CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Conference Paper

Analysis of self-interference within DAG tasks

José Fonseca
Vincent Nélis
Geoffrey Nelissen
Luis Miguel Pinho

CISTER-TR-150604

2015/07/07

Analysis of self-interference within DAG tasks

José Fonseca, Vincent Nélis, Geoffrey Nelissen, Luis Miguel Pinho

CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: jcnfo@isep.ipp.pt, nelis@isep.ipp.pt, grrpn@isep.ipp.pt, lmp@isep.ipp.pt

http://www.cister.isep.ipp.pt

Analysis of self-interference within DAG tasks

José Fonseca, Vincent Nélis, Geoffrey Nelissen and Luís Miguel Pinho
CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Portugal
Email: {jcnfo,nelis,grrpn,lmp}@isep.ipp.pt

I. Introduction

Few years ago, the frontier separating the real-time embedded domain from the high-performance computing domain was neat and clearly defined. Nowadays, many contemporary applications no longer find their place in either category as they manifest both strict timing constraints and work-intensive computational demands. The only way forward to cope with such orthogonal requirements is to embrace the parallel execution programming paradigm on the emergent scalable and energy-efficient multi-core/many-core architectures.

Although the parallel programming paradigm is nothing fundamentally new, the real-time community has only recently started to actively investigate the repercussions of using parallel tasks in the analysis of the worst-case behavior of the systems. Several parallel task models and respective schedulability analysis have been proposed in order to enrich classical schedulability theory with an answer to these new challenges [1]–[4]. Small steps have been taken so far, and the progress is still strongly coupled to the well-established literature for sequential tasks.

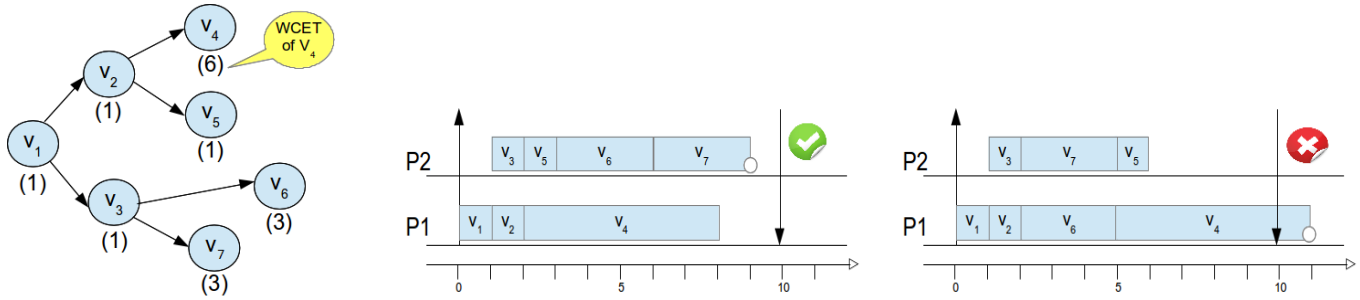
The most general model studied by the real-time community is the Directed Acyclic Graph (DAG) model [3]. It reflects sophisticated types of dynamic, fine-grained and irregular parallelism that can be achieved by using, e.g., the OpenMP programming model [5]. The resulting individual computational units (nodes of the DAG) are conventionally numerous and short-living for speed up purposes. Moreover, the nodes are mapped to OpenMP threads, and then scheduled through OS threads, according to a rather complex run-time dispatcher. Current works bypass all these run-time decisions and focus solely on the OS thread-to-core scheduling in a black-box manner. Unfortunately, the dynamic nature of the standard run-time manager along with the flexibility of the model complicate the derivation of accurate worst-case estimates both on execution time and on response time, which is even further aggravated by the choice of global scheduling schemes. In the following we describe one of the major sources of pessimism associated to such settings.

II. Problem description

Under the real-time DAG model, parallel tasks are typically characterized by a minimum inter-arrival time T_i, an end-to-end relative deadline D_i, and a directed acyclic graph $G_i = (V_i, E_i)$. Each node $v \in V_i$ represents a sequential sub-task and is associated with a wost-case execution time (WCET), whereas each arc $(v_a, v_b) \in E_i$ represents a unidirectional precedence constraint between two different nodes, with the interpretation that sub-task v_b cannot begin execution until sub-task v_a completes. No other restriction is imposed by the model (e.g., global synchronization points are not mandatory and there is no limit on the degree of parallelism). Additionally, critical path L_i and workload W_i are two metrics of interest also defined in the model. Critical path corresponds to the longest (w.r.t. WCET) path of sub-tasks for which there are arcs connecting every two adjacent sub-tasks. Workload is the maximum execution requirement of the task, i.e. the sum of all nodes WCETs. From a feasibility perspective, only the following two necessary (but not sufficient) conditions must be met: 1) $L_i \leq D_i$, and 2) $\sum \frac{W_i}{T_i} \leq m$, where m is the number of cores in the system.

Regarding the priority assignment scheme, most of the schedulability-related works that assume the DAG model consider a task- or job-level priority system, whether the priorities are static or dynamic. That is, no unique priorities are assigned to the individual sub-tasks, and thus all sub-tasks are assumed to inherit the priority of the parent task/job (i.e. the priority of the DAG). In this sense, the actual sub-task scheduling is unknown in the analysis. In practice, it is the run-time environment who decides on-the-fly which sub-tasks are executed first. Without entering into details, FIFO or LIFO, in conjunction Breadth-First Scheduling (BFS) or Work-First Scheduling (WFS), are the usual policies enforced by the run-time environment to arbitrate the sub-task-to-thread assignment. Therefore the dispatching decisions greatly depend on the time instant at which sub-tasks are released, which in turn varies according to the completion time of their predecessors. If on top of that we consider a global scheduling algorithm, then broadly speaking, ready sub-tasks may execute anywhere and in many different orders.

Due to this flexibility, interference of a DAG task on itself (referred as "self-interference" hereinafter) becomes a problem that should be considered by the scheduling algorithms and subsequently taken into account by the timing analysis in order to improve the utilization of the systems. Self-interference corresponds to the unexpected delay caused on the response time of a parallel task by its own sub-tasks. Although in general self-interference is an intrinsic property of parallel tasks, the main issue under the specified settings is that self-interference is not static. Meaning that two non-dependent sub-tasks may delay each other's execution in different instances of the same DAG task. The self-interference is thus mutual. Consequently, every path is penalized from a worst-case perspective. Let us consider a DAG task τ_i (as depicted in Fig. 1a) with deadline equal to 10 to be globally scheduled on a 2 cores platform. No other tasks co-exist in the system. Fig. 1b shows a schedule when sub-tasks v_4 and v_5 are dispatched first, resulting in a response time of 9 which clearly demonstrates that τ_i is feasible. However, it is possible that sub-tasks v_6 and v_7 get picked ahead, for instance due to a marginal early completion of v_3. In this scenario, the produced schedule (see Fig. 1c) causes τ_i to miss its deadline by 1 time unit, making the overall system infeasible. The fundamental difference resides on the self-interference shifted partially upon the critical path.



(a) DAG struture of τ_i. $L_i = 8$ and $W_i = 16$.

(b) A feasible global schedule for τ_i.

(c) A infeasible global schedule for τ_i.

Fig. 1: Feasibility problem of a DAG task τ_i with $D_i = 10$, under global scheduling.

While this simple example illustrates how dynamic self-interference may affect the feasibility of DAG tasks under global scheduling, the problem is further exacerbated when other tasks contend for access to the cores. The main reason behind relates to the fact that multiple paths contribute to the worst-case response time of lower priority tasks, and each one of such paths must be previously bounded according to the self-interference pattern that delays them the most.

It should also be noted that, although this paper focuses on core-level scheduling problems, the concept of self-interference propagates to the other sub-systems (e.g., memory access), posing even more challenges for WCET and overhead analysis.

III. Discussion on potential solutions

The self-interference problem has its roots on a too flexible software stack. In this section, we briefly argue about three different ways to restrict this flexibility and thus confine the problem. The first solution focuses on the sub-task scheduling, the second on the sub-task affinity, and the third on the DAG generation. Each one of the potential solutions minimizes the problem to a limited extent, hence a combination of them may lead to more satisfactory results.

1) **2-level priorities.** A separated fixed set of priorities is associated with the sub-tasks of each DAG task, so that the sub-task-to-thread assignment is no longer decided on-the-fly and the task scheduling is unaffected. Primarily the tasks dispute the cores, and only then the sub-tasks of the selected task contend for the access to its threads of execution. Sub-tasks run to completion because the preemption cost is generally unbearable for fine-grained parallel programming frameworks. Advantages: the self-interference is not completely static but it is easier to bound. Disadvantages: complexity of the queue management; how to assign priorities to the sub-tasks?
2) **Partitioned scheduling.** Each sub-task can execute only in one core. Different sub-tasks of the same DAG task may execute on different cores. The dispatching of sub-task is still done on a FIFO or LIFO basis. Advantages: the number of self-interference scenario may be largely reduced; some cores can be dedicated to critical paths with minimal slack; simplifies WCET-related analysis. Disadvantages: mapping problem is very likely to be NP-Hard.
3) **Extra precedence constraints**. Based on the feedback of timing analysis, create a procedure to provide the application's code with additional precedence constraints between certain sub-tasks. This kind of extra-functional dependencies can be enforced using OmpSs [6]. Advantages: optimize worst-case paths by establishing some order of execution. Disadvantages: cumbersome iterative process; restricts the degree of parallelism.

Acknowledgments

This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER Research Centre); by the North Portugal Regional Operational Programme (ON.2 – O Novo Norte) under the National Strategic Reference Framework (NSRF), through the ERDF, and by National Funds through FCT/MEC, within project NORTE-07-0124-FEDER-000063 (BEST-CASE, New Frontiers); and by the European Union under the Seventh Framework Programme (FP7/2007-2013), grant agreement nr. 611016 (P-SOCRATES).

References

[1] K. Lakshmanan, S. Kato, and R. R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *the 31st RTSS*, 2010, pp. 259–268.

[2] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *the 32nd RTSS*, 2011, pp. 217–226.

[3] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *the 33rd RTSS*, 2012, pp. 63–72.

[4] J. C. Fonseca, V. Nélis, G. Raravi, and L. M. Pinho, "A multi-dag model for real-time parallel applications with conditional execution," in *the 30th ACM/SIGAPP SAC*, 2015.

[5] OpenMP Architecture Review Board, "OpenMP application program interface version 4.0," 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[6] S. Royuela, A. Duran, and X. Martorell, "Compiler automatic discovery of ompss task dependencies," in *the workshop on Languages and Compilers for Parallel Computing*, 2012.