Journal of Systems Architecture 154 (2024) 103234

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Providing spatial isolation for Mixed-Criticality Systems

E. Tinto *, T. Vardanega

Department of Mathematics, University of Padua, Italy

ARTICLE INFO

Keywords:
Real-time systems
Mixed-Criticality Systems
Time and space partitioning
Memory isolation

ABSTRACT

Hard real-time systems, characterized by stringent timeliness requirements, occur in an increasing variety of industrial sectors. Some such domains carry important safety-critical concerns, notably avionics, space, and automotive. One common design trend across those domains seeks to reduce the number of computing devices embedded in them by integrating software applications of different criticality levels into one and the same onboard computer. A safety-savvy design approach however requires isolation among components of different criticality, to prevent unintended reciprocal interference across them. Isolation is traditionally achieved through partitioning. Partitioning, however, incurs low resource utilization as cautionary margins are used to inflate partition budgets over their anticipated needs. This situation has prompted research into alternative ways to integration that can safely afford higher levels of utilization. The Mixed-Criticality (MC) approach, which concentrates on the CPU scheduling problem, has yielded a large body of research results that show considerable gains in sustained utilization, but it has yet to meet all of the isolation requirements of safety-critical systems. This work presents a solution to augment a state-of-the-art MC solution with efficient and effective spatial isolation capabilities. Experimental results show that our solution provides adequate guarantees of temporal *and* spatial isolation with very small runtime overhead.

1. Introduction

1.1. Context and motivation

Real-time systems are characterized by the presence of timeliness requirements, whose strictness depends on the assurance level that the system is expected to achieve. The aerospace domain (avionics and space) is the traditional fore-bearer of safety-critical real-time systems. In that domain, the advent of the "fly-by-wire" revolution, pioneered by the famous Apollo program in the 1960s, resulted in the adoption of *federated* system architectures, where individual onboard computers were entirely dedicated to selected and distinct groups of functionalities (technically, partitions), in the pursuit of maximal isolation. Isolation extends across the dimensions of space, time, and fault. Federation soon proved too costly in terms of material, harness, and utilization. This observation gave rise to a shift toward the Integrated Modular Avionics (IMA) concept, as shown by [1], which allows more functional components to be deployed onto the same hardware resources, still granting isolation at the software level but reducing the number of onboard computers required by the system overall. An IMA solution essentially renounces direct hardware isolation guarantees, and therefore strives to nullify unintended interference among software components. IMA systems achieved such goal by means of static (spatial and temporal) resource allocation, using logical (as opposed to physical) partitions to isolate software components according to their level of criticality, regardless of their actual function. The resulting IMA partitions are criticality-driven aggregates, which renders them not functionally-cohesive. The dominant solution to partitioning over the spatial and temporal dimension nowadays is often referred to as Time and Space Partitioning (TSP), as noticed by [2,3]. The TSP approach is very convenient for extended contractual supply chains, typical of the aerospace domain, where the system parts developed by subcontractors need to be assembled without hazards and hassles by the prime contractor. The flip side of that bonus, though, is that the TSP approach suffers from low utilization – which contrasts with the urge to increase the functional value of the system that comes from committing more functions to software – due to the common practice of adding conservative (hence arbitrarily large) cautionary margins to partitions' resource budget.

The Mixed-Criticality Systems (MCS) approach arose little over a decade ago in the quest to attain higher levels of CPU utilization while protecting higher-criticality components from undue interference from lower-criticality ones. Following the intuition of [4], MCS allows tasks situated at different criticality levels to execute without logical or physical partitioning, hence effectively to co-exist, while granting that, in the event of a transient overload situation, high-criticality tasks will

* Corresponding author.
E-mail addresses: edoardo.tinto@phd.unipd.it (E. Tinto), tullio.vardanega@unipd.it (T. Vardanega).

https://doi.org/10.1016/j.sysarc.2024.103234
Received 28 July 2023; Received in revised form 31 May 2024; Accepted 3 July 2024
Available online 8 July 2024

be given precedence over low-criticality tasks, assuring timely completion for the former. The foundational work presented in [4] combines priority assignment heuristics and feedback-based response time analysis (RTA) to warrant larger CPU time margins – and hence higher guarantees of timely completion – to tasks with higher criticality. It is worth noting in passing that the notion of MCS as used in real-time scheduling research, which seeks integration without segregation, differs from the interpretation of the term mixed-criticality outside of that particular domain, where a partitioned system can be described as mixed-criticality, as in the works of [5,6] for example.

Vestal's work prompted a flurry of MC-centered research, all striving to improve the generality and the performance of the mixed-criticality scheduling solution. According to [7], higher schedulable utilization overall could be obtained by:

1. having all tasks initiate their life cycle with low-criticality execution time budget provisions;
2. deploying a run-time monitor to measure how long individual jobs actually execute;
3. in the event that a high-criticality task currently running should exceed its low-criticality budget without signaling completion, having the run-time monitor trigger a CPU mode change where only high-criticality tasks would be allowed to execute, and all low-criticality tasks would be held until normality is restored.

Assuming that overload situations are transient, normality would be restored as soon as the ready queue was void of high-criticality tasks. This MCS scheduling policy was called Adaptive Mixed Criticality (AMC) by its authors.

The AMC scheduler was initially designed for single-core scenarios. In a multicore processor architecture, however, discarding low-criticality tasks on mode change is not the only available option. As shown by [8], in fact, selected low-criticality tasks might migrate to another core with feasible CPU capacity for them, hence still in low-criticality mode. This approach, named "semi-partitioned" by its authors on account of the admissibility of selective task migration across cores, was shown to do better than the AMC scheduler for schedulable utilization in a dual-core scenario. The cited authors subsequently extended their work to quad-core architectures in [9].

More recently, a concrete implementation of a semi-partitioned AMC scheduler has been implemented by [10] based on an Ada runtime library targeted to an embedded dual-core processor, and made available at [11]. In addition to showing that the results claimed by [8] were largely – though not fully – reproducible with real-world technology, that work also provided empirical quantification of the gain in sustainable system utilization that could be attained by the Ada-based semi-partitioned MCS solution over a traditional TSP system, built on XtratuM [12] as hypervisor for the same processor target and identical application load. The cited MCS implementation was developed as an extension of the standard runtime of the Ada Ravenscar profile [13], with the Zynq-7000 SoC embedding an ARM Cortex-M family processor as target.

The works presented in the state-of-the-art literature in this field do not address spatial isolation, in spite of it being an essential requirement for safety-critical applications. Without addressing spatial isolation needs satisfactorily, MCS will not be able to supplant TSP systems in safety-critical industrial domains hungry for higher CPU utilization.

Dispensing with the use of hypervisor-enforced partitions carries the need for lighter-weight runtime mechanisms capable of preventing the occurrence of illicit memory accesses during execution, while also enabling controlled means of cross-criticality communication. In this work, we investigate the spatial dimension of isolation, building upon the proceeds of [10] and extending the Ada runtime library developed there to have it also support spatial isolation.

1.2. Related works

Within the research field rooted in the MCS approach, viz. a partitionless task-based architectural approach, some works have addressed spatial isolation. The matter can be addressed from two different angles, a hardware one relying on hardware-assisted mechanisms, and a software one, for which the underlying hardware layer is entirely transparent.

Regarding the former, namely research on hardware-assisted spatial isolation, different research trends can be recognized. In their works, [14,15] both address spatial isolation with the support of a memory management unit (MMU), which however is not general enough and therefore not viable for some of the application domains of our interest here (cf. e.g, [16]).

In [17,18], instead, the respective authors consider the use of customized processors, developed *ad hoc* for MCS, to tackle both temporal and spatial isolation.

The use of transactional memories, which grant atomicity in memory accesses in the presence of concurrency, is considered in [19,20], aiming at increasing isolation in shared memory, but without further discussing the issue of spatial isolation for non-shared ones.

An additional research trend investigates the possibilities emerging from cache-partitioning. The works of [21,22], and [23] consider the application of cache partitioning for assigning private space to groups of tasks, and its impact in a context with shared memory. The mechanism to perform cache partitioning in these works is hardware-based.

Recent works also consider the use of ARM TrustZone [24], which enforces isolation between distinct execution environments over the same hardware processor. Examples of this are provided in [25,26].

On the converse, software-based mechanisms for spatial isolation in MCS have never been considered, to the best of our knowledge. One the reasons for this remarkably uneven distribution in the works addressing spatial isolation in MCS might be the complexity of performing possibly substantial alterations in the compilation toolchain, needed to equip the base runtime libraries with the required capabilities, while still preserving portability. The quest for higher performance and the availability of sophisticated hardware facilities in commercial processors may also have boosted the preference for the exploration of hardware-assisted solutions.

We decided to investigate the software-based approach, aiming to produce a fitting solution for systems that do not possess and cannot afford hardware-assisted spatial isolation.

1.3. Contribution

Our contribution in this work is twofold:

- We present the design of a solution rooted in Ravenscar's model of concurrency that also addresses spatial isolation in a way amenable to static analysis, while preserving the temporal isolation guarantees achieved by [10]. Our goal is to support a viable alternative to logical partitioning, granting temporal *and* spatial isolation among (Ada) tasks, the central element in the Ravenscar model of execution, adopting a semi-partitioned MCS scheduling policy.
- We contribute to the public domain the software artifacts resulting from our extensions to the work of [10]. In addition to the spatial isolation and cross-criticality communication mechanisms, the scheduler has been adapted to support shared resources, in accordance with the Ravenscar profile restrictions, granting deferred suspension for jobs executing in a critical section during a mode change.

The remainder of this paper is organized as follows: Section 2 describes the baseline from which this work starts; Section 4 presents the proposed solution; Section 6 presents experimental results to evaluate the operation and the performance of our solution; Section 7 draws some conclusions and outlooks directions for future work.

2. Baseline for the investigation

2.1. Temporal isolation for MCS in Ada

The work presented in [8] describes a semi-partitioned dual-core instance of an AMC scheduler, using response time analysis of synthetic workloads to show the potential gain in schedulable utilization. The main characteristics of the model proposed by the authors of that work are: (1) two criticality levels, high (HI) and low (LO); (2) the stipulation that only LO tasks may migrate, exclusively after a per-core mode change and only toward a core that can host them feasibly; (3) the absence of (logical) resource sharing across tasks. Under these assumptions, they prove significant improvement over the schedulable utilization attainable by non-migratory MC algorithms.

The authors of [10] undertook to ascertain the reproducibility of the experiments performed in [8] and to make a quantitative comparison between an Ada Ravenscar based implementation of that particular MCS solution and a functionally-equivalent TSP system. In order to support the chosen MCS scheduling policy, the authors had to extend the standard Ada Ravenscar runtime by adding the following provisions:

- To each task, two execution-time budgets are assigned, a HI and a LO one. Those two parameters model the task's estimated worst-case execution time when running on a CPU in HI or LO mode.
- A run-time monitor to measure job execution time that operates as specified in the original work. In the event that a job exhausts its assigned LO execution-time budget without signaling completion, an interrupt is raised by the run-time monitor, which triggers a LO-to-HI mode change on the core where the event occurred.
- For each core, an additional scheduling queue has been introduced to enqueue the LO tasks that are not allowed to migrate after their CPU has entered HI mode. Those tasks are referred to as *frozen* tasks and are moved to that additional queue – hence invisible to scheduling – until normality is restored.
- An artificial null task is used to model the occurrence of an *idle tick*, which causes the return to normal mode, when, on a CPU in HI mode, no HI task is ready. When an *idle tick* is detected, the CPU returns to LO mode and all *frozen* tasks are returned to their original scheduling queue.

No changes occurred to the compilation system, as no syntactic provisions were required to implement the runtime features listed above. Indeed, the resulting system deflects from strict conformance with the specification of the Ravenscar profile, as the latter does not contemplate task migration, which instead is crucial to the MC scheduling policy of interest. This non-conformance is minor, though, as under the semi-partitioned scheduling regime migration occurs only between cores with feasible workloads, before and after migration.

To perform their empirical evaluation on a concrete implementation, the cited authors developed utilities to generate synthetic tasksets, which employ: (1) the Dirichlet Rescale (DRS) algorithm [27] to generate pseudo-random per-task utilization values within given utilization bounds at system-level; (2) the hyperperiod control technique by [28], tunable to limit the degree of harmonicity across task periods; and a configurable version of the Whetstone synthetic benchmark [29] for generating task workloads given period and target utilization.

The TSP system to compare against for schedulable utilization, was developed to be functionally equivalent to the MCS system, with a few notable architectural differences:

- Tasks execute *inside* logical partitions, with intra-partition scheduling delegated to an operating system (RTEMS) instance guest of the XtratuM hypervisor host. For each core, two partitions are defined, one for HI tasks and one for LO tasks.

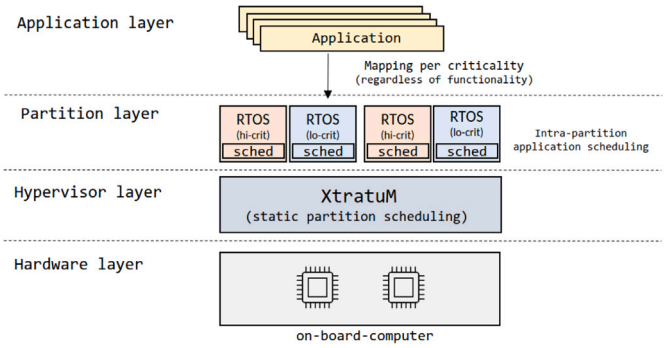


Fig. 1. TSP system architecture — An example of a TSP system using XtratuM as the hypervisor. Here, scheduling decisions are taken for partitions by the hypervisor, and for applications by the partition's internal scheduler.

- Inter-partition schedule is determined by an offline assignment created using a weighted round-robin algorithm. The XtratuM hypervisor manages partitions' scheduling but does not address in any way the intra-partition task scheduling, which is handled by the partition itself, as shown in Fig. 1.
- Partitions are static in composition and in assignment to cores; tasks do not migrate across partitions, nor do they move across cores.

The MCS and TSP solution were then comparatively evaluated in four experiments, under different types of tasksets. From such experiments, the authors of [10] show that MCS systems are able to sustain higher schedulable utilization, both peak and sustained, than TSP systems in a totally trustworthy manner.

2.2. Constraints on the solution

The MCS approach supported by the reference implementation in [10] is incomplete (in fact, void besides the standard visibility control warranted by the Ada model execution) as far as the spatial dimension of isolation goes. To overcome this limitation, we propose an extension of the runtime library and execution model that also addresses spatial isolation. Building on the previous reference implementation, our proposal inherits a number of architectural features or constraints:

- The runtime library targets the Ada programming language. Ada's use in the aerospace domains is well established, and the language itself provides a set of features in accordance with a model of computation that perfectly matches the scheduling provisions required by [8].
- Static and dynamic conformance with the Ravenscar profile is warranted in the reference implementation and in our extension to it, except that selected LO tasks are allowed to migrate across cores, when their migration can be accommodated by keeping all tasks at destination feasible. Other than that, the Ravenscar profile restrictions are very fit for safety-critical applications.
- The spatial isolation features to be added in this work should coexist with the temporal isolation features, coherently and consistently, programmatically and semantically, without the risk of undermining feature interaction or unwanted emerging features.
- The target hardware architecture used for the experimentation should be the same as used in [10], namely a Zynq7000 SoC with an ARM dual-core processor. This is not strictly required for the scheduling features, but moving to a different hardware would have required adapting the build infrastructure for the new target, without clear benefits for the outcomes of this work.

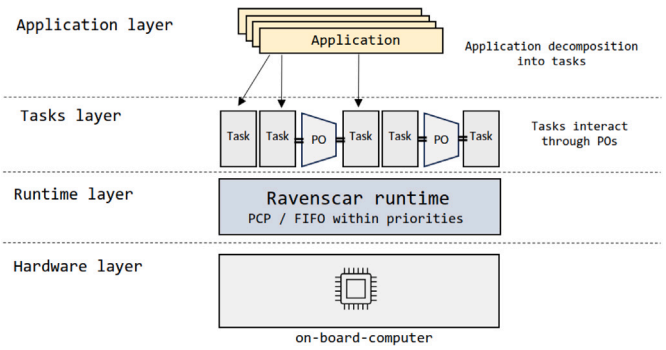


Fig. 2. Cross section of a Ravenscar application architecture, where tasks do not have any criticality level, and data-oriented communication among tasks happens through Protected Objects (POs).

3. Foundations to the solution

3.1. The Ravenscar profile

Provable predictability is not just an optional benefit in safety-critical systems; instead, it is a mandatory requirement. The full Ada programming language provides a rich set of concurrency features, capable of expressing complex scenarios and articulations of task-based architectures, but potentially precluding full formal verification of them. It is on account of this challenge that the Ada specification includes statically and dynamically enforced profiles. The Ravenscar profile has been designed with the intent of supporting a verifiable tasking model, restricting some of the concurrency features provided by the full Ada language, along with some further restrictions to the sequential parts of it. Ada's restrictions mechanism reflects the feature-composability trait of the language, which was designed to enable users to choose the range of primitive language features needed by the program, and exclude the others.

The Ravenscar concurrency model sees tasks as the main architectural elements of an application. Ravenscar tasks are statically defined (i.e., declared at the library level, hence created during elaboration, prior to execution), non-nested, and non-terminating. Tasks are said to be *ready* when they can execute instructions granted a processor, *suspended* when an event is required to enable the task to execute, or *blocked* when waiting for a shared logical resource (which may encapsulate a physical resource except for the CPU). The Ravenscar profile prescribes fixed-priority scheduling (FIFO within priority), and priority ceiling emulation for controlling access to shared resources.

Interactions between tasks take place through shared resources known in Ada as *protected objects* (POs), which grant exclusion and avoidance synchronization on concurrent access. The Ravenscar profile specification prohibits any operations that might cause the task to suspend while holding the lock on an object. Much like tasks, POs are statically defined, and their run-time *finalization* is not allowed. Simpler objects for task synchronization also exist in the Ravenscar profile: the *suspension objects* (SOs); they implement the abstraction of a simple semaphore, and also are statically defined as tasks and POs. POs support sophisticated forms of controlled resource sharing among tasks, with emphasis on synchronized access to protected operations. Using POs for writing and reading large data payloads incurs costs linear in the data footprint. This trait does not really accord with the Ravenscar quest for efficiency of synchronization. Accordingly, we deemed POs unfit for use in the implementation of the memory-transfer mechanisms envisioned for this work. We chose instead to develop our solution from the ground up, with Ravenscar-compliant constructs and features, in preference to using other components from the full language, such as for example the containers library.

Library level resources such as tasks, POs, and SOs are declared inside the memory space of an *environment task*; this provision requires all such components to have statically known size, including the stack of individual tasks (which obviously makes recursion a threat), and equally prohibits all forms of dynamic allocation.

The model implemented by the Ravenscar profile prevents the occurrence of deadlocks, as discussed in [13], and it is simple enough to be amenable to formal verification, while still offering tasking capabilities. Fig. 2 displays a vertical section of a Ravenscar-compliant application architecture.

3.2. Provisions for spatial isolation

The Ravenscar model is the starting point of our investigation, due to its suitability for safety-critical applications. [10] extend a Ravenscar runtime library for Zynq 7000 hardware with a semi-partitioned instance of an AMC scheduler, addressing temporal isolation and controlled migration of LO tasks during transient overload situations. However, the need for spatial isolation still needs to be fulfilled.

In order to address this requirement, we augmented this runtime environment with three additional features, which also represent distinct research contributions:

- An ownership mechanism similar to that of the Rust programming language [30], which allows using dynamic memory safely for cross-criticality communications.
- Disciplined use of Ada packages (the source level components of a program, also known as compilation units) and idiomatic programming to restrict visibility in a way that yields an equivalent of TSP partitioning, so that space isolation can be asserted statically, at compile time.
- An improved runtime scheduler that causes deferred suspension for LO tasks that should be frozen on a mode change, when they still have to commit an exclusive write or read to a cross-criticality message. (We shall discuss the wisdom of this feature in the sequel.)

4. Essentials of the proposed solution

This section illustrates the design of the features introduced in Section 3.2, which stand at the core of the proposed solution. The illustration is split in three complementary parts. First, we discuss how cross-criticality communications happen, which involves the ownership transfer mechanism, illustrated in Section 4.1.1, using the channel entity discussed in Section 4.1.2. Subsequently, in Section 4.2, we discuss the idiomatic programming style that leverages our provisions, yielding a solution arguably equivalent to classic TSP with much less runtime machinery. Finally, in Section 4.3, we present the scheduler extension that caters for the deferred freezing needed for cross-criticality communications to survive unscathed the occurrence of mode-change events at run time.

4.1. Cross-criticality communications

4.1.1. The ownership transfer mechanism

The Rust programming language supports a reference control mechanism known as *ownership*, which regulates in a statically verifiable manner how memory-stored items are managed. Ownership is preserved in the presence of concurrency [31]: only one task at a time can own the object denoted by a memory reference. The importance of this provision to our context is that its correctness can be asserted at compile time. [32] notes that this ability prevents the occurrence of a score of run-time errors that would otherwise occur from common-practice use of dynamic memory.

```
1  package Channel_Pool_Access is
2      --  This package uses generics.
3      . . .
4      generic
5          --  Element_Type is a generic element
6          --  that can be allocated as a message
7          --  in a channel pool.
8          type Element_Type is tagged private;
9
10     package Shared_Pointer is
11
12         type Accessor
13           (Element : access Element_Type)
14         is limited private
15         with Implicit_Dereference => Element;
16
17         type Element_Type_Reference
18           is limited private;
19
20         type Reference_Type is new
21           Ada.Finalization.Limited_Controlled
22         with record
23             Element : Element_Type_Reference;
24         end record;
25
26         ---------------------------------
27         --  Operations on Reference_Type
28         ---------------------------------
29         . . .
30
31     private
32
33         type Accessor
34            (Element : access Element_Type)
35            is null record;
36
37         type Element_Type_Reference
38            is access Element_Type;
39         for Element_Type_Reference'Storage_Pool
40         use
41           Channel_Pool_Instances
42           .High_Low_Channel_Pool;
43
44         procedure Free_Element is new
45         Ada.Unchecked_Deallocation
46           (Element_Type, Element_Type_Reference);
47
48     end Shared_Pointer;
49 end Channel_Pool_Access;
```

Listing 1: Selected declarations within the `Shared_Pointer` generic package, inside the `Channel_Pool_Access` package.

In this work, ownership is applied for cross-criticality communication precisely for the reason of having a mechanism that transfers data without permitting any sharing of it across criticality boundaries. The implementation of this mechanism would use dynamic memory precisely to allow it to be transferred from one owner to another, along directions agreed at design time. In this work we concentrate on the language runtime part of the implementation of this mechanism, without altering the compilation process, and argue that static compile-time checking that message ownership is preserved is a comparatively easy effort.

Ownership is modeled by a reference containing a private *access type*, a first-class type in Ada, which prevents using objects of this type outside of their scope of declaration. To ensure that references are not copied or managed improperly, their type (`Reference_Type` in Listing 1) is limited. In Ada, limited types cannot be assigned, hence they cannot be copied. Additionally, `Reference_Type` internal reference is a private type (Listing 1, line 23), thus it cannot be accessed directly. This design choice poses the problem of how the user can access the referenced object, given the visibility restrictions on the access type of `Element_Type`.

Our solution to this need uses an *accessor*, a construct enriched by the `Implicit_Dereference` attribute introduced with the 2012 revision of the language, which allows one to automatically dereference the pointed object, without disclosing its access type. Listing 1 illustrates the declaration of our accessor at lines 12–15. This approach holds as long as *anonymous access types* are not used in the program. The rationale of this choice is highlighted in Listing 4, as we shall discuss next. No explicit restriction is provided in Ada to exclude anonymous access types specifically. However, they are implicitly excluded in the Ravenscar profile owing to the exclusion of dynamic memory. In our solution we depart slightly from this particular trait, and therefore need to find other ways to ensure that the program does not employ anonymous access types. This check can easily be made statically before compilation.

A `Get` procedure is exposed to hide the accessor reference handling away from the user, as illustrated in Listing 2: the procedure returns an object of *accessor* type, declared as limited private (Listing 1, lines 12–15); its derefentiation happens automatically thanks to the implicit dereference attribute.

```
function Get (Reference : Reference_Type)
         return Accessor
is
begin
    return Accessor'
      (Element => Reference.Element.all'Access);
end Get;
```

Listing 2: `Get` procedure, returning an *accessor* with attribute `Implicit_Dereference` set.

```
procedure Move (Left : in out Reference_Type;
                Right : in out Reference_Type)
is
begin
    --  Here Left might be null or contain
    --  a valid reference
    Free (Left);
    --  Now Left.Element is null, any previously
    --  referenced object deallocated
    Left.Element  := Right.Element;
    --  The reference element in Right is
    --  copied into Left.
    --  Note that Element is an Access Type
    Right.Element := null;
    --  Here Right is set to null,
    --  deallocation is not required
end Move;
```

Listing 3: The *Move* procedure. Ownership (in the form of exclusive access rights) is passed from `Right` to `Left`; thereafter `Right` is set to `null` to revoke future access to the message.

```
procedure Some_Procedure is
    --  Some_Access_Type is an access type
    --  (not an anonymous one, though)
    type Some_Access_Type is access Message;
    Message_Access_T : Some_Access_Type;

    --  Message_Access_A is an anonymous
    --  access type variable
    Message_Access_A : access Message;

    Reference_1 : Reference_Type;
    Reference_2 : Reference_Type;

    --  other declarations
    . . .
```

```
16 begin
17     . . .
18     --  Proper way to access a message field
19     --  (dereference is performed automatically)
       .
20     --  Here 'Field' is an internal attribute of
21     --  Message
22     Shared_Pointer.Get (Reference_1).Field :=
23        "Value";
24
25     --  Compile-time error: left hand of
26     --  assignment must not be limited type
27     Reference_1 := Reference_2;
28
29     --  Compile-time error: wrong type
30     Message_Access_T :=
31        Shared_Pointer.Get (Reference_1).Element;
32
33     --  Compile-time error: wrong type
34     -- (and Element's type is private)
35     Message_Access_T := Reference_1.Element
36
37     --  The next instruction works, hence
38     --  explicit use of anonymous access type
39     --  should be forbidden
40     Message_Access_A :=
41        Shared_Pointer.Get (Reference_1).Element;
42     . . .
43 end Some_Procedure;
```

Listing 4: Illustration of the isolation guarantees achievable with the use of an *accessor*. Here, `Message` is the type of a message.

The central element of the ownership mechanism we implemented in this work is the `Move` procedure, which transfers the ownership of a dynamically allocated message. The `Move` procedure is the sole user-level action capable of altering the ownership of a message; it allows copying an access type object across two references, ensuring that messages without owner are immediately deallocated, and that all references without a referee are set to `null`. Listing 3 illustrates its implementation. Finally, Listing 4 exemplifies the use of an object of type `Reference_Type`. Line 22 shows how a user-defined procedure can modify a message through the use of `Get`; here, dereferencing is performed automatically. The next three statements result in compile-time errors. The assignment of `Reference_Type` objects (line 27) is not allowed due to the use of a limited type.

Also, copying the internal access type of a `Reference_Type` object (lines 30–31) is impossible; being it a private type, any other named access type (`Some_Access_Type` in the listing) would not match the internal element type. The same happens when trying to bypass the `Get` procedure as in line 35. Conversely, anonymous access types could be used to perform unconstrained copies of a `Reference_Type` object internal reference, as shown at lines 39–40: for that reason, the use of anonymous access type should be restricted.

4.1.2. Cross-criticality communication channels

The Ravenscar profile does not contemplate the notion of criticality, and the extended runtime library provided by [10], which does support criticality, does not consider resource sharing. Therefore, since the standard Ravenscar specification is unaware of mixed-criticality, resource sharing realized through POs takes place without warranting isolation between resources accessed by HI tasks and resources accessed by LO tasks.

In our solution, we propose to constrain the use of standard POs to intra-criticality communication, while providing a dedicated structure for addressing cross-criticality communications, named *channel*. In that manner, a partition that would not undertake inter-partition communication would correspond to a fully-conformant Ravenscar application.

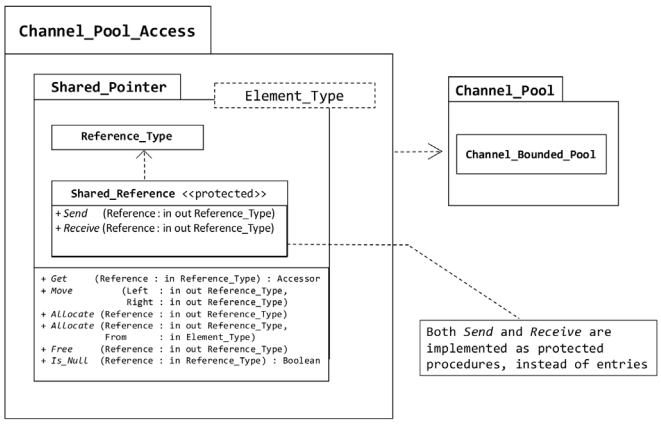


Fig. 3. Enriched package diagram exposing the structure of a channel.

Channels are intended for asynchronous (non-suspensive) message-passing across tasks at different criticality levels. Such communications would happen by tasks exchanging dynamic objects, referred to as *messages*, with exclusive access rights on messages managed in a manner inspired on the Rust ownership mechanism, described by [33].

Like everything else in a Ravenscar application, channels are statically defined objects, instantiated at elaboration time. Conversely, messages are dynamic entities of fixed size.

Channels are unidirectional and support multiple senders and receivers, even though, as for POs, the complexity arising from this multiplicity should be carefully assessed.

With the intent of providing a clear and well-defined semantic for cross-criticality communication, we impose that channels should hold no more than one message at a time. Channels would therefore only retain the message most recently sent, overwriting any previous message that was sent but not received. This is a design decision that simplifies the runtime semantics but may place considerable restrictions on the application's wishes on cross-criticality communication. Should application requirements for queue-based channels arise, our runtime structures could easily be adapted to that, only requiring design decisions on the queuing policy to be deployed.

Channels and POs are designed with different features for different purposes. Channels are targeted to serve cross-criticality communication, which requires the ability to transfer the ownership of memory payloads between sender and receiver in order that there be no sharing across criticality levels. This feature could not be soundly achieved with POs without derailing their intended purpose, nature and semantics, as noted in Section 3.1. The way we have designed them, channels also allow preserving messages for later use in the occurrence of a mode change, where LO tasks holding them are frozen and descheduled until further notice.

4.1.3. Implementing a channel

This work added some specialized packages to the runtime library originally presented in [11]. The `Channel_Pool_Access` package is one of such additions, which provides the implementation of channels. It also defines the generic package `Shared_Pointer`, as shown in Fig. 3. Here, `Element_Type` is a generic type, specified by the application as illustrated in the next section. `Element_Type` refers to the type of the messages a channel is allowed to transmit. A channel is made of three parts (cf. Fig. 3):

- An internal reference to a dynamically allocated message. The space dedicated for dynamic allocation is managed through a *storage pool*, i.e., a language-level memory object exposing *allocate* and a *deallocate* procedures, allowing the programmer to define the behavior of both operations (e.g., implementing automatic

memory reclamation). As noted, this provision is outside of the Ravenscar profile specification, but used in a controlled manner as this work proposes, it warrants an execution behavior that can be assured to the highest levels.

- A PO to grant exclusive access to the procedures accessing the shared reference: `Send` (Listing 5), and `Receive` (Listing 6). While being much lighter-weight than POs, Ada's SOs cannot suit our needs here as they do not warrant atomic actions other than for atomic variables (data represented on a single processor word and operated on by a single processor instruction), which messages are not. Idiomatically, in our solution, the call to the `Move` procedure, which transfers ownership, is embedded in the protected procedures `Send` and `Receive`.
- A pair of procedures, `Allocate` (Listing 7), and `Free` (Listing 8), to manage messages without directly accessing the channel's internal storage pool. All those procedures preserve ownership.

The PO internal to a channel does not contain any entries. In Ada, entries to POs support exclusion synchronization, prefixing exclusive acquisition of the resource with a Boolean condition, called *guard*; when the guard evaluates to false, the call is held in a per-guard wait queue that is notionally considered *within* the resource but without exclusive access to it; the guard is reevaluated every time the lock owner relinquishes the resource, in preference to attending to calls from outside of it. This algorithm, known as the eggshell model [34], prevents access starvation by construction, which is a precious feature in general. Calling entries, however, is exposed to potentially unbounded wait time, as there is no general way to tell when a guard will evaluate to true. For this reason, use of entries in the Ravenscar profile is only allowed when the resulting execution behavior models sporadic activation (which follows exactly that pattern). Other than that, entries are totally excluded in the PO internal to a channel because their semantics might conflict with the treatment of mode changes.

Consider, for example, an instance in which a HI task is waiting on a closed entry, and a LO task responsible for altering the state of the corresponding PO, consequently turning its guard to true, is frozen due to a mode change. Under these circumstances, without appropriate mitigation, the wait time of the HI task will extend as long as the mode change lasts causing massive undue interference on the HI task. Since they use physical memory, channel-handling procedures must be robust to preemption-induced reentrancy. `Allocate` and `Free`, which manage the dynamic memory that allows channels to exist, are implemented to execute at the highest per-CPU priority (code not shown here), which makes them preemption-free within a single Ravenscar partition. `Send` and `Receive`, which perform message exchange over channels, are implemented as protected procedures, which affords them exclusive access until completion. These provisions alone, however, do not suffice against multicore-level parallelism, where two or more Ravenscar partitions might attempt to access the same memory location. To prevent that from happening, our implementation uses an additional locking mechanism that operates across cores.

```
procedure Send (Reference : in out Reference_Type)
    is
begin
    --  Here, Internal_Reference might
    --  contain a reference to a message
    --  or be null
    Move (Internal_Reference, Reference);
    --  Here, Reference is null

    --  Store the arrival time of the
    --  last message
    Message_Arrival_Time := Ada.Real_Time.Clock;
end Send;
```

Listing 5: The *Send* protected procedure. `Internal_Reference` is a pointer to the message currently placed in the channel.

```
procedure Receive (Reference : in out
    Reference_Type) is
begin
    --  Here, Reference might contain
    --  a reference to a message or be null
    Move (Reference, Internal_Reference);
    --  Here, Internal_Reference is null
end Receive;
```

Listing 6: The *Receive* protected procedure.

```
procedure Allocate (Reference : in out
    Reference_Type) is
begin
    --  For elements of type Element_Type
    --  any new allocation is placed on
    --  a dedicated storage pool
    Reference.Element := new Element_Type;
end Allocate;
```

Listing 7: The *Allocate* procedure.

```
procedure Free (Reference : in out Reference_Type)
    is
begin
    Free_Element (Reference.Element);
    Reference.Element := null;
end Free;

procedure Free_Element is new
    Ada.Unchecked_Deallocation
        (Element_Type, Element_Type_Reference);
```

Listing 8: The *Free* procedure. `Free_Element` is a private operation (meaning it cannot be seen from the outside): the application should invoke `Free` instead.

Given a channel C, for communication from LO task τ_1 to HI task τ_2, the life cycle of message $\mathcal{M}$ is characterized by the following procedure invocations:

1. First, τ_1 invokes $C.allocate$, allocating a new message $\mathcal{M}$ on C's internal *storage pool*, and returning a reference to $\mathcal{M}$, encapsulated in a pointer data structure. At the end of $C.allocate$, the ownership of $\mathcal{M}$ belongs exclusively to τ_1: C does not retain any reference to $\mathcal{M}$ after returning.
2. τ_1 may decide to delete $\mathcal{M}$, by calling $C.free(\mathcal{M})$, or to initialize it.
3. Then, τ_1 can send $\mathcal{M}$, invoking $C.send(\mathcal{M})$. Two observations are in order here: (1) the sending task τ_1 does not name the receiver task (hence, the scopes of τ_1 and τ_2 remain isolated from each other), it simply (2) yields the ownership of $\mathcal{M}$ to the channel. At this point τ_1 has no reference to $\mathcal{M}$, as shown in Listing 5: its own reference (which was the only access point to the message) is now set to `null`; doing so prevents any future access to $\mathcal{M}$ by τ_1. Accordingly, the application should be ready to handle the occurrence of a `null` reference, which in Ada is associated to a runtime check by default.
4. If a new invocation of $C.send$ is issued, $\mathcal{M}$ is de-allocated by the channel itself, due to the semantic of `Move` (Listing 3), and a new message will replace it. Otherwise, at a certain point, τ_2 will invoke $C.receive$, and acquire the ownership of the message stored in C.
5. Any successive call to $C.receive$ will return a `null` reference to the caller. The application should be ready to handle such occurrences.

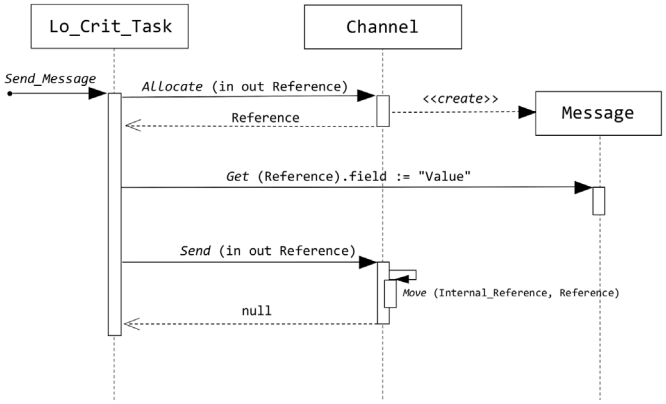
In our proposed solution, a finalization procedure has been defined for handling the de-allocation of a message going out-of-scope, for

example, at the end of a procedure call. Finally, it is worth underlying that message ownership is realized by granting exclusive access to a reference to the message itself.

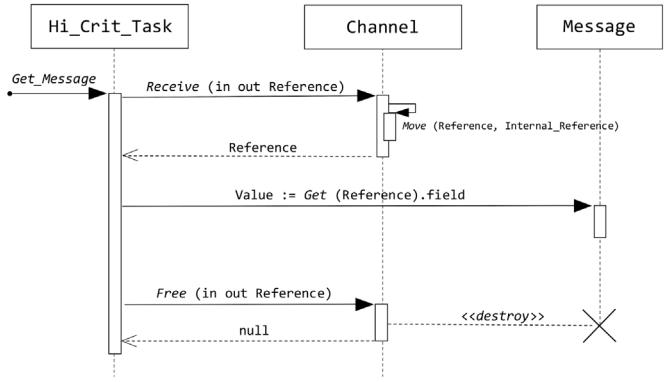
```
1  with Channel_Pool_Access;
2
3  package Channel is
4      --  The type of Element_Type.
5      --  The structure of this type is
6      --  application-specific
7      type Message is tagged record
8         Field : String (1 .. 5);
9      end record;
10
11     --  Instantiation of the generic
12     --  package Shared_Pointer
13     package Shared_Pointer is new
14       Channel_Pool_Access.
15         Shared_Pointer (Message);
16
17     --  Encapsulation of the Send
18     --  and Receive procedures
19     package Low_to_High_Channel is
20       procedure Send
21         (Reference :
22           in out Shared_Pointer.Reference_Type);
23       procedure Receive
24         (Reference :
25           in out Shared_Pointer.Reference_Type);
26    end Low_to_High_Channel;
27
28 end Channel;
29
30 package body Channel is
31
32     package body Low_to_High_Channel
33     is
34       Shared_Message :
35         Shared_Pointer.Shared_Reference;
36
37       procedure Send (Reference :
38         in out Shared_Pointer.Reference_Type)
39       is
40       begin
41          Shared_Message.Send
42            (Reference => Reference);
43       end Send;
44
45       procedure Receive (Reference :
46         in out Shared_Pointer.Reference_Type)
47       is
48       begin
49          Shared_Message.Receive
50            (Reference => Reference);
51       end Receive;
52    end Low_to_High_Channel;
53
54 end Channel;
```

Listing 9: Declaration of a `Channel` for element of type `Message`. Here, `Message` corresponds to the type `Element_Type`. This is an application level package leveraging the runtime library package `Channel_Pool_Access`.

Listing 9 illustrates how the instantiation of a channel occurs. An application-level package `Channel` is used to instantiate the `Shared_Pointer` generic package that embeds the channel implementation, provided in the extended runtime library within the `Channel_Pool_Access` package. The `Message` type corresponds to `Element_Type` in Fig. 3. It is application-specific, and the listing offers an example of a simple record with a `Field` of type `String`, as shown at line 8.



(a) A LO task creating and sending a message through a channel.



(b) A HI task receiving and freeing a message through a channel.

Fig. 4. Sequence diagrams – non-standard syntax – showing a Send and Receive exchange through a channel. Vertical bars stand for executions of actors (horizontal squares above the lifelines).

```
1  task body Low_Crit_Task is
2     Reference : Channel_Package.Reference_Type;
3  begin
4     . . .
5     loop
6         --  Allocate a new message object
7         Channel_Package.Allocate (Reference);
8
9         --  'Field' is an attribute of the
10        --  message type
11        Channel_Package.Get (Reference).Field
12           := "Value";
13
14        --  Send the message, hence lose
15        --  its ownership
16        Channel.Send (Reference);
17        pragma Assert
18           (Channel_Package.Is_Null (Reference)
19             = True);
20        . . .
21     end loop;
22 end Low_Crit_Task;
```

Listing 10: Some relevant sections of the body of a task allocating and sending a message through a channel. Notice that the receiver task is never mentioned in this scope.

Fig. 4(a) depicts how a HI task creates and sends a message through a channel across criticality levels. Fig. 4(b) displays how a LO task could access the message stored in a channel. The same use case is presented

in the Listings 10 and 11, showing some relevant portions of the body of two tasks exchanging a message.

As shown in lines 11 and 16, from Listing 10 and 11 respectively, a task aiming at accessing the content of a message, for either write or read operations, could do so by referring to the internal field of the message dereferenced with the `Get` procedure.

```
1  task body High_Crit_Task is
2     Reference : Channel_Package.Reference_Type;
3     value     : String (1..10);
4  begin
5     . . .
6     loop
7        --  Receive a message, if any,
8        --  from the channel
9        Channel.Receive (Reference);
10
11       --  Check if a message has been
12       --  successfully received
13       if Channel_Package.
14          Is_Null (Reference) /= False
15       then
16          value :=
17             Channel_Package.
18               Get (Reference).Field;
19       end if;
20
21       --  Deallocate the message object
22       Channel_Package.Free (Reference);
23       pragma Assert
24          (Channel_Package.Is_Null (Reference)
25            = True);
26       . . .
27    end loop;
28 end High_Crit_Task;
```

Listing 11: Some relevant sections of the body of a task receiving and deallocating a message through a channel.

4.2. Programmatic idioms

We propose an idiomatic programming approach to enforce isolation at the task level and the criticality level. In Ada, packages are the basic unit of modularization of the program structure: they are source-level artifacts. An Ada package contains three parts, one of which is optional: a public specification, which is the only part that is visible outside of the package itself; a private specification, which is visible only to the inside of the package and to its child packages, if any; and the implementation (called *body* in Ada speak), which cannot be viewed or accessed from the outside. We use packages to assure isolation among tasks and POs, which are the primary runtime entities of an Ada program in execution. Every individual task should be declared within a dedicated package: the rules of the language constrain the scope of visibility of that task to the internals of its own package. Visibility into other packages is achieved solely by prefixing an explicit directive of (visibility) inclusion to the declaration of one's own package: it is easy therefore to check statically who-views-what in the program as a whole. Ada strictness on the definition, type-bound use, and lifetime of pointers ensure that no intrusion can occur, whether intentionally or inadvertently that cannot be detected at compile time. In the same vein, every individual PO should be declared inside a dedicated package, to be solely included in the visibility scope of tasks at the same criticality level, pinned to the same core, which need to share data via such POs. In keeping with Ada's tenet, these visibility restrictions can be checked statically with ease. Tasks at different criticality levels can only functionally interact via channels. Again, every individual channel should be declared in a dedicated package and included in the visibility scope of the pertinent tasks. Again,

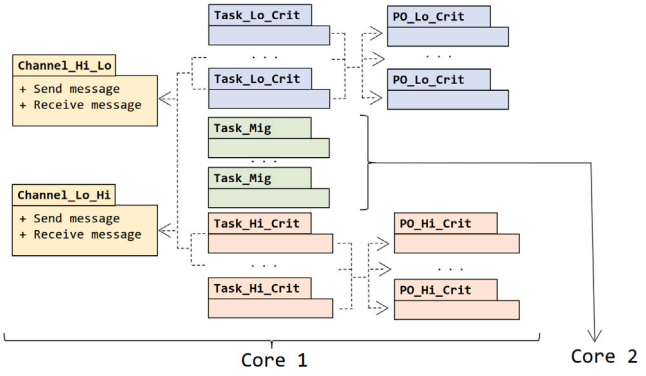


Fig. 5. Package diagram for the idiomatic programming approach introduced with this work. The package configuration is identical on the second core.

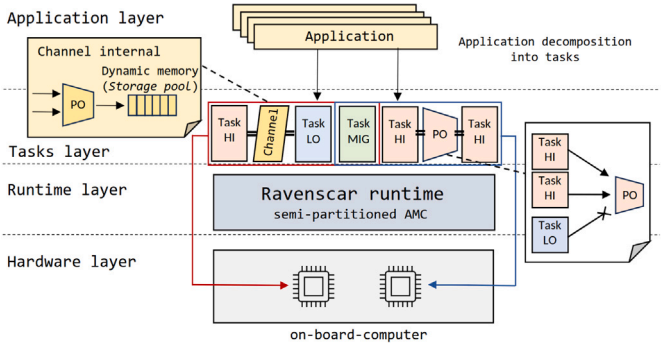


Fig. 6. Ravenscar architecture for MCS (with spatial isolation) – Our proposal, comprehensive of cross-criticality communication mechanisms (channels), where protected objects are specific for tasks at a certain criticality level.

static dependency checking is sufficient to discover illicit inclusions. Tracking dependencies for messages is equally possible, but much less interesting, really, in that messages always have exactly one owner at run time by design.

Fig. 5 describes graphically the idiomatic programming approach discussed here, where tasks, POs, and channels are declared inside of dedicated packages; in the image, colors are indicative of criticality levels. Notably, LO tasks marked as migratory are prevented from accessing any shared resource.

Fig. 6 illustrates the architecture of an application leveraging the extended runtime library, after the idiomatic programming style discussed here.

4.3. Deferred freezing

Introducing shared resources – channels in the particular case of this work – requires the provision of run-time mechanisms that prevent tasks from being frozen while holding any of them. In the model by [8], tasksets where LO tasks would fail to complete within the boundaries of their execution budget are deemed erroneous, and excluded from evaluation experiments. Consequently, the occurrence of a budget exceeded event is not contemplated in the response time analysis that ascertains the feasibility of the various execution scenarios. This exclusion may be reasonable in the context of laboratory experiments that use synthetic tasksets. When considering real-world applications, however, serious difficulties may arise in estimating the task execution budgets with sufficient accuracy. A real-world solution, therefore, should be able to respond soundly and promptly to any budget exceeded event, even if occurring within an otherwise expendable LO task. The deferred freezing provisions worsen the response time of HI tasks that happen to share resources with the LO tasks that incur the freezing event. Freezing

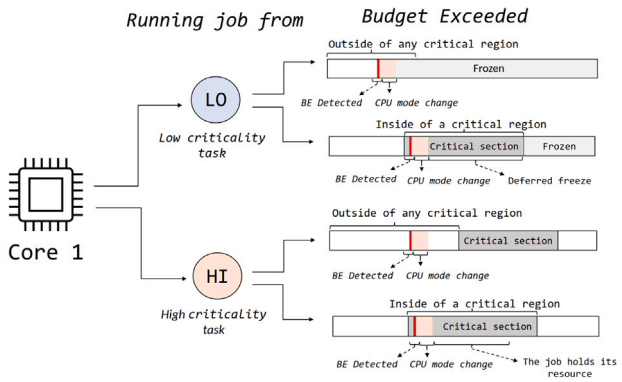


Fig. 7. Budget exceeded detected on Core 1 – Departing from the original MCS model in [8], in our solution the occurrence of a "budget exceeded" event occurring in a LO task is addressed through *deferred freezing*.

happens to LO tasks during a per-core LO-to-HI mode change. A task executing a critical section will not be frozen until it has released all its shared resources, as shown in Fig. 7. The runtime keeps track of whether a task holds a resource or not. When a mode change occurs, LO tasks holding a resource are allowed to remain active until they relinquish it, to be frozen right after that.

To implement this run-time mechanism, a flag is stored for each task, within the runtime-managed *Threads Control Block* (TCB): each time a task enters or exits a critical section, the flag is updated accordingly. In this manner, when a budget-exceeded event occurs, the runtime will scan all active LO tasks in its domain, checking the corresponding flag, to freeze immediately all of them that are currently running outside of any critical region. Those LO tasks that would still hold a shared resources at that point will be allowed to execute. (Of course, this eventuality will have to be addressed in response time analysis, as it would change the worst-case condition of cores.) When LO tasks in that condition eventually exit the critical region, running on a core that happens to be in HI mode, their control flag is updated, and the task is frozen until a HI-to-LO mode change occurs.

In our implementation, we chose to use deferred freezing for all LO tasks that happen to hold shared resources regardless with which other tasks at the time of the LO-to-HI mode change. The wisdom of this choice is for our runtime to provide a single, clear-cut semantics to the application during mode changes. Other policies might be devised in such situations, for example restricting deferred freezing just for LO tasks sharing resources (channels in our model) with HI tasks. Deciding what choice is optimal is not immediately obvious and would require further study that is outside of the scope of this paper.

5. Properties of the solution

The three additions outlined so far are sufficient to grant the following spatial isolation properties, grouped according to:

- lifetime (static/off-line vs. dynamic/run-time);
- isolation axis (by task vs. by criticality).

Table 1 lists all properties of interest.

Static configuration - by task. Static memory isolation by task is granted by:

- Idiomatic use of one distinct package for each task, hence task visibility is constrained to its own package.
- To prevent unintended modifications of any TCB (e.g., a task trying to alter its own stack boundaries) dependencies from a task to `System.Tasking` and `System.BB.Threads` should be forbidden. The GNAT compiler produces a warning for any dependency on `System`'s sub-packages. Hence, implementing a static check to this effect simply needs to compile with the `-gnatwe` flag. Inclusions directives made inside of sections of compilation units that are set to compiled with suppressed warnings (`pragma Warnings(Off)`) should be avoided as well.

Static configuration - by criticality. Each task can access only its own stack; however, communication mechanisms between tasks exist:

- Communication between tasks with the same criticality happens by means of POs, each declared and located in a dedicated package. Hence, illicit accesses from tasks with a different criticality level can be detected by static analysis, ahead of execution.
- Communication between tasks with different criticality levels happens by means of message-passing across statically defined channels, each one declared and located in a dedicated package. In this way, task dependencies to a specific channel could be assessed statically. On the other hand, messages are dynamic objects.

Dynamic/execution-time - by task. In the Ravenscar profile, allocation and deallocation from the standard global storage pool are forbidden by implicit application of pragma `No_Implicit_Heap_Allocations`. In the proposed model, only messages for inter-criticality communication are allocated dynamically. Allocation and deallocation for these messages are performed by the runtime library, through explicit allocators associated with a dedicated *storage pool*. Each message can be accessed at run time by exactly one task, referred to as the owner task, thanks to an ownership mechanism similar to Rust, and by actively forbidding the use of anonymous access types. Furthermore, this mechanism prevents errors such as multiple deallocations or dangling references.

Dynamic/execution-time - by criticality. Dynamic memory is the only space shared by tasks at different criticality levels, and the single-owner property also holds when ownership is moved across criticality levels.

6. Evaluation

6.1. Quality assessment

The effectiveness of the solution proposed here should be evaluated over two axes:

- The run-time cost incurred by our modified runtime library used in accord with the idiomatic programming style described earlier. The cost is comprehensive of the three features introduced. It does not consider the cost of formal verification, which we argue to be modest, given the size of a real-world application in the target domains.
- The runtime library, with the newly added features, should still be able to warrant temporal isolation; namely, the spatial isolation mechanisms introduced should complement the work of [10] without affecting the scheduler policy. Additionally, the programming model introduced here should complement the one adopted in [10], which can be done because the two address orthogonal concerns.

6.2. Run-time cost

To assess the run-time cost of the proposed runtime library, we measure the temporal cost of the four procedures exposed by a channel, altering the size of the message with the intent of highlighting the course of the time required for increasingly large messages. The experiment consists of a taskset of ten tasks, five HI-crit, and five LO-crit, exchanging messages of various payload sizes through the use of two channels with opposite directions, one from HI tasks to LO tasks, and another the other way around. Each message is an object containing a payload. The experiment has been repeated 10 times, applying linearly

Table 1
Isolation properties and their enabling mechanisms.

		Lifetime	
		Static	*Dynamic*
Isolation axis	*By Task*	• Idiomatic use of one distinct package for each task • Dependencies from a task to `System.Tasking` and `System.BB.Threads` should be forbidden. To implement this static check is sufficient to compile with the `-gnatwe` flag and without suppressed warnings (`pragma Warnings(Off)`)	• Dynamic memory is used exclusively for cross-criticality communications • Allocations and deallocations are performed by the runtime library, leveraging on a dedicated *storage pool* • Each dynamic message always has one owner at runtime
	By Criticality	• Each task can access only its own stack • Communication between tasks with the same criticality happens by means of POs, each one defined in a dedicated package • Communication between tasks with different criticality levels happens through channels, each one declared in a dedicated package	• Dynamic memory is the only space shared by tasks at different criticality levels • Single-ownership is preserved after ownership is moved across criticality levels

increasing payload size, and monitoring the execution time spent for each procedure involved in the message life cycle; measurements are then sent directly to an output interface where they are collected. An iteration of the experiment is concluded after at least 1000 sample measures have been collected.

This configuration is intended to provide measurements of the absolute cost of the runtime procedures presented in this work. An extensive comparison of our solution against a PO-based solution would be of no interest here, as POs and channels serve radically different purposes in our model. Incidentally, it is worth noting that a channel devoid of dynamic memory would effectively be a normal PO, while normal POs cannot support ownership without relying on dynamic memory.

The target environment is a Digilent Cora development board, with installed a Zynq 7000 SoC, featuring a dual-core ARM processor. An Ada application compiled over our runtime library has been flashed over the target hardware, while a Python script has been used to retrieve the measurements sent by the application over a USB/UART interface.

The experiment setup is available at [35], while the runtime library with the extensions required to support spatial isolation is available at [36].

The proposed solution encompasses three new features:

- An idiomatic use of the language features offered by the Ravenscar runtime.
- A mechanism to allow communication between pairs of tasks at different criticality levels, with message passing through channels.
- An extension of the scheduler for deferring freeze for LO tasks that happen to be communicating (hence holding a shared PO) when a mode change event occurs.

For each of these features, we discuss here the run-time cost incurred during the experiment.

Idiomatic use of language features. All additional checks are performed during static analysis; hence the run-time cost of this feature is null.

Message passing through channels. The results of our experiment are shown in Fig. 8.

For ten linearly spaced message-payload sizes, ranging from 0 to 1000 KB, 1000 samples have been acquired for each of the four procedures altering ownership, namely allocation (Fig. 8a), deallocation (Fig. 8b), send (Fig. 8c), and receive (Fig. 8d). To improve readability, only five data points are shown in the above plots. The application from which the measurements have been taken has been developed according to the idiomatic approach described earlier in this paper. Those measurements show that the run-time cost experienced during message allocation and deallocation does not increase linearly with the message size, on the reference hardware. The measurements also show that the time for placing (Fig. 8a) and removing (Fig. 8b) a message from a channel's storage pool always is less than 6 microseconds, varying in a range from 2.5 to 4.7 microseconds. Therefore, sending and receiving a message, hence moving ownership toward and from a channel, also have a near-constant cost.

With channels, contention arises from dynamic memory management (allocation and free procedures), and transfers of ownership (send and receive procedures); reading and writing access costs are paid only by the owner task, which is exempt from contention by definition. Overall, cross-criticality message-based communication incurs run-time overhead at three moments:

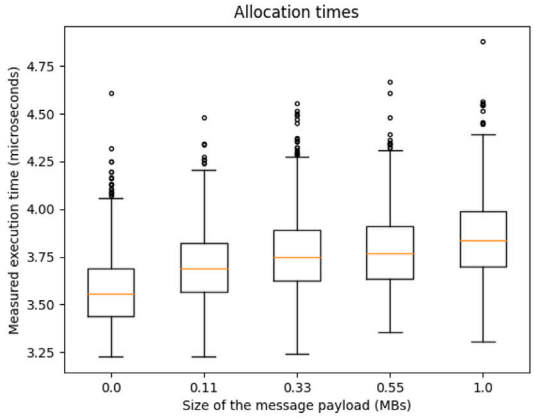
- During message allocation: the run-time cost of allocating a message is shown to not increase linearly with the size of the message (range [0–1 MB]), instead staying nearly constant for the target hardware.
- During a send or receive operation through a channel: these operations are implemented as protected procedures. The run-time cost of both operations is nearly constant, as it corresponds to the time required for exchanging a reference. Notably, executing this operation can generate priority-inversion blocking, which should be accounted for during response time analysis (RTA).
- During message deallocation: the situation is analogous to the case of allocation.

Scheduler's extension. A task executing inside a critical section should not be frozen in the event of a mode change, but its exclusion from the ready queue should be deferred to the end of the (outermost) critical section. This behavior has been implemented in the existing runtime library. The effects of this event on the schedulability of the system should be thoroughly evaluated with RTA. It might be argued that such a situation should be regarded as a system error needing contingency handling.

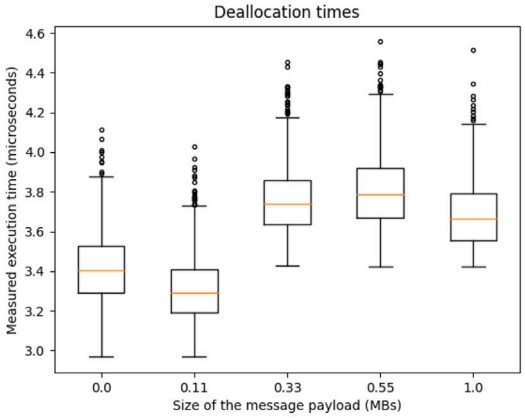
Additional cost. Following the Rust ownership approach to dynamic memory handling, heap (*storage pool*) memory exhaustion is not addressed by our runtime and should be managed by the application itself. A way to do so is to limit the number of tasks that can exchange data across different criticality levels, so as to ease static analysis of the maximum amount of memory that might be required for these operations during operation.

6.3. Features compatibility

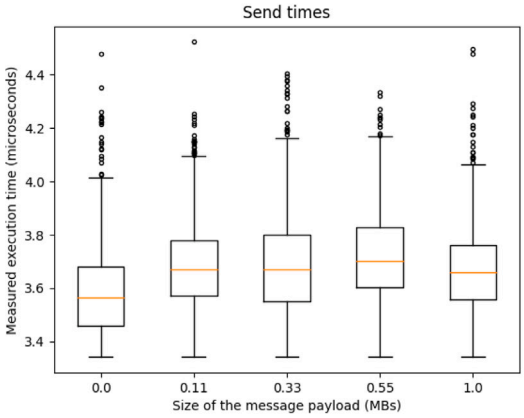
At the core of the contribution presented in [10] lies a scheduling policy that gives precedence to high-criticality tasks in the face of transient overload situations. In this work, we develop a spatial isolation strategy mindful of that scheduling algorithm, which we carried in our experimental application. The only alteration that we applied to the scheduler presented in [10] consists of the addition of the deferred



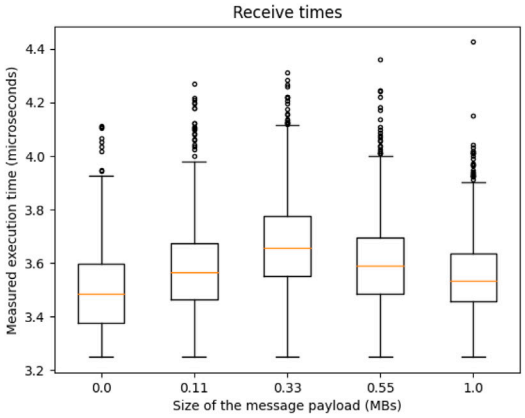
(a) Run-time cost for the allocation of messages with increasing size.



(b) Run-time cost for the deallocation of messages with increasing size.



(c) Run-time cost for a `Send` procedure over messages with increasing size.



(d) Run-time cost for a `Receive` procedure over messages with increasing size.

Fig. 8. The cost during execution-time of the four main procedures developed to support messages creation and exchange, in our experimental hardware, tend to be constant with an increase in message size, ranging from 0 to 1 MB. The overall cost of each operation seems never to be greater than 6 microseconds.

freezing mechanism. Deferring the suspension of low criticality tasks holding resources does not compromise the original guarantees of temporal isolation, but would definitely worsen system schedulability. As a result, the response time analysis (RTA) should be altered as well. Such an extension is not part of this work, but we deem it perfectly doable within the remit of the base scheduling model.

7. Conclusions

With this work, we aim to support the adoption of MCS in real-world applications in safety-critical scenarios. In this context, the sole guarantee of temporal isolation, however precious, is not sufficient to warrant adoption over the existing partitioning-based approaches, due to missing support for spatial isolation.

The model we have proposed here evolves from a concurrency model, that of the Ravenscar profile, developed for highly critical applications, amenable to static analysis and enriched by an instance of an AMC scheduler, developed by [10]. To provide spatial isolation guarantees comparable to TSP systems, we contribute three extensions: (1) an idiomatic programming style, which lends itself to static checking; (2) a cross-criticality communication mechanism inspired on Rust's ownership model; and (3) an extension of the scheduling policy that defers the freezing of LO tasks that hold POs when a LO-to-HI mode change occurs. The resulting solution is arguably sufficient to warrant temporal *and* spatial isolation.

A further contribution of this work is the extension of the Ravenscar runtime library with MC support, available here [36] under an open-source license.

Future works might explore two directions: exploring (1) how our runtime solution could be extended to address the fault dimension of isolation in MCS, and (2) more sophisticated policies of resource sharing fit for MCS, as discussed in [37].

CRediT authorship contribution statement

E. Tinto: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Investigation, Formal analysis, Data curation. **T. Vardanega:** Writing – review & editing, Writing – original draft, Validation, Supervision, Project administration, Methodology, Investigation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The source code developed in this work has been published through open-source repositories on GitHub, and the corresponding links can be found in the bibliography.

Acknowledgments

We wish to express our gratitude to Mattia Bottaro for his kind transfer of knowledge and technology that jump-started our work in this project. We are also grateful to the anonymous reviewers of this paper, for their thorough and insightful comments, and suggestions.

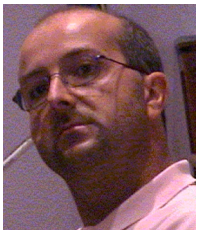
Funding sources

The work carried out in this project was partially funded under the National Recovery and Resilience Plan (NRRP), Italy, Mission 4 Component 2 Investment 1.4 - Call for tender No. 3138, 16 December 2021, by the Italian Ministry of University and Research funded by the European Union – NextGenerationEU.

References

[1] Aamir Mairaj, Preferred choice for resource efficiency: Integrated modular avionics versus federated avionics, in: 2015 IEEE Aerospace Conference, 2015, pp. 1–6.
[2] K. Ghose, S. Ray, O. Demir, D. Hogea, J. Imperato, A time and space partitioned avionics real-time file system, in: 24th Digital Avionics Systems Conference, Vol. 1, 2005, pp. 6.C.3–61.
[3] Justin Littlefield-Lawwill, Larry Kinnan, System considerations for robust time and space partitioning in integrated modular avionics, in: 2008 IEEE/AIAA 27th Digital Avionics Systems Conference, 2008, 1.B.1–1–1.B.1–11.
[4] Steve Vestal, Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance, in: 28th IEEE International Real-Time Systems Symposium, RTSS 2007, 2007, pp. 239–243.
[5] Shibarchi Majumder, Jens Frederik Dalsgaard Nielsen, Thomas Bak, Ærø : A platform architecture for mixed-criticality airborne systems, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (10) (2020) 2307–2318.
[6] Héctor Pérez, J. Javier Gutiérrez, Enabling data-centric distribution technology for partitioned embedded systems, IEEE Trans. Parallel Distrib. Syst. 27 (11) (2016) 3186–3198.
[7] S.K. Baruah, A. Burns, R.I. Davis, Response-time analysis for mixed criticality systems, in: 2011 IEEE 32nd Real-Time Systems Symposium, 2011, pp. 34–43.
[8] Hao Xu, Alan Burns, Semi-partitioned model for dual-core mixed criticality system, in: Proceedings of the 23rd International Conference on Real Time and Networks Systems, RTNS '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 257–266.
[9] H. Xu, A. Burns, A semi-partitioned model for mixed criticality systems, J. Syst. Softw. 150 (2019) 51–63.
[10] M. Bottaro, T. Vardanega, Evaluating a multicore mixed-criticality system implementation against a temporal isolation kernel, J. Syst. Archit. 130 (2022) 102688.
[11] Mattia Bottaro, Evaluating a multicore Mixed-Criticality System implementation against a temporal isolation kernel, https://github.com/BottCode/Ada-RTE-supporting-semi-partitioned-model.
[12] fentISS, XtratuM, https://www.fentiss.com/xtratum/.
[13] Alan Burns, Brian Dobbing, Tullio Vardanega, Guide for the use of the ada ravenscar profile in high integrity systems, Ada Lett. XXIV (2) (2004) 1–74.
[14] Matthieu Lemerre, Emmanuel Ohayon, Damien Chabrol, Mathieu Jan, Marie-Benedicte Jacques, Method and tools for mixed-criticality real-time applications within pharos, in: 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, 2011, pp. 41–48.
[15] Zubin Hu, Jianchao Luo, Xiyu Fang, Kun Xiao, Bitao Hu, Lirong Chen, Real-time schedule algorithm with temporal and spatial isolation feature for mixed criticality system, in: 2021 7th International Symposium on System and Software Reliability, ISSSR, 2021, pp. 99–108.
[16] Santiago Urueña, José A. Pulido, Jorge López, Juan Zamorano, Juan A. de la Puente, A new approach to memory partitioning in on-board spacecraft software, in: Fabrice Kordon, Tullio Vardanega (Eds.), Reliable Software Technologies – Ada-Europe 2008, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 1–14.
[17] Michael Zimmer, David Broman, Chris Shaver, Edward A. Lee, FlexPRET: A processor platform for mixed-criticality systems, in: 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium, RTAS, 2014, pp. 101–110.
[18] Joost Hoozemans, Jeroen van Straten, Stephan Wong, Increasing resource utilization in mixed-criticality systems using a polymorphic VLIW processor, J. Syst. Archit. 84 (2018) 2–11.
[19] Zaher Owda, Roman Obermaisser, A predictable transactional memory architecture with selective conflict resolution for mixed-criticality support in mpsocs, in: 2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing, 2015, pp. 158–162.
[20] Zaher Owda, Moisés Urbina, Roman Obermaisser, Mohammed Abuteir, Hierarchical transactional memory protocol for distributed mixed-criticality embedded systems, in: 2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress, DASC/PiCom/DataCom/CyberSciTech, 2016, pp. 334–343.
[21] Micaiah Chisholm, Namhoon Kim, Bryan C. Ward, Nathan Otterness, James H. Anderson, F. Donelson Smith, Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems, in: 2016 IEEE Real-Time Systems Symposium, RTSS, 2016, pp. 57–68.
[22] Benjamin Lesage, Isabelle Puaut, André Seznec, PRETI: Partitioned real-time shared cache for mixed-criticality real-time systems, in: Proceedings of the 20th International Conference on Real-Time and Network Systems, RTNS '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 171–180.
[23] Namhoon Kim, Bryan C. Ward, Micaiah Chisholm, James H. Anderson, F. Donelson Smith, Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning, Real-Time Syst. 53 (5) (2017-09) 709–759, 3.367.
[24] ARM Developer Documentation, What is TrustZone? https://developer.arm.com/documentation/102418/0101/What-is-TrustZone-.
[25] Pan Dong, Alan Burns, Zhe Jiang, Xiangke Liao, TZDKS: A new TrustZone-based dual-criticality system with balanced performance, in: 2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA, 2018, pp. 59–64.
[26] Zhe Jiang, Pan Dong, Ran Wei, Qingling Zhao, Yankai Wang, Dizhong Zhu, Yan Zhuang, Neil Audsley, PSpSys: A time-predictable mixed-criticality system architecture based on ARM TrustZone, J. Syst. Archit. 123 (2022) 102368.
[27] David Griffin, Iain Bate, Robert I. Davis, Generating utilization vectors for the systematic evaluation of schedulability tests, in: 2020 IEEE Real-Time Systems Symposium, RTSS, 2020, pp. 76–88.
[28] Joël Goossens, Christophe Macq, Limitation of the hyper-period in real-time periodic task set generation, in: Proceedings of the 9th International Conference on Real-Time Systems, 2001.
[29] H.J. Curnow, B.A. Wichmann, A synthetic benchmark, Comput. J. 19 (1) (1976) 43–49.
[30] The Rust Programming Language Documentation, Rust, https://www.rust-lang.org/.
[31] The Rust Programming Language Documentation, Using threads to run code simultaneously, https://doc.rust-lang.org/book/ch16-01-threads.html.
[32] Cyrille Comar, Claire Dross, Florian Gilcher, Yannick Moy, Dynamic memory management in critical embedded software, https://www.adacore.com/papers/dynamic-memory-management-in-critical-embedded-software.
[33] The Rust Programming Language Documentation, What is ownership? https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.
[34] Javier Miranda, Edmond Schonberg, Protected objects, in: GNAT: The GNU Ada Compiler, Free Software Foundation, 2004.
[35] Edoardo Tinto, Providing-spatial-isolation-experiment, https://github.com/TintoEdoardo/Providing-Spatial-Isolation-Experiment.
[36] Edoardo Tinto, Evaluating a multicore mixed-criticality system implementation against a temporal isolation kernel, https://github.com/TintoEdoardo/Ada-RTE-supporting-semi-partitioned-model.
[37] Nan Chen, Shuai Zhao, Ian Gray, Alan Burns, Siyuan Ji, Wanli Chang, MSRP-FT: Reliable resource sharing on multiprocessor mixed-criticality systems, in: 2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium, RTAS, 2022, pp. 201–213.

Edoardo Tinto, M.Sc. @ Padova, IT (2022), in February 2023 has enrolled as a Ph.D. student in the Brain, Mind, and Computer Science course at the University of Padova (IT). As a master student, he specialized in distributed and real-time systems. In his Ph.D. project he investigates the technical solutions to support the edge-to-cloud continuum of computing. The present submission proceeds from his master's thesis.



Tullio Vardanega, M.Sc. @ Pisa, IT (1986), Ph.D. @ TU Delft, NL (1998), is at the University of Padua, Italy, since January 2002. After working as PI from 1987 until mid-1991, he was with the European Space Agency (NL) until December 2001. He specializes in high-integrity real-time systems, edge-to-cloud continuum, software engineering, active learning, and informatics education. He has run several research collaborations in those ambits. He is a member of IEEE and ACM. He is the technical expert for Italy in ISO/IEC JTC1/SC22: WG9 (Ada) and WG23 (Programming Language Vulnerabilities). Since 2004, he has been the chairperson of Ada-Europe.