A Semi-Partitioned Model for Scheduling Mixed Criticality Multi-Core Systems

Hao Xu

PhD

University of York

Computer Science

July 2017

Abstract

Many Mixed Criticality scheduling algorithms have been developed with an assumption that lower criticality level tasks may be abandoned to guarantee the schedulability of higher criticality tasks when the criticality level of the system changes. But it is valuable to explore means by which all of the tasks remain schedulable throughout criticality level changes.

This thesis introduces a semi-partitioned model which allows all of the tasks to remain schedulable if only a bounded number of cores increase their criticality levels. In such a model, some lower criticality tasks are allowed to migrate instead of being abandoned. Different possible semi-partitioned approaches are proposed and analysed in this thesis. It is concluded from the experiments results that the semi-partitioned algorithm provides improved schedulability and performance of multi-core mixed criticality systems while enables all tasks to keep executing in the majority of scenarios.

This thesis also includes the consideration of migration overheads, the definition of fault tolerance models and the effects of the system architecture.

Contents

List of Figures

List of Tables

Acknowledgements

The development of this thesis is the result of the interaction with a number of people.

First of all I would like to thank my supervisor Prof. Alan Burns for his invaluable guidance, and my assessor Iain Bate as well, for their continuing support and patience. Most importantly, they supported my work with valuable suggestions. I am deeply grateful to Alan Burns for reviewing this thesis and providing helpful feedback.

I also want to express very special thanks to my colleagues in the Real-Time Systems Research Group at the University of York for the help they provided in my research. My thanks also go to all the staff of the Department of Computer Science for their help and support.

Finally, my deepest and sincere thanks go to my family, especially my wife, for their love, care and moral support during my study. They have contributed much to the successful completion of this work.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References. Part of the work in Chapter 3 is published in the paper "Semi-partitioned model for dual-core mixed criticality system" (Proceedings of the 23rd International Conference on Real Time and Networks Systems, 2015).

Chapter 1

Introduction

In real time systems, the correctness of a task not only relies on the logical correctness, but also relies on the time instance at which the result is generated [35]. Therefore, to guarantee the correctness of a set of tasks, real-time systems need to schedule the execution orders for tasks. Cyclic Executive Approach, Fixed Priority Scheduling (FPS), and Earliest Deadine First Scheduling (EDF) are widely used scheduling mechanisms, and all of these methods are applied under an assumption that the Worst-Case Execution Time (WCET) of each task is known. However, due to several uncertainties, such as the hit or miss of the cache memory, it is hard to calculate the exact WCET of a task. So the WCET value used in scheduling is generally estimated, and there are two main approaches to estimate the WCET of a task: theoretical path calculation and water mark observation. Referring to the theoretical path calculation, this approach enumerates all execution paths of a task which guarantees the estimated WCET value to be larger than the exact one. In addition, since this approach needs to enumerate all possible paths, the cost of this approach is quite high and the estimated value is often too pessimistic [87]. Furthermore, for many non-trivial kinds of code, these strict upper bounds are extremely pessimistic, and represent scenarios which are highly unlikely, or indeed impossible to occur [33]. Referring to the water mark observation, this approach executes the task a number of times and records the longest time (plus some safety margin) to be the estimated

WCET. This approach is less expensive comparing with the previous approach. But the estimated WCET value acquired by this approach cannot be guaranteed to be larger than the exact value, which may cause scheduling failure when using this approach. In all, the relationship between the estimated values from the above approaches and the exact value can be viewed in Figure 1.1.

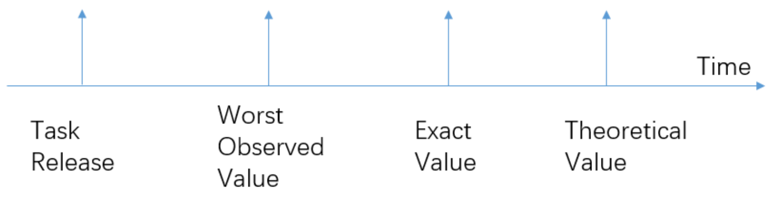


Figure 1.1: Estimated WCET values

According to the features of the two approaches, it can be indicated that if all of the tasks are scheduled using the theoretical estimated values, then the deadlines of the tasks will be guaranteed to be met. But since the theoretical estimated values are very pessimistic, the system capacity requirement will be high. Otherwise, if all of the tasks are scheduled using the observed estimated values, the system capacity requirement will be reduced from the previous case. But in such a scenario, there exists a risk that tasks may miss some of their deadlines at run-time. Thus, applying one estimation approach to all of the tasks in the system may not be appropriate for the system which contains a combination of tasks with different importance. For the tasks that missing their deadlines may cause the whole system to break down or even threaten the lives of humans, the theoretical path calculation approach is more appropriate as it guarantees the tasks will never execute beyond the estimated WCET value. For other tasks that are not quite essential to the system, water mark observation approach is sufficient to provide a WCET estimate value that a task will execute within the estimated value in the majority of cases. Overall, there exists a requirement on differentiating the importance of the tasks, and assigning different WCET estimation approaches to the tasks regarding to their importance. This leads to the definition of criticality levels and mixed-criticality systems.

1.1 Mixed Criticality System

Criticality may refer to the importance of the tasks, but the actual definition varies based on the scenarios of usage. A general division used to identify the importance of a task is to check the consequence of missing its deadline. If people may die or the whole system may break down due to the miss of a deadline, then such a task is called safety-critical; if only some functions of the software will be affected, then such a task is called mission-critical; other tasks are considered as non-critical. But criticality can also be defined to represent other concepts, including Safety Integrity Level (SIL), which makes some differences on the definition of the system model [56]. The 'criticality' used in this thesis will stick to the former one which checks the consequence of missing a deadline.

Generally, a system contained tasks with different criticality levels is called a Mixed Criticality System (MCS). According to Vestal's model [86], the higher the criticality level is, the larger the WCET estimated value is. In addition, Vestal's model also suggests that a criticality level is required for each core in a multi-core system, which indicates the criticality level to be assured at run-time. Based on that, the scheduling goal for Vestal's model is that when one core is required to assure a criticality level, each task on the core needs to be analysed with its corresponding estimated WCET value of that criticality level.

Based on Vestal's model (which will be reviewed in detail in the next chapter), many papers have been published to explore the field of Mixed Criticality Systems (MCS) both on uni-processor and multi-processors. Different system models have been identified and a variety of algorithms have been introduced. For example, Baruah and Burns extended Vestal's model so that not only the execution time but also the minimum interval period and deadline might be changed in order to meet the need of the task to complete before safety-critical time constraints [12]. According to their model, task may have different periods and deadlines at different criticality levels. Nevertheless, in this thesis, we propose to use the Vestal's original model that only WCET estimated value is changed for different criticality levels of a task.

1.2 Motivation

Although Vestal started the research of MCS, his original algorithm performs pessimistically on criticality inversion cases which will be discussed in the next chapter. According to that, a variety of algorithms, such as AMC [13] and EDF-VD [16], have been developed to improve the scheduling efficiency of MCS. However, most of these algorithms are invented under the assumption that the system only contains two criticality levels. In addition, some algorithms allow tasks with lower criticality levels to be abandoned for a period of time in certain scenarios in order to ensure the execution of high criticality tasks. Although these abandoned tasks will be brought back to execution later, it would be better if they were not abandoned in the first place. But it is not possible to solve such a problem on a uni-processor system as the maximum computation capability of a core is fixed and it is often too expensive to increase the performance of one core. Thus, many researches ([2],[64],[79]) have addressed MCS execution on a multi-core platform, since one of the key features of a multi-core platform is that tasks may migrate from one core to others during their execution, which provides more flexibility for scheduling.

Multi-core scheduling algorithms can generally be divided into three categories [44]: partitioned scheduling, global scheduling and semi-partitioned scheduling. Most of the researches conducted in multi-core MCS use partitioned scheduling since it statically maps tasks to processors. This partitioning provides a stable and predictable implementation that is preferable for safety critical applications. However, partitioned scheduling also provides isolation between the cores that denies the flexibility of the multi-core structure. Global scheduling allows tasks to migrate between different processors during execution which potentially provides higher overall utilization. But global scheduling is unpredictable and the failure of any task may result in the failure of the whole system which conflicts with the original intention of differentiating the importance of tasks. Thus, there exists a trade-off between the flexibility and predicability of the scheduling algorithms. Semi-partitioned scheduling is a mixture of the previous two algorithms in which hard real-time tasks may be statically mapped to processors and other tasks are able to migrate for schedulability. In other words, semi-partitioned scheduling provides predicability to partitioned tasks while provides flexibility to migration tasks.

Since the tasks with higher criticality levels requires predictability in order to guarantee their completions, most of the existing multi-core MCS researches are using the fully partitioned approach. In these existing works, tasks with lower criticality levels may be abandoned during criticality mode changes of the cores. In addition, as discussed above, the partitioned approach denies the flexibility of the multi-core structure. However, although global scheduling may introduce more flexibility into scheduling, it also introduces unpredictability of tasks. Considering that, semi-partitioned scheduling is suitable in this case as it is able to control the trade-off between the predictability and the flexibility for multi-core MCS. For example, tasks with higher criticality levels may be statically partitioned on processors in order to guarantee their executions, while tasks with lower criticality levels may be able to migrate during their executions. For instance, if core 1 enters into a higher criticality level, then some tasks originally executing on the core can be migrated to another core (core 2 in this case) to guarantee the schedulability of other tasks on the core (see Figure 1.2). According to the figure, there is still blank space in core 1 which indicates that with more available cores, it is likely that more tasks can be scheduled on the core.

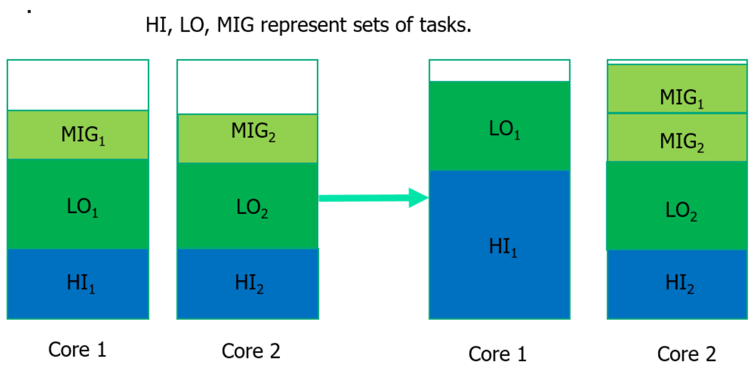


Figure 1.2: Semi-partitioned MCS Example

Currently, few paper has attempted to explore semi-partitioned algorithms in MCS. We propose to find a possible semi-partitioned scheduling algorithm on multi-core MCS in which all tasks remain schedulable in the majority of scenarios.

1.3 Thesis Proposition

The focus of this thesis lies on the semi-partitioned scheduling of multi-core MCS, and the thesis proposition can be stated as following:

A semi-partitioned approach to task placement on multiprocessor platforms can improve the performance of mixed-criticality systems, enabling all tasks to keep executing in the majority of scenarios.

The performance here refers to the schedulability, scalability and complexity, where complexity mainly refers to the runtime overheads.

1.4 Structure

The set of contributions stated in this thesis support the thesis proposition and are summarised in the following.

Chapter 2 provides the definition of various terms, analyse algorithms and notations used in this thesis. A critical review of relevant research topics in uni-processor mixed criticality scheduling algorithms, task allocation in multi-core mixed criticality system, and existing semi-partitioned mixed criticality scheduling algorithms is given. It illustrates a skeleton of the semi-partitioned algorithm studied in this thesis.

Chapter 3 starts the exploration of semi-partitioned algorithms on a dual-core platform with dual-criticality levels. In this platform, only low criticality tasks can migrate and tasks may only migrate from one defined core to another defined core. So the research focus in Chapter 3 is on the task allocation approaches and the algorithms to determine the migratable tasks. In this research, six approaches are proposed and evaluated.

Chapter 4 extends the model proposed in Chapter 3 to the n-core platform with dual-criticality levels. It provides a qualified definition of scenarios that all tasks need to be saved, and redefines the semi-partitioned model for a n-core platform. Based on the new model, this chapter explores four possible scheduling approaches for semi-partitioned scheduling for a 4-core platform. These approaches are evaluated, and the proposed approach is then extended to an n-core version.

Chapter 5 further extends the model to n-core systems with multi-criticality levels, and takes into consideration the influence from the system architecture. This chapter applies the new model to a 16-core platform with 3-criticality levels, and evaluates how the semi-partitioned algorithm may improve scheduling.

The research work described in this thesis is summarized in Chapter 6. Conclusion are drawn from the set of research contributions and prospective future work is considered.

Chapter 2

Literature Review

This chapter will first introduce the notations used in this thesis. Then it will provide some background information about real-time systems and a review of the works already done in uni-processor platform MCS, mainly upon Fixed Priority Scheduling and Earliest Deadline First Scheduling. After that, it will introduce the scheduling approaches mainly used in multi-core platforms. Then it will introduce several existing findings and knowledge, relating to semi-partitioned scheduling in multi-core systems. It will also introduce an existing semi-partitioned model MCS and a discussion of the differences between this model and the model to be developed in this thesis will be demonstrated. Finally, a summary will be given at the end of this chapter.

2.1 Notations and Assumptions

Since tasks may have different WCET for their different criticality levels, the notation for MCS is slightly different from the standardized notation for real-time systems. Table 2.1 shows the symbols used in this thesis.

In addition, the word "job" is used to represent one invocation/release of a "task", and the word "taskset" is used to represent a finite set of tasks. This thesis considers the sporadic task model, where the deadline of a task is smaller

Notation	Description
τ_i	Task i
D_i	The relative deadline of task τ_i
D_i'	The reduced deadline of task τ_i
T_i	The period of task τ_i
L_i	The criticality level of task τ_i
$C_i(L_i)$	The WCET estimation of task τ_i at criticality level L_i
$U_i(L_i)$	The utilisation of task τ_i at criticality level L_i
J_i	The release jitter of task τ_i
I_i	The interference time of the task τ_i
O_i	The overhead of the task τ_i
c_j	Core j
R_i	The response time of task τ_i

Table 2.1: MCS Notation

than or equal to its period/minimum release interval.

2.2 Real-Time System

As stated at the very beginning, the correctness of a real-time task not only relies on the produced results but also relies on the time it produces the results. Based on the focuses of correctness, real-time systems are mainly distinguished between hard and soft real-time systems. Referring to hard real-time systems, all tasks must finish all their releases within the given deadlines. While for soft real-time systems, meeting the deadlines is important, but the system will still function correctly if deadlines are occasionally missed.

In a hard or soft real-time system, the computer is usually interfaced directly to some physical equipment and is dedicated to monitoring or controlling the operation of that equipment. Since the key feature of these applications is the role of the computer acting as an information processing component within a larger engineering system, such applications are also known as embedded computer systems. In order to activate systems to keep up with the environment, tasks are structured to be either time-triggered or event-triggered. For time-triggered tasks, all computation activities are periodic and have a defined cycle time, for example 10 ms, and are released for execution by an internal clock. Based on that, such tasks are also known as periodic tasks. For event-triggered tasks, the environment explicitly controls the release for execution of some software activities. Such tasks are also known as aperiodic tasks. If there exists a bound on how often the releasing event can occur in a time interval, such tasks are named sporadic. This thesis focuses on systems with sporadic and periodic tasks.

In a concurrent program, it is not necessary to specify the exact order in which tasks execute as the usage of synchronization primitives, such as mutual exclusion, enforces the local ordering constraints. However, the general behaviour of the program exhibits significant non-determinism. If the program is correct, then its functional outputs will be the same regardless of internal behaviour or implementation details. While the program's outputs are identical with all these possible interleaving, the timing behaviour may vary considerably. A task with a tight deadline may request to execute first in order to meet the program's temporal requirements.

A real-time system needs to restrict the non-determinism found within concurrent systems, which indicates the usage of a scheduler. Regarding to that, a large number of different scheduling approaches, such as Fixed-Priority Scheduling and Earliest Deadline First scheduling, have been introduced to address this problem. Cheng et al. [40] provides a systematic classification of the scheduling approaches (Figure 2.1). According to the scheduling decision time, the schedulers can be divided into two types: static and dynamic. A scheduler is called static if it makes its scheduling decisions at compile time; while a scheduler is called dynamic if it makes its scheduling decisions at run time, selecting one out of the current set of ready tasks. In static scheduling, the schedulers can be further divided into two types: preemptive and non-preemptive. In preemptive scheduling, there will be an immediate switch to any released higher-priority task. In non-preemptive scheduling, the currently executing task will not be interrupted until it decides on its own to release the allocated resources. In general, the preemptive scheduling is preferred since it enables high-priority (short deadline) tasks to be more reactive [35]. A test that determines whether a set of tasks can

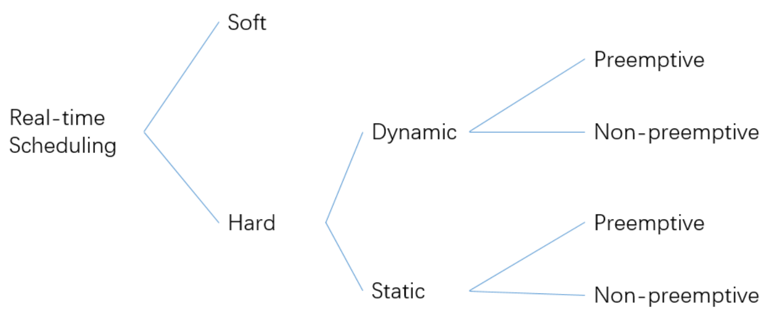


Figure 2.1: Taxonomy of Real-time Scheduling Algorithms [40]

be scheduled (each task meet its deadline) is called a schedulability test. The test is generally distinguished between exact, necessary, and sufficient schedulability tests [65]. A schedulability test is defined to be sufficient if a positive outcome guarantees that all deadlines are always met. While a test can be labelled as necessary if the failure of the test will indeed lead to a deadline miss at some point during the execution of the system. A sufficient and necessary test is exact and hence is in some sense optimal; a sufficient but not necessary test is pessimistic.

There exist a number of different scheduling approaches. We will introduce two common algorithms, fixed-priority scheduling and earliest deadline first scheduling, as these two algorithms are most widely used.

Fixed-Priority Scheduling

Fixed-Priority Scheduling (FPS) is the most widely used approach and is the main focus of this thesis. In FPS, each task has a fixed and static priority which is computed pre-run-time. The system will always execute the available task with the currently highest priority first. It is worthwhile to note that this priority value is derived from the temporal requirements of the task, not the importance of the task. In addition, priority 1 (or 0) implies the highest priority, as this is the normal usage in most scheduling analysis.

There exist several priority assignment schemes for FPS. Rate Monotonic Priority Assignment (RMPA) is a simple optimal priority assignment scheme for tasks that have $T = D$. In RMPA, each task is assigned a priority based on its period. The shorter the period, the higher the priority (e.g. for task τ_i and task τ_j, $T_i < T_j \Rightarrow P_i > P_j$). This assignment is optimal in the sense that if any task set can be scheduled with a fixed-priority assignment scheme, then the given task set can also be scheduled with RMPA [70].

Leung et al. [68] have defined a priority assignment scheme, named as Deadline Monotonic Priority Assignment (DMPA), caters for tasks that have $D \leq T$. DMPA behaves similarly to RMPA but assigns priorities inversely proportional to the length of the relative deadlines of the tasks. Thus, the task with the shortest deadline is assigned the highest priority while the task with the longest deadline is assigned the lowest priority. It can be observed that DMPA will provide the same priority order to RMPA for cases that have $D = T$. Leung et al. [68] also prove that DMPA is an optimal static priority scheme for uni-processor scheduling.

However, both RMPA and DMPA are only optimal for simple task models. They cannot provide optimal priority ordering for tasks with arbitrary deadlines. Audsley et al. [5] introduces a theorem and algorithm, also known as Audsley's Optimal Priority Assignment (AOPA), for assigning priorities in arbitrary situations.

Theorem: If task p is assigned the lowest priority and is feasible, then, if a feasible priority ordering exists for the complete taskset, an ordering exists with task p assigned the lowest priority.

According to this theorem, if any task is schedulable at the lowest value, then the task can be assigned that priority, and the scheduling problem is therefore reduced to be a subproblem to assign the other $N-1$ priorities. This progress can be reapplied to the reduced task set. Hence through successive reapplication, a complete priority ordering can be obtained if one exists.

It is worthwhile to mention that their theorem is proved by using the response time analysis equations (will be mentioned later in this section) that if a task has the lowest priority then it suffers interference from all high-priority tasks and this interference is not dependent upon the actual ordering of these higher priority tasks. Thus, according to David and Burns [43], there exists pre-conditions for applying the AOPA. In detail, for a schedulability test S to be compatible with the AOPA algorithm, it must comply with three conditions stated below:

- Condition 1: The schedulability of a task τ_i may, according to test S, depend on any independent properties of tasks with priorities higher than i, but not on any properties of those tasks that depend on their relative priority ordering.
- Condition 2: The schedulability of a task τ_i may, according to test S, depend on any independent properties of tasks with priorities lower than i, but not on any properties of those tasks that depend on their relative priority ordering.
- Condition 3: When the priorities of any two tasks of adjacent priorities are swapped, the task being assigned the higher priority cannot become unschedulable according to test S, if it is previously schedulable at the lower priority.

The above paragraphs have introduced several priority assignment schemes. With the priorities assigned to the tasks, scheduling tests can be used to determine whether a taskset is schedulable or not by fixed priority scheduling. Liu and

Layland [70] show that by considering only the utilization of the taskset, a test for schedulability can be obtained (when the rate monotonic priority ordering is applicable and used):

$$\sum_{i=1}^{N}\left(\frac{C_i}{T_i}\right) \leq N(2^{1/N} - 1) \tag{2.1}$$

They show that for large N, the bound asymptotically approaches 69.3%. Hence any taskset with a combined utilization of less than 0.692 will always be schedulable by a preemptive priority-based scheduling scheme, with priorities assigned by the rate monotonic algorithm. However, the utilization-based tests for FPS have two significant drawbacks: the tests are not exact, and the tests are not really applicable to a more general task model. In considering that, Response Time Analysis (RTA) is introduced to provide an exact scheduling test for general task models. According to Burns and Wellings [34], response time analysis has two stages. In the first stage, an analytical approach is used to predict the worst-case response time (R) of each task. The next stage is to compare these response values with the deadlines of the corresponding tasks.

In a pre-emption scheme, since the task with the highest-priority will always execute once it is released, its worst-case response time equals its own computation time (that is, $R = C$). Other tasks will suffer interference from higher-priority tasks. The interference time refers to the time spent on executing higher-priority tasks when a low-priority task is runnable. So for a general task τ_i, the response time R_i can be represented as equation (2.2), where I_i is the maximum interference that task i can experience in any time interval $[t, t + R_i)$.

$$R_i = C_i + I_i \tag{2.2}$$

The maximum interference occurs when all higher-priority tasks are released at the same time as the task. Assume that all tasks are released at time 0 and consider one task τ_j with higher priority than task τ_i, then the maximum interference from task τ_j can be obtained as the number of release of the task τ_j

multiple with the WCET of task τ_j, which is $\lceil \frac{R_i}{T_j} \rceil * C_j$. Substituting these values for all higher-priority tasks will get the maximum interference suffered by task τ_i. Thus, the response time analysis for task τ_i is shown as equation (2.3)(Joseph and Pandya [61]), where $hp(i)$ stands for the set of tasks with higher priority than task τ_i:

$$R_i = C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.3}$$

Although the formulation of the interference equation is exact, the actual amount of interference is unknown as R_i is unknown. According to equation (2.3), due to the ceiling functions, it is difficult to solve and obtain the value R_i. As the characteristic of fixed-point equation, there will be many values of R_i that form solutions to the equation, and the smallest positive value represents the worst-case response time for the task. The simplest way of solving the equation is to form a recurrent relationship (Audsley et al. [5]):

$$\omega_i^{n+1} = C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{\omega_i^n}{T_j} \right\rceil C_j \tag{2.4}$$

The set of values $\omega_i^0, \omega_i^1, \omega_i^2, ..., \omega_i^n, ...$ is monotonically non-decreasing. When $\omega_i^n = \omega_i^{n+1}$, a possible solution to the equation is found. If $\omega_i^0 < C_i$, then ω_i^n is the smallest value and hence is the worst-case response time for the task. If the equation does not have a solution, the ω values keep rising. Once ω_i gets bigger than the period of task τ_i, it can be assumed that the task will not meet its deadline. According to that, the starting value, e.g. ω_i^0, must be no larger than the unknown final solution. Since $R_i \geq C_i$, C_i is a safe starting point which is efficient enough for most of the cases. Davis et al. [45] have proposed more efficient starting values if the efficiency of the response time analysis is an issue. Since the calculation of response time analysis is quite lengthy, we only show the last step of the recursive calculations when showing examples in later chapters.

The response time analysis is an exact scheduling test. If a taskset passes the test, then all of the tasks will meet all their deadlines. Otherwise, if a taskset fails the test, then a task will miss its deadline at run-time (if WCET are exact). In addition, the original response time analysis equation can be extended to deal with more complicated and practical situations. In this thesis, we will focus on the extended version relating to the release jitter issue. Release jitter is a key issue in distributed systems and this problem mainly happens when a sporadic task τ_s is released by a periodic task τ_p. Although the periodic task and the sporadic task have the same release frequency, the maximum interferences from these two tasks on lower priority tasks are different. This difference can be observed by considering two consecutive executions of the sporadic task τ_s. Assume that the sporadic task τ_s is released when the periodic task τ_p finishes its executing. The first release of the sporadic task τ_s occurs at time R_p. Assume there is no interference for the second release of periodic task τ_p, then it finishes at time $T_p + C_p$. That is, the sporadic task τ_s is released at time $T_p + C_p$. According to that, the two executions of the sporadic task are not separated by T_p but by $T_p + C_p - R_p$. The maximum variation in the task's release is termed the release jitter of the task. For a task with release jitter J, it is released at times 0, $T - J$, $2T - J$, and so on. Audsley et al. [5] analyse the possible interferences from tasks with the release jitter, and extend the original response time analysis equation as follows:

$$R_i = C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{2.5}$$

This equation can also be solved by forming a recurrent relationship.

Earliest Deadline First

Earliest Deadline First (EDF) was introduced by Liu and Layland [70] in 1973. According to the EDF algorithm, an arrived task with the earliest absolute deadline will be executed first. The EDF algorithm is proved to be optimal among all scheduling algorithms on a uni-processor for periodic and sporadic tasksets. This factor makes the EDF algorithm a common dynamic priority scheduling algorithm for real-time systems.

Liu and Layland [70] provide a necessary and sufficient schedulable analysis for cases when all tasks relative deadlines are equal to their periods ($D = T$). The schedulability condition is that the total utilization of the taskset should be less than or equal to 1. For periodic tasks which have arbitrary deadlines, Leung and Merrill [67] propose that a taskset is schedulable if and only if all absolute deadlines can be met in the busy period ($[0, maxs_i + 2H]$, where s_i is the start time of task τ_i and H is the least common multiple of the task periods). Later on, Baruah et al. [17] extend the algorithm to fit sporadic task systems. They show that the taskset is schedulable if and only if for all time lengths t, the maximum execution time requirement ($h(t)$) of all tasks, which have both their arrival times and their deadlines within a contiguous interval of length t, shall be smaller than or equal to the time length t. The mathematical expression can be shown as:

$$\forall t > 0, h(t) = \sum_{i=1}^{n} max\left\{0, 1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor C_i \right\} \leq t \tag{2.6}$$

Based on this equation, an upper bound of the value t can be calculated to be used for testing schedulability of the system. The original algorithm (PDA) needs to test all of the values of t from 0 to the upper bound which requires quite a large load of calculation. Zhang and Burns [89] introduce a new algorithm (QPA) which can dramatically reduce the calculation load while keeping the testing necessary and sufficient.

2.3 Uni-processor Mixed Criticality Systems

A mixed-criticality system can be defined as a finite set of components. Each component has a level of criticality, L, and contains a finite set of sporadic tasks. Each task, τ_i, has its period T_i, deadline D_i, worst case execution time C_i and criticality level L_i. There exist a number of different models to explain the relationships between the attributes of the tasks and the criticality level of the tasks. In Vestal's model [86], he assumes that the higher the degree of assurance required, the larger the task execution time needed to guarantee the completion of the task. That is, if a task τ_i has a set of assure level (criticality levels), $L = 1, 2, 3, 4$ with 4 being the highest, then the WCET estimations for task τ_i shall have the relationship as $C_i(L_1) \leq C_i(L_2) \leq C_i(L_3) \leq C_i(L_4)$. Baruah and Burns [12] extend Vestal's model by thinking that not only the execution time but also the minimum interval period and deadline may be changed in order to meet the need for guaranteeing the completion of the task. With more concerns in practical, such as hardware failures, Ekberg and Yi [49] extends Vestal's model by implementing the model with a Directed Acyclic Graph (DAG). But due to the nature of a DAG, the mode change cannot be reversed.

This thesis uses Vestal's original model that only WCET estimations of the tasks may be changed during mode changes. In addition, the main parts of the thesis considers MCS with two criticality levels, also known as dual-criticality systems. In such a system, two criticality levels are named as HI and LO, with HI more critical than LO. Tasks with HI criticality level are named as HI-crit tasks while tasks with LO criticality level are named as LO-crit tasks.

2.3.1 Fixed Priority Scheduling

Regarding to fixed priority scheduling in MCS, since tasks may have different WCET, RMPA and DMPA are no longer optimal for such tasksets. However, Vestal [86] proves that the use of AOPA is still optimal for MCS, and this proof is formalised by Dorin et al. [47]. Based on that, a variety of scheduling approaches are developed.

Vestal's Approach (SMC-no)

Vestal [86] firstly introduces a new way of using Audsley's algorithm to perform his approach to MCS in 2007. According to Vestal's approach, priorities of high and low criticality tasks are able to be interleaved in order to provide flexibility in scheduling. However, interleaved tasks need to be considered as if they are of the same criticality level. So the response time analysis equation for Vestal's approach can be written as:

$$R_i = C_i(L_i) + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(L_i) \tag{2.7}$$

Although Vestal's approach does not require any runtime monitors, it can be quite expensive to perform as several low criticality level tasks must be calculated with their parameters at the high criticality level.

Static Mixed Criticality (SMC-run)

Baruah and Burns [12] analyse Vestal's approach and indicate that the pessimistic of Vestal's approach is mainly caused by the criticality inversion. Criticality inversion happens when a LO-crit task has a higher priority than a HI-crit task. In this case, Vestal's original approach will use the HI-crit WCET estimated values of the LO-crit tasks to calculate the interferences. Baruah and Burns indicates that such problems can be eliminated with help from a runtime monitor. According to their extended model, when calculating the interference of task τ_j for task τ_i, three cases need to be considered:

- If $L_i = L_j$, then $C_j(L_j)$ shall be used as the tasks are at the same level of criticality.
- If $L_i < L_j$, then $C_j(L_i)$ shall be used since the lower level of assurance is needed for task τ_i.
- If $L_i > L_j$, then criticality inversion is spotted. If $C_j(L_i)$ is used, than the algorithm works as Vestal's original model. If $C_j(L_j)$ is used, then task τ_j needs to be guaranteed that it shall not execute for more than this value, which can be achieved by using a run-time monitor.

Task	C(LO)	C(HI)	T	D	L	P
τ_1	10	16	24	24	HI	4
τ_2	1	-	6	6	LO	1
τ_3	1	-	8	8	LO	2
τ_4	1	-	12	12	LO	3

Table 2.2: AMC Example Taskset

Thus, with the help of the run-time monitor, the response time analysis for SMC-run can be written as:

$$R_i = C_i(L_i) + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(min(L_i, L_j)) \tag{2.8}$$

Adaptive Mixed Criticality (AMC-rtb)

Adaptive Mixed Criticality is a further extension based on SMC-run by increasing the usage of the run-time monitor. The main idea in AMC is to abandon LO-crit tasks in order to save HI-crit tasks. In this way, some cases not schedulable in SMC will be schedulable in AMC [13]. For example, considering the example taskset in Table 2.2. It is evident that neither task τ_2 nor task τ_3 can be assigned the lowest priority as WCET of task τ_1 is larger than their deadlines. If task τ_4 is assigned the lowest priority, then the sum of the WCET of other three tasks is already equal to the deadline, which indicates that task τ_4 will miss its deadline. Therefore, only task τ_1 may be assigned the lowest priority. Based on SMC-run, the RTA for τ_1 will be:

$$R_1 = 28 = 16 + \left\lceil \frac{28}{6} \right\rceil 1 + \left\lceil \frac{28}{8} \right\rceil 1 + \left\lceil \frac{28}{12} \right\rceil 1 = 16 + 5 + 4 + 3 \tag{2.9}$$

Since $R_1 = 28 > 24 = D_1$, the example seems not to be schedulable for SMC. But if task τ_2, τ_3 and τ_4 are abandoned when τ_1 finishes its LO-crit execution bound, then the response time analysis for task τ_1 will change to:

$$R_1' = 18 = 10 + \left\lceil \frac{18}{6} \right\rceil 1 + \left\lceil \frac{18}{8} \right\rceil 1 + \left\lceil \frac{18}{12} \right\rceil 1 = 10 + 3 + 3 + 2 \tag{2.10}$$

Therefore task τ_1 can execute for $C_1(HI) - C_1(LO) = 16 - 10 = 6$ time unit to reach its HI-crit WCET. Thus, the total response time for task τ_1 will be $18 + 6 = 24 \leq D_1$, which indicates that the example is schedulable.

The original AMC is designed for a dual-criticality system(tasks are either in LO-crit or in HI-crit), and the rules for AMC can be viewed as [13]:

- There is a criticality level indicator, Γ, initialized to LO.
- While the indicator Γ remains in LO ($\Gamma \equiv LO$), all tasks are executed according to general priority order(start with the highest).
- If a executing task is monitored not finishing after using up its LO-crit budget, then the criticality level indicator will be set to HI ($\Gamma \leftarrow HI$).
- Once the indicator is HI ($\Gamma \equiv HI$), LO criticality level tasks will be stopped and forbidden from releasing, while HI criticality level tasks will keep on executing.
- Detect the circumstances to reset the criticality level indicator to LO($\Gamma \leftarrow LO$).

According to the above rules, the schedulable test for AMC consists of three phases of analysis. The first phase is to verify the schedulability of LO-crit mode, when all of the tasks are executing with their LO-crit budgets. The response time analysis for this phase is shown in equation (2.11).

$$R_i(LO) = C_i(LO) + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i(LO)}{T_j} \right\rceil C_j(LO) \tag{2.11}$$

The second phase is to verify the schedulability of HI-crit mode, when only HI-crit tasks are executing and execute with their HI-crit budgets. The response time analysis for this phase is shown in equation (2.12) where $hpH(i)$ stands for the set of HI-crit tasks with higher priority than that of task τ_i.

$$R_i(HI) = C_i(HI) + \sum_{\tau_j \in hpH(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(HI) \tag{2.12}$$

The change in criticality level has a number of similarities to systems that move between different operational modes (although there exists some significant differences [28],[56]). The literature on mode change protocols [84],[62] highlights one important problem that a system can be schedulable in every mode, but not schedulable during a mode change. Thus, the third phase is to check the schedulability of the progress of criticality change. Since exact analysis of this phase is unlikely to be tractable [13], a sufficient analysis can be carried out by assuming that HI-crit tasks execute in their HI-crit budgets while LO-crit tasks execute in their LO-crit budgets before the system changes to HI-crit mode. In this case, for HI-crit task τ_i, interferences from HI-crit tasks will not be affected by changing the time when system enters HI-crit mode, but interferences from LO-crit tasks will increase if that time increases. So $R_i(LO)$, the time that τ_i finishes all its LO-crit budget, is the latest time point the criticality change may occur. In all, the response time analysis for this phase is shown in equation (2.13) where $hpL(i)$ stands for the set of LO-crit tasks with higher priority than that of task τ_i.

$$\begin{aligned} R_i(HI)' = &C_i(HI) + \sum_{\tau_j \in hpH(i)} \left\lceil \frac{R_i(HI)'}{T_j} \right\rceil C_j(HI) \\ &+ \sum_{\tau_k \in hpL(i)} \left\lceil \frac{R_i(LO)}{T_k} \right\rceil C_k(LO) \end{aligned} \tag{2.13}$$

Note $R_i(HI)' > R_i(HI)$ so that if the task is deemed schedulable with $R_i(HI)'$, it is deemed to be schedulable with $R_i(HI)$.

AMC-max

The AMC algorithm is further improved by checking all the possible time points to initiate the criticality change rather than just using the point when HI criticality task used up its LO mode execution budget. Assume a criticality mode change is triggered by some task, say τ_s, executing for more than its $C_s(LO)$ budget occurs at some arbitrary time s. If this event impacts on task τ_i then $s < R_i(LO)$, and the priority of τ_s must be equal to or greater than that of task τ_i; otherwise task τ_i will have completed before the criticality change happens. Based on this, the response time analysis of task τ_i can be seen as equation (2.14), where $I_s(LO)$ represents the interference suffers from LO-crit tasks with higher priorities, $I_s(HI)$ represents the interference suffers from HI-crit tasks with higher priorities.

$$R_{i,s} = C_i(HI) + I_s(LO) + I_s(HI) \tag{2.14}$$

For LO-crit tasks, according to AMC rules, they are prevented from executing after time s. Their interference time can be bounded by their maximum release, which is $\lceil \frac{s}{T} \rceil$. However, in order to include interference from tasks as soon as they are released, $\lfloor \frac{s}{T} \rfloor + 1$ is a safer choice. Thus, the worst-case interference of LO-crit tasks is:

$$I_s(LO) = \sum_{j \in hpL_i} (\left\lfloor \frac{s}{T} \right\rfloor + 1) C_j(LO) \tag{2.15}$$

The interferences from HI-crit tasks can be differentiated into two cases according to their priorities. Those with priorities greater than τ_s must have completed their latest release before s, while those with priorities equal to or less than τ_s may not yet completed. Hence, in this case, their latest release may need $C(HI)$. In all, only jobs with a deadline before s contribute a $C(LO)$ value.

Consider the interference from task τ_k (has a higher priority) at time t with $t > s$. The maximum number of releases of task τ_k is $\lceil \frac{t}{T_k} \rceil$, and the maximum number of releases before s is $\lceil \frac{s}{T_k} \rceil$. Thus, the maximum number of releases that can fit into the HI-crit mode period is $\lceil \frac{t-s}{T_k} \rceil + 1$ (when $T = D$). For cases with $D < T$, this maximum number can be improved by reducing an interval in which t_k is not executing (time between its deadline and the next release):$\lceil \frac{t-s-(T_k-D_k)}{T_k} \rceil + 1$. But if s is small and D_k is close to T_k, this improved value may include more jobs than actually presented in the interval which makes the value pessimistic. So the maximum number of releases of task k can be further improved to be:$M(k, s, t) = min \lceil \frac{t-s-(T_k-D_k)}{T_k} \rceil + 1, \lceil \frac{t}{T_k} \rceil$.

Therefore, the analysis equation for AMC-max is:

$$\begin{aligned} R_{i,s} = & C_i(HI) + \sum_{j \in hpL_i} (\left\lfloor \frac{s}{T} \right\rfloor + 1) C_j(LO) + \\ & \sum_{k \in hpH_i} M(k, s, R_{i,s}) + (\left\lceil \frac{t}{T_k} \right\rceil M(k, s, R_{i,s}) C_k(LO)) \end{aligned} \tag{2.16}$$

The last step is to define the values of s that need to be considered. The possible value range for s is $[0, R_i(LO))$. By examining the equation, if s increases, the LO-crit task part will increase while the HI-crit task part will decrease, and $R_{i,s}$ will only increase at the points when a LO-crit task is released. Thus, only the time points, when a LO-crit task is released, need be checked. Although s is restricted to the interval $[0, R_i(LO))$ and only certain values need be explored, this can be a large number for a sizeable application. Accoridng to that, AMC-max may be time consuming to calculate in certain cases.

Take Table 2.2 as an example. As computed before, the response time for task τ_1 in LO-crit mode is 18. During this time period, task τ_2 has been released 3 times, task τ_3 has been released 3 times, and task τ_4 has been released twice. Based on that, time points $t = 6, 8, 12, 16$ need to be checked:

- $R_{1,6} = 16 + (1 + 1 + 1) + 0 = 19$
- $R_{1,8} = 16 + (2 + 1 + 1) + 0 = 20$
- $R_{1,12} = 16 + (2 + 2 + 1) + 0 = 21$

- $R_{1,16} = 16 + (3 + 2 + 1) + 0 = 22$

It has a solution of 22 at the worst case, which is smaller than that of 24 based on AMC-rtb.

Evaluation of FPS

Baruah et al. [13] have produced an experiment to compare the scheduling efficiency of the approaches introduced above. They also compare the algorithms with a upper bound and a lower bound.

- UB-H&L: A composite upper bound of UB-L and UB-H test. UB-L is an upper bound on taskset schedulability obtained by considering execution of all tasks at the LO-crit level and using DMPA. UB-H is an upper bound based solely on the schedulability of the HI-crit tasks executing with their HI-crit budgets and using DMPA.
- AMC-max: Approach 4 described in Section 2.3.1
- AMC-rtb: Approach 3 described in Section 2.3.1
- SMC-run: Approach 2 described in Section 2.3.1
- SMC-no: Approach 1 described in Section 2.3.1
- CrMPO: Criticality Monotonic Priority Ordering. Task priorities are ordered first according to criticality levels (HI-crit first) and then according to deadlines (shortest deadline first). Response time analysis is used to determine if the taskset is schedulable with HI-crit tasks executing with HI-crit budgets and LO-crit tasks executing with LO-crit budgets.

Their comparing results can be seen in Figure 2.2. According to the figure, it can be observed that SMC-run outperforms SMC-no by a large margin which indicates that the usage of a run-time monitor significantly improves schedulability. It is also observed that AMC-rtb has a further improvement on the performance of SMC-run and AMC-max has a slightly better performance than AMC-rtb. Based on that results, AMC-max provides the best performance than all other algorithms. However, as stated in the previous section, AMC-max may requires too much calculation in certain cases while the performance improvement from AMC-rtb is quite small. Considering this payoff, AMC-rtb is recommended as the most appropriate algorithm in FPS on uni-processor platforms.

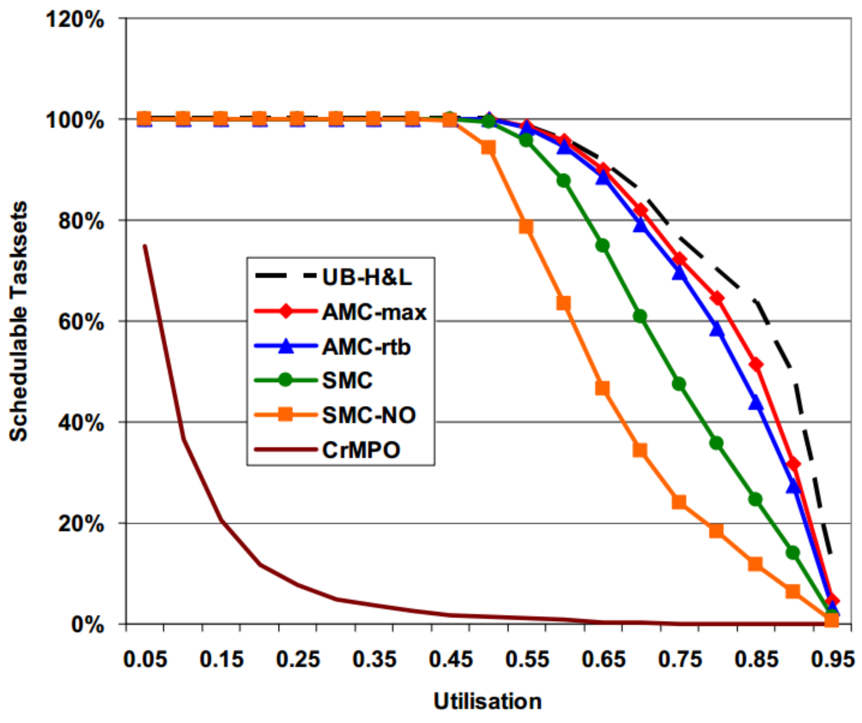


Figure 2.2: FPS MCS Comparison [13]

The AMC-rtb approach is extended by Zhao et al. ([90],[91]) in 2013 to incorporate preemption thresholds [80] into the model. They demonstrate a reduction in stack usage and improved performance for some parameter ranges. Burns and Davis [31] introduce another approach to combine AMC-rtb with the usage of deferred preemption ([27],[42]), and demonstrate a significant improvement over fully preemptive AMC-rtb. The improvement they observed is obtained by having a final non-preemptive region at the end of each criticality and by combining the assignment of priority with the determination of the size of these regions.

Fleming and Burns [52] extend the models to an arbitrary number of criticality levels. They observed that AMC-rtb remains as good approximation to

AMC-max, and AMC-max becomes computational expensive for increased levels. According to their model, LO-crit tasks will be aborted, MID-crit and HI-crit tasks will execute with MID-crit budgets. A sufficient response time analysis for this AMC criticality mode change is represented in equation (2.17), where $hpM(i)$ stands for the taskset that contains all the MID-crit tasks which have higher priority than tasks τ_i and $R_i(LO)$ stands for the response time of the task when the system is in LO-crit mode. This equation is later used in Chapter 3.

$$\begin{aligned} R_i =& C_i(MID) + \sum_{j \in hpH(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(MID) \\ &+ \sum_{k \in hpM(i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k(MID) \\ &+ \sum_{l \in hpL(i)} \left\lceil \frac{R_i(LO)}{T_l} \right\rceil C_l(LO) \end{aligned} \tag{2.17}$$

2.3.2 Earliest Deadline First

In general uni-processor scheduling, Earliest Deadline First (EDF) has better performance than Fixed Period Scheduling (FPS) due to its high utilization bound [70]. Several works have been done on applying EDF to MCS. Baruah and Vestal [18] first considered MCS with EDF scheduling in 2008. Park and Kim [77] later introduce a slack-based mixed criticality scheme. They propose a combination usage of on-line and off-line analysis to run HI-crit jobs as late as possible while LO-crit jobs executing in generated slack. Ekbery and Yi ([49],[57]) propose a more complete analysis for EDF scheduled MCS.

Demand-bound EDF

In Ekbery and Yi's model ([49],[57]), the system has two behaviour modes and each HI-crit task has been assigned two relative deadlines. One deadline is the original deadline of the task, the other is an artificial earlier deadline that is used to increase the likelihood of HI-crit tasks executing before LO-crit tasks. In the LO-crit mode, all LO-crit tasks execute normally, while HI-crit tasks execute with a smaller deadline (comparing to their original deadlines). When a HI-crit task is detected to execute exceeding its LO-crit budget, the system will change from LO-crit mode to HI-crit mode. In the HI-crit mode, all LO-crit tasks will be abandoned and HI-crit tasks will execute with their original deadlines.

In their later work ([58],[50]), the original scheme is improved and extended to include changes to all task parameters and to incorporate more than two criticality levels. Easwaran [48] provides a tighter analysis, focuses on the dual-criticality platform, and Yao et al. [39] improves the model by using an improved schedulability test for EDF (a scheme called QPA [89]) and genetic algorithm (GA) to find better artificial deadlines.

Virtural Deadline (EDF-VD)

Baruah et al. ([11],[9],[10]) introduce a similar scheme, named EDF-VD. EDF-VD is designed for dual-criticality systems and tasks with the same period and deadlines ($T = D$). Since the execution order of the tasks is controlled by the absolute deadlines in EDF, EDF-VD uses a factor x to reduce the deadlines of all tasks. With this factor, D'_i is calculated prior to runtime for each HI-crit task. HI-crit tasks will execute with a smaller deadline by general EDF scheduling with LO-crit tasks unless some task executes beyond its budget. In detail, if a task τ_i arrives at time-instant t:

- If $L_i = LO$, then τ_i will be assigned a deadline equal to $t + D_i$
- If $L_i = HI$, then τ_i will be assigned a virtual deadline equal to $t + D'_i$

When a certain task is detected executing over the budget, then all of the LO-criticality tasks need to be discarded immediately while HI-criticality tasks will execute with their normal deadlines. Based on that, a taskset S will only be schedulable for EDF-VD if two requirements are met:

- All tasks will be schedulable in its LO-criticality behaviour when the parameters of HI-crit tasks are scaled.
- All HI-crit tasks are schedulable with their original parameters when a budget overrun is detected.

They demonstrate both theoretically and via evaluations that EDF-VD is an effective scheme. In later work, Baruah [8] has generalised the MCS model to include criticality-specific values for period and deadline as well as WCET.

2.3.3 Other Approaches

Aiming to develop an efficient scheme that can be used at run-time, Masrur et al. [73] propose an intermediate approach based on EDF-VD and Demand-bound EDF. In their approach, two scaling factors are used to adjust the deadlines of tasks.

Lipari and Buttazzo [69] propose a reservation-based approach to address EDF scheduling of MCS with two criticality levels. In their model, deadlines for the HI-crit tasks are chosen to guarantee the execution of HI-crit tasks and to maximise the amount of capacity which can be reclaimed when HI-crit tasks execute with their LO-crit budgets. In effect, LO-crit tasks run in capacity reclaimed from HI-crit tasks. Their approach reserves sufficient budget for HI-crit tasks, but if these HI-crit tasks only execute with their LO-crit budgets, a set of LO-crit tasks can be guaranteed.

Su at al. ([82],[83]) introduce a different approach to use spare capacity. They take advantage of the elastic task model [36] in which the period of a task can change. They propose a minimum level of service for each LO-crit task τ_i that is defined by a maximum period, T_i^{max}. The system is only schedulable when all HI-crit tasks use their HI-crit budgets and all LO-crit tasks use their LO-crit budgets and T^{max} values. At run-time, if HI-crit tasks use their LO-crit budgets, then the LO-crit tasks can run more frequently. They demonstrate that their approach performs better than EDF-VD under certain scenarios.

EDF Evaluation

Since EDF will not be used in this thesis, we will not consider further details of the EDF algorithms. The reason we do not use EDF is based on two considerations. The first consideration is that EDF belongs to the class of dynamic scheduling, which provides less execution predictability than static scheduling. If the HI-crit tasks are safety-critical, EDF can not be a preferable choice. The second consideration is that EDF algorithms use deadlines of tasks to manipulate the execution order, and both of the algorithms introduced above modify the deadlines of HI-crit tasks to increase their execution priorities. However, we have observed that migrating a task will cause reduced deadline issues (which will be stated in the next chapter), and these migratable tasks are LO-crit tasks. In that case, these migrated LO-crit tasks may affect the execution order of the accepting core, which may cause the HI-crit tasks on that core become non-schedulable. In all, although EDF is a good scheduling apporach, this thesis is focused on FPS.

2.3.4 Comparing FPS and EDF

Although EDF dominates FPS in general uni-processor scheduling, Vestal has proved that it is not applicable for Mixed Criticality System (MCS), as there exists MCS examples which can be scheduled by FPS but not by EDF [18]. An example is provided in Table 2.3.

Task	C(LO)	C(HI)	T	D	L
τ_1	1	2	4	4	HI
τ_2	5	-	7	7	LO

Table 2.3: FPS VS EDF

According to the table, τ_1 is a HI-crit task while τ_2 is a LO-crit task. For FPS, since the execution time for τ_2 is larger than the deadline of τ_1, τ_2 will be assigned the lowest priority. As the HI-crit task has the highest priority, it will always meet its deadline. While the response time for τ_2, when all tasks are using their LO-crit values, is 7, which equals to its deadline, so τ_2 will also meet its deadline. Therefore, this taskset is schedulable for FPS. For EDF, if τ_1 exceeds its LO-crit budget at the first release, then τ_2 will execute for the next five time units. For the second release of τ_1, which starts at the time unit 8, it will miss its deadline. In addition, static scheduling (such as FPS) provides better execution prediction than dynamic scheduling (such as EDF). Thus, FPS is preferred to EDF in MCS, and it is the scheme used in this thesis.

2.4 Multi-core Platforms

As embedded applications become more and more complicated, embedded system designers rely more on multi-core platforms to obtain high computing performance [38], [88]. In addition, Pollack's Rule [23] indicates that the performance increase benefits are only the square root of the increase in core complexity in a uni-core platform. For example, only 40% more performance is delivered when the logic gates in a processor core are doubled. Due to this constraint, the industry is changing its gear toward multi-core architectures rather than continuing to pursue high performance under the single processor architecture.

There are extensive researches published on real-time scheduling for homogeneous multi-core systems ([4], [3], [63], [66]). These scheduling algorithms can be largely categorized into three classes: global scheduling (e.g. [3]), partitioned scheduling (e.g. [66]) and semi-partitioned scheduling (e.g. [63]). In partitioned scheduling, each task is assigned to a dedicated processor and executes solely on that particular processor. According to that, the problem of multi-processor scheduling is reduced to a set of uni-processor ones after tasks are partitioned. In addition, partitioned scheduling does not lead to job migration costs which can influence the schedulability of the system. However, it has been shown that in worst cases, all partitioned scheduling approaches may cause deadlines to be missed if the system utilization exceeds 50% ([72], [63]).

In global scheduling, all tasks first enter a global queue, and then execute on any available processors in order. Thus, in global scheduling, each task can potentially execute on any processor. This scheduling approach contains optimal algorithms, such as Pfair [15], [14] and LLREF [41]. Any periodic taskset is schedulable by those algorithms if the utilization of the taskset does not exceed 100%. However, Carpenter et al. [37] have compared the partitioned scheduling and the global scheduling, and have indicated that both scheduling approaches have their own pros and cons and none of them dominates the other in terms of schedulability. In addition, since fixed-priority algorithms are often adopted by commodity real-time operating system for practical usage, global scheduling provides few advantage over partitioned scheduling from the viewpoint of fixed-priority algorithms. To the best of our knowledge, existing fixed-priority algorithms based on global scheduling ([4], [3], [7]) also have a taskset utilization bound no greater than 50%.

Semi-partitioned scheduling can be seen as an intermediate solution between the partitioned and global scheduling approaches. The basic idea with semi-partitioned scheduling is to execute tasks according to a static job migration pattern with the goal to reduce run-time overheads by controlling the number of job migrations. In classical semi-partitioned scheduling, tasks are firstly assigned, as much as possible, to single processors according to the partitioned scheduling approach. The jobs of the tasks that cannot be assigned to a single processor are then allowed to migrate between a fixed set of particular processors. According to George et al. [54], there exist two main approaches to assign the migrating jobs: job portion migration and job migration.

Kato et al. [63] first propose the idea to split tasks in the semi-partitioned approach. Their approach splits a task into several portions and each portion of the task is assigned to a dedicated processor. They propose a portioning heuristic that minimizes the number of processors required to execute a task by assigning the maximum possible duration to the WCET of a portion while preserving the schedulability of the task. By splitting tasks, the overall system utilization can be significantly improved. As stated before, the best known utilization bound for either global or partitioned fixed-priority scheduling algorithms is no more than

50%. For the semi-partitioned scheduling, Lakshmanan et al. [66] have shown an utilization bound of 65%, and Guan et al. [59] have improved this bound to the traditional Liu and Laylands bound, i.e. 69.3%. However, above approaches are built with the assumption that no constraint is imposed when splitting tasks. In practice, the programming code and the resources used by a task are highly likely to influence the number of portions the task can be split into.

The other approach, referred to as the restricted migration case by Funk and Baruah [53], only allows a job to execute on one processor for each release while different jobs of the same task can be executed on different processors. Although this approach does not provide high utilization bound as the previous one, this approach avoids the problems caused by splitting jobs. In addition, according to George et al. [54], this migration pattern introduces fewer overheads than the job portion migration approach as migrations are only done at job boundaries (at most one migration per job). In theoretical analysis, it is common to assume that job migrations take zero time. But in practice, several activities (acquiring locks, making a scheduling decision, performing a context switch, etc.) need to be performed before the migration of a job and such activities have a cost [20]. Thus, in terms of practice, the restricted migration approach is preferred.

2.5 Multi-core MCS

Multi-processor systems can be divided into three types: heterogeneous, homogeneous and uniform [44]. In order to simplify the system model, most of the algorithms in MCS are using homogeneous multi-core platforms, where the processors are identical and the execution time for tasks remains the same on all processors. So if not specified, the multi-core systems mentioned in this thesis are homogeneous.

2.5.1 Task Allocation

Although there exist three types of multi-core scheduling algorithms, most papers published are using partitioned scheduling. The reason is that partitioning provides a stable and predictable implementation that is preferable for safety critical applications. Regarding partitioned scheduling, the most essential issue is to allocate tasks to cores.

According to Michael and Johnson [74], unless $P = NP$, only heuristic solutions can be gained for scheduling a set of n real-time tasks on m processors even when the tasks have identical criticality levels and share the same release time and deadline ($D = T$). By importing extra criticality levels, the problem gets more complicated which implies that the partitioning problem mentioned above will also be NP-Hard. Bin-packing algorithms, include First-Fit (FF), Best-Fit (BF) and Worst-Fit (WF), have been applied to the multi-processor scheduling of general real-time system [71], and can be extended to be applied in the mixed-criticality context.

First-Fit (FF)

First-Fit allocates the task to the first processor on which it "fits". The word "fit" here means that the task can be successfully scheduled along with the other tasks that are already allocated to that processor. For example, consider the taskset contained in Table 2.4 which needs to be allocated to a 3-core platform (cores c_1, c_2 and c_3). The allocation manipulated by the First-Fit algorithm will be as following:

1. Schedule task τ_1 to core c_1. Success. Allocate task τ_1 to core c_1.
2. Schedule task τ_2 to core c_1. Fail.
3. Schedule task τ_2 to core c_2. Success. Allocate task τ_2 to core c_2.
4. Schedule task τ_3 to core c_1. Success. Allocate task τ_3 to core c_1

Task	C	T	D
τ_1	5	10	10
τ_2	6	10	10
τ_3	4	10	10
τ_4	5	10	10

Table 2.4: Bin Packing Example

5. Schedule task τ_4 to core c_1. Fail.

6. Schedule task τ_4 to core c_2. Fail.

7. Schedule task τ_4 to core c_3. Success. Allocate task τ_4 to core c_3.

According to the allocation results, task τ_1 and τ_3 are assigned to core c_1, task τ_2 is assigned to core c_2, and τ_4 is assigned to core c_3. Thus, it can be observed that First-Fit algorithm tends to make better usage of the first few cores rather than giving an average allocation. In addition, First-Fit algorithm is quite straightforward if the scheduling process is quite simple.

Best-Fit (BF)

Best-Fit algorithm allocates the task to the processor with the smallest unused capacity among all of the processors on which it fits. So the performance of BF is quite different from FF as it needs to check the schedulability state of each core. Still consider the taskset contained in Table 2.4, if this taskset is allocated to a 3-core platform (cores c_1, c_2 and c_3), then BF will allocate the tasks as following:

1. Schedule task τ_1

   to core c_1. Success. The core has utilization of 0.5.

   to core c_2. Success. The core has utilization of 0.5.

   to core c_3. Success. The core has utilization of 0.5.

2. Since allocating task τ_1 to any core provides the same core utilization, allocating task τ_1 to any core is fine. In this case, allocate task τ_1 to core c_1.

3. Schedule task τ_2

   to core c_1. Fail.

   to core c_2. Success. The core has utilization of 0.6.

   to core c_3. Success. The core has utilization of 0.6.

4. Since allocating task τ_2 to core c_2 and core c_3 provide a same core utilization, allocating task τ_2 to any available core is fine. In this case, allocate task τ_2 to core c_2.

5. Schedule task τ_3

   to core c_1. Success. The core has utilization of 0.9.

   to core c_2. Success. The core has utilization of 1.0.

   to core c_3. Success. The core has utilization of 0.6.

6. Since allocating task τ_3 to core c_2 provides the highest utilization, task τ_3 is allocated to core c_2.

7. Schedule task τ_4

   to core c_1. Success. The core has utilization of 1.0.

   to core c_2. Fail.

   to core c_3. Success. The core has utilization of 0.5.

8. Since allocating task τ_4 to core c_1 provides the highest utilization, task τ_4 is allocated to core c_1.

As it is revealed from the allocation results, task τ_1 and τ_4 are assigned to core c_1, task τ_2 and τ_3 are assigned to core c_2, and no task is assigned to core c_3. Based on the allocation results, Best-Fit provides better usage of the computing ability of each core, and tends to provide a constrictive distributed allocation.

Worst-Fit (WF)

Worst-Fit allocates the task to the processor with the largest unused capacity. This algorithm also needs to check the schedulability state on each core, which performs similarly to BF but towards a totally opposite target. Still consider the taskset contained in Table 2.4, if this taskset is allocated to a 3-core platform (cores c_1, c_2 and c_3), then WF will allocate the tasks as following:

1. Check the utilization of all of the cores:

   core c_1 has utilization of 0.

   core c_2 has utilization of 0.

   core c_3 has utilization of 0.

2. The unused capacity for each core is the same. Schedule task τ_1 to core c_1. Success. Allocate task τ_1 to core c_1.

3. Check the utilization of all of the cores:

   core c_1 has utilization of 0.5.

   core c_2 has utilization of 0.

   core c_3 has utilization of 0.

4. Core c_2 and c_3 have the largest unused capacity. Schedule task τ_2 to core c_2. Success. Allocate task τ_2 to core c_2.

5. Check the utilization of all of the cores:

   core c_1 has utilization of 0.5.

   core c_2 has utilization of 0.6.

   core c_3 has utilization of 0.

6. Core c_3 has the largest unused capacity. Schedule task τ_3 to core c_3. Success. Allocate task τ_3 to core c_3.

7. Check the utilization of all of the cores:

core c_1 has utilization of 0.5.

core c_2 has utilization of 0.6.

core c_3 has utilization of 0.4.

8. Core c_3 has the largest unused capacity. Schedule task τ_4 to core c_3. Success. Allocate task τ_4 to core c_3.

According to the allocation results, task τ_1 is assigned to core c_1, task τ_2 is assigned to core c_2, and task τ_3 and τ_4 are assigned to core c_3. It can be observed that Worst-Fit algorithm tends to provide an average distributed allocation.

Task Sorting

It is known that the order of a taskset will affect the performance of bin-packing algorithms and it has been shown that decreasing utilization order improves the performance of bin-packing algorithms in general real-time systems [71]. It is reasonable to assume that the decreasing utilization order would also be helpful in the MCS context. In addition, as criticality level is included, the order of criticality may also have influence on the performance of the algorithms. Kelly et al. [64] compare the efficiency of bin-packing algorithms among different taskset orders based on utilization and criticality. In their experiment, they name two taskset orders as Decreasing Utilization (DU) and Decreasing Criticality (DC). Regarding to DU, tasks with high utilization values are allocated first. However, because each MCS task is associated with multiple utilization values, such an ordering requires a single utilization value to be identified for each task. In their experiment, they use a nominal utilization ($u_i(L_i)$ to represent the value of the task‘s utilization at the specific criticality level of the task) for each task. Regarding to DC, tasks are ordered according to criticality and tasks at the same criticality level are further ordered by decreasing nominal utilization. After tasks are allocated, they use RM Priority Assignment and Audsleys Optimal Priority Assignment to assign the task priority on each processor for different taskset orders: RM for DU, Audsleys for DC.

Task	C(LO)	C(HI)	u(LO)	u(HI)	T	D	L
τ_1	4	16	0.2	0.8	20	20	LO
τ_2	12.5	17.5	0.25	0.35	50	50	HI
τ_3	12	18	03	0.45	40	40	HI
τ_4	4	5	0.4	0.5	10	10	LO

Table 2.5: DC-Audsley VS DU-RM 1

According to their analysis, they find tasksets that are successfully scheduled by either DU-RM or DC-Audsley but not both. Take Table 2.5 as an example taskset. Regarding to the first example, if FF is used as the task allocation heuristic, consider DU-RM scheme, where tasks are ordered according to their nominal utilizations ($u_1(1) = 0.2$, $u_2(2) = 0.35$, $u_3(2) = 0.45$, $u_4(1) = 0.4$) and RM priorities are used, the tasks are allocated in the order of τ_3, τ_4, τ_2 and τ_1. Based on the RM scheme, τ_3 and τ_4 can be successfully assigned to core c_1. Neither τ_1 nor τ_2 fit on core c_1, which means they both need to be placed on core c_2. However, according to RM, τ_1 will be assigned higher priority, which makes τ_2 miss its deadline even at its first release (in HI-criticality mode). Therefore, this taskset is not schedulable with DU-RM. Consider DC-Audsley scheme, the tasks are allocated in the order of τ_3, τ_2, τ_4 and τ_1. Based on Audsleys scheme, τ_3 and τ_2 can be successfully assigned to core c_1, while τ_1 and τ_4 can also be successfully assigned to core c_2. Hence, this taskset is schedulable with DC-Audsley.

However, it is also observed that there exist cases that are schedulable by DU-RM but not DC-Audsley. Take the following taskset as an example (Table 2.6), if FF is used as the bin-packing heuristic, considering DC-Audsley, the tasks are allocated in the order of τ_1, τ_2 , τ_3 and τ_4. According to Liu and Layland [70], the utilization of FPS on uni-processor must be smaller than 1. Thus, if τ_1 and τ_2 are assigned to c_1, τ_3 and τ_4 can only be allocated to core c_2. As the sum of the utilization of τ_3 and τ_4 is 1, they are schedulable in EDF but not FPS. Therefore, such taskset is not schedulable with DC-Audsley. Considering DU-RM, the tasks are allocated in the order of τ_1, τ_3, τ_4 and τ_2. According to RM scheme, τ_1 and

Task	C(LO)	C(HI)	u(LO)	u(HI)	T	D	L
τ_1	14	26	0.35	0.6	40	40	HI
τ_2	24	32	0.3	0.4	80	80	HI
τ_3	66	72	0.55	0.6	120	120	LO
τ_4	180	200	0.45	0.5	400	400	LO

Table 2.6: DC-Audsley VS DU-RM 2

τ_3 can be successfully allocated to core c_1 while τ_1 is assigned higher priority. τ_4 and τ_2 can also be successfully allocated to core c_2 while τ_2 is assigned higher priority. Thus, such taskset is schedulable with DU-RM.

Kelly et al. have also explored the performance of the allocation schemes DU-RM, DU-Audsley, DC-RM and DC-Audsley. They generate different tasksets with increasing sum of the utilization and check the scheduling successful ratio of the allocation schemes. In their experiment, they fix the task number of each taskset to 40, and the number of processors to 4. According to their results (Figure 2.3), DC-Audsley outperformes other scheduling schemes throughout the experiments, which draws their conclusion that general First-Fit decreasing criticality is the best task allocation scheme for mixed criticality fixed priority scheduling. This result is used in the allocation scheme developed in this thesis.

2.5.2 Idle Tick on Multi-core

As described in the uni-processor section, several algorithms, like AMC and EDF-VD, require the system to detect an idle tick to get back to the default settings. It may not be a problem in uni-processor or partitioned architectures, but in a migrating system, it may take quite a long time to find an idle tick or even such a time instance may not exist. For example, considering the taskset in Table 2.7 under a dual-criticality level MCS. Assume that the taskset is scheduled by FPS that τ_1 has the highest priority while τ_3 has the lowest. If τ_1 and τ_3 are released at

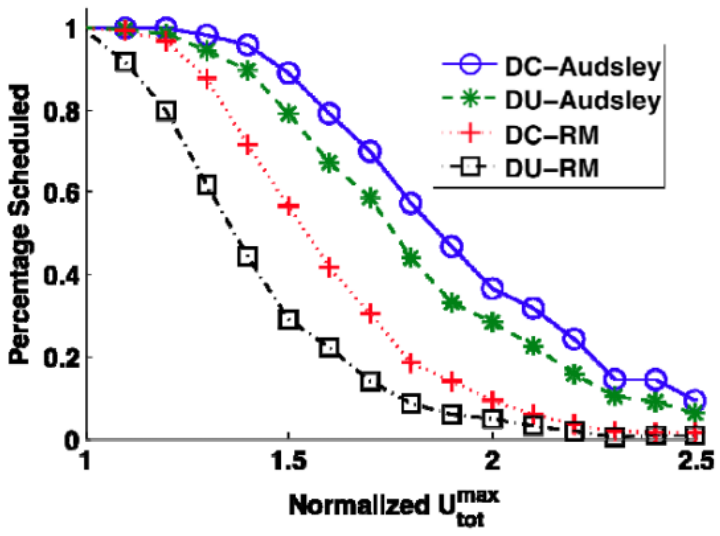


Figure 2.3: Task Sorting Algorithm Comparison [64]

Task	C(LO)	C(HI)	T	D	L
τ_1	6	9	10	10	HI
τ_2	6	9	10	10	HI
τ_3	3	-	10	10	LO

Table 2.7: No Idle Tick Taskset

time $t = 0$, τ_2 is released at time $t = 5$ and τ_1 exceeds its LO-criticality execution budget at time $t = 6$, then both τ_1 and τ_2 execute in their LO-crit budgets from their next release and the system will never become idle (Figure 2.4).

Santy et al. [81] propose a protocol to address this problem. According to their protocol, suppose an overrun occurs at time t_{over} and no job exceeds its WCET budget at criticality level L after t_{over}. For every task τ_i in the decreasing order of priority, the protocol identifies a time instant f_i satisfying:

- $f_i \leq f_{i-1}$ (with $f_0 = t_{over}$)
- τ_i has no active job at time f_i

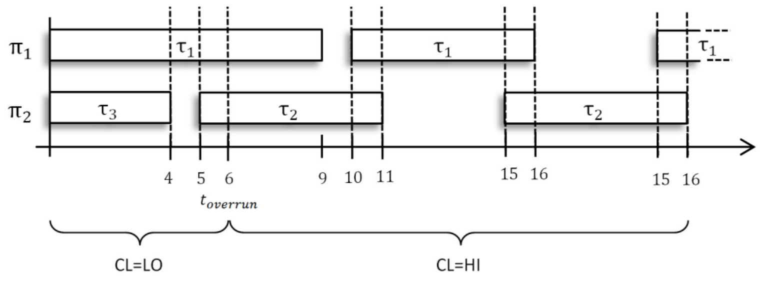


Figure 2.4: No Idle Tick [81]

Then, as soon as such an instant has been found for the lowest priority task τ_n, the criticality of the system can be decreased safely to Level L and all the suspended tasks with a criticality greater than or equal to Level L can be reactivated. In detail, (assuming tasks are numbered in priority order), at time t_{over}, the protocol checks whether task τ_1 (the highest priority task belonging to τ) has an active job $J_{1,k}$. If it is the case, then the protocol waits for that job to complete its execution, and the time when it completes will be assigned as f_1. Otherwise, $f_1 = f_{over}$. The protocol then looks for the earliest instant after f_1 where τ_2 has no active job. Again, if at time f_1, τ_2 has no active job, then $f_2 = f_1$. Otherwise, the protocol waits for the job to complete. These steps are iteratively performed for each task based on the priority order.

The protocol has an assumption that an overrun is unusual. That is, if the protocol identifies a time instant f_i for task τ_i, then the task will not overrun its WCET until the protocol is able to decrease the criticality level to Level L_i. Nevertheless, if the task τ_i is detected to exceed its execution budget before f_i, the protocol needs to be aborted and restart all over again from the beginning.

For example, considering a dual-criticality system, let the taskset $S = \tau_1, \tau_2, \tau_3, \tau_4$ consists of four implicit-deadline sporadic tasks with the parameters in Table 2.8:

Consider an identical multi-core platform (c_1 and c_2), if tasks τ_2, τ_3, τ_4 release a job at time $t = 0$, and task τ_1 releases a job at time $t = 2$. At time $t = 4$, task τ_3 does not signal its completion, which increases the criticality level of the system from LO-crit to HI-crit. Therefore, task τ_4 is suspended. At time $t = 4$, the protocol is launched and checks whether task τ_1 has an active job. Since

Task	C(LO)	C(HI)	T	D	L	P
τ_1	2	3	5	5	HI	1
τ_2	2	4	9	9	HI	2
τ_3	4	9	11	11	HI	3
τ_4	1	-	5	5	LO	4

Table 2.8: Idle Tick Example Taskset

the first job of task τ_1 is still executing, the protocol must wait for that job to finish at time $t = 5$. Since task τ_2 has finished its first release at time $t = 2$, the protocol skips τ_2 and considers τ_3 instead. At time $t = 5$, τ_3 has an active job, the protocol must wait for that job to finish, which is at time $t = 9$. By then, as τ_3 is the lowest priority task among HI-crit tasks, no other tasks in the taskset S will have an active job (τ_4 is abandoned). Thus, at time $t = 9$, the criticality of the system can be decreased to LO-crit, thereby reactivating task τ_4. The scenario is illustrated in Figure 2.5.

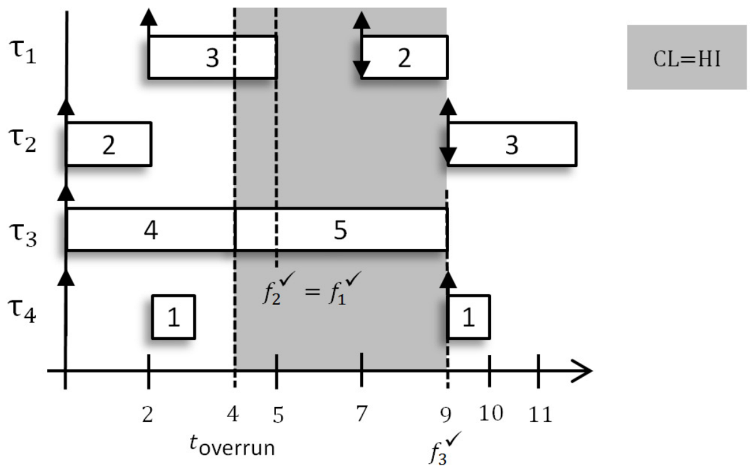


Figure 2.5: Idle Tick Example [81]

Bate et al. [22], [21] propose a more aggressive scheme for returning a system back to its LO-criticality mode. In their approach, a bailout protocol is introduced for dual-criticality systems. The protocol is mainly focused on single processor systems (but is applicable for individual cores in a partitioned multi-processor) and the fixed priority preemptive scheduling of sporadic tasks with constrained deadlines.

At run-time, dual-criticality systems are typically defined to be in one of two modes: LO-crit mode and HI-crit mode. But the bailout protocol defines three modes: normal mode, bailout mode and recovery mode. Normal mode corresponds to the traditional LO-crit mode while bailout and recovery modes correspond to the traditional HI-criticality mode. The normal mode behaves similarly to the traditional LO-crit mode. In the bailout mode, HI-crit tasks take out a loan if they execute for more than their LO-crit budgets. Other tasks repay the loan by either not executing at all or executing for less than expected. When the loan is repaid, the system enters the recovery mode. In the recovery mode, LO-crit tasks are still prevented from executing while HI-crit tasks are executing with their LO-crit budgets. If any HI-crit task executes for its LO-crit budget without signalling completion, the system will re-enter bailout mode. When the first release of the HI-crit task with the lowest priority in the recovery mode completes its execution with its LO-crit budget, the system transits to the normal mode. They demonstrate, using a scenario-based assessment, the bailout protocol returns the system to the normal mode much quicker than the 'wait for idle tick' scheme.

2.5.3 Communication among Cores

With a more complete platform such as a multi-processor or System on Chip or Network on Chip, more resources have to be shared between criticality levels. A main design issue is raised for multi-core MCS is how to ensure the behaviour of low criticality components does not adversely impact on the behaviour of higher criticality components. Thus, for a bus-based architecture, it is necessary to control access to the bus so that applications on one core do not impact unreasonably on applications on other cores. Pellizzoni et al. [78] show that a task can suffer a 300% increase in its WCET due to memory access interference even when it only spends 10% of its time on fetching from external memory on an 8-core system.

Tobuschat et al. [85] have developed a NoC explicitly to support MCS. Their IDAMC protocol uses a back suction technique [46] to maximise the bandwidth given to low (or non) critical messages while ensuring that high criticality messages arrive by their deadlines. Burns, Harbin and Indrusiak ([32],[60]) expand the wormhole routing scheme [75] for a NoC to provide support for mixed criticality traffic. An alternative to having a NoC be used for all traffic is proposed by Audsley ([6],[55]). They advocate the use of a separate memory hierarchy to link each core to off chip memory. A criticality aware protocol is used to pass request and data through a number of efficient multiplexers. If the volume of requests and data is criticality dependent then analysis similar to that used for processor scheduling can be used on this memory traffic.

As NoC is widely supported in MCS, this thesis considers NoC in the analysis developed in Chapter 5.

2.6 Semi-partitioned Scheduling

Al-Bayati et al. [1] recently introduce a dual-partitioned scheduling approach, which allows HI-crit tasks to be statically mapped to processors at all times while LO-crit tasks executing with limited migration, under the assumption that both processors will go into HI-crit mode at the same time. Their model consists of two steady modes and a migrating process. In the steady modes (LO-crit mode and HI-crit mode), tasks are fully-partitioned to each core unless a criticality change of the system is detected. During the criticality change, LO-crit tasks can be migrated to other cores to provide flexibility. Thus, a LO-crit task τ_i may have two different designated processors $c_i(LO)$ and $c_i(HI)$. In addition, Al-Bayati et al.'s model also assume that LO-crit tasks will get a decreased release frequency ($T_i(LO) < T_i(HI)$), but the WCET will remain the same ($C_i(HI) = C_i(LO)$). In all, the response time analysis for two steady modes of each core is shown in equation (2.18).

$$
\begin{aligned}
R_i(LO) =& C_i(LO) + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i(LO)}{T_j(LO)} \right\rceil C_j(LO) \\
R_i(HI) =& C_i(HI) + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i(HI)}{T_j(HI)}) \right\rceil C_j(HI)
\end{aligned}
\tag{2.18}
$$

However, their model only checks the schedualbilities of two states but omits checkings on the mode change itself, which makes their results incomplete. In addition, they make an assumption that LO-crit tasks would have increased periods in the HI-criticality mode which decreases the execution rate of these tasks. The scheme proposed in this thesis has different requirements and analysis.

2.7 Summary

This chapter firstly provides some background knowledge of real-time systems and reviews existing works upon uni-processor MCS. It concludes that AMC-rtb is the most appropriate algorithm to be extended to semi-partitioned scheduling. Then it discusses why the semi-partitioned approach is an appropriate approach for multi-core MCS. After that, it introduces three bin-packing algorithms, the DC-Audsley task sorting algorithm, and a protocol to address the idle tick issue in multi-core MCS. At the end of this chapter, it discusses the problem observed in the semi-partitioned model introduced by Al-Bayati et al. [1].

Chapter 3

Semi-partitioned Model on Dual-core platform with Two Criticality Levels

The previous chapter gives an overview of topics that are related to the scheduling algorithms in MCS. The review shows that most existing efficient algorithms are built with the assumption that tasks with lower criticality levels may be abandoned to guarantee the execution of tasks with higher criticality levels. From this chapter, we will propose an appropriate semi-partitioned approach for multi-core MCS.

This chapter, as a start of the exploration, will focus on the simplest case of MCS that there are only two criticality levels (HI-crit and LO-crit) and only two cores in the system. Due to such settings, only LO-crit tasks may migrate and there is one possible core for tasks to migrate to. Thus, the research in this chapter mainly focuses on the definition of the semi-partitioned algorithm model and the approaches to determine which tasks shall be migratable in a taskset.

3.1 Semi-partitioned Model

As stated in the motivation in Chapter 1, one of the purposes is to save the LO-crit tasks that are abandoned during mode changes on cores in other algorithms. Thus, the migration progress shall only happen when the system observes a mode change on one core. If both cores enter HI-crit mode, then none of the cores can accept migratable tasks, and these migratable tasks have to be abandoned. In consideration of the need of safety-critical applications, all HI-crit tasks need to be statically allocated to each core. Regarding LO-crit tasks, some may also be statically allocated to cores while others are allowed to migrate to other cores in order to provide flexibility for mode changes. Assume that the criticality level change of one core has no effects on the other, the basic properties of the model are as follows:

- If all tasks execute within their LO-crit budgets then all deadlines are met and no tasks migrate.
- No LO-crit task is allowed to exceed its LO-crit budget.
- If HI-crit tasks on one core exceed their LO-crit budgets, then some LO-crit tasks will migrate, but ALL LO-crit tasks and HI-crit tasks remain schedulable.
- If HI-crit tasks on more than one core exceed their LO-crit budgets, then some LO-crit tasks will be abandoned, but all HI-crit tasks remain schedulable (without migration).

For example, for a dual-core platform, the dispatching of jobs for execution occurs according to the following rules:

- Each core consists of a criticality level indicator Γ, which is initialized to LO.
- For each core, while ($\Gamma \equiv LO$), task with highest priority is selected for execution.

- If a LO-crit task executes for its LO-crit budget without signalling completion, its current release shall be terminated. If a HI-crit task executes for its LO-crit budget without signalling completion, then the criticality level indicator Γ_i for this core c_i will be changed to HI.
- Once $\Gamma \equiv HI$, if the criticality level indicator of the other core remains in LO, then all HI-crit tasks may execute with their HI-criticality budgets while some LO-crit tasks will keep executing with their LO-crit budgets and other LO-crit tasks will immediately migrate their current release onto the other core. If the criticality level indicator of the other core is HI, then all of the LO-crit tasks currently executing on the core need to be abandoned while HI-crit tasks execute within their HI-crit budgets.

By allowing LO-crit tasks to migrate, the semi-partitioned scheduling model can schedule tasksets that are not schedulable by a non-migration scheduling model. For instance, consider the taskset in Table 3.1 to be scheduled on a dual-core platform. Task τ_1 and τ_2 are HI-crit tasks and task τ_3 is a LO-crit task. Considering the HI-crit WCET and the period, task τ_1 and τ_2 cannot be both allocated to the same core. Assume that task τ_1 is allocated to core c_1 and task τ_2 is allocated to core c_2. Since task τ_1 and τ_2 have the same attributes, allocating task τ_3 to either core c_1 or core c_2 has the same effect. Assume that task τ_3 is allocated to core c_1, due to the period of task τ_3 is equal to the LO-crit WCET of task τ_1, task τ_3 needs to be assigned a higher priority. In this case, task τ_1 will first execute 1 time unit, then task τ_3 will execute 1 time unit. Task τ_3 then preempt task τ_1 and execute another 1 time unit, and τ_1 will finish its LO-crit execution at time 4. Thus, task τ_1 and τ_3 are schedulable on core c_1 in LO-crit mode. However, when a criticality mode change occurs (task τ_1 not finishing after executing its LO-crit WCET), task τ_1 needs to execute one more time unit but it is preempted by τ_3. After τ_3 executing one more time unit, τ_1 can only finish its HI-crit budget at time 6 which means it misses its deadline. Due to that, the taskset is not schedulable by a non-migration scheduling model. But with the support of migration, task τ_3 can be migrated to core c_2 when a criticality mode change occurs on core c_1. In this scenario, task τ_1 is able to execute one more time unit at time 4 and finish its HI-crit budget at time 5, while task τ_2 and τ_3 are schedulable on core c_2. In summary, this simple example shows that the semi-partitioned scheduling model dominates any non-migration scheduling approach.

Task	C(LO)	C(HI)	T	D	L
τ_1	2	3	5	5	HI
τ_2	2	3	5	5	HI
τ_2	1	-	2	2	LO

Table 3.1: Example Taskset

This example shows the basic idea of the semi-partitioned algorithm, the detailed mechanism of the model can be illustrated with the help of state variables. Assume that a taskset S contains several tasks in two criticality levels (HI-crit and LO-crit). If this taskset is to be scheduled on a two cores platform (c_1 and c_2) by the semi-partitioned algorithm, then on each core there shall exist three types of tasks: HI-crit tasks, statically allocated LO-crit tasks and migratable LO-crit tasks. Let HI_i represent the set of HI-crit tasks on core c_i, LO_i represent the set of statically allocated LO-crit tasks and MIG_i represent the set of migrating LO-crit tasks, then the following relationship can be obtained:

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

In steady mode, all these tasks are statically partitioned on each core and executing with their LO-crit budgets. Define state X to represent this phase, then the relationship between tasks and cores may be viewed as:

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

If a criticality change occurs on one core (c_i), then HI-crit tasks (HI_i) will execute with their HI-crit budgets. For LO-crit tasks, some of them (LO_i) still execute on the core with their LO-crit budgets while the others (MIG_i) need to migrate to other cores as there is not enough space for them on the core. Define state $Y(1)$ to represent the case that core c_1 enters its HI-crit mode, then tasks in MIG_1 will be migrated from core c_1 to core c_2 and the relationship between tasks and cores may be viewed as:

- $Y(1)_1 = LO_1 \cup HI_1$
- $Y(1)_2 = LO_2 \cup HI_2 \cup MIG_1 \cup MIG_2$
- $S = Y(1)_1 \cup Y(1)_2$

Define state $Y(2)$ to represent the case that core c_2 enters its HI-crit mode, then tasks in MIG_2 will be migrated from core c_2 to core c_1 and the relationship between tasks and cores may be viewed as:

- $Y(2)_1 = LO_1 \cup HI_1 \cup MIG_1 \cup MIG_2$
- $Y(2)_2 = LO_2 \cup HI_2$
- $S = Y(2)_1 \cup Y(2)_2$

In state $Y(1)$, taskset MIG_1 is migrated from core c_1 to core c_2, and in state $Y(2)$, taskset MIG_2 is migrated from core c_2 to core c_1. For tasks in $Y(1)_1$ and $Y(2)_2$, the migration progress does not affect their priorities. For tasks in $Y(1)_2$ and $Y(2)_1$, since extra tasks have been migrated to these two cores, it is likely that the original priority orders will be affected. So the priority orders in $Y(1)_2$ and $Y(2)_1$ may need to be recalculated offline. For these migrated tasks, it is not defined whether they have finished or partly-completed or even not yet started before migration occurs. Assume the migrations have no cost, all of the migrating tasks may need to execute all their LO-crit budgets on newly allocated cores in order to guarantee their completion. In addition, these tasks are likely to be released certain time before migrating, which means they have reduced deadlines (D^*) after migration. To compare the exact value of such deadline reduction is unlikely to be tractable as all of the release patterns need to be considered, so a sufficient analysis can be obtained by applying the smallest reduced deadline to each migrating task.

Theorem(1): For task τ_i, the worst case after migration is that it needs to execute all its LO-crit budget in a reduced deadline of $D_i^* = D_i - (R_i - C_i)$, where R_i is the worst-case response time for task t_i in state X.

Proof. Assume the latest release of task τ_i is t_0. The task migrates at time $t_0 + t$ and has completed a units of its current release job. Then in order to meet the deadline, task τ_i needs to finish the rest of job, which is $C_i - a$, within time $D_i - ((t_0 + t) - t_0) = D_i - t$ after migration if no migration costs are considered. In addition, for two tasks with same period, if task τ_1 needs to finish C units in D and task τ_2 needs to finish $C + x$ in $D + x$, then τ_2 is harder to be scheduled in FPS than τ_1. In other words, if τ_2 is deemed to be schedulable, then τ_1 is also schedulable. Thus, for τ_i, it has worst case when it has to schedule $C_i - a + a = C_i$ within time $D_i - t + a = D_i - (t - a)$. The value of $(t - a)$ may be understood as the time that task τ_i is pre-emptied before migration, and the maximum of this value is $R_i - C_i$ when task τ_i has been pre-emptied for a maximum time without executing any of its job. In all, the worst case for τ_i is that it has to execute all its LO-crit budget (C_i) in a reduced deadline of $D_i^* = D_i - (R_i - C_i)$. □

The above paragraphs have considered how the semi-partitioned algorithm may improve the scheduling efficiency when only one core has increased to the HI-crit mode. But in reality, both of the cores may be in HI-crit mode at the same time. There are two possible situations in which both of the cores are in HI-crit mode: both cores increase into HI-crit mode at the same time or two cores increase into HI-crit mode one after another. Regarding to the first case, as both cores enter their HI-crit mode, migrating tasks have no place to execute. Therefore, these tasks need to be abandoned while HI-crit tasks and statically allocated LO-crit tasks are still able to guarantee their completion. Define state

BX to represent this case based on state X, the relationship among tasksets can be obtained as:

- $BX_1 = LO_1 \cup HI_1$
- $BX_2 = LO_2 \cup HI_2$
- $S = BX_1 \cup BX_2 \cup MIG_1 \cup MIG_2$

Regarding to the latter case which is based on the situation that one core has already entered HI-crit mode, the schedulability tests of tasks on that core have been covered in the previous section. Referring to the core that enters HI-crit mode later, since extra LO-crit tasks have been executing on the core, only abandoning the migrating tasks may not guarantee the execution of HI-crit tasks. Considering that, all LO-crit tasks on that core need to be abandoned. Define state $BY(1)$ to represent this situation based on state $Y(1)$ and state $BY(2)$ to represent this situation based on state $Y(2)$, the relationship among tasksets can be obtained as:

- $BY(1)_1 = LO_1 \cup HI_1$
- $BY(1)_2 = HI_2$
- $S = BY(1)_1 \cup BY(1)_2 \cup MIG_1 \cup MIG_2 \cup LO_2$
- $BY(2)_1 = HI_1$
- $BY(2)_2 = LO_2 \cup HI_2$
- $S = BY(2)_1 \cup BY(2)_2 \cup MIG_1 \cup MIG_2 \cup LO_1$

3.1.1 Response Time Analysis

The analysis of the semi-partitioned model is quite similar to that of AMC. The schedulable test consists of a three-phase analysis. The first phase is to verify the schedulability of states X_1 and X_2 when all the tasks are partitioned on two cores and executing within their LO-crit budgets. The response time analysis for this phase is shown in equation (3.1) where $chp(i)$ stands for the set of all tasks with higher priority than that of task τ_i on the same core:

$$\forall \tau_i \in X : R_i(LO) = C_i(LO) + \sum_{j \in chp(i)} \left\lceil \frac{R_i(LO)}{T_j} \right\rceil C_j(LO) \tag{3.1}$$

The second phase is to verify the schedulability of states $Y(1)_1$ and $Y(1)_2$ and states $Y(2)_1$ and $Y(2)_2$ in the semi-partitioned model when some tasks have been migrated from one core to another. For states $Y(1)_1$ and $Y(2)_2$, the cores enter HI-crit mode where HI-crit tasks are executing with their HI-crit budgets while LO-crit tasks are executing with their LO-crit budgets (this is represented as $C_i(L_i)$ for task τ_i). The response time analysis for these tasks is shown in equation (3.2).

$$\begin{aligned} \forall \tau_i \in & (Y(1)_1 \cup Y(2)_2) : \\ & R_i(HI) = C_i(L_i) + \sum_{j \in chp(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(L_j) \end{aligned} \tag{3.2}$$

Meanwhile, tasks in $Y(1)_2$ and $Y(2)_1$ are executing in LO-crit mode that all tasks are executing with their LO-crit budgets. Additionally, since migrating tasks are released before moving to the cores, the release jitters of these tasks need to be considered when calculating their interference upon other tasks. In order to guarantee the schedulability, the maximum pre-emptive time of the migrating tasks are used as the release jitters when performing response time analysis. According to this, if task τ_i migrates to another core, the release jitter will be $J_i = R_i - C_i(LO)$; otherwise, $J_i = 0$. The response time analysis for these tasks is shown in equation (3.3).

$$\begin{aligned} \forall \tau_i \in & (Y(1)_2 \cup Y(2)_1) : \\ & R_i(LO)^* = C_i(LO) + \sum_{j \in chp(i)} \left\lceil \frac{R_i(LO)^* + J_j}{T_j} \right\rceil C_j(LO) \end{aligned} \tag{3.3}$$

The last phase is to check the schedulability of the criticality change progress which consists of two parts. The first part is to check the schedulability of cores entering HI-crit mode. This progress is quite similar to the mode change progress from LO-crit level to MID-crit level in three criticality levels (LO-crit, MID-crit and HI-crit) AMC [52]. But in the semi-partitioned model, migration tasks will be "aborted" on the core when migration happens while other LO-crit tasks remain executing in their LO-crit budgets, and HI-crit tasks start to execute with their HI-crit budgets. Comparing with the three criticality levels AMC, migrating tasks in semi-partitioned model perform similarly to LO-crit tasks in AMC, other LO-crit tasks in semi-partitioned model perform similarly to MID-crit tasks in AMC but execute with their LO-crit budgets rather than MID-crit budgets, and HI-crit tasks perform similarly to HI-crit tasks but execute with their HI-crit budgets rather than MID-crit budgets. Thus, by modifying equation (2.17), a sufficient response time analysis for the semi-partitioned model can be obtained as equation (3.4) where $chpH(i)$ stands for the taskset that contains all the HI-crit tasks which have higher priority than task τ_i on the same core, $chpL(i)$ stands for the taskset that contains all the non-migrating LO-crit tasks which have higher priority than tasks τ_i on the same core, $chpMIG(i)$ stands for the taskset that contains all the migrating LO-crit tasks which have higher priority than tasks τ_i on the same core.

$$\begin{aligned}
\forall \tau_i \in & (Y(1)_1 \cup Y(2)_2) : \\
& R_i(HI)^* = C_i(L_i) \\
& \qquad + \sum_{j \in chpH(i)} \left\lceil \frac{R_i(HI)^*}{T_j} \right\rceil C_j(HI) \\
& \qquad + \sum_{k \in chpL(i)} \left\lceil \frac{R_i(HI)^*}{T_k} \right\rceil C_k(LO) \\
& \qquad + \sum_{l \in chpMIG(i)} \left\lceil \frac{R_i(LO)}{T_l} \right\rceil C_l(LO)
\end{aligned} \tag{3.4}$$

The other part of the last phase is to check the schedulability of migrating tasks. As stated in the semi-partitioned model, these tasks have a reduced deadline for their current release during migrating. As equation (3.3) represents the response time analysis for these migrating tasks after migration, their results need to be compared with reduced deadlines to decide their schedulability. Equation (3.5) shows the calculation of the reduced deadlines for migrating tasks.

$$\forall \tau_i \in (MIG_1 \cup MIG_2) : D_i^* = D_i - (R_i(LO) - C_i) \tag{3.5}$$

In addition, according to equation (3.4), it seems that setting LO-crit tasks with high priority will bring more contributions to scheduling other tasks. Furthermore, according to equation (3.5), tasks with higher priority are likely to have relatively smaller pre-emptive time which leads them to have relatively larger reduced deadlines. It makes such tasks easier to be scheduled. However, such tasks may also have quite high priorities after migration which will bring in more impacts on statically allocated tasks, including HI-crit tasks. So there is a payoff when determining the choice of migrating tasks.

Referring to the case that both cores increase to their HI-crit mode, the schedulability test for the first situation that both cores enter HI-crit mode at the same time has already been covered in equation (3.2). For the latter case, it is similar to AMC algorithm that all LO-crit tasks need to be abandoned during the mode change. The response time analysis of HI-crit tasks can be represented as equation (3.6) :

$$\begin{aligned} R_i(HI)^{**} = & C_i(HI) + \sum_{j \in chpH(i)} \left\lceil \frac{R_i(HI)^{**}}{T_j} \right\rceil C_j(HI) \\ & + \sum_{k \in chpL(i)} \left\lceil \frac{R_i(LO)^{*}}{T_k} \right\rceil C_k(LO) \end{aligned} \tag{3.6}$$

This completes all the analyses required to test the schedulability of a dual-criticality system on a dual-core platform.

3.1.2 Analysis Example

This section will show an example to illustrate the response time analysis discussed above. Regarding the taskset in Table 3.2, τ_1, τ_2, τ_3 and τ_4 are assigned to c_1 and τ_4 may migrate to c_2 when c_1 enters high mode, while τ_5, τ_6, τ_7 and τ_8 are assigned to c_2 and τ_8 may migrate to c_1 when c_2 enters high mode. According to the priorities, for core c_1, $P_3 > P_2 > P_4 > P_1$; for core c_2, $P_7 > P_5 > P_8 > P_6$.

Task	C(LO)	C(HI)	T	D	L	P	c	MIG
τ_1	8	16	36	36	HI	7	1	NO
τ_2	3	4	12	12	HI	3	1	NO
τ_3	1	-	6	6	LO	1	1	NO
τ_4	1	-	12	12	LO	5	1	YES
τ_5	4	5	12	12	HI	4	2	NO
τ_6	10	20	56	56	HI	8	2	NO
τ_7	1	-	9	9	LO	2	2	NO
τ_8	1	-	12	12	LO	6	2	YES

Table 3.2: Example Taskset

For the semi-partitioned algorithm, the schedulability test will be done in five phases: state X, state $Y(1)$ with the migration progress $X \Rightarrow Y(1)$, state $BY(1)$, state $Y(2)$ with the migration progresses $X \Rightarrow Y(2)$, and state $BY(2)$. For state X, all of the tasks are executing on their LO-crit budgets.

- $X_1 = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ and $P_3 > P_2 > P_4 > P_1$
  - $R_3(LO) = 1 < 6 = D_3$
  - $R_2(LO) = 3 + \lceil \frac{4}{6} \rceil \times 1 = 4 < 12 = D_2$
  - $R_4(LO) = 1 + \lceil \frac{5}{6} \rceil \times 1 + \lceil \frac{5}{12} \rceil \times 3 = 5 < 12 = D_4$
  - $R_1(LO) = 8 + \lceil \frac{20}{6} \rceil \times 1 + \lceil \frac{20}{12} \rceil \times 3 + \lceil \frac{20}{12} \rceil \times 1 = 20 < 36 = D_1$
- $X_2 = \{\tau_5, \tau_6, \tau_7, \tau_8\}$ and $P_7 > P_5 > P_8 > P_6$
  - $R_7(LO) = 1 < 9 = D_7$

- $R_5(LO) = 4 + \lceil\frac{5}{9}\rceil \times 1 = 5 < 12 = D_5$
- $R_8(LO) = 1 + \lceil\frac{6}{9}\rceil \times 1 + \lceil\frac{6}{12}\rceil \times 4 = 6 < 12 = D_8$
- $R_6(LO) = 10 + \lceil\frac{23}{9}\rceil \times 1 + \lceil\frac{23}{12}\rceil \times 4 + \lceil\frac{23}{12}\rceil \times 1 = 23 < 56 = D_6$

For state $Y(1)$, task τ_4 has migrated from core c_1 to core c_2, and HI-crit tasks on c_1 will execute on their HI-crit budgets while all other tasks remain executing with their LO-crit budgets. In addition, task τ_4, as a migrating task, needs to use its reduced deadline for checking its schedulability.

- $Y(1)_1 = \{\tau_1, \tau_2, \tau_3\}$ and $P_3 > P_2 > P_1$
  - $R_3(HI)^* = 1 < 6 = D_3$
  - $R_2(HI)^* = 4 + \lceil\frac{5}{6}\rceil \times 1 = 5 < 12 = D_2$
  - $R_1(HI)^* = 16 + \lceil\frac{36}{12}\rceil \times 4 + \lceil\frac{36}{6}\rceil \times 1 + \lceil\frac{20}{24}\rceil \times 2 = 36 \leq 36 = D_1$ as the worst case happens when $t = R_1(LO) = 20$
- $Y(1)_2 = \tau_4, \tau_5, \tau_6, \tau_7, \tau_8$ and $P_7 > P_5 > P_4 > P_8 > P_6$
  - $D_4^* = 12 - (5 - 1) = 8$
  - $J_4 = 5 - 1 = 4$
  - $R_7(LO)^* = 1 < 9 = D_7$
  - $R_5(LO)^* = 4 + \lceil\frac{5}{9}\rceil \times 1 = 5 < 12 = D_5$
  - $R_4(LO)^* = 1 + \lceil\frac{6}{9}\rceil \times 1 + \lceil\frac{6}{12}\rceil \times 4 = 6 < 8 = D_4(r)$
  - $R_8(LO)^* = 1 + \lceil\frac{7}{9}\rceil \times 1 + \lceil\frac{7}{12}\rceil \times 4 + \lceil\frac{7+4}{12}\rceil \times 1 = 7 < 12 = D_8$
  - $R_6(LO)^* = 10 + \lceil\frac{32}{9}\rceil \times 1 + \lceil\frac{32}{12}\rceil \times 4 + \lceil\frac{32+4}{12}\rceil \times 1 + \lceil\frac{32}{12}\rceil \times 1 = 32 < 56 = D_6$

For state $BY(1)$, as core c_2 also increases to HI-crit mode, LO-crit tasks τ_4, τ_7 and τ_8 on this core need to be abandoned while HI-crit tasks τ_5 and τ_6 will execute in their HI-crit budgets. Core c_1 stays unchanged from state $Y(1)$, so no extra checks are required.

- $BY(1)_2 = \tau_5, \tau_6$ and $P_7 > P_5 > P_4 > P_8 > P_6$

- $R_5(HI)^{**} = 5 + \lceil \frac{5}{9} \rceil \times 1 = 6 < 12 = D_5$ as the worst case happens when $t = R_5(LO)^* = 5$
- $R_6(HI)^{**} = 20 + \lceil \frac{55}{12} \rceil \times 5 + \lceil \frac{32}{9} \rceil \times 1 + \lceil \frac{32}{12} \rceil \times 1 + \lceil \frac{32}{12} \rceil \times 1 = 55 < 56 = D_6$ as the worst case happens when $t = R_6(LO)^* = 32$

For state $Y(2)$, task τ_8 has migrated from core c_2 to core c_1, and HI-crit tasks on c_2 will execute on their HI-crit budgets while all other tasks remain executing with their LO-crit budgets. In addition, task τ_8, as a migrating task, needs to use its reduced deadline for checking its schedulability.

- $Y(2)_1 = \{\tau_1, \tau_2, \tau_3\, \tau_4, \tau_8\}$ and $P_3 > P_2 > P_4 > P_8 > P_1$
  - $D_8^* = 12 - (6 - 1) = 7$
  - $J_8 = 6 - 1 = 5$
  - $R_3(LO)^* = 1 < 6 = D_3$
  - $R_2(LO)^* = 3 + \lceil \frac{4}{6} \rceil \times 1 = 4 < 12 = D_2$
  - $R_4(LO)^* = 1 + \lceil \frac{5}{6} \rceil \times 1 + \lceil \frac{5}{12} \rceil \times 3 = 5 < 12 = d_4$
  - $R_8(LO)^* = 1 + \lceil \frac{6}{6} \rceil \times 1 + \lceil \frac{6}{12} \rceil \times 3 + \lceil \frac{6}{12} \rceil \times 1 = 6 < 7 = D_8(r)$
  - $R_1(LO)^* = 8 + \lceil \frac{23}{6} \rceil \times 1 + \lceil \frac{23}{12} \rceil \times 3 + \lceil \frac{23}{12} \rceil \times 1 + \lceil \frac{23+5}{12} \rceil \times 1 = 23 < 36 = D_1$
- $Y(2)_2 = \{\tau_5, \tau_6, \tau_7\}$ and $P_7 > P_5 > P_6$
  - $R_7(H)^* = 1 < 9 = D_7$
  - $R_5(H)^* = 5 + \lceil \frac{6}{9} \rceil \times 1 = 6 < 12 = D_5$
  - $R_6(H)^* = 20 + \lceil \frac{48}{9} \rceil \times 1 + \lceil \frac{48}{12} \rceil \times 5 + \lceil \frac{23}{12} \rceil \times 1 = 48 \leq 56 = D_6$ as the worst case happens when $t = R_6(LO) = 23$

For state $BY(2)$, as core c_1 also increases to HI-crit mode, LO-crit tasks τ_4, τ_7 and τ_8 on this core need to be abandoned while HI-crit tasks τ_5 and τ_6 will execute in their HI-crit budgets. Core c_2 stays unchanged from state $Y(2)$, so no extra checks are required.

- $BY(2)_1 = \tau_1, \tau_2$ and $P_3 > P_2 > P_4 > P_8 > P_1$
  - $R_2(HI)^{**} = 4 + \lceil \frac{4}{6} \rceil \times 1 = 5 < 12 = D_5$ as the worst case happens when $t = R_2(LO)^* = 4$
  - $R_1(HI)^{**} = 16 + \lceil \frac{36}{12} \rceil \times 4 + \lceil \frac{23}{6} \rceil \times 1 + \lceil \frac{23}{12} \rceil \times 1 + \lceil \frac{23}{12} \rceil \times 1 = 36 \leq 36 = D_1$ as the worst case happens when $t = R_1(LO)^* = 23$

As all of the sufficient response time analyses above are less than or equal to their deadlines, this taskset is schedulable by using the semi-partitioned algorithm.

If the non-migration algorithm is applied to schedule this taskset, neither core c_1 nor c_2 is schedulable since $R_1 = 16 + \lceil \frac{44}{12} \rceil \times 4 + \lceil \frac{44}{6} \rfloor \times 1 + \lceil \frac{44}{12} \rfloor \times 1 = 44 > 36 = D_1$ and $R_6 = 20 + \lceil \frac{57}{12} \rceil \times 5 + \lceil \frac{57}{9} \rceil \times 1 + \lceil \frac{57}{12} \rceil \times 1 = 57 > 56 = D_6$. As migration only occurs if necessary and the above example shows that multi-core platform delivers improved schedulability, it can be concluded that the semi-partitioned algorithm dominates any non-migration algorithm.

3.1.3 Returning to LO-crit Mode

This section will address the issue of returning to LO-crit mode. There are three possible cases based on the number of cores in HI-crit mode and which core is returning to LO-crit mode.

The first case occurs when only one core is in its HI-crit mode. In this case, once the core (core c_1 in state $Y(1)$ or core c_2 in state $Y(2)$) in HI-crit mode experiences an idle tick, it can return to LO-crit mode and the next release of migrated tasks will be on their original processor.

The second case may happen when both cores are in HI-crit mode and the core which enters HI-crit mode later is returning to LO-crit mode. In this case, once the core (core c_2 in state $BY(1)$ or core c_1 in state $BY(2)$) in HI-crit mode experiences an idle tick, it can return to LO-crit mode and all of the tasks previously abandoned, including the allocated LO-crit tasks and migrating tasks belong to both cores, will start to execute on this processor.

The last case happens when both of the cores are in HI-crit mode and the core which enters HI-crit mode earlier (including the case that both cores enter HI-crit mode at the same time) is returning to LO-crit mode. In this case, once the core (core c_1 in state $BY(1)$, core c_2 in state $BY(2)$, and either core c_1 or c_2 in state BX) in HI-crit mode experiences an idle tick, all of the migrating tasks belong to both cores will start executing on this processor.

3.1.4 Migration Overhead Consideration

In the above algorithms, it is assumed that the overheads caused by the migration progress is small and can be omitted. In this section, we will consider the influence from the overheads and explore how it may affect the semi-partitioned algorithm.

As the overheads occur when LO-crit tasks migrate, the main effect of overheads upon semi-partitioned algorithm lies on the response time analysis of migrating tasks. Thus, the migrating tasks require to spend extra units of time before their execution on the migrate destination cores. Considering that, the effect from overheads is quite similar to that from pre-emption, which causes release jitter and a reduced deadline. Let O_i represent time it takes for task τ_i to migrate, then equation (3.3) can be modified as below to consider the influence from these overheads:

$$\begin{aligned}
&\forall \tau_i \in (Y(1)_2 \cup Y(2)_1): \\
&\quad R_i(LO)^{**} = C_i(LO) + \sum_{j \in chps(i)} \left\lceil \frac{R_i(LO)^{**}}{T_j} \right\rceil C_j(LO) \\
&\quad + \sum_{k \in chpMIG(k)} \left\lceil \frac{R_i(LO)^{**} + J_k + O_k}{T_k} \right\rceil C_k(LO) \qquad (3.7) \\
&\forall \tau_i \in (MIG_1 \cup MIG_2): \\
&\quad D_i^{**} = D_i - J_i - O_i
\end{aligned}$$

3.2 Semi-partitioned Configuration

The previous section has described how to determine whether a given set of tasks with fixed priorities and allocated cores is schedulable by the semi-partitioned algorithm. This section will consider how to apply the semi-partitioned algorithm to allocate a set of tasks to a dual-core platform. Given that bin-backing algorithms will be used for task allocation, the first step is to sort the set of tasks into a specified order. Since the primary target of MCS is to guarantee the execution of HI-crit tasks, it will be efficient to check if all the HI-crit tasks are schedulable first, which means it is helpful to put all of the HI-crit tasks in front of LO-crit tasks. As stated in Chapter 2, criticality-aware utilization descending order provides better performance than others in First-Fit partitioned MCS. It can be assumed that criticality-aware utilization descending order may also provide good performance in semi-partitioned MCS as the majority of the tasks are still partitioned. However, different task orders perform differently under different bin-packing algorithms. So other possible task sorting orders, such as criticality-aware period descending order and criticality-aware deadline descending order, are required to be evaluated.

According to the migration mechanism stated in the semi-partitioned model, setting a task as migratable will add extra computation load to the system. So it will be better to minimise the chance of setting a task as migratable. Considering that, the next step is to check whether the taskset is schedulable with the non-migration algorithm. The semi-partitioned algorithm will only be applied when the non-migration algorithm cannot schedule the taskset. The non-migration algorithm simply assigns tasks using First-Fit bin packing algorithm and checks the response times when HI-crit tasks execute with HI-crit budgets and LO-crit tasks execute with LO-crit budgets. Note that only LO-crit tasks are migratable in the semi-partitioned algorithm, so if the non-migration algorithm is not able to schedule all of the HI-crit tasks then the taskset will not be schedulable by the semi-partitioned algorithm.

Regarding the semi-partitioned algorithm, there are several possible approaches based on the choice of bin packing algorithms and selection of migration tasks. As stated in Section 2.5.1, First-Fit, Best-Fit and Worst-Fit are the three mostly commonly used bin-packing algorithms in MCS, but it is uncertain which method performs best for the semi-partitioned algorithm. For the choice of migration tasks, there are two main approaches. The first approach simply set the current fetched task migratable. In the second approach, the set of LO-crit already allocated tasks are considered candidates of migration (highest priority first).

For priority assignment, Audesly's optimal priority scheme will be used to assign priorities for all the task. An important issue is that migration tasks will be assigned two priorities: one for its original core, the other for its destination core. These values will be determined during task assignment. A detailed semi-partitioned approach is shown as follows:

1. A task is fetched from the sorted taskset.
2. Assign the task to one of the cores according to the chosen bin packing algorithm (FF or BF or WF).
3. Use Audsley's algorithm to assign priorities for all tasks and check whether all of the tasks are schedulable.
4. If an un-schedulable task is found, try to assign the task to the other core and assign the priority order and do the checking again.
5. If both cores have been checked and neither of them can schedule the fetched task, setting the fetched task or allocated tasks migratable will be considered.
6. Assign the fetched task to one of the cores according to the chosen bin packing algorithm (FF or BF or WF) and set the task migratable based on one of the approaches.
7. Use Audsley's algorithm to assign priorities for all tasks on both cores considering migration effects and check whether all of the tasks on both cores are schedulable.

8. If an un-schedulable task is found, try to assign the task to the other core and do the checking again.

9. If both cores have been checked and still neither of them can schedule the fetched task, then the task is not schedulable by the semi-partitioned algorithm.

10. Fetch another task to start a new loop until the taskset is empty or a task is detected un-schedulable.

3.2.1 Software Configuration

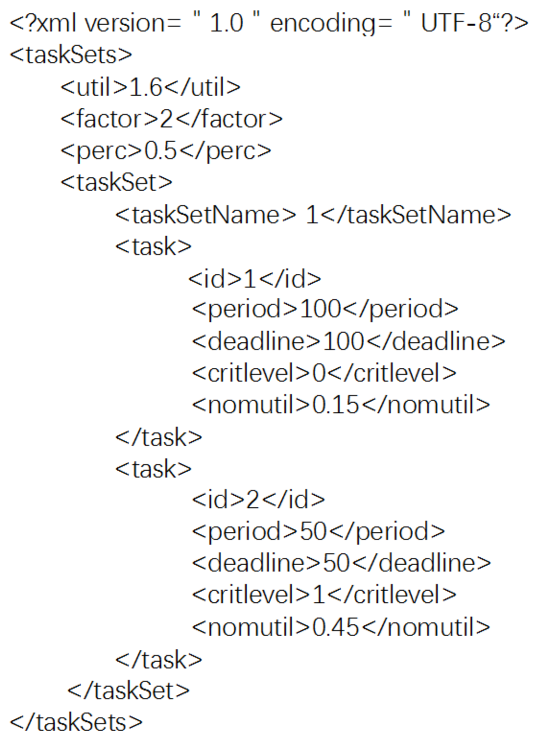
According to the model, the semi-partitioned algorithm should have better performance than the non-migration algorithm based on the original Vestal's algorithm. But it is uncertain how much the semi-partitioned algorithm has improved the scheduling efficiency. This section will introduce an experiment designed to compare the performance among six semi-partitioned approaches and also to compare the semi-partitioned approaches against the non-partitioned approach.

In order to reveal the performance of the semi-partitioned algorithm, software is produced to check the performance of different semi-partitioned approaches and the non-migration algorithm. The software consists of three parts. The first part of the software will generate tasksets. Tasks are randomly set to be HI-crit tasks or LO-crit tasks but the percentage of HI-crit tasks is controlled to be a fixed number. For all HI-crit tasks, their HI-crit WCETs have a fixed relationship with their LO-crit WCETs. These two fixed values will be changed in the experiments to explore the performance of different algorithms among different taskset settings. In order to gain uniform distributed parameters, the UUnifast-discard algorithm [30] is used to generate "nominal" utilization (a "nominal" utilization represents the LO-crit utilization for a LO-crit task or the HI-crit utilization for a HI-crit task), and the Log-uniform algorithm [51] is used to generate periods. Other parameters of each task can be calculated based on these two values ($D = T, C(L_i) = U_i(L_i) * T$).

The generated tasksets are stored as XML (eXtensible Markup Language) files. The reason to use XML is that XML is designed to be both human-readable and machine-readable, so that we can manually check the reliability of the scheduling results [26]. One XML file is designed to contain 10000 tasksets and each taskset in this file has the same utilization u. Regarding to that requirement, the XML file is configured to have a super parent node, named as taskSets. The super parent node taskSets contains a util attribute, which represents the same utilisation u, a factor value to store the fixed relationship between HI-crit and LO-crit WCET, a percentage value to store the ratio between HI-crit and LO-crit tasks, and 10000 parent nodes, named as taskSet. Each parent node taskSet contains a unique ID, named as taskSetName, and a number of nodes, named as task. The node task contains the essential attributes to form a task: id stands for a unique name in the taskset to differentiate each task; period literally stands for the interval time between each release of the task; deadline literally refer to the relative deadline of the task; critlevel is a integer value to indicate the criticality level of the tasks and 0 represents the lowest criticality level; nomutil stands for the nominal utilization of the task. An example of the XML file can be viewed in Figure 3.1.

The second part of the software is to pre-sort each taskset before scheduling. As stated in the task allocation section, all tasks will be sorted in criticality-aware utilization descending order. In such order, HI-crit tasks will be placed in front of all LO-crit tasks, and both HI-crit tasks and LO-crit tasks are in utilization descending order independently. The last part of the software is to test the success rate of scheduling the tasksets by different scheduling algorithms. The response time analysis in Section 3.1.1 is implemented in the software tool. Once the success rate is obtained, the results are stored in an XLS (Microsoft Excel) file for diagramming and further analysis.

In addition, in order to verify the reliability of the software, there exists a special method in the software to output the scheduling result of a specific approach upon a specific taskset. Combining with the corresponding XML file, a manually check is possible to be done to examine the reliability.



```
<?xml version= " 1.0 " encoding= " UTF-8“?>
<taskSets>
    <util>1.6</util>
    <factor>2</factor>
    <perc>0.5</perc>
    <taskSet>
        <taskSetName> 1</taskSetName>
        <task>
            <id>1</id>
            <period>100</period>
            <deadline>100</deadline>
            <critlevel>0</critlevel>
            <nomutil>0.15</nomutil>
        </task>
        <task>
            <id>2</id>
            <period>50</period>
            <deadline>50</deadline>
            <critlevel>1</critlevel>
            <nomutil>0.45</nomutil>
        </task>
    </taskSet>
</taskSets>
```

Figure 3.1: A Simple Example of XML file

3.3 Evaluation

We investigate the performance of six semi-partitioned approaches (Table 3.3) and compare them with the non-migration algorithm. The non-migration algorithm is chosen as the lowest bound of performance. Figure 3.2 (it would be better to look at the colour version) shows the percentage of tasksets that are schedulable for a system of 12 tasks, with half of those tasks having HI-crit (tp=0.5) and the HI-crit execution budget is double of the LO-crit execution budget (f=2). The Y-axis shows the percentage of the successful tasksets while the X-axis shows the sum of nominal utilizations of the tested taskset. The sum of utilization ranges only from 1.6 to 2.2 to amplify the results shown in the figure. Note that the utilization is larger than 2 due to the use of nominal utilization.

We observe that all of the semi-partitioned schedulability tests outperform the non-migration algorithm by a considerable margin. For example, as shown by the black lines, Semi2WF can schedule 61% of tasksets with utilization of 1.9 while non-migration algorithm can only schedule 42%. There exists an improvement

Notation	Description
Non-migration	The non-migration approach
Semi1FF	Semi-partitioned approach that migrates the fetched task and uses First Fit bin packing algorithm
Semi1BF	Semi-partitioned approach that migrates the fetched task and uses Best Fit bin packing algorithm
Semi1WF	Semi-partitioned approach that migrates the fetched task and uses Worst Fit bin packing algorithm
Semi2FF	Semi-partitioned approach that migrates the “highest” priority tasks and uses First Fit bin packing algorithm
Semi2BF	Semi-partitioned approach that migrates the “highest” priority tasks and uses Best Fit bin packing algorithm
Semi2WF	Semi-partitioned approach that migrates the “highest” priority tasks and uses Worst Fit bin packing algorithm

Table 3.3: Real-time System Notation

around $\frac{61-42}{42} * 100\% = 45.23\%$ from Semi2WF over the non-migration algorithm. This is expected as the semi-partitioned algorithms allow more LO-crit tasks to be scheduled. Comparing all of the semi-partitioned algorithms, the algorithms that migrate allocated tasks (Semi2) perform slightly better than those algorithms which only migrate fetched tasks (Semi1). This is also to be expected as the former type of algorithm checks more possibilities and is more likely to migrate LO-crit tasks with higher priorities, which, as discussed earlier, may improve scheduling. Among Semi2 algorithms, Semi2WF has the best performance when the sum of utilization is smaller than 1.9 while Semi2FF outperforms others in the other cases.

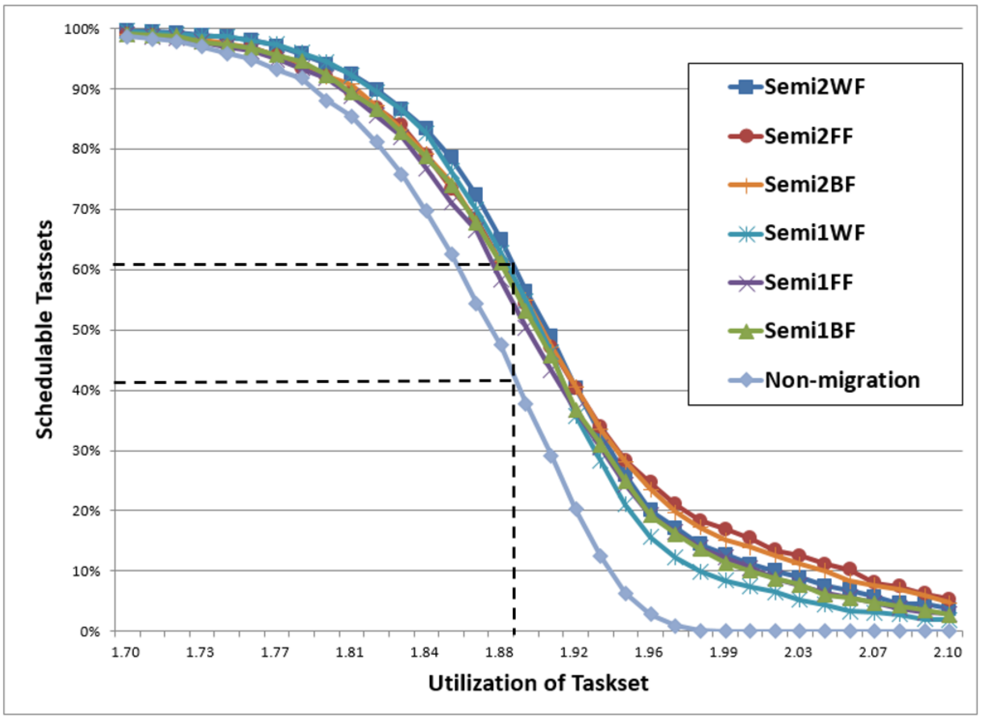


Figure 3.2: Percentage of Schedulable Tasksets (better in color version)

In order to explore the performance of the algorithms relating to the criticality factor ($C(HI)/C(LO)$) and the percentage of HI-crit tasks. Weighted schedulability measure $W_y(p)$ [19] is used for schedulability test y as a function of parameter p to reduce a 3-dimensional plot to 2 dimensions:

$$W_y(p) = (\sum_{\forall\Gamma} u(\Gamma) * S_y(\Gamma, p)) / \sum_{\forall\Gamma} u(\Gamma) \tag{3.8}$$

Regarding equation (3.8), for each value of p, it combines results for all of the tasksets Γ generated for all of a set of equally spaced utilization levels (same as that in previous figure, 1.6 to 2.2 in steps of 0.012). $S_y(\Gamma, p)$ is the binary result (1 or 0) of schedulability test y for a taskset Γ with parameter value p while $u(\Gamma)$ represents the utilization of taskset Γ.

We show how the results are changed by varying one key parameter at a time. Figure 3.3 varies the criticality factor, Figure 3.4 varies the percentage of HI-crit tasks and Figure 3.5 varies the size of a taskset. The X-axis stands for the parameter examined and Y-axis represents the weighted value.

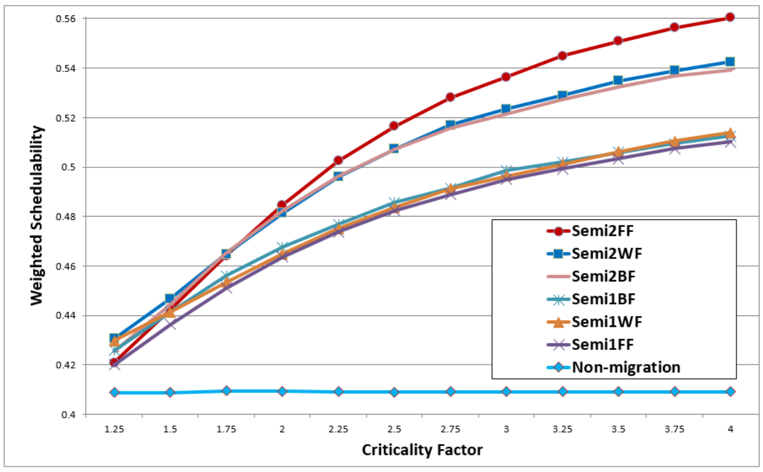


Figure 3.3: Varying the Criticality Factor

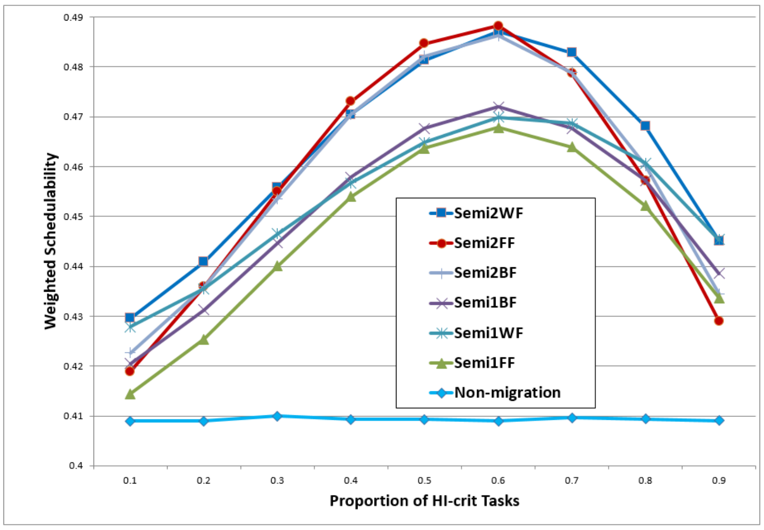


Figure 3.4: Varying the Criticality Proportion

According to Figure 3.3, Semi2WF has the best performance when criticality factor is smaller than 2 while Semi2FF outperforms others in the rest of the cases. In addition, all semi-partitioned algorithms have increased performance as the criticality factor increases. This is to be expected as the increase of WCET differences between different criticality levels allows more scheduling potential for migration tasks. According to Figure 3.4, the performance of the semi-partitioned algorithms has formed an inverted U-shape curve since each end of the interval represents a one-criticality taskset, and hence the priorities are optimal. Regarding to individual performance, Semi2WF has the best performance in most of the cases ($0 < p < 0.8$) while Semi1WF has the best performance when

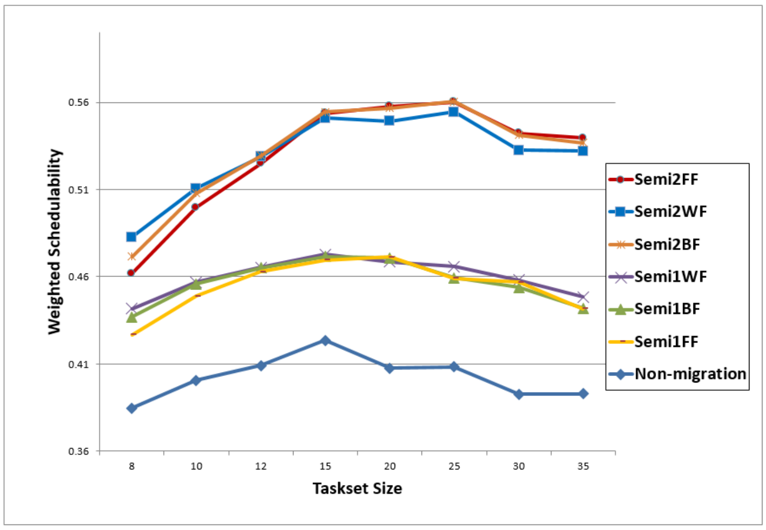


Figure 3.5: Varying the Taskset Size

the percentage of criticality tasks is quite high ($0.8 < u < 1$). According to Figure 3.5, the performance of the semi-partitioned algorithms have formed an inverted U-shape curve. This is expected as tasks are relatively large in small sized tasksets, which adds difficulty in finding acceptable migrating tasks. While in large sized tasksets, the interference from high priority tasks gets increased which adds difficulty to the schedulability of migrating tasks due to the reduced deadline and release jitter issues. Regarding to individual performance, Semi2WF has the best performance when the size of the taskset is small ($n \leqslant 12$) while Semi2FF has the best performance in the rest of the cases ($n > 12$).

Overall, Semi2WF and Semi2FF have the best performance in the majority of the cases. Thus, a combined usage of Semi2WF and Semi2FF may be the most appropriate method of scheduling a two criticality level MCS on a dual-core platform.

3.3.1 Overhead Influence

In this section, we explore the influence of the size of the overhead on the schedulability of the semi-partitioned algorithm. A key problem in considering overhead is that the overhead value is difficult to estimate. Brandenburg and Anderson [24] propose that the migration overhead can only be observed indirectly. In addition, Brandenburg et al. [25] indicates that the migration overhead is heavily dependent on the working set size of each task. Thus, it is valuable to find reasonable values to assume for the overhead. Bastoni et al. [19] conclude from their experiments that the migration delays do not depend significantly on the task set size but strongly relating to the pre-emption length. Based on their findings, we assume the migration overhead be a proportion of its response time in the steady mode. In addition, since we are evaluating the Semi2 algorithms that tries to migrate tasks with highest priorities, the response time of these tasks is close to their WCET. As we are using estimation values to find a rough scheduling efficiency effect from the overhead, we simply use the proportion of the WCET to represent the overhead values. For instance, for migratable task τ_i, overhead O_i equals to a proportion μ of its WCET C_i, that is $O_i = C_i * \mu$. This value μ is used to indicate the size of the overhead in the system and the same value is used for all of the migratable tasks.

In the experiment, we have checked the schedulability of the semi-partitioned algorithm with three different overhead proportion μ:0.05, 0.10 and 0.15. The non-migration algorithm is chosen as the lowest bound of performance while the semi-partitioned algorithm without the consideration of overhead is chosen as the upper bound. Figure 3.6 shows the scheduling efficiency of the semi-partitioned approach (a combination usage of Semi2WF and Semi2FF) with different overhead proportions in the scenario that $t = 12, p = 0.5, f = 2$. The X-axis represents the utilization of the taskset and the Y-axis represents the schedulability rate. It can be observed from the results that when the proportion is smaller than or equal to 0.10, the semi-partitioned approach still outperforms the non-migration approach. Thus, it can be indicated that in this scenario, the semi-partitioned approach is a better choice when the overhead proportion is smaller than or equal to 0.10 of the task's WCET.

Based on the result observed in figure 3.6, 0.10 behaves as a boundary number for the overhead influence of the specific scenario. We define the boundary number to be the largest two digit value for which the semi-partitioned approach still outperforms (at least a 5% improvement in schedulability) the non-migration

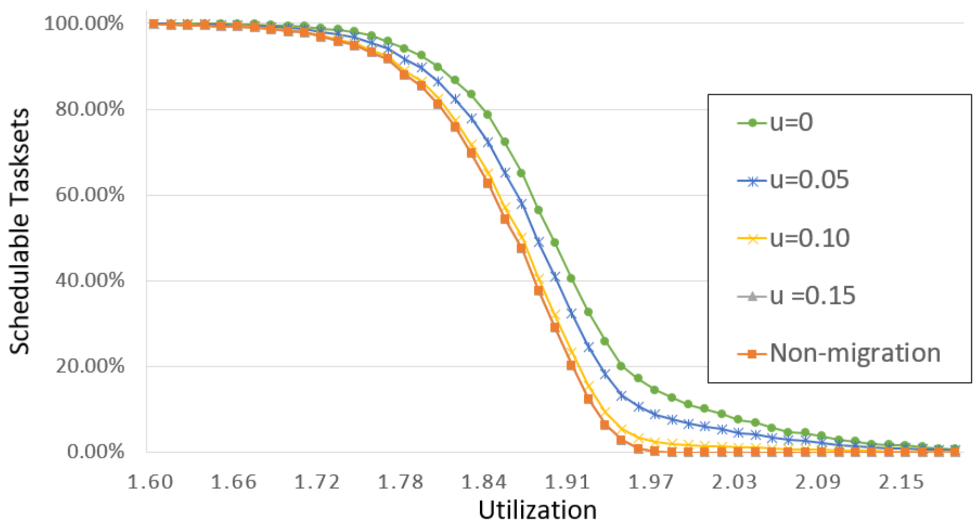


Figure 3.6: Overhead Exploration

approach with such an overhead. We propose that by exploring the boundary number of different scenarios, we may be able to make a summary of the overhead influence on the semi-partitioned approach. Figure 3.7 shows the relationship between the boundary number and the criticality factor f, Figure 3.8 shows the relationship between the boundary number and the number of tasks in a taskset, and Figure 3.9 shows the relationship between the boundary number and the percentage of HI-crit tasks in the taskset.

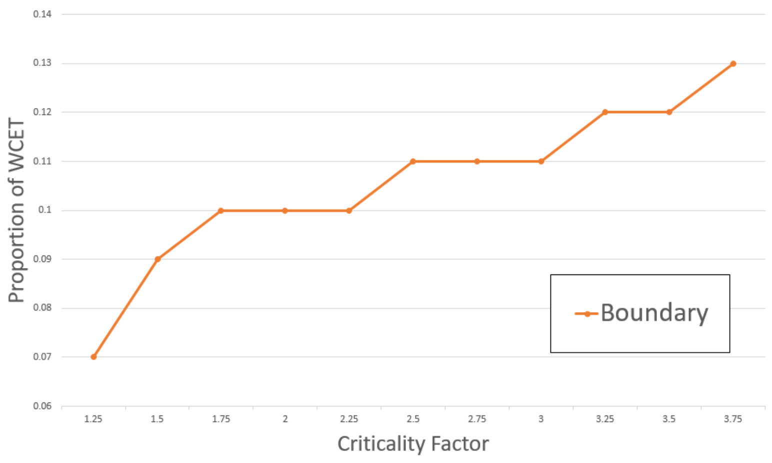


Figure 3.7: Varying the Criticality Factor

It can be observed that increasing the factor will ease the influence from the overhead, while both percentage and number of tasks form an inverted U-shape curve. According to that, it is hard to generate a solid summary. Thus,

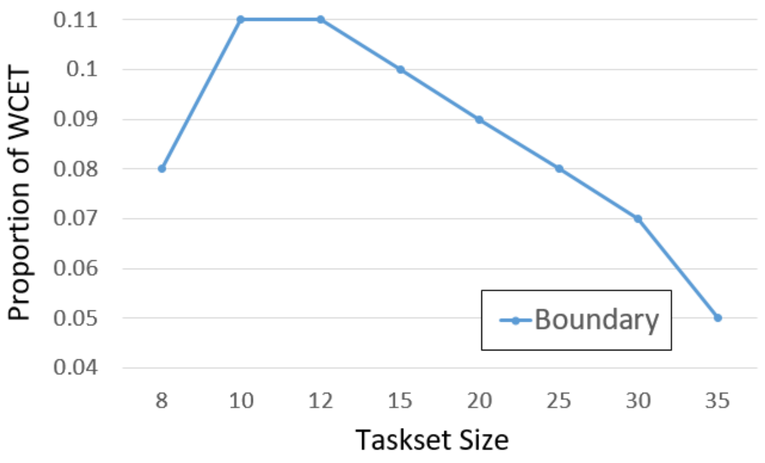


Figure 3.8: Varying the Number of Tasks

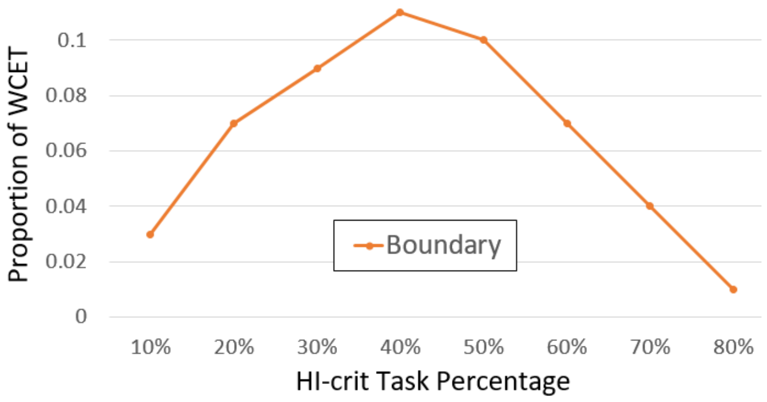


Figure 3.9: Varying the Percentage of HI-crit Tasks

it is suggested that the tolerance of overhead shall be analysed case by case. Nevertheless, in almost all scenarios if the cost of migration is less than 5% of a task WCET (which is a valid assumption), the semi-partitioned approach is beneficial.

3.4 Summary

In this chapter, we have introduced a semi-partitioned model for a dual-core MCS with two criticality levels. The model allows some LO-crit tasks on the mode changing core to migrate to the other core if only one core enters HI-crit mode. If both cores are in HI-crit mode, all HI-crit tasks are guaranteed to be schedulable while some LO-crit tasks shall be abandoned. Then we have provided a detailed response time analysis of the semi-partitioned model and illustrated the equations with an example. During the analysis, we observe that there exists a release jitter issue and a reduced deadline issue caused by the migration progress. For the migrating tasks, it is not defined whether they have finished or partly-completed or even not yet started before the migration occurs, and all of the migrating tasks still need to execute their remaining LO-crit budgets on newly allocated cores. Considering that, the migration tasks have reduced deadlines (D^*) after migration. It is difficult to compare the exact value of such deadline reduction since all of the release patterns need to be considered. Therefore, a sufficient analysis is proposed, which can be obtained by applying the smallest reduced deadline to each migrating task. This chapter has proved that for task τ_i, the worst case after migration is that it needs to execute all its LO-crit budget in a reduced deadline of $D_i^* = D_i - (R_i - C_i)$, where R_i is the worst-case response time for task t_i when the core is in LO-crit mode and the release jitter is $J_i = R_i - C_i$. Based on that, migrating tasks with relatively high priority are likely to have small release jitter and small effect from the reduced deadline issue.

After that, we introduce six scheduling approaches based on two possible task migration schemes and three bin-packing algorithms during the discussion of the configuration of the semi-partitioned model. Then we set up an experiment to compare the performances of the approaches proposed. According to the results, we observe that all of the proposed approaches have better performance than the non-migration algorithm. It is also observed that the semi-partitioned approach which migrates the "highest" priority migratable LO-crit tasks with Worst-Fit bin-packing algorithm and the semi-partitioned approach which migrates the "highest" priority migratable LO-crit tasks with First-Fit bin-packing algorithm have better performance than others in all of the scenarios. Based on the findings, we suggest that a combining usage of Semi2WF and Semi2FF shall be the most appropriate method for scheduling a dual-criticality level MCS on a dual-core platform. We also explore the effect of the overhead caused by migration. It is observed that the boundary case changes during different scenarios and therefore it is suggested to analyse the tolerance of overhead case by case.

In the next chapter, we will explore the semi-partitioned model on a multi-core platform.

Chapter 4

Semi-partitioned Model for a Multi-core Platform with Two Criticality Levels

The previous chapter has explored the semi-partitioned model for a dual-core platform, and proposed an appropriate approach to schedule a set of tasks in such a system. Although it is studying the simplest case of MCS, the findings are extendable to other cases when the taskset and the migration destination are fixed. This chapter will be based on the previous findings to extend the semi-partitioned model to a multi-core platform but still with two criticality levels.

This chapter will first identify the issues rising from extending the semi-partitioned model from a dual-core platform to a multi-core platform; and will then redefine the system model based on this extension. Then this chapter will explore scheduling upon a four-core platform, and introduce several possible migration models. An experiment will be set up to compare the efficiency of the proposed migration models, and an evaluation will be given based on the results. After that this chapter will extend the findings from a four-core platform to a multi-core platform. A summary will be given at the end of this chapter.

4.1 Redefine the Model

In the semi-partitioned model for dual-core platform, LO-crit migratable tasks shall migrate if only one core enters HI-crit mode while these tasks need to be abandoned if both of the cores enter HI-crit mode. According to that, there exists a boundary number n_b for the semi-partitioned approach: if less than or equal to n_b cores enter the HI-crit mode, LO-crit tasks may migrate and all tasks keep executing within their corresponding criticality level budgets; otherwise, if more than n_b cores enter the HI-crit mode, only HI-crit tasks guarantee their executions. Literally, $n-1$ can be the boundary number for a n-core platform. However, criticality change is a rare event and $n-1$ cores all in HI-crit mode is an extremely rare event. Thus, the determination of this boundary number n_b is an essential issue in the multi-core semi-partitioned approach. If n_b is quite small, then LO-crit tasks may need to be abandoned in many cases which is against the initial purpose of the design. If n_b is quite large, then the schedulability of the model will be quite low as the scheduling requirement becomes quite strict.

We propose that the problem can be addressed by using a probability calculation. Assume that the probability of one core entering HI-crit mode in a sufficient long period of time is fixed, and the criticality mode change on each core is independent. Based on probability knowledge, the probability of exactly m cores, being in HI-crit mode at the same time, can be calculated. Assume the probability of a core in HI-crit mode is p, then the probability of m cores, in a n cores system, are all in HI-crit mode in the same period of time is given by, where C_n^m represents the binomial coefficient function of choosing m out of n :

$$f(m,n) = C_n^m p^m (1-p)^{n-m} = \left\{\frac{n!}{m!(n-m)!}\right\} p^m (1-p)^{n-m} \tag{4.1}$$

Based on that, the probability of more than X cores entering HI-crit mode at the same time can be expressed as equation (4.2).

$$F(X, n) = \sum_{i=X+1}^{n} f(i, n) \tag{4.2}$$

According to equation (4.2), $F(X, n)$ will represent the probability of the case that the system needs to abandon LO-crit tasks. So if there exists a tolerance standard, p_{tol}, then the smallest number X which meets the tolerance standard ($F(X, n) \leq p_{tol}$) can be calculated. Assuming $n_b = 2$ is a reasonable requirement in a four-core system, p_{tol} can be set as $F(2, 4)$. Assume that $p = 10^{-4}$, an exploration is made to find out appropriate boundary number n_b for certain cases (Table 4.1).

n	n_b	$F(n_b, n)$	p_{tol}
4	2	4.00E-12	4.00E-12
8	3	7.00E-15	4.00E-12
16	3	1.82E-13	4.00E-12
32	3	3.59E-12	4.00E-12
64	4	7.59E-14	4.00E-12
128	4	2.62E-13	4.00E-12
256	5	3.61E-13	4.00E-12
512	6	1.68E-13	4.00E-12
1024	7	2.67E-13	4.00E-12

Table 4.1: Probability Table

According to the table, there exists a trend that the increment of the boundary number n_b is much slower than the increment of the number of cores. It indicates that the fault model introduced is likely to be extendable to many core platforms. However, it is also observed from the table that the exact boundary numbers n_b are irregular and hard to be calculated generally. We have tried to fit the curve and find that there is a small difference between $log_2 n$ and n_b (Table 4.2). Regarding to that, we propose to use $log_2 n$ instead of the exact values.

For system with other number of cores, it can be proved that $\lceil log_2(n) \rceil$ is an appropriate boundary number to fulfill the tolerance percentage requirement.

n	n_b	log_2n
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

Table 4.2: log_2n VS n_b

Lemma 4.1.1. *For systems with* $\{w|2^n + 1 \leq w \leq 2^{n+1}\}$ *cores,* $n + 1$ *is an appropriate boundary number.*

Proof. It can be prove by induction.

- The possibility of more than K enter HI-crit mode in an n-core platform and an (n+1)-core platform can be represented as $F(K, n) = \sum_{i=K+1}^{n} f(i, n)$ and $F(K, n + 1) = \sum_{i=K+1}^{n+1} f(i, n + 1)$.

- The difference between the two functions can be viewed as :$F(K, n + 1) - F(K, n) = (f(K+1, n+1) - f(K+1, n) + (f(K+2, n+1) - f(K+2), n) + ... + (f(n, n + 1) - f(n, n)) + f(n + 1, n + 1))$.

- For each pair $f(S, n + 1)$ and $f(S, n)$, they can be compared by using division. $f(S, n + 1)/f(S, n) =$ $\left\{\frac{(n+1)!}{S!(n+1-S)!}\right\} p^S(1 - p)^{n+1-S} * \left\{\frac{S!(n-S)!}{n!}\right\} \frac{1}{p^S} \frac{1}{(1-p)^{n-S}} = \frac{n+1}{n+1-S} * (1 - p)$.

- Since $p = 0.0001$, $(1 - p) \approx 1$ and $n + 1 > n + 1 - s$, $f(S, n + 1)/f(S, n)$ shall be larger than 1.

- In that case, $f(S, n + 1)$ is larger than $f(S, n)$.

- According to that, $F(K, n+1)$ is larger than $F(K, n)$, which indicates that $F(n+1, 2^{n+1})$ has the largest value in all these situations.
- Since it is shown that p_{tol} is larger than $F(n+1, 2^{n+1})$, all these situations shall fulfill the requirement.
- Thus, $n+1$ would be an appropriate boundary number for systems with $\{w | 2^n + 1 \leq w \leq 2^{n+1}\}$ cores.
- Since $\lceil log_2(2^n+1) \rceil = n+1$ and $\lceil log_2(2^{n+1}) \rceil = n+1$, n+1 can be replaced by $\lceil log_2(NumberOfTheCores) \rceil$.
- In other words, $\lceil log_2(n) \rceil$ is an appropriate boundary number for systems with n cores.

□

Summing up all of the findings above, we propose a semi-partitioned model for a n-core system as following:

- If all tasks execute within their LO-crit budgets, then all deadlines are met and no tasks migrate.
- No LO-crit task is allowed to exceed its LO-crit budget.
- If HI-crit tasks on no more than $\lceil log_2(n) \rceil$ cores exceed their LO-crit budgets, then some LO-crit tasks will migrate, but ALL LO-crit tasks and HI-crit tasks remain schedulable. These migrating tasks will be divided and migrate to paired cores that are currently in LO-crit mode.
- If HI-crit tasks on more than $\lceil log_2(n) \rceil$ cores exceed their LO-crit budgets, then some LO-crit tasks will be abandoned, but all HI-crit tasks remain schedulable (without migration).

However, this value $\lceil log_2(n) \rceil$ is a calculated reasonable value so that the semi-partitioned scheduling model can be evaluated. In practical, other issues may affect the actual determination of the boundary number. In addition, we assume a fully connected platform where a task can migrate from any core to any other core at a fixed overhead cost in this chapter. This assumption will be removed in the next chapter.

4.2 Semi-partitioned Model on Four-core Platform

As the boundary number issue has been addressed and the model has been redefined for the multi-core platform, the next step is to solve the migration destination problem. In a dual-core platform, migratable tasks have only one core to migrate towards. But in a multi-core platform, tasks may literally migrate to any possible cores, which may cause the whole system to become unpredictable and hard to analyse. This section will explore this migration destination issue on a four-core platform. It will first introduce four allocation models and provide a brief exploration upon the working mechanisms of these models. A response time analysis upon these models will be given afterwards, as well as a comparison of the models based on the analysis. An evaluation of the models will be given at the end of this section.

4.2.1 Migration Models

For a four-core platform, if only one core enters HI-crit mode, the migration tasks have three possible cores to migrate towards. We propose three models based on the distribution of these migration tasks.

- Model 1 represents the model that all migration tasks migrate to one core.
- Model 2 represents the model that all migration tasks migrate to two cores.
- Model 2a represents a varient on Model 2 (see description below).

- Model 3 represents the model that all migrate tasks migrate to three cores.

The following subsection will introduce the details of these models, including the relationship among cores and how the migration tasks are divided to migrate to different cores. Note for this 4-core system, mode changes on less than or equal to two cores must be tolerated without the loss of scheduling.

Model 1

Model 1 is a naive model that migration tasks on each core may only migrate to one core. The model can be viewed as Figure 4.1, where the rectangles stand for cores and arrows stand for migration routes.

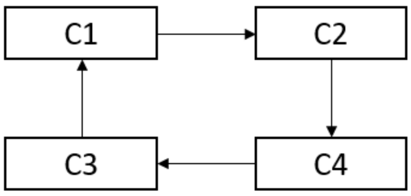


Figure 4.1: Model 1

According to the figure above, the migration routes form a circle which indicates that it is always possible to find an available core (a core that still in LO-crit mode) following the routes. Figure 4.2 and Figure 4.3 indicate two example scenarios of Model 1. The core in grey indicates that this core is currently in HI-crit mode; the thin arrow indicates a load of tasks migrate from one core to another; the thick arrow indicates different steps of the scenarios (the left hand side of the arrow is step 1 while the right hand side is step 2).

Based on these scenarios, the migrating load seems to be the main issue of Model 1. In Step 2 of Scenario 1 and Scenario 2, an extremely heavy task load is migrated to Core c_4 while no task migrates to Core c_3, which will undoubtedly affect the schedulability of the model. Based on this observation, the issue of this model lays on the heavy migration load during the second migration progress.

Model: Scenario 1 (Figure 4.2)

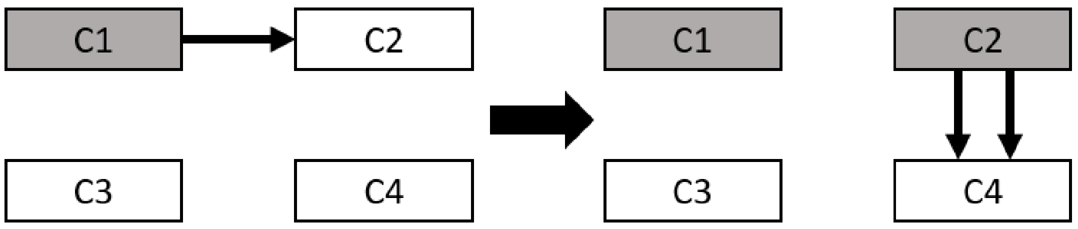


Figure 4.2: Model 1 Scenario 1

1. Core c_1 enters HI-crit mode, all of the migratable tasks on Core c_1 will migrate to Core c_2.

2. Core c_2 enters HI-crit mode, all of the migratable tasks on Core c_2, including tasks migrated from Core c_1, will migrate to Core c_4.

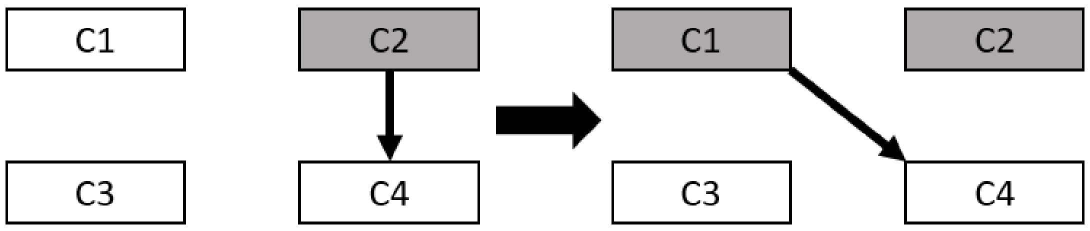


Figure 4.3: Model 1 Scenario 2

Model: Scenario 2 (Figure 4.3)

1. Core c_2 enters HI-crit mode, all of the migratable tasks on Core c_2 will migrate to Core c_4.

2. Core c_1 enters HI-crit mode, all of the migratable tasks on Core c_1 will migrate to Core c_2. But since Core c_2 is already in HI-crit mode, these migrating tasks will migrate to Core 4 directly.

Model 2

This model allows migration tasks to migrate to two cores rather than one. This increment of the migrating destinations leads to two issues: how to decide which two cores to migrate to and how the migration tasks shall be divided. Regarding to the first issue, Model 2 pairs cores into four groups: (Core c_1, Core c_2), (Core c_1, Core c_3), (Core c_2, Core c_4) and (Core c_3, Core c_4). Each core has two group-mate cores and migration tasks originally on the core will only migrate to the group-mate cores if they are available. For example, Core c_1 is paired with Core c_2 and Core c_3. If Core c_1 enters HI-crit mode, all of the migratable tasks on Core c_1 will migrate to Core c_2 and Core c_3. The model can be viewed as Figure 4.4.

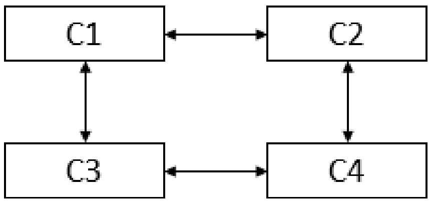


Figure 4.4: Model 2

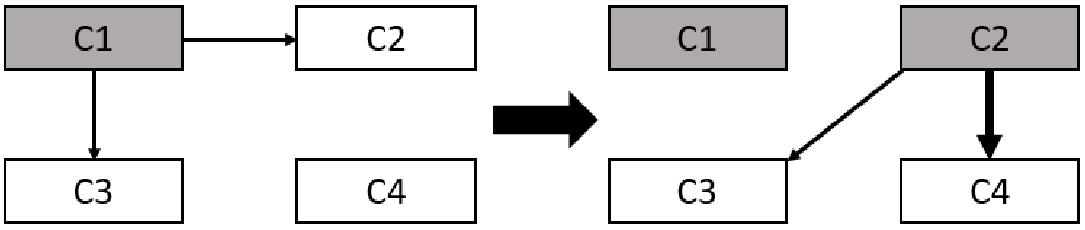


Figure 4.5: Model 2 Scenario 1

Model: Scenario 1 (Figure 4.5)

1. Core c_1 enters HI-crit mode, migratable tasks on Core c_1 will split into two parts and migrate to Core c_2 due to the pairing relationship (Core c_1, Core c_2) and Core c_3 due to the pairing relationship (Core c_1, Core c_3).

2. Core c_2 enters HI-crit mode, all of the migratable tasks originally on Core c_2 will migrate to Core c_4 due to the pairing relationship (Core c_2, Core c_4), and all of the migratable tasks from Core c_1 will migrate back to Core c_1. But since Core c_1 is already in HI-crit mode, these tasks will migrate to Core c_3. (In practice, these tasks will directly migrate to core c_3.)

Model: Scenario 2 (Figure 4.6)

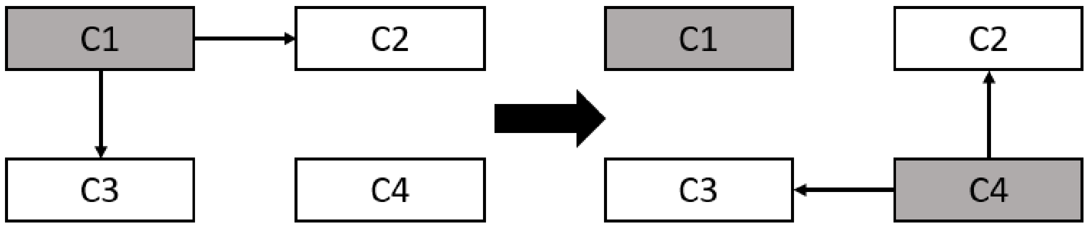


Figure 4.6: Model 2 Scenario 2

1. Core c_1 enters HI-crit mode, all of the migratable tasks on Core c_1 will split into two parts and migrate to Core c_2 due to the pairing relationship (Core c_1, Core c_2) and Core c_3 due to the pairing relationship (Core c_1, Core c_3).

2. Core c_4 enters HI-crit mode, all of the migratable tasks on Core c_4 will split into two parts and migrate to Core c_2 due to the pairing relationship (Core c_2, Core c_4) and Core c_3 due to the pairing relationship (Core c_3, Core c_4).

The split here (and following ones used in other models) is actually using WF bin-packing algorithm to assign the migratable tasks to other available cores. The reason to use WF algorithm is that WF provides a more balancing task distribution than other bin-packing algorithms.

Model 2a

This model is a possible variant on Model 2, but later on we will show that this model is dominated by Model 2. In Model 2a, cores are also paired into several groups and only tasks between groups may migrate to each other. The difference between this model and Model 2 is that this model migrates the whole migratable tasks to one paired core rather than splits the task load and migrates to two paired cores. For example, assume that Core c_1 is paired with Core c_2 and Core c_3. If Core c_1 enters HI-crit mode, all of the migratable tasks on Core c_1 will migrate to Core c_2 due to the pairing relationship (Core c_1, Core c_2). Then if Core c_2 also enters HI-crit mode, all of the migratable tasks from Core c_1 shall migrate to Core c_3 due to the pairing relationship (Core c_1, Core c_3), while the migratable tasks originally on Core c_2 may migrate to other cores regarding to the pairing relationship for Core c_2. The model can be viewed as Figure 4.4.

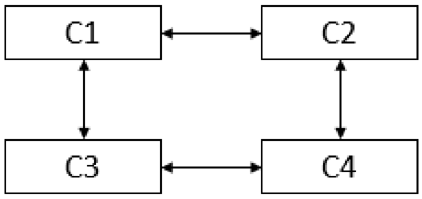


Figure 4.7: Model 2a

From the figure, the cores are paired into four groups: (Core c_1, Core c_2), (Core c_1, Core c_3), (Core c_2, Core c_4) and (Core c_3, Core c_4). Based on that, there are two possible scenarios:

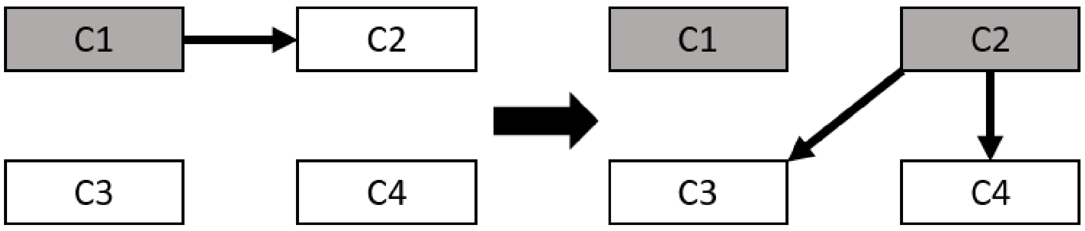


Figure 4.8: Model 2a Scenario 1

Model: Scenario 1 (Figure 4.8)

1. Core c_1 enters HI-crit mode, all of the migratable tasks on Core c_1 will migrate to Core c_2 due to the pairing relationship (Core c_1, Core c_2).
2. Core c_2 enters HI-crit mode, all of the migratable tasks originally on Core c_2 will migrate to Core c_4, while all of the migratable tasks from Core c_1 will migrate to Core c_3.

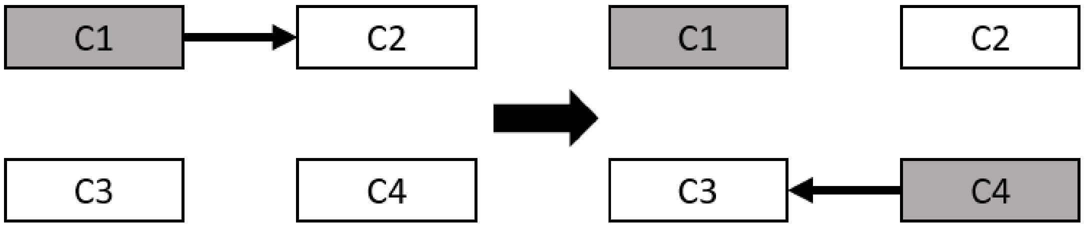


Figure 4.9: Model 2a Scenario 2

Model: Scenario 2 (Figure 4.9)

1. Core c_1 enters HI-crit mode, all of the migratable tasks on Core c_1 will migrate to Core c_2 due to the pairing relationship (Core c_1, Core c_2).

2. Core c_4 enters HI-crit mode, all of the migratable tasks on Core c_4 will migrate to Core c_3 due to the pairing relationship (Core c_3, Core c_4).

Although this model looks quite similar to Model 2, there exists a problem that the migration task load may be quite heavy in certain scenarios, for instance, step 2 in Scenario 1. A detailed explanation on why Model 2a is dominated by Model 2 will be given in the Section 4.2.2.

Model 3

Model 3 is a quite different model from all of the previous ones. In this model, tasks are allowed to migrate to all of the cores currently in the LO-crit mode. Figure 4.10 shows how Model 3 may be viewed as.

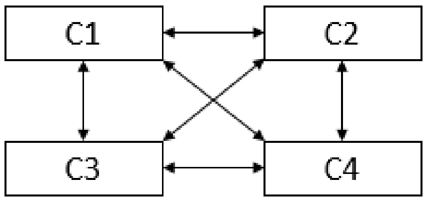


Figure 4.10: Model 3

As shown in the figure, migratable tasks can migrate to all of the cores which makes a maximum usage of the computation ability of the system. Here is a possible scenario of Model 3:

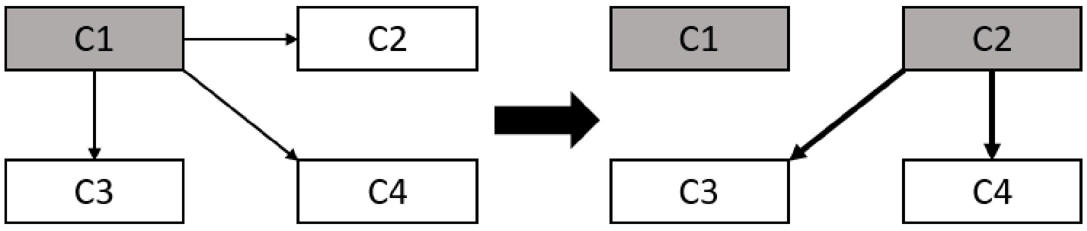


Figure 4.11: Model 3 Scenario 1

Model: Scenario 1 (Figure 4.11)

1. Core c_1 enters HI-crit mode, all of the migratable tasks on Core c_1 will split and migrate to all other cores in LO-crit mode (Core c_2, Core c_3 and Core c_4).
2. Core c_2 enters HI-crit mode, all of the migratable tasks on Core c_2 will split and migrate to all other cores in LO-crit mode (Core c_3 and Core c_4)

4.2.2 Model Analysis

The previous section has introduced four possible allocation models. This section will give a detail exploration of the models proposed, especially based upon response time analysis. It will describe Model 1, Model 2 and Model 3 and their corresponding response time analysis. It will also illustrate why Model 2a is dominated by Model 2.

Model 1

In this model, cores are chained in a circle. Assume a platform contains four cores (c_1, c_2, c_3 and c_4), then cores shall be chained as $c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow c_4 \rightarrow c_1$. If core c_1 enters its HI-crit mode, then LO-crit tasks on core c_1 may migrate to core c_2. If core c_2 also enters HI-crit mode, then these migrated tasks from core c_1 shall migrate to core c_3.

To be detailed, assume that a taskset S contains several tasks with two criticality levels (HI-crit and LO-crit). If this taskset is to be scheduled on a four-core platform by Model 1, then on each core there shall exist three types of tasks: HI-crit tasks, statically allocated LO-crit tasks and migrating LO-crit tasks. Let HI_i represent the set of HI-crit tasks on core c_i, LO_i represent the set of statically allocated LO-crit tasks and $MIG_{i,j,k}$ represent the chain relationship of $i \rightarrow j \rightarrow k$. Then the following relationship can be obtained:

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

In the steady state mode, all these tasks are statically partitioned on each core and executing with their LO-crit budgets. Define state X to represent this phase, then the relationship between tasks and cores can be viewed as:

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

Regarding state X, all tasks are executing with their LO-crit budgets. In this case, the response time analysis of all tasks is given by equation (4.3):

$$\forall \tau_i \in X : R_i = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(LO) \tag{4.3}$$

If a criticality change occurs on one core (c_i), then HI-crit tasks (HI_i) will execute with their HI-crit budgets. For LO-crit tasks, some of them (LO_i) still execute on the core with their LO-crit budgets while the others ($MIG_{i,j,k}$) need to migrate to other cores as there is not enough space for them on the core. Define state $Y(1)$ to represent the case that core c_1 enters its HI-crit mode, then tasks in $MIG_{1,2,3}$ will be migrated from core c_1 to core c_2 and the relationship between tasks and cores can be viewed as:

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

Regarding to this state, the behaviour of the cores is quite similar to the semi-partitioned model analysed in the dual-core platform. According to that, the reduced deadlines and release jitters need to be applied to migrating tasks, and the worst case is given by equation (4.4):

$$\begin{aligned}
&\forall \tau_i \in MIG_{1,2,3}: \\
&\qquad D_i' = D_i - (R_i - C_i(LO)) \\
&\qquad J_i = R_i - C_i(LO)
\end{aligned} \tag{4.4}$$

Thus, the response time analysis of core c_1 and core c_2 in this state is given by equation (4.5):

$$\begin{aligned}
&\forall \tau_i \in Y(1)_1: \\
&\qquad R_i(MIX) = C_i(L_i) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(MIX)}{T_j} \right\rceil C_j(L_i) \\
&\qquad\qquad + \sum_{\tau_k \in chpMIG(i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k(LO) \\
&\forall \tau_i \in Y(1)_2: \\
&\qquad R_i(LO)' = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(LO)' + J_j}{T_j} \right\rceil C_j(LO)
\end{aligned} \tag{4.5}$$

If one more core enters HI-crit mode, there exists two different scenarios. The first scenario is that the core, which does not accept any migrated tasks, enters HI-crit mode. For example, if core c_3 enters HI-crit mode, then all of the migratable tasks on core c_3 shall migrate to core c_4 due to the chained relationship. The relationship between tasks and cores can be viewed as below, where $Y(1,3)$ represents the state that core c_1 enters HI-crit mode first and core c_3 enters HI-crit mode later:

- $Y(1,3)_1 = Y(1)_1$
- $Y(1,3)_2 = Y(1)_2$

- $Y(1,3)_3 = LO_3 \cup HI_3$
- $Y(1,3)_4 = X_4 \cup MIG_{3,4,1}$
- $S = Y(1,3)_1 \cup Y(1,3)_2 \cup Y(1,3)_3 \cup Y(1,3)_4$

The response time analysis equation of core c_3 and core c_4 in this state is same as that of core c_1 and core c_2 in the previous state Y_1.

The second scenario is that the core, which accepts migrated tasks, enters HI-crit mode. In this scenario, not only the migratable tasks on the core, but also the accepted migrated tasks need to migrate to another core. If core c_2 enters HI-crit mode, then all of the migratable tasks on core c_2 shall migrate to core c_3, and the relationship between tasks and cores can be viewed as:

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

Regarding to this scenario, taskset $MIG_{1,2,3}$ migrates a second time. Since the migrate progress will cause the reduced deadline and the release jitter issues to the task, a task migrating a second time may suffer from a further effect caused by the above issues. In other words, the release jitter and the reduced deadline effects are stackable. It is observed that the worst case happens when a task migrates to one core and further migrates to another core in one release. For instance, τ_i waits a maximum time $(R_{i,m} - C_i)$ before it starts to execute on core c_m and then migrates to core c_n. Then it waits another maximum time $(R_{i,n} - C_i)$ before it starts to execute on core c_n and migrates to core c_o. Thus, for these tasks, the worst case of release jitters and reduced deadlines is given by equation (4.6):

$$\begin{aligned}
&\forall \tau_i \in MIG_{1,2,3}: \\
&\qquad D_i'' = D_i - (R_{i,m} - C_i) - (R_{i,n} - C_i) \\
&\qquad J_i' = (R_{i,m} - C_i) + (R_{i,n} - C_i)
\end{aligned} \tag{4.6}$$

However, despite the changes to reduced deadlines and release jitters of the tasks migrating a second time, the response time analysis for other tasks remains the same as that in the other scenario.

If further cores enter HI-crit mode, then all of LO-crit tasks on that core need to be abandoned as the number of cores in HI-crit mode exceeds the boundary number. For example, if core c_3 enters HI-crit mode, then the relationship between tasks and cores can be viewed as:

- $Y(1,2,3)_1 = Y(1)_1$
- $Y(1,2,3)_2 = Y(1,2)_2$
- $Y(1,2,3)_3 = HI_3$
- $Y(1,2,3)_4 = X_4$
- $S = Y(1)_1 \cup Y(1)_2 \cup Y(1)_3 \cup Y(1)_4 \cup MIG_{1,2,3} \cup MIG_{2,3,4}$

Based on the state view above, only HI-crit tasks are executing in core c_3 while all of the LO-crit tasks are abandoned. The response time analysis of core c_3 in this state is given by equation (4.7):

$$\begin{aligned}
&\forall \tau_i \in Y(1,2,3)_3: \\
&\qquad R_i'(HI) = C_i(HI) + \sum_{\tau_j \in chph(i)} \left\lceil \frac{R_i(HI)'}{T_j} \right\rceil C_j(HI) \\
&\qquad\qquad + \sum_{\tau_k \in chpl(i)} \left\lceil \frac{R_i(LO)' + J_k'}{T_k} \right\rceil C_k(LO)
\end{aligned} \tag{4.7}$$

If all of the response time analysis for all possible states have passed, then the taskset is deemed to be schedulable by Model 1.

Model 2

This model pairs the cores so that tasks may only migrate between paired cores. Assume the platform contains four cores (c_1, c_2, c_3 and c_4), then four pairs will be generated: (c_1, c_2), (c_3, c_4), (c_1, c_3) and (c_2, c_4). If core c_1 enters its HI-crit mode, then LO-crit tasks on core c_1 may only migrate to either core c_2 or c_3 but not core c_4. Assume that all of the migratable tasks on core c_1 have migrated to core c_2, if core c_2 also enters HI-crit mode then tasks, which previously migrated to core c_2, will have to migrate to core c_3, while all of the migrating tasks originally on core c_2 will migrate to core c_4. Based on that, the schedulability test of this model can be simplified into several dual-core semi-partitioned models, which is simpler than the previous model.

Assume that a taskset S contains several tasks in two criticality levels (HI-crit and LO-crit). If this taskset is to be scheduled on a four-core platform by semi-partitioned algorithm, then there shall exist three types of tasks on each core: HI-crit tasks, statically allocated LO-crit tasks and migrating LO-crit tasks. Let HI_i represent the set of HI-crit tasks on core c_i, LO_i represent the set of statically allocated LO-crit tasks, and $MIG_{i,j,k}$ represent the set of LO-crit tasks which will migrate from core c_i to core c_j if core c_i enters HI-crit mode and migrate to core c_k if core c_j also enters HI-crit mode. Then the following relationship can be obtained:

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

In the steady state mode, all these tasks are statically partitioned on each core and executing with their LO-crit budgets. Define state X to represent this phase, then the relationship between tasks and cores can be viewed as:

- $X_1 = LO_1 \cup HI_1 \cup MIG_{1,2,3} \cup MIG_{1,3,2}$
- $X_2 = LO_2 \cup HI_2 \cup MIG_{2,1,4} \cup MIG_{2,4,1}$

- $X_3 = LO_3 \cup HI_3 \cup MIG_{3,1,4} \cup MIG_{3,4,1}$
- $X_4 = LO_4 \cup HI_4 \cup MIG_{4,2,3} \cup MIG_{4,3,2}$
- $S = X_1 \cup X_2 \cup X_3 \cup X_4$

Regarding state X, all tasks are executing with their LO-crit budgets. In this case, the response time analysis of all tasks is given by equation (4.8):

$$\forall \tau_i \in X : R_i = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(LO) \tag{4.8}$$

If a criticality change occurs on one core (c_i), then HI-crit tasks (HI_i) will execute with their HI-crit budgets. For LO-crit tasks, some of them (LO_i) still execute on the core with their LO-crit budgets, while the others (MIG_i) need to migrate to other cores as there is not enough space for them on the core. Define state $Y(1)$ to represent the case that core c_1 enters its HI-crit mode, then tasks in MIG_1 will migrate from core c_1 to core c_2 and the relationship between tasks and cores can be viewed as:

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

Regarding state $Y(1)$ where only core c_1 enters HI-crit mode, HI-crit tasks on this core will execute with their HI-crit budgets while LO-crit staying tasks will execute with their LO-crit budgets. All of the tasks on other cores will still execute with their LO-crit budgets. Reduced deadlines and release jitters will be applied to migrating tasks, and the worst case is given by equation (4.9):

$$\begin{aligned} &\forall \tau_i \in MIG_{1,2,3} \cup MIG_{1,3,2} : \\ &\qquad D'_i = D_i - (R_i - C_i(LO)) \\ &\qquad J_i = R_i - C_i(LO) \end{aligned} \tag{4.9}$$

Thus, the response time analysis of this state is given by equation (4.10):

$$\begin{aligned} &\forall \tau_i \in Y(1)_1 : \\ &\quad R_i(MIX) = C_i(L_i) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(MIX)}{T_j} \right\rceil C_j(L_i) \\ &\qquad + \sum_{\tau_k \in chpMIG(i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k(LO) \\ &\forall \tau_i \in Y(1)_2 \cup Y(1)_3 : \\ &\quad R_i(LO) = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(LO) + J_j}{T_j} \right\rceil C_j(LO) \end{aligned} \tag{4.10}$$

If another core enters HI-crit mode, there are two possible scenarios: the core which receives migrated tasks enters its HI-crit mode, and the core which does not have any migrated tasks enters its HI-crit mode. Regarding to the first case, assume core c_4 enters HI-crit mode in state $Y(1)$, then HI-crit tasks on core c_4 will execute with their HI-crit budgets and migratable LO-crit tasks will migrate to core c_2 and core c_3. The relationship between tasks and cores can be viewed as:

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

In this scenario, the response time analysis is similar to the previous state $Y(1)$, which will not be repeated.

Regarding the latter scenario, assume core c_2 enters HI-crit mode in state $Y(1)$, then HI-crit tasks on core c_2 will execute with their HI-crit budgets, migratable LO-crit tasks that originally allocated on core c_2 will all migrate to core c_4 as core c_1 is already in HI-crit mode, and the migratable LO-crit tasks previously migrated from core c_1 will migrate to core c_3. The relationship between tasks and cores can be viewed as:

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

Regarding state $Y(i,j)$, core c_j, which has not accepted any migrated tasks, enters HI-crit mode after core c_i has entered HI-crit mode. After that, HI-crit tasks on this core will execute with their HI-crit budgets while LO-crit staying tasks will execute with their LO-crit budgets. All of the tasks on other cores will still be executing with their LO-crit budgets. Reduced deadlines and release jitters will be applied to migrating tasks, and in this case only migrating tasks originally from core c_i will suffer from further reduced deadlines and release jitters issues. Equation (4.11) shows the worst case:

$$\begin{aligned} &\forall \tau_i \in MIG_{1,2,3}: \\ &\qquad D_i" = D_i' - (R_i(LO) - C_i(LO)) \\ &\qquad J_i' = J_i + (R_i(LO) - C_i(LO)) \end{aligned} \tag{4.11}$$

Thus, the response time analysis of core c_2, core c_3 and core c_4 in state $Y(i,j)$ is given by equation (4.12):

$$\forall \tau_i \in Y(1,2)_2 :$$
$$\begin{aligned} R_i(MIX)' = C_i(L_i) &+ \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(LO)}{T_j} \right\rceil C_j(L_i) \\ &+ \sum_{\tau_k \in chpMIG(i)} \left\lceil \frac{R_i(LO) + J_k}{T_k} \right\rceil C_k(LO) \end{aligned} \tag{4.12}$$
$$\forall \tau_i \in Y(1,2)_3 \cup Y(1,2)_4 :$$
$$R_i(LO)' = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(LO)' + J_i'}{T_j} \right\rceil C_j(LO)$$

If further cores enter HI-crit mode, then both migration and non-migration LO-crit tasks on the mode changing core need to be abandoned to guarantee the execution of HI-crit tasks as the number of cores in HI-crit mode exceeds the boundary number. Assume core c_3 enters HI-crit mode in state $Y(1,2)$, then the relationship between tasks and cores can be viewed as:

- $Y(1,2,3)_1 = Y(1)_1$
- $Y(1,2,3)_2 = LO_2 \cup HI_2$
- $Y(1,2,3)_3 = HI_3$
- $Y(1,2,3)_4 = X_4 \cup MIG_(2,1,4) \cup MIG_{2,4,1}$
- $S = Y(1,2,3)_1 \cup Y(1,2,3)_2 \cup Y(1,2,3)_3 \cup Y(1,2,3)_4$
  $\cup LO_3 \cup (MIG_{3,1,4} \cup MIG_{3,4,1}) \cup (MIG_{1,3,2} \cup MIG_{1,2,3})$

Regarding this state, only HI-crit tasks on core c_3 are executing with their HI-crit budgets while all migrating LO-crit tasks on the core are abandoned. The response time analysis of core c_3 in this state is given by equation (4.13):

$$\forall \tau_i \in Y(1,2,3)_3 :$$
$$\begin{aligned} R_i(HI)' = C_i(HI) &+ \sum_{\tau_j \in chpH(i)} \left\lceil \frac{R_i(HI)'}{T_j} \right\rceil C_j(HI) \\ &+ \sum_{\tau_k \in chpL(i)} \left\lceil \frac{R_i(LO)' + J_k'}{T_k} \right\rceil C_k(LO) \end{aligned} \tag{4.13}$$

If all of the response time analysis for all possible states have passed, then the taskset is deemed to be schedulable by Model 2.

Model 2a

As stated before, Model 2a is quite similar to Model 2. It also pairs the cores into several groups and migrating tasks may only migrate within the groups. However, instead of splitting the tasks, Model 2a fully migrates all of the migratable tasks from one core to another. Thus, the response time analysis for Model 2a is similar to that for Model 2. But due to the different migration mechanism, we observe that Model 2a requires a heavier migration load than Model 2 in certain scenarios, which causes some tasksets to be schedulable by Model 2 but not Model 2a. For example, assume a taskset S contains 16 tasks, and has been assigned to a four-core platform as shown in Table 4.3.

According to the example, as the tasks on each core have the same parameters, a schedulability check on core c_1 is representative for all of the cores in the steady mode. If a non-migration algorithm is tried to schedule the taskset, we can get the response time as in Table 4.4. From the table, we can get $R_4(HI) \geq 87$ which is larger than its deadline. According to that, this taskset is un-schedulable by the non-migration algorithm.

If Model 2 is used to schedule the taskset, we may assume that cores are paired as (c_1, c_2), (c_3, c_4), (c_1, c_3) and (c_2, c_4). In this case, the response time analysis of tasks on core c_1 can be viewed in Table 4.5.

If core c_1 enters HI-crit mode, task τ_2 will migrate to core c_3 and task τ_3 will migrate to core c_2. Since task τ_3 has a larger release jitter and a smaller reduced deadline, the schedulability test on core c_2 will be harder than that on core c_3. Thus, examining the schedulability on core c_2 will be sufficient in this special example. The response time analysis of tasks on core c_2 can be viewed in Table 4.6.

If core c_2 also enters HI-crit mode, task τ_3 will migrate to core c_3 while task τ_6

Task	C(LO)	C(HI)	T	D	J	L	c
τ_1	1	2	10	10	-	HI	c_1
τ_2	4	-	35	35	-	LO	c_1
τ_3	4	-	35	35	-	LO	c_1
τ_4	5	45	85	85	-	HI	c_1
τ_5	1	2	10	10	-	HI	c_2
τ_6	4	-	35	35	-	LO	c_2
τ_7	4	-	35	35	-	LO	c_2
τ_8	5	45	85	85	-	HI	c_2
τ_9	1	2	10	10	-	HI	c_3
τ_{10}	4	-	35	35	-	LO	c_3
τ_{11}	4	-	35	35	-	LO	c_3
τ_{12}	5	45	85	85	-	HI	c_3
τ_{13}	1	2	10	10	-	HI	c_4
τ_{14}	4	-	35	35	-	LO	c_4
τ_{15}	4	-	35	35	-	LO	c_4
τ_{16}	5	45	85	85	-	HI	c_4

Table 4.3: Example Taskset

and task τ_7 will migrate to core c_4. Since task τ_6 and task τ_7 have larger release jitters and smaller reduced deadlines, the schedulability test on core c_4 will be harder than that on core c_3. Thus, examine the schedulability on core c_4 will be sufficient in this special example. The response time analysis of tasks on core c_4 can be viewed in Table 4.7.

According to the results, all of the response time are smaller than the corresponding deadlines, which indicates that the taskset is deemed to be schedulable by Model 2.

If Model 2a is used to schedule the taskset, same as previously, we may assume that cores are paired as (c_1, c_2), (c_3, c_4), (c_1, c_3) and (c_2, c_4). In this steady state mode, the response time analysis of tasks on core c_1 shall be the same as that in Model 1. If core c_1 enters HI-crit mode, task τ_2 and task τ_3 will migrate to core

Task	C(LO)	C(HI)	T	D	J	L	c	p	R(LO)	R(HI)
τ_1	1	2	10	10	-	HI	c_1	1	1	2
τ_2	4	-	35	35	-	LO	c_1	2	5	6
τ_3	4	-	35	35	-	LO	c_1	3	9	10
τ_4	5	45	85	85	-	HI	c_1	4	15	≥ 87

Table 4.4: Example Core Analysis

Task	C(LO)	C(HI)	T	D	J	L	c	p	R(LO)	R(HI)
τ_1	1	2	10	10	-	HI	c_1	1	1	2
τ_2	4	-	35	35	-	LO	c_1	2	5	-
τ_3	4	-	35	35	-	LO	c_1	3	9	-
τ_4	5	45	85	85	-	HI	c_1	4	15	67

Table 4.5: Example Core Analysis

c_2. The response time analysis of tasks on core c_2 can be viewed in Table 4.8.

If core c_2 also enters HI-crit mode, task τ_2 and task τ_3 will migrate to core c_3 while task τ_6 and task τ_7 shall migrate to core c_4. Since task τ_6 and task τ_7 have larger release jitters and smaller reduced deadlines, the schedulability test on core c_4 will be harder than that on core c_3. Thus, examine the schedulability on core c_4 will again be sufficient in this special example. The response time analysis of tasks on core c_4 can be viewed in Table 4.9.

It can be observed that due to the release jitter, $R_4(LO)$ is much larger than that in Model 2 which leads to a failure on scheduling τ_4 in the HI-crit mode. According to that, this taskset is unschedulable by Model 2a.

The above example is quite straightforward due to the tasks on each core are the same in static mode. In actual cases, all of possible migrations need to be checked. However, the example is sufficient to illustrate that Model 2a is dominated by Model 2.

Task	C(LO)	C(HI)	T	D	J	L	c	p	R(LO)	R(HI)
τ_5	1	2	10	10	-	HI	c_2	1	1	2
τ_3	4	-	35	30	5	LO	c_2	2	5	-
τ_6	4	-	35	35	-	LO	c_2	3	9	-
τ_7	4	-	35	35	-	LO	c_2	4	14	-
τ_8	5	45	85	85	-	HI	c_2	5	19	73

Table 4.6: Example Core Analysis

Task	C(LO)	C(HI)	T	D	J	L	c	p	R(LO)	R(HI)
τ_{13}	1	2	10	10	-	HI	c_4	1	1	2
τ_7	4	-	35	25	10	LO	c_4	2	5	-
τ_6	4	-	35	30	5	LO	c_4	3	9	-
τ_{14}	4	-	35	35	-	LO	c_4	4	14	-
τ_{15}	4	-	35	35	-	LO	c_4	5	18	-
τ_{16}	5	45	85	85	-	HI	c_4	6	24	77

Table 4.7: Example Core Analysis

Model 3

In this model, tasks are allowed to migrate to any possible core to maximize the scheduling flexibility. When only one core enters HI-crit mode, then some LO-crit tasks may stay on the core executing with their LO-crit executing budgets while some other LO-crit tasks will migrate to other cores. In this model, these migrating LO-crit tasks will be separated "equally" to all cores. This "equally" here not only represents the number of tasks but also needs to consider the sum of the utilization of the migration tasks on each core. If another core also enters HI-crit mode, then some LO-crit tasks, which are originally executing on the core, may stay on the core executing with their LO-crit executing budgets, while some other LO-crit tasks and the LO-crit tasks migrated to the core will migrate to other cores which are in LO-crit mode. These migrating tasks will also migrate "equally". If a further core enters HI-crit mode, then all of the LO-crit tasks on the core will be abandoned in order to guarantee the execution of the HI-crit

Task	C(LO)	C(HI)	T	D	J	L	c	p	R(LO)	R(HI)
τ_5	1	2	10	10	-	HI	c_2	1	1	2
τ_3	4	-	35	30	5	LO	c_2	2	5	-
τ_2	4	-	35	35	1	LO	c_2	3	9	-
τ_6	4	-	35	35	-	LO	c_2	4	14	-
τ_7	4	-	35	35	-	LO	c_2	5	18	-
τ_8	5	45	85	85	-	HI	c_2	6	24	77

Table 4.8: Example Core Analysis

Task	C(LO)	C(HI)	T	D	J	L	c	p	R(LO)	R(HI)
τ_9	1	2	10	10	-	HI	c_4	1	1	2
τ_7	4	-	35	21	14	LO	c_4	2	5	-
τ_6	4	-	35	25	10	LO	c_4	3	9	-
τ_{10}	4	-	35	35	-	LO	c_4	4	14	-
τ_{11}	4	-	35	35	-	LO	c_4	5	18	-
τ_{12}	5	45	85	85	-	HI	c_4	6	33	≥ 87

Table 4.9: Example Core Analysis

tasks.

Assume that a taskset S contains several tasks in two criticality levels (HI-crit and LO-crit). If this taskset is to be scheduled on a four-core platform (c_1, c_2, c_3 and c_4) by semi-partitioned algorithm, then there shall exist three types of tasks on each core: HI-crit tasks, statically allocated LO-crit tasks and migrating LO-crit tasks. Let HI_i represent the set of HI-crit tasks on core c_i, LO_i represent the set of statically allocated LO-crit tasks and $MIG_{i,j,k}$ represent the set of LO-crit tasks that will migrate from core c_i to core c_j if core c_i enters HI-crit mode and migrate to core c_k if core c_j also enters HI-crit mode. Then the following relationship can be obtained:

- $S = (LO_1 \cup LO_2 \cup LO_3 \cup LO_4) \cup (HI_1 \cup HI_2 \cup H_3 \cup H4)$
  $\cup ((MIG_{1,2,3} \cup MIG_{1,2,4}) \cup (MIG_{1,3,2} \cup MIG_{1,3,4}) \cup (MIG_{1,4,2} \cup MIG_{1,4,3}))$
  $\cup ((MIG_{2,1,3} \cup MIG_{2,1,4}) \cup (MIG_{2,3,1} \cup MIG_{2,3,4}) \cup (MIG_{2,4,1} \cup MIG_{2,4,3}))$

$\cup((MIG_{3,1,2} \cup MIG_{3,1,4}) \cup (MIG_{3,2,1} \cup MIG_{3,2,4}) \cup (MIG_{3,4,1} \cup MIG_{3,4,2}))$
$\cup((MIG_{4,1,2} \cup MIG_{4,1,3}) \cup (MIG_{4,2,1} \cup MIG_{4,2,3}) \cup (MIG_{4,3,1} \cup MIG_{4,3,2}))$

In the steady state mode, all these tasks are statically partitioned on each core and executing with their LO-crit budgets. Define state X to represent this phase, then the relationship between tasks and cores can be viewed as:

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

Regarding state X, all tasks are executing with their LO-crit budgets. In this case, the response time analysis of all tasks can be viewed as equation (4.14):

$$R_i = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(LO) \tag{4.14}$$

If a criticality change occurs on one core (c_i), then HI-crit tasks (HI_i) will execute with their HI-crit budgets. For LO-crit tasks, some of them (LO_i) still execute on the core with their LO-crit budgets while the others (MIG_i) need to migrate to other cores as there is not enough space for them on the core. Unlike the Dual-core model, these migrating tasks may migrate to all possible cores rather than only one core. Define state $Y(1)$ to represent the case that core c_1 enters its HI-crit mode, then tasks in MIG_1 will be migrated from core c_1 to core c_2 and the relationship between tasks and cores can be viewed as:

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

Regarding state $Y(i)$ where only core c_i enters HI-crit mode, HI-crit tasks will execute with their HI-crit budgets while some LO-crit tasks will execute with their LO-crit budgets on the core c_i. All of other tasks will execute with their LO-crit budgets. Reduced deadlines and release jitters will be applied to migrating tasks, and the worst case of these migrating tasks is given by equation (4.15):

$$\begin{aligned}
&\forall \tau_i \in MIG_{1,2,3} \cup MIG_{1,2,4} \cup MIG_{1,3,2} \cup MIG_{1,3,4} \cup MIG_{1,4,2} \cup MIG_{1,4,3} : \\
&\qquad D'_i = D_i - (R_i - C_i(LO)) \\
&\qquad J_i = R_i - C_i(LO)
\end{aligned} \tag{4.15}$$

Thus, the response time analysis for state $Y(i)$ is given by equation (4.16):

$$\begin{aligned}
&\forall \tau_i \in Y(1)_1 : \\
&\qquad R_i(MIX) = C_i(L_i) + \sum_{\tau_j \in chpS(i)} \left\lceil \frac{R_i(MIX)}{T_j} \right\rceil C_j(L_i) \\
&\qquad\qquad + \sum_{\tau_k \in chpMIG(i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k(LO) \\
&\forall \tau_i \in Y(1)_2 \cup Y(1)_3 \cup Y(1)_4 : \\
&\qquad R_i(LO) = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(LO) + J_j}{T_j} \right\rceil C_j(LO)
\end{aligned} \tag{4.16}$$

If a further criticality change occurs on core c_j, then all of the migratable LO-crit tasks on this core needs to migrate to other cores while all of the HI-crit tasks execute with their HI-crit budgets. Define state $Y(1,2)$ to represent the case that core c_2 also enters HI-crit mode after core c_1 enters HI-crit mode, the relationship between tasks and cores can be viewed as:

- $Y(1,2)_1 = LO_1 \cup HI_1$
- $Y(1,2)_2 = LO_2 \cup HI_2$
- $Y(1,2)_3 = Y(1)_3 \cup (MIG_{2,3,1} \cup MIG_{2,3,4}) \cup MIG_{2,1,3} \cup MIG_{1,2,3}$
- $Y(1,2)_4 = Y(1)_4 \cup (MIG_{2,4,1} \cup MIG_{2,4,3}) \cup MIG_{2,1,4} \cup MIG_{1,2,4}$
- $S = Y(1,2)_1 \cup Y(1,2)_2 \cup Y(1,2)_3 \cup Y(1,2)_4$

Regarding this state, HI-crit tasks will still execute with HI-crit budgets while some LO-crit tasks will execute with LO-crit budgets on the core c_i. All of the other tasks will execute with their LO-crit budgets. Taskset $MIG_{1,2,3}$ and $MIG_{1,2,4}$ will migrate a second time. As discussed in Model 1, further reduced deadlines and release jitters will be applied to these migrating tasks and the worst case of them is given by equation (4.17):

$$\begin{aligned}
&\forall \tau_i \in MIG_{1,2,3} \cup MIG_{1,2,4} : \\
&\qquad D_i" = D_i' - (R_i(LO) - C_i(LO)) \\
&\qquad J_i' = J_i + (R_i(LO) - C_i(LO))
\end{aligned} \tag{4.17}$$

Thus, the response time analysis for core c_2, c_3 and c_4 in this state is given by equation (4.18):

$$\forall \tau_i \in Y(1,2)_2 :$$

$$\begin{aligned} R_i(MIX)' = C_i(L_i) &+ \sum_{\tau_j \in chpS(i)} \left\lceil \frac{R_i(MIX)'}{T_j} \right\rceil C_j(L_i) \\ &+ \sum_{\tau_k \in chpMIG(i)} \left\lceil \frac{R_i(LO) + J_k}{T_k} \right\rceil C_k(LO) \end{aligned} \tag{4.18}$$

$$\forall \tau_i \in Y(1,2)_3 \cup Y(1,2)_4 :$$

$$R_i(LO)' = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(LO)' + J'_j}{T_j} \right\rceil C_j(LO)$$

If more cores enter HI-crit mode, then only HI-crit tasks on these cores will remain executing while all LO-crit tasks on these cores need to be abandoned. Assume core c_3 enters HI-crit mode, then the relationship between tasks and cores can be viewed as:

- $Y(1,2,3)_1 = LO_1 \cup HI_1$
- $Y(1,2,3)_2 = LO_2 \cup HI_2$
- $Y(1,2,3)_3 = HI_3$
- $Y(1,2,3)_4 = Y(1)_4 \cup (MIG_{2,4,1} \cup MIG_{2,4,3}) \cup MIG_{2,1,4} \cup MIG_{1,2,4}$
- $S = Y(1,2)_1 \cup Y(1,2)_2 \cup Y(1,2)_3 \cup Y(1,2)_4 \cup LO_3 \cup (MIG_{3,1,2} \cup MIG_{3,1,4}) \cup (MIG_{3,2,1} \cup MIG_{3,2,4}) \cup (MIG_{3,4,1} \cup MIG_{3,4,2}) \cup (MIG_{1,3,2} \cup MIG_{1,3,4}) \cup (MIG_{2,3,1} \cup MIG_{2,3,4}) \cup MIG_{2,1,3} \cup MIG_{1,2,3}$

Based on the state, all migrating LO-crit tasks on core c_3 are abandoned while HI-crit tasks are executing with their HI-crit budgets. The response time analysis for the core in such state is given by equation (4.19):

$$\forall \tau_i \in Y(1,2,3)_3 :$$

$$\begin{aligned} R_i(HI) = C_i(HI) &+ \sum_{\tau_j \in chpH(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(HI) \\ &+ \sum_{\tau_k \in chpL(i)} \left\lceil \frac{R_i(LO)' + J'_k}{T_k} \right\rceil C_k(LO) \end{aligned} \tag{4.19}$$

If all of the response time analysis for all possible states have passed, then the taskset is deemed to be schedulable by Semi-partitioned Model 3.

4.2.3 Evaluation of the Models

The previous section has derived sufficient response time analysis for all of the allocation models introduced. In this section, we will introduce an experiment to compare the scheduling efficiency of the allocation models. At the end of this section, the comparison results will be studied and a recommended approach is proposed.

Experiment Configuration

Software is developed to explore the efficiency of the three models. The configuration of the software is fairly similar to that mentioned in Section 3.2.1. The software consists of three parts. The first part generates tasksets and stores these tasksets in XML files. Each tasksets node contains 10000 tasksets. In order to gain uniform distributed parameters, UUnifast-discard algorithm is used to generate nominal utilizations and Log-uniform algorithm [51] is used to generate periods. Other parameters of each task are calculated based on these two values. The second part of the software pre-sorts each taskset in criticality-aware utilization descending order. The last part of the software contains the response time analysis mentioned in Section 4.2.2 and explores the scheduling success rate of the three models.

Results and Comparison

We investigate the performance of Model 1, Model 2 and Model 3 and compare them with the non-migration algorithm. The non-migration algorithm is chosen as the lowest bound of performance. Figure 4.12 shows the percentage of the tasksets that are schedulable for a system of 24 tasks (in the setting that half of the tasks are HI-crit tasks and the criticality factor is 2, $P = 0.5, f = 2$). The Y-axis shows the percentage of the successfully scheduled tasksets while the X-axis shows the sum of nominal utilizations of the tested taskset. The sum of utilizations ranges from 3.2 to 4.6 in steps of 0.028 to amplify the view of the results.

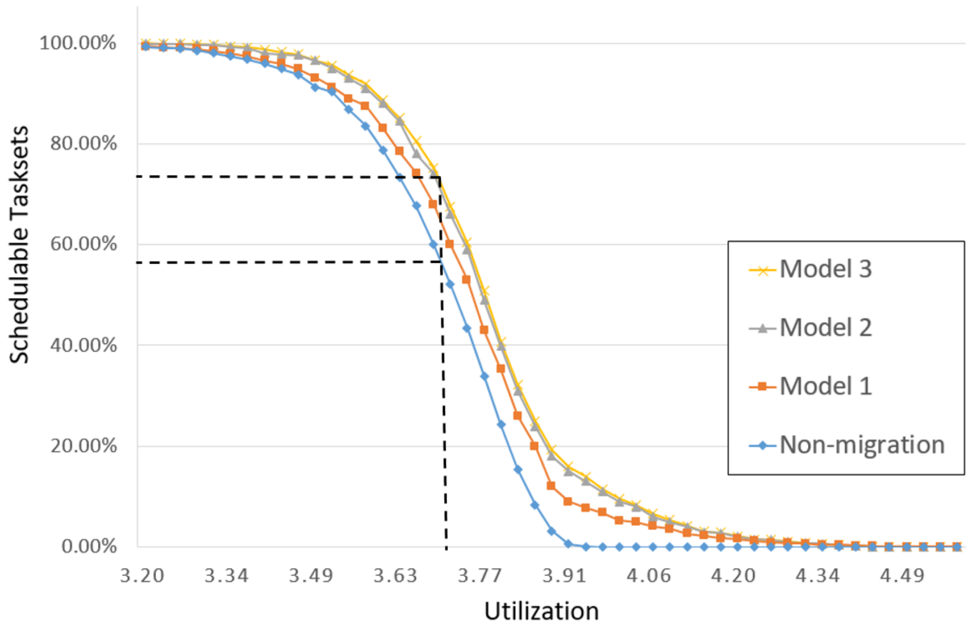


Figure 4.12: Percentage of Schedulable Tasksets

From the above figure, it can be observed that all of the models outperform the non-migration one by a considerable margin. For example, as shown by the black lines, Model 3 can schedule around 75% of the tasksets when the taskset utilization is around 3.7, while non-migration model can only schedule around 57% of the tasksets. The improvement of schedulability from Model 3 towards non-migration is about $\frac{75-57}{57} * 100\% = 31.58\%$, which is significant. Comparing all of the semi-partitioned methods, Model 3 has the best performance, but the difference between Model 3 and Model 2 is not large.

In order to explore the performance of the algorithms relating to criticality factor $(C(HI)/C(LO))$ and the percentage of HI-crit tasks, weighted schedulability measurement is also used. We show how the results are changed by varying one key parameter at a time. Figure 4.13 varies the criticality factor, Figure 4.14 varies the percentage of HI-crit tasks and Figure 4.15 varies the size of the taskset. The X-axis stands for the parameter examined and Y-axis represents the weighted value. According to Figure 4.13, Model 3 has the best performance, while Model 2 provides slightly less schedulability. In addition, both models have increased performance as the criticality factor increases. This is to be expected as the increase of WCET difference between different criticality levels allows more scheduling potential for migrating tasks.

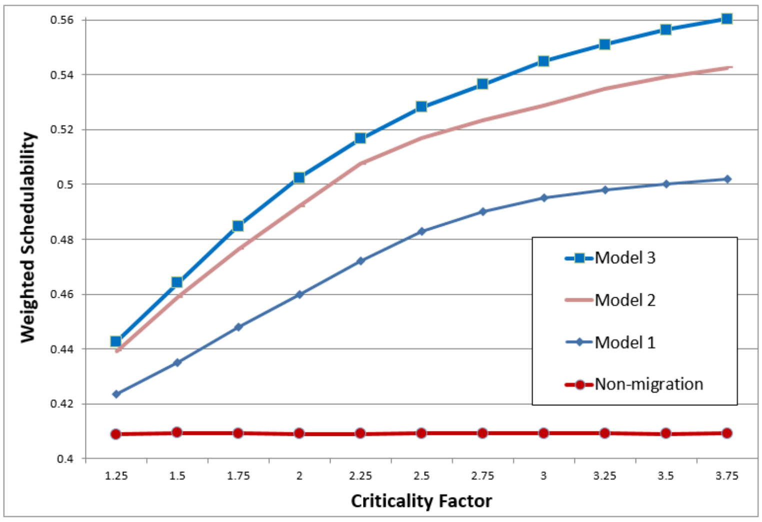


Figure 4.13: Varying the Criticality Factor

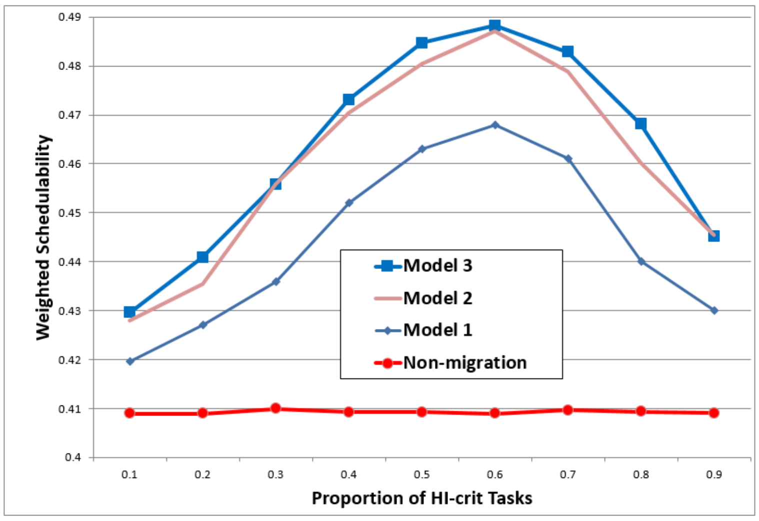


Figure 4.14: Varying the Criticality Percentage

According to Figure 4.14, the performance of the semi-partitioned algorithms has formed an inverted U-shape curve since each end of the interval represents a one-criticality taskset, and hence the priorities are optimal. Regarding to the

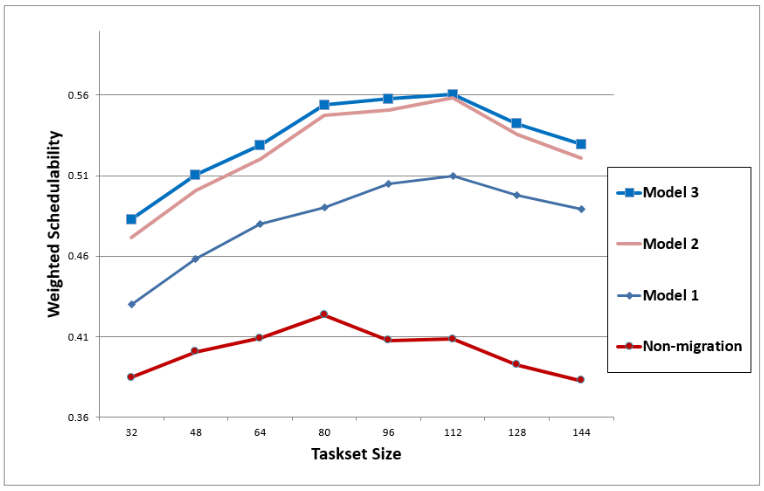


Figure 4.15: Varying the Taskset Size

individual performance, Model 3 has the best performance while Model 2 provides slightly less schedulability. In addition, it is observed that the difference between Model 3 and Model 2 decreases when the percentage of the criticality tasks is approaching 0.6.

According to Figure 4.15, the performance of the semi-partitioned algorithms also forms an inverted U-shape curve. This is expected as tasks are relatively large in small sized tasksets which adds difficulty in finding acceptable migrating tasks, while in large sized tasksets, the interference from high priority tasks increases, as well as the effects from release jitters, which adds difficulty to the schedulability of migrated tasks with reduced deadlines. Regarding to the individual performance, Model 3 still has the best performance while the performance of Model 2 is slightly poorer.

4.2.4 Recommended Approach

Overall, it is observed that Model 3 provides the best schedulability in all cases. However, the schedulability difference between Model 3 and Model 2 is not that significant. As it has been argued that Model 3 has much more complex scheduling analysis, Model 2 is suggested to be the most appropriate model for 4-core MCS with two criticality levels.

4.3 Extending to n-core Platform

The previous section has indicated that migration Model 2, which pairs the cores into groups, is the most suitable and scalable migration model for multi-core platforms. This section will discuss how to extend Model 2 to an n-core platform. It will first show two detailed examples of extending the migration model to an 8-core platform and a 7-core platform. Then it will show a general extending mechanism.

4.3.1 8-core Platform Example

In a 4-core platform, the boundary number is 2 and each core has two paired cores. In addition, migratable tasks are split into two parts when migrating. Based on this information, it is reasonable to assume that for a 8-core platform, since the boundary number is 3, each core shall have three paired cores and migratable tasks shall be split into three parts when migrating. Thus, the problem becomes how the cores can be paired to fulfill the above requirements. We propose a clone algorithm to solve the problem.

- Assume that there exists a 4-core platform (c_1, c_2, c_3, c_4) and cores are paired into four pairs $(c_1, c_2), (c_1, c_3), (c_2, c_4), (c_3, c_4)$ as previous in Model 2.
- Create a clone 4-core platform (c'_1, c'_2, c'_3, c'_4) and pair the cores in the same way $(c'_1, c'_2), (c'_1, c'_3), (c'_2, c'_4), (c'_3, c'_4)$.
- Pair the original cores with the clone cores: $(c_1, c'_1), (c_2, c'_2), (c_3, c'_3), (c_4, c'_4)$.
- Replace c'_1 by c_5, c'_2 by c_6, c'_3 by c_7 and c'_4 by c_8 in all of the pairs generated.

According to the algorithm, we can obtain the pairing relationship for a 8-core platform as following: (c_1, c_2), (c_1, c_3), (c_2, c_4), (c_3, c_4), (c_5, c_6), (c_5, c_7), (c_6, c_8), (c_7, c_8), (c_1, c_5), (c_2, c_6), (c_3, c_7), (c_4, c_8), which fulfills the requirement that each core is paired with three different cores. In addition, by using the same algorithm iteratively, we can extend the pairing relationship to 16-core platforms, 32-core platforms, ..., 2^n-core platforms.

4.3.2 7-core Platform Example

A 7-core platform is a quite special case. According to the semi-partitioned model definition, the boundary number for this platform is still 3 but it is not possible to pair the cores so that each core has three different paired cores. This can be proved by contradiction as following:

1. Assume there exists a pairing method to pair 7 cores so that each core is paired to three different cores.
2. There exist $3 \times 7 = 21$ relationships between the cores.
3. Pair one core to another always results in 2 relationships.
4. There does not exist a possibility to create an odd number of pairing relationships. Contradiction found.
5. Therefore, 7 cores cannot be paired into groups so that each core has three different paired cores.

As the boundary number calculated by $\lceil log_2(n) \rceil$ is slightly larger than the exact boundary number, it is acceptable to make one core have a smaller boundary number (2 in this scenario) and only pair with two cores. Thus, a modified clone algorithm can be used to solve the pairing problem for a 7-core platform.

- Assume that there exists a 4-core platform (c_1, c_2, c_3, c_4) and cores are paired into four pairs $(c_1, c_2), (c_1, c_3), (c_2, c_4), (c_3, c_4)$ as previous in Model 2.
- Create a clone 4-core platform (c'_1, c'_2, c'_3, c'_4) and pair the cores in the same way $(c'_1, c'_2), (c'_1, c'_3), (c'_2, c'_4), (c'_3, c'_4)$.
- Pair the original cores with the clone cores: $(c_1, c'_1), (c_2, c'_2), (c_3, c'_3), (c_4, c'_4)$.

- Replace c_1' by c_5, c_2' by c_6, c_3' by c_7 and c_4' by c_8 in all of the pairs generated.
- Delete all of the pairing relationship with core c_8: $(c_6, c_8), (c_7, c_8), (c_4, c_8)$.
- For each two deleted pairing relationship, create a new pairing relationship between two different cores excluding core c_8: (c_6, c_7)

According to the algorithm, we can get the pairing relationship for a 7-core platform as following: (c_1, c_2), (c_1, c_3), (c_2, c_4), (c_3, c_4),(c_5, c_6), (c_5, c_7), (c_1, c_5), (c_2, c_6), (c_3, c_7), (c_6, c_7). Regarding this pairing relationship, for core c_1, c_2, c_3, c_5, c_6, c_7, each has three paired mates, while core c_4 has two paired mates.

4.3.3 General Algorithm

Regarding to a n-core platform, the boundary number is $\lceil log_2(n) \rceil$. There exist two different situations: n is an even number or n is an odd number. If n is an even number, then there exists an integer k such that $n = 2 \times k$. In order to apply Model 2, the cores in the system require to be paired so that each core has $\lceil log_2(n) \rceil$ paired mates. By applying the clone algorithm, the pairing relationship of the n-core platform can be generated by finding the pairing relationship of a k-core platform where the boundary number is $\lceil log_2(n) \rceil - 1 = \lceil log_2(2 \times k) \rceil - 1 = log_2(2) + \lceil log_2(k) \rceil - 1 = \lceil log_2(k) \rceil$. Thus, finding the pairing relationship of a k-core platform will solve the pairing problem for this n-core platform.

If n is an odd number, then there exists an integer k' such that $n = 2 \times k' - 1$. Similar to the previous scenario, in order to apply Model 2, the cores in the system require to be paired so that most cores have $\lceil log_2(n) \rceil$ paired mates. By applying the modified clone algorithm, the pairing relationship of the n-core platform can be generated by finding the pairing relationship of a k'-core platform where the boundary number is $\lceil log_2(n) \rceil - 1 = \lceil log_2(2 \times k' - 1) \rceil - 1 = log_2(2) + \lceil log_2(k') \rceil - 1 = \lceil log_2(k') \rceil$. Thus, finding the pairing relationship of a k'-core platform will solve the pairing problem for this n-core platform. In all, the pairing relationship of a n-core platform can be generated by a recursion usage of the clone algorithm and the modified clone algorithm. When the pairing relationship between cores is settled, migratable tasks may be split into the boundary number of groups and migrate to the paired cores when required by the system.

4.4 Summary

This chapter has explored the semi-partitioned model on a multi-core platform. It first addresses the boundary number determination problem by the use of a probability calculation. In consideration of easier usage, it is proposed that $\lceil log_2(n) \rceil$ shall be used as the boundary number for an n-core system. That is, for a n-core system, all tasks shall remain schedulable if no more than $\lceil log_2(n) \rceil$ cores enter HI-crit mode. This chapter then explores the task allocation problem on a four-core platform. Four task allocation models are proposed and analysed by response time analysis and experiments. According to the results observed and the consideration of calculation complexity, it is suggested that Model 2, which splits the migration task load within paired cores, will be the most appropriate task allocation model for a four-core system. In addition, this chapter has provided an iterative algorithm to manipulate a possible pairing relationship for an n-core platform to apply Model 2. In the previous chapter, we illustrate that the combination usage of Semi2WF and Semi2FF provides the best scheduling performance for a dual-core platform. In other words, when the migration source core and the migration destination core are fixed, Semi2 algorithm is an appropriate approach to determine which task to be migratable. Based on that, we propose an appropriate semi-partitioned model for a n-core system as:

- Each core is paired with $\lceil log_2(n) \rceil$ cores.
- Semi2 approach is used to determine which LO-crit tasks shall be migratable.
- If Core c_i enters HI-crit mode and the total number of the cores in HI-crit mode is no more than $\lceil log_2(n) \rceil$, migratable tasks on Core c_i migrate

"equally" (by the use of WF bin-packing algorithm) to the paired cores which are still in LO-crit mode. All tasks guarantee their executions.

- If Core c_i enters HI-crit mode and the total number of the cores in HI-crit mode is more than $\lceil log_2(n) \rceil$, all LO-crit tasks on Core c_i will be abandoned. Only HI-crit tasks guarantee their executions.

Chapter 5

Extended NoC Version with More Criticality Levels

The previous chapters have introduced appropriate approaches to schedule tasks in a multi-core platform with two criticality levels. However, the result has assumed that the migration costs between different cores are the same. In this chapter, we will extend the model to multi-criticality levels. In addition, we will explore the system architecture influence on the semi-partitioned model.

This chapter will first discuss the influence from increasing the number of criticality levels, and propose how the model can be extended. Then it will discuss the effects of considering the migration difference between cores and provide a detailed analysis of the issues when extending the semi-partitioned model to Network-on-Chip-based multi-core platforms (NoC for short). Then, this chapter will introduce a new model for scheduling tasks on a multi-core NoC system with three criticality levels. An experiment is set up to show how the semi-partitioned model outperforms the non-migration model. An evaluation will be made at the end.

5.1 Extend Criticality Levels

This section will focus on extending the semi-partitioned model to more than two criticality levels. The direct influence from increasing the number of criticality levels is that each task has more WCET estimations for different criticality levels. In a system with n criticality levels, each task shall have n WCET estimations for different levels by definition. Thus, scheduling such a system will be quite complicated. In order to ease the calculation capacity and the complexity of the model, we adopt a model that only uses two WCET estimations for each task ([29],[76]). One is the WCET estimation at the lowest criticality level, while the other is the WCET for the task at its own criticality level. Take a three criticality system ($LO < MID < HI$) as an example, we may get the following performances based on this model:

- All of the tasks execute with their LO-crit budgets by default.
- If a MID-crit task exceeds its LO-crit budget, then the system will enter MID-crit mode, in which all of the MID-crit tasks will execute with their MID-crit budgets, while LO-crit tasks and HI-crit tasks will still execute with their LO-crit budgets.
- If a HI-crit task exceeds its LO-crit budget, then the system will enter HI-crit mode, in which all of the HI-crit tasks will execute with their HI-crit budgets, MID-crit tasks will execute with their MID-crit budgets, and LO-crit tasks will still execute with their LO-crit budgets.

According to this behaviour, it can be observed that tasks with the lowest criticality level will only have one WCET estimation. For other tasks, if the task exceeds its WCET budget for the lowest criticality level, it will execute with the other WCET estimation and the system will increase to the criticality level of that task. Due to that, the change of the criticality level of a core may skip the middle levels. For example, the core may increase directly from LO to HI when a HI-crit task exceeds its LO-crit budget. In addition, tasks will still be executing with their own criticality budgets if the system is executing in a higher criticality level. In all, a general model for a core with n criticality levels (0 represents the lowest level and n represents the highest level) can be generated as following:

- All tasks execute with their WCETs for criticality level 0 ($C(0)$) by default.
- No task may exceed its own criticality budget.
- If task τ_i with $L_i > 0$ exceeds its 0-crit budget ($C_i(0)$), the system will enter L_i-crit mode and task τ_i will execute with its own criticality budget $C_i(L_i)$. If task τ_k has a smaller or equal criticality level ($L_k \leq L_i$), it will execute with its highest criticality level budget ($C_k(L_k)$); otherwise, it will still execute with the lowest criticality budget ($C_k(0)$).
- No task is abandoned.

5.1.1 Model and Analysis

Although the aim of the semi-partitioned model is to allow all tasks to remain schedulable, it is unwise to fail the schedulability test due to guaranteeing the execution of the lowest criticality level tasks when the system is in a high criticality level. So it is essential to redefine the criticality level to be assured of in the semi-partitioned model for a multi-core platform. A reasonable suggestion is to try to save all of the tasks that have one criticality level lower than the criticality level of the system. Consider a dual-core three-criticality system, based on the dual-core two-criticality system explored in Chapter 3, we may get the following scenarios:

- Both cores are in LO-crit mode, all of the tasks remain schedulable.
- Core c_1 enters MID-crit mode while core c_2 remains in LO-crit mode, migratable LO-crit tasks on core c_1 migrate to core c_2. All of the tasks remain schedulable.

- Core c_2 enters MID-crit mode while core c_1 remains in MID-crit mode, all LO-crit tasks on core c_2 are abandoned. All of the MID-crit and HI-crit tasks remain schedulable.
- Core c_1 enters HI-crit mode while core c_2 remains in MID-crit mode, migratable MID-crit tasks on core c_1 migrate to core c_2 while LO-crit tasks on core c_1 are abandoned. All of the MID-crit and HI-crit tasks remain schedulable.
- Core c_2 enters HI-crit mode while core c_1 remains in HI-crit mode, all MID-crit tasks on core c_2 are abandoned. All of the HI-crit tasks remain schedulable.

Based on the above scenarios, the criticality level to be assured of mainly depends on the highest criticality of the system, while the boundary number still plays an important role upon determining the criticality level to be guaranteed execution in the system. In detail, if the number of the cores in the current highest criticality level L is smaller than or equal to the boundary number, then all of the tasks with criticality levels larger than or equal to $L-1$ can be saved. Otherwise, only tasks with criticality levels larger than or equal to L can be guaranteed to be schedulable.

However, the above scenarios only consider the situation that the criticality level of the system increases gradually, while the model allows the criticality level to increase directly. Again take the dual-core three-criticality system as an example, core c_1 may mode change directly from LO-crit to HI-crit. In such a situation, core c_1 needs to guarantee the execution of HI-crit tasks, so that migratable MID-crit tasks need to migrate to core c_2 and all LO-crit tasks need to be abandoned. In addition, since core c_1 is in HI-crit mode, all of the MID-crit tasks, including the tasks migrating away, are executing with their MID-crit budgets. As only one core is in HI-crit mode, the system is required to guarantee the execution of all MID-crit tasks. Thus, in order to guarantee the execution of the MID-crit tasks on core c_2, c_2 is therefore forced to enter MID-crit mode and abandon all LO-crit tasks. In order to differentiate such forced mode change state from the original mode change state, we name such a mode as MID'-crit. According to that, in semi-partitioned multi-level MCS, criticality level increase on one core may lead to system mode changes on other cores.

In all, the general model for multi-level semi-partitioned model may be viewed as following:

- All cores execute in criticality level 0, and the global state is set to be 0.
- If one core enters level i and the global state is lower than i, then the global state is set to i. The tasks with criticality levels lower than $i-1$ on the core will be abandoned, a proportion of the tasks with criticality level of $i-1$ will stay on the core while others will migrate to other cores, the tasks with criticality levels higher than $i-1$ will stay on the core.
- If one core enters level i, the global state is i, and the number of the cores in level i is smaller than the boundary number, then the global state is unchanged, the tasks with criticality levels lower than $i-1$ on the core will be abandoned, the migratable tasks with criticality level of $i-1$ will migrate to available cores while the other tasks with criticality level of $i-1$ will stay executing on the core, and the tasks with criticality levels higher than $i-1$ will also stay executing on the core.
- If one core enters level i, the global state is i, and the number of the cores in level i is equal to or larger than the boundary number, then the global state is unchanged, the tasks with criticality levels lower than $i-1$ on the core will be abandoned, the migratable tasks with criticality level of $i-1$ will be abandoned while the other tasks with criticality level of $i-1$ will stay executing on the core, and the tasks with criticality levels higher than $i-1$ will also stay executing on the core.
- If one core enters level i and the global state is higher than i, then the global state is unchanged, and the tasks with criticality levels lower than i on the core will be abandoned, while the tasks with criticality levels equal to or higher than i will also stay executing on the core.

5.2 Considering System Architecture

The previous section has concluded the multi-criticality level effects on the semi-partitioned model. This section will focus on the effects from considering the architecture of the cores in the system. The models in previous chapters are built with two assumptions: the criticality mode of all of the cores can be acquired instantly at run time, and the migration cost is sufficient small that it can be ignored. However, with the increment of the number of the cores in the system, these two assumptions become unsustainable. Regarding the first assumption, there are two possible structures of the criticality mode information delivery system: a global one and a local one. For the global one, it is assumed that there exists a special controlling core c_c which has direct access to criticality mode status of each core while each core also has direct access to this core for the information. In this structure, the criticality modes of all of the cores are possible to be acquired instantly at run time. It is a typical distributed consensus problem that since all of the cores need access to the same controlling core, problems occur due to synchronised periods. For example, for a n core two criticality level system and the boundary number is n_b, based on the previous models, it can be indicated that if no more than n_b cores enter HI-crit mode then all of the tasks are schedulable. Assume that there are $n_b - 1$ cores currently in HI-crit mode in the system, and two cores c_j and c_k enter HI-crit mode at the same time, then one of the two cores, say c_j, will access the controlling core c_c first. The controlling core c_c will only notice that there are currently n_b cores in HI-crit mode without knowing that c_k is also in HI-crit mode due to the synchronise problem. Thus, it will suggest the migratable LO-crit tasks on core n_j to migrate rather than being abandoned. According to that, core n_k may receive some migrating tasks which may cause the scheduling problem on HI-crit tasks on the core. In all, the global structure may cause critical failures. For the local one, it is assumed that each core contains a counting procedure and when a core enters HI-crit mode, it will broadcast this information to all of the cores. For this structure, the increasing number of the cores in the system may cause a significant increase of the latency of the broadcasting information. This latency may cause a core to make a wrong estimation of the number of HI-crit cores currently in the system, which leads to a wrong decision on whether to allow migrating or abandoning. In summary, the first assumption seems no longer suitable for a system with a large number of cores.

Regarding to the second assumption, considering the system architecture and transferring bus issues, the migration costs from one core to other cores can be significantly different. For example, consider a $16 - core$ NoC-based system (Figure 5.1), the migration distance from core $c_{1,1}$ to core $c_{4,4}$ is much larger than that from core $c_{1,1}$ to core $c_{1,2}$. Since the transfer speed is typically the same in a NoC, the migration cost from core $c_{1,1}$ to core $c_{4,4}$ is much larger than that from core $c_{1,1}$ to core $c_{1,2}$. Based on this problem, it is proposed to extend the previous model with a consideration of the architecture of the system.

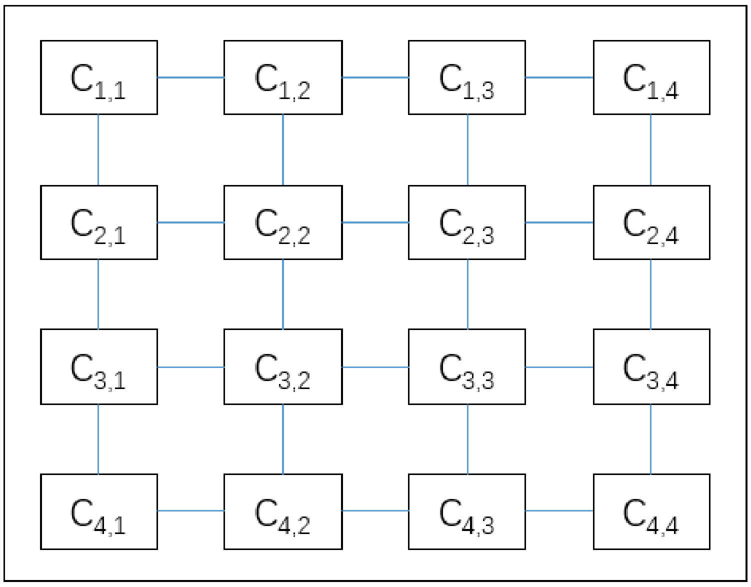


Figure 5.1: 16-core NOC

There exists many system architectures and many of them have quite different characteristics. This section will focus on the MCS on a regular X=Y NoC platform because the environment of most of the cores are similar in such a NoC, which will ease the scheduling problem. In addition, this section only considers MCS with two criticality levels, HI-crit and LO-crit. As it is mentioned before, the migration costs between cores mainly rely on the distances between cores in the NoC. It can be deduced that the minimum migration cost is the migration cost between neighbours and this cost between neighbours shall be all the same. Considering that, it is proposed to limit the migration options from all possible cores to all possible neighbour cores. In addition, based on the location of the core in the NoC, the boundary number of migration decision is variable. To be detailed, cores in a NoC are divided into three types, which are corner, edge and normal, according to their locations. As these names suggest, the corner type represents all of the cores in the four corners; the edge type stands for all of the cores in the four edges except the ones in corners; the normal type represents the rest of the cores. Take the above $16 - core$ NoC system (Figure 5.1) as example, the cores can be divided as:

- Corner: $c_{1,1}$, $c_{1,4}$, $c_{4,1}$, $c_{4,4}$
- Edge: $c_{1,2}$, $c_{1,3}$, $c_{2,1}$, $c_{2,4}$, $c_{3,1}$, $c_{3,4}$, $c_{4,2}$, $c_{4,3}$
- Normal: $c_{2,2}$, $c_{2,3}$, $c_{3,2}$, $c_{3,3}$

Regarding the corner type, according to the shape of the NoC, there are always four cores in this type no matter how many cores in the system. Although it seems that this type is a kind of minority case in the system, it is a critical type in the system since the core of this type only has two neighbours and both of the neighbours are in the edge type. According to that, cores in corner type have the least migration flexibility than all other cores in the system. Thus, it is proposed to give a low priority on allocating migratable tasks to these cores. In addition, the boundary number for this type of cores is 0. That is, if none of the neighbour cores are in HI-crit mode, then when a criticality mode change occurs to the core, the migratable tasks belong to the core may split and migrate to neighbour cores. Otherwise, all migratable tasks on the core need to be abandoned to guarantee the execution of the HI-crit tasks.

Regarding the edge type, cores have three neighbours. Two of the neighbours belong to the edge or corner type while the other is of the normal type. Comparing with the corner type, the cores in the edge type have slightly more migration options. Considering that, the boundary number for this type of cores is suggested to be 1. That is, if less than or equal to one of the neighbour cores is in HI-crit mode, then when a criticality mode change occurs to the core, the migratable tasks belong to the core may split and migrate to other neighbour cores still in the LO-crit mode. Otherwise, all migratable tasks on the core need to be abandoned to guarantee the execution of the HI-crit tasks.

Regarding the normal type, cores in this type have four neighbours. Consider the shape of the NoC, two of the neighbours are guaranteed to belong to the normal type, while the rest of the neighbours may have three different combinations: two normal types, one normal type and one edge type, two edge types. For the first two combinations, the increment of the number of neighbours improves the migration options. Based on that, the boundary number of these two cases is suggested to be 2. That is, if less than or equal to two of the neighbour cores are in HI-crit mode, then when a criticality mode change occurs to the core, the migratable tasks belonging to the core may split and migrate to other neighbour cores still in the LO-crit mode. Otherwise, all migratable tasks on the core need to be abandoned to guarantee the execution of the HI-crit tasks.

In all, the relationship between the boundary number and the type of the core can be seen in Table 5.1.

Type of the core	Number of the neighbours	Boundary number
Corner	2	0
Edge	3	1
Normal	4	2

Table 5.1: Relationship between Type and Boundary

5.3 Semi-partitioned Model on 16-core NoC Platform with 3-criticality Levels

The previous two sections have separately discussed the possible effects from increasing the number of the criticality levels in the system and the consideration of the system architecture, and provided the extended model correspondingly. This section will combine the findings above to construct an extended semi-partitioned model for a 16-core NoC-based mixed criticality system with 3-criticality levels.

5.3.1 Response Time Analysis

Since only local information can be acquired by each core, a general system model is not applicable for this semi-partitioned model. In addition, the location of the cores affects the boundary number so that cores contain different boundary numbers. Thus, unlike the analysis parts in previous chapters, the analysis of the cores are separated into three parts. Each part represents a type of core.

Cores of the Corner Type

Corner cores are paired with both of their neighbours. Thus, when a corner core needs to migrate tasks, the task load may split and migrate to two neighbour cores. In addition, according to the analysis in Section 5.2, the boundary number for the corner core is 0. Assume that a set of tasks S containing three criticality levels is scheduled on the corner core c_c and its two neighbour cores are core c_1 and c_2, then on each core there exist five types of tasks: HI-crit tasks, statically allocated MID-crit tasks, migratable MID-crit tasks, statically allocated LO-crit tasks and migratable LO-crit tasks.

Let HI_i represent the set of HI-crit tasks on core c_i; MID_i represent the set of statically allocated MID-crit tasks; LO_i represent the set of statically allocated

LO-crit tasks; $MIGM_{i,j}$ represent the set of migratable MID-crit tasks which will migrate to core c_j if core c_j is not in HI-crit mode; $MIGL_{i,j}$ represent the set of migratable LO-crit tasks which will migrate to core c_j if core c_j is in LO-crit mode; $MIGM_{i,j,k}$ represent the set of migratable MID-crit tasks which will migrate to core c_j if core c_j is not in HI-crit mode, and migrate to core c_k if core c_j is in HI-crit mode but core c_k is not in HI-crit mode; $MIGL_{i,j,k}$ represent the set of migratable LO-crit tasks which will migrate to core c_j if core c_j is in LO-crit mode, and migrate to core c_k if core c_j is not in LO-crit mode but core c_k is in LO-crit mode.

Since the scheduling test in this part focuses on the corner core c_c, the migrating tasks from the neighbour cores to other cores are not considered. In addition, the corner core c_c cannot differentiate whether the neighbour cores c_1 and c_2 have accepted any migrating LO-crit tasks from other cores or not. But having these tasks in neighbour cores does not affect the schedulability test on the corner core. Thus, in order to simplify the analysis, it is safe to assume that there is no migrating LO-crit tasks from other cores on two neighbour cores. Then the following relationship can be obtained, where symbol $*$ represents all other cores except the corner core c_c and neighbour core c_1 and c_2:

- $S = LO_c \cup MID_c \cup HI_c \cup MIGL_{c,1} \cup MIGL_{c,2} \cup MIGM_{c,1} \cup MIGM_{c,2}$ $\cup LO_1$ $\cup MID_1$ $\cup HI_1$ $\cup MIGL_{1,c,*}$ $\cup MIGL_{1,*,c}$ $\cup MIGL_{1,*,*}$ $\cup MIGM_{1,c,*}$ $\cup MIGM_{1,*,c}$ $\cup MIGM_{1,*,*}$ $\cup LO_2$ $\cup MID_2$ $\cup HI_2$ $\cup MIGL_{2,c,*}$ $\cup MIGL_{2,*,c}$ $\cup MIGL_{2,*,*}$ $\cup MIGM_{2,c,*}$ $\cup MIGM_{2,*,c}$ $\cup MIGM_{2,*,*}$

In the steady state mode, all these tasks are statically partitioned on each core and executing with their LO-crit budgets. Define state X to represent this phase, then the relationship between tasks and cores can be viewed as:

- $X_c = LO_c \cup MID_c \cup HI_c \cup MIGL_{c,1} \cup MIGL_{c,2} \cup MIGM_{c,1} \cup MIGM_{c,2}$

- $X_1 = LO_1 \cup MID_1 \cup HI_1 \cup MIGL_{1,c,*} \cup MIGL_{1,*,c} \cup MIGL_{1,*,*} \cup MIGM_{1,c,*} \cup MIGM_{1,*,c} \cup MIGM_{1,*,*}$

- $X_2 = LO_2 \cup MID_2 \cup HI_2 \cup MIGL_{2,c,*} \cup MIGL_{2,*,c} \cup MIGL_{2,*,*} \cup MIGM_{2,c,*} \cup MIGM_{2,*,c} \cup MIGM_{2,*,*}$

- $S = X_c \cup X_1 \cup X_2$

Regarding state X, all tasks are executing with their LO-crit budgets. In this case, the response time analysis of all the tasks is given by equation (5.1):

$$\forall \tau_i \in X : R_i = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(LO) \tag{5.1}$$

If core c_c enters MID-crit mode while core c_1 and c_2 remain in LO-crit mode, then the migratable LO-crit tasks on core c_c will split and migrate to cores c_1 and c_2. All tasks remain schedulable in this scenario. The relationship between tasks and cores can be viewed as below, where Y(c,1,2H,1H) represents the state formed by the progress that core c enters MID-crit mode, then core c_1 enters MID-crit mode, then core c_2 enters HI-crit mode and core c_1 enters HI-crit mode as the final mode change:

- $Y(c)_c = LO_c \cup MID_c \cup HI_c \cup MIGM_{c,1} \cup MIGM_{c,2}$

- $Y(c)_1 = X_1 \cup MIGL_{c,1}$

- $Y(c)_2 = X_2 \cup MIGL_{c,2}$

- $S = Y(c)_c \cup Y(c)_1 \cup Y(c)_2$

In this state, all of the MID-crit tasks remaining on core c_c are executing with MID-crit budgets while all of the other tasks are executing with their LO-crit budgets. The migratable LO-crit tasks are executing with their LO-crit budgets on the new core and suffering from reduced deadline and release jitter issues influence. Thus, the response time analysis of cores in this state is given by equation (5.2), where $(\delta L_i, MID)$ is a function that if $L_i > MID$ it returns LO; if $L_i = MID$ it returns MID; if $L_i < MID$ it returns L_i: (on the next page)

$$\begin{aligned}
&\forall \tau_i \in MIGL_{c,1} \cup MIGL_{c,2} : \\
&\qquad D'_i = D_i - (R_i - C_i(LO)) \\
&\qquad J_i = R_i - C_i(LO) \\
&\forall \tau_i \in Y(c)_c : \\
&\qquad R_i(MID) = C_i(\delta L_i, MID) + \sum_{\tau_j \in chpL(i)} \left\lceil \frac{R_i(MID)}{T_j} \right\rceil C_j(LO) \\
&\qquad + \sum_{\tau_k \in chpM(i)} \left\lceil \frac{R_i(MID)}{T_k} \right\rceil C_k(MID) + \sum_{\tau_l \in chpH(i)} \left\lceil \frac{R_i(MID)}{T_l} \right\rceil C_l(LO) \\
&\qquad + \sum_{\tau_j \in chpM(i)} \left\lceil \frac{R_i(MID)}{T_j} \right\rceil C_j(MID) + \sum_{\tau_m \in chpMIGL(i)} \left\lceil \frac{R_i}{T_m} \right\rceil C_m(LO) \\
&\forall \tau_i \in Y(c)_1 \cup Y(c)_2 : \\
&\qquad R_i(LO)' = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(LO)' + J_j}{T_j} \right\rceil C_j(LO)
\end{aligned} \tag{5.2}$$

If core c_c enters MID-crit mode while one or two neighbour cores are in MID'-crit mode, then all of the migratable LO-crit tasks on core c_c need to be abandoned. Considering the worst case that both core c_1 and core c_2 are in MID-crit mode (the actual order of which core enters MID-crit does not matter, say core c_1 first), and assume that all of the LO-crit migratable tasks on these cores that can be migrate to core c_c have been migrated, define state $Y(1,2)$ to represent this phase and relationship between tasks and cores can be represented by:

- $Y(1,2)_c = X_c \cup MIGL_{1,c,*} \cup MIGL_{1,*,c} \cup MIGL_{2,c,*} \cup MIGL_{2,*,c}$
- $Y(1,2)_1 = LO_1 \cup MID_1 \cup HI_1 \cup MIGM_{1,c,*} \cup MIGM_{1,*,c} \cup MIGM_{1,*,*}$
- $Y(1,2)_2 = LO_2 \cup MID_2 \cup HI_2 \cup MIGM_{2,c,*} \cup MIGM_{2,*,c} \cup MIGM_{2,*,*}$
- $S = Y(c)_c \cup Y(c)_1 \cup Y(c)_2 \cup MIGL_{1,*,*} \cup MIG_{2,*,*}$

Accordingly, we can get the new response time for the tasks on the core c_c as equation (5.3):

$$\forall \tau_i \in Y(1,2)_c :\quad R_i(LO)' = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(LO)' + J_j}{T_j} \right\rceil C_j(LO) \tag{5.3}$$

In this case, if core c_c enters MID-crit tasks, then all of the LO-crit tasks on the core need to be abandoned. In addition, it is the worst case for c_c entering MID-crit mode.

- $Y(1,2,c)_c = MID_c \cup HI_c \cup MIGM_{c,1} \cup MIGM_{c,2}$
- $LO_c \cup MIGL_{c,1} \cup MIGL_{c,2} \cup MIGL_{1,c,*} \cup MIGL_{1,*,c} \cup MIGL_{2,c,*} \cup MIGL_{2,*,c}$ are abandoned

Regarding state $Y(1,2,c)_c$, MID-crit tasks will execute with their MID-crit budgets and HI-crit tasks will execute with their LO-crit budgets. Thus, the response time analysis of core c_c can be seen as equation (5.4):

$$\begin{aligned} &\forall \tau_i \in Y(1,2,c)_c : \\ &R_i(MID)' = C_i(\delta L_i, MID) + \sum_{\tau_j \in chpM(i)} \left\lceil \frac{R_i(MIG)'}{T_j} \right\rceil C_j(MID) \\ &+ \sum_{\tau_k \in chpH(i)} \left\lceil \frac{R_i(MIG)'}{T_k} \right\rceil C_k(LO) + \sum_{\tau_l \in chpL(i)} \left\lceil \frac{R_i(LO)'}{T_l} \right\rceil C_l(LO) \end{aligned} \tag{5.4}$$

If core c_c enters HI-crit mode from state $Y(1,2,c)$, then the migratable MID-crit tasks will split and migrate to core c_1 and c_2 and all of the MID-crit tasks are guaranteed to be schedulable. The relationship between cores can be represented by:

- $Y(1,2,c,cH)_c = MID_c \cup HI_c$
- $Y(1,2,c,cH)_1 = Y(1,2,c)_1 \cup MIGM_{c,1}$
- $Y(1,2,c,cH)_2 = Y(1,2,c)_2 \cup MIGM_{c,2}$

In this state, all of the MID-crit tasks remaining on core c_c are executing with MID-crit budgets while all of the HI-crit tasks are executing with their HI-crit budgets. The migratable MID-crit tasks are executing with MID-crit budgets at the new core and suffering from reduced deadline and release jitter issues influence. Thus, the response time analysis of cores in this state is given by equation (5.5):

$$
\begin{aligned}
&\forall \tau_i \in MIGM_{c,1} \cup MIGM_{c,2} :\\
&\qquad D_i' = D_i - (R_i(MID)' - C_i(MID))\\
&\qquad J_i = R_i(MID)' - C_i(MID)\\
&\forall \tau_i \in Y(1,2,c,cH)_c :\\
&\qquad R_i(HI) = C_i(\delta L_i, HI) + \sum_{\tau_j \in chpM(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(MID)\\
-&\qquad + \sum_{\tau_k \in chpH(i)} \left\lceil \frac{R_i(HI)}{T_k} \right\rceil C_k(HI) + \sum_{\tau_l \in chpMIGM(i)} \left\lceil \frac{R_i(MID)'}{T_l} \right\rceil C_l(MID)\\
&\forall \tau_i \in Y(1,2,c,cH)_1 \cup Y(1,2,c,cH)_2 :\\
&\qquad R_i(MID)'' = C_i(\delta L_i, MID) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(MID'' + J_j}{T_j} \right\rceil C_j(\delta L_i, MID)
\end{aligned}
\tag{5.5}
$$

Considering another scenario that core c_1 and c_2 enter HI-crit first, and all of the MID-crit migratable tasks on these cores that can migrate to core c_c have been migrated, then the core is forced to mode change to MID-crit' mode and abandon all of the LO-crit tasks on core. The tasks on core c_c can be represented by:

- $Y(1H,2H)_c = MID_c \cup HI_c \cup MIGM_{c,1} \cup MIGM_{c,2} \cup MIGM_{1,c,*} \cup MIGM_{1,*,c} \cup MIGM_{2,c,*} \cup MIGM_{2,*,c}$

Accordingly, we can get the new response time for the tasks on the core c_c as equation (5.6):

$$\forall \tau_i \in Y(1H,2H)_c: \quad R_i(MID)'' = C_i(\delta L_i, MID) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(MID'' + J_j}{T_j} \right\rceil C_j(\delta L_i, MID) \tag{5.6}$$

In this case, if core c_c enters HI-crit tasks, then all of the MID-crit tasks on the core need to be abandoned. In addition, it is the worst case for c_c entering HI-crit mode.

- $Y(1H, 2H, cH)_c = HI_c$
- $MID_c \cup MIGM_{c,1} \cup MIGM_{c,2} \cup MIGM_{1,c,*} \cup MIGM_{1,*,c} \cup MIGM_{2,c,*} \cup MIGM_{2,*,c}$ are abandoned

Regarding state $Y(1H, 2H, cH)_c$, only HI-crit tasks execute with their LO-crit budgets. Thus, the response time analysis of core c_c can be seen as equation (5.7):

$$\begin{aligned} \forall \tau_i \in & Y(1H,2H,cH)_c: \\ & R_i(HI)' = C_i(\delta L_i, HI) + \sum_{\tau_j \in chpH(i)} \left\lceil \frac{R_i(HI)'}{T_j} \right\rceil C_j(HI) \\ & + \sum_{\tau_k \in chpM(i)} \left\lceil \frac{R_i(MID)''}{T_k} \right\rceil C_l(MID) \end{aligned} \tag{5.7}$$

For completion, the scenario that core c_c increases directly from LO-crit mode to HI-crit mode needs to be considered. This scenario only happens when core c_1 and c_2 are in LO-crit or LO'-crit or MID-crit or MID'-crit mode (if c_1 is already in HI-crit mode then core c_c is forced to mode change to MID'-crit mode). In any of the above situations, the migratable MID-crit tasks will migrate to core c_1 and c_2. Considering that, the worst case happens when core c_1 and core c_2 are both in MID-crit mode and all of the LO-crit migratable tasks on these cores that can migrate to core c_c have been migrated, then the tasks on core c_c can be represented as:

- $Y(1,2,cH)_c = MID_c \cup HI_c$
- $Y(1,2,cH)_1 = Y(1,2)_1 \cup MIGM_{c,1}$
- $Y(1,2,cH)_2 = Y(1,2)_2 \cup MIGM_{c,2}$
- $LO_c \cup MIGL_{c,1} \cup MIGL_{c,2} \cup MIGL_{1,c,*} \cup MIGL_{1,*,c} \cup MIGL_{2,c,*} \cup MIGL_{2,*,c}$ are abandoned

Regarding this state, the migratable MID-crit tasks perform unusually as they are executing with MID-crit budgets. The reduced deadline and release jitter issues are calculated using the LO-crit response time and LO-crit budgets. In addition, the interference from these migrating tasks is also calculated using their LO-crit budgets. Thus, the response time analysis of the cores can be viewed as equation (5.8):

$$\begin{aligned}
&\forall \tau_i \in MIGM_{c,1} \cup MIGM_{c,2} :\\
&\qquad D_i'' = D_i - (R_i(LO)' - C_i(LO))\\
&\qquad J_i' = R_i(LO)' - C_i(LO)\\
&\forall \tau_i \in Y(1,2,cH)_c :\\
&\qquad R_i(HI)'' = C_i(\delta L_i, HI) + \sum_{\tau_j \in chpM(i)} \left\lceil \frac{R_i(HI)''}{T_j} \right\rceil C_j(MID)\\
&\qquad + \sum_{\tau_k \in chpH(i)} \left\lceil \frac{R_i(HI)''}{T_k} \right\rceil C_k(HI) + \sum_{\tau_l \in chpMIGM(i)} \left\lceil \frac{R_i(LO)'}{T_l} \right\rceil C_l(LO)\\
&\qquad + \sum_{\tau_m \in chpL(i)} \left\lceil \frac{R_i(LO)'}{T_m} \right\rceil C_m(LO)\\
&\forall \tau_i \in Y(1,2,cH)_1 \cup Y(1,2,cH)_2 :\\
&\qquad R_i(MID)'' = C_i(\delta L_i, MID) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(MID'' + J_j}{T_j} \right\rceil C_j(\delta L_i, MID)
\end{aligned} \tag{5.8}$$

If all of the above schedulability tests have been passed, then the tasks on the corner core are deemed to be scheduled.

Cores of the Edge Type

For the cores in the edges, they have three neighbours and hence are paired with three cores. Thus, when the edge core needs to migrate tasks, the task load may split and migrate to the neighbour cores. In addition, according to the analysis in the previous section, the boundary number for the edge core is 1. Assume that a set of tasks S containing three criticality levels is scheduled on the corner core c_c and its three neighbour cores c_1, c_2 and c_3, then on each core there exist five types of tasks: HI-crit tasks, statically allocated MID-crit tasks, migratable MID-crit tasks, statically allocated LO-crit tasks and migratable LO-crit tasks. Since the scheduling in this part focuses on the edge core, the migrating tasks from the neighbour cores to other cores are not considered. In addition, the edge core c_e cannot differentiate whether the neighbour core c_1, c_2 and c_3 have accepted any migrating LO-crit tasks or not. But having these tasks in neighbour cores does not affect the schedulability test on the focused core. Thus, in order to simplify the analysis, it is safe to assume that there is no migrating LO-crit task from other cores on two neighbour cores. Then the following relationship can be obtained:

- $S = LO_e \cup MID_e \cup HI_e \cup MIGL_{e,*,*} \cup MIGM_{c,*,*} \cup LO_1 \cup MID_1 \cup HI_1 \cup MIGL_{1,c,*} \cup MIGL_{1,*,c} \cup MIGL_{1,*,*} \cup MIGH_{1,c,*} \cup MIGH_{1,*,c} \cup MIGH_{1,*,*} \cup LO_2 \cup MID_2 \cup HI_2 \cup MIGL_{2,c,*} \cup MIGL_{2,*,c} \cup MIGL_{2,*,*} \cup MIGH_{2,c,*} \cup MIGH_{2,*,c} \cup MIGH_{2,*,*} \cup LO_3 \cup MID_3 \cup HI_3 \cup MIGL_{3,c,*} \cup MIGL_{3,*,c} \cup MIGL_{3,*,*} \cup MIGH_{3,c,*} \cup MIGH_{3,*,c} \cup MIGH_{3,*,*}$

In the steady state mode, all these tasks are statically partitioned on each core and executing with their LO-crit budgets. Define state X to represent this phase, then the relationship between tasks and cores can be represented by:

- $X_e = LO_e \cup MID_e \cup HI_e \cup MIGL_{e,*,*} \cup MIGM_{e,*,*}$
- $X_1 = LO_1 \cup MID_1 \cup HI_1 \cup MIGL_{1,e,*} \cup MIGL_{1,*,e} \cup MIGL_{1,*,*} \cup MIGH_{1,e,*} \cup MIGH_{1,*,e} \cup MIGH_{1,*,*}$

- $X_2 = LO_2 \cup MID_2 \cup HI_2 \cup MIGL_{2,e,*} \cup MIGL_{2,*,e} \cup MIGL_{2,*,*} \cup MIGH_{2,e,*} \cup MIGH_{2,*,e} \cup MIGH_{2,*,*}$
- $X_2 = LO_3 \cup MID_3 \cup HI_3 \cup MIGL_{3,e,*} \cup MIGL_{3,*,e} \cup MIGL_{3,*,*} \cup MIGH_{3,e,*} \cup MIGH_{3,*,e} \cup MIGH_{3,*,*}$
- $S = X_c \cup X_1 \cup X_2 \cup X_3$

Regarding state X, all tasks are executing with their LO-crit budgets. In this case, the response time analysis of all the tasks is given by equation (5.9):

$$\forall \tau_i \in X : R_i = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(LO) \tag{5.9}$$

Since the boundary number is 1 for the edge cores, if core c_e enters MID-crit mode while another core is already in MID-crit mode, the LO-crit tasks on core c_e are still migratable. Thus, if core c_3 enters MID-crit mode and all of the LO-crit migratable tasks on these cores that can migrate to core c_e have been migrated, then the tasks on core c_e can be represented by:

- $Y(3)_e = X_(e) \cup MIGL_{3,e,*} \cup MIGL_{3,*,e}$

Accordingly, we can get the new response time for the tasks on the core c_e as equation (5.10):

$$\begin{aligned} \forall \tau_i \in & Y(3)_e : \\ & R_i(LO)' = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(LO)' + J_j}{T_j} \right\rceil C_j(LO) \end{aligned} \tag{5.10}$$

If core c_c then enters MID-crit mode while core c_2 and c_3 remaining in LO-crit mode, then the migratable LO-crit tasks on core c_c will split and migrate to core c_1 and c_2. In addition, the migrated tasks accepted by core c_e from core c_3 are abandoned. The relationship between tasks and cores can be represented by:

- $Y(3,e)_c = LO_c \cup MID_c \cup HI_c \cup MIGM_{e,*,*}$

- $Y(3,e)_1 = Y(3)_1 \cup MIGL_{e,1,*} \cup MIGL_{e,3,1}$
- $Y(3,e)_2 = Y(3)_2 \cup MIGL_{e,2,*} \cup MIGL_{e,3,2}$
- $Y(3,e)_3 = Y(3)_3$
- $MIGL_{3,e,*} \cup MIGL_{3,*,e}$ are abandoned

In this state, all of the MID-crit tasks remain on core c_e are executing with MID-crit budgets while all of the other tasks are executing with their LO-crit budgets. The migratable LO-crit tasks are executing with LO-crit budgets at the new core and suffering from reduced deadline and release jitter issues influence. Thus, the response time analysis of cores in this state is given by equation (5.11).

$$\begin{aligned}
&\forall \tau_i \in \cup MIGL_{e,1,*} \cup MIGL_{e,3,1} \cup MIGL_{e,2,*} \cup MIGL_{e,3,2} :\\
&\qquad D_i' = D_i - (R_i(LO)' - C_i(LO))\\
&\qquad J_i = R_i(LO)' - C_i(LO)\\
&\forall \tau_i \in Y(3,e)_e :\\
&\qquad R_i(MID) = C_i(\delta L_i, MID) + \sum_{\tau_j \in chpL(i)} \left\lceil \frac{R_i(MID)}{T_j} \right\rceil C_j(LO)\\
&\qquad + \sum_{\tau_k \in chpM(i)} \left\lceil \frac{R_i(MID)}{T_k} \right\rceil C_k(MID) + \sum_{\tau_l \in chpH(i)} \left\lceil \frac{R_i(MID)}{T_l} \right\rceil C_l(LO)\\
&\qquad + \sum_{\tau_j \in chpM(i)} \left\lceil \frac{R_i(MID)}{T_j} \right\rceil C_j(MID) + \sum_{\tau_m \in chpMIGL(i)} \left\lceil \frac{R_i(LO)'}{T_m} \right\rceil C_m(LO)\\
&\forall \tau_i \in Y(3,e)_1 \cup Y(3,e)_2 :\\
&\qquad R_i(LO)' = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(LO)' + J_j}{T_j} \right\rceil C_j(LO)
\end{aligned} \tag{5.11}$$

If core c_e enters MID-crit mode while one or two or three neighbour cores are in MID'-crit mode, then all of the migratable LO-crit tasks on core c_e need to be abandoned. The schedulability test for core c_e in this scenario is covered by the above scenario.

If all of the neighbour cores are in MID-crit mode, and assume that all of the LO-crit migratable tasks on these cores that can migrate to core c_e have been migrated, then the tasks on core c_e can be represented by:

- $Y(1,2,3)_e = X_(e) \cup MIGL_{1,e,*} \cup MIGL_{1,*,e} \cup MIGL_{2,e,*} \cup MIGL_{2,*,e} \cup MIGL_{3,e,*} \cup MIGL_{3,*,e}$

According to that, we can get the new response time for the tasks on the core c_c as equation (5.12):

$$\begin{aligned} &\forall \tau_i \in Y(1,2,3)_e : \\ &\quad R_i(LO)'' = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(LO)' + J_j}{T_j} \right\rceil C_j(LO) \end{aligned} \tag{5.12}$$

In this case, if core c_e enters MID-crit tasks, then all of the LO-crit tasks on the core need to be abandoned. In addition, it is the worst case for c_e entering MID-crit mode.

- $Y(1,2,3,e)_e = MID_e \cup HI_e \cup MIGM_{e,*,*}$
- $LO_c \cup MIGL_{e,*,*} \cup MIGL_{1,e,*} \cup MIGL_{1,*,e} \cup MIGL_{2,e,*} \cup MIGL_{2,*,e} \cup MIGL_{3,e,*} \cup MIGL_{3,*,e}$ are abandoned

Regarding state $Y(1,2,3,e)_e$, MID-crit tasks will execute with their MID-crit budgets and HI-crit tasks will execute with their LO-crit budgets. Thus, the response time analysis of core c_e can be seen as equation (5.13):

$$\begin{aligned} &\forall \tau_i \in Y(1,2,3,e)_e : \\ &\quad R_i(MID)' = C_i(\delta L_i, MID) + \sum_{\tau_j \in chpM(i)} \left\lceil \frac{R_i(MIG)'}{T_j} \right\rceil C_j(MID) \\ &\quad + \sum_{\tau_k \in chpH(i)} \left\lceil \frac{R_i(MIG)'}{T_k} \right\rceil C_k(LO) + \sum_{\tau_l \in chpL(i)} \left\lceil \frac{R_i(LO)''}{T_l} \right\rceil C_l(LO) \end{aligned} \tag{5.13}$$

Still since the boundary number is 1, if core c_e enters HI-crit mode while another core is already in HI-crit mode, the MID-crit tasks on core c_e are still migratable. Thus, if core c_3 enters HI-crit core from state $Y(1,2,3,e)$ and all of the MID-crit migratable tasks on these cores that can migrate to core c_e have been migrated, then the tasks on core c_e can be represented by:

- $Y(1,2,3,e,3H)_e = MID_e \cup HI_e \cup MIGM_{e,*,*} \cup MIGM_{3,e,*} \cup MIGM_{3,*,e}$

Accordingly, we can get the new response time for the tasks on the core c_c as equation (5.14):

$$\forall \tau_i \in Y(1,2,3,e,3H)_e : \\ R_i(MID)'' = C_i(\delta L_i, MID) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(MID'' + J_j}{T_j} \right\rceil C_j(\delta L_i, MID) \tag{5.14}$$

If core c_e enters HI-crit mode from state $Y(1,2,3,e,3H)$, then the migratable MID-crit tasks will split and migrate to core c_1 and c_2. In addition, the accepted MID-crit migrated tasks on core c_e need to be abandoned. The relationship between cores can be represented by:

- $Y(1,2,e,3H,eH)_c = MID_e \cup HI_e$
- $Y(1,2,e,3H,eH)_1 = Y(1,2,e,3H)_1 \cup MIGM_{e,1,*} \cup MIGM_{e,3,1}$
- $Y(1,2,e,3H,eH)_2 = Y(1,2,e,3H)_2 \cup MIGM_{e,2,*} \cup MIGM_{e,3,2}$
- $Y(1,2,e,3H,eH)_3 = Y(1,2,e,3H)_3$
- $MIGM_{3,e,*} \cup MIGM_{3,*,e}$ are abandoned

In this state, all of the MID-crit tasks remaining on core c_e are executing with MID-crit budgets while all of the HI-crit tasks are executing with their HI-crit budgets. The migratable MID-crit tasks are executing with MID-crit budgets at the new core and suffering from reduced deadline and release jitter issues influence. Thus, the response time analysis of cores in this state is given by equation (5.15).

$$
\begin{aligned}
&\forall \tau_i \in MIGM_{e,1,*} \cup MIGM_{e,3,1} \cup MIGM_{e,2,*} \cup MIGM_{e,3,2} : \\
&\qquad D_i' = D_i - (R_i(MID)' - C_i(MID)) \\
&\qquad J_i = R_i(MID)' - C_i(MID) \\
&\forall \tau_i \in Y(1,2,3,e,3H,eH)_e : \\
&\qquad R_i(HI) = C_i(\delta L_i, HI) + \sum_{\tau_j \in chpM(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(MID) \\
&\qquad + \sum_{\tau_k \in chpH(i)} \left\lceil \frac{R_i(HI)}{T_k} \right\rceil C_k(HI) + \sum_{\tau_l \in chpMIGM(i)} \left\lceil \frac{R_i(MID)''}{T_l} \right\rceil C_l(MID) \\
&\forall \tau_i \in Y(1,2,c,cH)_1 \cup Y(1,2,c,cH)_2 : \\
&\qquad R_i(MID)'' = C_i(\delta L_i, MID) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(MID'' + J_j}{T_j} \right\rceil C_j(\delta L_i, MID)
\end{aligned}
\tag{5.15}
$$

Considering another scenario that core c_1 , c_2 and c_3 enter HI-crit first, and all of the MID-crit migratable tasks on these cores that can migrate to core c_e have been migrated, then the core is forced to mode change to MID'-crit and abandon all of the LO-crit tasks on core. The tasks on core c_e can be represented as:

- $Y(1H,2H,3H)_e = MID_e \cup HI_e \cup MIGM_{e,*,*} \cup MIGM_{1,e,*} \cup MIGM_{1,*,e} \cup MIGM_{2,e,*} \cup MIGM_{2,*,e} \cup MIGM_{3,e,*} \cup MIGM_{3,*,e}$

Accordingly, we can get the new response time for the tasks on the core c_c as equation (5.16):

$$\begin{aligned}\forall \tau_i \in & Y(1H,2H,3H)_e : \\ & R_i(MID)''' = C_i(\delta L_i, MID) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(MID''' + J_j}{T_j} \right\rceil C_j(\delta L_i, MID)\end{aligned} \tag{5.16}$$

In this case, if core c_e enters HI-crit tasks, then all of the MID-crit tasks on the core need to be abandoned. In addition, it is the worst case for c_e entering HI-crit mode.

- $Y(1H,2H,3H,eH)_c = HI_e$
- $MID_e \cup MIGM_{e,*,*} \cup MIGM_{1,e,*} \cup MIGM_{1,*,e} \cup MIGM_{2,e,*} \cup MIGM_{2,*,e} \cup MIGM_{3,e,*} \cup MIGM_{3,*,e}$ are abandoned

Regarding state $Y(1H,2H,3H,eH)_e$, only HI-crit tasks execute with their LO-crit budgets. Thus, the response time analysis of core c_e is given by equation (5.17):

$$\begin{aligned}\forall \tau_i \in & Y(1H,2H,3H,eH)_e : \\ & R_i(HI)' = C_i(\delta L_i, HI) + \sum_{\tau_j \in chpH(i)} \left\lceil \frac{R_i(HI)'}{T_j} \right\rceil C_j(HI) \\ + & \sum_{\tau_k \in chpM(i)} \left\lceil \frac{R_i(MID)'''}{T_k} \right\rceil C_l(MID)\end{aligned} \tag{5.17}$$

For completion, the scenario that core c_e increases directly from LO-crit mode to HI-crit mode needs to be considered. This scenario only happens when core c_1, c_2 and c_3 are in LO-crit or LO'-crit or MID-crit or MID'-crit mode (if c_1 is already in HI-crit mode then core c_e is forced to mode change to MID'-crit mode). In any of the above situations, the migratable MID-crit tasks will migrate to core c_1, c_2 and c_3. Considering that, the worst case happens when core c_1, c_2 and c_3 are all in MID-crit mode and all of the LO-crit migratable tasks on these cores that can be migrate to core c_e have been migrated, then the tasks on core c_e can be represented as:

- $Y(1,2,3,eH)_c = MID_e \cup HI_e$
- $Y(1,2,3,eH)_1 = Y(1,2,3)_1 \cup MIGM_{e,1,*}$
- $Y(1,2,3,eH)_2 = Y(1,2,3)_2 \cup MIGM_{e,2,*}$
- $Y(1,2,3,eH)_3 = Y(1,2,3)_3 \cup MIGM_{e,3,*}$
- $LO_e \cup MIGL_{e,*,*} \cup MIGL_{1,e,*} \cup MIGL_{1,*,e} \cup MIGL_{2,e,*} \cup MIGL_{2,*,e} \cup MIGL_{3,e,*} \cup MIGL_{3,*,e}$ are abandoned

Regarding this state, the migratable MID-crit tasks perform quite unusually as they are executing with MID-crit budgets, the reduced deadline and release jitter issues are calculated using the LO-crit response time and LO-crit budgets. In addition, the interference from these migrating tasks is also calculated using their LO-crit budgets. Thus, the response time analysis of the cores can be viewed as equation (5.18):

$$
\begin{aligned}
&\forall \tau_i \in MIGM_{e,*,*}:\\
&\quad D_i'' = D_i - (R_i(LO)' - C_i(LO))\\
&\quad J_i' = R_i(LO)' - C_i(LO)\\
&\forall \tau_i \in Y(1,2,3,eH)_e:\\
&\quad R_i(HI)'' = C_i(\delta L_i, HI) + \sum_{\tau_j \in chpM(i)} \left\lceil \frac{R_i(HI)''}{T_j} \right\rceil C_j(MID)\\
&\quad + \sum_{\tau_k \in chpH(i)} \left\lceil \frac{R_i(HI)''}{T_k} \right\rceil C_k(HI) + \sum_{\tau_l \in chpMIGM(i)} \left\lceil \frac{R_i(LO)'}{T_l} \right\rceil C_l(LO)\\
&\quad + \sum_{\tau_m \in chpL(i)} \left\lceil \frac{R_i(LO)'}{T_m} \right\rceil C_m(LO)\\
&\forall \tau_i \in Y(1,2,3,eH)_1 \cup Y(1,2,3,eH)_2 \cup Y(1,2,3,eH)_3:\\
&\quad R_i(MID)'' = C_i(\delta L_i, MID) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(MID'' + J_j}{T_j} \right\rceil C_j(\delta L_i, MID)
\end{aligned}
\tag{5.18}
$$

If all of the above schedulability tests have been passed, then the tasks on the edge core are deemed to be scheduled.

Cores in the Normal Type

The Schedulability test of the cores in normal type is mostly similar to that of the cores in edge type. The only difference is that the boundary number is 2 for the cores in normal type. According to that, the response time analysis will not be repeated here. The detailed analysis can be seen in the Appendix.

5.3.2 Task Allocation

Unlike the previous models, although cores in a NoC have the same computing capability, they perform differently due to their locations in the semi-partitioned algorithm. According to that, when allocating tasks, the difference among cores needs to be considered. Based on the number of neighbours and the boundary number settings, cores of the corner type have the least flexibility while cores of the normal type have the most. As tasks are sorted by decreasing utilization criticality aware order, the task allocation progress will be divided into three steps.

The first step is to allocate the HI-crit tasks. For HI-crit tasks, since these tasks will not migrate, they are most suitable to be allocated to cores in the corner type. Thus, when allocating the HI-crit tasks, corner type cores have the highest priority, edge the next, while normal type cores have the lowest priority.

The next step is to allocate the MID-crit tasks. Although MID-crit tasks can be set migratable, the migrating progress will add reduced deadline and release jitter issues which adds extra scheduling burden. Thus, the MID-crit tasks are initially allocated as non-migratable tasks to the cores, unless the non-migration algorithm cannot schedule the tasks. In addition, due to the algorithm used to determine the migratable tasks, the initially non-migratable tasks may change to be migratable. Based on the consideration above, when allocating the MID-crit tasks, normal type cores have the highest priority as they provide the best migrating flexibility, edge the next, and corner type cores have the lowest priority.

The last step is to allocate the LO-crit tasks. Since the LO-crit tasks can also be set migratable, with the same reason as that for MID-crit tasks, normal type cores have the highest priority while corner type cores have the lowest priority.

5.3.3 Evaluation

The previous sections have derived the sufficient response time analysis and introduced an appropriate task allocation approach. In this section, we will introduce an experiment to compare the scheduling efficiency of the semi-partitioned algorithm against the non-migration algorithm.

Experiment Configuration

Software is developed to explore the efficiency of the three models. It is produced to compare the performance of the semi-partitioned algorithm and non-migration algorithm. The configuration of the software is fairly similar to that mentioned in Section 3.2.1. The software consists of three parts. The first part generates tasksets and stores them in XML files. Each tasksets node contains 10000 tasksets. In order to gain uniform distributed parameters, UUnifast-discard algorithm is used to generate nominal utilizations and Log-uniform algorithm [51] is used to generate periods. Other parameters of each task are calculated based on these two values. The second part of the software pre-sorts each taskset in criticality-aware utilization descending order. In such order, HI-crit tasks will be placed in front of all MID-crit tasks while MID-crit tasks will be placed in front of all LO-crit tasks, and each criticality level tasks are in utilization descending order independently. Then the software allocates the tasks based on the method introduced above, and compares the scheduling efficiency between the semi-partitioned approach and the non-migration approach.

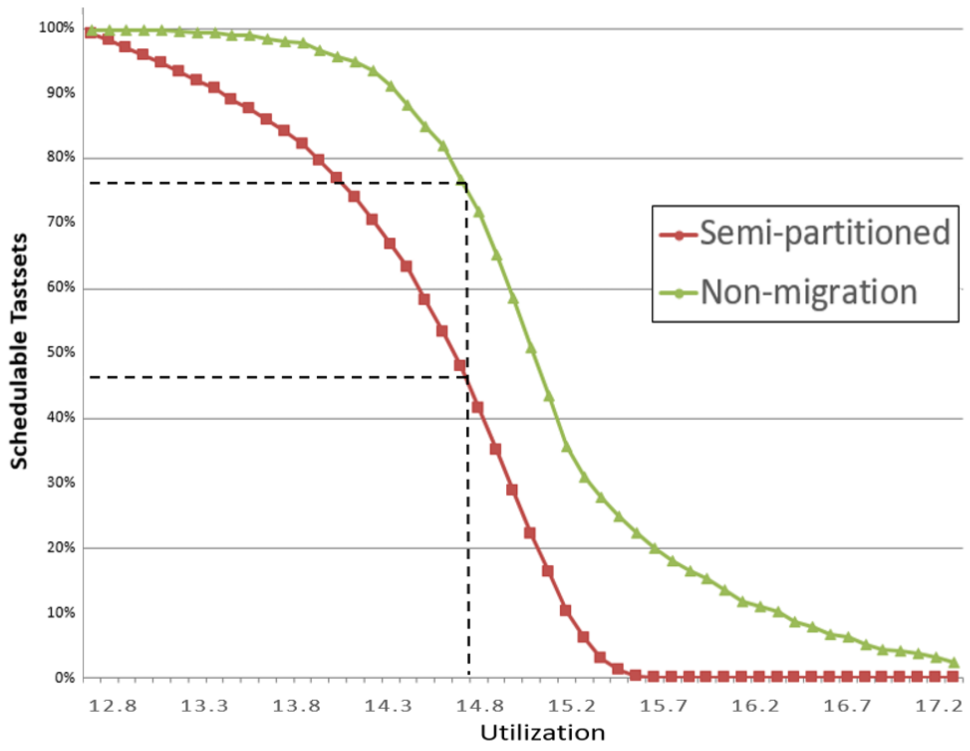


Figure 5.2: Semi-partitioned and Non-migration Comparison

Evaluation

We investigate the performance of the semi-partitioned algorithm compared with the non-migration algorithm. Figure 5.2 shows the percentage of tasksets that are schedulable for a 16-core system of 96 tasks, with on average equal number of LO-crit, MID-crit and HI-crit tasks, and the WCET estimation factor is 2 ($f = 2$). The Y-axis shows the percentage of the successful tasksets while the X-axis shows the sum of nominal utilizations of each taskset. The nominal utilization stands for $C(LO)$ for LO-crit tasks, $C(MID)$ for MID-crit tasks, and $C(HI)$ for HI-crit tasks. The sum of utilization ranges from 12.8 to 17.6 to amplify the results. According to the results, semi-partitioned algorithm has a significant improvement margin compared with non-migration algorithm. For example, as shown with black lines in the figure, the semi-partitioned algorithm schedules 77% of the tasksets with utilization of 14.86 while the non-migration one only schedules 49%. There is a $\frac{77-49}{49} * 100\% = 57.14\%$ schedulability improvement from the semi-partitioned algorithm over the non-migration algorithm in this scenario.

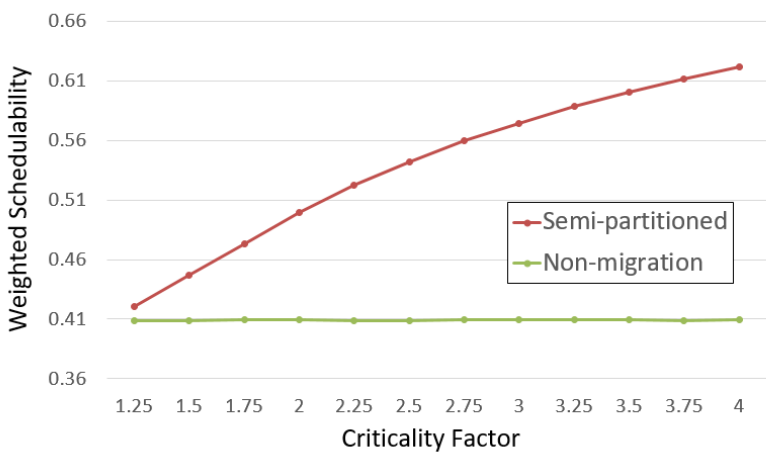


Figure 5.3: Varying the Criticality Factor

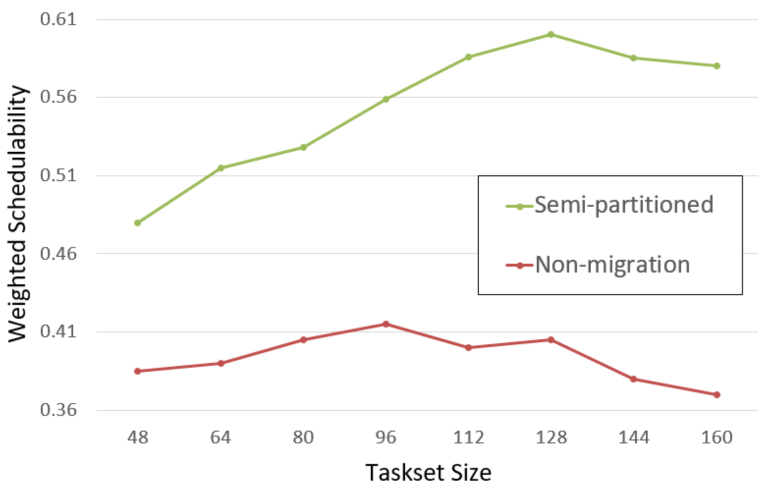


Figure 5.4: Varying the Taskset Size

In order to further explore the performance of the algorithms relating to the criticality factor and the size of the taskset, weighted schedulability measure algorithm is used to reduce the 3-dimensional plot to 2 dimensions. Figure 5.3 varies the criticality factor and Figure 5.4 varies the size of each taskset. In these weighted figures, the X-axis stands for the parameter examined and Y-axis represents the weighted value. Both figures indicate that semi-partitioned algorithm provides a significant better performance than non-migration algorithm. Regarding to Figure 5.3, the schedulability increases rapidly with the increase of the factor value. It is expected as there exist three criticality levels, the HI-crit WCET estimation increases much faster than two criticality scenarios, which provides more scheduling space for cores in the LO-crit mode. Regarding

Figure 5.4, the semi-partitioned approach no longer forms an inverted U-shape curve as it did in the experiments in previous chapters. The reason is possibly that migration is only permitted between neighbour cores in NoC and migratable tasks never migrate twice (not referring to migrating back) which lowers the influence of increasing the number of tasks. However, when the number of tasks in the taskset is sufficiently large, the schedulability of semi-partitioned approach is expected to decrease.

5.4 Summary

This chapter first proposes a model that each task only uses two WCET estimated values, and extends the semi-partitioned model based on the new model. Then it introduces the influence from considering the system architecture of the cores, and proposes a semi-partitioned model of a NoC-based multi-core platform. This chapter then provides a detailed response analysis for the semi-partitioned model on a 16-core NoC 3-criticality system, as well as a task allocation approach. An experiment is then introduced to assess the performance of the semi-partitioned model and it is shown that semi-partitioned algorithm has a significant improvement over the non-migration algorithm.

Although the intuition behind the analysis of a three criticality level NoC is straightforward, the actual response time analysis is complex and detailed. Three different types of cores are identified and each of them has a number of different scenarios to model. Overall, the improvement in performance comes with a considerable burden in terms of understanding of the analysis.

Chapter 6

Conclusion

The main focus of the research undertaken is to find an appropriate semi-partitioned algorithm for the multi-core mixed-criticality system. In the following sections, the contributions of the thesis are summarized and an outlook is given on future works that can extend the semi-partitioned algorithm further.

6.1 Summary of Contributions

The central proposition of the thesis (see Section 1.3) claims that*A semi-partitioned approach to task placement on multiprocessor platforms can improve the performance of mixed-criticality systems, enabling all tasks to keep executing in the majority of scenarios.* The performance here refers to the schedulability, scalability and complexity, where complexity mainly refers to the runtime overheads.

Chapter 1 and Chapter 2 provide an introduction and an examination of works related to the mixed-criticality systems and task allocation algorithms. By comparing the existing algorithms, considering scheduling efficiency and predictability, Adaptive Mixed Criticality (AMC-rtb) is chosen to be the template algorithm to extend to the semi-partitioned model. Several issues are identified when extending the algorithm, and these issues are addressed by Chapter 3, Chapter 4 and Chapter 5.

Chapter 3 addresses the issue about determining which tasks to migrate. This chapter first shows why the semi-partitioned scheduling model dominates any non-migration scheduling model. It then explores the semi-partitioned algorithm on a dual-core platform with dual-criticality levels, where only LO-crit tasks may migrate and the migration destination is fixed. Six possible approaches are proposed based on the task allocation algorithms and the migration algorithms. It is concluded that a combination usage of Semi2FF and Semi2WF provides the most appropriate method to schedule tasks in a dual-core platform with dual-criticality levels in the situation when only LO-crit tasks may migrate and the migration destination is fixed. This chapter also considers the influence of the overhead caused by migration. According to the experiment results, the overhead value that the semi-partitioned approach may tolerant varies in different scenarios, for realistic values the proposed approach is nearly always beneficial.

Chapter 4 firstly addresses the definition of "majority scenarios" that all tasks need to be saved by a probability calculation and the idea of a boundary number. This chapter redefines the semi-partitioned model for an n-core platform: the semi-partitioned algorithm will save all of the tasks if no more than $\lceil log_2(n) \rceil$ cores are in HI-crit mode. Then this chapter addresses the migration destination choice problem by exploring the semi-partitioned algorithm on a 4-core platform with dual-criticality levels. Four semi-partitioned models are proposed, and it is concluded that the migration Model 2, which pairs the cores with a calculated amount of cores and splits the migration task loads within the paired cores, is the most appropriate approach with a combined consideration over schedulability, scalability and complexity. This chapter then explains how the Model 2 can be applied to an n-core platform.

Chapter 5 firstly extends the model to multi-criticality levels. In order to ease the scheduling complexity, it proposes to use a model where only two WCET estimated values are used by each task. It is observed that the new model allows the criticality level of a core to increase directly rather than level by level, and the migration progress may cause the accepting core to mode change. In addition, it identifies the tasks to save based on the boundary number and the current criticality level. The next part of this chapter discusses the possible effects from considering the system architecture of the cores. It proposes a mixed-criticality system on a NoC to use local information to determine the criticality level to be assured to and provides a relationship between the location of the tasks in a NoC and the boundary number. In the last part of this chapter, the semi-partitioned model on a 16-core NoC-based 3-criticality system is analysed, and an experiment is set up to identify how much the semi-partitioned algorithm improves the schedulability from the non-migration algorithm.

Considering the thesis proposition made in the beginning, different semi-partitioned scheduling models have been proposed for corresponding multi-core MCS settings in these three chapters. In these semi-partitioned scheduling models, all of the tasks keep schedulable in most of the scenarios which meets the requirement from the hypothesis. In addition, they use the same scaling method to calculate the boundary scenario when tasks with lower criticality levels need to be abandoned if more cores enter higher criticality mode and the recommended algorithms are proved to have better performance than other proposed algorithms. According to the experiments results, it is observed that the semi-partitioned scheduling algorithm has a significant improvement on the schedulability over the non-migration partitioned algorithm for all of the multi-core MCS settings. Overall, the material illustrated in these three chapters is sufficient to satisfy the demands made in the thesis proposition.

6.2 Future Work

Due to time constraints, all of the comparisons done in the thesis use schedulability tests. Although all of the test results have been checked, it would be invaluable to run a simulation of the semi-partitioned model to make the results more convincing. In addition, the overhead is assumed to only relate to the WCET estimations of the tasks, while cache miss and other elements may also affect the overhead caused by the migration progress. Regarding to that, the work could be followed up by a full implementation of a multi-core system from which the overhead can be measured.

In the extended NoC model, it is assumed that each task only has two WCET estimations, which eventually causes a potential problem of criticality skipping. It is possible that using the original criticality model may bring some benefits in certain scenarios, which is valuable to explore.

Appendix

This Appendix completes the analysis provided in Chapter 5. It contains the analysis of cores that are neither corner nor edge.

For the cores of the normal type, they have four neighbours. According to the analysis in Section 5.2, the boundary number for the normal core is 2. Assume that the set of tasks S containing three criticality levels are scheduled on the normal core c_n and its four neighbour cores c_1, c_2, c_3 and c_4, then on each core, there exist five types of tasks: HI-crit tasks, statically allocated MID-crit tasks, migratable MID-crit tasks, statically allocated LO-crit tasks and migratable LO-crit tasks. Since the scheduling in this part focuses on the normal core, the migrating tasks from the neighbour cores to other cores are not interested here. In addition, the normal core c_n cannot differentiate whether the neighbour core c_1, c_2, c_3 and c_4 have accepted any migrating LO-crit tasks or not. But having these tasks in neighbour cores does not affect the schedulability test on the focused core. Thus, in order to simplify the analysis, it is safe to assume that there is no migrating LO-crit tasks from other cores on two neighbour cores. Then the following relationship can be obtained:

- $S = LO_e \cup MID_e \cup HI_e \cup MIGL_{e,*,*} \cup MIGM_{c,*,*} \cup LO_1 \cup MID_1 \cup HI_1 \cup MIGL_{1,n,*} \cup MIGL_{1,*,n} \cup MIGL_{1,*,*} \cup MIGM_{1,n,*} \cup MIGM_{1,*,n} \cup MIGM_{1,*,*} \cup LO_2 \cup MID_2 \cup HI_2 \cup MIGL_{2,n,*} \cup MIGL_{2,*,n} \cup MIGL_{2,*,*} \cup MIGM_{2,n,*} \cup MIGM_{2,*,n} \cup MIGM_{2,*,*} \cup LO_3 \cup MID_3 \cup HI_3 \cup MIGL_{3,n,*} \cup MIGL_{3,*,n} \cup MIGL_{3,*,*} \cup MIGM_{3,n,*} \cup MIGM_{3,*,n} \cup MIGM_{3,*,*} \cup LO_4 \cup MID_4 \cup HI_4 \cup MIGL_{4,n,*} \cup MIGL_{4,*,n} \cup MIGL_{4,*,*} \cup MIGM_{4,n,*} \cup MIGM_{4,*,n} \cup MIGM_{4,*,*}$

In the steady state mode, all these tasks are statically partitioned on each core and executing with their LO-crit budgets. Define state X to represent this phase, then the relationship between tasks and cores can be represented as:

- $X = X_n \cup X_1 \cup X_2 \cup X_3 \cup X_4$
- $X_n = LO_n \cup MID_n \cup HI_n \cup MIGL_{n,*,*} \cup MIGM_{n,*,*}$
- $X_1 = LO_1 \cup MID_1 \cup HI_1 \cup MIGL_{1,n,*} \cup MIGL_{1,*,n} \cup MIGL_{1,*,*} \cup MIGM_{1,n,*} \cup MIGM_{1,*,n} \cup MIGM_{1,*,*}$
- $X_2 = LO_2 \cup MID_2 \cup HI_2 \cup MIGL_{2,n,*} \cup MIGL_{2,*,n} \cup MIGL_{2,*,*} \cup MIGM_{2,n,*} \cup MIGM_{2,*,n} \cup MIGM_{2,*,*}$
- $X_3 = LO_3 \cup MID_3 \cup HI_3 \cup MIGL_{3,n,*} \cup MIGL_{3,*,n} \cup MIGL_{3,*,*} \cup MIGM_{3,n,*} \cup MIGM_{3,*,n} \cup MIGM_{3,*,*}$
- $X_4 = LO_4 \cup MID_4 \cup HI_4 \cup MIGL_{4,n,*} \cup MIGL_{4,*,n} \cup MIGL_{4,*,*} \cup MIGM_{4,n,*} \cup MIGM_{4,*,n} \cup MIGM_{4,*,*}$

Regarding state X, all tasks are executing with their LO-crit budgets. In this case, the response time analysis of all tasks is given by equation (6.1):

$$\forall \tau_i \in X : R_i = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(LO) \tag{6.1}$$

Since the boundary number is 2 for the edge cores, if core c_n enters MID-crit mode while two neighbour cores are already in MID-crit mode, the LO-crit tasks on core c_n are still migratable. Thus, considering the worst case, if core c_3 and c_4 enter MID-crit mode first and all of the LO-crit migratable tasks on these cores that can migrate to core c_n have been migrated, then the tasks on core c_n can be represented as:

- $Y(3,4)_n = X_(n) \cup MIGL_{3,n,*} \cup MIGL_{3,*,n} \cup MIGL_{4,n,*} \cup MIGL_{4,*,n}$

Accordingly, we can get the new response time for the tasks on the core c_n as equation (6.2):

$$\begin{aligned}\forall \tau_i \in & Y(4)_n : \\ & R_i(LO)' = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(LO)' + J_j}{T_j} \right\rceil C_j(LO)\end{aligned} \tag{6.2}$$

If core c_n then enters MID-crit mode while core c_1 and c_2 remain in LO-crit mode, then the migratable LO-crit tasks on core c_n will split and migrate to core c_1 and c_2. Due to the defined mechanism, the migrated LO-crit tasks accepted by core c_n are abandoned. The relationship between tasks and cores can be represented as:

- $Y3,4,n = Y(3,4,n)_n \cup Y(3,4,n)_1 \cup Y(3,4,n)_2 \cup Y(3,4,n)_3 \cup Y(3,4,n)_4$
- $Y(3,4,n)_n = LO_n \cup MID_n \cup HI_n \cup MIGM_{n,*,*}$
- $Y(3,4,n)_1 = Y(3)_1 \cup MIGL_{n,1,*} \cup MIGL_{n,3,1} \cup MIGL_{n,4,1}$
- $Y(3,4,n)_2 = Y(3)_2 \cup MIGL_{n,2,*} \cup MIGL_{n,3,2} \cup MIGL_{n,4,2}$
- $Y(3,4,n)_3 = Y(3,4)_3$
- $Y(3,4,n)_4 = Y(3,4)_4$
- $MIGL_{3,n,*} \cup MIGL_{3,*,n} \cup MIGL_{4,n,*} \cup MIGL_{4,*,n}$ are abandoned

In this state, all of the MID-crit tasks remaining on core c_n are executing with MID-crit budgets while all of the other tasks are executing with their LO-crit budgets. The migratable LO-crit tasks are executing with LO-crit budgets at the new core and suffering reduced deadline and release jitter issues influence. Thus, the response time analysis of cores in this state is given by equation (6.3).

$$
\begin{aligned}
&\forall \tau_i \in \cup MIGL_{n,1,*} \cup MIGL_{n,3,1} \cup MIGL_{n,4,1} \cup MIGL_{n,2,*} \cup MIGL_{n,3,2} \\
&\qquad \cup MIGL_{n,4,2} : \\
&\qquad D_i' = D_i - (R_i(LO)' - C_i(LO)) \\
&\qquad J_i = R_i(LO)' - C_i(LO) \\
&\forall \tau_i \in Y(3,4,n)_n : \\
&\qquad R_i(MID) = C_i(\delta L_i, MID) + \sum_{\tau_j \in chpL(i)} \left\lceil \frac{R_i(MID)}{T_j} \right\rceil C_j(LO) \\
&\qquad + \sum_{\tau_k \in chpM(i)} \left\lceil \frac{R_i(MID)}{T_k} \right\rceil C_k(MID) + \sum_{\tau_l \in chpH(i)} \left\lceil \frac{R_i(MID)}{T_l} \right\rceil C_l(LO) \\
&\qquad + \sum_{\tau_j \in chpM(i)} \left\lceil \frac{R_i(MID)}{T_j} \right\rceil C_j(MID) + \sum_{\tau_m \in chpMIGL(i)} \left\lceil \frac{R_i(LO)'}{T_m} \right\rceil C_m(LO) \\
&\forall \tau_i \in Y(3,4,n)_1 \cup Y(3,4,n)_2 : \\
&\qquad R_i(LO)' = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(LO)' + J_j}{T_j} \right\rceil C_j(LO)
\end{aligned}
\tag{6.3}
$$

If core c_n enters MID-crit mode while three or more neighbour cores are in MID'-crit mode, then all of the migratable LO-crit tasks on core c_n need to be abandoned. The schedulability test for core c_n for this scenario is covered by the following scenario. Consider the scenario that all of the neighbour cores are in MID-crit mode, and assume all of the LO-crit migratable tasks on these cores that can migrate to core c_n have been migrated, then the tasks on core c_n can be represented as:

- $Y(1,2,3,4)_n = X_(n) \cup MIGL_{1,n,*} \cup MIGL_{1,*,n} \cup MIGL_{2,n,*} \cup MIGL_{2,*,n} \cup MIGL_{3,n,*} \cup MIGL_{3,*,n} \cup MIGL_{4,n,*} \cup_{4,*,n}$

Accordingly, we can get the new response time for the tasks on the core c_n as equation (6.4):

$$\forall \tau_i \in Y(1,2,3,4)_n : \\ R_i(LO)'' = C_i(LO) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(LO)' + J_j}{T_j} \right\rceil C_j(LO) \tag{6.4}$$

In this case, if core c_n enters MID-crit tasks, then all of the LO-crit tasks on the core need to be abandoned. In addition, it is the worst case for c_n entering MID-crit mode.

- $Y(1,2,3,4,n)_n = MID_n \cup HI_n \cup MIGM_{n,*,*}$
- $LO_c \cup MIGL_{n,*,*} \cup MIGL_{1,n,*} \cup MIGL_{1,*,n} \cup MIGL_{2,n,*} \cup MIGL_{2,*,n} \cup MIGL_{3,n,*} \cup MIGL_{3,*,n} \cup MIGL_{4,n,*} \cup MIGL_{4,*,n}$ are abandoned

Regarding state $Y(1,2,3,4,n)_n$, MID-crit tasks will execute with their MID-crit budgets and HI-crit tasks will execute with their LO-crit budgets. Thus, the response time analysis of core c_n can be seen as equation (6.5):

$$\begin{aligned} \forall \tau_i \in & Y(1,2,3,4,n)_n : \\ & R_i(MID)' = C_i(\delta L_i, MID) + \sum_{\tau_j \in chpM(i)} \left\lceil \frac{R_i(MIG)'}{T_j} \right\rceil C_j(MID) \\ & + \sum_{\tau_k \in chpH(i)} \left\lceil \frac{R_i(MIG)'}{T_k} \right\rceil C_k(LO) + \sum_{\tau_l \in chpL(i)} \left\lceil \frac{R_i(LO)''}{T_l} \right\rceil C_l(LO) \end{aligned} \tag{6.5}$$

Still since the boundary number is 2, if core c_n enters HI-crit mode while two neighbour cores are already in HI-crit mode, the MID-crit tasks on core c_n are still migratable. Thus, if core c_3 and core c_4 enter HI-crit mode from state $Y(1,2,3,4,n)$ and all of the MID-crit migratable tasks on these cores that can migrate to core c_n have been migrated, then the tasks on core c_n can be viewed as:

- $Y(1,2,3,4,n,3H,4H)_n = MID_n \cup HI_n \cup MIGM_{n,*,*} \cup MIGM_{3,n,*}$ $\cup MIGM_{3,*,n} \cup MIGM_{4,n,*} \cup MIGM_{4,*,n}$

According to that, we can get the new response time for the tasks on the core c_n as equation (6.6):

$$\forall \tau_i \in Y(1,2,3,4,n,3H,4H)_n : \\ R_i(MID)'' = C_i(\delta L_i, MID) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(MID'' + J_j}{T_j} \right\rceil C_j(\delta L_i, MID) \tag{6.6}$$

If core c_n enters HI-crit mode from state $Y(1,2,3,4,n,3H,4H)$, then the migratable MID-crit tasks will split and migrate to core c_1 and c_2. In addition, the accepted MID-crit migrated tasks on core c_n need to be abandoned. The relationship among cores can be viewed as:

- $Y(1,2,3,4,n,3H,4H,nH)_n = MID_n \cup HI_n$
- $Y(1,2,3,4,n,3H,4H,nH)_1 = Y(1,2,3,4,n,3H,4H)_1 \cup MIGM_{n,1,*}$ $\cup MIGM_{n,3,1} \cup MIGM_{n,4,1}$
- $Y(1,2,3,4,n,3H,4H,nH)_2 = Y(1,2,3,4,n,3H,4H)_2 \cup MIGM_{n,2,*}$ $\cup MIGM_{n,3,2} \cup MIGM_{n,4,2}$
- $Y(1,2,3,4,n,3H,4H,nH)_3 = Y(1,2,3,4,n,3H,4H)_3$
- $Y(1,2,3,4,n,3H,4H,nH)_4 = Y(1,2,3,4,n,3H,4H)_4$
- $MIGM_{3,n,*} \cup MIGM_{3,*,n} \cup MIGM_{4,n,*} \cup MIGM_{4,*,n}$ are abandoned

In this state, all of the MID-crit tasks remaining on core c_n are executing with MID-crit budgets while all of the HI-crit tasks are executing with their HI-crit budgets. The migratable MID-crit tasks are executing with MID-crit budgets at the new core and suffering from reduced deadline and release jitter issues influence. Thus, the response time analysis of cores in this state is given by equation (6.7).

$$
\begin{aligned}
\forall \tau_i \in & MIGM_{n,1,*} \cup MIGM_{n,3,1} \cup MIGM_{n,4,1} \cup MIGM_{n,2,*} \cup MIGM_{n,3,2} \\
& \cup MIGM_{n,4,2} : \\
& D_i' = D_i - (R_i(MID)' - C_i(MID)) \\
& J_i = R_i(MID)' - C_i(MID) \\
\forall \tau_i \in & Y(1,2,3,n,3H,4H,nH)_n : \\
& R_i(HI) = C_i(\delta L_i, HI) + \sum_{\tau_j \in chpM(i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(MID) \\
& + \sum_{\tau_k \in chpH(i)} \left\lceil \frac{R_i(HI)}{T_k} \right\rceil C_k(HI) + \sum_{\tau_l \in chpMIGM(i)} \left\lceil \frac{R_i(MID)''}{T_l} \right\rceil C_l(MID) \\
\forall \tau_i \in & Y(1,2,3,4,n,3H,4H,nH)_1 \cup Y(1,2,3,4,n,3H,4H,nH)_2 : \\
& R_i(MID)'' = C_i(\delta L_i, MID) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(MID'' + J_j}{T_j} \right\rceil C_j(\delta L_i, MID)
\end{aligned}
\tag{6.7}
$$

Considering another scenario that core c_1 , c_2, c_3 and c_4 enter HI-crit mode first, and all of the MID-crit migratable tasks on these cores that can migrate to core c_n have been migrated, then the core is forced to mode change to MID-crit' mode and abandon all of the LO-crit tasks on the core. The tasks on core c_n can be viewed as:

- $Y(1H,2H,3H,4H)_n = MID_n \cup HI_n \cup MIGM_{n,*,*} \cup MIGM_{1,n,*} \cup MIGM_{1,*,n} \cup MIGM_{2,n,*} \cup MIGM_{2,*,n} \cup MIGM_{3,n,*} \cup MIGM_{3,*,n} \cup MIGM_{4,*,n} \cup MIGM_{4,n,*}$

According to that, we can get the new response time for the tasks on the core c_c as Equation (6.8):

$$
\begin{aligned}
\forall \tau_i \in & Y(1H,2H,3H,4H)_n : \\
& R_i(MID)''' = C_i(\delta L_i, MID) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(MID''' + J_j}{T_j} \right\rceil C_j(\delta L_i, MID)
\end{aligned}
\tag{6.8}
$$

In this case, if core c_n enters HI-crit tasks, then all of the MID-crit tasks on the core need to be abandoned. In addition, it is the worst case for c_n entering HI-crit mode.

- $Y(1H, 2H, 3H, 4H, nH)_n = HI_n$
- $MID_n \cup MIGM_{n,*,*} \cup MIGM_{1,n,*} \cup MIGM_{1,*,n} \cup MIGM_{2,n,*} \cup MIGM_{2,*,n} \cup MIGM_{3,n,*} \cup MIGM_{3,*,n} \cup MIGM_{4,n,*} \cup MIGM_{4,*,n}$ are abandoned

Regarding state $Y(1H, 2H, 3H, 4H, nH)_n$, only HI-crit tasks executes with their LO-crit budgets. Thus, the response time analysis of core c_n can be seen as Equation (6.9):

$$\begin{aligned}\forall \tau_i \in & Y(1H, 2H, 3H, eH)_n : \\ & R_i(HI)' = C_i(\delta L_i, HI) + \sum_{\tau_j \in chpH(i)} \left\lceil \frac{R_i(HI)'}{T_j} \right\rceil C_j(HI) \\ & + \sum_{\tau_k \in chpM(i)} \left\lceil \frac{R_i(MID)'''}{T_k} \right\rceil C_l(MID)\end{aligned} \tag{6.9}$$

For completion, the scenario that core c_n increases directly from LO-crit mode to HI-crit mode needs to be considered. This scenario only happens when core c_1, c_2, c_3 and c_4 are in LO-crit or LO'-crit or MID-crit or MID'-crit mode (if c_1 is already in HI-crit mode then core c_n is forced to mode change to MID'-crit mode). In any of the above situations, the migratable MID-crit tasks will migrate to core c_1, c_2, c_3 and c_4. Considering that, the worst case happens when core c_1, c_2, c_3 and c_4 are all in MID-crit mode and all of the LO-crit migratable tasks on these cores that can migrate to core c_n have been migrated, then the tasks on core c_n can be viewed as:

- $Y(1, 2, 3, 4, nH)_n = MID_n \cup HI_n$
- $Y(1, 2, 3, 4, nH)_1 = Y(1, 2, 3, 4)_1 \cup MIGM_{n,1,*}$
- $Y(1, 2, 3, 4, nH)_2 = Y(1, 2, 3, 4)_2 \cup MIGM_{n,2,*}$

- $Y(1,2,3,4,nH)_3 = Y(1,2,3,4)_3 \cup MIGM_{n,3,*}$
- $Y(1,2,3,4,nH)_4 = Y(1,2,3,4)_4 \cup MIGM_{n,4,*}$
- $LO_n \cup MIGL_{n,*,*} \cup MIGL_{1,n,*} \cup MIGL_{1,*,n} \cup MIGL_{2,n,*} \cup MIGL_{2,*,n} \cup MIGL_{3,n,*} \cup MIGL_{3,*,n} \cup MIGL_{4,*,n} \cup MIGL_{4,n,*}$ are abandoned.

Regarding this state, the migratable MID-crit tasks perform quite special that although they are executing with MID-crit budgets, the reduced deadline and release jitter issues are calculated using the LO-crit response time and LO-crit budgets. In addition, the interference from these migrating tasks is also calculated using their LO-crit budgets. Thus, the response time analysis of the cores can be viewed as equation (6.10):

$$\begin{aligned}
&\forall \tau_i \in MIGM_{n,*,*}:\\
&\qquad D_i'' = D_i - (R_i(LO)' - C_i(LO))\\
&\qquad J_i' = R_i(LO)' - C_i(LO)\\
&\forall \tau_i \in Y(1,2,3,4,nH)_n:\\
&\qquad R_i(HI)'' = C_i(\delta L_i, HI) + \sum_{\tau_j \in chpM(i)} \left\lceil \frac{R_i(HI)''}{T_j} \right\rceil C_j(MID)\\
&\qquad + \sum_{\tau_k \in chpH(i)} \left\lceil \frac{R_i(HI)''}{T_k} \right\rceil C_k(HI) + \sum_{\tau_l \in chpMIGM(i)} \left\lceil \frac{R_i(LO)'}{T_l} \right\rceil C_l(LO)\\
&\qquad + \sum_{\tau_m \in chpL(i)} \left\lceil \frac{R_i(LO)'}{T_m} \right\rceil C_m(LO)\\
&\forall \tau_i \in Y(1,2,3,4,nH)_1 \cup Y(1,2,3,4,nH)_2 \cup Y(1,2,3,4,nH)_3 \cup Y(1,2,3,4,nH)_4:\\
&\qquad R_i(MID)'' = C_i(\delta L_i, MID) + \sum_{\tau_j \in chp(i)} \left\lceil \frac{R_i(MID'' + J_j}{T_j} \right\rceil C_j(\delta L_i, MID)
\end{aligned} \tag{6.10}$$

If all of the above schedulability tests have been passed, then the tasks on the normal core are deemed to be scheduled.

Bibliography

[1] Z. Al-Bayati, Q. Zhao, A. Youssef, H. Zeng, and Z. Gu. Enhanced partitioned scheduling of mixed-criticality systems on multicore platforms. In *Design Automation Conference (ASP-DAC), 2015, 20th Asia and South Pacific*, pages 630–635. IEEE, 2015.

[2] J. H. Anderson, S. K. Baruah, and B. Brandenburg. Multicore operating-system support for mixed criticality. In *Proceedings of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*. Citeseer, 2009.

[3] B. Andersson. Global static-priority preemptive multiprocessor scheduling with utilization bound 38%. In *International Conference on Principles of Distributed Systems*, pages 73–88. Springer, 2008.

[4] B. Andersson, S. Baruahand, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 193–202. IEEE, 2001.

[5] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[6] N. C. Audsley. Memory architectures for NoC-based real-time mixed criticality systems. *Proc. WMC, RTSS*, pages 37–42, 2013.

[7] T. P. Baker. An analysis of fixed-priority schedulability on a multiprocessor. *Real-Time Systems*, 32(1-2):49–71, 2006.

[8] S. K. Baruah. Schedulability analysis for a general model of mixed-criticality recurrent real-time tasks. In *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pages 25–34. IEEE, 2016.

[9] S. K. Baruah, V. Bonifaci, G. DAngelo, A. Marchetti-Spaccamela H. Li, S. Van Der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *24th Euromicro Conference on Real-Time Systems (ECRTS), 2012*, pages 145–154. IEEE, 2012.

[10] S. K. Baruah, V. Bonifaci, G. D'angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems. *Journal of the ACM (JACM)*, 62(2):14, 2015.

[11] S. K. Baruah, V. Bonifaci, G. D'Angelo, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. Mixed-criticality scheduling of sporadic task systems. In *ESA*, pages 555–566. Springer, 2011.

[12] S. K. Baruah and A. Burns. Implementing mixed criticality systems in Ada. In *Reliable Software Technologies-Ada-Europe*, pages 174–188. Springer, 2011.

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

[14] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[15] S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pages 280–288. IEEE, 1995.

[16] S. K. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 13–22. IEEE, 2010.

[17] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium*, pages 182–190. IEEE, 1990.

[18] S. K. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Real-Time Systems*, pages 147–155. IEEE, 2008.

[19] A. Bastoni, B. Brandenburg, and J. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. *Proceedings of OSPERT*, pages 33–44, 2010.

[20] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Is semi-partitioned scheduling practical? In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 125–135. IEEE, 2011.

[21] I. Bate, A. Burns, and R. I. Davis. A bailout protocol for mixed criticality systems. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 259–268. IEEE, 2015.

[22] I. Bate, A. Burns, and R. I. Davis. An enhanced bailout protocol for mixed criticality embedded software. *IEEE Transactions on Software Engineering*, 43(4):298–320, 2017.

[23] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.

[24] B. Brandenburg and J. Anderson. On the implementation of global real-time schedulers. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 214–224. IEEE, 2009.

[25] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Real-Time Systems Symposium, 2008*, pages 157–169. IEEE, 2008.

[26] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML). *World Wide Web Journal*, 2(4):27–66, 1997.

[27] A. Burns. *Preemptive priority based scheduling: An appropriate engineering approach.* 1993.

[28] A. Burns. System mode changes-general and criticality-based. In *Proceedings of 2nd Workshop on Mixed Criticality Systems (WMC)*, pages 3–8. RTSS, 2014.

[29] A. Burns. An augmented model for mixed criticality. In *Proc. of Dagstuhl seminar on Mixed Criticality on Multicore/Manycore Platforms*, pages 0098–5589, 2015.

[30] A. Burns and R. I. Davis. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40, 2010.

[31] A. Burns and R. I. Davis. Adaptive mixed criticality scheduling with deferred preemption. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 21–30. IEEE, 2014.

[32] A. Burns, J. Harbin, and L. S. Indrusiak. A wormhole noc protocol for mixed criticality systems. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 184–195. IEEE, 2014.

[33] A. Burns and B. Littlewood. Reasoning about the reliability of multi-version, diverse real-time systems. In *Real-Time Systems Symposium (RTSS)*, pages 73–81. IEEE, 2010.

[34] A. Burns and A. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX.* Pearson Education, 2001.

[35] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX.* Addison-Wesley Educational Publishers Inc, 2009.

[36] G. C. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*, pages 286–295. IEEE, 1998.

[37] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. H. Anderson, and S. K. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms., 2004.

[38] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance Sparc CMT processor. *IEEE micro*, 29(2), 2009.

[39] Y. Chen, Q. Li, Z. Li, and H. Xiong. Efficient schedulability analysis for mixed-criticality systems under deadline-based scheduling. *Chinese Journal of Aeronautics*, 27(4):856–866, 2014.

[40] S. Cheng, J. A. Stankovic, and K. Ramamritham. Scheduling algorithms for hard real-time systems — a brief survey. 1987.

[41] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 101–110. IEEE, 2006.

[42] R. I. Davis and M. Bertogna. Optimal fixed priority scheduling with deferred pre-emption. In *Real-Time Systems Symposium (RTSS), 2012, 33rd IEEE*, pages 39–50. IEEE, 2012.

[43] R. I. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40, 2011.

[44] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35, 2011.

[45] R. I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Transactions on Computers*, 57(9):1261–1276, 2008.

[46] J. Diemer and R. Ernst. Back suction: Service guarantees for latency-sensitive on-chip networks. In *Networks-on-Chip (NOCS)*, pages 155–162. IEEE, 2010.

[47] F. Dorin, P. Richard, M. Richard, and J. Goossens. Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-time systems*, 46(3):305–331, 2010.

[48] A. Easwaran. Demand-based scheduling of mixed-criticality sporadic tasks on one processor. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 78–87. IEEE, 2013.

[49] P. Ekberg and W. Yi. Outstanding paper award: Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Real-Time Systems (ECRTS)*, pages 135–144. IEEE, 2012.

[50] P. Ekberg and W. Yi. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Real-time systems*, 50(1):48–86, 2014.

[51] P. Emberson, R. Staggord, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time System (WATERS)*.

[52] T. Fleming and A. Burns. Extending mixed criticality scheduling. *Proc. WMC, RTSS*, pages 7–12, 2013.

[53] S. Funk and S. K. Baruah. Restricting EDF migration on uniform heterogeneous multiprocessors. *TSI. Technique et science informatiques*, 24(8):917–938, 2005.

[54] L. George, P. Courbin, and Sorel Y. Job vs. portioned partitioning for the earliest deadline first semi-partitioned scheduling. *Journal of Systems Architecture*, 57(5):518–535, 2011.

[55] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens. A globally arbitrated memory tree for mixed-time-criticality systems. *IEEE Transactions on Computers*, 66(2):212–225, 2017.

[56] P. Graydon and I. Bate. Safety assurance driven problem formulation for mixed-criticality scheduling. *Proc. WMC, RTSS*, pages 19–24, 2013.

[57] N. Guan, P. Ekberg, M. Stigge, and W. Yi. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *Real-Time Systems Symposium (RTSS)*, pages 13–23. IEEE, 2011.

[58] N. Guan, P. Ekberg, M. Stigge, and W. Yi. Improving the scheduling of certifiable mixed-criticality sporadic task systems. *Technical Report 2013–008*, 2013.

[59] N. Guan, M. Stigge, W. Yi, and G. Yu. Fixed-priority multiprocessor scheduling with liu and layland's utilization bound. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 165–174. IEEE, 2010.

[60] L. S. Indrusiak, J. Harbin, and A. Burns. Average and worst-case latency improvements in mixed-criticality wormhole networks-on-chip. In *Real-Time Systems (ECRTS), 27th Euromicro Conference on IEEE*, pages 47–56. IEEE, 2015.

[61] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[62] A. Burns K. Tindell and A. Wellings. Mode changes in priority preemptively scheduled systems. In *Real-Time Systems Symposium, 1992*, pages 100–109. IEEE, 1992.

[63] S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 23–32. IEEE, 2009.

[64] O. R. Kelly, H. Aydin, and B. Zhao. On partitioned scheduling of fixed-priority mixed-criticality task sets. In *Trust, Security and Privacy in Computing and Communications*, pages 1051–1059. IEEE, 2011.

[65] H. Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[66] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *21st Euromicro Conference on Real-Time Systems, 2009. ECRTS'09*, pages 239–248. IEEE, 2009.

[67] J. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information processing letters*, 11(3):115–118, 1980.

[68] J. Y. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation*, 2(4):237–250, 1982.

[69] G. Lipari and G. Buttazzo. Resource reservation for mixed criticality systems. In *Proceeding of Workshop on Real-Time Systems: the past, the present, and the future*, pages 60–74, 2013.

[70] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[71] F. Liu, A. Narayanan, and Q. Bai. Real-time systems. 2000.

[72] J. M. López, J. L. Díaz, and D. F. García. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, 2004.

[73] A. Masrur, D. Müller, and M. Werner. Bi-level deadline scaling for admission control in mixed-criticality systems. In *21st International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2015 IEEE*, pages 100–109. IEEE, 2015.

[74] R. G. Michael and D. S. Johnson. Computers and intractability: A guide to the theory of NP-completeness. *WH Freeman & Co., San Francisco*, 1979.

[75] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, 1993.

[76] D. Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Real-Time Systems Symposium*, pages 291–300. IEEE, 2009.

[77] T. Park and S. Kim. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *Embedded Software (EMSOFT)*, pages 253–262. IEEE, 2011.

[78] R. Pellizzoni, A. Schranzhofer, J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 741–746. European Design and Automation Association, 2010.

[79] P. Rodriguez, L. George, Y. Abdeddaim, and J. Goossens. Multi-criteria evaluation of partitioned EDF-VD for mixed-criticality systems upon identical processors. In *Workshop on Mixed Criticality Systems*, 2013.

[80] M. Saksena and Y. Wang. Scalable real-time system design using preemption thresholds. In *Proceedings of Real-Time Systems Symposium, 2000, the 21st IEEE*, pages 25–34. IEEE, 2000.

[81] F. Santy, G. Raravi, G. Nelissen, V. Nelis, P. Kumar, J. Goossens, and E. Tovar. Two protocols to reduce the criticality level of multiprocessor mixed-criticality systems. In *Proceedings of the 21st International conference on Real-Time Networks and Systems*, pages 183–192. ACM, 2013.

[82] H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 147–152. EDA Consortium, 2013.

[83] H. Su, D. Zhu, and D. Mossé. Scheduling algorithms for elastic mixed-criticality tasks in multicore systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 19th International Conference on IEEE 2013*, pages 352–357. IEEE, 2013.

[84] K. Tindell and A. Alonso. A very simple protocol for mode changes in priority preemptive systems. *Universidad Politécnica de Madrid, Tech. Rep*, 1996.

[85] S. Tobuschat, R. Ernst P. Axer, and J. Diemer. IDAMC: A NoC for mixed criticality systems. In *RTCSA*, pages 149–156, 2013.

[86] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium 2007, 28th IEEE International*, pages 239–243. IEEE, 2007.

[87] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al. The worst-case execution-time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[88] W. Wolf. Multiprocessor system-on-chip technology. *IEEE Signal Processing Magazine*, 26(6), 2009.

[89] F. Zhang and A. Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Transactions on Computers*, 58(9):1250–1258, 2009.

[90] Q. Zhao, Z. Gu, and H. Zeng. Integration of resource synchronization and preemption-thresholds into EDF-based mixed-criticality scheduling algorithm. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on*, pages 227–236. IEEE, 2013.

[91] Q. Zhao, Z. Gu, and H. Zeng. PT-AMC: Integrating preemption thresholds into mixed-criticality scheduling. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 141–146. IEEE, 2013.