Challenges and Design Principles for Implementing Slot-Based Task-Splitting Multiprocessor Scheduling

Paulo Baltarejo Sousa, Björn Andersson, and Eduardo Tovar
CISTER-ISEP Research Center
Polytechnic Institute of Porto
4200-072 Porto, Portugal
{*pbsousa,bandersson,emt*}*@dei.isep.ipp.pt*

***Abstract*—Consider the problem of scheduling a set of sporadic tasks on a multiprocessor to meet deadlines even at high processor utilizations. We assume that task preemption and migration is allowed but because of their associated overhead, their frequency of use should be kept small. Task-splitting (also called semi-partitioning) is a family of algorithms that offers these properties. An algorithm in this class assigns most tasks to just one processor but a few tasks are assigned to two or more processors, and they are dispatched in a way that ensures that a task never executes on two or more processors simultaneously. A certain type of task-splitting algorithms, called slot-based split-task dispatching, is of particular interest because of its ability to schedule tasks at high processor utilizations. Unfortunately, no slot-based task-splitting algorithm has been implemented in a real operating system so far.**

Therefore, in this paper, we discuss challenges and design principles for implementing slot-based task-splitting algorithms on multiprocessor systems running the Linux kernel.

I. Introduction

The real-time systems research community has developed a comprehensive toolkit comprised of scheduling algorithms (RM and EDF), schedulability tests and implementation techniques which have been very successful: they are currently taught at major universities world-wide; they are incorporated in design tools and they are widely used in industry. Unfortunately, the results were limited to computer systems with a single processor only.

Today, a multiprocessor implemented on a single chip (called *multicore*) is the preferred platform for many embedded real-time applications and this brings the pressing need for developing an analogous toolkit for multicores. Such a toolkit for multicore should ideally exhibit the same properties as the uniprocessor toolkit exhibited and that engineers valued: (i) high utilization bound; (ii) few preemptions; (iii) dispatchers with low time-complexity; and (iv) the ability to provide pre-run-time guarantees to schedule sporadically arriving tasks to meet deadlines even with deadlines much shorter than the minimum inter-arrival times.

During recent years, the research community has therefore created a family of real-time scheduling algorithms which exhibit these properties. This family of algorithms is called *task-splitting* or *semi-partitioning* [1], [2], [3], [4], [5], [6], [7], [8]. Recent evaluations based on simulation experiments [3] and implementations in real operating systems [9] have demonstrated the excellent performance of this class of algorithms. The key idea of these algorithms is that they assign most of the tasks to just one processor but some of the tasks (called *split tasks*) are assigned to two or more processors. Uniprocessor dispatchers are used on each processor but they are modified to ensure that a split task never executes on two or more processors simultaneously.

One particularly interesting class of task-splitting algorithms is those algorithms where time is subdivided into timeslots such that within timeslots, processor reserves are carefully positioned with a time offset from the beginning of a timeslot. A split task is assigned to two or more processor reserves located on different processors and the positioning of the processor reserve in time is statically assigned (relative to the beginning of a timeslot) so that no reserves serving the same split task overlap in time — see Fig. 2(a). Among the types of split-task scheduling algorithms, this is the class that provides the highest utilization bound. In addition, its run-time dispatching does not depend on any data structures that are shared among all processors and therefore it has the potential to scale to multicore processors with a very large number of processors. For these reasons, we believe an implementation of a slot-based task-splitting algorithm would be valuable.

Three implementations of multiprocessor scheduling algorithms have recently been developed. $\mathrm{Litmus}^{\mathrm{RT}}$ [10] provides a modular framework for different scheduling algorithms (global-EDF and pfair algorithms) for the Linux kernel. Kato *et al.* [9] has also created a modular framework (called RESCH) for using other algorithms than $\mathrm{Litmus}^{\mathrm{RT}}$ (partitioned and semi-partitioned scheduling) for the Linux kernel. Faggioli *et al.* [11] has implemented global-EDF in the Linux kernel and made it compliant with POSIX interfaces. The implementation of $\mathrm{Litmus}^{\mathrm{RT}}$ and the POSIX compliant implementation do not support the class task-splitting at all and hence they are not in the scope of our interest. The framework by Kato *et al.* shares some of our goals in that it provides an implementation of task-splitting algorithms. But it uses another type of task-splitting (that is

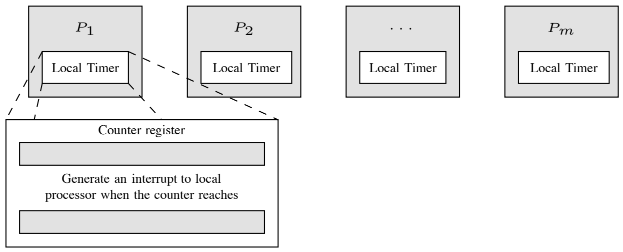


Figure 1. Each processor (P_i) has a local timer.

not slot-based split-task dispatching) which cannot guarantee to meet deadlines at high processor utilization. Hence, the current research literature provides no answer to the question whether slot-based task-splitting multiprocessor scheduling can be implemented and whether it works in practice.

This paper presents challenges and design principles for implementing sloted-based task-splitting algorithms such as [2] and [3]. This is relevant since the algorithm in [3] was (and still is) the algorithm that, in theory, has the best ability (among state-of-art algorithms) to offer pre-run-time guarantees to arbitrary-deadline sporadic tasks on a multiprocessor.

II. Background

A. System model

Consider n tasks and m identical processors. A task τ_i is uniquely indexed in the range $1..n$ and a processor in the range $1..m$. Each task τ_i is characterized by worst-case execution time C_i and minimum inter-arrival time T_i and by the time that the execution must be completed, the deadline (D_i). We assume $0 \leq C_i \leq D_i$.

A processor p executes at most one task at a time and no task may execute on multiple processors simultaneously. The system utilization is defined as $U_s = \frac{1}{m} \cdot \sum_{i=1}^{n} \frac{C_i}{T_i}$.

We assume that (i) all processors have the same instruction set and data layout (e.g. big-endian/little-endian), (ii) all processors execute at the same speed and (iii) the speed at which a task executes is independent of which processor it executes on. We assume that the execution speed of a processor does not depend on activities on another processor (for example whether the other processor is busy or idle or which task it is busy executing) and also does not change at runtime. In practice, this implies that (i) if the system supports simultaneous multithreading (Intel calls it *hyperthreading*) then this feature must be disabled and (ii) features that allow processors to change their speed must be disabled. We assume that each processor has a local timer — see Fig. 1. that provides two functions: (i) one function allows reading the current real-time (that is is not calender time) as an integer; and (ii) another function makes it possible to set up the timer to generate an interrupt x time units in the future, where x can be specified.

Consider n=m+1 tasks with T_i=1 and $C_i = 0.5+\epsilon$ (where ϵ is a positive number smaller than $1/6$) to be scheduled on m processors. It is easy to see that if task migration is not allowed then there is a processor which is assigned at least two tasks. And on this processor, the utilization exceeds 100% and hence a deadline miss occurs. This is problematic since $U_s = \frac{m+1}{m} \cdot (0.5+\epsilon)$ which becomes 0.5 as $m \to \infty$ and $\epsilon \to 0$; that is, a deadline miss can occur although only 50% of the entire processing capacity is requested.

Researchers observed [12], [1] that if the execution-time of a task could be "split" into two pieces then it is possible to meet deadlines. For example, assign task τ_i with $i \in \{1, 2, 3, \ldots, m\}$ to processor P_i and assign task τ_{m+1} to two processors (for example P_1 and P_2) so that a job by τ_{m+1} executes $0.25 + \frac{\epsilon}{2}$ units on one of the two processors and $0.25 + \frac{\epsilon}{2}$ units on the other. This makes it possible to meet deadlines, assuming that the two "pieces" of task τ_{m+1} are dispatched so that they never execute simultaneously.

Many recent algorithms are based on this idea and they differ in (i) how tasks are assigned to processors and split before run-time and (ii) how tasks are dispatched, particularly, how split tasks are dispatched at run-time. The two approaches for split-task dispatching that we believe are the most promising are (i) *job-based split-task dispatching* [6] and (ii) *slot-based split-task dispatching* [2]. Job-based split-task dispatching splits a job into two or more subjobs and forms a sequence of subjobs from the subjobs that originate from the same job and sets the arrival time of a subjob equal to the absolute deadline of its preceding subjob. Job-based split-task dispatching provides a utilization bound greater than 50% and few preemptions. It has been implemented in a real operating system and through experimental studies [9] of that implementation it was found to outperform many other non-split approaches. The main drawback of job-based split-task dispatching is that utilization bounds greater than 65% have not been attained.

Slot-based split-task dispatching subdivides time into equal-duration timeslots whose beginning and end are synchronized across all processors; the end of a timeslot of processor p contains a reserve and the beginning of a timeslot of processor p+1 contains a reserve, and these two reserves supply processing capacity for a split-task — see Fig. 2(a). Slot-based split-task dispatching causes more preemptions than job-based split-task dispatching but, in return, it offers higher utilization bounds (higher than 65% and configurable for up to 100%) [2] and best performance (among all algorithms, not only task-splitting algorithms) for providing pre-run-time guarantees to arbitrary-deadline tasks [3].

Despite the good performance of slot-based split-task dispatching in theory, the current research literature provides no answer to the question whether slot-based task-splitting multiprocessor scheduling can be implemented and whether it works in practice.

B. Challenges

From Fig. 2(a), we can identify three challenges for implementing slot-based split-task dispatching:

C1. Timeslots must begin at the same time on all processors;

C2. A split-task must migrate instantaneously in the beginning of a timeslot;

C3. The reserves should begin and end at precisely specified time instants.

Since Moore's law causes the number of processor cores in multicore systems to increase exponentially with time we believe it is also important that an implementation of a multiprocessor scheduling algorithm has a dispatching overhead that is low as a function of the number of processors — ideally independent of the number of processors. This poses no challenges for scheduling non-split tasks. For split-tasks however this brings the following two additional challenges:

C4. The run-time overhead of migration (manipulation of data structures and concurrency control) should be independent of the number of processors;

C5. The run-time overhead due to handling of timers (reading the current value of a real-time clock; setting up a timer to generate an interrupt signal at a certain time) should be independent of the number of processors.

We will address these challenges in the next section. Challenges C1 and C3 will be resolved using high-resolution local timers to each processor. Challenges C4 and C5 will be resolved through carefully designed data structures which avoids synchronization between processors and the local timers will help us overcome C5. The challenge C2 is fundamental however — we can resolve it by a minor redesign of the actual scheduling algorithm. Consider Fig. 2(a) again. It shows that task τ_2 must migrate instantaneously at certain instants; this occurs at time 0, time S, time $2S$, etc. Actually, this situation can introduce additional complexity to the dispatching algorithm, which can imply more overhead and more preemptions. To illustrate this, let us assume that the current time t is infinitesimal higher than S. Therefore, the dispatcher of processor P_2 will select task τ_2 to be executed, but it must not do that without checking if task τ_2 has already relinquished processor P_1. Due to many factors, like interrupt handling or timer drift just to mention some, it could happen that task τ_2 has not relinquished processor P_1 yet at that time. Since, task τ_2 cannot execute on both processors then the dispatcher of processor P_2 has to select other task to be execute on it. This can be avoided if the reserve on processor P_2 starts slightly later — see Fig. 2(b).

III. How to implement slot-based task-splitting

In order to cope with challenges listed in Section II, we recommend that an implementation of a task-splitting follows the following design principles:

P1. Each processor should have its own run-queue (the queue that stores tasks which have outstanding request for execution). The run-queue of processor p should store non-split tasks assigned to processor p. The run-queue of each processor should support the operations `insert_task`, `peek_highest_priority_task` and `extract_highest_priority_task` with low-time complexity.

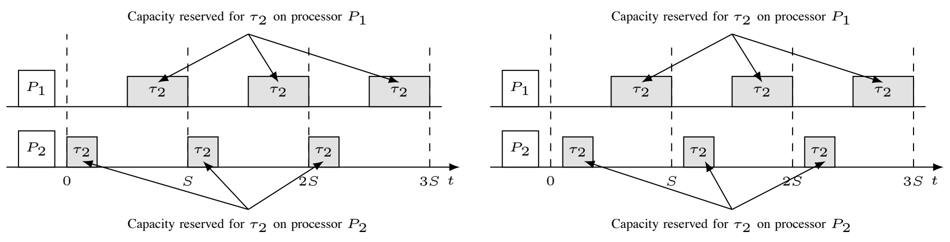
P2. For each processor p, there should be a data structure with two variables `hi_split` and `lo_split`. The variable `hi_split` of processor p and the variable `lo_split` of processor p+1 should point to the process control block for the task that is split between them. If no such task exist then these pointers are NULL.

P3. Each processor should have a variable called `begin_curr_timeslot`. It should hold a time which is no larger than the current time and it should never be less than current time minus S (timeslot length). The variable `begin_curr_timeslot` should be incremented by S to ensure this. This assures that the beginning of the timeslot on each processor is synchronized and avoids the lock mechanism that would be necessary if this variable was global.

P4. Each processor should have a timer-queue of events in the future. This should always include the time of the beginning of the next timeslot, that is `begin_curr_timeslot` + S. If applicable, it also contains the time when the reserve in the beginning of the timeslot ends and also the time when the reserve in the end of the timeslot begins. Whenever the timer queue changes (for example an event has expired and therefore should be removed from the timer queue, or a new event is inserted into the timer queue), the processor should disable interrupts, set up a timer x time units in the future where x is the time of the earliest event in the timer queue minus current time, and then enable interrupts. This is a standard approach for timers and it ensures that cumulative drift because of finite speed of the processor does not occur (see page 38 in [13] for discussion).

P5. The operating system should implement a `delay_until` system call (see page 38 in [13] which makes it possible for a task to sleep until an absolute time. This is important for implementing periodically arriving tasks without suffering from cumulative drift [13].

These design principles will be followed to implement the scheduling algorithm proposed in [2] using the Linux kernel 2.6.28. This kernel version is provided by the required


Capacity reserved for τ_2 on processor P_1
P_1
τ_2
P_2
τ_2
0
S
$2S$
$3S$
t
Capacity reserved for τ_2 on processor P_2
Capacity reserved for τ_2 on processor P_1
P_1
τ_2
P_2
τ_2
0
S
$2S$
$3S$
t
Capacity reserved for τ_2 on processor P_2

(a) Original split-task dispatching

(b) New split-task dispatching

Figure 2. An example of the operation of slot-based task-splitting multiprocessor scheduling. Task τ_2 is a split-task. A non-split task executes only on its dedicated processor; it can execute in a reserve but it does so with a lower priority than a split task.

tools to statisfy these design principles: (i) each processor holds its own run-queue and it is easy to add new fields to it; (ii) it has already implemented red-black trees that are balanced binary trees whose nodes are sorted by a key and the most operations are done in $O(\log n)$ time; (iii) it has the high resolution timers infrastructure that offers a nanosecond time unit resolution and can be set on per-cpu; (iv) it is very simple to add new system calls and finally (v) it comes with the modular scheduling infrastructure that became easy to add a new scheduling policy to the Linux kernel.

IV. Conclusions

We have shown the challenges and design principles for implementing slot-based task-splitting multiprocessor scheduling algorithms. These kind of scheduling algorithms for multiprocessor systems offer higher utilization bounds and best performance (among all algorithms, not only task-splitting algorithms), however, the current research literature provides no answer to the question whether slot-based task-splitting multiprocessor scheduling can be implemented and whether it works in practice. These design principles will be followed to implement the scheduling algorithm proposed in [2] using the Linux kernel 2.6.28.

Acknowledgements

This work was partially funded by the Portuguese Science and Technology Foundation (Fundação para a Ciência ea Tecnologia - FCT) and Luso-American Development Foundation (FLAD).

References

[1] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemption," in *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Application (RTCSA 06)*, Sydney, Australia, 2006, pp. 322–334.

[2] B. Andersson and K. Bletsas, "Sporadic multiprocessor scheduling with few preemptions," in *20th Euromicro Conference on Real-Time Systems (ECRTS 08)*, Prague, Czech Republic, 2008, pp. 243–252.

[3] B. Andersson, K. Bletsas, and S. Baruah, "Scheduling arbitrary-deadline sporadic tasks on multiprocessors," in *29th IEEE Real-Time Systems Symposium (RTSS 08)*, Barcelona, Spain, 2008, pp. 385–394.

[4] K. Bletsas and B. Andersson, "Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound," in *30th IEEE Real-Time Systems Symposium (RTSS 09)*, Washington, DC, USA, 2009, pp. 385–394.

[5] K. Lakshmanan, R. Rajkumar, and J. Lehoczky, "Partitioned fixed-priority preemptive scheduling for multi-core processors," in *21st Euromicro Conference on Real-Time Systems (ECRTS 09)*, Dublin, Ireland, 2009, pp. 239–248.

[6] S. Kato and N. Yamasaki, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *21st Euromicro Conference on Real-Time Systems (ECRTS 09)*, Dublin, Ireland, 2009, pp. 239–248.

[7] ——, "Portioned EDF-based scheduling on multiprocessors," in *8th ACM/IEEE International Conference on Embedded Software (EMSOFT'08)*, Atlanta, GA, USA, 2008, pp. 139–148.

[8] ——, "Real-time scheduling with task splitting on multiprocessors," in *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 07)*, Daegu, Korea, 2007, pp. 441–450.

[9] S. Kato, R. Rajkumar, and Y. Ishikawa, "A loadable real-time scheduler suite for multicore platforms," Technical Report CMU-ECE-TR09-12, Tech. Rep., 2008.

[10] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LITMUS^{RT} : A testbed for empirically comparing real-time multiprocessor schedulers," in *27th IEEE Real-Time Systems Symposium (RTSS 06)*, Rio de Janeiro, Brazil, 2006, pp. 111–126.

[11] D. Faggioli, M. Trimarchi, F. Checconi, and C. Scordino, "An EDF scheduling class for the Linux kernel," in *11th Real-Time Linux Workshop (RTLWS 2009)*, Dresden, Germany, 2009.

[12] J. H. Anderson, V. Bud, and U. C. Devi, "An EDF-based scheduling algorithm for multiprocessor soft real-time systems," in *17th Euromicro Conference on Real-Time Systems (ECRTS 05)*, Palma de Mallorca, Balearic Islands, Spain, 2005, pp. 199–208.

[13] A. Burns, *Concurrency in Ada*. Cambridge University Press, 1998.