Towards an OpenMP Specification for Critical Real-time Systems

Maria A. Serrano[1,2], Sara Royuela[1,2], and Eduardo Quiñones[1]

[1] Barcelona Supercomputing Center (BSC), Barcelona, Spain
{maria.serranogracia,sara.royuela,eduardo.quinones}@bsc.es
[2] Universitat Politecnica de Catalunya (UPC), Barcelona, Spain

Abstract. OpenMP is increasingly being considered as a convenient parallel programming model to cope with the performance requirements of critical real-time systems. Recent works demonstrate that OpenMP enables to derive guarantees on the functional and timing behavior of the system, a fundamental requirement of such systems. These works, however, focus only on the exploitation of fine grain parallelism and do not take into account the peculiarities of critical real-time systems, commonly composed of a set of concurrent functionalities. OpenMP allows exploiting the parallelism exposed within real-time tasks and among them. This paper analyzes the challenges of combining the concurrency model of real-time tasks with the parallel model of OpenMP. We demonstrate that OpenMP is suitable to develop advanced critical real-time systems by virtue of few changes on the specification, which allow the scheduling behavior desired (regarding execution priorities, preemption, migration and allocation strategies) in such systems.

1 Introduction

There is an increasing demand to introduce parallel execution in critical real-time systems to cope with the performance demands of the most advanced functionalities, e.g., autonomous driving and unmanned aerial vehicles. In this regard, OpenMP is a firm candidate [22,40] due to its capability to efficiently exploit highly parallel and heterogeneous embedded architectures, and its programmability and portability benefits. OpenMP is already supported in several embedded platforms for instance, the Texas Instruments Keystone II [37] or the Kalray MPPA [18]. Moreover, OpenMP is being evaluated to be supported in future versions of the Ada language [26], used to develop safety critical systems.

Current critical real-time systems are composed of a set of independent and recurrent pieces of work, known as *real-time tasks*, implementing the functionalities of the system. This model enables to exploit the inherent *concurrency* of the system when the number of available cores is low, as it is the case of the Infineon Aurix, a 32-bit micro-controller used in automotive that features six cores [5]. With the newest highly parallel embedded architectures targeting the critical real-time market, the number of available cores has increased significantly, enabling to exploit fine-grain parallelism within each real-time task as well. This is the case for instance, of the Kalray MPPA, featuring a fabric of 256 cores [18].

However, critical real-time systems must provide strong *safety* evidences on the *functional* and *timing* behavior of the system. In other words, the system

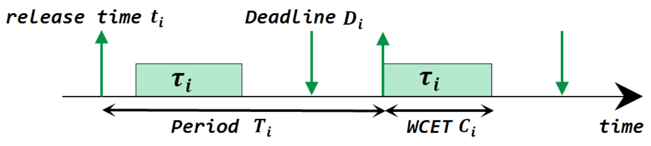


Fig. 1: Real-time task representation.

must guarantee that it operates correctly in response to its inputs, and that system operations are performed within a predefined time budget. A recent work evaluated the suitability of OpenMP from a functional perspective [29]. This paper complements that work and evaluates OpenMP from a timing behavior perspective. In this context, recent studies have shown the similarities between the structure and syntax of the OpenMP tasking model and the *Direct Acyclic Graph* (DAG) scheduling model [40], which enables to verify the timing constraints of parallel real-time tasks [13]. These similarities allow the analysis of the timing behavior of a single real-time task parallelized with OpenMP [35,38].

This paper extends previous works, and analyses the use of OpenMP (as it is in version 4.5 [3]) to implement critical real-time systems. We focus on the design implications and the scheduling decisions to efficiently exploit fine grain parallelism within real-time tasks and concurrency among them, while guaranteeing the timing behavior according to current real-time practices.

2 Critical Real-Time Systems

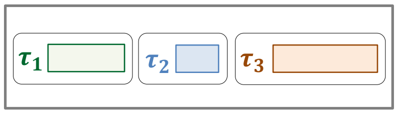
2.1 The Three-Parameter Sporadic Tasks Model

Critical real-time systems are represented as a set of recurrent and independent real-time tasks $\mathcal{T} = \{\tau_1, \tau_2, ...\tau_n\}$. Each execution of a real-time task is known as a *job*; the time at which a job is triggered is known as *release time* and it is denoted by t_i. A recurrent task can be *periodic*, if there is an exact time between two consecutive jobs, or *sporadic*, if there is a minimum time between jobs. In both cases, they can be triggered either by an internal clock or by the occurrence of an external event, e.g., a sensor.

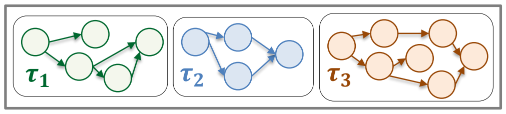
Traditionally, the *three-parameter sporadic tasks model* [25] is used to characterize critical real-time systems composed of sequential tasks that run concurrently on a platform. In this model, each task τ_i is represented with the tuple $\langle C_i, T_i, D_i \rangle$, where C_i is the Worst-Case Execution Time (WCET), i.e., an estimation of the longest possible execution time of τ_i; T_i is the period, or the minimum time between two consecutive jobs of τ_i; and D_i is the deadline at which τ_i must finish (see Figure 1). Critical real-time systems must guarantee that, for each task, its deadline is met, i.e., $\forall \tau_i \in \mathcal{T}$, $t_i + C_i \leq D_i$.

2.2 The Sporadic DAG Tasks Model

In the recent years, the complexity of real-time tasks have significantly increased to incorporate advanced functionalities, e.g., image recognition. With the objective of providing the level of performance needed, the code within each real-time



(a) 3-parameter sporadic tasks



(b) Sporadic DAG tasks

Fig. 2: Real-time system models.

task can be further parallelized. In this context, the use of the *sporadic DAG task model* [13] enables to characterize *parallel real-time tasks*[3] with the tuple $\tau_i = \langle G_i, T_i, D_i \rangle$. $G_i = (V_i, E_i)$ is a DAG representing the parallelism exposed within a real-time task. $V_i = \{v_{i,1}, \ldots, v_{i,n_i}\}$ denotes the set of nodes that can potentially be executed in parallel, where n_i is the number of nodes within τ_i. $E_i \subseteq V_i \times V_i$ denotes the set of edges between nodes, representing the precedence constraints existing between them: if $(v_{i,1}, v_{i,2}) \in E_i$, then $v_{i,1}$ must complete before $v_{i,2}$ begins its execution. In this model, each node $v_{i,j} \in V_i$ is characterized by its WCET, denoted by $C_{i,j}$. Finally, as in sequential real-time tasks, T_i and D_i represent the period and deadline of the parallel real-time task τ_i. This model is considered in the integrated modular avionics (IMA) [2] and the AUTOSAR [20] frameworks, used in avionics and automotive systems, respectively.

Figure 2 shows a taskset composed of three real-time tasks: in Figure 2a, tasks are modelled with the three parameter sporadic tasks model, i.e., real-time tasks are sequential and run concurrently; in Figure 2b, tasks are modelled with the sporadic DAG tasks model, i.e., real-time tasks have been parallelized, enabling to exploit both, concurrency and fine-grain parallelism.

2.3 Parallelizing a Single Real-time Task with OpenMP

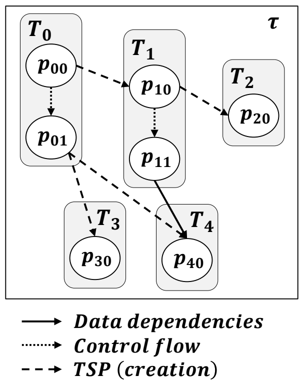
Several works demonstrate that the OpenMP tasking model resembles the sporadic DAG task scheduling model when considering a single real-time task [40][41] [35][24][38]. Hence, the OpenMP tasking model can be used to parallelize a real-time task, modelled as a DAG, upon which timing guarantees can be provided. Given an OpenMP-DAG $G = (V, E)$, nodes in V correspond to the portions of code that execute uninterruptedly between two *Task Scheduling Points (TSPs)*, referred as *parts of a task region* in the OpenMP specification, and considered as *task parts* henceforward. Edges in E correspond to explicit synchronizations (for instance, defined by the `depend` clause), TSPs (for instance, defined by the `task` construct) and control flow precedence constraints (defined by the sequential execution order of task parts from the same OpenMP task).

Figure 3a shows an example of a real-time task τ parallelized with the OpenMP tasking model, and Figure 3b shows the corresponding OpenMP-DAG. Nodes of the OpenMP-DAG represent the seven tasks parts generated within the four explicit tasks and the implicit task executing the single region. For instance, task T_1 (line 6 of Figure 3a), is composed of task parts $part_{10}$ and $part_{11}$ (nodes p_{10} and p_{11} in Figure 3b). Edges represent (1) the data dependence between $T1$

[3] *Parallel real-time tasks* denote real-time tasks which exploit parallelism within them. These tasks are also concurrent among them.

```
void parallel_RT_task() {                    // τ
#pragma omp parallel
#pragma omp single nowait                    // T0
{
  part00
  #pragma omp task depend(out:x)             // T1
  {
    part10
    #pragma omp task                         // T2
    { part20 }
    part11
  }
  part01
  #pragma omp task                           // T3
  { part30 }
  #pragma omp task depend(in:x)              // T4
  { part40 }
}}
```

(a) Real-time task parallelized with OpenMP tasks.



(b) OpenMP-DAG.

Fig. 3: Example of an OpenMP real-time task.

and T_4; (2) the TSP after the creation of tasks T_1 to T_4, e.g., at the end of task part p_{10} task T_2 is created; and (3) control flow dependences, e.g, task parts p_{10} and p_{11} from T_1 execute sequentially. All threads are synchronized in the implicit barrier at the end of the `parallel` construct (not shown in Figure 3b).

3 Developing Critical Real-time Systems with OpenMP

This section analyses the use of the OpenMP tasking model to develop a critical real-time system, from two different perspectives: 1) how to efficiently exploit parallelism within real-time tasks and among them, and 2) how to express the recurrence of real-time tasks.

3.1 Parallelizing Several Concurrent Real-time Tasks

In critical real-time systems, the scheduler plays a key role as it must guarantee that all real-time tasks execute before its deadline. To do so, real-time schedulers implement the following features (Section 4 provides a detailed analysis): 1) *tasks priorities*, which determine the urgency of each real-time task to execute; 2) *preemption strategies*, which determine when a real-time task can be temporarily interrupted if a more urgent task is ready to execute; and 3) *allocation strategies*, which determine the computing resources (cores) in which tasks can execute.

As introduced in the previous section, current works consider OpenMP only to exploit parallelism within a single real-time task. As a result, each real-time task defines its own OpenMP parallel environment. This becomes a black box for the scheduler, which can not control the resources used by real-time tasks.

In order for the scheduler to have full control over the execution of the real-time tasks (and their parallel execution), the complete taskset must be included within a single OpenMP application. To do so, one option is to exploit nested parallel regions, i.e., to enclose the real-time tasks, each defining its own parallel

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task priority(p1) // τ1: OpenMP-DAG1
    {  RT_task_1()  }
    #pragma omp task priority(p2) // τ2: OpenMP-DAG2
    {  RT_task_2()  }
    ...
    #pragma omp task priority(pn) // τn: OpenMP-DAGn
    {  RT_task_n()  }
}
```

Fig. 4: Critical real-time system implemented with OpenMP parallel tasks.

region (see Figure 3a), within an outer parallel region. In this case, the OpenMP framework manages two scheduling levels: one in charge of scheduling the real-time tasks (outer parallel region), and another one in charge of scheduling the parallel execution within each real-time task (inner parallel regions). Interestingly, this approach enables the first level scheduler to use the `priority` clause associated to the `task` construct to determine the priority of each real-time task (see Section 4). However, this solution is not valid as the first-level scheduler cannot control the parallel execution of each real-time task. In other words, the team of threads of each real-time task is (again) a black box for the first-level scheduler. Hence, preemption and allocation strategies cannot be implemented.

Clearly, the control of the OpenMP threads executing each of the real-time tasks is key to support a fine-grain control over the whole parallel execution. To do so, we propose to define *a common team of OpenMP threads to execute all the real-time tasks.* Figure 4 shows the implementation of a real-time system in which each real-time task $\tau_i \in \mathcal{T}$ is encapsulated within an OpenMP task, and implemented in a function *RT_task_X()*. The code in Figure 3a could represent an example of one of these functions. However, the `parallel` and `single` constructs (lines 2 and 3, respectively) must be removed, and a `taskwait` synchronization construct must be included at the end of the function (line 18). These changes are not shown due to lack of space. In this design, a single real-time scheduler will be in charge of scheduling both, the OpenMP tasks implementing the real-time tasks (with an associated priority given by the `priority` clause), and the nested OpenMP tasks implementing the parallel execution of each real-time task.

3.2 Implementing Recurrent Real-Time Tasks in OpenMP

The OpenMP tasking model is very convenient to implement critical real-time systems based on DAG scheduling models. However, OpenMP lacks an important feature of these systems, *the notion of recurrency.* As presented in Section 2, the execution of real-time tasks can be either periodic or sporadic triggered by an event, e.g., an internal clock or a sensor.

With the objective of including recurrency in the OpenMP execution model, we propose to incorporate a new clause, named `event`, associated to the `task` construct. This clause enables to define the release time of the OpenMP tasks implementing real-time tasks. The syntax of the `event` clause is as follows:

```
#pragma omp task event(event-expression)
```

where *event-expression* is an expression, if it evaluates to true the associated OpenMP task is created. This expression represents the exact moment in time[4] at which the real-time task release occurs or the external event that must occur for the real-time task to release a new job. The expression is true whenever the task releases, and shall evaluate to false after the task creation. However, the `event` clause is not enough to state the synchrony between the event that triggers a real-time task and the actual execution of that task. In languages such as Ada, which are intrinsically concurrent, events are treated at the base language level, thus an Ada task triggering an event will launch an *entry* (a functionality) of a different task. But OpenMP is defined on top of C, C++ and Fortran, languages intrinsically sequential, that do not typically provide these kind of features. Following, we analyze three different approaches to associate the occurrence of an event and the execution of a real-time task:

– *Managed by the base language*: a simple approach would use the base language to implement an infinite loop containing the set of real-time tasks with their corresponding events and priorities. This solution however renders one thread useless, executing the control loop. Interestingly, C++11 introduces multi-threading support, adding features to define concurrent execution.
– *Managed by the operating system*: based on the previous approach, the thread executing the control loop may be freed at the end of each iteration, and the operating system may return the thread to the control loop in a period of time shorter than the minimum period of a task (ensuring no job is missed).
– *Managed by the OpenMP API*: a different approach would be implementing the concept of *persistent* task [27] in the OpenMP API, pushing the responsibility for checking the occurrence of an event to the OpenMP runtime.

A deeper evaluation of the most suitable solution to implement critical real-time system is of paramount importance to promote the use of OpenMP in critical real-time environments. This evaluation is out of the scope of this paper and remains as a future work.

Interestingly, this new `event` clause would allow to unequivocally identify which OpenMP tasks implement real-time tasks, differentiating them from the OpenMP tasks used to parallelize each real-time task. The real-time system implemented in Figure 4 must therefore include the `event` clause associated to each `task` construct at lines 4, 6 and 9.

4 Implementing Real-time Scheduling features in the OpenMP Task Scheduler

One of the most important components of critical real-time systems is the *real-time scheduler*, in charge of, not only assigning the execution of real-time tasks to the underlying computing resources, but also guaranteeing that all tasks execute

[4] Real-Time Operating Systems (RTOS) provide time management mechanisms and timers to determine the release time or deadline of real-time tasks.

before its deadline. In the context of multitasking systems, the scheduling policy is normally priority driven [16], i.e., real-time tasks have a *priority* assigned and the preference to execute is given to the highest-priority tasks. A scheduler may preempt a running task if a more urgent task is ready to execute. The interrupted task resumes later its execution. Moreover, different scheduling algorithms place additional restrictions as to where real-time tasks are allowed to execute. Overall, real-time schedulers can be classified based on: 1) task priorities, 2) preemption strategies and 3) allocation strategies. Following, we describe how these features can be supported by the OpenMP specification.

4.1 Priority-driven Schedulers Algorithms

Depending on the restrictions of how to assign priorities to real-time tasks, priority-based schedulers are classified as follows [11]: (1) *Fixed Task Priority (FTP)*, (2) *Fixed Job Priority (FJP)*, and (3) *Dynamic Priority (DM)*. In FTP, each real-time task has a unique fixed priority. This is the case of the rate-monotonic (RM) scheduler that assigns the priorities based on the period (i.e., tasks with smaller periods have higher priority). In FJP, different jobs of the same real-time task may have different priorities. This is the case of the earliest deadline first (EDF) scheduler that assigns greater priorities to the jobs with earlier deadlines. In DM, the priority of each job may change between its release time and its completion. This is the case of the least laxity (LL) scheduler that assigns the priorities based on the laxity[5] of a job.

In OpenMP, the `priority` clause associated to the `task` construct matches the priority representation of real-time tasks for the FTP scheduling. However, the OpenMP specification (version 4.5) states that *"the priority clause is a hint for the priority of the generated task [..] Among all tasks ready to be executed, higher priority tasks are recommended to execute before lower priority ones. [...] A program that relies on task execution order being determined by this priority-value may have unspecified behavior"*. As a result, the current behavior of the `priority` clause does not guarantee the correct priority-based execution order of real-time tasks. Therefore, the development of OpenMP task schedulers in which the `priority` clause truly leads the scheduling behavior is essential for real-time systems. Moreover, the *priority-expression* value defined at real-time task level must be inherited by the corresponding child tasks implementing parallelism within each real-time task. By doing so, the OpenMP task scheduler can preempt the OpenMP tasks conforming a low priority real-time task in favour of higher priority tasks.

Regarding the implementation of EDF and LL schedulers, a new clause, named `deadline`, associated to the `task` construct is needed. This clause will enable to define the deadline of the real-time task upon which EDF and LL schedulers are based. The syntax of the `deadline` clause is as follows:

[5] The *laxity* of a job at any instant in time is defined as its deadline minus the sum of its remaining processing time and the current time.

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task deadline(D_1) event(e_1) // τ_1 : OpenMP−DAG_1
  { part_00
    #pragma omp task depend(out:x)     // T_1
    { part_10
      #pragma omp task                 // T_2
      { part_20 }
      part_11
    }
    part_01
    #pragma omp task                   // T_3
    { part_30 }
    #pragma omp task depend(in:x)      // T_4
    { part_40 }
  }
  ...
  #pragma omp task deadline(D_n) event(e_n) // τ_n : OpenMP−DAG_n
  {   ...   }
}
```

Fig. 5: OpenMP real-time system designed for a deadline-based scheduler.

`#pragma omp task deadline(`*deadline-expression*`)`

where the *deadline-expression* is the expression that determines the time instant at which the OpenMP task must finish. Similarly to the `priority` clause, the *deadline-expression* associated to an OpenMP task implementing a real-time task must be inherited by all its child tasks. This allows the scheduler to identify those OpenMP tasks with the farthest deadline, and preempt them to assign the corresponding OpenMP threads to those tasks with the closest deadline. The `deadline` clause is not compatible with the `priority` clause, as both are meant for determining the priority of a task for different scheduling algorithms.

Figure 5 shows an example of an OpenMP real-time system, when the scheduler is EDF or LL, and so the `deadline` clause is required. Real-time tasks $\tau_1 ... \tau_n$ have a deadline and an event associated to them. Notice that, in case of a fixed task priority scheduler, the `deadline` clause would be replaced by a `priority` clause. Real-time task τ_1 corresponds to the real-time task represented in Figure 3. All child tasks inherit the deadline of the parent task, for instance, T_1, T_2, T_3 and T_4 inherit the deadline D_1.

4.2 Preemption Strategies

The real-time scheduling theory defines three different types of preemption strategies: (1) *fully-preemptive* (FP), (2) *non-preemptive* (NP), and (3) *limited preemptive* (LP). The FP strategy [9] preempts the execution of low priority tasks as soon as a higher priority task releases. This strategy allows high-priority tasks not to suffer any blocking due to low priority ones. However, it may lead to prohibitively high preemption overheads, mainly related to task context switches and migration delays [15], which may degrade the predictability and performance of the system. The NP strategy [14] executes real-time tasks until completion with no interruption. This strategy offers an alternative that avoids preemption

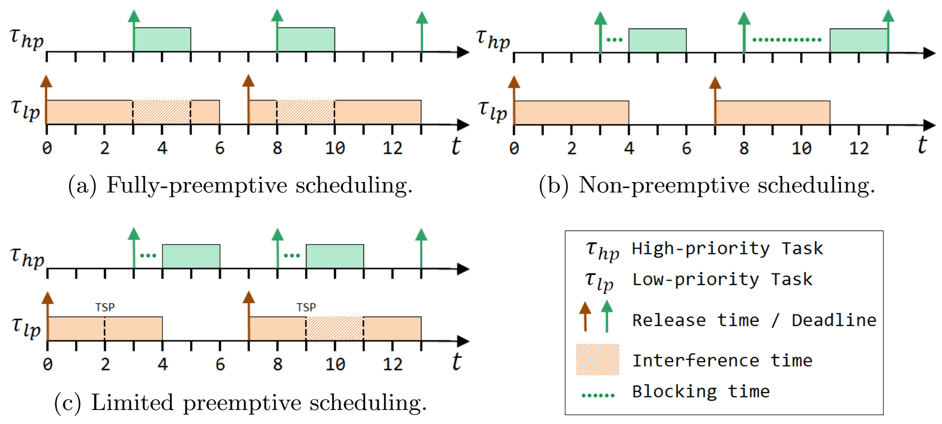

τ_{hp}
τ_{lp}
0 2 4 6 8 10 12 t
(a) Fully-preemptive scheduling.
(b) Non-preemptive scheduling.
TSP
TSP
(c) Limited preemptive scheduling.
τ_{hp} High-priority Task
τ_{lp} Low-priority Task
Release time / Deadline
Interference time
Blocking time

Fig. 6: Scheduling preemption strategies in a single core.

related overheads at the cost of potentially introducing significant blocking effects to higher priority tasks. So far, this strategy has only been considered for sequential real-time tasks. The reason is that parallel real-time tasks may require a different number of computing resources during its execution, and the NP strategy shall guarantee that these resources are always available to avoid preemption operations. Finally, the LP strategy [17] has been proposed as an effective scheduling scheme that reduces the preemption-related overheads of FP, while constraining the blocking effects of NP, thus improving predictability. In LP, preemptions can only take place at certain points during the execution of a real-time task, dividing its execution in non-preemptive regions.

Figure 6 illustrates the three preemption strategies presented above. It considers a task set composed of two tasks: a high-priority task τ_{hp}, with a WCET $C_{hp} = 2$, and a period $T_{hp} = 5$ time units, and a low-priority task τ_{lp}, with a WCET $C_{lp} = 4$, and a period $T_{lp} = 7$ time units. In order to facilitate the explanation, we consider that: a single core is used and real-time tasks are sequential. Moreover, the deadline is equal to the period, so arrows represent the release time of a given job, and the deadline of the previous job. In the FP strategy (Figure 6a), as soon as τ_{hp} is released, at times $t = 3$ and $t = 8$, τ_{lp} is preempted and it resumes as soon as τ_{hp} has finished. In the NP strategy (Figure 6b), although τ_{hp} is released at time instant $t = 3$, it must wait 1 time unit, until τ_{lp} finishes. At the following τ_{hp} release, it must wait again 3 time units. In the LP strategy (Figure 6c), τ_{lp} defines one preemption point (named as TSP). In the first release of τ_{hp}, at time instant $t = 3$, the preemption point of τ_{lp} has already passed, so τ_{hp} must wait until τ_{lp} has finished. In the second release, at time instant $t = 8$, τ_{hp} must wait only until the preemption point of τ_{lp}, at $t = 9$. Then τ_{lp} is preempted, and τ_{hp} starts its execution.

Interestingly, the OpenMP tasking model implements an LP strategy as explained in Section 2.3: OpenMP tasks are preemptable only at TSPs, dividing the task into multiple non preemptive task part regions. Accordingly, the OpenMP runtime can preempt OpenMP tasks and assign its corresponding threads to a different OpenMP task based on the priorities. It is worth noting that OpenMP provides the `taskyield` construct, which allows the programmer to explicitly define additional TSPs. However, regarding task scheduling points, the OpenMP API states that *"the implementation may cause it to perform a task switch"* and regarding the `taskyield` clause, *"the current task can be suspended in favor of execution of a different task"*. This means that an implementation is not forced to perform a task switch in any case. However, in real-time scheduling a TSP must be evaluated, meaning that if a higher priority task is ready at that point, then the lower priority one must be suspended. Therefore, limited preemptive OpenMP schedulers must implement the evaluation of each TSP occurrence.

Interestingly, this laxity in the OpenMP specification, which establishes that threads are allowed to, but not forced to, suspend a task at TSPs, supports the implementation of NP strategies. By simply disabling the suspension of tasks at those points, the OpenMP scheduler would be non-preemptive. In fact, for sequential real-time tasks, this is the default preemption strategy, since there are no implicit TSPs. In this case, it is worth noting that the `taskyield` construct allows the implementation of the LP strategy in sequential real-time tasks as well.

Finally, OpenMP does not support the implementation of FP scheduling strategies because that would require the runtime to preempt the execution of OpenMP tasks at any point of its execution. In any case, as we stated above, FP is not a desirable strategy due to very high preemption overheads it may cause, which can degrade the predictability of the system.

4.3 Allocation and Migration Strategies

There exist three strategies to allocate the execution of real-time tasks to the underlying computing resources (in our case, cores): (1) the *static allocation*, which statically assigns real-time tasks to cores at design time, with the objective of increasing the predictability and minimizing the response time of the overall system; (2) the *dynamic allocation*, in which the allocation is performed based on runtime information, such as the state of the platform (e.g., computing and communication resources available), the set of ready tasks, or the location of input data; (3) the *hybrid allocation*, which statically allocates a subset of real-time tasks, while the rest are dynamically scheduled.

Moreover, real-time schedulers define *migration strategies* to stablish the cores in which real-time tasks are permitted to execute. These strategies can be grouped in three categories: (1) *global scheduling* algorithms allow any real-time task to execute upon any core, allowing jobs from the same real-time task to migrate, (2) *partitioned scheduling* algorithms assign each real-time task to a core so that each job of a real-time task executes always on the same core, and (3) *federated scheduling* algorithms that combine global and partitioned schedulers for a subset of tasks. Typically, the dynamic allocation strategy is built upon global scheduling, whereas static allocation is built upon partitioned scheduling.

Although the OpenMP specification says nothing about allocation strategies, current OpenMP systems are performance-driven, and so all runtime implementations are based on dynamic scheduling. However, static allocation strategies have been proposed for OpenMP as well [24]. In case of migration strategies, the OpenMP *tied* tasking model (the default one) limits the implementation of global schedulers. *Tied* tasks are those that, when suspended, can only be resumed by the same thread that started its execution. As a result, a real-time task implemented as an OpenMP tied task will not be able to migrate. This is not the case of *untied* task, that can be resumed by any thread in the team. In this case the `untied` clause attached to the `task` directive is required.

OpenMP task to OpenMP thread Mapping

With the objective of increasing time predictability, most of the real-time schedulers consider a direct mapping between real-time tasks and cores. This includes two conditions: (1) threads are mapped to cores in a one-to-one manner, and (2) threads are not allowed to migrate between cores.

OpenMP threads are an abstraction of the computing resource upon which OpenMP tasks execute. As stated in Section 3, this paper considers a single team of threads to execute all OpenMP tasks. This enables the real-time scheduler to have full control over the execution of OpenMP tasks over threads. However, OpenMP threads are further assigned to the operating system, hardware threads and cores (referred to as *places* in OpenMP), existing other levels of scheduling out of the control of the OpenMP scheduler.

Fortunately, the OpenMP specification provides mechanisms to consider a single real-time scheduler and so fulfilling the two conditions stated above. On one hand, the `requires` directive, which will be introduced in the next OpenMP specification, version 5.0 [4], allows to specify the features an implementation must provide in order for the code to compile and execute correctly. This may be useful to express the minimum number of cores that the target architecture must provide to guarantee a one-to-one mapping, as required by the system. On the other hand, OpenMP defines the *bind-var* internal control variable together with the `proc_bind` clause, which allow to control the binding of OpenMP threads to cores, enabling to define different thread-affinity policies. Finally, the *place-partition-var* internal control variable controls the list of places available.

Overall, an OpenMP framework intended to implement a critical real-time system must obey the following constraints: (a) *place-partition-var* := `cores`, so that each OpenMP place corresponds to a single core; and (b) *bind-var* := `close`, so that OpenMP threads are consecutively assigned to places (forbidding threads migration between places). Once OpenMP threads are assigned to cores, this affinity must not be modified. Therefore, the `proc_bind` clause must be forbidden or ignored. Moreover, we propose to use the `requires` directive along with the `ext_min_cores` clause and an integer value, to determine the minimum number of threads (and so, cores) necessary to correctly execute the system.

4.4 Evaluation of Current OpenMP implementations

This section evaluates how priorities and preemptions are treated in the OpenMP runtime implementation provided by GCC 8.1 [28] and Nanos++ [8]. To do so, we consider the source code presented in Figure 7a, in which two real-time tasks, T_1 and T_2, are created, T_1 having lower priority than T_2. Moreover, T_1 includes an

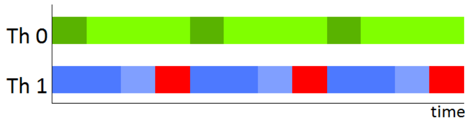
```
 1 #pragma omp parallel
 2 #pragma omp single
 3 {
 4   while (1) {
 5     #pragma omp task priority(1) // T1
 6     {
 7       work_11();
 8       #pragma omp taskyield
 9       work_12();
10     }
11     work_01();
12     #pragma omp task priority(2) // T2
13     {  work_21();  }
14     work_02();
15 }}
```

🟩 *work*_01()
🟩 *work*_02()

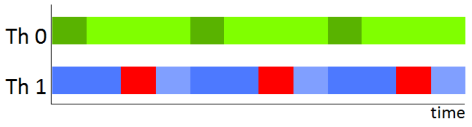
T_1 Low priority task
🟦 *work*_11()
🟦 *work*_12()

T_2 High priority task
🟥 *work*_21()

(a) OpenMP code.



(b) Execution traces, GCC 8.1.



(c) Execution traces, Nanos++.

Fig. 7: Real-time system example: LP scheduling and the `priority` clause.

explicit TSP by means of the `taskyield` construct. Therefore, T_1 is divided into two non-preemptive task parts. Sequential real-time tasks and two threads have been considered for simplicity. Current OpenMP implementations only support dynamic allocation and global scheduling.

Critical real-time systems must honor the priority of each task because these determine preeminence of some tasks over others. Moreover, in the LP strategy, low priority tasks are preempted at preemption points (TSPs) in favour of high priority ones to guarantee that all tasks meet its deadline. Hence, in the example shown in Figure 7a, T_1 gets first the idle thread as it is created before T_2. However, if T_2 is already created (and ready to execute) at the TSP of T_1 (line 8), T_1 must be preempted and the thread must be assigned to T_2 to honor priorities.

The execution traces[6] of three iterations of the source code presented in Figure 7a are shown in Figure 7b, using GCC 8.1, and Figure 7c, using Nanos++. Green blocks represent the execution of the code within the `single` construct (*work_01* and *work_02*) in the thread *Th* 0. Blue blocks represent the execution of T_1 (*work_11* and *work_12*) in *Th* 1. Red blocks represent the execution of T_2 (*work_21*) in *Th* 1. The exact expected behavior is observed in Nanos++, since T_2 executes between the two task parts of T_1. However, in GCC, T_2 executes after T_1 completes, because the preemption point of T_1 is not honored: T_2 is not executed as soon as possible.

Overall, although current OpenMP runtimes are not ready to support the development and execution of critical real-time systems, Nanos++ already implements some of the fundamental features needed by critical real-time systems. This is not the case of GCC 8.1.

[6] Traces obtained with Extrae and Paraver performance monitoring tools [6,7].

5 Related work

The performance requirements of advanced embedded critical real-time systems entails a booming trend to use multi-core, many-core and heterogeneous architectures. As we stated in early sections of this paper, OpenMP has been already considered to cope with these performance needs [21,1]. In this context, OpenMP has been analyzed regarding the two features that are mandatory in such restricted systems: timing analysis and functional safety.

From a timing perspective, there is a significant amount of work considering the time predictability properties of OpenMP. Despite the fork-join was firstly considered [22], the tasking model seems to be more suitable given its capabilities to define fine grain, both structured and unstructured parallelism. For this reason several works [40,35,38,24] studied the OpenMP tasking model and its similarities with the sporadic DAG scheduling model. However, none of these works consider a complete real-time system, but a unique non-recurrent real-time task. The schedulability analysis of a full DAG task-based real-time system has been addressed for homogeneous architectures under different scheduling strategies [23,12,33,34,19,10]. Recently, a response-time analysis has been proposed for a DAG task supporting heterogeneous computing [36]: the OpenMP accelerator model is proposed to address heterogeneous architectures. From a functional safety perspective, OpenMP is considered as a convenient candidate to implement real-time systems, although some features and restrictions must be addressed [29]. Based on the potential of existent correctness techniques for OpenMP, it could be introduced in safe languages such as Ada [30,32,31], widely used to implement safety-critical systems. The Ada Rapporteur Group is considering the introduction of OpenMP into Ada [26] to exploit fine grain parallelism.

Finally, as embedded systems usually have tight constraints regarding resources such as memory (e.g., the Kalray MPPA has 2MB shared memory [18]), different approaches for developing lightweight OpenMP runtime systems [41,39] coexist. These studies are meant to efficiently support OpenMP in such constrained environments. For instance, the memory used at runtime is reduced when the task dependency graph of the applications is statically derived.

6 Conclusions

OpenMP is a firm candidate to address the performance challenges of critical real-time systems. However, OpenMP was originally intended for a different purpose than critical real-time systems, for which guaranteeing the correct output is as important as guaranteeing it within a predefined time budget. In this paper, we evaluate the use of the OpenMP tasking model to develop and execute the sporadic DAG-based scheduling model upon which many critical real-time systems are based on, e.g., IMA and AUTOSAR used in avionics and automotive respectively. Concretely, we propose the use of a single team of threads to implement and execute both, concurrent real-time tasks and the parallelism within them. Two new clauses, `event` and `deadline`, are proposed to allow the implementation of recurrent real-time tasks and FJP and DM schedulers. Moreover, we analyze some important features already provided in the OpenMP API: the `priority` clause and the TSPs. The defined behavior of these two features is not desirable for critical real-time systems. In both cases, it must be a prescriptive modifier, instead of a hint (the case of the `priority` clause) or a possibility of occurrence (the case of TSPs). In order to implement limited preemptive scheduling, the most suitable preemptive strategy for OpenMP real-time systems, it must be guaranteed that, at each preemption point (TSP), if there is a higher priority task ready, the running task is suspended in favor of the highest priority task. Overall, correctly addressing all these features in the specification is of paramount importance to use OpenMP in critical real-time systems.

Nevertheless, some design implications need a deeper analysis and evaluation. This is the case, for instance, of the event-driven execution model not supported in OpenMP, which remains as future work.

Acknowledgments

The research leading to these results has received funding from the Spanish Ministry of Science and Innovation, under contract TIN2015-65316-P, and from the European Union's Horizon 2020 Programme under the CLASS Project (www.class-project.eu), grant agreement No 780622.

References

1. P-SOCRATES European Project (Parallel Software Framework for Time-Critical Many-core Systems). http://p-socrates.eu
2. ARINC Specification 653: Avionics Application Software Standard Standard Interface, Part 1 and 4 (2012)
3. OpenMP Application Programming Interface. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (2015)
4. OpenMP Technical Report 6: Version 5.0 Preview 2. http://www.openmp.org/wp-content/uploads/openmp-TR6.pdf (2017)
5. AURIX^{TM} Safety joins Performance. https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/aurix-safety-joins-performance/ (2018)
6. Barcelona Supercomputing Center: Extrae Release 3.5.2. https://tools.bsc.es/extrae
7. Barcelona Supercomputing Center: Paraver Release 4.7.2. https://tools.bsc.es/paraver
8. Barcelona Supercomputing Center: OmpSs 1.0 Specification. https://pm.bsc.es/ompss-docs/specs/ (2016)
9. Baruah, S.: Techniques for multiprocessor global schedulability analysis. In: Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS) (2007)
10. Baruah, S.: The federated scheduling of constrained-deadline sporadic DAG task systems. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE) (2015)
11. Baruah, S., Bertogna, M., Buttazzo, G.: Multiprocessor Scheduling for Real-Time Systems. Springer (2015)
12. Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A.: The global EDF scheduling of systems of conditional sporadic dag tasks. In: Proceedings of the 27th IEEE Euromicro Conference on Real-Time Systems (ECRTS) (2015)
13. Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stougie, L., Wiese, A.: A generalized parallel task model for recurrent real-time processes. In: Proceedings of the 38th IEEE Real-Time Systems Symposium (RTSS) (2012)
14. Baruah, S.K., Chakraborty, S.: Schedulability analysis of non-preemptive recurring real-time tasks. In: Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS) (2006)
15. Bastoni, A., Brandenburg, B., Anderson, J.: Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. Proceedings of OSPERT (2010)
16. Buttazzo, G.C.: Hard real-time computing systems: predictable scheduling algorithms and applications, vol. 24. Springer Science & Business Media (2011)

17. Buttazzo, G.C., Bertogna, M., Yao, G.: Limited preemptive scheduling for real-time systems. a survey. IEEE Transactions on Industrial Informatics 9(1) (2013)
18. De Dinechin, B.D., Van Amstel, D., Poulhiès, M., Lager, G.: Time-critical computing on a single-chip massively parallel processor. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE) (2014)
19. Fonseca, J., Nelissen, G., Nelis, V., Pinho, L.M.: Response time analysis of sporadic DAG tasks under partitioned scheduling. In: Proceedings of the 11th IEEE Symposium on Industrial Embedded Systems (SIES) (2016)
20. GbR, A.: AUTomotive Open System ARchitecture (AUTOSAR), Standard v4.1. http://www.autosar.org (2014)
21. Hanawa, T., Sato, M., Lee, J., Imada, T., Kimura, H., Boku, T.: Evaluation of multicore processors for embedded systems by parallel benchmark program using OpenMP. In: Proceedings of the International Workshop on OpenMP (IWOMP) (2009)
22. Lakshmanan, K., Kato, S., Rajkumar, R.: Scheduling parallel real-time tasks on multi-core processors. In: Proceedings of the IEEE 31st Real-Time Systems Symposium (RTSS) (2010)
23. Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A., Buttazzo, G.C.: Response-time analysis of conditional DAG tasks in multiprocessor systems. In: Proceedings of the 27th IEEE Euromicro Conference on Real-Time Systems (ECRTS) (2015)
24. Melani, A., Serrano, M.A., Bertogna, M., Cerutti, I., Quiñones, E., Buttazzo, G.: A static scheduling approach to enable safety-critical openmp applications. In: Proceedings of the 22nd IEEE Asia and South Pacific Design Automation Conference (ASP-DAC) (2017)
25. Mok, A.K.: Task management techniques for enforcing ED scheduling on periodic task set. In: Proceedings of the 5th IEEE Workshop on Real-Time Software and Operating Systems (1988)
26. Pinho, L.M., Quiñones, E., Royuela, S.: Combining the tasklet model with OpenMP. In: 19th International Real-Time Ada Workshop (2018)
27. Pop, A., Cohen, A.: A stream-computing extension to OpenMP. In: Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers. pp. 5–14. ACM (2011)
28. project, G.: GNU libgomp (November 2015), uRL: https://gcc.gnu.org/projects/gomp/
29. Royuela, S., Duran, A., Serrano, M.A., Quiñones, E., Martorell, X.: A functional safety OpenMP for critical real-time embedded systems. In: Proceedings of the International Workshop on OpenMP (IWOMP) (2017)
30. Royuela, S., Martorell, X., Quiñones, E., Pinho, L.M.: OpenM tasking model for Ada: safety and correctness. In: Proceedings of the Ada-Europe International Conference on Reliable Software Technologies (2017)
31. Royuela, S., Martorell, X., Quiñones, E., Pinho, L.M.: Safe Parallelism: Compiler Analysis Techniques for Ada and OpenMP. In: Ada-Europe International Conference on Reliable Software Technologies. Springer (2018)
32. Royuela, S., Pinho, L.M., Quiñones, E.: Converging Safety and High-performance Domains: Integrating OpenMP into Ada. In: Design, Automation Test in Europe Conference Exhibition (2018)
33. Serrano, M.A., Melani, A., Bertogna, M., Quiñones, E.: Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE) (2016)
34. Serrano, M.A., Melani, A., Kehr, S., Bertogna, M., Quinones, E.: An analysis of lazy and eager limited preemption approaches under DAG-based global fixed priority scheduling. In: Proceedings of the International Symposium on Real-Time Distributed Computing (ISORC) (2017)
35. Serrano, M.A., Melani, A., Vargas, R., Marongiu, A., Bertogna, M., Quiñones, E.: Timing characterization of OpenMP4 tasking model. In: Proceedings of the IEEE International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES) (2015)
36. Serrano, M.A., Quiñones, E.: Response-time analysis of DAG tasks supporting heterogeneous computing. In: Proceedings of the Annual Design Automation Conference (DAC) - to appear - (2018)
37. Stotzer, E., Jayaraj, A., Ali, M., Friedmann, A., Mitra, G., Rendell, A.P., Lintault, I.: OpenMP on the low-power TI keystone II ARM/DSP system-on-chip. In: Proceedings of the International Workshop on OpenMP (IWOMP) (2013)
38. Sun, J., Guan, N., Wang, Y., He, Q., Yi, W.: Scheduling and analysis of real-time OpenMP task systems with tied tasks. In: Proceedings of the IEEE Real-Time Systems Symposium (RTSS) (2017)
39. Tagliavini, G., Cesarini, D., Marongiu, A.: Unleashing Fine-Grained Parallelism on Embedded Many-Core Accelerators with Lightweight OpenMP Tasking. IEEE Transactions on Parallel and Distributed Systems (2018)
40. Vargas, R., Quiñones, E., Marongiu, A.: Openmp and timing predictability: a possible union? In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE) (2015)
41. Vargas, R.E., Royuela, S., Serrano, M.A., Martorell, X., Quiñones, E.: A lightweight openmp4 run-time for embedded systems. In: Proceedings of the IEEE 21st Asia and South Pacific Design Automation Conference (ASP-DAC) (2016)