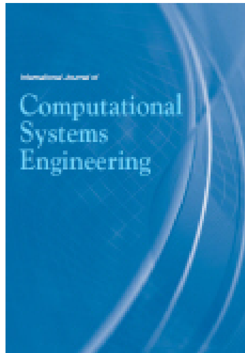
INDERSCIENCE PUBLISHERS
Linking academia, business and industry through research



International Journal of Computational Systems Engineering

ISSN online: 2046-3405 - ISSN print: 2046-3391
https://www.inderscience.com/ijcsyse

Integrated online and offline scheduling of real-time tasks using a co-processor scheduling unit towards dual-mode kernels

Yacine Laalaoui

DOI: 10.1504/IJCSYSE.2022.10054961

Article History:

Received:	04 March 2022
Last revised:	10 August 2022
Accepted:	10 August 2022
Published online:	19 May 2023

Copyright © 2022 Inderscience Enterprises Ltd.

Integrated online and offline scheduling of real-time tasks using a co-processor scheduling unit towards dual-mode kernels

Yacine Laalaoui

Department of Information Technology,
Taif University,
Taif, Kingdom of Saudi Arabia
Email: y.laalaoui@tu.edu.sa

Abstract: In this paper, we integrate online and offline scheduling of real-time tasks using two processing elements. The scheduling task executes on the first PE (called scheduling PE or the co-processor). User tasks execute on the second PE (called main PE). The main objective is to decrease the scheduling overhead from the main PE. The scheduling PE executes both online and offline algorithms. Thus, it runs on two different modes, scheduling and dispatching. The scheduling mode executes first at selection instants using an online algorithm. The offline algorithm executes in parallel to users tasks. Once the offline algorithm finds a feasible schedule, the scheduling PE switches to dispatching mode. We further describe a new task states that fit the proposed design. Finally, we explain the resolvability of the problem of nonpreemptive scheduling using the IBM ILOG-CP solver and Xu and Parnas's algorithm using the proposed design.

Keywords: real-time tasks; scheduler; dispatcher; optimal scheduling; online; offline; co-processor scheduling.

Reference to this paper should be made as follows: Laalaoui, Y. (2022) 'Integrated online and offline scheduling of real-time tasks using a co-processor scheduling unit towards dual-mode kernels', *Int. J. Computational Systems Engineering*, Vol. 7, No. 1, pp.19–29.

Biographical notes: Yacine Laalaoui has received his BSc, MSc and PhD from the Ecole National Superieur d'Informatique of Algeria in 2002, 2005, and 2010 respectively. Currently, he is an Associate Professor in Information Technology Department at Taif University. His main research interests span artificial intelligence and operations research fields with particular emphasise on scheduling, planning, and knapsacks to solve problems in real-time systems, cloud computing, and transportation domains.

1 Introduction

Scheduling tasks is one of most important challenges in real-time operating systems (RTOS). RTOS kernels use either schedulers or dispatchers. The scheduler is used if an online algorithm is integrated into the kernel. The dispatcher is used if an offline algorithm is utilised to find a desired solution before integration of that solution into the kernel (Xu and Parnas, 2000). Online algorithms tend to be effective in cases where the problem is polynomially solvable which are rare in practice. The offline counterpart has been proposed to bridge the limitation of online algorithms; but, they are expensive in computational point of view due to the NP-hardness of most of scheduling problems (Garey and Johnson, 1979).

1.1 Online scheduling

In online scheduling, task arrival is not known to the scheduler. Each admitted task is inserted into the ReadyQueue. The latter list is used to manage the order of arrival of tasks awaiting their execution. Each task T_i is assigned a heuristic value according to its timing requirements, usually called *priority*. This heuristic value is used to fix the order of T_i in ReadyQueue before it takes the processor. After priority assignment, the scheduler selects tasks according to the decreasing or increasing order of their priorities. The instant of taking the processor depends on the computation times of the task currently occupying the processor. The start time of T_i depends also on the scheduler ability to insert idle times. When T_i has been assigned the processor, it will be removed from ReadyQueue.

It is known that online schedulers consider three states for each task: *ready*, *blocked* and *running* (Tanenbaum and Woodhull, 2006). ReadyQueue is the set that contains all tasks in *ready* state while BlockedQueue is the set that contains all tasks in *blocked* state. When a task is admitted to be executed, the system decides whether to put this task in *ready* state or in *blocked* state. The task waits until the

Copyright © 2022 Inderscience Enterprises Ltd.

current system time coincides with its release date to be moved from *blocked* state to *ready* state.

Whatever the implementation of ReadyQueue (linked-list or heap), its size is kept relatively small to reduce the search for the task with the highest priority. When a task is selected for execution by the scheduler, it is then removed from ReadyQueue. In preemptive scheduling, the task in *running* state can be moved again to ReadyQueue if another task with higher priority is released in ReadyQueue. This is not the case in non-preemptive scheduling where the running task releases the processor after completing all its computing units. Often, tasks are periodic and new requests for a task are activated at regular time intervals. The system decides to move those requests again to *ready* or *blocked* states.

The major issue in this design is the size of the ReadyQueue list. When this size is large, it automatically causes much system overhead and maybe timing constraints violation. Moreover, since the scheduling problem is NP-Hard in most of the cases, there is no guarantee to find the optimal schedule using a simple heuristic even if the size of the list is small.

1.2 Offline scheduling and dispatching

Offline schedulers have the aim of producing desired solutions out-field. Unlike the online approach, the offline scheduling algorithm has a complete knowledge about the task set to be scheduled. The result is saved into an array data structure to be consulted later at run-time by the dispatcher.

The dispatcher is the RTOS component that assigns jobs to processors based on their order in the pre-prepared array using $O(1)$ time complexity. The dispatcher is invoked the first time to select the first job to take the processor. The dispatcher is invoked again after the completion of the first job. Similarly, the second job is selected from the pre-prepared array. All remaining jobs will be selected after each others with respect to the pre-computed order.

The common property of all offline algorithms is the exponential time complexity because of the NP-hardness of the problem. But, some other existing algorithms have an exponential space complexity too such as those based on branch-and-bound. Most of the proposed algorithms that solve the problem of scheduling tasks under timing constraints could be found in Shepard and Gagne (1991), Xu and Parnas (1992), Xu (1993), Cavalcante (1997) and Abdelzaher and Shin (1999).

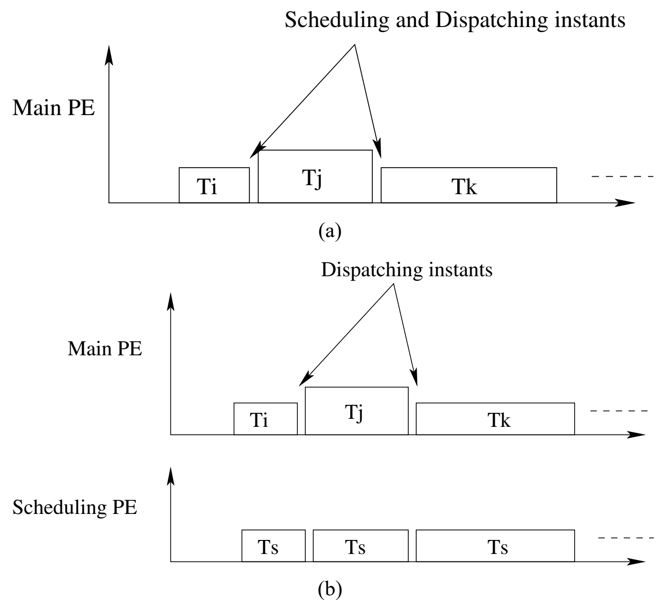
1.3 Aim of this paper

In this this paper, we propose a co-processor scheduling approach that combines both online and offline approaches in the hope of producing better results.

This paper aims the execution depicted in Figure 1(b) instead of the classical execution where both the dispatcher and the scheduler modules are executed on a single processing element (PE) Figure 1(a).

In Figure 1(b), the dispatcher module is executed by the main processing element (MPE) while the scheduling module is executed by the scheduling processing element (SPE). The task called T_s is the scheduling program. It has variable computation durations because of the variation of the size of ReadyQueue. In this design, SPE prepares the task with the highest priority and the dispatcher reads only that task instead of searching it from the set ReadyQueue.

Figure 1 Scheduling and dispatching times. T_s is the scheduling task, (a) serial execution of user and system tasks (b) parallel execution of the scheduling overhead



Notes: T_i, T_j and T_k are user tasks.

The remainder of this paper is organised as follows. Section 2 presents the most related works. Section 3 gives a colloquial motivation. Section 4 shows our task model. Section 5 describes the proposed design with some other consideration in Section 6. Section 7 shows the experimental work. The paper is concluded in Section 8.

2 Related work

Burleson et al. (1999) proposed a scheduling coprocessor for their spring system. Their proposed design uses an iterative heuristic to find good solutions. This means the non-optimality of their scheduling approach. Starner et al. (1996) proposed also scheduling unit to increase time predictability in real-time systems. In the latter design, the scheduling unit executes a priority-based simple heuristic algorithm in preemptive context. Salcic et al. (2006) proposed a scheduler support unit for reactive microprocessors. All cited works use simple heuristics which are limited in terms of finding good solutions. Such limitation is due to the lack of using optimal algorithms. To overcome this problem, efficient artificial intelligence (AI) algorithms and solvers are required.

Wang et al. (2003) integrated online and offline schedulers to address the problem of accommodating event-driven tasks into time-driven schedules. The offline algorithm produces a pre-schedule of the time-driven workload with sufficient embedded slacks to accommodate the event-driven workload. The online scheduler follows the execution order provided by the offline scheduler to assign tasks to the processor. Their proposed approach uses a single PE that schedules and executes users tasks. Further, the offline component executes independently of the whole system to produce the desired solution. A similar approach have been proposed in Isovic and Fohler (2009) to address multiple sets of tasks: periodic, sporadic, and aperiodic.

All existing approaches are either pure online or pure offline. Purely online approaches can be found in Burleson et al. (1999), Starner et al. (1996) and Salcic et al. (2006). Purely offline approaches can be found in Xu and Parnas (1992, 1993) and Cavalcante (1997). Purely online approaches use, solely, simple heuristics that are executed by the co-processor unit while purely offline approaches use, solely, complex AI solvers to be executed by the co-processor unit or by the same processor executing users tasks. Additionally, in purely offline approaches such as Wang et al. (2003) and Isovic and Fohler (2009), the online component cannot start until the completion of the offline component.

In this paper, we propose the integration of both scheduler and dispatcher into the same RTOS kernel. To this end, we use a parallel architecture with two PEs to schedule and execute users tasks.

3 Colloquial motivation

In this section we give a brief colloquial example to the reader in order to simplify the presentation of the proposed approach.

Consider a mechanical repair shop with two persons, the manager and the technician. The manager is responsible on accounting and scheduling repair requests. The technician is taking care of the repair tasks. When requests arrive at early morning, the manager starts doing scheduling, then he will give the resulting plan to the technician to start the execution (serving clients) according to the provided plan.

The manager can either give an exact solution or an approximate one. The exact solution provides the optimal value, for example the optimal makespan (latest completion time). The approximate solution gives a feasible solution not necessarily the optimal one. The fact is that getting the optimal solution using an exact algorithm is expensive in computational viewpoint due to the NP-hardness of the scheduling problem. This means, the manager can stay days doing the scheduling task if the number of arrived requests is big. At this time, the technician is idle awaiting the result of the scheduling task. It is clear that this situation is not practical at all; the technician should not be idle. Therefore, providing an approximate solution by the manager seems to be more practical simply because it can be provided quickly (using polynomial time algorithm).

In this paper, we propose to combine both approaches. The manager computes a quick plan and gives it back to the technician to start serving clients. Once the technician starts working, the manager will try to get the optimal plan whatever the time that will take. Once the optimal plan is ready, the technician can switch from the approximate plan to the optimal one.

From the operating systems point of view, the co-processor scheduling unit gives a quick approximate schedule to the main processor (to execute user activities). Then, the co-processor unit will take its time to find the optimal solution; once found, the main processor can switch from the approximate schedule to the optimal schedule. This design can be of potential use in real-time systems where the problem of scheduling remains a major issue.

4 Preliminaries

4.1 Notations and definitions

The following notations and definitions are used along the present paper:

- n: The number of tasks to be scheduled.
- T_i: A task i, $i = 1, ..., n$.
- $\Pi = \{T_i | i = 1, ..., n\}$ it is the set of all tasks to be scheduled.
- r_i: Release date of the task T_i.
- C_i: Computation time of the task T_i.
- D_i: Deadline of the task T_i. It is assumed that it must be greater than or equal to its computation time C_i, otherwise no feasible schedule could exist.
- d_i: The relative deadline of the task T_i.
- P_i: Period of the task T_i.
- LCM: The least common multiple of all task periods from Π. It is also called the meta-period. It defines the scheduling interval [0, LCM] in which all tasks must be scheduled. Each task T_i from Π has exactly $\frac{LCM}{P_i}$ jobs (called also requests or invocations). In a periodic real-time system, the same jobs are released at regular LCM meta-periods infinitely. Finding the sequence of satisfied job within the first meta-period suffices to predict the real-time system (Xu and Parnas, 1993).
- $\Gamma = \{T_{i,k} | 1 \leq i \leq n, 1 \leq k \leq \frac{LCM}{P_i}\}$: It is the set of all jobs resulting from each task.
- $s_{i,k}$: The start time of the kth job of the task T_i.
- $e_{i,k}$: The end time of the kth job of the task T_i.
- D_ReadyQueue: It is the set of ready jobs computed by an offline algorithm. This set is consulted periodically by the dispatching module.

- S_ReadyQueue: For convenience, we use this notation to emphases the set of ready jobs of an online kernel instead of ReadyQueue notation.
- ReadyRequest: It stores the most eligible job to move into running state either in dispatching mode or scheduling mode.
- NEA: Non-efficient algorithm that is often used in online approaches. We assume that NEA has a polynomial time complexity.
- EA: Efficient algorithm that is often used in offline approaches. We assume that EA is optimal and it has an exponential time complexity.

A task T_i in a schedule is said to be *satisfied* when its deadline is met; otherwise it is said to be *unsatisfied*.

4.2 Task model

A typical real-time application is a finite set Γ of periodic independent tasks to be scheduled on a single PE architecture. Each task T_i is characterised by the standard timing parameters: $<r_i, C_i, D_i, P_i>$. Let $\mathcal{S}$ be the sum of all jobs' computing units:

$$\mathcal{S} = \sum_{i=1}^{|\Gamma|} C_i \tag{1}$$

Let $\mathcal{U}$ be the system load for the task set Π that is defined as follows:

$$\mathcal{U} = \sum_{i=1}^{n} \frac{C_i}{P_i} \tag{2}$$

We assume that preemption between tasks is not allowed and resources can be shared without high level synchronisation tools (Yacine and Nizar, 2014). Non-preemptive scheduling has many benefits over preemptive scheduling that can be summarised in the following:

- Non-preemptive schedulers offer much less switching context and system overhead.
- Non-preemptive schedulers on single processor systems guarantee exclusive access to shared resources without utilising complex resource management protocols.
- Non-preemptive schedulers are widely used in message scheduling in distributed systems where the atomic unit to send and receive data is a frame with a fixed size.
- Non-preemptive schedulers are more suitable in real-time data-bases where transactions are usually performed non-preemptively.
- Device access and I/O tasks are performed non-preemptively.

4.3 Objective function

The objective is to find the optimal/near-optimal solution which is a sequence of all satisfied jobs. If this solution does not exist, then a solution that has the maximum lateness ($max_lateness$) would be returned. This means tasks are authorised to violate their deadlines. The lateness (l_i) of each job J_i is defined as follows:

$$l_i = MIN(e_i - d_i). \tag{3}$$

The maximum lateness is taken over all obtained latenesses:

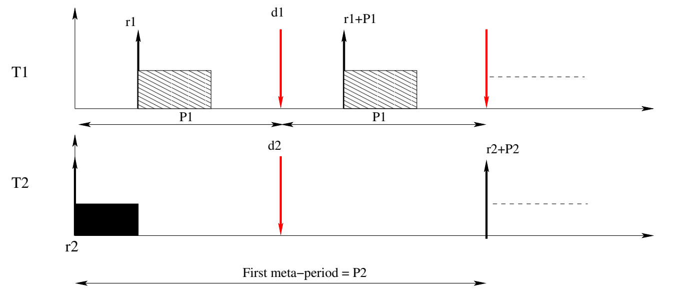
$$max_lateness = MAX_{J_i \in \Gamma}(l_i) \tag{4}$$

Our final target is to assign jobs to the processor in accordance to their order in the optimal/near-optimal value with the respect to the objective function. For example, if the NEA produces a solution with some jobs violating their timing constraints, the EA algorithm attempts to find a solution with all jobs meeting their timing requirement if it exists. In the remainder of this paper, we will refer to this solution as the *desired solution*.

4.4 Example

Figure 2 shows an example of running two periodic tasks during the first meta-period. The first task has exactly two jobs and the first task has exactly one job over the interval [0, LCM]. LCM is equal the period of the second task P2. P2 is equal the double of P1.

Figure 2 Example of two periodic tasks during the first meta-period (see online version for colours)



Notes: Up-arrows are release dates and down-arrows are deadlines. (r1, d1, P1) and (r2, d2, P2) are tasks' timing parameters (release dates, relative deadlines and periods respectively).

5 The proposed approach

The proposed approach uses parallel processing to solve the problem of scheduling non-preemptive tasks. We also combine online and offline scheduling to reach the optimal/near-optimal sequencing of all jobs.

5.1 Two PEs architecture

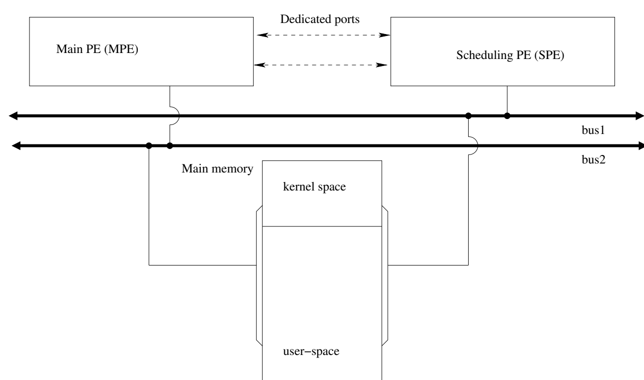
First, we describe the suitable hardware architecture. We use two PE called the SPE and the MPE. The first is the co-processor unit that is dedicated to execute the scheduling task. The second is dedicated to execute the user tasks. The SPE is tasked to prepare the desired solution by scheduling ready requests. The MPE executes ready requests one-by-one in the order prepared by the SPE. Therefore, the load on the MPE will be significantly decreased compared to the classical design in which both user and systems tasks are executed on the single processing unit.

We propose to use only one common memory to share data between both SPE and MPE. A cache memory can be added to the MPE for better performance. There is no need to add such type of memory for the SPE since the real content of ReadyQueue is required during each scheduling time. Any change made by the first PE (SPE) in the shared memory should be visible to the second PE (MPE).

Another important component in this architecture is the linking bus for possible communications between SPE, MPE, and the shared memory. Two buses have been used to connect all components together. The first bus connects the MPE with the shared memory. The second bus connects the SPE with shared memory too. This means that each PE can access the bus on its own and no contention is possible.

We propose to use a RAM memory (with a cache memory for better performance) that can connect two buses and as a result it can be shared by two PEs as can be seen in Figure 3. Each PE has its own bus to connect to the shared memory. The access to non-shared zones can be performed easily without any bottleneck. The protection of shared zones would be left to the programmer by using standard protection tools such as semaphores.

Figure 3 The overall hardware architecture



5.2 Scheduling and dispatching modes

Our second point to explain is the two modes of the kernel: scheduling and dispatching modes. The scheduling mode runs if the desired solution has not yet been found. Once the latter solution is found, the kernel switches to the dispatching mode. Therefore, a Boolean variable is needed to check which mode to use by the kernel. Let us call $B_Dispatch$ such Boolean variable. $B_Dispatch$ is set initially to $false$ since the scheduling mode is firstly used before finding the desired solution. Algorithm 1 shows how to switch between scheduling and dispatching modes. Notice that this code is a system task and it executes in parallel to user's tasks on the SPE.

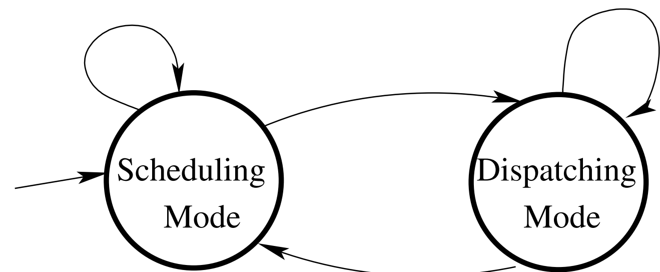
Algorithm 1 Dispatching and scheduling switches

```
1: if not (B_Dispatch) then
2:     use scheduling mode
3: else
4:     use dispatching mode
5: end if
```

If $B_Dispatch$ is $false$, then the kernel uses the scheduling mode. This means that usual heuristics, such as RM and EDF are used to select jobs to be assigned to the MPE. The variable $B_Dispatch$ is set to $true$ once the desired solution has been found, i.e., once the EA algorithm terminates with that desired solution. If $B_Dispatch$ is true, then the dispatching mode will be used infinitely or at least until changes of the current task set, i.e., after admission of new task(s).

Figure 4 shows a finite automata representation of a dual-mode kernel. When the kernel starts, it enters the scheduling mode. The kernel stays (or looping) in the scheduling mode until finding the desired solution by the SPE. The transition to the dispatching mode is performed in both directions scheduling to/from dispatching. The kernel stays (looping) in dispatching mode infinitely or until changes of the task set, i.e., after arrival of new tasks.

Figure 4 Finite automata representation of a dual-mode kernel



5.3 Scheduling and dispatching modules

Our third point to explain is the scheduling and dispatching modules. First, we propose to break ReadyQueue set into two sets called D_ReadyQueue and S_ReadyQueue. S_ReadyQueue is used during the scheduling mode while D_ReadyQueue is used during the dispatching mode because the order of jobs is different during each mode. D_ReadyQueue is the set of jobs at *ready* state during the scheduling mode. D_ReadyQueue is the set of jobs that are ready during the dispatching mode.

5.3.1 Dispatcher module

The dispatching module is executed by the SPE and it is simplified as much as possible to reduce the time complexity to $O(1)$. The SPE reads each ready task from the shared zone ReadyRequest stored in the main memory. There is no need to search for the most eligible request from ReadyQueue. Let T_d be the dispatching task and C_d its computation time. This task executes once the desired solution has been found and stored in D_ReadyQueue set.

5.3.2 Scheduling module

In fact, the SPE has two main tasks to execute when the relevant event is received namely the scheduling task, noted T_s, and the dispatching task, noted T_d. The scheduling task T_s is a composite task that includes the following sub-tasks:

1 *Selects the next job from S_ReadyQueue:* If the complete feasible solution has not yet been prepared by the SPE, then the SPE works as usual in online mode. The SPE selects the next job to be written into ReadyRequest zone according to a plain heuristic such as EDF and RM. Notice that The SPE takes jobs from the set S_ReadyQueue and not from D_ReadyQueue set. Let us call this task $T_{s,1}$ and $C_{s,1}$ is its computation time.

2 *Calculation of the LCM:* This calculation involves adding the relevant jobs for each task according to its period. Each new job is appended to the set $open$. The latter set is the input to the planning (scheduling) algorithm to produce the desired solution including jobs that will be stored in D_ReadyQueue. It is well know that the task of calculating the LCM has an exponential time complexity; thus we limit our study to cases where periods are harmonic (Choquet-Geniet and Grolleau, 2004). Let us call this task $T_{s,2}$ and $C_{s,2}$ is its computation time. The latter task executes independently of the task $T_{s,1}$.

3 *Execution of the scheduling algorithm:* This task executes in parallel to the user tasks to find the desired solution. It reads inputs from the set $open$ using producer/consumer paradigm. Notice that the *producer* is standard while the *consumer* has to read all non-blocked requests from the buffer and put them into $open$. The input to the scheduling algorithm is the set $open$ and the output is the set D_ReadyQueue. The nature of the scheduling algorithm has an impact on the efficiency of this new design since it takes more time than simple heuristics. This issue will be discussed separately in the next sections. Let us call this task $T_{s,3}$ and $C_{s,3}$ is its computation time.

Let C_s be the computation time of the scheduling task T_s. C_s can be computed as follows:

$$C_s = C_{s,1} + C_{s,2} + C_{s,3} \tag{5}$$

$T_{s,1}$ is the task that should be executed first in order to select the next job from S_ReadyQueue because the expected desired solution is not ready at time t equal 0. Further, this task should not be preempted by any other task namely $T_{s,2}$ and $T_{s,3}$. If $T_{s,2}$ could be preempted only by the task $T_{s,1}$. The latter preemption happens in case of a great number of jobs that must be appended to the set $open$. $T_{s,2}$ resumes its execution after the completion of $T_{s,1}$ (selection of one job only). $T_{s,3}$ is the task that takes the longest computation time since it executes the EA algorithm. $T_{s,3}$ executes after the completion of $T_{s,2}$ because after that time the set *open* becomes full of all jobs to be scheduled. The task $T_{s,3}$ could be preempted by $T_{s,1}$ only.

5.3.3 Required number of meta-periods

Given a set of jobs Γ to be scheduled within a period equals the meta-period LCM. The objective is to find the minimum number of meta-periods sufficient to reach the desired solution by the SPE. In other words, how many meta-periods are required to find the desired solution?

Let z be the minimum number of meta-periods to reach this desired solution. z can be computed as follows:

$$z = \left\lceil \frac{C_s}{\mathcal{S}} \right\rceil \tag{6}$$

where C_s is the computation time of the scheduling task which is also the time taken by the search algorithm to find the desired solution.

It is worth to note that the SPE is not given the total time T_s during each meta-period since the latter PE executes the scheduling task in addition to the search task. During each meta-period, the SPE takes a slot from T_s until finding the desired solution. Therefore, the total number of required meta-periods should be minimised. Since the latter problem is NP-Hard, the time T_s taken by the SPE is exponential (Jeffay et al., 1991). The ideal case is that an EA algorithm is able to find the optimal solution within only one meta-period (z equal to 1). In other words, the sum of all jobs $\mathcal{S}$ processing times is enough for the EA to find the desired solution.

To simply the proposed design, we assume that the earliest switching to dispatching mode could be done during the second meta-period. This means that the desired solution has been found during the first meta-period and z is greater than or equal 1. Switching from scheduling to dispatching mode during the first meta-period is left for future research works.

5.3.4 Example

Consider an example of a job set Γ with an LCM equal 500 units of time. Assume that the SPE could execute the scheduling algorithm during 250 units of time during each LCM. Recall that the SPE will be busy in doing the scheduling task only when the MPE is busy running user tasks. The 250 units of times is the value of $\mathcal{S}$ that is equal the sum of all jobs' computing units. Assume that the scheduling algorithm takes 1,000 units of time to

find out the desired solution. Thus, the number of required meta-periods z is equal $\frac{1,000}{S}$ that is equal 4. After 4 meta-periods, the desired solution will be ready and the RTOS kernel must switch to the dispatching mode instead of the continues use of the scheduling mode.

5.4 *Task states and shared memory zones*

Our fourth point to explain is the states of real-time task when the co-processor is utilised to schedule tasks. In the literature, there is a missing piece of information, yet important, which is the adequate task states when using two communicating PEs to schedule/run real-time task sets. In the present section, we will describe our new states of a real-time task.

When a task is accepted by the admission controller, it is in the created state and it is saved into a shared memory zone called buffer. When the ready task T_i is selected for execution (to running state), its next release T_{i+1} moves to the state blocked and it is stored in buffer zone. The SPE reads task requests from that shared zone to prepare the desired solution where selected requests are moved now to the new state called *ready-ahead*. The most eligible task is then selected and written into the shared zone called ReadyRequest with a flag ready state. The MPE executes the dispatcher module to take the ready request from ReadyRequest zone and move it to the running state.

Communication points between both PEs are buffer and ReadyRequest zones. Therefore, an adequate communication and synchronisation tools are required to protect them again simultaneous access. The producer/ consumer paradigm could be used to protect the buffer and signals to protect ReadyRequest. It is worth to note that all used data structures are stored in the kernel space. The SPE keeps all *ready-ahead* requests in both *open* and D_ReadyQueue where the latter contains the expected desired solution and the former contains a set of jobs not yet appended to the expected execution plan (D_ReadyQueue). When the SPE terminates the scheduling task, the most eligible request is selected from the front of the set D_ReadyQueue and written into ReadyRequest zone. Then, the MPE selects the ready job from ReadyRequest zone.

Communications between both MPE and SPE are done using *producer/consumer* paradigm. The communication points are shown in Figure 5 in red color called buffer and ReadyRequest. The buffer contains N items while ReadyRequest has only one item and the access to this zone is done in exclusive mode using signal/wait mechanism on dedicated ports for example. It is worth to note that the producer is a standard module while the consumer has to read all the buffer items and put them into the set *open*. In fact, we mean by a standard module the action of putting/getting only one item from/into the shared buffer and here we make a slight modification of the consumer to allow moving more than one item from buffer to *open*.

Figure 5 Tasks states in non-preemptive scheduling on PEs (see online version for colours)

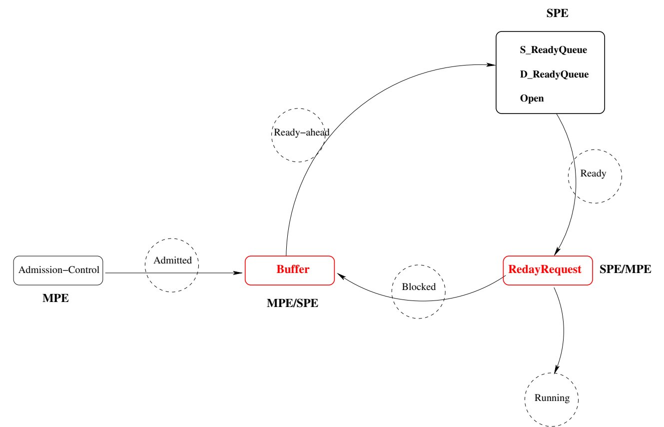


Figure 6 Communication zones and task's states, (a) buffer zone (b) ReadyRequest zone (see online version for colours)

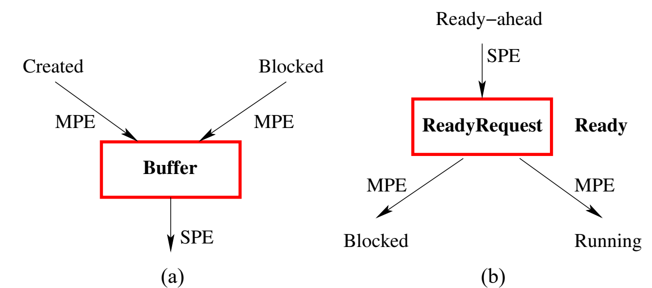
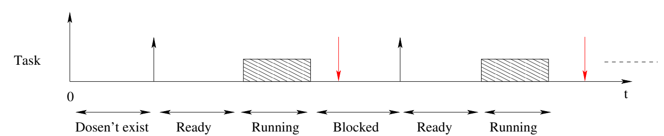


Figure 7 Tasks states along the execution (see online version for colours)



6 Further considerations

6.1 *Context switching in SPE*

The SPE executes two different algorithms sequentially: the NEA and the EA algorithms. The NEA, such as RM or EDF, executes to select the next job to be sent to the MPE. Recall that our aim in this paper is to reduce the scheduling overhead to $O(1)$ time complexity. This means that the next job to be executed on the MPE should be ready before the completion of the job currently using the MPE. The NEA is executed before finding the desired solution and after that the RTOS kernel switches to the dispatching mode as stated above. Once the next job has been selected using the NEA, the second algorithm executes the search algorithm to find the desired solution.

Each time the scheduling mode executes (task $T_{s,1}$), there is no need to save the context since this mode ends at selecting the task of the highest priority. The EA is executed until finding the desired solution. The task of executing the EA algorithm must be preempted from time

to time to switch the SPE to the scheduling mode. Each preemption needs saving the context of the EA.

Further, the context must be restored when the EA algorithm is executed. Notice that the first time when the EA is executed, there is no context to restore as it is shown in Algorithm 2.

Algorithm 2 Context saving/restoring points

if not ($B_Dispatch$) **then**
  1. Execute the non-efficient algorithm ($T_{s,1}$)
  $Restore - the - context - if - any$
  2. Calculate the LCM ($T_{s,2}$)
  $Save - the - context - if - preemption$
  $Restore - the - context - if - any$
  3. Execute the efficient algorithm ($T_{s,3}$)
  $Save - the - context - if - preemption$
else
  Use dispatching mode (T_d)
end if

The overhead of switching the context on the SPE ends once the desired solution is found. No more context switching would be needed. It is clear that this overhead on the SPE does not affect the user tasks on the MPE. Thus, it does not affect the whole system performance.

6.2 Offline scheduling algorithm to use

The SPE executes a specific EA scheduling algorithm to prepare the desired solution. The input to the EA algorithm is the set $open$, and its output is a solution stored in D_ReadyQueue. In AI point of view, this problem is a constraint satisfaction problem (CSP) and algorithms that solve this type of problems are optimal or non-optimal (Yacine and Nizar, 2014). Optimal algorithms, that we called EA algorithms, provide the guarantee of finding the optimal solution. Optimal algorithms are either complete-and-repair (Minton et al., 1992) or backtracking algorithms Rossi et al. (2006). Complete-and-repair algorithms have often exponential space complexity because they are often based on branch-and-bound. Backtracking algorithms have often polynomial/pseudo-polynomial space complexity because they based on depth-first-search (Van Beek, 2006).

Now, the question is that which scheduling algorithm is more suitable to the proposed design? Complete-and-repair or backtracking? Let us examine the complexity of each class of algorithms. Complete-and-repair algorithms have been widely used in offline scheduling of real-time tasks (Shepard and Gagne, 1991; Xu and Parnas, 1992; Xu, 1993; Cavalcante, 1997; Abdelzaher and Shin, 1999). Since the problem under investigation is NP-hard, then the space complexity is exponential because all these techniques are based on branch-and-bound. The size of the search tree grows exponentially with the input job set size. Thus, the kernel-RAM space could be exhausted if the RTOS integrates this type of techniques.

Remark 6.1: If the RTOS integrates a branch-and-bound algorithm, then there is a risk of exhausting the kernel RAM space.

Often, the kernel space is smaller than the user space. Therefore, it is not recommended to use branch-and-bound algorithms to produce the desired solution by the SPE to avoid the risk of exhausting the kernel space. In turn, backtracking algorithms have polynomial/pseudo-polynomial complexity. Thus, it is almost impossible to exhaust the whole RAM space when these approaches are used. This feature makes backtracking approaches more suitable to be integrated into an RTOS kernel.

To summarise, a suitable scheduling algorithm for the above architecture should have two main properties:

1. *Continues improvement of the solution quality:* The used algorithm should be able to improve the solution quality until finding the optimal one. Optimal algorithms have this feature since they guarantee to reach the optimal solution.
2. *Linear space complexity:* The used algorithm must have a polynomial/pseudo-polynomial space complexity to keep the main memory less used by the OS kernel. We strongly believe that there is no need to go for an algorithm with an exponential space complexity to avoid memory exhaustion situations.

7 Experimental study

The objective of the present experimental work is provide the reader an idea on how the proposed approach works. The evaluation of existing algorithms is not within the scope of the present paper. We will use two existing algorithms and we will show their behaviour when applied to schedule sample real-time task sets.

7.1 Experimental setup

7.1.1 IBM ILOG-CP solver

IBM ILOG-CP solver is one of the widely used tools in constraint programming. IBM ILOG-CP is an exact solver that uses efficient constraints programming techniques namely Backtracking and propagation algorithms to search for desired solutions. In the present study, we used IBM ILOG-CP solver version 12.2 on laptop machine with MS Windows operating system, Intel CORE i5 processor and 6 GB of RAM space. In software engineering point of view, there is no need to develop from scratch an ILOG-like solver to be integrated into a RTOS kernel because this solver offers very useful application programming interfaces. IBM ILOG-CP solver has a specification language to write a constraint program for the problem to be solved (Laborie and Rogerie, 2008, 2009). The constraint program that we wrote to solve the problem

of single processor scheduling of tasks with their own timing constraints non-preemptively is shows in Figure A1. This constraint program uses *interval variables* and *nonOverlap* features to boost the search performance of this solver.

7.1.2 Xu and Parnas algorithm

Xu and Parnas is a widely used brand-and-bound algorithm to solve the problem of preemptive scheduling of hard-real-time tasks under timing, precedence and exclusion constraints on single processor architecture. The implementation of this algorithm has been done using C++ on the same laptop machine. In order to force the non-preemptive scheduling of the input task sets, we have added exhaustively exclusion constraints between each pair of tasks. This algorithm has been added to the current study in order to show empirically that algorithms with exponential space complexity can generate a great number of nodes; thus can lead to the exhaustion of the RAM space.

7.1.3 Time quantification

In all our experimental work, we disregarded the operating system overhead times. We took the time reported by the solver only which is the amount of time to find the optimal solution for each task set. To compute property the optimal solution, we need the quantification of the time units used in all timing parameters <r, C, D, P>. In the following, we assume that each computing unit is equal to 0.001 second.[1] For example if a task T has <1, 2, 10, 20> timing parameters, then the new parameters according to the proposed assumptions are: <0.001, 0.002, 0.01, 0.02>. If the LCM is equal to 500, then the new value is $500 * 0.001$ which is equal to 0.5 second. If the $\mathcal{S}$ value is 250, then its new value is $250 * 0.001$ which is equal to 0.25 second. Thus, if IBM ILOG-CP solver takes ten seconds to find the optimal solution, then there are 10/(0.25) meta-periods which is equal to 40. After 40 meta-periods, no more time will be wasted by the SPE.

7.2 Experimental results

7.2.1 Results on nine-pumpe problem instance

Mine-pumpe is a real-world example used in academia that contains 782 real-time jobs according to the timing parameters set in Cavalcante (1997). The corresponding system load $\mathcal{U}$ is 0.3045. This job set can be considered relatively large. $\mathcal{S}$ is equal to 9,135 and LCM is equal to 30,000. Thus, the quantified timing parameters are 30 seconds and 9.135 seconds for LCM and $\mathcal{S}$ respectively. This job set is solvable at root node using IBM ILOG-CP solver within a very short time slot equal to 0.1 seconds. Thus, there are 0.1/9.135 which should be approximated to 1. This means that only one meta-period is quite enough to find the solution of jobs that satisfy all timing constraints.

This example confirms that sometimes even large real-time task sets could be solved within a very short period. It is not recommended to let the online scheduler violating deadlines infinitely if the input task set is easy to solve using an EA algorithm.

Table 1 IBM ILOG-CP solver results

Task set	*#jobs*	$\mathcal{S}$	$\mathcal{U}$	*ILOG-CP*		*Xu and Parnas*		
				Time (sec)	*z*	*#nodes*	*Time (sec)*	*z*
I1	54	462	0.924	37	81	4	0.211	*1*
I2	55	463	0.926	7.5	17	4	0.222	*1*
I3	56	464	0.928	7	16	4	1.77	*4*
I4	57	467	0.934	16	35	4	0.237	*1*
I5	58	468	0.936	16	35	8	1.474	*4*
I6	55	483	0.966	8	17	6	0.363	*1*
I7	56	484	0.968	7.5	16	6	1.61	*4*
I8	57	485	0.97	7.5	16	6	0.4	*1*
I9	58	486	0.972	7.5	16	6	0.415	*1*
I10	59	487	0.974	7.7	16	6	1.565	*4*
I11	60	488	0.976	3.5	8	6	0.471	*1*
I12	61	474	0.948	5.5	12	6	0.42	*1*
I13	62	476	0.952	2.7	6	7	1.452	*4*
I14	63	477	0.954	13	28	7	1.422	*3*
I15	64	480	0.96	6	*13*	1,812	142.825	298
I16	65	481	0.962	6	*13*	1,812	149.88	312
I17	57	489	0.978	40	82	12	1.84	*4*
I18	63	485	0.97	35.5	*74*	7,401	496.381	1,024
I19	63	481	0.962	13.5	29	8	1.348	*3*
I20	64	482	0.964	73.5	153	9	1.211	*3*
I21	65	485	0.97	33	69	18	1.663	*4*
I22	66	486	0.972	33	68	25	2.492	*6*
I23	67	487	0.974	34	70	25	3.383	*7*
I24	68	488	0.976	30	62	25	3.318	*7*
I25	69	489	0.978	35	72	25	3.175	*7*
I26	54	458	0.916	8	18	15	1.242	*3*
I27	67	484	0.968	17.5	37	6	0.6	*2*
I28	68	485	0.97	35	73	6	0.633	*2*
I29	69	486	0.972	35	73	6	1.32	*3*
I30	70	487	0.974	159	327	6	1.297	*3*

Notes: The time is measured in seconds. the LCM is 500 units of time. Each unit of time corresponds to 0.001 second.

7.2.2 Results on pseudo-random task sets

We have developed a pseudo random generator to get 30 task sets. The pseudo-random generator uses an initial well known task set that have a feasible solution. Based on such feasible solution, we add more tasks by exploiting available idle times. In this way, we ensure that each generated task set would have a feasible solution too.

The used initial task set contains 26 jobs with a load 0.8 while the LCM is equal 500. All generated task sets are difficult to solve since the IBM ILOG-CP solver takes a long time to find the optimal solution using the constraint program described in Appendix.

Table 1 shows the time taken by each solver ILOG-CP and Xu and Parnas. The latter solver succeeds in finding the feasible solution within a time less than ILOG-CP

solver in most of the cases. Nevertheless, Xu and Parnas algorithm gives a very long time in three cases. This difference of behaviour is related to the nature of algorithm itself. ILOG-CP solver uses backtracking techniques with only one unsatisfied job at the end of each partial feasible solution while Xu and Parnas's algorithm uses branch-and-bound techniques and it attempts to satisfy many unsatisfied jobs at each node.[2]

Xu and Parnas algorithm has the drawback of the increased size of the search tree. As can be seen in Table 1, this algorithm can generate 7,400 nodes in relatively small task sets (with a number of jobs below 63). This result provides a good example when the scheduling can generate a great number of jobs for small task sets. Therefore, we believe that this algorithm is not suitable to be built into an RTOS kernel upon the risk of exhausting the kernel RAM space.

We found that IBM ILOG-CP solver takes the longest time to find the feasible solutions as can be seen in Table 1. However, its space cost is very small and it has been neglected. Therefore, no risk to exhaust the kernel RAM space.

8 Conclusions

In this paper, we proposed the integration of both scheduler and dispatcher into the same RTOS kernel. To this end, we have used a parallel architecture with two PEs to schedule and execute users tasks. The scheduling is performed on the first PE, called SPE. User tasks are executed on the second PE, called MPE. The SPE executes both online and offline algorithms. This means it runs on two different modes namely, scheduling and dispatching modes. The scheduling mode is executed first at selection instants using an online algorithm. The offline algorithm is executed in parallel to users tasks. Once the offline algorithm finds an optimal/near-optimal schedule, the SPE switches to dispatching mode. We have also described a new task states that fits the proposed design with possible communications between both PEs.

The new design is also subject to many possible improvements. Multiprocessor systems, sporadic and aperiodic tasks, preemptive context with sharing resources and data dependencies are among the important challenges in real-time systems area which should be studied extensively according to the proposed design. We strongly believe that this new design is very promising to help solving those challenges.

Acknowledgements

The author would like to thank anonymous reviewers for their valuable comments to improve the presentation of this work.

References

Abdelzaher, T.F. and Shin, K.G. (1999) 'Combined task and message scheduling in distributed real-time systems', *IEEE Transaction on Parallel and Distributed Systems*, Vol. 10, No. 11, pp.1179–1191.

Burleson, W. et al. (1999) 'The spring scheduling coprocessor: a scheduling accelerator', *IEEE Transactions VLSI System*, Vol. 7, No. 1, pp.38–47.

Cavalcante, S.V. (1997) *A Hardware-Software Co-Design System for Embedded Real-Time Applications*, PhD thesis, University of Newcastle upon Tone.

Choquet-Geniet, A. and Grolleau, E. (2004) 'Minimal schedulability interval for real-time systems of periodic tasks with offsets', *Theoretical Computer Science*, Vol. 310, Nos. 1–3, pp.117–134.

Garey, M.R. and Johnson, D.S. (1979) *Computers and Intractability. A Guide to the Theory of NP-Completeness*, Freeman, New York, USA.

Isovic, D. and Fohler, G. (2009) 'Handling mixed sets of tasks in combined offline and online scheduled real-time systems', *Real-Time Systems*, Vol. 43, No. 3, pp.296–325.

Jeffay, K., Stanat, D.F. and Martel, C.U. (1991) 'On non-preemptive scheduling of periodic and sporadic tasks', *Proceedings of the 12th IEEE Symposium on Real-Time Systems*, December, pp.129–139.

Laborie, P. and Rogerie, J. (2008) 'Reasoning with conditional time-intervals', *21st International FLAIRS Conference*.

Laborie, P. and Rogerie, J. (2009) 'Reasoning with conditional time-intervals', *22nd International FLAIRS Conference*.

Minton, S. et al. (1992) 'Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems', *Artificial Intelligence*, Vol. 58, Nos. 1–3, pp.161–205.

Rossi, F., Van Beek, P. and Walsh, T. (2006) *Handbook of Constraint Programming*, Elsevier Publisher, ISBN: 1574-6525.

Salcic, Z. et al. (2006) 'The spring scheduling coprocessor: a scheduling accelerator', *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp.368–372.

Shepard, T. and Gagne, M. (1991) 'A pre-run-time scheduling algorithm for hard real-time systems', *IEEE Transaction on Software Engineering*, Vol. 17, No. 7, pp.669–677.

Starner, J. et al. (1996) 'Real-time scheduling co-processor in hardware for single and multiprocessor systems', *EUROMICRO Conference*, pp.509–512.

Tanenbaum, A. and Woodhull, A. (2006) *Operating Systems Design and Implementation*, Prentice Hall, ISBN-10: 0131429388.

Van Beek, P. (2006) 'Backtracking techniques for constraint satisfaction problems', in Rossi, F., Van Beek, P. and Walsh, T. (Eds.): *Handbook of Constraint Programming*, Chapter 4, pp.85–118, Elsevier Publisher, Netherlands.

Wang, W., Mok, A.K. and Fohler, G. (2003) 'Pre-scheduling: integrating offline and online scheduling techniques', *Lecture Notes in Computer Science*, Vol. 2855, pp.356–372.

Xu, J. (1993) 'Multiprocessor scheduling of processes with release times, deadlines, precedence and exclusion relations', *IEEE Transaction on Software Engineering*, Vol. 19, No. 2, pp.139–154.

Xu, J. and Parnas, D. (1992) 'Pre-run-time scheduling of processes with exclusion relations on nested or overlapping critical sections', *11th IEEE International Phoenix Conference on Computers and Communications*, Scottsdale, USA, April, pp.774–782, ISBN: 0-7803-0605-8.

Xu, J. and Parnas, D. (1993) 'On satisfying timing constraints in hard-real-time systems', *IEEE Transaction on Software Engineering*, Vol. 19, No. 1, pp.70–84.

Xu, J. and Parnas, D. (2000) 'Priority scheduling versus pre-run-time scheduling', *The International Journal of Time-Critical Computing Systems*, Vol. 18, No. 1, pp.7–23.

Yacine, L. and Nizar, B. (2014) 'Pre-run-time scheduling in real-time systems: current researches and artificial intelligence perspectives', *Expert Systems with Applications*, Vol. 41, No. 5, pp.2196–2210.

Notes

1 Notice that this value is a proposed value to show the behaviour of the global design. A more realistic value according to a specific hardware could be chosen.

2 Recall that our aim is this paper is not to assess the efficiency of both algorithms.

Appendix

IBM ILOG-CP constraint program

Table A1 IBM ILOG-CP constraint program: single processor scheduling of jobs under timing constraints

```
using CP;
int lcm = ...;
range ilcm = 0...lcm;
int n = ...;
range T = 1...n;
int r[T] = ...;
int c[T] = ...;
int d[T] = ...;
dvar int start[T] in ilcm;
dvar int latness;
dvar interval x [a in T] in r[a]...d[a] size c[a];
maximize lateness == min(a in T) (d[a]-startOf(x[a])-c[a]);
subject to {
   forall(a in T){
            ct1: start[a] == startOf(x[a]);
            ct2: start[a] ≥ r[a];
            ct3: start[a] + c[a] == endOf(x[a]);
            ct4: lateness ≥0; // a decision problem
}
noOverlap(all(a in T) x[a]);
}
```