Journal of Systems Architecture 118 (2021) 102223

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Hardware resources contention-aware scheduling of hard real-time multiprocessor systems☆

José María Aceituno, Ana Guasque *, Patricia Balbastre, José Simó, Alfons Crespo

Universitat Politècnica de València, Camino de Vera s/n, Valencia, 46022, Valencia, Spain

ARTICLE INFO

Keywords:
Multicore
Real-time
Scheduling
Memory contention
MILP

ABSTRACT

In hard real-time embedded systems, switching to multicores is a step that most application domains delay as much as possible. This is mainly due to the number of sources of indeterminism, which mainly involve shared hardware resources, such as buses, caches, and memories. In this paper, a new task model that considers the interference that task execution causes in other tasks running on other cores due to memory contention is proposed. We propose a scheduling algorithm that calculates the exact interference. We also analyse and compare existing partitioning algorithms and propose three strategies to allocate tasks to cores to schedule as many tasks as possible and minimise total interference.

1. Introduction

The release of the first dual-core processor in 2001 began a migration of computing platforms from single-cores to multicore architectures. In response to the increased use of multicore processors, the Certification Authorities Software Team (CAST) published Position Paper CAST-32 A named 'Multi-core Processors' [1]. This paper identifies topics that could impact the safety, performance and integrity of airborne software systems executing on multicore processors and provides objectives intended to guide the production of safe multicore avionic systems. For example, objective MCP_Software_1 requires that evidence is produced to demonstrate that all hosted software components function correctly and have sufficient time to complete their execution when operating in their multicore environment. In many domains such as avionics, space, or industrial control systems, hard real-time constraints, safety and security issues, and certification assurance levels are commonly required.

Hard real-time multiprocessor systems are commonly implemented using partitioned scheduling rather than global scheduling. In the partitioned approach, the tasks are statically partitioned among the processors, i.e., each task is assigned to a processor and is always executed on it. Static assignment is key to meeting the temporal requirements needed by certification authorities. Nevertheless, multiprocessor systems add many sources of indeterminism. Processors may contend for shared resources, such as memory. During the lifetime of a piece of software several processors may reach a part of the programs that lead to heavy bus loads and each access that the processors attempt may collide with accesses generated by another processor. It is necessary to analyse software images in the context of a multi-core system and not just analyse the software when it is running without contenders [2].

The timing behaviour of a task in a multicore system is affected not only by the software running on it and its inputs, but also by contention for resources such as buses, caches, and GPUs that are shared with tasks running on other cores. This contention causes interference in the timing behaviour of the task [3]. We can define the interference as a delay on the expected execution time of a set of tasks in a multicore system due to the contention produced for simultaneous accesses to the shared hardware.

Different techniques can be implemented to deal with these sources of indeterminism. The worst-case interference can be estimated and added to the worst-case execution time (WCET) of the task. This results in an over-estimated and very pessimistic model. Moreover, this value depends on the execution of other cores and so it is difficult to find a worst case. We can apply techniques to reduce or enhance the predictability of the interference due to memory contention. Our work is in the middle: we consider worst-case interference, but we do not assume that it is produced for every task. We propose a scheduling algorithm that counts the exact interference while assuming that the interference a task produces for other tasks due to contention is bounded. Based on the implementation of the scheduling algorithm we deduce a way to allocate tasks to cores so that we group tasks into cores to produce as little interference as possible.

☆ Funding: This work was supported by the Spanish Science and Innovation Ministry (predictable and dependable computer systems for Industry 4.0) under Grant MICINN: CICYT project PRECON-I4 and Grant TIN2017-86520-C3-1-R.

* Corresponding author.
E-mail address: anguaor@ai2.upv.es (A. Guasque).

https://doi.org/10.1016/j.sysarc.2021.102223
Received 19 February 2021; Received in revised form 15 June 2021; Accepted 20 June 2021
Available online 26 June 2021

It is important to note that the measurement about the time that a task spends on accessing shared hardware resources (i.e., reading and writing memory operations) is outside the scope of this paper. Considerable research has been done in this area. We assume that this is a value previously measured for the used hardware, and so we do not restrict our work to a specific type of hardware resource contention.

We centre our efforts on dealing with the limitations and requirements of the software, and it is important to note that, according to [4], the configuration of the software architecture can reduce the interference delay, and this means that indeterminism will be reduced as the number interruptions is reduced. On previous referenced work, interference is reduced with temporal and spatial isolation. But in our work, we centre on allocation and scheduling tasks in cores.

In summary, the main contributions of this paper are the definition of a task model that includes the interference due to contention for shared hardware resources, allocation strategies to reduce this interference, and a scheduling algorithm for the proposed model.

The rest of the paper is organised as follows: Section 2 presents the relevant works in partitioned multicore systems scheduling. In Section 3, the system model is described and the main contributions are highlighted. Section 4 describes the contention-aware scheduling algorithm, while in Section 5 three allocation algorithms are proposed. The evaluation of the proposals is presented in Section 6, while conclusions and further work are given in Section 7.

2. Related works

Scheduling for multicore platforms was the subject of many research works, surveyed in [5]. As we know, in multi-core scheduling there are two main branches, partitioned scheduling, and global scheduling. In this work we are focusing on partitioned scheduling. To generate a schedule plan in a multicore partitioned system, the following steps are defined:

- Allocation of tasks to cores.
- Perform the schedule generation for each core.

Since our work is framed around hard real-time systems, we will focus on the state of the art of partitioned allocation (where) and static scheduling (when).

The task allocation problem is analogous to the bin packing problem and is known to be NP-hard in the strong sense [6].

A number of heuristics are available for solving the bin packing problem. Some of the most well-known [7,8] are:

- First fit (FF). Each item is allocated into the first bin that it fits into without exceeding the maximum capacity of the bin. If there is no bin available, a new bin is opened.
- Best fit (BF). This algorithm allocates each item into the fullest bin where it fits, and as with FF, possibly opening a new bin if the item does not fit into any currently open bin.
- Worst fit (WF). WF allocates each item into the bin that leaves most remaining capacity, that is, the emptiest bin. It will also open a new bin if no bin is available to allocate the item.

In addition to these heuristics, there are other bin packing algorithms used to solve the allocation problem. Coffman et al. [9] presents a survey and classification of these algorithms. In [10] a state-of-the-art about contention delays is presented.

Previous algorithms are sensitive to the order of the items. For example, if lightweight items are allocated first, accommodating large items in the gaps they leave is a difficult task. There are several methods to order the items before allocating them into cores. The most used technique consists of ordering the items according to their weight, i.e., utilisation and decreasing utilisation is one of the main variants. The decreasing utilisation method (DU) puts the items in decreasing order by utilisation. In this way, WF, BF, and FF become WFDU (worst fit decreasing utilisation), BFDU (best fit decreasing utilisation) and FFDU (first fit decreasing utilisation).

All the previous results are highly theoretical and do not consider delays due to hardware resource contention. In [11] a deep study of the sources of unpredictability is analysed under two categories: primary sources (caches, FSB, memory, and memory controller); and secondary sources (hardware-prefetching, power saving strategies, translation look-aside buffer, misses, system management interrupts). This topic is also analysed in [12] where the sources of timing interference in single-core, multicore, and distributed systems are presented. As stated in this paper, memory interference can jeopardise system feasibility. It is shown in [13] that there are cases where memory interference can cause a worst case response time increase as high as 300%, even for tasks that spend only 10% of their time fetching memory in an eight-core system.

There are some works that try to reduce contention delays by using specific task models. The predictable execution model (PREM) is introduced in [14] which splits a task into a read communication phase and an execute phase. A similar technique is used in [15] that calculates task scheduling and contentions with the objective of minimising the schedule makespan by letting the technique decide when it is necessary to avoid or consider interference. The shared bus is arbitrated using a round-robin policy and the task model considers a DAG (direct acyclic graph) to separately read, execute, and write operations so the WCET of the execute phase can be measured in isolation. We do not split tasks and so our model is a classical periodic task model. For DAG task models, [16] proposes a scheduling method that applies the LET (Logical Execution Time) paradigm and considers communication timing between nodes to reduce contention.

Other approaches, such as the one presented in [17], try to reduce interference costs using synchronisation-based interference models and appropriate memory allocation schemes.

In [18], a feedback control scheme is proposed to ensure the execution of critical cores in a mixed-criticality partitioned system. The controller limits at hypervisor level the use of the memory bus of non-critical cores when they reach a limit. Performance monitor counters are used to establish the number of bus accesses. In [19], Casini et al. propose an analysis of memory contention where an optimisation problem is formulated to bound the memory interference by leveraging a three-phase execution model and holistically considering multiple memory transactions issued during each phase.

One of the first papers that introduced the interference parameter in the temporal model is presented in [4]. In this paper, for a partitioned system in the aerospace domain, the WCRA parameter is defined (Worst Case number of shared Resource Accesses). This value is added to the WCET and this results in a predictable but pessimistic scheduling plan. Similarly, the concept of interference-sensitive Worst-Case Execution Time (isWCET) is proposed in [20]. In [21] a dynamic approach is presented that safely adapts isWCET schedules during execution by relaxing or completely removing isWCET schedule dependencies (depending on the progress of each core). The concept of isWCET is similar to our work but the proposals are centred around minimising the effect of interference with new scheduling methods while our work focuses on allocation algorithms.

The work lines of [22] and [23] deserve mention. In these approaches, interference is analysed in a similar way to our work, as it is also represented as a parameter. However, their work is based on the interference produced by the DRAM memory as a shared resource in a multicore system while our proposal is agnostic with respect to the hardware resource used. Their work also considers only fixed priority scheduling while our proposal can be used with fixed and dynamic priorities. In [23] memory interference is reduced by partitioning DRAM banks and by a BestFit-based allocation algorithm. Clearly, a model that considers a specific hardware resource and a specific scheduling algorithm will further reduce contention delays, but our aim is to provide a more general model that can be used for any shared hardware.

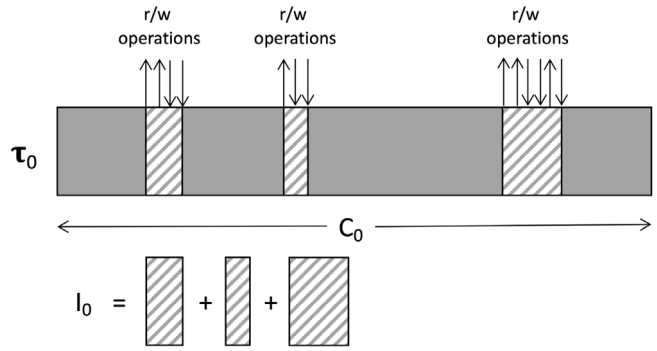


Fig. 1. Example of task interference.

3. Task model and contributions

3.1. Periodic task model

In a multicore system, there is a set of m homogeneous cores $M_1, \ldots, M_m$ that enable different tasks to be executed at the same time. Each core has allocated N_k tasks. We assume that migration is not allowed so we will follow the partitioned approach. In a hard real-time system, there is a set of n independent real-time tasks $\tau = [\tau_1, \ldots, \tau_n]$, where each task generates a set of jobs ($\tau_{ij}, 1 \leq i \leq n, j \geq 0$) that must be completed before the due time. This work considers that all tasks are periodic and characterised by $\tau_i = (C_i, D_i, T_i, I_i)$, where C_i is the WCET, D_i is the deadline, T_i is the period and I_i is the worst-case interference time (explained in detail in 3.2). We assume a constrained deadline task model ($D_i \leq T_i$) and periodic or sporadic tasks.

The hyperperiod of the task set, H, is the smallest interval of time after which the periodic patterns of all the tasks are repeated, and it is calculated as the least common multiple of the task periods. Any task τ_i has A_i activations throughout H. Therefore, $A_i = \frac{H}{T_i}$.

The utilisation of a task τ_i is calculated as the relation between the computation time and the period, $U_i = \frac{C_i}{T_i}$. The utilisation of a core M_k is the sum of the utilisation of all tasks that belong to this core: $U_{M_k} = \sum_{\tau_i \in M_k} U_i$. The total utilisation of the system is the sum of the utilisation of all cores: $U_\tau = \sum_m U_{M_k}$.

3.2. Worst case interference time

In multicore systems, unlike monocore, the resources of the system are shared by different tasks at the same time and different cores may simultaneously need a particular resource, such as buses or memory because of the nature of the process they are executing. At this point, we can assert that during an execution of a multicore system there will probably be a contention or interference between different cores that will delay the expected execution of the processes. As remarked in [10] from contention to access to shared resources arises the interference effect between various tasks that are allocated and executed on different processors.

I_i is the worst-case time that τ_i uses for reading and writing memory operations. It is illustrated in Fig. 1, where we can see that all the time that τ_0 spends on reading and writing operations is part of the interference parameter I_0. Task execution times are depicted in solid rectangles while interference is depicted in dashed rectangles. From the point of view of other tasks, I_i is the extra time that τ_i produces in other tasks executing at the same time on all other cores due to contention. For example, let us suppose that $\tau_i = \{10, 50, 75, 3\}$, this would mean that the computation time of τ_i is 10 time units, of which 3 are dedicated to memory accesses. In the worst case, we can assume that all the tasks running on other cores at the same time as τ_i will be delayed 3 units in their execution.

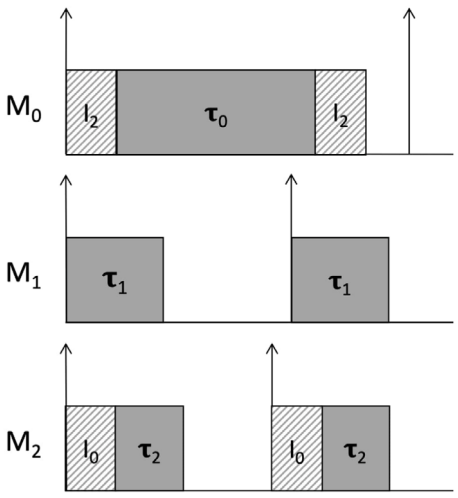


Fig. 2. Example of influence of interference on scheduling.

Fig. 1 shows the interference parameter from the point of view of the task, and this interference implies a delay in the execution of tasks in other cores. Note that I_0 is represented as a whole piece when it affects other tasks while in Fig. 1 it is represented as separate pieces. From now on, the interference will always be represented as a whole piece since we will investigate the effect it has on other tasks in other cores. The following example illustrates this effect.

Example. Consider a set of three tasks τ_0, τ_1, and τ_2 allocated on a platform with three cores as shown in Fig. 2. Suppose that $I_0 = I_2 = 1$ and $I_1 = 0$, that is, τ_0 and τ_2 are the tasks that share the resources, so they provoke and receive interference. However, τ_1 does not use the resources and so suffers no interference. Every time that τ_0 is executed at the same time as τ_2, both tasks increase their computation times: τ_0 increases I_2 units due to the interference caused by τ_2 while τ_2 increases I_0 units due to the interference caused by τ_0.

From Fig. 2, it is deduced that if contention is considered, the total utilisation of a task depends on its computation time and period, as well as the interference received from other tasks. Moreover, this interference does not have to be considered in all activations, and only in those where there is execution on both cores. Therefore, we define the equivalent real values for U_i, U_{M_k} and U_τ:

$$U_i' = U_i + U_i^{int} \tag{1}$$

being U_i^{int} the utilisation due to the interference caused by other tasks to τ_i.

Expressing U_i' with respect to the hyperperiod, we obtain:

$$U_i' = \frac{A_i C_i}{H} + \frac{I_i^T}{H} \tag{2}$$

being I_i^T the total interference that τ_i receives due to the execution of contenders in other cores.

In the same way, the real utilisation of a core M_k is:

$$U'_{M_k} = \sum_{\tau_i \in M_k} U_i' \tag{3}$$

And for the total real utilisation of task set τ:

$$U'_\tau = \sum_m U'_{M_k} \tag{4}$$

3.3. Contributions

As far as the authors are aware there is no scheduling algorithm that schedules a system like the algorithm presented in this paper where the interference is calculated at its exact value.

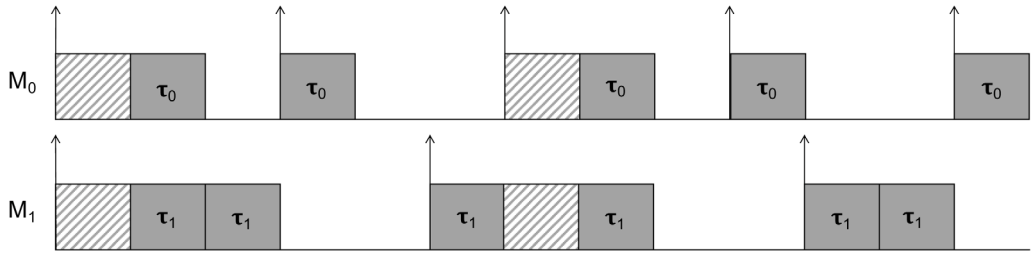


Fig. 3. Execution chronogram of the example.

With this model, the interference plays a key role in determining the schedulability of multicore systems. In addition, we will use the knowledge of interference to propose new allocation algorithms and compare them with existing ones. Therefore, the main contributions of this work are:

- Definition of a task model that considers interference delays due to contention of shared hardware resources.
- Proposal of a new scheduling algorithm that copes with the new task model. This scheduling algorithm is formulated to be included in any priority-based scheduling algorithm for monocores.
- Proposal of three allocation algorithms and comparison with existing ones in terms of schedulability and real utilisation of the system.

4. Contention aware scheduling algorithm

This section describes the rules that a scheduling algorithm must follow to consider the exact interference that a task suffers in each activation. It is important to note that this scheduler does not try to reduce interference. We will do this in the allocation of tasks to cores in Section 5.

We initially need some definitions:

Definition 1. A task is defined as a receiving task when it accesses shared hardware resources and suffers an increase in its computation time due to the interference produced by other tasks allocated in other cores.

Definition 2. A task is defined as a broadcasting task when it accesses shared hardware resources and provokes an increase in computation time in other tasks allocated in other cores due to contention.

If $I_i = 0$, τ_i is neither a broadcasting nor a receiving task. If $I_i > 0$, τ_i will be a broadcasting and receiving task if there is at least one task τ_j in other cores whose $I_j > 0$.

Interference is produced whenever two broadcasting tasks run at the same time in different cores. The instant in which an interference may occur is when one of the two following situations occurs:

- A receiving task τ_i is released. In this moment, active broadcasting tasks in other cores cause interference for τ_i.
- A broadcasting task τ_j is released. In this moment, τ_j causes interference for active receiving tasks in other cores.

Moreover, as a task can receive interference from more than one task (if there are more than two cores), and in different instants of time, it will be necessary to record the interference produced by each task in a matrix.

Definition 3. Let W be a binary matrix of $n \times n \times H$. At each instant t the value of W_{ijt} indicates whether τ_i provokes interference for τ_j or not, in the following way:

- $W_{ijt} = 1$: there is interference.
- $W_{ijt} = 0$: there is not interference.

This matrix will be used to establish when a task τ_i must increase its computation time because of the interference caused by other tasks running in different cores.

Property 1. *If two tasks τ_i and τ_j are allocated to the same core M_k then $W_{ijt} = 0$ and $W_{jit} = 0$ for all $t = 0, \ldots, H$.*

Property 2. *If a task τ_i has $I_i = 0$ then $W_{ijt} = 0$ and $W_{jit} = 0$ for all $t = 0, \ldots, H$ and for all $j = 1, \ldots, n$.*

We are going to illustrate the behaviour of W with an example. Let us consider the following task set, $\tau = [\tau_0, \tau_1]$ with $\tau_0 = (1, 3, 3, 1)$ and $\tau_1 = (2, 5, 5, 1)$, allocated in a dual-core system, τ_0 is allocated in M_0 and τ_1, in M_1.

Fig. 3 shows the resulting rate monotonic [24] scheduling plan of the system that also considers interference due to memory contention. Rate monotonic is a static fixed-priority scheduling algorithm in which the priority of a task is inversely proportional to its period. As can be seen in the figure, there is an increase in the execution times of τ_0 and τ_1 by I_1 and I_0 whenever other tasks are released. If τ_0 or τ_1 releases when the other task is not active, interference is not added.

The values of W for each time instant t are depicted in Fig. 4. As is pointed in the comments, whenever there is a release of one of the tasks while the other is still active, the corresponding value of the matrix changes from 0 to 1. When a task τ_i finishes its activation, all the elements of W in the ith file are 0. At each instant t, if W_{ijt} is 1, this means that τ_i is active at instant t and has caused an interference for τ_j.

Looking at Fig. 3, we see that the total interference suffered by τ_0 is 2, while the interference suffered by τ_1 throughout H, is also 2. Therefore, the real utilisation of each core is:

$$U'_{M_0} = \frac{C_0}{T_0} + \frac{I_0^T}{H} = \frac{1}{3} + \frac{2}{15} = 0.46$$

$$U'_{M_1} = \frac{C_1}{T_1} + \frac{I_1^T}{H} = \frac{2}{5} + \frac{2}{15} = 0.53$$

As a rule, from the study of W we can see that an interference is caused at time t when W_{ijt} changes from 0 to 1, that is, when $W_{ijt} - W_{ij(t-1)} = 1$ for all tasks τ_j not allocated in the same core as τ_i.

In this way, besides accounting for the total interference received I_i^T we can define and calculate this value for each activation, that is, the contention-aware execution time:

Theorem 1. *The contention-aware execution time C'_{is} of τ_i in activation s is the sum of C_i plus the interferences caused by running tasks in other cores and can be calculated as:*

$$C'_{is} = C_i + \sum_{\tau_j \notin M_k} \left(W_{jisT_i} + \sum_{t=sT_i+1}^{t=(s+1)T_i-1} \max\left(W_{jit+1} - W_{jit}, 0\right) \right) \cdot I_j \qquad (5)$$

Proof. In the interval of the s-th activation, $(sT_i, (s+1)T_i - 1)$ the number of interferences caused by a broadcasting task τ_j is equal to the

t	W	Comment
0	$\begin{pmatrix}0&1\\1&0\end{pmatrix}$	τ_0 and τ_1 release
1	$\begin{pmatrix}0&1\\1&0\end{pmatrix}$	
2	$\begin{pmatrix}0&0\\1&0\end{pmatrix}$	τ_0 finishes its first activation
3	$\begin{pmatrix}0&0\\0&0\end{pmatrix}$	τ_1 finishes its first activation
4	$\begin{pmatrix}0&0\\0&0\end{pmatrix}$	
5	$\begin{pmatrix}0&0\\0&0\end{pmatrix}$	τ_1 releases, but τ_0 is not active
6	$\begin{pmatrix}0&1\\1&0\end{pmatrix}$	τ_0 releases while τ_1 is active
7	$\begin{pmatrix}0&1\\1&0\end{pmatrix}$	
8	$\begin{pmatrix}0&0\\0&0\end{pmatrix}$	τ_0 and τ_1 finish
9->15	$\begin{pmatrix}0&0\\0&0\end{pmatrix}$	τ_0 and τ_1 are not active at the same time

Fig. 4. W values for the example.

number of times that W_{ji} changes from 0 to 1 from instant t to $t+1$, multiplied by I_j. If the change is the other way round, that is, from 1 to 0, then there is no interference. In this way, we must only consider low-to-high transitions of the W matrix. This is expressed in the term $\max\left(W_{jit+1} - W_{jit}, 0\right)$. The term W_{jisT_i} accounts for the interference at the initial instant in which τ_i releases. □

The relation between I_i^T and C'_{is} is then:

$$I_i^T = \sum_s (C'_{is} - C_i) \quad \forall s = 0, \ldots, A_i - 1 \tag{6}$$

The interference matrix W is used to establish the exact computation time that a task should execute at each activation to consider its worst-case execution time C_i and the time added by the contention C'_{is}. This matrix must be calculated at each instant t as part of the online scheduling algorithm.

Listing 1 shows the pseudo-code (Python-like) of a priority (fixed or dynamic) scheduling algorithm, while Listing 2 shows the pseudo-code with the modifications needed to calculate C'_{is} added to the previous algorithm.

Listing 1: Priority based scheduling algorithm for m processors

```
1  #variables definition and initialisation
2  for t in range(H):
3      for k in range(m):
4          runningTask[k] = HigherPriority(M_k)
5      for k in range(m):
6          currentTask[k] = runningTask[k]
7          #account for execution
8          finished = Execute(currentTask[k], k)
9          if finished:
10             C'_i = C_i
```

Listing 2: Contention-aware scheduling algorithm

```
1  #variables definition and initialisation
2  for t in range(H):
3      for k in range(m):
4          runningTask[k] = HigherPriority(M_k)
5      for k in range(m):
6          τ_i = runningTask[k]
7          if τ_i != currentTask[k] && t % T_i == 0:
8              for s in range(m):
9                  if s != k and I_i > 0:
10                     τ_j = runningTask[s]
11                     W[j][i] = 1
12                     C'_i += I_j
13         else:
14             for s in range(m):
15                 if s != k and I_i > 0:
16                     τ_j = runningTask[s]
17                     if W[j][i] == 0:
18                         W[j][i] = 1
19                         C'_i += I_j
20     for k in range(m):
21         currentTask[k] = runningTask[k]
22         #account for execution
23         finished = Execute(currentTask[k], k)
24         if finished:
25             C'_i = C_i
26             for j in range(n):
27                 W[j][i] = 0
28                 W[i][j] = 0
```

The algorithm first selects the task to run on each core according to the selected algorithm. In this case, the task with the highest priority is chosen for each core (line 4). For each core, the condition to add the interference is then evaluated (line 7). If the task selected to run (τ_i) is different from the previous task (context switch) and τ_i has released in time t, then all the running tasks in the rest of the cores change from 0 to 1 the value in i column to indicate that these tasks cause interference for τ_i (line 11). C'_i then increases its value by the interference of the other cores running tasks (line 12). If there is not a context switch or the running task τ_i has resumed from preemption (line 13), it is possible that in the meantime (from preemption to resume) some tasks in other cores have been released, and so their interference must be considered. This can be seen if W_{ij} is 0, which means that some task τ_j has been released in another core while τ_i was preempted. In this case, the interference is added (lines 18 and 19) and recorded in W_{ij}. All the tasks are executed once all the cores have been checked for interference. Those that finish the execution of their activation, set their corresponding row (as a broadcasting task) and column (as a receiving task) in W to 0 (lines 27 and 28). Note that it is not necessary to define W as a three-dimensional matrix (t dimension is removed) as the calculation of $C'_i s$ is done on the fly. The same happens with C'_{is}. This reduces notably the overhead of the algorithm.

After the completion of the scheduling on H, I_i^T is calculated for each task and, so can compute the real utilisation of the system U'_τ.

In terms of computational complexity, this algorithm consists of three main loops. The first one is inherited from Listing 1 and it corresponds with the complexity of RM or EDF algorithm by m cores.

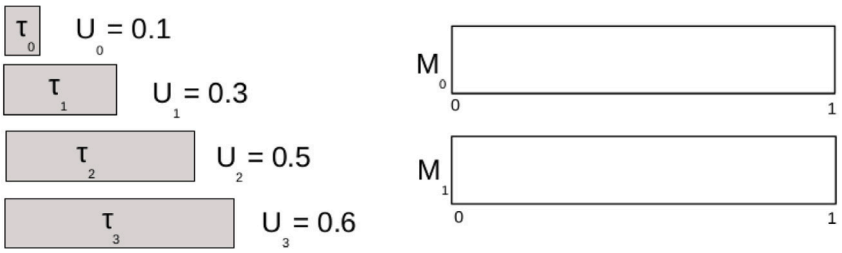


Fig. 5. Task set and available cores in a system before allocation.

The second loop has computational complexity of $O(m^2)$. The third loop has computational complexity of $O(m \cdot n)$. Assuming that there are more tasks than cores, $n > m$ and then, the computational complexity of algorithm of Listing 2 is $O(m \cdot n)$.

5. Task allocation algorithms

Prior to scheduling, it is necessary to allocate tasks to cores. As commented in Section 2, different allocation heuristics for partitioned multiprocessor systems exist. These have the goal of maximising the number of tasks to allocate while assuring feasibility. However, with our model, the real utilisation of the core is no longer U_{M_k} but U'_{M_k} because the real utilisation of tasks U'_i is increased by the interferences.

The total interference received for each task I_i^T (and, equivalently, C'_{is} in each activation) is difficult to estimate since it depends on the scheduling of the tasks allocated to each core. For this reason, it is not obvious to estimate in advance I_i^T or C'_{is} to calculate the real utilisation per core, so we can know before scheduling whether a task allocation is going to be schedulable or not.

The goal of this section is to study W_{ijt} to derive allocation algorithms that try to take into account the possible interference generated and received so that real utilisation (U'_{M_k}) is decreased while ensuring feasibility.

At any instant t, the interference that a task τ_i allocated in core M_k causes all other tasks on other cores to look at their corresponding i row of W in which values are 1 and multiply it by I_i. To reduce the interference caused, we are interested in having as many zeros as possible in the W matrix. The elements of row i that are always zero are:

- when $j = i$
- when for j column, τ_j is allocated to the same core than τ_i
- when $I_i = 0$.

Therefore, a bound for the maximum interference that a core M_k can receive is:

$$maxW_k = \sum_{\substack{\tau_i \in M_k \\ I_i \neq 0}} \sum_{\tau_j \notin M_k} I_j \tag{7}$$

This value cannot be reached at an instant because two tasks on the same core cannot be active at the same time. But this bound can be reached in an interval $[s \cdot T_i, (s+1) \cdot T_i]$.

Finally, assuming that all values of W that can be 1 are indeed 1, we have that the maximum value of the sum of all elements of W is:

$$maxW = \sum_{\forall k} maxW_k = \sum_{k=1}^{m} \sum_{\substack{\tau_i \in M_k \\ I_i \neq 0}} \sum_{\tau_j \notin M_k} I_j \tag{8}$$

This is not a bound of $\sum_j I_j^T$ since this matrix changes as t changes. The total value of the interference can be greater if, for example, two activations of τ_j interfere with one activation of τ_i.

From the previous analysis of the W matrix we can intuitively sense that a task allocation algorithm that unbalances the load between cores, or that minimises Eq. (8), will tend to have less real utilisation than an allocation algorithm that balances the load between cores. However, a task allocation algorithm that balances the load between cores will tend to schedule more task sets that an unbalanced algorithm.

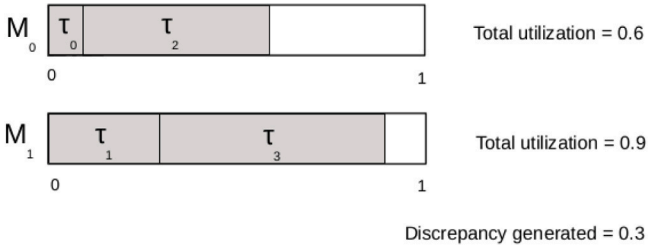


Fig. 6. Allocation of 4 tasks in 2 cores.

An unbalanced load is defined as a load that tries to allocate the maximum number of tasks on the minimum number of cores is low. One approach to exactly measure how balanced a system is with respect to utilisation is by using the so-called utilisation discrepancy, that we define as follows [25]:

Definition 4. The utilisation discrepancy UD_τ can be defined as the difference between the maximum and minimum utilisation of a multicore system.

$$UD_\tau = \max_{k=0}^{m-1}(U_{M_k}) - \min_{k=0}^{m-1}(U_{M_k}) \tag{9}$$

To understand this concept, let us suppose the following task set: τ_0, τ_1, τ_2 and τ_3 with their corresponding utilisations: U_0, U_1, U_2, U_3, and two cores, M_0 and M_1. We do not provide all the temporal parameters because they are not relevant for this example. Fig. 5 represents tasks and cores. Grey boxes are dimensioned depending on the task utilisation while cores are dimensioned with white boxes with a total length of one (maximum utilisation that can be reached by the core).

We then make the allocation process and assign every task a core. We have different options to make the allocation, for example, we could assign τ_0 and τ_2 to M_0 and τ_1 and τ_3 to M_1. In this case, $U_{M_0} = 0.6$ and $U_{M_1} = 0.9$. Therefore, according to the definition of discrepancy, in this allocation case, the discrepancy would be 0.3. Fig. 6 shows the resulting allocation.

When tasks are allocated to different cores, the discrepancy would be maximised if there is no other combination that produces a greater discrepancy than this one. And in the opposite way, the discrepancy would be minimised if the task allocation produces the minimum possible discrepancy value. In this case, we can say that the cores have a balanced utilisation task load, and an unbalanced load on the contrary.

For the reasons explained above, we can derive new allocation algorithms for our model. We will propose three new allocation algorithms:

- Maximise utilisation discrepancy, UD_τ. From now on, this algorithm is called UDmax.
- Minimise utilisation discrepancy, UD_τ. From now on, this algorithm is called UDmin.
- Minimise Eq. (8). From now on, this algorithm is called Wmin.

Table 1
Model notation.

SETS AND INDICES	
i	Tasks $\tau_i \in \{0, 1, 2, \dots, n-1\}$
k	Cores $M_k \in \{0, 1, 2, \dots, m-1\}$
PARAMETERS	
C_i	Worst case execution time of τ_i
T_i	Period of τ_i
U_i	Theoretical utilisation of τ_i
I_i	Interference factor of τ_i over other tasks
DECISION VARIABLES	
UDmin and UDmax models	
O_{ik}	Allocation matrix. 1 if τ_i allocated in M_k, 0 otherwise
U_{M_k}	Theoretical utilisation of core k
UD_τ	Utilisation discrepancy of the task set τ
Wmin model	
O_{ik}	Allocation matrix. 1 if τ_i is allocated in core k and 0 otherwise
U_{M_k}	Theoretical utilisation of core k
$maxW\,k$	Maximum value of the sum of all elements of W for core k
$maxW$	Maximum value of the sum of all elements of W for all cores

Below, we will describe the three allocation algorithms. We will check if the previous observations hold in Section 6. The allocators will be implemented as an integer programming formulation.

5.1. UDmin And UDmax allocators

Because of their similarities, we will describe UDmin and UDmax together. For that, we define a set of parameters and variables shown in Table 1, that also defines the parameters and variables for Wmin.

Both allocators have the same constraints, but the objective is to minimise or maximise the discrepancy.

We have that for UDmin the objective is:

$$\text{Minimise} \quad UD_\tau \tag{10}$$

and for UDmax the objective is:

$$\text{Maximise} \quad UD_\tau \tag{11}$$

s.t:

$$\sum_{\forall k} O_{ik} = 1 \qquad \forall i \tag{12}$$

$$\sum_{i \in k} U_i \cdot O_{ik} = U_{M_k} \qquad \forall k \tag{13}$$

$$U_{M_k} \leq 1 \qquad \forall k \tag{14}$$

$$UD_\tau = \max_{k=0}^{m-1}(U_{M_k}) - \min_{k=0}^{m-1}(U_{M_k}) \tag{15}$$

$$O_{ik} \in \{0, 1\} \tag{16}$$

$$U_{M_k} \geq 0 \tag{17}$$

The model constraints are defined in Eqs. (12), (13), (14) and (15). In constraint (12), we ensure that a task is allocated in one and only one core. The total utilisation per core is calculated as the sum of the task utilisations that belong to that core (Eq. (13)) and is less than or equal to 1 (Eq. (14)). In constraint (15) we calculate the utilisation discrepancy as the difference between the biggest and lowest task utilisation from all cores. Consequently, the objective function is to minimise the discrepancy in the case of UDmin and maximise the discrepancy in the case of UDmax. Finally, Eqs. (16) and (17) represent the decision variable domains.

Constraint (15) is not linear but is easily expressed with this formulation that avoids a large set of linear and special-ordered set constraints, plus a number of auxiliary decision variables. It is directly supported by the solver API by performing the transformation to a corresponding Mixed Integer Programming formulation automatically and transparently during the solution process. As this constraint is not linear and the model has binary and integer variables, it is a Mixed Integer Non-Linear Programming (MINLP) problem.

5.2. Wmin allocator

To formulate the Wmin allocator through integer programming, let us also use the notation in Table 1. The objective function of this allocator is to minimise Eq. (8).

$$\text{Minimise } maxW = \sum_{\forall k} maxW_k \tag{18}$$

s.t:

$$\sum_{\forall k} O_{ik} = 1 \qquad \forall i \tag{19}$$

$$\sum_{i \in k} U_i \cdot O_{ik} = U_{M_k} \qquad \forall k \tag{20}$$

$$U_{M_k} \leq 1 \qquad \forall k \tag{21}$$

$$\sum_{\substack{\tau_i \in M_k \\ I_i \neq 0}} \sum_{\tau_j \notin M_k} I_j = maxW_k \qquad \forall k \tag{22}$$

$$O_{ik} \in \{0, 1\} \tag{23}$$

$$U_{M_k}, maxW_k \geq 0 \tag{24}$$

Constraints defined in Eqs. (19), (20), and (21) were explained in Section 5.1 and define the capacity of each core.

Eq. (22) calculates the maximum interference provoked by all cores, following Eq. (8) previously defined. Eqs. (23) and (24) represent the decision variable domains.

As the model has binary and integer variables, it is a Mixed Integer Linear Programming (MILP) problem.

6. Evaluation

6.1. Experimental conditions

The simulation scenario developed for this work is depicted in Fig. 7. It is divided into five steps:

- Generation of the load (see Section 6.1.1).
- Allocation (see Section 5).
- Validation of the allocation phase (see Section 6.1.2).
- Scheduling (see Section 6.1.3).
- Validation of the scheduling phase (see Section 6.1.4).

The automatic load generator generates a task set with the process described in Section 6.1.1. This task set is the input for the six allocators: existing FFDU, BFDU, and WFDU (which correspond with the FF, BF and WF heuristics described in Section 2 and in which the items are ordered according to decreasing utilisation DU) and UDmin, UDmax, and Wmin proposed in this work. The result of the allocations is then evaluated to check their feasibility. The feasible allocations and the task set are the input for the scheduler, that generates the six scheduling plans. If they are schedulable, their performance parameters will be stored. This sequence is repeated to complete enough simulations.

We use the Gurobi optimiser 9.0 [26], from Gurobi Optimisation, Inc., which is a powerful optimiser designed from scratch to run in multi-core and with the capability to run in parallel mode. It achieved performance improvements with each version and provides a Python interface. Since version 9.0, Gurobi can solve non-convex quadratic optimisation problems and also general constraints such as those described in Section 5.1.

All allocators described in previous sections are executed on an Intel Core i7 CPU with 16 GB of RAM.

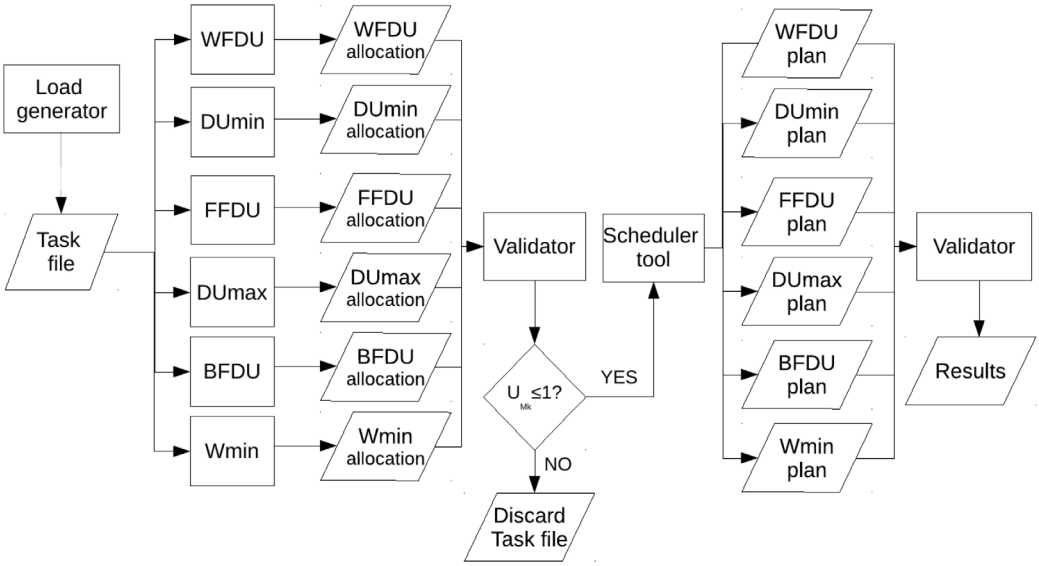


Fig. 7. Experimental evaluation overview.

Table 2
Experimental parameters.

	Experimental parameters			
Number of cores	2	4	8	10
Theoretical utilisation	1	2	4	5
Number of tasks	4	12	20	28
Number of broadcasting tasks	2	3	5	7

6.1.1. Load generator

The load is generated using a synthetic task generator. The number of tasks in each set and the total system utilisation depends on the number of cores in which they are allocated. As these experiments are conducted in 2, 4, 8, and 10 cores, we set a reasonable number of tasks and a feasible load for each number of cores.

Given the system utilisation value and the number of tasks for each set, the utilisation is shared among the tasks using the UUniFast discard algorithm [5]. Periods are generated randomly in [20,1000] and computation times are deduced from the system utilisation. Without loss of generality, deadlines are set to be equal to periods, although they could be constrained to be less than or equal to periods.

Table 2 defines the experimental parameters selected for the evaluation process. The total utilisation for each task set depends on the number of cores in which the set is allocated to and is equal to 50 percent of the maximum load of all cores. For example, the maximum load in a system with four cores is four ($U_\tau \leq m$), i.e., 400%. Therefore, the load of a task set allocated to four cores is set to two. The number of broadcasting tasks is set to 25% of the number of tasks in each set and $I_i = 1\ \forall i$, being τ_i a broadcasting task. In other words, the extra time that τ_i produces for other tasks executing at the same time on all other cores due to contention is equal to one unit of time. As a reminder, if a task is broadcasting, it will interfere with the tasks allocated in other cores (τ_j) if their coefficient $I_j \neq 0$. The more broadcasting tasks, the greater the interference that may be produced. Note that in the case of 2 cores, the percentage of broadcasting tasks is set to 50%. This is due to the fact that, if only 25% is considered, only one task is broadcasting and then $U'_\tau = U_\tau$.

6.1.2. Validation of the allocation phase

The first validation phase consists of checking if all tasks have been allocated to cores and ensuring that the maximum capacity per core is not exceeded i.e. $U_{M_k} \leq 1 \quad \forall k = 0, \ldots, m-1$. If any of the allocators cannot allocate the task set, this task set is discarded and a new one is generated. As we assume implicit deadlines for this evaluation, the previous condition is sufficient. In any event, we do not use it as a condition since U_τ is not the real utilisation of the system and will be increased by the interference in the scheduling phase. Therefore, in our evaluation, constrained deadlines can be used without loss of generality.

6.1.3. Scheduling phase

In this phase, the contention aware scheduling algorithm proposed in Section 6.1.3 is executed independently for all cores for the six allocations obtained in the allocation phase. In this work, the EDF scheduling algorithm [24] is used and, therefore, the task with the highest priority will be the task with the shortest relative deadline. Note that any other priority based algorithm is applicable (by implementing it in line 4 of Listing 2). As an output of this phase, U'_τ is obtained since the algorithm in Listing 2 obtains the exact interference.

6.1.4. Validation of the results

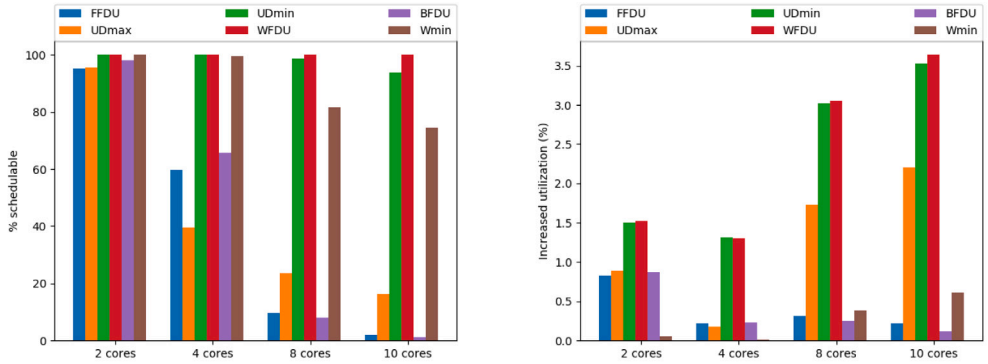
The validation of the scheduling plans involves two steps. Firstly, we must check feasibility to ensure that all deadlines are met in the hyperperiod. Secondly, some performance parameters are obtained to compare different methods. Specifically, we obtain the relation between the theoretical utilisation of the system and the real utilisation of the system for each set, measured after the scheduling phase. Moreover, once the evaluation of all sets has finished, the parameters that must be evaluated are:

- Schedulability ratio. The percentage of task sets with feasible scheduling plans over total task sets with feasible allocations.
- Increased utilisation. The increase in utilisation with respect to the theoretical utilisation. This is measured as $1 - \sum_k \frac{U_{M_k}}{U'_{M_k}} = 1 - \frac{U_\tau}{U'_\tau}$.

Previous parameters are evaluated for a certain number of cores and also for a certain percentage of broadcasting tasks.

6.2. Experimental results

The selection of the initial utilisation of the system and the number of broadcasting tasks does affect the final outcomes. For example, in a set with four tasks allocated to a dual-core system in which the initial utilisation is under but near to two and three of the tasks are broadcasting, it is highly probable that the increase in utilisation due



(a) Percentage of schedulable task sets depending on the number of cores and allocators.

(b) Increased utilisation depending on the number of cores and allocators.

Fig. 8. Experimental values of schedulability and increases in utilisation as a function of the number of cores and allocators.

to interference makes the system infeasible. In this section, the results obtained in the evaluation are commented.

Fig. 8 depicts the results of the experimental evaluation following the parameters in Table 2 for different numbers of cores. Fig. 8(a) shows the percentage of schedulable sets as a function of the number of cores in the system. As seen in the figure, as the number of cores increases, the schedulability ratio decreases. Allocators such as FFDU, BFDU, or UDmax minimise the number of used cores and, therefore, the load per core is high. The increase in load due to interference means that the cores with high utilisation cannot feasibly schedule the tasks and so the system becomes infeasible. Moreover, the greater the number of cores in the system, the more interference is produced. Therefore, unbalanced cores (FFDU, BFDU, or UDmax allocators) become infeasible when the number of cores increases. Likewise, balanced cores with similar loads (UDmin and WFDU allocators) can afford the increase of utilisation due to interference and the percentage of schedulability is almost 100% for all experiments. The Wmin allocator presents very good schedulability ratios, even in those systems with a lot of cores.

Fig. 8(b) represents the increased utilisation of the system. As previously, the more cores in the system, the more interference is produced and, therefore, the more overloaded the cores are in comparison with the initial utilisation. WFDU, UDmin, and Wmin represent this behaviour. However, allocators as FFDU, UDmax, or BFDU differ from the previous allocators, especially in the case of ten cores. This is because they possess very low schedulability ratios (if any) and the increase in utilisation is calculated from a small feasible sample. We observe that, in the case of two cores, the overload is greater than in the case of four cores. This is because in two cores, the percentage of broadcasting tasks is 50% and in other cases it is 25% (see Section 6.1.1). Again, the more broadcasting tasks, the greater the utilisation.

From Fig. 8 we can conclude that there is not an allocator that dominates the others in both schedulability and increased utilisation. The allocators that balance the load, such as UDmin and WFDU, almost always ensure the schedulability of the sets at the expense of the increase in load (about 3.5% in the case of ten cores). Allocators with unbalanced loads, such as FFDU, BFDU, and UDmax, cannot ensure schedulability, especially for a large number of cores, but the increase in utilisation is less than UDmin and WFDU. However, Wmin presents a good ratio of schedulability and a small increase in utilisation in comparison with other allocators.

In all the cases, the percentage of schedulability decreases with the number of cores (and consequently, with the number of produced interferences) and the utilisation increases with the number of cores.

Fig. 9 depicts the schedulability ratio and the increase in utilisation as a function of the allocators. It is calculated as the average of the values represented in Fig. 8. From these figures we can conclude that FFDU, BFDU, or UDmax allocators achieved up to 43% of schedulability with a small increase in system utilisation due to interference. However, WFDU or UDmin achieve almost 100% of schedulability at the expense of a 2.3% increase in utilisation. Previous allocators work in an opposite way: the former achieve an unbalanced load, which reduces the feasibility when interference appears, and the latter ensures feasibility but system utilisation increases. Finally, the Wmin allocator provides a high schedulability ratio (up to 89%) with an increase in utilisation of only 0.266%.

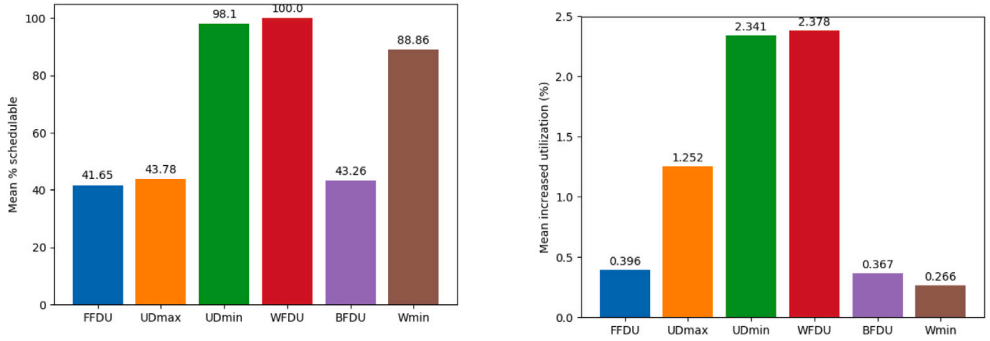
Fig. 10 shows the influence of the number of broadcasting tasks in the schedulability ratio and the increase of utilisation, for each allocator evaluated in this work and following the evaluation parameters in Table 2. From Fig. 10(a) we can conclude that the schedulability ratio generally decreases with the number of broadcasting tasks. Only WFDU and UDmin achieve 100% schedulability for the whole range of broadcasting tasks. FFDU, BFDU, and UDmax are the allocators with the lowest schedulability ratio.

In contrast, from Fig. 10(b) we can deduce that the more broadcasting tasks, the greater is system utilisation. This is a common behaviour for all allocators. WFDU and UDmin are the allocators with the highest increase in utilisation, while FFDU and BFDU reveal the lowest increases. Wmin always presents an intermediate behaviour, i.e., its schedulability decreases depending on the number of broadcasting tasks and utilisation increases up to 0.6% (which are considerable values in comparison with other allocators).

Therefore, it can be deduced that increasing the number of broadcasting tasks produces an increase in system utilisation and a reduction in the schedulability ratio for all allocators except for WFDU, and this ensures schedulability in the studied range of broadcasting tasks. With a major number of broadcasting tasks, WFDU will also reduce its schedulability but we can-not observe this behaviour in the studied range of broadcasting tasks.

Finally, the solution times for the proposed MILP approaches are measured and depicted in Fig. 11 (note that the y-axis is represented on a logarithmic scale and the solution time values are included in the graph for ease of analysis and neatness). For this evaluation, we have conducted experiments with the experimental values described in Table 2.

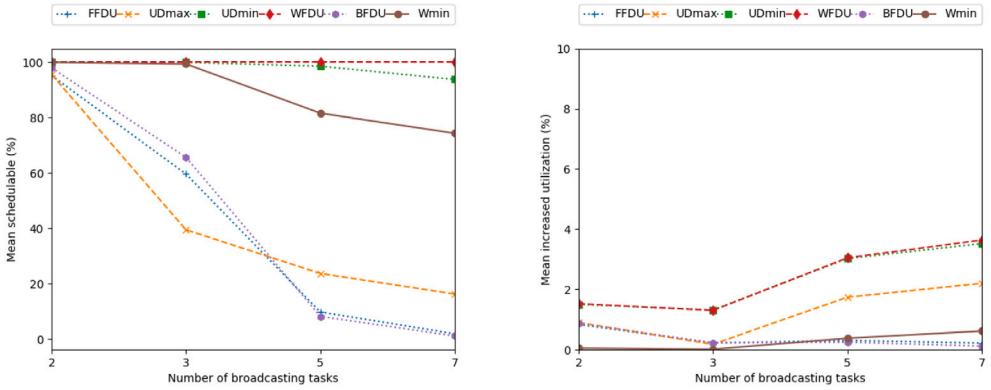
Fig. 11 shows that, as the number of cores increases, the solution time increases. This is because the more cores in the system, the more tasks and broadcasting tasks are considered. UDmin is the approach that takes longest to calculate the solution. This is because the algorithm tries to share the load as much as possible. On the contrary, UDmax unbalances the load, that is, at least one of the cores will remain empty (if possible) and at least one will be full, and so the discrepancy



(a) Percentage of schedulable task sets depending on the allocators.

(b) Increased utilisation depending on the allocators.

Fig. 9. Experimental values of schedulability and increase in utilisation as a function of the allocators.



(a) Percentage of schedulable task sets.

(b) Increased utilisation.

Fig. 10. Experimental values of schedulability and increment of utilisation as a function of the number of broadcasting tasks.

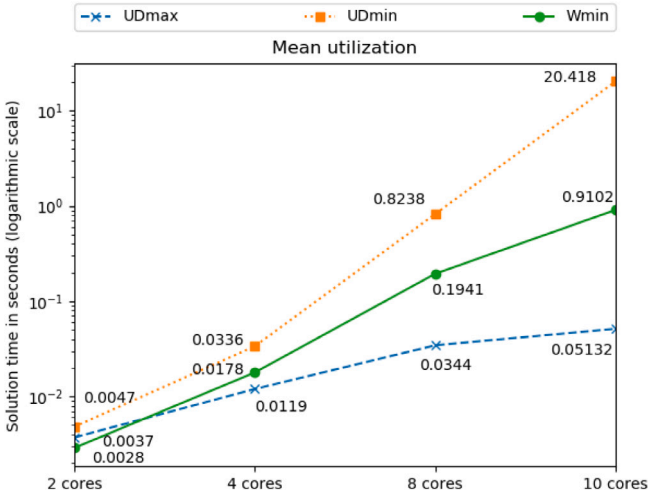


Fig. 11. Solution time for MILP approaches.

is maximised. Obtaining this solution is faster than obtaining a balanced solution. The Wmin allocator is again an intermediate proposal.

We can conclude that if a precise adjustment in terms of discrepancy is required, the UDmax and UDmin approaches are the most suitable solutions. In the remaining cases, their analogue heuristics FFDU, BFDU, and WFDU provide similar and faster solutions. As before, the Wmin approach provides intermediate solution times.

7. Conclusions

This paper has proposed a new task model that considers the delay produced by the contention of hardware shared resources in hard real-time multiprocessor systems. Along with the new model, a scheduling algorithm that considers the exact interference produced for each task is proposed. Until now, the problem was solved by significantly increasing the WCET to cope with the worst case. Our proposal is not so pessimistic without jeopardising feasibility. Moreover, three allocators have been proposed and compared with well-known existing allocators.

According to the experimental evaluation, we can conclude that FFDU, BFDU and UDmax allocator algorithms behave similarly, and UDmin and WFDU allocator algorithms also show similar behaviour. In the case of FFDU, BFDU and UDmax allocators, we can conclude that they reach a low rate of increase in utilisation (which is an advantage), but on the other hand, their rates of schedulability are very low. They may be the best choice for two core architectures.

The opposite case occurs with UDmin and WFDU allocators. They present an excellent rate of schedulability, but they are greatly affected by interference and so if a system needs to prioritise a low utilisation rate then UDmin and WFDU are not the best options. Wmin is clearly an intermediate option because it shows non-extreme rates in increases of utilisation and schedulability. Hence, according to the needs and requirements of a system, Wmin may be a suitable option.

We plan to further investigate the schedulability of task sets with worst-case interference time parameters by deriving an utilisation bound and proposing new scheduling algorithms to decrease interference.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Multi-core Processors—Position Paper CAST-32A, Technical Report, 2016.
[2] Cobham, Multi-core software considerations, 2015, Available at https://www.gaisler.com/doc/antn/GRLIB-AN-0005.pdf (2015/10/28).
[3] Multicore Timing Analysis for DO-178C, https://www.rapitasystems.com/downloads/multicore-timing-analysis-do-178c, Rapita Systems.
[4] J. Galizzi, F. Vigeant, L. Perraud, A. Crespo, M. Masmano, E. Carrascosa, V. Brocal, P. Balbastre, F. Quartier, F. Milhorat, WCET and Multicores with TSP, in: DASIA 2014 DAta Systems in Aerospace, 2014.
[5] R. Davis, A. Burns, Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems, in: 2009 30th IEEE Real-Time Systems Symposium, 2009, pp. 398–409, http://dx.doi.org/10.1109/RTSS.2009.31.
[6] D. Johnson, Near-Optimal Bin Packing Algorithms (Ph.D. thesis), Massachusetts Institute of Technology, Dept. of Math., 1973.
[7] Y. Oh, S.H. Son, Allocating fixed-priority periodic tasks on multiprocessor systems, Real-Time Syst. 9 (3) (1995) 207–239, http://dx.doi.org/10.1007/BF01088806.
[8] E.G. Coffman, M.R. Garey, D.S. Johnson, Approximation algorithms for bin packing: A survey, in: Approximation Algorithms for NP-Hard Problems, PWS Publishing Co., USA, 1996, pp. 46–93.
[9] E.G. Coffman Jr., J. Csirik, G. Galambos, S. Martello, D. Vigo, Bin packing approximation algorithms: Survey and classification, in: P.M. Pardalos, D.-Z. Du, R.L. Graham (Eds.), Handbook of Combinatorial Optimization, Springer New York, New York, NY, 2013, pp. 455–531, http://dx.doi.org/10.1007/978-1-4419-7997-1-35.
[10] G. Fernandez, J. Abella, E. Quiñones, C. Rochange, T. Vardanega, F. Cazorla, Contention in multicore hardware shared resources: Understanding of the state of the art, in: WCET, 2014.
[11] D. Dasari, B. Akesson, V. Nélis, M.A. Awan, S.M. Petters, Identifying the sources of unpredictability in COTS-based multicore systems, in: 2013 8th IEEE International Symposium on Industrial Embedded Systems, SIES, 2013, pp. 39–48, http://dx.doi.org/10.1109/SIES.2013.6601469.
[12] T. Mitra, J. Teich, L. Thiele, Time-critical systems design: A survey, IEEE Des. Test 35 (2) (2018) 8–26, http://dx.doi.org/10.1109/MDAT.2018.2794204.
[13] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, L. Thiele, Worst case delay analysis for memory interference in multicore systems, in: 2010 Design, Automation Test in Europe Conference Exhibition, DATE 2010, 2010, pp. 741–746, http://dx.doi.org/10.1109/DATE.2010.5456952.
[14] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, R. Kegley, A predictable execution model for COTS-based embedded systems, in: 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium, 2011, pp. 269–279, http://dx.doi.org/10.1109/RTAS.2011.33.
[15] B. Rouxel, S. Derrien, I. Puaut, Tightening contention delays while scheduling parallel applications on multi-core architectures, ACM Trans. Embed. Comput. Syst. 16 (5s) (2017) http://dx.doi.org/10.1145/3126496.
[16] S. Igarashi, T. Ishigooka, T. Horiguchi, R. Koike, T. Azumi, Heuristic contention-free scheduling algorithm for multi-core processor using LET model, in: 2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications, DS-RT, 2020, pp. 1–10, http://dx.doi.org/10.1109/DS-RT50469.2020.9213582.
[17] S. Reder, J. Becker, interference-aware memory allocation for real-time multi-core systems, in: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2020, pp.148–159.
[18] A. Crespo, P. Balbastre, J. Simó, J. Coronel, D. Gracia Pérez, P. Bonnot, Hypervisor-based multicore feedback control of mixed-criticality systems, IEEE Access 6 (2018) 50627–50640, http://dx.doi.org/10.1109/ACCESS.2018.2869094.
[19] D. Casini, A. Biondi, G. Nelissen, G. Buttazzo, A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling, in: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2020, pp. 239–252, http://dx.doi.org/10.1109/RTAS48715.2020.000-3.
[20] J. Nowotsch, Interference-sensitive Worst-case Execution Time Analysis for Multi-core Processors (Ph.D. thesis), 2014.
[21] S. Skalistis, A. Kritikakou, Dynamic interference-sensitive run-time adaptation of time-triggered schedules, in: ECRTS 2020 - 32nd Euromicro Conference on Real-Time Systems, 2020, pp. 1–22, http://dx.doi.org/10.4230/LIPIcs.ECRTS.2020.4.
[22] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, R. Rajkumar, Bounding memory interference delay in COTS-based multi-core systems, in: 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium, RTAS, 2014, pp. 145–154, http://dx.doi.org/10.1109/RTAS.2014.6925998.
[23] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, R. Rajkumar, Bounding and reducing memory interference in COTS-based multi-core systems, Real-Time Syst. 52 (3) (2016) 356–395, http://dx.doi.org/10.1007/s11241-016-9248-1.
[24] C.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, J. ACM 20 (1) (1973) 46–61, http://dx.doi.org/10.1145/321738.321743.
[25] A. Crespo, P. Balbastre, J. Simo, P. Albertos, Static scheduling generation for multicore partitioned systems, in: K.J. Kim, N. Joukov (Eds.), Information Science and Applications (ICISA) 2016, Springer Singapore, Singapore, 2016, pp. 511–522.
[26] Gurobi optimizer reference manual, Inc. Gurobi Optimization, 2019.

José María Aceituno was born in Valencia, Spain in 1982. He received the B.S in computers management from the University of Castellon in 2012 and M.S. degrees in Artificial Intelligence in Universitat Politècnica de València, in 2016.

From 2016 to 2019, he was teacher of superior grade formative in development web application and multiplatform in Ilerna Online, Spain. He is currently a Ph.D. candidate in distributed systems at Universitat Politècnica de València.

Ana Guasque was born in Valencia, Spain, in 1987. She received the B.S. degree in industrial engineering from the Universitat Politècnica de València, Spain, in 2013 and the M.S. degree in automation and industrial computing in the same university in 2015. She received the Ph.D. degree in industrial engineering from the same university in 2019.

She is currently working as a researcher in Universitat Politècnica de València. Her main research interests include real-time operating systems, scheduling and optimisation algorithms and real-time control.

Patricia Balbastre is Associate Professor of Computer Engineering at the Universitat Politècnica de València. She graduated in Electronic Engineering in the same university in 1998 and obtained the Ph.D. degree in Computer Science in 2002. Her main research interests include real-time operating systems, dynamic scheduling algorithms and real-time control.

José Simó received the M.S. degree in industrial engineering and the Ph.D. degree in computer science from the Universitat Politècnica de València (UPV), Spain, in 1990 and 1997, respectively. Since 1990, he has been involved in several Spanish and European research projects mainly related to Real-Time Systems and Industrial and Embedded Collaborations.

He is currently a Professor with the Department of Computer Engineering, UPV. His current research is focused on the development of real-time embedded systems, autonomous systems, and robotics.

Alfons Crespo is Professor of the Department of Computer Engineering of the Technical University of Valencia. He received the Ph.D. in Computer Science from the Technical University of Valencia, Spain, in 1984. He held the position of Associate professor in 1986 and full Professor in 1991. He leads the group of Industrial Informatics and has been the responsible of several European and Spanish research projects. His main research interest include different aspects of the real-time systems (scheduling, hardware support, scheduling and control integration,...). He has published more than 60 papers in specialised journals and conferences in the area of real-time systems.