Hard Real Time and Mixed Time Criticality on Off-The-Shelf Embedded Multi-Cores

Albert Cohen*, Valentin Perrelle†, Dumitru Potop-Butucaru*, Marc Pouzet*, Elie Soubiran‡, Zhen Zhang*
*INRIA and DI, ENS, Paris, France,
†CEA, LIST, 91191, Gif-sur-Yvette, France,
‡Alstom Transport and IRT SystemX, France

***Abstract*—The paper describes a pragmatic solution to the parallel execution of hard real-time tasks on off-the-shelf embedded multiprocessors. We propose a simple timing isolation protocol allowing computational tasks to communicate with hard real-time ones. Excellent parallel resource utilization can be achieved while preserving timing compositionality. An extension to a synchronous language enables the correct-by-construction compilation to efficient parallel code. We do not explicitly address certification issues at this stage, yet our approach is designed to enable full system certification at the highest safety standards, such as SIL 4 in IEC 61508 or DAL A in DO-178B.**

***Index Terms*—Mixed criticalities, Multi-core, Embedded real-time system, Synchronous Language, Time-triggered execution**

I. Introduction

This paper presents the design of a synchronous language enabling hard real-time applications to run on off-the-shelf multi-core platforms.[1] The language and methodology ensure the isolation of time-critical tasks from the non-time-critical ones under the following three hypotheses:

1) most of the computational load takes place in non-time-critical tasks;
2) it is possible to program the reaction to the absence of timely data, when non-time-critical tasks are delayed;
3) the target multiprocessor provides means to strictly prioritize memory accesses of one or more processors executing time-critical tasks, or to fully isolate such accesses into a scratch-pad memory; and the target also supports asymmetric multiprocessing (e.g., bare-metal execution on one core and Linux on another).

We illustrate our approach and validate it on a train signaling use case provided by Alstom Transport. It is representative of the complexity in terms of vital/non-vital code interweaving, operational performance and availability constraints. The system function is called "Passenger Exchange" (PE). This function takes control of the train when safely docked at a station; it organizes the exchange of passengers (train and station doors opening/closing) while protecting them from any untimely train movement or non-aligned doors opening, and finally gives the departure authorization when all safety conditions are met. The functional specification is made of more than 300 requirements (natural language and SysML), and the system function is composed of about twenty sub-functions.

The PE application is partitioned into tasks, some of them being safety-critical and hard real-time, and some of them being mission-critical but non-vital. These tasks expose input and output signals and may result from the compilation of a synchronous block-diagram language. Unlike most related work on mixed criticality [2], *dependences and communication among tasks of different criticality are allowed*. There is a simple reason why we can afford such a breach of criticality partitions: our approach composes tasks of different *time*-criticalities, *without relaxing any other validation requirement*. In other words, *all tasks may still be certified at the highest (relevant) level of safety*, but we acknowledge that *only a subset of the tasks needs to be time-predictable and validated against real-time constraints*. Let us discuss this hypothesis on the PE application. The computation of the doors that are safe to open (e.g., because they are not aligned) is safety-critical, as well as the task preventing train departure if safety conditions are not met (e.g. the doors are open or opening). Less vital tasks are in charge of operating doors with respect to the mission and to a time table; these are still mission-critical since the quality of service depends on the rare occurrence of timeouts. In this example we identify two occurrences of mission-critical to safety-critical communication. First, in order to ensure that door commands (mission-critical) do not lead to an accident, they must be checked against the enabled set of doors (safety-critical). Second, the departure authorization (safety-critical) must be computed regarding door commands to ensure that no opening commands will be executed after the authorization has been given. Such communication patterns are quite common in the case study.

In this paper, we will be using a modified version of the PE application. These modifications decouple computational aspects of the original safety-critical components. These computational tasks are amenable to parallelization and aggressive optimization, while satisfying a relaxed set of soft real time constraints. As a result, a safety critical component is split into a non-time-critical and a time-critical task, both of them being certified at the highest levels of safety. On the other hand, several less critical components with no connection to safety critical tasks have been coalesced for didactic reasons.

In the following, we focus on the validation of the hard

[1]This work is supported by the Technological Research Institute (IRT) SystemX, partially funded by the French public program "Investissement d'Avenir". It is also partially funded by the EMC2 ECSEL and ITEA ASSUME projects.

real time requirements of the time-critical components of the system. The difficulty being that interferences on shared buses and caches of conventional multicores make it impossible in general to establish a practically useful worst-case execution time of a given task [16]. Our goal is to design a software stack and composition methodology enabling hard real-time control code to be isolated from timing interference, while exploiting parallelism among non-time-critical tasks, and still allowing for communications between the two. To solve this apparently paradoxical and infeasible set of constraints, our language provides an automatic inference mechanism for "late" communications between time-criticality levels. Time compositionality may be implemented through mode changes, and introduced incrementally into existing design and validation flows.

II. State of the art

Thread-level parallelism has become unavoidable in any area where performance matters. While specific designs are emerging that combine predictability and performance—e.g. [16]—off-the-shelf multiprocessors designed for mass-market areas are not well suited to timing analysis. Indeed, it is necessary to establish strict bounds on the worst case execution time (WCET) to address the time-predicability requirements of safety-critical systems [19]. If contention of shared resources can not be avoided, the complexity and imprecision of these techniques worsens dramatically. A survey of these researches can be found in a recent paper [12]. Our approach is not to improve timing analyses themselves, but to make those more effective by *controlling how the system is designed, from specification to code generation*. We also hope to reduce the reliance on timing analysis on large parts of its code: *ideally, most of the software components would not need a fully safe worst-case execution time characterization, even though the full system remains globally time-predictable and provably safe*.

We are interested in mass-market commercial off-the-shelf (COTS) platforms, but we believe our approach will also be applied to more predictable classes of multiprocessors, such as the Kalray MPPA [14], [8], increasing resource efficiency. Such extensions are left for future work. Numerous hardware components impact WCET analysis on multiprocessors [7], such as shared caches and shared busses, etc. The main solutions attempt to reduce the general problem to a composition of sequential WCET analyses, enforcing a strict isolation at all levels of the memory hierarchy. For example, software-cache partitioning has been realized for ARM Cortex A9 [18], and other approaches using scratch-pad and multi-ported memories are also possible. Our proposal builds on these ideas to implement spatial and temporal partitioning.

Our proposal was also inspired by the Logical Execution Time (LET) paradigm [15] where the correctness of the system relies on observable input and output times independently of the actual execution time of the system's components. We extend LET with communications across time-criticality partitions, introducing a new protocol for tightly controlling timing isolation. We also leverage the multiple levels of time-criticality in real-world safety-critical applications to relax the timing isolation of parts of the system, improving overall efficiency and reducing certification costs without jeopardizing the safety of the full system. This approach has also been applied to Automotive control applications, but without enforcing hard timing isolation and compositionality [4].

Finally, when compared with the state of the art in mixed-criticality real-time scheduling, our paper proposes a totally different approach. Isolation between low-criticality and high-criticality components is ensured not only temporally, but also functionally, by means of language design and code generation. This approach is fully complementary to the mixed-criticality task models proposed in the real-time scheduling community. As such, it could be a contribution towards aligning academic work on mixed-criticality systems with the notion of mixed criticality introduced in industry standards [11].

III. A mixed-time-critical synchronous language

We designed a simple mixed-time-critical extension of an existing synchronous dataflow language: Heptagon [13].[2] It is a research language and compiler with a Lustre-like syntax, analogous to the textual language of Scade Suite.[3] Heptagon features state of the art constructs such as finite state machines and functional arrays with in-place operations. Its compiler implements a clock calculus upon which the generation of efficient embedded code is built,[4] and a number of optimizations to reduce control flow and memory management overhead.

The original PE application has been completely implemented in Heptagon, with only low-level I/O and system calls implemented in C. This choice allowed to faithfully implement the original specification, facilitating the application of formal methods or manual certification procedures. Although using mainly a natural language, the specification describes the functions to be implemented in an equational way which suits easily a dataflow language. Heptagon has been used to describe the tasks themselves as well as the target-specific code which describes how tasks communicate. We were also able to compile and test the different components early, then proceed with their integration and static scheduling, preserving timing isolation in a compositional way. In later development and validation stages, the tasks have been identified and mapped to specific processors, scheduled and executed in separation, with no functional changes to the program and reusing its high-level communication code. The detailed presentation and discussion of the parallelization and distribution features is out of the scope of the paper (some of the principles can be found in Gérard et al. [5] and

[2]See http://heptagon.gforge.inria.fr for documentation, source code and applications.

[3]http://www.esterel-technologies.com/products/scade-suite.

[4]Clocks can be seen as a type system for sequences of boolean conditions controlling the presence or the absence of values in stream variables, or the stepping of synchronous, stateful nodes in a process network.

Delaval et al. [9]). Instead, we outline the proposed extensions to HEPTAGON, and in particular how to programmatically control what happens when a task misses its deadline. For this purpose, we extend our language with the notion of "tasks" and "punctuality".

A *task* is defined as a dataflow node which is the smallest partition of the synchronous program after static scheduling and code generation. Tasks may then be amenable to dynamic or time-triggered scheduling. A task is a reactive program, a property inherited from HEPTAGON nodes: it activates repeatedly in response to a signal, its inputs need to be present before it activates (dataflow semantics) and the task's outputs are present after it terminates its reaction. After the beginning of the task and before its end, the task does not communicate with any other task. It is the programmer's duty to choose which dataflow nodes in the node instantiation tree will be tasks and thus to define the granularity at which the application is deployed on the target platform.

If ever a task were to be instantiated inside another task, the compiler would simply ignore this information and continue as if the task was a simple node. This does not guarantee time and space isolation across the "embedded" and "embedding" task, but we found it sufficient to ensure time and space isolation across top-level tasks in the system.

Note: the HEPTAGON language allows the programmer to declare "external" nodes and tasks which can be resolved at link time. This allows specific nodes to be written in plain C.

A task is called *unpunctual* if it is not time-critical. In this case, the task can miss its deadline. It may be left to run to completion, ignoring its outputs, or it may be killed or preempted by the system. When it happens, the outputs of the task are not present. To reflect this possibility, we say that the outputs of the task are also unpunctual. We extend the type system of HEPTAGON by assigning a punctuality to each variable and each task. Our language requires that the punctuality of the output variables is always the same as the punctuality of the task. These unpunctual values can be transmitted to other nodes. However, to exploit these values, the programmer must distinguish two cases: either the value has been computed on time or the unpunctual computation timed out and it is not available. The operator `ontime` takes an unpunctual variable argument and returns a boolean which evaluates to true when the variable can be read. In other words, the expression `ontime v` is the clock of the unpunctual variable `v`. This means that the `merge` operator—combining multiple flows with mutually exclusive clocks in HEPTAGON—can be used to build a punctual value by combining the actual value when present and a "default" value otherwise.

In many cases, the "default" value to be used in place of the actual value when a task missed its deadline is a static constant. This happens when there is a value which is always safe. For instance in our case, it is always safe that the component sends no commands or to considers that there is no door correspondence. The programmer can then give a default value to the unpunctual variable which will be used when the actual value is not available. When an unpunctual variable is set to its default value, its clock is the one of the node. (Or the one explicitly declared if it is different.) Sometimes, specifying a default value is not powerful enough. For instance, in Model Predictive Control (MPC), a more suitable reaction may be to repeat the command issued in the last cycle but following a suboptimal, faster prediction [17], [1].

IV. MIXED-TIME-CRITICAL FLOW AND PLATFORM

We review the tool flow and run-time support for the compilation and execution of synchronous programs with multiple levels of time criticality.

A. Compilation

The extensions to HEPTAGON require three main changes.

1) The `task` keyword is a synonym of `node`. It is supported as an annotation in the intermediate representation, which can then be read by the back-end to generate a scheduling table [6] or communication code.
2) The type system is extended with a punctuality property.
3) A transformation pass abstracts the punctuality information such that the resulting program have the same semantics but is lowered to conventional synchronous code without the mixed-time-criticality constructions. Further down, the usual compilation flow can be applied.

We add a "punctuality" attribute to the type of expressions and variables. The type system verifies that variables assigned from an unpunctual task application are also unpunctual and that unpunctual variables may only be passed as argument when the corresponding parameter is also unpunctual. The new construct `ontime` is always typed as boolean.

The additional transformation pass uses clocks to represent the punctuality. Each unpunctual variable is split into two variables: one is a boolean signal representing whether the value has been computed on time or not, the other is the actual value which is only defined on the former clock. The `ontime v` expression for any variable `v` is translated to either `true` if the variable `v` is punctual or to the clock of `v` otherwise.

A task with an unpunctual parameter is split the same way. HEPTAGON allows one parameter to be the clock of another and thus supports this construction.

When a default value is provided for an unpunctual variable, the real value is selected when present, and the default one otherwise; it corresponds to the `merge` construct in HEPTAGON.

The clock variable for each unpunctual variable needs an input from the system telling whether the task generating the encapsulated value has completed on time or not. We introduce for each task call a fresh abstract function which provides this information. It is the responsibility of the back-end to stub these functions appropriately.

It is worth noting that by relaxing the type system to be a bit more lenient on unpunctual arguments we could have allowed interesting constructions, at the cost of additional compilation efforts: since unpunctual tasks are not time-critical, it should be possible to execute them one after another, waiting for the previous task to terminate normally without killing it when

```
node check_command(door_command : command; door_map : int)
   returns (safe_command : command)
let
  safe_command = if door_map <> -1 then door_command else None;
tel

task check_commands(unpunctual door_commands : command^n; door_map : int^n)
   returns (safe_commands : command^n)
let
  if ontime door_commands then
    safe_commands = map<<n>> check_command(door_commands, door_map);
  else
    safe_commands = None^n;
  end
tel
```

Figure 1. Simplified implementation of a safety-critical function which ensures that there are no command for a train or platform door which is not aligned with a corresponding platform or train door, respectively. The `door_map` array provides a valid description of which pairs of doors are aligned, and the `door_commands` is the array of commands computed for each door. The latter is *unpunctual* as it results from a best-effort computation in a mission-critical task; when commands are not computed in time, it is always safe for the passenger that the doors receive no commands.

a deadline is missed. At worst, the unpunctual task missing its deadline could be preempted to ensure it does not use resources (CPU, memory, etc.) which belong to any time-critical task. Thus depending on the criticality of a task, the behavior to follow when a data is not computed on time would not be the same. If the task is time-critical, we use the constructions introduced in this paper to program the temporary transition to a degraded mode. If the task is not time-critical then it can wait until its data are computed. This solution may accumulate delay if multiple tasks miss their deadline or the delay of a task can be compensated by another. For this purpose, we could have allowed unpunctual arguments to be passed to unpunctual tasks without declaring them as unpunctual, but this would involve heavier changes in the type system.

B. Distribution and parallel code generation

At this stage of our research, the code generation procedure is only partially implemented and thus semi-automatic.

Like in LUSTRE, a HEPTAGON program can be seen as a dataflow graph where vertices are operators or instances of other nodes. A program is defined by its root node. All nodes instanciated from a parent task node can be statically scheduled by the HEPTAGON compiler into a single step function. Calls to the node's internal operators and child nodes may even be inlined if desirable.[5] After the program has been compiled, we obtain a dataflow graph whose vertices are tasks and whose edges are communications between these tasks.

In our case study, most dataflow operators are copies of outputs of one task to the input of another, or field selection where a subset of the data is being copied. Other operators, such as arithmetic ones, are also encapsulated into tasks. The punctuality of an encapsulating task is completely determined by the type system which imposes that the operands and the result of an operators must have the same punctuality.

In summary, the first transformation pass produces a dataflow graph where, after inlining and encapsulation, all vertices are tasks and where edges are communications. These tasks must then be allocated and scheduled. Although the HEPTAGON compiler has its own scheduler, it is intended for pure sequential scheduling and for a totally different optimisation goal. Thus, our approach relies on the existence of external tools to allocate and schedule tasks. These tools must take into account the cost of communications as well as the individual execution cost of tasks. For instance, we have implemented a back-end for the LOPHT [3] scheduler: this back-end allow us to feed the scheduler with the list of tasks, data dependencies, clocks and constraints. LOPHT then produces a scheduling table for all these tasks. [6].

Basically, the allocator gives a color to each vertex in the dataflow graph while the scheduler gives a topological order to those vertices. In our framework, allocation just splits punctual and unpunctual tasks. For each computation resource—i.e. for each color in the dataflow graph—one may generate sequential code. This code calls each task in the order given by the scheduler. When the input of a task is produced by another task on the same computation resource, a simple data copy takes place. When the two tasks are allocated in different computation resources, the generated code uses the communication protocol described in the next section.

C. Partitioning and time-triggered communication protocol

Systems with mixed time-criticalities require a strong assurance on the worst case execution time (WCET) of most safety critical tasks. However this is almost impossible to achieve without temporal and spatial partitioning, due to the shared resources of conventional multi-cores [16]. Concretely, concurrent accesses on shared resources, such as a shared L2 cache or a shared scratch-pad memory bank, result in contentions, which prohibit timing analysis at the system level. Building on partitioning techniques, we propose a time-triggered communication protocol to realize contention-avoidance and timing isolation.

A strict spatial partitioning requires independent physical memory and computing units. This is for two reasons: first, to

[5]Recursion is forbidden.

keep the register and memory context of safety-critical tasks away from malware or bugs; second, to avoid concurrent accesses on shared resources, including instruction caches. Such spatial partitioning can be achieved on off-the-shelf hardware platforms, such as the Zynq 7K [23]. It integrates an ARM MPcore (Dual Cortex-A9) and a FPGA permitting asymmetric multiprocessing (AMP) configurations. Each ARM core can run its own software stack, set up separately in the shared DDR or on chip memory (OCM). Precise timers and bus arbitration policies can also be controlled. Moreover, building on the embedded FPGA, a variety of memory and computing units can be implemented, e.g., for communication buffers, and for controlling external I/O. Therefore the time-critical and non-time-critical tasks can be fully isolated on the physical platform. This does come with a significant cost however, as communications and DDR accesses from the time-critical level must be temporally isolated from any other activity. Such temporal isolation may dramatically reduce the effective parallelism available in the program. In particular, to the best of our knowledge, it is not possible to execute two time-predictable tasks in parallel on such a platform.

Our time-triggered communication protocol aims at realizing sufficient temporal partitioning without destroying the potential for parallel execution. This protocol controls not only shared communication buffers by building access time frame and deadline in order to avoid contentions. But it can also tolerate timeout events in order to avoid delaying the time-critical functions by non-time-critical ones. And it achieves this inspite of the presence of communications between the two levels.

In this protocol, each task is split into phases falling in one of six classes:

- time-critical computing (TCCP),
- time-critical copy to buffer (TCTB),
- time-critical copy from buffer (TCFB),
- non-time-critical computing (NTCCP),
- non-time-critical copy to buffer (NTCTB), and
- non-time-critical copy from buffer (NTCFB).

To set the execution deadlines, we rely on a supplied WCET for each of these phases. Fig. 2 shows a sample chronogram of the time-triggered protocol. In this simple example, there are two parallel execution chains, one for safety-time critical tasks, another for mission critical ones, they are executed in parallel on different CPUs.

Phase I: On the time-critical level, at the end of the date $T0$ that is the deadline of previous function, the TCTB function is executed to copy the data to buffer. Its WCET $W0$ is used to define its deadline $T1$ ($T1 = T0 + W0$).

On the non-time-critical level, $T1$ is the deadline for the previous functions. It should be noted that this deadline is not a hard one: late events are tolerated.

Phase II: The NTCFB function of the non-time-critical level transfers the data from the buffer at date $T1$. Then the NTCCP function deals with the data and generates the output results. Finally the NTCTB function transfers the data to the buffer. The sum of WCETs $Y0 + Y1 + Y2$ of NTCFB, NTCCP and NTCTB is used to define the date of $T2$. The temporal gap between $T1$ and $T2$ can be filled by other TCCP functions according to the task scheduling strategy. The WCET value $W1$ should be less than $T2 - T1$.

Phase III: The TCFB function copies the data from the buffer when the data is ready, if no timeout takes place. Otherwise the late event is handled appropriately by the second TCCP task. The maximal WCET value of LCFB and of the backup function defines the deadline date $T3$.

In fact, if missing a deadline on an *unpunctual* communication is harmless, a default value may be enough to react to the timeout event. Otherwise, a more evolved backup function is needed, and its WCET needs to be accounted for at design time, to set up the timeout accordingly. On the other hand, if a timeout should be considered as a fault, the approach is not applicable and the task should not be considered as non-time-critical; fail-safe or degraded mode transitions, or fault-tolerant approaches might be considered.

It should be noted that, not only a communication from non-time-critical to time-critical, but also time-unpredictable *communications* between two time-critical tasks or two parallel time-critical execution chains (operating in parallel on two different CPUs) may be *unpunctual*. This feature makes the protocol suitable for network-on-chip (NoC) systems, such as the Kalray MPPA manycore processor [14]. On such a chip, time-critical tasks can executed in parallel in the different clusters, while implementing *unpunctual* communication between clusters to reduce overhead and certification cost.

V. Validation

As mentioned above, the PE application is coded according to its original functional specification. This specification, written in SysML, details functional and behavioral aspects, risk levels (vital or non-vital), the component architecture, communications and interfaces. We split the vital functions into time critical and non-time critical ones according to their timing requirements, but any atomic behavior is not broken down.

Although the PE application is a reasonably simple use case, it has more than 300 functional and behavioral requirements already: it is a representative industrial SIL (Safety Integrity Level) application. Many other application areas combine the needs for computational components in the loop of safety-critical control. From engine control to monitoring, control engineers need computational tasks in order to improve trajectories, leading to resource optimization, emission reductions, longer maintenance cycles, etc. Model-predictive control (MPC) is one of the best representatives of such computational control applications.[6] Its applicability to real-time systems has so far been limited by its time impredictability [25], making it a prime candidate for mixed-time-critical execution.

Fig. 3 presents a simplified dataflow graph of the PE application. The following safety requirements must be met by time-critical tasks:

[6] http://www.mpc.berkeley.edu/mpc-course-material

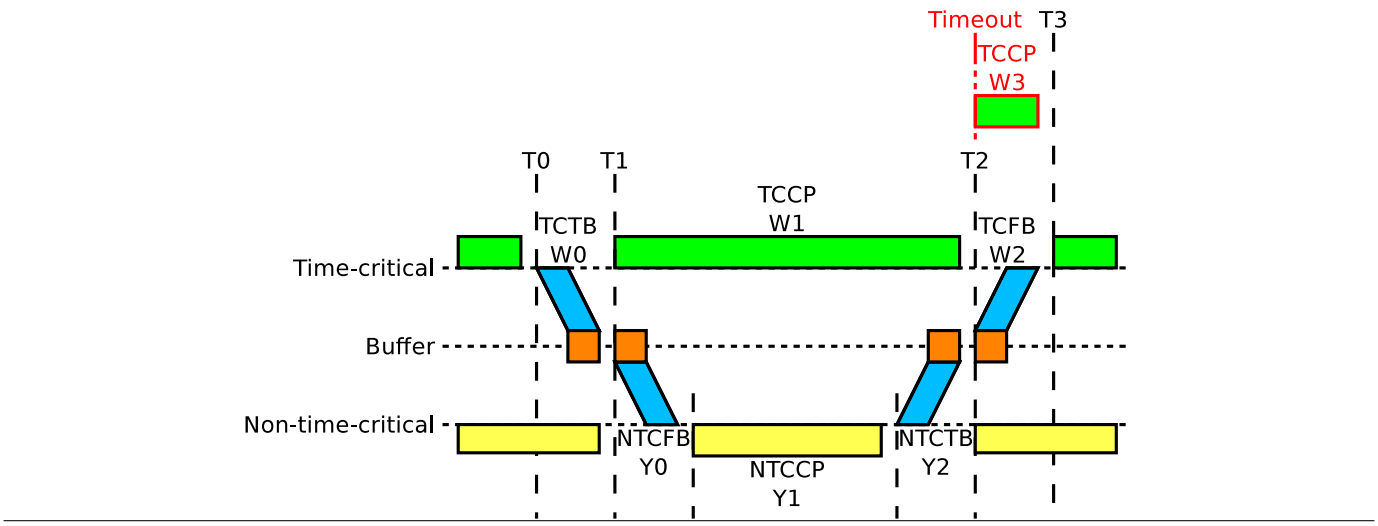

Figure 2. Chronogram of ideal time-triggered communication protocol.

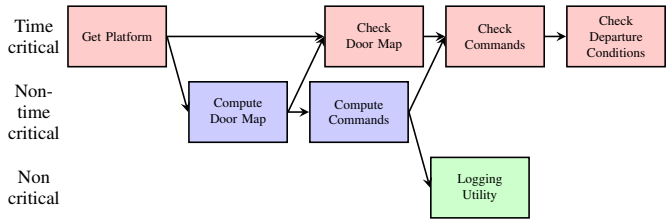

Figure 3. PE application modeling graph.

1) train/platform doors may only open if properly aligned;
2) if the train is not immobilized, doors cannot be opened;
3) doors must be closed to allow the train to depart.

The non-time-critical part sets three other requirements:

1) open/close commands are issued according to the mission;
2) inform passengers of an imminent opening/closing;
3) send warnings to the traffic supervision when the passenger exchange cannot be completed.

These last three requirements are handled by two non-time-critical functions. The first one computes a mapping from train doors to platform doors and vice versa, such that matching doors are physically aligned. The second one uses this mapping to compute commands to be sent to the doors according to the current mission.

To meet the safety requirements, these commands must be checked. If a command breaks one of the requirements, it is canceled: it is always safe, according to these requirements to send no commands. This check relies on three time-critical functions. First, the physical position of the platform doors is retrieved. Then, the door mapping computed by the corresponding mission-critical function is checked using these physical positions and the current train position. Finally, the commands are checked against this mapping and information about the train docking state. The last requirement is ensured by a fourth function which issues a departure authorization when no commands have been sent for the last few seconds.

The application also has a non-critical logging function which records the commands history.

We show in Listing 1 the HEPTAGON source code, declaring two *unpunctual* variables, `door map` and `door commands`. The generated C code is presented in Listing 2. Two functions are generated, for the time-critical code and one for the non-time-critical code, respectively. The communication functions `send` and `receive` have the same parameters: an identifier for the channel buffer, the address and the size of the payload, and the time-criticality of the sender or receiver. The result of the `receive` function is a boolean which indicates whether the communication has been done on time or a timeout has occurred. This boolean can only be false when communicating from a non-time-critcical task to a time-critical one. The communication mechanism is illustrated on a custom system configuration and platform in the next subsection.

A. Hardware/software implementation

We selected an off-the-shelf hardware platform, the Zynq 7K SoC ZedBoard [24] as our experimental target. It provides a pair of ARM cores and a FPGA. This flexible platform is very popular in mixed critical execution environments and in hardware-software codesign. In this paper, we leverage the ARM cores for their flexibility and performance on typical software stacks, and rely on the FPGA only for configuring the local memory and bus interfaces, and for implementing communication buffers.

As mentioned in Section IV-C, our first goal is to realize a strict spatial partitioning on the Zynq 7K. This is achieved through asymmetric multi-processing (AMP). The Zynq 7K permits at least two kinds of AMP configurations [22], between two ARM cores and between ARM and FPGA based soft-cores. We selected the ARM core-only AMP configuration to suit our performance-driven implementation strategy:

- ARM core 1 executes the time-critical tasks in a bare-metal environment. The software stack is allocated on

Listing 1. Snippet of the HEPTAGON implementation of the PE.

```
node passenger_exchange(train_position : int)
   returns (safe_door_commands : command^n; departure_authorization : bool)
var
  platform : int;
  unpunctual door_map : int^n;
  safe_door_map : int^n;
  unpunctual door_commands : command^n;
let
  platform = get_platform(train_position);
  door_map = compute_door_map(platform);
  safe_door_map = check_door_map(door_map, platform);
  door_commands = compute_commands(door_map);
  safe_door_commands = check_commands(door_commands, safe_door_map);
  departure_authorization = check_departure_conditions(safe_door_commands);
tel
```

Listing 2. C code generated by the HEPTAGON compiler.

```
void passenger_exchange_tc(int train_position, command safe_door_commands[8],
      bool* departure_authorization) {
  int platform, door_map[8], safe_door_map[8];
  command door_commands[8];
  bool ontime1, ontime2;

  get_platform(train_position, &platform);
  send(0, &platform, sizeof(int), TC);
  ontime1 = receive(1, door_map, sizeof(int) * 8, TC);
  check_door_map(ontime1, door_map, safe_door_map);
  ontime2 = receive(2, door_commands, sizeof(command) * 8, TC);
  check_commands(ontime2, door_commands, safe_door_map, safe_door_commands);
  check_departure_conditions(safe_door_commands, departure_authorization);
}

void passenger_exchange_ntc() {
  int platform, door_map[8];
  command door_commands[8];

  receive(0, &platform, sizeof(int), NTC);
  compute_door_map(platform, door_map);
  send(1, door_map, sizeof(int) * 8, NTC);
  compute_commands(door_map, door_commands);
  send(2, door_commands, sizeof(command) * 8, NTC);
}
```

the on chip memory (OCM) of 256KB (code and data).

- ARM core 0 executes the non-time-critical and non-critical tasks in a Linux environment: Petalinux [20]. The software stack is allocated on the DDR (512MB).
- The communication buffers are implemented on the FPGA.

Figure 4 details the hardware IPs used in the Vivado tool chain [21] for our system configuration.

a) Processing_system7_0: is the IP used to configure a couple of ARM codes. We use almost all default configuration values. It should be noticed that, CPU0 uses both L1 and L2 caches, but CPU1 uses only the L1 cache, in order to avoid concurrent access on the shared L2. The communication buffers on the FPGA are not cached.

b) Proc_sys_reset_0: is the IP used to reset FPGA components controlling by the Processing System.

c) Axi_mem_intercon: is the interconnection IP used to connect the AXI master components with the AXI slave ones. This is a $1 \rightarrow 3$ crossbar as there are one master (processing element) and three slave components: memory block, ZedBoard LEDs and switch GPIO. The latter two interfaces are used during PE application timeout injection and functional test.

d) MEM: is a block memory wrapper containing two IPs, one for the AXI BRAM controller and the other one for a block memory generator. In our implementation, we realize a 32KB memory buffer.

The resource utilization metrics are presented in the table below. Only a small fraction of the FPGA is used.

On the ARM Core1, tasks are time triggered relying on a snoop control unit (SCU) 32 bits local timer. We experimented with two ways to configure the timer: a fixed frequency static value and a dynamic value derived from to executing task's WCET. Both of them were tested successfully.

As the ZedBoard Zynq 7K SoC can operate at 666 MHz, we can set up a static timer reaching a relatively high frequency such as 100 KHz, 500 KHz or even 1 MHz, to accomodate multiperiodic schedules and a wide diversity of task WCET. A "jiffies" counter is used to hold the number of time ticks that have occurred since a task was initiated. At the end of the task,

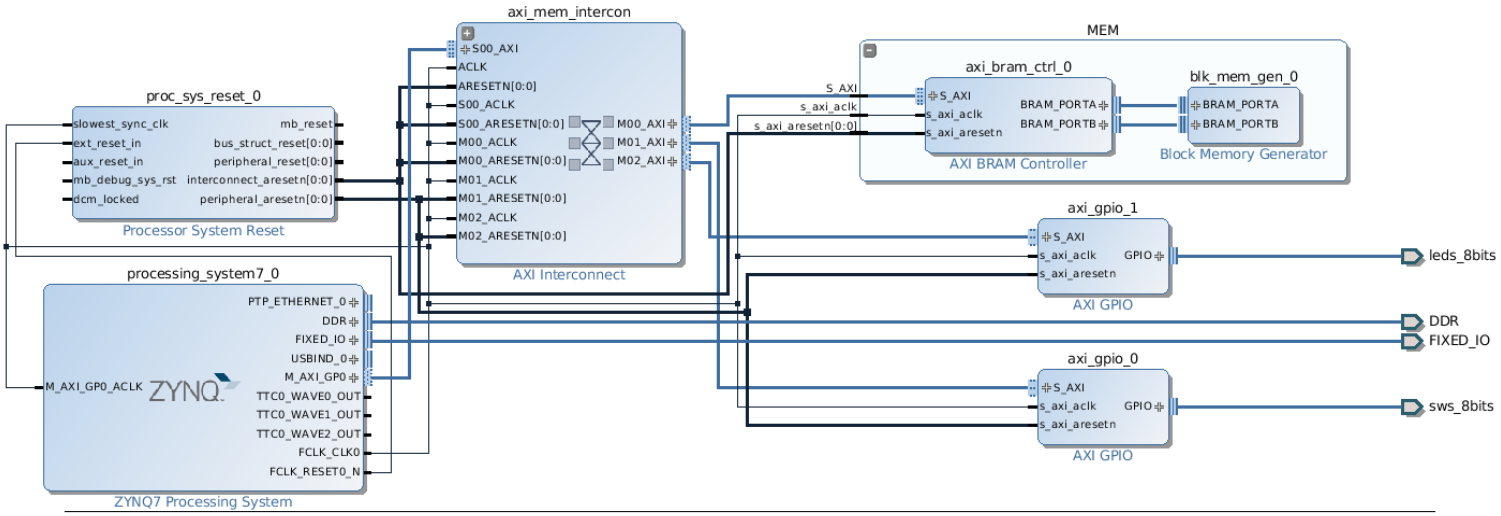

Figure 4. Hardware IPs (Vivado tool chain) used in our implementation.

Resource	Used	Available	Util%
Slice LUTs	2345	53200	4.40%
Slice Registers	2815	106400	2.64%
Block RAM Tile	8	140	5.71%

Figure 5. Mainly resource utilization ratio.

a comparison between the expected WCET and the measured jiffies allows to determine if the deadline was reached.[7] If it did, the next task executes and the jiffies counter is reset to zero. Otherwise, a loop spins until the expected deadline. When using a dynamic timer, it is defined according to the WCET of the operating task, with a "deadline" variable set to detect to the WCET. The rest proceeds identically as with a static counter.

The dynamic scheme requires slightly more code generation effort but it saves CPU resources, saving the need to handle intermediate interrupts. For example in the PE application, the task's largest WCET is 120× the shortest WCET, which means at least 120 timer interrupts can be saved using a dynamic timer.

On the ARM Core0, we did not yet realize time-triggered execution on Linux, but enter a spin loop instead to check if the data is ready. To avoid interferences on the communication buffer due to round robin arbitration, we have to time-isolate the non-time-critical side of the communication as well, which doubles the WCET of the corresponding communication function. This overhead is very lightweight as the communication function has a low cost compared with other components.

The chronogram of the execution is illustrated in Fig. 6;[8] the digits correspond to the following tasks: (1) Get Platform, (2) Compute Door Map, (3) Check Door Map, (4) Compute Commands, (5) Check Commands, (6) Check Departure Conditions, (7) Logging Utility.

As we just presented, the time-critical communication function A runs in parallel with the spin loop, effectively wasting parallel computing resources for a short time period. On the other hand, for most of the execution, functions (3) and (4) run in parallel, showing the benefit of our mixed time-critical design. More complex applications and scalable platforms would make even more effective use of the approach.

Listing 3. C structure of for unpunctual communication.

```
typedef struct communication
{
  volatile int *ready;

  // For unpunctual communication
  volatile int *timeout;

  volatile int *cycle_count;

  volatile void *payload;

  int size;
} communication_t;
```

Listing 3 details the language C structure dedicated to unpunctual communications. This structure contains five fields, *ready* signals that a communication packet is ready, *timeout* informs about missing packets when reaching a timeout, *cycle_count* stores the packet's logical instance cycle (a.k.a. period), *payload* wraps the packet data and *size* represents the size of the payload. We use the "C" and "D" communications of Figure 6 to explain this C implementation.

As shown in the Figure 7, we declare a sender variable C (type of communication_t) on the non-time-critical side, and a receiver variable D (type of communication_t) on the time-critical side.

e) The ready *fields:* of C and D point to a 32 bits physical address allocated on the FPGA Mem Buffer—0x40000000.

[7]We do not attempt to deal with missed deadlines on time-critical tasks, which should be handled as critical faults at another level (e.g., a degraded mode of operation).

[8]The lengths are not on (timing) scale.

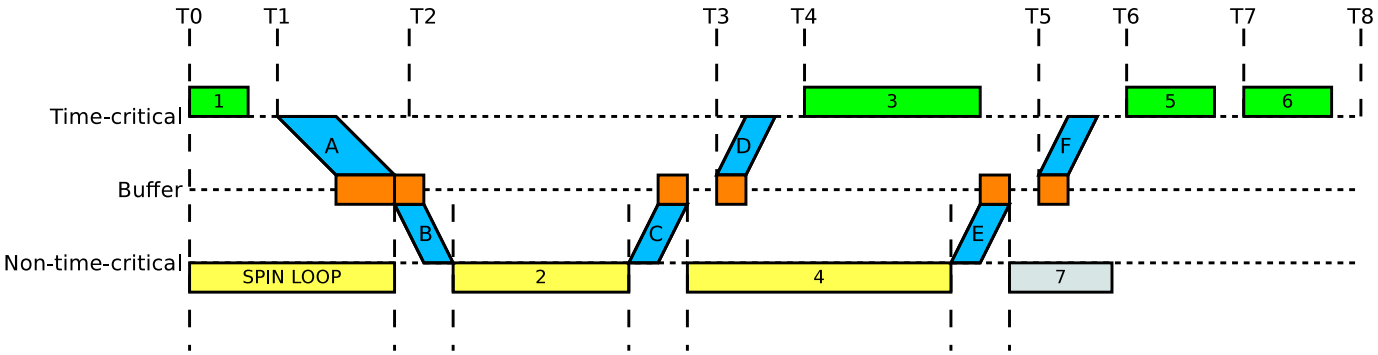

Figure 6. Execution chronogram of PE application.

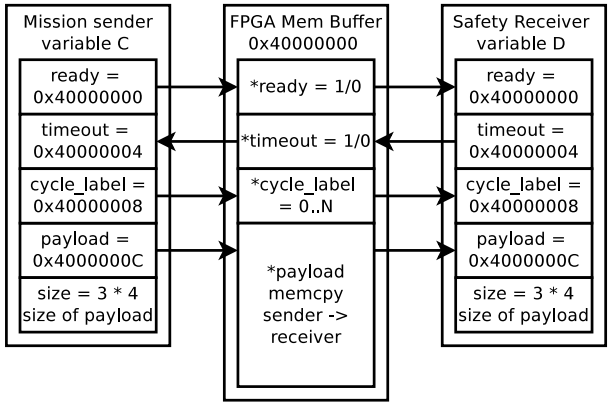

Figure 7. Implementation of the "C" and "D" communication of Figure 6.

The sender variable C produces the *ready* signal, it enables *ready* once the packet *payload* has been copied to the buffer, i.e., when the packet is ready for the receiver. The "ontime" operator is implemented by checking the *ready* field at the receiver side when the deadline date is reached. If *ready* is not enabled, a timeout occurs. The receiver should execute a backup function.

f) The timeout *fields:* of C and D point to the following 32 bits physical address—0x40000004. The receiver variable D produces the *timeout* signal, it enables *timeout* when the deadline date T3 is reached and packet *payload* is not yet ready. That is to say, when the packet timeout occurs. The sender variable C is the consumer; when it seens an enabled *timeout*, it aborts the current packet transmission.

g) The cycle_count *fields:* of C and D point to the next 32 bits physical address—0x40000008. The sender variable C is the producer, the cycle count of the non-time-critical task is sent to the receiver. If it is different from the receivers', a full-cycle delay accumulation occured. That means the sender and the receiver have lost their synchronous association, which is a more severe situation, but still one that the protocol aims to tolerate. It may happen it the non-time-critical task is seriously delayed due to a chaotic accumulation of interferences. In such a case, the simplest approach would be to skip the upcoming instances of the non-time-critical task(s) until the synchrony of the cycle counts can be restored. Optimized methods to handle such severe and accumulated delays are outside the scope of this paper, but the reader interested in advanced strategies may refer to [10].

h) The payload *fields:* of C and D point to the following 32 bits physical address—0x4000000C. To preserve timing isolation, the packet's contents is first copied from the sender to the buffer, then from the buffer to the receiver.

The memory overhead of our time-triggered communication protocol are is limited to the additional communication variables (cycle count, timeout and ready). On the PE application, only 6 such variables are needed to implement the (two) cross-time-criticality communications.

B. Early experiments

Our first experimental validation was to test the mixed-time-critical version of PE application with its original functional self-test, which takes the form of a set of scenario simulations. This test was passed successfully, which proved that the time-triggered protocol does not change the PE function.

Next, we replayed the functional test with an additional timeout injection hazard. We did not change the test scenarios, but inserted controllable delays in the functions (2) Compute Door Map and (4) Compute Commands. Practically, these delays took the form of simple device I/O on the Zync platform, checking if the corresponding ZedBoard switch is on. if yes, a random number of loop iterations were executed in order to simulate a non-predictable delay. If this induces a timeout, the backup function or default values are used, as implemented in the mixed-time-critical synchronous program. The trace of the PE application proved the absence of timing violations or incorrect commands on the time-critical tasks.

As noticed earlier, when a huge delay affects the function (2) or (4), delay accumulation may occur and the *cycle_count* fields of the sender and receiver may differ. Our current implementation simply aborts the PE application by an exception for now, as an illustration of the self-diagnosis potential of the communication protocol. Of course, a complete implementation should skip some upcoming tasks and resynchronize accordingly instead.

VI. Conclusion

We presented an application where mixed criticality resides at the application level, or even at function level, rather than the system level. Moreover, all functions remain safety critical, and the different criticalities we consider are focused on timing predictability and requirements instead. Different time predictability requirements allowed us to expose parallelism and optimize resource utilization, compared to a much more conservative timing isolation of all components. We demonstrate the feasibility of *hard real-time and parallel execution of safety-critical tasks on a conventional embedded multicore platform*. A timing isolation protocol allows best-effort, functionally validated tasks to communicate with hard real-time ones, while (1) *preserving timing compositionality*, and (2) *satisfying all hard real-time constraints* in the complete application. To program and certify such applications, we proposed an extension to a *synchronous language enabling correct-by-construction code generation and parallel execution* on a multiprocessor platform. Based on this experience, we advocate for a multidimensional approach to mixed criticality where *timing constraints are managed separately from other system validation aspects*. We also advocate for a holistic approach, where the design flow of complex control applications takes into account different levels of timing predictability: our proposal is one step towards the construction of such a flow. We encourage control engineers to detail the different admissible modes, *limiting the extent of hard real-time components, and defining the hard real-time reactions to the late availability of non-time-critical data*.

References

[1] F. Borrelli, A. Bemporad, and M. Morari. *Predictive control for linear and hybrid systems*. To appear, 2016.

[2] A. Burns and R. Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, 2013.

[3] T. Carle, D. Potop-Butucaru, Y. Sorel, and D. Lesens. From dataflow specification to multiprocessor partitioned time-triggered real-time implementation. Research Report RR-8109, INRIA, Oct. 2012.

[4] D. Claraz, F. Grimal, T. Leydier, and R. Mader. Introducing multi-core at automotive engine systems. In *ERTS*^{2}, Feb. 2014.

[5] A. Cohen, L. Gérard, and M. Pouzet. Programming parallelism with futures in lustre. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 197–206, New York, NY, USA, 2012. ACM.

[6] A. Cohen, V. Perrelle, D. Potop-Butucaru, E. Soubiran, and Z. Zhang. Mixed-criticality in railway systems: A case study on signalling application. In *Workshop on Mixed Criticality for Industrial Systems (WMCIS'2014)*, 2014.

[7] D. Dasari, B. Akesson, V. Nelis, M. A. Awan, and S. M. Petters. Identifying the sources of unpredictability in cots-based multicore systems. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, pages 39–48. IEEE, 2013.

[8] B. D. de Dinechin, D. van Amstel, M. Poulhiè, and G. Lager. Time-critical computing on a single-chip massively parallel processor. In *DATE invited paper for special session SD1 on Predictable Multi-Core Computing*, pages 24–28, Dresden, Germany, Mar. 2014.

[9] G. Delaval, A. Girault, and M. Pouzet. A type system for the automatic distribution of higher-order synchronous dataflow programs. *SIGPLAN Not.*, 43(7):101–110, June 2008.

[10] J. P. Erickson, N. Kim, and J. H. Anderson. Recovering from overload in multicore mixed-criticality systems. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 775–785, 2015.

[11] A. Esper, G. Nelissen, V. Nélis, and E. Tovar. How realistic is the mixed-criticality real-time system model? In *RNTS*, pages 155–164, Lille, France, Nov. 2015.

[12] G. Fernandez, J. Abella, E. Quiñones, C. Rochange, T. Vardanega, and F. J. Cazorla. Contention in Multicore Hardware Shared Resources: Understanding of the State of the Art. In H. Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 31–42, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[13] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet. A Modular Memory Optimization for Synchronous Data-Flow Languages. Application to Arrays in a Lustre Compiler. In *Languages, Compilers and Tools for Embedded Systems (LCTES'12)*, Beijing, June 12-13 2012. ACM.

[14] Kalray. Kalray MPPA®-256 integrated manycore processor.

[15] C. M. Kirsch and A. Sokolova. *Advances in Real-Time Systems*, chapter The Logical Execution Time Paradigm, pages 103–120. 2011.

[16] A. Pyka, M. Rohde, and S. Uhrig. A real-time capable coherent data cache for multicores. *Concurrency and Computation: Practice and Experience*, 26(6):1342–1354, 2014.

[17] Y. Wang and S. Boyd. Fast model predictive control using online optimization. *IEEE Transactions on Control Systems Technology*, 18(2):267–278, 2010.

[18] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Making shared caches more predictable on multicore platforms. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 157–167. IEEE, 2013.

[19] R. Wilhelm and J. Reineke. Embedded systems: many cores – many problems. *SIES*, 12:176–180, 2012.

[20] Xilinx. PetaLinux tools.

[21] Xilinx. Vivado design suite.

[22] Xilinx. Xilinx Zynq multi-os support (AMP & hypervisor).

[23] Xilinx. Zynq-7000 all programmable SoC.

[24] ZedBoard.org. ZedBoard.

[25] M. N. Zeilinger, D. M. Raimondo, A. Domahidi, M. Morari, and C. N. Jones. On real-time robust model predictive control. *Automatica*, 50(3):683 – 694, 2014.