A static scheduling approach to enable safety-critical OpenMP applications

Alessandra Melani[1], Maria A. Serrano[2,3], Marko Bertogna[4],
Isabella Cerutti[1], Eduardo Quiñones[2], Giorgio Buttazzo[1]

[1] Scuola Superiore Sant'Anna - Pisa, Italy
[2] Barcelona Supercomputing Center (BSC) - Barcelona, Spain
[3] Universitat Politècnica de Catalunya - Barcelona, Spain
[4] Università di Modena e Reggio Emilia - Modena, Italy
{alessandra.melani, i.cerutti, g.buttazzo}@sssup.it, marko.bertogna@unimore.it
{maria.serranogracia, eduardo.quinones}@bsc.es

Abstract— Parallel computation is fundamental to satisfy the performance requirements of advanced safety-critical systems. OpenMP is a good candidate to exploit the performance opportunities of parallel platforms. However, safety-critical systems are often based on static allocation strategies, whereas current OpenMP implementations are based on dynamic schedulers. This paper proposes two OpenMP-compliant static allocation approaches: an optimal but costly approach based on an ILP formulation, and a sub-optimal but tractable approach that computes a worst-case makespan bound close to the optimal one.

I. Introduction

Parallel programming models are fundamental to exploit the massively parallel computation capabilities of many-core embedded architectures (e.g., Kalray MPPA [1], TI Keystone II [2]). To that aim, current architectures already incorporate them in their software developer kits (SDKs) to provide the abstraction level required to program parallel applications, while hiding the complexity of the underlying processing platform. This paper focuses on OpenMP [3], the de-facto standard for shared memory parallel programming in high-performance computing (HPC) that is being adopted also in parallel real-time embedded systems, e.g., MPPA and Keystone II embedded architectures support OpenMP in their SDKs.

OpenMP incorporates a *tasking model* that enables very sophisticated types of fine-grained and irregular parallelism, in which the programmer may define explicit tasks and their related data dependencies. At run-time, tasks are scheduled in a *team of threads* according to the two types of tasking models, i.e., *tied* and *untied*, effectively utilizing many-core architectures. Interestingly, recent works [4, 5] have demonstrated that the structure and syntax of an OpenMP program resembles the Directed Acyclic Graph (DAG) scheduling model [6, 7, 8], enabling the time predictability of OpenMP programs. [9] further computed a worst-case response time analysis of common dynamic scheduling strategies used by OpenMP run-times for the untied model, and acknowledged the impossibility of deriving an accurate schedulability analysis for the tied model due to its non-work-conserving scheduling nature.

This work is funded by the EU projects P-SOCRATES (FP7-ICT-2013-10) and HERCULES (H2020/ICT/2015/688860), and the Spanish Ministry of Science and Innovation under contract TIN2015-65316-P.

However, despite the proven timing predictable behaviour of the untied tasking model when using current dynamic schedulers, the use of OpenMP is not allowed in certain high-criticality real-time systems that guarantee a predictable execution by binding tasks to cores in a static fashion. This is the case, for example, in the automotive domain, in which the static allocation of system components defines a valid *application configuration*, for which the application is tested and validated [10, 11]. This configuration defines a specific order in which components process data, which in turns impact on the *end-to-end latency* between sensors and actuators [12], e.g., the gas pedal (sensor) and the injection (actuator). The use of static allocation is therefore of paramount importance for these types of systems to guarantee the correct functionality by (i) statically determining where each task will execute and (ii) simplifying the certification activities reducing the run-time configurations to be considered. Moreover, the only alternative to provide timing predictability for the tied tasking model is by means of static allocation solutions, due to the non-work-conserving nature when using dynamic scheduling.

This paper poses the first step towards the adoption of OpenMP in safety-critical systems by proposing two OpenMP-compliant static allocation strategies that comply with the restrictive predictability requirements of these systems, whilst exploiting the performance opportunities brought by the latest many-core embedded architectures. The first strategy derives a computationally expensive but optimal allocation solution based on a non-trivial ILP formulation, which computes the minimum possible response time achievable for a given OpenMP application. The second strategy is based on well-known sub-optimal heuristic strategies that allow providing response times comparable to the optimal one, but within a much smaller computational complexity, and that can be adopted in the case of OpenMP applications with a very large number of tasks. Interestingly, the first strategy provides also a reference point to evaluate any other (static or dynamic) scheduling solution. Finally, this paper completes the analysis of the OpenMP tasking model, by providing a response time analysis of the tied model by means of static allocation strategies. Experiments on a real case-study and randomly generated workloads prove that our OpenMP-compliant static allocation reduces the worst-case makespan compared with existing worst-case response time

analysis of dynamic schedulers.

II. Related Work

Several parallel task models have been proposed in the literature to analyze the timing behaviour of parallel real-time applications: the fork-join [13], the synchronous parallel [14], and the DAG task model [6, 8]. Although being inspired by the most common parallel programming paradigms, none of these models and corresponding schedulability analyses is directly applicable to OpenMP. Moreover, none of these works addresses static solutions, which is the focus of this paper.

Affinities and differences between DAG scheduling and the OpenMP tasking model were first identified in [4], and the only schedulability analysis approach addressing the two execution models supported by OpenMP, i.e. tied and untied, has been developed in [9]. In particular, the work in [9] reasons about the capability of the OpenMP specification to provide precise and tight timing guarantees on the most common dynamic schedulers, i.e., *breadth-first scheduler* (BFS) and *work-first scheduler* (WFS), implemented in most OpenMP runtimes. Moreover, it computes a response time analysis for the untied model, and proves that timing guarantees can only be derived by means of static solutions due to the non-work-conserving nature of the tied scheduling model. In addition, a compiler method has been proposed in [5] to construct an augmented DAG compliant with the OpenMP semantics. This implementation enables the practical applicability of the schedulability analysis presented in [9]. As a result, in Section VI, our approach will be only evaluated in comparison with [9], which is the only other work that explicitly targets scheduling strategies for OpenMP applications.

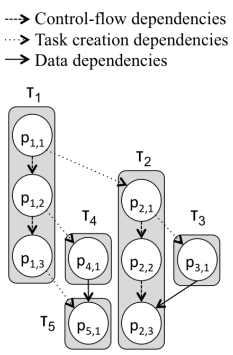
III. OpenMP Tasking Model

In OpenMP, an executing task may be suspended and the hosting thread can be rescheduled to a different task. The points in the program where this can happen are called *task scheduling points* (*TSP*), because they are associated to a scheduling decision. TSPs, which occur upon task creation/-completion, synchronization, `taskyield` and `target` directives, divide task regions into *task-parts*, which are uninterruptedly executed from start to end. As a result, task-parts become the de-facto units of any scheduling solution (either static or dynamic) of tasks to threads [4]. Moreover, OpenMP defines two tasking models: *tied* and *untied*. A task is defined as tied unless the `untied` clause is present in the task construct. Parts from a tied task can execute only in the same thread that started its execution, whereas parts from the same untied task can execute in any available thread.

Therefore, when a *TSP* is encountered, the set of *task scheduling constraints* (*TSC*) defined by the OpenMP specification must be fulfilled, depending on whether the task is *tied* or *untied*. In addition, OpenMP defines an extra TSC for tied tasks (named *TSC 2*) that limits the scheduling of new tasks to threads depending on the set of tied tasks suspended on it. If this set is empty, any new tied task may be scheduled in the considered thread. Otherwise, a new tied task may be scheduled in the considered thread only if it is a descendant task of every suspended task in the set. The descendant relationships of task τ_i are τ_i's child tasks and child's descendant tasks

```
1 #pragma omp parallel num_threads(3) {
2 #pragma omp single { // τ1
3   p1,1
4   #pragma omp task { // τ2
5     p2,1
6     #pragma omp task // τ3
7     { p3,1 }
8     p2,2
9     #pragma omp taskwait
10    p2,3
11  }
12  p1,2
13  #pragma omp task depend
14                (out:a) // τ4
15  { p4,1 }
16  p1,3
17  #pragma omp task depend
18                (in:a) // τ5
19  { p5,1 }
20 }}
```

(a) OpenMP source code.



(b) OpenMP-DAG.

Fig. 1. Example of an OpenMP program composed of tied tasks (a) and its corresponding OpenMP-DAG (b).

(similarly, antecedents tasks of τ_i are τ_i's parent task and parent's antecedents tasks). Finally, OpenMP defines the `depend` clause, which describes a list of input/output data dependencies existing among tasks. If a task has an input dependence on a variable, it cannot start executing until the set of tasks having an output dependency on the same variable complete. Dependences can only be defined among sibling tasks (first-level descendant tasks of the same parent task).

Figure 1a shows an OpenMP program composed of five tied tasks. The figure also shows the parts in which tasks are divided due to the TSPs task creation/completion of τ_2, τ_3 and τ_4, and the `taskwait` directive, e.g., τ_1 is composed of $p_{1,1}$, $p_{1,2}$ and $p_{1,3}$ (lines 3, 12 and 16).

A. System Model

There is a tight correspondence between the structure and syntax of an OpenMP program and the sporadic DAG task model [4], recently introduced in the real-time community [6, 7, 8]. The sporadic DAG model represents a parallel application by means of a DAG $G = (V, E)$, a minimum interarrival time T and a relative deadline D. Each vertex $v \in V$ denotes a sequential operation or job, while the edges represent precedence constraints between jobs, that is, if $(v_1, v_2) \in E$, then job v_1 must complete its execution before job v_2 can start executing. When a DAG task is released at time t, vertices become ready to execute as precedence constraints are fulfilled, and all jobs must finish before time $t + D$. Each DAG task instance is released with a minimum separation of T time units to the following one.

The execution of an OpenMP task-part resembles the execution of a vertex in the DAG, while OpenMP synchronization directives can be modeled as edges in the DAG. Exploiting this similarity, this paper considers an OpenMP application modeled as a (single) OpenMP-DAG G composed of N OpenMP tasks $\tau_1, \ldots, \tau_N$. Each task τ_i is composed of n_i task-parts $p_{i,1}, \ldots, p_{i,n_i}$. The Worst-Case Execution Time (WCET) of task-part $p_{i,j}$ of task τ_i is denoted as $C_{i,j}$. Figure 1b shows the corresponding OpenMP-DAG of the program shown in Figure 1a, obtained using the compiler technique presented in [5]. In

the Figure, there exist a task creation dependency among task-parts $p_{1,1}$ and $p_{2,1}$, a control-flow dependency among $p_{1,1}$ and $p_{1,2}$ and a data dependency among $p_{4,1}$ and $p_{5,1}$.

The total number of threads where task-parts can be executed on a multi-/many-core platform is denoted as m and specified in the `num_threads` clause. The *volume* of an OpenMP-DAG is defined as the sum of the WCETs of all its task-parts, i.e., $\sum_{i=1}^{N}(\sum_{j=1}^{n_i} C_{i,j})$. Also, the OpenMP application may be recurring (or sporadic), as long as the makespan[1] of the OpenMP-DAG does not exceed the period (or minimum inter-arrival time) of the application. We finally assume all OpenMP-DAG parameters are integer.

IV. Optimal Static Allocation of OpenMP-DAGs

The problem of optimally allocating OpenMP task-parts to threads can be modeled with an Integer Linear Programming (ILP) formulation. The problem aims to determine the minimum time interval needed to execute a given OpenMP application on m threads, considering both, tied and untied tasking models. In other words, we seek to derive the optimal mapping of task-parts to threads so that the task-set makespan is minimized.

A. 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.

A.1 Input parameters

(1) m: number of threads available for execution; (2) N: number of OpenMP 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 task-part $p_{i,j}$ of task τ_i; (7) rel_i: set of tasks having a relative relationship with τ_i (either as antecedents or descendants) as defined by the task creation dependencies.

A.2 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.

[1]The *makespan* of a set of precedence constrained jobs is defined as the total length of the schedule (i.e., response-time) of the collection of jobs.

A.3 Objective function

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

A.4 Initial Assumptions

(i) *The first part of the first task must begin at time* $t = 0$, i.e., $\psi_{1,1} = 0$

(ii) *The first task is executed by thread* 1.

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

A.5 Constraints

(i) *Each task is executed by only one thread.*

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

This constraint enforces the `tied` scheduling clause, i.e., for each task τ_i, only one binary variable $X_{i,k}$ is set to 1 among the m variables referring to the available threads.

(ii) *All parts of each task are allocated to the same thread.*

$$n_i \cdot X_{i,k} = \sum_{j=1}^{n_i} Y_{i,j,k} \quad \forall i \in \{1, \ldots, N\}, \forall k \in \{1, \ldots, m\} \tag{2}$$

This constraint establishes the correspondence between the $X_{i,k}$ and $Y_{i,j,k}$ variables. Please note that the constraint $n_i = \sum_{j=1}^{n_i}\sum_{k=1}^{m} Y_{i,j,k}$ is not needed since it is already implied by constraints (i) and (ii).

(iii) *All precedence requirements between task parts must be fulfilled.*

$$\begin{gathered}\forall i, w \in \{1, \ldots, N+1\},\ \forall j \in \{1, \ldots, n_i\},\\ \forall z \in \{1, \ldots, n_w\} \mid P_{w,z} \in succ_{i,j},\\ \psi_{i,j} + C_{i,j} \leq \psi_{w,z}.\end{gathered} \tag{3}$$

For each pair of task-parts, if a precedence constraint connects them because of a control-flow, task creation of data dependency, then the latter cannot start until the former has completed execution. Notice that this constraint also applies to the sink task τ_{n+1}.

(iv) *The execution of different task-parts must not overlap.*

$$\begin{gathered}\forall i, w \in \{1, \ldots, N\}, \forall j \in \{1, \ldots, n_i\}, \forall z \in \{1, \ldots, n_w\},\\ \forall k \in \{1, \ldots, m\} \mid (w \neq i)\ \vee\ (j \neq z),\\ (Y_{i,j,k} = 1\ \wedge\ Y_{w,z,k} = 1) \Rightarrow\\ (\psi_{i,j} + C_{i,j} \leq \psi_{w,z}\ \vee \psi_{w,z} + C_{w,z} \leq \psi_{i,j})\end{gathered}$$

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:

$$\begin{gathered}\forall i, w \in \{1, \ldots, N\}, \forall j \in \{1, \ldots, n_i\}, \forall z \in \{1, \ldots, n_w\}, \\ \forall k \in \{1, \ldots, m\} \mid (w \neq i) \ \vee \ (j \neq z), \\ \psi_{i,j} + C_{i,j} \leq \psi_{w,z} + M(2 + a_{i,j,w,z,k} - Y_{i,j,k} - Y_{w,z,k}) \\ \psi_{w,z} + C_{w,z} \leq \psi_{i,j} + M(3 - a_{i,j,w,z,k} - Y_{i,j,k} - Y_{w,z,k}),\end{gathered} \tag{4}$$

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

(v) *Task Scheduling Constraint 2 (TSC 2) must be satisfied.*

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

This constraint imposes that parts from a task cannot be allocated to a thread where parts from another task that is neither a descendant nor antecedent of the considered task is suspended. This is equivalent to say that if parts from two tasks not related by any descendancy 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 or equal than the starting time of the first task-part of the other one. As for constraint (iv), this constraint can be rewritten as:

$$\begin{gathered}\forall i, w \in \{1, \ldots, N\}, i \neq w, T_w \notin rel_i, \forall k \in \{1, \ldots, m\}, \\ \psi_{i,n_i} + C_{i,n_i} \leq \psi_{w,1} + M(2 + b_{i,w,k} - X_{i,k} - X_{w,k}) \\ \psi_{w,n_w} + C_{w,n_w} \leq \psi_{i,1} + M(3 - b_{i,w,k} - X_{i,k} - X_{w,k}).\end{gathered} \tag{5}$$

Note that all constraints (except constraint (iii)) need not be applied to τ_{N+1}.

B. 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: $Y_{1,1,1} = 1$

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

$$\sum_{k=1}^{m} Y_{i,j,k} = 1 \quad \forall i \in \{1, \ldots, N\}, \forall j \in \{1, \ldots, n_i\} \tag{6}$$

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

C. 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 [15]. 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 $O(N^2p^2m)$, where $p = \max_{i=1,\ldots,N} n_i$. Given the problem complexity and poor scalability of the ILP formulation, the next section proposed an efficient heuristic for providing sub-optimal solutions within a reasonable amount of time.

V. Sub-optimal Static Allocation of OpenMP-DAGs

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 [16, 17]. 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 priority rules allow selecting the next job to be executed in the set of ready jobs.

The priority rules that have been shown to perform well in the context of parallel machine scheduling are [16, 17]:

- *Longest Processing Time* (LPT): the job with the longest WCET is selected;
- *Shortest Processing Time* (SPT): the job with the shortest WCET is selected;
- *Largest Number of Successors in the Next Level* (LNSNL): the job with the largest number of immediate successors is selected;
- *Largest Number of Successors* (LNS): the job with the largest number of successors overall is selected;
- *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 a sub-optimal, yet efficient static allocation algorithm considering both the *tied* and *untied* tasking model, to map task-parts on the different threads following one of the above-mentioned priority rules, so that the partial ordering between task parts is respected.

A. Tied Tasks

Algorithm 1 instantiates the procedure for the case of tied tasks, for which existing heuristic strategies cannot be applied directly. The algorithm takes the structure G of an OpenMP-DAG and the number of available threads m defined in the `num_threads` clause as inputs. It returns as outputs a heuristic allocation of tied OpenMP tasks to threads, i.e., a vector

ψ representing the starting times of task-parts in the obtained schedule, a mapping θ of task-parts to threads, and the corresponding value of makespan μ.

Algorithm 1 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} ← ψ_{i,j} + 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};
20:             end if
21:             F_{k,z} ← F_{k,z} + 1
22:             if F_{k,z} == SIZE(INEDGES_{k,z}) then
23:                 R ← APPEND(P_{k,z}, R)
24:             end if
25:         end for
26:     end while
27:     μ = max_{i=1}^{m} L_i
28:     return (μ, ψ, θ)
29: end procedure
```

The reasoning behind the algorithm is to allocate ready-task parts to the first available thread, following a pre-determined rules of selection among ready tasks, while enforcing the specific semantics of the OpenMP tied tasking model.

First, a list R of ready task-parts is initialized with the initial task-part of the first task, i.e. $p_{1,1}$, 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 4-26 iterates until all task-parts have been allocated, i.e., until the size of list A, which contains the allocated task-parts, reaches the total number of parts in the task-set ($\sum_{i=1}^{N} n_i$). Such value is passed as input to the procedure as part of the graph structure G. 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 $p_{i,j}$ selected according to one of the priority rules described above. The allocation of the selected task-part must 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 $p_{i,j}$ is the first part of τ_i, i.e. when $j = 1$, (line 7), then the task is allocated on thread k; otherwise, its allocation must have been previously defined (according to the tied scheduling clause, all task parts must be allocated on the same thread). 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 task-part (lines 17-25). Once the counter reaches the number of its immediate predecessors, the task part may be appended to the list of ready vertices (line 23). Finally, the makespan μ for the obtained allocation is returned as output. The procedure also returns the vector ψ, which stores the starting times of task parts in the final schedule, and the vector θ, which stores the task-to-thread mapping.

The algorithm runs in polynomial time in the size of the task-set; specifically, the time complexity is $O(N^2p^2)$.

B. Untied Tasks

Algorithm 1 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 need not be performed. On the other hand, the algorithm must keep track of the thread associated to each task-part (instead of each task).

VI. EVALUATION

In this section, the proposed allocation strategies are evaluated in terms of running time and performance. To the best of our knowledge, no other static approaches are directly comparable with our proposed strategies. Therefore, they are compared with the schedulability bound derived in [9], which upper-bounds the response time of an OpenMP application composed of untied tasks, considering a dynamic scheduler.

Experiments have been performed on an Intel®Core™ i7-4770K CPU 3.50 GHz with 16GB RAM, with the ILP solver IBM ILOG CPLEX Optimization Studio v.12.61.

A. A real case study: an OpenMP 3D path planning (3DPP) application

This application is used for airborne collision avoidance in UAVs [18] to compute the path between the current position and the target position, while avoiding obstacles in a 3D environment.We tested it with two different program inputs: (1) *3DPP1* generating an OpenMP-DAG with 66 tasks and 129 task-parts; and *3DPP2* generating a DAG of 130 tasks and 257 task-parts. The OpenMP-DAGs have been obtained with the compiler technique presented in [5].

The WCET of each task-part has been computed by measuring the high watermark execution time running in "isolation". Then, a safety marging of 20% has been added[2]. Since the two DAGs have a nesting level of parallelism equal to 2 (i.e., all tasks are descendants of the first (master) task), the formulation for the tied and untied cases yield the same solutions. The experiments have been performed assuming $m = 8$; two other configurations ($m = 2$ and $m = 4$) have been examined for 3DPP2, which is computationally intensive and has a very high degree of connectivity.

Worst-case makespan results are reported in Table I. Concretely, each application configuration is evaluated with: (1)

[2]The most common industrial practice to obtain WCET values still relies on software simulation and testing, reinforced by the application of *safety margins* [19].

TABLE I
CASE STUDY RESULTS.

	3DPP1 (m=8)	3DPP2 (m=2)	3DPP2 (m=4)	3DPP2 (m=8)
ILP-BF	254	506	506	506
SPT	317	824	660	571
LPT	254	659	577	530
LNS	254	715	506	506
LNSNL	300	748	619	549
LRW	254 (7s)	717 (11m41s)	506 (11m48s)	506 (11m53s)
BOUND-untied	331	827	666.5	586.25

the optimal allocation based on the ILP formulation (labeled as *ILP-BF*)[3]; (2) the sub-optimal allocation based on the heuristics presented in Section V (labeled as *SPT, LPT, LNS, LNSNL*, and *LRW*; and (3) the schedulability bound derived in [9], which upper-bounds the response time of an OpenMP application composed of untied tasks, considering a dynamic scheduler. The work in [9] also highlighted the complexity of deriving a tight upper-bound on the response time in the case of tied tasks, for which no schedulability analysis exists. In case of the LRW, we report the running time of LRW in parenthesis as well (the most computationally intensive heuristic).

Although ILP-BF constantly provides the best worst-case makespan compared with the sub-optimal static allocation heuristics, results are very similar, with a 38% of variation in the worst-case (SPT for the 3DPP2 with $m = 2$). The running time of LRW reported in the Table shows that, while for 3DPP1 the solution is found rapidly, the running time of 3DPP2 appears significantly larger (but still reasonable), due to its big and complex structure. Finally, BOUND-untied always overestimates the ILP solution of at least 15% (65% in the worst-case), mainly because the bound is tight for very peculiar graph structures that are not representative of the general behavior. Although for this particular application none of the heuristic strategies clearly dominates the others, all of them are able to effectively reduce the pessimism determined by BOUND-untied.

B. Synthetic OpenMP-DAG generation

The synthetically generated task graphs compliant with the OpenMP semantics are generated as follows: Initially, the number N of tasks in the system is uniformly chosen in the interval $[N_{min}, N_{max}]$, while the number of parts n_i of each task is randomly selected as an integer in the interval $[1, n_{max}]$. Each task part is labeled with a value of WCET $C_{i,j}$ uniformly selected in the interval $[1, 10]$. Then, the precedence constraints between task parts, i.e., control-flow, task creation and data dependencies (See Figure 1b) are generated. First, control flow dependencies are assigned between any pair of consecutive task parts to guarantee the correct order of execution among them. Then, task creation dependencies are determined as follows: First, descendance levels $\ell_1, \ldots, \ell_k$, $k \leq n$, are randomly assigned to tasks, making sure that each level contains at most as many tasks as the number of task parts in the previous level (since each task part corresponds to a TSP where at most one task can be created). Also, for any pair of tasks τ_i and τ_j, if $i < j$ then $\ell_i \leq \ell_j$. Second, dependencies are randomly assigned between parts of tasks belonging to consecutive descendance levels by enforcing that each task τ_i can generate at most n_i tasks. Finally, data dependencies are created between pairs of tasks τ_i and τ_j ($i < j$) belonging to the same descendance level (i.e., $\ell_i = \ell_j$) by adding a dependency between p_{i,n_i} and $p_{j,1}$, with probability $P_s = 0.2$.

[3]Due to the high complexity of this approach, the best found solution is recorded after running the solver for 5 hours. In all cases, however, the ILP-BF converged very rapidly (∼10 sec.) to the best found solution.

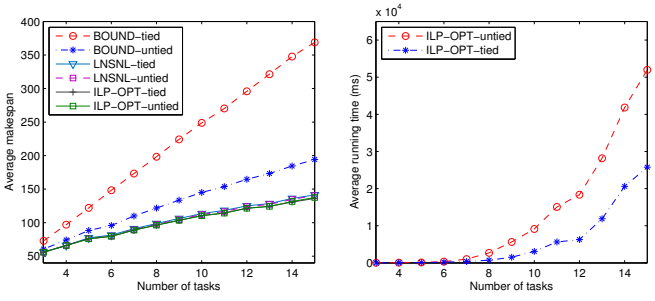
C. Synthetic OpenMP-DAG experimental results

C.1 Small task sets

A first set of experiments has been performed to evaluate the optimal solutions computed by ILP solver as a function of the number N of tasks in the system. For each value of N, 500 random instances are generated.

Figs. 2a and 2b report the average makespan and the average running time of the ILP solver respectively, for $N \in [3, 15]$, $m = 4$ and $n_{max} = 8$ and for both tied and untied tasks. Fig. 2a shows that on average the optimal makespan in the tied case (labeled as *ILP-OPT-tied*) is as for the untied case (labeled as *ILP-OPT-untied*), and that the solutions differ in only few instances with minor difference of makespan value. The figure also compares the optimal solutions to the schedulability upper-bound for untied tasks (labeled as *BOUND-untied*) given in [9] and the pessimistic bound for the tied case (labeled as *BOUND-tied*) given by the volume of the OpenMP-DAG. For the largest task-sets, i.e. $N = 15$, the bounds for tied and untied tasks over-estimate the optimal solution of about 43% and 170%, respectively. The figure also highlights the excellent performance of the heuristic approach based on LNSNL priority rule, which closely matches the optimal solutions and slightly outperforms on average the other priority rules (not reported for readability).

Although the number of constraints and variables of the ILP formulation for untied tasks grow asymptotically as those for tied tasks in the worst case, the running time of the ILP solver (Fig. 2b) is significantly larger in the untied case. This difference is due to the larger solution space size of the problem for untied tasks. As expected, in both cases the running time to solve the ILP grows exponentially as N increases. The same trends of Figs. 2a and 2b have been observed also when varying the number of threads, hence the corresponding plots are not reported.



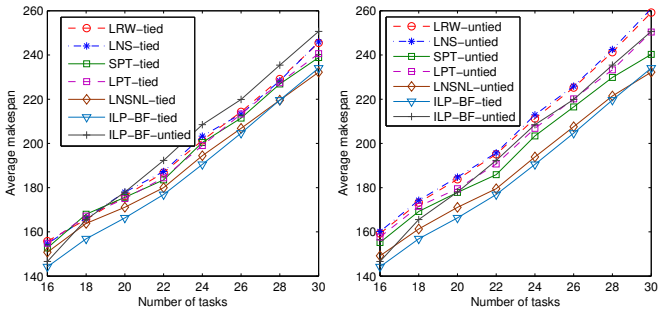
(a) (b)

Fig. 2. Average makespan (a) and running time (b) as a function of N, with $m = 4$ and $n_{max} = 8$.

C.2 Large task sets

A second set of experiments aims at evaluating the scalability of the heuristics proposed in Section V for task sets in which the ILP solver is unable to find the optimal solution in a reasonable time. For each value of N, 100 task-set instances have been generated and for each of them, the best feasible solution found by the ILP solver (ILP-BF) in 300 s is collected. Simulation results have confidence interval values of at least 10%, with a confidence level of 95%.

Figs. 3a and 3b depict the average makespan for the tied and untied case when $m = 4$, $n_{max} = 8$ and N is varied in the range $[16, 30]$. As in the case of 3DPP, the different priority rules yield a similar performance, but in average the LNSNL priority rule slightly outperforms the others, and even ILP-BF-tied for high values of N. As the problem size increases, ILP-BF-untied has a significantly slower convergence than ILP-BF-tied (see also Fig. 2b), leading to a significant performance gap between the two curves. Also, the heuristic strategies are able to quickly find nearly optimal solutions. For instance, for $N = 30$, the execution time of LRW (the most computationally intensive heuristic strategy) is on average of about 9.84 ms and 22.23 ms, for the tied and untied case, respectively. In the worst-case, LRW-tied and LRW-untied take 147.62 ms and 242.48 ms, respectively. In Figs. 3a and 3b, the schedulability bounds (not reported for readability) are as the one observed in Fig. 2a.



(a) (b)

Fig. 3. Average makespan as a function of N, with $m = 4$ and $n_{max} = 8$ for the tied (a) and untied (b) case.

VII. Conclusions

The adoption of OpenMP is fundamental for an efficient exploitation of many-core embedded systems. However, OpenMP relies on dynamic scheduling strategies, which is not allowed in certain safety-critical domains in which the use of static allocation guarantees the correct functionality of the system. This paper proposes an ILP formulation to derive an optimal static allocation compliant with the OpenMP tasking model. With the objective of reducing the complexity of the ILP solver, the paper also proposes five heuristics for an efficient (although sub-optimal) allocation. Results show a significant reduction in the worst-case makespan comared with an existing schedulability upper-bound (for untied tasks only). Moreover, the proposed heuristics perform very well, closely matching the optimal solutions.

References

[1] B. D. de Dinechin, D. van Amstel, M. Poulhies, G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *DATE*, 2014.

[2] *Texas Instruments. The 66AK2H12 Keystone II Processor*, http://www.ti.com/product/66AK2H12.

[3] "OpenMP Application Program Interface, Version 4.5," http://www.openmp.org, November 2015.

[4] R. Vargas, E. Quiñones, and A. Marongiu, "OpenMP and timing predictability: A possible union?" in *DATE*, 2015.

[5] R. Vargas, S. Royuela, M. A. Serrano, X. Martorell, and E. Quiñones, "A lightweight OpenMP4 run-time for embedded systems," in *ASP-DAC*, 2016.

[6] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *RTSS*, 2012.

[7] A. Melani, M. Bertogna, V. Bonifaci, A. M. Spaccamela, and G. Buttazzo, "Response-time analysis of conditional DAG tasks in multiprocessor systems," in *ECRTS*, 2015.

[8] M. A. Serrano, A. Melani, M. Bertogna, and E. Quiñones, "Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions," in *DATE*, 2016.

[9] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones, "Timing characterization of OpenMP4 tasking model," in *CASES*, 2015.

[10] M. Panić, S. Kehr, E. Quiñones, B. Böddecker, J. Abella, and F. J. Cazorla, "RunPar: An Allocation Algorithm for Automotive Applications Exploiting Runnable Parallelism in Multicores," in *CODES+ISSS*, 2014.

[11] S. Kehr, E. Quiñones, B. Boeddeker, and G. Schaefer, "Parallel Execution of AUTOSAR Legacy Applications on Multicore ECUs with Timed Implicit Communication," in *DAC*, 2015.

[12] M. Di Natale, W. Zheng, C. Pinello, P. Giusto, and A. S. Vincentelli, "Optimizing end-to-end latencies by adaptation of the activation events in distributed automotive systems," in *RTAS*, 2007.

[13] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *RTSS*, 2010.

[14] A. Saifullah, K. Agrawal, C. Lu, and C. D. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *RTSS*, 2011.

[15] J. K. Lenstra and A. H. G. Rinnooy Kan, "Complexity of scheduling under precedence constraints," *Operations Research*, vol. 26, no. 1, 1978.

[16] M. L. Pinedo, *Scheduling: theory, algorithms, and systems*. Springer Science & Business Media, 2012.

[17] K. E. Raheb, C. T. Kiranoudis, P. P. Repoussis, and C. D. Tarantilis, "Production scheduling with complex precedence constraints in parallel machines," *Computing and Informatics*, vol. 24, no. 3, 2012.

[18] C. Rochange, A. Bonenfant, P. Sainrat, M. Gerdes, J. Lobo, et al., "WCET analysis of a parallel 3D multigrid solver executed on the MERASA multi-core," in *WCET*, 2010.

[19] E. Mezzetti and T. Vardanega, "On the industrial fitness of WCET analysis," in *WCET*, 2011.