OpenMP and Timing Predictability: A Possible Union?

Roberto Vargas
Barcelona Supercomputing Center (BSC)
and Technical University of Catalonia (UPC)
roberto.vargas@bsc.es

Eduardo Quinones
Barcelona Supercomputing Center (BSC)
eduardo.quinones@bsc.es

Andrea Marongiu
Swiss Federal Institute of
Technology in Zurich (ETHZ)
a.marongiu@iis.ee.ethz.ch

***Abstract*—Next-generation many-core embedded platforms have the chance of intercepting a converging need for high performance and predictability. Programming methodologies for such platforms will have to promote predictability as a first-class design constraint, along with features for massive parallelism exploitation. OpenMP, increasingly adopted in the embedded systems, has recently evolved to deal with the programmability of heterogeneous many-cores, with mature support for fine-grained task parallelism. While tasking is potentially very convenient for coding real-time applications modeled as periodic task graphs, OpenMP adopts an execution model completely agnostic to any timing requirement that the target application may have. In this position paper we reason about the suitability of the current OpenMP v4 specification and execution model to provide timing guarantees in many-cores.**

I. Introduction

Fueled by the recent technological advancements and market trends, we are witnessing the convergence of High-Performance Computing (HPC) and Embedded Computing (EC) systems. High-end EC systems are increasingly concerned with providing HPC-like performance in real-time, challenging the performance capabilities of current architectures. The advent of next-generation many-core embedded platforms has the chance of intercepting this converging need for "*predictable high-performance*", given that appropriate programming paradigms for massive parallelism exploitation in a predictable way are devised [**?**].

OpenMP [**?**], the de-facto standard for shared memory parallel programming, has been succesfully used for decades in the HPC domain and has recently gained much attention also in the embedded field [**?**] [**?**] [**?**] [**?**] [**?**]. Originally focused on data-parallel, loop-intensive types of applications, OpenMP has evolved over the years to express fine-grained, irregular and highly-dynamic *task* parallelism. The latest specification v4.0 was further augmented with features to express dependencies among such tasks. OpenMP v4.0 tasking retains certain similarities to the formalisms used to describe real-time applications (e.g., task graphs), that makes it a good candidate to fill the existing gap between i) a convenient programming model for embedded manycores and ii) state-of-the-art techniques for scheduling with timing guarantees. However, OpenMP adopts a programming interface and a parallel execution model that is completely agnostic to any timing requirement that the target application may have.

In this position paper we reason about the possibility of adopting the current OpenMP v4 specification to provide timing guarantees in embedded manycores. To the best of our knowledge, we are the first to consider a real integration between state-of-the-art techniques for real-time scheduling and the OpenMP *tasking* execution model, all the previous attempts in this direction (e.g., [**?**] [**?**]) being limited to using OpenMP v2.5 directives as a mere programming frontend to describe a task-graph. Specifically, with this paper we try to answer the following questions: *Can OpenMP tasks be used to describe a real-time application? What possibly needs to be changed or improved at various levels to enable classical timing analysis and real-time scheduling within the OpenMP tasking model? How to leverage standard real-time scheduling techniques without violating the semantics of the OpenMP execution model?* To address these issues we delve into a detailed analysis of the OpenMP v4.0 specification and semantics, particularly focusing on three key points: i) how to derive a task graph which conveys appropriate information for scheduling constraints and ii) worst-case execution time (WCET) analysis; iii) how to ensure that OpenMP task scheduling constraints do not clash with traditional real-time scheduling techniques.

II. Real-Time Scheduling of Parallel Applications

The *task model* [**?**], either *sporadic or periodic*, is a well-known model in scheduling theory to represent real-time systems. In this model, real-time applications are typically represented as a set of n *recurrent* tasks $\tau = \{\tau_1, \tau_2, .., \tau_n\}$, each characterized by three parameters: worst-case execution time ($WCET$), period (T) and relative deadline (D). With the introduction of multi-core processors, new scheduling models have been proposed to better express the parallelism that these architectures offer. This is the case of the *sporadic DAG model* [**?**] [**?**] [**?**] [**?**] [**?**], which generalizes the *fork-join* execution model to allow exploitation of parallelism *within* tasks. In the sporadic DAG model each task (called *DAG-task*) is represented with a *directed acyclic graph* (DAG) $G = (V, E)$, plus T and D. Each node $v \in V$ denotes a sequential operation or *job*, characterised by a *worst-case execution time* (WCET) estimation. Edges represent dependencies between jobs: if $(v_1, v_2) \in E$ then the job v_1 must complete its execution before job v_2 can start executing. The DAG captures scheduling constraints imposed by dependencies among jobs and it is annotated with WCET estimation of each job.

III. An Overview of OpenMP Tasking

An OpenMP program starts with a single thread of execution, called the *master* or *initial* OpenMP thread[1], that runs sequentially. When the thread encounters a `parallel` construct, it creates a new *team* of threads, composed of itself and $n-1$ additional threads (n being specified with the `num_threads` clause). The use of *worksharing* constructs allows specifying how the computation within a parallel region is partitioned among threads. In this paper, we focus on the `task` construct and its associated execution model.

[1] In the rest of the paper, the term "thread" refers an OpenMP thread.

```
1  #pragma omp parallel num_threads(10) {
2  #pragma omp master {
3  #pragma omp task { // T0
4     part00
5     #pragma omp task depend(out:x) // T1
6                      final(true)
7     {
8        part10
9        #pragma omp task { part4 } // T4
10       part11
11    }
12    part01
13    #pragma omp task depend(in:x) // T2
14    { part2 }
15    part02
16    #pragma omp taskwait
17    part03
18    #pragma omp task { part3 } // T3
19    part04
20 }}}
```

Fig. 1. Example of an OpenMP program using tasking worksharing construct

When a thread encounters a `task` construct, a new *task region* is generated from the code contained within the task. The execution of the new task region can be then assigned to one of the threads in the current team for immediate or deferred execution, based on additional *task-scheduling* clauses: `depend`, `if`, `final` and `untied`. The `depend` clause allows to describe a list of `in`, `out` or `inout` dependences on target data items. If a task has an `in` dependence on a variable, it cannot start execute until the set of tasks that with `out` and/or `inout` dependences on the same variable complete. Dependences can only be defined among *sibling*[2] tasks. When an `if` clause is present and its associated expression evaluates to false, the new generated task becomes *undeferred* and executed immediately by a thread of the team. The current task region is suspended until the new task completes. When a `final` clause is present and its associated expression evaluates to true, all its child tasks are *undeferred* and *included* tasks, meaning that the encountering threads itself executes sequentially all the new descendants. By default, OpenMP tasks are *tied* to the thread that first starts their execution. If such tasks are suspended, they can later only be resumed by the same thread. When a `untied` clause is present, the task is not tied to any thread and so in case it is suspended it can later be resumed by any thread in the team. All tasks bound to a given parallel region are guaranteed to have completed at the implicit barrier at the end of the parallel region, as well as at any other explicit `barrier` construct. Synchronization over a subset of explicit tasks can be specified with the `taskwait` construct, which forces the encountering task to wait for all its first-level descendants to complete before proceeding.

Figure **??** shows an example OpenMP program . The code enclosed in the `parallel` construct defines a team of 10 threads. The `master` worksharing construct at line 3 specifies that only the *master* thread executes the associated block of code. At line 4, a new task region T_0 is created and assigned to a thread in the team. When the thread executing T_0 encounters the `task` constructs at lines 7, 17 and 23, new tasks T_1, T_2 and T_3 are generated. Similarly, the thread executing T_1 will create task T_4 at line 11. Tasks T_1 and T_2 include a `depend` clause that defines a dependence on the memory reference x, so T_2 cannot start until T_1 finishes. T_4 is an *included task* because its parent T_1 contains a `final` clause that evaluates to *true*, so T_1 is suspended until T_4 finishes. All tasks are guaranteed to have completed at the *implicit barrier* at the end of the parallel region at line 26. Task T_0 waits on the `taskwait` at line 20 until tasks T_1 and T_2 (not its descendant task T_4) have completed before proceeding past the `taskwait`.

[2]First-level descendants of the same *parent* task.

IV. Similarities and Differences between OpenMP and the DAG Model

Although the current specification of OpenMP lacks any notion of real-time scheduling semantics, such as deadline, period or WCET, the structure and syntax of an OpenMP program have certain similarities with the DAG model. The most immediate features of the OpenMP programming interface that could be thought of for describing a task graph are the `task` directive, the `depend` clause and the `taskwait` directive. Intuitively, the first describes a job in V in the DAG model; while the second and the third describe the *edges* (data dependence and synchronization) in E in the DAG model. However, this DAG would not convey enough information to derive a real-time schedule that complies to the semantics of the OpenMP tasking execution model. This section discusses the main OpenMP characteristics that makes it difficult to compare with DAG models.

A. The OpenMP Tasking Execution Model

OpenMP defines *task scheduling points* (TSP) as points in the program where the encountering task can be suspended, and the hosting thread can be rescheduled to a different task. TSPs occur upon task creation and completion, and at task synchronization points such as `taskwait` directives, explicit and implicit barriers[3]. TSPs divide task regions into *parts* executed uninterrupted from start to end. Different parts of the same task region are executed in the order in which they are encountered. The example shown in Figure **??** identifies the parts in which each task region is divided: T_0 is composed of part_{00}, part_{01}, part_{02}, part_{03} and part_{04}; T_1 is composed of part_{10} and part_{11}; and T_2, T_3 and T_4 are composed of part_2, part_3 and part_4 respectively. When a task encounters a TSP, program execution branches into the OpenMP runtime system, where task schedulers 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 team. The order in which these two actions are applied is not specified by the standard, but it is subject to the following *task scheduling constraints* (TSC):

1) An *included* task must be executed immediately after the task is created.
2) Scheduling of new *tied* tasks is constrained by the set of task regions that are currently tied to the thread, and that are not suspended in a `barrier` region. If this set is empty, any new *tied* task may be scheduled. Otherwise, a new *tied* task may be scheduled only if the new *tied* task is a *child task* of the task region.
3) A dependent task shall not be scheduled until its task data dependencies are fulfilled.
4) When a task is generated by a construct containing an `if` clause for which the conditional expression

[3]Additional TSPs are implied at constructs `target`, `taskyield`, `taskgroup` that we do not consider in this paper for simplicity.

evaluates to false, and the previous constraints are already met, the task is executed immediately after generation of the task.

To correctly capture the precedence constraints defined by the OpenMP specification with DAG-based real-time scheduling models it is required not only to know the dependencies among tasks but also to determine: 1) the point in time of each TSP; 2) the task creation clauses that influence the scheduling of a task.

B. WCET estimation and OpenMP Tasks

TSPs also have an impact on the timing analysis of OpenMP tasks, as the timing behaviour of tasks depends on the decisions taken by the execution model upon TSPs. The sporadic DAG model annotates each job in V with its WCET estimation. However, the execution model of OpenMP tasks differs from the execution model of jobs in V in a fundamental aspect: A node in the DAG model is a sequential operation that cannot be interrupted; instead an OpenMP task can legally contain multiple TSPs at which the task can be suspended or resumed as specified by the TSC (explained in Section **??**). As a result, the timing behaviour of an OpenMP task not only depends on the computation of the task itself, but also on the timing behaviour of its descendant tasks and the dependencies existing among them. Thus, the WCET estimation of T_0 depends on the timing behaviour of T_1, T_2, T_3 and T_4. Similarly, the timing behavior of T_1 depends on T_4. Task creation clauses also influence the timing behaviour of a task. This is the case of the `final` clause, which guarantees that T_4 will be executed undeferred by the same thread that executes task region T_1. In absence of the `final` clause, T_4 would not necessarily execute immediately, and so the timing behaviour of T_1 would change as well.

V. OpenMP and Timing Predictability: A Possible Union?

In this section we discuss how to overcome the difficulties of applying DAG-based models to OpenMP execution model highlighted previously, focusing on three key elements: *1) How to reconstruct from code analysis (TSPs) an OpenMP task graph that resembles the DAG-task structure, 2) How to apply WCET analysis to the nodes of the graph* and 3) *how to schedule OpenMP tasks based on DAG-task methodologies so that TSCs are met.*

A. Reconstructing the OpenMP DAG should consider the TSP

The execution of a task *part* resembles the execution of a job in V, i.e., it is executed uninterrupted. To that end, we propose to consider task *parts* and not tasks as nodes in V. Figure **??** shows the OpenMP-DAG of the example presented in Figure **??**, in which task *parts* are the nodes in V. T_0 is decomposed in its corresponding parts P_{00}, P_{01}, P_{02}, P_{03} and P_{04}, with a TSP at the end of each (creation of tasks T_1, T_2, and T_3 for P_{00}, P_{01} and P_{03}, and the `taskwait` construct for P_{02}). Similarly, T_1 is decomposed in P_{10} and P_{11} with the creation of task T_4 at the end of P_{10}.

Depending of the TSP encountered at the end of a task *part* (task creation or completion, task synchronization), we distinguish three different *dependency types*: Control flow dependencies (dotted arrows) that force parts to be scheduled in the same order as they are executed within the task, TSP dependencies (dashed arrows) that force tasks to start/resume execution after the corresponding TSP, and synchronization dependencies (solid arrows) that force the sequential execution of tasks as defined by the `depend` clause and task synchronization constructs. The OpenMP-DAG is annotated with all dependence types (with no need to differentiate them).

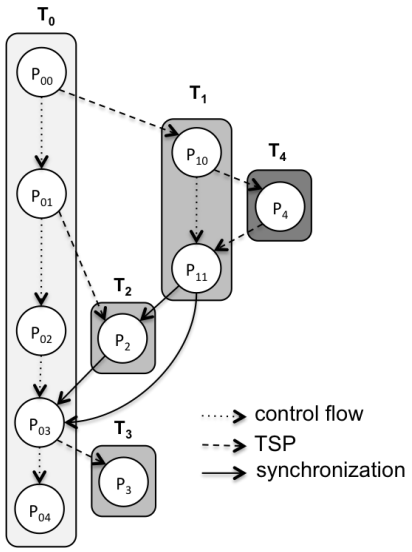

Fig. 2. OpenMP-DAG composed of task region *parts*.

Besides the `depend` clause, the `if` and `final` clauses also affect the order in which task region parts are executed. In both cases the encountering task is suspended until the newly generated task completes execution. In order to model the undeferred and included task behaviour, we introduce a new edge in E. In Figure **??**, a new dependence between P_4 and P_{11} is inserted such that task region T_1 does not resume its execution until the included task T_4 finishes.

B. Timing analysis should be applied to task region parts

To comply to the DAG-model, nodes in V in the OpenMP-DAG should also be annotated with the WCET estimation of the corresponding task region parts. By constructing the OpenMP-DAG based on the knowledge of TSPs the timing analysis of each node has a WCET which is independent of any dynamic instance of the OpenMP program (i.e., how threads may be scheduled to tasks and parts therein). The timing behaviour of task *parts* will only be affected by interferent access to shared resources [**?**].

C. Real-time task scheduling must not violate the TSC

When applying standard techniques for real-time scheduling of multicores, it is mandatory that the semantics specified by OpenMP TSC are not violated. Task creation clauses not only define new precedence constraints, as shown in Section **??**, but they also define the way in which tasks, and *parts* therein, are scheduled according to the TSC defined in Section **??**. This is the case of `if`, `final` and `untied` clauses, as well as the default behavior (*tied* tasks).

TSC 1 imposes *included* tasks to be executed immediately by the encountering thread. In this case, the scheduling of the OpenMP-DAG can follow two strategies: (1) It may assign a higher priority to included tasks or (2) it may consider both the task *part* that encounters it and the complete *included* task

```
1 #pragma omp task // T1
2 {
3   #pragma omp task if(false) {...} // T2
4 }
5 #pragma omp task    {...} // T3
```

Fig. 3. Example of an OpenMP fragment of code with *tied tasks*.

region as a unique unit of scheduling. In Figure **??**, the former case would give T_4 the highest priority. The latter case would consider P_{10} and P_4 as a unique unit of scheduling.

TSC 2 does not allow scheduling new *tied* tasks if there are other suspended *tied* tasks already assigned to this thread that are parents of the new task. Figure **??** shows a fragment of code in which this situation can occur. Let's assume that T_1, which is not a parent of T_3, is executed by *thread 1*. When T_1 encounters the creation point of T_2, it is suspended because of TSC 4, and it cannot resume until T_2 finishes. Let's consider that T_2 is being executed by a different thread instead, e.g. *thread 2*. If T_2 has not finished, when the creation point for T_3 is reached, T_3 cannot be scheduled on *thread 1* because TSC 2 is not accomplished, even if *thread 1* is idle. As a result, *tied* tasks constrain the scheduling opportunities of the OpenMP-DAG. Ideally, every time a new *tied* task is generated it should be scheduled onto a different thread in order to accomplish the TSC 2.

TSC 3 imposes tasks to be scheduled respecting the task dependencies. This information is already contained in the OpenMP-DAG.

TSC 4 states that *undeferred* tasks execute immediately if TSCs 1, 2 and 3 are met. `untied` tasks are not subject to any TSC, allowing *parts* of the same task to execute on different threads, so when a task is suspended, the next *part* to be executed can be resumed on a different thread. Therefore, one possible scheduling strategy for `untied` tasks which satisfied TSC 4 is not to schedule *undeferred* and `untied` task *parts* until *tied* and *included* tasks are assigned to a given thread. This guarantees that TSC 1 and 2 are met. This is because task *parts* of *tied* and *included* tasks are bound to the thread that first started their execution, which reduces significantly their sheduling opportunities. `untied` and *undeferred* task *parts* instead have higher degree of freedom as they can be scheduled to any thread of the team.

For the OpenMP-DAG to convey enough information to devise a TSC-compliant scheduling, each node in V must be augmented with the *type of task* (`untied`, *tied*, *undeferred* and *included*). In Figure **??**, T_0 is marked as *tied*, T_1, T_2 and T_3 are marked as `untied` and T_4 is marked as *included*.

VI. Conclusion and Future Work

This paper is the first to evaluate the suitability of OpenMP v4 specification and execution model to deliver the performance and predictability required by modern real-time applications on embedded many-cores. We study how to construct an OpenMP task graph which contains enough information to allow the application of real-time DAG scheduling models, from which timing guarantees can be derived. We identify task *parts* (non-preemptible code segments) as the program unit to which standard WCET estimation and scheduling can be safely applied, and we explain how various types of dependencies, besides explicit constructs, are implied in the OpenMP execution model. Implicit and explicit dependences can be captured in a OpenMP-DAG, to correctly model program behaviour as defined by the OpenMP specification. Finally, we reason about the effect that OpenMP *task scheduling constraints* may have on the traditional real-time scheduling of this new OpenMP-DAG. Our study shows that ***the union between OpenMP and timing predictability is possible***, and paves the way for new scheduling techniques compatible with the OpenMP-DAG.

VII. Acknowledgments

This work was supported by EU project P-SOCRATES (FP7- ICT-2013-10) and by Spanish Ministry of Science and Innovation grant TIN2012-34557.

References

[1] B. Lisper, "Towards Parallel Programming Models for Predictability," in *Workshop on WCET Analysis*, 2012.

[2] OpenMP Architecture Review Board, "OpenMP Application Program Interface, Version 4.0," October 2013. [Online]. Available: http://www.openmp.org

[3] E. Stotzer, et al., "OpenMP on the Low-Power TI Keystone II ARM/DSP System-on-Chip," in *OpenMP in the Era of Low Power Devices and Accelerators*, ser. Lecture Notes in Computer Science, A. Rendell, B. Chapman, and M. Mller, Eds. Springer Berlin Heidelberg, 2013, vol. 8122, pp. 114–127. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40698-0_9

[4] A. Marongiu, et al., "Improving the programmability of STHORM-based heterogeneous systems with offload-enabled OpenMP," in *Proceedings of the 1st International Workshop on Many-core Embedded Systems, (MES'2013)*, 2013, pp. 1–8.

[5] B. Chapman, et al., "Implementing OpenMP on a high performance embedded multicore MPSoC," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1–8.

[6] C. Wang, et al., "libEOMP: A Portable OpenMP Runtime Library Based on MCA APIs for Embedded Systems," in *Workshop on PMAM*, 2013.

[7] P. Burgio, et al., "Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters," in *DATE*, 2013.

[8] K. Lakshmanan, et al., "Scheduling parallel real-time tasks on multi-core processors," in *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, Nov 2010, pp. 259–268.

[9] K. Agrawal, et al., "A real-time scheduling service for parallel tasks," in *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, ser. RTAS '13, 2013, pp. 261–272.

[10] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, oct 2011.

[11] V. Bonifaci, et al., "Feasibility Analysis in the Sporadic DAG Task Model," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, July 2013, pp. 225–233.

[12] S. Baruah, et al., "A generalized parallel task model for recurrent real-time processes," in *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, Dec 2012, pp. 63–72.

[13] A. Saifullah, et al., "Multi-core real-time scheduling for generalized parallel task models," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, Nov 2011, pp. 217–226.

[14] S. Baruah, "Improved multiprocessor global schedulability analysis of sporadic dag task systems," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, July 2014.

[15] J. Li, et al., "Outstanding paper award: Analysis of global edf for parallel tasks," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, July 2013, pp. 3–13.

[16] P. Radojković, et al., "On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 34:1–34:25, jan 2012.