Multicore Scheduling for Large Safety-Critical Applications: Beyond Single-Core Equivalence

Peter Csaba Ölveczky[1] and José Meseguer[2]

[1] University of Oslo, Oslo, Norway
[2] University of Illinois, Urbana-Champaign, USA

Abstract. The *single-core equivalence* (SCE) technology provides the *robust partitioning* of multicore processors needed for estimating worst-case executing times (WCETs) and for FAA certification, which requires that change or failure of lower-criticality software does not affect higher-criticality software. However, SCE only allows running an application in a single core, and therefore cannot deal with emerging large safety-critical applications that do not fit in a single core. In this paper we introduce the *physically multicore, logically single-core* (PMLS) technology that extends SCE to such large applications—where different tasks may share variables and therefore have critical sections—which cannot be executed in a single core. The idea is to use well-known single-core methods, such as the single-core priority ceiling protocol (PCP), for task synchronization. We study schedulability properties of PMLS, formalize PMLS executions in Maude, and use model checking to find a task allocation that makes the large application schedulable (if one exists).

1 Introduction

Multicore processors are now ubiquitous and provide significantly improved *average* performance for *general-purpose* applications. In this paper we target *safety-critical* applications where tasks *must* meet hard real-time deadlines. In particular, we present a real-time scheduling solution for large avionics applications that must use more than one core and must meet the multicore partitioning requirements of the U.S. Federal Aviation Administration (FAA) [2, 23].

However, timing analysis in multicore systems is made hard, if not impossible, by the various forms of *interference* between *independent* applications that are executed in different cores. Such inter-core interferences include sharing DRAM, memory bus bandwidth, last-level caches, I/O channels, and so on [25]. Experiments by Lockheed Martin on its Space Systems Testbed using an 8-core chip (Freescale P4080) showed that a task's worst-case execution time (WCET) can increase by as much as 600% when it runs concurrently with logically independent tasks in other cores [25]. Using a multicore chip *as is*, the inter-core interferences therefore essentially turns a task's WCET into a random variable; furthermore, any schedulability analysis of safety-critical tasks in one core can be invalidated by workload changes in *other* cores.

Robust partitioning of cores makes possible schedulability analysis and certification of each core *independently* of the workloads in other cores. FAA allowed developers of safety-critical tasks to use only *one* core per multicore chip until all interference channels between cores were identified, constrained, and accounted for in timing analysis and testing. Robust partitioning is required for certification by the FAA and the European Union Aviation Safety Agency (EASA) [3]. Furthermore, DO-178C certification for software-based aerospace systems [1] was developed for single-core chips and has yet to address inter-core interference.

The *single-core equivalence* (SCE) architecture [25] is a package of technologies jointly designed by academia and industry to provide robust partitioning. SCE bounds the inter-core interferences and *allows timing analysis* and certification of software in a core independently of workloads in other cores.

In SCE, each core executes an independent application (set of tasks). However, in the era of big data and UAVs, applications are becoming ever larger. Some large applications—such as advanced airborne monitoring and navigation applications—may no longer fit in one core, and must therefore be executed on multiple cores. For example, areas near airports will be increasingly crowded with UAVs, which airplanes must take into account during take-off and landing.

This paper presents a scheduling methodology for such large avionics applications that must use more than one core and must meet FAA's robust partitioning requirements [2,23]. More generally, we develop a methodology that extends SCE to make it possible to execute large safety-critical applications that cannot fit in one core (and therefore are not schedulable using SCE), but must meet strict timing requirements. Such large applications have tasks that access shared variables, and whose critical sections access both shared variables and "private" variables only accessible by a single task.

We propose and formalize a methodology called *physically multicore, logically single-core* (PMLS), which extends SCE to large applications, using well-understood *single-core* scheduling and task synchronization, and which uses formal model checking methods to automatically synthesize an allocation of tasks and task fragments onto the different cores so that desired properties are met.

For large applications where all tasks sharing some resource do *not* fit in a single core, we must execute (parts of) some of these tasks in different cores. This can lead to different cores accessing some shared or private variables, as well as having to move the execution of a task across cores. We therefore propose in Section 6 an architecture which extends SCE to this more general setting, without jeopardizing SCE's timing guarantees.

We introduce the PMLS methodology in Section 5. The key idea is to allocate tasks, and even different segments of a task, onto different cores, so that the well-understood *single-core* priority ceiling protocol (PCP) [24] is used for task synchronization and scheduling. Furthermore, multicore cache coherence problems are avoided, since a variable accessible in multiple cores is only accessed by one task, and all variables shared across tasks are located in a single core.

There is a vast body of work on multicore scheduling. However, to the best of our knowledge, only SCE-based approaches are designed to support the FAA

robust partitioning requirement for multicore scheduling. The FAA's robust partitioning requirement aims to enable the reuse of the well-established single-chip software timing certification process; this is also exactly the goal of PMLS. Our contribution appears to be the first to extend the SCE methodology to large safety-critical applications that cannot fit into a single core.

Section 7 studies schedulability and task allocation properties of PMLS:

- Compares different "task splitting policies" for allocating the execution of a "synchronization task" onto two cores.
- Demonstrates the need to consider also non-worst-case execution times of different task segments in the schedulability analysis. Unlike in the single-core case, it is not sufficient to consider only worst-case execution times.
- Shows that "each task makes its first deadline" does not imply schedulability.

Having to combine different task splitting policies for optimal schedulability makes it hard to find an allocation of task fragments on a given number of cores so that the large application does not miss any deadline. Furthermore, having to consider also non-maximal execution times and having to go beyond checking whether each task makes its first deadline to ensure schedulability make it hard to decide whether a given PMLS task allocation can lead to missed deadlines. In Section 8 we therefore formalize PMLS executions in rewriting logic [15] using the Maude [16] language. We can then use Maude model checking to check whether a given task allocation can lead to a missed deadline. We use this model checker (and Maude's meta-programming capabilities) to automatically synthesize a PMLS task allocation so that no deadlines are missed.[3]

Finally, Section 2 discusses related work, and Section 3 provides background on PCP and Maude. Section 4 presents some assumptions about the tasks and the execution environment, and Section 9 gives some concluding remarks.

2 Related Work

The work most closely related to ours is the work on SCE, which partitions the resources shared by cores in such a way that each core can be used as if it were a single-core chip [25]. This is a set of co-designed protocols that include work on DRAM management [30], memory bus bandwidth management [31], I/O in Integrated Modular Avionics [11], real-time cache management [13], and estimating worst-case WCET in a multicore chip [14]. As SCE, our work supports certification of avionics applications; but it generalizes SCE by dropping the requirement that applications must fit in a single core (or even the requirement that all tasks sharing a variable must fit into a single core). Nevertheless, PMLS can be implemented on multicore chips that satisfy SCE requirements and support core-switching and lockdown of variables in the last-level cache.

Among literature on multicore scheduling and task synchronization, Anderson et al. investigate a cache-aware Pfair-based scheduling scheme for real-time

[3] At the moment, our model checker only takes one task splitting policy into account; the general case will be supported in future work.

tasks on multicore platforms [4], Rajkumar developed the first multiprocessor synchronization protocol [22], and Brandenburg et al. compare the multiprocessor real-time synchronization protocols M-PCP, D-PCP, and FMLP [6]. None of these papers address the certification challenges tackled in this paper.

Sun et al. [27] also target schedulability on multicore processors. They partition the cache, allocate tasks to cores, and perform schedulability analysis. In contrast to our contribution, they do not consider shared resources among tasks, and therefore avoid task migration (whereas in our setting a task may have to execute its non-critical-section code in one core, and its critical-section code in another core). Sun et al. therefore do not need to extend resource-sharing protocols, such as PCP, to the multicore setting, which is a problem we address in this paper. Furthermore, in the absence of task migration, they can perform schedulability analysis based on WCETs, whereas we show in Section 7 that deadlines can be missed when some job fragments execute for *less time* than their WCETs, even if deadlines are *not* missed when all job fragments execute for their entire WCET. Having to consider non-WCET executions is why we use model checking for schedulability analysis in Section 8.

Xu et al. [29] develop heuristics for allocating *independent* tasks onto cores and partitioning cache and memory bandwidth to achieve schedulability on a multicore processor. Since the tasks do not access shared variables, our contribution contrasts to Xu et al. [29] as to the work by Sun et al. [27].

While synchronization cores have been used in multicore scheduling [10, 21, 28], they do not use single-core methods, but instead use multicore versions such as M-PCP and MSRP, to schedule a big application that does not fit in a single core.

Hang, Manolios, and Papavasileiou [9] use integer linear programming modulo theories (IMT) to synthesize architectural models for cyber-physical systems. Part of this work involves mapping tasks to different processors so that the tasks are schedulable. Hang et al. consider simpler static non-preemptive cyclic scheduling. In contrast, we consider—and synthesize schedulable task allocations for—preemptive *multicore* scheduling, and show in this paper that static schedulability checking in this setting seems to be a very hard problem. Hang et al. combine an integer linear programming solver with specialized decision procedures for scheduling. The interesting aspect is that the model checker presented in this paper could serve exactly as such a decision procedure for scheduling large applications on multiple cores. This opens up the intriguing possibility of combining the methods of Hang et al. with our model checker and our approach to multicore task synchronization to synthesize the architecture of large safety-critical CPSs on multicore architectures.

Finally, on the rewriting logic side, the Real-Time Maude language and tool [19] has been used to model and analyze *single-core* scheduling algorithms [17] and the *single-core* priority inheritance protocol [20]. In contrast, our paper targets the *multicore* setting, with task migration and task splitting, and with the need to consider also non-WCET execution times in the schedulability analysis.

3 Preliminaries

Single-Core Scheduling. We use generalized rate-monotonic scheduling [26] (with preemption), where tasks with shorter periods have higher priorities. In standard rate-monotonic scheduling it is easy to check schedulability of a task set using Liu and Layland's critical instant theorem [12]: assuming that all tasks start their periods at time 0, it is sufficient to check that (the first job of) each task makes its first deadline. Furthermore, it is sufficient to analyze the case when each job executes for as long as possible (its WCET). However, in our setting, different fragments of the same task can execute in two cores, leading to something akin to "self-suspending" tasks, which are hard to analyze efficiently [7].

When multiple tasks access shared resources, they must do so in critical sections. This could, however, lead to (possibly unbounded) priority inversion: a high-priority task that wants to access its critical section could be blocked by a low-priority task executing inside the critical section (which in turn could be preempted by a medium-priority task executing outside the critical section, and so on). The *priority ceiling protocol* (PCP) [24] is a well-known real-time synchronization protocol that ensures that tasks are deadlock-free and that a higher-priority task can be blocked at most once by a simple or nested critical section of a lower-priority task. That is, a higher-priority task can be blocked for at most the largest WCET of a critical section of a lower-priority task.

Rewriting Logic and Maude. Rewriting logic [15] is a computational logic for specifying concurrent systems as *rewrite theories* of the form $(\Sigma, E \cup A, R)$, where, (i) Σ is a *signature* of typed *function symbols* with types and subtypes (called sorts and subsorts), (ii) E a set of (possibly conditional) equations, (iii) A is a set of equational axioms such as associativity and commutativity, so that equational deduction is performed *modulo* the axioms A, and (iv) R is a set of *labeled conditional rewrite rules* $[l] : t \longrightarrow t'$ **if** *cond*. $(\Sigma, E \cup A, R)$ specifies a concurrent system whose states belong to the algebraic data type defined by $(\Sigma, E \cup A)$ and whose local transitions are defined by R.

Real-time systems can be defined in rewriting logic as real-time rewrite theories [18], where ordinary rewrite rules model *instantaneous* change, and time advance is modeled by "tick" rewrite rules $[tick] : \texttt{\{}t\texttt{\}} \longrightarrow \texttt{\{}u\texttt{\} in time } \tau$ **if** *cond*, where τ is a term of sort `Time`, t and u are terms of sort `System`, and where the *entire* state always has the form `{`s`}`.

As shown in [5], real-time rewrite theories can be specified and formally analyzed using the rewriting-logic-based Maude tool [8]. A function f is declared `op` f `:` $s_1 \ldots s_n$ `->` s. Equations and rewrite rules are introduced with, respectively, keywords `eq`, or `ceq` for conditional equations, and `rl` and `crl`.

`class` C `|` $att_1 : s_1$`,` $\ldots$`,` $att_n : s_n$ declares a *class* C of objects with attributes att_1 to att_n of sorts s_1 to s_n. An *object* o of class C is represented as a term `<` $o : C \mid att_1 : val_1, \ldots, att_n : val_n$ `>`, where o is the object's *identifier*, and val_1 to val_n are the values of the attributes att_1 to att_n. A system state is modeled as a term of sort `Configuration`, and is a *multiset* of objects and messages.

Maude provides several analysis methods, including rewriting, reachability analysis, and linear temporal logic (LTL) model checking, all of which can be made time-bounded [5]. The command `red` *expr* reduces the expression *expr* to its normal form using the equations E. Given a state pattern *pattern* and an (optional) condition *cond*, the following `search` command searches for a state reachable from *init* that matches *pattern* and satisfies *cond*:

`search [1]` *init* `=>*` *pattern* [`such that` *cond*] `.`

Maude supports *meta-programming*, where a Maude module ("program") M (and a term t, etc.) can be (meta-)represented as a Maude *term* $\overline{M}$ of sort `Module` (resp, a term $\overline{t}$ of sort `Term`, etc.). We can then define Maude functions on such (meta-)terms, to analyze and manipulate Maude specifications. Maude provides (meta-level) functions mirroring its analysis commands. For example, `metaSearch(`$\overline{M}$`,` $\overline{\textit{init}}$`,` $\overline{\textit{pattern}}$`,` $\overline{\textit{cond}}$`, '*, unbounded, 0)` returns the (meta-representation of the) solution to the above search command in the module M, and returns `failure` if such a state cannot be reached.

4 Assumptions and Notation

Since aperiodic events can be handled by periodic servers, we only consider periodic tasks. A task T_i sharing variables with other tasks is called a *synchronization task*; other tasks are called *independent tasks*. Each job of a synchronization task T_i executes three disjoint code segments $s_{i,1}$, $css_{i,2}$, and $s_{i,3}$, where $s_{i,1}$ and $s_{i,3}$ denote non-critical-section code segments and $css_{i,2}$ a critical section code segment. The (single) code segment of an independent task T_j is denoted $s_{j,1}$.

We build on *single-core equivalent* (SCE) technology [25]. In SCE, the worst-case execution time of a task (segment) s depends on how many cores are used. $WCET(k, s)$ denotes the worst-case execution time of s when k cores are used. For a given multi-core architecture, $WCET(k, s)$ can be measured or computed based on $WCET(1, s)$ [14]. We typically assume a given bound b on the number of cores to be used, and just write $WCET(s)$ for $WCET(b, s)$.

For the chosen b, we denote by $e_{i,1}$, $e_{i,3}$, and $cs_{i,2}$, respectively, $WCET(b, s_{i,1})$, $WCET(b, s_{i,3})$, and $WCET(b, css_{i,2})$. The period of a task T_i is denoted P_i. The deadline of each task is the end of its period.

Our model differentiates between the different shared variables. However, for simplicity we assume that there are no *nested* critical sections; this assumption holds in most real-time applications. We use the single-core priority ceiling protocol [24] for task synchronization. All tasks in a task set $\mathcal{T} = T_1, \ldots, T_n$ are indexed: a lower index implies a higher priority.

Notation: We usually write e_i for an independent task T_i, and $e_{j,1}$; $cs_{j,2}$; $e_{j,3}$ for a synchronization task T_j. Furthermore, when the shared variable is given or understood, we often write $\langle T_j : e_{j,1} - cs_{j,2} - e_{j,3};\ p_j \rangle$ for the synchronization task T_j with period p_j, and write $\langle T_i : e_{i,1};\ p_i \rangle$ for the independent task T_i with maximal execution time $e_{i,1}$ and period p_i.

For example, $\langle T_1 : 5 - 2 - 3;\ 15\rangle$ denotes a *synchronization task* T_1, where the WCET (relative to the assumed number of cores to use) of its code segment *before* the critical section is 5, the WCET of its critical section code segment is 2, the WCET of its code after the critical section is 3, and its period is 15. Likewise, $\langle T_2 : 8;\ 16\rangle$ denotes an *independent task* T_2 whose WCET is 8 and whose period is 16. The task T_1 has higher priority than T_2, since it has lower index (which follows from its shorter period in rate-monotonic scheduling).

5 Overview of PMLS Scheduling of Large Applications

The *physically multi-core, logically single-core* (PMLS) methodology extends single-core equivalence [25] to large applications whose tasks do not fit into one core. PMLS provides conditions for allocating such a large application task set, satisfying the assumptions in Section 4, onto the cores of a multicore chip so that single-core PCP is used for task synchronization.

Our approach to allocate and execute large applications using SCE techniques and single-core PCP can be summarized as follows:

- One core, called the *synchronization core*, or *syn_core*, executes the critical section code of all tasks. The entire code segment of some synchronization tasks and/or independent tasks may also be allocated to *syn_core*, as long as *syn_core* remains schedulable with single-core PCP.
- Other cores, called *execution cores*, $exec\_core_i$, for $i \in \{1, \ldots, b-1\}$, execute independent tasks (that do not fit in *syn_core*).
- If this does not make the application schedulable, we can also move fragments of synchronization tasks to execution cores, *as long the critical section fragment of each synchronization task is executed in the synchronization core.* Assuming in this paper that a synchronization task is executed in at most two cores, there are different choices (or "policies") for *which* fragments of a synchronization task to execute in the different cores:
  1. Maybe the most natural choice is to let the synchronization task T_i (that does not fit entirely in the *syn_core*) execute in an execution core, *except* that its critical section fragment $cs_{i,2}$ is executed in the *syn_core*.
  2. Only the *tail* segment $e_{i,3}$ is executed in an execution core, whereas the head $e_{i,1}$ and the critical section $cs_{i,2}$ both are executed in *syn_core*.
  3. Only the *head* fragment $e_{i,1}$ is executed in an execution core, whereas the critical section and the tail both are executed in *syn_core*.

  Section 7 shows that these *task splitting policies* are incomparable in strength.
- All variables shared across tasks are stored in *syn_core*. So are the private variables of all synchronization tasks which fully fit into the *syn_core*, and the private variables of the independent tasks allocated to *syn_core*. Otherwise, we distinguish between:
  - *private variables used across cores*, which appear *both* in the critical section code (executed in *syn_core*) and in the non-critical-section code of a task allocated to an execution core; and

- *single-core private variables*, private variables not appearing in critical section code (if any).

Any task allocation algorithm satisfying the above conditions ensures that all jobs have unique access to their critical section.

Private variables used across cores are accessed from both the *syn_core* and an execution core. They are stored in the last-level cache, as explained in Section 6. Nevertheless, multicore cache coherence is not needed since:

- Variables accessed "concurrently" by multiple tasks reside in *syn_core*.
- A private variable shared across cores is only accessed by a single task.

6 The PMLS Architecture

Single-core equivalence (SCE) provides robust partitioning of cores, so that each core can be used as if it were a single-core chip [25]. In particular, if we declare that we want to use k of the m cores on a chip, each core gets allocated $\frac{1}{k}$ of the available resources, including the last-level cache (which is shared among all cores). SCE allows us to estimate WCETs carefully, and ensures that the WCET is independent of applications running on the other cores.

The PMLS architecture introduced in this section reuses SCE technology and implementations to support the execution of large applications on multicore chips. If all synchronization tasks of our large application fit in one core, we can use SCE as is. Otherwise, we must extend SCE for two reasons:

1. Core switching of tasks, where the execution of a single (synchronization) task switches between two cores, is needed in PMLS for multicore tasks. (Fragments of) such tasks are executed in execution cores, but their critical sections are executed in *syn_core*.
2. The private variables of a multicore task which appear in both the critical-section code and in the non-critical-section code must be accessible from both cores executing the task.

This section explains how the PMLS architecture addresses these two issues without jeopardizing the good properties of SCE, including providing reliable WCETs and avoiding cache coherence problems.

To avoid discussing many similar cases, we assume in this section that we use the *first* task splitting policy in Section 5; i.e., both fragments $e_{i,1}$ and $e_{i,3}$ of a multicore task T_i are executed in an execution core. For a task T_j we then denote by $core(T_j)$ the core in which its non-critical-section fragments are executed.

6.1 Core Switching

SCE does not need core switching, since SCE assumes that all tasks of an application fit in a single core. PMLS requires core switching when executing multicore tasks. This can be done by operating system calls. However, the extra time required by core switching must be accounted for in timing analyses.

If δ is the (usually very short) time needed to switch cores executing a task, then in the worst-case scenario, when k cores are used by the application, a task may have to wait for time $k \cdot \delta$ to switch cores. This is because of the extremely unlikely event that all cores have tasks that want to migrate at the exact same time; these system calls must then be buffered until the migration takes place.

The PMLS solution to ensure that timing analyses are correct is to add time $k \cdot \delta$ to the WCET of both $e_{i,1}$ and $e_{i,3}$, for each synchronization task T_i. In the schedulability discussions in this paper, we assume that this has been done.

The addition to SCE should not affect the isolation from other applications running in other cores, which never access the cores used by our large application.

6.2 Cross-Core Accessible Variables

We have different "types" of variables in a PMLS execution:

1. Variables accessed by multiple tasks are only accessed inside critical sections.
2. Private variables, only accessed by a single task, of *single-core tasks*.
3. Private variables of *multicore tasks*, which can appear:
   (a) both inside and outside the task's critical section;
   (b) only outside the task's critical section; or
   (c) only inside the task's critical section.

We call variables of type (3-a) *cross-core accessible* variables.

Under SCE partitioning, PMLS allocates all shared variables (type 1) and type (3-c) variables to syn_core, and allocates any variable of types (2) and (3-b) of a task T_i to the core $core(T_i)$. The new challenge is the cross-core accessible variables. The PMLS solution is to allocate a partition of the last-level cache for the exclusive purpose of storing all cross-core accessible variables of all tasks, and to lock them down there already in the initialization phase. Although multiple cores will access these variables, multicore cache coherence problems do not occur in the last-level cache, since only *one* task will access each such variable.

Since we allocate cross-core accessible variables to an entirely new partition of the last-level cache during initialization, and lock them down there so that they are never removed from that cache, the rest of the execution can be done entirely in an SCE chip as is. The only requirement is support for the lockdown in the cache of such variables (see [13]).

Reserving a partition of the last-level cache to the cross-core accessible variables can be done by "using" $k + 1$ cores for the large application allocated to k cores, and storing all cross-core accessible variables in the last-level cache "part" of the extra core.[4] Alternatively, we may not reserve an entire core—which will be idling—just to use its partition of the last-level cache. If l cores are used for all applications, we can instead partition all shared resources into $l+1$ parts, and use the "extra" partition of the last-level cache for cross-core accessible variables.

[4] If there are too many cross-core accessible variables for one partition of the last-level cache, we can use multiple new partitions.

Concerning WCET estimations and their accuracy, after the initialization phase, the execution should be faster, since the cross-core accessible variables are always available in the last-level cache, and therefore do not have to be read from, or written to, DRAM. Furthermore, there are no operations extraneous to SCE after initialization. The only "change" is that an instruction to read a cross-core accessible variable from cache finds it in the new partition of the cache.

Note, however, that in both variations, if a total of l cores are used in the chip for application executions, then the WCETs should be based on $WCET(l+1)$. Since all operations after the initialization are "standard" SCE operations, robust partitioning still holds, so that independent applications running on other partitions have no effects on our task's execution times.

To summarize, if an implementation of SCE supports lockdown of variables in a chosen partition of the last-level cache and its RTOS supports core switching, then such an SCE implementation can be used for PMLS executions.

7 Task Splitting Policies and Schedulability

This section studies schedulability properties of PMLS.

We start by studying the three different task splitting policies mentioned in Section 5 for dividing the execution of a synchronization task onto the synchronization core and an execution core. A natural question to ask is whether one policy is better than another. We show in Section 7.1 that the three policies are incomparable in strength: For each policy, there are task sets that are only schedulable with that policy, and not with the other two.

In Section 7.2 we show, for each task splitting policy, that it is *not* sufficient to consider only worst-case execution times when analyzing schedulability. Some task sets that are schedulable when each segment executes as long as possible can miss their deadlines when either the first non-critical-section fragment or the critical section fragment executes for less time than its WCET.

To check schedulability of a task set under PMLS, we must therefore consider all possible combinations of policies for splitting the execution of a multicore synchronization task, as well as all possible execution times for all job fragments. This motivates our model-checking-based task allocation algorithm in Section 8.

"Checking schedulability" in similar systems often amounts to using the critical instant theorem (i.e., all tasks start at the same time) and checking whether each task makes its first deadline. Experimenting with our model checker, we found that this is *not* the case for PMLS, as we explain in Section 7.3. This emphasizes the need for model-checking-based schedulability analysis until analytic schedulability analysis methods are developed for PMLS.

7.1 Comparing Task Splitting Policies

In PMLS a synchronization task executes in at most two cores, where the critical section fragment is executed in the *synchronization core*. If the execution of a synchronization task T_i must be split over two cores, we can think of, as

mentioned in Section 5, at least three different *policies* for executing its non-critical-section fragments $e_{i,1}$ and $e_{i,3}$:

(1) Both non-critical segments $e_{i,1}$ and $e_{i,3}$ are executed in some execution core, and the critical section segment $cs_{i,2}$ is executed in the synchronization core.
(2) Only the tail fragment $e_{i,3}$ is executed in the execution core, whereas both $e_{i,1}$ and $cs_{i,2}$ are executed in the synchronization core.
(3) Only the head fragment $e_{i,1}$ is executed in an execution core, whereas both $cs_{i,2}$ and $e_{i,3}$ are executed in the synchronization core.

The following examples show that these three task splitting policies are incomparable in strength: sometimes only policy (1) makes the system schedulable, sometimes only policy (2), and some other times only policy (3).

The following example shows a task set where moving *both* non-critical fragments of a task makes the system schedulable, whereas neither of the other two policies work; task is, only policy (1) makes the task set schedulable:

Example 1. Consider a task set with two tasks $\langle T_1 : 0 - 8 - 3;\ 20\rangle$ and $\langle T_2 : 3 - 10 - 0;\ 20\rangle$. This task set is schedulable when we execute both the tail of T_1 and the head of T_2 in the execution core. However, if either one of these non-critical-section fragments are executed in the synchronization core, then the processor utilization of the synchronization core becomes $\frac{3+8+10}{20} > 1$. □

The following example shows that sometimes only policy (2) makes a task set schedulable:

Example 2. Consider the tasks $\langle T_1 : 2-2-1;\ 5\rangle$, $\langle T_2 : 4;\ 5\rangle$, and $\langle T_3 : 0-1-0;\ 5\rangle$ to be executed on two cores. This task set is schedulable with policy (2): the tail of T_1 and entire T_2 are executed on the execution core; the rest on the synchronization core. Executing entire T_1 in the synchronization core does not work, since the critical section of T_3 must also be executed there, leading to utilization $\frac{6}{5}$ of that core. Moving both head and tail of T_1 (policy (1)) to the execution core leads to processor utilization $\frac{7}{5}$ of that core, and moving only the head of T_1 to the execution core (policy (3)) leads to utilization $\frac{6}{5}$. □

Finally, we can "mirror" the task set in Example 2 to show a task set where *only* policy (3) makes the task set schedulable:

Example 3. The task set $\langle T_1 : 1 - 2 - 2;\ 5\rangle$, $\langle T_2 : 4;\ 5\rangle$, and $\langle T_3 : 0 - 1 - 0;\ 5\rangle$ is only schedulable on two cores using policy (3): the execution core executes the first part (1) of T_1 and entire T_2. This makes the system schedulable, and an argument similar to the one in Example 2 shows that no other policy works. □

7.2 Accounting for Non-WCET Executions

In single-core schedulability analysis it is enough to consider the case when all task executions take the maximum possible time (WCET). However, in our

multicore setting the worst situation(s) may happen when (multicore) tasks do *not* execute their segments for the longest possible time.

The following five examples show that taking *only* WCETs into account can make a multicore task set (appear) "schedulable," even when a deadline *can* be missed if some segment executes for *less* than its WCET.

Examples 4 and 5 show this when the critical-section code executes for less than its WCET, for different task splitting policies:

Example 4. Consider tasks $\langle T_1 : 1 - 1 - 1;\ 4\rangle$ and $\langle T_2 : 3;\ 6\rangle$, both executing in *exec_core₁* under task splitting policy (1).

If all code segments of T_1 take the worst-case time to execute, namely, 1, then we have the following scenario: T_1 executes in *exec_core₁* in the time interval $[0, 1]$, executes in *syn_core* in time $[1, 2]$, executes in *exec_core₁* in time $[2, 3]$, idles in time $[3, 4]$, executes in *exec_core₁* in time $[4, 5]$, and executes in *syn_core* in time $[5, 6]$. Task T_2 can then execute in *exec_core₁* in the time intervals $[1, 2]$, $[3, 4]$, and $[5, 6]$, and therefore makes its first deadline. The task set seems schedulable according to such WCET analysis.

However, if the *second* job of T_1 does *not* execute its critical section, then T_1 executes in *exec_core₁* at times $[0, 1]$, $[2, 3]$, $[4, 5]$, and $[5, 6]$. That leaves only the time intervals $[1, 2]$ and $[3, 4]$ in *exec_core₁* for T_2 before time 6; T_2 therefore misses its first deadline. □

We can slightly modify this example to show that we have the same "problem" also for task splitting policy (2):

Example 5. Consider the tasks $\langle T_1 : 0 - 1 - 2;\ 4\rangle$ and $\langle T_2 : 3;\ 6\rangle$, both of which execute in *exec_core₁* under task splitting policy (2). If all job segments execute maximally, then T_2 can execute in *exec_core₁* during the times $[0, 1]$ and $[3, 5]$, making its first deadline. If the critical section code of the *second* job of T_1 is not executed, then T_1 executes in the execution core at times $[1, 3]$ and $[4, 6]$, so that T_2 can only execute during $[0, 1]$ and $[3, 4]$, missing its deadline 6. □

Examples 4 and 5 show that for schedulability analysis, we must also consider cases where the execution time of the *critical section code* in some instances is smaller than its WCET, for both task migration policies (1) and (2). This seems harder to show for policy (3), since the fragment after the shortened fragment also executes in *syn_core*. The following examples do the same thing, for all three task splitting policies, for the (first) *non-critical-section code fragment*:

Example 6. We have two synchronization tasks $\langle T_1 : 1 - 2 - 0;\ 4\rangle$ and $\langle T_2 : 1 - 1 - 1;\ 6\rangle$, where T_1 is executes *exec_core₁*, under task splitting policy (1), and T_2 executes entirely in *syn_core*.

If all segments execute as long as possible, then we have the following scenario: T_1 executes in *exec_core₁* in time $[0, 1]$, executes in *syn_core* in time $[1, 3]$, idles in time $[3, 4]$, executes in *exec_core₁* in time $[4, 5]$, and executes in *syn_core* in time $[5, 7]$. T_2 can therefore execute (in *syn_core*) at times $[0, 1]$, $[3, 4]$, and $[4, 5]$, thereby making its first deadline. The task set is therefore schedulable when all segments execute as long as possible.

If the *second* job of T_1 skips $e_{1,1}$, then T_1 executes in *syn_core* at times $[1, 3]$ and $[4, 6]$. T_2 can then only execute at times $[0, 1]$ and $[3, 4]$ before time 6, and therefore cannot finish its first job before its first deadline. □

We "mirror" Example 6 to obtain the same result for task splitting policy (2):

Example 7. Let both tasks[5] $\langle T_1 : (1 - \delta) - \delta - 2;\ 4\rangle$ and $\langle T_2 : 3;\ 6\rangle$ execute in *exec_core₁* under task splitting policy (2). With maximal execution times, T_1 executes in *syn_core* at times $[0, 1]$ and $[4, 5]$, and in *exec_core₁* at times $[1, 3]$ and $[5, 7]$, so that T_2 can execute in *exec_core₁* at times $[0, 1]$ and $[3, 5]$, making its first deadline. However, again, if the first segment $e_{1,1}$ of the *second* job of T_1 is not executed, then T_1 would execute in *exec_core₁* at times $[1, 3]$ and $[4 + \delta, 6 + \delta]$, so that T_2 misses its first deadline. □

We modify Example 7 to obtain the result for task splitting policy (3):

Example 8. Consider the tasks $\langle T_1 : 1 - \delta - (2 - \delta);\ 4\rangle$ and $\langle T_2 : 1 - 1 - 1;\ 6\rangle$, with T_1 is assigned to *exec_core₁*, under task splitting policy (3), and T_2 is fully executing in *syn_core*. Like in the other examples, with maximal executions, the tasks are schedulable with this allocation; however, if T_1 skips the first segment of its second instance, then T_2 misses its first deadline. □

7.3 Critical Instant Theorem and First-job Completion

Experimentation with our model checker, described in Section 8, revealed that schedulability analysis based on the critical instant theorem and checking whether the first job of each task makes its deadline is *not sufficient* to ensure that PMLS executions do not miss hard deadlines, as the following example shows:

Example 9. Let $\langle T_1 : 0 - 2 - 0;\ 4\rangle$ execute in *syn_core* and let $\langle T_2 : 2 - 2 - 1;\ 5\rangle$ execute in *exec_core₁* under task splitting policy (1). It is easy that see that both tasks make their first deadline when starting at time 0. However, our model checker revealed the following missed deadline: T_1 idles in its first three periods (any job will do; the model checker outputs the shortest path to the missed deadline), and T_2 idles in its first two periods. At time 10, T_2 starts a new round with job $2 - 1 - 1$. At time 12, T_2 has remaining job $1 - 1$ and remaining time 3 until its deadline. At time 12, T_1 starts its next round, with job $0 - 2 - 0$. That is, at time 12, both T_1 and T_2 want to access critical section. The higher-priority task T_1 executes in *syn_core* in time $[12, 14]$, blocking T_2 from executing its critical section until time 14, so that T_2 misses its third deadline at time 15. □

In other words, this example shows that if T_1 starts at time 2 and T_2 starts at time 0, then T_2 misses its first deadline, while this is not the case if both start at time 0, invalidating the critical instant theorem for such systems.

The question is what this means for the results in Sections 7.1 and 7.2, where we write "is schedulable" based on "make first deadline" reasoning. The results in

[5] We add a very small value δ so that T_1 actually is a synchronization task.

Section 7.1 should hold, since all periods are the same. Furthermore, our model checker has confirmed the "is schedulable" results in Examples 1, 4, 5, and 6.[6]

8 Synthesizing Schedulable Task Allocations

We have specified in Maude an executable *formal* model of all possible PMLS executions of a given task set and a given PMLS task allocation, for two reasons:

1. Reachability analysis in our model allows us to automatically check whether or not a given task allocation can lead to missed deadlines.
2. We use this model checker to find a task allocation onto the available cores that makes the large application schedulable, if such an allocation exists.

At the moment our model checker *only* considers task splitting policy (1). Since we show in Section 7 that sometimes other such policies may make a task set schedulable, we should extend our model checker to consider all possible task allocations with all possible combinations of task splitting policies.[7]

8.1 Formalizing PMLS Executions

We formalize executions of a set of tasks, allocated to a set of cores, and executing under task splitting policy (1), in Maude. Since Section 7 shows that deadlines can be missed when some job fragment(s) execute for shorter times than their WCETs (or not at all), when a new period begins, the job to be executed by the task in this period is selected *nondeterministically*, and can be *any* job where each job fragment executes for its WCET, or less, or not at all. In this way we cover *all possible behaviors* of the system (under task splitting policy (1)).

We model a synchronization task t as an object

```
< t : SynTask | job : job,          period : p,          priority : pri,
                leftOfPeriod : τ,   leftOfJob : remJob,  state : s,
                homeCore : c,       currentPriority : currPri >
```

where *job* denotes its "maximal" job as a term of the form

`exec(`e_1`) ; enterCS(`x`) ; exec(`cs`) ; exitCS(`x`) ; exec(`e_3`).`

p is the period of the task, *pri* is its (static) priority, and c is the core to which the task is allocated; these values do not change during executions. The dynamic values are: *currPri*, denoting the task's current dynamic priority; τ is the time until the task's next deadline; *remJob* denotes what "job fragments" remain to be executed in this period; and the task's execution state s is either `executing`, `idle`, `waiting`, or `blocked(`t'`)`.

An independent task is represented as an object of class `IndTask` with the same attributes, except for `homeCore`.

The synchronization core is modeled as an object

[6] At the moment, our model checker only considers task splitting policy (1), but allows to also model check when all jobs are maximal.

[7] All formal models and analysis commands are available at http://olveczky.se/PMLS

`< synCore : SynCore | tasks :` *tasks*, `vars :` *shVarData* `>`

The attribute `tasks` contains the tasks currently "residing" in the synchronization core. `vars` contains, for each shared resource, its status (`blocked(`t`)` or `free`), and the highest priority of the tasks that will access the resource. Execution cores are modeled as objects of the class `ExecCore` that do not have the `vars` attribute.

Example 10. The initial state of the system in Example 4 is

```
op example26-4 : -> GlobalSystem .
eq example26-4 =
{< synCore : SynCore | sharedVars : var(x, free, 1),  tasks : none >
 < execCore(2) : ExecCore | tasks :
   < t1 : SynTask |
           job : (exec(1) ; enterCS(x) ; exec(1) ; exitCS(x) ; exec(1)),
           period : 4, priority : 1, currentPriority : 1, leftOfPeriod : 0,
           leftOfJob : nil,  state : idle,  homeCore : execCore(2) >
   < t2 : IndTask | job : exec(3), period : 6, priority : 2,
           leftOfPeriod : 0, currentPriority : 2, leftOfPeriod : 0,
           leftOfJob : nil, state : idle > >} .
```

Both tasks are just finishing a period: they have zero time remaining of the period and nothing left to execute of the "previous" job. The nondeterministically chosen jobs for the next period will soon be in their `leftOfJob` attribute. □

The behaviors are formalized by 28 rewrite rules, the first of which is[8]

```
crl [newPeriod-A1] :
    < C : Core | tasks : (OTHER-TASKS
      < T : Task | job : JOB,  period : PERIOD, currentPriority : PRI1,
                   leftOfPeriod : 0, leftOfJob : nil, state : idle >
      < T1 : Task | state : executing,  leftOfJob : exec(NZT) ; JOB1,
                    currentPriority : PRI2 >) >
   =>
    < C : Core | tasks : (OTHER-TASKS
      < T : Task | leftOfPeriod : PERIOD, leftOfJob : exec(NZT2) ; REST-JOB,
                   state : if PRI1 < PRI2 then executing else waiting fi >
      < T1 : Task | state : if PRI1 < PRI2 then waiting else executing fi >)
    >
    if chooseJob(JOB) => exec(NZT2) ; REST-JOB .
```

In this rule, a core `C` "contains" tasks `T` and `T1`, and possibly other tasks (captured by the variable `OTHER-TASKS`). The class `Task` is a superclass of `SynTask` and `IndTask`, so these tasks could be synchronization tasks and/or independent tasks. `Core` is also a superclass, so this core could be either a synchronization core or an execution core. In the left-hand of the rule, the task `T` has finished its current period (`leftOfPeriod : 0`) and has also finished executing its job

[8] We do not show (mathematical) variable declarations in Maude, but follow the convention that the Maude variables are written in all capital letters.

(`leftOfJob : nil`). The *other* task `T1` is currently `executing`. This rewrite rule starts a new period of `T`: its relative deadline is reset to its period (`leftOfPeriod : PERIOD`). `T` then *nondeterministically* selects its next job; in this case it will start with executing outside the critical section (`chooseJob(JOB) => exec(NZT2) ; REST-JOB`). If `T`'s priority is higher than the currently executing task (`PRI1 < PRI2`), then `T` starts `executing`, otherwise `T` will be `waiting`. `chooseJob(`*job*`)` nondeterministically rewrites to any job that is "smaller or equal" to *job*.

Our model contains an equation

```
op deadlineMiss_left_timeLeft_ : TaskId Job Time -> Configuration .
ceq {OTHER-CORES
     < C : Core | tasks : < T : Task | leftOfPeriod : TIME, leftOfJob : JOB >
                          OTHER-TASKS >}
   = {deadlineMiss T left JOB timeLeft TIME} if duration(JOB) > TIME .
```

that transforms a state in which some task `T` in some core `C` has less time remaining until its deadline (`TIME`) than the duration `duration(JOB)` of the job that remains in this round, into a term `{deadlineMiss T left JOB timeLeft TIME}`.

8.2 Analyzing Schedulability

We can analyze whether a given task allocation is schedulable by searching for a missed deadline, i.e., a "state" `{deadlineMiss` t `left` j `timeLeft` $t2$`}`:

Example 11. We check whether a missed deadline can be reached in Example 4 using the following command:

```
Maude> search [1] example26-4 =>* {deadlineMiss T left JOB timeLeft TIME} .

Solution 1 (state 170)   rewrites: 13364 in 3ms cpu (3ms real)
T --> t2       JOB --> exec(1)       TIME --> 0
```

The output shows a reachable missed deadline: task `t2` still has to execute `exec(1)`, but has no time left in its current period. (As expected, no missed deadline was found in this example when all jobs executed "maximally.") □

Example 12. The initial state `example26-1` of the system in Example 1 is

```
{< syn-core : SynCore | sharedVars : var(x, free, 1), tasks : none >
 < exe-core : ExecCore | tasks :
  < t1 : SynTask | job : (exec(0) ; enterCS(x) ; exec(8) ; exitCS(x) ; exec(3)),
                   period : 20, priority : 1, ... >
  < t2 : SynTask | job : (exec(3) ; enterCS(x) ; exec(10) ; exitCS(x) ; exec(0)),
                   period : 20, priority : 2, ... > >}
```

As expected, no missed deadline can be reached from this state:

```
Maude> search [1] example26-1 =>* {deadlineMiss T left JOB timeLeft TIME} .

No solution.     states: 11018  rewrites: 1086084 in 295ms cpu
```

8.3 Synthesizing Schedulable Task Allocations

If there exists a PMLS allocation of tasks onto the user-defined number of cores (currently only with task splitting policy (1)) that makes a large application schedulable, we can find such a schedulable task allocation by:

1. generating all possible PMLS task allocations, and
2. for each such task allocation, perform the above search command to check whether the task allocation is schedulable.

We have implemented this process in Maude, and used Maude's meta-programming facilities to automate the last step.

We define a sort `SetOfGlobalSystems` for `;;;`-separated *sets* of terms of sort `GlobalSystem` (which are our states). The function `generateAllAllocs` takes as arguments the tasks to be allocated and the available cores, and generates all possible task allocations.

The function `findAllocation` is the main function that finds a schedulable PMLS task allocation if one exists. It takes as argument a task set and a set of cores, uses `generateAllAllocs` to generate all possible task allocations, and uses the above reachability analysis for each such task allocation. If, for some task allocation, the `search` does *not* find a missed deadline, then that allocation is returned. This function is defined in Maude as follows:

```
op findAllocation : Configuration GlobalSystem -> GlobalSystem .
op findAllocation : SetOfGlobalSystems -> GlobalSystem .

eq findAllocation(TASKS, CORES)
 = findAllocation(generateAllAllocs(TASKS, CORES)) .

eq findAllocation(ALLOC ;;; ALLOCS)
 = if metaSearch(['TEST-EXEC-PCP], upTerm(ALLOC),
       ''{_'}['deadlineMiss_left_timeLeft_['T:TaskId,'JOB:Job,'TIME:Time]],
                 nil, '*, unbounded, 0) == failure
   then ALLOC else findAllocation(ALLOCS) fi .

eq findAllocation(empty) = {none} .  --- no good task allocation
```

The second equation iterates over all possible task allocations. If (meta-)search from the allocation `ALLOC` cannot find a missed deadline (i.e., returns `failure`), then the (initial state with) allocation `ALLOC` is returned. Otherwise, the function `findAllocation` recursively tries to find a good task allocation from the remaining allocations `ALLOCS`. If all such allocations have been exhausted without finding a good one, then the last equation returns the "empty" state `{none}`.

The first argument to `metaSearch` shows that we are executing the module `TEST-EXEC-PCP`, the second argument gives the meta-representation of `ALLOC`, and the third argument is the meta-representation of the search pattern.

Example 13. Let us check whether there is a schedulable task allocation of the two tasks $\langle T_1 : 1 - 2 - 0; 4\rangle$ and $\langle T_2 : 1 - 1 - 1; 6\rangle$ in Example 6 on two cores:

```
Maude> red findAllocation(
  < t1 : SynTask | job : (exec(1) ; enterCS(x) ; exec(2) ; exitCS(x) ; exec(0)),
                   period : 4, priority : 1, homeCore : noId, ... >
  < t2 : SynTask | job : exec(1) ; enterCS(x) ; exec(1) ; exitCS(x) ; exec(1),
     period : 6, priority : 2, ..., homeCore : noId >,
 {< syn-core : SynCore | sharedVars : var(x, free, 1), tasks : none >
  < exe-core : ExecCore | tasks : none >}) .

result GlobalSystem:
{< exe-core : ExecCore | tasks : < t1 : SynTask | ... >
                                 < t2 : SynTask | ... > >
 < syn-core : SynCore | tasks : none, ... >}
```

We see that the tasks are schedulable if both are allocated to `exe-core`. □

Example 14. As expected, there is no schedulable task allocation of the tasks $\langle T_1 : 2-2-1; 5\rangle$, $\langle T_2 : 3; 5\rangle$, and $\langle T_3 : 0-1-0; 5\rangle$ on two cores with the model checker's currently supported task splitting policy (1):

```
Maude> red findAllocation(
  < t1 : SynTask | job : (exec(2) ; enterCS(x) ; exec(2) ; exitCS(x) ; exec(1)),
                   period : 5, priority : 1, homeCore : noId, ... >
  < t2 : IndTask | job : exec(3), period : 5, priority : 2, ... >,
  < t3 : SynTask | job : exec(0) ; enterCS(x) ; exec(1) ; exitCS(x) ; exec(0),
                   ... >
 {< syn-core : SynCore | sharedVars : var(x, free, 1), tasks : none >
  < exe-core : ExecCore | tasks : none >}) .

result GlobalSystem: {none}
```

Example 15. The task set $\{\langle T_1 : 1-1-1; 3\rangle$, $\langle T_2 : 2; 3\rangle$, $\langle T_3 : 1-1-2; 9\rangle$, $\langle T_4 : 3; 9\rangle$, $\langle T_5 : 1-1-1; 11\rangle\}$ is schedulable on three cores: T_5 is allocated to the synchronization core, T_1 and T_3 are allocated to one execution core, and T_2 and T_4 are allocated to the other execution core. If the period of T_3 is 8, then there is no schedulable task allocation (with task migration policy (1)). □

The task allocation algorithm in the worst case checks all possible task allocations, and for each of them must cover all possible combination of jobs "smaller than or equal to" each task's WCET job. For up to four tasks, and with reasonable WCET values, this can be done almost instantaneously. However, Example 15, with five tasks and three cores, needed four minutes (on a MacBook Pro with M1 chip) to find a task allocation, and 28 minutes to find that no schedulable task allocation exists when the period of T_3 is reduced to 8.

9 Concluding Remarks

To the best of our knowledge PMLS is the first work that makes possible the certification of safety-critical avionics applications that do not fit in a single

core. This is achieved using single-core scheduling and synchronization methods. PMLS can be implemented on multicore chips satisfying SCE requirements that support core switching and lockdown of variables in the last-level cache.

We investigated some fundamental properties of PMLS executions and task allocations, which showed that: (i) shorter execution times can lead to missed deadlines not possible when each task fragment executes for its WCET; (ii) different "policies" of migrating parts of a task on two cores must sometimes be combined to achieve schedulability; and (iii) checking whether each task makes its first deadline is not sufficient for checking schedulability. These insights motivated a formal model checking approach to analyze schedulability of a given task allocation, and to automatically finding a task allocation that makes a large application schedulable on the available number of cores.

In future work we will extend our schedulability analysis and task allocation tool to all three task splitting policies, and extend PMLS to tasks with multiple and/or nested critical sections. Our model-checking-based approach may not scale to applications with many tasks or long execution times, although significant performance improvement can be achieved by *parallelizing* the process of finding a schedulable task allocation. Important future work is therefore to develop fast, but non-optimal, task allocation algorithms for PMLS.

Acknowledgments. We cordially thank Lui Sha for suggesting to us to work on the problem solved in this paper and for many discussions over the years about this problem. His ideas and insights have greatly helped us develop the formal model and solution presented here. We of course take full responsibility for any possible mistakes in the paper. We also thank the anonymous reviewers for useful comments on an earlier version of this paper. The first author was supported by NATO SPS program project G6133 ("SymSafe").

References

1. Discover DO-178C guidance. https://www.do178.org/
2. AC 20-193: Use of multi-core processors. Federal Aviation Administration, https://www.faa.gov/documentLibrary/media/Advisory_Circular/AC_20-193.pdf (2024)
3. AMC 20-193: Use of multi-core processors. pp. 661-675 of EASA's General Acceptable Means of Compliance for Airworthiness of Products, Parts and Appliances (AMC-20), https://www.easa.europa.eu/downloads/134971/en (2022)
4. Anderson, J.H., Calandrino, J.M., Devi, U.C.: Real-time scheduling on multicore platforms. In: 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006). pp. 179–190. IEEE Computer Society (2006)
5. Bae, K., Olarte, C., Ölveczky, P.C.: Modeling and analyzing real-time systems in rewriting logic. In: Meseguer, J., Varela, C.A., Venkatasubramanian, N. (eds.) Concurrent Programming, Open Systems and Formal Methods: Essays Dedicated to Gul Agha to Celebrate His Scientific Career. Lecture Notes in Computer Science, vol. 16120, pp. 494–535. Springer Nature Switzerland (2026)

6. Brandenburg, B.B., Anderson, J.H.: A comparison of the M-PCP, D-PCP, and FMLPon LITMUSRT. In: Principles of Distributed Systems (OPODIS 2008). LNCS, vol. 5401, pp. 105–124. Springer (2008)
7. Chen, J.J., Nelissen, G., Huang, W.H., Yang, M., Brandenburg, B., Bletsas, K., Liu, C., Richard, P., Ridouard, F., Audsley, N., Rajkumar, R., Niz, D., Brüggen, G.: Many suspensions, many problems: a review of self-suspending tasks in real-time systems. Real-Time Systems **55**(1), 144–207 (Jan 2019)
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude – A High-Performance Logical Framework, LNCS, vol. 4350. Springer (2007)
9. Hang, C., Manolios, P., Papavasileiou, V.: Synthesizing cyber-physical architectural models with real-time constraints. In: Proc. Computer Aided Verification (CAV'11). pp. 441–456. Lecture Notes in Computer Science, Springer (2011)
10. Kim, H., Wang, S., Rajkumar, R.: vMPCP: A synchronization framework for multi-core virtual machines. In: IEEE Real-Time Systems Symposium (RTSS 2014). pp. 86–95. IEEE Computer Society (2014)
11. Kim, J., Yoon, M., Bradford, R.M., Sha, L.: Integrated modular avionics (IMA) partition scheduling with conflict-free I/O for multicore avionics systems. In: IEEE 38th Annual Computer Software and Applications Conference, (COMPSAC 2014). pp. 321–331. IEEE Computer Society (2014)
12. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM **20**(1), 46–61 (1973)
13. Mancuso, R., Dudko, R., Betti, E., Cesati, M., Caccamo, M., Pellizzoni, R.: Real-time cache management framework for multi-core architectures. In: 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2013). pp. 45–54. IEEE Computer Society (2013)
14. Mancuso, R., Pellizzoni, R., Caccamo, M., Sha, L., Yun, H.: WCET(m) estimation in multi-core systems using single core equivalence. In: 27th Euromicro Conference on Real-Time Systems, ECRTS 2015. IEEE (2015)
15. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992)
16. Ölveczky, P.C.: Real-Time Maude and its applications. In: Rewriting Logic and Its Applications (WRLA 2014). LNCS, vol. 8663, pp. 42–79. Springer (2014)
17. Ölveczky, P.C., Caccamo, M.: Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In: Fundamental Approaches to Software Engineering (FASE 2006). LNCS, vol. 3922, pp. 357–372. Springer (2006)
18. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theor. Comput. Sci. **285**(2), 359–405 (2002)
19. Ölveczky, P.C., Meseguer, J.: The Real-Time Maude tool. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). LNCS, vol. 4963, pp. 332–336. Springer (2008)
20. Ölveczky, P.C., Prabhakar, P., Liu, X.: Formal modeling and analysis of real-time resource-sharing protocols in Real-Time Maude. In: 2008 IEEE International Symposium on Parallel and Distributed Processing. pp. 1–8 (2008)
21. Rahman, M.M.: Process synchronization in multiprocessor and multi-core processor. In: 2012 International Conference on Informatics, Electronics & Vision (ICIEV). pp. 554–559 (2012)
22. Rajkumar, R.: Real-time synchronization protocols for shared memory multiprocessors. In: 10th International Conference on Distributed Computing Systems. pp. 116–123 (1990)

23. Rushby, J.: Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Tech. Rep. DOT/FAA/AR-99/58 and NASA/CR-1999-209347, Office of Aviation Research, Washington, D.C., and NASA Langley Research Center, Hampton, VA (2000), https://www.tc.faa.gov/its/worldpac/techrpt/ar99-58.pdf
24. Sha, L., Rajkumar, R., Lehoczky, J.: Priority inheritance protocols: an approach to real-time synchronization. IEEE Transactions on Computers **39**(9), 1175–1185 (1990)
25. Sha, L., Caccamo, M., Mancuso, R., Kim, J., Yoon, M., Pellizzoni, R., Yun, H., Kegley, R., Perlman, D.R., Arundale, G., Bradford, R.M.: Real-time computing on multicore processors. IEEE Computer **49**(9), 69–77 (2016), longer version: https://hdl.handle.net/2142/55672
26. Sha, L., Rajkumar, R., Sathaye, S.S.: Generalized rate-monotonic scheduling theory: a framework for developing real-time systems. Proc. IEEE **82**(1), 68–82 (1994)
27. Sun, B., Roy, D., Kloda, T., Bastoni, A., Pellizzoni, R., Caccamo, M.: Co-optimizing cache partitioning and multi-core task scheduling: Exploit cache sensitivity or not? In: IEEE Real-Time Systems Symposium, RTSS 2023. pp. 224–236. IEEE (2023)
28. Wu, J., Hong, X.: Energy-efficient task scheduling and synchronization for multicore real-time systems. In: 2017 IEEE 3rd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing, (HPSC) and IEEE International Conference on Intelligent Data and Security (IDS), 2017. pp. 179–184. IEEE (2017)
29. Xu, M., Phan, L.T.X., Choi, H.Y., Lin, Y., Li, H., Lu, C., Lee, I.: Holistic resource allocation for multicore real-time systems. In: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 345–356 (2019)
30. Yun, H., Mancuso, R., Wu, Z.P., Pellizzoni, R.: PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In: 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2014). pp. 155–166. IEEE Computer Society (2014)
31. Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., Sha, L.: MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In: 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2013). pp. 55–64. IEEE Computer Society (2013)