CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

Scheduling Parallel Real-Time Tasks using a Fixed-Priority Work-Stealing Algorithm on Multiprocessors

Cláudio Maia

Luís Nogueira

Luis Miguel Pinho

CISTER-TR-130607

Version:

Date: 06-19-2013

Scheduling Parallel Real-Time Tasks using a Fixed-Priority Work-Stealing Algorithm on Multiprocessors

Cláudio Maia, Luís Nogueira, Luis Miguel Pinho

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 for scheduling parallel real-time tasks. The proposed model uses a work-stealing approach to schedule real-time parallel task sets at runtime, where each job may present a nested fork-join structure, generate an arbitrary number of parallel jobs, and each parallel job inherits the timing properties of the job that spawns it.

© CISTER Research Unit
www.cister.isep.ipp.pt

Scheduling Parallel Real-Time Tasks using a Fixed-Priority Work-Stealing Algorithm on Multiprocessors

Cláudio Maia, Luís Nogueira, Luis Miguel Pinho
CISTER/INESC TEC, ISEP
Polytechnic Institute of Porto
Portugal
{*crrm, lmn, lmp*}*@isep.ipp.pt*

***Abstract*—This paper proposes a model for scheduling parallel real-time tasks. The proposed model uses a work-stealing approach to schedule real-time parallel task sets at runtime, where each job may present a nested fork-join structure, generate an arbitrary number of parallel jobs, and each parallel job inherits the timing properties of the job that spawns it.**

***Keywords*-Parallel Real-time, Work-Stealing, Scheduling**

I. Introduction

Multicores offer an opportunity to maximise performance and, through parallelism, execute more complex and computing-intensive tasks whose stringent timing constraints can be difficult to meet through traditional multiprocessing of sequential task models. Nevertheless, most results in multiprocessor real-time scheduling concentrate on sequential tasks running on multiple processors or cores [1]. Moreover, current parallel models are still restrictive in nature, i.e. they are static, less general, and based on task decomposition. Task decomposition enables the application of well-known schedulability analysis techniques, but it requires all the information to be known *a priori*, therefore exploiting parallelism only when the system is not schedulable in a sequential way.

There exists a number of applications that take advantage of multiprocessor architectures by exploiting the features provided by frameworks such as Cilk [2] and OpenMP [3]. These frameworks allow the application programmers to divide the applications into smaller blocks which can be assigned to different CPUs so that they can execute in parallel. In order to dynamically schedule the parallel blocks in a load balancing manner, such frameworks use the work-stealing algorithm, proposed by Blumofe and Leiserson [4]. This algorithm is formally proven to be asymptotically optimal for scheduling fully-strict computations, i.e. computations in which all join dependencies from a thread point to its parent thread.

Traditional work-stealing considers a pool of worker threads (the schedulable entities). Each worker thread possess its own local double-ended queue (deque) and is mapped to a core. A task may generate new subtasks and such subtasks are enqueued into the local deque of the worker thread. Worker threads execute tasks from their own deques in a LIFO order, but whenever they become idle they steal work from a randomly chosen core deque's tail (accessed in a FIFO order), i.e. tasks or subtasks that were enqueued first in the deque of the chosen core.

Work-stealing has the advantages of reducing task contention, load balancing the workloads, and preserving data locality [4]. Nevertheless, it cannot be directly applied in real-time systems as it may cause priority inversion, which may eventually lead to deadline misses depending on the timing properties of the task sets. A motivational example to this problem is presented next.

Example 1: Assume a system with two cores and two worker threads, WT_1 and WT_2. In core 1, WT_1 is executing a medium priority task (T_m), and in core 2 WT_2 is executing a high priority task (T_h). Now let us assume that T_m generates medium priority subtasks which are enqueued into core 1's deque. If at this particular time instant a new high priority task is ready for execution (T_{h2}), T_m is preempted. If T_{h2} also generates subtasks, these subtasks are enqueued into core 1's deque, more specifically pushing older subtasks (of medium priority) to the end of the queue. If at this time instant, core 2 becomes idle, it is allowed to steal work from the core 1's deque. By doing so, the candidates for stealing are the subtasks of T_m.

This is clearly an example of priority inversion. In order to have a correct behaviour, both cores should have been executing the subtasks generated by T_{h2} in parallel. However, on the other hand if stealing was not allowed, core 2 would be idle and the system would be wasting resources.

It is our belief that a modification of work-stealing may be useful in real-time systems. In this paper we present a scheduling model that is a variation of the traditional work-stealing approach in the sense that it considers the real-time behaviour of parallel applications which may be modelled as real-time nested fork-join tasks.

The objectives of the proposed approach are the scheduling of tasks that inherently present a parallel behaviour, and the support for dynamic systems where the internal parallel structure may not be known in advance. Moreover, by using

work-stealing, the approach is able to take advantage of data locality and load balancing of the workloads, which may reveal useful in environments composed of real-time nested fork-join tasks.

The remainder of this paper is organised as follows. Section II presents the parallel real-time tasks literature. Section III describes the system model. In Section IV, the approach is described and finally, Section V concludes the paper by presenting future work.

II. Related Work

Considering the scheduling of parallel real-time tasks, Goossens and Berten [5] redefined a classification from the parallel literature. According to this classification, the model presented in this paper is considered to be composed of malleable tasks (i.e. the number of processors assigned to a job is determined by the scheduler at runtime). Although we consider a set of fixed-priority parallel real-time tasks, the processor assignment of the jobs generated by the parallel tasks is performed in a dynamic manner.

The study of scheduling malleable tasks was covered by Jansen [6] with the objective of minimizing the makespan. Collette et al. [7] study the problem of the global scheduling of sporadic task systems on multiprocessor platforms considering job-level parallelism. Korsgaard and Hendseth [8] proposed a pessimist (although sustainable) schedulability test for task systems composed of malleable task scheduled with global Earliest Deadline First (EDF).

Drozdowski [9] considers the problem of scheduling parallel tasks with the objective of minimizing the makespan. Han and Lin [10] prove that the problem of scheduling parallelisable jobs with a fixed priority is NP-Hard. Manimaran et al. [11] proposed a variant of non-preemptive EDF that considers parallel real-time tasks. Goossens and Berten [5] proposed a scheduling algorithm for parallel real-time tasks based on gang scheduling.

Lakshmanan et al. [12] proposed a scheduling algorithm to schedule periodic real-time tasks that follow a fork-join structure on multiprocessor systems. This model is restrictive in the sense that all parallel segments have the same number of threads and this number cannot be greater than the number of processors in the system. Saifullah et al. [13] build their work upon previous work ([12]) and present a more general synchronous task model for scheduling parallel real-time tasks with a fork-join structure. However, their model does not present any limitations on the number of parallel threads per segment or even on the number of parallel threads executing at the same time in the system. Both models use task decomposition to schedule the tasks in order to apply well-known schedulability analysis techniques.

Concerning the application of work-stealing into real-time systems, Mattheis et al. [14] provide an upper bound on the latency for different work-stealing strategies suitable for stream processing applications, however without considering system predictability. Nogueira and Pinho [15] propose a server-based approach combined with work-stealing to support parallel tasks. More recently, Nogueira et al. [16] present an approach that combines global EDF with work-stealing, however this approach only covers simple fork-join tasks. The approach presented in this paper differs from the previous approaches in the sense that it presents a model that combines work-stealing with real-time for scheduling fixed-priority nested fork-join tasks.

III. System Model

We consider the problem of scheduling independent jobs on a system comprised of m identical processors/cores with uniform memory access (UMA). A fully preemptive system is assumed where any job executing may be preempted at any time instant and resumed later without any cost. At any given time instant, the jobs with the highest priority among the ready jobs are the ones executing.

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. Furthermore, each task releases a sequence of jobs, $J_i = \{j_1, j_2, ..., j_j\}$, at periodic time intervals separated by T_i time units. Each job has an implicit deadline equal to $D_i = T_i$.

During execution, the jth job may spawn a set of k *parallel jobs* or in short *p-jobs*, $pJ_{i,j} = \{pJ_{i,j,1}, pJ_{i,j,2}, ..., pJ_{i,j,k}\}$. Parallel jobs are sequential threads that decompose the job's workload so that its execution can be parallelised. 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 cores, i.e. sequential and parallel parts.

Each job as well as p-jobs may spawn other p-jobs (i.e. nested structure), and each p-job instance may have a variable worst-case execution time. Let C_j^T denote the total worst-case execution time of job j, C_j^{Seq} the total sequential worst-case execution time of j, and $C_{j,k}^{Par}$ the worst-case execution time of the kth p-job spawned by job j. The *total worst-case execution time* of job j is given by: $C_j^T = C_j^{Seq} + \sum_{l=1}^{k} C_{j,l}^{Par}$.

Concerning the timing properties, each p-job instance $pJ_{i,j,k}$ inherits the timing properties from the jth job that spawns it. Thus, the kth 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.

For the purpose of modelling jobs and p-jobs, it is possible to represent the job tree of the jth job as 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 a sequential part of a job or a p-job spawned during the execution of job j. Furthermore, each vertex has associated

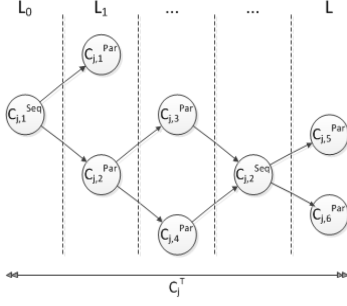

Figure 1. Job tree (DAG) of the jth job of Task τ_i

with it its 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. $v_i, v_j \in V$.

As the system is comprised of identical processors with UMA, the proposed model does not take into account any communication cost between any two nodes in the graph. Therefore, communication costs are assumed to be zero. 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, i.e each job/p-job is parent of the spawned p-jobs in the sense that it is responsible for the creation of the p-jobs.

The *maximum execution time* of a job j is defined to be C_j^T, i.e. its total worst-case execution time. This interval of time represents the time a job takes to execute in a single processor without preemption.

The *minimum execution time* P_j^T 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_j^T is defined as follows: $P_j^T = \sum_{v_v \in L_i} max(C_{j,v}), i = 0, 1, ..., L$, where v_v represents the vth vertex that is part of level L_i and L denotes the number of levels in the graph, as depicted in Figure 1. The interpretation for P_j^T is that even if the number of processors in the system is infinite, the jth job takes at least P_j^T units of execution time to complete. If $P_j^T > D_i$ then the system is not schedulable (see Proposition 1 in [12]).

The *utilisation factor* u_j of job j, i.e. the fraction of processor time that is dedicated to the execution of the jth job, is the ratio of the job's execution time to its period, and is defined as follows: $u_j = \frac{C_j^T}{T_i}$. 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 periodic task sets, a necessary and sufficient condition for feasibility is $U(\tau) \leq m$ ([17]).

IV. Real-Time Work Stealing

The proposed approach is a modified version of the work-stealing algorithm. Instead of using a single non-priority deque to store ready tasks, which would cause priority inversion as shown previously, we propose the addition of per-core priority deques. The priority deques are ordered by priority, so instead of stealing randomly, idle cores steal

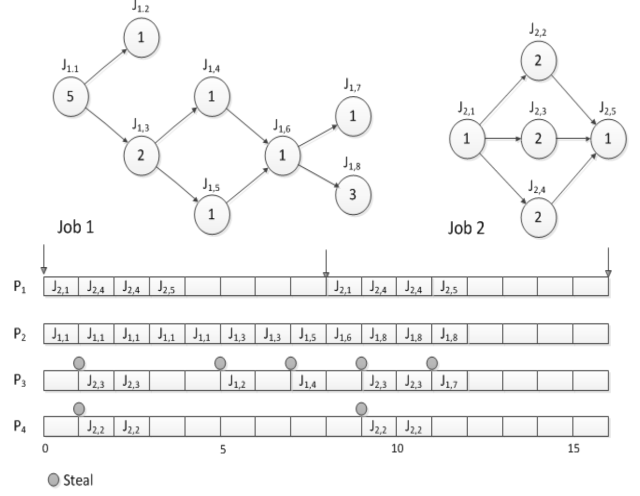

Figure 2. Example of real-time work-stealing

jobs from the highest priority deque. An example of the behaviour of the algorithm is presented next.

Example 2: Figure 2 depicts a scenario of a system consisting of four identical processors and a possible schedule for two jobs. Only the first 16 time units are depicted in the figure. Job 1 has a $C_j^T = 15$ and $T_i = 20$, and Job 2 presents a typical fork-join structure with $C_j^T = 8$ and $T_i = 8$. Jobs are released at $t = 0$ and Job 2, with highest priority, is assigned to P_1, and Job 1 to P_2 respectively.

In the first period of Job 2 (i.e. from $t = 0$ to $t = 8$), processors P_3 and P_4 are idle and therefore help P_1 executing the p-jobs spawned by Job 2 at $t = 1$ and the p-jobs spawned by Job 1 at $t = 5$ and $t = 7$.

In the second period of Job 2 (i.e. from $t = 8$ to $t = 16$), at $t = 9$ Job 2 spawns three p-jobs ($J_{2,2}$, $J_{2,3}$ and $J_{2,4}$) and Job 1 spawns two p-jobs ($J_{1,7}$ and $J_{1,8}$). As Job 2 has a higher priority than Job 1, processors P_3 and P_4 will steal p-jobs from the highest priority deque, and therefore both steal $J_{2,3}$, $J_{2,2}$, respectively. At $t = 11$, there is still one p-job left in P_1's deque ($J_{1,7}$), which is stolen by P_3.

Concerning the details of the approach, there is a Global Submission Queue (GSQ) ordered by priority. The ready jobs from the task set are enqueued in the GSQ. From this queue, each core chooses the next job for execution, the highest priority jobs execute first. Once a job is picked up from the GSQ, it is executed in that core until completion or until preempted. In case a preemption occurs, the processor enqueues the preempted job/p-job in the respective local deque, and executes the higher priority job/p-job that was released, leaving the execution of the preempted job to be resumed in this core or another core depending on the priority of other jobs. Preemptions occur when a higher priority job is released or a higher priority job spawns new p-jobs. In the latter case, and in order to avoid priority inversion, such p-jobs must preempt other lower priority jobs/p-jobs executing in the system.

Parallel jobs are stored in the local deques according to their priority. As a job may spawn an arbitrary number of p-jobs, local queues are used to store them in order to benefit from data locality. As the system evolves through time, p-jobs are executed by a core until no more p-jobs exist. When a core is idle, it steals the highest priority job/p-job from the local deque of the core that has this job/p-job.

The advantages of this model are the following: (i) when a job/p-job is stolen from the other cores' deques the migration overhead is supported by the idle core; (ii) keeping jobs/p-jobs in local deques maximises data locality in caches; (iii) load balancing workloads considering their real-time properties.

V. Future Work

This paper presents a model that combines work-stealing with real-time with the objective of scheduling task sets composed of nested fork-join tasks. Such tasks present several challenges from a real-time systems perspective due to their characteristics.

From a schedulability analysis perspective, important decisions are yet to be considered namely: (i) precedence constraints among tasks; (ii) worst-case behaviour of a nested fork/join considering stealing; (iii) migration costs and preemption costs. Such decisions will influence the schedulability conditions that must be derived in order to assure that a particular task set can be scheduled with the proposed variant of work-stealing for real-time systems.

Concerning the algorithmic behaviour, other alternatives may be applied in order to obtain better schedulability results, as for instance a stolen p-job may not be the target of a preemption and therefore execute in a non-premptive fashion, or even allow for random steals as long as these can be bounded in order to assure system schedulability.

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) and ref. FCOMP-01-0124-FEDER-020447 (REGAIN); 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] 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.

[2] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, pp. 212–223, May 1998.

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] M. Drozdowski, "Real-time scheduling of linear speedup parallel tasks," *Inf. Process. Lett.*, vol. 57, no. 1, pp. 35–40, Jan. 1996.

[10] C.-C. Han and K.-J. Lin, "Scheduling parallelizable jobs on multiprocessors," in *IEEE Real-Time Systems Symposium*, 1989, pp. 59–67.

[11] G. Manimaran, C. S. R. Murthy, and K. Ramamritham, "A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems," *Real-Time Syst.*, vol. 15, no. 1, pp. 39–60, Jul. 1998.

[12] 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.

[13] 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.

[14] S. Mattheis, T. Schuele, A. Raabe, T. Henties, and U. Gleim, "Work stealing strategies for parallel stream processing in soft real-time systems," in *Proceedings of the 25th international conference on Architecture of Computing Systems*, ser. ARCS'12, 2012, pp. 172–183.

[15] L. Nogueira and L. M. Pinho, "Server-based scheduling of parallel real-time tasks," in *Proceedings of the tenth ACM international conference on Embedded software*, ser. EMSOFT '12, 2012, pp. 73–82.

[16] L. Nogueira, J. Fonseca, C. Maia, and L. Pinho, "Dynamic global scheduling of parallel real-time tasks," in *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, 2012, pp. 500–507.

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