A semi-partitioned model for Mixed Criticality Systems - review

Michele Tagliabue
Student ID 1206966
Real-Time Systems

Abstract—Since Vestal presented his paper in 2007, much research has been done in the Mixed Criticality System domain. Moreover, with the spread of multicore systems also in the real-time context, research has shifted toward an application of the MCS model to the multicore environment. The earliest models specified that, if the system entered HI-crit mode, lower priority tasks would be abandoned. However, Xu and Burns have developed a model that promises, in the face of certain conditions, to be able to not abandon the LO-crit tasks. The cited authors started from a dual-core environment to get to multi-core systems. This paper will go through an overview of their work, giving when and as far as possible also a critical judgment.

1 Introduction

Multi-core is now a standard in real-time systems. The need for computational power is increasing and satisfying it with a single-core system becomes increasingly difficult: to increase CPU power often you need to higher the frequency, which can cause both system instability and lower efficiency: CPU power consumption often rises as the square of the core clock frequency while performance grows less than linearly. These are key aspects in a real-time system, so the world is opting for multicore CPUs. However, this choice introduces some new challenges that need to be addressed in order to reach an efficient and feasible multi-core system. One of these is certainly to determine in which core tasks' jobs execute. Various approaches can be used:

- *Partitioned scheduling*, where tasks are statically allocated to a specific core, without the ability to migrate. For each core, tasks are scheduled using a single-processor scheduling protocol (like EDF), so there is one scheduler for each core. Although a system that uses this approach is easier to analyze (and to compute WCETs), the main problem is its low utilization bound: it's very easy to build an example with a utilization bound smaller than 50% overall;
- *Global scheduling*, where tasks can execute in any core available and they can be preempted and migrated from one core to another and, unlike partitioned scheduling, tasks are managed by a single "global" scheduler. While tasks migration allows putting the cores' idle time to use, it introduces an unpredictable factor to the system, since it might be hard to know in advance which task is executing on an arbitrary core c_i at an arbitrary time T_k, making the feasibility analysis very difficult (if not impossible);
- *Semi-partitioned scheduling*, where tasks are split in two categories: statically allocated tasks and migratable tasks. Task categorization and initial core allocation are decided before the execution of the system.

Moreover, there is a trend to combine components with different level of criticality onto a single hardware platform: the result is a Mixed Criticality System. More formally, a mixed criticality system can be abstracted as a finite set of tasks, where each task $\tau_i = \{L_i, T_i, \pi_i, C_i, D_i\}$ has a level of criticality L_i, a period T_i, a priority π_i, a function $C_i(L_i)$ that gives Worst Case Execution Time (a.k.a. execution time budget) at a criticality level L_i, a relative deadline D_i. Since the WCET value is estimated with different levels of assurance, the following assertion must hold: $L_1 \leq L_2 \Rightarrow C_i(L_1) \leq C_i(L_2)$ for any task τ_i. A MCS has also an attribute, L, which is the current criticality level of the whole system and it's initially set with the lower possible value $L_i | \forall j \neq i \; L_i < L_j$ and only tasks with a criticality level equal or higher than L are assured to be schedulable, while the others might be dropped. If a job with criticality level $L_t > L$ executes for more than $C_i(L)$, system switches to a higher criticality level.

Researchers realized that in a multicore system, tasks with priority less than L can be migrated to another core instead of being dropped. This is the reason why multicore mixed criticality systems, like the one presented in [2], are a very hot topic in the research world.

After examining the main concepts, section 2 presents an overview of response time analysis on single core (or partitioned multi core) environment, section 3 focuses on the dual-core semi partitioned model presented in [3] and section 4 examines the multi-core evolution presented in [2]. Eventually, section 5 discusses the work and draws some conclusions.

2 Response time analysis for single-core MCS

The first scheduling analysis for MCS was introduced by Vestal in 2007 [4][2] and it was based on a modified version of Audsley's priority assignment algorithm [5], which later was proven to be optimal. MCS with FPS scheduling are now analyzed with AMC (Adaptive Mixed Criticality) [6].

2.1 Audsley's priority assignment algorithm

The aim of the algorithm is to efficiently assign priorities to tasks in a fixed-priority scheduling system. More precisely, it operates as follows, assigning priorities from the smallest (less important) to the greatest (more important) task:

1. No task of the taskset has a priority assigned;
2. The priority *p* that is going to be assigned is initialized with the smallest one, so *p = 0;*
3. From the tasks with no priority assigned, choose the one that could be schedulable if it had priority *p+1.* If more than one task meet this criterion, choose one of them arbitrarily. If no such task exists, the system is not schedulable with PFP scheduling, end*;*
4. Assign priority *p+1* to the chosen task*;*
5. Assign to *p* the next priority, so *p=p+1;*
6. If there are still tasks with no priority assigned, go back to step 3.

The main point of the algorithm is that, at every iteration, in step 3 are executed at most *n-1* comparisons. This is possible thanks to the use of Joseph-Pandya [7] response time formula:

$$R_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{Tj} \right\rceil C_j \tag{1}$$

To compute the response time of task τ_i it is needed to know only which tasks have a higher priority than τ_i (which, in the Audsley's algorithm, are tasks without a priority assigned), not the actual priority value. This aspect allows reducing the complexity of the algorithm from *n!* to *n(n+1)/2*.

2.2 Vestal's algorithm

Vestal's algorithm is a modified version of Audsley's one that can be applied to MCS. There are only two differences in the procedure:

- The response time of a generic task τ_i is computed with a slightly modified version of the Joseph-Pandya formula, which takes in account also the critical level of the task:

$$R_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{Tj} \right\rceil C_j(L_i) \tag{2}$$

- In step 3, if more than one task that meet schedulability criteria, the one with greatest scaling factor is chosen. The scaling factor is the highest value by which all execution times C_i (L_i) can be multiplied while still maintaining the system feasible. In other words, it represents a measure of the quantity of work that can be added in a proportional manner to all the tasks for a specific schedulability analysis and a specific priority-assignment algorithm. [8] proved that Vestal's approach is optimal.

2.3 Adaptive Mixed Criticality

Very often a Mixed Criticality System has one or more "safety-critical" components that are subject to strict certification, while the others might not [6]. While assigning priority and testing schedulability of a system of this kind, two main aspects need to be considered:

- *Run-time robustness*, which means that, in case of system overload, low-criticality (LO-crit) tasks must be suspended before high-criticality (HI-crit) ones. In an informal way, if you need to choose which task to abandon, choose the lower-priority one;
- *Static verification*, which means that critical components of the system should be easily analyzable by Certification Authorities (CA's), and the conservative analysis done by the CA's should be as similar as possible to the one conducted by the system designer.

For this purpose, [6] introduced AMC, a priority assignment scheme derived from SMC (Static Mixed Criticality). Its main characteristic is the choice to deschedule all tasks of criticality *L* or lower if any job of criticality *L* executes for more than *C(L)* budget in order to guarantee enough time to higher criticality jobs. To do so, AMC requires a run time monitor, a system component that monitors the execution time of each task τ_i and trips if it executes for more than $C_i(L_i)$. Priorities in AMC are assigned to task using a version of Audsley's priority assignment algorithm, so the complexity of finding the optimal priority order for a taskset of *n* tasks is *n(n+1)/2* [2].

If the MCS has only two criticality levels, HI-crit and LO-crit, the response time analysis for AMC requires only three steps [6]:

1. Check the schedulability of LO-crit mode, which can easily be done with an adapted version of the *Joseph-Pandya* formula:

$$R_i(LO) = \sum_{j \in hp(i)} \left\lceil \frac{R_i(LO)}{Tj} \right\rceil C_j(LO) \tag{3}$$

2. Check the schedulability of HI-crit mode, proceeding in a similar way to the previous point:

$$R_i(HI) = \sum_{j \in hp(i)} \left\lceil \frac{R_i(HI)}{Tj} \right\rceil C_j(HI) \tag{4}$$

3. Check the schedulability of the criticality change. This exact analysis is possible only when the critical instant is clear, which is not the case for AMC. In fact, as evidence in [6], critical instant of AMC does not always coincide with the moment when sporadic tasks are released together. From the analysis of SMC, the following equation is derived:

$$R_i^* = \sum_{j \in hpH(i)} \left\lceil \frac{R_i^*}{Tj} \right\rceil C_j(HI) + \sum_{k \in hpL(i)} \left\lceil \frac{R_i(LO)}{T_k} \right\rceil C_k(LO) + C_i(HI) \tag{5}$$

where *hpH(i)* means HI-crit tasks with higher priority than *i* and *hpL(i)* means LO-crit tasks with higher priority than *i*. In brief, in this case we are just worried with HI-crit tasks, and the maximum interference they can suffer from LO-crit tasks is given by $R_i(LO)$, the time when τi ends its LO-crit budget and the criticality change occurs. Finally, we can assume that HI-crit tasks are executing with their HI-crit budget, and this is a conservative approach, since for every τi, $C_i(LO) \le C_i(HI)$.

3 SEMI-PARTITIONED MODEL ON DUAL-CORE PLATFORM

3.1 Overview

The direct predecessor of [2] is [3], where the same authors set up a semi-partitioned on a dual core platform. The semi partitioned model in [3] has two criticality levels (HI and LO crit) and works as follows:

- Each HI-crit task is statically allocated to a specific core;
- Some LO-crit tasks are also statically allocated;
- The remaining LO-crit tasks can migrate, like in global scheduling;
- Each core has a criticality level indicator Γ initialized with LO;
- During execution, LO-crit tasks cannot exceed their LO-crit budget. If an HI-crit task on one core exceeds his LO-crit budget, the core enters HI-crit mode (Γ=HI), so that non-statically allocated LO-crit tasks are migrated to the other core, while all tasks remain schedulable;
- If also the other core enters HI-crit mode (so both cores are in HI-crit mode at the same time), LO-crit tasks are abandoned, while HI-crit tasks continue executing.

It should be noted that, for the sake of brevity, a core in LO-crit mode or HI-crit might be (even if improperly) called respectively "LO-crit core" or "HI-crit core".

3.2 Model

Let S be the taskset running on a dual-core MCS. Tasks in S can be:

- HI-crit statically allocated to the first core (*HI1*);
- HI-crit statically allocated to the other core (*HI2*);
- LO-crit statically allocated to the first core (*LO1*);
- LO-crit statically allocated to the last core (*LO2*);
- LO-crit migratable, initially allocated to the first core (*MIG 1*);
- LO-crit migratable, initially allocated to the last core (*MIG2*).

S can be represented with the following relation:

$$S=(LO_1 \cup LO_2) \cup (HI_1 \cup HI_2) \cup (MIG_1 \cup MIG_2)$$

Let Γi be the criticality level of core *ci*. If Γ1 ≡ Γ2 ≡ LO, all tasks are executing with their LO-crit budget, then each core *i* is in what we define state *Xi* (steady state). The situation can be represented as follow:

$$X_1=LO_1 \cup HI_1 \cup MIG_1$$
$$X_2= LO_2 \cup HI_2 \cup MIG_2$$
$$S=X_1 \cup X_2$$

If only one core enters HI-crit mode, so Γ1 ≠ Γ2, migratable tasks of that core are migrated to the other core, which is still in LO-crit mode. More formally, let cj be the core in HI-crit mode (Γj = HI) and ck the one still in LO-crit mode (Γk = LO). This state is defined as Y(j):

$$Y(j)_j=LO_j \cup HI_j$$
$$Y(j)_k= LO_k \cup HI_k \cup MIG_k \cup MIG_j$$
$$S=Y(j)_1 \cup Y(j)_2$$

Since core *k* has now extra tasks compared to the starting configuration, priority orders in Y(j)*K* might be recalculated. Furthermore, migrated tasks have a reduced deadline D*, which, in worst case, is D* = Di - (Ri - Ci) when task still need to execute all its LO-crit budget after migration. There might be moments when both cores are in HI-crit mode (Γ1 ≡ Γ2 = HI). In this case, some tasks need to be dropped to give enough space to HI-crit tasks to execute with their HI-crit budget. However, we must distinguish between two possible behaviors, depending on the moment in which both cores enter HI-crit mode. The first is that both cores enter HI-crit mode at the same time. In this case, MIG tasks have no core to migrate to, so they must be abandoned. Both cores execute HI-crit tasks (with their HI-crit budget) and statically allocated LO-crit tasks. We can define this situation as state BXi, formally:

$$BX_1=LO_1 \cup HI_1$$
$$BX_2= LO_2 \cup HI_2$$
$$S=\mathrm{B}X_1 \cup BX_2 \cup MIG_1 \cup MIG_2$$

The second (and last) possible behavior happens when one core (*ck*) enters HI-crit mode while the other (*cj*) has already entered. Since ck has been executing also MIGj tasks (as represented before in scenario *Y(j)k*), it might need to drop all LO-crit tasks (both migratable and statically allocated) to allow HI-crit tasks to execute with their HI-crit budget. The situation can be described as follows:

$$BY(j)_j=LO_j \cup HI_j$$
$$BY(j)_k= HI_k$$
$$S=\mathrm{B}Y(j)_1 \cup \mathrm{B}Y(j)_2 \cup MIG_k \cup MIG_j \cup LO_k$$

3.3 Analysis

The analysis of the model above is quite like the analysis we've done in section 2.3 for *AMC*. It consists of three steps:

1. Schedulability test of state X, where for each core *ck* Γk ≡ LO, so tasks are fully partitioned and execute with their LO-crit budget:

$$\forall \tau_i \in X: \; R_i(LO) = C_i(LO) + \sum_{j \in chp(i)} \left\lceil \frac{R_i(LO)}{Tj} \right\rceil C_j(LO) \quad (6)$$

   where *chp(i)* means all task on the same core of τi with a higher priority than τi.

2. Schedulability test of states $Y(j)_j$ and $Y(j)_k$, where cj is the core in HI-crit mode (Γj = HI) and ck the one in LO-crit (Γk = LO). Here we must distinguish between the analysis of $Y(j)_j$ and $Y(j)_k$. In the first case, HI-crit tasks are executing with their HI-crit budget, while LO-crit statically allocated tasks are executing with their LO-crit budget:

$$\forall \tau_i \in Y(j)_j: \; R_i(HI) = C_i(L_i) + \sum_{k \in chp(i)} \left\lceil \frac{R_i(HI)}{T_k} \right\rceil C_k(L_k) \quad (7)$$

Instead, the analysis of $Y(j)_k$ must consider release jitters of migrating tasks, which in the worst case is $J_i = R_i - C_i(LO)$, so the response time analysis becomes:

$$\forall\tau_i \in Y(j)_k : R_i^*(LO) = C_i(LO) + \sum_{k \in chp(i)} \left\lceil \frac{R_i^*(LO) + J_k}{T_k} \right\rceil C_k(LO) \tag{8}$$

3. Schedulability test of the criticality change itself, which must be done in two steps: checking the schedulability of cores entering HI-crit mode and checking the schedulability of migrating tasks. In the first case, statically allocated LO-crit tasks remain in execution with their LO-crit budget, while HI-crit tasks start to execute with their HI-crit budget, so the interference from migratable tasks occurs only as long as task's LO-crit budget is not expired. Thus, the following equation summarizes the analysis:

$$\forall\tau_i \in Y(j)_j : R_i^*(HI) = C_i(L_i) + \sum_{k \in chpH(i)} \left\lceil \frac{R_i^*(HI)}{T_k} \right\rceil C_k(LO) + \sum_{l \in chpL(i)} \left\lceil \frac{R_i^*(HI)}{T_l} \right\rceil C_l(LO) + \sum_{m \in chpMIG(i)} \left\lceil \frac{R_i\ (LO)}{T_m} \right\rceil C_m(LO) \tag{9}$$

At last we need to check the schedulability of tasks when system becomes in state BY. In this context, LO-crit tasks need to be abandoned during mode change. The equation below follows:

$$R_i^{**}(HI) = C_i(HI) + \sum_{k \in chpH(i)} \left\lceil \frac{R_i^{**}(HI)}{T_k} \right\rceil C_k(HI) + \sum_{l \in chpL(i)} \left\lceil \frac{R_i^*(LO)}{T_l} \right\rceil C_l(LO) \tag{10}$$

Finally, complexity of such equations for a taskset of n tasks is $O(n^2)$.

3.4 Task allocation

Besides RTA, it is time to see how to assign priorities to tasks in a specific taskset and bound them to the cores. So, here's the generic algorithm:

1. Sort the taskset with a criticality-aware order, like criticality-aware utilization descending. This is important because it puts all HI-crit tasks in front of LO-crit ones, so the schedulability of the HI-crit tasks can be checked before (if HI-crit tasks are not schedulable, the whole taskset is not schedulable with a semi-partitioned approach);
2. Fetch a task from the sorted taskset;
3. Assign one core to the fetched task according to a chosen bin-packing heuristic. Bin packing is a NP-hard problem, so there are numerous heuristics that allow arriving at an approximate solution. For this context, it has proven by [2] that the best heuristics are Worst Fit or First Fit;
4. Assign priorities to all task using Audsley's algorithm and check if the configuration found allows all tasks to be schedulable;
5. When an unschedulable task is found, verify if assigning it to the other core changes the situation yielding it a schedulable configuration;
6. If scheduling the fetched LO-crit task cannot be done by any of the two cores, consider setting it as migratable.
7. According to the results of step 3 and 6, assign the fetched task to the core computed on step 3 and set it migratable if decided in step 6;
8. Check for each core if the configuration found in the previous step is schedulable, assigning priorities to each task with Audsley's algorithm;
9. In case of unschedulable task, assign it to the other core and check again;
10. In case of unschedulability also on the other core, the task is not schedulable by the semi-partitioned algorithm;
11. While the taskset is not empty or a task is found to be unschedulable, return to step 2.

There is an inaccuracy in step 7: based on the approach chosen, you can set migratable the current fetched task (described in [3] as *Semi1*) or the highest priority LO-crit task among all migratable tasks (named *Semi2*). This, in combination with the bin-packing heuristic, will impact the optimality of the solution. Evaluation conducted in [3] highlights how *Semi2FF* and *Semi2WF*, which migrate the highest priority tasks among all LO-crit migratable task and use respectively First-Fit and Worst-Fit as bin-packing approximation, are the ones which perform best.

4 EXTENDING SEMI-PARTITIONED MODEL ON MULTICORE

In [2], which is the paper under review, the real new work begins with theorizing about the existence of a number, n_b, which, for a system of n cores, represents the maximum number of cores that can be simultaneously in HI-crit mode without the need to drop LO-crit tasks but only to migrate some. For example, for the dual-core MCS described in previous section, $n = 2$ (since it's dual core) and $n_b = 1$ (because with only one core in HI-crit mode, no task is abandoned). The author of [2] said that values n_b is crucial, since it can determine success or failure of the model. In fact, a small value of n_b means that most of the times LO-crit tasks need to be abandoned, so the advantages in schedulability performance of the model are quite low. Inversely, high values of n_b means more migrations, so more complexity, so less implementability of the model. If we assume that the probability p for the event "a core is entering HI-crit mode" is fixed, we can compute the probability of the event "m core out of n are in HI-crit mode at the same time". In fact, the latter, which [2] calls $f(m,n)$, is a simple binomial distribution:

$$f(m,n) = \binom{n}{m} p^m (1-p)^{n-m} \quad (11)$$

so, the probability of more than X cores entering HI-crit mode at the same time is:

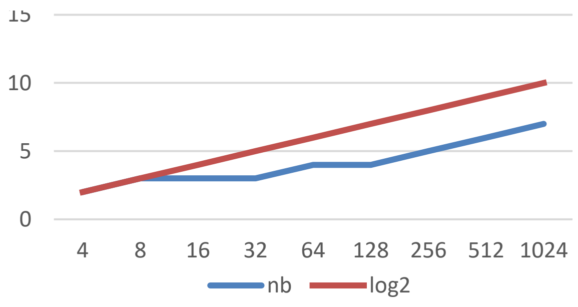
$$F(X,n) = \sum_{i=X+1}^{n} f(i,n) \quad (12)$$

Summing up, *F(X,n)* represents the probability of the needs of abandoning LO-crit tasks, given *p* (probability of a core entering HI-crit mode) as a static parameter. If an acceptable upper bound value for *F(X,n)* is "statically" defined, which will serve as a tolerance bound (p_{tol}), it's possible to compute n_b starting from *n*. In other words, it's possible to find the number of cores out of *n* which can enter HI-crit mode without the need to abandon any task by finding the largest number *X* for which $F(X,n) \leq p_{tol}$. If, by assumption, p_{tol} is assigned the value *F(2,4)*, we obtain values reported in table 1. As shown, the values obtained are very similar to $\lceil log2(n) \rceil$, so, if $p = 10^{-4}$ is assumed, $\lceil log2(n) \rceil \geq n_b$. More technically, for systems with $\{w \mid 2^n+1 \leq w \leq 2^{n+1}\}$ cores, *n+1* is assumed as a "safe" boundary number.

Table 1: maximum values of X in relation to cores number and log_2 [2]

n	n_b	$log_2(n)$
4	2	2
8	3	3
16	3	4
32	3	5
64	4	6
128	4	7
256	5	8
512	6	9
1024	7	10

Figure 1: Comparison between log_2 and n_b [2]



Now there are all the elements needed to propose a semi-partitioned model for multi-core MCS. Basically, it's a revisitation of what the same author had already proposed in [3] (and it was reported on this document at section 3.1):

- Tasks can be of three types: HI-crit, statically allocated LO-crit and migratable LO-crit. The first two are statically allocated to a core, while those belonging to the last category are assigned to a core, but they can migrate to another one in some cases later described;
- When the system starts, all tasks are executing with their LO-crit budget on the core previously assigned to them. No HI-crit task is allowed to exceed its HI-crit budget and no LO-crit task is allowed to exceed its LO-crit budget;
- If a HI-crit task exceed his LO-crit budget, his core (c_i) enters HI-crit mode. In this case, if number of cores in HI-crit mode is less or equal than $\lceil log_2(n) \rceil$, LO-crit migratable tasks migrate from c_i to a paired LO-crit mode core; otherwise some LO-crit tasks will be abandoned to allow HI-crit tasks to execute with their HI-crit budget;
- If a core returns to LO-crit mode (after it experienced an idle tick), tasks that have migrated from it returns to their host core.

Now that the semi-partitioned approach is defined, it's time to address the migration problem which consists of determining the destination core for a task that needs to migrate.

4.1 Semi-partitioned model on four-core platform

The main difference between the semi-partitioned model defined in [3] and the one defined in [2] is the migration problem. In fact, in [3] having only two cores means that if core c_1 enters HI-crit mode his task can be migrated only to core c_2 or abandoned (It depends on Γ_2). On a four-core system, for example, if core c_1 enters HI-crit mode, migratable tasks have three possible destinations: c_2, c_3 and c_4. Also, migratable tasks can be migrated all to one core or they can be split among some of them. As can be seen, it's important to define a migration policy because it can impact performance and complexity (so analyzability) of the system.

On a four-core platform, if only one core enters HI-crit mode, it's possible to distinguish between three migration models:

- Model 1: all migrating tasks have only one destination core;
- Model 2: all migrating tasks are distributed among two cores;
- Model 3: all migrating tasks can migrate to any of the remaining three cores.

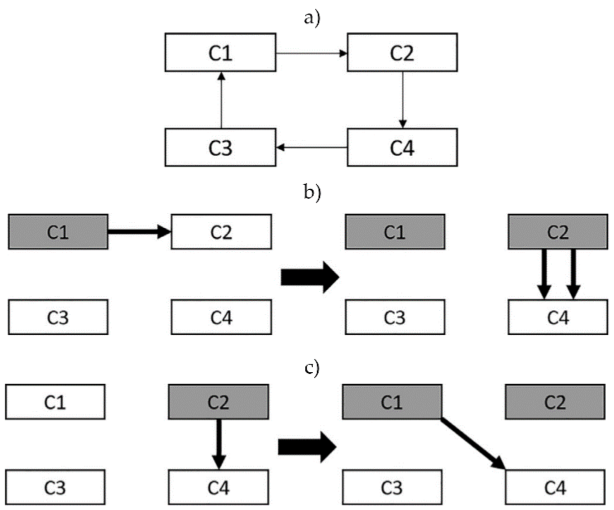
As stated in the previous section (table 1), it must be taken into consideration that for a four-core system the boundary number $n_b = 2$, so two HI-crit cores at the same time must be tolerated without abandoning any task.

4.1.1 Model 1

Model 1 allows migratable tasks to migrate to only one core, determined by a fixed route.

In *figure 1 a)* we can see a representation of the model, where rectangles are cores (white LO-crit, gray HI-crit), thin arrows the migration routes, thicker arrows indicate task migration and thickest arrow different steps. As can be seen, the migration route is a clockwise circle ($c_1 \rightarrow c_2 \rightarrow c_4 \rightarrow c_3 \rightarrow c_1$), so, starting from any core, it's always possible to reach a LO-crit one if it exists.

Figure 2: Model 1 [2]



In *figure 1 b)* we can see the scenario where c_1 enters HI-crit mode, so its migratable tasks are migrated to c_2. Then also c_2 enters HI-crit mode so, following the route, its migratable tasks are migrated to c_4. Here it's important to note that, because only two cores are in HI-crit mode, no task must be abandoned, so c_4 hosts migratable tasks of both core c_1 and c_2.

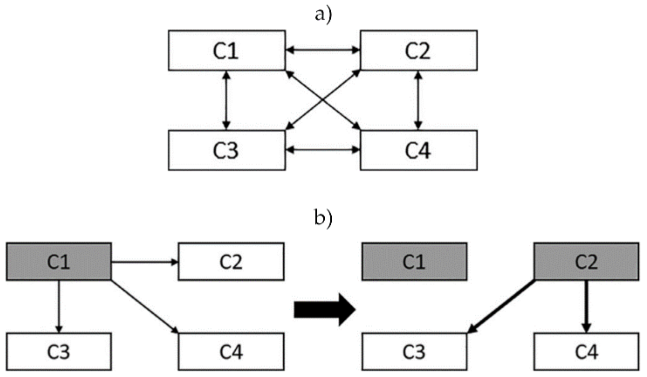
Figure 1 c) shows the scenario where c_2 enters HI-crit mode, so its migratable tasks are migrated to c_4. Then also c_1 enters HI-crit mode and, following the route, its migratable tasks should be migrated to c_2. But, since c_2 is already in HI-crit mode, c_1's migratable tasks are migrated to c_4. Note that, also in this case, c_4 hosts both c_1's migratable tasks and c_2's ones. This leads to an unbalance of the workload, especially if migratable tasks have a huge load. This is the problem that Model 2 and Model 3 try to solve

4.1.2 Model 2

In this model, each core is paired with his two neighbors, so migratable tasks have two possible cores to migrate to. More in depth, in a four-core system there are four pairs of cores: (c_1, c_2), (c_1, c_3), (c_2, c_4), (c_3, c_4). Here the problem is to choose where to migrate each task belonging to the migratable set of each core. An example of this model is given in *figure 3*.

Figure 3 b) shows a scenario where core c_1 enters HI-crit mode, so its migratable tasks are divided between cores c_2

Figure 3: Model 2 [2]



and c_3, thanks to the relations (c_1, c_2) and (c_1, c_3). Then, also c_2 enters HI-crit mode and it's migratable tasks should be split between c_1 and c_4 because of the relations (c_1, c_2) and (c_2, c_4). Note that, since c_2 hosts also c_1's migratable tasks, the latter try to come back to c_1, but, since c_1 is in HI-crit mode, they are migrated to c_3 thanks to the relation (c_1, c_3).

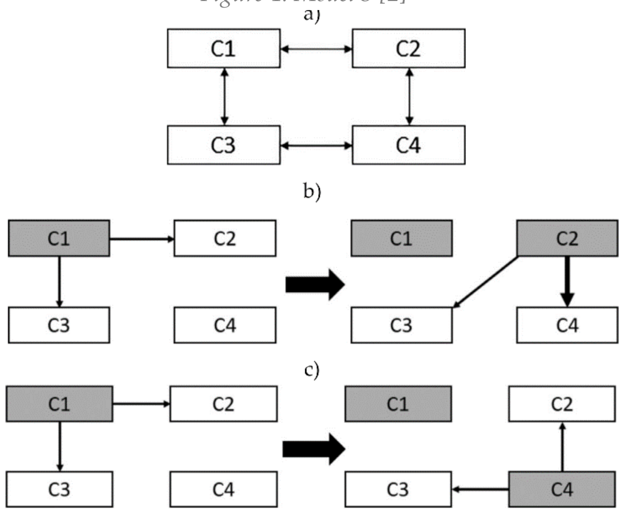
A second scenario is given by *figure 3 c)*. Core c_1 enters HI-crit mode, so, as stated before, its migratable tasks are divided between cores c_2 and c_3. Then, also c_4 enters HI-crit mode, so its migratable tasks are divided between core c_2 and c_3, since the pairing relations (c_2, c_4) and (c_3, c_4).

The partition of tasks between the various cores is an instance of the *bin-packing* problem. As discussed in section 3.1, this is a NP-hard problem, so there are various heuristics that give an approximated solution. In this situation, the heuristic that provides a balanced distribution is Worst-Fit (WF), which is the one used here.

4.1.3 Model 3

In this model (shown in *figure 4*), each core is paired

Figure 4: Model 3 [2]



with any other core in the system, so tasks can migrate to any core in LO-crit mode. While this allows the model to make the most out of the computational capacity of the system (tasks can be distributed in proportion to each core's load), on the other it can be very complex to analyze, as we will see in next sections. Anyway, *figure 4 b)* represents a scenario where core c_1 enters HI-crit mode, so his migratable tasks are distributed among c_2, c_3 and c_4 with a Worst-Fit heuristic. When also core c_2 enters HI-crit mode, all its migratable tasks (so also a subset of c_1's migratable tasks) are split between c_3 and c_4, which are the only cores still in LO-crit mode. As it is immediately evident, load is more balanced than, for example, Model 1. The next section discusses the various models in detail.

4.2 Model analysis

4.2.1 Model 1

According to this model, each core is paired with the next one, so in a system with four cores, they can be paired based on the following path: $c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow c_4 \rightarrow c_1$. So, if c2 enters HI-crit mode, its migratable tasks are moved to c3, while if also c3 enters HI-crit mode, migratable tasks of c2 and c3 shall be moved to c4. To represent the model on a four-core platform, let's define S as a taskset that contains three types of tasks:

- HI-criticality: they're denoted with HI$_i$, which means HI-criticality tasks allocated to core c_i;
- LO-criticality statically allocated: for each core c_i, its LO-criticality statically allocated tasks are indicated with LO$_i$;
- Migratable LO-criticality, which are denoted with MIG$_{i,j,k}$: migratable LO-criticality tasks bounded to core c_i but that can migrate to c_j and eventually to c_k.

So, *S* can be defined as:

$$S = (LO_1\ U\ LO_2\ U\ LO_3\ U\ LO_4)\ U\ (HI_1\ U\ HI_2\ U\ HI_3\ U\ HI_4)\ U\big(MIG_{1,2,3}\ U\ MIG_{2,3,4}\ U\ MIG_{3,4,1}\ U\ MIG_{4,1,2}\big)$$

The "normally-operative" state of the system on each core c_i can be defined as X$_i$ (same of *section 3.2*):

- $X_1 = LO_1\ U\ HI_1\ U\ MIG_{1,2,3}$
- $X_2 = LO_2\ U\ HI_2\ U\ MIG_{2,3,4}$
- $X_3 = LO_3\ U\ HI_3\ U\ MIG_{3,4,1}$
- $X_4 = LO_4\ U\ HI_4\ U\ MIG_{4,1,2}$

In relation to X$_i$, $S = X_1\ U\ X_2\ U\ X_3\ U\ X_4$. In this state, for each core c_i, Γ_i (criticality level of core c_i) ≡ LO. This means that all tasks are scheduled, partitioned through the cores and all of them are executing with their LO-crit budget. In this case, response time analysis uses Joseph-Pandya's formula:

$$\forall \tau_k \in X_i : R_k(LO) = \sum_{j \in chp(k)} \left\lceil \frac{R_k(LO)}{T_j} \right\rceil C_j(LO)$$

(13)

If a HI-crit task on core c_i executes for more than its LO-crit budget, Γ_i is set to HI and c_i enters HI-crit mode. In this case, HI-crit tasks (*HI$_i$*) execute with their HI-crit budget, statically allocated LO-crit tasks (*LO$_i$*) continue to execute (with their LO-crit budget) while migratable LO-crit tasks (*MIG$_{i,j,k}$*) are migrated to c_j. So, formally, we end up with a new state *Y(j)$_k$*, which is the state of core c_j when core c_j enters HI-crit mode:

- $Y(1)_1 = LO_1$ U HI_1
- $Y(1)_2 = LO_1$ U HI_1 U $MIG_{2,3,4}$ U $MIG_{1,2,3}$
- $Y(1)_3 = X_3 = LO_3$ U HI_3 U $MIG_{3,4,1}$
- $Y(1)_4 = X_4 = LO_4$ U HI_4 U $MIG_{4,1,2}$
- $S = Y(1)_1$ U $Y(1)_2$ U $Y(1)_3$ U $Y(1)_4$

As can be noted, the system is as if it were composed of two dual core subsystems described in *section 3*, one in steady state (*S$_1$*) and the other in a state where only one core entered HI-crit mode (*S$_2$*). Accordingly, response-time analysis is quite similar and has to address some problems: migrated tasks have a reduced deadline and their release jitter need to be taken into account:

$$\forall \tau_i \in MIG_{1,2,3} : D_i' = D_i - \big(R_i(LO) - C_i(LO)\big)$$
$$J_i = R_i(LO) - C_i(LO)$$

(14)

Since the system has some components in HI-crit mode and others in LO-crit mode, we can say the system is in a mixed state (MIX). So, response time analysis is given by this formula:

$$\forall \tau_i \in Y(1)_1 : R_i(MIX) = C_i(L_i) + \sum_{k \in chp(i)} \left\lceil \frac{R_i(MIX)}{T_k} \right\rceil C_k(L_k) + \sum_{j \in chpMIG(i)} \left\lceil \frac{R_i(LO)}{T_j} \right\rceil C_j(LO)$$

$$\forall \tau_i \in Y(1)_2 : R_i(LO)' = C_i(LO) + \sum_{j \in chp(i)} \left\lceil \frac{R_i(LO) + J_j}{T_j} \right\rceil C_j(LO)$$

(15)

where *chpMIG(i)* are tasks belonging to MIG$_{i,j,k}$.

If another core (c_h) enters HI-crit mode, two scenarios are possible: either c_h is hosting migratable tasks of the other HI-crit core (c_j) or is not. The latter is, informally, as if one of the two cores belonging to subsystem still in steady state (*S$_1$*) entered HI-crit mode, so it translates from state X to state Y. So, for example, if c_3 enters HI-crit mode, we obtain the following situation:

- $Y(1,3)_1 = Y(1,3)_3$
- $Y(1,3)_2 = Y(1)_2$
- $Y(1,3)_3 =$ LO$_3$ U HI$_3$
- $Y(1,3)_4 =$ LO$_4$ U HI$_4$ U MIG$_{4,1,2}$ U MIG$_{3,4,1}$
- S $= Y(1,3)_1$ U $Y(1,3)_2$ U $Y(1,3)_3$ U $Y(1,3)_4$

Since $Y(1,3)_3$ and $Y(1,3)_4$ are "twins" of $Y(1)_1$ and $Y(1)_2$, the response time analysis is the same.

If c_h is hosting migratable tasks of the other HI-crit core, things are a bit different. In fact, in this case migratable tasks of both c_j and c_h need to migrate to another core. For example, if c_2 enters HI-crit mode, the scenario (that we call Y$_{1,2}$ for further reference) is the following:

- $Y(1,2)_1 = Y(1)_1$
- $Y(1,2)_2 =$ HI$_2$ U LO$_2$
- $Y(1,2)_3 =$ X$_3$ U $MIG_{2,3,4}$ U $MIG_{1,2,3}$
- $Y(1,2)_4 =$ X$_4$
- $S = Y(1,2)_1$ U $Y(1,2)_2$ U $Y(1,2)_3$ U $Y(1,2)_4$

Here, since MIG$_{1,2,3}$ is migrated for the second time, its tasks might experience a potentially narrower deadline and higher jitter. For example, if $\tau_i \in$ MIG$_{1,2,3}$ is migrated from c_1 to c_2 and from c_2 to c_3 in a single release, its deadline and jitter would be:

$$D_i'' = D_i - \big(R_{i,1} - C_i\big) - \big(R_{i,2} - C_i\big)$$
$$J_i' = \big(R_{i,1} - C_i\big) + \big(R_{i,2} - C_i\big)$$

(16)

where $R_{i,n}$ means response time of task τ_i on core c_m. This consideration, however, does not change the behavior of the response time analysis, which is the same as described before.

Another scenario is when a third core enters HI-crit mode, which implies that all its LO-crit tasks need to be descheduled because the number of HI-crit cores, which is 3, is greater than the boundary number n$_b$ for a four-core system, which is 2. For example, if in Y$_{1,2}$ core c_3 enters HI-crit mode, the scenario that is created is the following:

- $Y(1,2,3)_1 = Y(1)_1$
- $Y(1,2,3)_2 = Y(1,2)_2$
- $Y(1,2,3)_3 =$ HI$_3$
- $Y(1,2,3)_4 =$ X$_4$

- S=$Y(1,2,3)_1$ U $Y(1,2,3)_2$ U $Y(1,2,3)_3$ U $Y(1,2,3)_4$

It follows that response time analysis is:

$\forall\tau_i \in Y(1,2,3)_1$:

$$R_i(HI)' = C_i(HI) + \sum_{k\in chp(i)} \left\lceil \frac{R_i(HI)}{T_k} \right\rceil C_k(HI) + \sum_{j\in chpL(i)} \left\lceil \frac{R_i(LO)' + J'_j}{T_j} \right\rceil C_j(LO) \quad (17)$$

4.2.2 Model 2

In this model, each core is paired with its two neighbors, so its tasks migrate to two destinations. In a four-core environment four different pairs exist: *(c1, c2), (c2, c4), (c1, c3)* and *(c3, c4)*. So, if a core (for example *c1*) enters HI-crit mode, its migratable tasks are split in two groups and migrated to the two paired cores (*c2* and *c3*). If also *c2* enters HI-crit mode, both its migratable tasks and *c1*'s migratable ones that were executing over here needs to be migrated. Since *c1* is still in HI-crit mode, it cannot host any migratable task, so its migratable tasks that were executing on *c2* are transferred to *c3* due to the pairing relationship *(c1, c3)*, while c2's own migratable tasks migrates to *c4*.

In this way we can define the initial partitioning of the taskset (using the same notation of model 1):

$S = (LO_1\ U\ LO_2\ U\ LO_3\ U\ LO_4)\ U\ (HI_1\ U\ HI_2\ U\ HI_3\ U\ HI_4)$
$U\ (MIG_{1,2,3}\ U\ MIG_{1,3,2})U\ (MIG_{2,1,4}\ U\ MIG_{2,4,1})$
$U\ \ (MIG_{3,1,4}\ U\ \ MIG_{3,4,1})\ U\ (MIG_{4,2,3}\ U\ \ MIG_{4,3,2})$

As was the case with the first model, initially all tasks are statically partitioned to each core and they're executing with their LO-crit budget. Let's define this steady state X_i (where *i* is the core) as:

- $X_1 = LO_1\ U\ HI_1\ U\ MIG_{1,2,3}\ U\ MIG_{1,3,2}$
- $X_2 = LO_2\ U\ HI_2\ U\ MIG_{2,3,4}\ U\ MIG_{2,4,1}$
- $X_3 = LO_3\ U\ HI_3\ U\ MIG_{3,1,4}\ U\ MIG_{3,4,1}$
- $X_4 = LO_4\ U\ HI_4\ U\ MIG_{4,1,2}\ U\ MIG_{4,3,2}$
- $S = X_1\ U\ X_2\ U\ X_3\ U\ X_4$

This state is analogous to state X described in Model 1, hence the response time formula is also the same (13).

As for Model 1, if, for example, an HI-crit task on core *c1* executes for more than his LO-crit budget, *Γ1* is set to HI, so the core enters HI-crit mode. Now, only HI-crit and LO-crit continue to execute on c*1*, while $MIG_{1,2,3}$ and $MIG_{1,3,2}$ migrate respectively to cores *c2* and *c3*. So, for each core c_k we end up with the state $Y(1)_k$:

- $Y(1)_1$=LO_1 U HI_1
- $Y(1)_2$= X_2 U $MIG_{1,2,3}$
- $Y(1)_3$= X_3 U $MIG_{1,3,2}$
- $Y(1)_4$= X_4
- S=$Y(1)_1$ U $Y(1)_2$ U $Y(1)_3$ U $Y(1)_4$

Since migrated tasks might experience jitter and reduced deadlines, *equation (14)* must be used to estimate worst case for taskset $MIG_{1,2,3}$ and $MIG_{1,3,2}$. Also, to compute response time for cores c1, c2 and c3, *equation (15)* must be used.

Same considerations on response-time analysis can be done if another core which is not hosting other cores' migratable tasks enters HI-crit mode. Continuing with the previous example, this means that *c4* enters HI-crit mode, so its HI-crit tasks start to execute with their HI-crit budget and its migratable tasks are migrated to core *c2* and *c3*. So, we end up with:

- $Y(1,4)_1$=$Y(1)_1$
- $Y(1,4)_2$= $Y(1)_2$ U $MIG_{4,2,3}$
- $Y(1,4)_3$= $Y(1)_3$ U $MIG_{4,3,2}$
- $Y(1,4)_4$= $LO_4\ U\ HI_4$
- S=$Y(1,4)_1$ U $Y(1,4)_2$ U $Y(1,4)_3$ U $Y(1,4)_4$

Analysis is similar to the previous case: tasks belonging to $MIG_{4,2,3}$ and $MIG_{4,3,2}$ might experience release jitter and reduced deadline, both of which can be computed using equation *(14)*. Response time of all tasks needs to be computed with equation *(15)*.

A slightly different behavior happens if a core which is hosting other core's migratable tasks, enters HI-crit mode. For example, if system is in *Y(1)* state and *c2* enters HI-crit mode, *HI2* tasks start to execute with their HI-crit budget, so $MIG_{2,3,4}$, $MIG_{2,4,1}$ and $MIG_{1,2,3}$ tasksets have to be moved to another core. In particular, the latter has two possibilities, going back to *c1* or migrating to *c3*, but since *c1* is still in HI-crit mode, it needs to be moved to *c3*. The scenario is the following:

- $Y(1,2)_1$=$Y(1)_1$
- $Y(1,2)_2$= LO_2 U HI_2
- $Y(1,2)_3$= X_3 U $MIG_{1,3,2}$ U $MIG_{1,2,3}$
- $Y(1,2)_4$= X_4 U $MIG_{2,1,4}$ U $MIG_{2,4,1}$
- S=$Y(1,2)_1$ U $Y(1,2)_2$ U $Y(1,2)_3$ U $Y(1,2)_4$

So, $MIG_{1,2,3}$ tasks might experience release jitter and reduced deadline, which can be computed (and must be taken into account) with equation *(14)*. It's important to notice that tasks on *c3* and *c4* are still executing with their LO-crit budget, so equation *(15)* gives a response-time value for their tasks.

If a third or more core enters HI-crit mode, their LO-crit tasks (both LO_i and HI_i) needs to be descheduled to guarantee HI-crit tasks enough time budget (because number of HI-crit cores exceed boundary value *nb*. For example, if system is in Y(1,2) state and core c3 enters HI-crit mode, we came in the following situation:

- $Y(1,2,3)_1$=$Y(1)_1$
- $Y(1,2,3)_2$= $Y(1,2)_2$
- $Y(1,2,3)_3$= HI_3
- $Y(1,2,3)_4$= $Y(1,2)_4$
- S=$Y(1,2,3)_1$ U $Y(1,2,3)_2$ U $Y(1,2,3)_3$ U $Y(1,2,3)_4$

Since in core c3 are executing only HI-crit tasks (with their HI-crit budget), equation *(17)* shall be used to compute response time.

4.2.3 Model 3

According to this model, migratable tasks "equally" migrate to any core in LO-crit mode. The "equally" adverb means that migratable tasks are allocated among the various LO-crit cores taking into account the overhead on a core that hosting them will cause. In other words, while migrating a task, its ability to increase the utilization rate of the host core must be taken into account. This behavior will be repeated if a second core enters HI-crit mode, so migratable tasks originally allocated to it and ones it was hosting need to be migrated "equally" to the other two LO-crit cores. Since $n_b = 2$, if a third or fourth core enters HI-crit mode, all LO-crit (statically allocated, migratable and migrated) tasks executing on that core must be abandoned. More formally, let's define a taskset S as:

$$S = (LO_1 \, U \, LO_2 \, U \, LO_3 \, U \, LO_4) \, U \, (HI_1 \, U \, HI_2 \, U \, HI_3 \, U \, HI_4) \\ U \, ((MIG_{1,2,3} \, U \, MIG_{1,2,4}) \, U \, (MIG_{1,3,2} \, U \, MIG_{1,3,4}) \\ U \, (MIG_{1,4,2} \, U \, MIG_{1,4,3})) \, U \, ((MIG_{2,1,3} \, U \, MIG_{2,1,4}) \\ U \, (MIG_{2,3,1} \, U \, MIG_{2,3,4}) \, U \, (MIG_{2,4,1} \, U \, MIG_{2,4,3})) \\ U \, ((MIG_{3,1,2} \, U \, MIG_{3,1,4}) \, U \, (MIG_{3,2,1} \, U \, MIG_{3,2,4}) \\ U \, (MIG_{3,4,1} \, U \, MIG_{3,4,2})) \, U \, ((MIG_{4,1,2} \, U \, MIG_{4,1,3}) \\ U \, (MIG_{4,2,1} \, U \, MIG_{4,2,3}) \, U \, (MIG_{4,3,1} \, U \, MIG_{4,3,2}))$$

As for the other models, in steady state X_i all tasks are statically partitioned on each core and all of them are executing with their LO-crit budget:

- $X_1 = LO_1 \, U \, HI_1 \, U \, (MIG_{1,2,3} \, U \, MIG_{1,2,4})$
  $U \, (MIG_{1,3,2} \, U \, MIG_{1,3,4}) \, U \, (MIG_{1,4,2} \, U \, MIG_{1,4,3})$
- $X_2 = LO_2 \, U \, HI_2 \, U \, (MIG_{2,1,3} \, U \, MIG_{2,1,4})$
  $U \, (MIG_{2,3,1} \, U \, MIG_{2,3,4}) \, U \, (MIG_{2,4,1} \, U \, MIG_{2,4,3})$
- $X_3 = LO_3 \, U \, HI_3 \, U \, (MIG_{3,1,2} \, U \, MIG_{3,1,4})$
  $U \, (MIG_{3,2,1} \, U \, MIG_{3,2,4}) \, U \, (MIG_{3,4,1} \, U \, MIG_{3,4,2})$
- $X_4 = LO_4 \, U \, HI_4 \, U \, (MIG_{4,1,2} \, U \, MIG_{4,1,3})$
  $U \, (MIG_{4,2,1} \, U \, MIG_{4,2,3}) \, U \, (MIG_{4,3,1} \, U \, MIG_{4,3,2})$
- $S = X_1 \, U \, X_2 \, U \, X_3 \, U \, X_4$

Since tasks are statically partitioned and all execute with LO-crit budget, equation *(13)* must be used to compute response time.

If a core (for example c_1) enters HI-crit mode, migratable tasks on that core must be transferred to the other three cores in an equal manner. Thus, we come to the following state:

- $Y(1)_1 = LO_1 \cup HI_1$
- $Y(1)_2 = X_2 \cup (MIG_{1,2,3} \cup MIG_{1,2,4})$
- $Y(1)_3 = X_3 \cup (MIG_{1,3,2} \cup MIG_{1,3,4})$
- $Y(1)_4 = X_4 \cup (MIG_{1,4,2} \cup MIG_{1,4,3})$
- $S = Y(1)_1 \cup Y(1)_2 \cup Y(1)_3 \cup Y(1)_4$

This state is very similar to *Y(i)* as discussed for Model 2, so similar considerations should be made: migrated tasks suffer release jitter and reduced deadline computed via equation *(14)*, while response time analysis is provided by equation *(15)*.

If a second core (for example c_2) enters HI-crit mode, all migratable tasks currently executing on the core (both originally allocated or migrated) needs to go to another one. The model representing this state is the following:

- $Y(1,2)_1 = Y(1)_1$
- $Y(1,2)_2 = LO_2 \cup HI_2$
- $Y(1,2)_3 = Y(1)_3 \cup (MIG_{2,3,1} \cup MIG_{2,3,4})$
  $\cup MIG_{2,1,3} \cup MIG_{1,2,3}$
- $Y(1,2)_4 = Y(1)_4 \cup (MIG_{2,4,1} \cup MIG_{2,4,3})$
  $\cup MIG_{2,1,4} \cup MIG_{1,2,4}$
- $S = Y(1,2)_1 \cup Y(1,2)_2 \cup Y(1,2)_3 \cup Y(1,2)_4$

For all cores except c_1 response time analysis is computed by equation *(15)*, since their situation is similar to *Y(i)*. Migratable tasks of core c_1 which previously were hosted by c_2 are migrated a second time. So, tasks belonging to $MIG_{1,2,3}$ and $MIG_{1,2,4}$ can incur in further release jitter and reduced deadline, which can be computed thanks to equation *(16)*.

If another core enters HI-crit mode, LO-crit tasks (migratable or not) that it was hosting needs to be descheduled to guarantee the feasibility of the system. So, for example, if core c_3 enters HI-crit mode:

- $Y(1,2,3)_1 = Y(1)_1 = LO_1 \cup HI_1$
- $Y(1,2,3)_2 = Y(1,2)_1 = LO_2 \cup HI_2$
- $Y(1,2,3)_3 = HI_3$
- $Y(1,2,3)_4 = Y(1,2)_4$
- $S = Y(1,2,3)_1 \cup Y(1,2,3)_2 \cup Y(1,2,3)_3 \cup Y(1,2,3)_4$

Since c_3's tasks are executing with their HI-crit budget, equation *(17)* must be used to compute response time. Other cores' behavior (and so the analysis) is analogous to the previous state.

Concluding, for each of the three models presented, if for a given taskset even a single deadline-miss occurs in the analysis, the taskset cannot be scheduled according to the model.

4.2 Models evaluation

The models presented in previous section have been compared to determine which one is the best choice. To do so, the authors of [2] have created a program which generates a random taskset compliant to three given parameters ("Percent", "Factor" and number of tasks in the taskset), pre-sorts it using descending criticality-aware criteria and eventually performs response time analysis (using all three models presented in previous section) to compute the scheduling performance of each model.

In the first phase, parameter "Percent" ("P") is used to determine the percentage of HI-crit task over the total number of tasks in the taskset, while parameter "Factor" ("f") indicates the multiplication factor applied to LO-crit WCETs to obtain the corresponding HI-crit value.

In the third phase, different tests have been conducted. The first one examined a taskset of 24 tasks, P = 0.5 (so 12 HI-crit tasks) and f = 2 (HI-crit budget twice as bigger as LO-crit one). Results are reported in *figure 5*. As can be seen, it is very evident that the semi-partitioned model in all its forms outperforms the partitioned one as utilization increases. In this regard, the black dotted lines highlight the difference of 31.58% of schedulability between Model 3 and non-migration approach around 3.7 utilization. Between the various semi-partitioned models, Model 3 performs better, even if the difference with Model 2 is quite marginal.

The other three tests are conducted varying each time one parameter, starting from criticality factor (*f*), continuing with *P* and finishing with the size of the taskset. In order to allowing the results to be represented with a 2D plot (since both the parameter and the schedulability percentage must be represented), weighted schedulability is used. From the second test, whose result is represented in *figure 6*, emerges how all semi-partitioned models have an increasing performance trend with increasing criticality factor, but Model 2 and Model 3 performs significantly better than Model 1. Third test results are shown in *figure 7*, from which it is clear how Model 3 performs better than Model 1 and Model 2, but the difference with the lat-

Figure 5: Result of the first test [2]

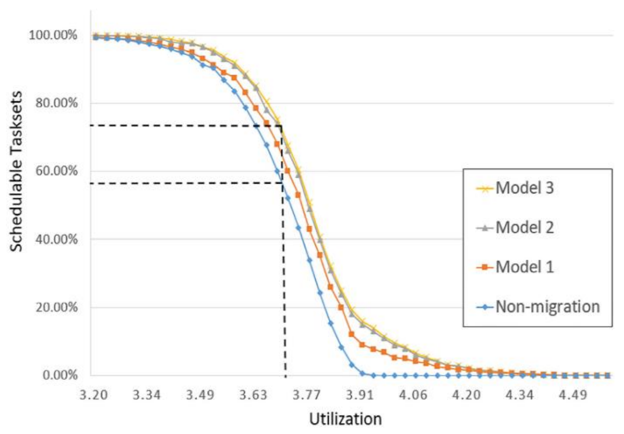


Figure 6: Result of the second test [2]

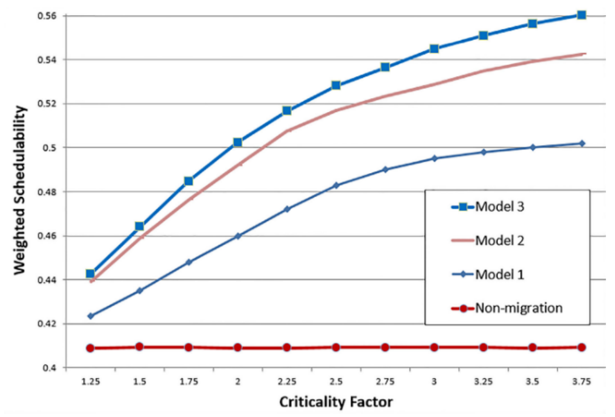
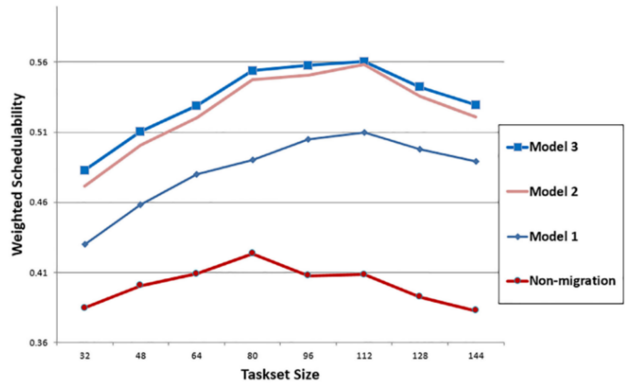
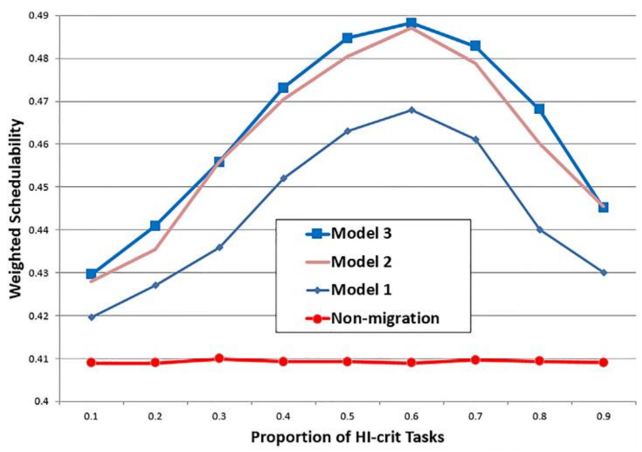


Figure 7: Result of the third test [2]



ter is not so marked. However, for all three models, schedulability increases until the relationship between high priority and low priority reaches 0.6, and then it starts to decrease (forming an inverted U shape). The last test (results in figure 8) shows how the semi-partitioned approach performs when varying the taskset size. In all its forms, semi-partitioned approach performs better than partitioned one. Internally, Model 3 has again the best performance, but the advantage over Model 2 is quite limited. It can be noted that results shown in figure 7 and 8 have an inverted U-shape. In figure 7 the reason is that, with a small taskset, for the same value of utilization, tasks are bigger, so it's difficult to find a core that has enough space to host migratable tasks, while with a large taskset both interference from higher priorities tasks and

Figure 8: Result of the fourth test [2]



tasks' release jitter increase. In figure 8, instead, what is varying is the proportion of HI-crit tasks. With a very small or a very high proportion of HI-crit task (so the lower and upper limit of the X-axis values) the behavior of the system is similar to a single-criticality one, so very few migrations happen: with a very small proportion of HI-crit tasks, the probability for a core to enter HI-crit mode is very low, while in case of pretty much only HI-crit tasks, the system behaves similar to a fully partitioned single criticality one. In both cases, the number of migrations is very limited, so the semi-partitioned algorithm cannot make a significant increase in performance. Instead, when the number of HI-crit and LO-crit tasks are similar, the performance improvement made by all the semi-partitioned models is more relevant.

In general, we saw how Model 3 perform better than the other two, but it is also the most complicated to implement and, since difference in performance with Model 2 seems rather small, the Model 2 is the best choice.

4.3 Beyond four cores

After the examination of the semi-partitioned model applied on a four-core environment, examples of how to apply the Model 2 on an eight-core and seven-core system will be shown. Eventually, the general algorithm will be presented.

On a four-core system, the boundary number, which we call n_{b4} was 2, so each core was paired with two neighbors and migratable tasks were split in two subsets (one for each paired core). So, because the boundary number for an eight-core platform is 3, each core should have three paired cores, and migratable tasks should be split in three subsets.

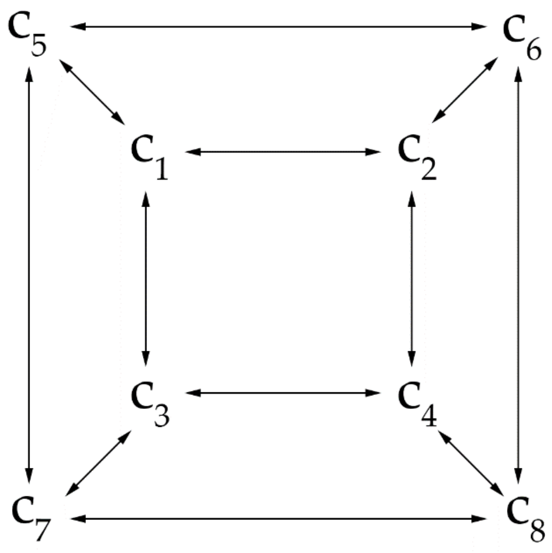
Figure 9 represents an example of this configuration. It should be noted that it appears as two four-core subsystem (c_1,c_2,c_3,c_4) and (c_5,c_6,c_7,c_8) with four more pairs. Each subsystem has a twin core in the other, with which it has a pair relationship. So, (c_1,c_5), (c_2,c_6), (c_3,c_7) and (c_4,c_8) are the twin pairs. This "trick" can be used to apply semi-partitioned model on 2^n-core mixed criticality system. Generalizing, from this reasoning the following cloning algorithm can be derived for a n-core platform with even number of cores:

1. Assume the existence of a k-core platform P ($c_1, .., c_k$) with $k = \lceil n/2 \rceil$, with $(n_{bk} * k) / 2$ different relations, where $n_{bk} = \lceil log2(k) \rceil$;
2. Create a twin clone system of k core P' ($c'_1, .., c'_k$) where each core is paired as the "original" k-core system. So, for example, if in P exists the pair (c_1,c_2), in P' the pair (c'_1, c'_2) must be created;
3. Pair the twin cores to each other. So $\forall i \in \{1..k\}$ create the pair (c_i, c'_i);
4. $\forall i \in \{1..k\}$ replace core c'_i with c_{i+k}

If the system has an odd number of cores, some of them might have one less paired core. If for example we take a seven-core system (n_{b7} = 3), it's impossible to pair each core with exactly other two in such a way that each core is connected to at most other three. An example of a seven-core configuration is given by *figure 10*. Cores are paired in a way similar to the eight-core system, except that core c_8 and its relations are deleted, and for each two of them deleted one new relation is added between two different cores which have a number of relation smaller than n_{b7} (on the example, relation between cores c_6 and c_7 is added). Again, from these observations it is possible to derive the general algorithm for a n-core platform with an odd number of cores:

1. Execute the previous algorithm;
2. Let's define $q = k*2 - n$ as the number of cores in "overflow" (cores that are present in the model but not present in the actual n-core system);
3. Remove all cores $c_j : n<j\leq n+q$ and their pair relationships;
4. For each two relationship removed, create a new pair relationship between two different cores $c_a, c_b : a,b\leq n$ which have less than $n_{bn} - 1$ associated cores.

Figure 9: example of eight-core configuration



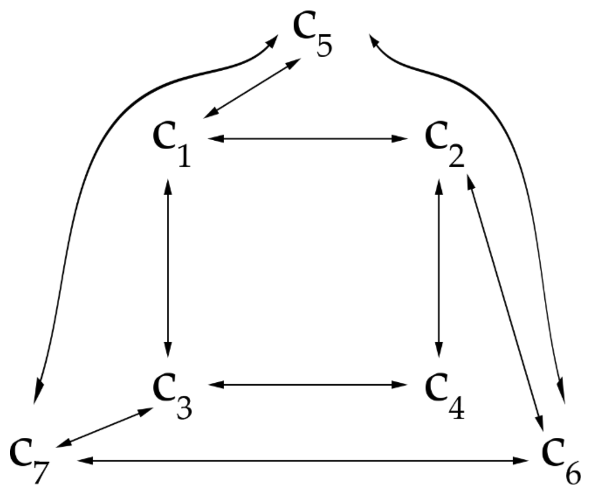
These two algorithms can be used recursively to obtain the solution to the pairing problem for a generic n-core platform, with the dual-core semi-partitioned model described in [3] and four-core semi-partitioned model described in [2] as base cases.

Figure 10: example of a seven-core configuration

In conclusion, the semi-partitioned model for an n-core system is created as follows:

- Apply the generic pairing algorithm described above to pair each core with $n_b = \lceil log2(n) \rceil$ other cores;
- If necessary, mark some tasks as migratable using a combination of Semi2WF and Semi2FF described in [3] (and in *section 3.4*);
- If a core enters HI-crit mode and the total number of cores in HI-crit mode is less or equal than n_b, its migratable tasks are redistributed "equally" to the other paired LO-crit cores and no task is descheduled (all deadlines are met).
- If a core enters HI-crit mode and the total number of cores in HI-crit mode greater than n_b, all LO-crit tasks running on are descheduled, but it's guaranteed that HI-crit tasks will met their deadlines.

5 Discussion



5.1 General considerations

After having reviewed all the concepts covered by the paper, it's time to make some considerations about it. First, it's fair to say that it represents a glimmer of light in the context of multicore real time systems. Mixed Criticality System is a very powerful concept that allows different components (so, different subsystems) to be merged into a single system, allowing a simplification of the architecture. However, single-core MCS are not so effective, because in case of increase in global criticality level, lots of tasks might be abandoned. Since multi-core permits a workload redistribution, multi-core MCS have the potential to drop less tasks if criticality change increment happens. This was the idea behind the study presented by the paper, which, at least in theory, provided us with a very promising model.

An important merit to be attributed to this work is certainly to have demonstrated that it is possible to obtain good performance in terms of schedulability by linking each core with a limited number of others (n_b). This allows to simplify a lot the design and the analysis of the system without having a sensible performance drawback.

However, an imperfection is present in the definition of the computational steps needed to compute value n_b for a *n*-core system. Since f(m,n) (defined in *equation 11*) is a binomial distribution, its values cannot be negative and, since F(X,n) (*equation 12*) is a sum of n-X-1 values of f(m,n), its value decrease as X increases. This implies that n_b has to be redefined as "the smallest number X which meets the tolerance standard ($F(X,n) \leq p_{tol}$)". Also, when n = 4, the correct value of n_b is 2.62e-12 and not 2.62e-13. However, this in no way compromises the robustness of the model analysis or validity presented.

At a higher level, one problem could be represented by tasks' migration overhead. In fact, even if this model considers reduced deadlines and increment in release jitter, migrate a task is a costly operation. [9] did some simulations inducing a migration overhead proportional to the WCET and noticed that even with a migration overhead of 10% WCET, this model performs better than a fully partitioned one. However, migration overheads are difficult to predict (and can fluctuate), so further simulations and tests should be carried out.

Also, it would have been interesting to compare performance of a system with an odd number of core (*k*-core) with the ones of a *(k-1)*-core system. In fact, since in platform with odd cores number there is at least one of them which has less paired cores than the other, performance gain might be affected.

A limitation factor of this model might be the presence of only two criticality levels. Burns himself during a conference in 2015 stated that the criticality levels should be five for the solution to be applicable. About that, [9] presents a variation of the model for a triple criticality system, where tasks and core can have three criticality level (HI, MID and LO). However, despite performance of the model seems good, the analysis is very complicated. Furthermore, to simplify the analyzability, in that model tasks can have only two WCETs, so they can only take two levels of criticality (HI and LO, HI and MID or MID and LO). This fact can be interpreted as an indication of the difficulty of scalability of the number of criticality levels of this model.

Another possible critical issue might be in the response-time analysis. In fact, at least in steady state (X) tasks are fully partitioned and the analysis considers only tasks on the same core to compute the response time. However, have a *n*-core system is different than having *n* single-core system because of shared resources (like buffers). So, albeit marginally, tasks might be influenced by other ones on different cores.

5.2 Migration overhead estimation

[3] concluded by saying that further analysis of the semi-partitioned model should take migration overhead into account. So, it makes sense to try to understand if indeed there is an overhead when migrating a task to other CPU and, if so, what is causing it and how much is it.

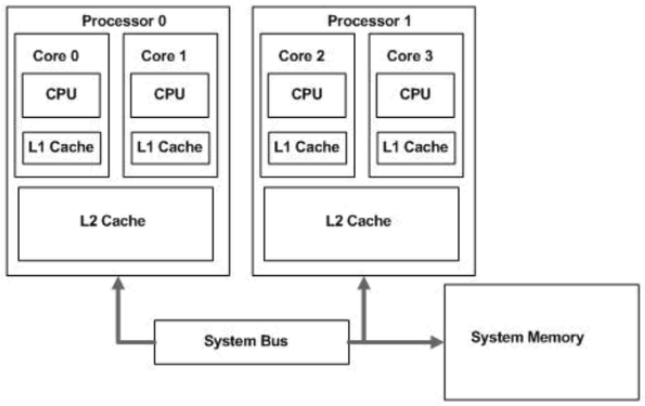
5.2.1 Migration overhead in theory

In modern systems there is a component that acts as an intermediary between RAM and the CPU: the cache memory. The latter is faster and closer to the CPU than RAM (but smaller), and its purpose is to store data and instruction frequently used by the running program in order to avoid accessing to RAM as much as possible. In fact, RAM access is via a bus which, in case of a multi-core system, is shared among cores.

Cache memory is divided into various levels (L1, L2, L3) and the way they work is strongly dependent to the CPU microarchitecture. In particular:

- L1 (instruction and data), which is internal to each core, so no bus access is needed. This is the fastest and the smallest type of cache;
- L2 (slower and bigger than L1) which, based on implementation, is shared among all cores, shared among a subset of cores (like in Intel Xeon Phi [10]) or private to each core;
- L3 (bigger and slower than L2), which is usually shared among cores. In some architecture each core can access (and write onto) every portion of L3 cache, while some others have different fraction of the L3 cache allocated to every core, which means that a core cannot directly access and modify other cores' cached data.

Figure 11: example of microarchitecture using shared L2 cache among group of cores [11]



So, if a task's job is migrating on another core, it might lose a part of the performance boost given by the cache mechanism. But the exact quantity of lost boost is strongly platform dependent. Also, if the CPU architecture is like the one on *figure 11*, the migration cost can be different based on the target core. In fact, migrating from *core 0* to *core 1* might be less expansive compared to migrating from *core 0* to *core 2*, because of the shared L2 cache.

Another hypothesis that derives from what was written above is that migration overhead could depend on RAM speed: a faster RAM could mitigate the loss of the cache boost.

5.2.1 Experiment configuration

The above statements give rise to the following three observations. Firstly, since (a part of) migration overhead seems to be generated by additional RAM access, a benchmark to estimate it must include memory operations. Moreover, to obtain precise measurements, the benchmark must run in an environment with less executing process as possible to avoid interference. Eventually, it must consider that migration costs may vary depending on the target core.

So, the benchmark itself is composed of a timed computation, from now on called C(n), where *n* is the complexity (this parameter will be analyzed later), iterated 10 000 times. For each even iteration, before performing the calculation, the process is migrated. So, given *i* the current iteration:

- If *i* is even, the detected time will be composed by *migration overhead* + *time needed to compute C(n)*;
- If *i* is odd, the detected time will be composed by only the *time needed to compute C(n)*.

The computation C(n) consists of some operations on an integer array of size *n*. More in depth, the array is scrolled from index *0* to *n-1* and to the value of the current cell are added values taken from two random cells. The random elements are present in order to avoid exploration orders that a priori favor or penalize the performance boost given by the exploitation of the cache in odd iterations.

At the implementation level, the benchmark is made of a C program running on Minimal Linux Live, a Linux distro that has nothing else than kernel and terminal, without network support. In this way, the benchmark process should not suffer too much from other processes interference.

The migration itself is done by calling *sched_setaffinity* (part of the standard C Linux scheduler library) and *sleep* methods. The latter is needed because *sched_setaffinity* only tells the scheduler on which core(s) the current process should execute, but it does not perform the migration immediately. Instead, it waits until the current process is preempted, and this is what the *sleep* method is for.

The benchmark is ran multiple times, each one with a different value of *n* starting from n=100, so the array is 0.4KB, up to n=1000000, so the array is 4MB, each run increasing n by 100. Those values, although they are chosen in a totally arbitrary way, should allow to have a set of scenarios wide enough to draw some conclusions on the migration overhead.

Finally, the execution times of each iteration are taken using the standard *clock_gettime*, knowingly that results obtained could contain inaccuracy. However, the benchmark considers the mean execution time (on 10 000 tests), so the impact of the *clock_gettime* inaccuracy should be partially mitigated. In section 5.2.3 (Future Works) a solution to obtain more accurate readings will be discussed.

5.2.3 Results

The benchmark was executed on two different computers three times (results shown are the mean of all three executions):

- **PC1**, with an Intel i7 4770k (Haswell) quad core processor, 8MB "SmartCache" (64KB of L1 and 256KB of L2 cache per core, and 8MB of L3 cache shared among cores), disabled "HyperThread" (so no virtual cores were active) and 16GB of DDR3 800/1333/2133 MHz RAM;
- **PC2**, with an Intel Pentium D 930 (Presler) dual core, 16KB of L1 and 2MB of L2 cache per core (a total of 4MB), and 2GB of DDR2 667MHz RAM.

Figure 12: Migration overhead on PC1

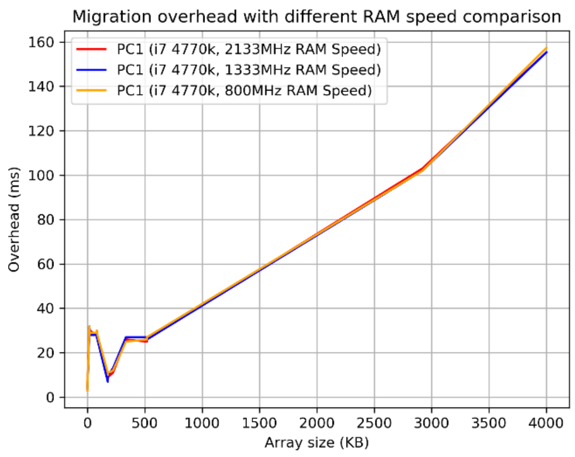
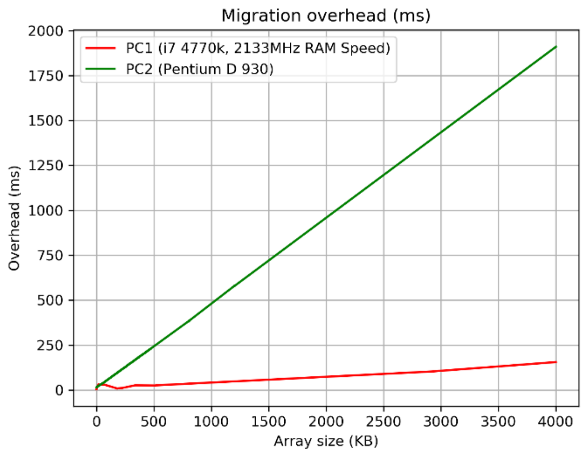


Figure 13: Comparison between PC1 and PC2



Results obtained are shown in *figure 12* and *figure 13*. *Figure 12* compares the migration overhead obtained on the same computer but with different ram speed. It's evident that RAM speed, at least on PC1, does not affect the migration overhead, although it leads to an increase in C(n) total execution time. Also, a "strange" behavior can be noted for value of array size smaller than about 500KB: migration overhead rapidly increase, then it stabilizes, then it decreases (until about 240KB) and suddenly increases (forming a sort of V). The increase in migration cost at the beginning and after 240KB could be due to running out of L1 and L2 cache respectively. Also, at around 2900KB, the overhead starts to rise faster. This can be because, with Intel SmartCache technology, the L3 cache is shared among all core in the sense that each one has access to a slice of the cache (the size of the slice is changed dynamically). So, maybe the maximum free amount of cache that a core has access, considering that there are also other system processes running, is about 3MB. Indeed, this aspect needs to be verified running the benchmark on other Intel CPUs that use this technology.

Figure 13 compares the results of the two test computers. It's impressive how PC2 overhead increases linearly and more rapidly compared to PC1. For example, when *n=1000000* (so array size is 4MB), the average overhead detected over 10 000 runs is 1910ms for PC2 (and the average non-migrating execution time detected was 130300ms), while the corresponding value for PC1 was about 155ms (with an average non-migrating execution time of 24255ms). Contextualizing, PC1 has a % overhead of 0.64%, while PC2 about 1.4%. However, for "quicker" tasks (so for small values of *n*), the migration costs are greater in percentage of the execution time. For example, if n=100, the percentage migration overhead is about 167%, while for PC2 the same value is the 127% of the non-migrating counterpart. Those numbers show that the migration cost of a task strongly depends on the computer on which the software is executing. Also, in some cases (like PC2), the overhead might be predictable with a discrete amount of precision, while in other contexts (like PC1 for small array size) a right prediction might be harder.

5.2.4 Final considerations and future works

The experiment certainly shows that the migration cost can be very costly, and it depends on many factors. In [9] the author said that, for an overhead of 10% or less of the task execution time, the dual-core semi-partitioned model outperforms the non-migrating approach. However, if a system is composed predominantly of tasks with small execution time, the migration cost (in percentage) can be 10 times greater than what the author of [9] was expecting, potentially undermining the validity of the model.

Since the tasks migration costs are strongly system-dependent, and one certain limit of this work was the ability to experiment on only two computers (which maybe haven't the most common architecture on real-time environment), it's desirable that future work will try to benchmark other systems with different characteristics.

Another weakness of this work is how the time measurements are done. In fact, there's a way to make them more precise: the use of PAPI library. This is a C/Fortran library which allows use the hardware performance counters available on the processor. It has a method, *PAPI_get_real_usec*, which allows to get a reliable timestamp and it could replace the usage of *clock_gettime* in the benchmark. In this case, PAPI was not used due to hardware incompatibility: it does not support Intel Presler architecture [12], which is the one used in PC2.

References

[1] A. Burns, R. I. Davis, "Mixed Criticality Systems – A review (thirteenth edition)"

[2] Xu, H., Burns, A., 2019. Semi-partitioned model for mixed criticality system. In: Journal of Systems and Software Volume 150, April 2019, Pages 51-63, Networks Systems. ACM, pp. 257–266.

[3] H. Xu, A. Burns, Semi-partitioned model for dual-core mixed criticality system, Proceedings of the 23rd International Conference on Real Time and Networks Systems, ACM (2015), pp. 257-266

[4] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance", Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International, IEEE (2007), pp. 239-243

[5] N. C. Audsley, "Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times," Technical Report YCS 164, Department of Computer Science, University of York, November 1991

[6] Baruah, S.K., Burns, A., Davis, R.I., 2011. Response-time analysis for mixed criticality systems. In: Real-Time Systems Symposium. IEEE, pp. 34–43

[7] Joseph M, Pandya P (1986) Finding response times in a real-time system. Comput J 29(5):390–395

[8] Dorin, F., Richard, P., Richard, M., Goossens, J., 2010. "Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities." Real-Time Syst. 46 (3), 305–331.

[9] Xu, H., 2017. A semi-partitioned model for scheduling mixed-criticality multi–core systems. Department of Computer Science, University of York, UK Ph.D. thesis.

[10] Intel Xeon Phi Architecture Overview, https://bit.ly/2qeqVPY

[11] "How is L2 cache shared between different cores in a CPU?" https://bit.ly/2qpyZ05

[12] PAPI supported architectures, https://bit.ly/33eVq5E. last visit on 2019-11-23