In *DATE*, 2003.