Timely Error Detection for Effective Recovery in Light-Lockstep Automotive Systems

Carles Hernandez, Jaume Abella

Barcelona Supercomputing Center (BSC-CNS)

carles.hernandez@bsc.es, jaume.abella@bsc.es

Abstract—Safety-relevant systems in the automotive domain often implement features such as lockstep execution for error detection, and reset and reexecution for error correction. Light-lockstep has already been adopted in some such systems due to its relatively low implementation cost given that it does not require deep changes into non-lockstep hardware. Instead, as only off-core activities (i.e. data/addresses sent) need to be compared across different cores, light-lockstep designs are lowly intrusive. This approach has been proven sufficient to guarantee functional correctness of the system *in the presence of errors in the cores*, in particular in relation with certification against safety standards such as ISO26262 in the automotive domain. However, error detection in light-lockstep systems may occur long after the error actually occurs, thus jeopardising timing guarantees, which are as critical as functional ones in hard real-time systems.

In this paper we analyse the timing behaviour of errors due to transient and permanent faults in light-lockstep systems. Our results show that the time elapsed until an error is detected can be inordinately large, especially for permanent faults. Based on this observation and building upon the specific characteristics of light-lockstep systems, we propose *LiVe* (Lightly Verbose), a new mechanism to enforce the early detection of errors, due to both transient and permanent faults, thus enabling the computation of tight error detection timing bounds. We also analyse how existing mechanisms for error recovery in multicore systems increase their effectiveness when light-lockstep operates in *LiVe* mode in the context of mixed-criticality workloads.

I. INTRODUCTION

Electronics in automobiles provide an increasing amount of complex functionality, with features such as brake assist, active lane keeping, adaptive cruise control, etc. Today's vehicles are becoming more and more reliant on electronic components. Modern cars can have software components consisting of up to 100 million lines of code, and each car contains up to 70 ECUs working together under complex conditions [13]. The trend is that software and performance requirements will continue increasing in the foreseeable future. Recently Gartner Inc. has reported that “*semiconductor content of safety systems (in automotive) will almost double, from \$2.2 billion in 2009 to \$4.3 billion in 2014*” [15].

For instance, in the automotive domain many manufacturers have started incorporating systems like airbag modules, electronic parking brakes, tracking and stability control, tire-pressure monitoring and x-by-wire technology [23], [30] in the last few years. In particular, x-by-wire technology (for instance, brake-by-wire and steer-by-wire) replaces several mechanical control systems with electronic control systems using electromechanical actuators and human-machine interfaces

such as pedal and steering feel emulators [5], [10]. Hence, traditional components such as the steering column, intermediate shafts, pumps, hoses, belts, etc. can be eliminated from the vehicle. Although these safety systems have the potential to provide a big improvement in terms of reliability, multicores and modern semiconductor technologies, both needed for performance, are inherently unreliable. Therefore, electronics becomes the reliability-bottleneck. Moreover, existing safety-critical electronic systems call for increased performance. For instance, in the early 90s the software for an ABS system required an electronic control unit (ECU) at 16MHz and only 128KB of memory. By 2004, it required an ECU at 250MHz (a 15X increase) and around 1MB of memory (an 8X increase) according to ARM data [36]. The next generation of electric cars will only be realisable with significantly more powerful ECUs.

Those semiconductor technologies needed for performance reasons, however, lead to an increased number of transient faults due to higher susceptibility to cosmic rays and alpha particles, as well as due to intermittent faults caused by small defects that grow enough due to degradation to produce faults under some particular environmental conditions (e.g., low voltage, high temperature) [3]. Permanent faults also arise either because defects grow enough until they cause permanent faults, or simply because they escaped post-silicon test [3]. Transient and permanent faults lead to errors that can be tolerated to some extent in some markets, but cannot in CRTES where stringent correctness constraints call for means to prevent faults from jeopardising the safety of those systems. This is an issue for 65nm and beyond since Failure In Time (FIT)¹ rates are above the threshold affordable for many CRTES (e.g., automotive, avionics, space).

Some existing error detection and correction techniques can cope with both functional and *timing* correctness required in CRTES in the presence of faults. For instance, lockstep cores have been deployed in many systems [19], [17] at different granularities. This has been shown to be highly effective to detect errors, but if lockstep is applied at fine grain (e.g., comparing the output of each instruction or the output of each pipeline stage) it becomes too expensive due to (i) large IP modifications required, (ii) lack of flexibility to use cores in non-lockstep mode, and (iii) complex validation of the circuitry in charge of performing a number of across-core comparisons every cycle.

¹ FIT corresponds to 1 failure per 10⁹ hours of operation.

Alternatively, light lockstep cores offer lower cost and higher flexibility. Such a design suits very well mixed-criticality systems where cores may run critical tasks in lockstep mode and non-critical tasks in non-lockstep mode, as it is the case of the Infineon AURIX [20], which implements a 3-core processor based on the TriCore architecture (2 lockstep cores and 1 non-lockstep one), and the STMicroelectronics SPC56XL60/54 family [33], which implements a 2-core processor based on the Power architecture [18].

Under light lockstep mode, outputs are shared through the shared communication network for error detection. While this method is able to detect all faults occurring in the cores, it has a significant disadvantage: the time errors take to reach the bus and be detected can be long and is unbounded. With the increasing demand for more computation resources in safety-critical real-time systems, an efficient usage of computation resources is crucial. In mixed-criticality environments consisting of several applications with different criticality levels running concurrently in the same processor (in lockstep and non-lockstep mode) effectively allocating computation resources to tasks cannot be achieved if error detection mechanisms allow significant amounts of computation resources to be wasted. For example, when several applications are to be scheduled in the same processor, error detection latency directly affects the probability of a task to be schedulable [11]. This is a critical issue since CRTES must provide both, functional and *timing* correctness, which must be proven against the corresponding functional safety standards such as ISO26262 [21] for automotive and DO-178B [32] for avionics. Moreover, resetting the system to restart the faulty function may not be doable in the context of mixed-criticality multicores since other functions with potentially higher criticality levels may also be running in other cores. Thus, appropriate recovery mechanisms are needed (i.e. software triggered and/or based on checkpointing), but their efficiency is jeopardised if it cannot be determined a (close) point in time when state was still fault-free.

In this paper we analyse the timing behaviour of errors, both due to transient and permanent faults, and their implications in terms of certification in light lockstep systems, and provide an effective solution to limit the delay between error manifestation and detection. In particular, the main contributions of this paper are as follows:

- An analysis of the timing behaviour of errors due to transient and permanent faults in the context of light lockstep processors resembling the Infineon AURIX [20], proving that a non-negligible number of errors may remain undetected long after they actually manifest. We refer to those errors as *long lag errors* (LLE for short). Further we show how permanent faults lead to a higher number of LLE than transient faults due to the lower masking factor of the former.
- A low cost solution, *LiVe*, to enforce the early detection of errors in light lockstep processors by operating in a **L**ightly **V**erbose mode, thus enforcing periodic checks of the architectural state through the shared communication network.
- We show how some of the cross-domain hardware/software challenges that LLE introduce in the con-

text of the ISO26262 automotive safety standard can be efficiently addressed by using *LiVe*. In particular, we illustrate how *LiVe* facilitates error recovery.

The rest of the paper is organised as follows. Section II provides some background on lockstep-based error detection in safety-critical automotive systems. Section III presents our simulation framework and the timing analysis of error detection in light lockstep processors. Section IV introduces *LiVe*, our approach to enforce early error detection and presents some results. Section V reviews certification-friendly recovery mechanisms and how *LiVe* helps facing certification challenges. Related work is described in Section VI. Section VII draws the main conclusions of this paper.

II. ERROR DETECTION FOR AUTOMOTIVE SAFETY-CRITICAL APPLICATIONS

Redundant execution has been regarded as an effective approach for error detection, either by means of time or space replication. In particular, time replication (e.g., reexecution in the same core) [31], [27] is particularly suitable to detect soft errors. However, some significant hardware modifications are required if reexecution must occur simultaneously in a simultaneously multi-threaded (SMT) core. Alternatively, the program can be executed serially twice in a single-threaded core, but this may roughly double execution time. Either way, errors produced due to permanent and intermittent faults (e.g., those caused due to degradation or “telegraph radio noise” [4]) are very likely to repeat in both executions, thus remaining undetected. Instead, space replication requires the execution of the program in two distinct cores, typically simultaneously. Such an approach is able to detect any type of error as long as diverse implementation for the cores is used as, for instance, in the AURIX processor [20]. Diverse implementation consists of using, for instance, different sets of gates or gate designs to implement the same function so that identical layout patterns under exactly the same stress (inputs, voltage, temperature, etc.) – and so with similar process variations and similar mean time to failure – are avoided as much as possible. Next, we review the granularity at which comparison across redundantly executed programs can be performed and some particular lockstep implementations.

A. Sphere of Replication

The granularity at which outputs of redundantly executed programs are compared is usually known as the *sphere of replication* [29] or SoR for short. Such SoR can be defined at the granularity of instruction so that the output of each instruction is compared across redundant threads. However, the overhead of this approach across threads is non-negligible and cannot be performed across the shared communication network to reach memory and I/O since it would become a bottleneck if the outcome of all instructions is to be compared (i.e. values and addresses). Therefore, specific queues and communication channels are required to perform such per-instruction comparison, which is expensive. However, error detection occurs immediately when the execution of the faulty instruction completes.

Fig. 1. Schematics for different SoR.

Conversely, other approaches rely on a SoR defined at the memory and I/O interface granularity, so that only off-core activity is compared across threads. Those approaches rely on the fact that once the program finishes, only memory and I/O state matters, so in-core activity can be ignored. Thus, this approach suffices to detect meaningful errors. Also, its overheads are much lower than those when the SoR is defined at the instruction level since it is only needed to snoop the information flowing through the shared communication network to check that addresses and values sent from the cores match. Unfortunately, there is no guarantee on how long an error will take to manifest at the off-core SoR. Therefore, it is possible that an error occurs at the beginning of the execution but does not manifest until the program is about to finish. Those errors, LLE, are typically not an issue in many domains where timing is non-critical, but they jeopardise safety of CRTES, which rely on both functional and timing correctness.

B. Lockstep Systems

Many commercial systems can be clearly classified according to their SoR. For instance, HP non-stop servers [9] perform lockstep execution at a very coarse SoR since full boards are replicated and only off-board activity is compared (see Figure 1(d)). Due to such coarse granularity, intrusiveness of lockstep in the design is really low and can be easily and efficiently implemented in software layers only. Although error detection is guaranteed, long time may elapse since errors occur until they are detected. Nevertheless, target domains for such systems only care about functional correctness and timing only matters to some extent. False positives can occur in these designs if a fault manifests beyond the board boundaries but has no semantic impact in the application, thus not becoming an error. However, the likelihood of faults being errors at off-board granularity is high.

In the opposite side we can find some examples of processors delivering lockstep at the instruction (Figure 1(b)) or

even processor stage SoR (Figure 1(a)), such as for instance the PowerPC 750GX [19]. Those systems deliver immediate error detection since the time elapsed between error occurrence and its detection is upper-bounded by the latency of the longest-latency instruction. However, intrusiveness of lockstep implementation is huge as many values that are typically unobservable from outside the core must now be observable, and the amount of information shared across cores for error detection is high, thus requiring high-bandwidth communication means. Note that this kind of designs are also likely to raise many false positives due to faults that will not become errors due to fault masking. For instance, a fault may affect a register whose value will be overwritten before being read.

Intermediate solutions such as the Freescale Qorivva MPC5643L microcontroller [17] deliver lockstep at the off-core SoR (see Figure 1(c)), thus requiring low hardware support but failing to provide timing guarantees as those provided by instruction-level SoR. This low-cost design choice is the one considered in this work, where we address its timing issues. Although false positives can also occur, many of them are already filtered inside cores, so the fraction of false positives are expected to be much fewer than for for instruction or processor state SoR.

III. TIMING ANALYSIS OF ERROR DETECTION

Next we analyse the timing behaviour of error occurrence and detection in a light lockstep system as the one in Figure 1(c) whose SoR is defined at the off-core activity level.

A. Processor Model

We consider a processor resembling the Infineon AURIX 3-core processor [20]. In particular, we analyse the behaviour of two of its cores operating in lockstep mode. As in the AURIX processor, those cores operate in a way that the leading thread runs a few cycles ahead of the trailing thread. A simple hardware checker is placed in between the trailing core and the shared communication network. During lockstep operation, the hardware checker stalls the bus accesses of the trailing core and snoops leading thread activity (data, interrupts, exceptions, etc.). Buffers are needed to retain data to be sent (leading core) or compared (trailing core) until the bus is available. However, such buffering does not differ from the one needed for regular bus accesses in non-lockstep cores. On each bus access of the leading core, the checker compares the values (address and data if any) against those of the trailing core. On a mismatch an error is reported raising the corresponding interrupt. If no mismatch is detected, the trailing core remains in the same state as the leading core. For instance, leading and trailing cores are allowed to proceed if the memory or I/O request does not require any answer as in a write operation. However, only the write operation of the leading core is effectively sent. Alternatively, both cores may remain waiting for an answer (e.g., on a read operation). In such case only the read request from the leading core is sent and the answer is read by both cores. This operation mode guarantees that lockstep operation is transparent for the rest of the system and both cores operate

Fig. 2. Transient and permanent fault propagation latency (mean, median, max) across registers (address (A), data (D) and special purpose (SP)). Some registers show empty latency values as they are not used by the executed benchmarks.

with, at most, a time difference matching the time since the leading core sends a bus request until the trailing core snoops it. Note that the checker must operate at least at the same speed as the bus to keep pace with it without introducing any stall. This is not an issue since, even if the bus can send one request per cycle, simple comparators complete comparisons comfortably within one cycle.

Such architecture has been modelled with an enhanced version of the SoCLib simulation framework [24] with TriCore 1.3 binaries to implement a cycle-accurate pipelined in-order core architecture similar to the AURIX processor [20], widely used in the automotive domain.

B. Fault Injection

We inject single bit upsets (SBU) and stuck bits (both stuck-at-zero and stuck-at-one) to model transient, intermittent, and permanent faults. Note that intermittent faults will behave first as transient faults until degradation makes them large enough so that errors occur frequently or faults become permanent.

Transient and permanent faults can be injected in any processor component. However, in our experiments we have injected faults only in the register file based on the observation that faults that are not quickly propagated to the shared bus and hence, detected, will end-up in the register file. On one hand, 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 observable in the shared bus and thus, detected by the other lockstep core. On the other hand, 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 CRTES, so write operations are immediately propagated to the shared bus, thus allowing lockstep to immediately detect errors.

In general, faults in logic and latches/flip-flops reach caches and/or registers quickly and, if transient, disappear from logic and latches/flip-flops. Faults in caches are quickly propagated to the shared bus and thus, errors are detected soon by means of lockstep execution. Faults in the register file, instead, may

remain undetected for a long time and so the register file is the main target for fault injection. We perform fault injection in all registers (and only in registers) in the specification to study the timing behaviour of faults. If other registers exist in hardware and are not part of the specification, our approach should be applied on those registers analogously. Part of our future work consists of considering write-back caches so that faults may remain undetected for a long time in cache, as for the register file.

Faults may occur homogeneously across registers due to faults affecting them directly. However, they may be affected differently due to faults occurring in other processor components (e.g., combinational logic) that propagate to those registers when written. Therefore, we perform fault injection homogeneously in all registers and present combined results in two different ways:

- *Pure average.* All registers are given the same weight when computing the time elapsed between fault injection and error manifestation in the bus. Thus, this method corresponds to fault injection in the register file where all registers are equally vulnerable.
- *Weighted average.* Results for each register are weighted in accordance with the number of times they are written during execution, thus considering the impact of faults in logic and latches that propagate to particular registers.

We use the EEMBC Autobench benchmark suite [28], which is a well-known suite reflecting the current real-world demand of some automotive embedded systems. Given N_{reg} registers, each benchmark is executed $N_{reg} \cdot 100$ times, so 100 times per register. In any execution the particular register under consideration is marked as faulty exactly once at a cycle chosen randomly across the number of cycles of a fault-free execution. Then, 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 assumed to be irrelevant as the program finishes execution and they have not reached any observable device (memory or I/O). Analogously, faults that disappear without propagating become also irrelevant (e.g., because the register holding the fault is overwritten before the fault propagates). This process is known

Fig. 3. Comparison of error timing behaviour for transient and permanent faults.

Fig. 4. Transient fault masking factor.

as fault masking.

C. Results

Figure 2 shows the result of averaging out how long faults take to propagate to the bus (error latency). Error latencies due to transient faults are shown in Figure 2(a) while error latencies due to permanent faults are shown in Figure 2(b). In both plots error latency values are provided per register to show error latency variability across different registers. Registers with no bar in the plots correspond to those registers not used by the benchmark suite except D0, whose transient faults were always masked and never propagated, as shown later in Figure 4. Regardless of the type of the injected faults, errors in some frequently accessed registers (e.g., D4, D3, A4, A5) are rapidly propagated to memory and thus, shortly detected. However, a relevant number of LLE are present for a significant number of registers. For example, faults injected in special purpose (SP) registers remain long time undetected as the contents of those registers take long to propagate to the bus. Other registers devoted to some specific functions such as A10 (stack pointer) and A13 (return pointer) also take long to propagate, as they are accessed mainly at subroutine calls and returns.

If we pay attention to maximum values, we observe that some errors can propagate after several tens of million cycles after the fault actually occurred, thus wasting plenty of computation time, and thus challenging whether deadlines are met for critical tasks. In fact, we have observed that, for the benchmarks we have considered, average and maximum detec-

tion latency is only slightly shorter than the actual execution time. We have confirmed this by modifying the number of iterations of the EEMBC benchmarks, which allows increasing/decreasing their execution time. For processors running at 200-300MHz the maximum observed latencies translate into hundreds of milliseconds.

As shown, maximum error detection latency due to permanent and transient faults is very similar. However, differences in the timing behaviour of both type of faults are quite significant. In this regard, average latency for errors due to permanent faults is significantly higher than for transient faults. Figure 3 shows the fraction of errors with detection latencies above 10,000 and 100,000 cycles for transient and permanent faults. Note that the fraction of LLE is computed with respect to the total number of errors detected, not the total number of faults injected. As shown in the figure, in the case of permanent faults the fraction of LLE is much more significant than for transient faults. On average, the fraction of errors with detection latencies above 100,000 cycles is 5.1% and 0.4% for permanent and transient faults, respectively. The reason for the different behaviour across transient and permanent faults is the masking factor. Masking makes a significant fraction of the transient faults to be masked, and so disappear, before detection. Figure 4 shows the fraction of transient faults injected that are detected as errors, undetected, and masked. As shown in this figure, a very large fraction of the transient faults injected is masked. Note that some transient faults injected remain undetected at the end of the execution

Fig. 5. Average and weighted average latency.

because the register where faults have been injected are never used again in those benchmarks.

Finally, we are also interested on studying the behaviour of errors depending on whether they occur in the register file or in other components (e.g., functional units, latches, etc.) propagating to the register file. For that purpose, we have computed pure average and weighted average latency results (Fig. 5). Pure average results correspond to errors in the register file, which occur with uniform probabilities across registers. Weighted results take into account how often registers are written, as those registers written more often have higher probabilities of storing wrong values propagated due to faults in other components. Weighted average latency values reported are lower than pure average latency values. This is mainly because D15 (implicit data) and A15 (implicit address), which are the registers written more often during the execution, have lower average latency. However, despite of those lower weighted latency values, large latencies are frequent enough to challenge the lockstep architecture timing correctness as they would show up late in many cases.

IV. LiVe: ENFORCING EARLY ERROR DETECTION

As explained in Section III, in a processor with a write-through data cache all errors either reach the shared communication network almost immediately or reach the register file. Once a register holds wrong data, such data can be overwritten, propagated to other registers or propagated to the shared communication network. If wrong data are eventually sent to the network before the end of the execution of the program – a software component in automotive systems – the fault can lead to wrong program results. However, once a register holds wrong data it is non-obvious whether a correct execution state will be reached (no register holds wrong data) or wrong data will be eventually sent through the network. Thus, it is more convenient detecting a potential error soon than waiting for the fault to, hopefully, disappear (being masked) before propagating.

The simplest way to detect errors is exposing register values in the network so that they can be compared and any wrong value identified. For that purpose we propose *LiVe*, which stands for **L**ightly **V**erbose operation. *LiVe* relies on sending register file contents through the network so that error detection features of lockstep can detect any discrepancy across values in the different cores. Next we describe the mechanism in detail and evaluate its performance impact.

A. Hardware Model

LiVe is an operation mode suitable for a processor model as the one described in Section III-A. *LiVe* focuses on improving the error detection latency of light lockstep architectures. *LiVe* does not provide any further error detection capability and only those errors that can be detected with the light lockstep operation will be detected when using those processors in *LiVe* mode. The baseline processor we use employs two functionally identical cores, the leading and the trailing core, and a hardware checker in charge of detecting errors originated at cores affecting the program execution (see Figure 6). In this baseline processor faults affecting leading and trailing cores producing the same error manifestation remain undetected. To minimize this effect leading and trailing cores use diverse hardware implementations based on the use of different logic functions, different layouts, and placement, in such a way that the probability of experiencing the very same fault in the two cores simultaneously, thus leading to the same error manifestation, is minimized. Moreover, leading and trailing cores do not execute the same code at the same time which also minimizes the likelihood of simultaneous identical error manifestations. Execution in the trailing core is shifted by few cycles. In particular its execution is delayed by at least the number of cycles required to send data from the leading core to the checker in the trailing core.

B. Detailed Design

LiVe defines a *Maximum Detection Interval* (or MDI for short). Such MDI is the maximum time that 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.

Two different approaches exist for sending register values through the shared communication network:

- *Burst transmission*. This approach relies on sending all registers MDI cycles after the last time they were sent. Thus, a counter is set to MDI and decremented every cycle. Whenever it reaches zero, it is set back to MDI to start decrementing it again in the next cycle and all registers are sent uninterruptedly. If a shared bus or the like is used, this approach may cause some significant bus contention periodically, thus creating some undesirable timing effects (glitches) if those bursts occur simultaneously with other events (e.g., communications from other cores).

Fig. 6. Steps followed in *LiVe* for error detection in a register.

- *Sporadic transmission.* Alternatively, MDI can be defined as $MDI = N_{reg} \cdot IRTI$, where N_{reg} is the total number of registers and $IRTI$ stands for *Inter-Register Transmission Interval*. Then, one register is sent every $IRTI$ cycles in a round-robin fashion so that any register is sent exactly every MDI cycles. By doing so bursts are avoided and any other component competing for the shared bus will only be affected by the transmission of a single register every $IRTI$ cycles.

In *LiVe* we consider the sporadic transmission approach since it is the one with fewer side effects as glitches are avoided, thus preventing issues related to the time-alignment of events across components.

Registers communication is performed by means of non-blocking write operations (cores do not wait for any answer) whose destination address is not mapped into any device. By doing so, the register value will be exposed into the network and the core executing the trailing thread will snoop the value for error detection, but no functional impact will occur because of those write operations.

Error detection occurs as shown in Figure 6. First, both cores initiate the transmission of a particular register value (R_i in the figure). Then, the trailing core stalls until the leading core communication can be snooped. Eventually, leading core R_i value is granted access to the shared communication network and the trailing core can snoop it. Values are then compared and errors (if any) detected.

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), (ii) how much time is elapsed since a register holds a wrong value until it is sent through the network (MDI cycles at most), and (iii) 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, which depends on N_{reg} (a fixed value) and $IRTI$. Therefore, $IRTI$ must be set

Fig. 7. Performance degradation for different MID values.

low enough so that MDI is also low, but large enough so that little network bandwidth is wasted. There is no particular rule to set $IRTI$ (and so MDI) since affordable overheads and error detection latencies depend on the user needs. However, as a rule of thumb, one would expect $IRTI$ to be set so that MDI is in the order of few microseconds given that programs implementing safety-related functions may last in the order of few milliseconds [37]. For instance, if the architecture under consideration has $N_{reg} = 32$ and cycle time is 4ns (so frequency is 250MHz), $IRTI = 100$ would lead to a MDI of 3,200 cycles, so $12.8\mu s$. Next we study the impact of $IRTI$ and MDI in performance.

C. Performance

In order to study the performance overhead of *LiVe*, we use the same evaluation framework as in Section III. Although there is a tradeoff between $IRTI$ and performance overhead caused due to shared communication network contention, CRTES are typically designed not to experience high contention to guarantee low worst-case execution time (WCET) values. For instance, AURIX [20] uses a crossbar where no contention can be experienced in the network itself. Still, contention can be experienced in the device sending data if other data are ready to be sent (e.g., the core may be willing to write some data in memory). In our case we can use such crossbar as long as the core executing the trailing thread can snoop data from the leading thread crossbar.

Figure 7 shows the impact on average execution time of *LiVe* for different $IRTI$ values. Performance overhead of *LiVe* is huge for very low $IRTI$ values (e.g., $IRTI=3$, so $MDI \approx 100$) as all register values are sent in a short time frame (one register every three cycles). However, the performance overhead becomes negligible as $IRTI$ grows, being 6% for 30 cycles ($MDI \approx 1,000$) and less than 1% for values above 300 cycles ($MDI \approx 10,000$). Results confirm the effectiveness of our *LiVe* approach, which has negligible impact on performance while reducing the detection time for LLE, which takes several millions of cycles to propagate in some cases and, potentially, can take even longer if *LiVe* is not in place.

D. Analyzing *LiVe* Suitability for Harsh Environments

LiVe enforces faults in the register file to be visible before they naturally manifest. One side of the coin is that periodically sending register contents to the shared interconnection

Fig. 8. Estimated Masking factor for different MID values.

network makes error detection latencies to be upper-bounded. The other side of the coin is that a non-negligible fraction of the transient faults would have been masked but *LiVe* detects them as errors before such masking occurs. Note, however, that early detection of errors that would have been masked otherwise does not impact systems's safety. On an error detection the system can transition to a safe state. This may imply that during such transition and the recovery process the system is unavailable. However, unavailability is a lower-magnitude problem than safety. Moreover, power and performance of the system may also be affected due to the overheads introduced by the more frequent transition to a safe state and the recovery steps needed.

To quantify the impact of the masking effect degradation introduced by *LiVe* we compute the raw masking factor of several applications and how it degrades due to *LiVe*. We compute the raw masking factor by measuring the fraction of injected faults that do not lead to an error detected by the lockstep mechanism. In particular, we measure the ratio between masked faults and injected faults.

We compute the masking factor when *LiVe* is used for MDI values of 10,000 and 100,000 cycles, which show to have low performance overhead in Figure 7. Results for the different masking factors are shown in Figure 8. As shown, raw masking factors for transient faults when *LiVe* is not in place are around 68%. *LiVe* decreases slightly the masking factor down to 63% when MID is 100,000 cycles. If MDI is 10,000 cycles, masking factor decreases down to 49% on average. This behaviour is expected as most of the faults are masked quickly after they occur, thus preventing *LiVe* from prematurely detecting them. However, as MDI decreases, the likelihood of exposing wrong values that would have been masked later increases, thus leading to error detections that decrease the masking factor.

V. CERTIFICATION-FRIENDLY RECOVERY MECHANISMS

Hardware faults such as transient and permanent 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 any error and transitioning to a fail-safe state. As stated in previous sections, in the context of CRTES, error detection can be satisfactorily carried out by deploying light-lockstep architectures operating in *LiVe* mode. In this section, we analyse which recovery mechanisms are suitable for light-lockstep architectures and how those mechanisms benefit from using the system operating in *LiVe* mode. We first review the implications of

certification in error detection and recovery. Then, we show how *LiVe* helps recovery mechanisms to effectively meet certification requirements.

A. Certification Implications

Systems implementing safety-related functions need to go through a certification process. For instance, certification against ISO26262 standard [21] is required in the automotive domain. Certification is deemed as an expensive process since the number of test cases required for the validation and verification (V&V) of systems grows in pace with the complexity of those systems. Thus, V&V of multicore systems in charge of executing several safety-related functions is, at least, challenging. In the context of the hardware product development, certification activities focus on the following three aspects [21]: (1) the analysis of potential hardware faults and their effects, (2) the implementation of the technical safety concept, and (3) the coordination with the software development. For the analysis of the potential faults and their effects, extensive fault-injection experiments and tests have to be carried out to obtain suitable values of diagnostic coverage and latent fault metrics. The implementation of the technical safety concept includes the evidence of the effectiveness of the safety mechanisms to reach a fail-safe state fast enough when a fault occurs. Finally, in the coordination with the software development, the functions related to the error detection, indication, and handling of safety-related hardware elements are included. Note that hardware and software developments are tightly coupled and the interactions of those elements in relation to system's safety need to be investigated as well.

For instance, *recovery through repetition* is one of the accepted error handling methods in ISO26262 in the automotive domain. Such method consists of resetting the particular hardware components involved in the faulty execution and reexecuting affected software components (or some of its runnables as described in the AUTOSAR standard for automotive software design [6])². Additionally, according to ISO26262-6:2011 clause 10.4.3, generating test cases for software resource usage testing must allow determining the maximum execution time of the program under analysis to prove the schedulability of the integrated system. However, as shown in Section III, regular light-lockstep architectures do not provide an upper-bounded error detection latency and, this is at odds with having an upper-bounded execution time of the program.

B. Recovery of Errors due to Transient Faults

The lockstep processor model used always operates in lockstep mode. This means that two instances of the same application run in the leading and trailing cores simultaneously as required to fulfill the highest criticality levels in ISO26262. Since lockstep execution is a form of error detection but not correction per se, on an error, an interrupt is triggered and the real-time operating system (RTOS) on top is in charge of

²Note that similar concepts apply to the avionics case since avionics systems build upon the Integrated Modular Avionics (IMA) [1] software paradigm, which has significant similarities with the automotive one [35].

taking any recovery action needed. The most trivial recovery mechanism is the one that, on the occurrence of an error during application execution, 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 [21] has several limitations. As shown in Section III, error detection may occur long after error occurrence, thus leading to long time wasted performing useless computation. While this is not a challenge for functional correctness, it is detrimental for timing correctness since processor usage increases and available resources may be exhausted, leading to missing the scheduled deadlines so that the transition to a fail-safe state occurs too late or the unavailability of the system lasts too long. Furthermore, to cope with the demand for increased functionality, it is desirable to allow multiple safety-relevant applications execute in the same system simultaneously using multicore microcontrollers. If multiple safety-relevant applications with different criticality levels run simultaneously, resetting the system may not be acceptable. For instance, it cannot be allowed that an ASIL B or ASIL C application triggers 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.

In this context, checkpointing and rollback recovery mechanisms arise as an effective solution to cope with the need for selective recovery from errors of each application and to perform the recovery short after errors occur to improve the schedulability of the system in the presence of faults [12]. Checkpointing and rollback recovery requires saving snapshots of the state of the application periodically and, on an error, rollback the state to the latest error-free snapshot. Checkpoints are performed at several points in the execution of a program and in every checkpoint 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 cost of storing the application snapshots is non-negligible and both, the memory and computation time overhead due to storing the snapshot, have to be considered in the feasibility analysis. In general, checkpointing is cheap at task boundaries where just few inter-task messages need to be stored. This makes user-controlled checkpointing to be the most appealing checkpointing mechanism in the context of CRTES where typically storage space is highly constrained.

C. Recovery of Errors due to Permanent and Intermittent Faults

When *recovery through repetition* mechanisms are used in the presence of permanent faults, task execution is systematically interrupted before completion as permanent faults can lead to systematic errors that reproduce in every repetition in the same point of task's execution. For intermittent faults,

Fig. 9. Flow diagram of the permanent and transient errors recovery mechanism.

that may reproduce in several consecutive repetitions of the task, task completion is neither guaranteed. Therefore, to efficiently tolerate permanent and intermittent faults further safety mechanisms are required. Otherwise, *recovery through repetition* schemes will end-up looping infinitely and system's safety – or at least availability – will be compromised. The first requirement to effectively recover from permanent faults is fault diagnosis. The basic criterion to classify faults as either permanent or transient is error persistence [7]. For instance, this approach has already been explored in the context of faults in cache memories [2]. Note that more complex diagnosis mechanisms relying on the activation reproducibility [16] allow distinguishing between intermittent and permanent faults.

However, in the automotive domain either permanent or intermittent faults require transitioning the system to fail-safe state. In this context, error persistence is a valid metric to allow the system to take the correct action. Error persistence can be measured in terms of the number of consecutive non-error free task executions. When the error persistence is above a given threshold, an exception is raised and the system needs to transition to a fail-safe state. Safety-critical systems require the existence of safe-points that allow the system to inform the user or other sub-systems that a given task was not able to complete. Error persistence threshold is set according to the expected transient fault rate λ , which is a known parameter. If we assume a Poisson error distribution, the average number of errors in a time interval T is given by λT . Then, the expected number of errors affecting a given task is proportional to the task worst-case execution time (WCET). In general, the number of faults K_{th} above which a fault is to be considered permanent is an integer value no lower than $\lceil WCET * \lambda + N_k \rceil$, where N_k is a safety margin that has to be set according to the number of times a task can be allowed to reexecute before transitioning the system to a fail-safe state.

D. Timely Recovery with LiVe

Systems in charge of safety-related functions have to tolerate errors due to both transient and permanent faults on a timely manner. To ensure timing guarantees in the presence of errors there are two main possibilities: (1) the deadline is missed and a recovery mechanism is provided so that the system's safety is not compromised. The recovery in this case

Fig. 10. Normalized recovery time in light-lockstep systems with and without *LiVe*.

may decide not to re-execute the task and simply take the system to a safe state. (2) The system is designed in such a way that enough execution time slack (safety margin) is provided to allow one or several task re-executions (i.e. to tolerate a given number of errors).

Obviously, in a recovery-through repetition approach the safety margin is not enough to recover from more than a given number of errors. On every error diagnosing the type of fault causing the error is required. This is done based on the actual number of re-executions and the re-execution threshold (K_{th}) to either restart the task again or to set the system to a fail-safe state. The complete recovery process flow for permanent and transient fault recovery is illustrated in Figure 9. In the case of transient faults the exact number of errors that can be tolerated (recovered from) depends on when the error appears and which task the error affects, as this determines the exact amount of computation time required for the recovery. The fault-tolerant capabilities of a system running several critical tasks can be represented with a given scheduling probability [12]. The use of *LiVe* increases the probability of the system to remain schedulable as it minimizes the computation time that is wasted. Having more computation time ensures maximizing the probability of scheduling a task in the system. In other words, using *LiVe* reduces the probability of tasks missing their deadlines.

In the case of regular light-lockstep schemes the presence of LLE compromises the effectiveness of recovery mechanisms based on *recovery through repetition*. When a single task is running in the microcontroller, a LLE detected with lockstep, as any other error, forces the system to be reset and the task to be re-executed. However, in order to tolerate LLE due to permanent and transient faults very large safety time margins are needed. In practice, timing guardbands of about $K_{th} \times WCET_t$, where $WCET_t$ stands for the WCET of the task, are required in accordance with the observations in Section III, where it is shown that some LLE experience a detection latency that is in the order of task's execution time.

On the contrary, the use of *LiVe* ensures that errors are strictly detected in around MDI cycles. For instance, in the architecture considered errors manifest off-core or reach a register in few cycles (less than 10). Thus, given a MDI of 10,000 cycles, any error is detected in at most 10,010 cycles (e.g., 50 microseconds at 200MHz). If *LiVe* is not used, then error detection latency can be as high as the execution time of

tasks, which typically last in the order of some milliseconds (so around 2 orders of magnitude more than with *LiVe*).

Once an error is detected, our mechanisms also provide tight upper-bounds to the recovery and diagnosis latency as shown in Figure 10. Again, our comparison against the baseline case shows that latencies decrease from some milliseconds to some tens of microseconds only. Therefore, probabilities to successfully schedule all tasks despite errors increase.

When *LiVe* is deployed in the lockstep processor the safety time margin is drastically reduced. For instance, if *LiVe* is deployed with a MDI of 100,000 cycles and $K_{th} = 2$, 200,000 cycles are required to detect a permanent fault. Figure 10 shows the required diagnosis time for permanent faults in a light-lockstep architecture with and without *LiVe*. In this figure we show the time required for recovery when using *LiVe* with MDI=10,000 and MDI=100,000, and the time required for recovery in a regular light-lockstep architecture (*observed* and *safe* bars). Time is normalised with respect to 2 times the actual execution time of the benchmarks. Bars labeled as *observed* represent the recovery time for the maximum error latencies observed in the experiments, whereas bars labeled as *safe* represent the required time needed to guarantee a worst-case recovery. For worst-case recovery we consider that the error can potentially be detected in the last cycle of task's execution. For instance, as shown, the diagnosis latency is only $2 \times 0.25\%$ of the *cacheb* program execution time with *LiVe* (MDI=100,000 cycles) instead of roughly $2 \times WCET_{cacheb}$.

Finally, in the context of mixed-criticality systems running on top of multicore microcontrollers there are further issues to be addressed. If *LiVe* is not in place, LLE complicate the recovery process when multiple applications run concurrently. On the one hand, resetting the system for every faulty task execution is not doable when having applications with different criticality levels. On the other hand, if an error can be potentially detected after tens or hundreds of milliseconds, it is very difficult – if at all possible – to determine the exact point at which the error was produced in order to perform the recovery of the corresponding task and all other dependent tasks. In this context, the use of *LiVe* simplifies the recovery process as error detection latency can be upper-bounded to easily identify recovery points.

VI. RELATED WORK

Literature on error detection and correction is abundant. However, particular constraints of CRTES industry, such as

very high coverage, functional and timing correctness needs, and certification processes, pose severe limitations on the techniques that can be effectively adopted. Typically, CRTES industry relies on error detection and correction codes for memory devices such as main memory and caches [14] and space redundancy for the remaining devices as this allows to deal with any type of error [25], [19], [17], [20], [34]. Note that solutions based on time redundancy such as redundant multi-threaded (RMT) processors [31], [27] are typically dismissed as they do not provide guarantees to detect faults due to degradation and “telegraph radio noise” [4].

Solutions based on lockstep are popular and used in recent products to provide functional correctness. The design trade-offs of different implementations of lockstep architectures for the automotive domain have been analysed in [8]. Light lockstep architectures are attractive as little hardware overhead is incurred and those designs can be easily used in non-lockstep mode as proposed in [22]. In [26] is proposed a flexible redundancy scheme that is able to maximize the utilization of computation resources in the presence of applications with different criticality levels. Mechanisms described in [22] and [26] are orthogonal to our proposal, which is used only under lockstep operation. It must be noted that the approach in [26] reduces the hardware overhead due to redundancy by making trailing and leading cores to smartly share L1 data cache. This reduction in the hardware resources comes, however, at the expense of limiting the flexibility of the design, since running in non-lockstep mode, as needed for lowly critical applications, may not be possible. Finally, several light lockstep systems have been recently proposed by some of the main chip vendors in the automotive domain [20], [17], [34]. However, light lockstep fails to provide timing correctness as needed in safety-related functions, since errors may be detected long after they occur, thus potentially exhausting the time slack available for re-executing faulty functions. To the best of our knowledge, our solution, *LiVe*, is the first approach guaranteeing functional and timing correctness on top of light lockstep CRTES.

VII. CONCLUSIONS

Safety-critical functions require means for error detection and correction to guarantee functional and timing correctness, which must be proven against functional safety standards such as ISO26262 in the automotive domain. Lockstep execution has been widely deployed for that purpose. However, redesigning cores for timely error detection raises costs and decreases flexibility as it is unclear how lockstep cores can operate in non-lockstep mode, as needed in mixed-criticality systems.

In this paper we propose *LiVe*, a simple solution to enable both functional and timing correctness in light lockstep systems where design costs are kept low and flexibility is attained so that cores can be used in either lockstep or non-lockstep mode. We show how *LiVe* fits the needs of safety standards and prove its negligible impact in performance. We also analyse the side effects of *LiVe* in the context of error recovery and prove that existing recovery mechanisms can be much more effective on top of processors implementing *LiVe*.

The investigation of recovery mechanisms paired with *LiVe* other than periodic checkpointing are left for future work.

ACKNOWLEDGMENTS

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 has been 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] J. Abella, P. Chaparro, X. Vera, J. Carretero, and A. Gonzalez. On-line failure detection and confinement in caches. In *IOLTS*, 2008.
- [3] J. Abella et al. Towards improved survivability in safety-critical systems. In *IOLTS*, 2011.
- [4] M. Agostinelli et al. Erratic fluctuations of SRAM cache Vmin at the 90nm process technology node. In *IEDM*, 2005.
- [5] S. Amberkar, J.G. D'Ambrosio, B.T. Murray, J. Wysocki, and B.J. Czerny. A system-safety process for “by-wire” automotive systems. In *SAE World Congress*, 2000.
- [6] AUTOSAR. *Technical Overview V2.0.1*, 2006.
- [7] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.
- [8] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini. Fault-tolerant platforms for automotive safety-critical applications. In *In Proc. of the Intl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, pages 170–177. ACM Press, 2003.
- [9] D. Bernick et al. Nonstop advanced architecture. In *DSN*, 2005.
- [10] E.A. Bretz. By-wire cars turn the corner. *IEEE Spectrum*, 38(4), 2001.
- [11] 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.
- [12] 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.
- [13] R.N. Charette. This car runs on code. In *IEEE Spectrum online*, 2009.
- [14] 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.
- [15] P. Clarke. Automotive chip content growing fast, says gartner. In <http://www.eetimes.com/electronics-news/4207377/Automotive-chip-content-growing-fast>, 2011.
- [16] Jaime Espinosa, David de Andrs, Juan-Carlos Ruiz, and Pedro Gil. The challenge of detection and diagnosis of fugacious hardware faults in vlsi designs. In Marco Vieira and JooCarlos Cunha, editors, *Dependable Computing*, volume 7869 of *Lecture Notes in Computer Science*, pages 76–87. Springer Berlin Heidelberg, 2013.
- [17] Freescale Semiconductor. *Qorivva MPC5643L Microcontroller Data Sheet*, 2013.
- [18] IBM. Power ISA version 2.06 revision B.
- [19] IBM. *PowerPC 750GX Lockstep Facility. Application note*, 2008.
- [20] Infineon. AURIX - TriCore datasheet. highly integrated and performance optimized 32-bit microcontrollers for automotive and industrial applications.
- [21] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.
- [22] C.M. Jeffery and R.J.O. Figueiredo. A flexible approach to improving system reliability with virtual lockstep. *Dependable and Secure Computing, IEEE Transactions on*, 9(1):2–15, Jan 2012.
- [23] G. Leen and D. Heffernan. Expanding automotive electronic systems. *IEEE Computer*, 35(1), 2002.

- [24] LiP6. SoCLib. www.soclib.fr/trac/dev.
- [25] 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.
- [26] Brett H. Meyer, Nishant J. George, Benton H. Calhoun, John Lach, and Kevin Skadron. Reducing the cost of redundant execution in safety-critical systems using relaxed dedication. In *DATE*, pages 1249–1254. IEEE, 2011.
- [27] S.S. Mukherjee, M. Kontz, and S.K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *ISCA*, 2002.
- [28] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.
- [29] S.K. Reinhardt and S.S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA*, 2000.
- [30] S. Rohr, R. Lind, R. Myers, W. Bauson, W. Kosiak, and H. Yen. An integrated approach to automotive safety systems. *SAE Automotive Engineering International magazine*, September 2000.
- [31] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *FTC*, 1999.
- [32] RTCA and EUROCAE. *DO-178B / ED-12B, Software Considerations in Airborne Systems and Equipment Certification*, 1992.
- [33] STMicroelectronics. 32-bit power architecture microcontroller for automotive SIL3/ASILD chassis and safety applications.
- [34] STMicroelectronics. *32-bit Power Architecture microcontroller for automotive SIL3/ASILD chassis and safety applications*, 2014.
- [35] T. Ungerer et al. parMERASA - multi-core execution of parallelised hard real-time applications supporting analysability. In *Euromicro DSD*, 2013.
- [36] B. Vittorelli. Automotive systems. where silicon meets the road. *Information Quarterly Building Blocks for Embedded Applications*, 3(4), 2004.
- [37] F. Wartel et al. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *SIES*, 2013.