DOT/FAA/TC-16/51

Federal Aviation Administration
William J. Hughes Technical Center
Aviation Research Division
Atlantic City International Airport
New Jersey 08405

Assurance of Multicore Processors in Airborne Systems

July 2017

Final Report

U.S. Department of Transportation
Federal Aviation Administration

NOTICE

Technical Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
DOT/FAA/TC-16/51		

4. Title and Subtitle	5. Report Date
ASSURANCE OF MULTICORE PROCESSORS IN AIRBORNE SYSTEMS	July 2017
	6. Performing Organization Code 220410

7. Author(s)	8. Performing Organization Report No.
Laurence H. Mutuel, Xavier Jean, Vincent Brindejonc, Anthony Roger, Thomas Megel, E. Alepins	

9. Performing Organization Name and Address	10. Work Unit No. (TRAIS)
Thales Avionics, Inc. 2733 S Crystal Drive, suite 1200 Arlington, VA 22202	11. Contract or Grant No.

12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
Federal Aviation Administration William J. Hughes Technical Center Aviation Research Division Atlantic City International Airport, NJ 08405	Final Report, 11/10/14 – 11/08/2016
	14. Sponsoring Agency Code AIR-134

15. Supplementary Notes

The FAA William J. Hughes Technical Center Aviation Research Division Technical Monitor was Srini Mandalapu.

16. Abstract

This report documents the issues related to software assurance applied to multicore processors (MCPs) and the safety implications pertaining to the use of MCPs in flight-critical applications. In recent years, the use of MCPs in avionics has supported the increase in performance and level of integration of safety-critical functions. However, MCPs stretch the current hardware and software assurance processes. As MCPs were not initially designed for aircraft applications, a preemptive investigation of the potential safety concerns is warranted.

The main concern regarding the use of MCPs in the safety-critical aerospace domain is their lack of predictability. Therefore, their use must be integrated into a systems approach wherein the need for determinism is considered for each function implemented on the MCP. This statement justifies the use of a top-down safety method as the primary approach to be applied to all failure conditions, including those specific to MCPs. The outcome is a set of qualitative and quantitative safety requirements, and the allocation of development assurance level (DAL) to each software application according to the identified criticality level.

The proposed approach to interference analysis in the context of safety processes is close to partitioning analyses. It is composed of two complementary analyses: a top-down analysis followed by a bottom-up analysis. The top-down analysis allows for isolating high-level sources of non-determinism affected by the function/task allocation to cores, the software scheduling strategy, and the selection of MCPs based on usage domain (UD). This consideration of UD is used to orient and bound the complementary bottom-up analysis. Finally, the top-down analysis prepares for the determination of mitigation strategies for the sources of non-determinism that remain in the UD. The bottom-up analysis is conventional from a safety standpoint. The key point is that the complexity of MCPs no longer allows for claims of exhaustiveness unless the top-down analysis is performed beforehand to bind its scope. The interference-aware safety analysis consists of three sequential steps: the identification of an interference path; performance of an interference analysis to tag each interference channel as acceptable, unacceptable, unbounded, or faulty; and determination of interference mitigation. The last step is to implement mitigation for interference channels identified as bounded but unacceptable, unbounded, and faulty. The mitigations can be internal or external to the processor. This report details each of the steps of the proposed safety analyses.

The limitations in existing guidance can be mitigated by the implementation of the proposed complementary top-down and bottom-up analyses. This approach is close to that for integrated modular avionics and is, therefore, dedicated to complex computational systems with high integration levels. It is perfectly adapted to MCP concerns.

17. Key Words	18. Distribution Statement
Multicore processors, Software assurance, Safety, Non-determinism, Interference, Failure, Usage domain, Safety method	This document is available to the U.S. public through the National Technical Information Service (NTIS), Springfield, Virginia 22161. This document is also available from the Federal Aviation Administration William J. Hughes Technical Center at actlibrary.tc.faa.gov.

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	121	

ACKNOWLEDGEMENTS

This research has been coordinated with the technical experts and reviewers at Thales Avionics SAS by Cyril Marchand. Didier Regis, Marc Fumey, Helene Misson, and Guy Berthon acted as internal reviewers for this report.

TABLE OF CONTENTS

APPENDICES

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVATIONS AND ACRONYMS

AEH	Airborne electronic hardware
AMP	Asymmetrical multiprocessing
API	Application program interface
ARP	Aerospace Recommended Practices
ASIL	Automotive Safety Integrity Level
BIST	Built-in self-test
CASP	Concurrent Autonomous chip self-test using stored test patterns
CFG	Control flow graph
CM	Certification Memorandum
COTS	Commercial off-the-shelf
CPU	Core processing unit
DAL	Development Assurance Level
DDR	Double data rate
DMA	Direct memory access
DRAM	Dynamic random access memory
EASA	European Aviation Safety Agency
ECC	Error-correcting code
FDAL	Functional development assurance level
FHA	Functional hazard assessment
FMEA	Failure modes and effects analysis
I/O	Input/output
IDAL	Item development assurance level
IMA	Integrated modular avionics
IOMMU	Input/output memory management unit
IP	Intellectual property
LRU	Least recently used
MCP	Multicore processor
MMU	Memory management unit
NoC	Network-on-Chip
OS	Operating system
PCIe	Peripheral Component Interconnect Express
PLRU	Pseudo least recently used
PSSA	Preliminary system safety assessment
QoS	Quality of service
RTCA	Radio Technical Commission for Aeronautics
RTOS	Real-time operating system
SCP	Single-core processor
SEU	Single-event upset
SMP	Symmetrical multiprocessing
SoC	System-on-Chip
SRAM	Static random access memory
SWAT	SoftWare Anomaly Treatment
TDMA	Time division multiple access
UD	Usage domain
WCET	Worst case execution time

EXECUTIVE SUMMARY

This report documents the issues related to software assurance applied to multicore processors (MCPs) and the safety implications pertaining to the use of MCPs in flight-critical applications.

Most safety-critical avionics systems are defined as “hard real-time,” meaning that they must perform their function within predefined deadlines. In some cases, missing a single deadline at system level is considered a failure condition as per Title 14 of the Code of Federal Regulations subpart 1309, for which severity may be classified as catastrophic. The fail-safe principle requires this single hazard to be appropriately mitigated.

On the other hand, the increase in the integration level of safety-critical avionics functions and the need for increased computational performance induce a more widespread use of MCPs. Thanks to these MCPs, avionics computers can host more and more safety-critical functions. The capabilities of MCPs stretch the current assurance processes for both software and hardware. As these processors were not initially designed with aircraft applications in mind, a preemptive investigation of the potential safety concerns is warranted. The main concern related to the use of commercial off-the-shelf MCPs in the aerospace safety-critical domain is their lack of predictability. Therefore, the use of MCPs must be integrated into a systems approach wherein the need for determinism is considered for each function implemented on the MCP. This statement justifies the use of a top-down safety method as the primary approach. In addition to classical safety analysis on mono-core processors, special attention is required when MCPs are used, as some of the additional features of the MCPs could cause specific failure modes.

In the safe design process, top-down safety analysis techniques—such as functional hazard assessment and preliminary system safety assessment—are applied to all failure conditions, including those specific to MCPs. The outcome is a set of qualitative and quantitative safety requirements and the allocation of development assurance levels (DAL) to each software application according to the identified criticality level. This conventional process supports the justification for implementing different functions on an MCP. Consideration of the DAL refinement process allows for dissociating real-time constraints from safety constrains, so that both aspects should be analyzed in parallel but separately as two independent sets of constraints. In addition, the classification of sources of non-determinism into cache type sources, buffer type sources, and interference sources provides a structure to investigate non-determinism in MCPs.

Interferences are hazards specific to MCPs. From the component manufacturer’s viewpoint, interference is not a dysfunctional behavior but a performance bottleneck; for the avionics manufacturer, however, interferences are considered dysfunctional behavior. Therefore, the failure modes related to interferences and their effects in terms of loss of integrity, loss of availability, or non-deterministic behavior of embedded applications must be identified and may require mitigation by design.

The proposed approach to interference analysis in the context of safety processes is close to partitioning analyses. It is composed of two complementary analyses: a top-down analysis followed by a bottom-up analysis. The top-down analysis allows for isolating high-level sources of non-determinism affected by the function/task allocation to cores, software scheduling

strategy, and selection of MCPs based on usage domain (UD). This consideration of UD is used to orient and bound the complementary bottom-up analysis.

Finally, the top-down analysis prepares for the determination of mitigation strategies for the sources of non-determinism that remain in the UD. The bottom-up analysis is conventional from a safety standpoint. The key point is that the complexity of MCPs no longer allows for claims of exhaustiveness, unless the top-down analysis is performed beforehand to bind its scope.

The interference-aware safety analysis consists of three sequential steps: the identification of an interference path; performance of an interference analysis to tag each interference channel as acceptable, unacceptable, unbounded, or faulty; and determination of interference mitigation. The identification of interference paths is constrained by the presence of black-box components within the processor. For increased confidence in the result of this step, it is recommended to pair the bottom-up identification with a top-down analysis that evaluates which components are used when embedded software triggers operations to shared resources (unused paths do not need to be further covered). For interference paths on which interferences are actually observed (i.e., an interference channel), the interference analysis will identify if the interference channel has a bounded impact compatible or not with the set interference penalty. If the interference penalty cannot be assessed (unbounded channel), the impact of the interference channel on the software execution time is not properly known; thus, mitigation is required. Finally, if the interference channel triggers a failure mode within the processor (faulty channel), a formalized safety analysis must be conducted with the MCP manufacturer. The last step is to implement mitigation for interference channels identified as bounded but unacceptable, unbounded, and faulty. The mitigations can be internal or external to the processor. Internal mitigations are restrictions set on the MCP, while external mitigations deal with monitoring aspects and sanction strategy.

The interference-aware safety process is to be instantiated on a case-by-case basis based on the equipment, the criticality, and the selected MCP and its internal components' level of detail. Acceptability criteria should be proposed as early as possible in the safe design phase and discussed with the MCP manufacturer.

This report details each of the steps of the proposed safety analyses. A generic fault model for MCP-related non-determinism is developed, including the identification of sources for content and timing non-determinism, and used to illustrate the implementation of the approach. The proposed interference-aware safety analysis is detailed, including the definition of sources of interferences both at physical and logical levels, the determination of safety impacts, and their classification. Mitigation means are classified in terms of error detection, recovery and repair schemes for hardware faults, and design errors. For interference, internal mitigations are surveyed to include interference reduction, bounding, or elimination. Worst-case execution time techniques are also specifically called out as an example of the application of the proposed safety analysis process.

The limitations in existing guidance can be mitigated by the implementation of the proposed complementary top-down and bottom-up analyses. This approach is close to the approach developed for integrated modular avionics and is therefore dedicated to complex computational systems with high integration levels. It is perfectly adapted to MCP concerns.

1. INTRODUCTION

1.1 BACKGROUND

For a multitude of domains, safety-related or not, the user demand for significantly increased performance in reinforced size, weight, and power-constrained environments results in an increased share of multicore processors (MCPs) in the current market segment of highly complex semiconductor devices. On the other hand, legacy embedded avionics systems are based on single-core processors (SCPs) and may suffer from an obsolescence of these components.

Aircraft software applications themselves expand in capabilities that require increased computational performance (e.g., advanced signal processing, manipulation of large quantities of stored information) while achieving gains in size and power (e.g., integrated modular avionics [IMA]). MCPs address this need, so that their usage is foreseen to steadily increase. MCPs are currently used in airborne electronic hardware (AEH), although the technology is conservatively targeting MCPs with no more than two cores.

The capabilities of MCPs stretch the current assurance processes for both software and hardware. The supporting design and verification tools may not be adapted either. Because these processors were not initially designed with aircraft applications in mind, a preemptive investigation of the potential safety concerns is warranted. If the MCP technology driven by the overall market shows a faster growth than the aerospace market is able to address assurance concerns, the potential for a longstanding catch-up situation is possible. The investigation therefore needs to be not only preemptive but also predictive.

One specific safety-related concern is associated with the demonstration of deterministic behavior. The need for determinism is requested at aircraft level and depends on the aircraft function considered. These functions may develop several types of dysfunctional behaviors potentially linked to non-determinism issues.

1.2 PURPOSE

This report informs on the issues related to software assurance applied to MCPs and the safety implications of the use of MCPs in flight-critical applications.

Safety issues associated with the use of MCPs in flight-critical applications are associated with non-determinism. The link between dysfunctional behaviors visible to the users and non-determinism is investigated from both a safety and real-time standpoint. This report proposes to implement safety analyses with specific activities to identify MCP failure modes, assess the severity of their associated failure conditions, analyze the architecture for mitigation of these failure conditions, and determine the effectiveness of the mitigations implemented.

Section 2 describes the proposed approach which includes a top-down and a bottom-up analyses on the design. Furthermore, the potential for modulation of the effort is dependent on the distinction between real-time constraints and safety constraints. Lastly, a specific instantiation is

detailed for the proposed safety process applied to the most relevant cause of non-determinism within multicore processors: interferences.

Section 3 gives an overview of two architectures that represent both ends of the spectrum in avionics. Examples from both federated and integrated avionics architectures are used throughout this report.

Section 4 summarizes the existence guidance on MCPs. The identified gaps are addressed in the recommendation section.

Section 5 represents the first step of the safe design process. It discusses MCP failure modes, their description, and their allocation to the MCP architectural features. Specific attention is provided to interferences as the primary cause of non-determinism in MCP.

Section 6 represents the second step of the safe design process. It discusses the safety impacts of non-determinism and proposes a classification of these impacts for the investigation of interferences.

Section 7 represents the third and last step of the safe design process. It discusses the methods and techniques used to mitigate the impacts of MCP failures.

Section 8 collects specific examples where the proposed methods can be implemented. In particular, the performance of an interference analysis is explained in greater details. Worst-case execution time techniques are also detailed as they are commonly used to show mitigation of timing issues.

Section 9 collects recommendations extracted from the findings in sections 2–8. The proposed safety approach is justified from the gaps in existing guidance and the limitation of conventional assurance approaches. Activities supporting the performance of the recommended approach are defined and allocated to the MCP stakeholders. Finally, criteria are defined for each step of the proposed approach so that the achievement of determinism can be demonstrated.

Section 10 contains the references used in this report.

Appendix A is the glossary of terms, while Appendix B provides additional details on worst-case execution time techniques. Appendix C details activities and recommended justification data to be produced for commercial off-the-shelf (COTS) MCP assurance.

2. APPROACH

Most safety-critical avionics systems are defined as hard real-time, that is, they must perform their function within predefined deadlines. In some cases, missing a single deadline at system level is considered a failure condition (see Title 14 Code of Federal Regulations Part 2X.1309) that may be classified as catastrophic. The fail-safe principle requires this single hazard to be appropriately mitigated.

The increase in the integration level of safety-critical avionics functions induces a more widespread use of MCPs. As a consequence, avionics computers can host more and more safety-critical functions and are subjected to real-time constraints according to the criticality of these embedded functions.

The main concern related to the use of COTS MCPs, in the safety-critical domain, is their lack of predictability. The use of MCPs must be integrated in a systems approach wherein the need for determinism is considered for each function implemented on the MCP. This statement justifies the use of a top-down safety method as the primary approach. In addition to classical safety analysis on single core processors, special attention is required when MCPs are used, as some of the MCP additional features could cause specific failure modes.

2.1 GENERIC SAFETY PROCESS

The avionics safety process is comprised of a descending branch, called allocation, and an ascending branch, called integration [1], as shown in figure 1.

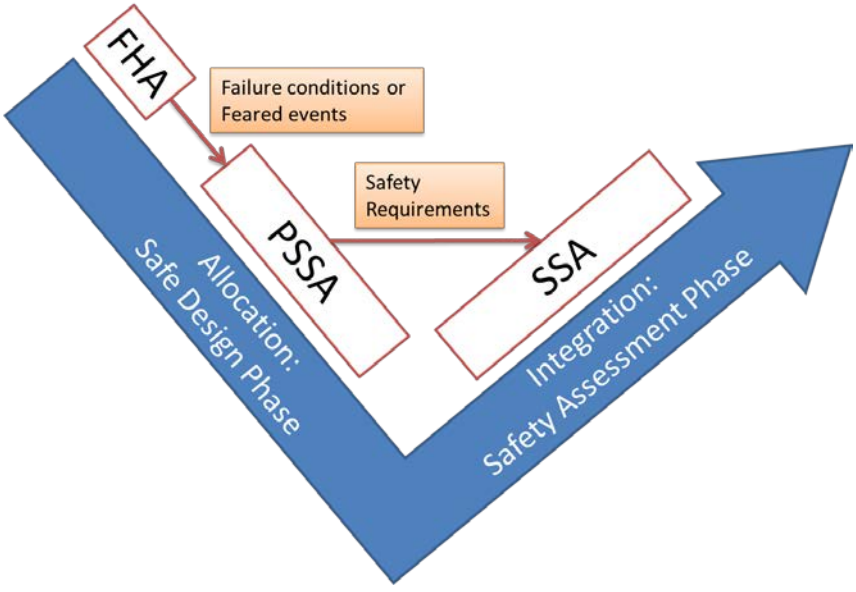

Figure 1. Overview of generic safety process

2.1.1 Top-down Methods for Safe Design Process

"Allocation" pertains to safe design and is performed using top-down analysis methods to design a product complying with the safety constraints. Such safety analysis techniques include the functional hazard assessment (FHA) and the preliminary system safety assessment (PSSA). The FHA identifies and classifies the failure conditions associated with the basic functions of the aircraft according to their severity. The PSSA defines the mitigation, by architecture, of the failure conditions, including the assignment to each function of a development assurance level (DAL) according to the severity of the failure condition.

These methods need to be applied to all hazards, including those specific to MCPs. The outcome is a set of qualitative safety requirements (e.g., defining the safety mechanisms to be implemented) and quantitative safety requirements (e.g., prescribed probability of failure, performance criteria for safety mechanisms).

According to Aerospace Recommended Practices (ARP) document 4754A [1], DALs are allocated to hardware and software items. Considering software applications, an item development assurance level (IDAL) is applied to each software application according to the identified criticality level. Several functions with different DALs can then be implemented on an MCP that can execute several software applications at the same time. The implementation must be such that a lower-level DAL application does not jeopardize an application developed with a higher DAL in order to guarantee the required level of safety in terms of availability and integrity.

2.1.2 Bottom-up Methods for Safety Verification Process

"Integration" pertains to the safety assessment of the design and is performed using methods to verify that the safety objectives set in the safe design phase have been met. The system safety assessment compiles proofs that the safety requirements of the previous phase were correctly implemented in the product. Only a few examples of failure modes and effects analysis (FMEA) are available in the literature that can specifically be used to assess the safety of MCPs.

The implementation of any appropriate mitigation must be planned, developed, documented, and verified to ensure its efficiency.

2.2 DISSOCIATING REAL-TIME CONSTRAINTS FROM SAFETY CONSTRAINTS

Real-time and safety constraints may appear to be related at the lowest breakdown level: a dysfunctional behavior can generate time non-determinism. At the higher levels, the correlation is less straightforward. Consider the example of a given aircraft function for which an associated failure condition is catastrophic and for which the real-time constraints are stringent (e.g., 10 milliseconds for flight control). There exist other functions on the aircraft with less stringent real-time constraints (e.g., over 100 milliseconds for fuel management) that are also associated with a catastrophic failure condition. On the other hand, functions involved with lower failure conditions may be required to meet very stringent deadlines (e.g., the communication function).

A second argument for such a position lies in the functional development assurance level (FDAL) refinement used in the ARP-4754A [1]. At aircraft level, the FDAL is directly (one-to-one) related to a failure condition. After refinement, for example from breaking down FDAL A into [FDAL A + FDAL C] or from breaking down FDAL A into [FDAL B + FDAL B], the resulting FDALs are no longer directly related to a failure condition but rather to a given design effort. In such a process, time constraints may not have been reduced; however, in the example refinement scheme [FDAL A → FDAL B + FDAL B], the same time constraint associated with the initial FDAL A may have been assigned to the refined level FDAL B. DAL B or DAL C functions can therefore inherit hard real-time constraints from an initial DAL A application. A functional design assurance level associated with the function development should thus not be assimilated to real-time constraints.

Safety and real-time aspects should be analyzed separately. It is recommended to address safety and real-time constraints in parallel but as two independent sets of constraints. The safety aspect corresponds to criticality levels that can be characterized by DALs, while real-time properties define the cycle duration (short versus long) or capacity to comply with deadlines (hard versus soft real-time). On IMA, cycle duration corresponds to partition period while process type (periodic or aperiodic) defines the system behavior (soft versus hard real-time).

Figure 2 sketches criticality versus real-time for existing avionic functions in a way that illustrates the non-equivalence between the safety criteria (e.g., DAL, dissimilarity constraints, fault-tolerance) and the real-time criteria (e.g., timing constraints in milliseconds, variability on latency). Eight avionics functions are considered with their corresponding criticality and typical real-time levels. The straight blue line indicates a correlation between criticality and real-time levels: The more critical the function, the tighter the real-time constraints become. For example, flight control is more critical than doors function; the real-time constraint for flight control is tighter (10–20 milliseconds) than for the aircraft doors function (50–100 milliseconds). This situation, however, is not a generality: Landing gear and fuel control have the same level of criticality but not the same timing constraints. This distinction is important toward facilitating the identification of non-determinism criteria and prioritizing their contribution.

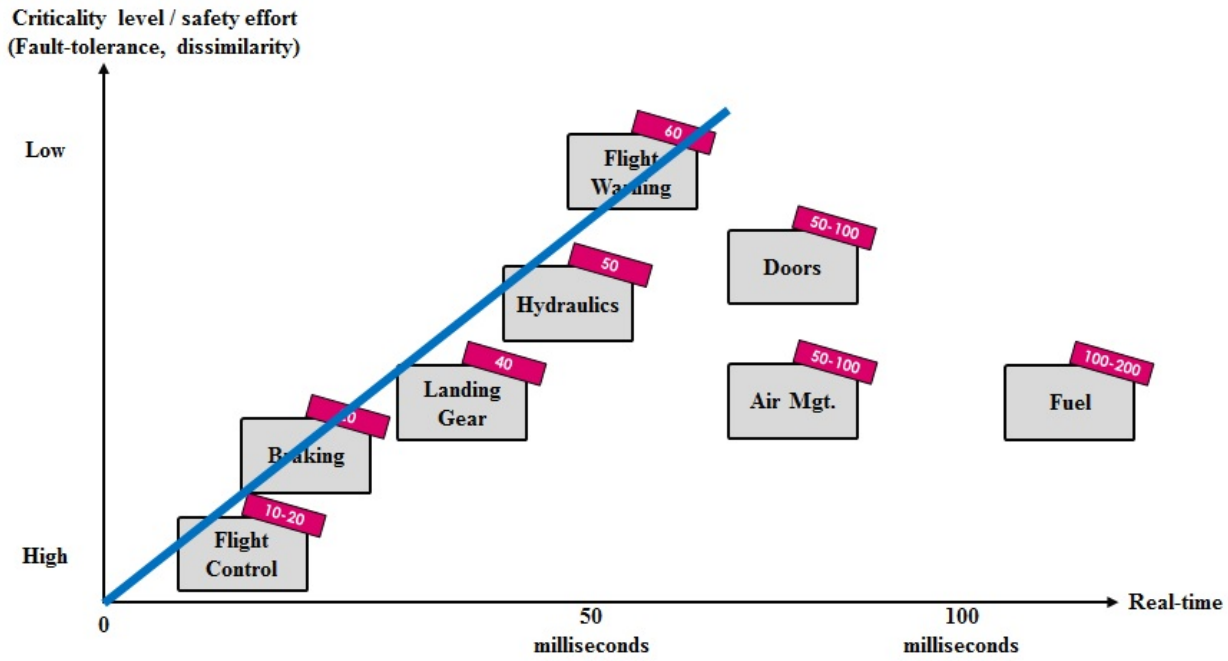

Figure 2. Correspondence between criticality level and real-time constraint

The objective of this section was to illustrate that the interplay between safety and real-time aspects is complex. The analysis should therefore first consider both aspects independently and then address their relationships and interdependencies. Real-time performance is of crucial interest to comply with time determinism. In a statistically optimized MCP, the effort to demonstrate hard real-time performance can be an important task—and it could be preferable to adapt the level of analysis to the real-time constraint level requested by the embedded applications.

2.3 INTERFERENCE-AWARE SAFETY PROCESS

The contribution of interferences to feared events or hazards can embody several aspects. Interferences between applications executing simultaneously on separate cores of an MCP may contribute to these feared events. The problem does not come from the interferences themselves but rather from their consequences.

The failure modes related to interferences and their effects in terms of loss of integrity, loss of availability, or non-deterministic behavior of hosted applications must be identified. Depending on these defects and the safety objectives, mitigations may be required. The coverage of these mechanisms and their sanctions—corrective actions taken at processor- or system-level—are then analyzed and validated by safety analysis for compliance with the system's behavior.

First, interferences are a bottleneck with respect to the performance assessment on a piece of equipment. Intuitively, software running on one core is slowed down by software running concurrently on other cores because of interferences. This is called an interference penalty. The challenge is related to the difficulty in foreseeing and managing inter-core interferences

stemming from the sharing of common resources (e.g., memory, system-level caches, interconnect, and input/output [I/O]) among multiple cores. Various applications are executed on different cores within the same time slot, which means that they must compete to access shared hardware resources. Accesses are managed by hardware arbitration mechanisms that allow one of the tasks to access the resource while delaying the others, resulting in a contention as shown in figure 3 (excerpted from [2]). From the delayed tasks' point of view, these additional delays, introduced because of the other concurrent tasks' unpredictable behavior, are interferences that break the time isolation principle required by avionics standards. From the component manufacturer's standpoint, this behavior is not dysfunctional but rather a performance bottleneck. For the avionics equipment provider, however, the behavior is considered dysfunctional.

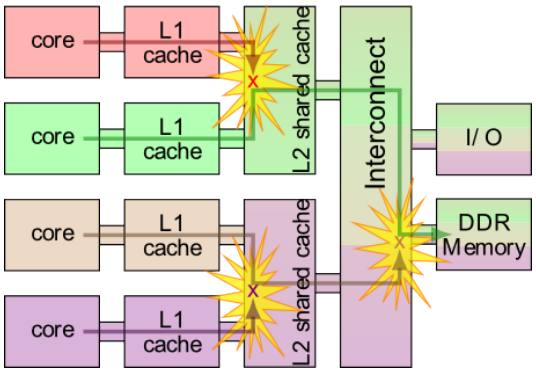

Figure 3. Interferences due to concurrent accesses on shared resources

Second, the situations in which interferences occur are numerous. Quantifying an interference penalty from exhaustive testing is impossible in many cases as the number of test scenarios grows exponentially. Therefore, a major challenge is to estimate an interference penalty from non-exhaustive test campaigns and to provide a defendable rationale that this estimated penalty is trustworthy. A chip manufacturer faces the same limits of exhaustiveness in testing as an equipment provider. Therefore, one or more configurations might trigger failure modes on the processor that were not covered by the manufacturer's tests. That case is no longer a matter of performance, but it impacts the system's integrity.

MCPs have dedicated hardware for spatial partitioning [3], including a memory management unit (MMU) and input/output memory management unit (IOMMU), so that enforcing spatial isolation today is considered a resolved issue. That is not the case for the timing aspects. To enable the usage of MCPs on safety critical systems, interferences need to be controlled and techniques need to be developed to exploit multicore performance benefits. To achieve this objective, new concepts have to be introduced, potentially relying on hardware specificities, as long as COTS processor manufacturers support them. That may impact processor selection processes.

2.3.1 Approach

This report describes a generic process, represented in figure 4, that covers several analyses performed on the processor (regardless of embedded software) or on the whole platform.

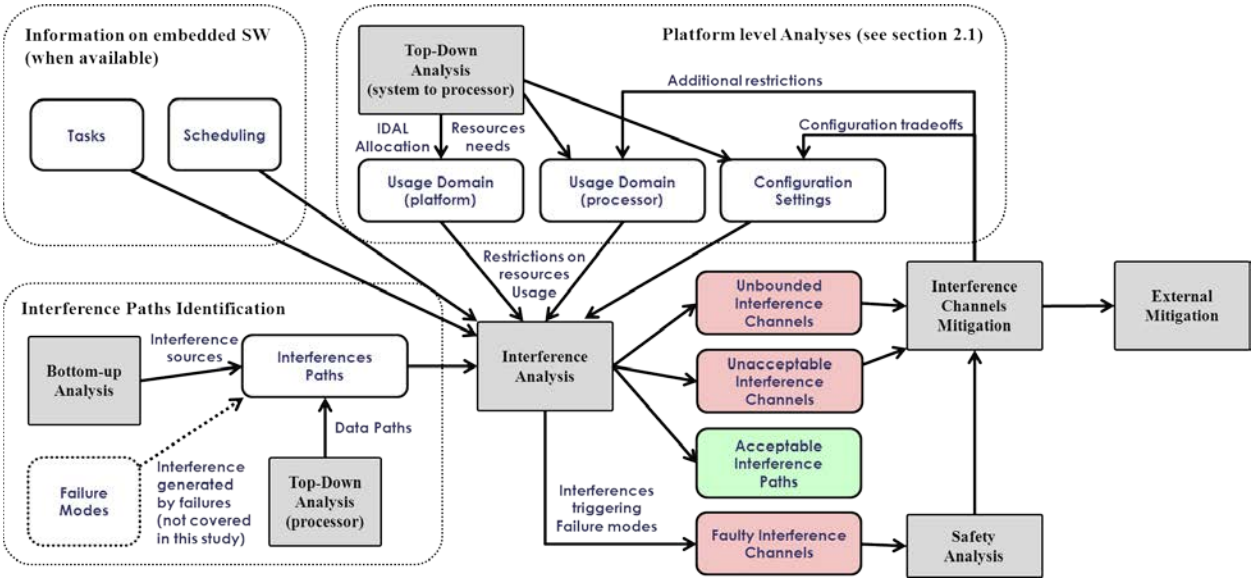

Figure 4. Overview of the proposed interference-aware safety process

The proposed philosophy is close to partitioning analyses, which is described in DO-297 [4]. Its process takes the following inputs:

- A top-down analysis that allocated safety objectives to the MCP in accordance with the top-level safety requirements' criticality and identified a functional failure path. Then, it allocated IDALs to all software items and refined their needs in terms of real-time requirements. Additionally, this analysis allocated software items to execution platforms and defined a usage domain (UD) both for the platform (e.g., scheduling rules, software development restrictions) and the processor (i.e., resources actually being used). All this information allowed restrictions on the use of shared resources within the processor. Additionally, the top-down analysis fixed acceptability conditions for the different stages of this process according to safety criteria.
- For federated systems, embedded software can be known with a high-level of details at an early stage of the equipment's development. That is not the case for integrated systems, such as IMA, whereas UD and representative applications compensate for the lack of knowledge on the final embedded software.

A complementary bottom-up approach driven by non-interference considerations can be deployed once the top-down analysis determines the appropriate level of exhaustiveness for the interference analysis as a function of real-time constraints and assurance level. The bottom-up interference analysis on the MCP allows for identifying sources of interference not linked to microprocessor failures and their effect on the system. Mitigations can then be implemented and justified in accordance with the identified failure effects.

The safety analysis is then completed to determine if the level of exhaustiveness and the specified sanctions are appropriate to ensure compliance with the safety requirements. The tolerance to interferences depends on the criticality of the implemented functions.

Note that sources of interferences linked to processor failure were not considered in details. These interferences are addressed through the performance of existing failure mode and effect analyses, as they are consequences of known failure modes.

2.3.1.1 Summary Benefits of the Top-down Approach

Studies on microprocessors have often been based on bottom-up analyses. These analyses were already more difficult to conduct for the last generation of SCP. The combinatory complexity brought by MCP only further increases the difficulty. Performing a top-down analysis prior to the bottom-up analysis leads to three major benefits:

- It allows for isolating high-level sources of non-determinism that can be affected by:
    - The function/tasks allocation on cores.
    - The selected software and scheduling strategies.
    - The MCP selection based on the coverage of the selected UD by the MCP specification domain.
- It allows for reducing the accessible UD in which sources of non-determinism have to be studied via a bottom-up analysis. As such, it orients and bounds the bottom-up analysis.
- It prepares for possible mitigation strategies if sources of non-determinism still remain in the UD.

2.3.1.2 Summary Benefits of the Bottom-up Approach

A bottom-up approach is conventional and is generally considered as being exhaustive, although the complexity of modern MCPs can no longer guarantee this characteristic without first performing a top-down approach.

Local sources of non-determinism are categorized into:

- Cache-type sources
- Buffer-type sources
- Interference sources

This classification provides a structured approach for investigating non-determinism in MCP.

2.3.2 Steps to be Performed

The proposed interference-aware safety process steps are outlined in sections 2.3.2.1 through 2.3.2.3.

2.3.2.1 Identification of Interference Paths

Thanks to a bottom-up approach, sources of interferences are listed, with a level of details that depends on the presence of black-box components within the processor. Confidence regarding the coverage can be obtained by coupling the result of this bottom-up analysis with a top-down analysis that evaluates which components are used when embedded software triggers operations to shared resources.

The output of this step is a set of interference paths (see figure 7).

2.3.2.2 Interference Analysis

This stage takes, as input, a set of restrictions driven by the UD over the platform, the processor, and its configuration. These restrictions apply at two levels:

- On the overall set of interference paths. Some of these paths may be unused and do not need to be covered.
- On specific interference paths. Specific configurations within these interference paths may be stated as unreachable (with the appropriate mitigation) and thus do not need to be covered for the interference analysis.

When interferences are actually observed on a given interference path, it is called an interference channel. The interference analysis will tag each interference channel as:

- Acceptable: The interference channel has a bounded impact, so that an interference penalty can be assessed. Moreover, this interference penalty copes with the equipment's functional domain.
- Unacceptable: The interference channel has a bounded impact. However, the interferences penalty that was assessed is too high.
- Unbounded: No interference penalty could be assessed on this interference channel. Its impact on software execution time is not properly known. Mitigation is required to meet performance requirements only.
- Faulty: The interference channel has triggered a failure mode within the processor. Further investigation in collaboration with the manufacturer is required and formalized through a safety analysis. That stage may be followed by mitigation.

An interference analysis is, at first, a matter of performance assessment. Assuming that, ideally, the processor has no failure mode triggered by interference channels, some interference channels will have a bounded and acceptable impact on software execution time. Others will not be bounded, or the bound will be unacceptable. They will be considered dysfunctional (i.e., out of the functional domain in an equipment provider's viewpoint but not from a processor manufacturer's).

Even if this is not the primary objective, an interference analysis may also find failure modes within the processor that were not discovered by the manufacturer. Hence, the equipment provider performing the analysis should be aware of these potential failure modes and, in case one is observed, should retain as much information as possible to allow for further investigations in collaboration with the manufacturer.

A major challenge of conducting the interference analysis is to reach a compromise between the complexity (in time and cost) of the test campaigns and the need to have a trustworthy coverage of interference situations.

2.3.2.3 Interference Mitigation

This step takes the set of tagged interference channels as input. For unbounded, unacceptable, and faulty interference channels, mitigation is required. Mitigations can either be internal or external to the processor. As shown in figure 5, an internal mitigation is a restriction (e.g., configuration setting, UD). For example, an internal mitigation can be placed in the way shared resources are used concurrently by requiring that no more than two core processing units (CPUs) be able to request the main memory at the same time. When mitigations are internal, they may alter the processor's UD and influence configuration trade-offs. In this case, the process described above is incremental. Indeed, the interference analysis should take into account these additional restrictions and re-evaluate the interference channels that are impacted.

External mitigations mainly deal with the monitoring aspects and sanction strategy in accordance with the safety objectives. Monitoring may be performed with a low-level granularity (e.g., by tracing hardware events on the processor) or at a higher level (e.g., by sanctioning deadline misses on software).

Finally, collecting statistics on interference may be used to fill a knowledge base that may be reused for other certified contexts, for instance with a higher level of criticality.

This proposed process should be instantiated on a case-by-case basis depending on the equipment being considered, its criticality level, the selected processor, and the level of detail of its internal components. The level of criticality of the equipment directly impacts the depth of analyses that will be performed.

Acceptability criteria can be proposed as early as possible—for instance during the safe design phase of the V cycle (section 2.1.1) and discussed with the processor's manufacturer. Trustworthiness in the interference analysis is obtained after the safety assessment phase.

3. COMPUTER ARCHITECTURES

For the purpose of illustrating the challenges associated with the use of MCPs, two examples of avionics computers are selected at both ends of the spectrum on issues such as real-time, safety constraints, certification, and architectural types (e.g., federated versus integrated architectures). The selected examples are an IMA platform and a flight control computer.

3.1 FEDERATED AND INTEGRATED AVIONICS ARCHITECTURES

Historically, two main trends have shaped current avionics architectures into federated or integrated architectures.

Federated avionics architectures represent the traditional or legacy approach. They are based on various computers for which there exists a one-to-one relationship between an avionics function and a computer. Consequently, these architectures require a significant number of point-to-point interconnections using legacy communication means between these computers to create a fully operational system. The main advantage of this architecture is that it leverages only a few complex COTS components.

On the other hand, IMA architectures bring additional flexibility thanks to standardized and common interfaces. IMA architectures have a significant impact on weight, as fewer wires and interfaces are required. Moreover, a single IMA computer can host multiple avionics functions with different levels of criticality. In today's avionics, IMA is primarily used for applications requiring high-performance computational power rather than for applications requiring hard real-time. The selected example in this report follows this approach. IMA's critical feature of resource sharing emphasizes the need to think globally during an IMA computer development process and to design an overall fault-tolerant architecture on top of commoditized computers. These architectures are more network-centric than computer-centric federated architectures.

In today's avionics, both architectures coexist within an aircraft. The findings and recommendations in this report are illustrated using a high-performance IMA computer for integrated avionics architecture and a flight control computer for federated avionics architecture. The objective of using both examples is not to advocate one over the other but rather to emphasize the trade-offs while choosing one over the other.

3.2 CLASSIFICATION CRITERIA

This section provides details on the main criteria and constraints to be considered during the AEH computer development and processor selection phases. These criteria include performance, real-time capability, safety, and certification.

3.2.1 Computing Performance

Computers are first categorized according to their performances. For both IMA and federated computers, computing performance is a good indicator of how fast an avionics function can be executed. For IMA computers, it also relates to how many functions can be hosted on a single computer. As such, performance is a raw bottom-up metric that is relevant when selecting components such as the processor. However, it does not directly provide an indication of whether real-time constraints will be met or not.

IMA computers are based on high-performance microprocessors because they typically target performance-demanding functions. On the other hand, federated computers (e.g., flight control system) typically host less demanding functions in terms of performance. Hosted functions, meanwhile, require meeting very stringent deadlines. Flight control computers are therefore based on microcontrollers, which embed less complex and more predictable processor cores.

3.2.2 Constraints of Real-time

Embedded avionics functions often face real-time constraints. As an example, avionics functions hosted on flight control computers have short real-time cycles of around 10 milliseconds. Computational margins are also very tight (on the order of a few milliseconds). Therefore, the computer design is generally based on a microcontroller, which embeds at least a simple processor core and a built-in memory with a predictable behavior.

Real-time constraints are currently less stringent on IMA computers. The need for hard real-time constraints is first associated with the function implemented on the computer. Because these computers currently aim at hosting several avionics functions, which need less hard real-time constraints, their main concern becomes computing performance. In the future, one could imagine stringent real-time constraints for IMA based on the functions they would embed.

Nevertheless, real-time constraints impact function allocation. Once the performance requirements are achieved, a provisioning assessment is considered [4] to allocate avionics functions to IMA computers. Finding such an allocation can be tricky because it involves a significant number of variables and criteria to be considered. One criterion of this provisioning assessment is the real-time constraints for the avionics functions. In general, these constraints

have longer real-time cycles compared with the federated flight control computer example, and the real-time constraint related to the variation of latency in delivering a response may range between 10 and 200 milliseconds.

Other criteria to be considered in the provisioning assessment are the safety aspects of the avionics functions.

3.2.3 Safety Considerations

IMA computers are sharing resources between functions that may have different levels of criticality. A less critical function (e.g., DAL D) must not impact the behavior of a more critical function (e.g., DAL A). Robust partitioning, as defined in [4], is a pillar of IMA computer design. Partitioning properties necessary for IMA computers require embedding elaborated safety mechanisms (e.g., MMU and IOMMU), which should be considered when selecting the processor.

Flight control computers are hosting one of the most important functions of the aircraft: the ability to control the aircraft in the air [1]. Consequently, flight control computers only embed highly critical functions. More generally, federated computers host dedicated avionics functions with a single associated DAL and are not sharing resources across mixed-criticality functions.

Finally, for both IMA and federated functions, specific safety architecture patterns, such as a command-monitoring or triplex architecture, can be selected for a highly critical avionics function to meet a prescribed safety target. Moreover, dissimilarity requirements on hardware or software components can be added for specific cases.

3.2.4 Certification Frameworks

Avionics functions are considered within different certification frameworks when they are to be hosted on IMA computers or on federated computers. Figure 5 is excerpted from [1] and depicts the relationships between applicable guideline documents.

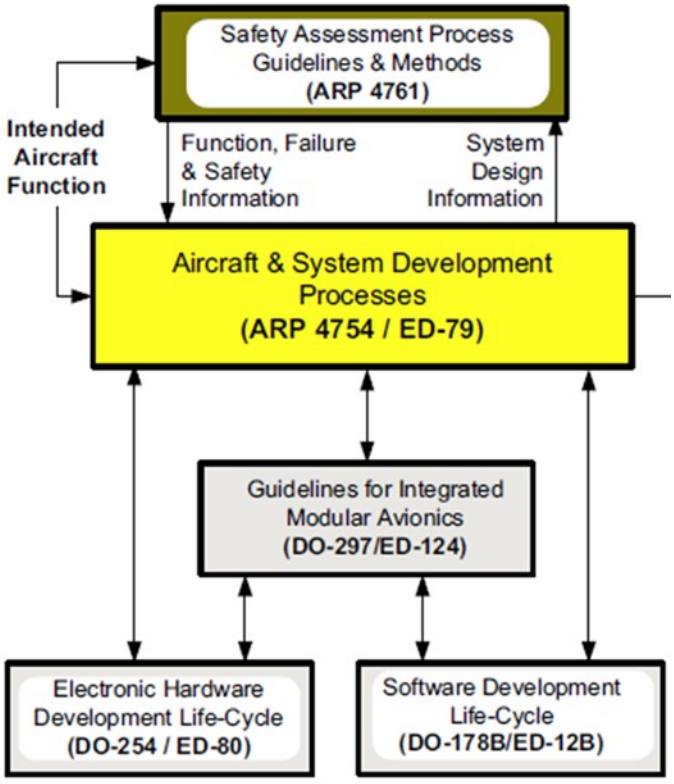

Figure 5. Certification frameworks

For IMA computer-based developments, DO-297 [4] defines an additional certification framework. The need for this addition was mainly justified to cover and specify the necessary additional activities to be performed in the presence of a high-level of resource sharing (both hardware and software resources) as is typical in IMA computers. The guideline document also details IMA stakeholders' roles and associated objectives and activities.

Federated architectures such as a flight control system rely on dedicated computers and consequently fall under the legacy certification framework (see documents in figure 5, excluding DO-297 document).

3.3 SUMMARY OF KEY FEATURES FOR COMPARING ARCHITECTURES

Table 1 summarizes key features for comparing federated and integrated avionics architectures for a high-performance application. Numerical values in the case of a federated architecture refer to a flight control computer. The table highlights that high-performance IMA modules are more constrained by performances than the flight control federated computers and that federated flight control computers are more constrained by real-time aspects than the example IMA modules.

Table 1. Key features of high-performance IMA and federated avionics architectures

Feature	IMA Platform	Flight Control Computer
Required computing performance	• High performance core processor • Branch prediction and speculative execution activated • One to several levels of cache activated	• Simple core processor • Possible branch prediction, no speculative execution • None or only one level of cache activated
Real-time constraint of function	• Ranging from 10–200 milliseconds	• Less than 10 milliseconds
Qualitative safety constraints (note)	• Embed applications of mixed criticality from DAL-D to DAL-A • Specific safety architecture patterns	• Embed applications of mixed criticality level that are not partitioned in a robust manner (ad-hoc segregation)
Certification framework	• RTCA DO-297 both conventional (SAE ARP4754A) and incremental certification processes	• Traditional: SAE ARP4754A minus DO-297

Note: Quantitative safety objectives are out of scope for this research.
DAL = development assurance level
RTCA = Radio Technical Commission for Aeronautics

The above distinctions emphasize the need to elicit the constraints to which the processors are submitted before tackling the issue of non-determinism in the MCP.

4. ISSUES WITH CURRENT GUIDANCE FOR MCP DEVELOPMENT PROCESS

Several avenues can be envisaged to cover the various issues associated with the use of MCPs. The following sections cover existing guidance in view of specific issues. The first avenue that has been explored was to address MCP issues in a conventional DO-178C [5]/DO-254 [6] and additive documents [7, 8] way. The recommendations in CAST Position Paper #32 [9] can be considered to be in that vein. For interferences, the FAA/European Aviation Safety Agency (EASA) Issue Paper/Certification Review Item, EASA certification memo on COTS assurance, and RTCA DO-297 are discussed.

4.1 VERIFICATION GUIDANCE AND CAST POSITION PAPER #32

As noted in the CAST Position Paper #32 [9], existing guidance in RTCA document DO-178C only covers the verification process for software installed on one core. It does not address the verification of software executing parallel tasks on separate cores of an MCP. Therefore, the verification of potential interference resulting from parallel execution cannot be performed. As interference is a major source of non-determinism, lack of guidance on adequate verification for MCPs is a major concern.

The major concern pertains to the dynamic control of threads in an MCP. In the absence of specific guidance, the applicant must demonstrate how they plan to adapt existing guidance to

demonstrate that the software behaves in a deterministic manner. The scope of the demonstration includes the selected MCP architecture and its impact on dynamic features, selected operating system (OS) and its control over parallel tasking, and specific applications hosted on the MCP. The details to be provided cover both the verification process and the tools used to support the demonstration of deterministic behavior.

Furthermore, the field of IMA offers guidance through industry standards on how to perform verification on a platform where hardware, software application, and OS tightly interact. The approach is incremental whereby the OS and any hardware/software interface is integrated and tested with the host hardware first, then each application is individually integrated and tested, and finally the entire system is tested.

CAST Position Paper #32 recommends that such a gradual approach be applied to MCP and organized to meet the specificity of the execution on multiple cores. The paper's six recommendations are recalled below:

- MCP_Software_2: The applicant has described in their software verification plan (SVP) the environment to be used for each software test activity. For any testing that will not be conducted using the target MCP, the applicant has described the environment that they intend to use, their rationale for using a different test environment, and the reason they consider that test environment to be sufficiently representative of the target MCP.

- MCP_Software_3: The applicant has verified that each OS and/or software interface with the MCP hardware complies with applicable objectives in DO-178B or DO-178C (e.g., DO-178C reference A-5 numbers 1–9, A-6 numbers 1–5) when installed on the MCP target.

- MCP_Software_4: The applicant has verified that each individual software application hosted on the MCP complies with the applicable objectives in DO-178B or DO-178C (e.g., DO-178C reference A-5 numbers 1–9, A-6 numbers 1–5) when all of the applications hosted on the MCP are executing in the intended final configuration of the processor and its hosted software.

- MCP_Software_5: The applicant has verified that the data and control coupling between all software components hosted on the MCP have been exercised during software requirement-based testing including exercising any implicit (e.g., through interconnect features) or explicit interfaces between the applications via shared memory and any mechanisms to control the access to shared memory—and that the data and control coupling is correct.

- MCP_Software_6: The applicant has conducted robustness testing of the interfaces and the features of the MCP both when software applications are executing individually and when all the software hosted on the MCP is executing, and has verified the compliance of all the hosted software with the applicable objectives in DO-178B or DO-178C (e.g., DO-178C reference A-6 numbers 1–5) and with the resource allocation to each application under these conditions.

With the exception of MCP_Software_3, all of the above recommendations are applicable to software with development assurance from C to A.

4.2 CURRENT GUIDANCE WITH RESPECT TO INTERFERENCES

The objective of this section is to illustrate the point of view of existing guidance on the question of interferences in MCPs. In some cases, the guidance has been interpreted to cover the case of interferences, while this was not the intention.

4.2.1 Revised FAA Issue Paper on MCP

The first issue of this document traces to the CAST Position Paper #32 [9] and EASA multicore certification review item (CRI). It was released in 2012 to tackle multicore-related issues and defined requirements regarding the use of dual-core MCPs that host safety-critical applications. The document is currently being revised by certification authorities and will be published as an FAA Issue Paper and EASA CRI. Its scope will extend beyond two active cores.

The main requirements in this document can be summarized as follows:

- Identify shared resources used in MCP.
- Identify configuration settings and define UD of MCPs.
- Identify interference channels and associated mitigations.
- Define relevant integration scheme depending on interference channel analysis.
- Define relevant monitoring of MCP.

Compliance to these requirements must be demonstrated.

4.2.2 EASA Certification Memorandum SWCEH-001

This currently published EASA Certification Memorandum (CM) [8] provides non-binding guidance material on COTS, in general, and on highly-complex COTS, which, per such guidance, includes MCPs. In addition to recommending activities to be performed and data to be collected depending on COTS complexity and criticality while taking into account relevant product service experience, the CM calls for quite a few additional expectations related more specifically to MCPs:

- The complexity of MCPs and the proprietary nature of their design call for more in-depth design data. Furthermore, it is assumed that these data are not readily available to users without specific agreements with the COTS supplier.
- The validation of the MCP UD is emphasized due to numerous built-in functionalities featured by MCPs that are potentially affecting the intended functions ultimately performed by such COTS.
- Partitioning analysis is an expected practice whenever COTS is involved in mechanisms implemented to ensure robust partitioning between independent software portions, which is the case for MCPs hosting multiple software functions.

4.2.3 RTCA DO-297

DO-297's [4] objective is to provide guidance for integrating hosted applications on an IMA platform. As hosted applications in an IMA context may require independence for system-level safety reasons (e.g., hypothesis taken in the FHA), it is necessary to provide an adequate partitioning solution on an IMA platform. The answer is robust partitioning. The overall approach is based on the identification of interference channels and relevant means to mitigate their effects with regard to the safety objectives.

MCP-related challenges are not necessarily relative to IMA, but there is a similarity (considering the CRI's objectives) because the identification and mitigation of interference channels is also the approach to obtain assurance for MCP usage.

5. MCP FAILURE MODES AND HAZARD IDENTIFICATION

Following the safety process indicated in section 2, the analysis follows the three steps indicated below:

1. Identification of functional failures or hazards is covered in this section.
2. Determination of the effect is addressed in section 6.
3. Definition of potential mitigation methods is covered in section 7.

5.1 MCP DESCRIPTION

In this section, the methodology described in [10] is applied to MCP to support the determination of failure modes. Three breakdown levels are envisaged:

- Black-box level: The failure modes pertain to the output flows of the MCP. As these failures are observable outside the MCP, architectural means are applicable as a means to detect, identify, and mitigate these failures.

- Grey-box level: The breakdown allows the identification of the key elements such as the interconnect, the shared caches, cores, and peripherals. The level of description must remain clear of intellectual property, but allow the identification of generic faults, failure modes, and their common causes.

- White-box level: This level of description would allow the identification of internal failure modes and the description of the failure path down to impact on the MCP output. This level of description is rarely accessible to the AEH manufacturer; The issue is therefore to achieve confidence in the coverage of the failure mode capture.

Three abstraction levels may be used, independently from the achievable breakdown level, for the analysis of information exchanged internally to the MCP and with its environment. These levels are:

- Functional level
- Logical level
- Physical level

The functional failures are typically expressed in either the loss or the malfunction of the function being rendered. At the logical level, the failure modes are described with respect to the states of the exchanged message. At the physical level, failure modes relate to the physical characteristics of the signals, which become ineffective for complex systems such as MCPs. One exception to this statement is the identification of failure modes related to the technology, such as single-event effect, voltage/current scaling, etc.

5.2 DESCRIPTION OF FAILURE MODES

The consideration of failure modes is the first step to determine a generic fault model for MCPs. A mode is defined as the way in which a given behavior is perceived. Once described, the modes can be sorted using a failure mode meta-classification at the logical level, applied to an MCP treated as a black-box [11]. This section describes the failure modes and builds such a meta-classification for modes associated with non-determinism.

5.2.1 Types of Failures

The content of this section is not specific to MCPs. The guideline document ARP4754A [1] distinguishes between:

- Random hardware failure: failure occurring at a random time, which results from one or more of the possible degradation mechanisms in the hardware [12].
- Systematic failure: failure related in a deterministic way to a certain cause, which can only be eliminated by a modification of the design or manufacturing process, operational procedures, documentation, or other relevant factors [12].

In terms of their impact, failures are classified as the following:

- Transient (or soft errors)
- Permanent (or hard errors)

5.2.2 Expression of Failure Modes at Logical Level

The methodology in [10] provides a way to classify logical-level failure modes into classes. Using the example of timing-related failure modes, the starting point is to consider the states in which a message transmission can be. The message can be:

- Normally received.
- Lost.
- Incorrect.
- Untimely received (with this condition being repetitive).

The causal factors for the failure of the message transmission are of two generic types:

- A structural or temporal disturbance of the message encapsulating information that prevents the entry into admissible states defined for the information.
- A disturbance in the transmission of the message (although it was sent at the right time) so that the state transition is not realized, is realized in an untimely manner, or is realized between two states that should not be linked (forbidden state transition).

The failure modes are therefore classified into [13]:

- Loss of message: This mode represents the absence of message delivery when it should have been emitted. As an independent mode, this mode is meaningful if the energy state at the loss of message is not an admissible logical state of the information. If the energy state is admissible (e.g., after a short circuit on the transmission line), the message will be interpretable by the receiver (e.g., as a zero value). A modeling decision on the part of the author of [10] leads to assimilating the above condition to an impossible transition of information to another state.
- Untimely transfer of message: This mode includes two sub-cases: 1) a message transfer that is in advance of the expected time, and 2) a late message, also referred to as abnormal latency.
- Abnormal sequence of messages.
- Untimely or forbidden transition of information: This failure mode is also denoted as information corruption of erroneous information transfer.
- Impossible transition of information: This failure mode merges with several of the categories above depending on the specific conditions. For example, if this failure mode affects only a part of an information burst, then it corresponds to erroneous information transfer. If it affects a complete information burst or several bursts, it corresponds to an abnormal latency in the information transfer.

Figure 6 illustrates the use of the above failure mode classification to categorize applicable modes of non-determinism.

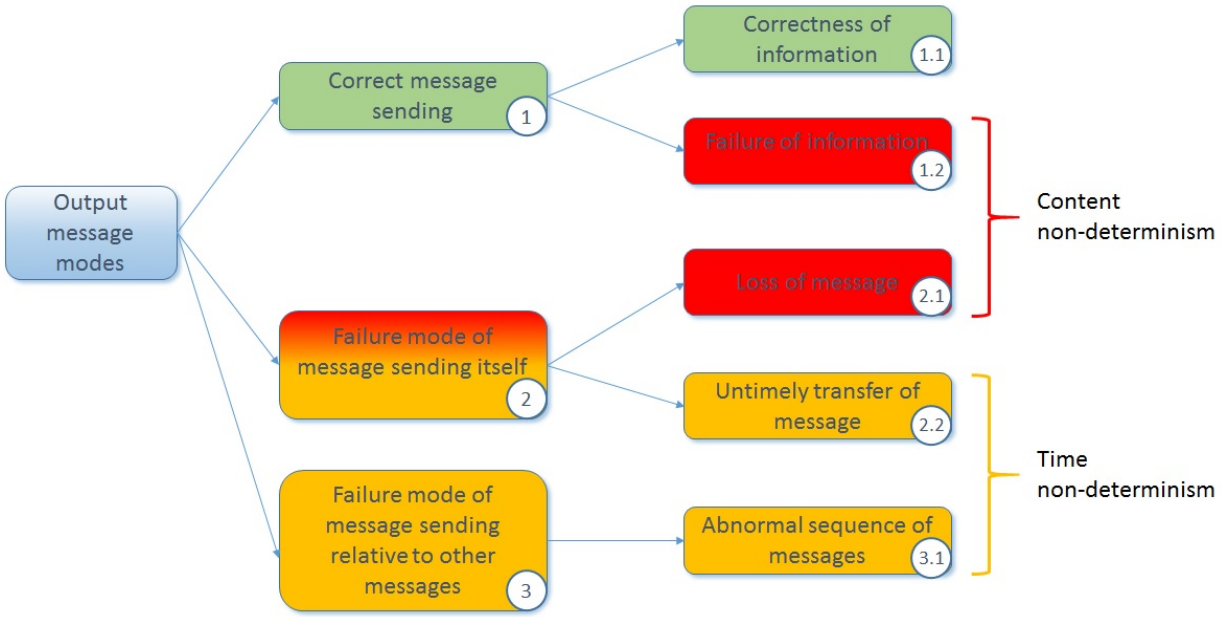

Figure 6. Relationship between failure mode classification and non-determinism modes

The possible behaviors resulting from an MCP sending a message are defined, in figure 6, as falling into three categories:

1. The message was delivered at the right time and in the right sequence. Either the content of the message is correct (case 1.1), or it is erroneous (case 1.2).
2. Sending the message fails. This failure mode can be reached either because the message was never delivered (case 2.1), or the message is untimely transferred (case 2.2). Note that "untimely transfer" in case 2.2 includes both cases where the message is abnormally early or delivered with an abnormal latency; most often, the transfer is late.
3. The message is sent out of sequence. This case illustrates abnormal sequences of messages (case 3.1).

Through this simple example, two modes of non-determinism can be defined:

- Non-determinism related to content with associated behaviors described as:
  - Transmission of erroneous output information (case 1.2)
  - Loss of one or several output messages (case 2.1)
- Non-determinism related to timing with associated behaviors described as:
  - Untimely transfer of message and in particular delayed message (case 2.2)
  - Abnormal sequence of messages (case 3.1)

Time non-determinism can be caused not only by the failures associated with the processor but also by behaviors compliant with the processor specification, such as missing application deadlines. Furthermore, the failures associated with the processor (systematic or random) are the only ones that can lead to content determinism. Therefore, in the framework of processors used in AEH, content determinism is ensured through the processor specification. More details regarding the sources of content and timing non-determinism can be found in section 6.1.2.1. In summary, non-determinism related to content points to structural causal factors (e.g., random hardware failures, manufacturing errors, or use of hardware feature outside specified domain). Non-determinism relating to timing points to temporal causal factors (e.g., uncertainty on cache content, uncertainty on buffer capacity, or interferences).

The definition for determinism from DO-297 covers both aspects of content and timing [4]. Nevertheless, it is useful to keep in mind the distinct definitions for content and time determinism because the causes for the associated non-deterministic behaviors are different.

An interesting and anecdotal comment is that the scientific papers reviewed for this research do not seem to agree on which source of error (temporal or structural) is the most relevant for MCPs. While the temporal sources are specific to the multicore aspect, the structural causal factors are common between single cores and multicores. Therefore, the traditional mitigation methods of fault detection combined with health monitoring are adapted to the MCP context.

5.2.3 Failure Modes at Hardware-Software Interface

The embedded software is an element of context for the MCP hardware. The software-hardware interface is therefore the source of additional (and new) failure modes. While these modes can be described as inherited from the comprehensive failure model described in the previous section, they are elicited below based on the responsibilities of the hardware toward the software:

The first responsibility is to realize the computation requested by the software, which is functionally described as:

- Get software instruction.
- Get data needed for the computation.
- Push the computation results.

The failure modes associated with these functions are:

- No program instruction or data retrieved
- Erroneous instruction or data retrieved
- Latency in data delivery (or maximum execution time drift)

In an IMA platform or MCP context where more than one application is embedded, an additional responsibility relates to partitioning of the software application. The associated failure mode can be expressed as inversion of tasks (tasks are not sequenced in the intended order across applications).

The non-respect of partitioning can lead to the following failure modes:

- Loss or blocking of program instruction output (equivalent to “no program instruction output”)
- Cross-corruption of two independent software applications (equivalent to “erroneous calculation output”)
- Latency in program instruction output

5.3 ALLOCATION OF HAZARDS ONTO MCP ARCHITECTURE

For convenience, the hazards have been sorted according to the MCP feature.

5.3.1 Hazards Associated with the OS/Hypervisor

When considering the MCP software and the OS or hypervisor, examples of hazards that can be identified as part of the FHA include:

- Deadlocks.
- Data corruption (for which one of the causal factors may also be related to the hardware).
- Instruction malfunction (can also be caused by hardware).
- Software execution slowdown.

Regarding the issue of instruction malfunction, in the context of single-core, instruction models using cooperative tasking do not require the implementation of mutual exclusion. These models are no longer valid in the context of MCP. Finally, the main causal factor in the failure condition of software slowdown is interference, which is addressed specifically in this report.

5.3.2 Hazards Associated with the Interconnect

When considering the interconnect, examples of hazards associated with transaction services include [14]:

- Silent loss of transaction.
- Silent transaction corruption.

The use of the adjective “silent” in this context means that the functional failure or hazard is not being signaled with an error (e.g., undetected). Corruption of transaction may be the result of transaction collision or can be due to an external event, such as a single-event upset (SEU).

Network-on-Chip (NoC) emerged as a technology for complex MCP-based System-on-Chip (SoC) to address performance and power consumption requirements for which global interconnect designs are becoming a bottleneck. NoCs establish a controlled communication mechanism between the different blocks, such as to minimize causal factors of collisions and errors [15]. The NoC provides alternative paths compared with traditional interconnect, which increases the reliability to hardware faults.

Process variation (decreasing feature size) has a direct correlation with the increase of delay faults, which can cause collision of packets and increase the latency—both of which are captured in transaction corruption [16]. Process variation affects the links used in communicating between different cores in NoC designs via delay variation, which may result in the violation of deadlines.

5.3.3 Core Failures

Core failures can be divided into two different types: permanent failure and transient failure. The main causal factor to permanent failure is manufacturing defects, which are detected and mitigated during production testing. Transient faults, on the other hand, are more difficult to mitigate. Their causal factors include cosmic radiation (e.g., single-event effect, alpha-particles) and noise (e.g., random telegraph noise).

5.4 IDENTIFICATION OF INTERFERENCE PATHS

As represented in figure 7, an interference path is a configuration wherein several initiators (e.g., CPUs) use various targets (e.g., memories, I/O) at the same time, while the electronic activity they create crosses one or more interference sources. For example, the core in the center of figure 6 uses the memory in the center along the path highlighted in pink through the right-most arbiter. That same arbiter is used by the direct memory access (DMA) in the top-right corner to use the slave I/O in the lower right corner along the path highlighted in green. Thus, the pink path and the green path may interfere.

This section proposes a classification of interference sources.

As depicted within the overall interference-aware safety process in figure 4, the identification of interference paths is composed of:

- A bottom-up analysis that lists the interference sources encountered on the processor.
- A top-down analysis that identifies, for given initiators, which targets are concerned by the operations. For instance, a common side effect of I/O operations is to request a memory to fetch an IOMMU configuration that will be cached. Potential sources of interferences affected by this side activity need to be taken into account.

The combination of these top-down and bottom-up analyses allows for building the set of interference paths.

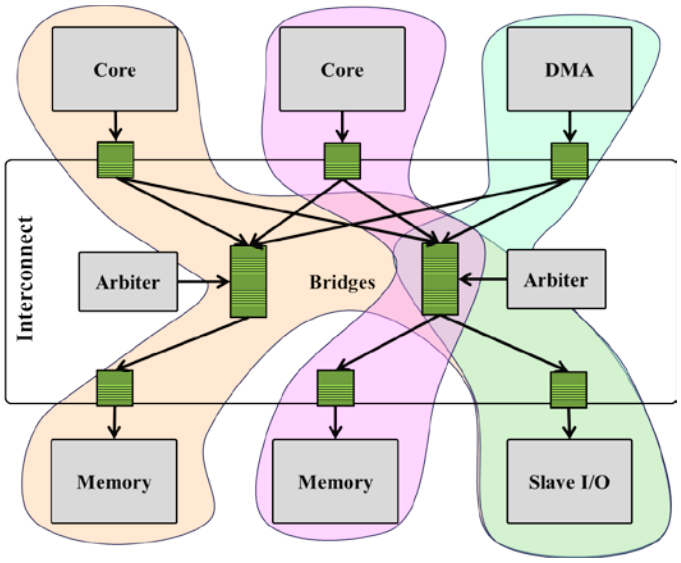

Figure 7. Example of interference path

5.4.1 Bottom-up Analysis to Identify Interferences Sources

Sources of interferences can be identified from a bottom-up analysis on the usage of the processor's shared resources and on the embedded OS, if any. This analysis depends on the restrictions imposed by a UD over the shared resources. The classification of interference sources mirrors the broader classification scheme for sources of non-determinism presented in section 5.2.2.

5.4.1.1 Physical Sources

These sources are linked to hardware components and mechanisms.

5.4.1.1.1 Internal Sources

Internal sources are components that may process concurrent activities. Examples of such components include the main memory, interconnects, and high-speed buses. These kinds of resources are included in the cache and buffer paradigms. Hardware interferences are being considered according to these paradigms.

5.4.1.1.1.1 Cache interferences

The cache paradigm was defined as the combination of a set of data that could be requested and a replacement logic that maintained this set of data according to the incoming requests. It is not limited to the usual cache hierarchy that is encountered in almost all processors, but also extends to features such as open pages table in double data rate (DDR) controllers.

Caches were associated with a source of non-determinism dealing with uncertainty on their content. The same applies to interferences. This source of interferences deals with non-simultaneous operations. Intuitively, an incoming operation will alter the cache's content by

triggering the replacement logic. When another operation arrives in the cache, requested data may have been removed by the replacement logic, thus entailing a cache miss. Even if operations occurred at different dates, they interfered.

5.4.1.1.1.2 Buffer interferences

The buffer paradigm is defined as a set of requests that is fed by incoming flow combined with an arbitration logic that selects the requests to be emitted. It refers, for instance, to arbitration mechanisms in the interconnect, but can also apply to request queues located in high-speed buses and DDR controllers.

Buffers generate interference when their arbitration logic depends on the presence of requests from different cores. In other words, it is a source of interferences between simultaneous operations.

5.4.1.1.2 External Sources

On an MCP, I/Os are often shared between several applications running on several cores. Some I/O controllers (e.g., peripheral component interconnect express [PCIe]) are capable of managing multiplexing on their own both for incoming and outgoing flows. That multiplexing can be described with cache and buffer paradigms. However, the processor has no fine view or control on incoming flows received by several remote peripherals. That is why they are refined as a distinct source of interferences.

5.4.1.2 Logical sources

These sources are linked to software mechanisms that are implemented within the platform software (e.g., real-time operating system [RTOS]).

The notion of buffer and cache paradigms also applies to logical sources of interferences, even if this kind of source is traditionally addressed in partitioning analyses. Locks and semaphores are examples of logical sources of interference that do not refer to cache or buffer paradigms. They can be requested at the same time by different tasks on each core, while, at most, one of them is granted the resources (and sometimes all tasks may fail).

5.4.2 Top-Down Analysis for Identification of Interference Paths

The set of interference paths can be built from a top-down analysis performed on initiators. Depending on the operations they can initiate, various components can be attained even when they are not the main targets of these operations. For instance, a COTS processor often contains internal cache memories, such as IOMMUs, that are loaded at runtime from a global configuration within the main memory. Thus, operations on I/Os may also interfere with concurrent operations on the main memory.

These targets and the resulting data paths (the components that are impacted) should be listed to determine what sources of interference they reach. Interference paths can be obtained by combining data paths from several initiators that share some interference sources.

5.5 PRIMARY CAUSAL FACTORS FOR UNRELIABLE HARDWARE OPERATION

The most important sources of unreliable hardware operation with the potential for failure are:

- Process variability: This causal factor may lead to heterogeneous operation of identical components on one chip.
- Process shrinking: This causal factor leads to soft or transient errors.
- Extreme operating conditions resulting in accelerated aging or wear-out.

These factors are not specific to multicore. Their effects are studied for COTS components that may be single processor based. The industry trend is, however, to integrate more and more processor cores on a single chip, exacerbating the effect of these causal factors.

6. DETERMINATION OF EFFECTS

The determination of the effects of a failure mode is the second step of the safety process.

6.1 SAFETY IMPACTS OF NON-DETERMINISM

This section continues the application of the safety-like approach that combines top-down and bottom-up analyses applied to MCP design. In the previous section, a classification of failure modes associated with non-determinism was presented. For the determination of safety impacts, the top-down analysis refines the need for determinism as it applies to the functions embedded on the MCP. The bottom-up analysis is more traditional in addressing the sources of non-determinism that can be encountered in the design of an MCP.

6.1.1 Top-down Analysis in MCP Design

System design widely applies top-down methods, for example, when the selection of a processor is based not only on the evaluation of low-level criteria such as performance scores (e.g., intrinsic computational capacities of the chip, its power consumption) but also on high-level non-functional aspects (e.g., leveraged buy-out, recurring engineering costs) that are driven by system requirements.

In the case of an MCP, enforcement of determinism is required and is primarily a matter of fault identification and coverage. The use of a top-down approach can bring additional high-level requirements pertaining to the execution model, easing the identification of issues of non-determinism. These additional requirements are, for instance, the mapping of functions onto the COTS perimeter, including task allocation onto cores and the function's use of shared resources (e.g., I/O, memory). The execution model also relates to the OS model used to execute the applications (i.e., symmetrical multiprocessing [SMP] and asymmetrical multiprocessing

[AMP]). This characterization of the resources and their usage is commonly known as the UD on an IMA platform.

The application of the top-down approach is a necessary step to set bounds on the investigation of sources for non-determinism as part of the bottom-up approach (discussed in section 6.1.2). While it facilitates industrial and certification processes by characterizing at high-level sources of non-determinism, this approach is not sufficient. The main benefits are to sort these sources of non-determinism so that the main contributor of non-determinism can be later identified and to recommend the most relevant approach at system/architecture level prior to breaking down to component level.

The top-down approach provides the selection criteria for the MCP itself and for selecting the IP to be activated or configured within the MCP. This configuration defines the UD of the MCP in which determinism is guaranteed.

For a generic design of avionic platforms with MCP, this UD can be viewed as new requirements to guarantee the correctness of the system to meet the global performance objective (e.g., bounded latencies). For IMA platforms, it corresponds to an extended UD compared with the existing one. Adding these system requirements will further ease the evaluation of the number of non-determinism sources, their types, and their importance. Also, the available (or the lack of) processor mechanisms to mitigate their effects can be evaluated (e.g., quality of service [QoS] function, control capability) at an early stage of the development.

This section develops the top-down approach along three axes: function or tasks allocation, execution model or software architecture, and SoC hardware architecture. The approach addresses both isolation and mitigation of sources for non-determinism through three main goals:

- The isolation of sources for non-determinism in the top-down strategy
- The limitation of accessible UD in which non-determinism has to be reduced
- The preparation of possible mitigation strategy if non-determinism sources are still present in the UD

6.1.1.1 Allocation of Functions or Tasks

The allocation of functions on different SCP computers connected to a common network (e.g., ARINC 664 standard) is already a possible source of non-determinism due to possible end-to-end latencies induced by network load. One constraint is then related to the network bandwidth requested by each function that can simultaneously post messages.

If SCP is replaced by MCP, a new contribution appears: An additional latency can be generated by conflicts inside the MCP between common resources access time requested by two functions that can simultaneously request accesses. It leads to a new cause of non-determinism at a very high level.

This identification also leads to a possible simplification in the strategies to guarantee determinism at MCP level. Some strategy for the mitigation, limitation, or avoidance of non-determinism sources can be identified at this level. For instance, one way to greatly reduce the potential interferences identifiable in the bottom-up approach for a given MCP is to allocate applicative tasks to a single core and I/O tasks to the other cores. With this, the bottom-up approach is no longer reciprocal between cores, as non-determinism sources of the applicative core can be analyzed using a known behavior of the other cores. This dissymmetric task allocation scheme also implies that IPs used by the applicative core and I/O cores are not the same. For example, I/O cores mostly access communication and analog IPs, whereas the applicative core essentially uses memories. Interferences will exist between the two regions, as they need to communicate with each other. However, the volume of interference can be reduced to that which can be tightly controlled and bounded. This strategy relies on MCPs having deterministic interconnects, which are often easier to demonstrate with microcontrollers.

For an IMA, a strategy is, for instance, that only one partition can run at a time so the responsibility of multicore usage could be delegated to functions that request high computation performances. This way, using MCP may reduce the efficiency but can also bind the non-determinism to the slice allocated to a given function.

6.1.1.2 Execution Model

The execution model is related to the software architecture, which can be realized by an OS or directly merged within the application in the case of small microcontrollers in federated architectures. Two aspects of an execution model for software architecture are distinguished: the resource management policy and the scheduling policy.

6.1.1.2.1 Resource Management Policy: SMP Versus AMP

SMP or AMP models focus on the way the resources of the MCP are used and managed. Resources are computational units such as cores, I/O, and memory. SMP relies on the use of one software layer (e.g., OS, hypervisor, or middleware) to manage any resources available on an MCP to execute the applications. It implies coherency between the parallel execution of applications and the use of hardware components (e.g., cores, cache, and memory).

Example of source of non-determinism:

This coherency will generate additional traffic and could generate non-determinism in terms of performance (e.g., additional uncontrolled transactions through a common interconnect). These transactions could have an impact on the latencies to access peripheral resources of the MCP.

On the contrary, AMP relies on the execution of several independent software layers dedicated to specific functions (e.g., control and execution) and MCP resources.

Example of source of non-determinism:

AMP or SMP can lead to uncontrolled accesses to a shared resource (software or hardware), which may generate non-determinism. For example, a race condition occurs if two applications try to access the same memory controller, but also if two applications try to access the same service of the OS application program interface (API) or library.

From the point of view of guaranteeing UD restrictions, resource management plays the role of an enabler. Resource management guarantees the respect of the restrictions on UD that were defined to keep the required level of determinism. An SMP software layer can apply these UD rules directly while an AMP would have to rely on a hypervisor layer to ensure the application of some rules. On an SMP system, a single OS runs on all of the CPUs so that the embedded OS (or the user configuring the OS) has control over the entire MCP. On an AMP system, multiple CPUs may have different architectures and embed different OSs. Such an OS has control over the CPU where it is embedded and the shared resources. However, the overall control of the MCP is shared among the cores. Therefore, there may be a problem of coherence in enforcing the UD when a core enforces it one way, while another enforces the UD in an incompatible manner. To correct that situation, an SMP software may be added with the role of enforcing the UD (e.g., static configurations performed at start-up). That SMP software may be a hypervisor, especially if the AMP software is an OS. An alternative approach may be to use a single configuration and a mechanism to designate a master. Both approaches work, but the solution with an SMP hypervisor seems to be more widespread. Resource management layer plays an important role in the mitigation strategy of non-determinism.

6.1.1.2.2 Scheduling Policy

Scheduling policy focuses on the mapping of applications onto the execution units (i.e., the cores). This mapping can follow three main families of strategy:

- Global strategy
- Partitioned strategy
- Semi-partitioned strategy

In the global strategy, one scheduler is responsible for the execution of applications running on several cores at the same time. The applications can be executed on cores without any affinity. The most restrictive strategy of this policy is to authorize only one application at a time to access any shared resources but at the cost of performance efficiency.

Example of source of non-determinism:

Uncontrolled migrations of processes between cores lead to additional execution overheads (due to context software switches, cache misses). These overheads are difficult to evaluate on the worst case execution time (WCET), especially if the MCP has a shared level of cache.

In the partitioned strategy, one scheduler is dedicated to a core, and the applications in a set are statically allocated. Practically, this is the easiest way to port legacy SCP code, but the control of shared resources is difficult to achieve without any other software layer (e.g., hypervisor).

The semi-partitioned strategy is a mix between the global and partitioned strategies. Applications can run on a limited number of cores (which can be a single core). This method can be used on an AMP model to define a precise perimeter on both cores and shared resources (e.g., I/O). For example, the migration of processes can be allowed only within a cluster of cores.

Traditionally, the AMP-partitioned model is considered the most straightforward approach to port legacy applications (i.e., SCP port), with one OS per core. It is a special and restrictive case of AMP.

Scheduling is not considered in general to be a source of limitation for UD. It is a powerful source of mitigation for non-determinism issues. Scheduling will control the manner in which the different functions or tasks will be timely distributed on the various cores. In particular, the detection of execution latency drift and associated sanctions will be applied at this level.

Regarding mixed criticality, the MCP offers the possibility to execute several applications at the same time, possibly with different FDAL[1] if a proper spatial partitioning is set. In this case, heterogeneous FDALs may run at the same time on different cores, regardless of the way these cores interfere (e.g., how one core may slow down another). With the distinction between safety and real-time constraints, state-of-the-art scheduling techniques using mixed criticality can be deployed; these techniques are more adapted and efficient than conventional methods based solely on real-time. For example, allocating higher priorities to more critical functions may seem a natural solution, but it may not be the most efficient. This aspect is different from the global, partitioned, or semi-partitioned strategies.

Example of source of non-determinism:

Uncontrolled execution can lead to the failure of the most critical application, a source of non-determinism at safety level. On an IMA platform, this issue is readily handled via the robust partitioning property.

6.1.1.3 SoC Hardware Architecture

The selection of an MCP is highly relevant with respect to the targeted applications. This selection impacts issues of determinism, UD limitations, and the available mitigations. In the case of an IMA module, the focus in general is on computational performance and will lead to favoring a high-performance symmetric MCP with several identical cores. These MCPs are selected from product lines designed for another market: Some selectable MCPs are network-centric with an appropriate sizing of I/O resources or pixel-centric with additional IPs for image/signal processing.

[1] Note that FDAL is used in the context of MCP sources of interference, as it drives the assurance that the application will not crash by itself.

Similar to SCPs in the same domain, these MCPs are designed to improve the average performance and not the determinism. This influences the hardware strategies for core computation and resource sharing among cores, potentially causing issues of non-determinism that will be detailed in the bottom-up analysis (section 6.1.2).

Due to their intrinsic complexity and possibly wide variety of usage, these MCPs boast a significant configurability (e.g., a factor of 10 can be observed between the last SCP generation and the present MCP generation with respect to addressable configuration register size). The definition of UD is a complex and highly important task. A miscalculation in the selection of the microprocessor (MCP or even SCP) will lead to a more complex definition of UD. The number of possible configurations in which determinism issues will have to be assessed will only be reduced later in the project's life, and therefore the issues of non-determinism will be more complicated to address.

This situation is already occurring in SCPs but becomes even more relevant for MCPs because they contain more IPs that can be masters (or initiators), which can untimely target shared resources causing timing or even content non-determinism.

To simplify this problem, an optimization problem is posed on the microprocessor architecture in which computational performances are maximized for the application while the specified domain is minimized with respect to the UD needed. Despite this effort, the specified domain remains very large in general and includes many IPs that are not useful for the UD; moreover, for IPs included in the UD, many are unused features. Reducing the ratio of unused versus used features is a factor of simplification that impacts the ability to ensure determinism. When it is achieved via deactivation, untimely activation of a deactivated IP or feature should be prevented. This prevention can be performed either by an external mechanism or by a verification process showing that their failure configuration is fail-silent.

SoC for federated applications are generally selected based on their response to real-time constraints, robustness, and health monitoring and fault management capabilities. For these reasons, they are often selected in the families of automotive microprocessors or even microcontrollers that have been developed for critical automotive functions and systems such as the engine control unit, braking system, and advanced driver assistance system.

These MCPs have specific architectures characterized by less complex IPs, simple homogenous or heterogeneous cores, and specific interconnects that are configurable to be deterministic by design. For example, it can be demonstrated that there is no collision between a core accessing one resource through the interconnect and another core accessing another resource. Interference can only occur when the cores access the same resource.

These characteristics make them suitable for federated architectures. Nevertheless, the selection of the right microprocessor for the characteristics of the embedded applications is also very decisive for managing the non-determinism issues. The choice of an interconnect that cannot natively support the segregation requested by the integration strategy may in fact be very detrimental for timing performances.

6.1.2 Bottom-up Analysis in MCP Design

A bottom-up approach aims at obtaining some guarantees over the whole MCP behavior from a systematic analysis of all the MCP components and their interactions. The analysis consists in studying sources of non-determinism, such as failure modes, and worst-case behaviors for predictability concerns. The analysis is performed on each component regardless of other component's behavior.

This approach is the most commonly developed in the literature and is included in the CAST Position Paper #32 [9]. Its main challenge is the need for information on the processor's architecture and behavior. While reaching a sufficient level of detail is often feasible for some components (e.g., CPU), it is less the case for key MCP components such as interconnects. Such a detailed and complete set of information can typically not be obtained from the manufacturer. Therefore, conservative hypotheses are commonly applied to keep the approach feasible.

In this section, such a bottom-up analysis is outlined on a meta-model of an MCP shown in figure 8. The MCP is modeled as a set of autonomous IPs that are connected by interconnects, also sometimes considered themselves as IP. Each IP may be functioning either proactively (thus dubbed "master") or reactively (dubbed "slave"). A master IP may initiate traffic at any time, while a slave IP only initiate traffic to answer requests from a master IP.

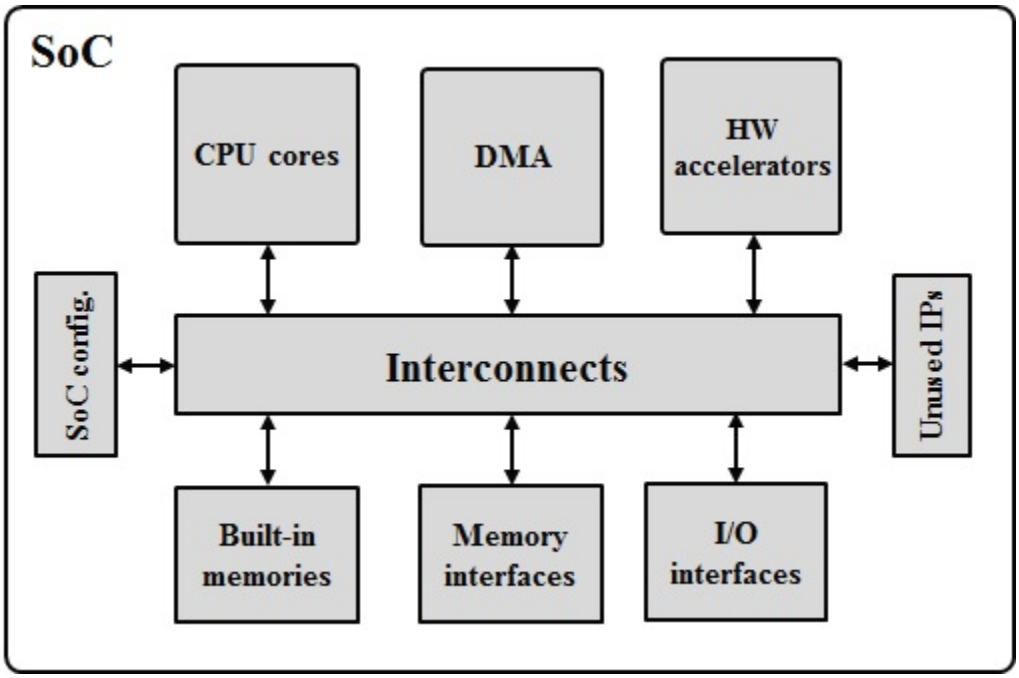

Figure 8. MCP meta-model

6.1.2.1 Generic Classification of Sources of Non-determinism

A bottom-up analysis identifies many sources of non-determinism. However, these sources may originate from similar phenomena and mechanisms. Therefore, a generic classification of sources of non-determinism is useful prior to performing a bottom-up analysis.

6.1.2.1.1 Sources of Content Non-determinism

Similar to any processor, an MCP performs two types of electronic operations: data processing and data exchange. Sources of content non-determinism affect:

- The soundness of operations for processed data with respect to the UD. The associated non-determinism may result from:
  - Persistent faults in the hardware either from design or manufacturing errors, random hardware failures, or aging effects.
  - The use of hardware features outside of the specified domain assessed by the MCP manufacturer.
  - The use of embedded microcode outside of the specified domain assessed by the MCP manufacturer.
  - An update of embedded microcode.
  - A misconfiguration of the component with respect to the UD.

These faults must be assessed using structural failure mode analyses (e.g., FMEA for random failures, except for aging effects for which other statistical models may be used). Moreover, a previous study led by the EASA explicitly recommended restricting the MCP usage to a UD that complies with the manufacturer's specifications [10, 14].

- The integrity of exchanged and/or stored data. This integrity can be corrupted by external events, such as SEUs or internal hardware random failures. Their probability of occurrence may be assessed from statistical data based on the type and size of internal memory and buffers used to exchange the data and other reliability information. On recent MCPs, error-correcting code (ECC) mechanisms are widespread to mitigate such effects.

As far as sources of content non-determinism are concerned, an MCP does not differ much from an SCP. The main difference remains the increased complexity of the processor's internal architecture, behavior, and configuration space.

6.1.2.1.2 Sources of Timing Non-determinism

The main sources of non-determinism associated with MCPs are related to timing issues. Interferences are the most cited source of timing non-determinism, but it is not the only one.

Sources of timing non-determinism are separated into two classes:

- Sources that are local to each component: As stated before, activities performed locally by components inside a processor (SCPs or MCPs) are data processing and data transfers. For example, ECC checking is data processing, while row selection in a dynamic random access memory (DRAM) is considered as data transfer. While sources related to data processing closely depend on each component, sources related to data transfer are generic and can be refined according to the following paradigms:

- Cache paradigm: Data is transferred to facilitate or accelerate further accesses. A cache is closely coupled with a replacement logic that manages its content. Uncertainty on cache content is therefore a recurrent source of non-determinism.
- Buffer paradigm: Buffers are used either to pipeline consecutive or concurrent accesses, or to hide the complexity of I/O internal logic. When a buffer is full, incoming requests are delayed and, thus, a global slow-down of the whole activity on the processor occurs. Uncertainty regarding a buffer's capacity to handle incoming requests is a source of non-determinism.

- Sources resulting from the combined activity of several master components on an MCP: Those sources correspond to interferences.

6.1.2.1.2.1 Issues Related to Cache Paradigm

Caches are commonly found in the hierarchical memory (e.g., Harvard architecture for L1 cache, unified backside L2 or L3 cache [17]), but this paradigm also applies to other components such as banks and row management performed by a DRAM controller, or hardware mechanisms performing posted I/O, such as PCIe. A cache is generally defined as a small set of memory coupled with a logic that processes requests for data at given addresses. It is in charge of fetching and flushing data according to the incoming requests. Many caches are closely coupled with a replacement algorithm.

Timing non-determinism in cache is linked to uncertainty or variability of the cache content, such as the presence of specific data in the cache. Indeed, when cache content is uncertain, it is not possible to know which requests will hit the cache and which will miss. A miss in a cache results in a penalty to request service time that has consequences on software execution time.

Content non-determinism in cache is linked to the integrity of the data, which is considered in a distinct manner from the presence of the data associated with timing non-determinism.

Table 2 describes sources of timing non-determinism in cache and refines the causes of cache content uncertainty.

Table 2. Sources of timing non-determinism for cache paradigm

Source of Non-determinism	Type	Rationale
Non-reproducibility of the sequence of incoming requests for one master	Local	The sequence of requests may directly result from software execution, which may be variable. For instance, IMA software may be interrupted by the OS, which makes cache content dirty.
Use of pre-fetching mechanisms	Local	To improve statistical performances, caches may embed pre-fetching mechanisms. They are triggered after receiving requests in order to anticipate further requests.
Cache replacement logic is non-analyzable	Local	For replacement policies such as LRU or FIFO, static analysis techniques are known [18]. However, other replacement policies may be poorly analyzable by static analyses techniques, as was the case for PLRU until recently [19].
Simultaneous accesses by multiple masters	Interference	Cache contents depend on the global activity of all masters. Therefore, only global cache content analyses can be performed. These analyses are more complex to perform, even if the problem has been addressed in the literature [20].

IMA = integrated modular avionics
OS = operating system
LRU = least recently used
FIFO = first-in first out
PLRU = pseudo least recently used

6.1.2.1.2.2 Issues Related to Buffer Paradigm

Buffers in processors serve two goals: 1) They can pipeline incoming requests sent by one or several masters, and 2) they can be considered as endpoints for some peripherals, thus masking the complexity of a component's behavior.

As shown in figure 9, a buffer is defined as a memory area with the capacity to store a given number of incoming requests. It is associated with:

- A processing logic to process and remove incoming requests from the buffer.
- A scheduling policy that selects the next request to be processed.

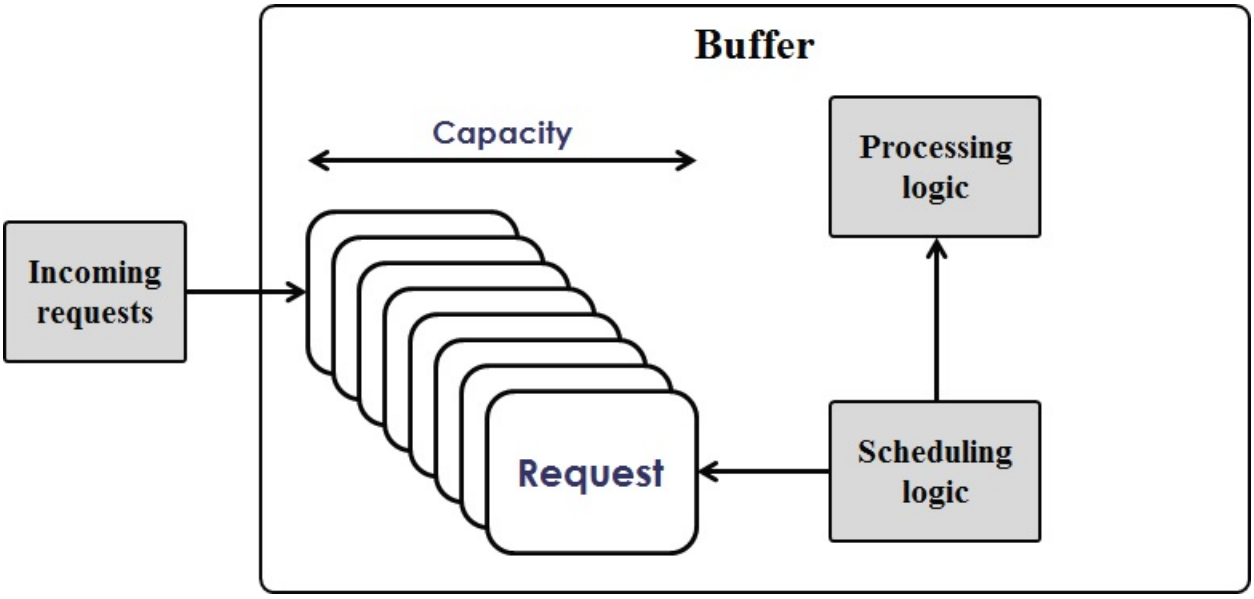

Figure 9. Buffer structure

As further detailed in table 3, sources of timing non-determinism associated with buffer paradigm can be refined as:

- Causes of uncertainty on the buffer's available space: When a buffer is full, incoming requests are delayed and entail a global slow-down of the whole activity on the processor. If a buffer is shared, there can be interferences, a source of non-determinism. If a buffer has a single master, it may be full if its master sends too many requests (e.g., bursts). Although this situation is not an interference, it remains a source of non-determinism, especially when it is unclear whether the buffer is full or not.

- Causes of uncertainty on the buffer's traversal time: When a request is stored within a buffer, it will wait some time before being actually processed by the dedicated logic.

Although interconnect structure is not disclosed, one can assume that some interconnects (such as Freescale's CoreNet™) fall within a buffer paradigm for non-determinism, in particular in order to avoid content non-determinisms [10].

Table 3. Sources of timing non-determinism for buffer paradigm

Source of Non-determinism	Type	Rationale
Lack of information or uncertainty regarding the number and capacities of buffers	Local	Buffer size and characteristics may be partially communicated by the MCP manufacturer.
Lack of information or uncertainty regarding the buffer's scheduling policy	Local	Buffer scheduling policies may be complex or undocumented. For example, incoming requests queues embedded in DRAM controllers are sometimes capable of rescheduling present requests when new requests are received.
Uncertainty regarding the buffer's emptying rate in the processing logic	Local	The processing logic empties the buffer with a rate that may be variable according to the incoming requests and the IP state.
Uncertainty regarding the amount of requests sent to a buffer by one master	Local	As for caches, the sequence of requests may directly result from software execution, which holds some level of variability due to different execution paths, and potential interrupts.
Requests sent to the buffer by multiple masters at the same time	Interference	The remaining capacity depends on the global activity of several masters, which requires guarantees at a global level.

MCP = multicore processor
DRAM = dynamic random access memory
IP = Intellectual Property

6.1.2.2 Bottom-up Approach on Hardware Resources

The objectives of a bottom-up approach are to identify a list of sources for non-determinism in as exhaustive as possible, analyze only the sources that may have an impact at a higher level of granularity, and filter out the sources with contributions that are negligible or masked by other sources. This last point is especially true for sources of timing non-determinism. Fundamentally, these sources introduce variability in the processing or transfer time of requests. When a request is sent by a device, it encounters several sources of non-determinism that will introduce, locally and individually, variability into the request's processing and transfer time. However, their individual contribution to the overall variability is not balanced across the sources, so that the main effort for mitigating the effect should be spent on major contributors.

In figure 10, we denote cache components (according to the definition of cache paradigm) with a round-corner yellow square, buffer components with a striped green rectangle, and processing components with a gray rectangle.

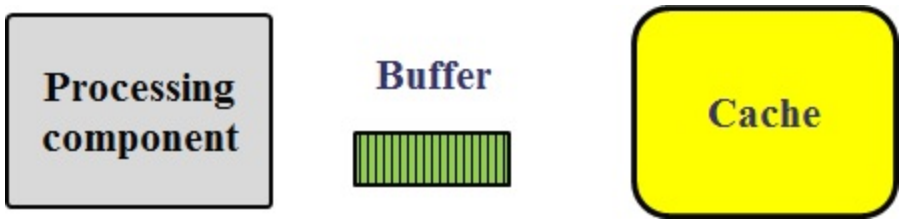

Figure 10. Graphical convention

6.1.2.2.1 Memory Hierarchy

The memory hierarchy contains embedded caches that are either located inside a CPU or shared among the CPUs on the SoC. These caches are small memories built with static random access memory (SRAM) technology and embedded replacement logic. Caches have a significant impact on the platform performance, and their use in airborne systems is common [20].

The memory hierarchy represented in figure 11 can be considered both as a source of non-determinism and as an opportunity to improve determinism on the overall processor.

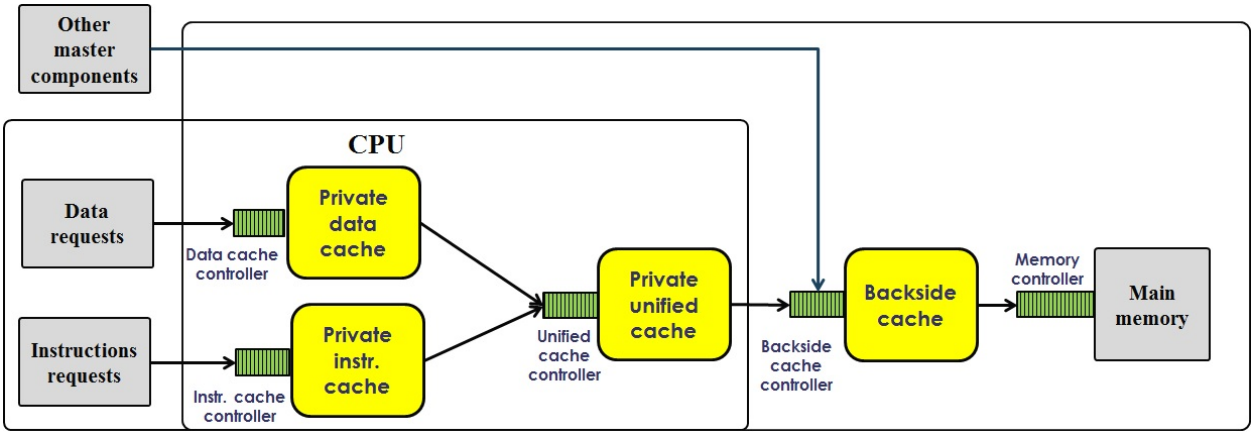

Figure 11. Example of memory hierarchy

Each level inherits from the sources of non-determinism relative to the cache paradigm and buffer paradigms when focusing on cache and memory controllers. Additionally the memory hierarchy reduces traffic density to the main memory. In turn, it decreases contentions on backside memory controllers, thus reducing interferences between concurrent software. Cache sizes and replacement policies are a key factor for reducing the traffic density. Moreover, SRAM and enhanced DRAM technologies used for cache implementation lead to a very low variability of timing of accesses and, therefore, ensure a more stable execution time of the software as long as the requests hit the caches.

6.1.2.2.2 DRAM Controllers

A significant number of MCPs provide one or more interfaces with external DRAM. These memories are organized in banks and are accessed through dedicated controllers. A DRAM controller may maintain a given number of open banks; This corresponds to a cache paradigm. Banks are organized as cell arrays with rows and columns, each cell containing data. The DRAM controller can only perform read and write in opened rows from opened banks. Each opening operation increases the memory's response time.

As shown in figure 12, a DRAM controller is composed of the following subsegments:

- The front-end performs the receipt and scheduling of requests (buffer paradigm). The objective of the DRAM controller (made of the front-end, back-end, and data management) is to provide both the best bandwidth and best response time, statistically speaking, to all requestors. To this end, the front-end segment embeds an arbiter that can reorder incoming requests to minimize opening rows or banks. It relies on tables that are organized according to the cache paradigm.

- The back-end performs request translation (i.e., translating an address into a series of actions) and request propagation in the DRAM control (e.g., implementation of the communication protocol with DRAM). Additionally, a DRAM controller is in charge of DRAM maintenance occurring during refresh operations. These operations are periodically scheduled at bank level. While a bank is being refreshed, its contents cannot be accessed.

- The data management segment of a DRAM controller includes operations on data, such as applying ECC.

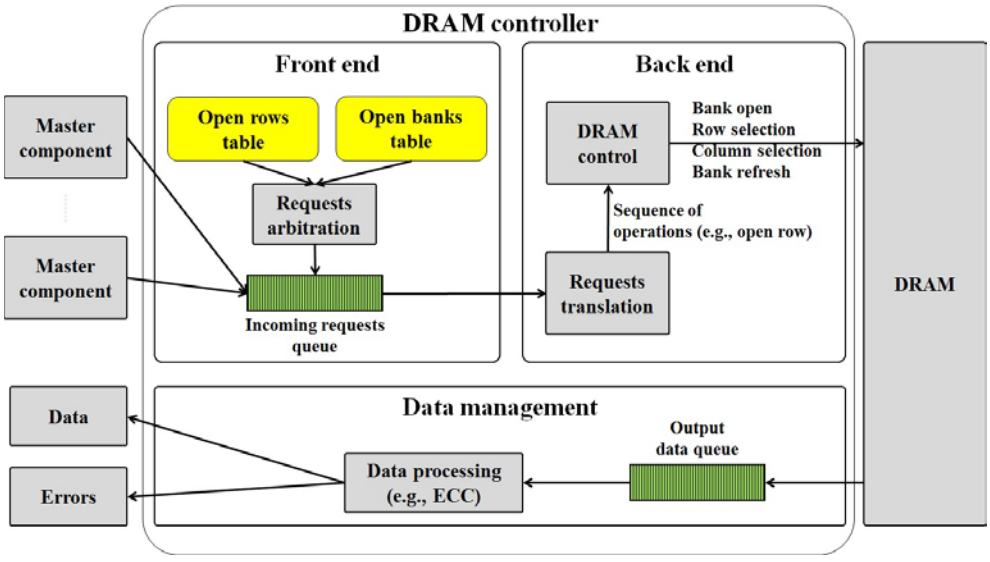

Figure 12. Overview of DRAM controller

Sources of non-determinism are inherited from the buffer and cache paradigms, with respect to the state and management of internal queues and the state of opened banks and rows. For both SCPs and MCPs, external DRAM is commonly considered as the largest source of contention for traffic initiated by master devices, such as CPU or DMA engines. Even in SCPs, interference scenarios are possible.

MCP manufacturers usually propose means to limit the uncertainty on bank state (opened or closed) along two approaches:

- Provide a memory space as large as possible and contiguously mapped by opened banks. This solution can be implemented by enabling interleaving mechanisms, which coordinate several DRAM controllers that separately maintain bank contexts at the same time. This approach is useful when a piece of embedded software requires a large memory space.

- Partition opened bank contexts to limit interferences related to cache paradigm. This approach is the opposite of the previous one, as it consists of providing each piece of software a dedicated context of opened banks and making sure that this context is not altered by other requests.

The issues of non-determinism related to DRAM controllers occur on both SCPs and MCPs. However, the observed behaviors are more common on MCPs because of the increasing number of masters potentially accessing the main memory at the same time.

6.1.2.2.3 Input/Output Interfaces

Several classes of I/O interfaces can be distinguished on a processor, whether SCPs or MCPs. The classification criteria are:

- The existence of support to concurrency at hardware level: When there is support to concurrency, the I/O controller can schedule requests received at the same time from several masters. When such support does not exist, the I/O interface must be multiplexed by software or privatized to one CPU.

- The assignment as master or slave: Depending on this criteria, the I/O interface may generate requests toward the processor's internal resources.

- Whether accesses are immediate or posted: When accesses are immediate (or synchronous), the CPU may stall while waiting for the access to be performed. Any variability in the management of incoming requests by the I/O controller directly impacts the software execution time. When accesses are posted, the contents of the request are stored in local memories and later processed by the I/O controller. The CPU is usually unaware of the date at which the request is processed, except when an error is raised.

Complex I/O interfaces, such as PCIe, support multiplexing at the hardware level and implement posted accesses. A generic representation of I/O interface is sketched in figure 13. The sources of non-determinism for this type of interface are inherited from the buffer paradigm.

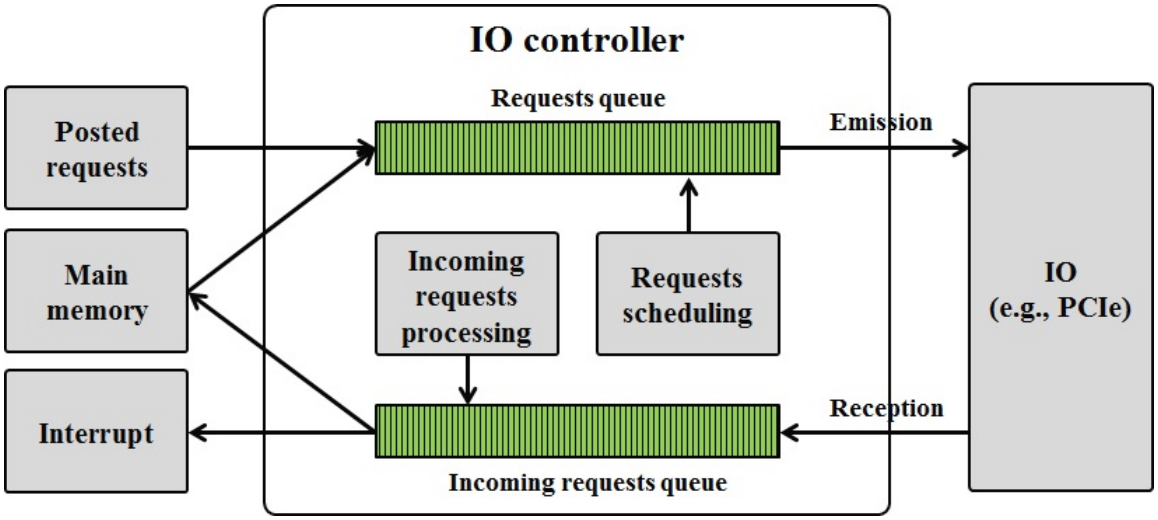

Figure 13. Overview of I/O interface

6.1.2.2.4 Interconnect (SCP versus MCP)

Interconnects are key components in MCPs in charge of propagating the traffic between components. A generic representation of an interconnect is shown in figure 14, for which all internal elements are represented according to the buffer paradigm. Each request, also called a transaction, is forwarded through a set of queues that follow the buffer paradigm. Each queue has an arbiter and associated policy.

Non-determinism within interconnects is therefore associated with non-determinism relative to internal buffers. When interconnect internal buffers are documented, or when the interconnect architecture is simple (e.g., pipelined bus), it may be possible to address the sources of non-determinism at a fairly low level of granularity. When it is not the case, black-box approaches, such as test campaigns ruled by the initiator-target model [21] or by the use of QoS mechanisms associated with in-service experience, can be proposed. However, the difficulty consists in deciding when to stop testing in order to have a correct compromise between hardware behavior coverage and complexity of the test campaign.

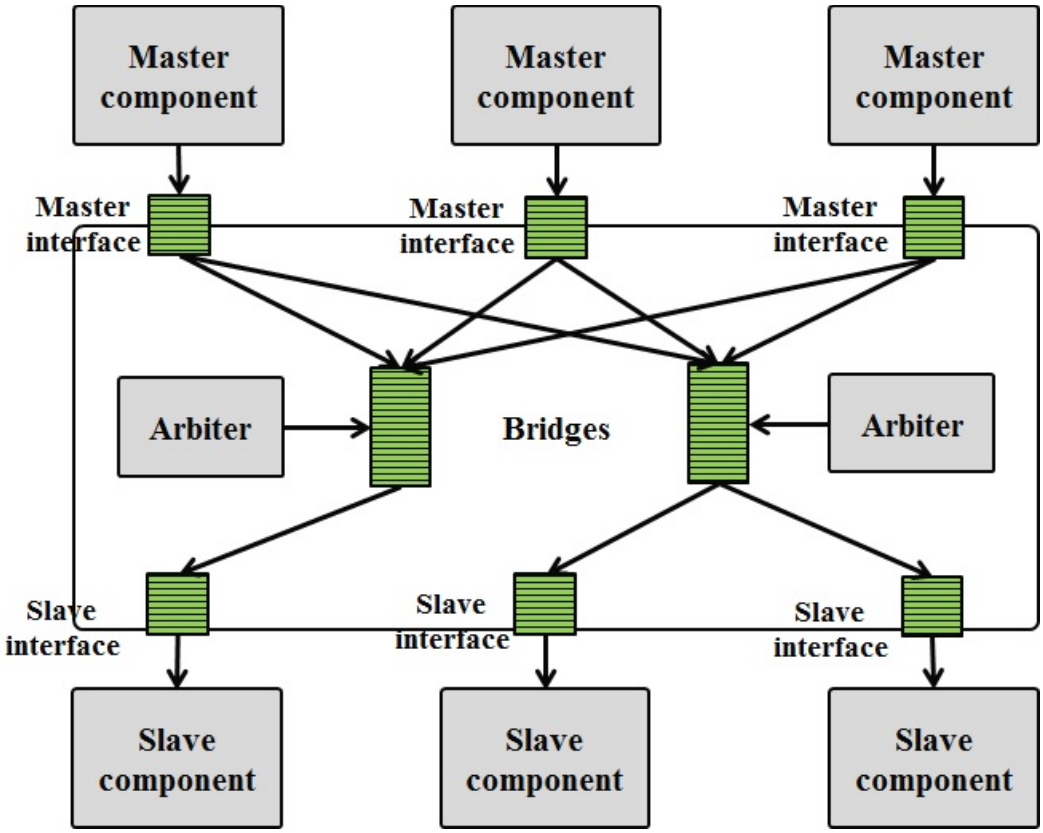

Figure 14. Overview of interconnect

6.1.2.2.5 Other Sources of Non-determinism

Processors, and in particular MCPs, can contain other sources for timing non-determinisms that are not investigated in this report, including DMA, cache coherence, and SoC configuration. None of these mechanisms is specific to MCPs, although their impact is more significant in MCPs. They can be investigated using the same principles as the ones described in the previous subsections.

6.1.2.3 Data Collection to Support Bottom-up Approach

Bottom-up analyses are difficult to conduct, and models for non-determinism are the result of numerous studies. Such studies are based on several types of data:

- COTS supplier public data, including:
  - The reference manual describing the features and operation of the MCP and the behavior of the main IPs
  - The core reference manual describing the features of the core

- The errata list detailing all known silicon errata on the component: The list can indirectly provide important information on the non-deterministic behavior of some IPs and its exploitation in the bottom-up analysis can lead to a reduction of UD.
- Application notes emitted by the manufacturer to configure or correctly use the device: These application notes often complement and particularize the reference manual and eliminate some ambiguities.

- COTS supplier data delivered under non-disclosure agreement
- In-service experience: Sources of non-determinism should be considered for an IP already in use in a microprocessor with in-service or test experience. A significant number of MCP IPs are reused from SCPs. The in-service experience cannot be used for interference reduction studies but may be useful for investigating local sources of non-determinism such as buffer and cache paradigms.
- Test data particularly coming from black-box tests that stress IPs and induce interferences on bottlenecks: This knowledge can be used on a case-by-case basis to confirm the information gained from other sources or complement missing information.

6.2 CLASSIFICATION OF IMPACT FOR INTERFERENCES

As shown in figure 4, an interference analysis takes as input a set of interference paths, filters it according to the UD restrictions (platform and processor), identifies interference channels, and tags each of them as:

- Acceptable: Interferences occur, but a bound has been found on the interference penalty, and this penalty meets the performance objectives for the equipment.
- Unacceptable: The interference penalty could be determined (i.e., a bound can be found), but it does not meet the performance requirements.
- Unbounded: The interference penalty could not be determined.
- Faulty: Tests on the interference path have triggered a failure mode that was not discovered or documented by the manufacturer. Further investigation is required.

There can be up to one interference penalty per interference channel, but a same interference penalty may apply to several interference channels. In the first three cases, the interference channels are not associated with a failure mode and, thus, are only considered from the performance standpoint. In the fourth case, the interference channel has triggered a failure mode and must be analyzed more in depth from a safety standpoint. Even if the failure modes are a concern for the equipment safety, they are addressed by several analyses that apply to complex COTS. Interference analysis can simply be aware of this eventuality and report them, if any are found.

6.2.1 Objectives and Results of the Interference Analysis

The main objective of the interference analysis is to provide a trustworthy performance assessment of the processor. The main challenges are:

1. Dealing with a high number of interference paths: In many cases, it will not be possible to identify all of them. Therefore, a sufficient subset may be covered, as long as a rationale can be provided to justify that the reduced coverage is sufficient. For instance, it can be shown that the gathered interference penalties apply to uncovered interferences paths.
2. Obtaining a trustworthy interference penalty on a given interference path, in spite of:

    a. The high complexity of hardware mechanisms.
    b. The eventuality of black-box and/or partially documented components within the processor.
    c. The limits of testing capacities (e.g., the impossibility of reaching heavy stress levels with given initiators). Defining test space and clarifying its limits in terms of which situations are reachable and which ones are not may be helpful.

For both challenges, termination criteria can be proposed to limit the complexity of the interference analysis. Meeting these criteria can be a sufficient condition to ensure the trustworthiness of interference penalties.

Various approaches and termination criteria can be proposed. They can be more or less formalized and computational. For instance, human-in-the-loop processes can be introduced, including experts' reviews, collaboration with the manufacturer, and collected in-service experience. The in-service experience can cover the processor itself or similar ones, such as the processors in the same series or using the same IPs. These approaches may be developed alongside formal analyses performed on the whole processor or specific components. The worst-case behavior of some components (e.g., SRAM) can be found, supporting the trustworthiness of interference penalty. However, the equipment provider performing the interference analysis should be aware that this might not always be the case. It is important to understand and document the limitations of the interference analysis, so that the rationale associated with the interference penalty is properly justified.

A specific concern points to processors that contain black-box components. Methods have been proposed to tackle black-box components to find failure modes in [21]. The following events are unsatisfactory from the point of view of performance:

1. Unbounded behavior within the limits of the test's exploration space: For instance, it can be necessary to consider an infinite test space when its bounds are not clearly defined. Therefore, asymptotic behaviors can be considered for test configurations that are not reached, but do not seem unreachable. These asymptotic behaviors may not be bounded; thus, the interference channel becomes unbounded.

2. Hidden mode changes and/or subcomponents' activation within the black-box component: These events introduce discontinuities in the processor's behavior (i.e., situations where adjacent tests entail non-adjacent behavior). Having discontinuities is not a problem in itself as long as their impact is covered. What is important is to secure the absence of singularities (i.e., finite discontinuities while the interval between them remains uncovered), so that the component's behavior is unknown. One of the results of the interference analysis dealing with black-box components can be the absence of singularities within a given limit according to metrics defined over the test space.

The interference analysis mainly brings obligations in terms of results (i.e., to have a sufficient coverage of interference paths, identify interference channels, and provide trustworthy interference penalties for each of them). Exhaustiveness is probably not reachable using any one method because of the "black-box" situation when using COTS. Nevertheless, trustworthiness should be reached by combining a semi-formalized approach, knowledge of the device from the MCP supplier, test campaigns, or any other method.

Finally, with this approach, the equipment provider is left the opportunity to propose and argue the methods together with their level of formalization, automation, and human intervention. This plan can be proposed and defended in front of certification authorities at an early stage of the certification process and then be refined during the safe design phase of the V cycle.

As an output, the interference analysis produces the set of tagged interference channels. The unacceptable and unbounded channels can be studied with a mitigation objective in mind. Faulty channels are analyzed from a safety standpoint first before being considered for mitigation.

6.2.2 Example of Interference Impact on Software

To illustrate the effect of interferences on software, consider Freescale's quad-core P5040 processor. It belongs to the QorIQ™ series and embeds four PowerPC e5500 cores, and is preferred by several equipment providers and organizations such as Multicore For Avionics.

In this example, the focus is on: 1) a small piece of code that performs sequential loads of 4K contiguous memory on a DRAM while the backward cache has been disabled ,and 2) on a piece of software within an application performing data fusion within the cockpit's software. This application is single-threaded and therefore will not benefit from intra-partition parallelism, such as SMP execution. For both cases, we collect memory access time and application's execution time while one core executes the benchmark and others execute a stressing benchmark over the same DRAM controller. At the lower level represented in figure 15, the impact of interferences between a single-core and a quad-core configuration rises to a factor of three.

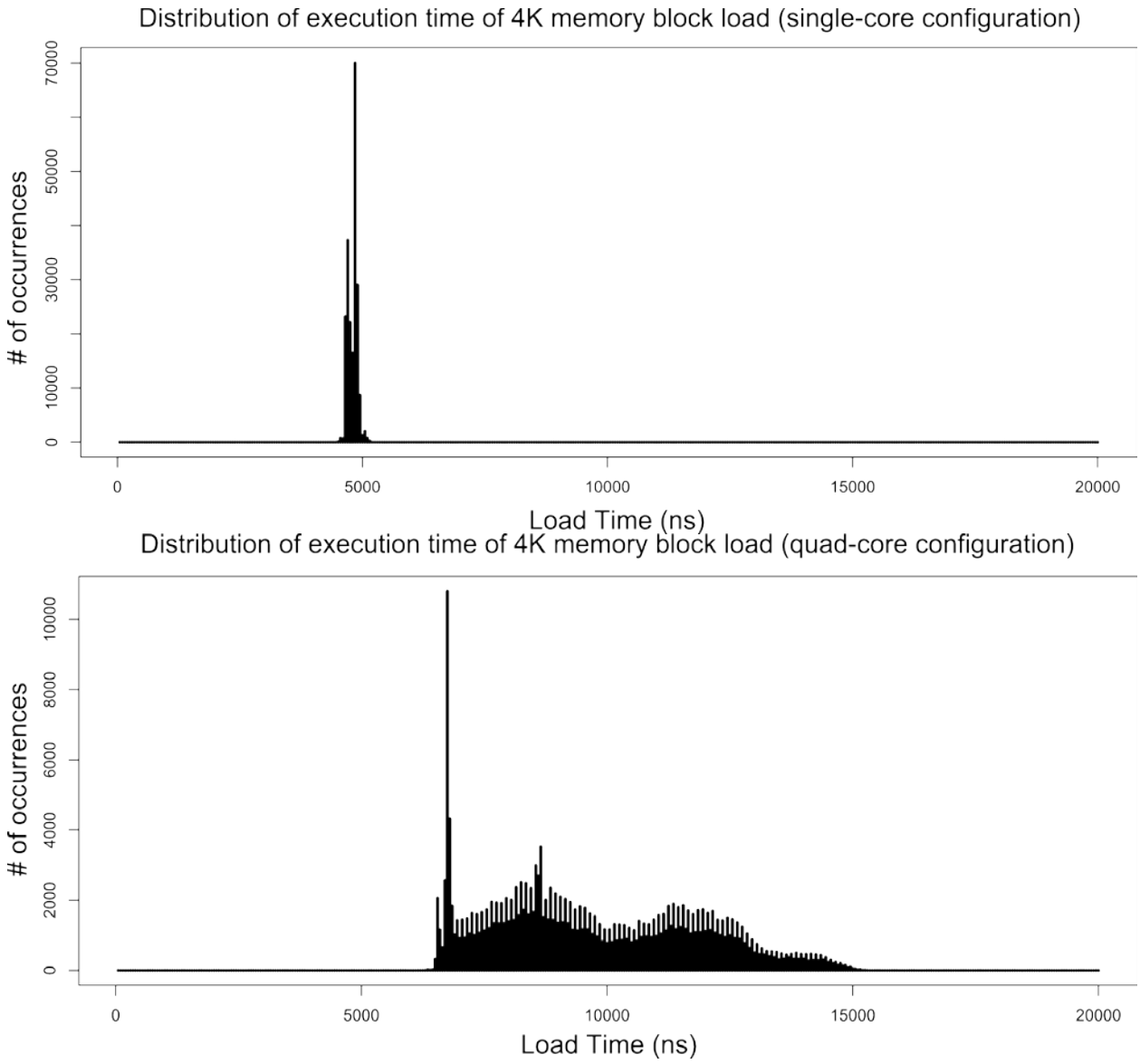

Figure 15. Example of impact of interferences on 4K memory load operations (1 vs. 4 cores)

As represented in figure 16, within the same time, interferences impact the execution of the application with a factor close to four (i.e., the number of cores). In the literature, even worse situations have been observed. For instance, Nowotsch, et al. [22] have observed a slow-down of the processor's activity with a factor of 20 on a P4080, which is an octo-core processor in the same series as the P5040.

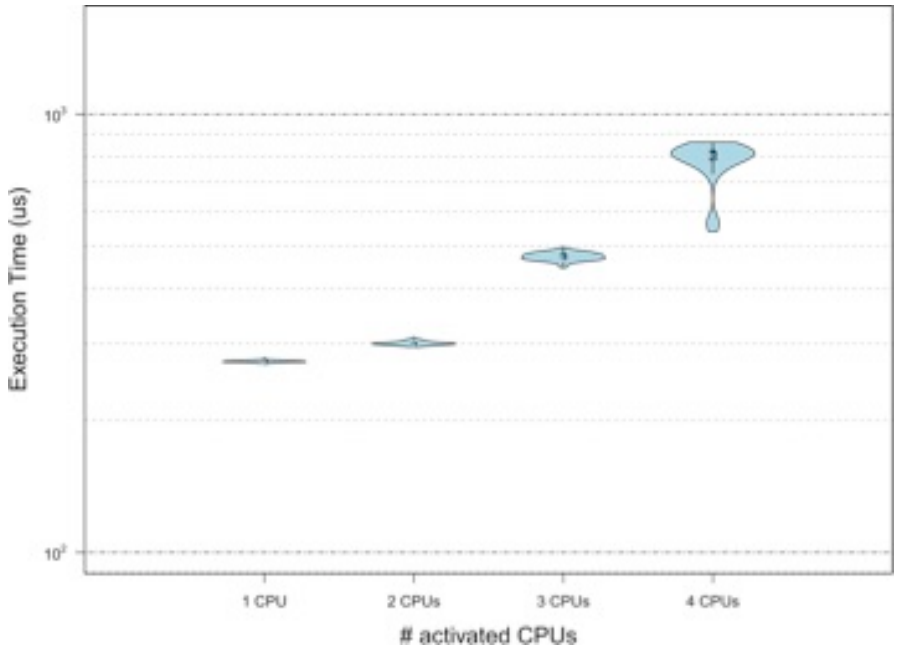

Figure 16. Impact of interferences on a piece of application vs. number of active cores

These experiments show that interferences are phenomena that significantly impact the embedded software's performances and make the whole equipment prone to timing failures. Therefore, their impact must be assessed with trusted methods and when this impact is too significant or unbounded, the appropriate mitigation has to be implemented in close relationship with the safety analyses performed at a higher level.

6.3 EFFECTS OF INTERCONNECT FAILURES

The functional failures associated with transaction services could lead to the faulty execution of the embedded software without raising any error. The associated analysis to determine the probability that these types of faults do not occur within the embedded system is called the interconnect integrity analysis, which is part of the mitigation by determination of interconnect UD (see table 4).

In cases of undetected internal failure, the interconnect may propagate an error to the targeted core and potentially to an external monitor. The interconnect integrity is a mitigation against the propagation of such errors. In this case, the platform-level reliability may benefit from the interconnect integrity by sanctioning the target core for the transaction whose corruption was stopped from propagating.

Nowotsch, et al. [22] performed an FMEA on a NoC that informed on the local effects and global effects of over 100 unique errors. This reference provides full coverage of single failure assessment and gives a deep insight into the various failure modes and their impact on task isolation.

The local effects include:

- Masked error: The subsequent block may ignore the error if it is only evaluated when other signals have a specific value
- Corrupted flow control unit
- Lost flow control unit
- Erroneous sending of flow control unit: the flow control unit is sent to the wrong port
- Delayed flow control unit transmission
- Blocked version control buffer

The global effects (system-level) include:

- Violation of QoS, which can be temporary or permanent
- Loss of packet, including the condition wherein the packet is delivered to the wrong recipient
- Corruption of packet
- Corruption of return route, which is relevant when acknowledgement is expected
- Blockage of version control buffer

7. MITIGATION MEANS

The determination of mitigation means is the third step of the safety approach. This section proposes several non-exclusive views of mitigation means and mitigation techniques. For more details, specific examples are provided in section 7.1 for implementations of these mitigation techniques.

7.1 MITIGATION TECHNIQUES BY MCP FEATURES

The list of MCP features that may impact the design in safety-critical systems include [14]:

- Variability of execution time
- Conflicts among services and/or transactions
- Cores interconnect switch
- Cache architecture structure
- Shared services
- Shared data
- Inter-core interrupts
- Access to peripherals
- Programming languages

Table 4 proposes examples of mitigation means for each of the above listed features.

Table 4. Mitigation means for MCP features

MCP Feature	Mitigation Means
Variability of execution time	WCET strategy for assessment, measurement, and continuous monitoring
Service/transaction conflicts	Software-controlled scheduling of tasks or processes
Cores interconnect switch	Interconnect usage-domain definition
Cache architecture structure	MCP-related cache management and cache consistency verified by trusted and privileged software
Shared services	Similar to the implementation of the airborne API, services are offered via a trusted and privileged software.
Shared data	Application of design rules. For example, all data that are not explicitly shared among cores are automatically tagged as private and duplicated in the cores.
Inter-core interrupts	Interrupts are accepted only when expected (wait-for-interrupt is a rule-based implementation) or restrictions are placed on the use of inter-processors interrupts.
Access to peripherals	Allocation of shared I/O configuration or shared memory space via trusted and privileged software. This allocation can be done either directly or via configuration-controlled configuration tables.
Programming languages	Determination of an adequate strategy for multiprocessing programming, such as preemptive versus cooperative strategy

WCET = worst case execution time
MCP = multicore processor
API = application program interface
I/O = input/output

7.2 ONLINE ERROR DETECTION, RECOVERY, AND REPAIR SCHEMES

Reference 23 investigates the deployment of online error detection, recovery, and repair schemes that can be applied against hardware faults and design errors. In MCP architectures, memories spanning a large portion of the die are successfully protected using ECCs. The focus of online error detection is thus the other elements: the cores, memory hierarchy control logic (i.e., memory consistency), and the interconnect logic.

The consumer electronic market primarily drives these smaller-size devices for which traditional redundancy-based solutions are unacceptable; these solutions carry a cost and penalties in area, power, and performance not compatible with the market. In lower-cost solutions, high-level symptom detectors focus on hardware faults propagating to the higher levels of the system [23]. These techniques assume checkpoint/rollback support to achieve recovery and mostly focus on transient faults. The reliability solution therefore consists of the combination of detection and a recovery method. Permanent faults, meanwhile, require a diagnosis mechanism to isolate the core affected by the fault so that it can be repaired and the system reconfigured.

Applicable online error detection techniques as described below can be classified as shown in figure 17:

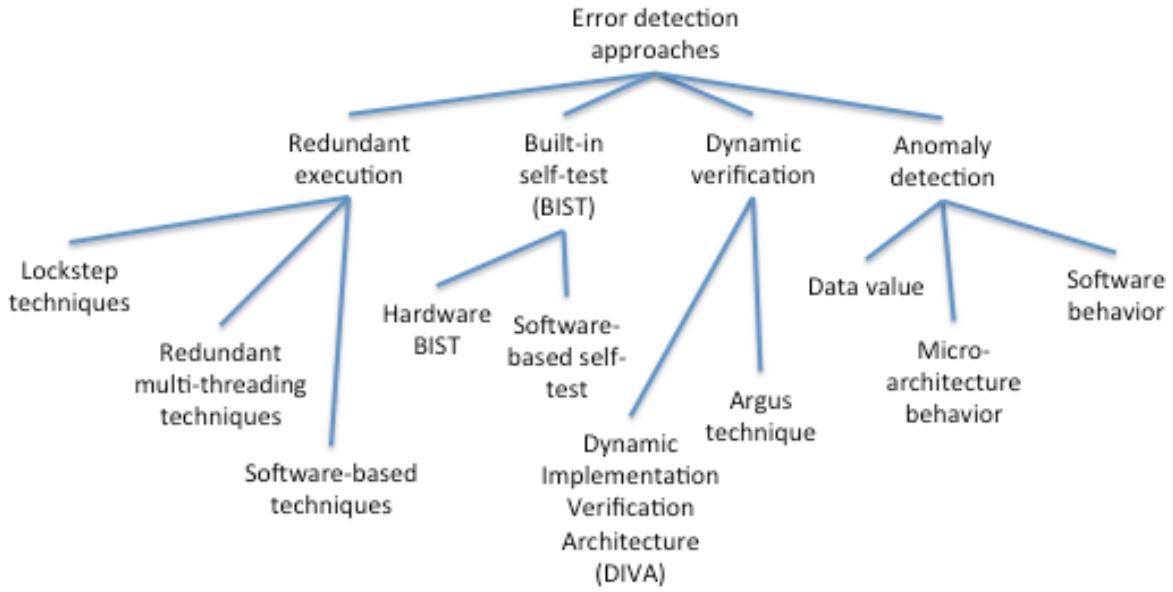

Figure 17. Topology of online error detection techniques [23]

- Redundant execution approaches: These techniques use the inherent replication of cores and threads in MCPs.
- Periodic built-in self-test (BIST) approaches: These techniques leverage the built-in test mechanisms traditionally used for the detection of manufacturing defects. They offer historically poor coverage for in-service failure (which is key in the context of process shrinking) and do not protect against soft errors.
- Dynamic verification approaches (also known as online testing): These techniques check that system-invariants are maintained independently from the implementation.
- Anomaly detection approaches

Periodic functional testing uses re-execution to compare the output produced in the same hardware at different times (e.g., temporal redundancy techniques). Soft errors are detected, but hard errors are left undetected in spite of re-execution. Detection of hard errors is based on re-execution on different hardware (e.g., spatial redundancy techniques), which can be costly in terms of additional area, power, and execution time. Finally, these techniques do not provide containment of hard errors, so as to apply hardware reconfiguration [24].

Online error recovery techniques can be classified into two broad categories:

- Forward error recovery: These techniques detect and correct errors without requiring a rollback/recovery mechanism. They must use redundancy (e.g., triple modular redundancy lockstep).

- Backward error recovery: These techniques are based on periodically saving the system state (checkpointing) and rolling back to the latest validated checkpoint following the detection of a failure.

Online error repair techniques typically leverage spatial or temporal redundancy to deactivate and isolate the component affected by the fault. The efficiency of these techniques depends on the coordination between the fault diagnosis and the isolation scheme (fault recovery) at the circuit or architectural level. SoftWare Anomaly Treatment (SWAT) is the comprehensive solution most cited in the literature survey. SWAT is a low-cost solution addressing hardware faults and consists of fault detection, fault diagnosis for multicore and single-core levels, and the fault recovery method.

Gizopoulos, et al. [25] goes one step further by investigating the management of homogeneous MCPs that accidentally become heterogeneous because of defects. Techniques such as core salvaging allow for recovering by exploiting cross-core redundancy and migrating offending threads to another core. This technique avoids disabling an entire core. The core cannibalization architecture allows one core to borrow resources from another core at the pipeline level to maintain a high number of operational cores. However, this technique puts more strain on the interconnect that supports the borrowing.

DeOrio, et al. [26] discusses the recovery of data after a core is disconnected following the recovery at the interconnect level (NoC architectures). This aspect is seldom investigated.

7.3 MITIGATIONS OF INTERFERENCES

Interferences mitigation comes at the end of the process and applies to interference channels. As an extension, we consider that mitigation techniques can also be injected in the initial definition of the UD, before performing the interference analysis, with the goal of making the analysis simpler.

As represented on the interference-aware safety process in figure 4, the mitigation can be internal and, thus, be reported as additional restrictions within the UD and/or specific configuration tradeoffs; or the mitigation can be external (i.e., hosted by a third-party hardware).

The case of an internal mitigation is the richest, as it implies new iterations with the interference analysis. In the first iteration, an internal mitigation will take care of the specificities of the hardware and software architectures in order to influence the definition of UD. Additionally, it may orient the equipment's development toward one of the following paradigms:

- Apply interference reduction techniques, use specific hardware features (if any), but do not try to bound interferences with software.
- Ensure that interferences are bounded within the interference channel by internal means.
- Ensure that interferences are eliminated by construction.

Each of the three paradigms above has its advantages and drawbacks. Selecting one of them has a strong impact on the rest of the equipment's lifespan, as it is significantly complex to recover from such a choice.

The case of external mitigation is more straightforward. Indeed, an external mitigation for an interference channel consists of a detection means and a sanction strategy, in accordance with safety objectives. Detection is the bottleneck of an external mitigation strategy, and detecting that an interference channel has "too much" interferences is not well-defined. A proactive detection strategy can be proposed and simply deployed. For instance, hardware events can be monitored at runtime, and unexpected events or UD violations can entail errors that are handled by health monitors, either internal or external. However, it becomes difficult to offer a rationale about the coverage of situations with unintended interferences.

Another approach consists of addressing detection means at a higher level, for instance by monitoring deadline misses. However, it becomes questionable to assimilate this detection as a consequence of interferences.

Finally, a long-term mitigation could also consist of collecting statistics on the processor's behavior both at the hardware and software levels. These statistics feed a database maintained by the equipment provider, which can reuse such information in projects for other equipment. Hence, unintended events, such as deadline misses, could be documented with a precise snapshot of the hardware and software conditions in which the problem occurred, so that the equipment provider could investigate.

No external mitigation strategy dealing with interferences seems to fit optimally. Interference mitigation should thus combine several detection means and propose adapted sanctions, in accordance with the safety objectives.

7.4 FAULT DETECTION, ISOLATION, AND RECONFIGURATION FOR FAIL-OPERATIONAL SYSTEMS

Process shrinking causes devices to degrade faster. The strategy at system level is therefore to build reliable systems on top of less reliable components because the probability is likely that the devices will degrade or even fail during the lifetime of the system. MCPs can be seen as an approach to resiliency in addition to being a solution to performance requirements. For MCPs to support fail-operational systems, the following mechanisms must be devised [23]:

- Detect or anticipate errors to prevent data corruption. Error anticipation may be implemented by monitoring components that are expected to fail or by annunciating an error immediately after the failure. Then, the error must be contained to the smallest possible component.
- Estimate the degradation of the different components and, in particular, the context of MCPs, the degradation of the different cores. The strategy is to activate error mitigation in highly degraded cores and reconfigure to use non-degraded cores for critical tasks.

- Reconfigure multicore to support fail-operational at system level. Reconfiguring the MCPs includes setting the operating parameters for each core, disabling the faulty core(s), and reallocating the core usage.
- Mitigate the degradation to maintain high performance over the lifetime of the processor. This involves mitigating causal factors, including electromigration, negative-bias temperature instability, and time-dependent dielectric breakdown (common with single core).

8. SELECTED EXAMPLES

The following subsections detail specific information using examples. Because of the different points of view that can be adopted, the techniques may repeat across different examples.

8.1 METHODS FOR INTERFERENCE ANALYSIS

The interference analysis is mainly focused on assessing the performance of the platform to leverage potential timing failures. Many different situations can be encountered with respect to the equipment's needs, depth of knowledge of the processor's architecture, and openness of the system (e.g., with an IMA). No generic method seems relevant; instead, a combination of several methods, more or less formalized, can be proposed with a case-by-case evaluation for each piece of equipment.

Some methods may be formalized or semi-formalized (i.e., relying on handwritten abstractions). Others may rely on engineering sense and involve experts' experience with MCPs, for instance on lower-criticality equipment or with processors from the same series. In all cases, the support from the MCP manufacturer is beneficial to the quality of the interference analysis.

This section details examples of methods that can be used within an interference analysis. These methods are formalized or semi-formalized.

8.1.1 Analysis of Processors Containing Black-Box Components

Many COTS processors embed black-box components, especially when these components bring competitiveness advantages. This is usually the case for interconnects and IPs for I/O control. Sometimes these components embed microcontrollers and execute microcode that is also proprietary. Thus their behavior is highly complex, and exhaustiveness of testing cannot be reached but only approximated. Termination of the analysis should then follow predefined termination criteria.

In this section, we propose examples of such termination criteria on a method that targets processors embedding black-box components.

8.1.1.1 Enumeration of Interference Paths

As mentioned earlier, the first challenge of an interference analysis is to reduce the set of interference paths down to one that is analyzable with an affordable complexity.

The initiator-target model [21] provides (and demonstrates) a formula to enumerate test classes on a processor containing black-box components. The processor is composed of initiators (e.g., CPU, DMA engines, master I/O, and targets [e.g., memories and slave I/O]). The notion of test class, as defined in [21], copes with the definition of interferences channels. Thus it is possible to apply the enumeration rules developed in the referenced paper.

The authors distinguish smart initiators (containing a memory and thus capable of performing master-to-slave communications) from non-smart initiators (with no built-in memory, thus capable of initiating slave-to-slave communication). For example, DMA engines are non-smart initiators. Considering these classes of initiators and the number of test classes, the number of interferences channels is given by:

$$\sum_{k_{si}=0}^{n_{si}} \binom{n_{si}}{k_{si}} n_t^{k_{si}} \cdot \sum_{k_{nsi}=0}^{n_{nsi}} \binom{n_{nsi}}{k_{nsi}} (n_t^2)^{k_{nsi}} - 1 = (1+n_t)^{n_{si}} \cdot (1+n_t^2)^{n_{nsi}} - 1 \qquad (1)$$

With these notations, n_{si} refers to the number of smart initiators, n_{nsi} refers to the number of non-smart initiators, and n_t refers to the number of targets. Table 5 illustrates the exponential growth of the number of interference channels for given processor configurations.

Table 5. Example of interference channel numbering

Smart initiators		Non-smart initiators	Targets		
Number of Cores	Master I/O	Number of DMA	Slave I/O	Number of memories (DDR, flash)	Number of interference paths
2	0	1	2 PCIe	2 DDR	424
2	0	1	2 PCIe + 1 CAN	2 DDR	935
2	1 PCIe	1	2 PCIe + 1 CAN	2 DDR	5615
4	0	1	2 PCIe	2 DDR	10,624
4	0	1	2 PCIe + 1 CAN	2 DDR	33,695
4	1 PCIe	2	2 PCIe + 1 CAN	2 DDR	5,256,575
8	0	1	2 PCIe	2 DDR	6,640,624
8	0	1	4 PCIe + 1 CAN	2 DDR	838,860,799
8	1 PCIe	2	4 PCIe + 1 CAN	2 DDR	335,544,319,999
12	0	1	2 PCIe	3 DDR	56,596,340,735
12	0	1	4 PCIe + 1 CAN	3 DDR	18,357,919,871,264
12	1 PCIe	2	4 PCIe + 1 CAN	3 DDR	10,739,383,124,690,000

CAN = controller area network

Except for very simple architectures and UDs, the sheer number of interference paths makes an exhaustive coverage impossible to reach (remember that a second analysis has to be performed on each interference path).

We can first note that the UD usually already restricts the allocation of shared resources to initiators. For instance, the platform can be configured to balance the workload over the shared resources. The values given in table 5 represent upper bounds; nonetheless, the set of interference paths still grows exponentially.

The following rationale can be employed to justify that an interference path is not tested:

- Partial ordering: A larger interference path already covers the interference path. This rationale works as long as no mode change occurs between the smaller and the larger interference paths (e.g., dynamic voltage and frequency scaling).
- Equivalence class: From the interference sources' point of view, the path is equivalent to another interference path. To make such an argument, architectural patterns can be exploited such as the symmetry of the processor's architecture and, more generally, any information stating that an initiator is treated by the hardware in the same way as another.

The last type of rationale may be used to build classes of interference paths for which the interference analysis would perform a more in-depth exploration of an item in each class; in other words, the interference analysis would investigate a representative of each equivalence class at each level within the predefined order.

8.1.1.2 Analysis of an Interference Path

The second challenge of the interference analysis is to get a correct coverage of the possible interference situations within a given interference channel.

The first point to note is that the state space is significantly large. Examples of parameters that can be explored include:

- Types of operations available for each initiator (e.g., read/write with different width)
- Length of transactional sequences and stress levels that can be reached by the embedded software
- Memory area targeted (in case a transaction targets a memory)

These parameters allow for defining metrics on test scenarios (i.e., by how much several scenarios differ from each other). The whole state space cannot be covered, so some situations will remain unexplored. As long as the processor's behavior is continuous, exploring close configurations can be sufficient. However, one issue related to processors that embed black-box components is the difficulty to predict mode changes (i.e., discontinuities in the processor's behavior due to specific inputs).

The intuition behind the evaluation of an interference channel is to observe the presence or absence of singularities (e.g., the explosion of transactions' propagation time outside of acceptable ranges). When a singularity is discovered, the interference channel is tagged as unbounded. Otherwise, the interference channel can be tagged as bounded.

This analysis makes the following assumptions on the processor:

- The processor's behavior is piecewise-continuous (i.e., adjacent tests lead to close interferences impact, except when a mode change leads to discontinuity).
- A limit exists on test metrics for which, concerning any pair of tests beyond this limit, at most one discontinuity is possible on the processor.

This method guarantees that no singularity is found while the test space is covered through a mesh with granularity that is lower than a given limit. In other words, the equipment provider defines metrics and mesh for the test space (i.e., the parameters that can be varied), where the mesh's granularity is predicated on the selected metrics. The test results would then state that no singularity has been observed over a mesh of granularity 'X.' The equipment provider proposes such a limit, which can vary per the criticality of the equipment. This limit sizes the number of tests to be performed and, thus, the complexity and cost of the interference analysis. The assumption that the microprocessor's interference level is piecewise continuous can be used to state that such an evaluation of interference over a specified mesh provides evidence; however, the demonstration is not complete. This approach allows for providing an answer tied to a non-exhaustive test campaign that is identified as such.

Finding singularities—that is, determining whether the interference channel is bounded or not—is one objective of this step. Another goal is to propose a trustworthy interference penalty derived from measurements. One of the problems is to deal with the imperfection of measurement methods, especially those using instrumented code. First, these measurement methods are biased by the intrinsic timing variability of the processor's behavior. For instance, several clock domains that are not synchronized can be encountered and introduced a random synchronization delay. Second, measurements are in many cases end-to-end. For read operations, this is a go-and-back-again operation. For write operations, the end-to-end measurement is more straightforward but may be harder to collect. For instance, it is hard to get the arrival date of a write transaction within a memory. Finally, measurements by code instrumentation may limit the range of parameters that can be explored. For instance, non-instrumented code may reach a higher level of stress on an interference channel than instrumented code.

As a conclusion, a trustworthy analysis of interference paths on processors containing black-box components may be possible using a systematic enumeration method, such as initiator-target, when:

- The UD is sufficiently restrictive on use-cases so that the number of interference paths remains manageable. Eventually, classes of interference paths may be built so that the analysis is performed on the whole classes.
- Granularity of evaluation has been established and is sufficiently high with regard to the safety objectives.

8.1.2 Methods for Global Interference Penalties

Approaches dealing with global interference penalties aim at jointly considering the task-set and platform during the interference analysis. These techniques can be applied to the equipment for which one stakeholder has visibility on both the platform and the embedded software. They rely on an in-depth analysis of each task to rebuild a detailed view of its use of shared resources. An interference penalty applied to each task can then be computed from that view.

The following methods have been proposed:

- Interference Sensitive WCET proposed by Nowotsch, et al. [27]: The authors use aiT, a static analyzer developed by AbsInt. They consider a given number of time windows, and for each one, they explore all execution paths of each task and compute a bound on shared resources usage during that time window. The global interference penalty is obtained by adding interference penalties computed for each time window.
- Application's signature proposed by Bin, et al. [28, 29]: The authors associate a signature to each component of the processor (i.e., an evaluation of the component's response or traversal time depending on its workload). The signature of a task is defined as the workload generated by this task on each component (e.g., interconnect, caches, or memory). It is possible to combine the signature of several tasks and compare them with the signature of components to compute a global interference penalty.
- A computational method proposed by Pellizzoni, et al. [30]: The method uses network calculus theory. The authors compute arrival curves that correspond to the workload sent by each task to shared resources. They define a service model for the processor and a set of equations that determine delays in arrival curves. By adding these delays, they obtain an interference penalty.

Global interference penalties have the advantage of being able to perform an end-to-end analysis of the interferences' impact. Their use in closed equipment (i.e., wherein the whole software is known) would be beneficial to the quality of the interference analysis. The main drawback is the difficulty in applying them early in the safe design process.

8.1.3 Local Analyses

A couple of techniques have been developed to perform fine-grain analysis on specific components that are the usual sources of interferences. These techniques mainly focus on interconnects and shared caches. As they are applied locally, they must be combined with other methods to constitute the interference analysis.

8.1.3.1 Shared Cache Analysis

Cache analyses techniques for shared caches extend existing methods developed for private caches and rely on static analyzers.

Historical methods consider the control flow graph of a software task and simulate the cache content during the execution for all possible executions [31]. A reference (instruction or data) may be classified as "always miss," "always hit," "first miss," or "unknown," depending on its presence in the cache with respect to the past executions of the sole task. Cache analyses have the following limitations:

- Data caches are hard to analyze because data references are often determined at runtime and vary among software execution. Approaches based on value analysis using abstract interpretation techniques have been proposed [32], and WCET evaluation tools, such as aiT, provide good support for data cache analyses.
- Cache analyses only support well-defined policies, such as LRU or PLRU [18]. Caches embedded on COTS processors sometimes have proprietary replacement policies—for example, streaming PLRU for Freescale processors.

For shared caches, the difficulty is to take into account conflicts between concurrent tasks without being too pessimistic. Yan and Zhang [33] represent such conflicts by annotating "always hit" and "always miss" states with a tag "except one," meaning that a concurrent task will alter this reference only once during its execution. This applies to structures such as loops that may conflict with sequential code. However, this approach still remains pessimistic. To improve on it, Hardy, et al. [20] propose a concept named "bypass." Intuitively, the authors state that many conflicts in shared cache are due to evictions from lower-level caches. A significant part of these evictions deals with cache lines that are used only once and thus have become useless. Hardy's approach consists in identifying such lines, also called "static single-usage blocks," and forcing the hardware to bypass the shared cache. Thus conflicts within the shared cache are significantly reduced.

The problem with shared cache analysis is that it presents solutions only in specific cases—for instance, considering only data or instruction-shared caches [34]. However, in general, especially with unified (data and instructions) caches, it remains an active research field.

8.1.3.2 Memory Analysis

The problem of memory's response time can be raised both for SRAM and DRAM technologies, with both of them being supported by COTS processors (SRAM technologies are used in most caches). Nevertheless, on SRAM, the problem is considered to be solved except for highly complex SRAM controllers [35].

Drepper provided an overview of a DRAM system (controller and memory) [36]. Paolieri, et al. [37] studied DRAM analysis. In this paper, the authors model an entire synchronous DRAM, including the controller and memory system. They provide static upper bounds for read and write operations. This approach yields promising results, but its limitation lays in its applicability to COTS processors that often have proprietary interleaving protocols.

8.1.3.3 Interconnects Analysis

Interconnects analysis have tackled the following problems:

- Worst-case behaviors of interconnects arbitration policies
- Worst-case situations regarding the interconnect protocol

Regarding the first problem, Bourgade, et al. proposed a study of several policies [38]. The authors highlighted the case of some protocols not giving bounds on the arbitration time. This is

the case, for instance, for fixed-priority and some classes of FIFO protocols. Like other families of local analyses, the limitations of this study is the lack of knowledge of COTS interconnects' arbitration protocols. Nevertheless, some interconnects IP, such as CoreLink™ from ARM® [39] and TeraNet™ from Texas Instruments [40], have their protocol and arbitration policies documented.

Regarding the second problem, corner cases with interconnect protocols have been studied by Shah, et al. on the AMBA™ protocol, which is widespread in ARM processors [41]. They have shown that bus-locking phenomena may slow down successive transactions during arbitration phases. This result also covers interference situations.

8.2 EXAMPLES OF INTERNAL MITIGATIONS FOR INTERFERENCE

In this section, state-of-the-art solutions are developed that can be used to provide internal mitigation for interferences.

Internal mitigations consisting of restrictions on the UD are enticing. For instance, it can be stated that some targets cannot be used by more than a given number of initiators or cannot be used beyond a given rate; the challenge is then to enforce such a restricted UD. Several techniques detailed in this section define mechanisms to implement such restrictions.

Additionally, several surveys, referenced in table 6, have been proposed in the literature.

Table 6. Surveys of interference mitigation techniques

Ref.	Mitigations covered	Comment
42	Interference reduction Interference bounding Interference elimination	This survey is more general than mitigation of interferences, as it also covers global methods for interference analysis.
43	Interference bounding Interference elimination	This survey only covers software implementations on COTS MCPs. The solutions presented in the survey are grouped under the label "deterministic platform software" (i.e., software components that do not bring new functionalities to payload software [e.g., applications] but provide interference mitigation).
44	Interference reduction Interference bounding Interference elimination	This survey could not be downloaded.

COTS = commercial off-the-shelf
MCP = multicore processor

The following sections (see sections 8.2.1–8.2.3) respectively detail:

- Techniques for interference reduction that do not remove or bound interferences by themselves but enhance the bounds on interferences when they exist
- Techniques for interference bounding: These techniques are implemented in regulation software solutions. The main idea consists of granting budgets that are periodically refilled (e.g., execution time, use of shared resources) for embedded software; allowed to execute until they have consumed their budget.
- Techniques for interference elimination: These techniques implement a control paradigm, which means that the use of shared resources is controlled by an external software according to a global policy.

Table 7 describes the classes of interference channels that can be targeted by the different interference mitigation techniques.

Table 7. Classes of interference mitigation techniques and targeted interference channels

Interference mitigation techniques	Targeted interference channels
Interference reduction: Interference reduction techniques will apply on interference channels for which the assessed level of interference is bounded, but too high to meet specified performances. They cannot bind an unbounded interference channel.	Unacceptable, bounded
Interference bounding: Interference bounding techniques will leave the interference channel active, but the impact of interferences on software performances will be bounded. They are applicable for unbounded interference channels for which no failure mode has been triggered.	Unacceptable, unbounded
Interference elimination: Interference elimination techniques are the most efficient ones regarding determinism aspects (but not necessarily regarding performances). They can be used to disable a given interference channel. Therefore, they are applicable for all kinds of interference channels, even those triggering failure modes.	Unacceptable, unbounded, faulty

8.2.1 Interference Reduction Techniques

Interference reduction techniques discussed in the literature apply to sources of interferences. They highlight specific mechanisms at the hardware or software level.

More specifically, solutions that target the following sources of interference are detailed:

- Shared caches: They have been identified for a long time as bottlenecks for performance assessment in MCPs. Hence, several solutions are proposed to reduce interferences within shared caches.
- Memory banks: The problem with memory banks is close to that for shared caches, even if their impact on performance is lower.
- Kernel locks: An issue in legacy RTOS migration to MCPs is to make independent operations, especially system calls, the least dependent on same lock as possible. That may require deep modifications to the OS design.

8.2.1.1 Cache Coloring

A cache management concept called “Colored Lockdown” has been proposed to reduce cache conflicts by coloring techniques [45, 46]. The technique aims at making a shared cache behave like a private one. Interferences are then addressed by the cache paradigm.

Set-associative caches are often represented as arrays (see figure 18), whose cells contain blocks, usually 64 bytes wide. No data can be located anywhere in the cache. As illustrated in figure 19, its address is split into three segments. The “set” segment determines in which row a piece of data is located. The “way” or column, is chosen by the cache replacement policy. When a cache block is selected to be evicted so that a new block can be loaded, it is within the same set.

Cache coloring aims at reducing interferences in caches by ensuring that data and instructions used by software running on different cores will be stored in different sets. Allocation and/or mapping of these data is complex and may require specific allocators; in [44], the page management system of Linux has been modified to do so. When a cache is properly colored, interferences are almost removed: Contents are independent, but access is not. Also, the equivalent private caches are efficient, as they keep a high associativity level.

In another approach, Ward, et al. have assigned colors to tasks with an underlying support for cache coloration, and alter the scheduler to minimize cache conflicts while keeping some dynamism within their system [47]. Several independent tasks may be maintained within the system while their conflicts are reduced over their execution.

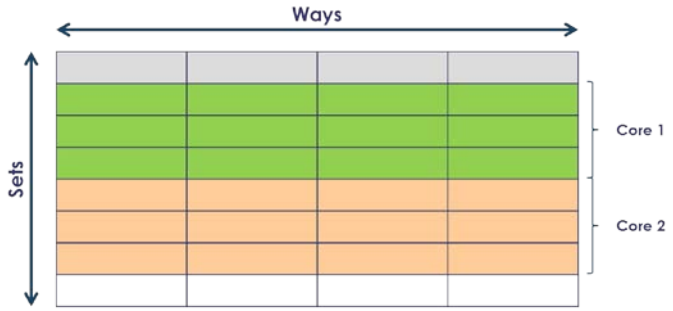

Figure 18. Example of colored cache shared between two cores

Figure 19. Address sections from the cache's point of view

8.2.1.2 Cache Partitioning

Cache partitioning is a more straightforward way, compared with cache coloring, of simulating private caches from a shared cache. As represented in figure 20, cache partitioning consists of restricting some ways to given address ranges, usually used by software running on specific cores. This method is supported by hardware on several processors, for instance on the Freescale QorIQ series. It is simple to deploy because it is just a matter of static configuration.

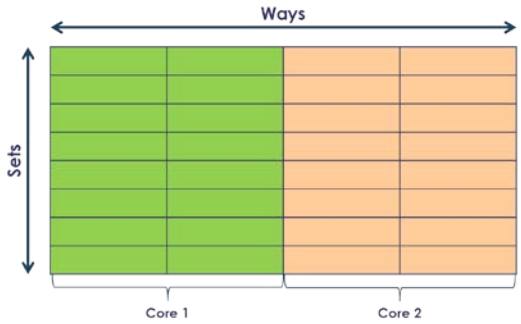

Figure 20. Illustration of cache partitioning

The drawback of cache partitioning is the lower efficiency of the equivalent private caches, as they have fewer ways.

8.2.1.3 Bank-Aware Memory Allocation

A solution called PALLOC has been developed within the "single-core equivalent virtual machines" framework to reduce interferences in open bank tables that are maintained by DRAM controllers [48]. This table was already classified within the cache paradigm; interferences linked to memory bank conflicts may have a significant impact [49].

Like Colored Lockdown, PALLOC is a dynamic memory allocator that is integrated within Linux. It is called when the kernel allocates memory, for instance, to copy process binaries to the main memory from persistent storage.

8.2.1.4 Kernel Locks Refinement

Many RTOS supporting MCPs have inherited kernel locks functionality from the legacy support of single-core processors. During the migration to multicore, special care must be taken to limit the number of conflicts on kernel locks due to simultaneous calls of kernel services by different cores. This problem is also called “lock trashing” and is described for Linux by Kleen [50].

Solutions have been proposed to limit the effects of lock trashing. For instance, Cui, et al. proposed an implementation of a lockless scheduler for MCPs [51]. A common practice, called “lock refinement,” consists of multiplying locks for services that are independent. Two difficulties arise. First, it is hard to say on a legacy OS which services are actually independent from each other. Second, the number of locks is likely to significantly increase, making the system harder to analyze. Solutions have been proposed to manage locks, for instance by Zhang [52], but it currently still remains an active research field for RTOS manufacturers.

8.2.2 Bounded Interference Solutions

Several solutions have been introduced to ensure that interferences have a bounded impact on software execution time. The main idea is to monitor the execution of embedded software and perform regulation (i.e., reconfigure the scheduler to alleviate the workload when some tasks are consuming too much shared resources).

In practice, each task will be allocated a budget, usually a number of accesses, granting it the right to freely access shared resources. That budget is periodically refilled and sized to fit with the task’s needs. If a task exceeds its budget, a regulation mechanism detects the overflow and delays it until its budget is renewed.

From these restrictions, an interference penalty can be evaluated based on the allocated budget and applied to each software task.

8.2.2.1 MemGuard

The regulation paradigm has been implemented within MemGuard [53], which is part of the single-core equivalent virtual machines framework [46]. As represented in figure 21, each core hosts a bandwidth regulator that shares the total bandwidth guaranteed at the DRAM controller level (i.e., 1.2GB/s in the original paper [53] among each core). Regulation is performed with hardware performance counters hosted on each core. These counters can be configured to count each access to shared resources and trigger an exception when that number exceeds the allocated budget.

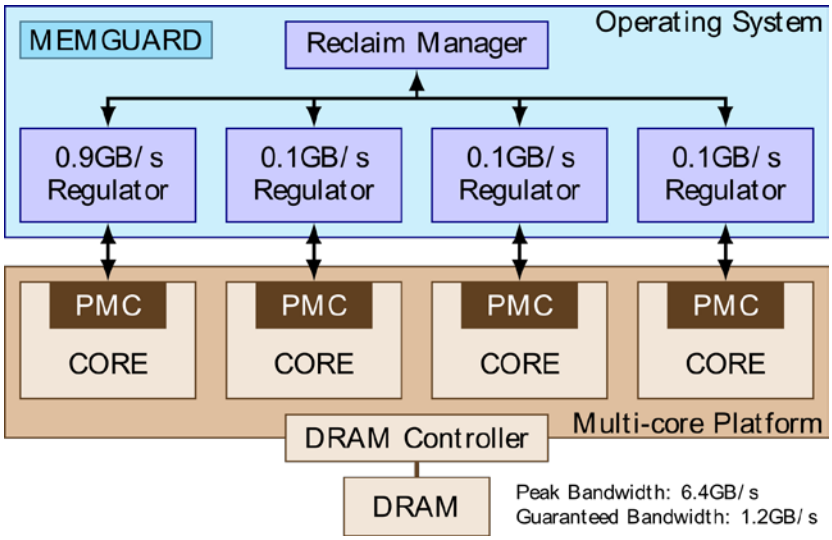

Figure 21. Overview of MemGuard mechanism

The authors have extended the MemGuard mechanism to improve the overall performance by allowing software to reclaim bandwidth that is unused by concurrent tasks. Even if this mechanism is not recommended for highly critical applications, it provides a significant performance improvement for equipment with a lower criticality level.

8.2.2.2 Mixed Critical Systems

A mixed-critical paradigm consists of co-running critical and non-critical software—the critical software must meet real-time criteria, while the non-critical does not and instead runs in a best effort way [54]. The objective related to interference is to ensure that low-criticality tasks do not interfere too much with highly critical ones.

A common view of mixed-critical systems consists of scheduling the whole task set in a nominal mode, wherein both critical and non-critical tasks coexist. When critical tasks have no choice but to use the entire processing time to meet their WCET, non-critical tasks are suspended. This kind of paradigm can be defended, but the problem of availability for non-critical tasks remains.

A less constrained framework has been proposed by Kiritikakou under the name of “distributed run-time WCET controller” [55]. Within this framework, tasks may be instrumented to monitor their own execution by inserting intermediate checkpoints. When a checkpoint is reached too late because of interferences (or other effects causing a slowdown), the WCET controller is notified and suspends the execution of non-critical tasks, as shown in figure 22. A critical task may also take the initiative to suspend non-critical tasks.

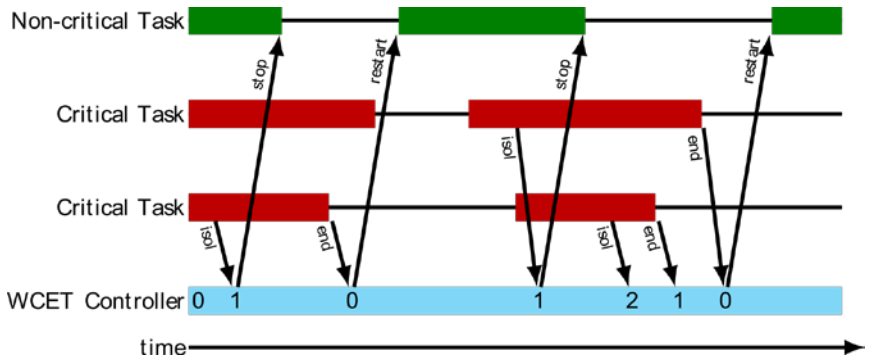

Figure 22. Overview of distributed WCET controller

Bounded interference solutions offer the advantage of being as less intrusive as possible. Therefore, their impact on software performance is minimized. They introduce guards on the use of shared resources by embedded software. However, they do not help in the analysis of the low-level behavior of the processor, which means that the number of interferences will be bounded, but their impact remains mandatory to assess.

8.2.3 Interference-Free Solutions

Interference-free solutions have been designed for equipment that has hard real-time constraints. They focus on the major sources of interferences, so that most interferences are actually eliminated. However, these solutions do not dispense with a proper interference analysis. For instance, slight jitters may remain relative to minor sources of interference; they must, nevertheless, be assessed.

Removal of interference on a buffer source is usually performed through timing isolation principles. These principles may be implemented within the scheduler (e.g., critical tasks have dedicated slots) or enforced between concurrent accesses from cores to shared resources. Historically, the second approach has been developed within dedicated hardware solutions [56–58]. However, these solutions have not, thus far, been embraced by COTS processors manufacturers. Thus the solutions presented in this section are only supported by software.

Interference-free solutions deployed on COTS processors are grouped in the control software paradigm [42].

8.2.3.1 Interference-Free Scheduling

Interference-free schedulers apply to systems that embed applications with heterogeneous levels of criticality. They run low-criticality applications in parallel while highly critical applications are executed alone on the processor, as represented in figure 23.

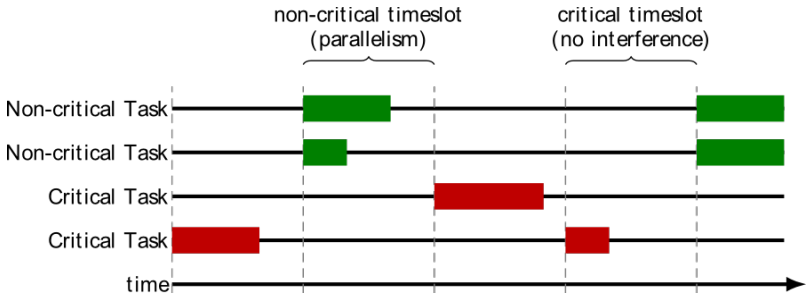

Figure 23. Example of interference-free schedule

This method has been published in [59–60]. Fundamentally, it does not solve the question of interferences on MCPs; rather, it proposes a workaround that fits with systems embedding little critical code.

8.2.3.2 Deterministic Execution Models

Deterministic execution models propose execution patterns that isolate phases where software initiates communications to shared resources, from phases where software executes from internal memories (e.g., private caches). Embedded software must be implemented according to these patterns.

When a piece of software performs an execution phase, it makes no use of shared resources and therefore cannot interfere with concurrent software. As illustrated in figure 24, communication and execution phases are scheduled, and communication phases are isolated according to time division multiple access (TDMA) principles.

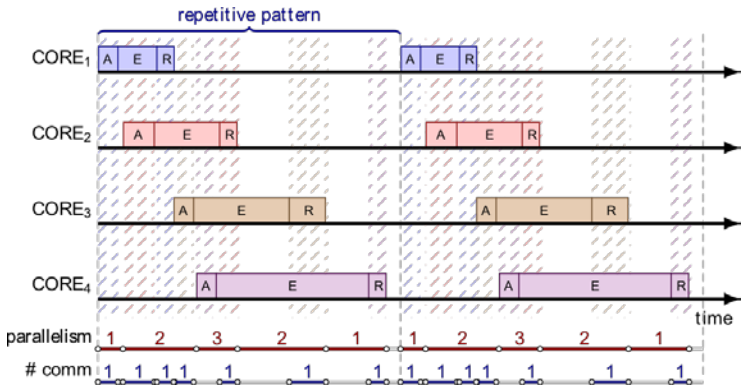

Figure 24. Example of deterministic execution model schedule [63]

The following execution models have been published in the literature:

- The Predictable Execution Model is proposed by Pellizzoni, et al. [61]. The authors introduce the notion of predictable intervals, which correspond to internal execution phases (i.e., without communication). This model scales to MCPs and takes into account interferences relative to the master I/O. The authors use an external device that interfaces with the I/O and can be programmed to cope with a TDMA policy and not interfere with the CPU.
- The Acquisition-Execution-Restitution pattern has been used to decouple memory and I/O communications [62–63]. The acquisition phase for a task consists of fetching data, instructions required for the execution phase, and fetching data from the I/O. During the restitution phase, the main concern is to update the I/O. An approach has been developed to position and optimally schedule execution phases and communication phases [2]. This approach consists of translating the set of tasks' parameters in constrained linear programming and letting a third-party solver find a schedule while maximizing the global response time.
- The Flush-Fetch-Execute is an execution model that has been developed to ease the deployment of code generated from synchronous execution models [64].

Timing analysis and the ability to schedule aspects of execution models have been developed by Schranzhofer, et al. [63, 65]. They showed that a deterministic execution model brings predictability to the execution of tasks on the MCP without impacting the performance. The main drawback is its difficulty in supporting legacy software.

8.2.3.3 Application Unaware Control Software

One drawback of deterministic execution models is the lack of support of legacy software, which introduces an additional effort for redesign. An approach has been proposed to make that control transparent with respect to embedded software [66–67]. This enables the reuse of legacy software, including RTOS, with minor porting effort. As represented in figure 25, the mechanism relies on a smart configuration of the CPU internal resources (e.g., MMU and caches) to make fetching operations transparent from the main memory.

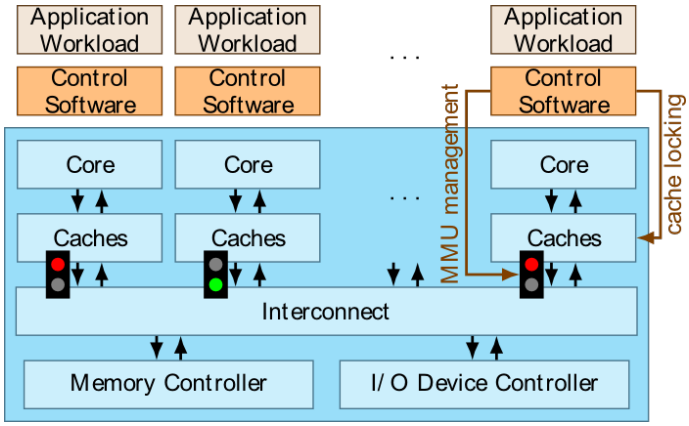

Figure 25. Overview of Marthy control software

This approach has been successfully deployed on a Freescale P5040. The prototype, called Marthy, has been included within the kernel of Topaz, which is a lightweight hypervisor developed by Freescale. However, Marthy's impact on software performance is variable. As described in [67], the slow-down is limited for some applications (e.g., a 30% overhead for a quad-core processor). For other applications, it may be a factor of 10 or even more. Even if optimizations are possible, the classes of applications for which this solution can be relevant are not known.

8.3 MITIGATION OF VARIABLE EXECUTION TIME BY WCET TECHNIQUES

The principal source of time non-determinism in MCPs is related to interferences as a causal factor to variability in execution time. This impact is the most significant in the context of safety-critical airborne software. The mitigation is based on the deployment of techniques to determine WCET.

WCET problems can be addressed using the safety process described in Section 2 because a deadline miss can be assimilated to a particular failure mode [68]. During the FHA, the failure condition associated with a deadline miss will be allocated a safety objective. The PSSA will determine safety mechanisms at the following levels:

8.3.1 At System Level

The safety mechanisms typically consist of redundancies and implementation of watchdogs for time elapsed. The timing constraints and the probabilities of failure are allocated to mechatronics elements (e.g., actuators, wires, and computers).

8.3.2 At Electronic Devices Level

The safety mechanisms primarily consist of timing surveillance. The probabilities of failure are allocated to I/O transmissions, PCIe communication, programmable logic device processing, and microprocessor processing.

8.3.3 At Microprocessor Level

Because the microprocessor is a COTS hardware component, the safety mechanisms can either be external to the microprocessor or implemented in the software. Additionally, the time constraints and probabilities of failure are constrained allocations because they can be adjusted only at software level. In the context of an MCP, the timing allocation needs to integrate a margin to account for interference; interference is also allocated a probability as a cause for the processor not to meet its deadline.

The safety assessment verifies the appropriate mitigation of deadline misses using the above safety mechanisms. The coverage rate of these mechanisms can be estimated through modeling and computation. Latency associated with software processing can be checked using WCET methods. Even if these methods do not provide a probabilistic WCET, it is nonetheless recommended to express it in terms of a probability of failure so that it can be readily integrated in classic safety analyses, such as fault-tree analyses. Reference 68 further indicates the applicability of the various WCET method classes to the safe design or safety assessment process based on the level of detail they expect. A synopsis is provided in Appendix B.

8.4 MITIGATION OF SINGLE EVENT IN MCP SCOREBOARD

Hari, et al. [69] proposed a recovery method for the scoreboard against an SEU that can either cause a dependent instruction to never be available (impossibility) or cause a dependent instruction to issue too early (untimely). The scoreboard consists of a table with as many rows as logical registers and as many columns as the maximum execution latency. A “1” in column *n* of row *r* indicates that register *r* will be available in the register file in *n* cycles from now. A “1” in column *0* indicates that the register is available in the register file. A “1” in column $n > 0$ indicates that the value is available in the proper bypass (if that bypass exists) or that the operand is not yet available.

In the first case of the dependent instruction to never be available, the implementation of a watchdog timer at the issue stage allows for the detection of the SEU effect (i.e., the processor being deadlocked) caused by a corrupted scoreboard. Once the problem is detected, the scoreboard can be safely reset because all previous instructions have been correctly executed and retired. Finally, the execution can resume.

In the second case of an untimely issue of the dependent instruction, a parity check is performed after the instruction issues. If the check is incorrect, the micro-operation is likely to have been issued prematurely. As no distinction regarding the severity of the error can be made by this method, the issue process is stopped—regardless of the effect of the error—so that the faulty instruction is contained within the issue queue. The pipeline is then drained, the scoreboard is reset, and correct execution can resume.

8.5 FAULT-TOLERANCE IN NETWORK-ON-CHIP SYSTEMS

Mitigation techniques implemented for fault-tolerance in NoCs systems include error detection and correction [70–72], stochastic communication techniques [73–74], adaptive routing, and various schemes combining spatial and temporal redundancy. All of these techniques apply at the lower level of the communication stack, so that their mitigation may not prevent the need for application restart at system level, which sets the system back to its initial state. The effect of such a restart at the system level may still be critical because of the time needed to reach the last valid state. For this reason, system-level fault-tolerant approaches investigate the rollback recovery concept. In the rollback concept, the system resumes execution after a detected failure from a consistent state (also called recovery line) that is further down the state path compared with the initial system state. This approach involves the saving of an intermediary state during error-free execution to build a consistent state. This process is called checkpointing [16].

8.6 PASSIVE REPLICATION FOR FAULT-TOLERANT SCHEDULING

Sun, et al. [75] investigated fault-tolerant scheduling based on scheduling copies of tasks on different cores. In particular, in primary backup task scheduling, two copies of a task are scheduled on two cores (i.e., the primary and the backup). The backup task is executed only if the primary fails; this technique is therefore a passive replication technique. To remain time-efficient, task overloading is implemented. The issue is that the management of the overloading scheme quickly becomes problematic based on the increased complexity of the tasking.

8.7 MITIGATION TECHNIQUES AT CORE LEVEL

8.7.1 Self-test

Concurrent autonomous chip self-test using stored test patterns (CASP) is a special type of self-test that is applicable over a wide change of MCPs ranging from microprocessors to networking chips and graphics processing processors. CASP's basic principles are to store specific and thorough test patterns in non-volatile storage media and to provide architectural and system-level support for MCP testing at the core level or over multiple cores [76]. This reference paper focuses the motivation to implement CASP on the reliability issues caused by aging-related failures. CASP is indicated for the implementation of circuit failure prediction [76–77] and self-test-based error detection and diagnosis.

Per Yi, et al. [76], the application of CASP to the OpenSPARC T1 octo-core processor consists of four phases:

1. Test scheduling: While the system operates normally, one or more cores may be selected by the CASP test controller to perform an online self-test.
2. Preprocessing: The core(s) selected for the self-test are isolated from the rest of the system, and execution on that core is stalled.
3. CASP testing: The CASP controller sets the signals needed for the test patterns and analysis of the core's response, loads the tests, performs the tests, and analyzes the answer for failure.
4. Return to normal operation by restoring communication and critical states

Note that other online self-testing approaches, such as BIST, roving emulation, and periodic functional testing, exist (see section 7.2). CASP was highlighted in this report because of its specific use for MCPs.

8.7.2 Architectural Mitigation

Classic fault-tolerant techniques can be applied to hardware in combination with redundancy. Ungsunan, et al. [78] proposes a combined hardware and software fault-tolerant approach using asymmetric cores to mitigate transient faults. In this architecture, all cores have the fault detection logic but differ in the amount of fault-tolerant hardware and resilient features to correct for the transient fault and soft errors. If a transient fault is detected at runtime, the affected core is reset. At the FHA level, the instantiation of this architecture requires the identification of processes as critical or non-critical as well as the identification of dependent processes (for containment).

A comparison of the MCP reliability using eight symmetric or asymmetric cores is performed in [78]. The result indicates that the system-level probability of failure is based on the architecture selection and the number of cores running critical processes. If the critical processes run on four or less of the cores, the asymmetric MCP is more reliable. If more than half of the cores are critical, the symmetric MCP has a higher reliability.

8.8 FAIL-SAFE AND FAIL-OPERATIONAL DESIGNS IN AUTOMOTIVE INDUSTRY

The automobile industry commonly uses MCPs, for example, in microcontrollers; the current dual-core designs are foreseen to rapidly evolve to quad-cores and octo-cores driven by higher processing power requirements. The cores are then likely to host functions with different automotive safety integrity levels (ASIL).

The approach proposed in [79] is to marry diverse redundancy in software to more traditional hardware redundancy in order to meet safety-critical requirements. The concept is called coded processing. The safety-critical parts, such as the control program, are specifically protected, fail-safe criteria are met, and safety verification is implemented.

The safety analysis applied to the problem is a Markov Chain in which the states correspond to the system's functional failures. Two types of errors are considered: 1) dangerous detected error (i.e., the system is in a dangerous state, but the failure diagnosis of the system detects the condition) and 2) dangerous undetected error. In a fail-safe system, the timely detection of dangerous failures results in the system entering a safe-state; the safe-state may cause the system to stop working. With the deployment of x-by-wire systems (e.g., full electric braking or steering), a purely mechanical fallback solution is no longer available, suppressing the countermeasure in the event of a detected error. At the state level, the system enters an error-state that can no longer be left for a safe-operation. Therefore, the system must be fail-operational.

The example provided in [79] is that of a system with two active channels running on an MCP where different ASIL are partitioned on different cores. When one channel fails, the service is continued based on the last valid result of the other channel. This state is a safe-state for the fail-safe system and an error-state for a fail-operational system.

Coded processing allows for increasing the percentage of dangerous states that are detected and is therefore an immense improvement to fail-safe systems. But for fail-operational systems, the improvement gained by coded processing is minor because deactivating the system is not an acceptable state.

8.9 PAIR AND SWAP TECHNIQUE FOR PROCESSOR'S DEGRADED MODE

Imai, et al. [80] proposed a processor-level fault tolerant technique called "Pair and Swap" to allow for graceful degradation in the presence of either permanent or transient faults. In the pair phase, two identical copies of a given task are executed on a pair of two cores. If the comparison of the output indicates no fault, each core repeats an execution and comparison cycle. If a fault is detected by mismatch, cores of the mismatched pair are swapped, and the test task is re-executed from the latest checkpoint. At the end of this "swap" phase, the fault is classified as "transient" or "permanent."

If the fault is permanent, the faulty core is readily identified and isolated, so that the MCP is reconfigured for operation in a degraded mode. If the fault is transient, the swapped pairs continue performing their tasking without reconfiguration. This solution offers a computational advantage over triple module redundancy, in which three cores are dynamically coupled.

9. RECOMMENDATIONS

9.1 IDENTIFIED GAPS IN EXISTING GUIDANCE

9.1.1 Context of Determinism

An additional definition for determinism should be added to the list to address determinism associated with reproducibility of behavior, as it is not correctly covered in the DO-297 definition above.

9.1.2 Limitations of Conventional Assurance Approach

Addressing the MCP issues highlighted in this report by conventionally applying DO-178B/C and DO-254 may be near impossible, in particular when aiming at verifying all potential states of execution; project constraints are incompatible with the required time and resources to conduct a strict bottom-up analysis. Moreover, the accumulation of design details, if possible, may be useless with regard to the questions addressed and impossible to exploit in the expected time to market.

9.2 RECOMMENDED APPROACH

In general, one of the questions to be answered in terms of supporting guidance is which one(s) to apply to MCP: DO-178C, DO-297, DO-254, DO-326, or a specific certification guidelines document for MCP. Two main trends exist:

- Adapt existing guidelines documents so that they can be applied to MCPs.

- Preserve existing guidelines, but add a system-level standard that would provide a suitable umbrella for the systems aspects and allow for the use of existing guidelines at software and hardware levels. This approach would be following that of DO-297 but targeting the instantiation of an MCP in a generic avionics system.

Amendments to DO-178C/DO-254 are a method being explored in order for these standards to be compliant with MCPs. It requires a complementary top-down point of view that is not traditionally the view taken by these standards.

An alternative could be to supplement these standards by a system-level top-down standard in a similar way to what has been done in IMA context with DO-297. This approach is less conventional than the DO-178C/DO-254 approach. It is, however, harmonious with the approach recommended in this report and summarized hereafter.

9.2.1 Recommended Activities

The approach is composed of the following steps:

1. Determination of the level of the various constraints that size performances of the MCP, in particular, for determinism consideration:

    a. Real-time constraints
    b. Safety constraints

2. Performance of top-down analysis aiming at:

    a. Isolating some high-level sources of non-determinism.
    b. Reducing the accessible UD in which non-determinism sources must be studied via a bottom-up analysis.
    c. Preparing possible mitigation strategies if non-determinism sources still remain in the UD.

3. Performance of bottom-up analysis complementing the top-down analysis in the UD. This analysis is performed around three topics:

    a. Local non-determinism (buffer and cache paradigms)
    b. Interferences between IP elements

c. Treatment of impact of failures on time and content determinism (this analysis is also called pathological case analysis)

This activity can be reduced in cases of service experience on some IPs. The possibility to lower the coverage of the bottom-up study can be considered on the basis of safety level to be reached (see table 8).

4. Mitigation of the remaining non-deterministic behaviors in the UD. This mitigation is performed in order to reach the level of determinism expected for the applications and with the uncertainty of real-time constraints.

Table 8. Proposed level of bottom-up studies determined by real-time and safety constraints

		Timing constraint	
		Hard real-time	Soft real-time
Safety constraint	DAL-A & DAL-B	High level of effort Full analysis for: • Local sources • Interferences • Failures Reduced only on the basis of: • Supplier input • In-service experience	Medium level of effort (safety centric) Reduced analysis on: • Local sources • Interferences Guarantying that content determinism is fulfilled and that timing determinism is definite in the limit of timing constraints.
	DAL-C & lower	Medium level of effort (Real-time-centric) Analysis for local sources, reduced only on the basis of: • Supplier input • In-service experience	Low Reduced bottom-up analysis

DAL= development assurance level

9.2.2 Pros and Cons of Conventional and Recommended Approaches

Table 9 presents the pros and cons for the two approaches described in this section.

Table 9. Summary of possible approaches for coverage of non-determinism in MCP

Solution	Advantages	Drawbacks
DO-254 & DO-178B/C approaches	This approach is conventional and well-known: • It has been exploited in complex electronic hardware. • It has been extended to COTS through the FAA order [7] or EASA report [8].	It develops mainly a bottom-up approach that is alone not applicable in an MCP context for use by industry.
Top-down coupled with bottom-up approach	• Dedicated to complex computational system with high integration level • Close to DO-297 approach developed in IMA framework where MCPs are of primary concern.	This approach is recommended but is not yet integrated in any standard or guideline document.

COTS = commercial off-the-shelf
IMA = integrated modular avionics
MCP = multicore processor

9.2.3 Definition of MCP Stakeholders and Associated Activities

In the DO-297 document, definitions and roles of IMA stakeholders have been recorded precisely in order to refine their responsibilities. Figure 26 illustrates the relationships between these stakeholders. A strong link is a relation that is always present whereas a weak link is dependent on the certification applicant habits and practices and, therefore, can differ from one aircraft certification to another. Indeed, the link between the application supplier and the platform supplier cannot exist if the system integrator is playing the appropriate intermediary role. This is the case when a certification applicant wants to guarantee a property of incremental qualification for its IMA system. This property enables the independence between the IMA platform and hosted application life cycles and their associated dependencies.

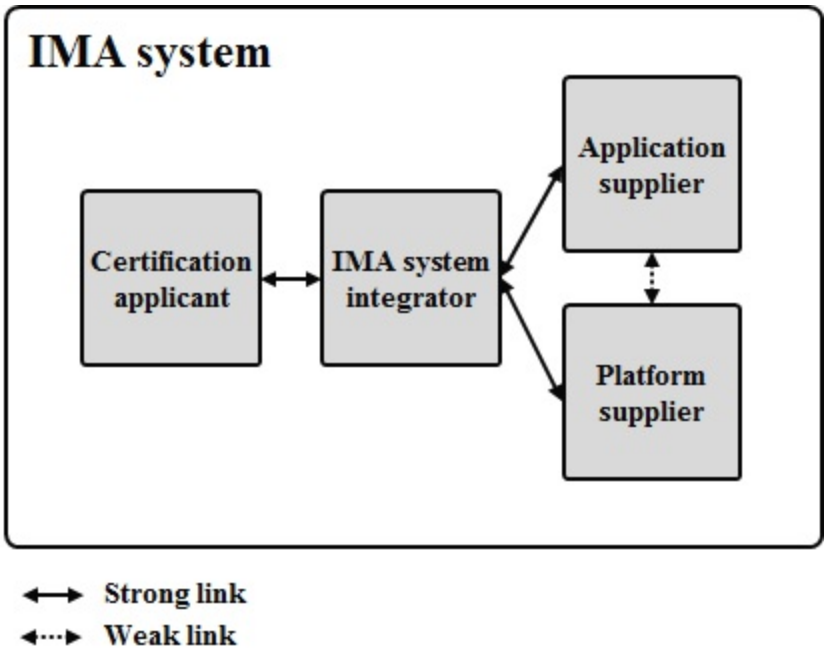

Figure 26. Relationship between stakeholders of MCP integration in bottom-up scheme

In the table 10, IMA stakeholders actively participating in the development of an IMA system are listed with their associated activities [4].

Table 10. Activities of IMA stakeholders

DO-297 stakeholder	Activities	Example of produced data
Certification applicant	• Demonstrate compliance with applicable aviation regulations. • Generate and validate all aircraft-level requirements and their allocation to subsystems.	N/A
IMA system integrator	Integrate platforms and hosted applications to produce the IMA system.	• Platform and hosted application configurations • Shared resource allocation
Platform and module suppliers	• Provide processing hardware and software resources (i.e., the processor plus the core software). • Provide development and configuration tools to support the IMA platform.	• Specification of IMA platform interfaces • Specification of shared resource allocation • Required resources and configuration data for the IMA platform
Application supplier	Provide hosted application.	• Specification of application external interfaces • Required resources and configuration data for the hosted application

IMA = integrated modular avionics

If we consider the top-down approach coupled with the bottom-up approach for MCP development, stakeholders and their associated activities must be defined. A significant number of commonalities can be found with DO-297, as MCPs are just another type of shared resource. Table 11 provides an example of the MCP stakeholders, with some of their associated activities.

Table 11. Summary of activities for MCP stakeholders

MCP stakeholder	Activities	Example of produced data
MCP system integrator	Integrate MCP platforms and hosted applications to produce the MCP system.	• Functions or tasks allocation on cores and scheduling • Platform and hosted application configurations (e.g., AMP or SMP) • Shared resource allocation (e.g., CPU cores, application, timeframe)
MCP platform supplier	• Provide processing hardware and software resources (i.e., the MCP processor plus the core software). • Provide core software execution model (i.e., AMP or SMP). • Provide development and configuration tools to support the MCP platform.	• Specification of MCP platform interfaces • Specification of shared resource allocation • Required resources and configuration data for the MCP platform
Application supplier	Provide hosted application.	• Specification of application external interfaces • Required resources (e.g., one or several CPU cores) and configuration data for the hosted application

MCP = multicore processor
AMP = asymmetrical multiprocessing
SMP = symmetrical multiprocessing
CPU = core processing unit

Nevertheless, activities are heavily dependent on the choices made for the MCP platform architecture. For example, the choice of an SMP strategy simplifies the activities of the MCP platform supplier because they no longer have to prove the independence between CPU cores. Indeed, at any given time, all CPU cores will be owned by one and only one hosted application. Therefore, partitioning infringement will no longer be possible. On the contrary, more activities fall on the application supplier to take benefit of all the CPU cores without compromising the safety assessment.

Finally, it is interesting to note that less complex federated platform developments are not in opposition with such MCP perspectives, but certain simplifications could be made, as several stakeholder roles may be covered by the same party.

9.3 CRITERIA FOR DETERMINISM

The criteria for declaring determinism has been achieved with either simple criteria binary in nature or complex criteria associated with decision trees.

9.3.1 Criteria Associated With Top-down Approach

These criteria are distinguished through their level of applicability, such as application-level, software-level, and safety-level.

9.3.1.1 Criteria at Application Level

The following criteria are applicable at application level:

- The number of safety criticality levels that can coexist on the different cores
- The application profile in terms of:
    - Real-time properties (short or long cycle)
    - The need to share data between the cores
- Related to the activation of the coherency mechanism at the software/hardware level:
    - The characterization of process activity (e.g., I/O intensive, memory intensive)

9.3.1.2 Criteria at Software Level

The following criteria are applicable at software level:

- The selected execution model and the underlying scheduling policy
- The OS capacity to manage parallel service calls (e.g., is a multicore API available?)

9.3.1.3 Criteria at Safety Level

The following criteria are applicable at safety level:

- The number of software layers that are responsible regarding the configuration of the MCP (this is the case on AMP, when several OS coexist)

In case of multiple software instances:

- The presence of a hardware/software mechanism to guarantee atomic access to the status and configuration registers
- The presence of a coherent UD between the several software instances (e.g., IPs enabled, access policy of shared resources)

9.3.2 Criteria Associated With Bottom-up Approach

These criteria include:

- The number of cores:
  - The number of cores is a relevant criterion when the increase in the number of cores may impact the buffer capacity (see section 5.4.1.1.1.2) or when considering the inter-core communication through a common pipe (e.g., the interconnect).
  - This criterion should be extended to the number of initiators (i.e., endpoints capable of issuing commands; see appendix A). For example, consider a dual-core MCP with two DMA and several complex IPs that can initiate transactions (e.g., an Ethernet frame decoder). Every one of the IPs could potentially interact in an ill-manner on the interconnect.
- The ratio of the number of targets (i.e., endpoints waiting for and answering the initiators' commands; see appendix A) of a given type to the number of initiators that can access the targets:
  - Consider a dual-core MCP, dual DMA (the initiators) with two DDR controllers (i.e., the targets). The above ratio is determined to be $\frac{2}{4}=\frac{1}{2}$. In other words, without UD restrictions, two initiators could access a DDR controller within the same time slice.
- The level of parallelism a resource can accept
- The SoC architecture:
  - Some architectures are more prone to non-deterministic behavior than others. Consider a SoC with a core cluster comprised of one core and of a dedicated memory allowing for local computation. This architecture is more deterministic that an architecture in which cores systematically rely on external memory, interfaced through a unique controller shared among cores, for their computational needs.

The analysis of determinism is influenced by several criteria:

- The mix of transactions changing the interference behavior (e.g., alternation of read and write)
- The density of transactions changing the interference behavior
- The frequency of occurrence of transactions
- The DRAM refresh and page miss patterns affecting the timing (this is dependent on the DRAM technology)
- Different controller configurations yielding different interference profiles (e.g., alignment, transaction reordering)

Exhaustiveness cannot, in general, be reached except for particular simple cases (e.g., SRAM), given the top-down limitations of UD.

10. CONCLUSIONS

The increased level of complexity introduced by multicore processors (MCPs) is pushing the limits of the feasibility of conventional bottom-up analyses of non-determinism issues in terms of needed time to complete, resource to allocate to the analysis, and information to obtain. To obtain a scope adjustment on the level of effort for the bottom-up analysis, this report recommends the performance of two additional steps prior to the bottom-up analysis:

1. An independent determination of applicable safety and real-time constraints
2. A top-down analysis

Performing a top-down analysis prior to the bottom-up analysis offers three major benefits:

1. The isolation of high-level sources of non-determinism and the architecture elements that affect them (e.g., function or task allocation onto cores, selection of software and scheduling strategies, selection of specific usage domain [UD])
2. The reduction of the accessible UD in which sources of non-determinism will be studied in the now bounded bottom-up analysis
3. The preparation of possible mitigation strategies if sources of non-determinism still remain in the UD

When performing the bottom-up analysis, the classification of local sources of non-determinism into cache type, buffer type, and source of interference provides a structure enabling the investigation of non-determinism in MCPs.

Out of the various sources of non-determinism in MCPs, interferences are undoubtedly an undesirable behavior for usage in avionics equipment regardless of the number of cores. Interferences make the processor's performance assessment complex to achieve and therefore raise safety issues. The main risk is to trigger timing failures on the whole equipment even if data failure modes might also be discovered during the interference analysis.

In line with the approach to safety analysis for non-determinism, the authors propose a safety process to specifically address the question of interference. The process is inspired by partitioning analyses performed on integrated modular avionics systems. No off-the-shelf generic solution seems to be feasible because of the wide variety of equipment, processors being used, level of criticality, and needs in terms of real-time requirements. Therefore, such a process must be instantiated on a case-by-case basis and presented in the certification plan as soon as possible. Its definition should contain acceptability and termination criteria that will be applied during the safety assessment phases.

Several points need be considered. First, no limitation on the number of cores seems relevant even if a processor that embeds more cores makes the interference analysis more complex. Second, exhaustiveness of analyses is not likely to be reachable on COTS processors containing either highly complex and/or black-box components. Thus, confidence should be obtained by combining several analysis methods (e.g., semi-formalized analyses coupled with information shared with the manufacturer and expert feedback on the processor, or similar ones in avionics). Safety experts must drive these methods. Similarly, the interference analysis should be performed with final applications, if possible. When it is not the case, representative applications fitting with the UD can be used, even if trustworthiness may be more complex to achieve.

Finally, interference mitigations, especially interference elimination techniques, do not displace the need for interference analysis, which brings evidence of interference control.

11. REFERENCES

1. SAE International. (2010). Guidelines for Development of Civil Aircraft and Systems, ARP-4754A.
2. Girbal, S., Pérez, D. G., Rhun, J. L., Faugère, M., Pagetti, C., & Durrieu, G. (2015). *A complete toolchain for an interference-free deployment of avionic applications on multi-core systems*. Paper presented at the 34th Digital Avionics Systems Conference, Prague, Czech Republic. Retrieved from https://safure.eu/downloads/paper_TRT.1.pdf.
3. Bost, E. (2013, May). Hardware Support for Robust Partitioning in Freescale QorIQ Multicore SoCs (P4080 and derivatives). Retrieved from http://www.nxp.com/files/32bit/doc/white_paper/QORIQHSRPWP.pdf.
4. Radio Technical Commission for Aeronautics. (December, 2005). DO-297, Integrated Modular Avionics Development Guidance and Certification Considerations. Retrieved from https://www.researchgate.net/publication/228799756_Integrated_Modular_Avionics_Development_Guidance_and_Certification_Considerations
5. Radio Technical Commission for Aeronautics. (December, 2011). DO-178C, Software Considerations in Airborne Systems and Equipment Certification. Retrieved from http://www.dcs.gla.ac.uk/~johnson/teaching/safety/reports/schad.html
6. Radio Technical Commission for Aeronautics. (April, 2000). DO-254, Design Assurance Guidance for Airborne Electronic Hardware.
7. FAA Order 8110.105. Simple and Complex Electronic Hardware Approval Guidance, Change 1 (2008).
8. EASA Report. (2012). Development Assurance of Airborne Electronic Hardware (CM-SWCEH-001).
9. Certification Authorities Software Team (CAST). (May, 2014). Multi-core Processors. Position Paper #32. Retrieved from https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32.pdf.
10. EASA. (November, 2013). *COTS-AEH—Use of Complex COTS (Components-Off-the-Shelf) in Airborne Electronic Hardware—Failure Mode and Mitigation*. Retrieved from https://www.easa.europa.eu/document-library/research-projects/easa201204#group-easa-downloads
11. Certification Authorities Software Team (CAST). (June, 2003). Addressing Cache in Airborne Systems and Equipment. Position Paper #20. Retrieved from https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-20.pdf.
12. IEC 61508 (2010). Functional Safety of Electrical/Electronic/Programmable Electronic Safety Related Systems, part 4, edition 2.0. Retrieved from http://www.iec.ch/functionalsafety/standards/page2.htm

13. Brindejonc, V., Marcuccilli, G., & Petit, S. (October, 2010). *System FMECA in the framework of ISO 26262*. Proceedings from the Lambda-Mu 17, Gentilly, France.

14. EASA. (December, 2012). *MULCORS—Use of Multicore Processors in Airborne Systems.* Retrieved from https://www.easa.europa.eu/document-library/research-projects/easa20116

15. Mondal, M., Ragheb, T., Wu, X., Aziz, A., & Massoud, Y. (2007). *Provisioning on-chip networks under buffered RC interconnect delay variations.* Proceedings from the 8th International Symposium on Quality Electronic Design, Washington, D.C.

16. Rusu, C., Grecu, C., & Anghel, L. (2008). *Coordinated versus uncoordinated checkpoint recovery for network-on-chip based systems.* Proceedings from the 4th IEEE International Symposium on Electronic Design, Test and Applications, Hong Kong.

17. Moyer, B. (2013). *Real world multicore embedded systems: A practical approach*, Waltham, MA: Newnes-Elsevier.

18. Reineke, J., Grund, D., Berg, C., & Wilhelm, R. (2007). Timing predictability of cache replacement policies. *Real-Time Systems*, *37*(2), 99-122.

19. Grund, D., & Reineke, J. (2010). *Toward precise PLRU cache analysis*. Paper presented at the 10th International Workshop on Worst-Case Execution Time Analysis. Retrieved from http://ecrts.eit.uni-kl.de/fileadmin/WebsitesArchiv/Workshops/WCET/Proceedings/WCET2010-Proceedings.pdf

20. Hardy, D., Piquet, T., & Puaut, I. (2009). *Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches.* Paper presented at the 30th IEEE Real-time System Symposium (RTSS). Retrieved from https://hal.inria.fr/inria-00380298/document.

21. Brindejonc, V., & Roger, A. (2014). *Avoidance of dysfunctional behavior of complex COTS used in an aeronautical context.* Proceedings from the Lambda-Mu RAMS Conference, Dijon, France.

22. Nowotsch, J., & Paulitsch, M. (2012). *Leveraging multicore computing architectures in avionics*. Proceedings from the 2012 European Dependable Computing Conference, Sibui, Romania.

23. Vera, X., & Abella, J. (2007). *Surviving to errors in multi-core environments* [PDF file]. Retrieved from http://people.ac.upc.edu/jabella/surviving_iolts07.pdf..

24. Chaparrp, P., Abella, J., Carretero, J., & Vera, X. (2008). *Issue system protection mechanisms*. Paper presented at the IEEE International Conference, Lake Tahoe, CA. Retrieved from http://iccd.et.tudelft.nl/2008/proceedings/599chaparro.pdf.

25. Gizopoulos, D., Psarakis, M., Adve, S.V., Ramachandran, P., Hari, S. Sorin, D., … Vera, X. (2011). *Architectures for online error detection and recovery in multicore processors* Retrieved from the http://ece.duke.edu/~sorin/papers/date11_special.pdf..

26. DeOrio, A., Aisopos, K., Bertacco, V., & Peh, L-S. (2011). *DRAIN: Distributed recovery architecture for inaccessible nodes in multi-core chips.* Proceedings from the 2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC 2011), San Diego, CA.

27. Nowotsch, J., Paulitsch, M., Buhler, D., Theiling, H., Wegener, S., & Schmidt, M. (2014). *Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement.* [PDF file]. Retrieved from http://ecrts.eit.uni-kl.de/fileadmin/files_ecrts14/documents/presentations/nowotsch.pdf.

28. Bin, J. (2014). *Controlling execution time variability using COTS for safety-critical systems* (Doctoral dissertation). Retrieved from WorldCat (892840653).

29. Bin, J., Girbal, S. G., Daniel, P., Grasset, A., & Merigot, A. (2014). *Studying co-running avionic real-time applications on multi-core COTS architectures*. Paper presented at the 7th European Congress On Embedded Real Time Software And Systems (ERTS). Toulouse, France. Retrieved from https://www.researchgate.net/publication/261721328_Studying_co-running_avionic_real-time_applications_on_multi-core_COTS_architectures

30. Pellizzoni, R. A., Schranzhofer, J. J., Chen, M., Caccamo, & Thiele, L. (2010). *Worst case delay analysis for memory interference in multicore systems*. Proceedings from the 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010), Dresden, Germany.

31. Mingsong, L., Nan, G., Wang, Y., Jan, R., & Reinhard, W. (2015). A survey on cache analysis for real-time systems. *ACM Computing Surveys.* Retrieved from http://user.it.uu.se/~yi/pdf-files/2015/lgyrw-acm15.pdf.

32. Huynh, B. K., Ju, L., & Roychoudhury, A. (2011). *Scope-aware data cache analysis for WCET estimation*. Proceedings from the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Chicago, IL.

33. Yan, J., & Zhang, W. (2008). *WCET analysis for multicore processors with shared L2 instruction caches.* Proceedings from the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium, Washington, D.C.

34. Zhang, W., & Jun, Y. (2012). Static timing analysis of shared caches for multicore processors. *Journal of Computing Science and Engineering, 6*, 267–278.

35. Aitken, R., & Idgunji, S. (2007). *Worst-Case Design and Margin for Embedded SRAM.* Proceedings from the 2007 Design, Automation & Test in Europe Conference & Exhibition, Nice, France

36. Drepper, U. (2007, November, 21). What every programmer should know about memory. Retrieved from www.akkadia.org/drepper/cpumemory.pdf.

37. Paolieri, M., Quiones, E., & Cazorla, F. J. (2013). Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions. *ACM Trans. Embed. Computing Systems, 12*(1s). doi: 10.1145/2435227.2435260.

38. Bourgade, R., Rochange, C., & Sainrat, P. (2011). *Predictable bus arbitration schemes for heterogeneous time-critical workloads running on multicore processors.* Proceedings from the 2011 IEEE 16th Conference on Emerging Technologies & Factory Automation (ETFA), Toulouse, France.

39. ARM. (2012). *CoreLink CCI-400 Cache Coherent Interconnect Technical Reference Manual, Rev G*, Cambridge, England: ARM Limited. Retrieved from http://infocenter.arm.com/help/topic/com.arm.doc.ddi0470-/.

40. Texas-Instruments. (2012). TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor.

41. Shah, H., Raabe, A., & Knoll, A. (2013). *Challenges of WCET analysis in COTS multicore due to different levels of abstraction.* Proceedings from the Workshop on High-performance and Real-time Embedded Systems (HiRES), Berlin, Germany. Retrieved from http://www6.in.tum.de/Main/Publications/Shah2013a.pdf.

42. Vasudevan, A. (2015). Techniques for Achieving Time-Predictability in Multicores—A Survey.

43. Girbal, S., Jean, X., le Rhun, J., Pérez, D. G., & Gatti, M. (2015). *Deterministic platform software for hard real-time systems using multicore COTS*. Proceedings from the 34th Digital Avionics Systems Conference, Prague, Czech Republic.

44. Chattopadhyay, S., Roychoudhury, A., Rosen, J., Eles, P., & Peng, Z. (2014). Time-predictable embedded software on multicore platforms: Analysis and optimization. *Foundations and Trends in Electronic Design Automation, 8*(3–4), 199–356.

45. Mancuso, R., Dudko, R., Betti, E., Cesati, M., Caccamo, M. & Pellizzoni, R. (2013). *Real-time cache management framework for multicore architectures*. Proceedings from the IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), Philadelphia, PA.

46. Illinois Digital Environment for Access to Learning and Scholarship (IDEALS). (2014). *Single core equivalent virtual machines for hard real-time computing on multicore processors*. Illinois: Sha, L., Caccamo, M., Mancuso, R., Kim, J.-E., Yoon, M.-K., & Pellizzoni, R. Retrieved from https://www.ideals.illinois.edu/handle/2142/55672

47. Ward, B. C., Herman, J. L., Kenna, C. J., & Anderson, J. H. (2013). *Making shared caches more predictable on multicore platforms*. Proceedings from the 25th Euromicro Conference on Real-Time Systems, Los Alamitos, CA.

48. Yun, H., Mancuso, R., Wu, Z.-P., & Pellizzoni, R. (2014). *PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms.* Proceedings from the 20th Real-Time and Embedded Technology and Applications Symposium, Berlin, Germany.

49. Moscibroda, T., & Mutlu, O. (2007). *Memory performance attacks: denial of memory service in multicore systems*. Proceedings from the 16th USENIX Security Symposium, Boston, MA.

50. Kleen, A. (2009). *Linux multicore scalability*. [PDF file]. Retrieved from http://www.linux-kongress.org/2009/slides/linux_multicore_scalability_andi_kleen.pdf.

51. Cui, Y., Zhang, W., Chen, Y., & Shi, Y. (2010). *A scheduling method for avoiding kernel lock thrashing on multicores.* Proceedings from the 16th International Conference on Parallel and Distributed Systems, Shanghai, China.

52. Zhang, X., & Zheng, H. (2014). *US Patent No. 9164765*. Washington, DC: U.S. Patent and Trademark Office.

53. Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., & Sha, L. (2013). *Memguard: Memory bandwidth reservation system for efficient performance isolation in multicore platforms* Proceedings from the 19th Real-Time and Embedded Technology and Applications Symposium, Philadelphia, PA.

54. University of York, Department of Computer Science. (2013). *Mixed criticality systems – a review*. York, U.K.: Burns, A., & Davis, R. Retrieved from https://www-users.cs.york.ac.uk/burns/review.pdf.

55. Kritikakou, A., Pagetti, C., Rochange, C., Roy, M., Faugere, M., Girbal, S., & Gracia Perez, D. (2014). *Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems.* Proceedings from the 22nd International Conference on Real-Time Networks and Systems, Versailles, France.

56. Lickly, B., Liu, I., Kim, S., Patel, H. D., Edwards, S. A., & Lee, E. A. (2008). *Predictable programming on a precision timed architecture*. Proceedings from the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, Atlanta, GA.

57. Ungerer, T., Cazorla, F., Sainrat, P., Bernat, G., Petrov, Z., Rochange, C., … Mische, J. (2010). Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro, 30*, 66–75.

58. Hansson, A., Goossens, K., Bekooij, M., & Huisken, J. (2009). CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation Electronic Systems, 14*(1), 2:1–2:24.

59. Fisher, S. SYSGO AG. (2013). *Certifying applications in a multi-core environment: The world's first multi-core certification to SIL 4*. Retrieved from http://www.wb-ip.com.au/uploads/3/8/8/4/38841341/01_2014_sil4_multicore_certification.pdf.

60. Vestal, R. (2010). *US Patent No. 8316368.* Washington, DC: U.S. Patent and Trademark Office.

61. Pellizzoni, R., Betti, E., Bak, S., Yao, G., Criswell, J., Caccamo, M., & Kegley, R. (2011). *A predictable execution model for COTS-based embedded systems*. Paper presented at the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). Chicago, IL. Retrieved from http://pertsserver.cs.uiuc.edu/~mcaccamo/papers/PREM_rtas11.pdf.

62. Durrieu, G., Faugère, M., Girbal, S., Gracia Pérez, D., Pagetti, C., & Puffitsch, W. (2014). *Predictable flight management system implementation on a multicore processor*. Paper presented at the 2014 Embedded Real Time Software and Systems, Toulouse, France. Retrieved from https://www.researchgate.net/publication/262949378_Predictable_Flight_Management_System_Implementation_on_a_Multicore_Processor.

63. Schranzhofer, A., Pellizzoni, R., Chen, J.-J., Thiele, L., & Caccamo, M. (2010). *Worst-case response time analysis of resource access models in multicore systems*. Proceedings from the 47th Design Automation Conference, New York, NY.

64. Jegu, V., Triquet, B., Aspro, F., Boniol, F., & Pagetti, C. (2012). *US Patent No. 08694747*. Washington, D.C.: U.S. Patent and Trademark Office.

65. Schranzhofer, A., Chen, J.-J. & Thiele, L. (2010). *Timing analysis for TDMA arbitration in resource sharing systems*. Paper presented at the 16th IEEE Real-Time and Embedded Technology and Applications Symposium, Stockholm, Sweden. Retrieved from http://embedded.cs.uni-saarland.de/literature/TimingAnalysisOfResourceAccessInterference.pdf.

66. Jean, X., Faura, D., Gatti, M., Pautet, L., & Robert, T. (2013). *A software approach for managing shared resources in multicore processors for IMA systems.* Paper presented at the 2013 IEEE/AIAA 32nd Digital Avionics Systems Conference, East Syracuse, NY. Retrieved from https://www.researchgate.net/publication/261208969_A_software_approach_for_managing_shared_resources_in_multicore_IMA_systems

67. Jean, X. (2015). *Hypervisor control of COTS multicores processors in order to enforce determinism for future avionics equipment*. (Doctoral dissertation). Retrieved from https://www.researchgate.net/publication/299489602.

68. Jean, X., Girbal, S., Roger, A., Megel, T., & Brindejonc, V. (2015). *Safety considerations for WCET evaluation methods in avionic equipment*. Paper presented at the 2015 IEEE/AIAA 34th Digital Avionics Systems Conference, Prague, Czech Republic. Retrieved from https://safure.eu/downloads/paper_TRT.3.pdf.

69. Hari, S. M-L., Li, P., Ramachandran, Choi, B., & Adve, S.V. (2009). *mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems*. Paper presented at the 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture, New York, NY. Retrieved from http://slideplayer.com/slide/5832553/

70. Bertozzi, D., Benini, L., & de Micheli, G. (2005). Error control schemes for on-chip communication links: The energy-reliability tradeoff. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, 24*(6), 818–831.

71. Murali, S., de Micheli, G., Benini, L., Theocharides, T., Vijaykrishnanm N., & Irwin, M. (2005). IEEE design and test of computers, *Analysis of Error Recovery Schemes for Networks on Chips*, *22*(5), 434-442.

72. Rossi, D., Angelini, P., & Metra, C. (2007). *Configurable error control scheme for NoC signal integrity.* Paper presented at the 13th IEEE International On-Line Testing Symposium, Crete, Greece. Retrieved from http://eprints.soton.ac.uk/368903/1/iolts07.pdf.

73. Dumitras, T., & Marculescu, R. (2003). *On-chip stochastic communication.* Proceedings from the 2003 Design, Automation and Test in Europe Conference and Exhibition, Washington, D.C.

74. Pirretti, M., Link, G. M., Brooks, R. R., Vijaykrishnan, N., Kandemir, M., & Irwin, M.J. (2004). *Fault tolerant algorithms for network-on-chip interconnect.* IEEE Computer Society Annual Symposium on VLSI, Lafayette, LA.

75. Sun, W., Xiong, N., Yang, L. T., & Rong, C. (2010). *Towards free task overloading in passive replication based real-time multiprocessors.* Proceedings from the 2010 IEEE 10th International Conference on Computer Information Technology, Bradford, U.K.

76. Li, Y., Makar, S., & Mitra, S. (2008). *CASP: Concurrent autonomous chip self-test using stored test patterns.* Paper presented at the 2008 Design, Automation and Test in Europe, Munich, Germany. Retrieved from https://www.date-conference.com/proceedings-archive/PAPERS/2008/DATE08/PDFFILES/07.5_1.PDF.

77. Agarwal, M., Paul, B., Zhang M., & Mitra, S. (2007). *Circuit failure prediction and its application to transistor aging*. Proceedings from the 25th IEEE VLSI Test Symposium, Berkeley, CA.

78. Ungsunan, P. D., Lin, C., Kong, X., & Gai, Y. (2009). *Improving multi-core system dependability with asymmetrically reliable cores*. International Conference on Complex, Intelligent and Software Intensive Systems, Fukuoka, Japan.

79. Braun, J. & Mottok, J. (2013). *Fail-safe and fail-operational system safeguarded with coded processing*. Proceedings from the 2013 IEEE EuroCon, Zagreb, Croatia.

80. Imai, M., Nagai, T., & Nanya, T. (2010). *Pair and swap: An approach to graceful degradation for dependable chip multiprocessors.* Proceedings from the 2010 International Conference on Dependable Systems and Networks Workshops, Chicago, IL.

APPENDIX A—GLOSSARY

Access pattern	Repetitive memory accesses, in opposition to local (one-shot) memory access.
Asymmetric multiprocessing (CAST #32)	Each individual functional process is permanently allocated to a separate core, and each core has its own OS. However, the OSs may be multiple copies of the same OS or be different from core to core.
Bound multiprocessing (CAST #32)	Extends symmetric multiprocessing by allowing the developer to bind any process and all of its associated threads to a specific core while using a common OS across all cores.
Branch misprediction	The CPU wrongly predicts the next instruction to be executed.
Cache coherence	Consistency of shared resource data that is stored in multiple local caches. Cache coherency protocol or coherency mechanisms are intended to manage the potential conflicts and maintain the consistency between the local cache and the shared memory.
Cache-hit	A request that can be served by simply reading the cache because the requested data is already contained in the cache.
Cache-miss	Opposite to cache-hit: The requested data has to be recomputed or fetched from its original storage location.
Component (RTCA DO-297)	A self-contained hardware or software part, database, or combination thereof that may be configuration controlled.
Coupling (control) (RTCA DO-297)	The manner or degree by which one software component influences the execution of another software component.
Coupling (data) (RTCA DO-297)	The dependence of a software component on data not exclusively under the control of that software component.
Data parallelism	In a multiprocessor system executing a single thread, each processor performs the same task on different pieces of distributed data.
Determinism (DO-297)	The ability to produce a predictable outcome generally based on the preceding operations. The outcome occurs in a specified period of time with some degree of repeatability.
Determinism (data)	Data values are always persistent once set and not affected by execution architecture or communication between cores.

OS = operating system
CPU = core processing unit

Determinism (error)	A mistake in requirements, design, or implementation
Determinism (execution)	The ultimate execution of a sequence of instructions can always be predicted. In other words, prefetching, out-of-order execution only affects performance.
Determinism (failure)	An occurrence that affects the operation of the MCP such that it can no longer function as intended
Determinism (resource access)	When an executable requires access to an input/output (I/O) or communication resource, that resource is available in a quantifiable and bounded manner.
Determinism (timing)	Worst case timing can be established with a known and bounded variance (or jitter).
Directed Acyclic Graph	A graph formed by a collection of vertices and directed edges in such a way that no directed cycle (loop) exists. In computer science, a DAG is used to model processes in which data flows in a consistent direction through a network of processors.
Hyperthreading	1. A high-performance computing architecture that simulates some degree of overlap in executing two or more independent sets of instructions. 2. A feature of certain Intel chips that makes one physical CPU appear as two logical CPUs. It uses additional registers to overlap two instruction streams in order to achieve gains in performance.
Initiator (see also Target)	An initiator refers to an endpoint sending commands to a target. For example, a computer can be an initiator and a data storage unit a target. In a client-server architecture, the client plays the role of an initiator and the server of the target. In the context of multicores, the initiators can be CPU cores and targets are the modules connected over the bus (e.g., ROM, RAM).

CPU = core processing unit
ROM = read-only memory
RAM = random access memory
DAG = directed acylic graph

Instruction-level parallelism	A measure of how many of the operations in a program can be executed simultaneously
Instruction set architecture	Part of the computer architecture related to programming, the precise definition of the interface between hardware and software. An ISA includes native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O.
Interference	To date, there is no common opinion agreement on the definition of interference in an MCP context. Interference is defined through its effect: slowdown of tasks due to concurrent tasking or perturbation in access times. The problem with this definition if that it does not support analysis, and these effects are not sufficiently discriminatory that they can be attributed to interference only (they may be caused by other phenomena). For this report, interference is an alteration of the processor's behavior seen by software running on one core, due to activities ordered by software running on other cores. A statement developed in this report considers that the impact of interferences is firstly a problem of performance assessment and timing determinism enforcement. Interferences are undesired phenomena that are considered by the manufacturer as belonging to the functional domain of the processor. Interferences are, however, seen as dysfunctional behaviors by the avionic equipment provider.
Interference analysis	An interference analysis is a process that considers interference paths and their usage according to the processor's UD. The analysis determines which paths are acceptable (from a performance's and safety's points of view) and which are not. The latter are called "interference channels" (see Interference channel).

ISA = instruction set architecture
MCP = multicore processor
I/O = input/output
UD = usage domain

Interference channel	An interference path for which interferences have actually been observed and do not cope with the equipment's functional domain
Interference path	A configuration wherein a given set of initiators (e.g., cores, DMA engines, master I/O) are allowed to communicate with a given set of targets (e.g., main memory, shared caches, slave I/O) without restriction Having an interference path does not necessarily mean interferences will actually occur. It only represents a configuration for which there is a risk of interference. Thus, interference paths can be seen as test cases wherein given initiators are allowed to request given targets (e.g., core #0 targets the main memory while core #1 targets the controller of the PCIe). Furthermore, the processor's behavior can be stated under each of these test cases. In other words: Having an interference path does not necessarily mean interferences will actually occur, while having an interference channel means undesirable interference occurs.
Interference source	A component on the processor that may entail interferences when used simultaneously by several cores or other initiators.. Examples of interference sources are shared caches, peripherals, and interconnects.
Locality (spatial)	Is exhibited by a reference when the data that is requested is stored spatially close to a data that has been requested already
Locality (temporal)	Is exhibited by a reference when data that has already been requested is requested again
Message passing	Type of explicit communication between concurrent components: exchange of messages, whether synchronously or asynchronously. The other type of explicit communication is shared memory communication.
Microarchitecture	Set of processor design techniques used to implement the instruction set. A same instruction set may have several different microarchitectures.

DMA = direct memory access
I/O = input/output
PCIe = Peripheral Component Interconnect Express

Multicore processor	A device that contains two or more independent processing cores. A core in the MCP is defined as a device that executes software. This includes virtual cores.
NP-Complete	Is said of a decision problem for which the solution can be quickly verified, while there is no known way to quickly find a solution in the first place. These problems are common in programming, and they are often solved using heuristic methods and approximation algorithms.
Operating system (RTCA DO-297)	1. The same as executing software 2. The software kernel that services only the underlying hardware platform 3. Software that directs the operations of a computer, resource allocation and data management, controlling and scheduling the execution of computer hosted applications, and managing memory, storage, I/O, and communication resources
Partitioning (RTCA DO-297)	An architectural technique to provide the necessary separation and independence of functions or applications to ensure that only intended coupling occurs.
Processor (RTCA DO-297)	A device used for processing digital data.
Process variation	Naturally occurring variation in the attributes of transistors when integrated circuits are fabricated.
Real-time system [A-1]	A system in which the correctness of the computations not only depends on their logical correctness, but also on the time at which the result is produced. In other words, a late answer (with respect to a specified time stamp) is a wrong answer.
Resource (RTCA DO-297)	Any object (e.g., processor, memory, software, or data) or component used by a processor, IMA platform, core software, or application. A resource may be shared by multiple applications or dedicated to a specific application. A resource may be physical (a hardware device) or logical (a piece of information).

MCP = multicore processor
I/O = input/output
IMA = Integrated Modular Avionics

Robust partitioning (RTCA DO-297)	A means for assuring the intended isolation of independent aircraft functions and applications residing in IMA shared resources in the presence of design errors and hardware failures that are unique to a partition or associated with application-specific hardware.
Target (see also Initiator)	An endpoint waiting for commands sent by an initiator and providing the required I/O. For example, a computer can be an initiator and a data storage unit a target. In a client-server architecture, the client plays the role of an initiator and the server of the target. In the context of multicores, the initiators can be CPU cores, and targets are the modules connected over the bus (e.g., ROM, RAM).
Task-level parallelism	Each processor executes a different thread (or process) on the same of different data. Task parallelism emphasizes the distribution of executing threads across different parallel computing nodes.
Usage domain (RTCA DO-297)	A declared set of characteristics for which it can be shown that: 1. The module is compliant to its functional, performance, and safety requirements as defined in the Module Requirements Specification. 2. The module meets all the assertions and guarantees regarding its defined allocate-able resources and capabilities. 3. The module performance is fully characterized, including fault and error handling, failure modes, and behavior during adverse environmental effects.
WCET function	The evolution of WCET as a function of task-level WCET.

IMA = integrated modular avionics
I/O = input/output
CPU = core processor unit
ROM = read-only memory
RAM = random access memory
WCET = worst case execution time

APPENDIX A—REFERENCES

A-1 Betti, E., (2010). Satisfying Hard Real-time Constraints Using COTS Components, Ph.D. Thesis, Universita di Roma Tor Vergata.

APPENDIX B—WORST CASE EXECUTION TIME EVALUATION ASPECTS

A significant number of avionics functions must fulfill hard real-time requirements, usually refined by deadlines applied to the whole system. At system level, the requirement is strict: Missing a deadline is a failure condition according to the definition of CS 25.1309/AC 25.1309-1A [B-1].

When this requirement is refined to a system's components, especially software ones, hard real-time constraints are still defined as deadlines. At this level of granularity, missing a deadline is a single failure in the acceptation of CS 25.1309/AC 25.1309-1A. Therefore, hard real-time constraints at system level appear to be less stringent, as missing a deadline at the software level cannot, on its own, lead to a catastrophic failure condition. However, it can be argued that this rule does not apply to hazardous failure conditions—and even at the catastrophic level, missing deadlines in software represents a risk that must be assessed.

At the software level, missing a deadline can occur for the following reasons:

- A scheduling anomaly occurred: The task was not allocated enough time to complete its execution within its deadline. Such an anomaly may result from a bad specification of the task set in the usage domain (UD). For example, several aperiodic tasks may be activated at the same time, demanding a heavy core processing unit load and thus making the task set not schedulable at a given moment. Scheduling problems are usually addressed by scheduling analysis techniques, which have a numerous literature, even for multicore processors [B-2]:

- A bad estimation of one task's computation time: The task's execution trace was correct but ran longer than expected. The worst case execution time (WCET) evaluation techniques [B-3] have been introduced to address this problem.

- Unexpected interferences occurred between tasks: A task was slowed down by concurrent tasks executed on different cores in a range that was not expected. On integrated modular avionics (IMA) modules, this problem could be addressed by the robust partitioning property. However this problem applies to all avionics systems embedding multicore processors (MCP) and not only IMA, and remains partially unsolved except in restrictive scenarios [B-4].

Figure B-1 synthesizes the concurrent approaches to the WCET estimation in the same framework.

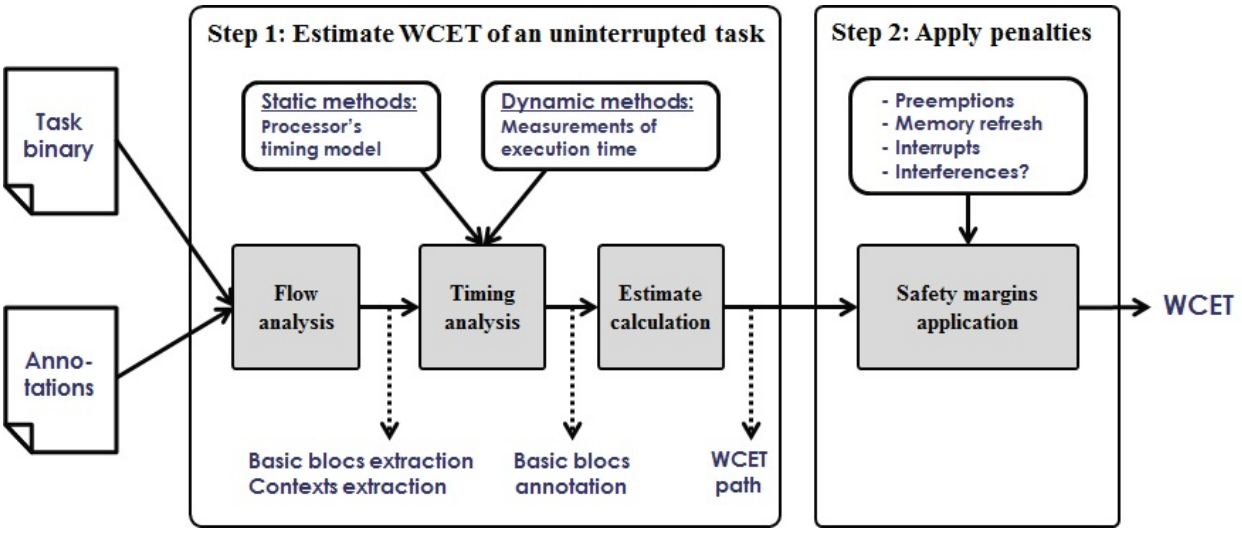

Figure B-1. Overview of the WCET evaluation flow

Historically, the WCET evaluation techniques were focused on the second issue, but some extensions have been proposed to take into account phenomena related to interferences on MCPs [B-5].

The following definition for WCET is proposed: The WCET of a task is the duration for which it is considered with a sufficient level of confidence that this task will fulfill its execution, whatever its execution path, the processor's initial state, and the events that may occur as long as they comply with the UD specification.

B. 1 OVERVIEW OF WCET EVALUATION TECHNIQUES

The WCET evaluation is commonly applied to each task. The process is comprised of two main steps.

STEP 1: EVALUATE THE WCET OF THE TASK WITHIN A SIMPLIFIED ENVIRONMENT

This step is the most studied in the literature and the most complex one. It aims at exploring the task's execution paths in order to find one that approximates the worst case and gathers the execution time in the most unfavorable hardware configuration [B-3]. The task is supposed to be uninterrupted, which means the effects of preemption and interrupts are not considered at this stage. Additionally, some hardware mechanisms may be ignored. The first step is further broken down in the following three sub-steps:

1. Flow analysis: The task's binary is explored to find out reachable execution paths. The raw binary may be explored automatically through static analysis techniques. However, it usually requires additional information from the user, commonly called annotations. This stage's output is a control flow graph (CFG) breakdown of the task in a set of basic blocks (usually functions and loops) and contexts that formalize relations between basic blocks.

2. Timing analysis: Each basic block's execution time is assessed from the analysis of a processor's model or measurements. This assessment may be performed for each context. This stage's output is a set of annotations of basic blocks' execution times.

3. Estimate calculation: The path that maximizes the execution time is determined from the annotated CFG. Analysis methods usually consider graph exploration techniques or constraint-solving techniques.

This step is more developed in the academic community than in the industry, even if industrial processes have evolved over the last few years. The first step ends with an intermediate WCET that is used in the second step.

STEP 2: APPLY PENALTIES TO THE INTERMEDIATE WCET

These penalties, also called safety margins, account for phenomena that were not considered in the previous phase. They correspond to hardware and software events such as:

- Operating system preemption and system ticks
- Interrupts management and other asynchronous events
- Dynamic random access memory refresh operations
- In some cases, interferences in MCP

This step is more developed in the industry than in the academic community. Historically, mitigation of timing non-determinism by safety margins was common in timing analysis processes. It is still the case today.

Some WCET analysis tools, like aiT and RapiTime [B-3], have almost fully automated this process and some of them support commercial off-the-shelf (COTS) processors. Their use is now generalized among industrial processes, even if many processes still involve human analyses and exhaustive test campaigns.

The WCET evaluation techniques are traditionally split between static (or analytic) and dynamic (or empirical) methods. More precisely, the distinction lies in the timing analysis of Step 1. Static methods rely on the analysis of software execution over a cycle-accurate model of the processor. This enables fine-grained analyses that identify worst case behaviors in components like pipeline and caches. aiT [B-6] is the most mature tool in this domain. Otawa [B-7] is also a good example of an open-source library for static WCET computation. Meanwhile, dynamic methods aim at gathering execution times from test campaigns that are expected to cover unfavorable hardware configurations. RapiTime [B-8] is an example of a mature tool in this aspect.

The extension of existing WCET evaluation methods to support MCP, either COTS or custom processors, is an active research field [B-9]. The main efforts are focused on the introduction of interference penalties, either in Step 1 during the timing analysis phase or in Step 2 as a global penalty. Interesting approaches can be found in [B-5, B-10, B-11].

Finally, the WCET evaluation methods produce bounds on the execution time of tasks. The interpretation of this bound may vary among communities, as it does in the literature. According to some papers, the WCET is a bound that shall never be exceeded, while for others it may be associated with a probabilistic assessment [B-12]. The idea in this report is not to close the debate but to point out that any method leads to a tradeoff between the following:

- The risks associated with the use of the method
    - The method may not cover all situations.
    - Abstractions used for the WCET evaluation, for instance processor models, may not be correct or be so inaccurate that the computed WCETs are too pessimistic.
- The risks associated with the evaluated WCET: Assuming that the method is correct, can the resulting WCET be exceeded?
- The precision of the evaluation and, thus, the system efficiency.

APPENDIX B—REFERENCES

B-1 FAA. (1988, June). Advisory Circular 25.1309-1A, *System Design and Analysis*. Washington, D.C.: U.S. Department of Transportation.

B-2 Davis, R. I., & Burns, A. (2011). A survey of hard real-time scheduling for multiprocessor systems, *ACM Comput. Surv.*, *43*(4), 35:1-35:44.

B-3 Wilhelm. R., Engblo, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., … Stenström, P. (2008). The Worst-Case Execution-Time problem overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst. 7*(3), 36:1-36:53.

B-4 Jean, X., Faura, D., Gatti, M., Pautet, L., & Robert, T. (2012). *Ensuring robust partitioning in multicore platforms for IMA systems.* Proceedings from the 31st IEEE/AIAA Digital Avionics Systems Conference (DASC), Williamsburg, VA.

B-5 Nowotsch, J., Paulitsch, M., Buhler, D., Theiling, H., Wegener, S., & Schmidt, M. (2014). 26th Euromicro Conference on Real-Time Systems (ECRTS): *Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement.*

B-6 Kästner, D., Wilhelm, R., Heckmann, R., Schlickling, M., Pister, M., Jersak, M., … Ferdinand, C. (2009). Leveraging Applications of Formal Methods, Verification and Validation. *Timing Validation of Automotive Software, 17,* 93-107, T. Margaria & B. Steffen (Eds.). Berlin: Springer.

B-7 Ballabriga, C., Cassé, H., Rochange, C., & Sainrat, P. (2010). Proceedings of the 8th IFIP WG 10.2 International Conference on Software Technologies for Embedded and Ubiquitous Systems: *OTAWA: an open toolbox for adaptive WCET analysis*.

B-8 Bernat, G., Davis, R., Merriam, N., Tuffen, J., Gardner, A., Bennett, M., & Armstrong, D. (2007). Identifying opportunities for worst-case execution time reduction in an avionics system. *Ada User Journal*, *28*(3), 189–195.

B-9 Girbal, S., Jean, X., le Rhun, J., Pérez, D. G., & Gatti, M. (2015). *Deterministic platform software for hard real-time systems using multicore COTS*. Proceedings from the 34th Digital Avionics Systems Conference, Prague, Czech Republic.

B-10 National Science Foundation. (2014). *Single core equivalent virtual machines for hard real-time computing on multicore processors.* Chicago, IL: Sha, L., Caccamo, M., Mancuso, R., Kim, J. E., Yoon, M. K., Pellizzoni, R., Yun, H., … Bradford, R. Retrived from http://rtsl-edge.cs.illinois.edu/memcl/files/SCE_tech_report.pdf.

B-11 Bin, J., Girbal, S. G., Daniel, P., Grasset, A., & Merigot, A. (2014). *Studying co-running avionic real-time applications on multi-core COTS architectures*. Paper presented at the 7th European Congress On Embedded Real Time Software And Systems (ERTS). Toulouse, France. Retrieved from https://www.researchgate.net/publication/261721328_Studying_co-running_avionic_real-time_applications_on_multi-core_COTS_architectures

B-12 Cazorla, F. J. Quiñones, E., Vardanega, T., Cucu, L., Triquet, B., Bernat, G., Berger, E., … Maxim, D. (2013). PROARTIS: Probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems (TECS), 12*(2), 94:1–94:26.

APPENDIX C—ACTIVITIES AND DATA RECOMMENDED TO BE PRODUCED FOR COTS ASSURANCE

The following is based on EASA CM SWCEH-001 [1] to tentatively recollect all activities and results expected from highly complex commercial off-the-shelf (COTS). Activities are gathered under the main objectives that are generally established as part of development assurance in support to certification.

With respect to the COTS airborne electronic hardware (AEH) description of functional purpose and architecture content:

For example: Criticality (e.g., allocated development assurance level [DAL]), novelty (e.g., new technology, service history), functionalities (various modes of operation), internal architecture (including internal interfaces, safety mechanisms, and level of integration), structure (e.g., hierarchal breakdown and interconnections), input/output (I/O), and hardware-software interfaces (HSI).

With respect to the intended function(s) (per 14 CFR 2X.1301) provided by COTS AEH:

For example: Usage domain (UD) definition (used/unused functions, activated/deactivated, configuration settings, including default configuration settings), UD validation (versus system, safety, and software requirements), UD verification (reviews, analyses and tests), errata capture and analysis (impacts on intended function and safety).

With respect to the technical suitability (per DO-254 section 11.2.1.6) of COTS AEH to application:

For example: Characteristics (electrical and logical), timing performance (the worst case execution time), deterministic access to resources (execution, I/O, data, etc.), specific usages (e.g., robust partitioning, resources management, communication protocols), and environmental conditions (temperature, range, power consumption, and reliability).

With respect to proper functioning (per 14 CFR 2X.1309) without anomalous behavior:

For example: Failure modes and effects summary for highly complex COTS AEH for critical applications. Integrate failure rates within upper-level failure modes and effects analysis of the unit of equipment. Perform functional failure part analysis at device level if a reduction of the DAL is sought. Provide directions for common mode and cause analyses at system level.

With respect to operating performance (per 14 CFR 2X.1309) under environmental conditions:

For example: Validation and verification against upper-level requirements and HSI, additional re-verifications in case of changes in the COTS AEH, qualification to environmental conditions at the unit of equipment level, and links with system/software architecture context for critical functions.

With respect to ensuring continued airworthiness of usage conditions:

For example: Document product service experience gained and lessons learned (e.g., errata workarounds, usage limitations, and errata discovered); follow-up on stability, maturity, configuration status, and monitor change notices; and associated change descriptions, document change impact analysis, and managing COTS AEH obsolescence.

APPENDIX C—REFERENCES

1. EASA Report. (2012). Development Assurance of Airborne Electronic Hardware (CM-SWCEH-001).