RAPITA Systems
A DANLAW Company

WNDRVR

WHITEPAPER

# Mitigation of interference in multicore processors for A(M)C 20-193


**Daniel Wright**
Technical Marketing Executive, Rapita Systems

**Karl Thyssen**
Multicore Analysis Engineer, Rapita Systems

**Olivier Charrier**
Functional Safety Specialist, Wind River

**Stefan Harwarth**
Specialist Systems Architect, Wind River

# CONTENTS

**1.** Multicore certification in avionics **1**

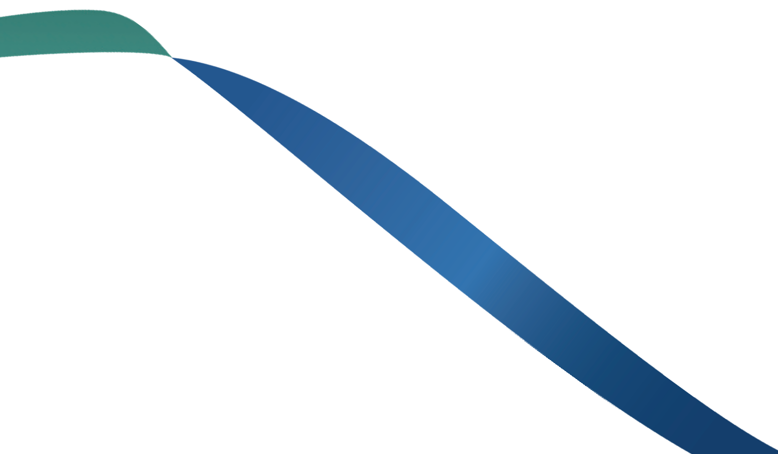
**2.** Interference and interference channels **2**

**3.** Identifying interference channels **4**
- 3.1 Platform analysis 5
- 3.2 Examples 6
- 3.3 What comes next? 7

**4.** Mitigating interference channels **8**
- 4.1 Hardware mitigation approaches 9
- 4.2 Software mitigation approaches 12
- 4.3 Hybrid mitigation approaches 15
- 4.4 Implementing mitigations 18

**5.** Verifying mitigations **20**
- 5.1 How to verify mitigations at the platform level 22
- 5.2 How to verify mitigations at the software level 23
- 5.3 Operating system platform verification 23

**6.** Conclusion **24**



# 1.

# MULTICORE CERTIFICATION IN AVIONICS

*With the embedded avionics industry's inevitable move towards the use of multicore processors for new projects, it is more important than ever to understand the certification landscape for multicore avionics systems.*

Increased adoption of multicore processors by the embedded avionics industry is being driven by ever-increasing demands for software functionality, improved SWaP (size, weight and power) characteristics, and increasing challenges in sourcing high-performance single-core processors. While using multicore processors for embedded avionics offers many benefits, doing so in DO-178C or ED-12C projects requires meeting the additional objectives of AC 20-193 (released January 2024; for DO-178C projects) or AMC 20-193 (released January 2022; for ED-12C projects).

Much of the challenge in meeting A(M)C 20-193 objectives hinges on understanding and mitigating multicore interference and providing evidence that the timing deadlines of hosted applications will always be met. In this paper, we do a deep dive on identifying interference channels, mitigation of interference and verification of mitigations as required to meet A(M)C 20-193's `MCP_Resource_Usage_3` objective, and we show how Wind River® and Rapita Systems solutions support this process.

***What is A(M)C 20-193?***

*A(M)C 20-193 guidance is a joint effort by the European Union Aviation Safety Agency (EASA) and Federal Aviation Assiociation (FAA).*

*It provides an acceptable means of compliance for showing that multicore processors (MCPs) used in airborne systems and equipment meet the necessary airworthiness specifications. This document is crucial for ensuring that MCPs, which are processors with multiple cores, operate safely and reliably in aviation environments.*

# 2.

# INTERFERENCE AND INTERFERENCE CHANNELS

*Multicore interference is the cornerstone of A(M)C 20-193 compliance. Interference can be caused by different cores in a multicore processor taking actions that interact with each other. This could include accesses to the same shared resource, synchronization events, coherency mechanisms and even thermal behavior.*

The impact of interference ranges from no noticeable effect to impacts on the execution behavior of hosted software, including properties such as average and worst-case execution time (WCET), predictability of timing behavior, and data coupling and control coupling.

Interference can originate from various sources on a multicore processor. A(M)C 20-193 call these “interference channels”, and define an interference channel as “a platform property that may cause interference between software applications or tasks.” Many different interference channels exist. These can come from multicore processors or peripheral devices on a platform. Multicore platforms tend to have more interference channels than you may first expect. Some of these are obvious sources of interference that can be easily identified from analysis of reference documentation (see *Platform Analysis* on page 5), while others are less obvious and may only be identified during testing (see *Platform Characterization* on page 22).

For discussion purposes, and as an aid to understanding where interference channels may come from and how they can be discovered, it can be helpful to categorize interference channels based on their properties. Throughout this paper, we will categorize interference channels as either Direct or Indirect.

**Direct interference channels** involve direct competition for resources between tasks or applications running on different cores. A common example of this is multiple cores sharing a cache, for example L2 cache, where one core can invalidate cache lines written by another core. This can cause cache misses, which have a performance penalty. We go into more detail on this example in *Identifying interference channels: Examples* on page 6. For a skilled engineer, many Direct interference channels can be intuitively discovered from a high-level understanding of the multicore platform and related devices. Analysis of technical reference documentation is needed to support this discovery and understand the technical definitions of the interference channel.

**Indirect interference channels** arise where hardware unpredictability can impact software behavior. This may be due to priorities or mode changes in hardware such as events triggering a cache policy change, or traffic triggering a routing change. An example of the latter arises when interconnect routing can depend according to traffic. On a ring buffer where multiple cores and devices are attached, there will be a shortest route between a given core and a device. If that route already has a lot of traffic, however, the system may reroute additional traffic to a different route, which may have a greater latency and more variability in execution time (**Figure 1**). Indirect interference channels are often more challenging to discover than Direct ones, and identifying them may require a more detailed understanding of the components in a multicore platform or devices and how they interact with each other. Analysis of technical reference documentation should support this discovery.

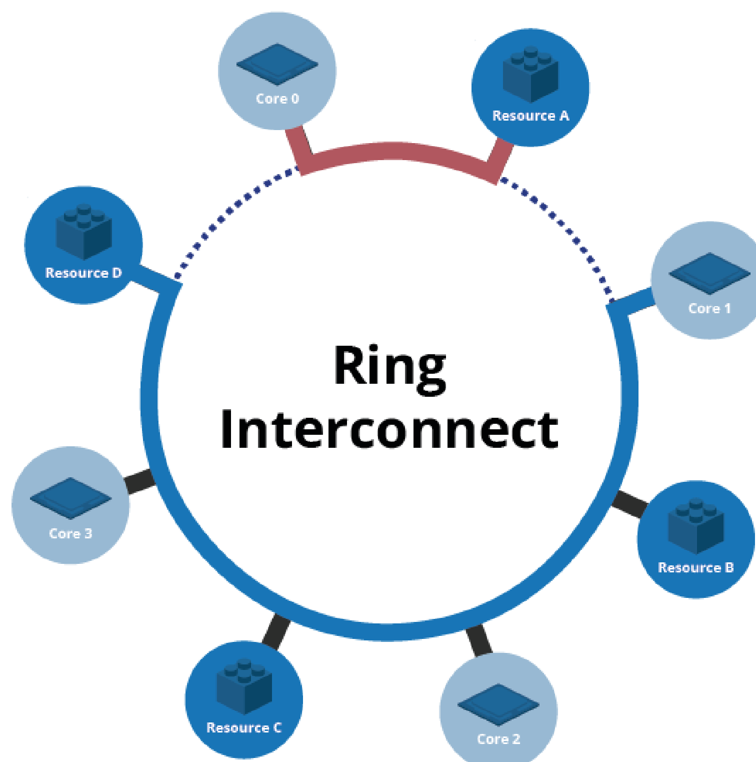



**Figure 1** – *Example of an indirect interference channel caused by contention for resources on a ring buffer. Due to accesses from Core 0 to Resource A, Core 1 is rerouted to take the shortest alternative route to access Resource D.*

3.

# IDENTIFYING INTERFERENCE CHANNELS

*Identifying interference channels is a key activity in A(M)C 20-193 projects.*

Interference is a central consideration in most A(M)C objectives, and objective `MCP_Resource_Usage_3` specifically asks a certification applicant to demonstrate that interference channels on a platform have been identified, and that any mitigations applied to those channels have been verified; see *Verifying mitigations* on page 20.

As the interference channels on a platform can have large effects on software performance and required verification effort, it is important to identify them as early as possible during an A(M)C 20-193 project. This is also important due to the scale of required activities – multicore platforms may include many more interference channels than you first think, with a typical platform having around 20-250 channels, depending on its complexity. Some of these may be easy to discover, while it may be more challenging to discover others (**Figure 2**).

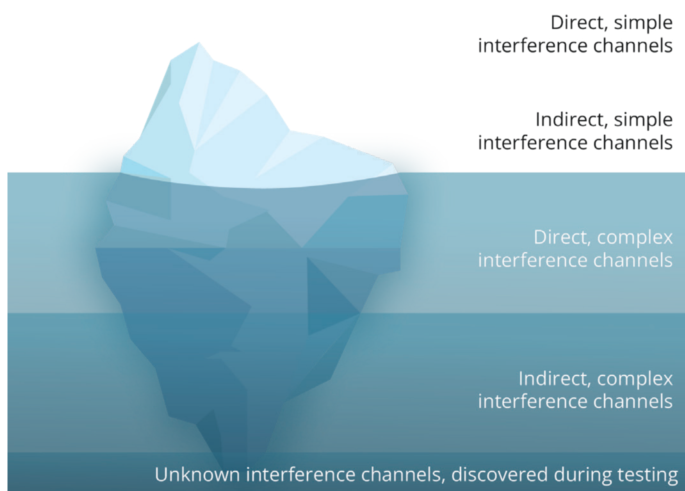



**Figure 2** – *Multicore platforms may have many more interference channels than you first think, and some may be difficult to discover.*

The hardware that is used in an A(M)C 20-193 project can have a huge impact on the interference channels in effect on a platform, on the impact of those interference channels, and on the required verification effort. Because of this, it makes sense to evaluate different platforms before embarking on formal activities in an A(M)C 20-193 project. Key elements of a platform that should be understood during such an evaluation include visibility of system interrupts and context switches, mitigation strategies (see *Mitigating interference channels* on page 8), and the availability of hardware event monitoring units (see *Platform Characterization* on page 22).

In this chapter, we will cover how you can identify interference channels (next section), present some examples of doing so (page 6), and briefly discuss what comes next (page 7).

## 3.1 How to identify interference channels (Platform Analysis)

If you are aiming for A(M)C 20-193 certification, then you should use a reproducible and well-defined process to identify interference channels on a platform. This process should be defined and included in your DO-178C/ED-12C planning documentation as per A(M)C 20-193's planning objectives.

Identification of interference channels requires a deep understanding of every component of a multicore platform. This can only be gained through analysis of detailed technical documentation about the behavior of a multicore processor and devices. To support A(M)C 20-193 compliance, you should ensure that this information is available and in sufficient detail for every element of the platform you intend to use.

Rapita Systems have identified interference channels on a range of multicore platforms with different processors and devices. Rapita identifies interference channels as part of Platform Analysis activities in Rapita's **MACH**[178] solution, which provides support for meeting A(M)C 20-193 objectives. **MACH**[178] includes processes that define how the analysis is performed, and templates to support writing DO-178C/ED-12C planning documents, including the Software Verification Plan.

The analysis involves first identifying hardware resources present on the platform, and then identifying and documenting the platform properties that can cause interference (interference channels) on each resource, as well as relevant information about those channels. This documentation includes a description of how each interference channel can cause interference between different applications or tasks, configuration settings that may affect each channel, references to the sources of information from which each interference channel was identified, and potential mitigations of each interference channel that have been identified during the analysis (see *Mitigating interference channels* on page 8).

In the previous chapter, we introduced terminology to categorize interference channels as being either Direct or Indirect to understand where interference channels come from and how they are discovered. For most multicore platforms, during an initial analysis, a range of Direct and Indirect interference channels will be identified and documented. Platforms are likely to have interference channels that can't be identified from an initial analysis, and which can only be discovered during verification and characterization of other interference channels (see *Platform Characterization* on page 22). When these are discovered, they are analyzed and documented.

## 3.2 Identifying interference channels: Examples

As an illustration of identifying interference channels, consider a hypothetical quad-core system with two levels of cache: L1, which is private to each core, and L2, which is shared by all cores (**Figure 3**). The L2 cache in the system is accessed using an interconnect shared by all cores through a shared cache controller. Many multicore processors share properties with this hypothetical system.

To identify interference channels on the platform, we need to understand in detail how our platform works by analyzing the technical documentation that describes the functionality of the cache. The following are just a few examples of possible interference channels relating to the cache architecture in this hypothetical system.

### *Example I: Invalidation of cache lines*

One interference channel we may identify on our platform is the invalidation of cache lines by other cores. Interference can result when one core places data in a cache line and another core replaces that data. Unable to retrieve cached data from the L2 cache as it has been replaced, the first core will need to access higher levels of the memory hierarchy to retrieve the data, and this comes with a performance penalty. This is a common interference channel associated with the cache architecture in our example.

### *Example II: Cache controller requests*

A less obvious example relates to cache controllers, which manage the requests being made to a cache. Cache controllers receive requests from cores, such as data linefills, instruction linefills, reads, and non-cacheable reads.
A cache controller is only able to service a limited number of simultaneous in-flight requests at once depending on its architecture and complexity. When multiple requests are made concurrently by multiple cores, a controller must serialize these operations. If sufficient requests are made, the queues and buffers that perform this serialization may become full, leading to new requests not being accepted. As this example involves direct contention for a resource (the cache controller), we'd class it as a Direct interference channel, but it is one that might not be identified from a preliminary analysis of interference channels.

### *Example III : Cache coherency protocols*

Taking a step down into the L1 cache, which is private to each core in our example, you might assume that there is no sharing of a resource, so no interference channels are present. However, a cache line may be shared between multiple L1 caches. If data is changed in one cache, it must be ensured that this change is broadcast to the other cores to ensure that no stale data is used. To support this, many platforms include cache coherency protocols (snoops) that are broadcast to other cores, allowing them to check whether the address of the modified data is also present in their private L1 cache. If the same address is present, it must be invalidated to ensure coherency, forcing the local core intending to use it to first access L2, which is associated with a higher latency. This is an example of an Indirect interference channel where performance can be affected not by direct contention for a resource, but by an indirect effect of the multicore architecture.

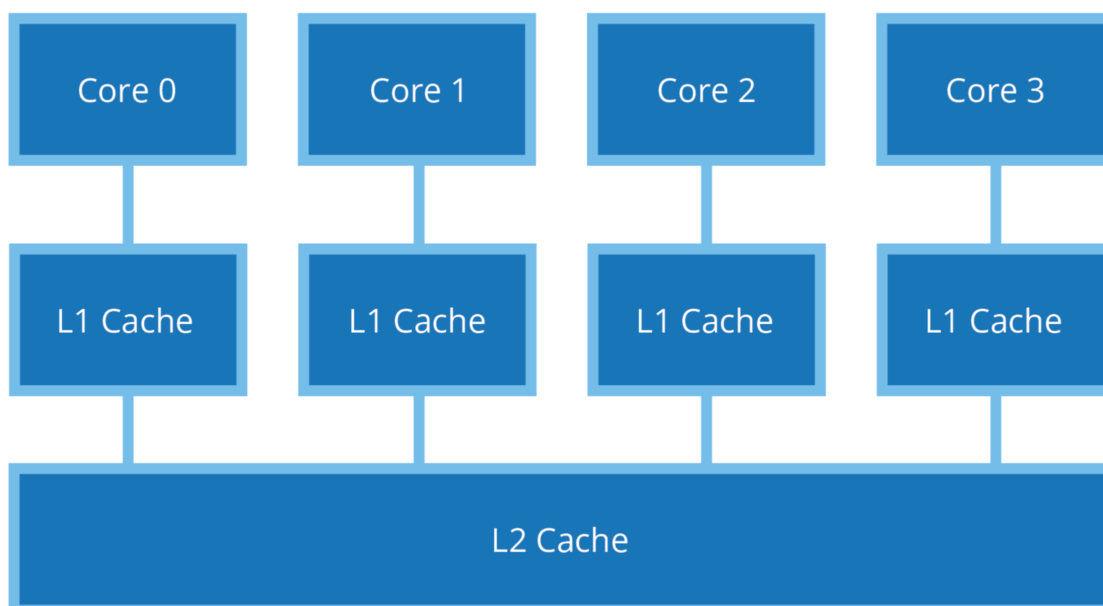



**Figure 3** – *A quad core system with private L1 and shared L2 caches. Example Interference Channels 1 and 2 may be present in the L2 Cache whilst the Example 3 can exist in the L1 caches, despite the fact that they are private.*

Examples 1 and 2 are Direct channels, while Example 3 is an Indirect channel using our definitions (see *Interference and interference channels* on page 2). For a given multicore platform, it may be possible to identify interference channels like this through analysis of reference documentation alone. On a real multicore platform, there would be many more Direct and Indirect channels that could be discovered from this type of analysis.

Real platforms will likely have channels that can't be detected through analysis alone, and which can only be discovered during testing (see *Platform Characterization* on page 22). At Rapita Systems, we have yet to encounter a multicore platform where no such interference channels were found. Examples of interference channels that can only be discovered through testing range in complexity from incorrect configuration register definitions to unintended features. One example of the latter that has been discovered by Rapita Systems is asymmetric bus arbitration on an old, widely-used multicore processor. On this processor, when all cores make accesses to saturate the bandwidth of the bus, core 0 will be treated preferentially, and this behavior can affect interference.

## 3.3 What comes next?

After you have identified the interference channels on a platform, there are two main approaches to achieving A(M)C 20-193 objectives.

The ideal approach is to find and implement a means to mitigate the interference channel and verify the mitigation according to A(M)C 20-193's `MCP_Resource_Usage_3` objective. For the purposes of this paper, and in alignment with A(M)C 20-193, we define an interference channel as having been mitigated if there is no observable impact from the channel on performance of the multicore system.

We will look at different approaches for mitigating interference channels in *Mitigating interference channels* on page 8, and at how mitigations can be verified in *Verifying mitigations* on page 20.

It is not always possible to mitigate an interference channel. Shared access to devices will almost always be required, including access to devices that cannot be partitioned easily on the hardware level, such as network interfaces. In these cases, interference can be managed through configuration or design to have better control over the level of actual interference at runtime. While this will not negate the need to verify the software's performance, it will allow the impact of the interference channel on performance to be minimized, making it more likely that the software will meet its performance requirements. This complex topic, falls out of the scope of this paper and may be discussed in a future paper.

If you have not mitigated an interference channel, then you will need to verify the software's timing behavior and data coupling and control coupling with respect to that interference channel to meet A(M)C 20-193 objectives `MCP_Software_1` and `MCP_Software_2`.

Mitigating a channel and verifying the mitigation is the preferable approach to take as, in addition to negating the impact of the interference channel on performance and making the system more reliable, it also negates the need for additional verification to meet A(M)C 20-193's `MCP_Software_1` and `MCP_Software_2` objectives, therefore reducing the overall verification effort.

# 4.

# MITIGATING INTERFERENCE CHANNELS

*Mitigating interference channels is crucial in the development of DO-178C/ED-12C multicore software. Mitigation is necessary to improve worst-case performance and reliability of the software, and it allows an applicant to reduce the scope of activities required to meet A(M)C 20-193's MCP_Software_1 and MCP_Software_2 objectives.*

> *For the purposes of this paper, an interference channel is defined as having been mitigated if there is no observable impact from the channel on the performance of a multicore system.*

The importance of mitigation should not be understated. Multicore projects are complex, and the verification activities required to meet A(M)C 20-193 objectives are expensive. The more that the performance and reliability of the software can be improved, and its verification simplified, the better.

There are different approaches to mitigating interference channels, and these can broadly be grouped into the following approaches:

- **Hardware-based approaches** – These are based on hardware capabilities and design. They may be configured in hardware, but they are often configured during initialization by a layer in the platform software, such as an RTOS or bootloader. Generally, these approaches are easier to implement than software and hybrid approaches, and they are more frequently used.
- **Software-based approaches** – These are based on the way the software is architected and designed, compiled and linked. This can also include the use of specific mitigation code that operates during runtime.
- **Hybrid hardware/software approaches** – These include elements of both hardware and software-based approaches.

All mitigations come with some trade-offs; usually, a sacrifice of average-case performance is needed to achieve better predictability and improved worst-case performance. Some mitigations may also impact the complexity of software, such as cache/bandwidth partitioning implementations or applying constraints on software architecture. When you first choose which mitigations to apply, you may choose to avoid these mitigations due to their expected impact on average-case performance.

If, during analysis of your software performance, you find that your software is particularly sensitive to a related interference channel, or if your software fails to meet its timing deadlines, you may choose to apply a mitigation that you had previously chosen not to use. As a result, you'll need to repeat some verification activities. Because of this, automation, traceability, and applying a well-defined and repeatable procedure is essential for efficiency. We cover this further in *Verifying mitigations* on page 21.

In the upcoming sections, we will share examples of approaches for mitigating interference based on the categories listed above.

Then, later in the chapter (page 18), we will discuss considerations for implementing mitigations, including the pros and cons of different approaches, and how much rework should be expected when selecting and verifying mitigations.

## 4.1 Hardware mitigation approaches

Some approaches to mitigating interference channels can broadly be categorized as being related to the hardware and its configuration and driven by the hardware design. These approaches are discovered during Platform Analysis along with other mitigation strategies (see *Platform Analysis* on page 22).

Common hardware mitigation approaches include the following:

- Disabling cores
- Disabling devices or features
- Avoiding the use of complex devices
- Hardware partitioning of shared resources

Some of these approaches have trade-offs. For example, applying an approach may be helpful for mitigating interference channels, but this may come at the expense of affecting average-case performance, the ease of analyzing interference channels, predictability in performance, or complexity.

### 4.1.1 Disabling cores

One approach you may consider to mitigate interference channels is to deactivate all but one core of a multicore processor to make it behave almost like a single-core processor (**Figure 4**).

You may assume that disabling all but one core will automatically mean that you no longer need to follow A(M)C 20-193 guidance, but this is not necessarily the case, and additional activities will always be required even if this strategy is used.

For example, if cores are disabled by hardware configuration settings, you will need to provide evidence that the cores are disabled and will remain so during operation, for example in the case of a single event upset, as per A(M)C 20-193's planning objectives, which provide guidance on deactivating cores.

Note that there are many other approaches to disabling cores. Some of these may eliminate or reduce your obligations for A(M)C 20-193 compliance, while others will not. Your approach to disabling cores should be discussed with your certification authority to determine the extent to which A(M)C 20-193 objectives apply to your project.

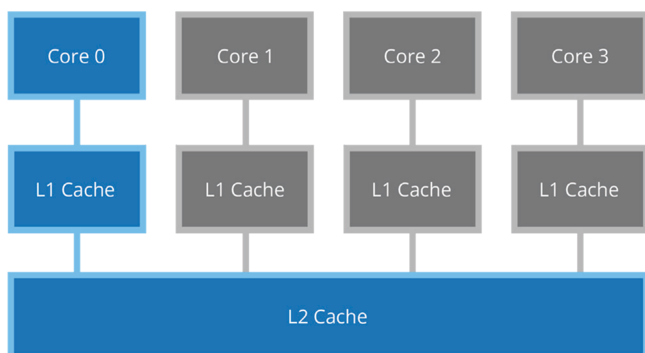



**Figure 4** – *Deactivating all but one core of a multicore platform can avoid the need to meet A(M)C 20-193 objectives entirely, depending on the method of deactivation.*

> ***Disabling cores***
>
> *Disabling cores for multicore processors and what that means for A(M)C 20-193 compliance is a complex topic.*
>
> *Rapita Systems have developed a White Paper dedicated to this.*
>
> *For more information, contact Rapita Systems.*

### 4.1.2 Disabling devices or features

If a device (such as a level of the memory hierarchy or a peripheral) or feature (such as cache stashing) is not needed, then disabling it can be an effective way of mitigating interference channels (**Figure 5**). This can be especially helpful where a device or feature has many associated interference channels.

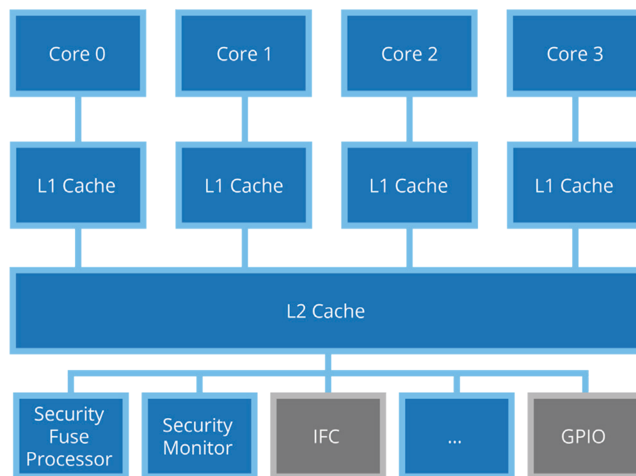



**Figure 5** – *Deactivating devices or features can mitigate all interference channels associated with them.*

Selecting which devices or features to disable depends on the architecture and functionality – not only of the device itself, but also of the platform. Disabling a level of cache such as L2, for example, will mitigate all interference channels associated with it, but it will also cause all accesses that would otherwise hit L2 to access main memory instead. This might have a pathological impact on interference channels associated with the buses or interconnects between L2 and main memory, as well as those associated with main memory.

The efficacy of trade-offs of performance for analyzability and complexity for this mitigation therefore depend entirely on the use case. If you are not going to use a feature, disabling it comes with no trade-off.

For example, if you are not going to use cache stashing, which allows cores or devices to preload the cache with data it "may" later need, you might as well disable the feature to reduce the scope of analysis activities associated with meeting A(M)C 20-193's software objectives.

Note that disabling a device or feature does not eliminate the need to perform verification activities, but it changes the activities that must be performed. A(M)C 20-193 requires evidence to be provided that disabled devices or features are disabled and will remain so during operation. These verification activities, however, require much less effort than those needed to analyze and characterize interference channels and their impact on hosted software.

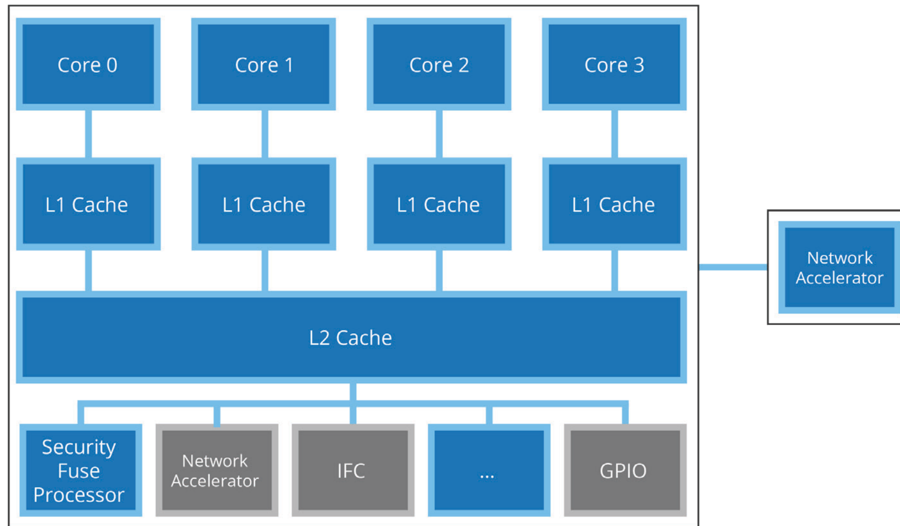



**Figure 6** – *A complex device will often have a greater number of interference channels associated with it than a simple one. Replacing a complex device with a simpler off-chip one mitigates all interference channels associated with it and thereby simplifies the effort required to meet A(M)C 20-193.*

### *4.1.3 Avoiding the use of complex devices*

In some cases, required functionality can be achieved through the use of simpler or more complex devices (**Figure 6**), and using a less complex device may help mitigate interference.

For example, rather than using deeply embedded on-chip peripherals with many interference channels, you may instead opt to use off-chip (I2C, SPI, PCIe etc.) peripherals with fewer potential interference channels.

As every interference channel adds additional A(M)C 20-193 activities, simplifying a system in terms of interference channel analysis will help reduce effort required for certification of the software, though it may increase effort required for certification of the hardware (DO-254).

### *4.1.4 Hardware partitioning of shared resources*

For many interference channels related to direct contention on shared resources such as a cache, main memory, or bandwidth, the simplest approach to mitigating the interference channels is to partition the resources.

Hardware partitioning is a technique whereby isolated access is enforced on non-overlapping parts of a resource. This usually includes the isolation of faults within each part and can reduce verification effort. Using the L2 cache as an example, this partitioning mechanism can mitigate interference channels associated with cache misses due to direct contention for cache lines from multiple cores (**Figure 7**).

Different partitioning methods have different performance and complexity implications. Applying this partitioning mechanism to the L2 cache would not necessarily mitigate all interference channels associated with the L2 cache, as there may be many other interference channels not related to direct cache line contention. Examples of this include internal buffers and queues that hold the lines being accessed. It may be possible to partition these, often through way partitioning, or it may not be possible due to lack of accessibility, customizability or even visibility.

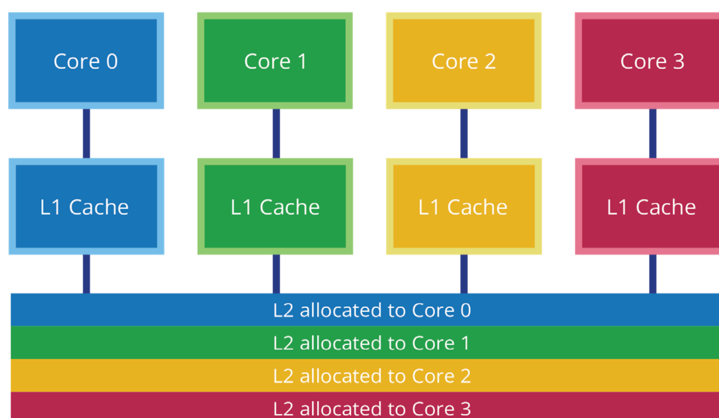



**Figure 7** – *Partitioning shared resources can mitigate interference channels; in this example, the L2 cache has been partitioned, preventing cores from evicting the data of other cores.*

## 4.2 Software mitigation approaches

Some approaches to mitigating interference channels can broadly be categorized as being based only on the software that executes during runtime. This can include the definition of a particular software architecture; the choice of programming constructs, standards or scheduling; or the implementation of a specific runtime behavior.

Before you begin to design and write software, it is important to consider the interference channels that may be present on the system (as identified by *Platform Analysis*, see page 5), and which of these you will mitigate by software approaches and how. After all, changing the software design, code or integration later in a project can be expensive.

Common software mitigation approaches include the following:

- Applying time partitioning
- Restricting resource usage to only one execution context
- Using a client/server architecture
- Avoiding the use of shared memory
- Limiting data exchange to scheduling boundaries
- Selecting programming constructs and standards
- Segregating criticality

As in A(M)C 20-193, the term *application* is used in this document to describe a software component that implements a set of functionality and that can be integrated into the system and executed without depending on other applications. When executing an application in the context of an operating system (OS) platform with a protected address space, the term *process* is often used, but bare-metal software or virtual machines in a hypervisor can equally be seen as applications according to this definition.

The term *task* when used in this document describes a single thread of execution flow within an application that can be scheduled to run on one of the cores of a multicore system. Applications can have one or more tasks, and these can all run on the same core, or on different cores.

### *4.2.1 Applying time partitioning*

Time partitioning is a technique with which software applications are isolated so that each application has no impact on the timing performance of another application. This is exceptionally challenging to implement in a multicore system due to the coupling effect of interference. Concurrent applications are coupled through the interference they generate and are sensitive to, even if there are no functional data or control dependencies within the software itself.

"Robust time partitioning" is referenced throughout the A(M)C 20-193 objectives and highlighted in the `MCP_Software_2` objective. A(M)C 20-193 assert that robust time partitioning would be achieved if no interference channels can cause applications to consume more than their allocated time resources (see box below for the full definition). To guarantee this for a system, it would be necessary to mitigate every interference channel such that there are no interference channels active across the entire platform. This is impossible to do in practice.

> ***Full A(M)C 20-193 definition for robust time partitioning***
>
> *Both AC 20-193 and AMC 20-193 state:*
>
> *"Robust time partitioning (on an MCP): this is achieved when, as a result of mitigating the time interference between partitions hosted on different cores, no software partition consumes more than its allocation of execution time on the core(s) on which it executes, irrespective of whether partitions are executing on none of the other active cores or on one, more than one, or all of the other active cores."*

Applying time partitioning can still, however, have many merits when it comes to mitigating interference channels. Allowing only a single task to use a particular shared resource during a defined amount of time can mitigate potential interference from shared accesses.

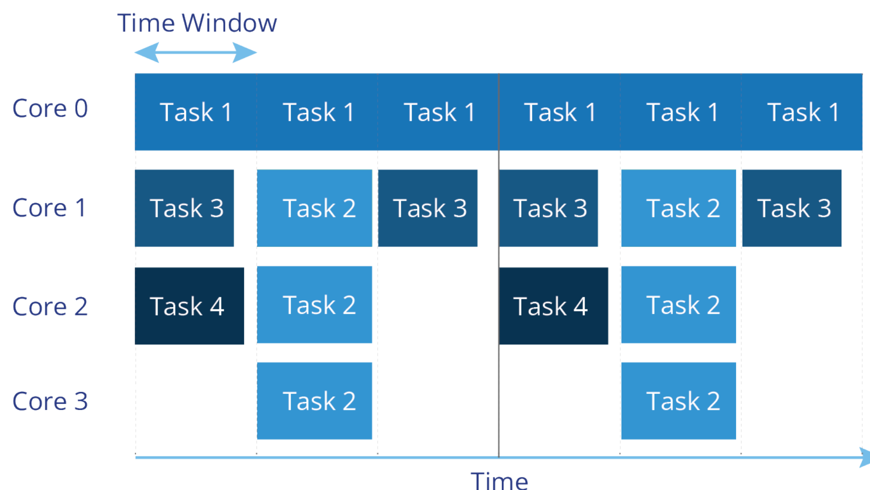



**Figure 8** – *Restricting the execution time of tasks to time windows can mitigate interference channel*

For example, for software with multiple tasks with heavy access patterns to main memory, time partitioning could be applied by scheduling these tasks in separate time slices to avoid concurrent accesses (**Figure 8**). This mitigation requires a timing event to trigger the start and stop of the thread. This timing event may make use of a shared resource that could exercise one or more interference channels. Additionally, you can architect your software to make time partitioning more effective by splitting functionality that accesses multiple resources into more granular tasks, each of which accesses fewer resources. For example, if you have a software component that performs read, process and write functionality, you could split this into three tasks, each with its own resource access profile. This makes it easier to time slice the software to take account of resource usage patterns. Time partitioning is typically implemented by allocating periodic time windows during which a particular task or application is executed.

While time partitioning can be an effective mitigation strategy, it constrains the scheduling and/or software architecture. Applying time partitioning to minimize concurrent use of shared resources requires a method of synchronization across cores. Implementing this by hand for a project can be expensive, but it is usually available out of the box in OS platforms such as VxWorks® and Helix™ Virtualization Platform.

### *4.2.2 Restricting resource usage to only one execution context*

In cases in which interference channels are related to just one resource, restricting access to this resource to only a single task on a specific core can be an effective mitigation. Implementing this mitigation can require different activities during software architecture, design, implementation, or verification to ensure that the resource is only used by one task on one core.

Some of the following mitigations such as *Using a client/server architecture* below and *Applying space partitioning* on page 16 can support this mitigation approach by controlling access to a restricted resource by other tasks, or guaranteeing exclusive access through hardware protection. While this can be an effective mitigation, the challenge of using the approach is in identifying the resources and interference channels on which it can be applied.

### *4.2.3 Using a client/server architecture*

It may be that the hardware operations that govern the mediation and serialization of accesses to a given resource are poorly or not at all documented, making them difficult to characterize or mitigate. Using a client/server model in your software architecture to mediate and serialize accesses to resources, thereby avoiding any hardware mechanisms with associated interference channels, mitigates interference from access serialization (**Figure 9**).

You may, for example, implement a storage server application that mediates and serializes accesses to non-volatile memory (NVM) for all applications. This form of software-mediated access to shared devices lets you tightly define how serialization occurs rather than leaving it at the mercy of hardware, and it can mitigate interference. While this mitigation can avoid the use of hardware features that are difficult to analyze, using it will often come at the expense of increasing the complexity of the software architecture.

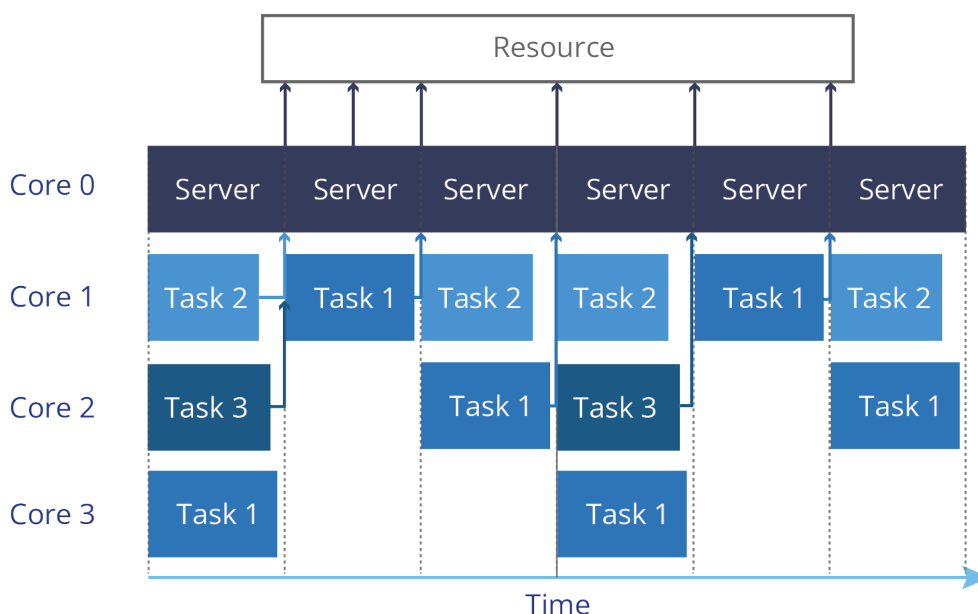



**Figure 10** – *Client/server architecture mediates and serializes accesses to resources. This avoids any hardware mechanisms with associated interference channels.*

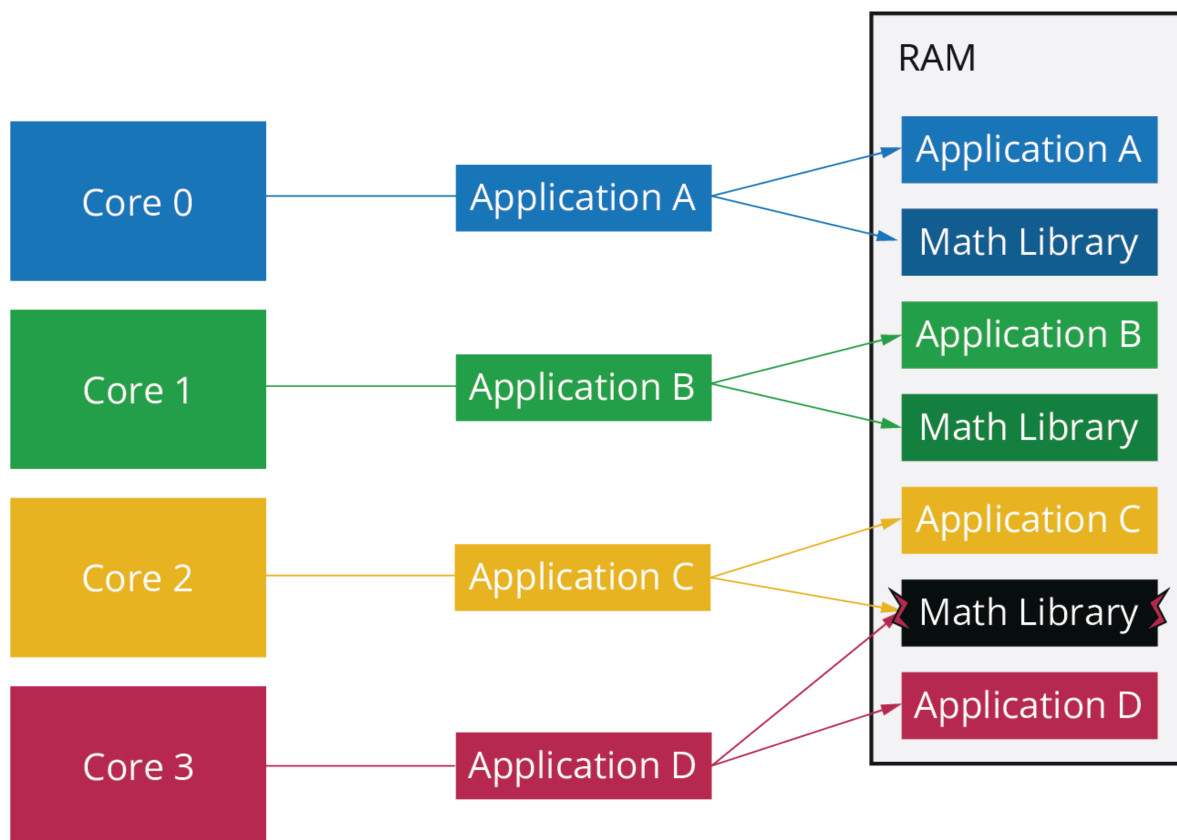



**Figure 10** – *Accessing separate copies of the Math Library in RAM for applications on core 0 and 1 mitigates interference channels related to shared memory access*

#### *4.2.4 Avoiding the use of shared memory*

Accessing the same memory pages from tasks on different cores concurrently is a source of interference analogous to some of those discussed previously. This can be mitigated by software design, for example by setting up separate copies of the OS components and libraries for each core, or by preventing allocation of the same memory pages from tasks on different cores (**Figure 10**).

However, exchanging data between, or synchronizing tasks on different cores typically requires memory pages to be shared or copied between cores, which can exercise interference channels. Restrictions on the API for data exchange can be used as a mitigation on the software level, including limiting data exchange to certain time windows as described in *Limiting data exchanged to scheduling boundaries* (right).

It is also possible to apply hardware-based mechanisms to add a protection level (e.g. the MMU or IOMMU). This hybrid approach can help support verification and validation activities and is described in *Applying space partitioning* on page 16.

#### *4.2.5 Limiting data exchange to scheduling boundaries*

Limiting data exchange between tasks to occur only at scheduling boundaries can mitigate interference. If multiple tasks execute concurrently with a high degree of coupling through communication or shared data, the sections that handle this communication will be particularly sensitive to associated interference channels. Through this coupling, interference effects that impact the availability of data can have cascading effects on the timing behavior of other tasks and applications.

Limiting data exchange to the end (writing/sending) and beginning (reading/receiving) of time windows ensures that the data will be available at the beginning of the next window with no coupling of timing behavior. This means that any interference channels exercised by this data exchange are mitigated as they can no longer impact the timing behavior of the software.

As well as having a positive impact on performance, this approach may also make multicore data and control coupling verification (as required by A(M)C 20-193's `MCP_Software_2` objective) easier.

### 4.2.6 Selecting programming constructs and standards

It is possible to mitigate some interference channels by careful selection of programming constructs and standards used in a project. Examples of this include restricting the use of shared memory or, for some niche interference channels, avoiding particular instructions such as `memset` or `icbi` (instruction block cache invalidate, which is available in some PowerPC® architectures). The policies that control cache behavior generally exist to improve average-case performance. As such, it may be that particular access patterns trigger policy changes. For example, if the default policy is a write-back mode, where write accesses from the CPU write the line into cache and then write it back to main memory, this generally caters well for quick repeat accesses to the newly written line.

However, if a lot of memory is written in a single transaction, this behavior may cause a performance penalty as the data could more efficiently be written directly to main memory. In this situation, the cache policy may automatically change to "read-allocate" mode, where cache lines are only filled when a read access is made, and writes instead write directly to main memory.

This cache policy change could have catastrophic consequences for other cores using the cache, and as such this is an (Indirect) interference channel that may need mitigation. Some ways to mitigate this interference channel include preventing cache policy changes or restricting the use of software behavior that may trigger the policy change, such as preventing the use of `memset` to write to large contiguous blocks of memory.

### 4.2.7 Segregating criticality

Naively, many projects approach the migration of single-core applications to multicore by attempting to separate applications based on Item Development Assurance Level (IDAL) on a per-core basis. While this seems sensible at first as it can maintain any space partitioning applied to the single-core counterpart, this makes it possible for applications of lower IDAL to generate interference that impacts applications of high IDAL on different cores. This can bring complex devices or functionality, and associated interference channels utilized by lower IDAL software into scope for analysis according to A(M)C 20-193.

It is therefore generally advisable to allocate functionality relating to different IDALs per time window rather than per core (**Figure 11**). Using time partitioning to allocate a time window per IDAL means that only the interference channels that relate to the usage domain of the high criticality application can impact its execution. This can reduce the scope of required certification activities as it can be considered a mitigation by design or avoidance of interference channels.

## 4.3 Hybrid hardware/software mitigation approaches

Some approaches to mitigation involve making choices or using methods in both hardware and software design and configuration. In these cases, neither hardware nor software methods alone are sufficient to implement the mitigation, and tight coupling between the hardware and software methods is needed to do so.

Some of these methods can be seen as extensions to the hardware or software mitigation approaches we discussed previously. The general comments made previously for both hardware and software-based mitigations also apply to hybrid mitigations.

Common hybrid hardware/software mitigation approaches include the following:

- Applying space partitioning
- Limiting accesses to shared resources
- Interrupt and exception handling
- Cache coloring

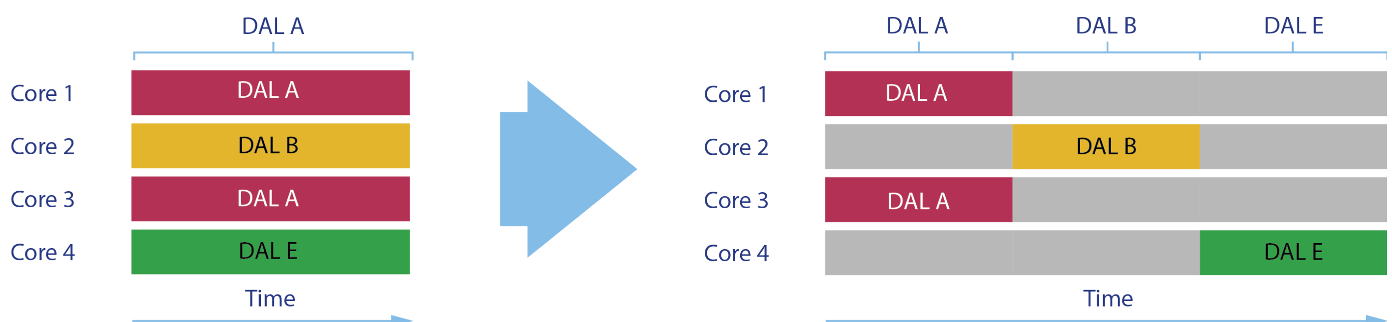



**Figure 11** – *While it may seem sensible to segregate functionality relating to different criticality per core, it's generally advisable to segregate criticality per time window instead*

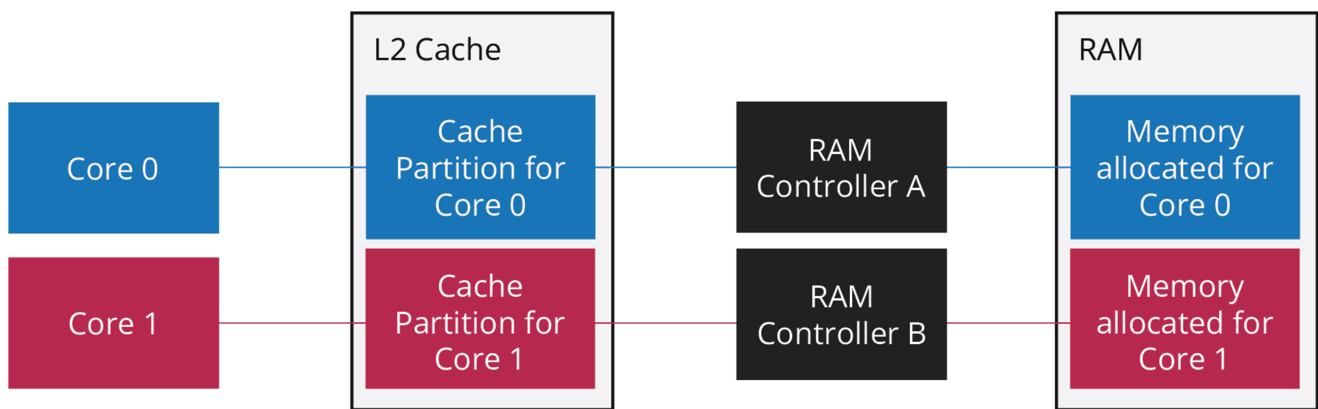



**Figure 12** – *The combination of cache partitioning and the use of certain RAM addresses can mitigate interference in the memory system through space partitioning.*

### *4.3.1 Applying space partitioning*

The extent of multicore interference in the memory system is related to the concurrent access from tasks on different cores. Common examples of interference channels in the memory system include cache snooping when accessing the same memory page, cache line eviction and memory controller bandwidth saturation. Powerful OS platforms such as VxWorks and Helix Platform can work around this by configuring the MMU to provide a contiguous view of virtual memory pages for each task or application, while the physical memory mapping is set up in a way to partition the memory resources and mitigate the related interference channels (**Figure 12**).

The availability of IOMMU and device virtualization goes one step further by partitioning accesses from peripheral devices to the memory system as well, and system-on-Chip (SoC)-specific partitioning features may provide similar options for on-chip devices.

### *4.3.2 Limiting accesses to shared resources*

When different applications can make accesses to shared resources concurrently as well as sequentially, interference can result, either immediately or when a threshold is reached. If a particular queue or buffer has a limited capacity and can only handle a limited throughput of accesses per second, this can be saturated by requests from multiple cores.

A memory controller, for example, can only handle a limited number of concurrent in-flight accesses. If the number of requests exceeds this number, the accesses must be serialized with some arbitration.

In the worst case, it could be that these queues and buffers are entirely saturated, and any further requests cannot be added.

Limiting accesses to devices or peripherals can mitigate these interference channels. This can be achieved in different ways. For example, configuring the MMU or IOMMU to only allow accesses from a specific application in each time window (see *Applying time partitioning* on page 12) can mitigate interference channels relating to bandwidth or access serialization. Accesses to or usage of a resource could also be budgeted for and monitored using hardware event monitoring units. If done correctly, this could ensure that thresholds are never exceeded. However, such mitigations can range in complexity and required verification effort, and using them can limit the available functionality considerably.

### *4.3.3 Interrupt and exception handling*

Even when following best practices by using polling instead of interrupts in a safety-critical system, interrupts will still be required to implement time-driven events or handle exceptions on the software level. Multicore interference occurs when the application that triggers the interrupt or the fault leading to an exception is running on a different core such as **(1)** in **Figure 13**, where an external device interrupt is targeted at the application on core 1, but is delivered to core 0 by the interrupt controller

The OS platform can provide system-level interrupt management by directing interrupts to the application's core with hardware support as in the case of **(2)** in **Figure 13**, and by masking device interrupts when an application is not scheduled. Time-based events can be reduced by adopting a tickless OS kernel architecture, such as that available in Helix Platform.

Another source of multicore interference is less obvious. When inter-process communication APIs are used in a multicore system, these can impact the execution of the destination task. For example, if data is sent to a task using pipes or message queues, this is likely to incur an immediate execution time penalty on the core receiving the communication as shown by **(3)** in **Figure 13**. Processors usually implement cross-core signaling with low interference through efficient doorbell or inter processor interrupts (IPI).

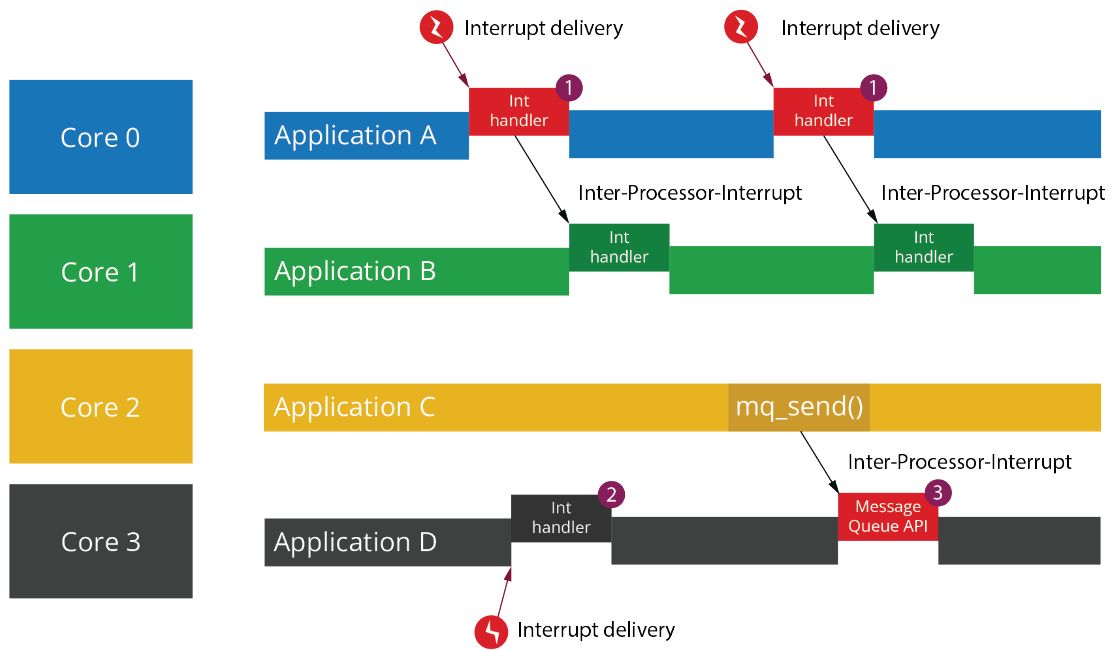



**Figure 13** – *Interrupts can trigger interference in different ways, as shown by the red boxes for cases (1) and (3).*

However, the complex actions in the API and scheduler that follow the signaling will increase the load on related interference channels, and a failed task may flood other cores with uncontrolled IPI load. Mitigation strategies include limiting cross-core signaling to predefined time windows similar to *Limiting data exchange to scheduling boundaries* on page 14. However, application developers may need to consider the lag when communicating across different cores. The ports and channels of ARINC 653 are a good example for API semantics that can be used with both polling and IPI-based signaling.

### *4.3.4 Cache coloring*

Cache coloring is a software partitioning method based on hardware characteristics that can be used to mitigate concurrent use of shared cache resources by multiple applications.

With cache coloring, the MMU is configured to map virtual addresses to physical addresses based on the relationship between cache lines and memory addresses (**Figure 14**). By mapping contiguous memory of an application to a set of physical memory pages that all end up in the same cache line of the shared cache, there is no concurrent cache eviction when other applications are similarly aligned to different cache lines. This is different to the cache coloring used for performance-optimized systems, where the number of cache lines used for each application is maximized to have the best utilization of the cache system.

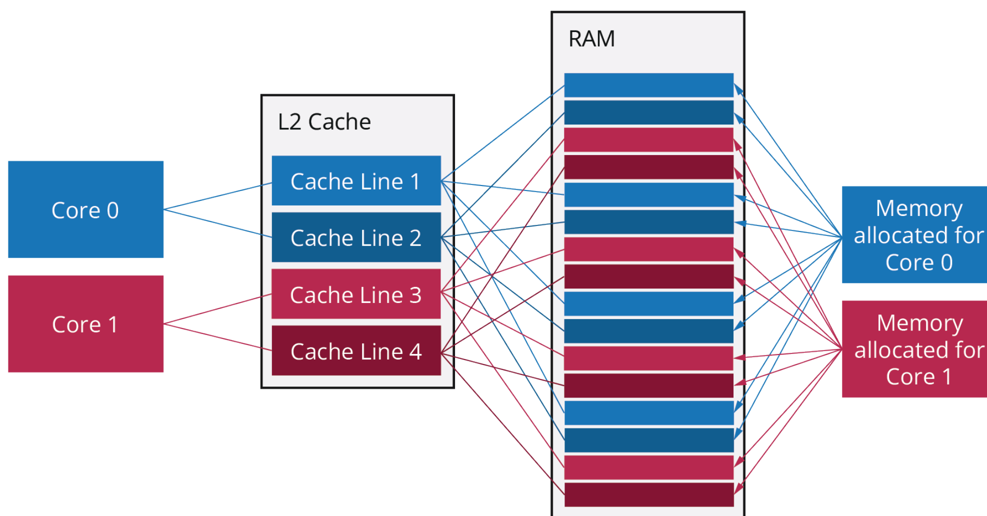



**Figure 14** – *Mitigation of cache concurrencies by allocating memory for Core 0 and 1 to addresses that are associated with different cache lines results in "cache-colored" memory usage.*

## 4.4 Implementing mitigations

Mitigation strategies are identified during *Platform Analysis* (see page 5). During this analysis, multiple potential mitigations for an interference channel may be identified. The mitigation strategies you choose to use may have implications on the configuration of the platform, or on the software architecture and design.

It is often easier to implement hardware configuration changes than software constraints, and mitigation strategies that involve configuration changes have the benefit of being verifiable without the software being available. This allows you to remove interference channels from the scope of performance verification early in a project. Strategies that involve making constraints to the software design, however, may be more robust with respect to change than those that involve configuration changes.

In cases where you choose to use a mitigation strategy and later find, for example, that it cannot be verified, is too complex, or has too high an overhead, you may need to explore using other mitigation strategies instead.

The cost of making changes to mitigations that involve configuration changes may be higher than other methods, as they may require you to repeat any platform characterization activities you have done previously (see *Platform Characterization* on page 22).

Whether you are applying hardware, software or hybrid strategies, the OS platform that you use can have a large impact on the ease of implementing and verifying a mitigation.

While the hardware mitigations described in *Hardware mitigation approaches* on page 9 are all statically configured or initialized at boot time, A(M)C 20-193 still requires that their status is monitored at runtime, for example by using safety monitoring software or a supervisor that prevents modification of the configuration at runtime.

For mitigations that are implemented within the software, the complexity of multicore processors and SoCs with many different internal and external devices requires a powerful software layer to be able to mitigate interference using the approaches we discussed above. State-of-the-art RTOSes such as VxWorks and the Wind River hypervisor contained in Helix Platform can provide the necessary features, and with their versatile configuration options, the most appropriate approaches can be selected to address the project's requirements for safety and performance.

The use of a multi-tiered OS architecture with a hypervisor and independent guest RTOS kernels can bring significant benefits when it comes to implementing interference mitigation approaches. Multi-tiered architectures offer clearer separation and increased isolation of applications in a multicore system, which makes it easier to demonstrate compliance to A(M)C 20-193 objectives, whereas monolithic multicore OS kernels tend to carry complex internal data and control couplings to implement device drivers, inter-process communication and stacks for networking or file systems. Implementing applications for a single-core RTOS or runtime environment, and then using distinct communication features across different applications and cores can greatly simplify A(M)C 20-193 verification. The abstraction between using a particular hardware resource and its functional usage through architectural standards like such as POSIX®, AUTOSAR or ARINC 653 is implemented in different layers of frameworks, RTOS, hypervisor and device drivers. It is usually opaque how a specific API call on the application level will exercise interference channels within all of these layers, but this must be taken into account during A(M)C 20-193 verification.

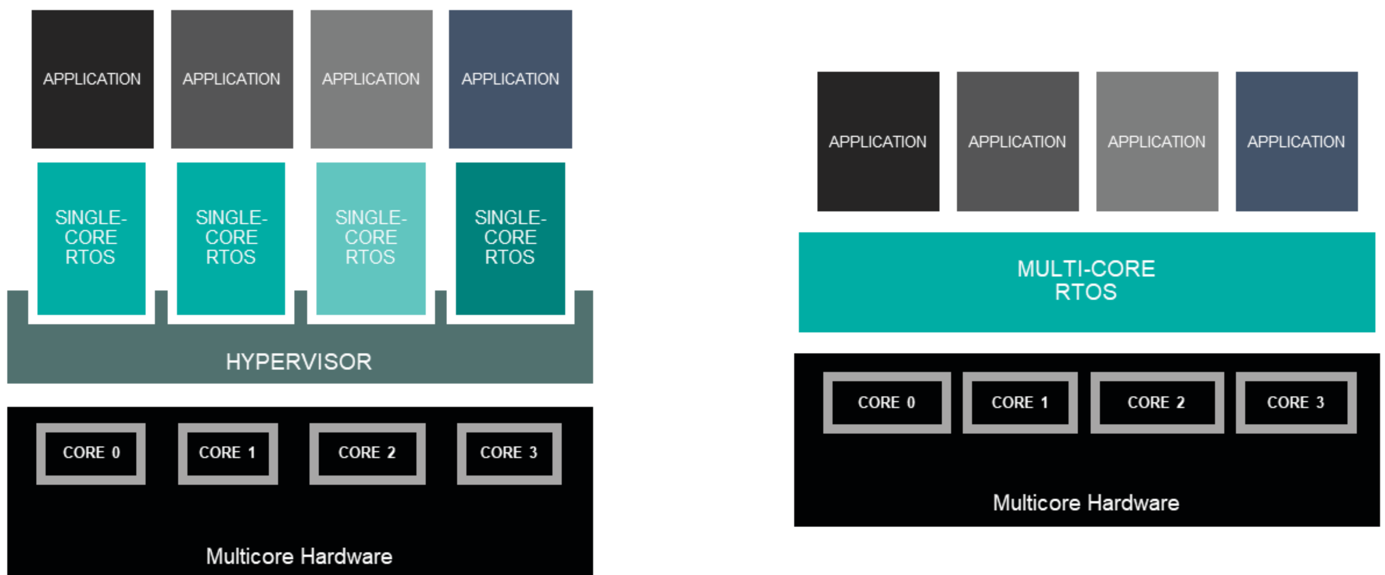



**Figure 15** – *Multi-tiered OS architecture versus monolithic architecture with single multicore OS.*

Common examples include:

- The OS platform layers make use of the MMU and IOMMU to introduce usage of cache and memory controllers that is different to the application software view of memory. This happens when the RTOS maps application memory to different physical memory areas, and the hypervisor even adds a second level of indirection, so that interference on the memory and caches cannot be managed from the application level.
- Usage of filesystem or network communication stacks can exercise interference channels in a non-obvious way. While this is directly traceable to file I/O or socket API calls in the application code, the mechanics of how the OS and runtime layers maintain the global state of the filesystem or network stack across different cores requires detailed analysis with regard to multicore interference.
- Other application APIs are also likely to exercise interference channels, for example when a semaphore released by a task on one core causes cross-core signaling and rescheduling of pending tasks on different cores, or when pipes and message queues pass data between cores using shared memory. In this case, the hardware interference channel is amplified by the functional interference due to the runtime internals and control coupling across cores.

While software is highly configurable, it is critical to analyze how different configuration settings affect multicore interference, as well as the overall performance characteristics of the system. In particular, the use of software and hybrid mitigations requires intimate knowledge about the software design and how it utilizes the underlying hardware.

This challenge can be overcome with pre-verified toolboxes that already implement various mitigation strategies, which is exactly what commercial OS platforms such as VxWorks and Helix Platform provide. The Wind River approach is to offer commercial off-the-shelf DO-178C certification evidence data packages for the OS platform that cover an entire processor family — such as Armv8-A or Intel® x86 64-bit — and come with formal guidance on how to integrate with specific hardware and application software. The design and integration documentation gives the information needed to open the black box and perform a full analysis of how the application's use of API and external interfaces will exercise the interference channels.

This out-of-the-box approach solves an important issue in the design process, as most of the software-related mitigation strategies have a direct influence on the design of the application or the system such as splitting functionality into different time windows or limiting data exchange between tasks.

# 5. VERIFYING MITIGATIONS

*After implementing a mitigation, you will need to verify it to meet A(M)C 20-193's MCP_Resource_Usage_3 objective.*

Your approach to verifying mitigations should have the following properties:

- It should be reproducible and well defined, and included in your DO-178C/ED-12C planning documentation as per A(M)C 20-193's planning objectives early in your project. This will support your DO-178C/ED-12C and A(M)C 20-193 compliance activities, and reduce rework costs.
- It must include a mechanism for measuring the performance of the system, both when there is no interference and when there is interference present on the system. Your approach to measuring performance will need to be measurement-based as no computational approach, such as static or statistical analysis, can effectively model the performance behavior of multicore computing platforms. To reduce costs, this approach should also support analyzing software performance to meet A(M)C 20-193's `MCP_Software_1` and `MCP_Software_2` objectives.

A sensible approach to verifying the impact of mitigation to meet the `MCP_Resource_Usage_3` objective is to analyze the impact of each interference channel through a measurement-based approach, see above, and determine whether mitigations applied to it are effective.

Many mitigations can be verified at the platform level. That is, for such mitigations, access to application software is not required to verify the mitigation. All hardware mitigations and some software and hybrid mitigations fall into this category. These mitigations can be verified early in a DO-178C/ED-12C project, reducing the need for further verification later in a project. An approach for this verification is presented in *Platform Characterization* on the next page.

Some mitigations, however, can only be verified when access to the application software is available. Software mitigations based on software architecture fall into this category. An approach for this verification is presented in *Software Characterization* on page 23.

The software characterization process will need to include the RTOS and hypervisor layers as well, as compliance to A(MC) 20-193 needs to be demonstrated for all software. Using an off-the-shelf OS platform where the vendor already provides verification results can make software characterization easier, as described in *Operating system platform verification* on page 23.

*If any of the mitigations you have applied disable a resource or feature, it will not be possible to characterize the corresponding interference channels.*

*In this case, you should instead register any configuration settings associated with the deactivation as critical configuration settings and ensure that the mitigations are implemented correctly and that they remain so during execution.*

*This provides evidence needed to meet the* `MCP_Resource_Usage_1` *and* 2 *objectives.*

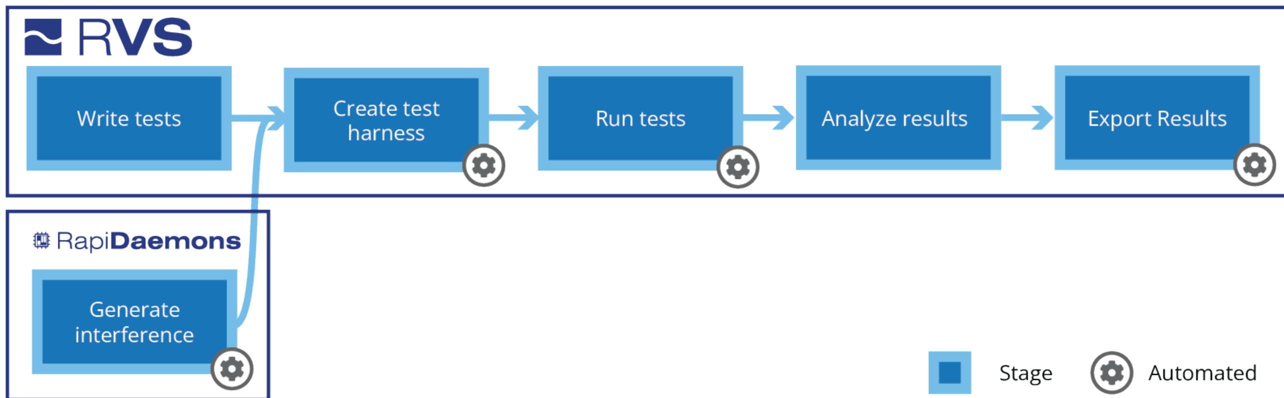



**Figure 16** – ***MACH***[178] *includes an efficient tool automation workflow to ensure test efficiency, traceability and reproducibility*

## 5.1 How to verify mitigations at the platform level (Platform Characterization)

Rapita Systems' **MACH**[178] solution provides an approach for generating interference on multicore platforms through a reproducible mechanism, and measuring the performance of a multicore system, both with and without interference present (**Figure 16**).

- Rapi**Daemons** are benchmark applications designed to target specific interference channels on multicore platforms. They support the measurement and generation of interference through a reproducible approach so that performance metrics to support A(M)C 20-193 compliance can be collected.
- The Rapita **Verification Suite** (R**VS**) enables the creation and execution of performance tests on a multicore platform to provide evidence for A(M)C 20-193 compliance. R**VS** supports running baseline performance tests on multicore platforms, and together with Rapi**Daemons**, enables the generation and execution of tests and collection of performance results when interference has been applied on the system. It also provides an automated export mechanism that allows compliance results to be generated and re-generated efficiently, and it supports the collection of performance counters from hardware event monitoring units (sometimes called performance monitoring units) on a multicore platform. These are used as an independent means to support the analysis and validate results, providing evidence that Rapi**Daemons** have the expected behavior on the multicore system and that resource usage is understood.

> *Detailed procedures for the Platform Characterization process are available in **MACH**[178] Foundations, which also includes template A(M)C 20-193 planning documents, templates and checklists, and white papers to support A(M)C 20-193 compliance.*

These technologies allow performance and mitigations to be verified through the following process (**Figure 17**). For each interference channel:

1. First, Rapi**Daemons** that can support the characterization are selected. This typically includes a victim Rapi**Daemon** that is sensitive to interference on a specific interference channel, as well as one or more Rapi**Daemons** that can generate interference on a specific interference channel (aggressive Rapi**Daemons**).
2. Then, with mitigations applied, the timing behavior of the victim Rapi**Daemon** running on one core of the system is measured while all other cores are idle. R**VS** is used to support the generation and execution of tests and collection and the review of results from these tests. Rapi**Test** supports the creation and automated execution of tests, while Rapi**Time** supports the collection of performance metrics and analysis and export of results.
3. Next, with mitigations still applied, a series of tests is conducted where a sensitive Rapi**Daemon** is running on one core as in step 2, and where interference is applied on one or more of the other cores using aggressive Rapi**Daemons**. R**VS** is again used to support this activity.
4. Results are analyzed using Rapi**Time** to determine whether the interference channel has been mitigated (there is no observable impact on performance of the sensitive Rapi**Daemon** between steps 2 and 3) or not. Results are also exported using Rapi**Time**.
5. If you want to understand the impact of applying the mitigation, steps 2 and 3 can be repeated without mitigations applied. In some cases, a channel may have negligible impact on performance even without mitigation. If you are analyzing the performance impact of an interference channel that has not been mitigated, this is the standard procedure.

Platform Characterization allows you to determine whether your mitigation strategies are effective at mitigating interference channels. Later in your A(M)C 20-193 workflow, you will need to provide evidence that your software meets its requirements while interference is present (e.g. to meet `MCP_Software_1` and `MCP_Software_2` objectives.

An advantage of demonstrating that an interference channel has been mitigated in the way described here is that it allows you to remove that interference channel from the scope of these future activities. No further verification should be required specific to the interference channel you have mitigated unless factors that could affect the mitigation, for example critical configuration settings in the multicore platform, change.

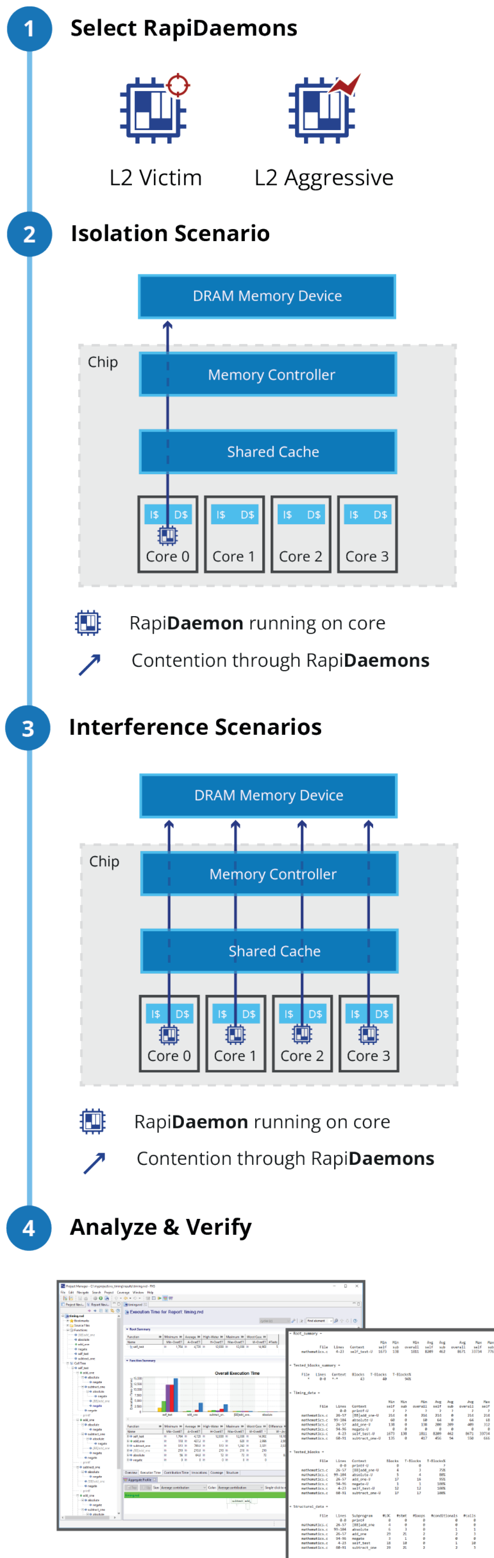



**Figure 17** – ***MACH***[178] *platform characterization workflow*

## 5.2 How to verify mitigations at the software level (Software Characterization)

Rapita Systems' **MACH**[178] approach includes a process for verifying mitigations at the software level, called Software Characterization. This process uses the same elements and a similar overall process to the Platform Characterization process with the following differences:

- Performance is measured with the software (rather than a victim Rapi**Daemon**) running on one core, with aggressive Rapi**Daemons** running on the other cores as in Platform Characterization.
- The mitigation will need to be reverified if the software changes.
- The resource usage of the software must be understood to ensure that sufficient resources are available, therefore the capture of values from different/additional hardware event monitoring units may be required for this activity.

For interference channels that have not been mitigated, the Software Characterization process can also be used to support `MCP_Software_1` and `MCP_Software_2` verification activities.

> *Detailed procedures for the Software Characterization process are available in **MACH**[178] Foundations, which also includes template A(M)C 20-193 planning documents, templates and checklists, and white papers to support A(M)C 20-193 compliance.*

## 5.3 Operating system platform verification

The challenge of A(M)C 20-193 compliance includes formal verification and full compliance to the objectives of DO-178C/ED-12C and A(M)C 20-193 for all software layers, including RTOS and hypervisor. The Wind River DO-178C certification evidence data packages for VxWorks and Helix Platform come with compliance statements for the multicore certification objectives that are gathered on reference hardware, along with guidance on how to achieve full compliance when the integration with the final hardware and application software is completed.

On top of this, the unique IBLL (independent build, link and load) feature in Helix Platform enables the DO-297/ED-124 role-based model for incremental system integration and certification. This approach from Integrated Modular Avionics (IMA) makes it possible to design and verify a generic combination of hardware and OS platform with interference mitigations in place, but without application software. Applications from different origins are then integrated later and can even be replaced independently from each other during the lifetime of the device without having to repeat the full suite of verification activities. This effectively decouples the lifecycle of applications and the underlying platform and powers true reuse in safety-critical multicore systems.

# 6. CONCLUSION

*The use of multicore processing for DO-178C/ED-12C avionics is inevitable, but using multicore processing in DO-178C/ED-12C software requires adherence to A(M)C 20-193 guidelines. Meeting these guidelines requires that interference and interference channels on the multicore platform are understood, mitigated and verified, so efficient approaches to each of these activities is key to the success of any project aiming for A(M)C 20-193 certification.*

Identifying interference channels and verifying mitigation are both crucial steps in achieving A(M)C 20-193 compliance. Rapita Systems' **MACH***178* solution supports these activities by providing documentation, tools and services to identify interference channels on multicore platforms, generate interference in a reproducible and automated manner, and verify the performance of multicore software in interference scenarios. The solution includes template compliance documents, processes and checklists that directly cover most of A(M)C 20-193's objectives.

VxWorks and Helix Platform are the foundation for easy implementation of many interference mitigation strategies and they take full advantage of the features of modern Arm, Intel and PowerPC processors. Off-the-shelf DO-178C certification data packages for these solutions support the verification of multicore interference in line with A(M)C 20-193 objectives. Clear separation and isolation with multi-tiered OS platforms help handle the complexity of multicore interference within the architecture and can enable independent verification mitigations before application software integration.

A(M)C 20-193 certification is an iterative process, and it is unreasonable to expect only a single run-through of some verification activities. Your choice to apply a specific mitigation or not, for example, may lead to an impact on your software performance that requires you to change this choice later and repeat some verification activities. Because of this, it is important to drive efficiency at every stage of the project, as supported by Wind River and Rapita Systems solutions.

In cases where it isn't feasible to mitigate all interference channels while delivering required performance, certification applicants should consider managing the impact of interference to improve software reliability and performance. We intend to address this in a future paper.

RAPITA Systems
A DANLAW Company

*Rapita Systems provides on-target software verification tools and services to the embedded aerospace and automotive electronics industries. Its solutions help to increase software quality, deliver evidence to meet safety and certification objectives (including DO-178C) and reduce project costs. With offices in the UK, Spain and US, it serves its solutions globally.*

## Daniel Wright

**Technical Marketing Executive**

Daniel Wright's roles at Rapita Systems include creating and curating technical marketing content, including collateral, blogs and videos, and capturing and maintaining data that is used to further develop Rapita's verification solutions to best meet the needs of the global aerospace market. Daniel received a PhD in Structural Biology from the University of York in 2016.

## Karl Thyssen

**Multicore Analysis Engineer**

Karl Thyssen graduated from the University of Heidelberg in Computer Science. After joining Rapita, he became a Multicore Analysis Engineer who works in the delivery of customer projects on multicore platforms, specializing in software analysis.

## About Rapita Systems

**Email**: info@rapitasystems.com

**LinkedIn**: linkedin.com/company/rapita-systems

**Find out more**: Learn more about multicore software verification and how to meet DO-178C objectives with a wide range of our webinars and whitepapers. Visit our website for more info. https://www.rapitasystems.com/resources

*Wind River is accelerating digital transformation across industries by delivering the software and expertise that enable the development, deployment, operations, and servicing of mission-critical intelligent systems from the edge to the cloud. Wind River technology is found in billions of products and is backed by world-class services and support and a broad partner ecosystem.*

## Olivier Charrier

**Functional Safety Specialist**

Olivier Charrier is a software engineer who obtained a Master's degree in Software Engineering (DESS) from Bordeaux University in 1989. He worked for Alsys/Aonix before joining Wind River in June 2001 as Senior Field Application Engineer for the South-western region of Europe dedicated to the Aerospace & Defence Market. In 2007, he became EMEA Aerospace & Defence Principal Engineer to support and coordinate EMEA wide A&D programs. Since January 2017, Olivier has been extending his scope to contribute to other markets, like Railway, Nuclear, Medical and Automotive, also adding new geo like APAC.

## Stefan Harwarth

**Specialist Systems Architect**

Stefan Harwarth is part of the European Field Application Engineering team working with customers in the Aerospace & Defense market. His focus is on leveraging Wind River's VxWorks RTOS and Helix Virtualization Platform in customer projects, including certification to safety standards such as DO-178C. Stefan graduated in Computer Science from the University of the Federal Armed Forces in Munich and spent several years working in Avionics software development before joining Wind River.

## About Wind River

**Email**: inquiries@windriver.com

**LinkedIn**: linkedin.com/company/wind-river

**Find out more**: Browse through white papers, videos, infographics, and more to explore the latest trends and technology for mission-critical intelligent systems. https://www.windriver.com/resources

RAPITA Systems
A DANLAW Company

WNDRVR

**Mitigation of interference in multicore processors for A(M)C 20-193**

*As the embedded avionics industry increasingly adopts multicore processors to meet demands for enhanced software functionality, improved SWaP characteristics, and high-performance needs, understanding the certification landscape, particularly A(M)C 20-193, becomes crucial.*

*This paper delves into identifying and mitigating multicore interference to ensure timing deadlines are met, showcasing how Rapita Systems and Wind River solutions support achieving A(M)C 20-193 objectives.*

**© Wind River Systems, Inc.** The Wind River logo™ is a trademark of Wind River Systems, Inc., and Wind River® and VxWorks® are registered trademarks of Wind River Systems, Inc.

**© Rapita Systems.** Rapita Systems logo™, Rapita Systems™, RVS™, MACH178™ and RapiDaemons™ are trademarks or Rapita Systems Ltd.