4

Mapping, Scheduling, and Schedulability Analysis

Paolo Burgio[1], Marko Bertogna[1], Alessandra Melani[1], Eduardo Quiñones[2] and Maria A. Serrano[2]

[1]University of Modena and Reggio Emilia, Italy
[2]Barcelona Supercomputing Center (BSC), Spain

This chapter presents how the P-SOCRATES framework addresses the issue of scheduling multiple real-time tasks (RT tasks), made of multiple and concurrent non-preemptable *task parts*. In its most generic form, the scheduling problem in the architectural framework is a dual problem: scheduling task-to-threads, and scheduling thread-to-core replication.

4.1 Introduction

In our framework, we assume threads in the same OpenMP application are statically *pinned* to the available cores in the platforms[1]. This approach has two advantages: (i) the lower layer of the software stack, namely the runtime and the operating system (OS) support, are much simpler to design and implement; and (ii) we remove one dimension from the scheduling problem, that is, we only need to solve the problem of assigning tasks (in our case, OpenMP task parts) to threads/cores. For this reason, and limited to this chapter, we use the words "mapping" and "scheduling" interchangeably. As explained in Chapter 3, when a task encounters a task scheduling point (TSP), program execution branches into the OpenMP runtime, where task-to-thread mapping can: (1) begin the execution of a task region bound to the current team or (2) resume any previously suspended task region bound to the current

[1]Still, to enable multitasking at the OS level, the OS can preempt threads from one OpenMP application in favour of another OpenMP application.

team, as defined by the *parallel* OpenMP construct. Note that, the order in which these two actions are applied is not specified by the standard. An ideal task scheduler will schedule tasks for execution in a way that maximizes concurrency while accounting for load imbalance and locality to facilitate better performance.

The following part of the chapter describes the design of a simple partitioned scheduler, detailing how to enforce a limited-preemption scheduling policy to limit the overhead related to context switches whenever higher-priority instances arrive while the cores are busy executing lower-priority workload. It is also called *static* approach.

Then, we introduce the so-called dynamic approach, where scheduling happens with the adoption of a global queue where all tasks are inserted, and from where they can potentially be fetched by any worker in the system. We also show how it can be enhanced to support task migration across computing threads and cores, in a work-conservative environment.

In the following part, we describe our overall framework for the schedulability analysis, and then we specialize it for static/partitioned approach and dynamic/global approach, respectively.

We then briefly discuss the scheduling problem in the multi-core system that powers the four I/O clusters present in the fabric.

4.2 System Model

In the framework, an application may consist of multiple RT task instances, each one characterized by a different period or minimum inter-arrival time, deadline and execution requirement (see Figure 4.1). Each RT task starts executing on the host processor and may include (OpenMP-compliant) parallel workloads to be offloaded to the many-core accelerator. Such a parallel workload needs then to be scheduled on the available processing elements (PEs).

The parallel execution of each RT task is represented by a direct acyclic graph (DAG) composed of a set of nodes representing task parts. Nodes are connected through edges that represent precedence constraints among different task parts of the same offload. A task part can be executed only if all nodes that have a precedence constraint over it have already been executed.

To comply with the OpenMP semantics, an RT task is not directly scheduled on the PEs. Instead, its parallel workload is first mapped to several OS threads (up to the number of PEs available), and then these OS threads are scheduled onto the available cores. Figure 4.2 summarizes

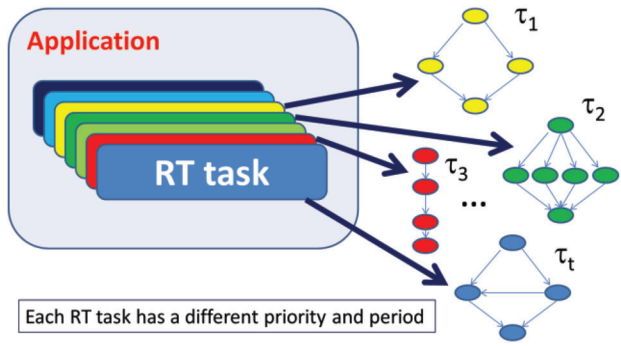

Figure 4.1 An application is composed of multiple real-time tasks.

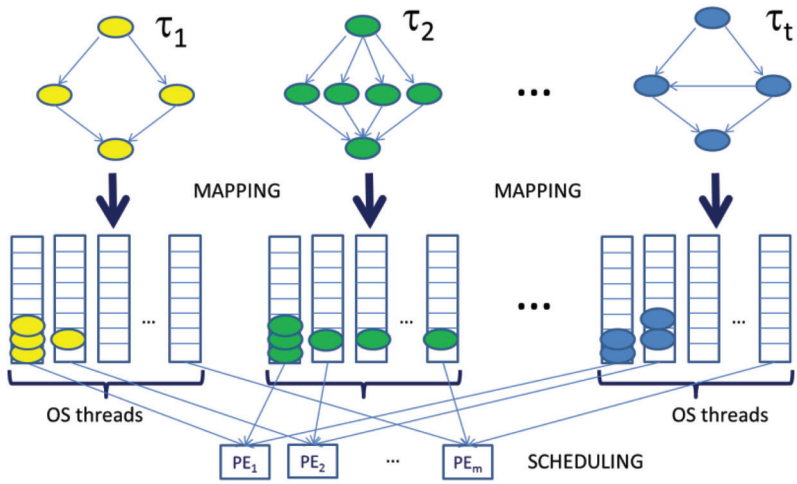

Figure 4.2 RT tasks are mapped to OS threads, which are scheduled on the processing elements.

the mapping/scheduling framework. Here, only partitioned/static approach is shown, where there is one task queue for each worker thread. In a fully dynamic/global approach, there is only one queue for every RT task, where all threads push and fetch work.

The number of OS threads onto which an RT task is mapped depends on mapping decisions. If the RT task does not present a large parallelism, it makes no sense to map it onto more than a limited number of threads. If instead the RT task has massively parallel regions, it may be useful to map it to a higher number of threads, up to the number of PEs in the many-core accelerator. A correct decision should consider the trade-off between

the OS overhead implied by many threads and the speed-up obtainable with a larger run-time parallelism. Note that creating a larger number of threads than necessary may impose a significant burden to the OS, which needs to maintain the context of these threads, schedule, suspend, and resume them, with an obvious increase in the system overhead.

The architectural template targeted in the project (described in Chapter 2) is a many-core platform where cores are grouped onto clusters. The testbed accelerator, Kalray MPPA of the "Bostan" generation, has 256 cores grouped into 16 clusters of 16 cores each. We consider only the threads offloaded to the same cluster. Note that the intra-cluster scheduling problem is the main problem to solve in our scheduling framework. The reason is that P-SOCRATES adopts an execution model, where the context of each RT task that may need to be accelerated is statically offloaded to the target clusters before runtime.

For the above reasons, the main problem is therefore how to efficiently activate and schedule the threads associated with the different RT tasks that have been offloaded to the same cluster. The threads of each RT task will contend to execute on the available PEs with the threads of the other RT tasks. A smart scheduler will therefore need to decide which thread, or set of threads, to execute at any time in each PE of the considered cluster, such that all scheduling constraints are met. Depending on the characteristics of the running RT tasks (priority, period, deadline, etc.) the scheduler may choose to preempt an executing thread or set of threads, to schedule a different set of threads belonging to a higher priority (or more urgent) RT task.

4.3 Partitioned Scheduler

In a traditional partitioned scheduler, OS threads are statically assigned to cores, so that no thread may migrate from one core to another. The scheduling problem then reduces to the design of a single-core scheduler. We start from this approach and design our task-to-thread scheduler.

4.3.1 The Optimality of EDF on Preemptive Uniprocessors

The earliest deadline first (EDF) scheduling algorithm assigns scheduling priority to jobs according to their absolute deadlines: the earlier the deadline, the greater the priority (with ties broken arbitrarily). EDF is known to be *optimal* for scheduling a collection of jobs upon a preemptive uniprocessor platform, in the sense that *if a given collection of jobs can be scheduled*

to meet all deadlines, then the EDF-generated schedule for this collection of jobs will also meet all deadlines [1]. To show that a system is EDF-schedulable upon a preemptive uniprocessor, it suffices to show the existence of a schedule meeting all deadlines — the optimality of EDF ensures that it will find such a schedule. Unfortunately, most of the commercial RTOSes do not implement the EDF scheduling policy. The main reasons are found in the added complexity of the scheduler, requiring timers to keep track of the thread deadlines, and in the agnostic behavior with respect to higher-priority workload. This last concern is particularly important for industrial applications that have a set of higher-priority instances whose execution cannot be delayed. With an EDF scheduler, a lower-priority instance overrunning its expected budget may end up causing a deadline miss of a higher priority instance that has a later deadline. Instead, with a Fixed Priority (FP) scheduler, higher-priority jobs are protected against lower-priority overruns, because they will always be able to preempt a misbehaving lower-priority instance. This makes FP scheduling more robust for mixed-criticality scenarios where RT tasks of different criticality may contend for the same PEs. For the importance of FP scheduling, we decided to implement a partitioned scheduler based on this policy.

4.3.2 FP-scheduling Algorithms

In an FP-scheduling algorithm, each thread is assigned a distinct priority (as in P-SOCRATES scheduling model) and every instance (a.k.a. job/RT task instance) released by the thread inherits the priority of the associated thread.

The rate-monotonic (RM) scheduling algorithm [1] is an FP-scheduling algorithm in which the priorities of the tasks are defined based on their period: tasks with a smaller period are assigned greater priority (with ties broken arbitrarily). It is known [1] that RM is an optimal FP-scheduling algorithm for scheduling threads with relative deadlines equal to their minimum inter-arrival times upon preemptive uniprocessors: if there is any FP-scheduling algorithm that can schedule a given set of implicit-deadline threads to always meet all deadlines of all jobs, then RM will also always meet all deadlines of all jobs.

The deadline monotonic (DM) scheduling algorithm [2] is another FP-scheduling algorithm in which the priority of a task is defined based on its relative deadline parameter rather than its period: threads with smaller relative deadlines are assigned greater priority (with ties broken arbitrarily). Note that RM and DM are equivalent for implicit deadline systems, since

all threads in such systems have their relative deadline parameters equal to their periods. It has been shown in [2] that DM is an optimal FP-scheduling algorithm for scheduling sets of constrained-deadline threads upon preemptive uniprocessors: if there is any FP-scheduling algorithm that can schedule a given constrained-deadline system to always meet all deadlines of all jobs, then DM will also always meet all deadlines of all jobs. DM is, however, known to not be optimal for systems where threads may have a deadline larger than their period.

4.3.3 Limited Preemption Scheduling

Preemption is a key concept in real-time scheduling, since it allows the OS to immediately allocate the processor to threads requiring urgent service. In fully preemptive systems, the running thread can be interrupted at any time by another thread with higher priority and be resumed to continue when all higher priority threads have completed. In other systems, preemption may be disabled for certain intervals of time during the execution of critical operations (e.g., interrupt service routines, critical sections, etc.). In other situations, preemption can be completely forbidden to avoid unpredictable interference among threads and achieve a higher degree of predictability (although higher blocking times).

The question of whether to enable or disable preemption during thread execution has been investigated by many authors under several points of view and it is not trivial to answer. A general disadvantage of the non-preemptive discipline is that it introduces additional blocking time in higher-priority threads, thereby reducing schedulability. On the other hand, preemptive scheduling may add a significant overhead due to context switches, significantly increasing the worst-case execution time. Both situations are schematized in Figure 4.3. CRPD in the figure stands for Cache-Related Preemption Delay, that is, the time overhead added to tasks' execution time due to cache cooling after a preemption.

There are several advantages to be considered when adopting a non-preemptive scheduler. Arbitrary preemptions can introduce a significant runtime overhead and may cause high fluctuations in thread-execution times, which degrades system predictability. Specifically, at least four different types of costs need to be considered at each preemption:

1. *Scheduling cost*: It is the time taken by the scheduling algorithm to suspend the running thread, insert it into the ready queue, switch the context, and dispatch the new incoming thread.

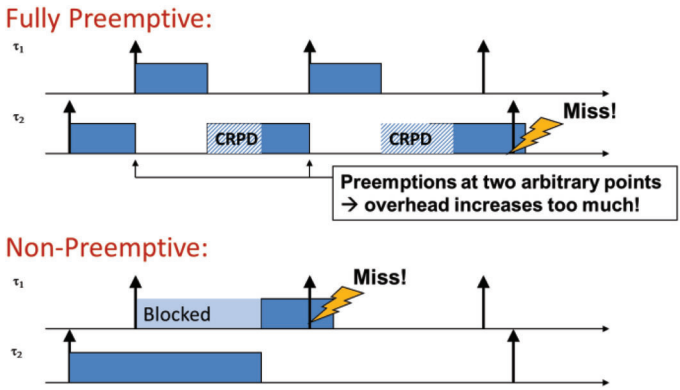

Figure 4.3 Fully preemptive vs. non-preemptive scheduling: preemption overhead and blocking delay may cause deadline misses.

2. *Pipeline cost*: It accounts for the time taken to flush the processor pipeline when the thread is interrupted, and the time taken to refill the pipeline when the thread is resumed.
3. *Cache-related cost*: It is the time taken to reload the cache lines evicted by the preempting thread. The WCET increment due to cache interference can be very large with respect to the WCET measured in non-preemptive mode.
4. *Bus-related cost*: It is the extra bus interference for accessing the next memory level due to the additional cache misses caused by preemption.

In order to predictably bound these penalties without sacrificing schedulability, we decided to adopt a limited preemption scheduler, which represents a trade-off between fully preemptive and non-preemptive scheduling. Note that this seamlessly integrates into the standard OpenMP tasking/execution model, where tasks can be preempted only at well-defined TSPs. See also Chapter 3.

4.3.4 Limited Preemption Schedulability Analysis

As in the fully preemptive case, the schedulability analysis of limited preemptive scheduling can be done analyzing the critical instant that leads to the worst-case response time of a given thread. However, differently from the fully preemptive case, the critical instant is not given by the synchronous arrival sequence, where all threads arrive at the same time, and all successive instances are released as soon as possible. Instead, in the presence of non-preemptive regions, the additional blocking from lower priority threads must

be taken into account. Hence, the critical instant for a thread τ_i occurs when it is released synchronously and periodically with all higher priority threads, while the lower priority thread that is responsible of the largest blocking time of τ_i is released one unit of time before τ_i.

However, the largest response time of a thread is not necessarily due to the first job after a critical instant but might be due to later jobs. Therefore, as shown in [3], the schedulability analysis needs to check all τ_i's jobs within a given period of interest that goes from the above described critical instant until the first idle instant of τ_i. Let K_i be the number of such jobs.

When analyzing the schedulability of limited preemptive systems, a key role is played by the last non-preemptive region. Let q_i^{last} be the length of the last non-preemptive region of thread τ_i. When such a value is large, the response time of τ_i may decrease because the execution of many higher-priority instances is postponed after the end of τ_i, thus not interfering with τ_i. This allows improving the schedulability over the fully preemptive approach.

The blocking tolerance β_i of thread τ_i is defined as the maximum blocking that τ_i can tolerate without missing its deadline. Such a value may be computed by the following pseudo-polynomial relation:

$$\beta_i = \min_{k \in [1, K_i]} \max_{t \in \Pi_{i,k}} \left\{ t - kC_i + q_i^{last} - \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j \right\}$$

where $\Pi_{\mathrm{i,k}}$ is the set of release times of jobs within the period of interest. The maximum allowed non-preemptive region of a τ_k is then given by:

$$NPR_k^{\max} \leftarrow \min_{i<k} \{\beta_i\}$$

Such a value determines the maximum spacing between two consecutive preemption points for each thread τ_k.

4.4 Global Scheduler with Migration Support

4.4.1 Migration-based Scheduler

The scheduling problem for single-core systems has already been solved with optimal priority assignments and scheduling algorithms back in the 1970s. In particular, RM assigning priorities with decreasing task periods, and DM assigning priorities with decreasing relative deadlines, are optimal priority

assignments for sporadic systems with, respectively, implicit and constrained deadlines. This means that if a sporadic or synchronous periodic task system can be scheduled with fixed priorities on a single processor, then it can also be scheduled using RM (for implicit deadlines) [4] or DM (for constrained deadlines) [2]. Also, the EDF — that schedules at each time-instant the ready job with the earliest absolute deadline — is an optimal scheduling algorithm for scheduling arbitrary collections of jobs on a single processor [3, 4]. Therefore, if it is possible to schedule a set of jobs such that all deadlines are met, then the same collection of jobs can be successfully scheduled by EDF as well. These observations allowed us to optimally select the scheduling policies for the partitioned scheduler that we will describe shortly.

When allowing tasks to migrate among different cores, such as in the case of OpenMP untied task model (see Chapter 3 for further information), things are much more complicated: EDF, RM, and DM are no more optimal and can fail even at very low utilizations (arbitrarily close to one) due to the so-called Dhall's effect [5]. Still, these are unlucky corner cases which do not often recur in practice. The alternative approaches that allow higher schedulability ratios are dynamic algorithms that however lead to a higher number of preemptions and migrations, allowing the priority of a job to change multiple times. Examples are Pfair [6, 7], BF [8], LLREF [9], EKG [10], E-TNPA [11], LRE-TL [12], DP-fair [13], BF^2 [14, 15], and RUN [16]. The optimality of the above algorithms holds under very restrictive circumstances, i.e., neglecting preemption and migration overhead, and for sequential sporadic tasks with implicit deadlines. In this case, they are able to reach a full schedulable utilization, equal to the number of processors. Instead, they are not optimal when tasks may have deadlines different from periods (it has been shown in [17] that an optimal scheduler would require clairvoyance), for more general task models including parallel regions, limited preemptions, and/or DAG-structures, as with the task models adopted in the P-SOCRATES project.

The additional complexity inherent to the implementation, runtime overhead, scheduling and schedulability analysis of dynamic scheduling algorithms, as well as in the lack of optimality properties with relation to the task model adopted in the project, made their applicability to the considered setting questionable. For this reason, we decided to opt for the static priority class of scheduling algorithms, which is far more used in a practical setting due to some particularly desired features. Systems scheduled with static priority algorithms are rather easy to implement and to analyze; they allow

reducing the response time of more critical tasks by increasing their priorities; they have a limited number of preemptions (and therefore migrations), bounded by the number of jobs activations in a given interval; they allow selectively refining the scheduling of the system by simply modifying the priority assignment, without needing to change the core of the scheduling algorithm (a much more critical component); they are easier to debug, simplifying the understanding of system monitoring traces and making it more intuitive to figure out why/when each task executes on which core; they are far more composable, i.e., changing any timing parameter of a lower-priority task does not alter the schedule of a higher-priority one, avoiding the need to recheck and re-validate the whole system.

4.4.2 Putting All Together

In our scheduling framework, the global scheduler will therefore consist of a fixed-priority scheduling algorithm. Each RT task is assigned a fixed priority, which is inherited by each one of its threads (there are at most m threads for each RT task). Threads that are *ready* to execute are ordered according to their priority in a global queue ("ready queue") from which the scheduler selects the m highest priority ones for execution, being m the number of available cores. These executing threads are popped from the queue and they change their state to *running*. New thread activations and incoming offloads are queued in the ready queue, based on their priorities. A blocked queue is also maintained with all suspended or waiting threads. Whenever a waiting thread is awakened, e.g., because the condition it was waiting for was satisfied, it is removed from the blocked queue and re-inserted into the ready queue according to its priority.

If the newly activated thread has a priority higher than one of the m *running* tasks, a preemption may take place, depending on the adopted preemption policy. With a fully preemptive scheduler, the preemption takes place immediately, as soon as the thread is (re-)activated. With a non-preemptive policy, the preemption is postponed until one of the running tasks finishes its execution. For this framework, we decided to adopt a limited preemption policy. According to this policy, threads are non-preemptively executed until they reach one of the statically defined preemption points, where they can be preempted if a higher priority thread is waiting to execute. This policy allows decreasing the preemption and migration overhead of fully preemptive policies, without imposing the excessive blocking delays experienced with non-preemptive approaches.

The problem with adopting a limited preemption scheduling policy is that it is necessary to define at which points to allow a preemption for each thread. Since requiring the programmer to manually insert suitable context-switch locations overly increases the programming complexity, we decided to automate the process by using meaningful information coming from the OpenMP mapping layer. In particular, the concept of TSP will be exposed to the scheduling layer in order to take informed decisions on when and where to allow a preemption. We will now detail this strategy.

4.4.3 Implementation of a Limited Preemption Scheduler

Arbitrary preemptions can introduce a significant runtime overhead and high fluctuations in thread-execution times, which degrades system predictability. These variations are due to multiple factors, including the time taken by the scheduling algorithm to suspend the running thread, insert it into the ready queue, switch the context, and dispatch the new incoming thread; the time taken to reload the cache lines evicted by the preempting thread; and the extra bus interference for accessing the next memory level due to the additional cache misses caused by preemption. Conversely, completely forbidding preemptions may cause an intolerable blocking to higher priority threads, potentially affecting their schedulability. For example, consider a system where a low-priority activity offloaded a parallel workload executing on all available cores. If a higher priority RT task now requests a subset of the cores to execute more important activities, it will need to wait until the low-priority ones are finished, eventually leading to a deadline miss. Such a miss could have been easily avoided by allowing preemptions.

With the limited preemption scheduling model adopted in the project, threads will execute non-preemptively until they reach a TSP. At these points, the execution control is moved back to the OpenMP runtime to decide which task (part) to map on that thread. Essentially, the mapper will fetch one of the tasks (belonging to the offload associated to the considered thread) that are ready to execute and map it to that thread. These are points that mark an interruption in the task-execution flow, potentially leading to context switches and/or some memory locality loss. In other words, TSPs are good candidate to be selected for potential preemption points, since they may represent a discontinuity in the continuous execution of a task, potentially requiring a new task to load new data to local memory. Taking advantage of these points seems reasonable to guarantee a reduced pollution of cache locality of an executing task, allowing a thread context switch only when a preemption causes less harm.

However, it remains to be shown how the information from the OpenMP runtime is to be propagated to the RTOS scheduling layer. Note that every RT task that is offloaded to the accelerator is managed by an instance of custom OpenMP runtime. This instance, among the other tasks, keeps track of the dependencies among the nodes of the RT task (the OpenMP task parts), and schedules for execution only those nodes whose dependencies have been satisfied. When a thread fetches a task from the pool for execution, it will continue uninterruptedly until it reaches a TSP. At this point, the runtime regains control, and it may decide to invoke the OS scheduler using a simple function call. The scheduler can then check whether there are new offload requests pending and/or there are blocked tasks that have been awakened. Potential higher-priority threads arrivals will then trigger a preemption, saving the context of the preempted thread and scheduling the higher priority one.

In this way, the OpenMP semantics of TSPs are propagated at RTOS scheduling level, allowing smarter decisions on the preemption locations. The timing analysis will also be significantly easier, since it will be sufficient to analyze the worst-case execution requirements of each task part, knowing that such code blocks will be executed without interruptions. The timing characterization of each task part will factor in the worst-case delay related to interfering instances, assuming each task part needs to (re-)load all required data from scratch. This makes the analysis robust and tractable, without requiring the timing analyzer to consider all possible instructions as potential preemption points but characterizing only the worst-case timing parameters of each individual task part. In Chapter 5, it is described how to obtain the maximum execution time of a task part, with and without including the additional time-penalty due to interference with other applications running concurrently. These two timing estimates are added to the characterization of every task part in the TDG produced by the compiler. This new TDG annotated with timing information is called the OpenMP-TDG and serves as input to our schedulability analysis.

Still, one may further reduce the number of potential preemption points, by not invoking the OS scheduler at each TSP. For example, with a Breadth-First mapping model, a task creating additional tasks will continue executing on its thread, without leading to a (task-level) context switch. In this case, it may be better not to invoke the OS scheduler at TSPs coinciding with task-creation directives, since the original task may continue executing without any discontinuity in the local context. A smarter option can be to invoke the OS scheduler only when the runtime decides to map a different task next

(e.g., because the current one is finished, or due to a work-first strategy, or because of a taskwait directive). These TSPs are more likely to lead to a cache locality loss, reducing the additional impact due to preemptions.

That said, in order to simplify the schedulability analysis and avoid long non-preemptive regions, we decided to invoke the scheduler at each TSP. Although it may be beneficial to reduce the number of preemption points, we opted for the simplest solution that allows us to provide a proof of concept of the proposed approach. In the evaluation phase, we will then identify the impact of the preemption points to the scheduling overhead.

4.5 Overall Schedulability Analysis

We now will describe the overall schedulability analysis of systems executing within the P-SOCRATES framework. The analysis is based on the computation of the worst-case response time of RT tasks concurrently executing on a given cluster of cores. Two different analyses are presented, depending on the mapping/scheduling mechanisms supported by the framework: (i) a dynamic solution based on a global scheduler allowing a work-conserving behavior, and (ii) a fully static solution based on a partitioned scheduler and a fixed task-to-thread mapping.

4.5.1 Model Formalization

On our overall framework, an OpenMP program is composed of recurring instances of a RT task (identified with a *target* OpenMP construct), which in turn is composed of task parts. Without loss of generality, in this paragraph, we consider [18] a set $\tau = \{\tau_1, \ldots, \tau_n\}$ of n sporadic conditional parallel tasks (cp-tasks) that execute upon a platform consisting of m identical processors. Each cp-task τ_k releases a potentially infinite sequence of jobs. Each job of τ_k is separated from the next by at least T_k time-units and has a constrained relative deadline $D_k <= T_k$. Moreover, each cp-task τ_k is represented as a directed acyclic graph $G_k = (V_k, E_k)$, where $V_k = \{v_{k,1}, \ldots, v_{k,nk}\}$ is a set of nodes (or vertices) and E_k is a set of directed arcs (or edges), as shown in Figure 4.4. Each node $v_{k,j}$ represents a sequential chunk of execution (or "sub-task") and is characterized by a worst-case execution time $C_{k,j}$. Preemption and migration overhead is assumed to be integrated within the WCET values, as given by the timing analysis. Arcs represent dependencies between sub-tasks, that is, an edge $(v_{k,1}, v_{k,2})$ means that $v_{k,1}$ must complete before $v_{k,2}$ can start executing. A node with no incoming arcs is referred to as

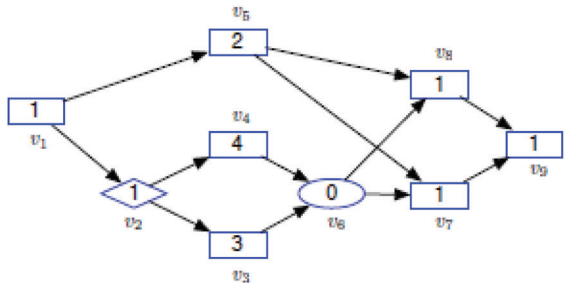

Figure 4.4 A sample cp-task. Each vertex is labeled with the WCET of the corresponding sub-task.

a source, while a node with no outgoing arcs is referred to as a sink. Without loss of generality, each cp-task is assumed to have exactly one source v_k^{source} and one sink node v_k^{sink}. If this is not the case, a dummy source/sink node with zero WCET can be added to the DAG, with arcs to/from all the source/sink nodes. The subscript k in the parameters associated with the task τ_k is omitted whenever the reference to the task is clear in the discussion.

In the cp-task model, nodes can be of two types:

1. Regular nodes, represented as rectangles, allow all successor nodes to be executed in parallel;
2. Conditional nodes, coming in pairs and denoted by diamonds and circles, represent the beginning and the end of a conditional construct, respectively, and require the execution of exactly one node among the successors of the start node.

Please note that this is a general solution for scheduling parallel recurring RT-Dags. In the specific domain of this project, where OpenMP is used as a frontend to specify DAGS, it may occur that the compiler cannot fully extract the DAG because there are conditionals that cannot be statically solved. See Section 3.4.3.2, "Missing information of the DAG", in Chapter 3, for a discussion about this issue.

To properly model the possible execution flows, a further restriction is imposed to the connections within a conditional branch. That is, a node belonging to a branch of a conditional statement cannot be connected to nodes outside that branch (including other branches of the same statement). This is formally stated in the following definition.

Definition 4.1. Let (v_1, v_2) be a pair of conditional nodes in a DAG $G_k = (V_k, E_k)$. The pair (v_1, v_2) is a conditional pair if the following holds:

1. If there are exactly q outgoing arcs from v_1 to nodes s_1, s_2, ..., s_q, for some $q > 1$, then there are exactly q incoming arcs into v_2 in E_k, from some nodes t_1, t_2,..., t_q.
2. For each $l \in \{1, 2, \ldots, q\}$, let V_l'E_l' denote all the nodes and arcs on paths reachable from sl that do not include node v_2. By definition, s_l is the sole source node of the DAG Gl':= (V_l'E_l'). It must hold that tl is the sole sink node of G_l'.
3. It must hold that V_l' and V_j' have a null intersection, for all $l \neq j$. Additionally, with the exception of (v_1, s_l) there should be no arcs in E_k into nodes in V_l' from nodes not in V_l', for each l in $\{1, 2, \ldots, q\}$.

A chain or path of a cp-task τ_k is a sequence of nodes $\lambda = (v_{k,a}, \ldots, v_{k,b})$ such that $(v_{k,j}, v_{k,j+1}) \in E_k$, for all $j \in [a,b]$. The length of a chain of τ_k, denoted by len(λ), is the sum of the WCETs of all its nodes. The longest path of a cp-task is any source-sink path of the task that achieves the longest length.

Definition 4.2. The length of a cp-task τ_k, denoted by L_k, is the length of any longest path of τ_k.

Note that L_k also represents the minimum worst-case execution time of cp-task τ_k, that is, the time required to execute it when the number of processing units is sufficiently large (potentially infinite) to allow the task to always execute with maximum parallelism. A necessary condition for the feasibility of a cp-task τ_k is that $L_k \leq D_k$.

In the absence of conditional branches, the classical sporadic DAG task model defines the volume of the task as the worst-case execution time needed to complete it on a dedicated single-core platform. This quantity can be computed as the sum of the WCETs of all the sub-tasks, that is $\sum_{v_{k,j} \in V_k} C_{k,j}$. In the presence of conditional branches, assuming that all sub-tasks are always executed is overly pessimistic. Hence, the concept of volume of a cp-task is generalized by introducing the notion of worst-case workload.

Definition 4.3. The worst-case workload W_k of a cp-task τ_k is the maximum time needed to execute an instance of τ_k on a dedicated single-core platform, where the maximum is taken among all possible choices of conditional branches.

Section 4.5 will explain in detail how the worst-case workload of a task can be computed efficiently.

The utilization U_k of a cp-task τ_k is the ratio between its worst-case workload and its period, that is, $U_k = W_k/T_k$. For the task-set τ, its total utilization U is defined as the sum of the utilizations of all tasks. A simple necessary condition for feasibility is $U \leq m$.

Figure 4.4 illustrates a sample cp-task consisting of nine sub-tasks (nodes) $V = \{v_1,\ldots,v_9\}$ and 12 precedence constraints (arcs). The number inside each node represents its WCET. Two of the nodes, v_2 and v_6, form a conditional pair, meaning that only one sub-task between v_3 and v_4 will be executed (but never both), depending on a conditional clause. The length (longest path) of this cp-task is $L = 8$, and is given by the chain (v_1, v_2, v_4, v_6, v_7, v_9). Its volume is 14 units, while its worst-case workload must take into account that either v_3 or v_4 are executed at every task instance. Since v_4 corresponds to the branch with the largest workload, $W = 11$.

To further clarify the restrictions imposed to the graph structure, note that v_4 cannot be connected to v_5, because this would violate the correctness of conditional constructs and the semantics of the precedence relation. In fact, if they were connected and v_3 were executed, then v_5 would wait forever, since v_4 is not executed. For the same reason, no connection is possible between v_4 and v_3, as they belong to different branches of the same conditional statement.

In the following sections, we will consider the dynamic approach consisting of a best-effort mapper, coupled with a fixed priority global scheduler. RT tasks are indexed according to their priorities, being τ_1 the highest priority one. For details on the scheduling algorithm and mapping, please refer to P-SOCRATES project's Deliverable D3.3.2 [19]. To understand the following analysis, it is sufficient to observe that the adopted scheduler allows a work-conserving behavior, never idling a core whenever there is some pending workload to execute.

4.5.2 Critical Interference of cp-tasks

We now present a schedulability analysis for cp-tasks globally scheduled by any work-conserving scheduler. The analysis is based on the notion of *interference*. In the existing literature for globally scheduled sequential task systems, the interference on a task τ_k is defined as the sum of all intervals in which τ_k is ready, but cannot execute because all cores are busy executing other tasks. We modify this definition to adapt it to the parallel nature of cp-tasks, by introducing the concept of critical interference.

Given a set of cp-tasks τ and a work-conserving scheduler, we define the *critical chain* of a task as follows.

Definition 4.4. The critical chain λ_k^* of a cp-task τ_k is the chain of nodes of τ_k that leads to its worst-case response-time R_k.

The critical chain of cp-task τ_k is in principle determined by taking the sink vertex v_k^{sink} of the worst-case instance of τ_k (i.e., the job of τ_k that has the largest response-time in the worst-case scenario), and recursively pre-pending the last to complete among the predecessor nodes (whether conditional or not), until the source vertex $v_{k,1}$ has been included in the chain.

A critical node of task τ_k is a node that belongs to τ_k's critical chain. Since the response-time of a cp-task is given by the response-time of the sink vertex of the task, the sink node is always a critical node. For deriving the worst-case response-time of a task, it is then sufficient to characterize the maximum interference suffered by its critical chain.

Definition 4.5. The critical interference I_k on task τ_k is defined as the cumulative time during which some critical nodes of the worst-case instance of τ_k are ready, but do not execute because all cores are busy.

Lemma 4.1. Given a set of cp-tasks τ scheduled by any work-conserving algorithm on m identical processors, the worst-case response-time of each task τ_k is

$$R_k = \text{len}(\lambda_k^*) + I_k. \tag{4.1}$$

Proof. Let r_k be the release time of the worst-case instance of τ_k. In the scheduling window $[r_k, r_k + R_k]$, the critical chain will require $\text{len}(\lambda_k^*)$ time-units to complete. By Definition 4.5, at any time in this window in which τ_k does not suffer critical interference, some node of the critical chain is executing. Therefore $R_k - I_k = \text{len}(\lambda_k^*)$.

The difficulty in using Lemma 4.1 for schedulability analysis is that the term I_k may not be easy to compute. An established solution is to express the total interfering workload as a function of individual contributions of the interfering tasks, and then upper-bound such contributions with the worst-case workload of each interfering task τ_k.

In the following, we explain how such interfering contributions can be computed, and how they relate to each other to determine the total interfering workload.

Definition 4.6. The critical interference $I_{i,k}$ imposed by task τ_i on task τ_k is defined as the cumulative workload executed by sub-tasks of τ_k while a critical node of the worst-case instance of τ_k is ready to execute but is not executing.

Lemma 4.2. For any work-conserving algorithm, the following relation holds:

$$I_k = \frac{1}{m} \sum_{\tau_i \in \mathcal{T}} I_{i,k}. \tag{4.2}$$

Proof. By the work-conserving property of the scheduling algorithm, whenever a critical node of τ_k is interfered, all m cores are busy executing other sub-tasks. The total amount of workload executed by sub-tasks interfering with the critical chain of τ_k is then mI_k. Hence,

$$\sum_{\tau_i \in \mathcal{T}} I_{i,k} = mI_k.$$

By reordering the terms, the lemma follows.

Note that when $i = k$, the critical interference $\mathrm{I}_{k,k}$ may include the interfering contributions of non-critical subtasks of τ_k on itself, that is, the self-interference of τ_k. By combining Equations (4.1) and (4.2), the response-time of a task τ_k can be rewritten as:

$$R_k = \mathrm{len}(\lambda_k^*) + \frac{1}{m} I_{k,k} + \frac{1}{m} \sum_{\tau_i \in \mathcal{T}, i \neq k} I_{i,k}. \tag{4.3}$$

In the following, we will show how to provide upper bounds on the unknown terms of Equation (4.3) for systems adopting a global fixed-priority scheduler with preemption support.

4.5.3 Response Time Analysis

In this section, we derive an upper-bound on the worst-case response-time of each cp-task using Equation (4.3). To this aim we need to compute the interfering contributions $\mathrm{I}_{i,k}$. In the sequel, we first consider the inter-task interference $(i \neq k)$ and then the intra-task interference $(i = k)$.

4.5.3.1 Inter-task interference

We divide the contribution to the workload of an interfering task τ_I in a window of interest between carry-in, body, and carry-out jobs. The *carry-in job* is the first instance of τ_i that is part of the window of interest and has release time before and deadline within the window of interest. The *carry-out job* is the last instance of τ_i executing in the window of interest, having a deadline after the window of interest. All other instances of τ_i are named *body jobs*. For sequential task-sets, an upper-bound on the workload of an interfering task τ_i within a window of length L occurs when the first job of τ_i starts executing as late as possible (with a starting time aligned with the beginning of the window of interest) and later jobs are executed as soon as possible (see Figure 4.5).

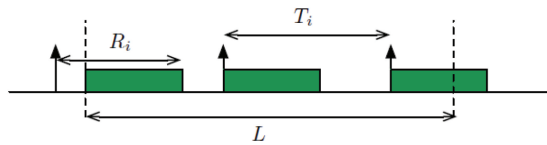

Figure 4.5 Worst-case scenario to maximize the workload of an interfering task τ_i in the sequential case.

For cp-task systems, it is more difficult to determine a configuration that maximizes the carry-in and carry-out contributions. In fact:

1. Due to the precedence constraints and different degree of parallelism of the various execution paths of a cp-task, it may happen that a larger workload is executed within the window if the interfering task is shifted left, i.e., by decreasing the carry-in and increasing the carry-out contributions. This happens for example when the first part of the carry-in job has little parallelism, while the carry-out part at the end of the window contains multiple parallel sub-tasks.
2. A sustainable schedulability analysis [10] must guarantee that all tasks meet their deadlines even when some of them execute less than the worst-case. For example, one of the sub-tasks of an execution path of a cp-task may execute for less than its WCET $C_{i,j}$. This may lead to larger interfering contributions within the window of interest (e.g., a parallel section of a carry-out job is included in the window due to an earlier completion of a preceding sequential section).
3. The carry-in and carry-out contribution of a cp-task may correspond to different conditional paths of the same task, with different levels of parallelism.

To circumvent the above issues, we consider a scenario in which each interfering job of task τ_i executes for its worst-case workload W_i, i.e., the maximum amount of workload that can be generated by a single instance of a cp-task. We defer the computation of W_i to Section 4.5.3. The next lemma provides a safe upper-bound on the workload of a task τ_i within a window of interest of length L.

Lemma 4.3. An upper-bound on the workloads of an interfering task τ_i in a window of

$$\mathcal{W}_i(L) = \left\lfloor \frac{L + R_i - W_i/m}{T_i} \right\rfloor W_i + \min(W_i, m \cdot ((L + R_i - W_i/m) \bmod T_i)).$$

length L is given by

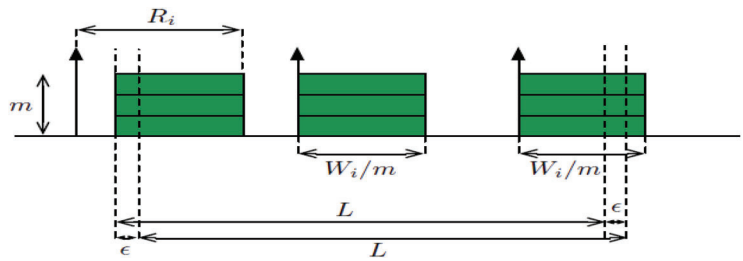

Figure 4.6 Worst-case scenario to maximize the workload of an interfering cp-task τ_i.

Proof. Consider a situation in which all instances of i execute for their worst-case workload $\mathrm{W_i}$. The highest workload within a window of length L for such a task configuration is produced when the carry-in and carry-out contributions are evenly distributed among all cores, as shown in Figure 4.6. Note that distributing the carry-in or carry-out contributions on a smaller number of cores may not increase the workload within the window. Moreover, other task configurations with a smaller workload for the carry-in or carry-out instance cannot lead to a higher workload in the window of interest: although a reduced carry-in workload may allow including a larger part of the carry-out (as in shifting right the window of interest by $\mathrm{W}_i = m$ in the figure), the carry-out part that enters the window from the right cannot be larger than the carry-in reduction.

An upper-bound on the number of carry-in and body instances that may execute within the window is

$$\left\lfloor \frac{L + R_i - W_i/m}{T_i} \right\rfloor,$$

each one contributing for $\mathrm{W_i}$. The portion of the carry-out job included in the window of interest is $(\mathrm{L} + \mathrm{R_i} - \mathrm{W_i}/m)$ *mod* $\mathrm{T_i}$. Since at most m cores may be occupied by the carryout job within that interval, and the carry-out job cannot execute for more than $\mathrm{W_i}$ units, the lemma follows.

4.5.3.2 Intra-task interference

We now consider the remaining terms of Equation (4.3), which take into account the contribution of the considered task to its overall response-time, and we compute an upper-bound on

$$Z_k \stackrel{\text{def}}{=} \text{len}(\lambda_k^*) + \frac{1}{m} I_{k,k}.$$

Lemma 4.4. For a constrained deadline cp-task system scheduled with any work-conserving algorithm, the following relation holds for any task τ_k:

$$Z_k = \text{len}(\lambda_k^*) + \frac{1}{m} I_{k,k} \leq L_k + \frac{1}{m}(W_k - L_k). \tag{4.4}$$

Proof. Since we are in a constrained deadline setting, a job will never be interfered with by other jobs of the same task. $\mathrm{W_k}$ being the maximum possible workload produced by a job of cp-task τ_k, the portion that may interfere with the critical chain λ_k is $\mathrm{W_k} - \text{len}(\lambda_k^*)$. Then, $\mathrm{I_{k,k}} \leq \mathrm{W_k} - \text{len}(\lambda_k^*)$. Hence,

$$\text{len}(\lambda_k^*) + \frac{1}{m} I_{k,k} \leq \text{len}(\lambda_k^*) + \frac{1}{m}(W_k - \text{len}(\lambda_k^*)). \tag{4.5}$$

Since $\text{len}(\lambda_k^*) \leq \mathrm{L_k}$ and $\mathrm{m} \geq 1$, the lemma follows.

Since $\mathrm{Z_k}$ includes only the contribution of task τ_k, one may think that the sum $[\text{len}(\lambda_k^*) + 1/\mathrm{m}\ \mathrm{I_{k,k}}]$ is equal to the worst-case response-time of τ_k when it is executed in isolation on the multi-core system (i.e., the makespan of τ_k).

However, this is not true. For example, consider the case of a cp-task τ_k with only one if-then-else statement; assume that when the "if" part is executed, the task executes one sub-task of length 10; otherwise, the task executes two parallel sub-tasks of length 6 each. When τ_k is executed in isolation on a two-core platform, the makespan is clearly given by the "if" branch, i.e., 10. When instead τ_k can be interfered with by one job of a task τ_i which executes a single sub-task of length 6, the worst-case response time of τ_k occurs when the "else" branch is executed, yielding a response time of 12. The share of the response time due to the term $\text{len}(\lambda_k^*) + 1/\mathrm{m}\ \mathrm{I_{k,k}}$ in Equation (4.3) is $6 + (1 = 2)6 = 9$, which is strictly smaller than the makespan. Note that $\text{len}(\lambda_k^*) + 1/\mathrm{m}\ \mathrm{I_{k,k}}$ does not even represent a valid lower bound on the makespan. This can be seen by replacing the "if" branch in the above example with a shorter subtask of length 8, giving a makespan of 8. For this reason, one cannot replace the term $\text{len}(\lambda_k^*) + 1/\mathrm{m}\ \mathrm{I_{k,k}}$ in Equation (4.4) with the makespan of τ_k.

The right-hand side of Equation (4.4) ($\mathrm{L_k} + 1/\mathrm{m}(\mathrm{W_k} - \mathrm{L_k})$) has been therefore introduced to upper-bound the term $\text{len}(\lambda_k^*) + 1/\mathrm{m}\ \mathrm{I_{k,k}}$. Interestingly, this quantity does also represent a valid upper-bound on the makespan of τ_k, so that it can be used to bound the response time of a cp-task executing in isolation. We omit the proof that is identical to the proofs of the given bounds, considering only the interference due to the task itself.

4.5.3.3 Computation of cp-task parameters

The upper-bounds on the interference given by Lemmas 4.3, 4.4, and 4.5 require the computation of two characteristic parameters for each cp-task τ_k: the worst-case workload $\mathrm{W_k}$ and the length of the longest chain $\mathrm{L_k}$. The longest path of a cp-task can be computed in exactly the same way as the longest path of a classical DAG task, since any conditional branch defines a set of possible paths in the graph. For this purpose, conditional nodes can be considered as if they were simply regular nodes. The computation can be implemented time linearly in the size of the DAG by standard techniques, see e.g., Bonifaci et al. [11] and references therein.

The computation of the worst-case workload of a cp-task is more involved. We hereafter show an algorithm to compute $\mathrm{W_k}$ for each task τ_k in time quadratic in the DAG size, whose pseudocode is shown in Algorithm 4.1.

The algorithm first computes a topological order of the DAG[2]. Then, exploiting the (reverse) topological order, a simple dynamic program can compute for each node the accumulated workload corresponding to the portion of the graph already examined. The algorithm must distinguish the case when the node under analysis is the head of a conditional pair or not.

Algorithm 4.1 Worst-case Workload Computation

```
1:  procedure WCW(G)
2:      σ ← TOPOLOGICALORDER(G)
3:      S(v^sink) ← {v^sink}
4:      for v_i ∈ σ from sink to source do
5:          if SUCC(v_i) ≠ ∅ then
6:              if ISBEGINCOND(v_i) then
7:                  v* ← argmax_{v∈SUCC(v_i)} C(S(v))
8:                  S(v_i) ← {v_i} ∪ S(v*)
9:              else
10:                 S(v_i) ← {v_i} ∪ ⋃_{v∈SUCC(v_i)} S(v)
11:             end if
12:         end if
13:     end for
14:     return C(S(v^source))
15: end procedure
```

[2] A topological order is such that if there is an arc from u to v in the DAG, then u appears before v in the topological order. A topological order can be easily computed in time linear in the size of the DAG (see any basic algorithm textbook, such as [17]).

If this is the case, then the maximum accumulated workload among the successors is selected; otherwise, the sum of the workload contributions of all successors is computed.

Algorithm 4.1 takes as input the graph representation of a cp-task G and outputs its worst-case workload W. In the algorithm, for any set of nodes S, its total WCET is denoted by C(S). First, at line 2, a topological sorting of the vertices is computed and stored in the permutation. Then, the permutation is scanned in reverse order, that is, from the (unique) sink to the (unique) source of the DAG. At each iteration of the for loop at line 4, a node v_i is analyzed; a set variable $S(v_i)$ is used to store the set of nodes achieving the worst-case workload of the subgraph including v_i and all its descendants in the DAG. Since the sink node has no successors, $S(v^{sink})$ is initialized to $\{v^{sink}\}$at line 3. Then, the function $SUCC(v_i)$ computes the set of successors of v_i. If that set is not empty, function $ISBEGINCOND(v_i)$ is invoked to determine whether v_i is the head node of a conditional pair. If so, the node v* achieving the largest value of C(S(v)), among v in $SUCC(v_i)$, is computed (line 7). The set S(v*) therefore achieves the maximum cumulative worst-case workload among the successors of v_i, and is then used to create $S(v_i)$ together with v_i. Instead, whenever v_i is not the head of a conditional pair, all its successors are executed at runtime. Therefore, the workload contributions of all its successors must be merged into $S(v_i)$ (line 10) together with v_i. The procedure returns the worst-case workload accumulated by the source vertex, that is $C(S(v^{source}))$.

The complexity of the algorithm is quadratic in the size of the input DAG. Indeed, there are $O(|E|)$ set operations performed throughout the algorithm, and some operations on a set S (namely, the ones at line 7) also require computing C(S), which has cost $O(|V|)$. So, the time complexity is $O(|V|\,|E|)$. To implement the set operations, set membership arrays are sufficient.

One may be tempted to simplify the procedure by avoiding the use of set operations, keeping track only of the cumulative worst-case workload at each node, and allowing a linear complexity in the DAG size. However, such an approach would lead to an overly pessimistic result. Consider a simple graph with a source node forking into multiple parallel branches which then converge on a common sink. The cumulative worst-case workload of each parallel path includes the contribution of the sink. If we simply sum such contributions to derive the cumulative worst-case workload of the source, the contribution of the sink would be counted multiple times. Set operations are therefore needed to avoid accounting multiple times each node contribution.

We now present refinements of Algorithm 4.1 in special sub-cases of interest.

4.5.4 Non-conditional DAG Tasks

The basic sporadic DAG task model does not explicitly account for conditional branches. Therefore, all vertices of a cp-task contribute to the worst-case workload, which is then equal to the volume of the DAG task:

$$W_k = \sum_{v_{k,j} \in V_k} C_{k,j}.$$

In this particular case, the time complexity to derive the worst-case workload of a task (quadratic in the general case), becomes O(|V|), i.e., linear in the number of vertices.

4.5.5 Series–Parallel Conditional DAG Tasks

Some programming languages yield series–parallel cp-tasks, that is, cp-tasks that can be obtained from a single edge by series composition and/or parallel composition. For example, the cp-task in Figure 4.5 is series–parallel, while the cp-tasks in Figures 4.2 and 4.6 are not. Such a structure can be detected in linear time [13]. In series–parallel graphs, for every head s_i of a conditional or parallel branch there is a corresponding tail $\mathrm{t_i}$. For example, in Figure 4.5, the tail corresponding to parallel branch head v_2 is v_9. Algorithm 4.1 can be specialized to series–parallel graphs. For each vertex u, the algorithm will simply keep track of the worst-case workload of the subgraph reachable from u, as follows. For each head vertex s_i of a parallel branch, the contribution from all successors should be added to s_i's WCET, subtracting, however, the worst-case workload of the corresponding tail t_i a number of times equal to the out-degree of s_i minus 1; for each head vertex s_i of a conditional branch, only the maximum among the successors' worst-case workloads is added to s_i's WCET. Finally, for all non-head vertices add the worst-case workload of their unique successor to their WCET. The complexity of this algorithm reduces then to O(|E|), i.e., it becomes linear in the size of the graph.

4.5.6 Schedulability Condition

Lemmas 4.3 and 4.4 and the bounds previously computed allow for proving the following theorem.

Theorem 4.1. Given a cp-task-set globally scheduled with global FP on m cores, an upper-bound R_k^{ub} on the response-time of a task τ_k can be derived by the fixed-point iteration of the following expression, starting with $R_k^{ub} = L_k$:

$$R_k^{ub} \leftarrow L_k + \frac{1}{m}(W_k - L_k) + \left\lfloor \frac{1}{m} \sum_{\forall i \neq k} \mathcal{X}_i^{ALG} \right\rfloor, \quad \mathcal{X}_i^{FP}$$

where:

$$\mathcal{X}_i^{FP} = \begin{cases} \mathcal{W}_i(R_k^{ub}), & \forall i < k \\ 0, & otherwise \end{cases};$$

because the interference from lower priority tasks can be neglected assuming a fully preemptive scheduler.

The schedulability of a cp-task system can then be simply checked using Theorem 4.1 to compute an upper-bound on the response-time of each task. In the FP case, the bounds are updated in decreasing priority order, starting from the highest priority task. In this case, it is sufficient to apply Theorem 4.1 only once for each task.

4.6 Specializing Analysis for Limited Pre-emption Global/Dynamic Approach

The response time analysis in Equation (4.3) can be easily extended [20] to incorporate the impact of the limited pre-emption strategy on DAG-based task-sets[3]. To do so, the factor that computes the inter-task interference must be augmented to incorporate the impact of lower-priority interference. Overall, the response time upper-bound can be computed as follows:

$$R_k^{ub} \leftarrow L_k + \frac{1}{m}(vol(G_k) - L_k) + \left\lfloor \frac{1}{m}(I_k^{l_p} + I_k^{h_p}) \right\rfloor$$

With LP, tasks are not only interfered with by higher-priority tasks, but also by already started lower-priority tasks whose execution has not reached a pre-emption point yet, and so cannot be suspended. In the worst-case scenario, when a high-priority task τ_k is released, all the m processors have just started executing the m largest NPRs of m different lower priority tasks. After τ_k started executing, it could be blocked again by at most $m - 1$ lower priority

[3]This section only considers LP with eager approach. In [28], we develop the analysis for Lazy approach as well. Interested readers are encouraged to refer to it for the complete analysis.

tasks at each pre-emption point. Therefore, for sequential task-sets, the lower priority interference is upper-bounded considering: (1) the set of the longest NPR of each lower-priority task and then (2) the sum of the m and $m-1$ longest NPRs of this set, as computed in [21]. This no longer holds for DAG-based task-sets, because multiple NPRs from the same task can execute in parallel. Next, we present two methods to compute the lower-priority interference in DAG-based task-sets.

4.6.1 Blocking Impact of the Largest NPRs (LP-max)

The easiest way of deriving the lower-priority interference is to account for the m and $m-1$ largest NPRs among all lower-priority tasks:

$$\Delta_k^m = \sum \max_{\tau_i \in l_p(k)}^{m} \left(\max_{1 \le j \le q_{i+1}}^{m} C_{i,j} \right)$$

$$\Delta_k^{m-1} = \sum \max_{\tau_i \in l_p(k)}^{m-1} \left(\max_{1 \le j \le q_{i+1}}^{m-1} C_{i,j} \right)$$

where $\sum max^{m}_{\tau_i \in lp(k)}$ and $\sum max^{m-1}_{\tau_i \in lp(k)}$ denote the sum of the m and $m-1$ largest values among the NPRs of all tasks $\tau_i \ \epsilon$ lp(k) respectively, while $max^{m}_{1 \le j \le q_{i+1}}$ and $max^{m+1}_{1 \le j \le q_{i+1}}$ denote the m and $m-1$ largest NPRs of a task τ_i. Despite its simplicity, this strategy is pessimistic because it considers that the largest m and $m-1$ NPRs can execute in parallel, regardless of the precedence constraints defined in the DAG.

4.6.2 Blocking Impact of the Largest Parallel NPRs (LP-ILP)

The edges in the DAG determine the maximum level of parallelism a task may exploit on m cores, which in turn determines the amount of blocking impacting over higher-priority tasks. This information must therefore be incorporated in the analysis to better upper-bound the lower-priority interference. To do so, we propose a new analysis method that incorporates the precedence constraints among NPRs, as defined by the edges in the DAG, into the LP response-time analysis. Our analysis uses the following definitions:

Definition 4.7: The LP worst-case workload of a task executing on c cores is the sum of the WCET of the c largest NPRs that can execute in parallel.

Definition 4.8. The overall LP worst-case workload of a set of tasks executing on m cores is the maximum time used for executing this set in a given execution scenario, i.e. fixing the number of cores used for each task.

Given a task τ_k, our analysis derives the lower-priority interference of lp(k) by computing new Δ_k^m and Δ_k^{m+1} factors in a three-step process:

1. Identify the LP worst-case workload of each task in lp(k) when executing on 1 to m cores;
2. Compute the overall LP worst-case workload of lp(k) for all possible execution scenarios;
3. Select the scenario that maximizes the lower-priority interference.

In order to facilitate the explanation of the three steps, the next sections consider an lp(k) composed of four DAG-tasks $\{\tau_1, \tau_2, \tau_3, \tau_4\}$ (see Figure 4.7), executed on an $m = 4$ core platform.

The nodes (NPRs) of τ_i are labeled as $v_{i,j}$ with their WCET ($C_{i,j}$) between parenthesis.

4.6.2.1 LP worst-case workload of a task executing on c cores

Given a task τ_i, this step computes an array μ_i of size m, which includes the worst-case workload of τ_i when NPRs are distributed over c cores, being

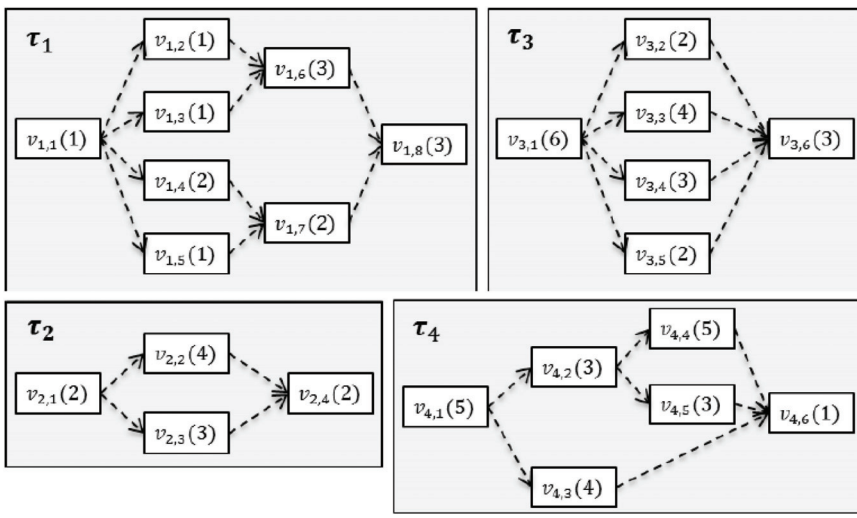

Figure 4.7 DAGs of lp(k) tasks; the $C_{i,j}$ of each node $v_{i,j}$ is presented in parenthesis.

c = {1,...,m} the index inside μ_i. Each element μ_i[c] is computed as follows:

$$\mu_i[c] = \sum \max_{c}^{parallel} \{C_{i,j}\}$$

where $max_c^{parallel}$ is the sum of the c largest NPRs of τ_i that can execute in parallel, maximizing the interference when using c cores. To this aim, the sum must consider the edges of τ_i's DAG to determine which NPRs can actually execute in parallel. Section 4.7.3 presents the algorithm that derives, for each NPR of τ_i, the set of NPRs from the same task that can potentially execute in parallel with it.

Table 4.1 shows the array μ_i for each of the tasks shown in Figure 4.7 with $m = 4$. For example, the worst-case workload μ_4 [2] occurs when NPRs $v_{4,3}$ and $v_{4,4}$ execute in parallel, with an overall impact of 9 time units. τ_2 has a maximum parallelism of 2, so μ_2 [3] and μ_2 [4] are equal to 0.

4.6.2.2 Overall LP worst-case workload

The lower-priority interference depends on how the execution of lp(k) is distributed across the m cores. We define $e^m = \{s_1,...,s_{p(m)}\}$ as the set of different execution scenarios (and so interference scenarios) of lp(k) running on m cores. p(m) is equal to the number of partitions[4] of m, and can be computed with the pentagonal number theorem from Euler's formulation:

$$\sum_q (-1)^q p\left(m - \frac{q(3q-1)}{2}\right)$$

where the sum is over all nonzero integers q (positive and negative) [22].

Table 4.1 Worst-case workloads of tasks in Figure 4.7

μ_1[c]	μ_2[c]	μ_3[c]	μ_4[c]
$C_{1,6}$ or $C_{1,8} = 3$	$C_{2,2} = 4$	$C_{3,1} = 6$	$C_{4,1}$ or $C_{4,4} = 5$
$C_{1,6} + C_{1,7} = 5$	$C_{2,2} + C_{2,3} = 7$	$C_{3,3} + C_{3,4} = 7$	$C_{4,4} + C_{4,3} = 9$
$C_{1,6} + C_{1,4} + C_{1,5} = 6$	0	$C_{3,3} + C_{3,4} + C_{3,2}$ or $C_{3,5} = 9$	$C_{4,4} + C_{4,3} + C_{4,5} = 12$
$C_{1,2} + C_{1,3} + C_{1,4} + C_{1,5} = 5$	0	$C_{3,2} + C_{3,3} + C_{3,4} + C_{3,5} = 11$	0

[4] In number theory and combinatory, a partition of a positive integer m is a way of writing m as a sum of positive integers. Two sums that differ only in the order of their summands are considered the same partition.

Table 4.2 the five possible execution scenarios assuming four cores [e_4, p(4) = 5]. The number of tasks being executed in each execution scenario s_l in e^m is given by its cardinality, i.e., $|s_l|$.

Each execution scenario s_l in e^m has an associated overall worst-case workload, computed as:

$$\rho_k[s_l] = \sum \max_{|s_l|}^{s_l} \{\mu_i\}$$

Where the right-hand side represents the sum of the $|s_l|$ largest combinations of μ_i that fits in the scenario s_l, and so maximizes the interference. Section 4.7.3 formulates the above equation as an ILP.

Table 4.3 shows the $\rho_k[s_l]$ of each execution scenario and the $\mu_i[c]$ considered in Table 4.1 and 4.2. For instance, the overall worst-case workload of s_3, $\rho_k[s3] = 19$ results when τ_4 executes on two cores ($\mu_4\ [2] = 9$), and τ_2 and τ_3 execute on one core each ($\mu_2\ [1] = 4$ and $\mu_3\ [1] = 6$).

4.6.2.3 Lower-priority interference

Finally, given the overall worst-case workload for each scenario $\mu_k[s_l]$, the lower-priority interference of lp(k) can be reformulated as the maximum overall worst-case workload among all scenarios:

$$\Delta_k^m = \max_{s_l \in e^m} \rho_k[s_l]$$

$$\Delta_k^{m-1} = \max_{s_l \in e^{m-1}} \rho_k[s_l]$$

Table 4.2 Five possible scenarios of taskset in Figure 4.7, assuming a four core system

$\mathbf{s_p} \in \mathbf{e^4}$	$\lvert\mathbf{s_p}\rvert$	**Execution scenario description**
$s_1 = \{1, 1, 1, 1\}$	4	Each task runs in 1 core
$s_2 = \{2, 2\}$	2	Each task runs in 2 cores
$s_3 = \{2, 1, 1\}$	3	1 task runs in 2 cores and 2 task in 1 cores each
$s_4 = \{3, 1\}$	2	1 task runs in 3 cores and 1 task in 1 core
$s_5 = \{4\}$	1	1 task runs in 4 cores

Table 4.3 Computed worst-case workload for each of the scenarios in Table 4.2

$\mathbf{s_l}$	$\rho_k[\mathbf{s_l}]$
s_1	$\mu_1[1] + \mu_2[1] + \mu_3[1] + \mu_4[1] = 18$
s_2	$\mu_2[2]$ or $\mu_3[2] + \mu_4[2] = 16$
s_3	$\mu_4[2] + \mu_2[1] + \mu_3[1] = 19$
s_4	$\mu_4[3] + \mu_3[1] = 18$
s_5	$\mu_3[4] = 11$

where the right-hand sides provide the maximum worst-case workload among e^m and e^{m-1} scenarios.

The lower-priority interference of lp(k) is given by the maximum $\rho_k[s_l]$, i.e., $\Delta_k^4 = 19$. On the contrary, the pessimistic approach selects the sum of the m largest NPRs among all lower-priority tasks, i.e., $\Delta_k^4 = C_{3,1} + C_{4,1} + C_{4,4} + C_{2,2} = 20$. The pessimism comes from the fact that nodes $v_{4,1}$ and $v_{4,4}$ cannot be executed in parallel. Similarly, $\Delta_k^3 = 15$, while the pessimistic approach gives $\Delta_k^3 = 16$.

Clearly, LP-ILP allows computing a tighter lower-priority interference, at the cost of increasing the complexity of deriving it, compared to the LP-max approach.

4.6.3 Computation of Response Time Factors of LP-ILP

We showed that the schedulability of a DAG-based task-set under LP-ILP can be checked in pseudo-polynomial time if, beside deadline and period, we can derive: (1) the worst-case workload generated by each lower-priority task τ_i (i.e., μ_i), and (2) the overall worst-case workload of lower-priority tasks for each execution scenario s_l in e^m (i.e., $\rho_m[s_l]$). The former can be computed at compile-time for each task, and it is independent from the task-set; the latter requires the complete task-set knowledge, and is computed at system integration time. In this section, we present the algorithms to compute these factors.

4.6.3.1 Worst-case workload of τ_i executing on c cores: μ_i[c]

$\mu_i[c]$ is determined by the set of c NPRs of τ_i that can potentially execute in parallel. As a first step, we identify for each NPR the set of potential parallel NPRs; then, we compute the interference of parallel execution when different numbers of cores are used.

(1) Computing the set of parallel NPRs: Given the DAG $G_i = (V_i, E_i)$, Algorithm 4.2 computes, for each NPR $v_{i,j}$ in V_i, the set of NPRs that can execute in parallel with it.

The algorithm takes as input the DAG of task τ_i, the topological order of G_i, and, for each node $v_{i,j}$, the sets:

1. SIBLING($v_{i,j}$), which contains the nodes which have a common predecessor with $v_{i,j}$;
2. SUCC($v_{i,j}$), which contains the nodes reachable from $v_{i,j}$; and

Algorithm 4.2 Parallel NPRs of $\boldsymbol{\tau}_i$

Input: *(1)* $G_i = (V_i, E_i)$; *(2)* TOPOLOGICAL-ORDER(G_i); *(3)* SIBLING $(v_{i,j})$, SUCC $(v_{i,j})$, PRED$(v_{i,j})$ $\forall v_{i,j} \in V_i$
Output: *Par*$(v_{i,j})$, $\forall v_{i,j} \in V_i$

```
1:  procedure PARALLEL-NPR
2:      for each v_{i,j} ∈ V_i do
3:          Par(v_{i,j}) ← ∅
4:          for each v_{i,l} ∉ SIBLING (v_{i,j}) do
5:              if (v_{i,j}, v_{i,l}) ∉ E_i and (v_{i,l}, v_{i,j}) ∉ E_i then
6:                  Succ ← SUCC (v_{i,l})\SUCC(v_{i,j})
7:                  Par(v_{i,j}) ← Par(v_{i,j}) ∪ {v_{i,l}} ∪ Succ
8:              end if
9:          end for
10:     end for
11:     for each v_{i,j} ∈ TOPOLOGICAL-ORDER(G_i) do
12:         for each v_{i,l} ∈ PRED(v_{i,j}) do
13:             Pred ← Par(v_{i,l})\ PRED(v_{i,j})
14:             Par(v_{i,j}) ← Par(v_{i,j}) ∪ Pred
15:         end for
16:     end for
17: end procedure
```

3. PRED($\mathrm{v_{i,j}}$), which contains the nodes from which $\mathrm{v_{i,j}}$ can be reached. It outputs, for each $\mathrm{v_{i,j}}$, the set Par($\mathrm{v_{i,j}}$), containing the nodes that can execute in parallel with it.

The algorithm iterates twice over all nodes in $\mathrm{V_i}$. The first loop (lines 2–10) adds to Par($\mathrm{v_{i,j}}$) (line 7) the set of sibling nodes $\mathrm{v_{i,l}}$ that are not connected to $\mathrm{v_{i,j}}$ by an edge (line 5), and the nodes reachable from $\mathrm{v_{i,l}}$ [SUCC($\mathrm{v_{i,l}}$)], discarding those connected to $\mathrm{v_{i,j}}$ by an edge (line 6). The second loop (lines 11–15), which traverses $\mathrm{V_i}$ in topological order, adds to Par($\mathrm{v_{i,j}}$) (line 14) the set of nodes Par($\mathrm{v_{i,l}}$) computed at line 7, being $\mathrm{v_{i,l}}$ a node from which $\mathrm{v_{i,j}}$ can be reached [$\mathrm{v_{i,l}}$ in PRED($\mathrm{v_{i,j}}$)]. From Par($\mathrm{v_{i,l}}$) we discard the nodes from which $\mathrm{v_{i,j}}$ can be reached (line 13).

As an example, consider node $\mathrm{v_{1,3}}$ of τ_1 in Figure 4.7. The first loop iterates over the sibling nodes $\mathrm{v_{1,2}}$, $\mathrm{v_{1,4}}$, and $\mathrm{v_{1,5}}$. None of them is connected to $\mathrm{v_{1,3}}$ by an edge (lines 4 and 5); also, SUCC($\mathrm{v_{1,2}}$) = {$\mathrm{v_{1,6}}$,$\mathrm{v_{1,8}}$}, SUCC($\mathrm{v_{1,4}}$) = {$\mathrm{v_{1,7}}$,$\mathrm{v_{1,8}}$}, and SUCC($\mathrm{v_{1,5}}$) = {$\mathrm{v_{1,7}}$,$\mathrm{v_{1,8}}$}. The algorithm discards from SUCC($\mathrm{v_{1,2}}$) nodes {$\mathrm{v_{1,6}}$,$\mathrm{v_{1,8}}$}, since they are already included in SUCC($\mathrm{v_{1,3}}$) (line 6). This is not the case of $\mathrm{v_{1,7}}$ in SUCC($\mathrm{v_{1,4}}$) and SUCC($\mathrm{v_{1,5}}$). Hence, we obtain Par($\mathrm{v_{1,3}}$) = {$\mathrm{v_{1,2}}$,$\mathrm{v_{1,4}}$,$\mathrm{v_{1,5}}$,$\mathrm{v_{1,7}}$}. The second

loop does not add new nodes to Par($v_{1,3}$) because the unique node from which $v_{1,3}$ can be reached is $v_{1,1}$, and Par($v_{1,1}$) is empty. When the second loop examines node $v_{1,7}$, the two sets Par($v_{1,4}$) and Par($v_{1,5}$) are considered, since $v_{1,4}$,$v_{1,5}$ in PRED($v_{1,7}$). Then, nodes $v_{1,2}$, $v_{1,3}$, and $v_{1,6}$ are included in Par($v_{1,7}$), since none of them belongs to PRED($v_{1,7}$).

(2) Impact of parallel NPRs on c cores: For any task τ_i, we present an ILP formulation to compute $\mu_i[c]$, i.e., the sum of the c largest NPRs in V_i that, when executed in parallel, generate the worst-case workload.

Parameters: (1) c, i.e., the maximum number of cores used by τ_i; (2) $v_{i,j}$ in V_i; (3) q_{i+1}, i.e., the number of NPRs; (4) $C_{i,j}$; and (5) $IsPar_{i,j,k}$ in {0,1}, i.e., a binary variable that takes 1 if $v_{i,j}$ and $v_{i,k}$ can execute in parallel, 0 otherwise.

Problem variables: (1) b_j in {0,1}, i.e., a binary variable that takes the value 1 if $v_{i,j}$ is one of the selected parallel NPRs, 0 otherwise, and (2) $b_{j,k} = b_j$ *OR* b_k with $b_{j,k}$ in {0,1}; $j \neq k$, i.e., an auxiliary binary variable.

Constraints:

1. $\sum_{j=1}^{q_{i+1}} b_j = c$, i.e., only c NPRs can be selected;
2. $\sum_{j=1}^{q_{i+1}} \sum_{k=j+1}^{q_{k+1}} b_{j,k}\ IsPar_{i,j,k} = c$, i.e., the selected NPRs can be executed in parallel; and
3. $b_{j,k} \geq b_j + b_k - 1$; $b_{j,k} \leq b_j$; $b_{j,k} \leq b_k$, i.e., auxiliary constraints used to model the logical *AND*.

Objective function: $\sum_{c=1}^{m} \sum_{\forall \tau_j \epsilon lp(k)} w_i^c \mu_i^c$.

4.6.3.2 Overall LP worst-case workload of lp(k) per execution scenario s_l: $\rho_k[s_l]$

Given the set lp(k) and an execution scenario s_l in e^m, we present an ILP formulation to derive $\rho_k[s_l]$, that is, the overall worst-case workload generated by lp(k) under s_l.

Parameters: (1) lp(k); (2) m; (3) s_l; and (4) $\mu_i[c]$, for all τ_i in lp(k), for all $c = 1,\ldots,m$.

Problem variable: w_i^c, i.e., a binary variable that takes the value 1 on the selected $\mu_i[c]$ that contributes to the worst-case workload, 0 otherwise.

Constraints:

1. $\sum_{c=1}^{m} \sum_{\forall \tau_j \epsilon lp(k)} w_i^c = |s_l|$, i.e., the number of tasks contributing to the worst-case workload must be equal to the size of the execution scenario;
2. For all τ_i in lp(k), $\sum_{c=1}^{m} w_i^c \leq 1$, i.e., each task can be considered at most in one scenario;
3. $\sum_{\forall \tau_j \epsilon lp(k)} w_i^c \geq 1$, c in s_l, i.e., for each number of cores considered in s_l, there exist at least one μ_i[c] that is selected;
4. $\sum_{c=1}^{m} \sum_{\forall \tau_j \epsilon lp(k)} w_i^c c = m$, the number of cores considered is m.

Objective function: $max \sum_{c=1}^{m} \sum_{\forall \tau_j \epsilon lp(k)} w_i^c \mu_i^c$.

4.6.4 Complexity

The complexity of the response time analysis is still pseudo-polynomial. We hereafter discuss the complexity of the LP-ILP analysis.

Algorithm 4.2 requires specifying for each node in V_i the sets SIBLING, SUCC and PRED, which can be computed in quadratic time in the number of nodes. Similarly, the complexity of Algorithm 4.1 is quadratic in the size of the DAG task, i.e., O($|V_k|^2$). The ILP formulation to compute μ_i[c] is performed for each task (except for the highest-priority one), and the number of cores ranges from 2 to m, hence the complexity cost is O(nm) O(ilp_A). It is important to remark that Algorithm 4.2 (as well as its inputs) and the ILP that computes μ_i[c] are executed at compile-time for each task and are independent of the task-set and the system where they execute.

$\rho_k[s_l]$ is computed for the execution scenarios e^m and e^{m-1}, and for each task τ_k (except for the lowest-priority task τ_n), hence the complexity cost is: O(n p(m)) O(ilp_B) + O(n p(m−1)) O(ilp_B). The cost of solving both ILP formulations is pseudo-polynomial, if the number of constraints is fixed [23]. Our ILP formulations have fixed constraints, with a function cost of O(ilp_A) and O(ilp_B) depending on $|V_k|$ and (m n) respectively.

Therefore, the cost of computing $\rho_k[s_l]$ for e^m dominates the cost of other operations; hence, the complexity of computing the lower priority interference is pseudo-polynomial in the number of tasks and execution scenarios, i.e., cores.

4.7 Specializing Analysis for the Partitioned/Static Approach

The use of dynamic schedulers in certain high-criticality real-time systems may be problematic. In the automotive domain, for example, the static allocation of system components (named runnables in the AUTOSAR nomenclature) define a valid application configuration, for which the application is tested and validated. This configuration defines a specific data-flow, i.e., an order in which components process data, and an end-to-end latency between sensors and actuators, e.g., the gas pedal (sensor) and the injection (actuator). A dynamic allocation instead generates different data-flows and sensor-actuator latencies that may result in invalid configurations. The use of static allocation is therefore of paramount importance for these types of systems to guarantee the correct functionality.

In this section[5], a static allocation of parallel applications is proposed based on the OpenMP4 tasking model, in order to comply with the restrictive predictability requirements of safety-critical domains. An optimal task-to-thread mapping is derived based on an ILP formulation, providing the best possible response time for a given parallel task graph.

Two different formulations are proposed to optimally deal with both the tied and untied tasking models. Then, different heuristics are proposed for an efficient (although sub-optimal) task-to-thread mapping, with a reduced complexity. Experiments on randomly generated workloads and a real case-study are provided to characterize the worst-case response time of the proposed mapping strategies for each tasking model. The results show a significant reduction in the worst-case makespan with respect to existing dynamic mapping methods, taking a further step towards the adoption of OpenMP in real-time systems for an efficient exploitation of future embedded many-core systems.

4.7.1 ILP Formulation

This section proposes an Integer Linear Programming (ILP) formulation to solve the problem of optimally allocating OpenMP tasks to threads. The problem is to determine the minimum time interval needed to execute a given OpenMP application on m threads, both in the case of tied and untied tasks. In other words, we seek to derive the optimal mapping of task (or task parts) to threads so that the task-set makespan is minimized.

[5]This section was published as a conference paper at AspDAC [30].

The system model is the same as in the previous sections, with the following modifications needed to account for the OpenMP task semantics. An OpenMP application is modeled as an OpenMP-DAG G composed of N tasks $\tau_1,\ldots,\tau_N$. Each task τ_i is composed of n_i parts $P_{i,1},\ldots,P_{i,ni}$. The Worst-Case Execution Time (WCET) of part $P_{i,j}$ of task τ_i is denoted as $C_{i,j}$. The total number of threads where tasks can be executed on a multi-core platform is denoted as m.

4.7.1.1 Tied tasks

The optimal allocation problem for tied tasks is modeled by starting from the set of tasks $\tau_1,\ldots,\tau_N$ and by adding a sink task τ_{N+1} with a single task part having null WCET (i.e., $C_{N+1,1} = 0$) and with incoming edges from the task parts without any successors in the original OpenMP-DAG.

The starting time of τ_{N+1} corresponds to the minimum completion time of the considered application; hence it represents our minimization objective.

Input parameters: (1) m: number of threads available for execution; (2) N: number of tasks in the system; (3) $C_{i,j}$: WCET of the j-th part of task τ_i; (4) G = (V, E): DAG representing the structure of the OpenMP application; (5) D: relative deadline of the OpenMP-DAG; (6) $succ_{i,j}$: set of immediate successors of part $P_{i,j}$ of τ_i; (7) rel_i: set of tasks having a relative relationship with τ_i (either as antecedents or descendants).

Problem variables: (1) $X_{i,k}$ in {0,1}: binary variable that is 1 if task τ_i is executed by thread k, 0 otherwise; (2) $Y_{i,j,k}$ in {0,1}: binary variable that is 1 if the j-th part of task τ_i is executed by thread k, 0 otherwise; (3) $\psi_{i,j}$: integer variable that represents the starting time of part $P_{i,j}$ of task τ_i (i.e., its initial offset in the optimal schedule); (4) $a_{i,j,w,z,k}$, $b_{i,w,k}$ in {0,1}: auxiliary binary variables.

Objective function: The objective function aims to minimize the starting time of the dummy sink task τ_{N+1}: min $\psi_{N+1,1}$ and represents the minimum makespan. A scheduling can be declared feasible if the minimum makespan is $\psi_{N+1,1} \leq D$.

Initial Assumptions: (i) The first part of the first task must begin at time $t = 0$: $\psi_{1,1} = 0$; (ii) The first task is executed by thread 1:

$$\begin{gathered} X_{1,1} = 1 \\ X_{1,k} = 0 \quad \forall k \in \{2,\ldots,m\} \\ Y_{1,j,1} = 1 \quad \forall j \in \{1,\ldots,n_1\} \\ Y_{1,j,m} = 0 \quad \forall j \in \{1,\ldots,n_1\}, \forall k \in \{2,\ldots,m\} \end{gathered}$$

Constraints

1. Each task is executed by only one thread:

$$\sum_{k=1}^{m} X_{i,k} = 1 \quad \forall_i \in \{1, \ldots, N\}$$

   This constraint enforces the tied scheduling clause, i.e., for each task τ_i, only one binary variable $X_{\mathrm{i,k}}$ is set to 1 among the m variables referring to the available threads.
2. All parts of each task are allocated to the same thread:

$$n_i \cdot X_{i,k} = \sum_{j=1}^{n_i} Y_{i,j,k} \quad \forall_i \in \{1, \ldots, N\}, \forall_k \in \{1, \ldots, m\}$$

   This constraint establishes the correspondence between the $X_{\mathrm{i,k}}$ and $Y_{\mathrm{i,j,k}}$ variables.
3. All precedence requirements between task parts must be fulfilled:

$$\forall_i, \omega \in \{1, \ldots, N+1\}, \forall_j \in \{1, \ldots, n_i\},$$
$$\forall_z \in \{1, \ldots, n_w\} \,|\, P_{\omega,z} \in succ_{i,j},$$
$$\psi_{i,j} + C_{i,j} \leq \psi_{w,z}.$$

   For each pair of task parts, if a precedence constraint connects them, then the latter cannot start until the former has completed execution. Notice that this constraint also applies to the sink task $\tau_{\mathrm{N+1}}$.
4. The execution of different task parts must be non-overlapping:

$$\forall_i, \omega \in \{1, \ldots, N\}, \forall_j \in \{1, \ldots, n_i\}, \forall_z \in \{1, \ldots, n_w\},$$
$$\forall_k \in \{1, \ldots, m\} \,|\, (\omega \neq i) \vee (j \neq z),$$
$$(Y_{i,j,k} = 1 \wedge Y_{w,z,k} = 1) \Rightarrow$$
$$(\psi_{i,j} + C_{i,j} \leq \psi_{w,z} \vee \psi_{w,z} + C_{w,z} \leq \psi_{i,j})$$

   In other terms, if two task parts are allocated to the same thread, then either one finishes before the other begins, or vice versa. This constraint can be written as:

$$\forall_i, \omega \in \{1, \ldots, N\}, \forall_j \in \{1, \ldots, n_i\}, \forall_z \in \{1, \ldots, n_w\},$$
$$\forall_k \in \{1, \ldots, m\} | (\omega \neq i) \vee (j \neq z),$$
$$\psi_{i,j} + C_{i,j} \leq \psi_{w,z} + M(2 + a_{a,j,w,z,k} - Y_{i,j,k} - Y_{w,z,k})$$
$$\psi_{w,z} + C_{w,z} \leq \psi_{i,j} + M(3 - a_{a,j,w,z,k} - Y_{i,j,k} - Y_{w,z,k})$$

where M is an arbitrarily large constant. Indeed, if $a_{\mathrm{i,j,w,z,k}} = 1$, then the first inequality is always inactive, while the second one is active only if $Y_{\mathrm{i,j,k}} = 1$ and $Y_{\mathrm{w,z,k}} = 1$. Similarly, if $a_{\mathrm{i,j,w,z,k}} = 0$, then the first inequality is active only if $Y_{\mathrm{i,j,k}} = 1$ and $Y_{\mathrm{w,z,k}} = 1$, while the second one is always inactive.

5. The Task Scheduling Constraint 2 (TSC 2) as described in Chapter 3 must be satisfied:

$$\forall_{i,\omega} \in \{1, \ldots, N\}, i \neq w, T_w \notin rel_i, \forall k \in \{1, \ldots, m\},$$
$$(X_{i,k} = 1 \wedge X_{w,z} = 1) \Rightarrow$$
$$(\psi_{i,n_i} + C_{i,n_i} \leq \psi_{w,1}) \vee (\psi_{w,n_w} + C_{w,n_w} \leq \psi_{i,1}).$$

This constraint imposes that one task cannot be allocated to a thread where another task that is neither a descendant nor an antecedent of the considered task is suspended. This is equivalent to saying that if two tasks not related by any descendance relationship are allocated to the same thread, then one of them must have finished before the other one begins. Therefore, the last task part of either task plus its WCET must be smaller than or equal to the starting time of the first task part of the other one. As for constraint (iv), it can be rewritten as:

$$\forall_{i,\omega} \in \{1, \ldots, N\}, i \neq w, T_w \notin rel_i, \forall k \in \{1, \ldots, m\},$$
$$\psi_{i,n_i} + C_{i,n_i} \leq \psi_{w,1} + M(2 + b_{i,w,k} - X_{i,k} - X_{w,k})$$
$$\psi_{w,n_w} + C_{w,n_w} \leq \psi_{i,1} + M(3 - b_{i,w,k} - X_{i,k} - X_{w,k}).$$

Note that all constraints [except constraint (iii)] need not be applied to $\tau_{\mathrm{N+1}}$.

4.7.1.2 Untied tasks

The ILP formulation proposed for tied tasks can be applied for untied tasks with the following modifications. The initial assumption (ii) is replaced as follows: $Y_{1,1,1} = 1$.

Since different parts of the same task are allowed to be executed by different threads, constraints (i) and (ii) are replaced by:

$$\sum_{k=1}^{m} Y_{i,j,k} = 1 \forall i \in \{1, \ldots, N\}, \forall j \in \{1, \ldots, n_i\}$$

and the variables $X_{\mathrm{i,k}}$ are no longer needed. Finally, constraint (v) does not apply for untied tasks and thus the auxiliary variables $b_{\mathrm{i,w,k}}$ are not needed.

4.7.1.3 Complexity

The problem of determining the optimal allocation strategy of an OpenMP-DAG composed of untied tasks has a direct correspondence with the makespan minimization problem of a set of precedence-constrained jobs (task parts in our case) on identical processors (threads in a team in our case). This problem, also known as job-shop scheduling, has been proven to be strongly NP-hard by a result of Lenstra and Rinnooy Kan [18]. The complexity of the problem for the tied tasks cannot be smaller than in the untied case. Indeed, when each task has a single task part, the problem for tied tasks reduces to that for untied tasks.

In the presented ILP formulations for both the tied and untied tasks, the number of variables and the number of constraints grow as $O(\mathrm{N}^2 p^2 m)$, where $p = \max_{\mathrm{i}=1,\ldots,\mathrm{N}} \quad n_{\mathrm{i}}$.

Given the problem complexity and poor scalability of the ILP formulation, the next section proposes an efficient heuristic for providing sub-optimal solutions within a reasonable amount of time.

4.7.2 Heuristic Approaches

In the context of production scheduling, several heuristic strategies have been proposed to solve the makespan minimization problem of precedence constrained jobs on parallel machines [20, 24]. More specifically, different priority rules have been proposed in the literature to sort a collection of jobs subject to arbitrary precedence constraints on parallel machines. Such ordering rules allow selecting the next job to be executed in the set of ready jobs.

The ordering rules that have been shown to perform well in the context of parallel machine scheduling are [20, 24]:

1. **Longest Processing Time (LPT)**: The job with the longest WCET is selected;

2. **Shortest Processing Time (SPT)**: The job with the shortest WCET is selected;
3. **Largest Number of Successors in the Next Level (LNSNL)**: The job with the largest number of immediate successors is selected;
4. **Largest Number of Successors (LNS)**: The job with the largest number of successors overall is selected;
5. **Largest Remaining Workload (LRW)**: The job with the largest workload to be executed by its successors is selected.

We build upon such results to make them applicable to the considered problem. At any time instant, the set of ready jobs of a given instance of an OpenMP-DAG corresponds to the set of task parts that have not completed execution and whose precedence constraints are fulfilled.

This section presents an algorithm for allocating tied and untied task parts on the different threads following one of the above-mentioned ordering criteria, such that the partial ordering between task parts is respected.

4.7.2.1 Tied tasks

Algorithm 4.3 instantiates the procedure for the case of tied tasks, for which existing heuristic strategies cannot be directly applied. The algorithm takes the structure G of an OpenMP-DAG and the number of available threads m as inputs, and it outputs a heuristic allocation of tied OpenMP tasks to threads.

The idea behind the algorithm is to allocate ready task parts to the first available thread, following a pre-determined criterion to choose among ready tasks, while enforcing the specific semantics of the OpenMP tasking model. First, a list R of ready task parts is initialized with $P_{1,1}$, and an array L of size m with null initial values is used to store the last idle time on each thread (lines 2–3). The while loop at lines 4–25 iterates until all task parts have been allocated, i.e., until the size of list A, which contains the allocated jobs, reaches the total number of parts in the task-set. At each iteration, a new task part is allocated to one of the threads. Specifically, at line 5, the index k of the earliest available thread is determined by function FirstIdleThread. Then, the procedure NextReadyJob returns the ready task part $P_{\mathrm{i,j}}$ selected according to one of the ordering rules described above. The allocation of the selected task part must always respect TSC 2. Hence, any time the first part of a new task is selected, the function must check its descendance relationships with the tasks currently suspended on thread k, stored in the list S_{k}. If $P_{\mathrm{i,j}}$ is the first part of τ_i (line 7), then it is allocated on core k; otherwise, it is

Algorithm 4.3 Heuristic allocation of an OpenMP application comprising tied tasks

```
1: procedure HEURTIED(G, m)
2:     A ← ∅; R ← P_{1,1}
3:     L ← ARRAY (m, 0) : S ← ARRAY(m, ∅)
4:   while SIZE(A)! = Σ_{i=1}^{N} n_i do
5:     k ← FIRSTIDLETHREAD(L)
6:     P_{i,j} ← NEXTREADYJOB(k, R, S_k, G)
7:     if j == 1 then
8:       θ_i ← k
9:       if j! = n_i then
10:            S_k ← APPEND(i, S_k)
11:      end if
12:    else if j == n_i then
13:      S_k ← REMOVE (i, S_k)
14:    end if
15:    ψ_{i,j} = max(L_{θ_i}, ψ_{i,j}); L_{θ_i}, ← L_{θ_i} + C_{i,j}
16:    A ← APPEND (P_{i,j}, A); R ← REMOVE (P_{i,j}, R)
17:    for P_{k,z}|(P_{i,j}, P_{k,z}) ∈ E do
18:      if ψ_{k,z} < ψ_{i,j} + C_{i,j} then
19:        ψ_{k,z} ← ψ_{i,j} + C_{i,j}; F_{k,z} = F_{k,z} + 1
20:        if F_{k,z} == SIZE(INEDGES_{k,z}) then
21:          R ← APPEND (P_{k,z}, R)
22:        end if
23:      end if
24:    end for
25:  end while
26:  return max_{i=1}^{m} L_i
27: end procedure
```

allocated on thread θ_i, according to the tied scheduling clause. Also, if that task part is not the final one (line 9), τ_i is appended to the list of tasks currently suspended on thread k. Otherwise, if $P_{i,j}$ is the final part of τ_i (line 12), τ_i can be removed from the list of tasks currently suspended on thread k. In both cases, the starting time of $P_{i,j}$ is updated, as well as the last idle time on thread k (line 15). In addition, $P_{i,j}$ is added to the list of allocated jobs and removed from the list of ready jobs (line 16). Once $P_{i,j}$ has been allocated, other jobs may become ready. All the successors of $P_{i,j}$ are scanned and an internal counter ($F_{k,z}$) is incremented for each vertex (for loop at lines 17–24). Once the counter reaches the number of its immediate predecessors, the task part may be appended to the list of ready vertices (line 21). Finally, the makespan corresponding to the generated allocation is returned. At the end of

the algorithm, $\psi_{\mathrm{i,j}}$ stores the starting time of any part $P_{\mathrm{i,j}}$ in the final schedule, and θ stores the mapping of tasks to threads.

The algorithm runs in polynomial time in the size of the task-set; specifically, the time complexity is $\mathrm{O}\left(\left(\sum_{i=1}^{N} n_i\right)^2\right)$.

4.7.2.2 Untied tasks

Algorithm 4.3 can be applied also in the case of untied tasks with some simplifications. In particular, the function NextReadyJob does not need to check the validity of TSC 2. Hence, the array S is not required, and all the operations on S at lines 7–14 do not need to be performed. On the other hand, the algorithm must keep track of the thread associated to each task part (instead of each task).

4.7.3 Integrating Interference from Additional RT Tasks

We now generalize the static setting by considering a set of n OpenMP applications modeled as a collection of OpenMP DAGs $\Gamma = \{\mathrm{G}_1,\ldots,\mathrm{G}_\mathrm{n}\}$. Each DAG is released sporadically (or periodically) and has a relative deadline $\mathrm{D_i}$, which is constrained to be smaller than or equal to its corresponding period (or inter-arrival time) T_i.

We assume that parts of each tasks are statically partitioned to the m available threads. At any time instant, the scheduler selects among the ready task parts the one that should be executed by a given thread according to partitioned fixed-priority preemptive scheduling. In addition, we assume that OpenMP applications are statically prioritized, i.e., each DAG $\mathrm{G_i}$ is associated with a unique (fixed) priority that is used by the scheduler to select which task parts should be executed at any time instant by any of the threads.

In order to compute an upper-bound on the response time R_i of a given OpenMP-DAG G_i, we proceed by computing an upper-bound on the response time of each task part in the OpenMP-DAG, following a predefined order dictated by any topological sorting of the DAG. At each step, the response time of the considered vertex is computed considering all its immediate predecessors, one at a time. A safe upper-bound on the response time of the vertex under analysis will be selected as the maximum of such values. The maximum response time among vertices without successors will be selected as upper-bound to the response time of the DAG-task G_i.

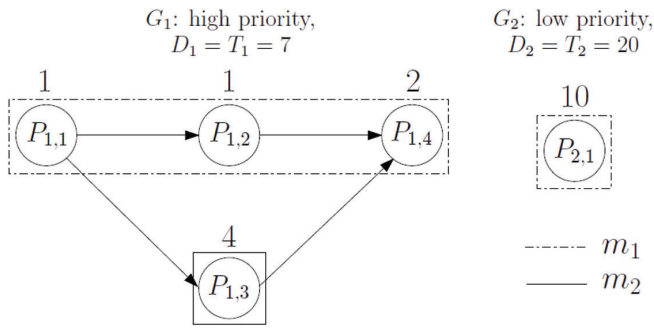

Figure 4.8 Tasks example.

4.7.4 Critical Instant

We hereafter prove that the synchronous periodic arrival pattern does not represent the worst-case release sequence for the OpenMP-DAG task model assumed. Consider a task-set composed of two OpenMP-DAG tasks G_1 and G_2, whose structure and parameters are illustrated in Figure 4.8. The figure also reports the static allocation of task parts to threads: parts $P_{1,1}$, $P_{1,2}$, $P_{1,4}$, and $P_{2,1}$ are allocated to thread m_1, while part $P_{1,3}$ is allocated to thread m_2.

We can immediately see that $R_1 = 7$, as G_1 is the highest priority RT task in the system. In order to compute the response time of G_2, we focus on thread m_1 and first consider the synchronous periodic arrival pattern for G_1, which produces the schedule in Figure 4.9a and yields a response time of 21 time units for G_2. However, if we consider the release pattern in Figure 4.9b, where the release of G_1 has an offset of two time units, we observe that the response time of G_2 becomes equal to 23.

This example shows that it is very difficult to exactly quantify the interference a task may suffer from higher-priority tasks in the worst-case. This is mainly due to the precedence constraints between parts of the same tasks, and to the fact that any vertex is allowed to execute on its corresponding thread only when all its predecessors (possibly allocated to different threads) have completed their execution. In order to overcome these problems, we derive a safe upper-bound on the response time of a given task by considering the densest possible packing of jobs generated by a legal schedule in any time interval. Specifically, we consider a pessimistic scenario (see Figure 4.9c):

- the first instance of a higher-priority task is released as late as possible;
- subsequent instances are released as soon as possible;
- higher-priority jobs are considered as if precedence constraints were removed (their WCET is "compacted").

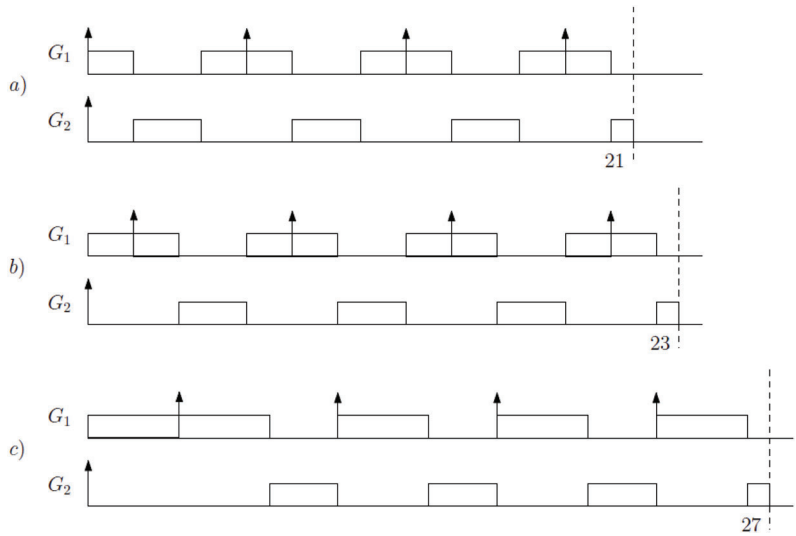

Figure 4.9 Different release patterns for the example of Figure 4.8. (a) represents the most optimistic case, while (c) the most pessimistic, i.e., yelding to the highest WCET. (b) represents an intermediate case.

4.7.5 Response-time Upper Bound

Algorithm 4.4 computes an upper-bound on the response time of an OpenMP-DAG by considering the above-described pessimistic scenario leading to the densest possible packing of higher-priority task parts:

The function SELFINTERFERENCE calculates the self-interference suffered by task part $P_{k,i}$ as the sum of the WCETs of all parts $P_{k,j}$ belonging to the same task and such that:

1. they are allocated to the same thread as $P_{k,i}$;
2. there is no path starting at $P_{k,i}$ that can reach $P_{k,j}$;
3. there is no path starting at $P_{k,j}$ that can reach $P_{k,i}$.

With the above algorithm in place, different heuristics can be proposed to find a feasible allocation of task parts to threads/cores. Among the ones we tried, we found that the best schedulability performances are obtained with a Best Fit approach that works as follows:

- It assigns RT tasks in non-increasing priority order, i.e., starting from the highest priority task and moving towards lower priority ones.
- For each task it defines a topological order for all task parts.

Algorithm 4.4 Upper-bound on the response time of an OpenMP-DAG by considering the densest possible packing of higher-priority task parts

```
1: procedure DENSESTPACKINGALG(G_k)
2:     σ → TOPOLOGICALORDER(G_k)
3:     for P_{k,i} ∈ σ from source to sink do
4:         R_max = max_{j∈PRED(k,i)} R_{k,j}
5:         S = SELFINTERFERENCE(k, i)
6:         R ← C_{k,i} + S
7:         R_prev ← 0
8:         while R ≠ R_prev do
9:             R_prev ← R
10:            R ← C_{k,i} + S
11:            for P_{h,j} such that h < k and θ_{h,j} == θ_{k,i} do
12:                R ← R + ⌈(R_prev + R_{h,j} − C_{h,j}) / T_h⌉ C_{h,j}
13:            end for
14:        end while
15:        If R_max + R > D_k then
16:            sched ← 0
17:            break
18:        else
19:            sched ← 1
20:            R_{k,i} ← R_max + R
21:        end if
22:    end for
23:    return {sched, R_{k,sink}}
24: end procedure
```

- Following the topological order, each task part is assigned to the core that minimizes its partial response time, i.e., the response time of the RT task until the considered task part.
- If any of the considered task parts has a partial response time that exceeds its relative deadline, the algorithm fails, declaring the RT task-set not schedulable.

The partial response time of each task part can be easily computed using Algorithm 4.4, executing the operations within the for loop at line 3. Once the selection is made for a task part, there is no need to recheck the schedulability of the parts already assigned belonging to higher priority tasks, since this last assignment does not interfere with them. However, it is necessary to reconsider the task parts belonging to the same RT task that may experience an increase in the interference. The only task parts that may be affected by the last task part assigned are those that have no precedence constraints with it. For these ones, we re-compute their partial response-time

after the new assignment. Since there is no backtracking in this case, the complexity of the heuristic remains reasonable, at the penalty of some added pessimism.

4.8 Scheduling for I/O Cores

This paragraph briefly describes the scheduler adopted at host level, i.e., in the I/O cores.

According to system requirements, the OS running on the host processor must be Linux. Moreover, the Linux kernel must be patched with the PREEMPT_RT patch[6]. This is an on-going project supported by the OSADL association[7] to add real-time performance to the Linux kernel by reducing the *maximum* latency experienced by an application, mainly through preemptible spinlocks and in-thread interrupt management (See also essential work in [25–38]). The patch makes the system more predictable and deterministic; however, it often increases the *average* latency. Currently, the patch only partially works on the reference platform due to missing support for SMP in the Linux kernel; full support will be added during the next months. Concerning the scheduling policy, the OS must provide a fixed-priority preemptive FIFO-scheduling algorithm. Therefore, the basic scheduling algorithm will be the SCHED_FIFO policy specified by the POSIX standard. The optional requirement R5.21 suggests to have a Linux kernel higher than 3.14 for investigating potential benefits given by the dynamic-priority SCHED_DEADLINE Linux scheduler. This possibility will be explored at a later stage of the project. Access to shared resources in the host cores is handled through the Priority Inheritance (PI) policy provided by the Linux kernel.

4.9 Summary

In this chapter, we described the design choices related to the implementation of a partitioned scheduler for allocating the computing resources to the different threads in the system. In particular, we detailed the thread model adopted in the project, and the local scheduler adopted at core level, based on fixed thread priorities.

[6]PREEMPT_RT Linux patch, https://rt.wiki.kernel.org

[7]OSADL, Open Source Automation Development Lab, http://www.osadl.org/

Such a scheduler has then been enhanced with the enforcement of a limited pre-emption scheduling policy that corresponds to the execution model supported by the OpenMP tasking model, as well as allowing increasing the predictability of the analysis, without sacrificing the schedulability. According to the limited pre-emption scheduling model, each thread can be pre-empted only at particular pre-emption points. The framework provides a method to compute the length of the largest non-preemptive region that can be tolerated by each thread (at each different priority). Then, threads execute along non preemptive regions. In a generic model such as the one introduced in this chapter, this means inserting the minimum possible number of preemption points such that the schedulability of higher priority thread is not affected. Of course, specifying this model so that it adheres to OpenMP semantics means that the identification of these preemption points exploits information inherited from the OpenMP task semantics, i.e., OpenMP TSPs will be used as potential candidates.

We then described the implementation of an enhanced global scheduler with migration support. Such a scheduler is integrated with the OpenMP dynamic mapping policy to allow for a work-conserving resource allocation of computing resources. The scheduler adopts a cluster-wide ready queue where threads are ordered according to their priorities. Preemptions are allowed only at task-part boundaries when a TSP is reached. TSPs are also natural polling points to deal with new incoming offloads without requiring interrupts.

The task model adopted, namely the cp-task model, generalizes the classic sporadic DAG task model by integrating conditional branches. The topological structure of a cp-task graph has been formally characterized by specifying which connections are allowed between conditional and non-conditional nodes. Then, a schedulability analysis has been derived to compute a safe upper-bound on the response-time of each task in pseudo-polynomial time. Besides its reduced complexity, the proposed analysis has the advantage of requiring only two parameters to characterize the complex structure of the conditional graph of each task: the worst-case workload and the length of the longest path. Algorithms have also been proposed to derive these parameters from the DAG structure in polynomial time. Simulation experiments carried out with randomly generated cp-task workloads and real test-cases clearly showed that the proposed approach is able to improve over previously proposed solutions for tightening the schedulability analysis of sporadic DAG task systems. The first formulation of the analysis considered a full-preemption model (see [18]). Then, it has been extended to limited

preemptive scheduling [24], and, finally, it has been specialized also for non-conditional DAGs [20, 29].

In this chapter, two methods have been proposed to compute the lower-priority interference: (1) a pessimistic but easy-to-compute method, named LP-max, which upper bounds the interference by selecting the NPRs with the longest worst-case execution time; and (2) a tighter but computationally-intensive method, named LP-ILP, which also takes into account precedence constraints among DAGs nodes in the analysis. Our results demonstrate that LP-ILP increases the accuracy of the schedulability test with respect to LP-max when considering DAG-based task-sets with different levels of parallelism.

The chapter then proposed an ILP formulation to derive an optimal static allocation compliant with the OpenMP4 tied and untied tasking model. With the objective of reducing the complexity of the ILP solver, five heuristics have been proposed for an efficient (although sub-optimal) allocation. Results obtained on both randomly generated task-sets and the 3DPP application (from the avionics domain) show a significant reduction in the worst-case makespan with respect to an existing schedulability upper-bound for untied tasks. Moreover, the proposed heuristics perform very well, closely matching the optimal solutions for small task-set, and outperforming the best feasible solution found by our ILP (after running the solver for a certain amount of time) for large task-sets and the 3DPP.

References

[1] Liu, C., Layland, J., Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM* 20, 46–61, 1973.

[2] Leung, J. Y. T., Whitehead, J., On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Perform. Eval.* 2, 237–250, 1982.

[3] Buttazzo, G., Bertogna, M., Yao, G., "Limited preemptive scheduling for real-time systems: a survey." *IEEE Transactions on Industrial Informatics*, 9, 3–15, 2013.

[4] Lehoczky, J., Sha, L., Ding, Y., "The rate monotonic scheduling algorithm: Exact characterization and average case behavior." In *Proceedings of the Real-Time Systems Symposium—1989, pp. 166–171. IEEE Computer Society Press*, Santa Monica, California, USA, 1989.

[5] Dhall, S. K., and Liu, C. L. On a real-time scheduling problem. *Operat. Res.*. 26, 127–140, 1978.

[6] Baruah, S., Cohen, N., Plaxton, G., and Varvel, D., Proportionate progress: A notion of fairness in resource allocation. *Algorithmica* 15, 600–625, 1996.

[7] Anderson, A., and Srinivasan, A., "Pfair scheduling: Beyond periodic task systems." In *Proceedings of the International Conference on Real-Time Computing Systems and Applications (Cheju Island, South Korea)*, IEEE Computer Society Press, 2000.

[8] Zhu, D., Mosse, D., and Melhem, R. G., "Multiple-resource periodic scheduling problem: how much fairness is necessary?" *24th IEEE Real-Time Systems Symposium (RTSS)* (Cancun, Mexico), 2003.

[9] Cho, H., Ravindran, B., and Jensen, E. D., "An optimal real-time scheduling algorithm for multiprocessors," *27th IEEE Real-Time Systems Symposium (RTSS)* (Rio de Janeiro, Brazil), 2006.

[10] Andersson, B., and Tovar, E., "Multiprocessor scheduling with few preemptions." In *Proceedings of the International Conference on Real-Time Computing Systems and Applications. (RTCSA)*, 2006.

[11] Funaoka, K., Kato, S., and Yamasaki, N., "Work-conserving optimal real-time scheduling on multiprocessors." In *Proceedings of the Euromicro Conference on Real-Time Systems*, 13–22, 2008.

[12] Funk, S., and Nadadur, V., "LRE-TL: An optimal multiprocessor algorithm for sporadic task sets." In *Proceedings of the Real-Time Networks and Systems Conference*, 159–168, 2009.

[13] Levin, G, Funk, S., Sadowski, C., Pye, I., Brandt, S, "DP-Fair: A Simple Model for Understanding Multiprocessor Scheduling." In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, Brussels, Belgium, pp. 1–10, 2010.

[14] Funk, S., Nelis, V., Goossens, J., Milojevic, D., Nelissen, G., and Nadadur, V., On the design of an optimal multiprocessor real-time scheduling algorithm under practical considerations (extended version). arXiv preprint arXiv:1001.4115, 2010.

[15] Nelissen, G., Su, H., Guo, Y., Zhu, D., Nelis, V., and Goossens, J., An optimal boundary fair scheduling. *Real-Time Sys. J.* 2014.

[16] Regnier, P., Lima, G., Massa, E., Levin, G., and Brandt, S., "RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," *IEEE 32nd Real-Time Systems Symposium (RTSS)*, 2011.

[17] Fisher, N., Goossens, J., and Baruah, S., Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Sys. J.* 45.1-2, 26–71, 2010.

[18] Melani, A., Bertogna, M., Bonifaci, V., Marchetti-Spaccamela, A., and Buttazzo, G., "Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems," in *27th Euromicro Conference on Real-Time Systems, ECRTS 2015*, Lund, Sweden, pp. 7–10, 2015.

[19] P-SOCRATES Deliverable 3.3.2. *Enhanced scheduler with migration support*. Delivery date: 31 March 2016.

[20] Serrano, M. A., Melani, A., Bertogna, M., and Quiñones, E., "Response-Time Analysis of DAG Tasks under Fixed Priority Scheduling with Limited Preemptions," in the *Design, Automation, and Test in Europe conference (DATE)*, Dresden, Germany, pp. 14–18, 2016.

[21] P-SOCRATES Deliverable 4.2.2. *Interference Model*. Delivery date: 31 March 2016.

[22] P-SOCRATES Deliverable 1.5.2. *Integrated Tool-chain*. Delivery date: 31 March 2016.

[23] Blumofe, R. D., and Leiserson, C. E., Scheduling multithreaded computations by work stealing. *J. ACM* 46, 720–748, 1999.

[24] Serrano, M. A., Melani, A., Kehr, S., Bertogna, M., and Quiñones, E., "An Analysis of Lazy and Eager Limited Preemption Approaches under DAG-based Global Fixed Priority Scheduling," in the *19th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC)*, Toronto, Canada, pp. 16–18, 2017.

[25] Anderson, T. E., "The performance of spin lock alternatives for shared-memory multiprocessors." In *IEEE Transactions on Parallel and Distributed Systems*, 1990.

[26] Craig, T. S., "Queuing spin lock algorithms to support timing predictability." In *Proc. Real-Time Sys. Symp*. pp. 148–157, 1993.

[27] Shen, C., Molesky, L. D., and Zlokapa, G., "Predictable synchronization mechanisms for real-time systems." In *Real-Time Systems*, 1990.

[28] Graunke, G., and Thakkar, S., "Syncronization algorithms for shared-memory multiprocessors." In *IEEE Computer*, 1990.

[29] Melani, A., Serrano, M. A., Bertogna, M., Cerutti, I., Quiñones, E., and Buttazzo, G., "A static scheduling approach to enable safety-critical OpenMP applications," in the *21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, Tokyo, Japan, pp. 16–19, 2017.

[30] P-SOCRATES Deliverable 3.1. *Resource Allocation Requirements*. Delivery date: 30 April 2014.

[31] P-SOCRATES Deliverable 5.2. *Operating Systems Support Prototypes*. Delivery date: 31 March 2015.

[32] P-SOCRATES Annex I – *Description of Work*, 2014.

[33] P-SOCRATES Deliverable 3.2. *Mapping Strategies*. Delivery date: 31 March 2015.

[34] Joseph, M., Pandya, P., Finding response times in a real-time system. *Comput. J.* 29, 390–395, 1986.

[35] Maia, C., Nogueira, L., and Pinho, L. M., "Scheduling Parallel Real-Time Tasks using a Fixed-Priority Work-Stealing Algorithm on Multiprocessors," in 8^{th} *IEEE Symposium on Industrial Embedded Systems*, Porto, Portugal, pp. 19–21, 2013.