AUTOMATYKA/ AUTOMATICS • 2012 • Vol. 16 • No. 2

http://dx.doi.org/10.7494/automat.2012.16.2.127

Jerzy Martyna*

Partition of Real-Time Application over Multicore Systems

1. Introduction

Modern multicore systems provide a new mechanism for increasing computational speed of computer systems. In the multicore systems the tasks are allocated to an individual core of a multicore system according to crucial objectives, such as guaranteeing the feasibility of the system under given performance requirements, satisfying the assumed energy-aware strategies for an individual core and the overall system, etc.

However, the multicore systems cause many problems for software designers too as well as for real-time system programmers in a number of concurrent applications and in many embedded real-time system applications. It is associated with the constraints and sharing of such resources, as timing dependencies between tasks, possible parallel execution of tasks, lack of main memory for allocation of the demand number of tasks, etc.

The resource management of multicore systems has been described in some papers. Among others, in the study by E. Bini *et al.* [2] the Actors project was presented. It was based on the Aquose architecture employed in the European Commission's Framework for Real-Time Embedded Systems and is devoted to the automatic partition of a time-sensitive application over multicore platforms. The software was based on Linux operating system. In the paper by A.K. Mok *et al.* [8] a concept of virtual processors modelled on using a bounded delay abstraction to represent a virtual platform was introduced. A bounded delay can be fully described by means of two parameters: bandwidth α, which measures the relative speed with which a resource is assigned to the application, and delay Δ, which represents the worst-case service delay. In the paper by Fahmy *et al.* [5] the collaborative scheduling algorithms of distributable real-time threads in dynamic, em bedded systems are proposed.

An algorithm based on gossip-style, called RTTG-DS, was suggested by the K. Han *et al.* [6]. This algorithm bounds thread blocking time and detects deadlocks. Currently,

* Jagiellonian University, Institute of Computer Science, Prof. S. Łojasiewicza 6, 30-348 Cracow, Poland

some new methodologies for abstracting the total computing power of multicore systems was proposed. In the paper by L. Abeni *et al.* [1] a temporal isolation achieved through a resource reservation technique was proposed. According to this technique the CPU processing capacity can be partitioned into a set of reservations, each equivalent to a virtual processor with a reduced speed. Thus, the resource reservation can be applied in order to isolate the behaviour of the real-time, as well as non real-time applications. Past efforts on resource management in a multicore platform were described by G. Buttazzo [3] as a general methodology for obstracting the total computing power available on a multicore platform by a set of virtual processors, in order to allocate applications independently of the physical platform.

In this paper we introduce a new algorithm with reduced complexity in the handling of real-time applications in a multicore system. This algorithm can find suboptimal partitions of real-time applications. The pseudocode of the proposed algorithm allows us to implement it in all computational environments. The simulation results provide an analytical framework for the performance evaluation of the proposed algorithm.

The rest of this paper is organized as follows. Section 2 gives the system model, the terminology and the notation used throughout the paper. Section 3 presents an algorithm for partitioning the real-time application into flows. Section 4 provides simulation results in order to evaluate the proposed method. Finally, Section 5 states our conclusions.

2. The System Model

Firstly, we model a real-time application as a set of tasks with given precedence constraints described by a directed acyclic graph (DAG).

We assume that each application possesses period T and must be computed within a given relative deadline. Each real-time application is composed of tasks described by the DAG graph. We assume that each task τ can sequential executed on a single core of multicore system. It is a part of a code which cannot be parallelized and must be executed sequentially.

Let d_i be a deadline assigned to task τ_i. It is characterized by a known worst-case execution time $C_i > 0$. For each task also determined is an activation time α_i relative to the activation of the first task of the application. It is obvious that for each task the following condition $[t + a_i, t + d_i]$ must be satisfied.

We introduce a precedence relation R to a partial ordering $P \in \Gamma \times \Gamma$ predecessor of τ_i for all tasks τ_i. Notation $\tau_i \prec \tau_j$ denotes that τ_i is a precedecessor τ_j. It means that task τ_j cannot start its execution before the completion of τ_i. We can define the following terms:

Def. 1 (Path P)

Path P is any subset of tasks $P \subseteq \Gamma$ totally ordered according to R, i.e. $\forall \tau_i, \tau_j \in P$ either $\tau_i \prec \tau_j$ or $\tau_j \prec \tau_i$.

Def. 2 (Sequential Execution Time C_S)

Sequential Execution Time C_S is the minimum time needed for the completion of the application on a uniprocessor system. It is equal to

$$C_s \overset{def}{=} \sum_{\tau_i \in \Gamma} C_i \quad (1)$$

Def. 3 (Parallel Execution Time C_P)

Parallel Execution Time to C_P is the minimum time needed for the completion of the application on a parallel architecture with an infinite number of cores., namely

$$C_p \overset{def}{=} \max_{P\, is\, a\, path} \sum_{\tau_i \in P} C_i \quad (2)$$

Def. 4 (Critical Path, *CP*)

Critical Path is a path *P* having $\sum_{\tau_i \in P} C_i = C_P$. In the example of Figure 1 the critical path is $CP = \{\tau_1, \tau_2, \tau_5, \tau_6\}$.

Figure 1 shows an example of a directed acyclic graph for a real-time application of six tasks with the following execution times:

$$C_1 = 5, C_2 = 4, C_3 = 6, C_4 = 12, C_5 = 8, C_6 = 6.$$

We assume that the application starts at time $t = 0$ and is periodically activated with a period $T = 35$. We consider a relative deadline D equal to the period.

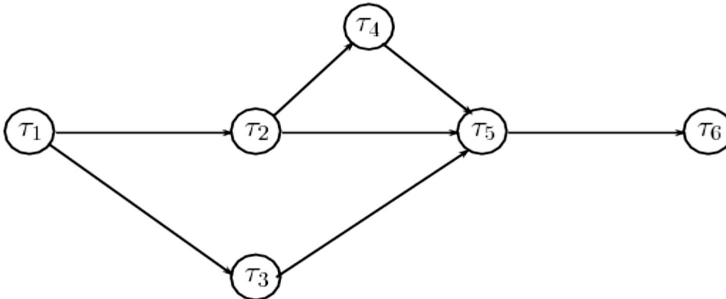

Fig. 1. A real-time application represented with directed acyclic graph

We can illustrate the parallel execution of the six tasks by means of the use of Gantt chart (Fig. 2). In such diagram each task starts in the timeline as soon as possible on the first available core in a multicore system. All arrows in the Figure 2 represent the synchronization points coming from the precedence graph. The given Gantt chart shows in each time slot the maximum number of cores which can be use to parallel execution of some tasks.

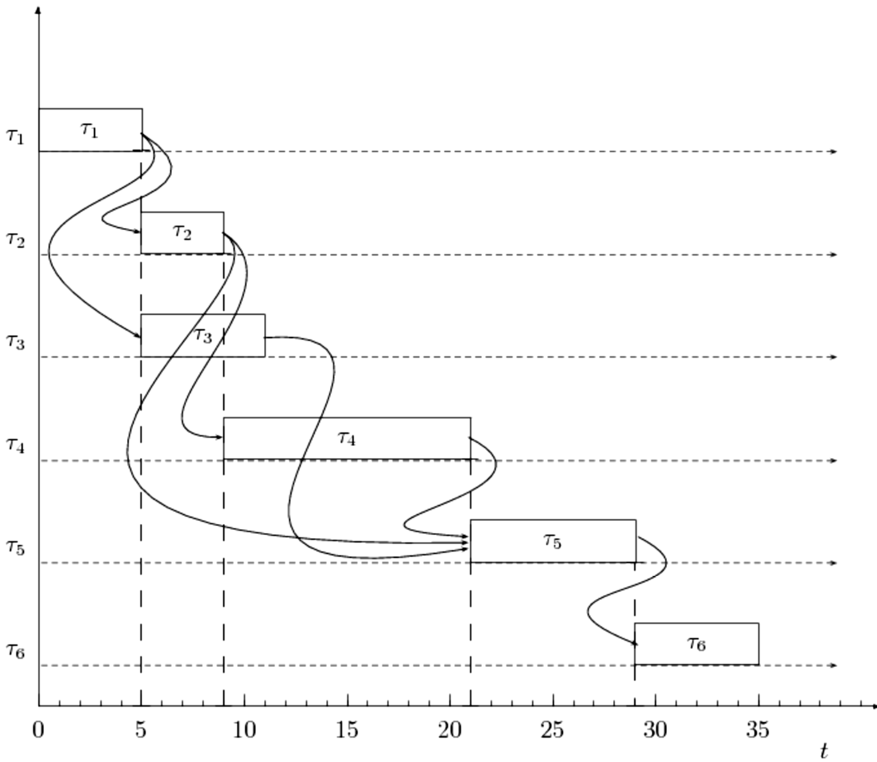

Fig. 2. Timeline representation

3. The Proposed Algorithm for a Real-time Application Partition over Multicore systems

In this section, we propose an algorithm for a real-time application partition over multicore systems.

The methodology for the application partition is according to the approach given by H. Chetto [4] and later used by A. K. Mok, *et al.* [8, 9]. Unfortunately, their algorithms are valid only for uniprocessor systems and do not consider the possibility of using parallel computations.

The first step into a partition of the real-time application is an assignment of a deadline to all tasks. The algorithm proceeds by assigning the deadline to task τ_i for which all successors have been considered. Thus, the deadline assigned to a task is given by

$$d_i = \min_{j:\tau_i \to \tau_j} (d_j - C_j) \tag{3}$$

where C_j is a computational time of task τ_j.

The second step into a partition of the real-time application is an assignment of the activation times. It is obvious that no task can be activated before all its predecessors have finished. The constraint that task τ_i is not activated before τ_k must be satisfied, namely

$$a_i \geq a_i^{prec} = \max_{k:\tau_k \to \tau_i, \tau_k \in \{F_k\}} \{a_k\} \quad (4)$$

In other words, task τ_i cannot be activated before the maximum deadline of the immediately preceding task is finished, namely

$$a_i \geq d_i^{prec} = \max_{k:\tau_k \to \tau_i, \tau_k \in \{F_k\}} \{d_k\} \quad (5)$$

where $\{d_k\}$ is a set of deadlines of immediate predecessors for task τ_i in a graph of real-time application. Thus, we can formulate a condition for task τ_i, namely

$$a = \max_{k:\tau_k \to \tau_i, \tau_k \in \{F_k\}} \{a_k, d_k\} \quad (6)$$

where $\{a_k\}$ and $\{d_k\}$ are the activation time and the deadline for the immediate predecessor for task τ_i.

The algorithm works as follows. In the first step, it assigns the deadlines times to root nodes, i.e. tasks without predecessors. Next, the algorithm assigns the activation times for all tasks $\tau_i, (i = 1, 2, ... |v|)$ for which all the precedessors have been considered. The pseudocode of the algorithm designed to guarantee the precedence constraints is given in Figure 3. The given algorithm provides the deadlines as late as possible and eliminates the same deadlines.

Step 1. *From all tasks without successors, indicate as v that task which has the maximum deadline* d_v*.*

Step 2. *Remove the task v from the set of tasks and the arcs that connect this task with unindicated tasks of the task set. Let* $|v| := |v| - 1$. *If* $|v| > 0$, *then repeat* **Step 1.**

Step 3.
repeat
select a task τ_v *with all successors modified;*
assign $d_v := \min_{v:\tau_v \to \tau_j} (d_j - C_j)$;
if a task τ_v with all predecessor modified;
assign $a_i := \max\{a_i^{prec}, d_i^{prec}\}$;
until $|v| \leq 0$;

Fig. 3. An algorithm for the deadline and the activation assignment

In the third step into the partition of a real-time application we allocate all the tasks over the multicore system. The complexity of the branch and the bound algorithm, which allows us to obtain the optimal partition of real-time application, is very high. Therefore, we propose the breadth search algorithm. To guaranteee the execution time of real-time application, we consider the realization of the tasks from the ciritical path of the tasks for a given application. For the remaining tasks and all the tasks for which the task execution before its deadline was impossible, we used the backtrack search [7] for finding the core that can per form a task j before its deadline.

The idea behind the backtrack search is that a multicore system possesses a list of cores. The extension of this list is the following: for the selected core only one number of the core is generated and it is added to this list. If this core has insufficient time for task τ_j to be executed and completed before its deadline, then a new number of the core is generated. When it is impossible, we return to the neighbour core for a given core for which we can generate descendants.

procedure *core_scheduler*(*i: number_of_cores*);
schedule all tasks from the critical path CP;
for $j := 1$ **to** v **do**
begin
if $F_j = 0$ {*the task j has not an immediate predecessor*}
then $t := r_i$ **else** $t_j := \max_{t \in F_j} \{\bar{c}_l\}$
endif; {*where* t_j *is the earliest time of execution for task j,* $\bar{c}_l$ *is the completion time for task l,* F_j *is the set of immediate predecessors for the task j*}
if $d_j \geq t_j + \min_i p_{ij}$ **then goto Step 4a**
else {*there is insufficient time for the task to be executed before its deadline*}
goto Step 4b
endif;
Step 4a. *backtrack_search(i, selected_core, a);*
Step 4b. *For the task* j *that can be executed before its deadline determine the best core, namely*

$a := \arg \min_{b \in W_v(i)} [\max\{E_b, t_j + C_j\}];$ {*where* $v = 0, 1, 2, ..., \gamma(i)$, $\gamma(i)$ *is the maximal rank of neighbouring with regard to the i-th core,* E_b *is the time in which the core b has been released,* $W_v(i)$ *is the number of neighbouring cores for the i-th core*}

Fig. 4. Procedure *core_scheduler*

```
procedure backtrack_search(i: number_of_cores; selected_core: boolean;
  a: number_of_selected_core);
begin
  extension(P, i); {extend the list P by the core i}
  while | P | ≠ 0 do
    begin
      a := select(P); {select the first core from the list P}
      if depth(a) = β(i) then
        remove(a,P); {core a has been removed from the list P. Backtrack}
        goto 1;
      endif;
  if arc_test(a) then
      remove(a,P); {If all arc for core a have been used, then the core a
                    is removed from the list P. Backtrack}
      goto 1;
    endif;
    b :=determine(a); { b has been determined as decendant of the core a}
    extension(P,b); { extend the list P by the core b}
    mark(a,b); { the arc(a,b) is denoted as used}
    if test(b) then
      selected_core := true;
      goto 2;
    else goto 1;
  endif;
label 1: selected_core := false end;
label_2: end;
```

Fig. 5. The procedure *backtrack_search*

A procedure called the *arc_test* has access to an arc for a given core. When this core has used all the arcs, then it is removed from the list of cores. Hence the backtrack occurs. The procedure *test* is devoted to checking the solution to the problem.

4. Simulation Experiments

We have simulated the implementation of the proposed algorithm for partitioning the real-time application into flows. The simulation was made for a multicore system with the strictly defined number of cores. For simplification reasons it was assumed that the throughput parameters of each core are identical. In our simulation study we assumed that the subset of dependent tasks in the real-time application can occur in the set of tasks. All possibile forms of precedence contraints in the subset of tasks are included in the simulation.

We assumed that the load of the i-th core in the multicore system is expressed as the arrival rate of the tasks. When the tasks have variable length, the load of the i-th core is defined here as:

$$\Theta_i = \lambda_i \left[\frac{task}{ms} \right] \cdot l_{mean} \left[\frac{instructions}{task} \right] \tag{7}$$

\where λ_i is the mean value of the task arrival rates at the core i, l_{mean} is the mean value of the task length measured as the average number of instructions required for its execution.

The load of the multicore system is the average value of the load of all cores in the multicore system, namely

$$\Theta = \frac{1}{M} \sum_{i=1}^{M} \Theta_i \cdot 100\% \tag{8}$$

where M is the number of cores in the multicore system.

It was assumed here that the simulation time unit is equal to one millisecond. Let the simulation time be divided into periods. The length of a period is 1000 simulation time units.

In our experiment, we studied the relation between the number of tasks in the real-time application and the load of the multicore system. In Figure 6 we show the dependence of n – an average number of tasks in the real-time application and the system load for a multicore system with 8 and 16 cores. In order to compare these results we used the algorithm for scheduling of independent tasks in the multiprocessor system proposed in McNaughtan [10]. As shown in Figure 6b a large number of cores allows us to obtain greater value of the mean number of the processed tasks.

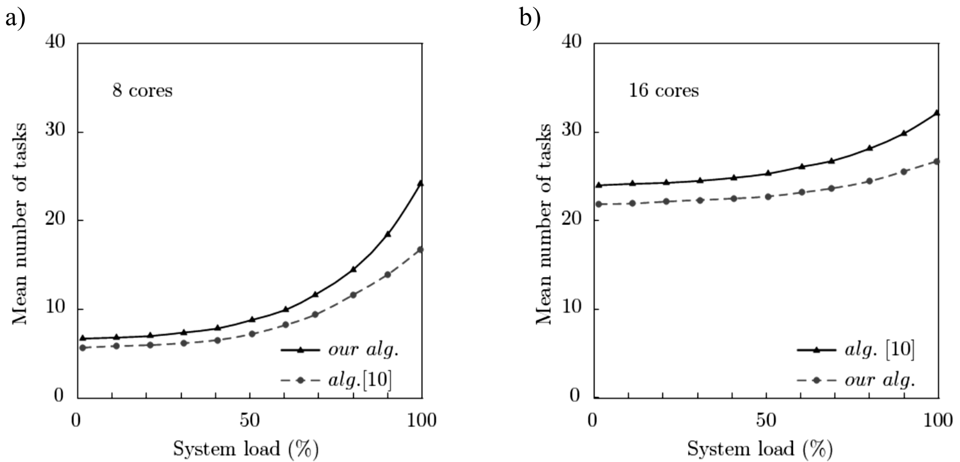

Fig. 6. Mean number of tasks of real-time application in dependence of system load for various number of cores in multicore system

Figure 7 shows the mean run time of the application in dependence of the number of tasks. It is worth mentioning that the heuristic algorithm requires less run time for the task scheduling.

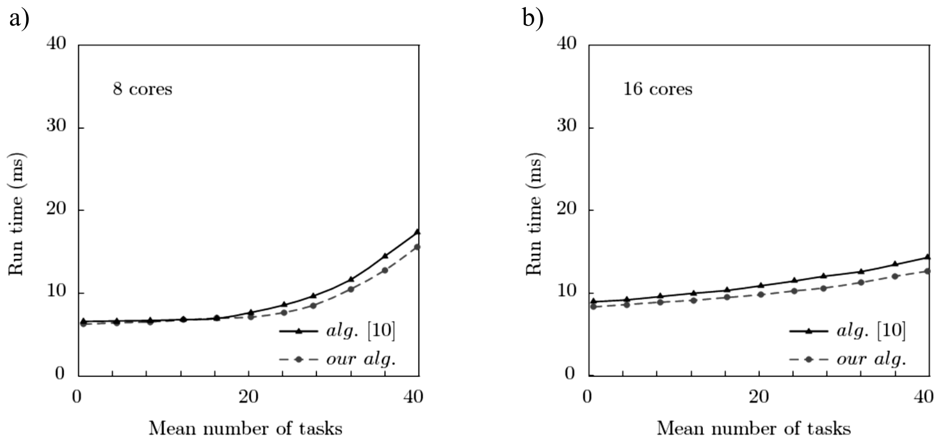

Fig. 7. Run time of tasks of real-time applications in dependence of mean number of tasks for various number of cores

5. Conclusion

We presented an algorithm for partition of a real-time application over a multicore system. We showed that our methodology allowed us to allocate tasks belonging to this application into parallel cores of a multicore system. It accelerated the speed of the time execution of a real-time application.

The contribution of this work was the development of an algorithm that can partition the application into tasks which are in most executed parallel in the multicore system. It minimizes the overall time required for the execution of a real-time application. Moreover, it takes into consideration the specified time constraints and the precedence relations between all tasks. Given the high complexity, we proposed a heuristic which minimizes the total demands of computational complexity and reduces the running time compared with the standard multiprocessor scheduling method.

In the future work we plan an extension of our model by including the calculation of the interprocessor communication between tasks within a real-time application.

References

[1] Abeni L., Buttazzo G., *Resource Reservation in Dynamic Real-Time Systems*. Real-Time Systems, vol. 27, No. 2, 2004, 123–167.

[2] Bini E., Buttazzo G., Eker J., Schorr S., Guerra R, Fohler G., Arzen K.-E., Romero Segovia V., Scordino C., *Resource Management on Multicore Systems: The Actors Approach*. IEEE Micro, 2011, 72–81.
[3] Buttazzo G., Bini E., Wu Y., *Partitioning Real-Time Applications Over Multicore Reservations*. IEEE Trans. on Industrial Informatics, vol. 7, No. 2, 2011, 302–315.
[4] Chetto H., Silly M., Bouchentouf T., *Dynamic Scheduling of Real-Time Tasks Under Precedence Constraints*. Real-Time Systems, vol. 2, No. 3, 1990, 181–194.
[5] Fahmy S., Ravindran B., Jensen E.D., *On Collaborative Scheduling of Distributable Real-Time Threads in Dynamic*. Networked Embedded Systems, [in:] IEEE Int. Symp. on Object Oriented Real-Time Distributed Computing (ISORC), 2008, 485–451.
[6] Han K., Ravindran B., Jensen E.D., *Exploiting Slack for Scheduling Dependent, Distributable Real-Time Threads in Mobile Ad Hoc Networks*. Int. Conf. on Real-Time and Network Systems (RTNS), 2007, 225–234.
[7] Knuth D.E., *The Art of Computer Programming*. Addison-Wesley, Reading, 1986.
[8] Mok A.K., Feng X., Chen D., *Resource Partition for Real-Time Systems*. [in:] Proc. 7th IEEE Real-Time Technology and Applications Symp., IEEE CS Press, 2001, 75–84.
[9] Mok A.K., Feng X., *Resource Partition for Real-Time Systems.* [in:] Proc. 7th IEEE Real-Time Technol. Appl. Symp., Taipei, Taiwan, May 2011, 75–84.
[10] McNaughtan R., *Scheduling with Deadlines and Loss Functions*. Management Science, 6, 1959, 1–12.