Priority Assignment and Application Mapping for Many-Cores Using a Limited Migrative Model

Borislav Nikolic
Konstantinos Bletsas
Stefan M. Petters

CISTER-TR-140204
Version:
Date: 2/11/2014

Priority Assignment and Application Mapping for Many-Cores Using a Limited Migrative Model

Borislav Nikolic, Konstantinos Bletsas, Stefan M. Petters

CISTER Research Unit

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: borni@isep.ipp.pt, ksbs@isep.ipp.pt, smp@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Most of present many-core scheduling and application mapping approaches rely on strict assumptions, e.g. uninterruptible availability of all cores, which significantly limits their scope of application. In this paper we consider a Limited Migrative Model (LMM). LMM is the novel approach in the real-time embedded domain, which gives the possibility to efficiently analyse scenarios that involve occasional core shutdowns for various reasons (e.g. energy and thermal management, load balancing, fault tolerance), thus allowing to embrace the full potential and flexibility of many-core platforms. The contributions of this work are as follows. First, we propose the classification of schedulability guarantees for LMM. Then, we introduce various priority assignment and application mapping techniques and subsequently study their impacts on (i) derived schedulability guarantees for critical and time-constrained applications, and (ii) the performance of best-effort workload. The experiments show that LMM is a valuable approach, even when assuming an uninterruptible availability of all cores, while it exhibits clear benefits in cases where, due to necessity to perform occasional core shutdowns, a higher amount of flexibility is desired.

Priority Assignment and Application Mapping for Many-Cores Using a Limited Migrative Model

Borislav Nikolić, Konstantinos Bletsas and Stefan M. Petters
CISTER/INESC-TEC, ISEP
Polytechnic Institute of Porto, Portugal
Email: {borni, ksbs, smp}@isep.ipp.pt

***Abstract*—Most of present many-core scheduling and application mapping approaches rely on strict assumptions, e.g. uninterruptible availability of all cores, which significantly limits their scope of application. In this paper we consider a *Limited Migrative Model* (LMM). LMM is the novel approach in the real-time embedded domain, which gives the possibility to efficiently analyse scenarios that involve occasional core shutdowns for various reasons (e.g. energy and thermal management, load balancing, fault tolerance), thus allowing to embrace the full potential and flexibility of many-core platforms. The contributions of this work are as follows. First, we propose the classification of schedulability guarantees for LMM. Then, we introduce various priority assignment and application mapping techniques and subsequently study their impacts on (i) derived schedulability guarantees for critical and time-constrained applications, and (ii) the performance of best-effort workload. The experiments show that LMM is a valuable approach, even when assuming an uninterruptible availability of all cores, while it exhibits clear benefits in cases where, due to necessity to perform occasional core shutdowns, a higher amount of flexibility is desired.**

I. Introduction

The miniaturisation process in the semiconductor technology reached the stage where further processing power enhancements related to single-cores are no longer affordable [1]. In order to continue the progress of computational devices, chip manufacturers took a design paradigm shift [2], [3], and started increasing the number of interconnected cores within a single chip. Nowadays, platforms containing several cores (*multi-cores*) and more than a dozen of cores (*many-cores*) have become commonplace in many scientific areas, e.g. high performance computing, while they are an emerging technology in others like real-time embedded systems.

Many-cores offer several beneficial possibilities, for instance, to integrate new functionalities and enhance existing ones, or to do energy/thermal management. However, issues such as unpredictability, scalability, the need for load balancing and core shutdowns are some of the challenges which have to be addressed before many-cores can be integrated into the real-time embedded domain. The existing state-of-the-art approaches in many-core scheduling and application mapping theories are not efficient in addressing all of the aforementioned challenges though, and this significantly limits their practical use. For instance, most approaches rely on strict assumptions, e.g. uninterruptible availability of all cores, which renders them inapplicable in scenarios where occasional core shutdowns are desirable and/or necessary for various reasons, such as energy/thermal management and fault tolerance. Partly, exceptions are two recent works of Baruah and Guo [4] and French et al. [5], where the authors discuss the scheduling of the mixed criticality workload upon unreliable processors, characterised by two speeds: normal and degraded.

The *Limited Migrative Model* [6] (LMM) is based on the foundations of the *multi-kernel* paradigm [7] – a novel OS design [7], [8], [9] and a promising step towards scalable and predictable many-cores. Moreover, LMM offers the possibility to efficiently analyse scenarios that involve occasional core shutdowns, which distinguishes it from the state-of-the-art approaches. Yet, the schedulability analysis of LMM is still an unexplored topic.

Contribution: This paper focuses on the schedulability aspects of LMM. First, we present the classification of schedulability guarantees (Section IV-B). Then, we propose a heuristic-based approach for priority assignment and application mapping, assuming that the workload is comprised of applications with different criticality levels[1] (Section IV-C). The objective is to derive schedulability guarantees for critical and time-constrained applications, and also to optimise the performance and establish a sense of fairness across best-effort applications (Section IV-A). Through simulations we study the impact of various mapping and priority assignment choices on derived guarantees, as well as on the overall system runtime behaviour, observed under different conditions (Section V).

II. Related work

Application mapping on many-core platforms has been one of the most investigated topics in the last decade [10]. Therefore, in this section we cover only works which are relevant to the real-time domain. Broadly classified, all state-of-the-art approaches fall into two categories, namely *non-migrative approaches* and *migrative approaches*. We firstly introduce these categories, then position LMM in that context, and finally elaborate on how this paper differs.

Non-migrative approaches (in the multi-core scheduling theory also known as *fully partitioned approaches*) assume that the workload is, at design time, statically assigned to different intellectual properties (e.g. CPU cores, DSP cores). Arranging the execution is the responsibility of single-core scheduler instances [11], that run independently on each core. Besides analysing the workload distribution with the objective of maximising the number of scheduled applications (bin-packing theory), these approaches also evaluate the position

[1]The approaches established in the real-time domain which address the schedulability of the mixed-criticality workload have a different perception of the problem then we do. The former provide schedulability analyses which take into account possible variations in worst-case execution times, while in our approach worst-case execution times do not change, but applications require different types of schedulability guarantees.

of cores within the chip with different objectives: to minimise the network power consumption [12], to minimise the average communication delay assuming bandwidth constraints [13], to derive a thermal-aware placement [14].

Migrative approaches are further classified into *semi-partitioned approaches* and *global approaches*. The first group (e.g. [15], [16]) assumes a static allocation, such that an application always executes on the same core (or cores if it migrates), where migrative applications also obey to static, offline-based decisions, i.e. always have to execute prescribed fractions of their work on assigned cores in a given order. Conversely, approaches termed as global (e.g. *Global FP* [17] and *Global EDF* [18]) comprise the possibility to migrate to, and execute on every core within the system.

As is evident, fully and semi-partitioned approaches are rigid, do not have the flexibility to support runtime load balancing and are not resilient to core shutdowns. Conversely, by allowing every application to execute on every core, global approaches inherently support load balancing, however, this amount of flexibility comes at a price – the necessity to maintain global structures (e.g. ready-queue) poses serious challenges when attempting practical implementations [19].

Gujarati et al. [20] introduced a model called APA, where each application may execute on an arbitrary number of cores, based on processor affinities. Baruah and Brandenburg [21] proposed the feasibility analysis for that model.

LMM [6] is an approach that has similarities with the APA model, in a sense that each application can execute only on a pre-selected subset of cores, which are decided at design time. Yet, in APA affinities are considered to be inputs, while in LMM the system designer decides the candidate cores. At runtime, an application can execute on any of the candidate cores, while migrations occur only on job boundaries (no job-level migrations). The greatest distinction of LMM from all the existing approaches is that release/migration decisions are explicitly detached from scheduling decisions, and are made by the applications *themselves*, not by scheduler entities. That is, the application decides on which core it will release its job, while a local kernel on that core is responsible to schedule it in a single-core fashion. This contributes to the scalability of the approach, and yet gives the flexibility to perform runtime load balancing. More details about LMM are given in Section III.

Assuming LMM, we already estimated the overheads of workload migrations [6], studied the worst-case communication delays [22] and studied the worst-case memory traffic delays [23]. In this paper we focus on the schedulability aspects of LMM, which is the last outstanding part that is necessary before developing a unified analysis for LMM.

III. Model

A. Hardware

The platform under consideration is a many-core system comprised of n homogeneous cores: $\mathcal{M} = \{\mathcal{M}_1, ..., \mathcal{M}_n\}$. Practical examples are Single-Chip-Cloud Computer manufactured by Intel [2] and the Tilera family of processors [3]. As already mentioned (Section II), the state-of-the art approaches have a common underlining assumption that the system always operates with the full capacity (all cores are always operational). Conversely, in this paper occasional core shutdowns

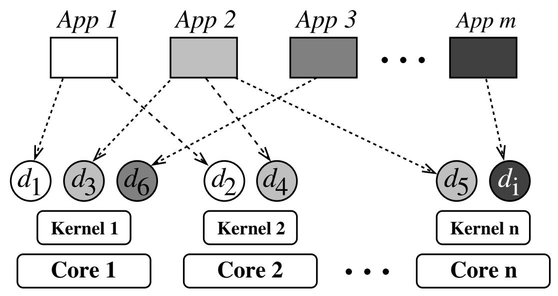


Fig. 1: Architectural structure of LMM

are allowed for various beneficial reasons, e.g. power/thermal management. Once a core is selected for shutting down, it will continue to execute the already assigned workload, however, it will reject new job releases. When the last previously assigned job completes, the core will become temporarily unavailable (e.g. sleep interval, cooling period). Depending on the purpose, a system designer might choose to apply more/less aggressive load balancing strategies, involving more/less frequent core shutdowns. As a means to control that, we introduce a parameter K, which symbolises the maximum number of concurrent core shutdowns. In this work we do not elaborate on the value of K, but only assume that it has already been specified.

B. Software Layers in LMM

In LMM, each core runs an independent kernel instance. Kernels are mutually connected, provide the basic communication infrastructure and form the multi-kernel entity. Each kernel exposes some of its functionalities to applications located on its core via system calls. Applications invoke system calls in order to communicate with other applications located on the same or other cores. Each kernel runs its own single-core fixed-priority scheduler with preemptions [11]. Kernels are also responsible for shutting down/rebooting the cores and performing necessary workload migrations. Practical examples of multi-kernel OSs are *Barrelfish* [7], *fos* [8] and *Quest-V* [9].

Each application can execute only on the respective subset of cores, selected at design time. The process of deciding the candidate cores for every application, we term *mapping* (one of the main contributions of this paper). On each of the selected cores, a copy of the executable code of that application exists, encapsulated within an entity called *dispatcher*. Dispatchers of the same application (each located on a different core) communicate with each other via agreement protocols [6] and derive release/migration decisions, i.e. will the migration occur, and if so, which core (dispatcher) will accommodate the next job. An elected dispatcher releases the next job on its core on behalf of the entire application. Once the job execution completes, the agreement protocol is performed again, so as to elect the core (dispatcher) for the next job release, and so on. Figure 1 illustrates the assumed model.

Notice, that in LMM each kernel performs the scheduling in a single-core fashion, similar to fully partitioned approaches. Yet, LMM supports application migrations, similar to global approaches. Thus, it appears that LMM embraces the favourable aspects from both concepts (scalability from the former and flexibility from the later). This possibility arises due to a very distinctive way the release/migration decisions are made – via agreement protocols, as briefly described in the previous paragraph. More details about LMM and agreement protocols are available in the following works [6], [22], [23].

The execution workload is represented with a sporadic application-set $\mathcal{A} = \{a_1, ..., a_z\}$, where each application $a_i\langle P_i, T_i, C_i, \mathcal{D}_i\rangle$ is characterised by its default priority – P_i, a minimum inter-arrival period – T_i, a required execution time – C_i and a set of dispatchers – $\mathcal{D}_i$. Each dispatcher d_j of an application a_i has an individual priority P_j, which is less than or equal to the default priority of its application, but not greater than it, i.e. $\forall d_j \in \mathcal{D}_i : P_j \leq P_i$. The priority assignment upon dispatchers is also one of the main contributions of this paper, as summarised in Section IV-A. The elected dispatcher d_j releases a job J_j on its core, where the job inherits dispatcher's priority – P_j. Jobs have implicit deadlines, a job released at time instant t, needs to execute for C_i time units before $t+T_i$. If it fails to do so, it has *missed its deadline*. Note, allowing dispatchers of the same application to have different priorities implies that job priorities of one application are not necessarily constant, but depend on dispatchers which are elected to release them. This is also a novel concept in the real-time domain.

C. Workload Classification

Every application in the system can be classified into one of the categories: *Safety-Critical Applications* (SCA), *Real-Time Applications* (RTA) and *Best-Effort Applications* (BEA). SCA are considered to be the highest-priority workload in the whole system. Therefore, strong guarantees regarding their schedulability requirements should be provided, ensuring that no missed deadlines of SCA will occur, even under circumstances that involve core shutdowns. RTA represent the medium-priority workload, and also require schedulability guarantees. However, the guarantees only need to hold when the system is working with the full capacity (no core shutdowns). Finally, BEA present the lowest-priority workload in the system. BEA can tolerate missed deadlines, however, that has an impact on the quality of service. Hence, we do not focus on deriving schedulability guarantees for BEA, but instead try to establish a notion of fairness, e.g. by spreading missed deadlines as evenly as possible across all BEA. This decision is motivated by the fact that, in many practical scenarios, maintaining all non-critical functionalities with reduced quality is more desirable than cancelling some of them (e.g. multimedia applications). The assumed workload classification and respectively posed requirements are inspired by the existing workload classification which is well established in the real-time embedded area [24].

IV. Proposed approach

A. Work Objectives

The objective of this work can be summarised with the following statement. Given the application-set $\mathcal{A}$, the platform $\mathcal{M}$ and the maximum allowed number of concurrent core shutdowns K, assign priorities to dispatchers and map them onto the platform ($\mathcal{A} \rightarrow \mathcal{M}$), such that:

- No missed deadline of SCA occurs, assuming that the system exhibits at most K concurrent core shutdowns.
- No missed deadline or RTA occurs when the system is working with the full capacity (no core shutdowns).
- The ratio of missed BEA deadlines is spread across all BEA as evenly as possible.

As already known, the application mapping for many-cores is an NP-Hard problem [12], hence searching for the optimal solution is, in most cases, prohibitively expensive. Nonetheless, even with infinite computational capacities, in this particular case the optimal solution could not be found at design time, since the "optimality" directly depends on core shutdown decisions, which are made at runtime. Therefore, we explore an alternative, heuristic-based approach, with the objective of finding a sub-optimal solution of acceptable quality.

B. Schedulability

Since different application classes have different schedulability requirements, in this section we focus on the classification of schedulability guarantees, which will be subsequently used when assigning priorities and while mapping.

1) **Offline Schedulability Guarantees:** If a dispatcher of an application passes an *offline schedulability test*, performed at design time, it means that the application can execute on its core as long as it is operational and will never miss a deadline due to the other workload residing on the same core. The test is performed by computing the worst-case response time for a fully preemptive fixed-priority system [11], treating that core as an independent single-core device and assuming the workload that can be generated by all dispatchers existing on that core.

$$R_i^{n+1} = C_i + \sum_{\forall d_j \in \mathcal{M}_x : P_j > P_i} \left\lceil \frac{R_i^n}{T_j} \right\rceil \times C_j \tag{1}$$

The worst-case response time of a job released by a dispatcher d_i on the core $\mathcal{M}_x$ consists of two terms (Equation 1). The first is the execution time of that job – C_i. Additionally, any higher priority dispatcher d_j residing on the same core $\mathcal{M}_x$ may release a job which can preempt the execution of the job under analysis, therefore from every such dispatcher the maximum possible interference has to be computed (the second term in Equation 1). Note, that Equation 1 has a recursive notion, thus is solved iteratively. If the computed value is less than or equal to the minimum inter-arrival period, i.e. $R_i^{n+1} \leq T_i$, an application is considered *offline schedulable* with its dispatcher d_i on the core $\mathcal{M}_x$.

2) **Online Schedulability Test:** If a dispatcher is not offline schedulable, it does not mean that a job of its application can never execute on that core without missing a deadline. It only implies that the ability to do so depends on the higher priority workload residing on that core at the moment of observation, i.e. which of the higher priority dispatchers are elected by their respective applications to release the jobs on their cores. Thus, in order to determine whether it can release a job and provide the guarantees that the deadline will not be missed, a dispatcher has to perform an online response time test (Equation 2).

$$R_i^{n+1} = C_i + \sum_{\forall J_j \in RQ(\mathcal{M}_x) : P_j > P_i} \widetilde{C_j} + \sum_{\forall d_k \in \mathcal{M}_x : P_k > P_i} \left\lceil \frac{t + R_i^n - r_k}{T_k} \right\rceil \times C_k \tag{2}$$

The response time consists of the execution time C_i, augmented by the higher priority content of the ready-queue $RQ(\mathcal{M}_x)$, where $\widetilde{C_j} \leq C_j$ stands for the remaining execution time of every higher-priority ready-queue job J_j, observed at the time instant t. A conservative assumption is that all higher priority dispatchers will be elected for their future releases, thus, the potential interference has to be considered from every such dispatcher d_k. It is calculated as the maximum number of occurrences in the interval between the next job release of d_k occurring at $r_k \geq t$, and the absolute response time $t + R_i^n$.

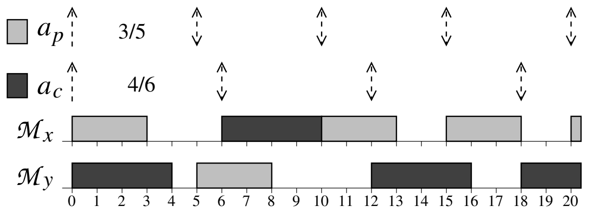


Fig. 2: Example of semi-schedulability

Notice, that r_k and t are absolute values, thus are represented with lower-case characters. Equation 2 is also recursive, and if a computed value is less than or equal to the minimum inter-arrival period ($R_i^{n+1} \leq T_i$), the application is *online schedulable* at the instant t on the core of its dispatcher d_i.

By passing the online schedulability test, the application gets guarantees from the system that its next job, released at t, can be executed on the core of its dispatcher d_i without missing a deadline. The guarantees are valid only for the next job, once its execution is completed the test should be performed again. At this stage, a reader may raise two very valid concerns:

- Is it practical to perform a recursive computation online?
- Is it practical to manage remaining execution times?

We address these concerns in the following way. For cases where solving Equation 2 might be prohibitively expensive, we propose a modification of the online schedulability test. A test is agnostic wits respect to remaining execution times and completes within a single iteration (Equation 3).

$$R_i = C_i + \sum_{\forall J_s \in RQ(\mathcal{M}_x):P_s>P_i} min\{C_s, r_s - T_s + R_s - t\} + \sum_{\forall J_u \in RQ(\mathcal{M}_x):P_u>P_i} C_u + \sum_{\forall d_k \in \mathcal{M}_x:P_k>P_i} \left\lceil \frac{t + T_i - r_k}{T_k} \right\rceil \times C_k \quad (3)$$

When compared to Equation 2, the first term is the same, while the computation of the interference due to future releases (the last term) is not recursive any more and is computed within a single iteration for the interval T_i. Conversely, the ready-queue content (the second and the third term) is computed differently. First, for each higher-priority job J_s, released at $r_s - T_s$, which has either offline or online guarantees that its execution will complete until $r_s - T_s + R_s$, we compute its remaining execution time by finding the smaller between (i) its total execution time and (ii) the difference between its worst-case response time, expressed in absolute values, and the current time instant t. For each higher-priority job J_u, which has been released without guarantees, we have to assume that its remaining execution time is equal to its worst-case execution time, because it *may* miss a deadline. Note, intermediate approaches are also possible, e.g. the remaining execution times are available, but the computation should be performed within a single iteration, and vice versa. For such approaches online schedulability tests can be derived by modifying Equations 2-3, which we do not cover, due to space constraints. In Section V we evaluate how the knowledge about the remaining execution times and the allowed number of iterations influence the efficiency of online schedulability tests.

3) **Semi-Schedulability Guarantees:** Even though multiple dispatchers of an application might be offline schedulable, each job will be executed by only one of them (an elected dispatcher), thus leaving the other dispatchers idle. We exploit this fact and recognise another form of guarantees, termed *semi-schedulability*. An application is semi-schedulable if none of its dispatchers is offline schedulable, however, at any time instant at least one is online-schedulable.

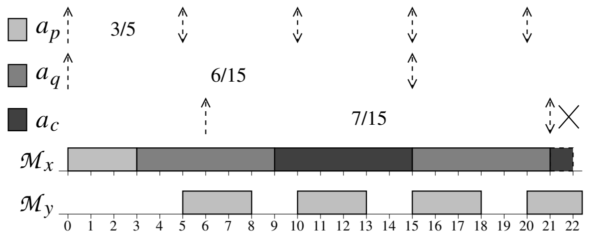


Fig. 3: Non-synchronised semi-schedulability

Consider two applications a_p and a_c, where the first one has a higher priority ($P_p > P_c$), and each dispatcher inherits a priority of a respective application, i.e. $\forall d_i \in \mathcal{D}_p : P_i = P_p$ and $\forall d_j \in \mathcal{D}_c : P_j = P_c$. Let a_p and a_c share two common cores - $\mathcal{M}_x$ and $\mathcal{M}_y$, and let a_p be offline schedulable on both of them. Additionally, assume that a_c is not offline schedulable on either, but the absence of a_p would make a_c offline schedulable on both $\mathcal{M}_x$ and $\mathcal{M}_y$. Since a_p can execute only on one core at any time instant, say $\mathcal{M}_x$, a core $\mathcal{M}_y$ might be able to accommodate a_c. And vice versa. Figure 2 illustrates one such example. The fractions represent execution times and minimum inter-arrival periods (C/T). Formally:

Definition 1. *An application a_c is considered semi-schedulable with respect to a higher priority application a_p, if a_c and a_p share at least two common cores, on which a_p is offline schedulable and a_c is not, but the absence of a_p would make a_c offline schedulable on both of them. a_c is called a semi-schedulable child and a_p is called a semi-schedulable parent.*

Semi-schedulability provides guarantees that, even though an application does not have offline schedulable dispatchers, at any time instant there will be at least one which can claim online schedulability, thus will be able to safely execute without missing a deadline. Semi-schedulability creates a co-scheduling parent-child relationship, which implies that, at certain points during runtime, some dispatchers may be prevented from being elected. Hence, semi-schedulable applications will have less flexibility when electing dispatchers for their next job releases. However, this does not violate the statement from Section II that the scheduling decision of each application is made by the application itself. This is possible because all the information that is necessary for deriving a release/migration decision (e.g. the workload state on candidate cores) is available to the dispatchers from their local kernels and is communicated during the agreement protocol. Semi-schedulability guarantees hold as long as both common cores remain operational (neither one is selected for shutting down).

The relationship between a parent and a child application is not trivial to analyse, and if the executions are not synchronised well, it can cause the child to miss deadlines. One such example is given in Figure 3. An application a_c is semi-schedulable with respect to a_p. Additionally, an application a_q exists in the system, with the priority lower than that of a_p, but higher than that of a_c, i.e. $P_p > P_q > P_c$. On both the cores, $\mathcal{M}_x$ and $\mathcal{M}_y$, the applications a_p and a_q are offline schedulable. a_c is not, but the absence of a_p would also make it offline schedulable. These conclusions can be reached by solving Equation 1 for this example, with the job parameters given as fractions C/T in Figure 3. Due to space constraints, the computation is omitted.

a_p completed its first job on $\mathcal{M}_x$, but for the next release at $t = 5$ decided to migrate to $\mathcal{M}_y$ and stays there until the end of the example. In order to release its job, a_c runs its agreement protocol at $t = 6$ and realises that the semi-schedulable parent is on $\mathcal{M}_y$, hence goes to $\mathcal{M}_x$. However, until its deadline, a job of a_c can not complete the execution, due to the higher priority workload of a_q and thus misses the deadline.

As seen, migrations of a parent application can make a child application unschedulable, even when organising their executions on different cores. In this example the first execution of a_p on $\mathcal{M}_x$ delayed an execution of a_q and created a workload backlog which consequently delayed the execution of a_c. Due to this delay, a_q requested more execution time than what is exhibited in the offline schedulability test performed for a_c, thus making it only a necessary but not a sufficient condition for semi-schedulability (i.e. the Equation 1 shows that a_q induces 6 time units of interference to a_c in the interval between its release and completion, while in the given example it sums up to 9 time units). Therefore, before performing the migration, a parent application should check whether its actions cause the unschedulability of its child. In our example, through its agreement protocol at $t = 5$, a_p should have checked whether its migration to $\mathcal{M}_y$ would cause a_c to be unschedulable on $\mathcal{M}_x$, and if so, act accordingly (i.e. either continue executing on $\mathcal{M}_x$, or go to some other core).

Therefore, during its agreement protocol at time instant t, a parent application should perform an online schedulability test for the next job release of a child application a occurring at r_a. It is equivalent to performing an online schedulability test for an artificial application a', with all the properties of a, except the release is at $r_{a'} = t$ and the period is extended to $T_{a'} = T_a + r_a - t$. An illustrative example is given in Figure 4, and Theorem 1 provides the proof.

Theorem 1. *Consider an application a, which is on the core $\mathcal{M}_x$ a semi-schedulable child with respect to a parent application a_p. Also consider that after a time instant t, any new release of a_p will occur on cores other than $\mathcal{M}_x$. Observed at t, the next future release of a, occurring at r_a, will be schedulable if and only if an artificial application a' with the same priority and the execution time ($P_{a'} = P_a \wedge C_{a'} = C_a$), but with the extended period $T_{a'} = T_a + r_a - t$, released on $\mathcal{M}_x$ at the time instant t is online schedulable.*

Proof: Proven by contradiction.

• Assume that a' is online schedulable, but a is not. As a is not schedulable, it did not receive the required C_a execution time units during its period (the interval between r_a and $r_a + T_a$). Since a' is schedulable and has the same schedulability requirements ($C_{a'} = C_a$), this means that it performed some computation before r_a. The fact that it performed some computation during the interval between $r_{a'}$ and r_a, suggests that an eventual busy interval of higher priority backlog caused by a_p completed before r_a. Thus, the offline schedulability condition (Equation 1) is sufficient, since it covers the worst-case. As a is offline schedulable on $\mathcal{M}_x$ excluding a_p, it has to be online schedulable as well. Contradiction has been reached.

• Assume that a' is not online schedulable, but a is. As a is schedulable, it received the required C_a execution time units during its period (the interval between r_a and $r_a + T_a$). Since a' has a larger period (the interval between $r_{a'}$ and $r_a + T_a$) of which a period of a is just a subset, it should have got at least the same amount of the execution time. As $C_{a'} = C_a$, a' has to be schedulable as well. Contradiction has been reached. ■

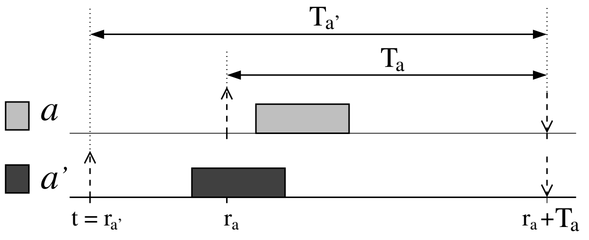


Fig. 4: Future release schedulability

The implications of Theorem 1 are that, during its release, a semi-schedulable parent application can perform the online schedulability test for the next future release of its semi-schedulable child and make a choice regarding its own future executions, such that the schedulability of a child application is preserved on at least one of semi-schedulable cores. By co-scheduling their executions, semi-schedulable applications can safely co-exist within the system (see Theorem 2).

Theorem 2. *Consider two semi-schedulable applications, a parent a_p and a child a_c, which share only two cores - $\mathcal{M}_x$ and $\mathcal{M}_y$. Assuming that $\mathcal{M}_x$ and $\mathcal{M}_y$ are fully operational, a_p and a_c can safely co-exist within the same system without missed deadlines.*

Proof: Proven by induction. Observe the time interval between two consecutive releases of a_p, termed *step*.

• Step one. Since this is the first release of a_p, no backlog workload exists which can jeopardise the schedulability of a_c, thus an offline schedulability excluding a_p (which holds) is sufficient. Two scenarios are possible. If a_c has already released a job, a_p may safely select either the other core, or one of the cores which a_p and a_c do not have in common. If a_c didn't release its first job yet, a_p can, due to the inexistence of backlog, safely choose any of the cores and leave the other core for a_c. Note, that a_c also has the possibility to choose one of the cores which it does not share with a_p. In either case, the next release of a_c will be online schedulable.

• Step $n+1$. At the beginning of the step $n+1$, a_p releases its job. As the assumption is that until the end of the step n no missed deadlines occurred, let us assume that the previous completed execution of the job of a_c occurred on $\mathcal{M}_x$. If the new execution of a_c already started before the step $n+1$, it had either safely continued its execution on the same core, or had selected some of the non-shared cores. Conversely, if the new release of a_c is yet to occur, then by applying Theorem 1 and Equation 2, a_p will check whether it can migrate to $\mathcal{M}_x$ and force a_c to go to $\mathcal{M}_y$ (or other cores), or not. In either case, the next release of a_c will be online schedulable. ■

4) **Blind Synchronisation:** So far we have proven that, as long as both semi-schedulable cores remain operational, semi-schedulable applications may safely co-reside in the same system and organise their executions in such a way that none of them misses a deadline. Yet, if the online schedulability test in not performed by solving Equation 2, but rather by solving a more pessimistic Equation 3, occasionally it may happen that the test reports the unschedulability of the child on both cores. In such cases, both semi-schedulable applications temporarily enter a *blind synchronisation mode*. By Definition 1, if a parent always executes on one semi-schedulable core, and the child on another, none of them can miss a deadline. We exploit that fact during the blind synchronisation mode, as follows:

Rule 1. *If, the core last used, from the semi-schedulable core pair, by each of the two applications (parent/child), is* ***different****, then each of the applications should choose the same core as before, over the other one.*

Rule 2. *If, the core last used, from the semi-schedulable core pair, by each of the two applications (parent/child), is* ***the same****, then whichever of them was released last on that core, should choose that core over the other one; the other application should accordingly choose the latter one over the former.*

Rule 3. *Both semi-schedulable applications can freely execute on other cores which are not semi-schedulable.*

Since there were no missed deadlines prior to the blind synchronisation mode, and given that during it no migrations of semi-schedulable applications across semi-schedulable cores will occur, from Definition 1 and Rules 1-3 it follows that no missed deadlines of these applications can occur.

5) **Discussion:** Semi-schedulability was earlier defined in the context of application pairs, i.e. a child application can have only one parent application, and vice versa. Yet, it is trivial to see that Theorems 1-2 also hold even under an extended definition of semi-schedulability wherein a parent application may have multiple semi-schedulable children applications but each child still has only one parent. For example, a 4-dispatcher parent may share two cores with one child and two different cores with another child. One could also consider allowing the children of the same parent to share cores with each other. However, such an extended model (i) would require additional proofs and (ii) more importantly, it would additionally reduce the flexibility of the approach, due to the necessity to perform co-scheduling not just between a parent and its children, but also between the children of a common parent. Hence, at this point we just conjecture the multiple-children approach, but do not employ it due to aforementioned reasons. In fact, in the experiments we only consider a 1:1 parent-child relationship.

Note, that the semi-schedulability property can be also studied from the perspective of *mode changes* [25], where the (in)existence of semi-schedulable applications on respective cores can be perceived as different system modes. Consequently, the analysis can be performed at design time, providing conditions under which semi-schedulable applications can migrate between cores. Thus, when releasing its job, a parent does not need to perform an online schedulability test for the next child's release. This is a potential topic for future work.

Let us now summarise how the aforementioned schedulability constructs are used at runtime. When an application runs its agreement protocol, all its dispatchers on currently operational cores participate. For clarity purposes, let us simplify the protocol to the extent that every dispatcher reports with a single variable $s \in \{\top, \bot\}$ about the service it can offer, regarding the next job release on its core. $\top$ means that a dispatcher can guarantee the execution on its core without missing a deadline, and $\bot$ that no guarantees can be provided.

• If a dispatcher is offline schedulable and its core is not selected for shutting down $s = \top$.

• If a dispatcher is a semi-schedulable parent and its execution will not cause the online unschedulability of a semi-schedulable child on all common cores and its core is not selected for shutting down $s = \top$.

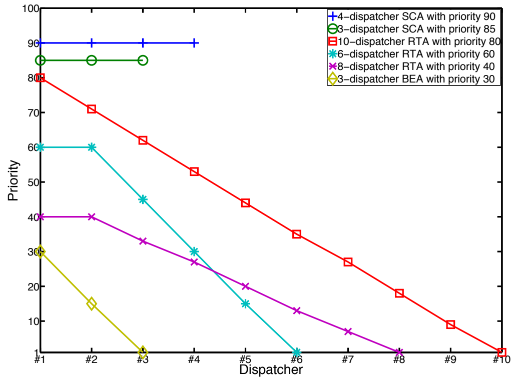


Fig. 5: Example of priority assignment upon dispatchers

• If a dispatcher is a semi-schedulable child and it passes the online schedulablity test on its core and its core is not selected for shutting down $s = \top$.

• If a dispatcher is not offline schedulable, nor semi-schedulable parent or child, and it passes the online schedulability test and its core is not selected for shutting down $s = \top$.

• In all other cases $s = \bot$.

How to elect a dispatcher, when multiple or none report $s = \top$, depends on the purpose of the system and is outside the scope of this paper. Hence, in our simulation-based evaluation, when these circumstances arise, in the former case we elect a random dispatcher of those which reported $s = \top$, while in the later any random dispatcher.

C. Priority Assignment and Mapping Process

The priority assignment and the mapping are mutually dependent activities, hence we present them in an interleaved fashion. We start with the most critical workload – SCA.

1) **Mapping SCA:** Since these applications tolerate no missed deadlines even under the worst-case conditions (with at most K concurrent core shutdowns), the fundamental mapping requirement for each SCA is to have at least $K+1$ dispatchers, and all of them mapped as offline schedulable. If any dispatcher can not be mapped as offline schedulable, a mapping process declares a failure. Therefore, to all SCA dispatchers we assign the default priorities of their respective applications (see applications with the priorities 90 and 85 in Figure 5).

SCA commence the mapping on an empty system, so most (if not all) of their dispatchers will have the possibility to choose from multiple places on the grid. We make an analogy with the bin-packing theory; dispatchers represent elements, cores symbolise bins, and the possibility of an element to fit in the bin is equivalent to testing a dispatcher's response-time on a core (Equation 1), where higher response-time is interpreted as a better mapping option (i.e. more efficiently packed elements). As an additional constraint, dispatchers of the same application can not go to the same core. We investigate three possible mapping options for SCA: (i) Best-Fit, (ii) Worst-Fit, (iii) Alternate-Fit (alternately mapping dispatchers of an application with the Best-Fit and Worst-Fit techniques).

2) **Mapping RTA:** Once all SCA are mapped, we focus on mapping RTA. As stated in Section IV-A, RTA require schedulability guarantees when the system is fully operational. Therefore, the first dispatcher of each RTA is assigned the application's priority and subsequently mapped as offline schedulable. If this is not possible, the second dispatcher is also assigned the application's priority, and both are attempted to be simultaneously mapped as semi-schedulable children of some already mapped higher priority application which is a potential semi-schedulable parent. If this is not possible either, the mapping process declares failure. When choosing the core to map the first dispatcher (in the case of offline schedulability) or when choosing the semi-schedulable parent to map the first and the second dispatcher (in the case of semi-schedulability), possible mapping options are similar to that of SCA: (i) Best-Fit, (ii) Worst-Fit and (iii) Alternate-Fit. Once this is done, there is no need to keep the priority of the rest of the application's dispatchers at the same level. Indeed, decreasing their priorities allows to preserve more schedulability resources for lower-priority RTA whose mapping did not start yet. Hence, after fulfilling the mapping condition (offline schedulability or semi-schedulability), the rest of the dispatchers of an application are assigned linearly decreasing priorities, with the last dispatcher having the least possible system priority, and they all undergo *speculative mapping* (see the next paragraph). An example of RTA priority assignment is given in Figure 5, where the application with the priority 80 managed to claim offline schedulability, while applications with priorities 60 and 40 could only claim semi-schedulability.

3) **Speculative mapping:** Unlike previous mapping stages, where the focus was on providing schedulability guarantees, when mapping speculatively the emphasis is on improving the system's overall runtime behaviour. In other words, dispatchers should not be necessarily mapped to cores where they can claim offline or semi-schedulability, but rather to cores where they have a high chance of claiming online schedulability at runtime. Therefore, the criterion for mapping is no longer the response-time, but the per-core utilisation, which is calculated as the sum of the utilisations of all contained dispatchers. The individual per-dispatcher utilisations are computed as follows: each dispatcher carries a fraction of the application's utilisation, but offline schedulable and semi-schedulable ones carry double the weight, as likelier to be elected.

We explain the speculative mapping with an illustrative example given in Figure 6, where dispatchers of the same color belong to the same application. For simplicity reasons, assume that (i) all applications have the same utilisation u, (ii) each dispatcher is either offline or semi-schedulable, and (iii) each dispatcher inherits the priority of its application. Consider that a dispatcher d_6, which belongs to the application with the lowest priority, will undergo a speculative mapping. Core 1 and Core 2 are not possible mapping options, because the application of d_6 already has dispatchers there (d_4 and d_5, respectively). From the remaining cores (Cores 3-5), a dispatcher is mapped to the one with the globally minimal utilisation. For example, the utilisation of Core 3 is equal to the sum of utilisations of d_2 and d_8. Their individual utilisations are $\frac{1}{3}u$ and $\frac{1}{2}u$, as their applications have 3 and 2 dispatchers, respectively. After computing the utilisation of every core, the conclusion is that the best mapping option is Core 4 (see Table I).

4) **Mapping BEA:** When mapping BEA the objective is to spread the ratios of BEA missed deadlines across all BEA as evenly as possible, i.e. maintain this particular notion of fairness, as described in Section IV-A. Thus, all BEA dispatchers are mapped speculatively. In order to further equalise

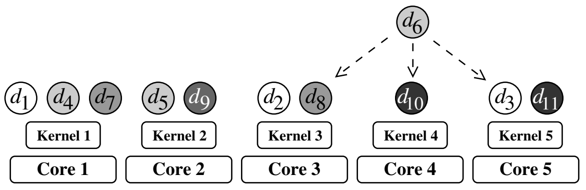


Fig. 6: Example of speculative mapping

Core 1	Core 2	Core 3	Core 4	Core 5
N/A	N/A	$\frac{5}{6}u$	$\mathbf{\frac{1}{2}u}$	$\frac{5}{6}u$

TABLE I: Speculative mapping computation

the consumption of schedulability resources, we assign linearly decreasing priorities to dispatchers of the same application (see the application with the priority 30 in Figure 5). In many cases this allows the first dispatcher of a lower priority application to have a higher priority than the second and subsequent dispatchers of some higher priority applications (even RTA), which gives it a scheduling precedence and contributes to the intended fairness.

Note, that during the entire mapping process, an ordered list of all dispatchers that are still not mapped is maintained. The ordering criterion is the non-increasing priority. The dispatchers are removed from the list and subsequently mapped in that order (e.g. in the example given in Figure 5 first the application with the priority 90 is entirely mapped, then also entirely the application with the priority 85, then the dispatchers $1-3$ of that with the priority 80, then the dispatchers $1-2$ of that with the priority 60, etc.). The rationale for this decision is that a currently mapped dispatcher does not have an influence on already mapped (higher-priority) workload, which reduces the complexity of the entire process from sub-quadratic – $O(|d|^2)$ to linear – $O(|d|)$, where $|d|$ denotes the total number of dispatchers in the application-set. Also note, that re-orderings of the list may occasionally be required in cases where RTA claim semi-schedulability, and hence the priorities of their respective unmapped dispatchers have to be elevated.

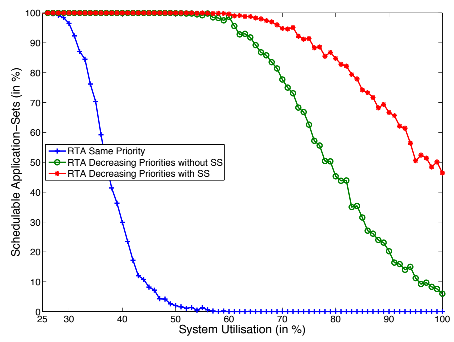
V. Evaluation

LMM should be compared with the most similar model - APA [20]. However, for that model several crucial aspects have not yet been defined. For instance, how to perform the mapping, which entity controls the migrations (the application or the scheduler), which scheduling policy is employed, how to handle core shutdowns? While these aspects are not defined, performing a fair and meaningful comparison between APA and LMM is not possible. Therefore, we focus solely on LMM and explore the impacts of different priority assignment and mapping strategies on the aspects which we consider the most relevant: (i) schedulability guarantees, (ii) the run-time behaviour assuming no core shutdowns, (iii) the runtime behaviour assuming core shutdowns. The simulations were performed on the extended version of the simulator *SPARTS* [26]. The simulation parameters are summarised in Table II. An asterisk sign denotes a uniformly distributed random value.

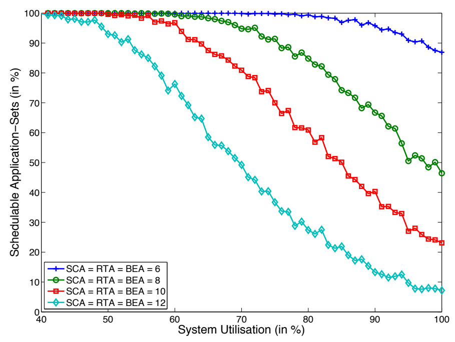
Platform size	**10 × 10**	Application-set size	**200**
Simulated time	**100 sec**	Application utilisation	**(0 - 0.7]***
SCA period	**[30 - 50]* msec**	RTA period	**[30 - 100]* msec**
BEA period	**[0.1 - 1]* sec**	SCA, RTA, BEA	**10%, 20%, 70%**

TABLE II: Simulation parameters

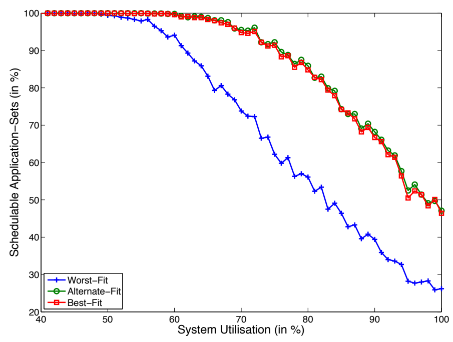
Experiment 1 (RTA Priorities and Semi-Schedulability): In this experiment we investigate the impacts of different prior-



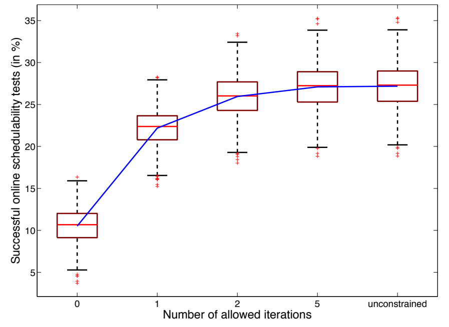
(a) RTA priorities and Semi-Schedulability



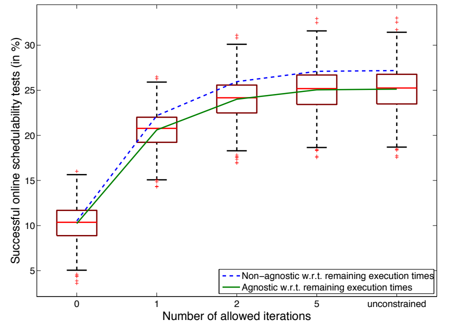
(b) Number of dispatchers



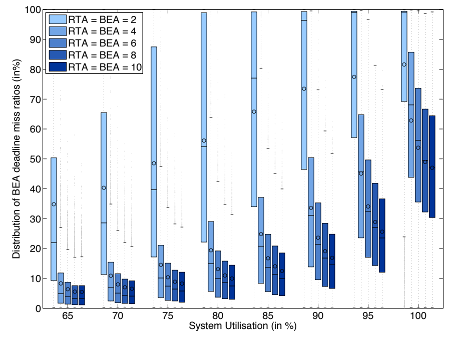
(c) Mapping strategies



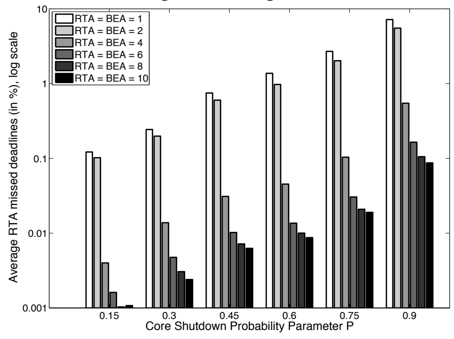
(d) Test using remaining execution times



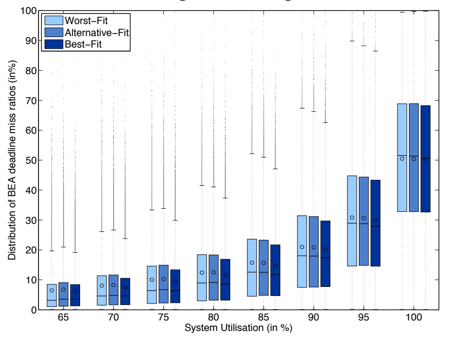
(e) Test not using remaining execution times



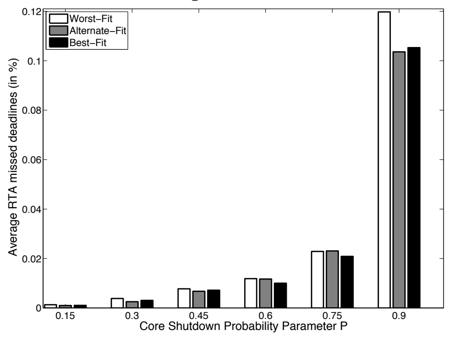
(f) Number of dispatchers without shutdowns



(g) Number of dispatchers with shutdowns



(h) Mapping strategies without shutdowns



(i) Mapping strategies with shutdowns

Fig. 7: Experimental evaluation

ity assignment techniques as well as the semi-schedulability on the provided schedulability guarantees. Recall (Section IV-A), a mapping is valid only if all requirements regarding schedulability guarantees are fulfilled, that is (i) every dispatcher of every SCA is offline schedulable, and (ii) every RTA has at least one dispatcher as offline schedulable, or a pair of dispatchers as semi-schedulable children. All application-sets, for which a mapping can be found, such that the aforementioned objectives are fulfilled, we refer to as *schedulable*.

In this experiment each application consists of 8 dispatchers, and the mapping is performed by employing the Best-Fit mapping technique. We analysed 3 different approaches, (i) all dispatchers of one application have the same priority, (ii) the priority assignment upon dispatchers is performed as described in Section IV-C, and (iii) the priority assignment is as in the previous case, but the semi-schedulability property (SS) is employed. We varied the system utilisation (x-axis) and observed the amount of schedulable application-sets (y-axis). Figure 7a shows that, assigning priorities as described in Section IV-C is an efficient strategy. Specifically, allowing dispatchers of RTA to have decreasing priorities helps to preserve more schedulability resources for lower-priority RTA that are yet to be mapped. Additionally, the tests reported a noticeable improvement when the SS technique is employed. As these strategies have proven to be beneficial, in the rest of the experiments both the semi-schedulability and the priority assignment, as proposed in Section IV-C, are employed.

Experiment 2 (Number of dispatchers): In this experiment we investigate how the number of SCA dispatchers impacts provided schedulability guarantees. Again, the Best-Fit mapping technique was employed. We analysed 4 different cases, where each application has $6, 8, 10$ and 12 dispatchers. The varied parameter is the system utilisation (x-axis), while the observed value is the number of schedulable application-sets (y-axis). Figure 7b demonstrates that the price of having more SCA dispatchers is expensive in terms of schedulability resources, however, at the same time it improves the resilience of the system and allows a higher number of concurrent core shutdowns without SCA missed deadlines.

Experiment 3 (Mapping strategies): This experiment focuses on different mapping techniques. Each application has 8 dispatchers. We analysed 3 different approaches, where the applications were mapped with (i) the Worst-Fit technique, (ii) the Alternate-Fit technique and (iii) the Best-Fit technique, as described in Section IV-C. Again, the varied parameter is the system utilisation (x-axis), while the observed value is the number of schedulable application-sets (y-axis). From Figure 7c it is visible that the Worst-Fit manages to map the least number of application-sets as schedulable. The other two techniques perform similar to each other, although the Alternate-Fit shows marginal improvements, because it tends to distribute the dispatchers more diversely, which in some cases results in better opportunities for semi-schedulability.

Experiment 4 (Online tests w/o remaining execution times): As the online schedulability tests are performed frequently, the performance of LMM depends on the trade-off between the complexity and the efficiency of these tests. In some scenarios performing the test by solving Equation 2 may be undesirably expensive, as it requires a fixed-point search algorithm and also the knowledge about remaining execution times. For that purpose, we proposed a lighter test (Equation 3), where a single computation is performed without the knowledge about the remaining execution times. In this experiment we want to investigate if the lighter tests are practical.

Each application has 8 dispatchers, and the Best-Fit mapping technique is employed. The system utilisation is fixed to 80%. We simulated 100 seconds of execution and measured the number of successful online schedulability tests. First, the tests were performed by taking into account the actual remaining execution times. We varied the number of allowed iterative computations, and if the value is not obtained within a given limit, a single computation is performed by assuming $R_i^0 = T_i$ in Equation 2. Figure 7d shows how the success ratio of the online schedulability test (y-axis) changes with the number of allowed iterations (x-axis). As seen, even if we allow just two iterations, the efficiency of the test compared to the exact test (i.e. unconstrained iterations) is barely affected. This is because the test converges fast anyway, almost always within few iterations. This is not surprising, as the applications contributing interference, in the recurrence relation, are few (i.e. just those that have dispatchers on the processor considered), unlike in global scheduling. In this sense, the analysis of LMM is quite scalable. Similarly, Figure 7e shows the success ratio of the online schedulability tests, but this time by being agnostic with respect to remaining execution times. We also plotted the average value from the previous figure, in order to ease the visual comparison. The trends are similar to the previous case, only a few iterations are needed. Moreover, being agnostic with respect to remaining execution times, as expected, has a negative effect, however the same is very mild. Thus, a light online test which is agnostic and has the limit of 5 iterations manages to succeed in more than 90% of the cases which were successful by the exact test (Equation 2). This crucial finding further motivates the research related to LMM.

Experiment 5 (Number of dispatchers and runtime): In this experiment, we investigate how the number of RTA and BEA dispatchers influences the runtime behaviour of the system. In other words, is it beneficial to have more RTA and BEA dispatchers? We set $K = 7$, i.e. a system should allow at most 7 concurrent core shutdowns. Thus, all SCA have 8 dispatchers. The number of RTA and BEA dispatchers is varied in the range $[1-10]$. The Best-Fit mapping technique was used. Let us first observe the behaviour of the system when no core shutdowns occur. Given that in these conditions no SCA, nor RTA missed deadlines can occur, of interest is the distribution of BEA missed deadline ratios. The execution was simulated for different system utilisations, and BEA missed deadlines were captured. Figure 7f shows that schemes with fewer dispatchers are more rigid and concentrate all BEA missed deadlines among very few applications. Conversely, schemes with more dispatchers clearly benefit from their flexibility, in a sense that BEA missed deadlines are evenly distributed among all applications. These trends do not reach a saturation point, but show systematic improvements as the number of dispatchers increases. This also validates the efficiency of the priority assignment techniques, and proves that by assigning priorities in a strategic manner we can benefit from the high number of dispatchers per application, and yet efficiently avoid the "suffocation effect" among applications. Note, for RTA = BEA = 1, all BEA missed deadline ratios are 100%. For better clarity, this case is omitted from Figure 7f.

We again investigate the runtime behaviour, but this time assuming core shutdowns. The duration of each shutdown is 1 second. In this and the next experiment the parameter P stands for the individual per-core probability of being selected for at least one shutdown, P^2 for at least two shutdowns, etc. All shutdowns of all cores must occur within the simulated interval of 100 seconds. Time instants at which each individual core will experience a shutdown were randomly generated, but without violating a constraint that at most $K = 7$ of them could be selected concurrently. We fixed the system utilisation to 80% and varied the parameter P (x-axis) and observed the average number of RTA missed deadlines (y-axis). Figure 7h, shows a clear benefit of having more dispatchers, for every value of P. A slight increase in the number of dispatchers may improve the resilience towards core shutdowns even by one order of magnitude, while any additional increase clearly contributes to the system flexibility to tolerate more frequent core shutdowns.

Experiment 6 (Mapping strategies and runtime): In this experiment, we investigate how different mapping techniques influence the runtime behaviour of the system. Again, $K = 7$. Each application has 8 dispatchers. First, we observe the system behaviour when no core shutdowns occur and focus on the distribution of BEA missed deadline ratios. We simulated all 3 proposed mapping techniques for different system utilisations (x-axis) and captured the ratio of BEA missed deadlines (y-axis). Figure 7h shows the results. It is noticeable that the Best-Fit technique achieves the best results, although the differences are very subtle and almost negligible.

Now, we investigate the runtime behaviour, but with core shutdowns. The system utilisation is 80% and the parameter P is varied (x-axis). We focus on RTA missed deadlines (y-axis). Figure 7i suggests that all techniques demonstrate a comparable performance.

Experiment 7 (Blind synchronisation): In this experiment, we investigate how often the blind synchronisation mode (BSM) occurs. We assume a setup identical to that of Experiment 4, with the only difference that now we focus on the releases of semi-schedulable applications which cause the BSM (y-axis). The varied parameter is the allowed number of iterations in the schedulability test recurrence (x-axis). For each value of the allowed number of iterations we performed the simulations, assuming two types of online schedulability tests, ones which are agnostic with respect to remaining execution times, and ones which are not. Figure 8 shows the results. It comes at no surprise that the agnostic tests, due to being more pessimistic, cause more frequent occurrences of the BSM, than the respective non-agnostic ones. However, the differences are

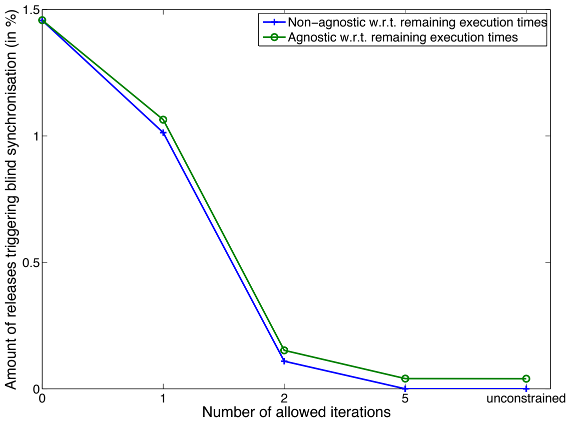


Fig. 8: Blind synchronisation mode

negligible. The explanation for this finding is twofold. First, Experiment 4 demonstrated that the pessimism of the agnostic tests, when compared to the respective non-agnostic ones, is not significant. Second, in many cases, the BSM requires specific (worst-case) conditions, which do not occur frequently during runtime.

As expected, allowing more iterations decreases the occurrences of the BSM, because the respective online schedulability tests become less pessimistic. This coincides with the findings of Experiment 4. In any case, even when using the most pessimistic tests (e.g. Equation 3), the BSM is triggered, on average, in just 1.5% of the releases of semi-schedulable applications. This shows that the conditions leading to the BSM arise very rarely during runtime, and also shows that for many semi-schedulable application pairs the BSM cannot occur, not even theoretically, irrespective of the employed online schedulability test.

Discussion: Assigning priorities as proposed in Section IV-C proved to be an efficient approach. Also, the semi-schedulability exhibited a huge impact on schedulability guarantees, which we consider very beneficial. Assigning the application's default priority to all $K+1$ dispatchers of SCA is costly, in terms of schedulability resources, but achieves the required schedulability guarrantee (i.e. at up to K concurrent core shutdowns). Understandably, providing strong guarrantees to SCA, for the event of core shutdowns (which is the main objective), commensurately "withholds" resources from RTA and BEA, but this is mitigated to a large extent by the flexibility of LMM.

Having more RTA and BEA dispatchers proved to be beneficial in both schemes, with and without core shutdowns. Due to the efficient priority assignment technique, schedulability guarantees for RTA are not influenced by the number of dispatchers per application. The additional system flexibility, brought by multiple dispatchers, indirectly through RTA and directly through BEA, contributes to the equal distribution of missed deadlines among BEA (assuming no core shutdowns) and minimises the number of RTA missed deadlines (assuming core shutdowns). However, as the number of dispatchers increases, the benefits from additional dispatchers start to level off, which may be an important factor when the communication delays [22] are taken into account.

Mapping with different mapping strategies has almost negligible effects. The Alternate-Fit approach is the most efficient in providing schedulability guarantees, while the Best-Fit technique is the best in terms of runtime behaviour, both with and without core shutdowns. The Worst-Fit approach performs worse than both the aforementioned techniques, in all investigated categories and its use cannot be justified.

Performing a light online schedulability test (agnostic with respect to remaining execution times, with at most 5 iterations) is in more than 90% of the cases as good as performing an exact test (Equation 2), while in only 0.04% of the releases of semi-schedulable applications it causes the blind synchronisation mode.

It is apparent that there is no single strategy which yields the best results under all circumstances. Facts such as the purpose of the system, the amount and the nature of the workload, the maximum number of concurrent core shutdowns K, core shutdown policies, the tolerable amount of RTA/BEA missed deadlines, are only few factors, out of many, which a system designer should take into account when choosing the strategy. We perceive the mapping process as an adaptive activity, where different strategies are attempted until reaching the solution with (i) the necessary amount of schedulability guarantees, (ii) the acceptable level of flexibility and resilience towards core shutdowns, and (iii) the satisfactory runtime performance.

VI. Conclusions and Future Work

In this paper we studied the Limited Migrative Model (LMM). We have classified the schedulability guarantees and subsequently proposed a heuristic-based method for priority assignment and application mapping. We elaborated on various mapping choices and analysed their impacts on the system. The experiments showed that LMM is a valuable approach even in scenarios where all cores are available, while it shows clear benefits in cases where a higher amount of flexibility is appreciated due to necessity to perform occasional core shutdowns. As future work, we plan to enhance the analysis to address more rigorous core shutdown and core failure scenarios. Also, we plan to extend the approach by migration-related overheads [6], communication and memory traffic analyses [22], [23], so as to unify the on-core and network analyses for LMM. Finally, how to provide a fair comparison of LMM against similar approaches (e.g. APA [20]) is an interesting problem to study.

References

[1] M. Lundstrom, "Moore's law forever?" *Science*, 2003.

[2] Intel, *The Single-chip Cloud Computer*, www.intel.com/content/www/us/en/research/intel-labs-single-chip-cloud-computer.html.

[3] Tilera, *TILE64^{TM} Processor*, www.tilera.com/products/processors/TILE64.

[4] S. Baruah and Z. Guo, "Mixed-criticality scheduling upon varying-speed processors," in *34rd RTSS*, 2013.

[5] A. French, Z. Guo, and B. Sanjoy, "Scheduling mixed-criticality workloads upon unreliable processors," in *11th WS Models & Algorithms for Planning & Scheduling Problems*, 2013.

[6] B. Nikolić and S. M. Petters, "Towards network-on-chip agreement protocols," in *12th EMSOFT*, 2012.

[7] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new os architecture for scalable multicore systems," in *SOSP*, 2009.

[8] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *SIGOPS Oper. Syst. Rev.*, 2009.

[9] M. D. Y. Li and R. West, "Quest-v: A virtualized multikernel for high-confidence systems," Tech. Rep., http://www.cs.bu.edu/~richwest/quest.html.

[10] P. K. Sahu and S. Chattopadhyay, "A survey on application mapping strategies for network-on-chip design," *J. Syst. Arch.*, 2013.

[11] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, 1973.

[12] J. Hu and R. Marculescu, "Energy-aware mapping for tile-based noc architectures under performance constraints," in *8th ASPDAC*, 2003.

[13] S. Murali and G. De Micheli, "Bandwidth-constrained mapping of cores onto noc architectures," in *7th DATE*, 2004, pp. 20 896–.

[14] W. Hung, C. Addo-quaye, T. Theocharides, Y. Xie, N. Vijaykrishnan, and M. J. Irwin, "Thermal-aware ip virtualization and placement for networks-on-chip architecture," in *Int. Conf. Comp. Design*, 2004.

[15] K. Bletsas and B. Andersson, "Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound," in *30th RTSS*, 2009.

[16] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *21st ECRTS*, 2009.

[17] T. P. Baker, "An analysis of fixed-priority schedulability on a multiprocessor," *Real-Time Syst. J.*, 2006.

[18] S. Baruah and T. Baker, "Schedulability analysis of global edf," *Real-Time Syst. J.*, 2008.

[19] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers," in *31st RTSS*, 2010.

[20] A. Gujarati, F. Cerqueira, and B. Brandenburg, "Schedulability analysis of the linux push and pull scheduler with arbitrary processor affinities," in *25th ECRTS*, 2013.

[21] S. Baruah and B. Brandenburg, "Multiprocessor feasibility analysis of recurrent task systems with specified processor affinities," in *34rd RTSS*, 2013.

[22] B. Nikolić, P. M. Yomsi, and S. M. Petters, "Worst-case communication delay analysis for many-cores using a limited migrative model," Tech. Rep., available at: http://www.cister.isep.ipp.pt/people/Borislav+Nikolic/publications/.

[23] ——, "Worst-case memory traffic analysis for many-cores using a limited migrative model," in *19th RTCSA*, 2013.

[24] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes," in *24th RTSS*, Cancun, Mexico, Dec 2003.

[25] J. Real and A. Crespo, "Mode change protocols for real-time systems: A survey and a new proposal," *Real-Time Syst. J.*, 2004.

[26] B. Nikolić, M. A. Awan, and S. M. Petters, "SPARTS: Simulator for power aware and real-time systems," in *8th IEEE ICESS*, 2011.