CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

Response-Time Analysis of Fork/Join Tasks in Multiprocessor Systems

Cláudio Maia
Luís Nogueira
Luis Miguel Pinho
Marko Bertogna

CISTER-TR-130701
Version:
Date: 07-09-2013

Response-Time Analysis of Fork/Join Tasks in Multiprocessor Systems

Cláudio Maia, Luís Nogueira, Luis Miguel Pinho, Marko Bertogna

CISTER Research Unit

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail:

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

Abstract

This paper proposes a model to analyse the response-time of parallel real-time tasks. The presented model is based on the fork/join model which is internally used by user-level frameworks to exploit the parallelism provided by the underlying architecture. The model considers tasks with fixed priorities and allows real-time jobs to generate an arbitrary number of parallel threads. Each parallel thread is scheduled at runtime by taking into account the timing properties of the job that spawns it.

Response-Time Analysis of Fork/Join Tasks in Multiprocessor Systems

Cláudio Maia, Luís Nogueira, Luis Miguel Pinho
CISTER-ISEP / INESC-TEC
Porto, Portugal
Email:{crrm, lmn, lmp}@isep.ipp.pt

Marko Bertogna
University of Modena
Modena, Italy
Email:{marko.bertogna}@unimore.it

***Abstract*—This paper proposes a model to analyse the response-time of parallel real-time tasks. The presented model is based on the fork/join model which is internally used by user-level frameworks to exploit the parallelism provided by the underlying architecture. The model considers tasks with fixed priorities and allows real-time jobs to generate an arbitrary number of parallel threads. Each parallel thread is scheduled at runtime by taking into account the timing properties of the job that spawns it.**

***Keywords*-Parallel Task Model, Job-level Parallelism, Real-Time**

I. Introduction

Nowadays most of the computer devices (e.g. PC, tablet, mobile phone) rely on multiple processors, and future generations of processors are expected to integrate thousands of simple processors into a single chip [1]. The turning point in the computer industry begun in 2001 when Sun Microsystems and IBM (in a separate effort) produced the dual-core processors, and later on in 2006 this type of processors became a mainstream technology powered by Intel and AMD.

The causes behind the paradigm shift are mostly concerned with the physical limitations of computer chips as an increase in the operating frequencies of the chips leads to an increase in power consumption as well as temperature. Therefore, and to overcome such limitations, instead of increasing the operating frequencies of the chips, chip manufacturers increased the number of computing units operating in parallel at lower frequencies.

The real-time and embedded systems domain was no exception to this shift. Industries such as automotive, aerospace, and avionic, design complex systems that require powerful hardware capable of supporting their functional and non-functional software requirements. It is therefore extremely important for each of these industries to incorporate new and state-of-the-art multiprocessor architectures in their products, not only because of the added computing capacity but also due to the size, weight, and power constraints of the hardware itself.

Traditional real-time applications are scheduled in a multiprocessor system by well-studied approaches, as it has been shown in a recent survey [2]. Nonetheless, the paradigm shift revealed, among others, that the problem of scheduling real-time tasks is no longer a problem of scheduling sequential tasks (i.e. without intra-task parallelism). New models for computing real-time tasks that consider *job-level parallelism* or *intra-task parallelism* should be employed to maximise the performance of the applications, and therefore the utilisation of the available processing capacity.

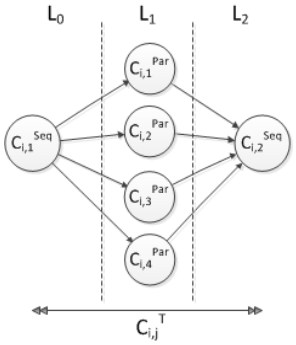

Figure 1. Job tree (DAG) of a job of Task τ_i

Frameworks such as Java Fork/Join [3] or OpenMP [4] help the application programmer divide the applications into sets of blocks which can then be scheduled in parallel in a multiprocessor architecture. Nevertheless, such parallelisation brings problems when application's timing constraints are considered, mainly because existing models are still restrictive (e.g. [5] and [6] transform a parallel task into a sequential task in order to apply well-known techniques used in traditional models).

The fork/join model is a model used by the above-mentioned frameworks to divide the applications into small blocks. In its basic form, the job of a task is composed of two sequential parts and a parallel part, as depicted in Figure 1. The first sequential part spawns several smaller units of execution that can be executed in parallel in order to exploit the inherent parallelism offered by multiprocessor architectures (in this paper these units are named *parallel jobs* or *p-jobs*). The number of parallel parts can be arbitrary large, as long as each parallel part is preceded by a sequential part and succeeded by another sequential part.

In this paper, the schedulability analysis of fork/join tasks from a response-time perspective is covered. We propose the

decomposition of a fork/join task into threads of execution, in order to improve the computation of the interference of each task on itself and on lower priority tasks. The proposed task model is composed of fixed-priority tasks where each real-time task can be a fork/join task, or a traditional sequential real-time task. The model assumes the graph of each task is known *a priori*, and the number of parallel jobs generated by each task can be greater than the number of cores in the platform.

The remainder of this paper is organised as follows. Section II presents the state of the art of parallel real-time tasks. Section III describes the system model. Section IV presents the possible approaches to perform response-time analysis of fork/join tasks. Finally, section V concludes the paper and presents the future work.

II. Related Work

Instead of considering a pure sequential task model, we consider the execution of parallel real-time tasks, i.e. job-level parallelism is allowed. In the domain of job-level parallelism, Goossens and Berten in [7] redefined a classification for different types of parallel tasks. Following this classification a job may be classified as rigid, moldable or malleable. A job is said to be rigid if the number of processors assigned to it is determined *a priori*, and this number does not change throughout job execution. A job is said to be moldable if the number of processors assigned to it is determined by the scheduler, but cannot change dynamically. Finally, a job is said to be malleable if the number of processors assigned to it is determined by the scheduler at runtime, and can change during the job's execution. Hence, a task is said to be: rigid if all of its jobs are rigid; moldable if all of its jobs are moldable; and malleable if all of its jobs are malleable. According to this classification, the parallel task model presented in this paper is considered to be composed of malleable tasks.

Malleable tasks were covered by Jansen [8], Collette et al. [9], and Korsgaard and Hendseth [10]. Jansen [8] focused on minimizing the makespan but without considering real-time constraints. Collette et al. [9] studied the problem of global scheduling of sporadic task systems on multiprocessors considering job-level parallelism. Korsgaard and Hendseth [10] proposed a sustainable schedulability test for malleable tasks scheduled with global Earliest Deadline First (EDF).

More recently, Lakshmanan et al. [5] and Saifullah et al. [6] focused on the study of scheduling fork/join tasks. Lakshmanan et al. [5] studied the scheduling of periodic real-time tasks that follow a fork-join structure on multiprocessor systems. In their proposed model, each parallel task is divided into a series of sequential and parallel segments, where all parallel segments must have the same number of threads and this number cannot be greater than the number of processors in the system. Moreover, the authors propose the task stretch transform algorithm in order to schedule fork/join tasks using traditional techniques. Saifullah et al. [6] present a synchronous task model for the scheduling of parallel real-time tasks with a fork-join structure. This model does not have any limitations on the number of parallel threads per segment and therefore is more general than [5]. The authors also proposed an algorithm to decompose the tasks into sequential tasks in order to use traditional schedulability analysis approaches.

III. System Model

We consider the problem of scheduling independent jobs on a system that comprises m identical processors with uniform memory access. In our model, a job is allowed to execute in more than one core at the same instant. A fully preemptive system is assumed where any job executing may be preempted at any instant and resumed later without any cost. At any given instant, the jobs with the highest priority among the ready jobs are the ones executing in the cores.

Let $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$ denote the set of n periodic tasks. Each task τ_i in the task set τ is characterised by a period T_i, a worst-case execution time requirement C_i, and a relative deadline D_i. Each task releases an infinite sequence of jobs at periodic time intervals separated by at least T_i time units. Each job has an implicit deadline equal to $D_i = T_i$ and a worst-case execution time requirement equal to C_i.

During execution, a job of τ_i may spawn a set of k "sub-jobs", denoted by *parallel jobs* or *p-jobs*, $pJ_i = \{pJ_{i,1}, pJ_{i,2}, ..., pJ_{i,k}\}$. The parallel jobs are sequential threads that decompose the job's workload so that its execution can be performed in parallel, therefore having the advantage of being executed in different processors in the same time instant (see Figure 1). Thus, each job has a set of instructions that are executed sequentially, and may have a set of instructions that can be executed in parallel upon m processors, i.e. a sequential part and a parallel part.

Note that the worst-case execution time C_i is equivalent to the time it takes to execute a job of τ_i in a single processor without preemption, i.e. executing all p-jobs sequentially. Let $C_{i,s}^{Seq}$ be the sequential worst-case execution time of the s-th sequential part of task τ_i, and $C_{i,p}^{Par}$ be the worst-case execution time of the p-th p-job spawned by task τ_i . Then, $C_i = \sum_{s=1}^{h} C_{i,s}^{Seq} + \sum_{p=1}^{k} C_{i,p}^{Par}$, where h and k are the number of sequential and parallel parts of τ_i, respectively.

Each p-job instance $pJ_{i,p}$ inherits the timing properties from the job that spawns it. Thus, the p-th instance of a p-job is characterised by the same period T_i and relative deadline D_i of the parent job. In this model, parallel jobs are independent, and with the exception of the processors, there are no other shared resources or critical sections.

The task structure is represented by a directed acyclic graph (DAG), denoted as $G_j = (V, E)$, as depicted in Figure 1. Each element in the set of vertices V represents the sequential parts of a job and the p-jobs spawned during

the execution. Each vertex has an associated worst-case execution time. Each element in the set of edges E represents the communication path between two vertices, v_i and v_j in the set V, i.e. $v_i, v_j \in V$. The proposed model does not take into account any communication cost between any two nodes in the graph. Nevertheless, a partial order in the execution is imposed which is deemed correct from the relation that exists between a job and its spawned p-jobs.

The *minimum execution time* P_i of a job j is defined to be the longest execution path in the task graph from the root vertex to the leaves, i.e. the critical path length. Formally, P_i is defined as $P_i = \sum_{v \in L_l} max(C_{i,v}), l = 0, 1, ..., L$, where v represents the v^{th} vertex that is part of level L_l and L denotes the number of levels in the graph (see Figure 1).

The *utilisation* u_i of task τ_i is the ratio between the task's execution time and period, $u_i = \frac{C_i}{T_i}$. For the task set τ, the *total utilisation factor* is defined as $U(\tau) = \sum_{i=1}^{n} \frac{C_i}{T_i}$. For implicit-deadline sequential task sets, a necessary and sufficient condition for feasibility is $U(\tau) \leq m$ ([11]). Nevertheless, for fork/join tasks this condition is only necessary [5]. It is important to mention that the fork/join model allows a task to have a utilisation larger than 100% while assuring that the cumulative utilization of the task set is no greater than m. This property prevents the serialisation of certain task sets to the implicit-deadline sequential case, as it would deem these task sets unschedulable.

IV. Response-Time Analysis

The response-time of a job is the amount of time that elapses between the release of the job and its completion time. From a real-time systems perspective, guaranteeing that the response-time of the tasks in the task set does not exceed their deadlines for all possible arrivals of the tasks, assures that the system is schedulable.

Lakshmanan et al. [5] analyse fork/join tasks from a feasibility perspective and provides the best-case and worst-case fork/join task structure. The best-case task structure is composed of m p-jobs which can be executed in parallel by fully utilising the m cores provided by the platform, with a cumulative utilisation of the task set that does not exceed m. The worst-case structure is based on infeasible task sets with a cumulative utilization closer to 100%, regardless of the number of processors present in the platform. An example of such task sets is given by taking a fork/join task with an implicit deadline equal to the *minimum execution time* and a short parallel region spanning all the cores in the platform; when this task is scheduled along with a sequential task with an arbitrarily small utilization and a deadline equal to the parallel region of the fork/join task, a deadline can be missed with a cumulative utilisation close to 100%.

From a response-time analysis perspective, the worst-case response-time of a fork/join task does not only depend on the interference caused by other higher priority jobs, but also on the precedence constraints between serial and parallel stages

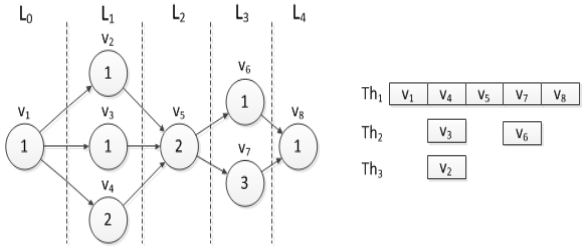

Figure 2. Decomposition approach example

of the task itself, as well as on the degree of parallelism of each stage and of the architecture.

In this paper we propose a global fixed-priority approach for fork/join tasks in which all the ready jobs/p-jobs are inserted in a global queue from where m processors pick the highest priority m jobs/p-jobs. In order to perform the response-time analysis of such tasks, we propose two possible approaches. The first approach considers for each job its execution time and the interference that it suffers from higher priority tasks. If the system is schedulable, the interference is bounded and the job execution time plus the imposed interference is always less than or equal to the job's deadline. If a job finishes its execution before its deadline, the available slack, given by the difference between the deadline D_i and the response time R_i, can be used to refine the computation of the worst-case interfering workload for other tasks, similarly to the method presented for sequential task sets in [12]. Moreover, a further refinement for fork/join tasks can be given by examining the response-time of each parallel stage of a task, allowing a tighter estimation of the interference imposed by such a task.

A second approach to response-time analysis is to consider a novel decomposition approach of a fork/join task, as depicted in Figure 2. In this approach, the fork/join task is decomposed into a set of threads of execution. There is a main thread of execution (Th_1 in the example) which is composed of all the sequential parts and the worst-case parallel part of each level L_i with an execution time of P_i. The remaining threads of execution (Th_2 and Th_3 in the example) are composed of sets of parallel jobs belonging to different levels in the graph, picking one parallel job from the remaining ones in each level L_i. The total number of threads of execution is given by the maximum out degree of all the sequential nodes in the graph. The algorithm that performs the decomposition is depicted in Algorithm 1.

Once the task decomposition into different threads is done, classic methods to bound the interfering contribution of each sequential thread can be applied, e.g., limiting the carry-in contributions to at most $m-1$ threads [13], limiting each thread interference to a task τ_i to at most $D_i - P_i + 1$ time-units [14], etc.

V. Conclusion

In this paper, we presented a model to analyse the response-time of fixed-priority fork/join tasks. Different

Algorithm 1 Task decomposition algorithm

$v_i \leftarrow RootVertex(G_j)$
function COMPUTE THREADS(v_i)
  $Create\ new\ list$
  $backtrackNode \leftarrow v_i$
  if $v_i\ not\ visited$ **then**
    $v_i \leftarrow visited$
    $add\ v_i\ to\ list$
    $VisitedNodes \leftarrow VisitedNodes + 1$
  end if
  for each $v_j\ in\ G_j\ such\ that\ v_i, v_j\ is\ an\ edge$ and $v_j\ is\ ordered\ in\ nonincreasing\ order\ by\ WCET$ **do**
    if $v_j\ not\ visited$ **then**
      $add\ v_j\ to\ list$
      $v_j \leftarrow visited$
      $v_i \leftarrow v_j$
      $VisitedNodes \leftarrow VisitedNodes + 1$
    else
      if $inDegree\ of\ v_j > 1$ **then**
        $v_i \leftarrow v_j$
      end if
    end if
  end for
  if $VisitedNodes < |V|$ **then**
    $Compute\ threads\ (\ backtrackNode\ )$
  end if
end function

methods are proposed to improve the response-time analysis of such task systems, including the decomposition of each fork/join task into sequential threads of execution. Future work includes a complete schedulability analysis of such tasks, considering the worst-case situations that lead to the largest possible interference for each task. Moreover, we believe the proposed model can be easily adapted to support strict fork/join tasks, where nested parallelism is allowed, and other general parallel task models.

ACKNOWLEDGMENT

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within projects Ref. FCOMP-01-0124-FEDER-022701 (CISTER), ref. FCOMP-01-0124-FEDER-020447 (REGAIN) and ref. FCOMP-01-0124-FEDER-012988 (SENODS); also by FCT and by ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/88834/2012.

REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[2] 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.

[3] D. Lea, "A java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*, ser. JAVA '00, 2000, pp. 36–43.

[4] OpenMP, "Openmp," http://openmp.org/, Jun. 2011.

[5] K. Lakshmanan, S. Kato, and R. R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, ser. RTSS '10, 2010, pp. 259–268.

[6] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," *Real-Time Systems Symposium, IEEE International*, vol. 0, pp. 217–226, 2011.

[7] J. Goossens and V. Berten, "Gang ftp scheduling of periodic and parallel rigid real-time tasks," *CoRR*, vol. abs/1006.2617, 2010.

[8] K. Jansen, "Scheduling malleable parallel tasks: An asymptotic fully polynomial-time approximation scheme," in *Proceedings of the 10th Annual European Symposium on Algorithms*, ser. ESA '02, 2002, pp. 562–573.

[9] S. Collette, L. Cucu, and J. Goossens, "Integrating job parallelism in real-time scheduling theory," *Inf. Process. Lett.*, vol. 106, no. 5, pp. 180–187, May 2008.

[10] M. Korsgaard and S. Hendseth, "Schedulability analysis of malleable tasks with arbitrary parallel structure," *Real-Time Computing Systems and Applications, International Workshop on*, vol. 1, pp. 3–14, 2011.

[11] W. A. Horn, "Some simple scheduling algorithms," *Naval Research Logistics Quarterly*, vol. 21, no. 1.

[12] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, 2007, pp. 149–160.

[13] N. Guan, M. Stigge, W. Yi, and G. Yu, "New response time bounds for fixed priority multiprocessor scheduling," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, 2009, pp. 387–397.

[14] M. Bertogna, M. Cirinei, and G. Lipari, "Improved schedulability analysis of edf on multiprocessor platforms," in *Real-Time Systems, 2005. (ECRTS 2005). Proceedings. 17th Euromicro Conference on*, 2005, pp. 209–218.