Partitioned Multiprocessor Scheduling of Mixed-Criticality Parallel Jobs

Guangdong Liu
University of Nebraska-Lincoln

Ying Lu
University of Nebraska-Lincoln

Shige Wang
General Motor Global Research and Development

Zonghua Gu
Zhejiang University

Follow this and additional works at: http://digitalcommons.unl.edu/cseconfwork
Abstract—Motivated by the increasing trend in embedded systems towards platform integration, there has been an increasing research interest in scheduling mixed-criticality systems. However, most existing efforts have concentrated on scheduling sequential tasks and ignored intra-task parallelism. In this paper, we study the scheduling of mixed-criticality parallel jobs on multiprocessor platforms. We propose a synchronous mixed-criticality job model, where each job consists of segments, each segment having an arbitrary number of parallel threads that synchronize at the end of the segment. A novel MinLoad algorithm is developed to decompose mixed-criticality parallel jobs into mixed-criticality sequential jobs. This decomposition enables us to leverage existing mixed-criticality scheduling algorithms and schedulability analysis to the multiprocessor scheduling of mixed-criticality parallel jobs. In addition, our MinLoad job decomposition algorithm is designed to make the decomposed mixed-criticality sequential tasks easier to schedule, and thus requires smaller-sized multiprocessor platforms for the mixed-criticality systems.

I. INTRODUCTION

Modern large real-time and embedded systems, such as those in avionics, automotive and robotics applications, typically comprise many diverse functions with different levels of criticality, or importance. Traditional approaches implement the system using a federated architecture. In such an architecture, software control components of different criticality levels have separate, dedicated devices for their execution. With only functionalities of the same criticality level sharing a computing system, all associated cost of acquisition, space, power, weight, cooling, installation, and maintenance increases. For better cost and efficiency, an increasing trend in embedded system design is to integrate applications and components of different criticality levels onto a common hardware platform. However, such mixed-criticality (MC) systems are subject to certifications of varying degrees of rigorousness, for validating the correctness of different subsystems on various confidence levels. For instance, in comparison with the system designer in designing, implementing, and testing the system, for certification the certification authority (CA) often mandates far more conservative assumptions about the worst-case behavior of the system. While the CA is only concerned with the correctness of the safety-critical part of the system, the system designer is responsible for ensuring that the entire system is correct, including the non-critical parts [18].

As research progresses in understanding MC systems, real-time scheduling of certifiable MC systems has been recognized to be a challenging problem. Initially, a number of papers considered uniprocessor MC scheduling and analysis [9], [45], [63], [53], [17], [62], [30]. With platforms of real-time and embedded systems migrating from single cores to multi-cores and, in the future, many-core architectures [21], researchers have begun to investigate multiprocessor MC scheduling [40], [46], [56], [18]. However, most existing efforts have concentrated on inter-task parallelism, where each task runs sequentially (and therefore can only run on a single core) and multiple cores are exploited by increasing the number of tasks. As pointed out by Li et al. [47], when a model is limited to inter-task parallelism, each individual task’s total execution requirement must be smaller than its deadline since individual tasks cannot run any faster than on a single-core machine. In order to enable tasks with higher execution demands and tighter deadlines, such as those used in autonomous vehicles, video surveillance, radar tracking, and robotic systems, we must enable parallelism within tasks [38], [68], [47]. Moreover, for many mixed-criticality applications such as autonomous driving, integration and cooperation of control functions are essential. With the architecture that consolidates relevant functions from cooperating controls into the same task to minimize runtime overhead, traditional inter-task parallelism seems too coarse of granularity, sometime even yields diminishing throughput of the system. As an example, considering a controller with consolidated powertrain control and driver assistance [58], when engaged, the driver assistance function interacts with powertrain control to accelerate or decelerate a vehicle. It is common that the functions from both controls requiring synchronization (e.g. the object tracking and the sign detection in the driver assistance and the speed control from the powertrain must be synchronized to achieve collision avoidance) are aggregated into a single task. In such a system, while there are many parallel operations, creating them as separate tasks will result in either too many tasks or long execution delay due to cross-task synchronization. Thus, it is desired to have intra-task parallelism in mixed-criticality systems to allow dynamic decision of parallel execution without introducing unnecessary waiting for synchronization. To fill in this research gap, this paper investigates multiprocessor scheduling of mixed-criticality parallel jobs.
There are two types of multiprocessor real-time scheduling [24]: global and partitioned scheduling. In the global scheduling [6], all eligible tasks are assembled into a single queue, from which the global scheduler selects tasks for execution. On the contrary, the partitioned approach allocates each task (or subtask) to a single processor, and processors are scheduled independently [7]. Most existing multiprocessor real-time scheduling approaches assume sequential tasks [24]. There have been some recent progresses on scheduling real-time parallel tasks on multiprocessors [47], [60], [61], [52]. They have, however, investigated regular, rather than MC, task systems. Multiprocessor scheduling of parallel tasks are further categorized into two types: one group of techniques schedules parallel tasks directly and the other first decomposes each parallel task into a set of sequential tasks and then applies a sequential task scheduling method to schedule decomposed tasks. Since a partitioned algorithm allocates each subtask to a single processor and then schedules processors independently, a partitioned algorithm must involve a task decomposition to exploit intra-task parallelism. In contrast, a global scheduler could be designed with or without a task decomposition.

Due to simplicity in design and implementation, partitioned approaches are often considered to be more practical than global scheduling approaches [5], [64], [32]. Thus, in this paper we focus on the development of a partitioned approach to schedule MC parallel jobs on multiprocessors.

We first present the related work in Section II. Then, a synchronous MC job model is proposed in Section III, followed by the problem formulation in Section IV. Section V briefly introduces the OCBP (Own Criticality-Based Priorities) algorithm that is adopted for scheduling MC sequential jobs on each processor. Section VI describes the algorithms and we present the simulation in Section VII. Section VIII concludes the paper.

II. RELATED WORK

Motivated by the increasing trend in embedded systems towards platform integration [66] as is evidenced by industry-wide initiatives such as IMA (Integrated Modular Avionics) for aerospace, and AUTOSAR (AUTomotive Open System ARChitecture) for the automotive industry, there has been an increasing research interest in scheduling mixed-criticality systems [21] and implementing these scheduling algorithms [35], [34]. Initially, researchers considered the problem of scheduling on a single processor a finite set of MC sequential jobs with criticality dependent execution times [9], [19], [45], [8], [55], [63]. This work has then been extended to scheduling a set of recurrent MC sequential jobs on a single processor [53], [39], [27], [44], [33], [17], [14], [62], [30], [29] and on multiprocessors [40], [65], [46], [56], [18], [59]. Recently, researchers began to investigate even more general MC task model [31], where not only task execution time depends on the criticality level, task period [11], [16], [22] and task deadline [31] could also be criticality dependent. In addition to schedulability analysis, some researchers investigated the issues of robustness, where task execution times are allowed to exceed their worst case estimates as long as the system remains schedulable [62], [34]. However, most existing work on scheduling mixed-criticality systems are limited to sequential programming models and ignore intra-task parallelism. Only a couple of researchers investigated the topic of scheduling MC parallel tasks [12], [13], [65]. Nevertheless, all these research efforts are based on static scheduling. To the best of our knowledge, this paper is the first study of priority-based scheduling of MC parallel jobs on multiprocessors.

There have been intensive research on scheduling parallel tasks without deadlines (e.g., [57], [67], [28], [26], [3]) and on scheduling real-time parallel tasks with soft deadlines [23], [4], [37], [48] and with hard deadlines [47], [60], [61], [41], [15], [42], [25], [51], [36], [20], [54], including our prior work on hard real-time scheduling of arbitrarily divisible tasks on multiprocessors [49], [50]. In earlier work on hard real-time scheduling of parallel tasks, researchers made simplifying assumptions about task models [42], [25], [51] or focused on a special type of parallel tasks [49], [50]. It is until recently that researchers began to investigate more realistic task models like synchronous [36], [41], [60], [52] and DAG task(s) [61], [47], [15], [20], [54]. They have, however, investigated regular (single-criticality), rather than mixed-criticality task systems.

III. JOB MODEL

This section formally defines the mixed-criticality parallel job model used in this paper. Although this paper focuses on the case where a system is comprised of a finite number of MC parallel jobs, the ideas and insights gained in this work will be extended to scheduling a set of recurrent MC parallel tasks in the future.

We consider a synchronous job model, where each parallel job consists of many computation segments, and each segment may contain many parallel threads which synchronize at the end of the segment. As been pointed out by Saifullah et al. [60], such tasks are generated by parallel loops, a construct common to many parallel languages such as OpenMP [1] and Intel’s CilkPlus [2].

A set of $n$ mixed-criticality (MC) synchronous parallel jobs $\tau = \{J_1, J_2, ..., J_n\}$ is assumed, where each MC job is characterized by a tuple of parameters: $J_i = ((J_i^1, J_i^2, ..., J_i^n), A_i, D_i, \chi_i)$, where

- $(J_i^1, J_i^2, ..., J_i^n)$ denotes job $J_i$ has $s_i$ number of segments, to be executed in sequence. That is, all threads of segment $k$ must complete before any thread of segment $k+1$ can start.
- $A_i \in \mathbb{R}^+$ is the release time.
- $D_i \in \mathbb{R}^+$ is the absolute deadline. We assume that $D_i \geq A_i$.
- $\chi_i \in \{LO, HI\}$ denotes the criticality. A HI-criticality job (a $J_i$ with $\chi_i = HI$) is one that is subject to certification, whereas a LO-criticality job (a $J_i$ with $\chi_i = LO$) is one that does not need to be certified.

Job $J_i$’s segment is also characterized by a tuple: $J_i^k = (m_i^k, c_i^k(LO), c_i^k(HI))$, where
• $m_i^k$ denotes the number of threads. We assume that threads of a segment, are independent from each other, can be executed in parallel, and have the same worst case execution time (WCET) estimates.

• $c_i^k(LO)$ is the thread WCET estimate that is used by the system designer (i.e., the WCET at the LO-criticality level).

• $c_i^k(HI)$ is the thread WCET estimate that is used by the certification authority (CA) (i.e., the WCET at the HI-criticality level).

Similar to previous research [18], we assume that

• $c_i^k(LO) \leq c_i^k(HI)$, i.e., the WCET estimate used by the system designer is never more pessimistic than the one used by the CA, and

• $c_i^k(LO) = c_i^k(HI)$ if $\chi_i = LO$ i.e., a LO-criticality job is aborted if any thread comprising the job executes for more than its LO-criticality WCET 1.

Upon release, a parallel job begins execution, where each thread of the job needs to execute for some amount of time $\gamma$. However, the value of $\gamma$ is unknown beforehand, but only becomes revealed by actually executing the thread until it signals that it has completed execution. If a thread of segment $J_i^k$ signals completion without exceeding $c_i^k(LO)$ units of execution, we say that it has exhibited LO-criticality behavior; if it signals completion after executing for more than $c_i^k(LO)$ but no more than $c_i^k(HI)$ units of execution, we say that it has exhibited HI-criticality behavior. If it does not signal completion upon having executed for $c_i^k(HI)$ units, we say that its behavior is erroneous. A parallel job has exhibited LO-criticality behavior if all threads comprising the job has exhibited LO-criticality behavior. A parallel job has exhibited HI-criticality behavior if any thread comprising the job has exhibited HI-criticality behavior and no thread of the job has exhibited erroneous behavior. A parallel job has exhibited erroneous behavior if any thread comprising the job has exhibited erroneous behavior.

Next, we define the minimum job length of LO-criticality and HI-criticality (i.e., the LO-criticality and HI-criticality job WCET on infinite number of processors)

\[
P_i(LO) = \sum_{k=1}^{s_i} m_i^k c_i^k(LO) \quad (1)
\]

\[
P_i(HI) = \sum_{k=1}^{s_i} m_i^k c_i^k(HI) \quad (2)
\]

We let

\[
W_i(LO) = \sum_{k=1}^{s_i} m_i^k c_i^k(LO) \quad (3)
\]

\[
W_i(HI) = \sum_{k=1}^{s_i} m_i^k c_i^k(HI) \quad (4)
\]

1As previous research [18], we assume that the run-time system provides support for ensuring that a thread does not execute for more than a specified amount.

respectively be the total LO-criticality and HI-criticality WCET on a single processor, also called the LO-criticality and HI-criticality work of the job.

IV. PROBLEM FORMULATION

In this paper, we investigate the multiprocessor mixed-criticality scheduling of parallel jobs. More precisely, we consider scheduling a job set $\tau$, described by the model in Section III, on a multiprocessor platform of $m$ identical processors.

Scheduling of MC parallel jobs. We define an algorithm for scheduling MC parallel job set $\tau$ to be correct if it is able to schedule $\tau$ such that

• If all parallel jobs exhibit LO-criticality behavior, then all of them receive enough execution between their release time and deadline to be able to signal completion; and

• If any parallel job exhibits HI-criticality behavior, then all HI-criticality jobs receive enough execution between their release time and deadline to be able to signal completion.

As explained in [18], if any job exhibits HI-criticality behavior, we do not require any LO-criticality jobs (including those that may have arrived before this happened) to complete by their deadlines. This is an implication of the requirements of certification: informally speaking, the system designer fully expects that all jobs will exhibit LO-criticality behavior, and hence is only concerned that they behave as desired under these circumstances. The CA, on the other hand, allows for the possibility that some parallel jobs may exhibit HI-criticality behavior, and requires that all HI-criticality jobs nevertheless meet their deadlines.

This paper focuses on a partitioned approach to schedule MC parallel jobs on multiprocessors. In such an approach, we will first decompose each MC parallel job into a set of MC sequential jobs by converting each thread of the parallel job into its own sequential job and assigning appropriate release times and deadlines to these jobs. Then, we will develop a partitioning strategy to allocate the decomposed MC sequential jobs to the $m$ processors. At last, an existing method will be used to schedule MC sequential jobs on each processor.

In developing such a partitioned approach, one of the biggest challenges is the design of the decomposition algorithm. The design objective is to decompose MC parallel jobs into MC sequential jobs that are either schedulable on $m$ processors whenever possible or requiring the least number of processors to be schedulable. The decomposition must be carried out in such a way that achieves this goal.

Since job decomposition and partitioning must be designed according to the job scheduling algorithm eventually used to schedule jobs on each processor, next section briefly introduces the OCBP (a uniprocessor MC) scheduling algorithm and a load-based schedulability test for OCBP. Although the OCBP algorithm is used to illustrate our approach, the ideas of job decomposition and partitioning can be generalized and similar algorithms can be developed for other uniprocessor MC scheduling algorithms as well.
V. THE OCBP SCHEDULING ALGORITHM

In [10], Baruah et al. developed a priority-based algorithm called OCBP (Own Criticality-Based Priorities) for uniprocessor mixed-criticality scheduling. The high-level description of the OCBP algorithm is as follows. Given a set $I$ of mixed-criticality sequential jobs, the algorithm determines off-line (i.e., prior to run-time) a total priority ordering of the jobs such that scheduling the jobs according to this priority ordering guarantees a correct schedule. Here, scheduling according to a priority ordering means that at each moment in time the highest-priority available job is executed.

In real-time scheduling, there is a common and well-known characterization metric called load, i.e., the maximal ratio of the processing demand and the processing capacity. Li et al. [43] defined the load metrics for mixed-criticality scenarios between the processing demand and the processing capacity. Li and Baruah [43] defined the load metrics for mixed-criticality systems and applied these metrics for the OCBP algorithm. The loads a processor can experience in LO-criticality and HI-criticality scenarios are determined as follows

$$\ell_{LO}(I) = \max_{0 \leq t_1 < t_2} \left( \frac{\sum_{j_i \in I} j_i \cdot d_i \cdot c_i(LO)}{t_2 - t_1} \right)$$

$$\ell_{HI}(I) = \max_{0 \leq t_1 < t_2} \left( \frac{\sum_{j_i \in I} j_i \cdot d_i \cdot c_i(HI)}{t_2 - t_1} \right)$$

Informally, $\ell_{LO}(I)$ is the largest load that the system designer expects to handle during run-time, while $\ell_{HI}(I)$ is the largest load that the CA expects to certify. Baruah et al. [10], [43] proved that a MC sequential job set $I$ is schedulable by the OCBP algorithm if it satisfies the following conditions

$$\ell_{LO}(I) \leq \frac{\sqrt{5} - 1}{2} \text{ and } \ell_{HI}(I) \leq \frac{\sqrt{5} - 1}{2}$$

VI. ALGORITHMS

This section presents three algorithms: a baseline EqualSlack job decomposition algorithm, our MinLoad job decomposition algorithm, and a job partitioning strategy. Combining a job decomposition (EqualSlack or MinLoad), the job partitioning, and the OCBP scheduling together gives us an (EqualSlack-Based or MinLoad-Based) algorithm for partitioned multiprocessor scheduling of mixed-criticality parallel jobs.

In a decomposition, each thread of a MC parallel job is converted to a MC sequential job, which is assigned new release time and deadline such that the precedence relation of the parallel job is maintained. Since we are decomposing synchronous jobs, threads of a common segment should be assigned a common release time and a common deadline, which are also called the release time and deadline of a segment. To maintain the precedence relation of job $J_i$, we must satisfy the following constraint for its adjacent segments $J_i^k$ and $J_i^{k+1}$: the release time of segment $J_i^{k+1}$ should equal the deadline of segment $J_i^k$.

A. EqualSlack Job Decomposition

We now present a baseline EqualSlack job decomposition algorithm to separate MC parallel jobs into MC sequential jobs. Given a job $J_i$, several steps are followed to determine its segment $J_i^k$’s release time $A_i^k$ and deadline $D_i^k$.

First, we calculate job $J_i$’s slack, which is defined as the difference between its deadline and its earliest finish time in HI-criticality scenario, i.e., $L_i = D_i - (A_i + \hat{p}(HI))$ (see Equation (2) for $\hat{p}(HI)$’s definition).

Second, when decomposing a job, EqualSlack algorithm distributes the slack evenly to the job’s segments. Since a MC parallel job has $s_i$ number of segments, each MC sequential job decomposed from $J_i$ has a slack of $\frac{L_i}{s_i}$. The release time $A_i^k$ and deadline $D_i^k$ of each segment are calculated accordingly.

$$A_i^k = \begin{cases} A_i & \text{if } k = 1 \\ D_i^{k-1} & \text{if } 1 < k \leq s_i \end{cases}$$

$$D_i^k = \begin{cases} A_i^k + \frac{D_i}{s_i} & \text{if } k = s_i \\ A_i^k + \hat{p}_i^k(HI) + \frac{L_i}{s_i} & \text{if } 1 \leq k < s_i \end{cases}$$

Once we have derived appropriate release time $A_i^k$ and deadline $D_i^k$, each segment $J_i^k$ is decomposed into $m_i^k$ number of identical MC sequential jobs: $(A_i^k, D_i^k, c_i^k(LO), c_i^k(HI), \chi_i)$. Totally, $\sum_{i=1}^{n} s_i \sum_{k=1}^{m_i^k}$ number of MC sequential jobs are generated from decomposing job set $\tau$.

B. MinLoad Job Decomposition

We develop a new MinLoad algorithm, which decomposes MC parallel jobs in such a way as to make the resultant MC sequential jobs easier to schedule, i.e., requiring less number of processors to be schedulable by the partitioning and OCBP algorithms. In Section V, we presented a sufficient condition: Equation (7), for a job set to be schedulable by the OCBP algorithm. By analyzing the condition, we think if we control the values of $\ell_{LO}(I)$ and $\ell_{HI}(I)$ of the resultant MC sequential job set $I$, i.e., using a job decomposition that minimizes $MaxLoad(I) = \max(\ell_{LO}(I), \ell_{HI}(I))$, we can make $I$ easier to schedule. Thus, we develop a heuristic algorithm, called MinLoad, to minimize the value of $MaxLoad(I)$ for the decomposed sequential job set $I$.

Algorithm Overview. Here, we provide a high-level overview of the MinLoad job decomposition algorithm. MinLoad algorithm first invokes EqualSlack algorithm, presented in Section VI-A, to get an initial decomposition $I$ of the parallel job set $\tau$. Then, MinLoad algorithm follows a systematic way to repetitively change parameters $A_i$ and $D_i$ of some segment $S_i$’s threads to reduce the value of $MaxLoad(I)$. This process stops when the parameters of jobs contributing to $MaxLoad(I)$ can no longer be modified to make $MaxLoad(I)$ smaller.

Detailed Description. We now provide a detailed description of our MinLoad algorithm, whose pseudo code is presented in Algorithm 1.

At the beginning of MinLoad algorithm, EqualSlack algorithm is invoked to generate the initial job decomposition $I$ (line 2 of Algorithm 1). Then, the current value of
MaxLoad(I) = max(\(\ell_{LO}(I), \ell_{HI}(I)\)) is calculated and the corresponding interval \([t_1, t_2]\) that has this maximum load is identified (line 3). According to Equations (5) and (6), we know \(t_1\) must be a job’s release time and \(t_2\) must be a job’s deadline. Since \(I\) is decomposed from synchronous job set \(\tau\), threads of a common parallel job segment are assigned a common release time and a common deadline. Since there are \(N = \sum_{i=1}^{n} s_i\) number of segments in parallel job set \(\tau\), we have at most \(N\) unique release time points and \(N\) unique deadlines in the resultant sequential job set \(I\). Thus, there are at most \(N^2\) number of different intervals for calculating MaxLoad(I). To facilitate the decomposition change, we add some data structures to \(I\) to record the structure of the original parallel jobs in \(\tau\), i.e., sequential jobs are organized in segment groups and jobs generated from a common segment must be changed together to keep their parameters always the same. These data structures are, however, only used by the MinLoad algorithm and are not passed to the partitioning and OCBP algorithms.

After identifying the interval \([t_1, t_2]\), the algorithm analyzes the segment of jobs that have contributed to the maximum load in \([t_1, t_2]\) and changes their parameters to make MaxLoad(I) smaller (lines 4-25). More specifically, when MaxLoad(I) = \(\ell_{LO}(I)\), if a segment \(S_i\)’s release time and deadline satisfy condition: \(t_1 \leq A_i \land D_i \leq t_2\), jobs generated from segment \(S_i\) have contributed \(\frac{m \times C_i(LO)}{t_2-t_1}\) amount of load to MaxLoad(I), where \(m\) is the number of threads in \(S_i\) and \(C_i(LO)\) denotes each thread’s LO-criticality WCET; when MaxLoad(I) = \(\ell_{HI}(I)\), if segment \(S_i\) belongs to a HI-criticality parallel job and \(S_i\)’s release time and deadline satisfy condition: \(t_1 \leq A_i \land D_i \leq t_2\), jobs generated from segment \(S_i\) have contributed \(\frac{m \times C_i(HI)}{t_2-t_1}\) amount of load to MaxLoad(I), where \(C_i(HI)\) denotes the HI-criticality WCET of \(S_i\)’s threads. The MinLoad algorithm picks such a segment and first tries to reduce the release time \(A_i\) of the segment’s jobs (lines 5-13). Since the goal is to reduce MaxLoad(I), we would like to decrease \(A_i\) such that \(A_i\) becomes less than \(t_1\). If the release time change fails, the MinLoad algorithm tries to increase the deadline \(D_i\) of the segment’s jobs (lines 14-21). Since the goal is to reduce MaxLoad(I), we would like to increase \(D_i\) such that \(D_i\) becomes larger than \(t_2\). There are, however, other constraints and effects that must be analyzed to ensure that the change indeed makes MaxLoad(I) smaller.

Segment Precedence Constraint. As mentioned, a job decomposition divides MC parallel jobs into a set of MC sequential jobs. In particular, each thread of a parallel job is converted to a sequential job, which is assigned new release time and deadline such that the precedence relation of the parallel job is still maintained. The following constraints for any adjacent segments of a job, say \(S_{i-1}, S_i,\) and \(S_{i+1}\) must be satisfied.

- The release time \(A_i\) of segment \(S_i\) must be equal to the deadline \(D_{i-1}\) of segment \(S_{i-1}\). Thus, to have a feasible job set \(I\), \(A_i \geq A_{i-1} + C_{i-1}(HI)\) must hold, where \(A_{i-1}\) is the release time of segment \(S_{i-1}\)’s jobs and \(C_{i-1}(HI)\) is the HI-criticality WCET of \(S_{i-1}\)’s jobs.
- The release time \(A_i\) of segment \(S_i\) must be less or equal to the deadline of segment \(S_i\) minus its HI-criticality WCET \(C_i(HI)\), i.e., \(A_i \leq D_i - C_i(HI)\).
- The release time \(A_{i+1}\) of segment \(S_{i+1}\) must be equal to the deadline \(D_i\) of segment \(S_i\). Thus, to have a feasible job set \(I\), \(D_i\) must be less or equal to the deadline \(D_{i+1}\) of \(S_{i+1}\) minus \(S_{i+1}\)’s HI-criticality WCET \(C_{i+1}(HI)\), i.e. \(D_i \leq D_{i+1} - C_{i+1}(HI)\).

Thus, to reduce \(A_i\), the new value must fall in the range \([A_{i-1} + C_{i-1}(HI), t_1]\), and to increase \(D_i\), the new value must fall in the range \((t_2, D_{i+1} - C_{i+1}(HI))\). When possible, the MinLoad algorithm uses a binary search method to pick a new value in these ranges to make MaxLoad(I) smaller.

Before we analyze the effects of a parameter change on the load of an interval, let us give some new definitions. Two load metrics corresponding to interval \([t_b, t_e]\) are given as follows.

\[
\ell_{LO}(t_b, t_e, I) = \sum_{J_i:t_b \leq A_i \land D_i \leq t_e} C_i(LO)
\]

\[
\ell_{HI}(t_b, t_e, I) = \sum_{J_i:t_b \leq A_i \land D_i \leq t_e} C_i(HI)
\]

The Effect of \(A_i\)’s Decrease on Interval \([t_1, t_2]\)’s Loads. Before the change, segment \(S_i\) contributes \(\frac{m \times C_i(HI)}{t_2-t_1}\) amount of load to \(\ell_{HI}(t_1, t_2, I)\) and \(\ell_{LO}(t_1, t_2, I)\) respectively. There are two cases that need to be analyzed separately.

- \(A_i \neq t_1\): After making \(A_i\) smaller than \(t_1\), \(\ell_{HI}(t_1, t_2, I)\) and \(\ell_{LO}(t_1, t_2, I)\) are reduced by \(\frac{m \times C_i(HI)}{t_2-t_1}\) and \(\frac{m \times C_i(LO)}{t_2-t_1}\) amount respectively.
- \(t_1\) is \(A_i\): In this case, to reduce \(A_i\) means to decrease \(t_1\). Assuming \(A_i\) is decreased to \(\hat{t}_1\), \(\hat{t}_1\) is also decreased to \(\hat{t}_1 = A_i\). The new LO load \(\ell_{LO}(\hat{t}_1, t_2, I)\) becomes

\[
\ell_{LO}(\hat{t}_1, t_2, I) = \sum_{J_i: \hat{t}_1 \leq A_i \land D_i \leq t_e} C_i(LO)
\]

Since \(\hat{t}_1 < t_1\), more jobs may be included when calculating the new LO load \(\ell_{LO}(\hat{t}_1, t_2, I)\). If a job \(J_k\)’s parameters satisfy the following condition: \(\hat{t}_1 \leq A_k < t_1 \land D_k \leq t_2\), its load is added to \(\ell_{LO}(\hat{t}_1, t_2, I)\). Let us denote

\[
C = \sum_{J_k: \hat{t}_1 \leq A_k \land D_k \leq t_2} C_k(LO)
\]
Combining all these analyses together, MinLoad algorithm determines whether or not reducing \(A_i\) to the new value is able to make the new \(MaxLoad(I)\) smaller than the old \(MaxLoad(I)\) (Note, the new and old \(MaxLoad(I)\) may correspond to different intervals.). As mentioned, if reducing the release time \(A_i\) of a relevant segment does not make \(MaxLoad(I)\) smaller, the MinLoad algorithm tries to increase the deadline \(D_i\) of the segment’s jobs. Since the goal is to make \(MaxLoad(I)\) smaller, we would like to increase \(D_i\) such that \(D_i\) becomes larger than \(t_2\).

The Effect of \(D_i\)’s Increase on \([t_1, t_2)\)’s Loads. Before the change, segment \(S_i\) contributes \(\frac{m_i \times C_i(H)}{t_2-t_1}\) amount of load to \(\ell_{HI}(t_1, t_2)\) and \(\frac{m_i \times C_i(LO)}{t_2-t_1}\) amount of load to \(\ell_{LO}(t_1, t_2, I)\) respectively. There are two cases that need to be analyzed separately

- \(D_i \neq t_2\): After making \(D_i\) larger than \(t_2\), \(\ell_{HI}(t_1, t_2, I)\) and \(\ell_{LO}(t_1, t_2, I)\) are reduced by \(\frac{m_i \times C_i(LO)}{t_2-t_1}\) and \(\frac{m_i \times C_i(H)}{t_2-t_1}\) respectively.
- \(t_2 \leq D_i\): In this case, to increase \(D_i\) means to increase \(t_2\). Assuming \(D_i\) is increased to \(\hat{D}_i\), the new \(LO\) load \(\ell_{LO}(\hat{t}_1, \hat{t}_2, I)\) becomes

\[
\ell_{LO}(\hat{t}_1, \hat{t}_2, I) = \sum_{J_i: \hat{t}_1 \leq A_i \land D_i \leq \hat{t}_2} C_i(LO) / (\hat{t}_2 - \hat{t}_1)
\]

Since our goal is to reduce \(MaxLoad(I)\), when the current maximum load \(MaxLoad(I) = \ell_{LO}(t_1, t_2, I)\) and \(\frac{\Delta C}{\Delta t} \geq \frac{C}{t}\), the algorithm will not change \(A_i\) to \(\hat{A}_i\). Only if the change reduces the load, i.e., when \(\frac{\Delta C}{\Delta t} \leq \frac{C}{t}\) and thus \(\ell_{LO}(\hat{t}_1, \hat{t}_2, I) \leq \ell_{LO}(t_1, t_2, I)\), will the change be made.

Similar analysis is made to evaluate the effect of \(A_i\)’s change on the \(HI\) load \(\ell_{HI}(t_1, t_2, I)\).

The Effect of \(A_i\)’s Decrease on Other Intervals’ Loads. Now, we analyze the effect of \(A_i\)’s decrease on the load of an arbitrary interval \([t_b, t_c]\). There are several cases

- If \(A_i \geq t_b\) and the corresponding deadline \(D_i \leq t_c\), then the effect on the loads in \([t_b, t_c]\) follows the same analysis as that in interval \([t_1, t_2]\).
- If \(A_i < t_b\) or the corresponding deadline \(D_i > t_c\), then reducing the value of \(A_i\) does not affect the loads in \([t_b, t_c]\).
- Assume \(S_i \) and \(S_i \) are segments of a parallel job and \(S_i \) is the segment preceding \(S_i\). Since \(A_i = D_{i-1}\), where \(D_{i-1}\) denotes the deadline of \(S_i\)’s jobs, to reduce \(A_i\) also decreases \(D_{i-1}\)’s value. The effect of the deadline reduction on loads is analyzed below.

The Effect of \(D_i\)’s Decrease on \([t_b, t_c]\)’s Loads. Now, we analyze the effect of \(D_i\)’s decrease on the load of an interval \([t_b, t_c]\). There are several cases

- If the corresponding release time \(A_i \geq t_b\) and \(D_i \leq t_c\), then reducing \(D_i\) within its constrained range does not change the loads in \([t_b, t_c]\).
- If the corresponding release time \(A_i < t_b\), then reducing \(D_i\) does not affect the loads in \([t_b, t_c]\).
- If the corresponding release time \(A_i \geq t_b\) and \(D_i > t_c\), then reducing \(D_i\) may increase \(\ell_{LO}(t_b, t_c, I)\) by \(\frac{m_i \times C_i(LO)}{t_c-t_b}\) amount if \(D_i\) becomes less or equal to \(t_c\). Similar effect holds for \(\ell_{HI}(t_b, t_c, I)\) if \(S_i\) is a HI-criticality job’s segment.
**Algorithm 1: MinLoad Job Decomposition**

**Input:** A MC parallel job set \( \tau \)

**Output:** A MC sequential job set \( I \)

//** Invoke the EqualSlack Algorithm to generate the initial sequential job set \( I \). To facilitate the decomposition change, we add some data structures to \( I \) to record the structure of the original parallel jobs in \( \tau \), i.e., sequential jobs are organized in segment groups. These data structures are, however, only used by the MinLoad algorithm and are not passed to the partitioning and OCBP algorithms. */

\( I = \text{EqualSlackAlg}(\tau) \)

while Find an interval \([t_1, t_2]\) with the maximal load \( \text{MaxLoad}(I) \) do

- FlagChange=\text{False}

  foreach Segment \( S_i \) of jobs in interval \([t_1, t_2]\) /* When \( \text{MaxLoad}(I) = \ell_{LO}(I) \), \( S_i \) is in \([t_1, t_2]\) if \( t_1 \leq A_i \wedge D_i \leq t_2 \). When \( \text{MaxLoad}(I) = \ell_{HI}(I) \), \( S_i \) is in \([t_1, t_2]\) if \( S_i \)'s jobs are of HI-criticality and \( t_1 \leq A_i \wedge D_i \leq t_2 \) */ do

    while The release time of \( S_i \)'s jobs can be decreased, i.e., using a binary search method to find a new value for \( A_i \), so that \( A_i \) is still in its constrained range but becomes less than \( t_1 \) do

      if the new \( \text{MaxLoad}(I) \) becomes smaller than the old \( \text{MaxLoad}(I) \) as a result of the change then

        Set the release time of \( S_i \)'s jobs to the new value and update the data structures
        FlagChange=\text{True}
        Break the For-Loop

    end

end

while The deadline of \( S_i \)'s jobs can be increased, i.e., using a binary search method to find a new value for \( D_i \), so that \( D_i \) is still in its constrained range but becomes greater than \( t_2 \) do

- if the new \( \text{MaxLoad}(I) \) becomes smaller than the old \( \text{MaxLoad}(I) \) as a result of the change then

      Set the deadline of \( S_i \)'s jobs to the new value and update the data structures
      FlagChange=\text{True}
      Break the For-Loop

end

if FlagChange=\text{False} /* no single-parameter change is found to make \( \text{MaxLoad}(I) \) smaller */ then

  Break the While-Loop

return \( I \)

---

- If the corresponding release time \( A_i < t_b \) or \( D_i > t_e \), then increasing the value of \( D_i \) does not affect the loads in \([t_b, t_e]\).
- Assume \( S_i \) and \( S_{i+1} \) are segments of a parallel job and \( S_{i+1} \) is the segment succeeding \( S_i \). Since \( D_i = A_{i+1} \), where \( A_{i+1} \) denotes the release time of \( S_{i+1} \)'s jobs, to increase \( D_i \) also increases \( A_{i+1} \)'s value. The effect of the release time increase on loads is analyzed below.

### The Effect of \( A_i \)'s Increase on \([t_b, t_e] \)’s Loads.

Now, we analyze the effect of \( A_i \)'s increase on the load of an interval \([t_b, t_e] \). There are several cases

- If \( A_i \geq t_b \) and the corresponding deadline \( D_i \leq t_e \), then increasing \( A_i \) within its constrained range does not change the loads in \([t_b, t_e]\).
- If \( A_i < t_b \) and the corresponding deadline \( D_i \leq t_e \), then increasing \( A_i \) may increase \( \ell_{LO}(t_b, t_e, I) \) by \( \frac{(t_e-t_i)}{t_e-t_b} \) amount if \( A_i \) becomes larger or equal to \( t_b \). Similar effect holds for \( \ell_{HI}(t_b, t_e, I) \) if \( S_i \) is a HI-criticality job’s segment.
- If the corresponding deadline \( D_i > t_e \), then increasing \( A_i \) does not affect the loads in \([t_b, t_e]\).

Combining all these analyses together, the MinLoad algorithm determines whether or not increasing \( D_i \) to its new value is able to make the new \( \text{MaxLoad}(I) \) smaller than the old \( \text{MaxLoad}(I) \) (Note, the new and old \( \text{MaxLoad}(I) \) may correspond to different intervals.). The algorithm stops when the parameters of jobs contributing to \( \text{MaxLoad}(I) \) can no longer be modified to make \( \text{MaxLoad}(I) \) smaller (lines 22-25).

### C. Partitioning Algorithm

After decomposing MC parallel jobs \( \tau \) into MC sequential jobs \( I \), a two-phase partitioning algorithm is developed to schedule \( I \) on the multiprocessor platform of \( m \) processors. Algorithm 2 gives the pseudo code of the partitioning algorithm.

The partitioning algorithm proceeds in two phases

1) During the first phase (Lines 2 to 13), only HI-criticality jobs are considered to be allocated to the multiprocessor platform. For a HI-criticality job, the partitioning algorithm considers an available processor only when the HI load \( \ell_{HI}(I[k]) \) of the processor does not exceed \( \sqrt{\frac{n-1}{2}} \) (Line 6). According to Equation (7), as long as each processor’s HI load does not exceed \( \frac{\sqrt{n-1}}{2} \), all HI-criticality jobs that have been allocated to the system are schedulable by the OCBP algorithm even in HI-criticality scenario.

2) During the second phase (Lines 14 to Line 25), only LO-criticality jobs are considered. For a LO-criticality job, the partitioning algorithm considers an available processor only when the LO load \( \ell_{LO}(I[k]) \) of the processor does not exceed \( \frac{\sqrt{n-1}}{2} \) (Line 18). According to Equation (7), as long as each processor’s LO load does not exceed \( \frac{\sqrt{n-1}}{2} \), all jobs that have been allocated
to the system are schedulable by the OCBP algorithm when all jobs exhibit LO-criticality behaviors.

Upon the completion of the algorithm, any unassigned jobs are considered failed.

Algorithm 2: Multiprocessor Job Partitioning

Input: A MC sequential job set I, Number of Processors m
Output: Job allocation array I[1 · · · m]

Initialize(I[1 · · · m])

foreach HI-criticality job Ji ∈ I do
  for k = 1; k ≤ m; k + + do
    Add Ji to I[k]
    Calculate ℓH1(I[k])
    if ℓH1(I[k]) > √7−1 2 then
      Remove Ji from I[k]
    else
      Remove Ji from I
      break
  end
end

foreach LO-criticality job Ji ∈ I do
  for k = 1; k ≤ m; k + + do
    Add Ji to I[k]
    Calculate ℓLO(I[k])
    if ℓLO(I[k]) > √7−1 2 then
      Remove Ji from I[k]
    else
      Remove Ji from I
      break
  end
end

return I[1 · · · m]

VII. EVALUATION

We have carried out simulations on randomly-generated mixed-criticality parallel jobs, where we apply EqualSlack-Based and MinLoad-Based partitioned algorithms to schedule the parallel job sets on multiprocessor platforms. A series of randomly generated job sets of different sizes are used. More precisely, the size of the parallel job set varies from 10 to 30. The release time and deadline of the parallel jobs are also randomly generated, in the range [10, 100) and [200, 1000] respectively. It is assumed that there are more LO-criticality jobs than HI-criticality jobs. Specifically, the number of LO-criticality jobs is twice of the number of HI-criticality jobs. The number of segments of each MC parallel job is randomly generated from 3 to 6. The number of threads for each segment is randomly generated from 2 to 6. We make the sum of the HI-criticality WCET of all segments of job Ji fall between 0.30 × |Di − Ai| and 0.20 × |Di − Ai|, while the distribution of the LO-criticality WCET sum to the segments is random. In other words, the following conditions must be satisfied when randomly generating the WCETs: ĉ1(HI) and ĉ1(LO) for segment Ji of job Ji.

0.3 × |Di − Ai| ≤ ∑ j=1 cs j (HI) ≤ 0.4 × |Di − Ai| (25)
0.1 × |Di − Ai| ≤ ∑ j=1 cs j (LO) ≤ 0.2 × |Di − Ai| (26)

After randomly generating a MC parallel job set τ, we apply either EqualSlack or MinLoad algorithm to convert it to a set of MC sequential jobs I. Then, the set of MC sequential jobs I are scheduled according to the partitioning algorithm (i.e., Algorithm 2). To compare the two algorithms, we use the number of processors required to make τ schedulable as the metric. Given a job set τ, a binary search approach is adopted to find these numbers for EqualSlack-Based and MinLoad-Based partitioned algorithms.

The simulation results are presented in Figure 1. The curves show the number of processors required by the two algorithms to make MC parallel job sets of different sizes, ranging from 10 to 30, schedulable. From these curves, we can see that our MinLoad-Based partitioned algorithm always requires less number of processors. In comparison to EqualSlack-Based algorithm, MinLoad-Based algorithm reduces the number of required processors by 12% to 32%. MinLoad algorithm achieves its design goal: it indeed decomposes MC parallel jobs in such a way that makes the resultant MC sequential jobs easier to schedule, i.e., requiring less number of processors to be schedulable by the partitioning and OCBP algorithms. These results have also proved our hypothesis: if we control the values of ℓLO(I) and ℓHI(I) of the resultant MC sequential job set I, i.e., by reducing MaxLoad(I) = max(ℓLO(I), ℓHI(I)), we can make I easier to schedule.

VIII. CONCLUSION

There has been an increasing research interest in scheduling mixed-criticality tasks in multiprocessor systems as multiprocessor technology becomes mainstream in processor design [18], [46]. However, most existing work on scheduling mixed-criticality systems are limited to sequential programming models and they are ineffective in exploiting the processing power of multiprocessor systems. In this paper, we have proposed a mixed-criticality parallel job model targeting at fully harnessing the power of multiprocessor systems. We have developed a novel job decomposition algorithm, called MinLoad, based on which a new partitioned algorithm is created to schedule mixed-criticality parallel jobs on multiprocessors. Comparing to a baseline EqualSlack job decomposition, our MinLoad method requires smaller-sized multiprocessor platforms for the mixed-criticality systems.
IX. ACKNOWLEDGEMENTS

The authors acknowledge support from General Motors Global Research & Development and NSFC award 61272127.

REFERENCES


