Fault and Timing Analysis in Critical Multi-Core Systems - A Survey with an Avionics Perspective

Andreas Löfwenmark
Saab Aeronautics
Linköping, Sweden
andreas.lofwenmark@saabgroup.com

Simin Nadjm-Tehrani
Department of Computer and Information Science
Linköping University
Linköping, Sweden
simin.nadjm-tehrani@liu.se

Abstract*—With more functionality added to future safety-critical avionics systems, new platforms are required to offer the computational capacity needed. Multi-core processors offer a potential that is promising, but they also suffer from two issues that are only recently being addressed in the safety-critical contexts: lack of methods for assuring timing determinism, and higher sensitivity to permanent and transient faults due to shrinking transistor sizes. This paper reviews major contributions that assess the impact of fault tolerance on worst-case execution time of processes running on a multi-core platform. We consider the classic approach for analyzing the impact of faults in such systems, namely fault injection. The review therefore explores the area in which timing effects are studied when fault injection methods are used. We conclude that there are few works that address the intricate timing effects that appear when inter-core interferences due to simultaneous accesses of shared resources are combined with fault tolerance techniques. We assess the applicability of the methods to currently available multi-core processors used in avionics. Dark spots on the research map of the integration problem of hardware reliability and timing predictability for multi-core avionics systems are identified.

I. Introduction

Added functionality in future avionics systems brings complexities to both design and operation of these systems and requires new platforms that offer more computational capacity. Multi-core processing offers a potential that the industry is exploring, and which opens up for new research questions in the context of safety-critical systems. Commercial off-the-shelf (COTS) multi-core processors are inherently less predictable because of shared resources [1,2] and the efforts of the chip manufacturers to optimize the throughput, which affect the analyses of worst-case execution time (WCET) and worst-case response time (WCRT). In the absence of new techniques, these analyses tend to result in very pessimistic estimates, which could negate the intended addition of computational capacity.

In this paper, we review the state of research that addresses the joint study of timing predictability and fault tolerance in the multi-core setting. The elementary fault classes of interest are dimension (hardware, software) and persistence (transient, permanent) [3]. We analyze proposed fault tolerance methods targeting these fault classes and whether their impact on WCET analysis is addressed. We also assess the applicability of the surveyed methods to safety-critical systems on multi-core platforms within a practical setting.

Avionics systems operate at high altitude and are more exposed to cosmic rays than electronics at ground-level. These cosmic rays have sufficient energy to alter the states of circuit components, resulting in transient faults, which could result in corrupted data (e.g., in caches and memory). These transient faults are often referred to as *soft errors*, as the effect of the fault will disappear when for instance a memory location is overwritten. Permanent hardware errors (e.g., resulting from deficiencies in manufacturing processes or component failure), are permanent in time. As the technology advances and the transistor sizes shrink, both permanent and transient faults will increase, thus making the soft errors a bigger problem for industries at sea level (e.g., automotive and railway) as well. Safety and reliability requirements of the system mandate making a serious attempt to make it fault-tolerant by masking both permanent and transient faults.

Caches occupy a large area on the processor chip and are vulnerable to soft errors. Therefore, to mitigate soft errors the caches are equipped with error detecting (and sometimes also error correcting) codes. The simplest detection type is parity where (typically) one extra bit is used per byte to indicate whether the number of 1-bits in the byte is even or odd. This can later be used to detect an error. An error-correction code (ECC) is required to correct errors and a common error correcting code is the single-error correction and double-error detection (SECDED) Hamming code [4].

Software faults are faults (bugs) in the software, which could result from the design or implementation phase. They are permanent in nature, but the effects of the bugs can be transient [5]. In this paper, we refer to these kinds of bugs as transient software faults. Other terms for software faults are Bohrbug (permanent), Mandelbug and Heisenbug (transient) [6,7].

The fault tolerance mechanisms used to ensure the safety and reliability of the system have to be verified and validated. Waiting for cosmic rays to alter the operational state of an observable component is not viable as a means of documentation. Fault injection can be used to introduce a fault in the component by more controllable means. Radiation beams can

be used to actually change the state in the hardware component or some other methods can be used to emulate a fault.

Other means, such as code review, static analysis and model checking, for verifying and validating the system and its fault tolerance mechanisms are also available (e.g., [8]) and often mandated by safety standards. However, in this paper we focus on fault injection as the assessment method for impacts of faults.

To use multi-core processors in a safety-critical system, both problems (i.e., timing predictability and fault tolerance) have to be addressed. Safety-critical systems have to produce the correct output within the allotted time and different fault tolerance methods have different impact on the execution time of the tasks in the system. We also have to ensure the verifiability of the chosen fault tolerance mechanism and the WCET estimates. To create repeatable experiments, we need appropriate fault injection methods.

Specifically, we seek answers to the following questions in a multi-core context:

1) Can we bound WCET estimates in presence of fault tolerance?
2) Can we validate fault tolerance and timeliness claims using fault injection?
3) What fault tolerance methods can be implemented with current COTS processors?

Reviewing the existing research in the area creates a base for understanding which building blocks are available from both perspectives (timing and fault tolerance), and for making informed architectural decisions when allocating applications to nodes, partitions or cores, as well as their impact on other resource boundaries (e.g., communication buses).

Others have surveyed related research areas. Mushtaq et al. [9] surveyed fault tolerance techniques for shared memory multi-core systems in 2011, which is similar to our work. However, they did not include the effects that fault tolerance techniques have on WCET estimations. Natella et al. [10] have performed a survey of software fault injection (SFI) for systems executing on single-core processors. Their survey does not cover multi-core platforms nor the applicability of fault injection to verify WCET estimation claims, which we aim to investigate.

While deploying multi-core platforms for high performance computing has attracted much attention including prediction models for performance of virtualised environments [11], there has been less attention paid to embedded systems in which throughput is not the only concern and indeed dependability concerns are as important [12]. This work intends to bring the need for multi-core based research in time-dependent and dependable systems to the forefront.

Although critical systems exist in several domains (e.g., avionics, automotive, railway) and we discuss topics relevant for all of them, in this paper we focus on the avionics domain.

The remainder of the paper is structured as follows. Section II includes the basic fault tolerance and fault injection concept relevant to the rest of the paper. Readers familiar with basic notions in dependability and safety can skip this section and move on to the next section. Section III and Section IV contain summaries of the surveyed papers related to fault tolerance (Question 1) and fault injection (Question 2) respectively. We discuss the papers and the implications of the covered methods on WCET estimates in Section V, where we also analyse the methods with regard to practical usability (Question 3). The paper is concluded in Section VI.

II. Basic Concepts

In this section, we review some basic notions used in the rest of the paper.

A. Fault Tolerance

We use the fundamental concepts of faults, errors, failures, transient, intermittent, permanent and so on, in accordance with the well-known notions of dependability [3].

Faults can originate from hardware or software. The majority of software faults found in production software are transient as most of the other bugs, those that always fail, should have been discovered in the process of design, review and test [7].

The process of making a system fault-tolerant can be summarized in four steps [9]: *proactive fault management*, *error detection*, *fault diagnosis* and *recovery*. In the proactive step one tries to predict failures and also prevent them from happening. Examples of proactive fault management are software rejuvenation and on-line checks during run-time (such as memory scrubbing). A watchdog can be used to detect timing errors, which could be a result of implementation mistakes or bit-flips due to high-energy particles (cosmic radiation). Both of these effects could result in the system being stuck in an infinite loop. Other types of errors can be treated using redundancy at different levels in the system. When an error is detected, methods to locate the faulty component and also the type of the fault can help with the handling of the error. Using Triple Modular Redundancy (TMR) for instance can help identify a faulty component. It is important to mitigate the fault before a failure is triggered.

Depending on the system, there are different ways to handle the fault (or error). One way is to shut down the system until the fault can be repaired, but for some systems this is not an option. In such cases, TMR can be used to handle faulty components if the probability of all three redundant components failing at the same time is considered low. Other solutions use a checkpoint and repair methodology, which periodically saves the execution state (creating a checkpoint) and when a fault is detected the execution is rolled back to the checkpoint. Fault masking is also a way of recovering from faults. TMR is an example of fault masking that uses three

redundant components and majority voting to mask deviating data. Single event upsets (SEU) are so called "soft errors" caused by a single energetic-particle strike, and single event functional interrupts (SEFI) are soft errors that cause the component to reset, lock-up, or otherwise malfunction in a detectable way, but does not require power cycling to restore operability.

B. Fault Injection

To study the impact of transient faults and to determine how they should be treated one could wait for a fault to happen, but this would be very time consuming as transient faults can be rare. Fault injection is a technique for evaluating the dependability of systems. It can be used to inject faults into a system with the aim of observing its behaviour and assessing the fault tolerance mechanisms. It is thereby a practical approach for achieving confidence in that faults cannot cause serious failures.

This is done by intentionally introducing faults in the hardware on which the application runs. Fault injection can take many forms and can be introduced in different phases of system development. Radiation beams is an example of physical injection methods that can be used to inject faults in a hardware circuit and can produce transient faults in random locations inside a hardware circuit, but it is a time-consuming and expensive task. Alternatives have been developed, some hardware-based, some software-based and some simulation-based [4].

Hardware-based fault injection includes radiation-beam testing, risking permanent damage to the tested device and requires special hardware instrumentation. Software-based fault injection is cheap and is achieved by altering the contents of CPU registers or memory while running the relevant application software on the hardware being tested. The faults are simulated using a fault model, and the selection of fault model is an issue that influences the potential outcomes of the injection experiments. Simulation-based fault-injection offers complete visibility inside the device under test provided that the simulation model of the hardware is detailed enough.

Software fault injection (SFI) attempts to simulate software faults through code changes, either at compile-time by modifying the source code or at run-time by using a trigger and then changing data in e.g., memory or registers. Mutation testing modifies existing lines of code and can be used to simulate faults unintentionally introduced by programmers. SFI can be used to inject code changes, data errors and interface errors. Addressing the questions of when, where and what to inject are important to create an efficient fault injection method.

C. Safety-Critical Systems

Regardless of which domain a critical system belongs to, it is subjected to international safety standards containing guidance for validation and certification. Table I shows the safety levels of a few major domains.

Table I
COMPARISON OF SAFETY LEVELS [13]

Domain	Safety Levels (high to low)				
Avionics DO-178C DAL[1]	A	B	C	D	E
Automotive ISO 26262 ASIL[2]	-	D	C/B	A	QM
General IEC-61508 SIL[3]	4	3	2	1	-
Railway EN 50128 SIL[3]	4	3	2	1	-

[1] Design Assurance Level
[2] Automotive Safety Integrity Level
[3] Safety Integrity Level

As an example, development of safety-critical airborne software is guided by the standard RTCA/DO-178C [14] containing a number of objectives to be fulfilled. The software is assigned a design assurance level (DAL), ranging from level A (most critical) to E (non-critical), depending on the criticality as determined in the safety assessment process.

In addition to RTCA/DO-178C, the certification authorities have issued the Certification Authorities Software Team position paper (CAST 32A) [15] that identifies topics impacting safety, performance, and integrity of an airborne software system executing on multi-core processors. It is required that probabilistic safety guarantees are provided to functionalities of different criticality as shown in Table II. Clearly, higher criticality levels require more stringent analyses of both hardware and software to ensure the assurance levels needed. Worst-case execution time (WCET) estimates need to be determined more thoroughly and tend to be higher (more pessimistic) in higher criticality levels.

Table II
FAILURE RATE PER RTCA/DO-178C CRITICALITY LEVEL

Level	Failure Condition	Failure Rate
A	Catastrophic	10^{-9}/h
B	Hazardous	10^{-7}/h
C	Major	10^{-5}/h
D	Minor	10^{-3}/h
E	No Effect	N/A

D. Mixed-Criticality Systems

Most embedded systems consist of many different functions. However, all functions in a system are not equally critical for correct service or mission (e.g., the flight control system in a commercial airliner or the braking system in a car is more critical than the infotainment system). A mixed-criticality system is a system hosting several different functions assigned different safety levels (DAL/ASIL/SIL) on the same computing platform. When estimating WCETs for the functions, the

most critical ones may get a more stringent analysis than the less critical ones.

The introduction of multi-core supports the migration from federated systems (functions implemented and packaged as self-contained units) to integrated systems. Integrated Modular Avionics (IMA) [16] and AUTomotive Open System ARchitecture (AUTOSAR) [17] are examples from the avionics and automotive domains respectively.

The multi-criticality task model was proposed by Vestal [18] after showing that existing scheduling theory could not address multiple criticality requirements. Multiple WCET estimates are specified for each task, one WCET for the criticality level the task belongs to and one for each lower level. Higher criticality levels demand greater levels of assurance and the WCET will therefore be more pessimistic for the same task at a lower criticality level. Similarly, treatment of tasks in terms of resource allocation, and analysis of fault tolerance can be differentiated depending on the task criticality.

E. WCET Estimations

WCET estimations are essential ingredients for establishing a predictable timing behaviour in safety-critical systems, but the introduction of multi-core processors has made their estimation even more difficult to perform. Having to account for failures complicates the task even further. WCET estimation can be classified as static or measurement-based.

A static analysis method computes the execution time of individual code blocks using a micro-architectural model of the target platform they will execute on. By design, it will find the longest path and can thus provide a safe (overestimated) upper bound instead of the WCET. The amount of overestimation is dependent on the micro-architectural model, which can be very difficult to produce for complex platforms (e.g., multi-core with multi-threading, branch prediction, pipelining and multi-level caches).

Measurement-based analysis can provide accurate execution times as the software is running on the target platform, but since it depends on actual execution it may be difficult to know whether the WCET path has been covered. There is always a risk that the WCET estimation is not equal to the real WCET.

Hybrid WCET estimation techniques also exist, where the application is executed (several times) on the target hardware and execution traces are collected. With the execution traces, together with knowledge of the code structure, a WCET estimate can be produced. For this to function, all statements in the application source code have to be executed during trace collection.

Probabilistic timing analysis (PTA) [19,20] can be used to calculate a probabilistic WCET (pWCET). The failure rates specified for different DALs (Table II) can be utilized during the pWCET calculations to find the relevant target probability. Two variants exist, static (SPTA) and measurement based (MBPTA), similar to the classical deterministic WCET estimations. These pWCET estimates are derived in such a way as to provide indications of likelihood (e.g., the estimates can be exceeded with a given probability [20]). Instead of requiring time-deterministic behaviour of the platform, PTA is based on randomization of the timing behaviour for some hardware (e.g., caches and shared buses).

Having covered the basic terms and notions, we now move on to review a selection of works within the real-time and fault-tolerant systems that we believe have the initial seeds needed to address our main problem, i.e. building integrated modular avionics (IMA) systems on multi-core platforms and provide the timing determinism and dependability requirements for these. We will therefore cover both methods from the single-core world and multi-core techniques.

III. Fault Tolerance

In this section, we aim to investigate Question 1, referring to worst-case execution time (WCET) bounds in presence of faults tolerance. The question can be viewed from two different perspectives: (a) Fault tolerance methods can consider WCET aspects; and (b) Timing analysis methods can be fault aware. We treat each perspective separately. In Section III-A we consider (a) and (b) is considered in Section III-B.

To validate claims about fault tolerance, fault injection methods are used, but the actual injection method is not a contribution of the papers reviewed in this section. Fault injection (related to Question 2) is surveyed in Section IV. Later, in Section V we will summarise all the categories covered here in a summary table (Table III) and discuss in the context of our aims. The selection of articles has been guided by the criteria of potential applicability to safety-critical systems assurance, since our intention is to finally judge where the gaps for applicability to safety-critical systems lie.

A. Fault Tolerance Methods

In this section we look at representative fault tolerance methods in presence of transient and/or permanent hardware faults. Some also consider software faults. An obvious category of fault tolerance works are those that exploit the multi-core platform itself as a means of achieving resilience to faults, namely by adopting *replication*. We begin by reviewing examples of such works, and then move on to works that use a *checkpointing and recovery* technique running on multi-core. In a third category, we consider the works that use *fault forecasting* to combine with the hardware-assisted fault detection, and finally we give an example of works that not only detect but also build in *recovery from attacks*. All these techniques are highly representative of methods that one needs to consider in a high-integrity and safety-critical system.

1) Multi-core platform used for replication: In systems with cost and space constraints, such as automotive, the use of replicated hardware is not feasible. A software-based fault tolerance method can be used in a *distributed system*, where tasks are replicated on other nodes. Kim et al. [21] present SAFER (System-level Architecture for Failure Evasion in Real-Time applications), incorporating fault tolerance methods to tolerate fail-stop processor and task faults. Hot and cold standbys can be configured for the primary tasks, where hot standbys execute the same instructions as the primary, but generate no output. The cold standbys on the other hand are dormant and only awoken if the primary fails. The state of the primary is communicated to the node with the cold backup to ensure that the cold standby can take over the role as primary. All replicas (hot and cold standbys) must be placed on nodes other than the one where the primary resides. A schedulability analysis including all tasks (primaries and backups) for the whole distributed system is also provided. Bhat et al. [22] address some of the limitations of SAFER, such as using the AUTomotive Open System ARchitecture (AUTOSAR) instead of Linux. The worst-case recovery time of the task model is analysed and they also present a task mapping heuristic to minimize the number of required nodes in the system and still meet the placement constraint.

Multi-core platforms can be used to enhance fault tolerance, in a similar way to the distributed systems described above, where redundant processes can execute on different cores instead of processors. Shye et al. [23] present process-level redundancy (PLR), a software-centric paradigm in transient fault tolerance on multi-core platforms. They use several cores to deploy redundant processes of the original (single-threaded) application process and then compare the results to detect transient faults. One master process is replicated several times to create the redundant slave processes (replicas). Since all processes execute the same instructions, care must be taken to ensure that any system state that is modified is only modified once as if the original process is executing by itself (preserving the semantics of the master process). This has been solved by a system call emulation unit that is inserted (using the LD_PRELOAD feature of Linux) between the process and the operating system. The emulation unit intercepts the application start (to create redundant processes as well as the original process) and the system calls, and ensures that system calls that return non-deterministic data (such as time of day) will be handled in a way that all replicas see the same value. This method works well on systems where throughput is not the primary goal. To be able to both detect and recover from transient faults, at least three redundant processes (the original process and two replicas) are needed.

Marshall et al. [24] present a framework for component-based systems, S^3RES, similar to PLR. S^3RES does not use system call emulation as PLR does, but redirects the input and output channels to a user-space voter process.

Quest-V, proposed by Missimer et al. [25], implements fault tolerance by leveraging hardware virtualization to isolate the cores of a multi-core processor. This way, redundant tasks that are typically run on separate processors to form a triple modular redundancy (TMR) system can be consolidated on one multi-core processor. To further improve fault detection, hashes are calculated on memory modified by the programs between synchronization points. These hashes are compared by the voter in the TMR system to detect deviations in the redundant computations.

While Shye et al. [23] only cover single-threaded applications, multi-threaded applications are addressed by Mushtaq et al. [26]. They introduce a record/replay approach to make multi-threaded shared memory requests deterministic in addition to the use of redundant processes. The order of all shared memory requests performed by the original process is recorded and later replayed by the replicas. Recovery is handled by checkpointing and rollback. This method requires communication between the original process and the replicas, and the memory used for the communication can also become corrupted, which makes it less reliable. Therefore, Mushtaq et al. extend their work and introduce deterministic multi-threading [27] instead of record/replay. The definition of deterministic multi-threading is that given the same input, the threads of a multi-threaded process always have the same lock interleaving. To ensure that the locks are acquired in the same order, they introduce logical clocks that are inserted at compile-time. When a thread is trying to acquire a lock, it is only allowed to do so if its logical clock has the minimal value. A number of optimizations are introduced to the logical clock handling to reduce the overhead. The fault tolerance method adds an average overhead of 49 percent to the execution time in absence of faults.

The fault tolerance methods discussed so far use uniform multi-core processors, where all cores are identical (i.e., so called homogeneous multi-core). Another variant is explored by Meisner and Platzner [28], where they combine “normal” cores with cores implemented in a field-programmable gate array (FPGA) (i.e., heterogeneous or hybrid multi-core). They propose a dynamic redundancy technique, named thread shadowing, which duplicates (shadows) a software or hardware thread during a time period. One advantage of using hybrid multi-cores is that a software thread can shadow a hardware thread, or the other way round (which is called trans-modal). Hardware threads are often much faster than software threads and this can be utilized in a trans-modal round-robin shadowing (i.e., one hardware thread shadows a number of software threads for a time period one at a time). This requires only one extra core, but does not provide continuous error detection. It is also possible to increase the number of cores to shadow each thread and provide a more comprehensive error detection, which is more suitable for transient faults. The former configuration can be used to detect permanent hardware faults.

Thread duplication and majority voting is suitable for masking

transient faults, but it is more difficult to detect permanent faults. The question is also what to do when identifying a permanent fault. Peshave et al. [29] use a reconfigurable Chip Multi-Core Processor (CMP) to provide redundancy in order to improve reliability. They mask transient faults and tolerate core-level permanent faults. The framework consists of a TMR system that uses dual-core CMPs, where one of the cores is deactivated and used as a redundant core in case of faults in the running one. Each of the three CMPs execute a copy of the application and the output from each is used in a voting procedure to produce the final result. The voter is included in a separate hardware block that monitors the system and can for instance switch to the redundant core on any CMP if a permanent fault is detected. This system is compared to a standard TMR system using single-core processors and they conclude that the dual-core system can tolerate more core-level permanent and transient faults than the single-core based system.

2) Software-assisted checkpoint and recovery: Even though the semiconductor technology evolution resulting in smaller and smaller transistors increases the sensitivity to both permanent and transient faults, the fault-free operation is still the common case. Hari et al. [30] focus therefore on light-weight detection mechanisms, whereas the relative rare operation of fault diagnosis is allowed to use more resources (e.g., execution time) and is performed by a replay task. They develop a diagnosis algorithm for multi-threaded applications on multi-core systems. The algorithm is based on repeated roll-back/replay and can deterministically replay execution from a previous checkpoint for a multi-threaded application, which is a requirement for a proper diagnosis. Several replays may be necessary to distinguish between software bugs, transient and permanent hardware faults, and to identify the faulty core in case of a permanent fault.

The components that implement the different fault tolerance methods, such as replication and checkpointing mentioned above, are also vulnerable to faults unless care is taken to protect them. Hoffmann et al. [31] propose dOSEK, which aims to be a robust real-time operating system that provides a reliable computing base (RCB) on which the fault-tolerant applications can be implemented. The RCB is the set of components that are expected to be reliable to ensure an operational fault tolerance method. To keep the RCB to a minimum, dOSEK is designed and implemented using static techniques to minimize the amount of code and data in the operating system vulnerable to transient faults. A code generator is also used to tailor the system. The operating system kernel is further hardened by using arithmetic encoding of kernel data structures to detect both data and control flow faults.

Song et al. [32] present C^3, which is a system-level fault tolerance mechanism implemented on the Composite component-based operating system. Components in Composite execute at user-level in a private protection domain. Even system-level functions, such as scheduling and memory management, are user-level components. C^3 focuses on recovery from transient faults without hardware or process-level replication. The goal of C^3 is to rebuild the internal state of a failed system component. This is done by tracking the state of components at the interface boundary and when a fault occurs, the component is "rebooted" and the saved state is restored. To better handle deadlines in the presence of faults, they include the overhead of C^3 in the presented schedulability analysis. Song and Parmer [33] extend Composite and C^3 with C'Mon, which is a monitoring infrastructure that keeps track of all communication in the system and validates that the behavior conforms to a model specified offline. C'Mon can detect and recover from latent faults in system services. The work also includes a system overhead aware schedulability analysis for systems using C'Mon.

3) Fault forecasting combined with error correction: The majority of transient faults will be spatial multi-bit faults as the technology scaling continues towards smaller and smaller feature sizes [34]. A spatial multi-bit fault is a single-event upset affecting more than one bit. If the affected bits belong to the same protection domain, the common single-error correction and double-error detection (SECDED) error correction code cannot be used for correction as it only corrects single-bit faults. Therefore, Manoochehri and Dubois [34] develop a formal model (PARMA+) to benchmark failure rates of caches with spatial multi-bit faults and different protection schemes. They focus on the cache because the caches occupy a large area on the processor chip and are vulnerable to soft errors, which will affect reliability. PARMA+ estimates the cache failures in time (FIT) rate. The FIT rate of a device is the number of failures that can be expected in one billion (10^9) device-hours of operation (e.g., 1000 devices for 1 million hours, or 1 million devices for 1000 hours each, or some other combination). A component having a failure rate of 1 FIT is then equivalent to having a mean time between failures (MTBF) of 1 billion hours. Most components have failure rates measured in hundreds or thousands of FITs. This is not really a fault tolerance method, but the model can help chip designers to configure reliability enhancing protection schemes in a more optimal way, resulting in more reliable components.

One fault forecasting method targeting railway transportation systems is Timed Fault Trees (TFTs) proposed by Peng et al. [35], which extends traditional fault tree analysis with temporal events and fault characteristics. TFTs can be used to predict and prevent accidents and also be applied to a system at design time.

Being able to predict a forthcoming failure is very important in cyber-physical systems. Chen and Sankaranarayanan [36] present a linear model-predictive scheme for monitoring linear systems. The monitor keeps a list of reachable set predictions and reports an unsafe incident or an alert when a prediction that is uncontrollable is detected. This way, one can switch from a high-performance controller, that may be unsafe, to a

safe controller.

Sangchoolie et al. [37] study the impact of multiple bit-flip faults and compare whether the number of silent data corruptions (SDCs) increases compared to single bit-flips. They also seek ways to prune the multi-bit fault injection space. Their conclusion is that multiple bit-flip faults do not result in higher rates of SDCs compared to single bit-flip experiments.

4) Hardware-assisted detection and attack recovery: The fault tolerance mechanisms described so far are all designed to handle random sporadic faults (and permanent faults in caches), but faults can also be used to attack a system. In this case, the faults are injected by an adversary in a well-planned manner. Hence, the fault-tolerant mechanisms discussed so far are not enough for fault-based attack resistance. Yuce et al. [38] propose FAME (Fault-attack Aware Microprocessor Extensions), a hardware-based fault detection that is continuously monitoring the system and a software-based fault response mechanism (software trap handler) that is invoked when a fault is detected. A fault control unit (FCU) collects information to ensure that a recovery is possible from the software trap handler. The FCU also flushes the processor pipeline and disables write operations to the memory and registers to ensure that no faulty results are committed to the processor state and the software trap handler is restarted if the fault detection unit (FDU) detects a fault during execution of the trap handler. This ensures the completion of the trap handler before resuming normal execution.

B. Fault-Aware Timing Analysis

Most of the fault tolerance methods presented in the previous section do not consider the impact they have on the execution (response) time of tasks and perform no timing analyses. The focus is on delivering the correct function and none of these methods can be used (without alterations) to answer Question 1. Some works [21,22,32,33] do take into account how the fault tolerance method impacts schedulability and provide a schedulability analysis. However, these are either on multi-processor systems (separate processors communicating over a network with no shared resources) or on single-core processors. Therefore, they cannot be directly applied to multi-core processors. Many embedded systems will have strict requirements on the worst-case execution time (WCET) of different functions, hence fault tolerance methods cannot ignore their impact (interference due to shared resources) on WCET and schedulability on multi-core systems. We proceed to review fault-aware timing analysis methods.

1) Fault-tolerant mixed-criticality scheduling: The FTMC scheduling algorithm for single-core systems proposed by Pathan [39] includes a fault tolerance perspective by using execution of backup tasks if faults are detected. Each task (both original and backup) has different WCETs on the different criticality levels (in this case LO and HI). The frequency of faults during a fixed time period is also considered for LO and HI criticality modes. The system will switch from LO to HI mode when either a task exceeds its LO-criticality WCET or the number of errors exceeds its threshold. Once the system has switched to HI-criticality mode all LO-criticality tasks are dropped. If any task errors are detected, a backup is executed and this could be re-execution of the original task (to handle transient hardware-faults) or execution of a diverse implementation (to handle potential software bugs). The error detection mechanism is assumed to be present in the platform already.

A similar method is proposed by Huang et al. [40] for handling transient hardware-faults, but instead of dropping less critical tasks when switching to HI-criticality mode the service can be degraded (e.g., by changing the inter-arrival time of these tasks). Safety requirements are introduced for each criticality level based on the probability-of-failure-per-hour metric (such as those in Table II) and a re-execution profile is defined for each task to ensure it meets the safety requirement. For the HI-criticality tasks a killing profile is defined specifying the number of re-executions that are allowed before LO-criticality tasks are dropped. The impact of task re-execution, task dropping and service degradation are thus bounded.

The above methods present scheduling algorithms that consider fault tolerance and WCET in a single-core setting. Kang et al. [41] apply the concepts of mixed-criticality and reliability to a multi-processor system-on-chip (MPSoC) using the standard model with two criticality levels (normal and critical state). Tasks are mapped to processing elements (PEs) and then locally scheduled. Re-execution and replication is used for reliability and droppable tasks are dropped when switching to the critical state. The dropped tasks are allowed to execute again at the next hyper-period once the system is switched back to normal mode. So, in this case the critical state is just a temporary state to cope with the faults without risking non-droppable tasks missing their deadline. The focus here is on bounding the worst-case response time (WCRT) by using static mapping and optimizations.

By integrating fault tolerance in the methods above, the transition to the critical state is performed for both detected faults and deadline misses even though different handling of the two may be more suitable. To overcome this and to improve the quality of service for the LO-criticality tasks, a four-mode model is proposed by Al-bayati et al. [42]. In addition to the standard two modes (LO, HI) the *transient fault* (TF) and the *overrun* (OV) modes are introduced. The TF mode will be transitioned to when a transient fault is detected, and task re-execution is needed and the OV is transitioned to when a task misses its deadline. The HI mode is used to cover the cases where both overrun and a transient fault are detected. LO-criticality tasks are dropped when the system enters OV or TF. Similar to Huang et al. [40] not all LO-criticality tasks are dropped, but they try to maximize the number of LO-criticality tasks that can run in each mode without affecting

the schedulability of the HI-criticality tasks.

Pathan also presents a global scheduling algorithm (FTM) for real-time sporadic tasks on multi-core platforms [43], but with no focus on mixed-criticality (this is left as future work). The algorithm considers a combination of active and passive redundancy, where active backups are executed in parallel on a different core from the primary task and passive backups are executed in sequence. Both permanent and transient hardware faults are considered. The application-level model considers errors to be detectable in both the primary task and in the backups. The stochastic behaviour of the actual fault model is inserted later, which results in a probabilistic analysis capable of assessing an application's ability to tolerate faults during run-time. A heuristic to help the designers configure the number of active backups is also presented.

2) Estimation of WCET: WCET estimation is a difficult task and most of the WCET analysis methods assume a fault-free execution, but this may no longer be an acceptable assumption as mentioned previously. If we continue to assume that components are fault-free while the probability of failures increases for circuits (e.g., due to miniaturization), there is a risk of underestimating the worst-case execution time of the components, for example when a fault leads to a cache miss where a cache hit was expected. Considering the impact of cache faults is particularly difficult, which is why we have focused on selecting works that are representative of this area.

With the shrinking sizes of components, the number of permanent faults increases and caches taking up a large area in the processor will be a non-negligible source of performance degradation. Slijepcevic et al. [44] present a measurement-based probabilistic timing analysis (MBPTA) approach for faulty caches. The method requires a hardware platform that is compliant with a probabilistic timing analysis (e.g., caches with random (re)placement). It also requires a mechanism to disable a cache line once a permanent fault is detected in that line. Execution times are measured on hardware with maximum degradation. Therefore, they introduce Degraded Test Mode (DTM) that allows a number of cache lines to be disabled. Using DTM it is possible to get measurements for a faulty cache on a fault-free processor so that probabilistic WCET (pWCET) estimates are safe. Slijepcevic et al. [45] continue by also handling transient faults and the timing impact of error detection, correction, diagnosis and reconfiguration. A lot of requirements are placed on the hardware, but they claim the result is a tight pWCET.

Chen et al. [46] perform static probabilistic timing analysis (SPTA) on instruction caches with random replacement on single-cores considering both permanent and transient hardware faults. Memory traces for single-path programs are used as input and the probability of exceeding a certain execution time is computed using state space techniques based on a non-homogeneous Markov chain model. The result of the Markov model is a timing analysis taking all cache states into account. However, the number of states grows polynomially with high exponent values as more and more memory addresses are requested. To overcome this they introduce a method that limits the number of states by using a lower number of memory addresses for the states.

The method used by Hardy and Puaut [47] is quite similar to what Slijepcevic et al. [44] present, but not as many requirements are posed on the hardware. They present an SPTA-based approach for instruction caches with least recently used (LRU) replacement. Their work is based on a low-level static analysis of the cache using abstract interpretation and a high-level WCET analysis using an integer linear programming (ILP) technique (Implicit Path Enumeration Technique (IPET)). Cache sets are independent, which means it is not necessary to explore all faulty cache configurations to obtain the penalty probability distribution, but rather compute the convolution of the set's probability distributions. They compute fault-free WCET and then derive an upper bound of the time penalties caused by fault-induced misses. The evaluation performed shows that for a given probability of a static random-access memory (SRAM) cell failure the pWCET estimates are significantly larger than the fault-free WCETs.

Hardy et al. [48] introduce two hardware based mitigation mechanisms, *Reliable Way (RW)* and *Shared Reliable Buffer (SRB)*. RW introduces a permanent fault resilient way for each cache set to capture the spatial locality of memory requests, which otherwise would be missed as all those requests would end up in an entirely faulty cache set. RW ensures that the cache performance is not worse than a direct-mapped cache with a size equal to the number of sets. SRB is also introduced to mitigate the increase in cache misses when a whole set is faulty, but at a lower hardware cost. The price for the lower cost is a performance that may be worse than RW when there are faults. The WCET estimation is adapted to using RW or SRB and their experimental evaluation shows a gain in pWCET of 48 percent and 40 percent for RW and SRB respectively, compared to their previous work [47].

Caches are of course not the only component that can affect WCET, Höfig [49] propose an SPTA approach considering faulty sensors. The Failure-Dependent Timing Analysis (FDTA) is based on tools such as *Enterprise Architect* (EA, a visual modeling and design tool based on OMG UML), *Simulink* (a graphical programming environment for modeling, simulating and analyzing multidomain dynamical systems) and *aiT* (a static timing analysis tool, based on abstract interpretation, for bounding WCET). Simulink is used to model the system and this model is imported into EA, where a safety analysis is performed to generate a fault tree. The failure modes resulting from this analysis are used to generate a new Simulink model for each mode in which a specific fault has been inserted. Code is generated and compiled for each of the fault-injected models and aiT is used to statically estimate the WCET. A probability is calculated for each of the resulting

WCET estimates providing a probabilistic guarantee that the deadline miss ratio of a task is below a given threshold.

To summarize, our review of the works in Sections III-A and III-B above shows that research works that focus on fault tolerance methods combined with multi-core elements have so far focused on using multi-core resources as a means of mitigating faults, and research that focuses on fault-aware timing analysis focuses on isolation of critical tasks or interference analysis *in the absence of faults*. We note that fault models that impact shared (hardware) resources (e.g., caches and DRAM) and thereby introduce additional inter-core interference in a multi-core are not well-studied.

IV. Fault Injection

To verify fault tolerance mechanisms such as those reviewed in Section III, one can use automated fault-injection tools to speed up the otherwise often time-consuming verification and validation. Most fault-injection tools focus on a particular type of fault (hardware or software fault) and on a particular component such as cache or memory, which occupy large parts of the chip area and are thereby extra vulnerable to faults. However, estimating the impact of faults on timing also needs an assumption of the frequency of faults, which is missing from the models adopted by most fault injection techniques.

Since there already exists a recent survey of fault injection methods for single-core platforms, Natella et al. [10], we complement their survey by selecting representative papers that address fault injection on multi-core platforms. We are not only interested in verifying the fault tolerance mechanism, but also timeliness (Question 2).

A. Fault Injection Emulating Hardware Faults

Most modern processors include capabilities for detecting and reporting errors in most processor units. Cinque and Pecchia [50] use this mechanism in Core i7 from Intel to emulate machine check errors (e.g., cache, memory controller and interconnection errors) by writing to registers associated with the error-reporting capability. They target virtualized multi-core systems and support fault injection at both hypervisor and guest-OS-level. This is an easy and lightweight fault injection method, but requires support for writing to these error-reporting registers. In the works below we consider both specific and generic hardware models as a basis of injecting faults.

Wulf et al. [51] present a software-based fault injection tool, SPFI-TILE, which emulates single-bit hardware faults in registers or memory on the many-core TILE64 using a debugger (the gnu debugger, GDB). They also present a data cache fault-injection extension called Smooth Cache Injection Per Skipping (SCIPS), which distributes fault injection probabilities evenly over all cache locations. As the cache is not directly accessible from software, they emulate faulty cache data by using the debugger to halt the selected tile (core) at a fault injection point, step to the next load instruction, inject a fault in the correct memory address, and then continue the execution to let the load instruction finish with the faulty data. SCIPS is used to balance the fault injection probabilities by randomly skipping several load instructions instead of injecting into the first load instruction after the location where the debugger halts the processor.

By using the *fork()* and *ptrace()* system calls and operating system signals (e.g., SIGSTOP and SIGCONT) Vargas et al. [52] develop a fault injection tool that is hardware-independent (but not operating system-independent) and can inject faults into general purpose registers, some selected special purpose registers and in memory regions. The parent process (after the fork() call) is used as the fault injector and the child process executes the application under analysis. This is actually similar to what GDB does under the hood (used by Wulf et al. [51]). Multi-threaded (pthreads or OpenMP) applications are supported as the fault injector queries the operating system for the number of threads in the child process and their IDs.

Software-based fault injection is easy to use and portable, but cannot be used to inject faults into parts that are not accessible from software. A full-system simulator can expose the internal state of the processor, which simplifies fault injection and no modifications to the device under test is required.

In most simulators one can save a checkpoint containing the current state of the simulated system. This feature can be used to inject faults if modification of the saved checkpoint is possible. One can save one or more checkpoints during a fault-free simulation, which is regarded the golden model and can be compared to checkpoints saved after a fault has been injected. Checkpoints can also be used to reduce the amount of re-executed code and speed up the simulation. This checkpointing and golden model method is used by several fault injection frameworks, of which a few ([53]–[56]) are described here. Carlisle et al. [53] use the Simics full-system simulator to develop DrSEUs (Dynamic robust Single-Event Upset simulator), which is used to simulate single event upsets (SEUs) and single event functional interrupts (SEFIs). They use the checkpointing capability of Simics to inject bit-flips into any of the components in the processor (e.g., general-purpose registers, program counter, Ethernet controller registers and translation-lookaside buffer entries). Caches and main memory are not targeted by DrSEUs. OVPSim-FIM is presented by Rosa et al. [54]. The golden checkpoints are created by executing the application on the original OVPSim. Faults (single bit-flips) are injected at a random time, in a random component (registers or memory) and then the simulation continues. Miele [55] adds SystemC/TLM, for modelling the architecture, to the methodology which offers the possibility to monitor system behavior at both the application and the architectural level. The emulator QEMU is used to inject faults

by Höller et al. [56] to analyse software countermeasures against attacks.

Petersén et al. [57] present a simulation-based platform for experimenting with fault injection and fault management, which utilizes an existing IEEE 1687 network for monitoring purposes. To simplify the experimenting, several parts of the platform are implemented using VHDL to model a multi-processor system-on-chip.

B. Fault Injection Emulating Software Bugs

Software fault injection (SFI) is a common technique to validate fault tolerance mechanisms in systems. Natella and Cotroneo [6] investigate whether SFI really does emulate transient software faults (mandelbugs) to a satisfactory degree. They perform a case study and analyse the SFI tool G-SWFIT, finding that the injected faults do not represent mandelbugs that well. This is because the injected faults are activated early in the execution phase and all process replicas are affected in a deterministic way.

Natella et al. [10] discuss several important aspects of SFI, such as how well the injected faults represent real faults, how efficiently the experiments can be performed and how usable the methods and tools are. Such a thorough analysis would also be needed for multi-core systems and would be useful when a sizeable body of works address injecting faults in multi-core systems.

V. Discussions

In our pursuit of answers to Question 1 and 2 we reviewed several state-of-the-art methods for dealing with fault tolerance in presence of hardware faults, by injection of (emulated) hardware faults, and in some cases software faults. In this section, we compare and evaluate these methods with regard to our posed questions.

Table III summarizes the surveyed papers and their respective focuses. Column two clarifies whether the method has a pronounced focus on critical systems as these may be a better starting point. We assume that generic application domains do not have any strict timing requirements (i.e., the main goal is performing the computations correctly). In the third column the main focus of the paper is presented, whether the method is applicable for designing or verifying (validating) a system, namely the space of fault tolerance techniques to choose from at design time, and the available injection techniques for verifying the adopted requirements. The fourth column specifies the type of faults that are considered and in the last column we indicate whether any impact on timing analysis is considered.

The following aspects will be considered in our detailed evaluation and discussion.

Portability to hard real-time systems We look at the facilities used from the underlying hardware and operating system and how (if) they can be ported to a constrained environment (real-time operating systems and design paradigms). As an example, safety-critical systems rarely allow dynamic memory allocation. (Question 3)

Inter-core interference When running mixed-criticality systems the temporal partitioning is fundamental. No function should be able to affect the execution of another. In a multi-core platform with shared resources multiple cores can access the same shared resource in parallel, which could affect the execution time of the applications on the cores. (Question 1 and 2)

WCET impacts Safety-critical systems have to perform the intended function in a timely fashion even under faulty conditions so fault tolerance should consider its impacts to worst-case execution time (WCET) and worst-case response time (WCRT) (or WCET/WCRT estimates should consider faults). (Question 1 and 2)

We start by looking at the execution environment for the methods and whether they depend on specific features of either the hardware or the operating system the applications execute on.

1) Portability to critical applications: Several of the proposed fault tolerance methods are implemented on Linux and rely on its features to alter the running process by preloading (*LD_PRELOAD*) dynamic shared libraries to extend system call functionality and override other shared library functions. Another example is *fork()*, which is used to dynamically create an identical copy of a process [24,26,27]. Process-level redundancy (PLR) as proposed by Shye et al. [23] uses Intel Pin for dynamic code patching on Linux. The methods are easily implemented using the support of the OS, but may be hard to port to a real-time operating system designed for safety-critical systems, where dynamic features are often disallowed. Linux is recognized as not being suitable for RTCA/DO-178C [14] design assurance level (DAL) C and higher in safety-critical avionics systems, where the OS needs to be certified to the same DAL as the most critical application running in the system. The same type of requirement exist also for other domains.

Simulators are often used to demonstrate the fault tolerance methods [30,44]–[46] because the existing commercial platforms do not support the controllability or observability required for the method to work. For some methods, where the required hardware support does not exist, additional hardware modifications are required for the method to become implementable. This makes it very difficult to deploy the methods in a real-world system as it may be hard to get the chip manufacturers to implement the needed features into their processors unless there is a big demand for these features. At least the avionics domain is quite small compared to the consumer market with general-purpose computers, mobile phones and tablets. The methods using additional hardware outside of the system-on-chip (SoC) have bigger potential of

Table III
CLASSIFICATION OF SURVEYED PAPERS

	Paper	Application Domain: Safety Critical	Application Domain: Generic	Fault: Tolerance	Fault: Injection	Hardware Faults: Transient	Hardware Faults: Permanent	Software Faults	Addresses Timing Analysis (WCET)
Multi-Core	Shye [23]		✓	✓		✓			
	Mushtaq [26]		✓	✓		✓			
	Mushtaq [27]		✓	✓		✓			
	Meisner [28]		✓	✓		✓	✓		
	Peshave [29]	✓		✓		✓	✓		
	Hari [30]		✓	✓		✓	✓	✓	
	Kang [41]	✓		✓		✓			✓
	Al-bayati [42]	✓		✓		✓			✓
	Slijepcevic [45]	✓		✓		✓	✓		✓
	Pathan [43]	✓		✓		✓	✓	✓	✓
	Missimer [25]	✓		✓		✓	✓	✓	
	Marshall [24]		✓	✓		✓			
	Kim [21]	✓		✓			✓	✓	✓
	Bhat [22]	✓		✓			✓	✓	✓
	Mushtaq [9][1]		✓	✓		✓	✓	✓	
	Cinque [50]	✓			✓	✓			
	Wulf [51]		✓		✓	✓			
	Vargas [52]		✓		✓	✓			
	Carlisle [53]		✓		✓	✓			
	Petersén [57]		✓		✓	✓	✓		
	Miele [55]		✓		✓	✓			
	Rosa [54]		✓		✓	✓			
Single-Core	Manoochehri [34]		✓	✓		✓			
	Pathan [39]	✓		✓		✓		✓	✓
	Huang [40]	✓		✓		✓			✓
	Höfig [49]	✓		✓		✓	✓		✓
	Slijepcevic [44]	✓		✓			✓		✓
	Chen [46]		✓	✓		✓	✓		
	Hardy [47]	✓		✓			✓		✓
	Hardy [48]	✓		✓			✓		✓
	Yuce [38]		✓	✓		✓			
	Hoffmann [31]	✓		✓		✓			
	Song [32]	✓		✓		✓			✓
	Song [33]	✓		✓		✓			✓
	Natella [6]		✓		✓			✓	
	Natella [10][1]		✓		✓			✓	

[1] Survey

actually making it to the industry. Implementations on field-programmable gate arrays (FPGAs) together with currently available components may be a way forward.

2) Inter-core interference: The most common surveyed methods for detecting and recovering from faults are process redundancy and re-execution [9,21]–[28,41]. Redundancy is particularly relevant when it comes to multi-core platforms, since plenty of resources exist for execution.

However, both redundancy and re-execution can be problematic on multi-cores due to inter-core interference when accessing shared resources. The re-execution will result in additional execution time and memory requests, which will lead to a variability in the response time (for tasks on other cores) as the cores contend for the shared resource. In a distributed multi-processor system, where redundant tasks are run on other nodes, we have no additional overhead and can perform a schedulability analysis [21,22] on each node. If we want to consolidate this system to a multi-core processor and run the redundant tasks on different cores, this will also lead to a variablility of the response time as the different processes access the same shared resources.

Most of the works focus on functional fault tolerance with no regard to the impact they have on the execution (or response) time as a result of this consolidation or re-execution on multi-core processors.

WCET estimation for a task in a multi-core system is increasingly difficult due to the dependence of what is executing on the other cores. There are proposed methods (e.g., [58,59]) that will allow WCET to be estimated for each task in isolation and then determining the maximum interference the task can be subjected to. In the integration phase where all tasks on all cores are brought together the final WCET (WCRT) can be determined and schedulability analysis can be performed. The works considering schedulability [39]–[42] on multi-core all assume that WCET estimates have been derived and are ready to be used. The response times estimated in these works have to be interference-aware as the fault tolerance method used is re-execution of a task, which may introduce additional memory requests that interfere with, or are interfered with, by tasks on other cores resulting in a variability of the

WCRT.

Delays for memory requests are modeled with a constant value in the works considering WCET estimates [44]–[49], which is not accurate considering the inter-core interference and the resulting timing variability for memory requests on a shared-memory multi-core system. Löfwenmark and Nadjm-Tehrani [58] present a methodology for estimating WCET in multi-core systems that includes measurements using a real-time operating system equipped for this purpose. To ensure an upper bound on the interference processes may experience, the memory requests issued from processes are monitored and the process is suspended when (if) the per-process allotted number of requests is exceeded. However, they also assume a fault free execution.

Related to the inter-core interference is the recovery method used for some caches, where the cache line in a write-through cache is automatically invalidated when a fault is detected. This also applies to instruction caches as no modified data exist. This may result in additional memory requests and thus increase the inter-core interference. This has not been considered in any work. Chen et al. [46] identify that faults in the instruction cache result in new memory requests to fetch the instructions again, but as the method is developed for a single-core processor the application to multi-cores is not well-understood.

3) WCET impacts: Triple Modular Redundancy using redundant hardware has no effect on WCET as the three units are executing independently and the voter logic has a well-defined delay, but it is expensive in terms of hardware as complete hardware components have to be tripled. For some systems this may still be the only viable option. On the other hand, using redundant processes on a multi-core does affect the WCET since the processes will execute concurrently and will access the same shared resources as mentioned above. Memory-intensive applications suffer from a larger overhead than CPU-intensive applications due to the higher cache miss rate and the subsequent larger number of memory requests [23].

We have surveyed works ([21,22,32,33]) that do consider the impact the fault tolerance method has on the schedulability and do provide a schedulability analysis. However, they are not targeting multi-core processors and are thus not considering any inter-core interference. Therefore, they cannot be directly applied to multi-core processors.

No fault tolerance method targets all features (e.g., register, cache, interconnect) of a processor. When determining which features to protect, the safety assessment process has to estimate the probabilities of faults and compare them to the probabilistic safety guarantees required by the criticality level (DAL/ASIL/SIL) of the software. The interconnect that is present in many multi-core platforms is often part of the manufacturer's intellectual property and details are seldom disclosed, making it difficult to assess its fault tolerance capabilities.

The use of probabilistic methods [44]–[49] does sound interesting and is worth a more in-depth study in the domain of safety-critical systems. However, Gil et al. [60] identify a number of open challenges for deriving probabilistic WCET (pWCET) using measurement-based probabilistic timing analysis (MBPTA) and the hardware requirement with random caches and buses may also be an obstacle. The avionics industry is currently focused on platforms with PowerPC processors and as far as we know these platforms do not support this. If the gain is big enough a change of processor family may be possible, but the avionics industry is slow-moving with long-lived projects.

Simulators can also be used for fault injection [53]–[57], making it possible to inject faults in basically every component of a processor. Simulations of complex processors can be very slow, offering a best-case simulation performance of 2-3 MIPS (33 injections per second) [54].

4) Validation by fault injection: Much of the research efforts in the software fault injection area are looking at the problem of producing representative software faults, that is, focusing on what to inject, where to inject it, and how to inject it. Software-based fault injection [50,52] affects the execution time and thus it may be hard to verify WCET estimates in presence of faults.

No software fault injection campaigns aiming at quantifying impacts on multi-core systems were identified, which is a bit surprising. Additional software faults when single-core-tested software is migrated to multi-core can arise since new race conditions or lack of locks can show up in interference scenarios. For instance, in a single core priority-based preemptive system, the software developer can often assume that a high-priority task and a low-priority task will not access the memory simultaneously, since the high-priority task will preempt the low-priority task or the low-priority task priority is temporarily raised using ceiling mechanisms. This may lead to applications failing to use a lock to properly synchronize access to the memory. In a multi-core system both of these tasks can run in parallel, resulting in simultaneous access to the memory with unpredictable consequences. Other problems may exist due to synchronization mechanisms that work well on a single core system, but lead to problems that surface only in a multi-core system.

Software faults that do not manifest themselves in every execution (earlier referred to as mandelbugs), where the effect is transient in nature, are not well represented by existing software fault injection tools [6]. This type of bug is hard to find during design time and pre-production testing. Thus, most of the bugs found in an operational system are classified as mandelbugs.

VI. Conclusion

Faults are becoming more and more common, not only in the very harsh environments such as the cruising altitude of an air-

craft or in space, but also due to miniaturization and software-intensive functionality. The shrinking sizes of transistors make the processors more sensitive to cosmic radiation and voltage variations, resulting in an increased number of transient faults. We reviewed papers that indicate an increase in permanent fault rates, indicating that fault-tolerant system development will be more important than before in many domains. The area should attract increasing attention.

In this survey we presented a representative analysis of the state-of-the-art fault tolerance and fault injection methods with the aim of addressing validated worst-case execution time/response time (WCET/WCRT) estimation within multi-core systems. We posed three questions that we set out to answer, unfortunately none of them have been answered in a satisfactory way. While many of these approaches are promising, several challenges remain.

We identify a lack of research on WCET/WCRT estimations under faulty conditions on safety-critical systems built with commercial off-the-shelf (COTS) multi-core platforms requiring temporal partitioning. This research is urgently required to safely deploy multi-cores in the a safety critical domain, and for certification authorities to accept and approve their usage.

We also identify the lack of research on fault tolerance induced memory requests and the effect on resource-monitoring multi-core COTS systems with deterministic timing.

WCET estimates in presence of permanent faults can be found and seem to be well-researched. Research on how to handle transient faults so far focuses on re-execution to ensure functionally correct tasks rather than investigating the impact that the transient faults have on WCET and schedulability.

None of the fault injection methods consider verification of timeliness, but a cycle-accurate simulator could perhaps be used for WCET estimation in presence of hardware faults provided the hardware model in the simulator is detailed enough. Furthermore, combining outcomes of fault injection experiments with models for analysis of WCET and WCRT in a systematic way, e.g. using formal methods, is a future direction in research.

To summarize, no work combining timing predictability and hardware reliability in presence of inter-core interference on multi-core systems are currently identified. This makes the topic an interesting area for future research. There are more or less isolated islands between work on fault tolerance and timing assurance in the multi-core setting. Also, fault injection platforms aiming at multi-core fault tolerance need more active research.

ACKNOWLEDGEMENT

This work was supported by the Swedish Armed Forces, the Swedish Defence Materiel Administration and the Swedish Governmental Agency for Innovation Systems under grant number NFFP6-2013-01203 and NFFP6-2014-00917.

REFERENCES

[1] A. Löfwenmark and S. Nadjm-Tehrani, "Challenges in Future Avionic Systems on Multi-Core Platforms," in *Proc. of 25th IEEE International Symposium on Software Reliability Engineering Workshops*, Nov 2014, pp. 115–119.

[2] G. Macher, A. Höller, E. Armengaud, and C. Kreiner, "Automotive embedded software: Migration challenges to multi-core computing platforms," in *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, July 2015, pp. 1386–1393.

[3] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan 2004.

[4] D. K. Pradhan, Ed., *Fault-tolerant Computer System Design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[5] Y. Huang, P. Jalote, and C. Kintala, "Two techniques for transient software error recovery," in *Hardware and Software Architectures for Fault Tolerance: Experiences and Perspectives*, M. Banâtre and P. A. Lee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 159–170. [Online]. Available: http://dx.doi.org/10.1007/BFb0020031

[6] R. Natella and D. Cotroneo, "Emulation of Transient Software Faults for Dependability Assessment: A Case Study," in *Dependable Computing Conference (EDCC), 2010 European*, April 2010, pp. 23–32.

[7] J. Gray, "Why Do Computers Stop and What Can Be Done About It?" Tandem Computers, Tech. Rep. 85.7, 1985.

[8] J. Hammarberg and S. Nadjm-Tehrani, "Formal verification of fault tolerance in safety-critical reconfigurable modules," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 268–279, Jun 2005. [Online]. Available: https://doi.org/10.1007/s10009-004-0152-y

[9] H. Mushtaq, Z. Al-Ars, and K. Bertels, "Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems," in *Design and Test Workshop (IDT), 2011 IEEE 6th International*, Dec 2011, pp. 12–17.

[10] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing Dependability with Software Fault Injection: A Survey," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 44:1–44:55, Feb. 2016. [Online]. Available: http://doi.acm.org.e.bibl.liu.se/10.1145/2841425

[11] Y. Cheng, W. Chen, Z. Wang, and Y. Xiang, "Precise contention-aware performance prediction on virtualized multicore system," *Journal of Systems Architecture - Embedded Systems Design*, vol. 72, pp. 42–50, 2017. [Online]. Available: https://doi.org/10.1016/j.sysarc.2016.06.006

[12] M. García-Valls, A. Casimiro, and H. P. Reiser, "A few open problems and solutions for software technologies for dependable distributed systems," *Journal of Systems Architecture - Embedded Systems Design*, vol. 73, pp. 1–5, 2017. [Online]. Available: https://doi.org/10.1016/j.sysarc.2017.01.007

[13] Wikipedia Foundation, "Automotive Safety Integrity Level," https://en.wikipedia.org/wiki/Automotive_Safety_Integrity_Level, Last accessed 16 March 2018.

[14] RTCA, Inc, "RTCA/DO-178C, Software Considerations in Airborne Systems and Equipment Certification," 2012.

[15] Certification Authorities Software Team, "CAST 32A Multi-core Processors," https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32A.pdf, 2016.

[16] RTCA, Inc, "RTCA/DO-297, Integrated Modular Avionics (IMA) Development, Guidance and Certification Considerations," 2005.

[17] AUTOSAR. [Online]. Available: http://www.autosar.org

[18] S. Vestal, "Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, Dec 2007, pp. 239–243.

[19] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim, "PROARTIS: Probabilistically Analyzable Real-Time Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, pp. 94:1–94:26, May 2013. [Online]. Available: http://doi.acm.org.e.bibl.liu.se/10.1145/2465787.2465796

[20] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. J. Cazorla, "Measurement-Based Probabilistic Timing Analysis for Multi-path Programs," in *2012 24th Euromicro Conference on Real-Time Systems*, July 2012, pp. 91–101.

[21] J. Kim, G. Bhatia, R. Rajkumar, and M. Jochim, "Safer: System-level architecture for failure evasion in real-time applications," in *2012 IEEE 33rd Real-Time Systems Symposium*, Dec 2012, pp. 227–236.

[22] A. Bhat, S. Samii, and R. Rajkumar, "Practical task allocation for software fault-tolerance and its implementation in embedded automotive systems," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2017, pp. 87–98.

[23] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, "PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 135–148, April 2009.

[24] J. Marshall, G. Bloom, G. Parmer, and R. Simha, "n-modular redundant real-time middleware: Design and implementation," in *Proceedings of the Embedded Operating Systems Workshop co-located with the Embedded Systems Week (ESWEEK 2016), Pittsburgh PA, USA, October 6, 2016.*, 2016. [Online]. Available: http://ceur-ws.org/Vol-1697/EWiLi16_15.pdf

[25] E. Messier, R. West, and Y. Li, "Distributed real-time fault tolerance on a virtualized multi-core system," in *Proc. of 10th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, July 2014, pp. 17–22.

[26] H. Mushtaq, Z. Al-Ars, and K. Bertels, "A user-level library for fault tolerance on shared memory multicore systems," in *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2012 IEEE 15th International Symposium on*, April 2012, pp. 266–269.

[27] ——, "Fault tolerance on multicore processors using deterministic multithreading," in *2013 8th IEEE Design and Test Symposium*, Dec 2013, pp. 1–6.

[28] S. Meisner and M. Platzner, "Thread Shadowing: Using Dynamic Redundancy on Hybrid Multi-cores for Error Detection," in *Reconfigurable Computing: Architectures, Tools, and Applications: 10th International Symposium, ARC 2014, Vilamoura, Portugal, April 14-16, 2014. Proceedings*, D. Goehringer, M. D. Santambrogio, J. M. P. Cardoso, and K. Bertels, Eds. Cham: Springer International Publishing, 2014, pp. 283–290. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-05960-0_30

[29] M. Peshave, F. B. Bastani, and I. L. Yen, "High-Assurance Reconfigurable Multicore Processor Based Systems," in *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, Nov 2011, pp. 220–226.

[30] S. K. S. Hari, M. L. Li, P. Ramachandran, B. Choi, and S. V. Adve, "mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009, pp. 122–132.

[31] M. Hoffmann, F. Lukas, C. Dietrich, and D. Lohmann, "dosek: the design and implementation of a dependability-oriented static embedded kernel," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2015, pp. 259–270.

[32] J. Song, J. Wittrock, and G. Parmer, "Predictable, efficient system-level fault tolerance in c^3," in *2013 IEEE 34th Real-Time Systems Symposium*, Dec 2013, pp. 21–32.

[33] J. Song and G. Parmer, "C'mon: a predictable monitoring infrastructure for system-level latent fault detection and recovery," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2015, pp. 247–258.

[34] M. Manoochehri and M. Dubois, "Accurate Model for Application Failure Due to Transient Faults in Caches," *IEEE Transactions on Computers*, vol. 65, no. 8, pp. 2397–2410, Aug 2016.

[35] Z. Peng, Y. Lu, A. Miller, C. Johnson, and T. Zhao, "Risk assessment of railway transportation systems using timed fault trees," *Quality and Reliability Engineering International*, vol. 32, no. 1, pp. 181–194, 2016. [Online]. Available: http://dx.doi.org/10.1002/qre.1738

[36] X. Chen and S. Sankaranarayanan, "Model predictive real-time monitoring of linear systems," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, vol. 00, Dec 2017, pp. 297–306. [Online]. Available: doi.ieeecomputersociety.org/10.1109/RTSS.2017.00035

[37] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2017, pp. 97–108.

[38] B. Yuce, N. F. Ghalaty, C. Deshpande, C. Patrick, L. Nazhandali, and P. Schaumont, "FAME: Fault-attack Aware Microprocessor Extensions for Hardware Fault Detection and Software Fault Response," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, ser. HASP 2016. New York, NY, USA: ACM, 2016, pp. 8:1–8:8. [Online]. Available: http://doi.acm.org.e.bibl.liu.se/10.1145/2948618.2948626

[39] R. M. Pathan, "Fault-tolerant and real-time scheduling for mixed-criticality systems," *Real-Time Systems*, vol. 50, no. 4, pp. 509–547, 2014. [Online]. Available: http://dx.doi.org/10.1007/s11241-014-9202-z

[40] P. Huang, H. Yang, and L. Thiele, "On the Scheduling of Fault-Tolerant Mixed-Criticality Systems," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 131:1–131:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593169

[41] S.-h. Kang, H. Yang, S. Kim, I. Bacivarov, S. Ha, and L. Thiele, "Static Mapping of Mixed-Critical Applications for Fault-Tolerant MPSoCs," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 31:1–31:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593221

[42] Z. Al-bayati, J. Caplan, B. H. Meyer, and H. Zeng, "A four-mode model for efficient fault-tolerant mixed-criticality systems," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 97–102.

[43] R. M. Pathan, "Real-time scheduling algorithm for safety-critical systems on faulty multicore environments," *Real-Time Systems*, vol. 53, no. 1, pp. 45–81, 2017. [Online]. Available: http://dx.doi.org/10.1007/s11241-016-9258-z

[44] M. Slijepcevic, L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla, "DTM: Degraded Test Mode for Fault-Aware Probabilistic Timing Analysis," in *2013 25th Euromicro Conference on Real-Time Systems*, July 2013, pp. 237–248.

[45] ——, "Timing Verification of Fault-Tolerant Chips for Safety-Critical Applications in Harsh Environments," *IEEE Micro*, vol. 34, no. 6, pp. 8–19, Nov 2014.

[46] C. Chen, L. Santinelli, J. Hugues, and G. Beltrame, "Static probabilistic timing analysis in presence of faults," in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, May 2016, pp. 1–10.

[47] D. Hardy and I. Puaut, "Static probabilistic worst case execution time estimation for architectures with faulty instruction caches," *Real-Time Systems*, vol. 51, no. 2, pp. 128–152, 2015. [Online]. Available: http://dx.doi.org/10.1007/s11241-014-9212-x

[48] D. Hardy, I. Puaut, and Y. Sazeides, "Probabilistic WCET estimation in presence of hardware for mitigating the impact of permanent faults," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 91–96.

[49] K. Höfig, "Failure-Dependent Timing Analysis - A New Methodology for Probabilistic Worst-Case Execution Time Analysis," in *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance: 16th International GI/ITG Conference, MMB & DFT 2012, Kaiserslautern, Germany, March 19-21, 2012. Proceedings*, J. B. Schmitt, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 61–75. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28540-0_5

[50] M. Cinque and A. Pecchia, "On the injection of hardware faults in virtualized multicore systems," *Journal of Parallel and Distributed Computing*, vol. 106, pp. 50 – 61, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731517300849

[51] N. Wulf, G. Cieslewski, A. Gordon-Ross, and A. D. George, "SCIPS: An emulation methodology for fault injection in processor caches," in *Aerospace Conference, 2011 IEEE*, March 2011, pp. 1–9.

[52] V. Vargas, P. Ramos, R. Velazco, J. F. Mehaut, and N. E. Zergainoh, "Evaluating SEU fault-injection on parallel applications implemented on multicore processors," in *Circuits Systems (LASCAS), 2015 IEEE 6th Latin American Symposium on*, Feb 2015, pp. 1–4.

[53] E. Carlisle, N. Wulf, J. MacKinnon, and A. George, "DrSEUs: A dynamic robust single-event upset simulator," in *2016 IEEE Aerospace Conference*, March 2016, pp. 1–11.

[54] F. Rosa, F. Kastensmidt, R. Reis, and L. Ost, "A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability," in *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, Oct 2015, pp. 211–214.

[55] A. Miele, "A fault-injection methodology for the system-level dependability analysis of multiprocessor embedded systems," *Microprocess. Microsyst.*, vol. 38, no. 6, pp. 567–580, Aug. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2014.05.008

[56] A. Höller, A. Krieg, T. Rauter, J. Iber, and C. Kreiner, "Qemu-based fault injection for a system-level analysis of software countermeasures against fault attacks," in *2015 Euromicro Conference on Digital System Design*, Aug 2015, pp. 530–533.

[57] K. Petersén, D. Nikolov, U. Ingelsson, G. Carlsson, F. G. Zadegan, and E. Larsson, "Fault injection and fault handling: An mpsoc demonstrator using ieee p1687," in *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*, July 2014, pp. 170–175.

[58] A. Löfwenmark and S. Nadjm-Tehrani, "Understanding Shared Memory Bank Access Interference in Multi-Core Avionics," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, ser. OpenAccess Series in Informatics (OASIcs), M. Schoeberl, Ed., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 1–11.

[59] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, "WCET(m) Estimation in Multi-core Systems Using Single Core Equivalence," in *Proc. of 27th Euromicro Conference on Real-Time Systems*, July 2015, pp. 174–183.

[60] S. J. Gil, I. Bate, G. Lima, L. Santinelli, A. Gogonel, and L. Cucu-Grosjean, "Open challenges for probabilistic measurement-based worst-case execution time," *IEEE Embedded Systems Letters*, vol. 9, no. 3, pp. 69–72, Sept 2017.