USING IMPRECISE COMPUTING FOR IMPROVED REAL-TIME SCHEDULING

A Dissertation

by

LIN HUANG

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jiang Hu
Committee Members,	Weiping Shi
	Riccardo Bettati
	Le Xie
Head of Department,	Miroslav M. Begovic

December 2019

Major Subject: Computer Engineering

ABSTRACT

Conventional hard real-time scheduling is often overly pessimistic due to the worst case execution time estimation. The pessimism can be mitigated by exploiting imprecise computing in applications where occasional small errors are acceptable. This leverage is investigated in a few previous works, which are restricted to preemptive cases. We study how to make use of imprecise computing in uniprocessor non-preemptive real-time scheduling, which is known to be more difficult than its preemptive counterpart. Several heuristic algorithms are developed for periodic tasks with independent or cumulative errors due to imprecision. Simulation results show that the proposed techniques can significantly improve task schedulability and achieve desired accuracy–schedulability tradeoff. The benefit of considering imprecise computing is further confirmed by a prototyping implementation in Linux system.

Mixed-criticality system is a popular model for reducing pessimism in real-time scheduling while providing guarantee for critical tasks in presence of unexpected overrun. However, it is controversial due to some drawbacks. First, all low-criticality tasks are dropped in high-criticality mode, although they are still needed. Second, a single high-criticality job overrun leads to the pessimistic high-criticality mode for all high-criticality tasks and consequently resource utilization becomes inefficient. We attempt to tackle aforementioned two limitations of mixed-criticality system simultaneously in multiprocessor scheduling, while those two issues are mostly focused on uniprocessor scheduling in several recent works. We study how to achieve graceful degradation of low-criticality tasks by continuing their executions with imprecise computing or even precise computing if there is sufficient utilization slack. Schedulability conditions under this Variable-Precision Mixed-Criticality (VPMC) system model are investigated for partitioned scheduling and global fpEDF-VD scheduling. And a deferred switching protocol is introduced so that the chance of switching to high-criticality mode is significantly reduced. Moreover, we develop a precision optimization approach that maximizes precise computing of low-criticality tasks through 0-1 knapsack formulation. Experiments are performed through both software simulations and Linux proto-

typing with consideration of overhead. Schedulability of the proposed methods is studied so that the Quality-of-Service for low-criticality tasks is improved with guarantee of satisfying all deadline constraints. The proposed precision optimization can largely reduce computing errors compared to constantly executing low-criticality tasks with imprecise computing in high-criticality mode.

DEDICATION

To my mother, my father, my grandfather, and my grandmother.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Jiang Hu, for his guidance and advice during my PHD program. He shows me how to be an independent researcher, and I have learned a lot from his attitude toward work and research. I'm lucky to have him as my advisor, and it's almost impossible for me to finish the PHD program withouth his guidance.

I would like to thank my committee members, Professor Weiping Shi, Professor Riccardo Bettati, Professor Le Xie of Texas A&M University, for their helpful suggestions on my research.

I would like to thank Professor Sachin S. Sapatnekar of University of Minnesota, Professor I-Hong Hou of Texas A&M University, Professor Youmeng Li of Tianjin University for their help and advice on my research. I have learned a lot from discussion with them.

Thanks to my mates in Texas AM University, Wenbin Xu, Jiafan Wang, Hao He, He Zhou, Chaofan Li, Justin Sun, Yanxiang Yang, Lang Feng, Hongxin Kong, Rongjian Liang, and Yaguang Li.

Finally, I would like to thank my parents and my wife for their support during my PHD program.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Jiang Hu, advisor, and Professor Weiping Shi, Professor Le Xie of the Department of Electrical and Computer Engineering, and Professor Riccardo Bettati of the Department of Computer Science and Engineering.

Some test cases for Chapter 4 were prepared by Professor Youmeng Li of Tianjin University.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by National Science Foundation (NSF) under Grant CCF-1525925 and CCF-1525749, and Office of Naval Research (N00014-18-1-2048).

NOMENCLATURE

WCET	Worst Case Execution Time
QoS	Quality of Service
DVFS .	Dynamic Voltage and Frequency Scaling
NP	Non-deterministic Polynomial-time
EDF	Earliest Deadline First
EDF-VD	Earliest Deadline First with Virtual Deadlines
ILP	Integer Linear Programming
fpEDF-VD	fixed-priority Earliest Deadline First with Virtual Deadlines
MC	Mixed-Criticality
IMC	Imprecise Mixed-Criticality
VPMC	Variable-Precision Mixed-Criticality
MC-DP-Fair	Mixed-Criticality Deadline Partition Fair
MCFQ	Mixed-Criticality Fluid scheduling with Quality of Service

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION [1]

Real-time system is widely used in applications such as flight control, medical equipment and automobiles, real-time monitors, etc. It requires both logical and temporal correctness. Logical correctness means the computing result needs to be correct, and temporal correctness means the task execution needs to be finished before required time (deadline). In our research, we work on *hard* real-time system, where missing deadline can lead to catastrophic result. Therefore the designers usually make the most conservative estimation of task execution time, which is called Worst Case Execution Time (WCET), in order to guarantee temporal correctness. Conventional hard real-time scheduling is often overly pessimistic due to the WCET estimation. In our research we explore imprecise computing to mitigate the pessimism of real-time scheduling, and thereby improve Quality of Service (QoS) for real-time system.

Imprecise computing, which is sometimes called approximate computing [1], is an unconventional approach to low power systems. The concept of imprecise computing appeared more than two decades ago [2]. Recently, its realization has made a lot of progress [1] and therefore its application in real-time system becomes more practical. It is based on the observation that occasional small computing errors are acceptable for applications like video/audio processing, recognition and mining, which are of growing interest. By intentionally allowing such errors in design, a computing system can operate with reduced power or improved speed. Imprecise computing can benefit real-time systems as well. In a virtual reality tracking system, for instance, deadline violations cause video discontinuity, which may hamper mission success. By contrast, errors at a few pixels of a small number of frames are usually indiscernible to human eyes and have much less

[1] Reprinted with permission from "Using imprecise computing for improved non-preemptive real-time scheduling " by Lin Huang, Youmeng Li, Sachin S.Sapatnekar, Jiang Hu, 2018. *Proceedings of Design Automation Conference (DAC)*, Page 1-6 , ©2018 IEEE, "Graceful degradation of low-criticality tasks in multiprocessor dual-criticality systems " by Lin Huang, I-Hong Hou, Sachin S.Sapatnekar, Jiang Hu, 2018. *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*, Page 159-169 , ©2018 ACM and "Improving QoS for global dual-criticality scheduling on multiprocessors " by Lin Huang, I-Hong Hou, Sachin S.Sapatnekar, Jiang Hu, 2019. *Proceedings of the International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Page 1-11, ©2019 IEEE.

serious consequence than deadline violations. As imprecise computing permits relatively short execution time, it can be adopted to avoid deadline violations when computing resource is under stress. In general, it offers an opportunity for improving schedulability and reducing the pessimism in real-time scheduling.

1.1 Non-preemptive Real-time Scheduling

The early works of considering imprecise computing in real-time scheduling [3, 2, 4] are restricted to preemptive cases. Meanwhile, imprecision must be applied prudently so that errors are well controlled in all conditions. In our research, we investigate how to use imprecise computing for improving non-preemptive real-time scheduling of independent tasks on uniprocessor. Compared to preemptive cases, non-preemptive scheduling implies less context switching overhead and has more predictable characteristics [5, 6]. In some scenarios, task preemption is even impossible or formidably expensive [5]. On the other hand, it is proved [5] that non-preemptive scheduling for periodic tasks with specific release times is NP-hard and its schedulability test is also NP-hard. Considering imprecise computing greatly escalates scheduling complexity and makes some preemptive problems NP-hard [2].

We propose several heuristic algorithms for scheduling periodic tasks considering imprecise computing with independent or cumulative errors. It is not trivial to extend preemptive techniques [3, 2] to our non-preemptive case. For tasks satisfying schedulability conditions in imprecise mode, our heuristics can guarantee that there is no deadline violation. In these algorithms, errors due to imprecise computing are controlled either offline with guarantee or online in the best effort. Simulation results from random and realistic cases indicate that our scheduling methods can indeed improve schedulability and significantly mitigate the pessimism of the worst case execution time model. The advantage is further demonstrated by a prototyping implementation in Linux system.

1.2 Mixed-criticality (MC) Scheduling

The drawback of pessimistic WCET estimation in conventional real-time scheduling can be partially addressed in mixed-criticality (MC) system scheduling, which has attracted a great deal of research attention in the past 10 years [7, 8, 9, 10, 11, 12]. Mixed-criticality scheduling categorizes tasks into different criticality levels and asymmetrically emphasizes high-criticality tasks. Its key elements can be described through a dual-criticality system, which starts with low-criticality mode. Once a high-criticality job executes longer than its worst case execution time (WCET), the system is switched to high-criticality mode, where very pessimistic WCET is applied for all high-criticality tasks. As such, high-criticality tasks still have guarantee for meeting their deadlines. However, such appealing advantage of MC systems comes with expensive price [12].

1. All low-criticality tasks are dropped in high-criticality mode to facilitate the guarantee for high-criticality tasks. Despite their low-criticality, these tasks are still very much needed and completely abandoning them is a heavy loss of Quality of Service (QoS).

2. The overrun of a single high-criticality job can force all the other high-criticality jobs (even without overrun) into high-criticality mode with the very pessimistic WCET, which undermines efficiency of resource utilization.

Both of the limitations recently received research attention. New techniques have been proposed [13, 14, 15, 16, 17, 18, 19, 20] to continue executing low-criticality tasks in high-criticality mode with graceful degradation. Other approaches [21, 22] define new protocols to prevent all high-criticality tasks from simultaneously entering high-criticality mode. Most of these works are geared toward uniprocessor scheduling. In reality, multiprocessor is a growing trend due to its performance advantage.

There are three main approaches to MC multiprocessor scheduling: (1) partitioning-based [10, 23, 24], (2) fluid-based [25, 26, 20] and (3) global scheduling [9, 10]. Each of them has its strength and weakness, and no one is absolutely superior to the others. Partitioning-based scheduling is appealing for its simplicity. On the other hand, it lacks flexibility and tends to cause under-utilization

of resources. Fluid-based scheduling can reach the theoretically optimal solution, but may incur frequent job preemptions and context switchings, which incur non-negligible overhead. Global scheduling can have both its performance and practicality between partitioning and fluid-based scheduling.

One approach to addressing the controversy of dropping low-criticality tasks is an elastic scheme [13, 17], where low-criticality tasks are continued with extended period in high-criticality mode. Another method is to reduce the priorities of low-criticality tasks [17]. However, it is noticed [27] that task period and priority are usually functional requirements and cannot be easily changed. A more viable approach to avoiding complete loss of low-criticality tasks is making use of imprecise computing [17, 18, 19, 20]. Each low-criticality task can alternatively be executed with imprecise computing, which causes inaccuracy in computing results but costs relatively short execution time. When a system is in high-criticality mode, low-criticality tasks can continue to execute with imprecise computing instead of being dropped. Such approach allows graceful degradation of low-criticality tasks in high-criticality mode. This model is called Imprecise Mixed-Criticality (IMC) system [19].

The schedulability conditions for several uniprocessor scheduling algorithms in IMC are established [17, 18, 19]. When deadline constraints are not tight, there is usually processor utilization slack for low-criticality tasks to continue with even precise computing. Based on this observation, the work of [20] extends the fluid-based multiprocessor scheduling [25, 26] to maximize precise computing of low-criticality tasks in high-criticality mode. A mixed-criticality system with such treatment of low-criticality tasks can be called Variable-Precision Mixed-Criticality (VPMC) system. This is perhaps the only published literature on VPMC and the only published work considering graceful degradation of low-criticality tasks for multiprocessor scheduling. Although the fluid-based scheduling [25, 26] is optimal in theory, it cannot be implemented with its original form on hardware as it depends on an unrealistic assumption that each processor can be partitioned into arbitrary fractions. MC-DP-Fair [25] is a practically implementable scheduling algorithm with schedulability equivalent to the fluid-based scheduling. However, the work of [20] has little dis-

cussion on MC-DP-Fair or the implementation issue, and MC-DP-Fair can result in an excessive amount of context switchings and hence a considerable overhead.

In our research, we study VPMC systems for some well-known multiprocessor scheduling methods that have not been well considered for the graceful degradation option yet. These include partitioned scheduling, fpEDF-VD based global scheduling [10] and MC-DP-Fair scheduling. The schedulability conditions of these methods are extended for considering low-criticality task executions in high-criticality mode. We found that the speedup factors of partitioned scheduling and fpEDF-VD are the same as before when they are applied in VPMC. We develop a precision optimization technique that maximizes precise computing of low-criticality tasks in high-criticality mode. This optimization is through formulation of 0-1 knapsack problem, which is optimally solved by dynamic programming.

Moreover, we focus on how to address the two limitations of dual-criticality system, the basic version of MC system, in global scheduling. First, a service preserving technique is developed to let all low-criticality tasks execute in high-criticality mode with imprecise computing, while all deadline constraints are satisfied. This is to improve QoS for high-criticality mode. Second, a deferred switching scheme is proposed to prevent high-criticality tasks from simultaneously entering high-criticality mode while all deadlines are still guaranteed to be enforced. This is to extend the low-criticality mode, which has better QoS than high-criticality mode. Further, these two techniques are combined into a unified framework such that the two limitations are concurrently mitigated.

The proposed techniques for VPMC are evaluated with both software simulation and Linux prototyping with consideration of overhead on context switching, execution monitoring and mode changes. Since continuing low-criticality tasks stresses processors more than dropping them, it is important to validate in a realistic setup [28]. The results show that the schedulability degradation due to continuing low-criticality tasks is often very small. In addition, VPMC systems with our precision optimization lead to significantly smaller errors than IMC systems. Although the fluid-based VPMC system [20] has the best schedulability in theory, it is outperformed by parti-

tioned scheduling when the overhead is considered. Additionally our service preserving technique alone outperforms the approach of DVD (Dual Virtual Deadline) in term of schedulability, and the effectiveness of our deferred switching technique is confirmed through simulations.

1.3 Contributions

The contributions of our research include:

- To the best of our knowledge, this is the first work on using imprecise computing for non-preemptive real-time scheduling.
- To the best of our knowledge, this is the first extensive study of IMC and VPMC systems on multiprocessors for several well-known scheduling approaches including partitioned, fpEDF-VD and MC-DP-Fair scheduling. The study covers schedulability analysis for these methods. We also show that the speedup factors of partitioned scheduling and fpEDF-VD scheduling are not changed in the VPMC model.
- An offline precision optimization technique is proposed to minimize errors from using imprecise computing by low-criticality tasks subject to schedulability constraints. This optimization can be optimally solved by dynamic programming.
- To the best of our knowledge, this is the first study of IMC, VPMC and their computing errors with consideration of overhead. Our work includes the first prototyping implementation of VPMC in contrast to only software simulations in the related previous works [18, 19, 20].
- To the best of our knowledge, this is the first work that simultaneously addresses both of the limitations for mixed-criticality scheduling on multiprocessors.

2. BACKGROUND [1]

2.1 Overview of Real-time Scheduling

In this section, we introduce some notations of real-time scheduling that are used in our research.

We work on *hard* real-time system, where deadlines can not be missed. Scheduling policies are studied to guarantee the completion of workload before deadlines.

- Scheduling policy: a scheduling policy (algorithm) is designed to decide which task to be allocated with computing resource (processor) at specific time on a platform. And *schedulability conditions* of a scheduling policy are used to check whether all the deadlines are guaranteed to be met before runtime. *Speedup factor* is a useful metric to verify the effectiveness of a scheduling policy, it is the minimum processor speed increasing factor (≥ 1) so that any task system that can be scheduled correctly by an optimal clairvoyant policy can be scheduled correctly by this scheduling policy [10].

- Workload model: we work on *sporadic* task model, which is a common *recurrent* task model in real-time scheduling. A *sporadic* task can generate infinite number of jobs, and a new job is released when it is available for execution, but the release times of two consecutive jobs are separated by a *minimum* time interval. A *sporadic* task is called *periodic* task when the release times of two consecutive jobs are separated by a *constant* time interval. In conventional real-time scheduling, one task has one Worst Case Execution Time (WCET), which is the execution time upper-bound for any job of this task, while one task can have more than

[1] Reprinted with permission from "Using imprecise computing for improved non-preemptive real-time scheduling " by Lin Huang, Youmeng Li, Sachin S.Sapatnekar, Jiang Hu, 2018. *Proceedings of Design Automation Conference (DAC)*, Page 1-6 , ©2018 IEEE, "Graceful degradation of low-criticality tasks in multiprocessor dual-criticality systems " by Lin Huang, I-Hong Hou, Sachin S.Sapatnekar, Jiang Hu, 2018. *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*, Page 159-169 , ©2018 ACM and "Improving QoS for global dual-criticality scheduling on multiprocessors " by Lin Huang, I-Hong Hou, Sachin S.Sapatnekar, Jiang Hu, 2019. *Proceedings of the International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Page 1-11 , ©2019 IEEE.

one WCETs in mixed-criticality scheduling, and detailed introduction will be given in the following sections of this chapter.

In our research, we work on both *preemptive* and *non-preemptive* systems, *preemptive* means current executing job can be suspended and another job can be chosen for execution, while for non-preemptive scheduling, a job needs to execute continuously till completion once it starts.

2.2 Imprecise Computing/Approximate Computing

Imprecise computing can be carried out at circuit, architecture and algorithm levels for datapath or numerical computations. Circuit level imprecision is mostly focused on imprecise arithmetic circuit designs [29, 1] or voltage overscaling [30]. The overscaling [30] and some dedicated designs [31] make computing accuracy runtime configurable. Architectural imprecision techniques include dedicated instructions [32] and imprecise accelerator based on neural network [33]. Different from circuit and architecture level imprecision, algorithm level imprecise computing is not an explicitly named discipline although imprecision widely exists in algorithm designs[2]. As an example, algorithmic imprecision can be realized by relaxing the convergence criterion in iterative algorithms. In the work of dynamic effort scaling [34], all of algorithm, architecture and circuit level imprecision are jointly explored. Overall, the substantial progress [1] on imprecise/approximate computing has made runtime computing precision reconfiguration feasible.

2.3 Non-preemptive Real-time Scheduling

2.3.1 Non-preemptive Real-Time Scheduling System Model

We consider to schedule a set of independent tasks $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$ onto a single processor. Each task $\tau_i, i = 1, 2, ..., n$, is a 3-tuple (r_i, C_i, d_i) representing task release time, worst case execution time and deadline. A periodic task τ_i is composed by multiple jobs $\{\tau_{i,1}, \tau_{i,2}, ...\}$, where $\tau_{i,j}$ is the j_{th} occurrence of task τ_i. The jobs of τ_i are released with period p_i, i.e., their release times and deadlines satisfy $d_{i,j} = r_{i,j} + p_i = r_{i,j+1}, j = 1, 2, ...$ In a hard real-time system, the execution of every task must be completed before its deadline. In practice, an execution time q_i may vary in a

[2]Approximate algorithm is a rigorous term with its dedicated meaning in computer science theory.

wide range. In order to ensure that no deadline violation is incurred by this uncertainty, the Worst Case Execution Time (WCET) $C_i > q_i$ is conventionally used in scheduling.

In non-preemptive systems, once a job starts, it must execute continuously on the processor till its completion. Please note a job in this case cannot be partitioned into mandatory and optional part and it must be executed as a whole. Thus, it does not have the convenience of preemptive scheduling that a task can be divided into small pieces to fill in processor idle time.

2.3.2 Schedulability of Non-Preemptive Real-Time Scheduling

The schedulability conditions for non-preemptive scheduling on uniprocessor are derived in [5] and shown as follows.

Theorem 1. *Let $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$, where $\tau_i = (p_i, C_i)$, be a set of periodic tasks sorted in non-decreasing order of period, i.e., if $i < j$, then $p_i \leq p_j$. If the following two conditions are satisfied, then $\mathcal{T}$ is schedulable.*

$$\sum_{i=1}^{n} \frac{C_i}{p_i} \leq 1 \tag{2.1}$$

$$C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor \cdot C_j \leq L, \ \ \forall i, 1 < i \leq n, \ \forall L, \ p_1 < L < p_i. \tag{2.2}$$

For jobs of a periodic task τ_i, their release time $r_{i,j}$ and deadline $d_{i,j}$ can be uniquely decided by the initial job release time $r_{i,1}$ and period p_i. Theorem 1 is necessary and sufficient for arbitrary initial release time, and thus only p_i needs to be considered here. Hence, a task is characterized by (p_i, C_i) instead of (r_i, C_i, d_i). In the sequel, release time r_i is still needed for considering specific tasks. For a specific set of release times, the theorem is sufficient but no longer necessary. Moreover, finding necessary and sufficient conditions for specific release times is an NP-hard problem. It is proved in [5] that if a set of tasks are schedulable according to Theorem 1, then non-preemptive Earliest Deadline First (EDF) scheduling [35] can always find a feasible schedule.

Condition (2.1) is to ensure the overall processor utilization does not exceed 1. This condition alone is the necessary and sufficient condition for a preemptive system to be schedulable, and the complexity of verifying this condition is constant. Condition (2.2) is to ensure that the workload

for any time interval of any task period is no greater than the length of the interval. The complexity of evaluating this condition is $O(p_n)$, where p_n is the largest task period, and therefore is pseudo-polynomial. Evidently, there is significant complexity escalation from preemptive to non-preemptive scheduling.

2.3.3 Imprecise Computing in Non-Preemptive Scheduling

Imprecise computing can be realized by either accuracy configurable circuits [30, 31] or algorithmic imprecision. Please note the imprecise computing is for only datapath or numerical computations, and not applicable to control flow or state computations. In a non-preemptive system, different task executions can be performed with different accuracy levels but the accuracy of one execution cannot be changed in the middle. We consider only one imprecision level in this work. Since our approaches are fundamentally enumerating discrete solution options, additional imprecision levels would not entail significant algorithm change.

The WCET b_i of task τ_i in imprecision mode must satisfy $b_i < C_i$, where C_i is the WCET for accurate mode. Meanwhile, an execution of job $\tau_{i,j}$ in imprecision mode produces a computing error $\epsilon_{i,j}$. Please note the error is single valued and would require composition of multiple errors from a multi-output system. By statistical analysis and pre-characterization, the mean error e_i for task τ_i can be obtained prior to scheduling. In independent error model, an error of job $\tau_{i,j}$ does not carry over to its subsequent job $\tau_{i,j+1}$ even if $\tau_{i,j+1}$ is in imprecision mode. In video rendering, for example, image processing error of one non-reference frame does not affect the next frame. In this scenario, one wishes to minimize the average error of a task. In the cumulative error model, an error of job $\tau_{i,j}$ propagates to its subsequent job. For example, in a target tracking computation, an error at one moment j is inherited in the computation of the next moment $j + 1$ and can be eliminated only when the computation at moment $j + 1$ is in accurate mode. To restrain such cumulative error, the number of consecutive jobs in imprecision mode must be limited.

2.4 Mixed-criticality (MC) Scheduling

2.4.1 Imprecise MC and Variable-Precision MC System Model

The system contains a set of independent sporadic tasks $\mathcal{T} = \{\tau_1, \tau_2, ...\}$ with implicit deadlines. Each task τ_i is composed by an infinite sequence of jobs $\{\tau_{i,1}, \tau_{i,2}, ...\}$, and is characterized by $(T_i, \chi_i, C_i^{LO}, C_i^{HI})$ where T_i is the minimal inter-arrival time that is the minimal time interval between two consecutive jobs of task τ_i, χ_i is the criticality level and C_i is the execution time. If job $\tau_{i,j}$ is released at time $r_{i,j}$, its deadline is $r_{i,j} + T_i$. In our work, we consider dual-criticality system model, which is justified in [11]. Then, the task set $\mathcal{T}$ is composed by two disjoint subsets of low-criticality tasks $\mathcal{T}_{lo}$ and high-criticality tasks $\mathcal{T}_{hi}$, and $\chi_i \in \{lo, hi\}$ indicates if task τ_i has low or high-criticality. Also, C_i^{LO} and C_i^{HI} indicate the execution time at low and high system criticality mode, respectively, and are based on the Worst Case Execution Time (WCET).

A such system starts with low-criticality mode. Once the execution time of a high criticality job $\tau_{i,j}$ exceeds its C_i^{LO}, the system is switched to high-criticality mode. If $\chi_i = hi$, $C_i^{HI} > C_i^{LO}$. In other words, high-criticality tasks are budgeted with more execution time in high-criticality mode. So far, the model has no fundamental difference from the conventional one [7, 12]. The key difference lies in the treatment of low-criticality tasks in high-criticality mode. We list related schemes below.

1. Conventional MC system [7, 8, 9, 10, 11, 12]: all low-criticality tasks are discarded in high-criticality mode. As such, C_i^{HI} for a low-criticality task is equivalent to zero.

2. Imprecise MC (IMC) system [18, 19]: a low-criticality task τ_i has a precise computing realization with execution time $\hat{C}_i$ and an imprecise implementation with execution time $\tilde{C}_i < \hat{C}_i$. The works of [18, 19] let $C_i^{LO} = \hat{C}_i$ and $C_i^{HI} = \tilde{C}_i$, which is a non-zero constant.

3. Variable-Precision MC (VPMC) system [20]: for a low-criticality task τ_i, $C_i^{LO} = \hat{C}_i$ and $C_i^{HI} \in \{\hat{C}_i, \tilde{C}_i\}$ corresponds to a decision variable. Since imprecise computing leads to errors, the objective of VPMC is to minimize computing errors, or maximize precise computing, for low-criticality tasks in high-criticality mode.

Please note VPMC and IMC follow the same schedulability condition for a scheduling method. When a schedulability condition is satisfied, there may be some processor utilization slack in high-criticality mode. IMC ignores the slack and applies imprecise computing for all low-criticality tasks. In contrast, VPMC attempts to utilize the slack in exchange for reducing imprecise computing and associated errors.

For each task τ_i, its utilizations in low and high-criticality mode are defined as

$$U_i^{LO} = \frac{C_i^{LO}}{T_i}, \text{ and } U_i^{HI} = \frac{C_i^{HI}}{T_i},$$

respectively. The total utilizations of all low-criticality tasks are

$$U_{lo}^{LO} = \sum_{\chi_i = lo} U_i^{LO} \text{ and } U_{lo}^{HI} = \sum_{\chi_i = lo} U_i^{HI}.$$

Similarly, the total utilizations for high-criticality tasks are defined as

$$U_{hi}^{LO} = \sum_{\chi_i = hi} U_i^{LO} \text{ and } U_{hi}^{HI} = \sum_{\chi_i = hi} U_i^{HI}.$$

Given m identical processors, a conventional scheduling is to decide when to execute each job on which processor. In our work, we consider preemptive scheduling, where a low-priority job can be preempted by a high-priority job during its execution.

2.4.2 IMC System Scheduling on Uniprocessor

The imprecise mixed-criticality (IMC) scheduling on uniprocessor [19] is based on EDF-VD (Earliest Deadline First with Virtual Deadlines) [8]. In EDF-VD, the implicit deadlines of all high-criticality tasks are scaled by a factor x. That is, for each high-criticality task τ_i, its virtual implicit deadline is $\hat{T}_i = x \cdot T_i$. The sufficient schedulability condition in low-criticality mode is given by the following theorem.

Theorem 2. *(Theorem 1 in [8]) If the following condition is satisfied, sporadic task set $\mathcal{T}$ is*

schedulable with EDF-VD method on uniprocessor in low-criticality mode.

$$U_{lo}^{LO} + \frac{U_{hi}^{LO}}{x} \leq 1 \tag{2.3}$$

This condition also tells that the scaling factor x can be decided by

$$x = \frac{U_{hi}^{LO}}{1 - U_{lo}^{LO}} \tag{2.4}$$

In high-criticality mode, the method of [19] continues to execute low-criticality tasks with imprecise computing, and derives the following sufficient schedulability condition.

Theorem 3. *(Theorem 2 in [19]) If the following condition is satisfied, sporadic task set $\mathcal{T}$ is schedulable with EDF-VD method on uniprocessor in high-criticality mode.*

$$xU_{lo}^{LO} + (1 - x)U_{lo}^{HI} + U_{hi}^{HI} \leq 1 \tag{2.5}$$

Theorem 4. *(Theorem 3 in [19]) Given a task set, if $\frac{U_{hi}^{LO}}{1-U_{lo}^{LO}} \leq \frac{1-(U_{hi}^{HI}+U_{lo}^{HI})}{U_{lo}^{LO}-U_{lo}^{HI}}$, where $U_{hi}^{HI}+U_{lo}^{HI} < 1$ and $U_{lo}^{LO} < 1$ and $U_{lo}^{LO} > U_{lo}^{HI}$, then this task set can be scheduled by EDF-VD with a deadline scaling factor x chosen in the following range*

$$x \in [\frac{U_{hi}^{LO}}{1 - U_{lo}^{LO}}, \frac{1 - (U_{hi}^{HI} + U_{lo}^{HI})}{U_{lo}^{LO} - U_{lo}^{HI}}] \tag{2.6}$$

2.4.3 Partitioned Scheduling on Multiprocessors

In this approach [10], $n = |\mathcal{T}|$ tasks are partitioned onto m unit-speed processors. After the partitioning, each task is never changed to another processor. As such, there is a fixed subset of tasks on each processor. Then, uniprocessor scheduling methods can be applied to each processor individually. In [10], a 2-phase task partitioning algorithm is described. In phase 1, high-criticality tasks are assigned to each processor one by one as long as the high-criticality utilization U_{hi}^{HI} for each processor does not exceed $\frac{3}{4}$. In phase 2, low-criticality tasks are further assigned to processors one by one following the condition that the low-criticality utilization $U_{lo}^{LO} + U_{hi}^{LO}$ is no greater than $\frac{3}{4}$. Alternatively, one can follow a different order of task assignment, which is

applied with the same schedulability check. This order is obtained by sorting with non-increasing utilization (U_i^{LO} for low-critical tasks and U_i^{HI} for high-critical tasks) regardless task criticality. It is shown in [36] that this alternative order never sacrifices overall schedulability and sometimes improves schedulability.

2.4.4 Global fpEDF Scheduling on Multiprocessors

The method of fpEDF (fixed-priority EDF) [37] is a state-of-art global scheduling approach for multiprocessor in conventional real-time systems without mixed-criticality. Fixed-priority means the priority of one job can not be changed during execution. For a task set $\bigcup_{\tau_i \in \mathcal{T}} (T_i, C_i)$ to be scheduled on m identical processors, fpEDF first chooses a subset $\mathcal{T}_{hp} \subset \mathcal{T}$ of at most $m-1$ tasks, each with utilization greater than $\frac{1}{2}$, and assigns them on m_{hp} processors with the highest priority. The priorities of remaining tasks $\mathcal{T}_{EDF} \subset \mathcal{T}$ are lower and scheduled according to EDF (Earliest Deadline First) principle on the other $m_{EDF} = m - m_{hp}$ processors. The schedulability condition for fpEDF is given in Lemma 1([37]).

Lemma 1. *Consider a task set $\bigcup_{\tau_i \in \mathcal{T}} (T_i, C_i)$ to be scheduled on m identical processors. Let U_{EDF}^{total} be the total utilization of the tasks in $\mathcal{T}_{EDF}$, and U_{EDF}^{max} be the maximum utilization of tasks in $\mathcal{T}_{EDF}$. If $U_{EDF}^{total} \leq m_{EDF} - (m_{EDF} - 1) \cdot U_{EDF}^{max}$ is satisfied, this task set $\mathcal{T}$ is schedulable by fpEDF method.*

2.4.5 Global Scheduling by fpEDF-VD on Multiprocessors

fpEDF-Virtual Deadline (fpEDF-VD) [9] is an extension of fpEDF to mixed-criticality systems. Virtual deadlines are enforced for high-criticality tasks. Each high-criticality task τ_j is mapped to $(\hat{T}_j, C_j^{LO})$ in low-criticality mode, where $\hat{T}_j = x \cdot T_j (0 < x < 1)$ is the virtual deadline that is enforced in both offline schedulability test and online execution. Each low-criticality task τ_i is mapped to a regular implicit deadline task (T_i, C_i^{LO}) in low criticality mode, and all low-criticality tasks are dropped in high-criticality mode. The schedulability conditions for fpEDF-VD are as follows.

- Task set $(\bigcup_{\tau_i \in \mathcal{T}_{lo}} (T_i, C_i^{LO})) \bigcup (\bigcup_{\tau_j \in \mathcal{T}_{hi}} (x \cdot T_j, C_j^{LO}))$ is schedulable on m processors in low-

criticality mode according to Lemma 1.

- Task set $\bigcup_{\tau_j \in \mathcal{T}_{hi}} ((1-x) \cdot T_j, C_j^{HI})$ is schedulable on m processors in high-criticality mode according to Lemma 1.

For a high-criticality task $\tau_j \in \mathcal{T}_{hi}$ in high-criticality mode, its implicit deadline $(1-x) \cdot T_j$ is used in the offline schedulability check. However, only its original deadline T_j needs to be enforced during online execution. By default, virtual deadline of a high-criticality task $\tau_j \in \mathcal{T}_{hi}$ refers to $x \cdot T_j$.

The schedulability condition in high-criticality mode leads to the following important conclusion, which is heavily used in our work.

Lemma 2. *([9]) If a mixed-criticality task set $\mathcal{T}$ is schedulable by fpEDF-VD, each of its high-criticality job $\tau_{j,k}$ in high-criticality mode can start from its virtual deadline $\hat{d}_{j,k}$ and guarantee to finish by actual deadline $d_{j,k}$ with execution time C_j^{HI} following fpEDF-VD scheduling.*

2.4.6 MC-DP-Fair Scheduling on Multiprocessors

DP (Deadline Partition) Fair [38] is a scheduling method for multiprocessor real-time system without mixed-criticality. Each task τ_i is defined with a density $\delta_i = \frac{C_i}{T_i}$, where C_i is its WCET and T_i is its minimal inter-arrival time. Time is divided into slices by deadline partitions, each of which is a distinct job release time or deadline. A time slice is a time interval between two consecutive partitions. If the length of a time slice is l, DP-Fair executes $\delta_i \cdot l$ amount of task τ_i in this slice. The schedulability condition of DP-Fair is specified as follows.

Theorem 5. *(Lemma 14 in [25]) A non-MC task set $\mathcal{T}$ is schedulable under DP-Fair iff $\sum_{\tau_i \in \mathcal{T}} \delta_i \leq m$, where m is the number of processors.*

MC-DP-Fair [25] is an extension of DP-Fair for mixed-criticality systems. A main change is that each task τ_i is assigned a virtual deadline $0 < V_i \leq T_i$. Let Γ be the earliest deadline partition after a system is switched to high-criticality mode. The virtual deadlines and the original deadlines are enforced before and after Γ, respectively. By carefully choosing the values of virtual deadlines,

MC-DP-Fair has schedulability equivalent to MC-Fluid [25], which is the theoretically optimal approach [25].

3. RELATED WORK [1]

3.1 Preemptive Real-time Scheduling Considering Imprecise Computing

There are some related works of preemptive real-time scheduling considering imprecise computing [3, 39, 40, 41, 2, 42, 4, 43] [2]. In these works, a task is partitioned into a mandatory part and an optional part. Only completing the mandatory part implies imprecision. For the computation to be accurate, the optional part must be completed after the mandatory part. Since preemption is allowed, the mandatory and optional parts can be executed separately with a precedence constraint. The optional part can be treated as a stand-alone task with soft deadline constraint. Please note a mandatory (or optional) part can be further divided due to preemption in these works. These techniques are not applicable to non-preemptive systems, where a task cannot be divided and executed disjointly.

In [44], soft real-time scheduling for multiprocessor system is studied with consideration of imprecise computing. It handles dynamic voltage and frequency scaling (DVFS) and thermal constraint at the same time. The work of [45] schedules tasks for parallel computing considering imprecise computing and DVFS. Every imprecise computing result is evaluated and the corresponding task is re-executed in accurate mode if the imprecision errors are too large. Evidently, such re-execution is not friendly to real-time systems.

3.2 Non-Preemptive Hard Real-Time Scheduling

The prior research on non-preemptive real-time scheduling is limited, at least partly due to its hardness. One classic work is [5], where the schedulability conditions are derived for unipro-

[1] Reprinted with permission from "Using imprecise computing for improved non-preemptive real-time scheduling " by Lin Huang, Youmeng Li, Sachin S.Sapatnekar, Jiang Hu, 2018. *Proceedings of Design Automation Conference (DAC)*, Page 1-6 , ©2018 IEEE, "Graceful degradation of low-criticality tasks in multiprocessor dual-criticality systems " by Lin Huang, I-Hong Hou, Sachin S.Sapatnekar, Jiang Hu, 2018. *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*, Page 159-169 , ©2018 ACM and "Improving QoS for global dual-criticality scheduling on multiprocessors " by Lin Huang, I-Hong Hou, Sachin S.Sapatnekar, Jiang Hu, 2019. *Proceedings of the International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Page 1-11, ©2019 IEEE.

[2] In some literature, imprecise computing is referred to as approximate computing or inexact computing.

cessors. In addition, it proves NP-hardness of several related problems. Its schedulability result is a critical foundation for our work. Non-preemptive scheduling for multiprocessors is studied in [46, 47]. In [48], the gap between preemptive and non-preemptive scheduling, in term of the processor speedup for a non-preemptive system to achieve the same schedulability as preemptive system, is studied.

3.3 Real-Time Scheduling and Dynamic Voltage and Frequency Scaling (DVFS)

There is one category of previous works that are seemingly different from ours but actually related. That is simultaneous real-time scheduling and DVFS [49, 50, 6, 51]. The voltage/frequency scaling in these works is limited to a range where no timing or computing error occurs. If the default computing is at low voltage/frequency or accurate mode, then both high voltage/frequency and imprecise computing provide another option for fast task execution, which facilitates improved schedulability. A pioneer work on simultaneous real-time scheduling and DVFS is [49], which is restricted to preemptive cases. Non-preemptive scheduling with DVFS is addressed in [50, 6, 51]. The works of [50, 6] study how to switch among different V/F levels in the middle of task execution. By contrast, accuracy change in imprecise computing is allowed only among different task executions, not during the execution of a task. In [51], it is assumed that task execution order is given and voltage/frequency is continuously tunable. Such assumptions are restrictive and not very practical. Despite the relations, these previous works cannot be easily extended to solve our problem.

3.4 Mixed-criticality (MC) Scheduling

The MC model was first described in [7]. Early works on MC scheduling are mostly for uniprocessors. One such work is EDF-VD [8], which extends the Earliest Deadline First scheduling with Virtual Deadlines such that time resource is reserved for meeting deadlines during low-criticality mode, high-criticality mode and the transition time at the beginning of high-criticality mode. This method is designed for the classic MC model [7] where all low-criticality tasks are dropped in high-criticality mode. There are only a few studies on imprecise computing of low-criticality

tasks in mixed-criticality systems. Most of them are built upon scheduling methods derived for the conventional MC model. One early work is [17], which is an extension of adaptive mixed-criticality scheduling [52]. Since this work also covers the other two approaches, reducing the priorities or increasing the periods of low-criticality tasks, its discussion on imprecise computing is restricted to response time analysis. Later, a mixed-criticality scheduling work dedicated to the imprecise computing model is introduced in [18]. This is a uniprocessor scheduling based on the fluid model [25, 26] and the algorithm is proved to have speedup factor of $\frac{4}{3}$. Another uniprocessor scheduling considering imprecise computing for low criticality tasks is [19], which is an extension to EDF-VD (Virtual Deadline) scheduling [8]. It derives a sufficient condition and speedup factor for allowing imprecise computing of low-criticality tasks under EDF-VD. All these works [17, 18, 19] execute low-criticality tasks with imprecise computing in high-criticality mode. By contrast, the latest work [20] allows some low-criticality tasks to be executed with full precision in high-criticality mode. It formulates an integer linear programming to decide which low-criticality tasks can continue with precise computing. Like [18], this work is also based on the fluid model and is applied with multiprocessors. The goal of [20] is very close to our work - maximizing precise computing of low-criticality tasks on multiprocessors. The key difference is that [20] is more focused on theoretical conditions while our work emphasizes more on practical realizations. Moreover, we study IMC and VPMC for other well-known multiprocessor scheduling techniques that have not been investigated before. Although fluid-based scheduling is theoretically very competitive, it cannot be directly implemented on hardware due to its restrictive assumption. Compared to [20], whose validation is by only software simulation, our work contains Linux prototyping validation that considers various overheads.

The limitations of the classic MC model have been identified [12] and several recent works are developed for corresponding mitigation. An online adjustment technique [21] is developed to reduce the number of low-criticality tasks that are dropped in high-criticality mode. In [22], multiple intermediate-criticality levels are introduced between low-criticality mode and high-criticality mode such that there is no need to simultaneously switch all high-criticality tasks to high-criticality

mode. The work of [53] enables return to low-criticality mode from high-criticality mode. Another approach is to continue low-criticality tasks in high-criticality mode with graceful degradation, such as imprecise computing [19]. All these techniques are developed for uniprocessors. For multiprocessors, an EDF-VD-based global scheduling method is proposed in [9] using the classic MC model. There are very few previous works on multiprocessor scheduling for addressing the limitations of the classic MC model. The work of [24] is a partitioning-based scheduling that aims to alleviate some limitations of the classic MC model. It first partitions tasks to processors in an effort for load balancing. Then, uniprocessor scheduling is performed for tasks assigned to the same processor. Tasks of the same processor are further divided into multiple groups, each of which contains only one high-criticality task so as to achieve isolation among high-criticality tasks. Processor time is allocated to different groups, where scheduling is performed separately. In high-criticality mode, low-criticality jobs are executed opportunistically without guarantee. As such, the limitation of dropping all low-criticality tasks is not well solved. It has an intermediate mode before entering high-criticality mode. However, all low-criticality tasks are suspended in this mode. In [54], new protocols are proposed for systems to return from high-criticality mode to low-criticality mode. A fluid-based scheduling [20] is introduced to allow execution of low-criticality tasks in high-criticality mode for multiprocessors. However, it follows the conservative model where all high-criticality tasks simultaneously enter high-criticality mode.

4. NON-PREEMPTIVE REAL-TIME SCHEDULING [1]

We work on using imprecise computing for non-preemptive real-time scheduling, and propose several heuristic algorithms for scheduling periodic tasks and considering imprecision with independent or cumulative errors.

4.1 Motivation Example

Table 4.1: An example of non-preemptive real-time scheduling.

Task	Period p	Accurate WCET C	Apprx WCET b	Actual ET q
τ_1	12	6	4	4, 5, 5
τ_2	18	7	4	6, 5
τ_3	36	10	6	8

A small example is described here to show the motivation of considering imprecise computing in real-time scheduling. The details of this example are listed in Table 4.1. If only accurate computing is considered, the schedulability check fails as

$$\sum_{i=1}^{3} \frac{C_i}{p_i} = \frac{6}{12} + \frac{7}{18} + \frac{10}{36} = \frac{42}{36} > 1.$$

However, the check can pass if one uses the WCET of imprecise mode as

$$\sum_{i=1}^{3} \frac{b_i}{p_i} = \frac{4}{12} + \frac{4}{18} + \frac{6}{36} = \frac{26}{36} \leq 1.$$

The same observation can be obtained for condition (2.2). Thus, imprecise computing allows aggressive scheduling without worrying deadline violation. By using our proposed techniques,

[1] Reprinted with permission from "Using imprecise computing for improved non-preemptive real-time scheduling " by Lin Huang, Youmeng Li, Sachin S.Sapatnekar, Jiang Hu, 2018. *Proceedings of Design Automation Conference (DAC)*, Page 1-6 , ©2018 IEEE.

where a task execution can select accurate mode even though the schedulability check is based on imprecision mode, one can reach a feasible scheduling with all jobs in accurate mode as shown in Figure 4.1. This example indicates that imprecise computing can reduce the pessimism of the conventional WCET model. At the same time, errors from imprecision can be controlled if imprecision mode is applied in a prudent manner.

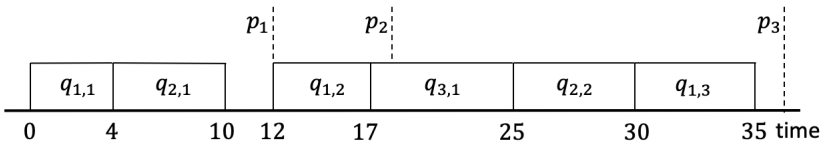

Figure 4.1: An example that is actually schedulable in accurate mode but fails schedulability check according to the WCET model.

4.2 Online Scheduling of Tasks with Independent Errors

In this section, an online algorithm is introduced for solving periodic tasks where imprecision errors are independent. The problem formulation is given below.

Problem 1. *Given a set of periodic tasks $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$, decide if each job $\tau_{i,j}$ is executed in accurate or imprecise mode, and its start time $s_{i,j}$ such that $r_{i,j} \leq s_{i,j}$, $s_{i,j} + q_{i,j} \leq r_{i,j} + p_i$ and the total error among all jobs in a hyper-period $\sum_{\forall i,j \in P} \epsilon_{i,j}$ is minimized.*

Please note hyper-period P is the least common multiple of all task periods. $q_{i,j}$ is the actual execution time regardless the accuracy level of job $\tau_{i,j}$ execution, and $\epsilon_{i,j}$ is the actual error when $\tau_{i,j}$ is executed in imprecision mode. Our algorithm designed for this formulation of periodic tasks can be easily adapted to sporadic tasks, because the online nature of this algorithm does not assume the prior knowledge of task release time and deadline, and the schedulability conditions employed in this algorithm are applicable for sporadic tasks.

If a given set of tasks passes its schedulability check using Theorem 1 based on imprecision mode WCET, our algorithm can guarantee that there is no deadline violation. A main effort of the algorithm is to increase the use of accurate mode in order to minimize imprecision errors. Our algorithm is EDF with Explicit Slack Reclamation, where the accuracy selection is based on explicit slack reclamation with constant complexity. When a job is to start, the algorithm checks if there is enough slack available for this job to be executed in accurate mode. There are three kinds of slacks: (1) individual slack; (2) idle-time slack; and (3) inter-job slack.

An individual slack $\psi_{i,j}$ is intrinsically available to every job $\tau_{i,j}$ and is estimated with the initial schedulability check before any job is started. A scaling factor γ is associated with every condition in Theorem 1 and their values can be found by solving the following equations:

$$\gamma \sum_{i=1}^{n} \frac{b_i}{p_i} = 1$$

$$\gamma_i^L \cdot (b_i + \sum_{j=1}^{i-1} \Big\lfloor \frac{L-1}{p_j} \Big\rfloor \cdot b_j) = L, \forall i, 1 < i \leq n, \forall L, p_1 < L < p_i.$$

We define γ_{min} to be the minimum γ value, i.e., $\gamma_{min} = \min_{\forall i, \forall L}(\gamma, \gamma_i^L, ...)$. If the set of tasks are schedulable when all of their jobs are in imprecision mode, then $\gamma_{min} \geq 1$. Hence, the individual slack $\psi_{i,j} = (\gamma_{min} - 1) \cdot b_i$. The individual slacks are computed only once at the beginning and their values can be used repeatedly throughout the online scheduling.

When a job $\tau_{i,j}$ is being scheduled online, and its nominal finish time $f_{i,j}$ is less than the minimum between its deadline $d_{i,j}$ and the release time r_{next} of its next job, then the idle time slack is $\psi_{i,j}^{idle} = \min(d_{i,j}, r_{next}) - f_{i,j}$. The nominal finish time $f_{i,j}$ for a job $\tau_{i,j}$ is the finish time assuming no slack reclamation is conducted and can be estimated as $current\_time + b_i + \psi_{i,j}^{k,l}$, where $\psi_{i,j}^{k,l}$ is inter-job slack introduced as follows.

An inter-job slack $\psi_{i,j}^{k,l}$ is the slack generated by job $\tau_{k,l}$ due to its early completion and passed to its next job $\tau_{i,j}$. This concept is elaborated through the example in Figure 4.2, where job 1 finishes at f_1' which is earlier than its nominal finish time f_1 and the next job according to EDF (job 2 here) has release time earlier than f_1. Then, the processor time from f_1' to f_1 is a slack time

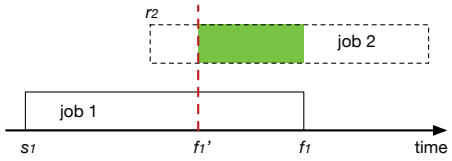

Figure 4.2: If job 1 actually finishes at f_1', which is earlier than its nominal finish time f_1, and job 2 is to be executed next, job 1 provides inter-job slack to job 2 as in the green region. The release times for job 2 is $r2$.

that can be applied to job 2, which is the green region in Figure 4.2. In general, the inter-job slack produced by $\tau_{k,l}$ and passed to $\tau_{i,j}$ is defined by

$$\psi_{i,j}^{k,l} = \max(f_{k,l} - \max(r_{i,j}, f'_{k,l}), 0) \tag{4.1}$$

Every job intrinsically has individual slack, which can be zero. It can be consumed by choosing accurate mode for this job. Even if a job is executed in imprecision mode, its individual slack expires when the job is completed. However, an individual slack can be recycled as inter-job slack. An idle-time slack generates opportunistically, and expires if it is not consumed. An inter-job slack $\psi_{i,j}^{k,l}$ can be consumed by executing job $\tau_{i,j}$ in accurate mode. If it is not consumed, it can assist $\tau_{i,j}$ to generate new inter-job slack or expires.

In the EDF with explicit slack reclamation algorithm, we use the following slack based check. When a job $\tau_{i,j}$ is being scheduled, its total slack is evaluated by

$$\psi_{i,j}^{total} = \psi_{i,j} + \psi_{i,j}^{idle} + \psi_{i,j}^{k,l}.$$

If $\psi_{i,j}^{total} \geq C_i - b_i$, this job is executed in accurate mode and otherwise in imprecision mode.

Proposition 1. *If a set of tasks are schedulable according to the WCET of imprecise mode, the EDF scheduling with explicit slack reclamation method guarantees that there is no deadline violation.*

Proof. Even with the slack reclamation, a job finish-time is never greater than that when all jobs are executed in the WCET of imprecise mode. Thus, it would not violate its deadline and this is

true for all jobs. □

The pseudo code for this algorithm is provided in Algorithm 1.

Algorithm 1: EDF scheduling with slack reclamation.

```
Check schedulability using b_i for all unexecuted jobs;
if schedulable then
    while ∃ unexecuted job do
        τ_{k,l} ← the job just completed;
        τ_{i,j} ← the next job according to EDF;
        f_{i,j} ← current_time + b_i + ψ^{k,l}_{i,j};
        r_next ← release time of the next job after τ_{i,j};
        ψ^{idle}_{i,j} ← min(d_{i,j}, r_next) − f_{i,j};
        ψ^{total}_{i,j} ← ψ_{i,j} + ψ^{idle}_{i,j} + ψ^{k,l}_{i,j};
        if ψ^{total}_{i,j} ≥ C_i − b_i then
            Start τ_{i,j} in accurate mode;
        else
            Start τ_{i,j} in imprecise mode;
        end
        Wait till completion of τ_{i,j};
        Estimate inter-job slack from τ_{i,j} to its next job according to Equation (4.1);
    end
end
```

4.3 Collaborative Scheduling of Periodic Tasks with Independent Errors

This section introduces three collaborative methods for solving Problem 1. Each of these methods is composed by an offline scheduling part and an online adjustment part. Since offline scheduling relies more on prior knowledge of the tasks, these methods are mostly for periodic tasks, whose release times are more predictable than sporadic tasks.

4.3.1 Offline ILP and Online Adjustment

The offline scheduling and accuracy selection is conducted for one hyper-period using integer linear programming (ILP). During task executions, some jobs are opportunistically adjusted from imprecise to accurate mode. The adjustment is performed with reference to the ILP result and thus

no schedulability check as Theorem 1 is needed. The ILP result provides an upper bound guarantee to imprecision errors.

A decision variable $y_{i,j}$ is defined to be 1 if $\tau_{i,j}$ is executed in imprecision mode and 0 otherwise. The offline scheduling finish time for job $\tau_{i,j}$ is denoted as $\hat{f}_{i,j}$. An indicator function $u_{i,j}(t)$ is equal to 1 if time t is in-between the start and finish time of $\tau_{i,j}$, and 0 otherwise. The ILP formulation is as follows.

$$
\begin{aligned}
\underset{y}{\text{minimize}} \quad & \sum_{\forall \tau_{i,j} | [r_{i,j}, d_{i,j}] \subseteq [0,P]} e_i \cdot y_{i,j} \\
\text{subject to} \quad & s_{i,j} \geq r_{i,j}, && \forall \tau_{i,j} | [r_{i,j}, d_{i,j}] \subseteq [0, P] \\
& \hat{f}_{i,j} = s_{i,j} + C_i + (b_i - C_i) \cdot y_{i,j}, && \forall \tau_{i,j} | [r_{i,j}, d_{i,j}] \subseteq [0, P] \\
& \hat{f}_{i,j} \leq r_{i,j} + p_i, && \forall \tau_{i,j} | [r_{i,j}, d_{i,j}] \subseteq [0, P] \\
& \sum_{\forall \tau_{i,j} | [r_{i,j}, d_{i,j}] \subseteq [0,P]} u_{i,j}(t) \leq 1 && \forall t, 0 < t \leq P \\
& s_{i,j} \in \mathbb{Z}_{\geq 0}, \;\; y_{i,j} \in \{0, 1\} && \forall \tau_{i,j} | [r_{i,j}, d_{i,j}] \subseteq [0, P]
\end{aligned}
$$

Where $s_{i,j}$ is the start time for job $\tau_{i,j}$, b_i is the constant WCET for task τ_i in imprecision mode and P is the hyper-period.

In the online adjustment, if a job $\tau_{i,j}$ finishes earlier than the offline finish time $\hat{f}_{i,j}$ specified by ILP, the next job can start immediately without waiting till its start time specified by ILP. The order of job executions is fixed and conforms to the ILP result. When a job $\tau_{i,j}$ is able to start at current time t_{cur} and $y_{i,j} = 1$, it is executed in accurate mode if and only if $t_{cur} + C_i \leq \hat{f}_{i,j}$, which is the finish time by ILP. Thus, the complexity of this online adjustment is constant.

4.3.2 ILP with Post-Processing and Online Adjustment

The ILP method described in Section 4.3.1 can guarantee the optimal solution according to the WCET estimation, but the actual execution time is usually shorter than WCET. We propose a post-processing to the ILP such that online adjustment may have more opportunities to improve the result. Meanwhile, the ILP constraints and optimality are not affected. The offline post-processing is based on three observations.

- For a job $\tau_{i,j}$, if the processor is idle after $\hat{f}_{i,j}$, which is obtained from ILP, we postpone its $s_{i,j}$ as much as possible without missing its deadline or conflicting with the execution of its next job. This is because the online accuracy adjustment is based on the condition of $t_{cur} + C_i \leq \hat{f}_{i,j}$ and increased $s_{i,j}$ as well as $\hat{f}_{i,j}$ would improve the chance of changing a job from imprecision to accurate mode. The increased offline $s_{i,j}$ does not affect the actual start time at runtime, when a job always starts immediately upon the availability of processor without waiting for $s_{i,j}$.

- If ILP schedules job $\tau_{k,l}$ to start immediately after $\tau_{i,j}$ and assigns both the jobs with the same accuracy level, we may swap their execution order such that the job with earlier release time starts earlier. This is based on the observation that a job with early release time has relatively large chance to reclaim the slack generated from prior job executions.

- If ILP schedules an accurate job to be executed right after an imprecise job, we swap the order of the two jobs subject to release time and deadline constraints. If an imprecise job is scheduled to be executed later, it may have more chance of reclaiming slacks from prior job executions. For example, in Figure 4.3, the imprecise job $\tau_{1,4}$ may reclaim slacks from $\tau_{3,2}$ after the swapping.

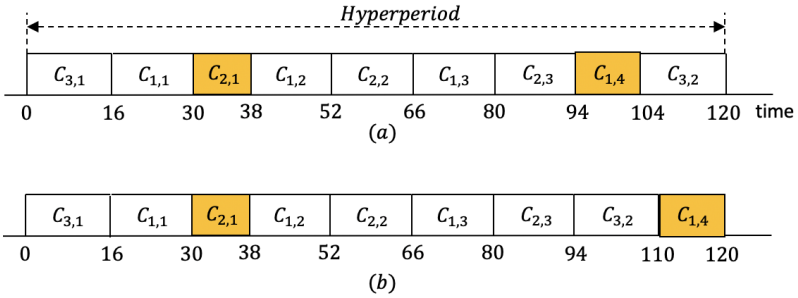

Figure 4.3: (a) ILP scheduling; (b) swapping imprecise job $\tau_{1,4}$ to be executed later in post-processing.

Algorithm 2: Post processing of ILP result.

Input : $\mathcal{T}^{ILP} = \{\Gamma^1, \Gamma^2 ...\}$ set of jobs scheduled by ILP for P

$Action \leftarrow false$;
do
  for *each job* $\Gamma^i \in \mathcal{T}^{ILP}$ **do**
    $Slack \leftarrow \min(d(\Gamma^i), s(\Gamma^{i+1})) - \hat{f}(\Gamma^i)$;
    if $Slack > 0$ **then**
      $s(\Gamma^i) \leftarrow s(\Gamma^i) + Slack$;
      $Action \leftarrow true$;
    end
    if $y(\Gamma^i) == y(\Gamma^{i+1})$ **then**
      if $r(\Gamma^{i+1}) < r(\Gamma^i)$ *and* $\hat{f}(\Gamma^{i+1}) \leq d(\Gamma^i)$ **then**
        swap Γ^i and Γ^{i+1};
        $Action \leftarrow true$;
      end
    else if $y(\Gamma^i) == 1$ *and* $y(\Gamma^{i+1}) == 0$ **then**
      if $r(\Gamma^{i+1}) \leq s(\Gamma^i)$ *and* $\hat{f}(\Gamma^{i+1}) \leq d(\Gamma^i)$ **then**
        swap Γ^i and Γ^{i+1};
        $Action \leftarrow true$;
      end
  end
while $Action == true$;

The pseudo code for the post processing is provided in Algorithm 2. For a job Γ^i, the superscript i is the execution order index decided by ILP. Thus, job Γ^{i+1} is scheduled to be executed right after Γ^i. If Γ^i is the last job in hyper-period P, then Γ^{i+1} can be treated as a dummy job with execution time 0 and is executed at the last moment of P. In this pseudo code, we use $r(\Gamma^i), s(\Gamma^i), \hat{f}(\Gamma^i), d(\Gamma^i)$ to denote the release time, start time, finish time and deadline for task Γ^i, respectively. Notation $y(\Gamma^i)$ is the result from ILP indicating accuracy level. If its value is 1, imprecision mode is selected for job Γ^i. Since all changes in the post processing are monotone and cannot be reversed, convergence is guaranteed. After the post-processing, the online adjustment part is the same as Section 4.3.1.

4.3.3 Flipped EDF and Online Adjustment

We propose another offline scheduling algorithm with all jobs being in imprecision mode. When it works together with online adjustment, this offline scheduling achieves comparable or

even better results compared to the ILP-based collaborative methods. For a hyper-period P, this algorithm schedules jobs from the last moment of P and proceeds backward to time 0, the starting point of hyper-period. Among all unscheduled jobs, it always chooses the job with the latest release time to schedule first. It selects imprecision mode for every job and schedules a job as late as possible without deadline violation or conflicting with jobs that have already been scheduled. One can think of this algorithm as EDF being performed in a flipped manner, where time axis is reversed and the release time and deadline of each job are exchanged. If the original EDF is like as-soon-as-possible scheduling, our flipped EDF is like as-late-as-possible scheduling.

According to [5], EDF can guarantee to find a feasible solution if the schedulability test of Theorem 1 is passed. Since the flipped EDF is fundamentally equivalent to EDF, it enjoys the same guarantee of finding feasible solution. After the offline flipped EDF, the online adjustment is performed in the same way as Section 4.3.1.

4.4 Scheduling Periodic Tasks with Cumulative Errors in Imprecision

When errors are cumulative, an error at one job $\tau_{i,j}$ can carry over to its next job $\tau_{i,j+1}$ and so on if the jobs of task τ_i are in imprecision mode contiguously. The errors can be cleared out only when at least one job execution is in accurate mode. As such, users wish to avoid that the jobs of a task continuously operate in imprecision mode. Hence, we attempt to constrain the number of consecutive imprecise job executions for each task like [2] and the problem formulation is as follows.

Problem 2. *Given a set of periodic tasks $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$, decide if each job $\tau_{i,j}$ is executed in accurate or imprecision mode, and its start time $s_{i,j}$ such that $r_{i,j} \leq s_{i,j}$, $s_{i,j} + q_{i,j} \leq r_{i,j} + p_i$ and the number of consecutive jobs in imprecision mode for each task τ_i is no greater than constraint B_i.*

We propose an online heuristic and an offline dynamic programming algorithm. For a problem with only constraints and without objective function, an offline-online collaborative approach is not helpful. If an offline algorithm can find feasible solution, then any online adjustment would

not improve the feasibility according to the problem formulation.

4.4.1 Online Heuristic

This online heuristic schedules jobs using EDF. In addition, it decides if to execute a job in accurate or imprecision mode. When a job $\tau_{i,j}$ is scheduled to start, there are four scenarios for deciding its accuracy mode. In the first scenario, there is error constraint violation if $\tau_{i,j}$ is executed in imprecision mode while schedulability check passes for accurate mode. Then, accurate mode is selected for this scenario. The second scenario is mirror case to the first one, where an imprecise execution would not violate the error constraint but accurate execution fails schedulability check. Imprecision mode is selected for this scenario. The third scenario is a difficult one, where both imprecision mode would violate error constraint and accurate mode does not satisfy schedulability conditions. In this scenario, we choose imprecision mode such that the theoretical guarantee of no deadline violation is still fulfilled.

The fourth scenario seems easy but actually can affect subsequent selections. This is when imprecision mode would not violate the error constraint and accurate mode passes schedulability check. We define and compare error slack and latency slack to tell if choose accurate mode for less error or imprecision mode for less risk of deadline violation, which can eventually become error constraint violation according to the third scenario. The error slack is defined as

$$ErrorSlack_i = \frac{B_i - \phi_i}{B_i},$$

where ϕ_i is the number of consecutive imprecise executions of jobs of task τ_i immediately before $\tau_{i,j}$. The latency slack is defined as

$$LatencySlack_{i,j} = \frac{d_{i,j} - s_{i,j} - C_i}{p_i}. \tag{4.2}$$

Please note the error slack is normalized within $(0, 1]$ while the latency slack is normalized within

$[0, 1]$. Then, we check the ratio

$$\rho = \frac{LatencySlack_{i,j}}{ErrorSlack_i}.$$

If $\rho < \theta$, where θ is a user specified threshold, job $\tau_{i,j}$ is executed in imprecision mode as the latency slack is tighter. The schedulability check here can be realized using the explicit slack reclamation as in Section 4.2. The implementation of slack reclamation can be integrated with the latency slack estimation.

4.4.2 Offline Dynamic Programming

The offline dynamic programming is a traversal of all jobs in a super period following the EDF principle. A super period is the minimum consecutive set of hyper-periods that can cover all scenarios of errors satisfying constraint B_i for all tasks. When a job is encountered in the traversal, both its accurate and imprecise executions are considered as two candidate solutions. Thus, the dynamic programming is an enumeration of different precision options in a decision tree. However, we do not need to examine the entire tree (or solution space). If a candidate solution has either deadline or error constraint violation, it is pruned without being propagated to consider with the next job. Please note that different precision choices may lead to different job execution orders, although all of them follow the principle of EDF. To further improve the computing efficiency, we propose two other solution pruning techniques.

The first pruning technique is based on solution dominance. Consider a set of candidate solutions $\mathcal{S}^k = \{S_1^k, S_2^k, ...\}$ for the same k jobs that have been processed in the traversal so far. A solution $S_i^k \in \mathcal{S}^k$ is characterized by $n+1$ tuple $(f_i^k, \lambda_{i,1}^k, \lambda_{i,2}^k, ..., \lambda_{i,n}^k)$, where f_i^k is the finish time of all the k jobs and $\lambda_{i,l}^k$ is the cumulative error for task τ_l. If $f_i^k = f_j^k$ and $\lambda_{i,l}^k \geq \lambda_{j,l}^k, 1 \leq l \leq n$, then solution S_i^k is dominated by S_j^k and can be pruned without being further propagated.

The second pruning technique is based on processor utilization. For a solution S_i^k, we estimate the best case processor utilization for the unprocessed jobs in the remaining time of the super period. By “best case", we mean the allowable error budget is maximally used by the unprocessed jobs. If this utilization exceeds 1, the corresponding solution is pruned without further propagation.

4.5 Experiment Results

Table 4.2: Testcase characteristics and schedulability.

Tesecases	#tasks	Utilization Accurate	#jobs/P	Schedulability	
				Accurate	Imprecise
Rnd1	3	1.13	9	No	Yes
Rnd2	3	1.88	9	No	No
Rnd3	5	1.93	15	No	Yes
Rnd4	3	1.6	9	No	Yes
Rnd5	3	0.45	17	No	Yes
Rnd6	7	3.8	22	No	Yes
Rnd7	10	4.43	38	No	Yes
Rnd8	12	2.91	60	No	Yes
Rnd9	15	1.93	24	No	Yes
Rnd10	17	4.99	126	No	Yes
Rnd11	20	3.57	105	No	Yes
Rnd12	22	5.47	130	No	Yes
Rnd13	25	7.12	163	No	Yes
IDCT	5	1.02	35	No	No

Table 4.3: Simulation results for periodic tasks with independent errors (error standard deviation is σ).

Test cases	EDF-Accurate	EDF-Imprecise		EDF+ESR(I)		ILP+OA		ILP+post+OA		Flipped EDF	
	Deadline violations	Mean error	σ	Mean error	σ	Mean error	σ	Mean error	σ	Mean error	σ
Rnd1	0	2.59	1.82	1.15	1.21	0.40	0.72	0.22	0.56	0	0
Rnd2	55%	2.16	1.64	1.51	1.39	1.19	1.19	0.67	0.86	0.72	0.93
Rnd3	29%	51.85	39.64	27.65	34.55	32.03	35.65	23.79	33.61	19.86	32.42
Rnd4	38%	3.60	3.44	2.77	3.35	1.10	1.08	0.94	0.99	2.06	3.07
Rnd5	0	0.62	0.31	0.22	0.19	0.29	0.22	0	0	0	0
Rnd6	27%	2.58	1.17	2.25	1.07	2.39	1.11	2.10	1.05	2.04	1.04
Rnd7	29%	82.03	27.25	68.87	25.01	65.26	24.25	62.78	23.49	55.83	22.13
Rnd8	43%	82.14	21.79	58.67	18.21	52.67	17.37	43.25	15.56	41.53	14.96
Rnd9	4%	2.59	1.12	1.23	0.77	0.44	0.46	0.32	0.39	0.32	0.39
Rnd10	28%	20.08	3.74	15.24	3.26	13.60	3.07	11.99	2.85	12.36	2.93
Rnd11	31%	5.09	1.03	3.93	0.90	3.09	0.83	2.51	0.74	2.40	0.71
Rnd12	24%	10.09	1.81	8.03	1.62	6.84	1.51	6.79	1.50	6.85	1.52
Rnd13	18%	87.25	27.24	70.10	24.78	42.95	19.30	41.17	18.85	44.90	19.89
IDCT	1%	2.71	1.52	0.81	0.66	0.17	0.12	0.03	0.02	0.02	0.02
Average	23%	25.38	-	18.74	-	15.89	-	14.04	-	13.49	-
Normalized	-	1	-	0.74	-	0.63	-	0.55	-	0.53	-

In our experiment, we compare the following methods:

- EDF-Accurate: EDF scheduling with all jobs in accurate mode.

- EDF-Imprecise: EDF scheduling with all jobs in imprecision mode.
- EDF+ESR: our EDF scheduling with explicit slack reclamation for periodic tasks with independent errors (Section 4.2).
- ILP+OA: our ILP scheduling with online adjustment for periodic tasks with independent errors (Section 4.3.1).
- ILP+Post+OA: our ILP and post-processing scheduling with online adjustment for periodic tasks with independent errors (Section 4.3.2).
- Flipped EDF: our flipped EDF scheduling with online adjustment for periodic tasks with independent errors (Section 4.3.3).
- EDF+ESR(C): our EDF scheduling with explicit slack reclamation for periodic tasks with cumulative errors (Section 4.4.1).
- DP(C): our dynamic programming based offline scheduling with accuracy selection for periodic tasks with cumulative errors (Section 4.4.2).

4.5.1 Simulation Results

Simulations are performed on 13 random testcases and 1 realistic case. In the random cases, actual job execution times are modeled as random variables following Gaussian distribution. The WCET is obtained by the mean value plus 6σ of the distribution, which is further augmented by a margin. The WCET to the best case execution time ratio is around 10. Errors from imprecision executions are also simulated as random variables satisfying Gaussian distribution. The error statistics are derived based on accuracy configurable circuit design [31]. The realistic case is IDCT computation on grayscale and RGB images of various resolution, which form 5 different tasks. The WCET and imprecise errors of IDCT computation are obtained from actual measurement. The testcase characteristics are summarized in column 2, 3 and 4 of Table 4.2. The third column is processor utilization when every job is executed in accurate mode and the fourth column is the number of jobs in hyper-period. Experimental results are based on simulating 10K hyper-periods for each task. Schedulability check according to Theorem 1 is performed on these testcases for

two scenarios: (1) all jobs in accurate mode and (2) all jobs in imprecise mode. The results are listed in the rightmost two columns of Table 4.2. None of these cases is schedulable in accurate mode according to Theorem 1. When all the jobs are in imprecision mode, every testcase except Rnd2 and IDCT is schedulable based on Theorem 1.

The main results are shown in Table 4.3. The EDF-accurate results do not have errors but have many deadline violations. Please note EDF-accurate can successfully schedule Rnd1 and Rnd5, although the cases fail the schedulability check. This discrepancy is largely due to the difference between actual execution time and WCET. All the other methods can always satisfy deadline constraints, but usually result in errors caused by imprecise computing. Our post-processing can reduce the normalized average error from 0.63 of the ILP to 0.55. The best result comes from Flipped EDF, which finds all accurate executions without deadline violation for Case 1 and 5. However, the Flipped EDF is restricted to cases with prior knowledge on task release time compared to online approaches. Moreover, its offline computation does not provide guarantee on errors like the ILP-based approach.

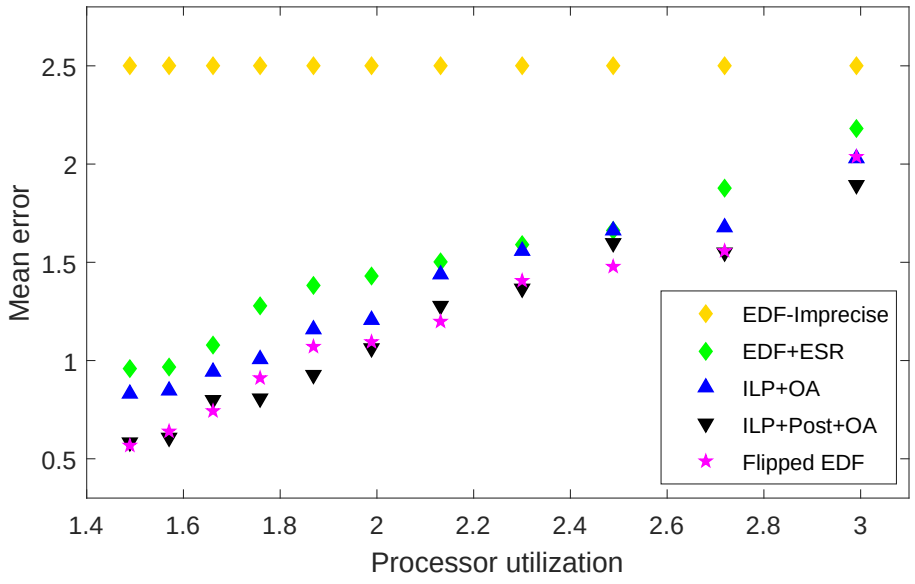

Figure 4.4: Mean error versus utilization.

We study the error versus processor utilization tradeoff and the result is depicted in Figure 4.4. The utilization is $\sum_{i=1}^{n} C_i/p_i$ based on the WCET of accurate executions. Please note utilization is greater than 1 for all the cases and this means these tasks are not schedulable according to Theorem 1 using the accurate execution WCET. However, all of these cases are successfully scheduled without deadline violation by using imprecise computing. Except EDF-imprecise, which always uses imprecision mode, all methods can reduce errors when utilization decreases. Among the methods, ILP+Post+OA and Flipped EDF produce the best result.

Table 4.4: Online runtime of EDF+ESR and ILP.

Test cases	EDF+ESR (ms)	ILP (s)
Rnd1	0.004	< 1
Rnd2	0.005	< 1
Rnd3	0.003	< 1
Rnd4	0.004	< 1
Rnd5	0.005	< 1
Rnd6	0.005	< 1
Rnd7	0.009	< 1
Rnd8	0.007	1.6
Rnd9	0.007	1.7
Rnd10	0.012	948
Rnd11	0.011	1164
Rnd12	0.013	1643
Rnd13	0.017	1842
IDCT	0.008	6.7
Average	0.008	401

We evaluated the computation runtime of these methods. Online computing usually takes a few μs and the ILP runtimes range from seconds to minutes. The actual online scheduling runtime of EDF+ESR is in column 2 of Table 4.4. EDF+ESR (Section 4.2) reclaims slacks online in a judicious manner such that the schedulability conditions, which are validated offline at the beginning, are not destructed. The theoretic complexity of each slack reclamation computation is constant. The rightmost column of Table 4.4 lists the runtime of offline ILP computation.

The simulation results for periodic tasks with cumulative errors are shown in Table 4.5. EDF+ESR(C) can guarantee to satisfy all deadline constraints. We intentionally make the cases very tight such

Table 4.5: Stress test results for periodic tasks with cumulative errors.

Testcases	Rnd1	Rnd2	Rnd3	Rnd4	Rnd5	Rnd6	Rnd7
Error Violations (EDF+ESR(C))	20%	26%	28%	21%	6%	53%	50%
Feasible?(DP(C))	Yes	No	No	Yes	Yes	No	No
Testcases	Rnd8	Rnd9	Rnd10	Rnd11	Rnd12	Rnd13	IDCT
Error Violations (EDF+ESR(C))	40%	12%	39%	46%	58%	49%	13%
Feasible?(DP(C))	No	Yes	No	No	No	No	Yes

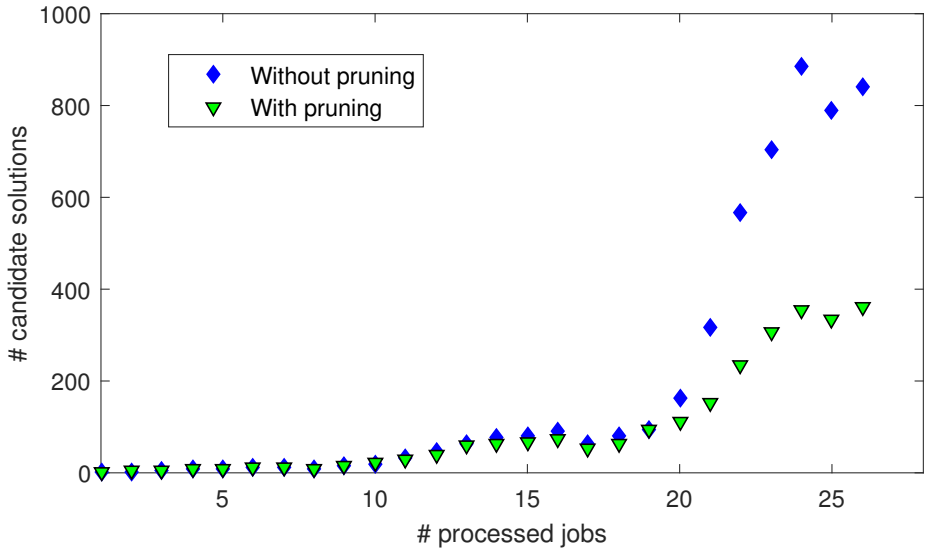

Figure 4.5: The number of candidate partial solutions with and without pruning.

that error constraint violations occur. The error constraint B_i for a task τ_i ranges from 1 to 6 in these cases. The results of EDF-Accurate and EDF-Imprecise are not shown here as the former one never causes errors and the later one has 100% error constraint violations. For cases Rnd1, Rnd4, Rnd5, Rnd9 and IDCT, the DP(C) can find feasible solutions satisfying both deadline and error constraints while EDF+ESR(C) results in violations of error constraints. Figure 4.5 shows the effectiveness of solution pruning in the dynamic programming for case Rnd7. We observe that the pruning can greatly reduce runtime as well.

4.5.2 Linux Prototyping Results

Table 4.6: Tasks in Linux system prototyping.

Task	Accurate WCET(s)	$\hat{\varepsilon}_{accurate}$	Imprecise WCET(s)	$\hat{\varepsilon}_{imprecise}$
τ_1	0.96	0.00001	0.55	20
τ_2	1.21	0.00001	0.27	0.5
τ_3	2.01	0.00001	1.18	5

We implemented EDF-Imprecise, EDF+ESR, Flipped EDF, ILP+Post+OA in Linux 4.6 on a 1200MHz ARM Cortex-A53 processor. The testcase is Newton-Raphson method for solving nonlinear equations numerically. The convergence criterion is $\hat{\varepsilon}_{accurate}$ ($\hat{\varepsilon}_{imprecise}$), which is tight (loose) for accurate (imprecision) mode. The WCETs are obtained by measuring the longest run-time among multiple random tests and augmenting with additional margin. There are three periodic tasks for solving three different kinds of equations. The statistics of this testcase are summarized in Table 4.6. Please note the equation for task 2 is relatively well behaved such that its execution time can reduce quickly when the convergence criterion is relaxed. The mean error results are depicted in Figure 4.6. Our ILP+Post+OA and Flipped EDF lead to much smaller errors than EDF-Imprecise. And the relative overhead ratio of these methods is at the level of 0.0001%.

4.6 Conclusion

Our work reports the first study result on using imprecise computing for non-preemptive real-time scheduling, to the best of our knowledge. Several heuristic algorithms are developed for periodic tasks with independent or cumulative errors. If a set of tasks pass an initial schedulability check where all jobs are assumed to be executed in imprecision mode, all of our algorithms can guarantee to find solution without deadline violation. At the same time, our algorithms either guarantee certain imprecision error bound or minimize errors in the best effort. Experimental results from both simulation and Linux system prototyping implementation show that using imprecision can greatly improve schedulability while our techniques provide desired error and deadline

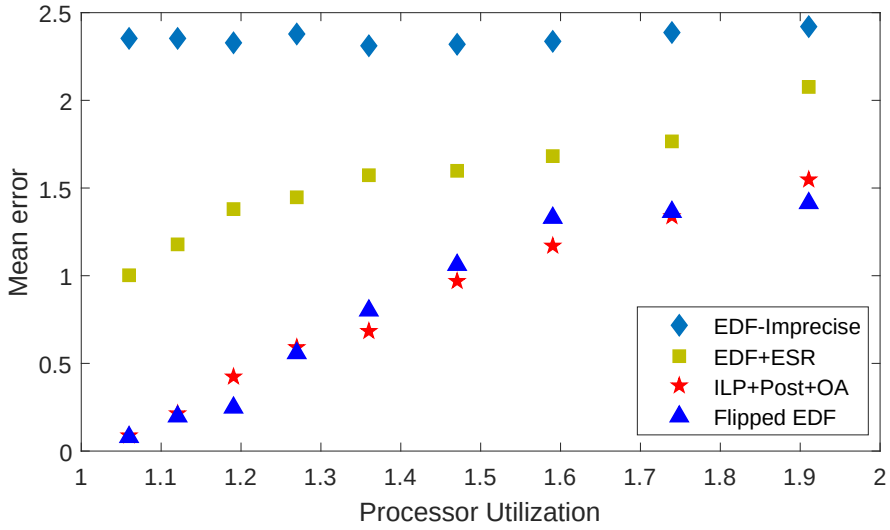

Figure 4.6: Mean error versus utilization from Linux prototyping.

tightness tradeoff.

5. MIXED-CRITICALITY SCHEDUING [1]

We study VPMC (Variable-Precision Mixed-Criticality) system for some well-known multiprocessor scheduling methods to consider low-criticality task executions in high-criticality mode. These methods include partitioned scheduling, fpEDF-VD based global scheduling and MC-DP-Fair scheduling. We also develop a optimization technique to allow low-criticality tasks to be executed with precise computing with schedulability guarantee. Moreover, we introduce two main techniques to mitigate both limitations of classic MC model simultaneously and thereby improve QoS of mixed-criticality systems. Both techniques are mostly built upon the fpEDF-VD. The first is a service preserving technique (Section 5.1.2.3) that allows all low-criticality tasks to execute in high-criticality mode with imprecise computing. Compared to the dual virtual deadline approach (Section 5.1.2.2), the proposed service preserving technique is less conservative and thereby facilitates improved schedulability. The second is a deferred switching scheme (Section 5.1.2.4), which has not been studied for multiprocessors, to the best of our knowledge. It will reduce the chance that a system switches into the very pessimistic high-criticality mode. The two techniques are unified into a single method, which is described in Section 5.1.2.5.

5.1 VPMC System Scheduling on Multiprocessors

Since VPMC and IMC systems follow the same schedulability conditions, we sometimes mention only one of them when a description is applicable for both kinds of models. Their difference is on how to exploit different computing precisions under the same schedulability constraints.

[1]Reprinted with permission from "Graceful degradation of low-criticality tasks in multiprocessor dual-criticality systems " by Lin Huang, I-Hong Hou, Sachin S.Sapatnekar, Jiang Hu, 2018. *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*, Page 159-169 , ©2018 ACM and "Improving QoS for global dual-criticality scheduling on multiprocessors " by Lin Huang, I-Hong Hou, Sachin S.Sapatnekar, Jiang Hu, 2019. *Proceedings of the International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Page 1-11, ©2019 IEEE.

5.1.1 Partitioned Scheduling

5.1.1.1 VPMC Partitioning with EDF-VD Scheduling

For EDF-VD on uniprocessor VPMC systems, we introduce a sufficient schedulability condition that has a form similar to that in conventional MC systems.

Lemma 3. *If a task set in VPMC system satisfies the condition* $max(U_{lo}^{LO}+U_{hi}^{LO}, U_{lo}^{HI}+U_{hi}^{HI}) \leq \frac{3}{4}$, *it is schedulable by EDF-VD on uniprocessor.*

Proof. According to Lemma 2 in [19], if $max(b + \alpha c, \lambda b + c) \leq S(\alpha, \lambda)$, then $\frac{\alpha c}{1-b} \leq \frac{1-(c+\lambda b)}{b-\lambda b}$, where $U_{hi}^{HI} = c$, $U_{hi}^{LO} = \alpha c$, $U_{lo}^{LO} = b$, $U_{lo}^{HI} = \lambda b$ and $S(\alpha, \lambda) = \frac{(1-\alpha\lambda)((2-\alpha\lambda-\alpha)+(\lambda-1)\sqrt{4\alpha-3\alpha^2})}{2(1-\alpha)(\alpha\lambda-\alpha\lambda^2-\alpha+1)}$. Based on Theorem 4 in [19], $S(\alpha, \lambda) \geq \frac{3}{4}$. As such, if $max(U_{lo}^{LO} + U_{hi}^{LO}, U_{lo}^{HI} + U_{hi}^{HI}) \leq \frac{3}{4}$, then $\frac{U_{hi}^{LO}}{1-U_{lo}^{LO}} \leq \frac{1-(U_{hi}^{HI}+U_{lo}^{HI})}{U_{lo}^{LO}-U_{lo}^{HI}}$, which is the sufficient schedulability condition for EDF-VD according to Theorem 4. □

In this method, the given tasks are first partitioned onto m unit-speed processors in the same order as that described in Section 2.4.3. When a task is assigned to a processor, the schedulability check is based on Lemma 3 instead of the conventional approach [10]. This change is to accommodate the IMC/VPMC model. This partitioning method is called *VPMC partitioning*. After the partitioning, the tasks on each processor are scheduled in the same way as EDF-VD under the IMC model [19] (Section 2.4.2). Under the same schedulability constraints, VPMC further allows some low-criticality task to execute with full precision in high-criticality mode.

Lemma 4. *If the VPMC partitioning is successfully completed, all tasks on each processor are schedulable using EDF-VD method.*

Proof. Each time a task is assigned to a processor in the VPMC partitioning, the schedulability condition specified by Lemma 3 is satisfied. If VPMC partitioning is successfully completed, the tasks on each processor satisfy the schedulability condition of Lemma 3 and therefore are schedulable by EDF-VD. □

It is shown in [10] that the partitioned scheduling of conventional MC model can achieve speedup factor of $\frac{8m-4}{3m}$ for m unit-speed processors. We show that the same speedup factor can be achieved for IMC/VPMC model through a proof similar to [10].

Theorem 6. *The speedup factor for VPMC partitioning with EDF-VD scheduling on m unit-speed processors is $\frac{8m-4}{3m}$.*

Proof. Suppose $i-1$ tasks have been successfully assigned and we are attempting to assign the i_{th} task τ_i onto a processor during the partitioning. Let $\tau(p_k)$ denote the set of tasks that have been successfully assigned to processor $p_k, 1 \leq k \leq m$. If the assignment of τ_i fails according to the schedulability check, then at least one of the following two inequalities must hold.

$$U_i^{LO} + \sum_{\tau_j \in \tau(p_k)} U_j^{LO} > \frac{3}{4} \tag{5.1}$$

$$U_i^{HI} + \sum_{\tau_j \in \tau(p_k)} U_j^{HI} > \frac{3}{4} \tag{5.2}$$

We can sum up inequality (5.1) for all the m processors to get $\sum_{j=1}^{i-1} U_j^{LO} > \left(\frac{3}{4} - U_i^{LO}\right) m$ $\Leftrightarrow \sum_{j=1}^{i} U_j^{LO} > (\frac{3}{4} - U_i^{LO})m + U_i^{LO}$, from which we can conclude

$$U_{lo}^{LO} + U_{hi}^{LO} > \left(\frac{3}{4} - U_i^{LO}\right) m + U_i^{LO} \tag{5.3}$$

Similarly, we can sum up inequality (5.2) for all the m processors to obtain

$$U_{lo}^{HI} + U_{hi}^{HI} > \left(\frac{3}{4} - U_i^{HI}\right) m + U_i^{HI} \tag{5.4}$$

If this task set can be scheduled by an optimal scheduling algorithm on m processors of speed s, we have $U_i^{LO} \leq s,\ U_i^{HI} \leq s,\ U_{lo}^{LO} + U_{hi}^{LO} \leq m \cdot s$ and $U_{lo}^{HI} + U_{hi}^{HI} \leq m \cdot s$.

If inequality (5.3) holds, $U_{lo}^{LO} + U_{hi}^{LO} > (\frac{3}{4} - U_i^{LO})m + U_i^{LO} \Leftrightarrow m \cdot s > \frac{3}{4}m - (m-1)s$ $\Leftrightarrow s > \frac{3m}{8m-4}$. Likewise we can obtain $s > \frac{3m}{8m-4}$ if inequality (5.4) holds. If $s \leq \frac{3m}{8m-4}$, this task set can be scheduled by the partitioning followed by EDF-VD on m unit-speed processors. Therefore, the speedup factor of the VPMC partitioning with EDF-VD is $\frac{8m-4}{3m}$. □

5.1.1.2 Enhanced VPMC Partitioning

We introduce two techniques to enhance the VPMC partitioning described in Section 5.1.1.1. The first improvement is to change the schedulability check in the partitioning from Lemma 3 to Theorem 4. From the proof of Lemma 3, we can tell the schedulability condition in Lemma 3 is sufficient for the schedulability condition in Theorem 4. We use an example to demonstrate that the Lemma 3 condition is not necessary for the Theorem 4 condition. The characteristics of this example task set are shown in Table 5.1. If the partitioning is based on Lemma 3, τ_1 and τ_2 are first assigned to processor p_1 and p_2, respectively. When we try to assign τ_3 to processor p_1, $max(U_{lo}^{LO} + U_{hi}^{LO}, U_{lo}^{HI} + U_{hi}^{HI}) = 0.9 > 0.75$. Alternatively, when we try to assign τ_3 to processor p_2, $max(U_{lo}^{LO} + U_{hi}^{LO}, U_{lo}^{HI} + U_{hi}^{HI}) = 0.8 > 0.75$. Hence, the assignment for τ_3 fails for both p_1 and p_2 according to Lemma 3. However, the assignment of τ_3 to p_2 satisfies the schedulability condition in Theorem 4 since $\frac{U_{hi}^{LO}}{1-U_{lo}^{LO}} = \frac{3}{5} \leq \frac{1-(U_{hi}^{HI}+U_{lo}^{HI})}{U_{lo}^{LO}-U_{lo}^{HI}} = \frac{2}{3}$. Thus, the condition in Lemma 3 is more conservative than Theorem 4 and applying Theorem 4 can identify more schedulable task sets.

Table 5.1: Scheduling on 2 unit-speed processors.

Task	χ_i	U_i^{LO}	U_i^{HI}
τ_1	hi	0.4	0.7
τ_2	hi	0.3	0.6
τ_3	lo	0.5	0.2

The second enhancement technique is to balance the utilizations of each processor between the two different criticality modes. More specifically, we attempt to make the difference between $U_{lo}^{LO} + U_{hi}^{LO}$ and $U_{lo}^{HI} + U_{hi}^{HI}$ on each processor as small as possible. The intuition is that a small difference or balanced utilization can avoid one criticality mode being a bottleneck of the whole system. This is inspired by the work on conventional MC systems [36], but applies to IMC/VPMC systems as well. Each time a task τ_i is to be assigned to a processor, all the processors are sorted

in non-decreasing order of $U_{lo}^{HI} + U_{hi}^{HI} - U_{lo}^{LO} - U_{hi}^{LO}$ and indexed from 1 to m. If $\chi_i = hi$, the attempts of assigning τ_i to a processor are in the order from 1 to m. Otherwise, the attempts follow the order from m to 1.

Lemma 5. *If the enhanced VPMC partitioning is successfully completed, all tasks on each processor are schedulable with EDF-VD scheduling.*

Proof. A successful assignment of tasks to a processor indicates the satisfaction of the condition in Theorem 4, which is a sufficient schedulability condition for EDF-VD. □

Lemma 6. *The speedup factor for the enhanced partitioning with EDF-VD scheduling is no greater than* $\frac{8m-4}{3m}$.

Proof. Since the condition of Lemma 3 is sufficient condition for the condition in Theorem 4, a failure of the enhanced partitioning, which implies violation of the condition in Theorem 4, indicates violation of the condition of Lemma 3, i.e., either inequality (5.1) or inequality (5.2) holds. Then, one can follow the same proof as Theorem 6 to derive the speedup factor of $\frac{8m-4}{3m}$. □

5.1.2 Global Scheduling by fpEDF-VD

5.1.2.1 Extension of fpEDF-VD for IMC and VPMC

When fpEDF-VD scheduling, which is briefly reviewed in Section 2.4.5, is applied with IMC/VPMC, the main change is that C_i^{HI} for each low-criticality task τ_i is no longer 0. This execution time change causes utilization change in high-criticality mode. The changed utilizations are evaluated by the same schedulability check described in Section 2.4.5 to tell if a given task set is schedulable with fpEDF-VD.

The tricky part is the transition from low-criticality mode to high-criticality mode. Let t^{hi} be the moment when the system enters high-criticality mode. We define d^{HI} as the earliest deadline (virtual deadline for high-criticality tasks) among all jobs that are active right after t^{hi}. We further define r^{HI} to be the earliest release time among jobs released after t^{hi}. We call $t^{HI} = \min(d^{HI}, r^{HI})$ the *critical moment*. After the critical moment, the schedulability check of high-criticality mode

can be applied without ambiguity. However, the transition time interval from t^{hi} to t^{HI} needs special consideration for IMC/VPMC systems. During the transition interval, there can exist carry-over jobs, which are jobs that are released before t^{hi} and have not been completed at t^{hi}. By the EDF-VD algorithm design, high-criticality carry-over jobs can be guaranteed to complete before their deadlines if the schedulability check is passed. If a low-criticality carry-over job $\tau_{i,j}$ has already executed at least $\tilde{C}_i$ amount of time at t^{hi}, we take its imprecise computing result [2] and quit this job. If $\tau_{i,j}$ has executed less than $\tilde{C}_i$, we continue it till t^{HI} and then quit. By disallowing low-criticality carry-over jobs after t^{HI}, the schedulability of all high-criticality jobs can be maintained. In the worst case, a low-criticality task may lose its job once during the transition interval.

The schedulability condition for our fpEDF-VD method is that both task systems $(\bigcup_{\chi_i=lo}(T_i, C_i^{LO}))\bigcup(\bigcup_{\chi_i=hi}(x * T_i, C_i^{LO}))$ and $(\bigcup_{\chi_i=lo}(T_i, C_i^{HI}))\bigcup(\bigcup_{\chi_i=hi}((1-x) * T_i, C_i^{HI}))$ are each (separately) schedulable on m processors by fpEDF (for low-criticality tasks, $C_i^{HI} = \tilde{C}_i$). The schedulability condition of fpEDF is Lemma 1 in Section 2.4.4.

Lemma 7. *If a task set satisfies $\frac{U_{hi}^{LO}}{1-U_{lo}^{LO}/s} + \frac{U_{hi}^{HI}}{1-U_{lo}^{HI}/s} \leq s$, then it is schedulable using our fpEDF-VD method on a speed s processor.*

Proof. fpEDF reduces to regular EDF on single processor [37]. If a task set satisfies $U_{lo}^{LO} + \frac{U_{hi}^{LO}}{x} \leq s$, or equivalently

$$x \geq \frac{U_{hi}^{LO}}{s - U_{lo}^{LO}} \tag{5.5}$$

task collection $(\bigcup_{\chi_i=lo}(T_i, C_i^{LO}))\bigcup(\bigcup_{\chi_i=hi}(x * T_i, C_i^{LO}))$ is schedulable using EDF on a speed s processor.

If the task set satisfies $U_{lo}^{HI} + \frac{U_{hi}^{HI}}{1-x} \leq s$, or equivalently

$$x \leq 1 - \frac{U_{hi}^{HI}}{s - U_{lo}^{HI}} \tag{5.6}$$

task collection $(\bigcup_{\chi_i=lo}(T_i, C_i^{HI}))\bigcup(\bigcup_{\chi_i=hi}(T_i - x * T_i, C_i^{HI}))$ is schedulable using EDF on a speed s processor.

We can prove this lemma when combining inequality (5.5) and inequality (5.6). □

Lemma 8. *If a task set satisfies* $max(U_{lo}^{LO} + U_{hi}^{LO}, U_{lo}^{HI} + U_{hi}^{HI}) \leq s$*, then it is schedulable by our fpEDF-VD method on a speed* ks *processor , where* $k = \frac{\sqrt{5}+1}{2}$.

Proof. If a task set satisfies $U_{lo}^{LO} + U_{hi}^{HI} \leq ks$, we know that it is schedulable on a speed ks processor, if not we need to show that (from Lemma 7),

$\frac{U_{hi}^{LO}}{1-U_{lo}^{LO}/ks} + \frac{U_{hi}^{HI}}{1-U_{lo}^{HI}/ks} \leq ks$, or equivalently,

$ks(U_{lo}^{LO} + U_{hi}^{LO}) + ks(U_{lo}^{HI} + U_{hi}^{HI}) - U_{lo}^{LO}(U_{lo}^{HI} + U_{hi}^{HI}) - U_{hi}^{LO}U_{lo}^{HI} \leq (ks)^2$

From $max(U_{lo}^{LO} + U_{hi}^{LO}, U_{lo}^{HI} + U_{hi}^{HI}) \leq s$, we have $U_{lo}^{LO} + U_{hi}^{LO} \leq s$, $U_{lo}^{HI} + U_{hi}^{HI} \leq s$, and from $U_{lo}^{LO} + U_{hi}^{HI} > ks$, we have $U_{lo}^{LO} > ks - U_{hi}^{HI} > ks - s$, then

$ks(U_{lo}^{LO} + U_{hi}^{LO}) + ks(U_{lo}^{HI} + U_{hi}^{HI}) - U_{lo}^{LO}(U_{lo}^{HI} + U_{hi}^{HI}) - U_{hi}^{LO}U_{lo}^{HI} < 2ks^2 - (k-1)s^2 = (k+1)s^2 = (ks)^2$, because $k + 1 = \frac{\sqrt{5}+3}{2} = k^2$.

□

Corollary 1. *(Corollary 1 in [9]) If a task set cannot be scheduled by algorithm fpEDF on* m *unit-speed processors, then it cannot be scheduled by preemptive uniprocessor EDF on a processor of speed (*m*+1)/2.*

From Lemma 8 and Corollary 1, we can have Corollary 2 as in [10],

Corollary 2. *If a task set satisfies* $max(U_{lo}^{LO} + U_{hi}^{LO}, U_{lo}^{HI} + U_{hi}^{HI}) \leq m$*, then it is schedulable by our fpEDF-VD method on* m *speed (*$\sqrt{5} + 1$*) processors.*

Corollary 3. *Any task set that can be scheduled by an optimal clairvoyant scheduling algorithm on* m *unit speed processors can be scheduled by our fpEDF-VD method on m speed (*$\sqrt{5} + 1$*) processors.*

Proof. If a task set can be scheduled by an optimal clairvoyant scheduling algorithm on m unit speed processors, it is necessary that $U_{lo}^{LO} + U_{hi}^{LO} \leq m$ and $U_{lo}^{HI} + U_{hi}^{HI} \leq m$, then this task set is schedulable by our fpEDF-VD method on m speed $(\sqrt{5} + 1)$ processors from Corollary 2. □

From Corollary 3, we can show that our fpEDF-VD method for IMC/VPMC has the speedup factor of $(\sqrt{5} + 1)$, which is the same as the fpEDF-VD scheduling of conventional MC systems.

5.1.2.2 Dual Virtual-Deadlines for fpEDF (fpEDF-DVD)

As pointed in Section 5.1.2.1, a direct extension of fpEDF-VD to IMC/VPMC model may result in one-time job abandonment for a low-criticality task during the transition from low-criticality to high-criticality mode. To avoid this loss, we propose to apply the virtual-deadline technique for low-criticality tasks in addition to high-criticality tasks. More specifically, each low-criticality task τ_i has deadlines $y \cdot T_i$ and $(1-y) \cdot T_i$ for low-criticality mode and high-criticality mode, respectively, where y is a scaling factor between 0 and 1. The value of y is found by sweeping between 0 and 1 and selecting the one that satisfies schedulability conditions (Section 2.4.5). This method is called fpEDF-DVD (fpEDF with dual virtual-deadlines).

Theorem 7. *If the virtual deadline based utilization of all tasks satisfy schedulability conditions in both low-criticality and high-criticality mode, the fpEDF-DVD scheduling guarantees all job completions before their deadlines and no job is abandoned.*

Proof. If the schedulability conditions (Section 2.4.5) are satisfied, all tasks are evidently schedulable by fpEDF in low-criticality mode and high-criticality mode. Special attention needs to be paid to carry-over jobs, which are released before the moment t^{hi} entering high-criticality mode and have not been completed at t^{hi}. Then, the low-criticality mode virtual-deadline for each carry-over job must be after t^{hi}. The virtual-deadlines partition a task period into low-criticality mode portion, which are $x \cdot T_i$ and $y \cdot T_i$, and high-criticality mode portion, which are $(1-x) \cdot T_i$ and $(1-y) \cdot T_i$, respectively. For the carry-over jobs, one can treat their low-criticality mode virtual-deadlines as their high-criticality mode release times, which are after t^{hi}. As the schedulability conditions are satisfied, even if the carry-over jobs start execution at their low-criticality virtual-deadlines, they are all schedulable for completion by their actual deadlines.

□

5.1.2.3 Service Preserving Method

The goal of this service preserving technique is to reduce the pessimism of fpEDF-DVD (Section 5.1.2.2) and thereby improve schedulability. It continues to execute low-criticality tasks in

high-criticality mode while all task deadlines are guaranteed to be met. All low-criticality tasks are executed with imprecise computing in high-criticality mode. The imprecise computing costs shorter execution time than precise computing and therefore $C_i^{LO} > C_i^{HI} > 0, \forall \tau_i \in \mathcal{T}_{lo}$.

The key issue is how to guarantee schedulability while low-criticality tasks are continued and consume processor time. In this technique, fpEDF-VD scheduling policy is used in low-criticality mode and high-criticality mode, and DP-Fair scheduling policy is used during the transition. In order to ensure schedulability for low-criticality mode, task set

$$(\bigcup_{\tau_i \in \mathcal{T}_{lo}} (T_i, C_i^{LO})) \bigcup (\bigcup_{\tau_j \in \mathcal{T}_{hi}} (x \cdot T_j, C_j^{LO}))$$

must be schedulable on m processors according to Lemma 1. Please note by scaling T_i by $x \in (0, 1)$, virtual deadline $x \cdot T_i$ is applied for all high-criticality tasks. We define the high-criticality mode condition as that task set

$$(\bigcup_{\tau_i \in \mathcal{T}_{lo}} (T_i - P, C_i^{HI})) \bigcup (\bigcup_{\tau_j \in \mathcal{T}_{hi}} ((1-x) \cdot T_j, C_j^{HI})) \tag{5.7}$$

must be schedulable on m processors according to Lemma 1, where P is a service preserving interval we introduce and will be elaborated later.

The transition from low-criticality to high-criticality mode is subtle and deserves a lot of attention [19]. The treatment of high-criticality tasks is the same as fpEDF-VD [9]. Consider a high-criticality job $\tau_{j,k}$ that is active at moment t^* of mode switching. Its virtual deadline satisfies $\hat{d}_{j,k} = r_{j,k} + x \cdot T_j \geq t^*$, where $r_{j,k}$ is the release time of job $\tau_{j,k}$, otherwise this job would have finished. Right after time t^*, the system enters high-criticality mode and the actual deadline $d_{j,k} = r_{j,k} + T_j$ is enforced. According to Lemma 2, the extra time budget $(1-x) \cdot T_j$ is sufficient for $\tau_{j,k}$ to finish with execution time C_j^{HI}. Therefore, high-criticality tasks are guaranteed to satisfy their deadlines.

The challenging part is how to handle low criticality tasks during the transition, where they can no longer be dropped as in the classic MC model. The non-zero C_i^{HI} for low-criticality tasks makes

the schedulability guarantee quite difficult. The first technique in section 5.1.2.1 may result in a one-time dropping of low-criticality jobs during transition. This is against the original intention of continuing all low-criticality tasks. Another technique in section 5.1.2.2 is dual virtual deadline (DVD) for avoiding such job loss. Unlike the original fpEDF-VD, where virtual deadline is applied only for high-criticality tasks, the DVD approach enforces virtual deadline for low-criticality tasks as well. Virtual deadline is effective for providing guarantee on meeting deadlines. However, it is basically a conservative resource reservation approach that makes schedulability condition more strict and hence causes under-utilization of resources. Applying virtual deadlines for both low-criticality tasks and high-criticality tasks would exacerbate the inefficiency and is an expensive price paid for avoiding one-time loss of low-criticality jobs.

We suggest a *service preserving interval* $[t^*, t^* + P]$, when only the active (carry-over) low-criticality jobs are executed by DP-Fair scheduling, all active high-criticality jobs are suspended and no newly arrival jobs are started. This is to facilitate that all active low-criticality jobs can be finished with imprecise computing during the transition. Meanwhile, the interval P is designed in a way that the schedulability of all the other jobs are still maintained. A critical basis for the service preserving interval is that execution time C_j^{HI} is accommodated after virtual deadline $\hat{d}_{j,k}$ for a high-criticality job $\tau_{j,k}$ in high-criticality mode according to Lemma 2 by fpEDF-VD [9]. The service preserving interval length is defined as

$$P = \min_{\forall \tau_j \in \mathcal{T}_{hi}} C_j^{LO} \tag{5.8}$$

Next, we will discuss schedulability of active jobs and those involving the service preserving interval.

Lemma 9. *By following fpEDF-VD, all high-criticality jobs can guarantee to meet their deadlines in high-criticality mode even if they are not executed in* $[t^*, t^* + P]$.

Proof. The high-criticality jobs involving the service preserving interval $[t^*, t^* + P]$ can be categorized into three cases, all of which will be discussed as follows.

Case 1: Overrun jobs. These are the high-criticality jobs that have executed C^{LO} time but have not finished (see Figure 5.1). At the end of the C^{LO} time, the system enters high-criticality mode when the moment is t^*. According to the schedulability conditions of fpEDF-VD, the virtual deadline $\hat{d}_{j,o}$ of an overrun job $\tau_{j,o}$ satisfies $\hat{d}_{j,o} \geq t^*$. The method of fpEDF-VD (Lemma 2) also indicates that all high-criticality jobs can execute C^{HI} after their virtual deadlines and finish before their actual deadlines in high-criticality mode. Since time C_j^{LO} has already been executed for job $\tau_{j,o}$ at t^*, deferring the rest of its execution by C_j^{LO} maintains the schedulability. In other words, the rest of the overrun job can start from $\hat{d}_{j,o} + C_j^{LO} = \hat{d}_{j,o} + C_j^{LO} + t^* - t^* = t^* + P_j$, where $P_j = \hat{d}_{j,o} + C_j^{LO} - t^*$. Since $\hat{d}_{j,o} \geq t^*$, $P_j \geq C_j^{LO}$. Therefore, postponing the execution of the rest of $\tau_{j,o}$ by C_j^{LO} will maintain the schedulability of overrun jobs.

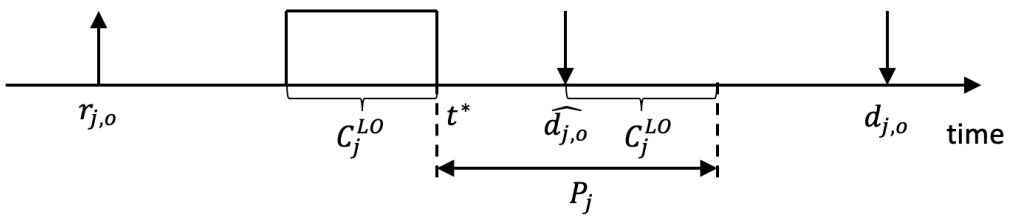

Figure 5.1: Case 1: service preserving interval for an overrun job.

Case 2: Active high-criticality jobs without overrun (see Figure 5.2). A high-criticality job $\tau_{j,k}$ has been executed $q_{j,k} < C_j^{LO}$ by t^*. Then, its rest portion can start from $\hat{d}_{j,k} + q_{j,k}$ with guarantee of meeting its deadline according to fpEDF-VD. Like Case 1, $\hat{d}_{j,k} + q_{j,k} = t^* + P_j$, where $P_j = \hat{d}_{j,k} + q_{j,k} - t^*$. By schedulability condition in low-criticality mode, $q_{j,k} + \hat{d}_{j,k} - t^* \geq C_j^{LO}$, then $P_j \geq C_j^{LO}$. Hence, such a job can be suspended in $[t^*, t^* + C_j^{LO}]$ without affecting its schedulability.

Case 3: High-criticality jobs arriving during the service preserving interval (see Figure 5.3). The release time $r_{j,k}$ of such a job $\tau_{j,k}$ satisfies

$$t^* \leq r_{j,k} \leq t^* + P. \tag{5.9}$$

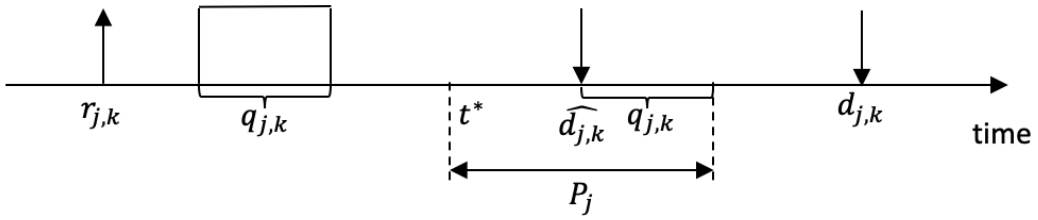

Figure 5.2: Case 2: service preserving interval for an active high-criticality job without overrun.

The schedulability conditions in fpEDF-VD require that

$$r_{j,k} + C_j^{LO} \leq \hat{d}_{j,k}. \tag{5.10}$$

Combing inequality (5.9) and (5.10), we have

$$C_j^{LO} \leq \hat{d}_{j,k} - r_{j,k} \leq \hat{d}_{j,k} - t^* = P_j$$

Since $\tau_{j,k}$ can guarantee finish before its deadline even if it starts from $\hat{d}_{j,k}$ according to fpEDF-VD, its start time can be deferred by P_j, which is lower bounded by C_j^{LO}.

Overall, all high-criticality jobs involving the service preserving interval can be deferred by C^{LO} without affecting their schedulability. Hence, deferring by $P = \min_{\forall \tau_j \in \mathcal{T}_{hi}} C_j^{LO}$ for all these jobs can still guarantee to meet their deadlines.

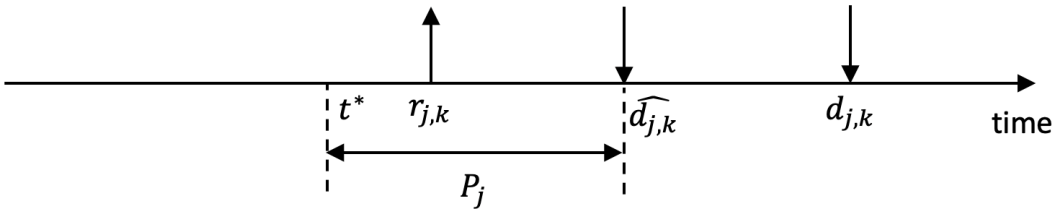

Figure 5.3: Case 3: service preserving interval for an immediate newly coming high-criticality job.

□

Next, we describe schedulability conditions for active low-criticality jobs during the service preserving interval $[t^*, t^* + P]$. At t^*, if a low-criticality job $\tau_{i,k}$ has already been executed for at least C_i^{HI}, it is terminated with imprecise computing result. An active (carry-over) low-criticality job $\tau_{i,k}$ means that it has been executed for $q_{i,k} < C_i^{HI}$ by t^*. The active low-criticality jobs are scheduled by the fluid-based DP-Fair method (see Section 2.4.6) in the service preserving interval, while fpEDF-VD is employed all the other time. Although fluid scheduling tends to entail frequent job preemptions, it is utilized only within the limited service preserving interval. The schedulability for DP-fair method is largely decided by the job density.

Lemma 10. *The density $\delta_{i,k}$ of an active low-criticality job $\tau_{i,k}$ in $[t^*, t^* + P]$ is no greater than* $\max(\frac{C_i^{HI}}{P}, \frac{C_i^{HI}}{C_i^{LO}})$.

Proof. This bound is derived from two cases. In one case, deadline $d_{i,k} \geq t^* + P$ as shown in Figure 5.4. In the worst case for the service preserving interval, entire job C_i^{HI} is executed by $t^* + P$, the density of this case is upper bounded as

$$\delta_{i,k}|_{d_{i,k} \geq t^* + P} \leq \frac{C_i^{HI}}{P}. \tag{5.11}$$

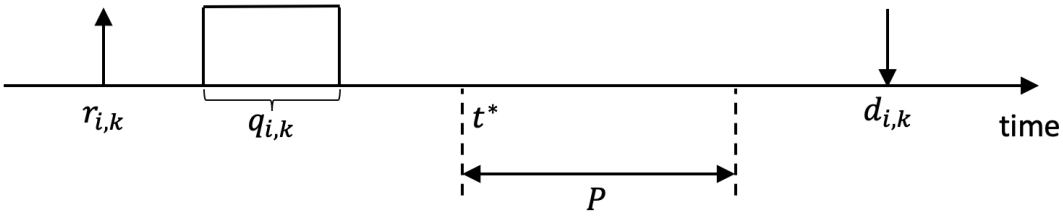

Figure 5.4: Active low-criticality job with deadline after $t^* + P$.

In the other case, $d_{i,k} < t^* + P$ as shown in Figure 5.5. If $q_{i,k}$ has been executed by t^*, the

density is estimated by

$$\delta_{i,k}|_{d_{i,k}<t^*+P} = \frac{C_i^{HI} - q_{i,k}}{d_{i,k} - t^*} \tag{5.12}$$

By the schedulability condition in low-criticality mode, $C_i^{LO} - q_{i,k} \leq d_{i,k} - t^*$. Therefore,

$$\delta_{i,k}|_{d_{i,k}<t^*+P} = \frac{C_i^{HI} - q_{i,k}}{d_{i,k} - t^*} \leq \frac{C_i^{HI} - q_{i,k}}{C_i^{LO} - q_{i,k}} \tag{5.13}$$

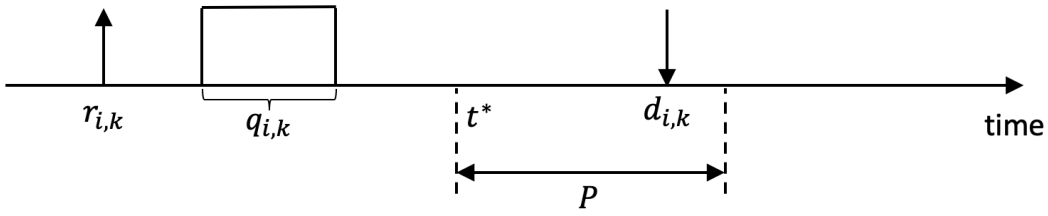

Figure 5.5: Active low criticality job with deadline before $t^* + P$.

Consider a function

$$f(x) = \frac{C_i^{HI} - x}{C_i^{LO} - x}, \quad 0 \leq x < C_i^{HI} < C_i^{LO}. \tag{5.14}$$

Since derivative $f^{'}(x) = \frac{C_i^{HI} - C_i^{LO}}{(C_i^{LO} - x)^2} < 0$, $f(x)$ is a monotone decreasing function and its maximum is at $x = 0$. Hence,

$$\delta_{i,k}|_{d_{i,k}<t^*+P} \leq \frac{C_i^{HI} - q_{i,k}}{C_i^{LO} - q_{i,k}} \leq \frac{C_i^{HI}}{C_i^{LO}} \tag{5.15}$$

By combining the two cases, we have

$$\delta_{i,k} \leq \max(\frac{C_i^{HI}}{P}, \frac{C_i^{HI}}{C_i^{LO}}) \tag{5.16}$$

□

In the worst case, every low-criticality task has an active job at t^*. According to Theorem 5

and Lemma 10, a sufficient condition for DP-Fair method to successfully schedule all the active jobs on m processors in $[t^*, t^* + P]$ is

$$\sum_{\forall \tau_i \in \mathcal{T}_{lo}} \max(\frac{C_i^{HI}}{P}, \frac{C_i^{HI}}{C_i^{LO}}) \leq m \tag{5.17}$$

Last, we discuss new low-criticality jobs that arrive in $[t^*, t^* + P]$. Our method does not allow such jobs to be executed until $t^* + P$. In other words, their execution is deferred by at most P. We specify that task set

$$(\bigcup_{\tau_i \in \mathcal{T}_{lo}} (T_i - P, C_i^{HI})) \bigcup (\bigcup_{\tau_j \in \mathcal{T}_{hi}} ((1-x) \cdot T_j, C_j^{HI}))$$

must be schedulable according to Lemma 1 in high-criticality mode. More specifically, a low-criticality task τ_i is scheduled with period (implicit deadline) $T_i - P$. Thus, with deferral of P, a low-criticality job arriving in $[t^*, t^* + P]$ is still schedulable.

Putting everything together, the service preserving policy is stated as follows.

Service preserving policy: *From the moment t^* switching to high-criticality mode to $t^* + P$, where $P = \min_{\forall \tau_j \in \mathcal{T}_{hi}} C_j^{LO}$, only active low-criticality jobs are executed with DP-Fair scheduling and all the other jobs can not be executed.*

From Lemmas 9 and 10, we can reach the following conclusion.

Theorem 8. *When applying the service preserving policy with fpEDF-VD scheduling, a task set $\mathcal{T}$ is schedulable on m identical processors if $\mathcal{T}$ satisfies the following schedulability conditions.*

- *task set*

  $$(\bigcup_{\tau_i \in \mathcal{T}_{lo}} (T_i, C_i^{LO})) \bigcup (\bigcup_{\tau_j \in \mathcal{T}_{hi}} (x \cdot T_j, C_j^{LO}))$$

  must be schedulable on m processors according to Lemma 1.

- $\sum_{\forall \tau_i \in \mathcal{T}_{lo}} \max(\frac{C_i^{HI}}{P}, \frac{C_i^{HI}}{C_i^{LO}}) \leq m$ *during* $[t^*, t^* + P]$, *where* $P = \min_{\forall \tau_j \in \mathcal{T}_{hi}} C_j^{LO}$.

- *task set*

$$(\bigcup_{\tau_i \in \mathcal{T}_{lo}} (T_i - P, C_i^{HI})) \bigcup (\bigcup_{\tau_j \in \mathcal{T}_{hi}} ((1-x) \cdot T_j, C_j^{HI}))$$

must be schedulable on m processors according to Lemma 1.

5.1.2.4 Deferred Switching Scheme

Although high-criticality mode guarantees that high-criticality tasks complete before deadlines in the worst case, it entails expensive price that the execution time estimation of all high-criticality tasks becomes overly pessimistic even if many of them do not have overrun. Moreover, low-criticality tasks would run in imprecise mode or are even dropped. The threshold of switching to high-criticality mode in the conventional protocol is quite low. That is, any single high-criticality job overrun triggers the mode switching. Generally, there are two approaches that can address this issue. One is to allow a system to switch back to low-criticality mode like bailout mode protocol [53, 54]. The other is to defer the switching into high-criticality mode like the work of [22]. We take the latter approach for multiprocessors while the work of [22] is for uniprocessor scheduling.

Our approach is built upon fpEDF-VD [9] with the observation that the conservativeness of fpEDF-VD allows room for such deferral. In the proposed scheme, a single high-criticality job overrun does not warrant immediate switching to high-criticality mode. Instead, the system enters a vigilant mode, which is almost identical as low-criticality mode except that the overrun job is monitored to decide if the system can recover back to low-criticality mode or must switch to high-criticality mode. The online monitoring and decision can still guarantee satisfaction of all deadline constraint even though C_j^{HI} is applied with the overrun job, while all low-criticality tasks are retained and all the other high-criticality tasks are scheduled with less pessimistic C_j^{LO}.

Vigilant mode is defined by the following characteristics.

- At low-criticality mode, if any high-criticality job $\tau_{j,o} \in \tau_j \in \mathcal{T}_{hi}$ does not finish after being executed C_j^{LO}, i.e., has overrun, then the system enters vigilant mode at this moment t'.

- Every low-criticality task $\tau_i \in \mathcal{T}_{lo}$ continues to execute with precise computing with esti-

mated execution time C_i^{LO}.

- Each non-overrun high-criticality job $\tau_{j,k}$ continues to execute with estimated execution time C_j^{LO}.
- Each overrun high-criticality job $\tau_{j,o}$ is scheduled with estimated execution time C_j^{HI} and priority lower than any low-criticality job and non-overrun high-criticality jobs.
- Each overrun high-criticality job $\tau_{j,o}$ is assigned a series of checkpoints $c_h(\tau_{j,o}), h = 0, 1, 2, ...$, when some conditions are checked with constant time to decide if $\tau_{j,o}$ allows to return to low-criticality mode, demands high-criticality mode or needs to stay at vigilant mode.
- The system switches to high-criticality mode if any overrun high-criticality job demands high-criticality mode.
- When no overrun high-criticality job demands high-criticality mode, the system stays at vigilant mode as long as any overrun high-criticality job needs so.
- The system returns to low-criticality mode when all overrun high-criticality jobs allow so.

The initial checkpoint for an overrun job $\tau_{j,o}$ is defined as

$$c_1(\tau_{j,o}) = \hat{d}_{j,o} + C_j^{LO} \tag{5.18}$$

where $\hat{d}_{j,o}$ is the virtual deadline by fpEDF-VD [9] for job $\tau_{j,o}$. An example of checkpoint is shown in Figure 5.6. Later on, it is likely that the checkpoint is updated to $c_2(\tau_{j,o}) > c_1(\tau_{j,o})$ and $c_3(\tau_{j,o}) > c_2(\tau_{j,o})$, and so on. For the convenience of representation, we specify $c_0(\tau_{j,o}) = \hat{d}_{j,o}$ and $q_0(\tau_{j,o}) = C_j^{LO}$. When time reaches the checkpoint $c_h(\tau_{j,o}), h = 1, 2, ...$, the amount of execution of $\tau_{j,o}$ from the previous checkpoint $c_{h-1}(\tau_{j,o})$ to $c_h(\tau_{j,o})$ is examined with constant time and the result determines three different outcomes.

1. If job $\tau_{j,o}$ is completed by $c_h(\tau_{j,o})$, it allows the system to return to low-criticality mode.

2. Zero amount has been done, then high-criticality mode is demanded.

3. If $q_h(\tau_{j,o})$ processor time has been spent on executing $\tau_{j,o}$ yet it is not completed, the job needs to stay at vigilant mode with the next checkpoint as $c_{h+1}(\tau_{j,o}) = c_h(\tau_{j,o}) + q_h(\tau_{j,o}), h = 0, 1, 2, ...$

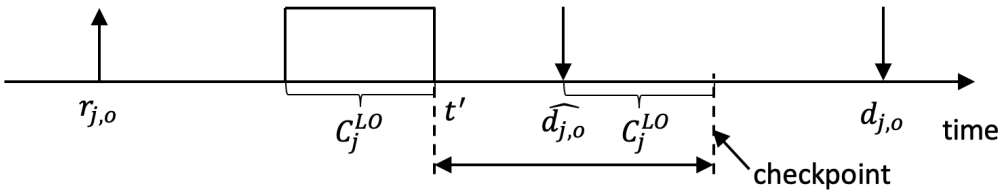

Figure 5.6: Illustration of checkpoint.

By introducing the vigilant mode, our approach can defer the switching to high-criticality mode. As the system can return to low-criticality mode from vigilant mode, the overall probability of entering high-criticality mode is also reduced. As such, low-criticality tasks are executed with improved quality and less conservative execution time estimation is applied for high-criticality jobs without overrun. Next, we will show that all jobs are guaranteed to satisfy their deadline constraints under this scheme. The key idea is to let overrun jobs have low priority in the vigilant mode so that the schedulability of the other jobs are not affected. At the same time, an overrun job reclaims time slack at runtime in an opportunistic manner. The online checking has constant complexity.

Lemma 11. *The deferred switching scheme guarantees that all jobs complete before their deadlines if they satisfy the schedulability conditions of fpEDF-VD and* $C_i^{HI} = 0$ *for all low-criticality tasks* $\tau_i \in \mathcal{T}_{lo}$.

Proof. We prove this for three kinds of jobs - all low-criticality jobs, non-overrun high-criticality

jobs and overrun high-criticality jobs, in all three modes - low-criticality mode, vigilant mode and high-criticality mode.

The low-criticality mode in this scheme is handled in the same way as fpEDF-VD for all kinds of jobs. Hence, all tasks can guarantee to satisfy deadlines if they meet the fpEDF-VD schedulability conditions.

The high-criticality mode in this scheme is also identical to that for fpEDF-VD. As such, all non-overrun high-criticality jobs can finish before their deadlines. In the classic MC system model, all low-criticality tasks are dropped in high-criticality mode. In Section 5.1.2.5, we will show how to unify the deferred switching scheme with the service preserving technique such that low-criticality tasks can continue to execute with imprecise computing.

In the vigilant mode, low-criticality tasks and non-overrun high-criticality jobs are treated in the same way as in low-criticality mode, except the presence of overrun high-criticality jobs. However, overrun high-criticality jobs have lower priority. Thus low-criticality tasks and non-overrun high-criticality jobs are not affected by those overrun high-criticality jobs, and their deadline can still be met with guarantee.

Last, we discuss overrun high-criticality jobs in the vigilant mode and high-criticality mode. The worst case execution time of an overrun job $\tau_{j,o}$ is C_j^{HI}, of which C_j^{LO} has already been executed. According to fpEDF-VD [9], entire execution time C_j^{HI} can be accommodated from its virtual deadline $\hat{d}_{j,k}$ to its actual deadline $d_{j,k}$ for any high-criticality job $\tau_{j,k}$ in high-criticality mode. As the overrun job has already been executed C_j^{LO}, time interval $[\hat{d}_{j,o} + C_j^{LO}, d_{j,o}]$ is sufficient to accommodate the rest execution time $C_j^{HI} - C_j^{LO}$. Please note the first checkpoint $c_1(\tau_{j,o}) = \hat{d}_{j,o} + C_j^{LO}$. Even if nothing of $\tau_{j,o}$ has been executed in $[t', c_1(\tau_{j,o})]$ the remaining $C_j^{HI} - C_j^{LO}$ part of the overrun job can guarantee to meet its deadline if the system switches to high-criticality mode at $c_1(\tau_{j,o})$.

When a new checkpoint is added with augmenting $q_h(\tau_{j,o})$ to the previous checkpoint, the condition is that $q_h(\tau_{j,o})$ has been executed from the previous checkpoint. As such, the amount of deferral of high-criticality mode start time is equal to the reduction of remaining execution

time of the overrun job. Therefore, switching to high-criticality mode at the new checkpoint still guarantees the satisfaction of deadline constraint for this overrun job. Overall, if an overrun job finishes after switching to high-criticality mode, it can guarantee to be completed before its actual deadline.

Since each checkpoint update is extended by $q_h(\tau_{j,o})$ that has been executed, the total extension after the virtual deadline cannot be greater than C_j^{HI}. In other words, the maximum possible checkpoint is $\hat{d}_{j,o} + C_j^{HI}$. If the system has not switched to high-criticality mode at $\hat{d}_{j,o} + C_j^{HI}$, job $\tau_{j,o}$ must have finished by $\hat{d}_{j,o} + C_j^{HI}$. Since fpEDF-VD entails that $\hat{d}_{j,o} + C_j^{HI} \leq d_{j,o}$, the job must have finished before its deadline during vigilant mode.

□

Although we propose to perform low-criticality jobs with precise computing in vigilant mode, sometimes, it is beneficial to execute them with imprecise computing. Since the execution time of imprecise computing is usually shorter than precise computing, more processor time can be saved and the chance of switching to high-criticality mode is decreased. The choice between precise computing and imprecise computing may depend on how low-criticality tasks are treated in high-criticality mode. If they are dropped, then even imprecise computing in vigilant mode is a QoS improvement. If low-criticality tasks are continued with imprecise computing in high-criticality mode, then it makes more sense to run them with precise computing in vigilant mode to have the advantage of deferring the mode switching to high-criticality mode.

5.1.2.5 Unified Deferred Switching and Service Preserving

When unifying the deferred switching and service preserving methods, a couple of changes need to made to the deferred switching part. Both of the techniques make use of Lemma 2 that fpEDF-VD accommodates C_j^{HI} after virtual deadline $\hat{d}_{j,o}$ for a high-criticality job $\tau_{j,o}$ in high-criticality mode. When both the techniques are applied at the same time, the same property cannot be utilized twice. This is the key reason for changing the deferred switching method here. The first change is that each overrun job can only have one checkpoint as opposed to possibly multiple

checkpoints described in Section 5.1.2.4. Since there is only one checkpoint, an overrun job not finished by the checkpoint immediately demands high-criticality mode. The second change is that the checkpoint is defined as $c(\tau_{j,o}) = \hat{d}_{j,o}$ in contrast to $c_1(\tau_{j,o}) = \hat{d}_{j,o} + C_j^{LO}$ in Section 5.1.2.4. In the unification, the service preserving part is the same as introduced in Section 5.1.2.3.

Since the change in the unified method is restricted to the deferred switching part, which is an online technique, the offline schedulability conditions of the unified method is the same as that for the service preserving method, which is stated in Theorem 8.

Lemma 12. *If the schedulability conditions in Theorem 8 are satisfied, the modified deferred switching in the unified method still maintains the schedulability.*

Proof. In the unified method, low-criticality mode and the high-criticality mode after the service preserving interval are identical to each stand-alone method. The vigilant mode is the same as low-criticality mode except the handling of overrun jobs. Overrun jobs are executed opportunistically in vigilant mode with their deadline guarantee provided by switching to high-criticality mode in time. The vigilant mode in the unified method is never longer than that in the stand alone deferred switching scheme. As such, the deadline guarantee of overrun jobs still relies on the scheduling in high-criticality mode. In the unified method, t^* is at a checkpoint $c(\tau_{j,o}) = \hat{d}_{j,o}$, which is covered in Case 1 in the proof of Lemma 9. The other kinds of jobs and cases discussed in the proof of Lemma 9 still hold in the unified method. The schedulability condition for the service preserving interval in the unified method is unchanged. Therefore, the unified method can guarantee that all tasks meet their deadlines if the schedulability conditions in Theorem 8 are satisfied.

□

5.1.3 Extension of MC-DP-Fair Scheduling for IMC and VPMC Systems

MC-DP-Fair is one realization of the fluid-based scheduling [25], which is not directly implementable by itself. Fluid-based scheduling associating Quality of Service for low critical tasks has been studied for VPMC in [20] and the method is called MCFQ, however, MC-DP-Fair scheduling for VPMC is barely discussed in [20]. Here, we show how to extend MC-DP-Fair scheduling to

VPMC-DP-Fair scheduling. In DP-Fair scheduling, an important concept is task density δ_i for task τ_i, which is usually equal to $\frac{C_i}{T_i}$ with a few exceptions. Fluid-based scheduling uses another concept, execution rate θ_i for τ_i, which is the fraction of a unit-speed processor allocated for executing τ_i.

For a low-critical task τ_i in VPMC-DP-Fair, $\delta_i^{LO} = \theta_i^{LO} = U_i^{LO}$ and $\delta_i^{HI} = \theta_i^{HI} = U_i^{HI}$, where the superscripts ^{LO} and ^{HI} indicate low-criticality and high-criticality mode, respectively. Its virtual deadline $V_i = T_i$. Please note $\delta_i^{HI} = 0$ in MC-DP-Fair. Let w_i be the length of time interval from job release time of τ_i to Γ, which is the earliest deadline partition after the system enters high-criticality mode.

Lemma 13. *In VPMC-DP-Fair scheduling, a low-criticality carry-over job of τ_i can be executed for at least $\tilde{C}_i$ time.*

Proof. Let C_i^{TR} denote the actual execution time of a carry-over job of τ_i.

$$C_i^{TR} = w_i \cdot \delta_i^{LO} + (T_i - w_i)\delta_i^{HI} = w_i \cdot U_i^{LO} + (T_i - w_i)U_i^{HI}$$

$$\geq w_i \cdot U_i^{HI} + (T_i - w_i)U_i^{HI} = T_i \cdot U_i^{HI} = \tilde{C}_i$$

□

For a high-criticality task τ_i, $\delta_i^{LO} = \theta_i^{LO}$, which is proved to be no greater than U_i^{HI} [20], and virtual deadline $V_i = C_i^{LO}/\theta_i^{LO}$. Its density in high-criticality mode is specified by [25]

$$\delta_i^{HI} = \frac{C_i^{HI} - \delta_i^{LO} \cdot w_i}{T_i - w_i}. \tag{5.19}$$

Lemma 14. *Given a task set that is deemed to be schedulable by MCFQ [20], if it is scheduled by VPMC-DP-Fair, then $\delta_i^{LO} \leq \theta_i^{LO}$ and $\delta_i^{HI} \leq \theta_i^{HI}$ for each task τ_i.*

Proof. For each task τ_i, $\delta_i^{LO} = \theta_i^{LO} \leq \theta_i^{LO}$. For each low-criticality task τ_i, we have $\delta_i^{HI} = \theta_i^{HI} \leq \theta_i^{HI}$. For each high-criticality task τ_i, since δ_i^{HI} is a variable depending on w_i according to

Equation (5.19), we need to show that the maximum value of δ_i^{HI} is no greater than θ_i^{HI}. Consider the derivative of δ_i^{HI} with respect to w_i

$$\frac{d\delta_i^{HI}}{dw_i} = \frac{C_i^{HI} - \delta_i^{LO} \cdot T_i}{(T_i - w_i)^2} = \frac{U_i^{HI} - \delta_i^{LO}}{T_i \cdot (T_i - w_i)^2}. \tag{5.20}$$

Since $\delta_i^{LO} = \theta_i^{LO} \leq U_i^{HI}$ [20], the derivative is non-negative and the function of Equation (5.19) is monotonically increasing. By definition, we know $w_i \leq V_i$. Thus, δ_i^{HI} has the maximum value when $w_i = V_i$,

$$\delta_{i,max}^{HI} = \frac{U_i^{HI} - U_i^{LO}}{1 - U_i^{LO}/\theta_i^{LO}} \tag{5.21}$$

In MCFQ [20], $\theta_i^{HI} = \frac{U_i^{HI} - U_i^{LO}}{1 - U_i^{LO}/\theta_i^{LO}}$, which is equal to $\delta_{i,max}^{HI}$, then we have $\delta_i^{HI} \leq \theta_i^{HI}$ for high-criticality tasks. □

Lemma 15. *Given a task set that is deemed to be schedulable by MCFQ, it is schedulable by VPMC-DP-Fair.*

Proof. Given a task set that is deemed to be schedulable by MCFQ, we have $\sum_{\tau_i \in \mathcal{T}} \theta_i^{LO} \leq m$ and $\sum_{\tau_i \in \mathcal{T}} \theta_i^{HI} \leq m$, then we have $\sum_{\tau_i \in \mathcal{T}} \delta_i^{LO} \leq \sum_{\tau_i \in \mathcal{T}} \theta_i^{LO} \leq m$ and $\sum_{\tau_i \in \mathcal{T}} \delta_i^{HI} \leq \sum_{\tau_i \in \mathcal{T}} \theta_i^{HI} \leq m$ from Lemma 14. Hence, low-criticality mode schedulability and high-criticality mode schedulability by Theorem 5 are satisfied and the task set is schedulable by VPMC-DP-Fair. □

5.2 Precision Optimization for VPMC Systems

5.2.1 Optimization Kernel

Under the VPMC model, there can be utilization slack for some processors when schedulability conditions are satisfied. The slack allows some low-criticality tasks to be executed with precise computing in high-criticality mode while the schedulability conditions are still satisfied. For a low-criticality task τ_i, the error of its imprecise computing is denoted by e_i. The error of a low-criticality task τ_i execution in high-criticality mode is denoted by e_i^{HI}, which is equal to e_i if it is executed with imprecise computing and otherwise 0. If each task τ_i has a weighting factor η_i indicating its importance, the precision optimization problem is stated as follows.

Problem 3. *Given a set of independent sporadic tasks $\mathcal{T} = \{\tau_1, \tau_2, ...\}$ in VPMC model and a scheduling method $\mathcal{S}$, decide if each low-criticality task τ_i is executed with precise or imprecise computing in high-criticality mode such that the total weighted error $\sum_{\chi_i=lo} \eta_i \cdot e_i^{HI}$ is minimized while the schedulability conditions for $\mathcal{S}$ are maintained.*

For each low-criticality task τ_i, let ΔU_i denote the additional processor utilization when its execution is changed from imprecise to precise computing and thus

$$\Delta U_i = \frac{\hat{C}_i - \tilde{C}_i}{T_i}. \tag{5.22}$$

Let $\bar{U}_{lo}^{HI}$ denote the maximal possible U_{lo}^{HI} under the schedulability constraint for a scheduling method. The *utilization slack* Ψ for low-critical tasks in high-criticality mode is defined as

$$\Psi = \bar{U}_{lo}^{HI} - U_{lo}^{HI} \tag{5.23}$$

Then, Problem 3 is essentially 0-1 knapsack problem. Let z_i be a binary decision variable for each low-criticality task τ_i. When $z_i = 1$, task τ_i is assigned to precise computing; otherwise it is executed with imprecise computing in high-criticality mode. The knapsack problem formulation is as follows.

$$\begin{aligned} &\text{maximize} \sum_{\chi_i=lo} \eta_i \cdot e_i \cdot z_i \\ &\text{subject to} \sum_{\chi_i=lo} \Delta U_i \cdot z_i \leq \Psi \\ &z_i \in \{0, 1\}, \ \ \forall \tau_i \in \mathcal{T}_{lo} \end{aligned} \tag{5.24}$$

In this formulation, the objective is to maximize the error reduction obtained from using precise computing compared to IMC model. The 0-1 knapsack problem is a well-known NP-complete problem. It can be optimally solved by dynamic programming with pseudo-polynomial complexity.

5.2.2 Utilization Slack Estimation and Customization for Different Scheduling Methods

5.2.2.1 Slack Estimation and Precision Optimization for Partitioned Scheduling

For partitioned scheduling, if $U_{lo}^{LO} + U_{hi}^{HI} \leq 1$, all tasks can be scheduled with EDF and all low-criticality tasks can be executed with precise computing. Hence, the slack estimation and precision

optimization is necessary only when $U_{lo}^{LO} + U_{hi}^{HI} > 1$. For both of the partitioned scheduling methods introduced in section 5.1.1, utilization slack is estimated for individual processors. On each processor, the maximal schedulable utilization $\bar{U}_{lo}^{HI}$ can be derived according to Theorem 2 and Theorem 3.

Theorem 9. *The utilization slack of a processor after the VPMC partitioning is* $\frac{1-U_{lo}^{LO}-U_{hi}^{LO}U_{lo}^{LO}-U_{hi}^{HI}+U_{lo}^{LO}U_{hi}^{HI}}{1-U_{lo}^{LO}-U_{hi}^{LO}} - U_{lo}^{HI}$.

Proof. From inequality (2.3), we can find the range of the scaling factor as

$$x \geq \frac{U_{hi}^{LO}}{1 - U_{lo}^{LO}} \tag{5.25}$$

Further, we know from inequality (2.5) that

$$U_{lo}^{HI} \leq \frac{1 - xU_{lo}^{LO} - U_{hi}^{HI}}{1 - x} \tag{5.26}$$

Taking derivative with respective to x on right-hand-side of inequality (5.26), we have

$$\frac{1 - U_{lo}^{LO} - U_{hi}^{HI}}{(1 - x)^2} \tag{5.27}$$

Since $U_{lo}^{LO} + U_{hi}^{HI} > 1$, the right-hand-side of inequality (5.26) is a decreasing function with respect to x. Then, $\bar{U}_{lo}^{HI}$ can be obtained by plugging RHS of inequality (5.25) into inequality (5.26):

$$\bar{U}_{lo}^{HI} = \frac{1 - U_{lo}^{LO} - U_{hi}^{LO}U_{lo}^{LO} - U_{hi}^{HI} + U_{lo}^{LO}U_{hi}^{HI}}{1 - U_{lo}^{LO} - U_{hi}^{LO}} \tag{5.28}$$

Therefore, the utilization slack is given by:

$$\Psi = \frac{1 - U_{lo}^{LO} - U_{hi}^{LO}U_{lo}^{LO} - U_{hi}^{HI} + U_{lo}^{LO}U_{hi}^{HI}}{1 - U_{lo}^{LO} - U_{hi}^{LO}} - U_{lo}^{HI} \tag{5.29}$$

□

5.2.2.2 Slack Estimation and Precision Optimization for fpEDF-VD Based Global Scheduling

Under fpEDF, a subset $\mathcal{T}_{hp} \subset \mathcal{T}$ of tasks are designated with the highest priority and $m_{hp} = |\mathcal{T}_{hp}|$ processors are allocated for them. Please note this allocation is not static, i.e., the m_{hp}

processors at one time may be different from the m_{hp} processors at another time. The other tasks $\mathcal{T}_{EDF} = \mathcal{T} - \mathcal{T}_{hp}$ follow EDF priority and are executed on $m_{EDF} = m - m_{hp}$ processors. Each low-criticality task $\tau_i \in \mathcal{T}_{hp}$ can always execute with precise computing in high-criticality mode, since an entire processor is allocated to one task in $\mathcal{T}_{hp}$ and this allocation is sufficient for precise computing.

For the fpEDF-VD-VPMC method described in Section 5.1.2.1, the utilization slack of $\mathcal{T}_{EDF}$ is estimated by the following statement according to Lemma 1.

Proposition 2. *The utilization slack for $\mathcal{T}_{EDF}$ on the m_{EDF} processors under fpEDF-VD-VPMC scheduling is $m_{EDF} - (m_{EDF} - 1) \cdot U_{EDF}^{max} - U_{EDF}^{total}$, where U_{EDF}^{max} and U_{EDF}^{total} are the maximal task utilization and total utilization for $\mathcal{T}_{EDF}$, respectively.*

This estimation can be applied with fpEDF-DVD-VPMC method described in Section 5.1.2.2. However, the partition of $\mathcal{T}_{hp}$ and $\mathcal{T}_{EDF}$ in Section 5.1.2.2 is different from that in Section 5.1.2.1 due to the virtual-deadlines applied to low-criticality tasks.

5.2.2.3 Utilization Slack Estimation for VPMC-DP-Fair Scheduling

The utilization slack for VPMC-DP-Fair Scheduling is estimated by $\Psi = m - \sum_{\tau_i \in \mathcal{T}} \theta_i^{HI}$, where θ_i^{HI} is the execution rate of task τ_i in high-criticality mode, which is computed according to [20].

5.3 Experimental Results

5.3.1 Evaluation of VPMC system scheduling methods

In this section, we evaluate our proposed methods for VPMC system scheduling. In our experiments, we evaluate the schedulability and computing errors of the following methods through software simulations and/or Linux prototyping:

- **Partition-MC**: Partitioned scheduling with the conventional MC model [10]. Since this method does not incorporate any approximations, its results are used to provide a reference level for schedulability, but cannot be used for comparing computing errors.

- **Partition-VPMC**: The partitioned scheduling method in Section 5.1.1.1 with precision optimization.
- **Partition-VPMC-E**: Enhanced partitioned scheduling (Section 5.1.1.2) with precision optimization.
- **fpEDF-VD-MC**: fpEDF-VD scheduling with the conventional MC model [10]. Since this method drops all low-criticality tasks in high-criticality mode, it is not included for error analysis.
- **fpEDF-VD-VPMC**: fpEDF-VD scheduling (Section 5.1.2.1) with precision optimization.
- **fpEDF-DVD-VPMC**: fpEDF dual virtual-deadline method (Section 5.1.2.2), with precision optimization.
- **Fluid-VPMC**: The MCFQ method [20] with precision optimization replaced by the dynamic programming technique in Section 5.2. Since the fluid-based scheduling is not directly implementable, this method is only evaluated with software simulation, to conduct the schedulability check and estimate error.
- **VPMC-DP-Fair**: The scheduling method described in Section 5.1.3 with precision optimization. Since this is an implementable realization of Fluid-VPMC, it is evaluated only through Linux prototyping.

5.3.1.1 Simulation Setup and Results

The testcases are randomly generated as follows:

- For each task set, the probability of a task being low-criticality (high-criticality) is 0.5.
- For a low-criticality (high-criticality) task τ_i, its utilization in low-criticality (high-criticality) mode U_i^{LO} (U_i^{HI}) is randomly chosen within the interval, $[0.05, 0.9]$, under a uniform distribution.
- The minimal inter-arrival time T_i of each task is randomly chosen from a uniform distribution in $[50, 500]$.

- For a low-criticality task τ_i, we set $C_i^{LO} = \hat{C}_i = T_i \cdot U_i^{LO}$, $\tilde{C}_i = k_{lo} \cdot \hat{C}_i$, where the scaling factor k_{lo} is randomly chosen from a uniform distribution in $[K_{lo}, 0.9]$, where K_{lo} is a parameter.
- For a high-criticality task τ_i, we set $C_i^{HI} = T_i \cdot U_i^{HI}$, $C_i^{HI} = k_{hi} \cdot C_i^{LO}$ and $1.1 \leq k_{hi} \leq K_{hi}$, where K_{hi} is a parameter.

For each low-criticality task τ_i, its imprecise computing error is randomly chosen from a uniform distribution between 1 and 10. We set error weighting factors (defined in Section 5.2) $\eta_i = 1$.

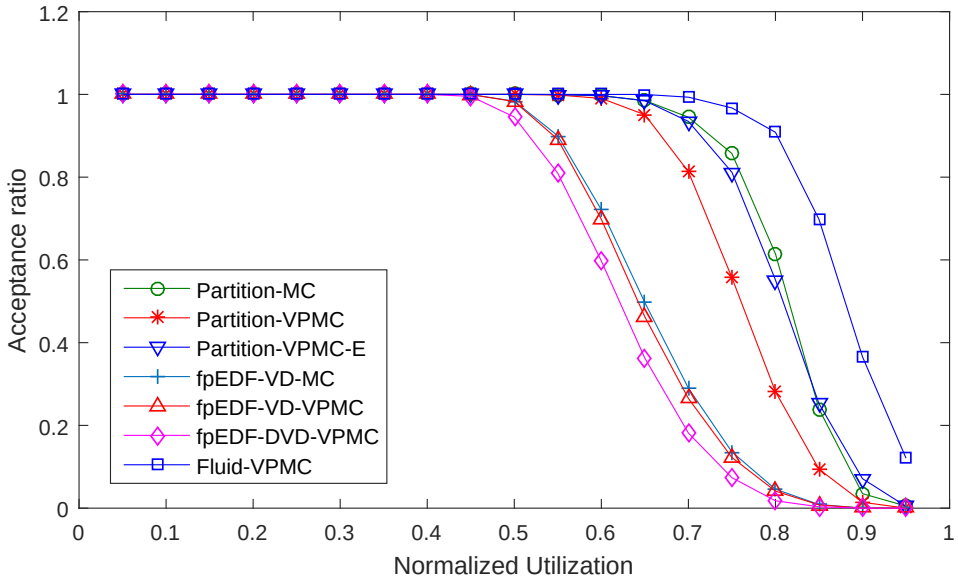

Figure 5.7: Acceptance ratio vs. normalized utilization of 4 processors ($K_{lo} = 0.1$, $K_{hi} = 5$).

Evaluation of the acceptance ratio: We first evaluate the acceptance ratio at several values of the utilization, U_i. For each U_i, we generate 10,000 testcases, and for each testcase, we iteratively add new tasks till $\max(U_{lo}^{LO} + U_{hi}^{LO},\ U_{lo}^{HI} + U_{hi}^{HI})$ reaches U_i. The acceptance ratios on 4 and 8 processors are depicted in Figures 5.7 and 5.8, respectively.

We see from these plots that Fluid-VPMC provides the best acceptance ratio (this is not surprising as the fluid-based scheduling is optimal in theory), while the three variants of fpEDF-VD

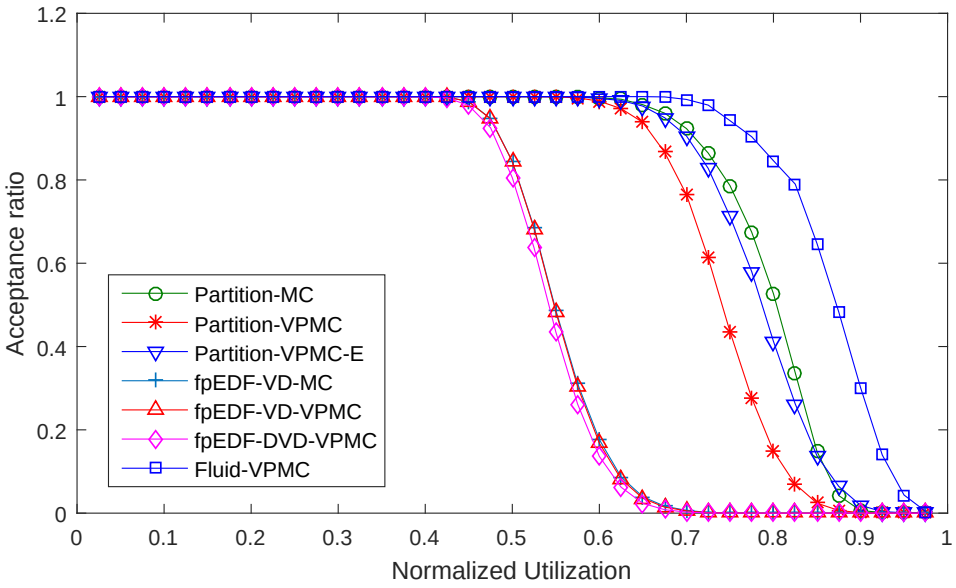

Figure 5.8: Acceptance ratio vs. normalized utilization of 8 processors ($K_{lo} = 0.1$, $K_{hi} = 5$).

have the lowest acceptance ratio due to their very conservative schedulability conditions. The acceptance ratio of fpEDF-VD-VPMC is very close to that of fpEDF-VD-MC, with an almost complete overlap in Figure 5.8. This implies that continuing low-criticality tasks at high-criticality mode hardly degrades schedulability. The dual virtual-deadline technique reduces acceptance ratio, but it guarantees that no low-criticality job is dropped while the fpEDF-VD-VPMC cannot provide such guarantees. The results also show that the enhancement techniques introduced in Section 5.1.1.2 can indeed improve schedulability of partitioned scheduling.

The simulations for Figures 5.7 and 5.8 do not consider overhead, which is important in practice. Overhead includes the time on context switching, job migration among processors, execution monitoring, scheduling computing, etc. For each of the VPMC methods, we estimate its overhead according to the Linux prototyping (Section 5.3.1.2) data. Then, the overhead is added into the task execution time for the simulation. The acceptance ratio results with consideration of overhead is shown in Figure 5.9. One can see that Fluid-VPMC is no longer the best due to its large overhead,

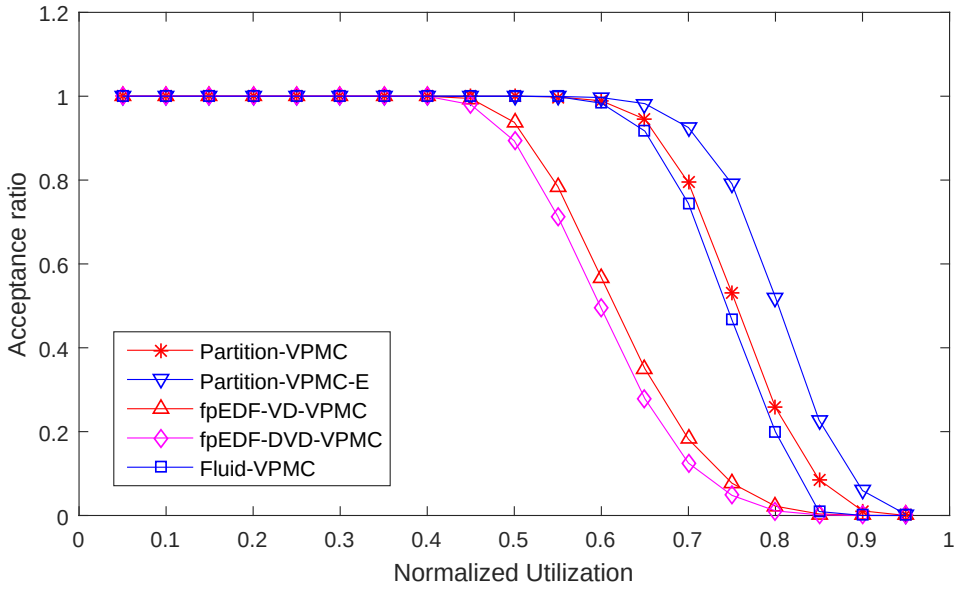

Figure 5.9: Acceptance ratio versus normalized utilization of 4 processors with consideration of overhead.

and the best results are obtained from partitioned scheduling. The gap between Fluild-VPMC and fpEDF-VD-VPMC also becomes smaller.

Evaluation of errors: Next, we evaluate computing errors of low-criticality tasks in high-criticality mode for different methods. Following the same testcase generation for evaluating the acceptance ratio, 1000 *schedulable* testcases are obtained at each utilization value. Figures 5.10 and 5.11 show the mean error with standard derivation among tasks as function of the normalized utilization. For a single testcase, minimizing mean error is equivalent to minimizing the total error as the number of tasks is a constant for the precision optimization. When evaluating multiple testcases, mean error is more like a normalized result that can avoid the result being dominated by a few cases. In both of the figures, errors from IMC is plotted besides those from other method. IMC is the model where all low-criticality tasks continue with imprecise computing in high-criticality mode. Hence, its error is the same for different scheduling methods. One can see that the VPMC model can provide large error reductions. Again, Fluid-VPMC provides the lowest error levels as its optimality allows more utilization slack for error reduction.

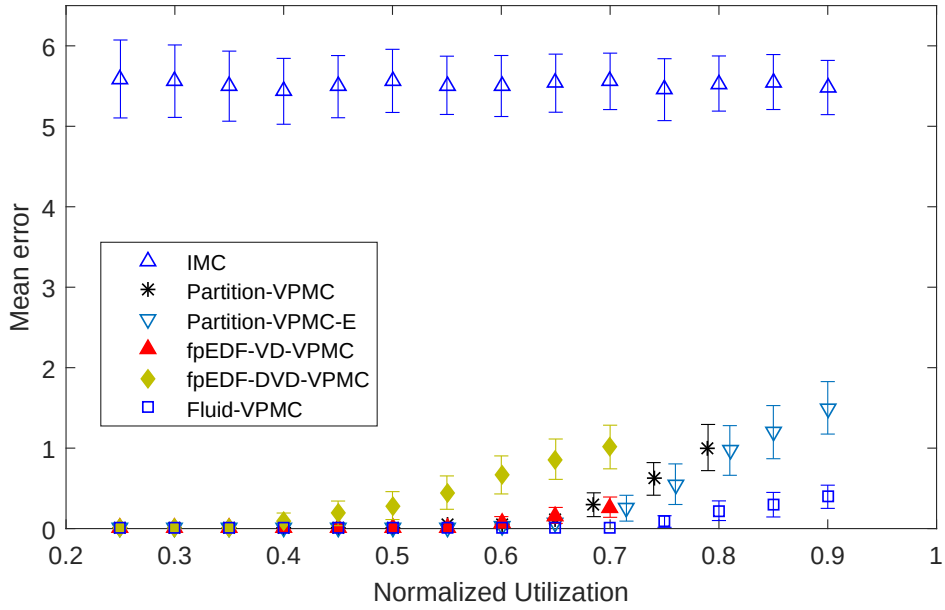

Figure 5.10: Mean error (with standard derivation) vs. normalized utilization of 4 processors ($K_{lo} = 0.1$, $K_{hi} = 5$).

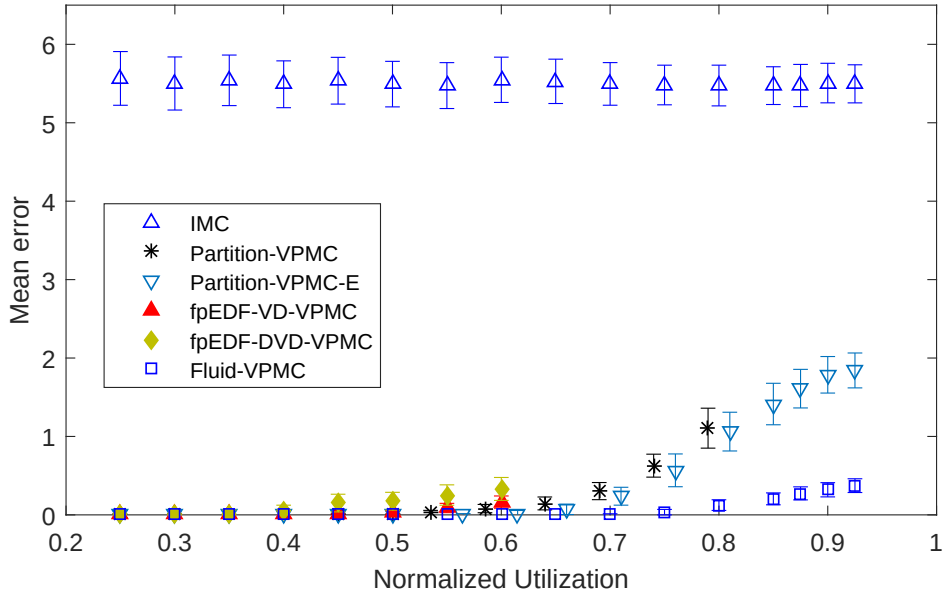

Figure 5.11: Mean error (with standard derivation) vs. normalized utilization of 8 processors ($K_{lo} = 0.1$, $K_{hi} = 5$).

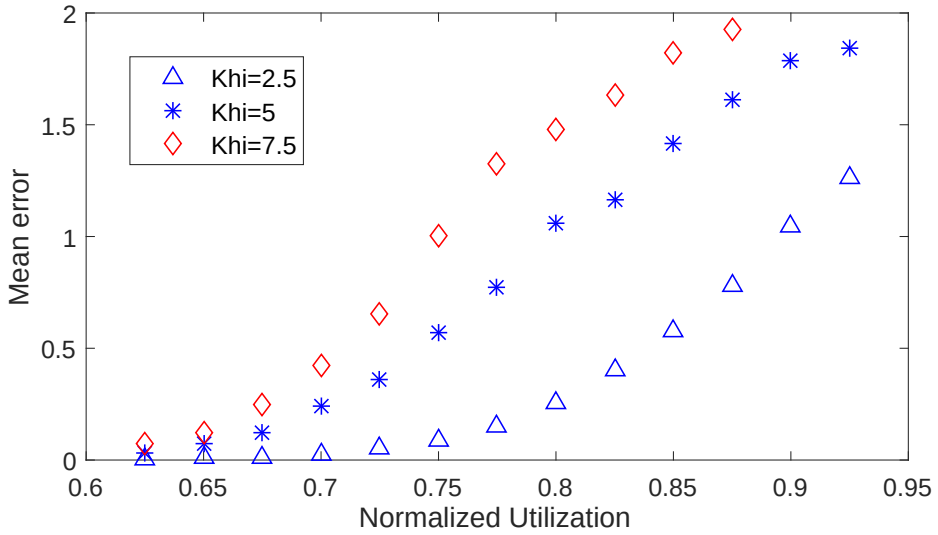

Figure 5.12: The effect of K_{hi} on errors for Partition-VPMC-E on 8 processors.

Figures 5.12 shows the effect of K_{hi} on errors for Partition-VPMC-E. In general, a large K_{hi} tends to cause large errors. We also studied the effect of parameter K_{lo} with result shown in Figure 5.13. Interestingly, the error grows as K_{lo} increases. For a large K_{lo}, the difference between precise and imprecise computing execution times is small, i.e., the additional utilization for changing imprecise computing to precise computing is small and applying precise computing becomes easier. On the other hand, a large K_{lo} increases the overall utilization and degrades schedulability.

5.3.1.2 Prototyping in Linux User Space

We evaluate the proposed techniques in the VPMC model with prototyping in Linux user space. Such prototyping can account for the overhead, which is neglected in software simulation. Moreover, the error model of the prototyping is more realistic. The prototyping is implemented in the user space of Linux 4.10 on a 4-processor machine, where each processor is Intel Core i3 with frequency 1.9GHz.

We create a managing thread that conducts the task scheduling at runtime. The managing thread can generate job threads. Each job thread is either in execution mode, or waiting mode when it is

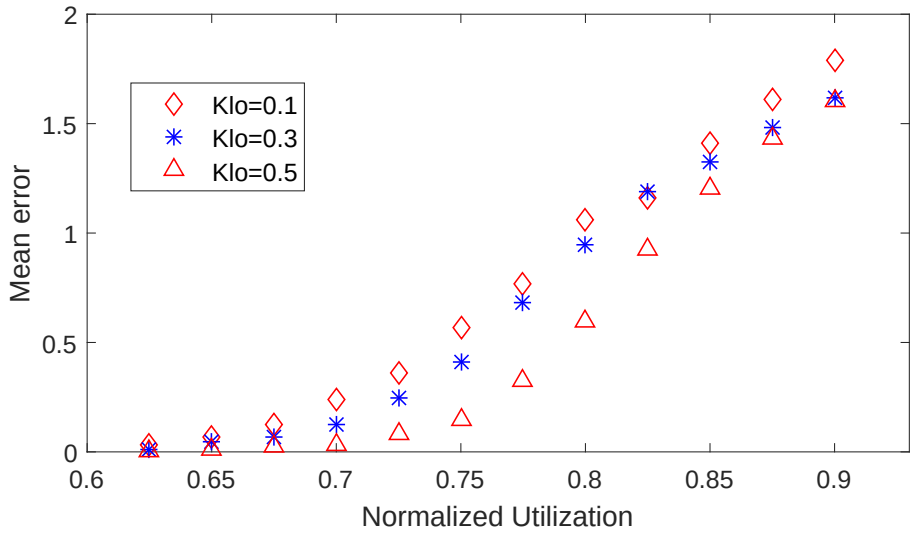

Figure 5.13: The effect of K_{lo} on errors for Partition-VPMC-E on 8 processors.

released, not completed and not being executed. The job thread management is performed at every time unit of 0.01 second, which allows sufficient resolution for the testcases. At each time unit, the managing thread checks if a new job is released, a job execution is completed, a job execution exceeds its WCET and if there is deadline violation. According to specific scheduling method, the managing thread decides if to start a waiting job, if a low-priority job being executed needs to be preempted, if to switch low-criticality mode to high-criticality mode, etc. Processor affinity is employed to assign a thread to certain processor.

For partitioned scheduling, the partition is performed offline in advance and there is one managing thread for each processor. For global scheduling like fpEDF-VD-VPMC and VPMC-DP-Fair, only one managing thread is needed for all processors. For the VPMC-DP-Fair method, the managing thread needs to maintain and update the deadline partitions, which are rounded to the time unit.

Two testcases are generated for the experiment in the Linux system. In the first case, all tasks are solving equations by the Newton-Raphson method. The second case is composed by tasks of both Newton-Raphson and the steepest decent method computing. For each case, tasks are

Table 5.2: Testcase characteristics for the Linux prototyping (the unit of execution time is second).

	Case 1				Case 2			
Task	χ_i	C_i^{LO}	C_i^{HI}	e_i	χ_i	C_i^{LO}	C_i^{HI}	e_i
τ_1	lo	0.76	0.47	20	lo	0.76	0.47	20
τ_2	hi	0.95	1.42	-	hi	0.95	1.42	-
τ_3	lo	0.67	0.21	0.5	lo	0.67	0.21	0.5
τ_4	hi	2.32	5.80	-	hi	2.32	5.80	-
τ_5	lo	1.65	0.98	5	lo	1.65	0.98	5
τ_6	hi	2.36	3.33	-	hi	2.36	3.33	-
τ_7	lo	1.40	0.71	8	lo	2.55	0.33	5
τ_8	hi	1.89	2.63	-	hi	4.10	5.10	-
τ_9	lo	0.76	0.27	5	lo	1.83	0.38	3
τ_{10}	lo	2.09	0.48	5	hi	1.07	2.31	-
τ_{11}	hi	0.47	0.73	-	lo	3.18	0.66	2
τ_{12}	-	-	-	-	hi	1.63	3.71	-

randomly designated with low-criticality or high-criticality. We run these cases repeatedly to find the maximum execution time of each task. The WCET is obtained by adding safety margins to the measured maximum execution time. Since both Newton-Raphson and the steepest decent method are iterative algorithms, their imprecise computing is realized by relaxing the termination criterion. The precise computation, which has a tight termination criterion, also results in a small errors, which is negligible in comparison with that of imprecise computing. The imprecise computing errors from low-criticality tasks are obtained from the results of the prototyped implementation. The characteristics of the two cases are summarized in Tables 5.2.

During the experiment, we vary the minimal inter-arrival time T_i of each task τ_i to obtain different utilizations. For each scheduling method, its schedulability condition is checked offline. For only the utilization conditions where the check is successfully passed, the tasks are run in the Linux system. In the first part of the Linux experiment, we investigate overhead including the time on context switching, job migration among processors, execution monitoring, scheduling computing, etc. In Figure 5.14, we show the average results of overhead ratio, which is the ratio of system time expended on the overhead to the total computation time. The VPMC-DP-Fair has quite large overhead, as large as 16% when utilization is high. Partitioned scheduling has the lowest overhead, and its overhead does not change much as the utilization increases. One advantage of

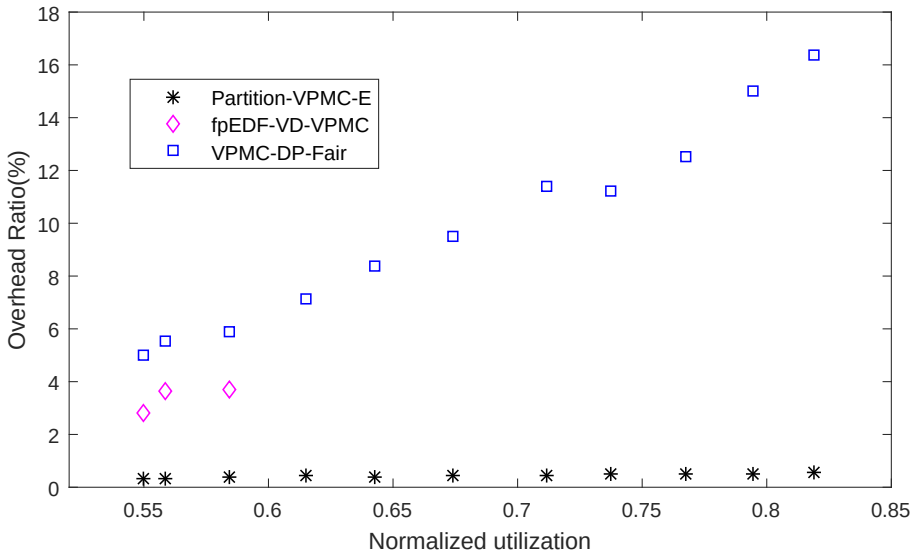

Figure 5.14: Overhead ratio vs. utilization (overhead includes the time on context switching, job migration among processors, execution monitoring, scheduling computing, etc.)

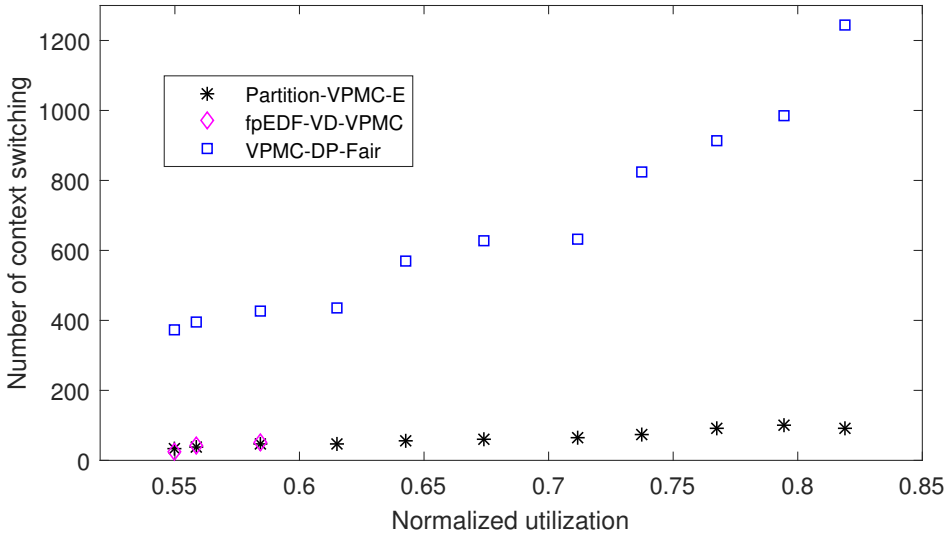

Figure 5.15: Number of context switchings vs. utilization.

partitioned scheduling is that there is no inter-processor migration overhead. Since the fpEDF-VD-VPMC method has relatively low schedulability, it does not produce much data for this figure. In Figure 5.15, we compare the numbers of context switchings for different methods. A large number implies large overhead and the results are indeed correlated with the overhead ratio in Figure 5.14.

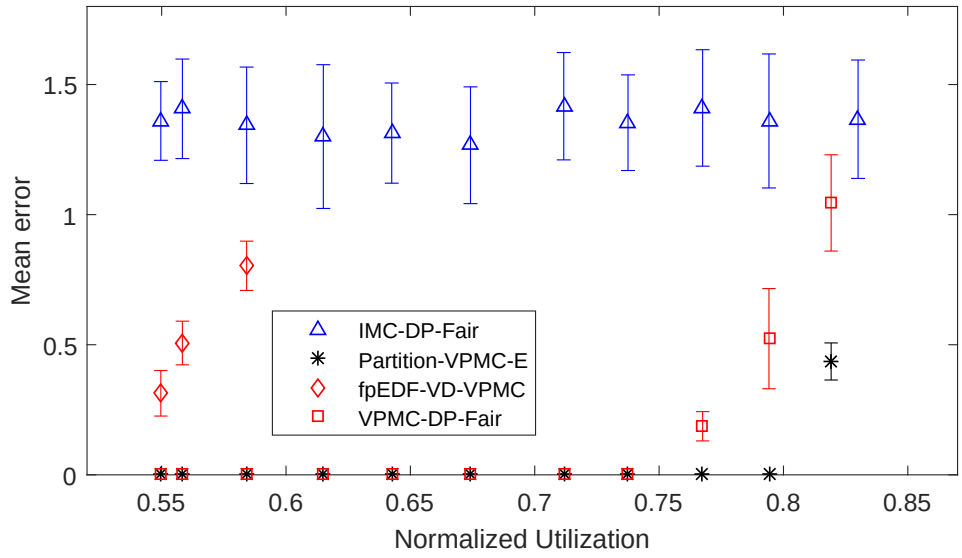

Figure 5.16: Mean error (with standard derivation) versus utilization from Linux prototyping.

We further evaluated the errors under imprecise computing for different methods on the two cases. The average results are plotted in Figure 5.16. The largest errors correspond to IMC-DP-Fair, where all low-criticality tasks are continued with imprecise computing in high-criticality mode, while the smallest errors are from Partition-VPMC-E. The errors from fpEDF-VD-VPMC are not good mostly due to its conservative schedulability, which does not allow much utilization slack for converting imprecise computing to precise computing for low-criticality tasks. The errors from VPMC-DP-Fair are generally small, but greater than Partition-VPMC-E. This is because its large overhead leads to lower slack for precise computing.

5.3.2 Evaluation of Service Preserving and Deferred Switching Techniques

In this section, we evaluate the effectiveness of our Service Preserving and Deferred Switching techniques. In the experiments, we evaluate and compare the following methods through software simulations.

- **Partitioning**: Partitioning-based scheduling method [10], where all low-criticality tasks are dropped in high-criticality mode.
- **fpEDF-VD**: fpEDF-VD scheduling [9], where all low-criticality tasks are dropped in high-criticality mode.
- **MC-DP-Fair**: The fluid-based MC-DP-Fair method [25], where all low-criticality tasks are dropped in high-criticality mode.
- **Dual-VD**: This is an extension to the fpEDF-VD scheduling such that virtual deadline is applied for both low-criticality and high-criticality tasks (Section 5.1.2.2). In this method, low-criticality tasks continue to execute with imprecise computing in high-criticality mode.
- **Deferred-Switching**: Our deferred switching method based on fpEDF-VD scheduling (Section 5.1.2.4). Low-criticality tasks are dropped in high-criticality mode.
- **Deferred-Switching-Apprx**: This method is the same as *Deferred-Switching* except that all low-critical tasks execute with imprecise computing in vigilant mode.
- **Service-Preserving**: Our service preserving method based on fpEDF-VD scheduling (Section 5.1.2.3). In this method, low-criticality tasks continue to execute with imprecise computing in high-criticality mode.
- **Unified**: The unified deferred switching and service preserving scheme based on fpEDF-VD scheduling (Section 5.1.2.5). In this method, low-criticality tasks continue to execute with imprecise computing in high-criticality mode.

5.3.2.1 Testcase Generation

The testcases in the experiments are randomly generated. For each testcase, the probability of a task being low-criticality (high-criticality) is 0.5. For each low-criticality task, we set its low-criticality mode utilization randomly in $[0.1, 0.9]$ under uniform distribution. Likewise, the high-criticality mode utilization of each high-criticality task is also randomly generated in $[0.1, 0.9]$, under uniform distribution. The minimal inter-arrival time T_i of each task τ_i is randomly chosen in $[100, 500]$ according to uniform distribution. For each low-criticality task τ_i, its execution times are set as $C_i^{LO} = T_i \cdot U_i^{LO}$ and $C_i^{HI} = k_L \cdot C_i^{LO}$, where the scaling factor k_L is randomly chosen in $[0.1, 0.9]$ following uniform distribution. For each high-criticality task τ_j, we set its high-criticality mode execution time $C_j^{HI} = T_j \cdot U_j^{HI}$. Its low-criticality mode execution time is obtained according to $C_j^{HI} = k_H \cdot C_j^{LO}$, where $1.1 \leq k_H \leq 7.5$.

5.3.2.2 Evaluation of Service Preserving

We evaluated the acceptance ratio of our service preserving technique and compared with the dual-VD method. Please note our unified scheme should have the same acceptance ratio as that of service preserving method, as both apply the same offline schedulability test. The deferred switching part is an online technique that does not affect the acceptance ratio. The other methods are not compared as they all drop low-criticality tasks in high-criticality mode. Therefore, they are not comparable with our method, which retains low-criticality tasks in high-criticality mode. The results for 4 processors and 8 processors are shown in Figure 5.17 and Figure 5.18, respectively. At each utilization, 10,000 testcases are randomly generated for evaluation. The difference between the two methods mainly exhibit around utilization 0.6, where our service preserving can improve as much as 50%.

5.3.2.3 Evaluation of Deferred Switching and Unified Method

In this part of experiment, we compare our deferred switching and the unified method with several previous works. The comparison is performed on two metrics. One is the number of completed low-criticality jobs before switching to high-criticality mode and the other is the mode

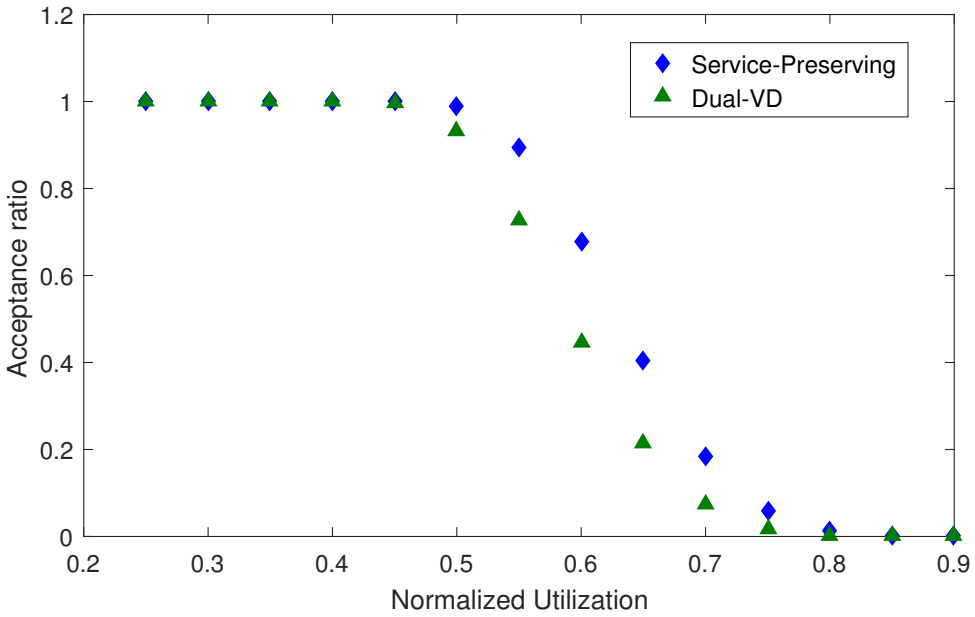

Figure 5.17: Acceptance ratio vs normalized utilization of 4 processors.

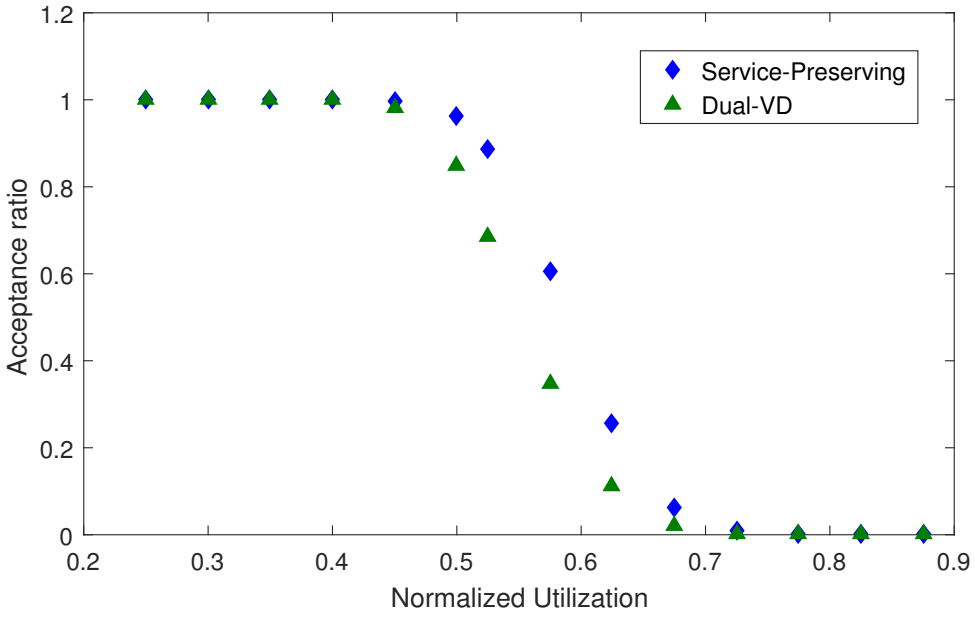

Figure 5.18: Acceptance ratio vs normalized utilization of 8 processors.

switching time. Compared to methods that drop low-criticality tasks in high-criticality mode, the deferred switching or more completed low-criticality jobs improves overall QoS. For each utilization, 1000 *schedulable* testcases are generated and each case is simulated 10 times to account for the random job execution time.

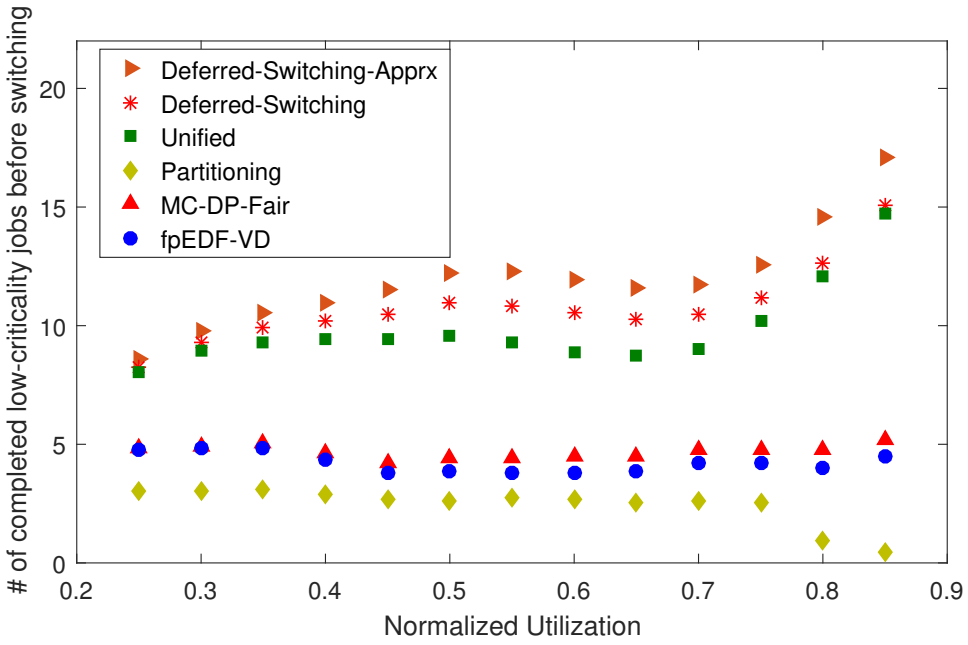

Figure 5.19: Number completed low-criticality jobs before mode switching vs normalized utilization of 4 processors, with overrun rate 0.2.

Figures 5.19 and 5.20 show the number of completed low-criticality jobs before mode switching for 4 processors with overrun rate of 0.2 and 0.5, respectively. The *overrun rate* is the probability that a high-criticality job has overrun. Both our deferred switching and unified methods complete significantly more low-criticality jobs than the previous works. Since the unified method needs to continue low-criticality jobs in high-criticality mode, it is more conservative than the stand alone deferred switching method. When the utilization is high, task minimal inter-arrival time T is relatively short. Consequently, the number of low-criticality jobs active in vigilant mode becomes large and the number of completed low-criticality jobs also increase accordingly. Therefore, the

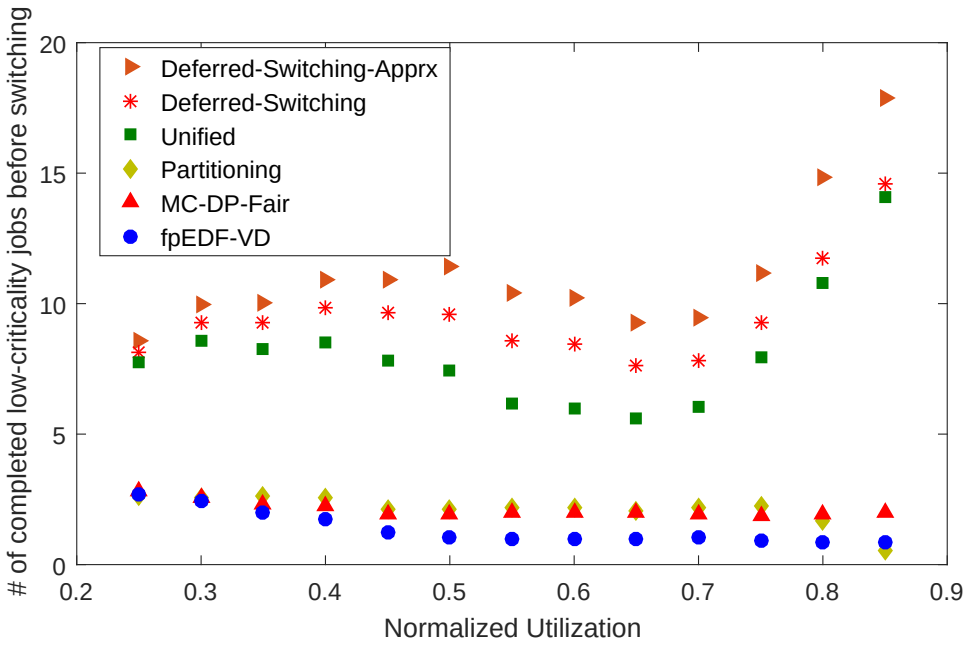

Figure 5.20: Number of completed low-criticality jobs before mode switching vs normalized utilization of 4 processors, overrun rate 0.5.

advantage of our proposed method is much more significant in heavily loaded cases. The results for 8-processor cases are shown in Figure 5.21 and 5.22, and similar trends as 4-processor cases can be observed, since the schedulability condition on 8-processor is relatively more conservative and it does not produce so much data as 4-processor. These figures also indicate that performing low-criticality tasks with imprecise computing in vigilant mode allows more low-criticality jobs to be completed than precise computing. Hence, our framework provides options with different tradeoff between computing precision and number of completed jobs.

In Figures 5.23 and 5.24, we compare the switching time to high-criticality mode among different methods. The overrun rate is 0.5 for the cases in both of the figures. Again, our methods can delay the switching for a long time compared to the previous works. This delay reduces the QoS loss in high-criticality mode. In choosing the value for x, which is the scaling factor for virtual deadline, we sweep from a small value toward large values, and stop whenever schedulability conditions are satisfied. When utilization is low, even small x values are sufficient for schedulability. When utilization is high, x must be chosen as greater or more balanced value. Since the

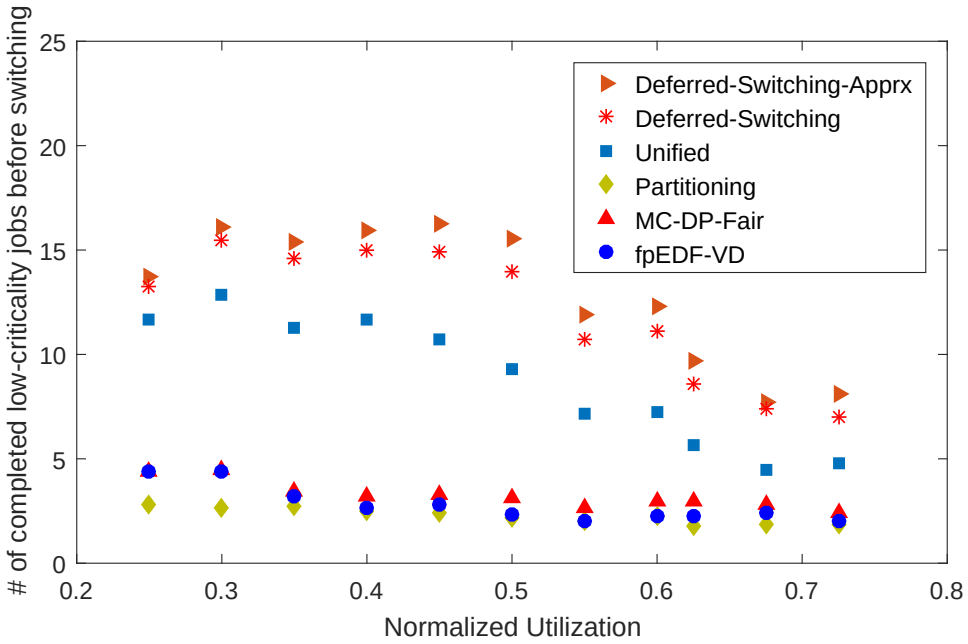

Figure 5.21: Number of completed low-criticality jobs before mode switching vs normalized utilization of 8 processors with overrun rate 0.2.

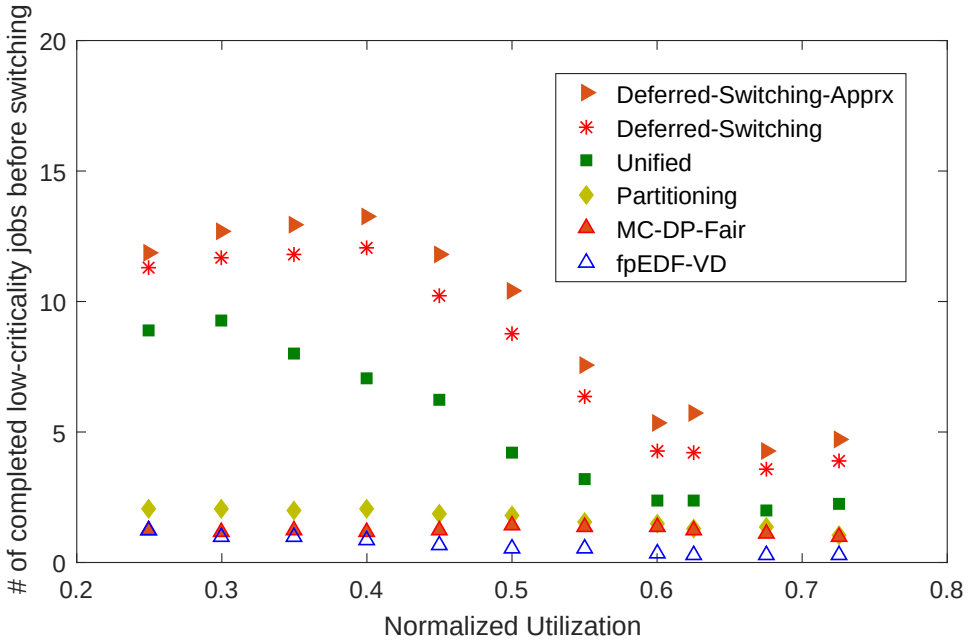

Figure 5.22: Number of completed low-criticality jobs before mode switching vs normalized utilization of 8 processors, with overrun rate 0.5.

checkpoints in deferred switching and the unified method largely depend on virtual deadlines, a relatively late virtual deadline contributes to greater deferral by our methods. By imprecise computing for low-criticality jobs in vigilant mode, the amount of deferral is increased according to the results shown in the figures.

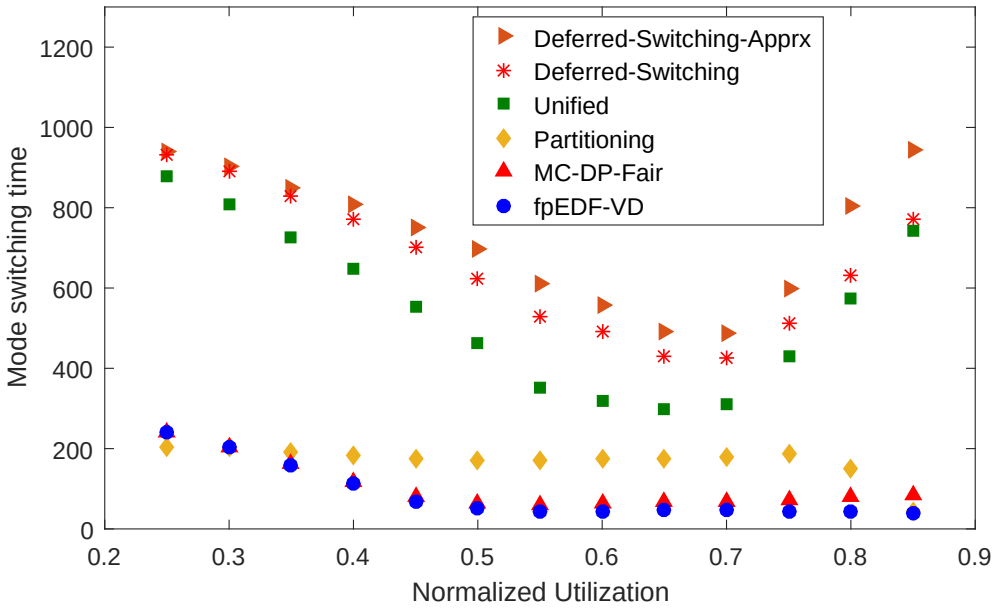

Figure 5.23: Mode switching time vs normalized utilization of 4 processors.

5.4 Conclusion

The conventional mixed-criticality system model, despite its popularity, is controversial as it drops all low-criticality tasks in high-criticality mode. Moreover, a single high-criticality job overrun causes immediate switching to high-criticality mode where all high-criticality tasks are scheduled with overly pessimistic execution time estimate. Most of previous works address these problems for uniprocessors while we focus on scheduling on multiprocessors. Recently, there are a few works for overcoming the drawback of dropping low-criticality tasks by continuing low-criticality tasks with imprecise computing or even precise computing. In our research, we

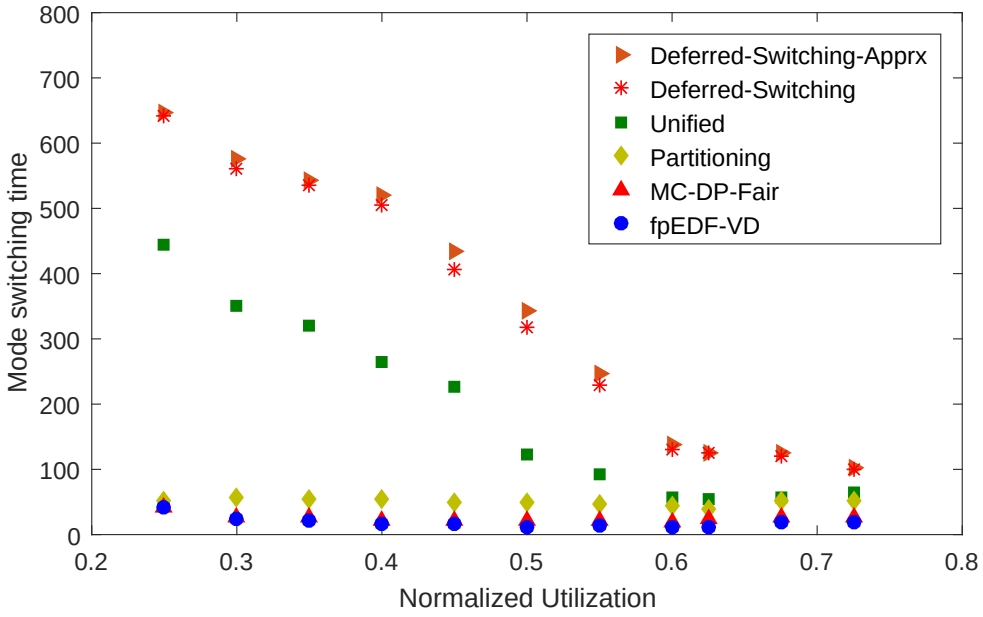

Figure 5.24: Mode switching time vs normalized utilization of 8 processors.

develop such graceful degradation techniques for partitioned scheduling and fpEDF-VD scheduling on multiprocessors. The proposed techniques are evaluated with both software simulations and Linux prototyping where overhead is considered. The results show that the graceful degradation approach can significantly improve computing quality with little sacrifice on schedulability. When the overhead is considered, the proposed partitioned scheduling outperforms the previous approach of fluid-based scheduling. Moreover, we developed a service preserving technique in the fpEDF-VD framework that allows all low-criticality tasks to execute with imprecise computing in high-criticality mode. It is less conservative and remarkably improves acceptance ratio compared to the method of dual virtual deadline. And a vigilant mode and online checkpoint method is proposed to deferred the switching to high-criticality mode. These two techniques are further unified into a single method. These techniques significantly improve Quality of Service (QoS) for low-criticality tasks in mixed-criticality systems.

6. SUMMARY AND CONCLUSIONS [1]

In our research, we work on using imprecise computing to improve real-time scheduling. We first investigate how to use imprecise computing for non-preemptive real-time scheduling, and propose several heuristic algorithms for scheduling periodic tasks with independent errors or cumulative errors. Our algorithms can guarantee that there is no deadline violation if a task set can pass the initial schedulability check. Meanwhile, our algorithms can either guarantee certain error bound due to imprecision offline or minimize error online in the best effort. Experiment results from both simulation and Linux prototyping show that our algorithms can improve schedulability and provide desired error and deadline tightness tradeoff.

Imprecise computing can provide graceful degradation for mixed-criticality system as well. Low-criticality tasks can be executed in imprecise computing or even precise computing in high-criticality mode rather than being dropped as in conventional mixed-criticality system model. We develop such graceful technique for well-know multiprocessor scheduling methods such as partitioned scheduling and global scheduling. These proposed algorithms are evaluated through simulation and Linux prototyping with consideration of overhead. Moreover, in the conventional mixed-criticality model, a single high-criticality job overrun causes immediate switching to high-criticality mode where all high-criticality tasks are scheduled with overly pessimistic WCET. We work on addressing both limitations for global scheduling on multiprocessors. We develop two techniques including service preserving technique and deferred switching technique to improve the Quality of Service (QoS) for low-criticality tasks. Our techniques can improve schedulability and allow more low-criticality tasks to be executed compared to previous works.

[1] Reprinted with permission from "Using imprecise computing for improved non-preemptive real-time scheduling " by Lin Huang, Youmeng Li, Sachin S.Sapatnekar, Jiang Hu, 2018. *Proceedings of Design Automation Conference (DAC)*, Page 1-6 , ©2018 IEEE, "Graceful degradation of low-criticality tasks in multiprocessor dual-criticality systems " by Lin Huang, I-Hong Hou, Sachin S.Sapatnekar, Jiang Hu, 2018. *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*, Page 159-169 , ©2018 ACM and "Improving QoS for global dual-criticality scheduling on multiprocessors " by Lin Huang, I-Hong Hou, Sachin S.Sapatnekar, Jiang Hu, 2019. *Proceedings of the International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Page 1-11, ©2019 IEEE.

REFERENCES

[1] J. Han and M. Orshansky, "Approximate computing: an emerging paradigm for energy-efficient design," in *Proceedings of European Test Symposium (ETS)*, pp. 1–6, IEEE, 2013.

[2] J. W. Liu, K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao, *Algorithms for Scheduling Imprecise Computations*, pp. 203–249. Springer US, 1991.

[3] J.-Y. Chung and J. W. S. Liu, "Algorithms for scheduling periodic jobs to minimize average error," in *Proceedings of Real-Time Systems Symposium (RTSS)*, pp. 142–151, IEEE, 1988.

[4] H. Aydin, R. Melhem, and D. Mosse, "Optimal scheduling of imprecise computation tasks in the presence of multiple faults," in *Proceedings of International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pp. 289–296, IEEE, 2000.

[5] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of periodic and sporadic tasks," in *Proceedings of Real-Time Systems Symposium (RTSS)*, pp. 129–139, IEEE, 1991.

[6] R. Jejurikar and R. Gupta, "Energy aware non-preemptive scheduling for hard real-time systems," in *Proceedings of Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 137–144, IEEE, 2005.

[7] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degree of execution time assurance," in *Proceedings of Real-Time Systems Symposium (RTSS)*, pp. 239–243, IEEE, 2007.

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

[9] H. Li and S. Baruah, "Global mixed-criticality scheduling on multiprocessors," in *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 166–175, IEEE, 2012.

[10] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin, "Mixed-criticality scheduling on multiprocessors," *Real-Time Systems*, vol. 50, no. 1, pp. 142–177, 2014.

[11] S. Baruah and Z. Guo, "Mixed-criticality job models: a comparison," *Workshop on Mixed-Criticality Systems*, 2015.

[12] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, p. 82, 2017.

[13] H. Su and D. Zhu, "An elastic mixed-criticality task model and its scheduling algorithm," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 147–152, IEEE, 2013.

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

[15] S. Ramanathan and A. Easwaran, "Mixed-criticality scheduling on multiprocessors with service guarantees," in *Proceedings of International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 17–24, IEEE, 2018.

[16] H. Su, D. Zhu, and D. Moss, "Scheduling algorithms for elastic mixed-criticality tasks in multicore systems," in *Proceedings of International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 352–357, IEEE, 2013.

[17] A. Burns and S. Baruah, "Towards a more practical model for mixed criticality systems," *Workshop on Mixed-Criticality Systems*, 2013.

[18] S. Baruah, A. Burns, and Z. Guo, "Scheduling mixed-criticality systems to guarantee some service under all non-erroneous behaviors," in *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 131–138, IEEE, 2016.

[19] D. Liu, J. Spasic, N. Guan, G. Chen, S. Liu, T. Stefanov, and W. Yi, "EDF-VD scheduling of mixed-criticality systems with degraded quality guarantees," in *Proceedings of Real-Time Systems Symposium (RTSS)*, pp. 35–46, IEEE, 2016.

[20] R. M. Pathan, "Improving the quality-of-service for scheduling mixed-criticality systems on multiprocessors," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 76, 2017.

[21] J. Lee, H. S. Chwa, L. T. Phan, I. Shin, and I. Lee, "Mc-adapt: Adaptive task dropping in mixed-criticality scheduling," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 163, 2017.

[22] G. Chen, N. Guan, B. Hu, and W. Yi, "EDF-VD scheduling of flexible mixed-criticality system with multiple-shot transitions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2393–2403, 2018.

[23] Z. Al-bayati, Q. Zhao, A. Youssef, H. Zeng, and Z. Gu, "Enhanced partitioned scheduling of mixed-criticality systems on multicore platforms," in *Proceedings of Asia and South Pacific Design Automation Conference (ASPDAC)*, pp. 630–635, IEEE, 2015.

[24] J. Ren and L. T. X. Phan, "Mixed-criticality scheduling on multiprocessors using task grouping," in *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 25–34, IEEE, 2015.

[25] J. Lee, K.-M. Phan, X. Gu, J. Lee, A. Easwaran, I. Shin, and I. Lee, "MC-Fluid: Fluid model-based mixed-criticality scheduling on multiprocessors," in *Proceedings of Real-Time Systems Symposium (RTSS)*, pp. 41–52, IEEE, 2014.

[26] S. Baruah, A. Eswaran, and Z. Guo, "MC-Fluid: simplified and optimally quantified," in *Proceedings of Real-Time Systems Symposium (RTSS)*, pp. 327–337, IEEE, 2015.

[27] R. Ernst and M. Di Natale, "Mixed criticality systems a history of misconceptions?," *IEEE Design and Test*, vol. 33, no. 5, pp. 65–74, 2016.

[28] L. Sigrist, G. Giannopoulou, P. Huang, A. Gomez, and L. Thiele, "Mixed-criticality runtime mechanisms and evaluation on multicores," in *Proceedings of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 194–206, IEEE, 2015.

[29] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-power digital signal processing using approximate adders," *IEEE Transactions on Computer-Aided Design*, vol. 32, pp. 124–137, Jan. 2013.

[30] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy, "Design of voltage-scalable meta-functions for approximate computing," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 1–6, IEEE, 2011.

[31] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On reconfiguration-oriented approximate adder design and its application," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 48–54, IEEE, 2013.

[32] S. Venkataramani, V. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pp. 1–12, 2013.

[33] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pp. 449–460, 2012.

[34] V. Chippa, A. Raghunathan, K. Roy, and S. Chakradhar, "Dynamic effort scaling: Managing the quality-efficiency tradeoff," in *Design Automation Conference (DAC)*, pp. 603–608, IEEE, 2011.

[35] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, pp. 46–61, Jan. 1973.

[36] S. Ramanathan and A. Easwaran, "Utilization difference based partitioned scheduling of mixed-criticality systems," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 238–243, IEEE, 2017.

[37] S. Baruah, "Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors," *IEEE Transactions on Computers*, vol. 53, no. 6, pp. 781–784, 2004.

[38] S. Funk, G. Levin, C. Sadowski, I. Pye, and S. Brandt, "DP-Fair: a unifying theory for optimal hard real-time multiprocessor scheduling," *Real-Time Systems*, vol. 47, no. 5, pp. 389–429, 2011.

[39] W. Shih, J. S. W. Liu, J. Chung, and D. W. Gillies, "Scheduling tasks with ready times and deadlines to minimize average error," *ACM SIGOPS Operating Systems Review*, vol. 23, pp. 14–28, July 1989.

[40] J.-Y. Chung, J. W. S. Liu, and K.-J. Lin, "Scheduling periodic jobs that allow imprecise results," *IEEE Transactions on Computers*, vol. 39, pp. 1156–1174, Sept. 1990.

[41] W.-K. Shih, J. W. S. Liu, and J.-Y. Chung, "Algorithms for scheduling imprecise computations with timing constraints," *SIAM Journal on Computing*, vol. 20, pp. 537–552, June 1991.

[42] C. Lee, W. Ryu, K. Song, K. Choi, G. Jung, and S. Park, "On-line scheduling algorithms for reducing the largest weighted error incurred by imprecise tasks," in *International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pp. 21–30, IEEE, 1998.

[43] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez, "Optimal reward-based scheduling for periodic real-time tasks," *IEEE Transactions on Computers*, vol. 50, pp. 111–130, Feb. 2001.

[44] C. Tan, T. S. Muthukaruppan, T. Mitra, and L. Ju, "Approximation-aware scheduling on heterogeneous multi-core architectures," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 618–623, IEEE, 2015.

[45] J. Yi, Q. Zhang, Y. Tian, T. Wang, W. Liu, E. H.-M. Sha, and Q. Xu, "ApproxMap: on task allocation and scheduling for resilient applications," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 318–323, IEEE, 2015.

[46] S. K. Baruah, "The non-preemptive scheduling of periodic tasks upon multiprocessors," *Real-Time Systems*, vol. 32, no. 1, pp. 9–20, 2006.

[47] N. Guan, W. Yi, Z. Gu, Q. Deng, and G. Yu, "New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms," in *Proceedings of Real-Time Systems Symposium (RTSS)*, pp. 137–146, 2008.

[48] A. Thekkilakattil, R. Dobrin, and S. Punnekkat, "Quantifying the sub-optimality of non-preemptive real-time scheduling," in *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 113–122, IEEE, 2013.

[49] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," *ACM SIGOPS Operating System Review*, vol. 35, pp. 89–102, Oct. 2001.

[50] F. Zhang and S. T. Chanson, "Processor voltage scheduling for real-time tasks with non-preemptible sections," in *Proceedings of Real-Time Systems Symposium (RTSS)*, pp. 235–245, IEEE, 2002.

[51] J. Mao, C. G. Cassandras, and Q. Zhao, "Optimal dynamic voltage scaling in energy-limited nonpreemptive systems with real-time constraints," *IEEE Transactions on Mobile Computing*, vol. 6, pp. 678–688, June 2007.

[52] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *Proceedings of Real-Time Systems Symposium (RTSS)*, pp. 34–43, IEEE, 2011.

[53] I. Bate, A. Burns, and R. I. Davis, "A bailout protocol for mixed criticality systems," in *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 259–268, IEEE, 2015.

[54] 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 International conference on Real-Time Networks and Systems (RTNS)*, pp. 183–192, ACM, 2013.