HAL
archives-ouvertes.fr

Partitioned Scheduling of Parallel Real-time Tasks on Multiprocessor Systems

Frédéric Fauberteau, Serge Midonnet, Manar Qamhieh

▶ **To cite this version:**

Frédéric Fauberteau, Serge Midonnet, Manar Qamhieh. Partitioned Scheduling of Parallel Real-time Tasks on Multiprocessor Systems. 23rd Euromicro Conference on Real-Time Systems (ECRTS'11), 2011, United States. pp.1-4. hal-00620398

HAL Id: hal-00620398

https://hal-upec-upem.archives-ouvertes.fr/hal-00620398

Submitted on 19 Mar 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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 of fork-join model have strict parallel segments with no 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. Ours avoid 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 compared to the previous one.

1. Introduction

Chip manufacturers are tending to build multiprocessors and multi-core processors as a solution to overcome the physical constrains of the manufacturing process, such as chip's size and heating. Because of that, parallel programming has gained a higher importance although it has been used for many years.

The concept of parallel programming is to write a code that can be executed simultaneously on different processors, and usually these programs are harder to be written 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 not be affected by reasons like 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.

From practical implementation's point of view, there exist certain libraries, APIs and models created specially for parallel programming like POSIX threads[3] and OpenMP [4], except those are not designed for real-time systems normally, but in this paper we will work on periodic real-time tasks of fork-join structure, the same structure OpenMP is based on, and which can be seen as a rigid type of real-time parallelism.

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 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, and this task always starts by a sequential segment, then it forks into several parallel independent threads (parallel segment) to be joined finally in another sequential segment. It is important to note that all parallel segments in a task shares the same number of processors, and it should be mentioned that tasks of this model have implicit deadline (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 equals to 1 for sequential segments.
- C_i^s is the Worst-Case Execution Time (WCET) of sequential segment, where s is an odd number and $1 \leq s \leq s_i$,
- P_i^s is the WCET of parallel segment, where s is an even number and $1 \leq s \leq s_i$,
- T_i is the period of the task.

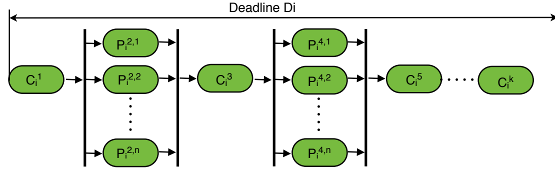


Figure 1. Fork-Join structure model.

What we can 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 equal to their deadlines.

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

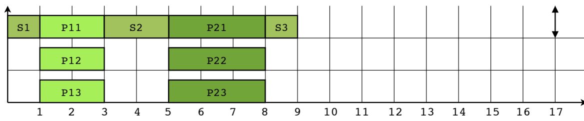


Figure 2. Example of fork-join task.

3. Related work

Due to the strict laxity of the parallel segments in the fork-join task model, Lakshmanan *et al.* in [5] propose an algorithm to increase the laxity of the parallel segments by reducing the parallelism in the fork-join model when possible. Their algorithm stretches the main thread to its deadline, as shown in Figure 3. It aims to execute as much parallel segments as possible in the master string thread (the thread that contains the sequential segments and it is 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's utilization. What remains of the parallel segments will be distributed on the available processors using partitioning algorithm called FBB-FFD (stands for Fisher Baruah Baker - First Fit Decreasing) [6].

This algorithm enhances the schedulability of parallel tasks of fork-join structure, by increasing the parallel segments deadline and getting rid of their strict execution time, as shown in the example of 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, then it 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 it will be clarified later in the analysis.

The number of job migrations in this algorithm could be either 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 will be 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.

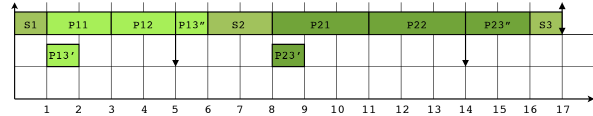


Figure 3. Task Stretch Transformation.

Task Stretch Transformation (TST) has a constraint when it comes to practical implementation, that in order to achieve a fully stretched master string, then 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 then they will 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* [7] (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 *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, we will keep 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 can 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 a 1 unit of time not used before the deadline), the parallel segment $P_{2,3}$ will not be used.

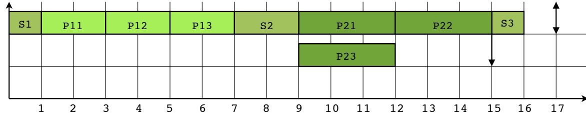


Figure 4. Segment Stretch Transformation.

In TST, the master string has to be filled with all 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. However, in SST, the master string will be filled initially 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 FBB-FFD partitioning algorithm. The laxity of the master string will increase since we did not fill it completely with parallel segments.

So, from a practical implementation point of view, the SST can be fully implemented on a standard Linux RT kernel with no special extensions or batches 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

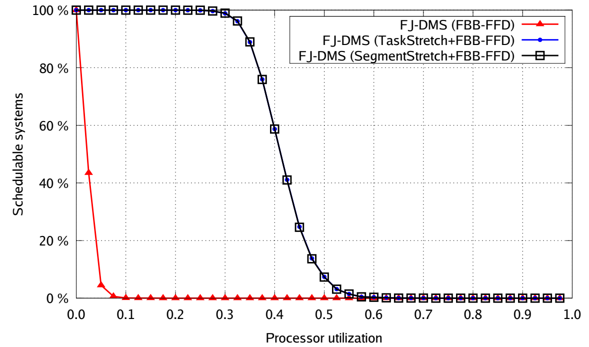
In order to provide a practical analysis for these algorithms, we are going to use *rtmsim* (stands for Real-Time Multiprocessor SIMulator). It is a free simulation software developed at *Université Paris-Est Marne-la-Vallée, France* [8]. This simulation software helps analyzing the performance of real-time scheduling algorithms by choosing one of the pre-coded approaches and run in through extensive simulation.

For our extensive simulation analysis, we considered 4 identical processors with taskset utilization varies from 0.025 to 0.975 times the number of processors in steps of 0.025, and for each utilization value we run 10,000 tasksets each of 16 parallel tasks with implicit deadline, which will be scheduled using FBB-FFD partitioning 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 got the results shown in Figure 5(a) (the curve with the rectangle points). And as we can see from this result, FBB-FFD failed to schedule the dataset after 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 $RBF^*(\tau_j, d_i) = e_j + \frac{e_j}{P_j} * d_i$



(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.

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

But by looking at the characteristics of the parallel segments, we can notice that they have offsets which means that they will not arrive all at the same time to be scheduled, and 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 is performed to compare TST and SST algorithms, by using the same model of extensive simulation described previously, the result of simulation is shown in Figure 5(a), where TST is the curve with the round points and SST is the curve with the square ones. As we can see, both curves are the same with no noticeable difference. There is a slight difference between these 2 algorithms as represented in Figure 5(b). The interesting result we can notice is the incomparability of these 2 algorithms.

6. Perspective

The temporal constraints of the theoretical real-time systems such as the worst case execution time, the deadline and the period, all these values can be estimated and specified. But when it comes to commercial real-time systems, some interferences and variations affect those constraints, the causes vary from the tasks to exceed their WCET, OS overheads to system interrupts [9], those variations made the constraints harder to be controlled and studied.

There exist some mechanisms to compute the variations in the temporal constraints and to analyze the interferences, among those we can mention as example the sensitivity analysis, which "provides useful information for changing the implementation by giving a measure of those computation times that must be reduced to achieve feasibility"[10]. And task's allowance which is defined as the maximum extra duration that can be granted to a faulty task without compromising the timeliness constraint of the task [11].

The principal idea of the algorithm TST is to design a full master string, where the processor's utilization is 100%, and there is no laxity which means the worst execution time of the task equals to its deadline. However, in our proposed algorithm SST, we stopped the migration and created a non-full master string by filling it with complete segments, which increased the laxity of the master string as well as the parallel segments, and it is a step forward to build a robust system.

In the future, we aim to provide an algorithm which computes a robust partitioning, in which we maximize the acceptable variations of the temporal constraints, taking into account the variations of possible WCET overruns. As well as maximizing the duration of a task without compromise missing its deadline [12].

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 eliminate the use of job migration, which makes it possible to be implemented on 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 will be to study the possibility of proposing a robust partitioning algorithm so as to maximize the laxity of the segments and tolerate the execution overruns of the parallel task model.

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] "Posix threads programming." [Online]. Available: https://computing.llnl.gov/tutorials/pthreads/

[4] "Openmp." [Online]. Available: http://www.openmp.org

[5] 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.

[6] 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.

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

[8] "rtmsim." [Online]. Available: http://igm.univ-mlv.fr/AlgoTR/rtmsim

[9] R. I. Davis and A. Burns, "Robust priority assignment for fixed priority real-time systems," in *Proc. of RTSS*, 2007, pp. 3–14.

[10] E. Bini, M. Di Natale, and G. C. Buttazzo, "Sensitivity analysis for fixed-priority real-time systems," in *Proc. of ECRTS*, 2006, pp. 13–22.

[11] L. Bougueroua, L. George, and S. Midonnet, "Dealing with execution-overruns to improve the temporal robustness of real-time systems scheduled fp and edf," in *Proc. of ICONS*, 2007, p. 8pp.

[12] F. Fauberteau, S. Midonnet, and L. George, "A robust partitioned scheduling for real-time multiprocessor systems," in *Proc. of DIPES*, 2010, pp. 193–204.