Partitioning Real-Time Application Over Multicore Reservations

Izchukwu George Enekwa
dept. Electronics Engineering
Hochshule Hamm-Lippstadt
Hamm, Germany
Izuchukwu-george.enekwa@stud.hshl.de

***Abstract*—In order to benefit fully from the computing power of a multicore platform, software must be designed to support parallel executional flows. At the same time, modern embedded systems contain many parallel applications with timing requirements, in order to achieve temporal isolation among critical operations running concurrently on the same platform, it is highly desirable in the kernel to provide resource reservations. The aim of this paper is to propose a general approach for abstracting the computing power of a multicore platform, so that applications can be allocated independently of the physical platform, usig a set of virtual processors.**

I. Introduction

The ability to optimize the available resources while meeting performance requirements is an important factor of an embedded system's design, In addition, new products demand higher performances, these requirements they demand are pushing the industry to implement multicore platforms in several domains, With a multicore architecture, you can increase processing speed while keeping the power consumption contained. A single processor operating at a higher frequency would increase its power consumption and cause serious heating problems.The allocation of tasks among processors in a multicore platform, however, impacts the number of active cores required to run an application, so minimizing this number is vital to various design objectives, such as guaranteeing the system's functionality under given performance requirements,turning off some cores or fitting large applications into platforms with decreased number of cores.

Also, the present day embedded systems is developing persistently and these products are frequently organised in a number of simultaneous applications, each comprising of a bunch of tasks with different qualities and imperatives and sharing similar assets. In such a situation, isolating the temporal behavior of real-time applications is crucial to prevent a reciprocated interference among critical functionalities. In this paper, techniques or methods whereby set of tasks for achieving temporal isolation are proposed. These are mostly focused on the issue of accomplishing a practical schedule or limiting the greatest reaction time among tasks. With the fast improvement of multicore implanted frameworks, limiting the accessible assets is of critical significance to save energy or keep chip temperature on check.

Identify applicable funding agency here. If none, delete this.

II. The Systems Modeling

First of all, a brief explanation to understand what Real-Time applications and Multicore are. a Real-Time application are applications which operates in real time; it senses, analyzes,and reacts to live streaming data as it happens. An application that is database-centric stores information in a database (on premises or in the cloud) for future analysis. A Multicore processor is an integrated computer circuit that enables communication between cores, these processors read and execute programs faster, and work simultaneously. They divide and assign all processing tasks appropriately, reduce power consumption and for greater performance generally.

Modeling real-time applications as a set of tasks is done by specifying a Directed Acyclic Graph (DAG) with precedence constraints. In particular, the DAG describes the application while considering the maximum level of parallelism. As a result, each task represents a sequential activity that needs to be executed on one core.

A. Notations and Terms

First we may denote max0,x as (x)0. The terms used throughout this paper are as follows.

Application Γ :

It is a set of n tasks with given precedence relations expressed by a Directed Acyclic Graph (DAG) Also, having in mind that the application is sporadic, which means that it is cyclically activated with a minimum interarrival time T (also referred to as period) and it is also mandatory that it completes within a given relative deadline D, which can be equal to or less than T (D$\leq T$).

Task Tι :

This portion of code must only be executed sequentially and it is also characterized by worst-case execution time C$\iota > 0$.

Tι

is also assigned a deadline dι

and first task of the application time (activation time) aι.

Putting all these together, when an application is activated at t the task Tι

must be executed in [t+ a$\iota, t + d\iota$],

these tasks are done by earliest deadline first scheduling (EDF) we introduced a precedence relation R which is formally defined as a partial ordering R $\subseteq \Gamma \times \Gamma$.

Notation T$\iota \prec Tj$

meaning that Tj cannot start executing before Tι.

This means that Tι

is an immediate prodecessor of Tj. The notation can also be seen as T$\iota \rightarrow Tj$

and T$\iota \prec T\kappa \prec Tj \Rightarrow (T\kappa = T\iota or T\kappa = Tj)$

Execution Time represented as C.

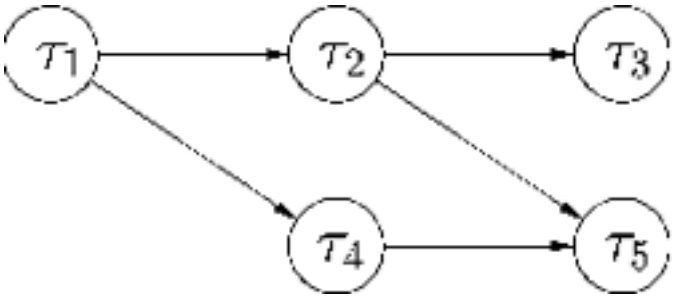


Fig. 1. A sample application represented with directed acyclic graph (DAG).

Fig. 1 shows an example of DAG illustration for a five tasks application with their repective execution times

C1 = 4, C2 = 1, C3 = 5, C4 = 2, C5 = 3

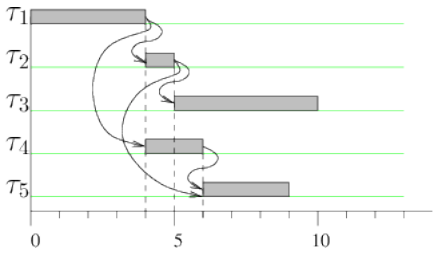


Fig. 2. Timeline representation.

Each task starts in the timeline as soon as possible on the first available core in a multicore system. For the application shown in Fig. 1, the timeline representation is illustrated in Fig. 2, here all arrows shown represents the synchronization points coming from the precedence graph. Timeline representation have the advantage that they clearly illustrate the concurrent nature of the application, showing in each time slot how many cores are required to perform the computation.Therefore, adding additional cores will not reduce the overall response time,since the DAG already reflects maximum parallelism.

- For the notation Path P: Any subset of tasks P$\subseteq \Gamma$ totally ordered according to R; i.e., $\forall T\iota, Tj \in PeitherT\iota \prec TjorTj \prec T\iota$.
- Sequential Execution Time Cs: By serializing all tasks in the DAG, this is the minimum time required to complete the application on a uniprocessor. This equals the sum of the computation times for all tasks. $C_s \stackrel{def}{=} \sum_{\tau_i \in \Gamma} C_i$
- Parallel Execution Time Cp: It represents the time it takes for an application to execute on an infinite number of cores on a parallel architecture. For the application in Fig 1, we have CP = 10. $C_p \stackrel{def}{=} \max_{P \text{ is a path}} \sum_{\tau_i \in P} C_i$
- Critical Path (CP): Here we have longest sequential operation in a parallel computation, notice the critical path in the example Fig. 1 is P = (T1, T2, T3) P having $\sum_{T_i \in P} C_i = C_P$

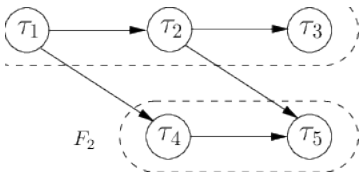


Fig. 3. Parallel flows in which the application can be divided.

Parallelizing an application allows for several options, from having a single flow to execute all tasks sequentially on one core (where maximum parallelism is not exploited/needed) to having a flow per task (maximum parallelism). A flow's definition may affect the total bandwidth required to run an application. Thus, we will attempt to find the best flow partition that minimizes the total bandwidth requirement.

III. The Proposed Algorithm for a Real Time Application Partition over Multicore

The methods for the application partition is described in this section, in accordance to the approach given by H. Chetto [4] and later used by A. K. Mok, et al. [8,9]

The first approach into a partition of real time application is the deadline assignment algorithm

for all (nodes without successors) set $D_i = D$;
while (there exist nodes not set) {
  select a task τ_k with all successors modified;
  set $d_k = \min_{j:\tau_k \rightarrow \tau_j} (d_j - C_j)$;
}

Fig. 4. Parallel flows in which the application can be divided.

For a given partition, the algorithm in Fig. 4, is used to assign deadline to task Tι,

after the activation and deadline time are assigned, a demand bound function is then used to evaluate the overall computation requirement of each flow and to determine the parameters of the corresponding virtual processor.

By summing the bandwidths computed for each flow using [7], the total bandwidth required to minimize the total bandwidth is determined and the partition with the lowest bandwidth is determined by carrying out branch and bound searches.

A. Activation and Deadline Assigning

According to the flow partitioning, activation times and deadlines are assigned to all tasks to meet timing constraints and precedence relationships. Assigning assignments are based on a method originally proposed by Chetto et al [20], which has been adapted for multicore systems and slightly modified to reduce bandwidth requirements. All tasks without successors are assigned the application deadline first. As soon as all successors have been considered, the algorithm assigns the deadline to the task. The deadline assigned to such task is

equation here . .....

$$d_i = \min_{j:T_i - T_j} (d_j - c_j)$$

Second step is the activation time, it is understandable that no task can be activated before other predecessors are completed. Therefore, it is better to make sure that Tι

is not activated before Tj's deadline. Thus,

equation . .....

$$a_i \geq a_i^{prec} = \max_{k:t_k \rightarrow t_j, t_k \in \{F_k\}} (a_k)$$

In a graph of real time application, (dk) represents the deadlines of immediate predecessors to task tι.

As a result, we can formulate a condition for task tι

namely equation .....

$$a = \max_{k:t_k \rightarrow t_i, t_k \in \{F_k\}} \{a_k, d_k\}$$

where (ak) and (dk) are described as the activation and deadline time for the predecessor for task tι.

Here is how the algorithm works, a root node, that is, a task without a predecessor, is assigned deadlines first. Activation times are next assigned for task tι, $(i = 1, 2, .. \mid \upsilon \mid)$

for which all predecessors are considered.

B. STEPS

- Step 1. From all tasks without successors, indicate as v that task which has the maximum deadline dv.
- Step 2. remove the task v from the set of tasks and the arcs that connect this task with unindicated task of the tasks set. Let $\mid \upsilon \mid := \mid \upsilon \mid -1. if \mid \upsilon \mid > 0,$

then repeat Step 1.

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

Fig. 5. Steps noted are an algorithm for the deadline and the activation assignment.

As part of the third step into partitioning a real time application, we assign all the tasks to multicore systems. Branch and bound algorithms, which enables us to partition real time applications optimically, are very complex. the breadth search algorithm is therefore proposed. In order to guarantee the performance of real time applications, we consider the realization of tasks from a circular path for a particular application. the remaining tasks, as well as the tasks that ould not be completed before their deadline, were found with the backtrack search [7] to find the core that is capable of completing the j task before its deadline.

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 $|P| \neq 0$ **do**
begin
$a := select(P)$; {*select the first core from the list P*}
if $depth(a) = \beta(i)$ **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*}

A procedure named arc test has permission to an arc for a given core,when the core uses all the arcs, then it is immediately removed from the list of cores.

```
      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. 6. The procedure backtrach search.

IV. EXPERIMENTS

A simulation of the partitioning algorithm proposed has been performed , transforming real-time applications into flows. This simulation was designed for multi-core systems , the strictly defined number of cores. It is assumed that throughput parameters of each core are exactly the same which is to say, for simplification they are identical. In this simulation study, assumptions 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:..... where i is the mean value of the

$$\Theta_i = \lambda_i \left[\frac{task}{ms}\right] \cdot lmean \left[\frac{instruction}{task}\right]$$

task arrival rates at the core i, lmean 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..... where M is the number of cores in the

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

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

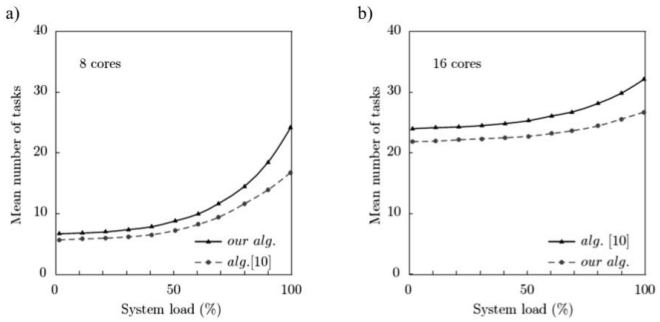


Fig. 6. Mean number of tasks of real-time application in dependence of system load for various number of cores in multicore system

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.

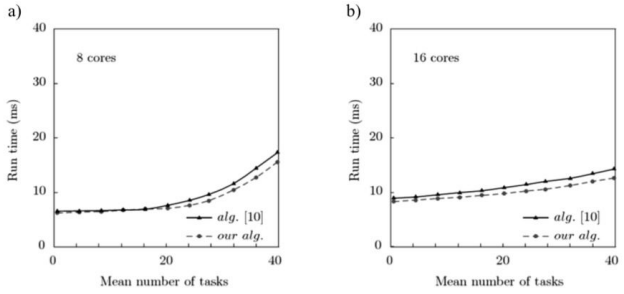


Fig. 7. Run time of tasks of real-time applications in dependence of mean number of tasks for various number of cores

V. CONCLUSION

The purpose of this paper was to present an algorithm for partitioning real-time applications over multiple cores systems. The methodology we used allowed us to assign tasks to this system. Multicore systems allow applications to be run in parallel, this speeds up the processing speed and execution of a real-time application. An algorithm was developed in this work to partition data In multicore systems, the application is divided into tasks that are executed in most parallel. In this way, the execution time of real-time applications is minimized. Furthermore, it takes into account any time constraints and precedence relationships throughout all tasks. Our heuristic reduces complexity by minimizing compared to other algorithms, it reduces the computational complexity and the running time scheduling multiprocessors with the standard method is simple. This model will be extended in the future to include the calculation of An application that communicates between tasks in real-time.

REFERENCES

[1] beniL.,ButtazzoG.,ResourceReservationinDynamicReal-TimeSystems.Real-TimeSystems, vol. 27, No. 2, 2004, 123167.

[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, 7281.

[3] uttazzo G., Bini E., Wu Y., Partitioning Real-Time Applications Over Multicore Reservations. IEEE Trans. on Industrial Informatics, vol. 7, No. 2, 2011, 302315.
[4] hetto H., Silly M., Bouchentouf T., Dynamic Scheduling of Real-Time Tasks Under Precedence Constraints. Real-Time Systems, vol. 2, No. 3, 1990, 181194.
[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, 485451.
[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, 225234.
[7] Knuth D.E., The Art of Computer Programming. Addison-Wesley, Reading, 1986.
[8] ok 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, 7584.
[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, 7584.
[10] cNaughtan R., Scheduling with Deadlines and Loss Functions. Management Science, 6, 1959,

VI. Appendix

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.

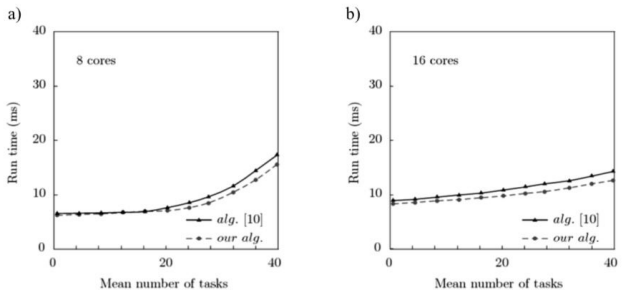


Fig. 7. Run time of tasks of real-time applications in dependence of mean number of tasks for various number of cores

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 $|v| \le 0$;

Fig. 7. A sample application represented with directed acyclic graph (DAG).

$$C^s \stackrel{\text{def}}{=} \sum_{\tau_i \in \Gamma} C_i.$$

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 $d_j \ge t_j + \min_i p_{ij}$ **then goto Step 4a**
else {*there is insufficient time for the task to be executed before its deadline*}

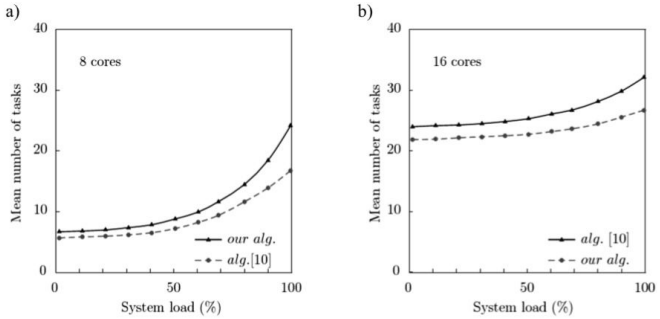
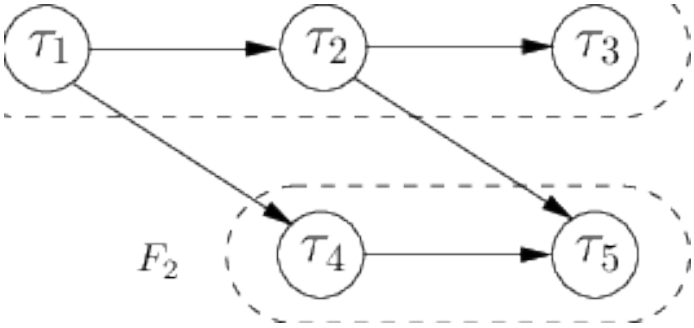


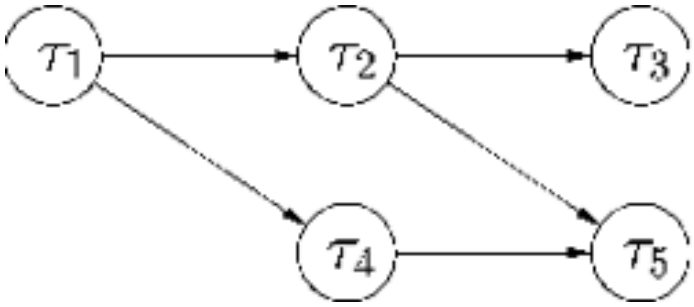
Fig. 6. Mean number of tasks of real-time application in dependence of system load for various number of cores in multicore system



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;
  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;
```