Partitioned Scheduling of Parallel Real-time Tasks on Multiprocessor Systems

Frédéric Fauberteau[†] Serge Midonnet[†] Manar Qamhieh[*†]
[†]*Université Paris-Est*
LIGM, UMR CNRS 8049
{frederic.fauberteau,serge.midonnet}@univ-paris-est.fr
[*]*ECE*
qamhieh@ece.fr

Abstract

In this paper, we focus on the scheduling of periodic fork-join real-time tasks on multiprocessor systems. Parallel real-time tasks in the fork-join model have strict parallel segments without laxity. We propose a partitioned scheduling algorithm which increases the laxity of the parallel segments and therefore the schedulability of tasksets of this model. A similar algorithm has been proposed in the literature but it produces job migrations. Our algorithm eliminates the use of job migrations in order to create a portable algorithm that can be implemented on a standard Linux kernel. Results of extensive simulations are provided in order to analyze the schedulability of the proposed algorithm, and to provide comparisons with the other algorithm proposed in the literature.

1. Introduction

Physical constraints of the manufacturing process, such as chip size and heating, are driving a tendency for chip manufacturers to build multi-processors and multi-core processors to further increase computational performance . Because of this, parallel programming has received increased attention, despite being in use for many years.

The concept of parallel programming is to write a code that can be executed simultaneously on different processors. These programs are typically harder to write than sequential ones, since it is necessary to keep the parallel partitions independent in order to execute them correctly on different processors at the same time. This condition might be affected by reasons like a shortage in processors, which requires the use of partitioning. In real-time systems and as we found in literature [1], [2], a parallel task can be:

- rigid if the number of processors is assigned externally to the scheduler and can't be changed during execution,
- moldable if the number of processors is assigned by the scheduler and can't be changed during execution,
- malleable if the number of processors can be changed by the scheduler during execution.

For a practical implementation, there are several libraries, APIs and models in existence created specifically for parallel programming, including POSIX threads and OpenMP, however they were not designed specifically for real-time systems.

In this paper we will work on periodic real-time tasks of fork-join structure, which is the same structure OpenMP is based on, and can be seen as a rigid type of real-time parallelism, since the number of processors is fixed by the task model. And we will propose a partitioned scheduling algorithm for this model of tasks, based on a prior work in the same field, without the use of job migration in order to be implemented on as standard Linux kernel.

The remainder of this paper is organized as follows: in Section 2, we present our task model. Section 3 describes a related work on the same model. Section 4 explains the proposed algorithm followed by the analysis in section 5. and we finish with the perspective and the conclusion in sections 6 and 7.

2. Fork-Join Model

As shown in Figure 1, the fork-join model defines a task as a collection of several segments, both sequential and parallel. This task always starts by a sequential segment, which then forks into several parallel independent threads (parallel segment) that finally join in another sequential segment. It is important to note that all parallel segments in a task share the same number of processors, and it should be mentioned that the tasks of this model have implicit deadlines (the deadline of a task equals its period).

Here is an example of the fork-join model: $\tau_i = ((C_i^1, P_i^2, C_i^3, ..., P_i^{s_i-1}, C_i^{s_i}), m_i, T_i)$ where:

- s_i is the total number of segments (sequential and parallel) and it is an odd number according to definition of the model,
- m_i is the number of parallel threads on which parallel segments will be executed. $m_i > 1$ for parallel segments, and equal to 1 for sequential segments.
- C_i^s is the Worst-Case Execution Time (WCET) of a sequential segment, where s is an odd number and $1 \leq s \leq s_i$,
- P_i^s is the WCET of a parallel segment, where s is an even number and $1 \leq s \leq s_i$,
- T_i is the period of the task.

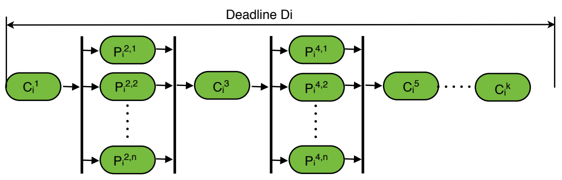


Figure 1. Fork-Join structure model.

What we notice about this model is the fact that by default all parallel segments have to finish their execution before the following sequential segment starts. Therefore these segments have strict laxity and their execution times are equal to their deadlines.

Figure 2 shows a fork-join task, which can also be represented according to the previous definition: (1, 2, 2, 3, 1), 3, 17).

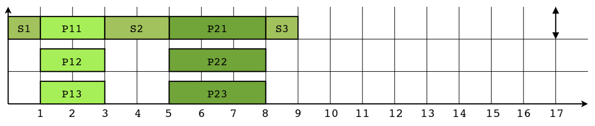


Figure 2. Example of fork-join task.

3. Related work

Due to the strict laxity of the parallel segments in the fork-join task model, Lakshmanan *et al.* in [3] proposed an algorithm to increase the laxity of the parallel segments by reducing the parallelism in the fork-join model whenever possible. Their algorithm stretches the main thread to its deadline, as shown in Figure 3. It aims to execute as many parallel segments as possible in the master string thread (the thread that contains the sequential segments, also considered as the entry and end point of the program). This master string will be stretched to its deadline so as to be executed on an exclusive processor with 100% processor utilization. What remains of the parallel segments will be distributed on the available processors using a partitioning algorithm called FBB-FFD (stands for Fisher Baruah Baker - First Fit Decreasing) [4].

This algorithm enhances the schedulability of parallel tasks of fork-join structure, by increasing the deadline of the parallel segments and getting rid of their strict execution time, as shown in Figure 2 and 3, parallel segment $P_{1,3'}$ has a deadline of 4 time units instead of 2, which was exactly the worst case execution time of that parallel segment. It then has to migrate to the master string so as to fill the master thread. This laxity in the deadline will increase the chances of the parallel segments to be scheduled using *FBB-FFD*, as will be clarified later in the analysis.

The number of job migrations in this algorithm could be 0, if the algorithm succeeded in scheduling all the parallel segments into the master string, creating a sequential task that will be executed on one processor. The other possibility for the number of job migrations is the number of parallel segments in the task, as shown in Figure 3 , both $P_{1,3}$ and $P_{2,3}$ are used to fill the slack time in the master string, and they both will migrate.

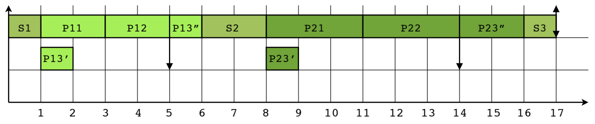


Figure 3. Task Stretch Transformation.

Task Stretch Transformation (TST) has a constraint when it comes to practical implementation, that is in order to achieve a fully stretched master string, job migration is inevitable. As shown in the Figure 3, segments $P_{1,3}$ and $P_{2,3}$ have to start execution on a certain processor. They will then migrate to the master string's processor in order to fill it. According to the paper, this can be easily implemented on a specific Linux system called *Linux/RK* [5] (stands for Linux Resource Kernel), which is a real-time extension to the Linux kernel to support the abstractions of a resource kernel. But our idea is to implement this algorithm directly on a standard Linux enhanced with the *PREEMPT_RT* kernel patch.

4. Segment Stretch Transformation

In order to eliminate the use of job migration, some modifications have to be done on the original pseudo-code, which we called *Segment Stretch Transformation* (SST). The basic idea of TST stayed the same, by trying to avoid the fork-join model by stretching the master string, but now it will be filled only with complete parallel segments with no migration. The following example will better explain the modifications.

We have a task τ_1 = ((1, 2, 2, 3, 1), 3, 17) as shown in Figure 2, which is a typical fork-join task. In Figure 2 we show the result of applying TST on τ_1. We notice that segment $P_{1,3}$ and $P_{2,3}$ have to be executed on 2 processors. But in SST and as shown in Figure 4, the master string is only filled by complete parallel segments. Even though the master string is not fully stretched (there still 1 unit of time not used before the deadline), the parallel segment $P_{2,3}$ will not be used.

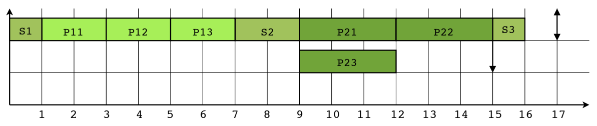


Figure 4. Segment Stretch Transformation.

In TST, the master string has to be filled with all of the parallel segments with equal partitions, which will increase the laxity of all the segments equally. But at the same time, it will increase the number of job migrations as well. In SST however, the master string will be initially filled with the pre-calculated number of parallel strings, then we check if we can add other single parallel strings to the master string (like $P_{1,3}$ in Figure 4). The remaining parallel segments of the task with the master string will be scheduled using the FBB-FFD partitioning algorithm. The laxity of the master string will increase since we did not fill it completely with parallel segments.

This transformation has changed the model type from being rigid into moldable, since the number of processors is assigned by the scheduler after creating the master string. And from a practical implementation point of view, the SST can be fully implemented on a standard Linux RT kernel with no special extensions or patches added, and by only using an ordinary function like *sched_set_affinity()*, each segment of the parallel task can be assigned to a specific processor according to the scheduling results of any partitioning algorithm (*e.g.* FBB-FFD).

5. Analysis

In order to provide a practical analysis for these algorithms, we are going to use *rtmsim* (stands for Real-Time Multiprocessor SIMulator). It is a free simulation software developed at *Université Paris-Est Marne-la-Vallée, France*. This simulation software helps analysing the performance of real-time scheduling algorithms by choosing a pre-coded approach to run in an extensive simulation.

Based on a simulation protocol of multiprocessor systems proposed by Davis *et al.* [6], which uses UUniFast as task-generation algorithm [7], we randomly generated 10,000 tasksets per processor's utilization varying from 0.025 to 0.975, each taskset of 16 parallel tasks, which makes it a total of 390,000 tasksets. FBB-FFD is used as scheduling algorithm.

We started the analysis by creating a dataset of parallel tasks of the fork-join model, and by using FBB-FFD directly to schedule this dataset, we observed the results shown in Figure 5(a) (the curve with the triangular points). But FBB-FFD failed to schedule the dataset beyond the processors' utilization of 0.1. This can be explained by knowing that FBB-FFD is using the following condition. For each task τ_i to be placed on processor k, the following condition has to be true:

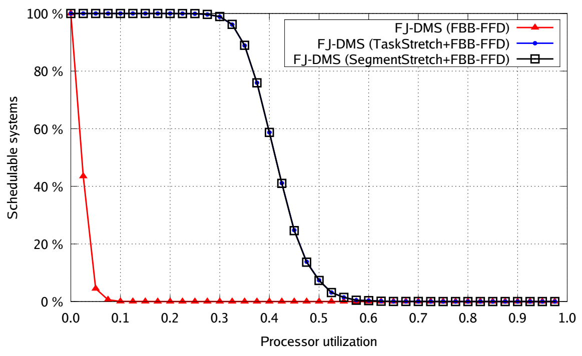
$$d_i - \sum_{\tau_j \in \tau(\pi_k)} RBF^*(\tau_j, d_i) \geq e_i$$

where τ_i is the task to be scheduled on processor k, π_k is the set of tasks already placed on processor k and $RBF^*(\tau_j, d_i) = e_j + \frac{e_j}{P_j} * d_i$

According to the previous condition, if the task to be scheduled has both an execution time and a deadline of the same value, then the condition will definitely fail if the processor has already other tasks to be executed on. And since parallel segments in the fork-join model always have execution times equal to their deadlines (Figure 2), then each parallel segment will need to be executed exclusively on a single processor.

But by looking at the characteristics of the parallel segments, we notice that they have offsets, which means that they will not all be scheduled at the same time. By using a suitable type of partitioning algorithm that can handle offsets we might be able to enhance the results of the simulation. FDD-RTA (First Fit Decreasing-Response Time Analysis) could be a good choice.

A second analysis was performed to compare TST and SST algorithms, by using the same model of extensive simulation described previously. The result of the simulation is shown in Figure 5(a), where TST is the curve with the round points, and SST is the



(a) Curves of comparison.

U_i	TST	SST	U_i	TST	SST
0.250	9991	9996	0.450	2477	2460
0.275	9967	9970	0.475	1380	1366
0.300	9887	9895	0.500	739	731
0.325	9614	9623	0.525	316	313
0.350	8889	8898	0.550	144	144
0.375	7596	7592	0.575	44	46
0.400	5872	5872	0.600	33	32
0.425	4110	4102	0.625	14	12

(b) Values of comparison.

Figure 5. Simulation results.

curve with square points. As we can see, there is no noticeable difference. There is a slight difference between these 2 algorithms as represented in Figure 5(b). The table shows the number of tasksets each algorithm succeeded to schedule for each utilization of the processor. The interesting result that we observe is the incomparability of these 2 algorithms.

6. Perspective

Sequential tasks in real-time systems are widely studied and analysed by a lot of algorithms designed specifically for this task model, therefore the transformation algorithms of parallel tasks like TST and SST are proposed. However, after applying the transformation and the creation of the master string, what remain of the parallel segments are considered as independent sequential tasks, without any consideration for their offset or their dependency on other segments, and they are scheduled by FBB-FFD scheduling algorithm which is not the most adaptive algorithm for parallel tasks of fork-join structure model.

In the future, we will focus on the parallel segments that are not included in the master string after applying the transformation algorithm, and we aim to provide a response time analysis for those segments in order to propose a specific scheduling algorithm deals with the specific constraints of this model of tasks (offset, dependency, ...). This can be done either by analysing the parallel segments as subtasks with fixed priority [8], or by creating slave strings from those parallel segments. Each one will be considered as a single task with an offset (which is the WCET of the first sequential segment) and suspension periods according to the number of the parallel segments in the slave string, and then apply a response time analysis on those tasks.

7. Conclusion

In this paper, we presented an algorithm that transforms parallel tasks of fork-join structure in order to increase the laxity of the parallel segments, and to eliminate the use of job migration, which makes it possible to implement on the standard Linux kernel. The analysis of this algorithm is performed by using extensive simulations in order to compare its performance with the original taskset model and TST algorithm. Our next step and as described in perspective section, will be to propose a scheduling algorithm for the segments of the parallel task under the fork-join structure after applying the transformation algorithm SST.

References

[1] J. Goossens and V. Berten, "Gang ftp scheduling of periodic and parallel rigid real-time tasks," in *Proc. of RTNS*, 2010, pp. 189–196.

[2] S. Kato and Y. Ishikawa, "Gang edf scheduling of parallel task systems," in *Proc. of RTSS*, 2009, pp. 459–468.

[3] K. Lakshmanan, S. Kato, and R. (Raj) Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Proc. of RTSS*, 2010, pp. 259–268.

[4] N. Fisher, S. Baruah, and T. P. Baker, "The partitioned scheduling of sporadic tasks according to static-priorities," in *Proc. of ECRTS*, 2006, pp. 118–127.

[5] S. Oikawa and R. Rajkumar, "Portable rk: A portable resource kernel for guaranteed and enforced timing behavior," in *Proc. of RTAS*, 1999, p. 111.

[6] R. I. Davis and A. Burns, "Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," *Real-Time Systems*, vol. 47, no. 1, pp. 1–40, 2010.

[7] E. Bini and G. C. Buttazzo, "Biasing effects in schedulability measures," in *Proceedings of the 16th Euromicro Conference on Real-time Systems (ECRTS)*. IEEE Computer Society, 2004, pp. 196–203.

[8] M. G. Harbour, M. H. Klein, and J. P. Lehoczky, "Fixed priority scheduling periodic tasks with varying execution priority," in *Proceedings of the 12th IEEE RTSS*, 1991, pp. 116–128.