2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing

Mixed-criticality scheduling upon varying-speed multiprocessors

Zhishan Guo Sanjoy Baruah
The University of North Carolina at Chapel Hill

***Abstract*—An increasing trend in embedded computing is the moving towards mixed-criticality (MC) systems, in which functionalities of different importance degrees (criticalities) are implemented upon a common platform. Most previous work on MC scheduling focuses on the aspect that different timing analysis tools may result in multiple WCET estimations for each "job" (piece of code). Recently, a different MC model has been proposed, targeting systems with varying execution speeds. It is assumed that the precise speed of the processor upon which the system is implemented varies in an *a priori* unknown manner during runtime, and estimates must be made as to how low the actual speed may fall. Prior work has dealt with uniprocessor platforms of this kind; the research reported in this paper seeks to generalize this prior work to be applicable to multicore platforms. In our method, a linear program (LP) is constructed based on necessary and sufficient scheduling conditions; and according to its solution, jobs are executed in a processor-sharing based method. Optimality of the algorithm is proved, and an example is constructed to show the necessity of processor sharing.**

I. INTRODUCTION

There is an increasing trend in embedded computing on moving towards *mixed-criticality* systems, where functionalities of different degrees of importance are implemented upon a common platform. This is evidenced by industry-wide initiatives such as IMA (Integrated Modular Avionics) for aerospace, and AUTOSAR (AUTomotive Open System ARchitecture) for the automotive industry.

Mixed-Criticality. Most prior work on mixed-criticality (MC) scheduling has focused on the model in which multiple worst case execution time (WCET) parameters are specified for each job. Under such a model, the larger value represents a "safer" estimate of the job's true WCET, while the smaller WCET is a less conservative one. Some of the jobs are designated as being safety-critical, and are assigned HI-criticality; the remaining ones are called LO-criticality jobs. In many cases, it may not be necessary to use the most conservative tool for validating the correctness of the system – less conservative tools should be sufficient, especially for less critical functionalities.

Since Vestal's pioneering work [13], there has been a large amount of research in mixed-criticality scheduling (see [7] for a review), where in the model each job is characterized by multiple WCETs. The general goal is to validate the correctness of highly critical functionalities under more pessimistic assumptions, while guaranteeing the correctness of less critical functionalities under less pessimistic ones.

Varying-Speed Processor. WCET tools make certain assumptions about the run-time behavior of the processor upon which the code is to execute; for example, the clock speed of the processor during run-time must be known to determine the rate at which instructions will execute. However, conditions during run-time, such as changes to the ambient temperature, the supply voltage, etc., may result in variations of the clock speed. In addition, processor speed could change during run-time in embedded devices that use dynamic frequency scaling in order to reduce energy consumption. Moreover, at the hardware level, innovations in computer architecture for increasing clock frequency can lead to varying speed clocks during run-time[1].

More recently, regarding this varying-speed property, a different MC model has been proposed [8] [5] in which it is assumed that the precise speed of the processor upon which the system is implemented varies in an *a priori* unknown manner during runtime. These studies suggest that when estimates are avaliable on how low the actual speed may drop to, there exists polynomial time *optimal* scheduling strategy for uniprocessor case.

Multiprocessor System. Embedded systems, especially safety-critical ones are increasingly implemented on multicore platforms. Furthermore, as these multicore platforms become more complex and sophisticated, their behaviors become less predictable. Larger variations will cause an increase of the pessimism to any conservative WCET-analysis tools.

Generally speaking, uniprocessor MC scheduling algorithms perform poorly on multicore platforms [4]. Regarding this issue, some recent studies have focused on mixed-criticality scheduling of multiprocessor systems; see, e.g., [1] [3] [2] [12] [14]. However, all of them address the multi-WCET model, which is generally NP hard even in uniprocessor case, and thus only provides non-optimal (approximated) solutions. Since it has been shown (in the uniprocessor case) that such NP-hardness no longer exists for varying-speed MC scheduling [5], in this paper we seek to propose optimal MC scheduling strategies specifically targeting varying-speed multiprocessors. To the best of our knowledge, there is no prior

Work supported by NSF grants CNS 1016954, CNS 1115284, and CNS 1218693; and ARO grant W911NF-09-1-0535.

[1]For example, ARM recently introduced a technique [6] for detecting whether signals are late at the circuit level within a CPU micro-architecture. Logical faults are prevented or recovered by delaying the next clock tick. This certainly introduces varying-speed property to higher (i.e., the software) levels.

978-1-4799-5079-9/14 $31.00 © 2014 IEEE
DOI 10.1109/DASC.2014.50

IEEE computer society

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on April 28,2023 at 17:42:16 UTC from IEEE Xplore. Restrictions apply.

work that targets mixed-criticality scheduling on varying-speed *multi*processors.

Paper Organization. In this paper, we target the scheduling problem on multiprocessor platform, where mixed-criticality arises from the varying-speed property of the system. The remainder of this paper is organized as follows. In Section II we formally describe the platform and workload models assumed in this work. In Section III we give a detailed description of our algorithm. Further discussions about both the problem and the proposed algorithm are provided in Section IV. Finally, Section V concludes the work and points out some further research directions.

II. MODEL

In this paper, we will be studying the scheduling of real-time *jobs* on multiprocessor platforms, under the following assumptions:

- Job preemption is permitted, with zero cost.
- Job migration is permitted, also with no penalty associated.
- Job parallelism is forbidden; i.e., each job may execute on at most one processor at any given instant.

We consider a workload model consisting of *independent jobs*[2]. In our model, a mixed-criticality real-time workload consists of basic units of work known as mixed-criticality jobs. Each *MC job* J_i is characterized by a 4-tuple of parameters: a release date a_i, a WCET c_i, a deadline d_i, and a criticality level $\chi_i \in \{\text{LO}, \text{HI}\}$.

A mixed-criticality multiprocessor *instance* $\mathcal{I}$ is specified by

- a finite collection of MC jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$, and
- m identical varying-speed processors that are characterized by both a normal speed (without loss of generality, assumed to be 1) and a specified *degraded processor speed threshold* $s < 1$.

Let $s_i(t)$ denote the processing speed of processor i at time t, $i = 1, ..., m$. The interpretation is that the jobs in $\mathcal{J}$ are to execute on a multiprocessor system that has two modes: a *normal* mode and a *degraded* mode.

Definition 1 (degraded mode): A system with m processors is in *degraded mode* at a given instant t if there exists *at least one* processor executing at a speed less than one; i.e., $\exists i, s_i(t) < 1$; and moreover, all processors execute at a minimum speed of s; i.e., $\forall i, s_i(t) \geq s$.

In normal mode, m processors execute at unit-speed and hence each completes one unit of execution per unit time, whereas in degraded mode, according to the definition, each processor completes at least s units of execution per unit time. It is not *a priori* known when, if at all, any of the processors will degrade: this information only becomes revealed during run-time when some processors actually begin executing at a slower speed. Besides the given two modes, we call the system in *error* mode if some processor executes below s.

[2]Finite collections of independent jobs may arise in *frame-based* approaches to real-time scheduling: the frame represents a collection of periodic jobs sharing a common period (or the frame is extended to the least common multiple of different tasks' periods and every job of each task is represented individually within the frame) with the entire frame repeatedly executed.

We seek to determine a *correct scheduling strategy*:

Definition 2 (correct scheduling strategy): A scheduling strategy for MC instances is *correct* if it possesses the properties that upon scheduling any MC instance $\mathcal{I} = (\mathcal{J}, m, s)$,

- if the system remains in normal mode throughout the interval $[\min_i\{a_i\}, \max_i\{d_i\})$, then all jobs complete by their deadlines; **and**
- if the system *never* operates at error mode, then HI-criticality jobs (J_i with $\chi_i = \text{HI}$) complete by their deadlines.

That is, a correct scheduling strategy ensures that HI-criticality jobs execute correctly regardless of whether the system runs in normal or degraded mode; LO-criticality jobs are required to execute correctly only if all processors execute throughout in normal mode. (One may argue that this is a rather restrictive definition, since we do not allow the case that a few processors to being nonfunctional, even when others execute at full speed. In Section IV-B we will discuss the problem based on an alternative, less restrictive, definition to degraded mode, where as far as all processors altogether, as a system, execute at an *average* speed of s, correctness to HI-criticality jobs will be guaranteed.)

Based on the definition of *correctness*, we can now define *optimality* in the following way:

Definition 3 (optimal scheduling strategy): An optimal scheduling strategy for MC instances possesses the property that if it fails to maintain correctness for a given MC instance $\mathcal{I}$, then no non-clairvoyant algorithm can ensure correctness for the instance $\mathcal{I}$.

III. PREEMPTIVE SCHEDULING

In this section we present, and show the correctness of, a simple polynomial-time algorithm for scheduling preemptable mixed-criticality instances. Its optimality is proved as well.

We start out with a general overview of our strategy. Given an instance $\mathcal{I} = \{\mathcal{J}, m, s\}$, prior to run-time we will construct a linear program to determine the amount of execution to be completed for each job within each interval. Such assignment will possess the property that each job J_i receives c_i units of execution over its scheduling window $[a_i, d_i)$. Details on this linear program construction will be provided in Subsection A, with an illustrative example alongside. During run-time, we will mimic a *processor sharing* strategy, under which the time line is partitioned into quanta. The length of each quantum is assumed to be small enough so that each processor will run at a "predictable" (and thus known) speed within it. Scheduling decisions are made according to assignments derived from the previous step. Degradation may occur at the beginning of any quantum. To guarantee that HI-criticality jobs will meet their deadlines if the system degrades, each HI-criticality job needs to be executed at its designated fraction over any past

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on April 28,2023 at 17:42:16 UTC from IEEE Xplore. Restrictions apply.

period of the interval. A detailed description will be provided in Subsection B.

A. Linear Program for Table Construction

We now describe how to construct a *linear program* such that a feasible solution for this linear program can be further used to construct the schedule. Without loss of generality, assume that the HI-criticality jobs in I are indexed $1, 2, \ldots, n_h$ and the LO-criticality jobs are indexed $n_{h+1}, \ldots, n$. Let $t_1, t_2, \ldots, t_{k+1}$ denote the at most $2n$ distinct values for the release date and deadline parameters of the n jobs, in increasing order ($t_j < t_{j+1}$ for all j). These release dates and deadlines partition the time-interval $\big[\min_i\{a_i\}, \max_i\{d_i\}\big)$ into k intervals, which will be denoted as $I_1, I_2, \ldots, I_k$, with I_j denoting the interval $[t_j, t_{j+1})$.

To construct our linear program we define $n \cdot k$ variables $x_{i,j}$, $1 \le i \le n; 1 \le j \le k$. Variable $x_{i,j}$ denotes the amount of execution we will assign to job J_i in the interval I_j, in the scheduling table that we are seeking to build.

First of all, since no job can be executed on more than one processor in parallel, the following two sets of inequalities need to be introduced for ensuring no capacity constraint is violated:

$$0 \le x_{i,j} \le s(t_{j+1} - t_j), \forall (i,j), 1 \le i \le n_h, 1 \le j \le k; \quad (1)$$

$$0 \le x_{i,j} \le t_{j+1} - t_j, \forall (i,j), n_h < i \le n, 1 \le j \le k. \quad (2)$$

The following n constraints specify that each job receives adequate execution when system remains in normal mode:

$$\Big(\sum_{j | t_j \ge a_i \ \wedge \ d_i \ge t_{j+1}} x_{i,j} \Big) \ge c_i, \forall i, 1 \le i \le n. \quad (3)$$

The following k inequalities specify the capacity constraints of each interval:

$$\big(\sum_{i=1}^{n} x_{i,j} \big) \le m(t_{j+1} - t_j), \forall j, 1 \le j \le k. \quad (4)$$

It should be evident that any scheduling table generated in this manner from $x_{i,j}$ values satisfying the above constraints will execute all jobs to completion upon a normal-mode (non-degraded) system. It now remains to add constraints for specifying the requirements that the HI-criticality jobs complete execution even in the event of the system degrading into faulty mode. It is evident that we only need to specify constraints for the most pessimistic degradation case – full degradation, where all processors run at the threshold speed s.

Considering the case when *full* degradation occurs at the beginning of each interval, capacity constraints of each interval need to be specified for all HI-criticality amounts:

$$\big(\sum_{i=1}^{n_h} x_{i,j} \big) \le s \cdot m(t_{j+1} - t_j), \forall j, 1 \le j \le k. \quad (5)$$

It is not hard to observe that the worst-case scenarios occur when the system transits to full degraded mode at the very *beginning* of an interval – that would leave the maximum load of HI-criticality execution remaining to be done on the degraded system. For each ℓ, $1 \le \ell \le k$, suppose that the full degradation of the system occurs at time-instant t_ℓ; i.e., the start of the interval I_ℓ. Henceforth, only HI-criticality jobs need to be guaranteed meeting deadlines. Thus for each possible deadline $t_m \in \{t_{\ell+1}, t_{\ell+2}, \cdots, t_{k+1}\}$, constraints must be introduced to ensure that the cumulative remaining execution requirement of all HI-criticality jobs with deadline at or prior to t_m can complete execution by t_m on a system with m processors each of minimum speed s. This is ensured by the following constraint:

$$\Big(\sum_{i:(\chi_i = \text{HI}) \wedge (d_i \le t_m)} \big(\sum_{j=\ell}^{m-1} x_{i,j} \big) \Big) \le s \cdot m(t_m - t_\ell). \quad (6)$$

To see why this represents the requirement stated above, note that for any job J_i with $d_i \le t_m$, $\big(\sum_{j=\ell}^{m-1} x_{i,j}\big)$ represents the remaining execution requirement of job J_i at time-instant t_ℓ. The outer summation on the left hand side of Equation (6) is simply summing this remaining execution requirement over all the HI-criticality jobs that have deadlines at or prior to t_m.

A moment's thought should convince the reader that rather than considering all t_m's in $\{t_{\ell+1}, t_{\ell+2}, \cdots, t_{k+1}\}$ as stated above, it suffices to only consider those that are deadlines for HI-criticality jobs.

The entire linear program is listed in Figure 1. It is trivial that violating any of the constraints will result in *incorrectness* of the scheduling. Thus we conclude that these conditions are *necessary*. If it could be further shown that they are also *sufficient*, we may conclude the *optimality* of our algorithm.

However, unlike the uniprocessor case studied in previous work [5], to make these conditions sufficient here, we need to mimic a *processor-sharing* scheduling strategy. Discussions on converting the solution of LP into a correct schedule with processor-sharing will be provided in a later subsection, and *optimality* will be shown based on the assumption that we can partition each interval into small enough quanta so that processor speed does not change inside each quantum.

Bounding the size of this LP. It is not difficult to show that the LP with linear constraints (1) - (6) is of size polynomial in the number of jobs n in MC instance $\mathcal{I}$:

- The number of intervals k is at most $2n - 1$. Hence the number of $x_{i,j}$ variables is $O(n^2)$.
- There are n constraints of the forms (1) or (2), n constraints of the form (3), and $2k$ constraints of the forms (4) and (5). The number of constraints of the form (6) can be bounded by $(k \cdot n_h)$, since for each $\ell \in \{1, \ldots, k\}$, there can be no more than n_h of t_m's corresponding to the deadlines of HI-criticality jobs. Since $n_h \le n$ and $k \le (2n-1)$, it follows that the number of constraints is $O(n) + O(n) + O(n) + O(n^2)$, which is $O(n^2)$.

Since it is known that a linear program can be solved in time polynomial of its representation [11] [10], our algorithm for generating the scheduling tables for a given MC instance $\mathcal{I}$ takes time polynomial in the representation of I.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on April 28,2023 at 17:42:16 UTC from IEEE Xplore. Restrictions apply.

Given MC instance $\mathcal{I} = (\mathcal{J}, m, s)$, with job release-dates and deadlines partitioning the time-line over $[\min_i\{a_i\}, \max_i\{d_i\})$ into the k intervals $I_1, I_2, \ldots, I_k$.
Determine values for the x_{ij} variables, $i = 1, \ldots, n, j = 1, \ldots, k$ satisfying the following **constraints:**

- For each pair (i, j), $1 \leq i \leq n_h, 1 \leq j \leq k$,
$$0 \leq x_{i,j} \leq s(t_{j+1} - t_j).$$
- For each pair (i, j), $1 \leq i \leq n, 1 \leq j \leq k$,
$$0 \leq x_{i,j} \leq t_{j+1} - t_j.$$
- For each i, $1 \leq i \leq n$,
$$\Big(\sum_{j \mid t_j \geq a_i \ \wedge \ d_i \geq t_{j+1}} x_{i,j}\Big) \geq c_i.$$
- For each j, $1 \leq j \leq k$,
$$\Big(\sum_{i=1}^{n} x_{i,j}\Big) \leq m(t_{j+1} - t_j);$$
$$\Big(\sum_{i=1}^{n_h} x_{i,j}\Big) \leq s \cdot m(t_{j+1} - t_j).$$
- For each pair (ℓ, m), $1 \leq \ell \leq k$, $\ell < m \leq (k+1)$
$$\Big(\sum_{i:(\chi_i = \text{HI}) \wedge (d_i \leq t_m)} \Big(\sum_{j=\ell}^{m-1} x_{i,j}\Big)\Big) \leq s \cdot m(t_m - t_\ell).$$

Fig. 1. Linear program for determining the amounts to be finished for each job within each interval.

B. Run-Time Scheduling

Given a solution to the linear program constructed in the previous subsection, we now need to derive a run-time scheduling strategy that assigns an amount of execution $x_{i,\ell}$ to processors during the interval I_ℓ, for each pair (i, ℓ). According to the design of the linear program, run-time scheduling is now an interval-by-interval business – arrangements need to be made according to the table (calculated by the LP). We will show in this subsection how to mimic a processor-sharing schedule to execute mixed-criticality amounts *within each interval* in this possibly heterogenous system (some processors may degrade while others may not at certain instants in time).

Within a given interval I_ℓ, we denote $f_{i,\ell} = x_{i,\ell}/(t - t_\ell)$ as the allocated fraction for a given amount $x_{i,\ell}$. According to Inequalities (4) and (5), we can derive the following bounds of these fractions:

$$f_{i,\ell} \leq s, \forall 1 \leq i \leq n_h; \quad (7)$$

$$f_{i,\ell} \leq 1, \forall n_h < i \leq n. \quad (8)$$

Definition 4 (lag): For any interval I_ℓ and an assigned amount $x_{i,\ell}$, its lag at any instant $t \in [t_\ell, t_{\ell+1})$ (within the interval) is given by:

$$lag(x_{i,\ell}, t) = t \cdot f_{i,\ell} - executed(J_i, t). \quad (9)$$

Equation 4 defines a measurement to the difference between an ideal schedule and the actual execution of a given job. Under such a definition, we know that at any instant, non-negative lag for a job indicates that the schedule is correct *so far* with respect to this job. We will provide a strategy that guarantees zero lag at the end of each interval for all jobs while the system remains normal, and only for HI-criticality ones otherwise.

It should not be surprising that with (sufficient) preemption and migration, we can mimic a *processor-sharing* scheduling strategy that deals with this problem *correctly*. To mimic a processor-sharing scheduling strategy, jobs are simultaneously assigned fractional amounts of execution according to the solution of the LP. This can be done by partitioning the time-line into quanta of length Δ, where Δ is an arbitrarily small positive number. For each quantum, each job is executed for a duration of $f_{i,\ell} \cdot \Delta$, where $f_{i,\ell}$ has been defined to be the fraction of the job within Interval I_ℓ. In this way, by the end (and also at the beginning) of each quantum, *lag* for any job is zero, which leads to correctness of the scheduling (thus far).

Now we have further reduced the original scheduling problem into the following: given a quantum of length Δ, m ordered processing speeds $\{s_1 \geq s_2 \geq ... \geq s_m \geq s\}$, and assigned fractions of mixed-criticality amounts $\{f_1, f_2, ..., f_n\}$, how to construct a feasible schedule on this heterogenous system [3]? We can use the following algorithm to schedule the amounts for each quantum (with length Δ), which is, in a larger picture, mimicking a processor-sharing schedule over the whole interval I_ℓ.

Without loss of generality, we assume that all fractions are sorted into decreasing order, and job IDs change accordingly for each quantum; i.e., $s \geq f_1 \geq ... \geq f_{n_h}$, and $1 \geq f_{n_h+1} \geq ... \geq f_n$

Algorithm Wrap-Around-MC(Δ, f)

- At the beginning of each quantum (with length Δ), sort both processor speeds $s_1, ..., s_n$ and assigned fractions $f_1, ..., f_n$ in decreasing order.
- Use slower processors to execute HI-criticality jobs. Consider HI-criticality fractions one by one in increasing order (smallest fit first), where a processor will not be used until all slower processors has been fully utilized (wrap-around).
- If the system is in normal node; i.e., $s_i \geq 1, \forall i$, continue the "wrap-around" process for LO-criticality jobs on remaining faster processors.
- During execution, execute jobs on each processor following the same (priority) order of assignments in previous steps.

The following example shows how Wrap-Around-MC algorithm works.

[3]An important assumption is that changes to the speed of any processor *only* occur at quantum boundaries. In some sense this assumption is impractical. However, we may assume any processor's execution speed will not change dramatically within a short period (with length Δ). In this way, one can always "predict" how slow the processor can be in the near future. This pessimistic prediction will give us a lower bound on the execution speed of the following short period, and can serve as the "current" processor speed in our model.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on April 28,2023 at 17:42:16 UTC from IEEE Xplore. Restrictions apply.

Example 1: Consider five jobs $J_1 = J_2 = \{0, 0.4, 1, \text{HI}\}, J_3 = \{0, 0.5, 1, \text{HI}\}, J_4 = \{0, 0.3, 1, \text{LO}\}, J_5 = \{0, 0.7, 1, \text{LO}\}$, to be scheduled on a platform of three varying-speed processors with degraded speed 0.5. Since all jobs share the same scheduling window, there is only one interval $Ii = [0, 1)$, and the LP has the solution $x_{11} = x_{21} = 0.4, x_{31} = 0.5, x_{41} = 0.3, x_{51} = 0.7$. Figures 2 and 3 show how Wrap-Around-MC would schedule these jobs under normal and a possible degraded mode respectively. For easier representation and understanding, we assume $\Delta = 1$ in the example without loss of generality – a smaller Δ would result in repeating of a shrinking version of the same schedule.

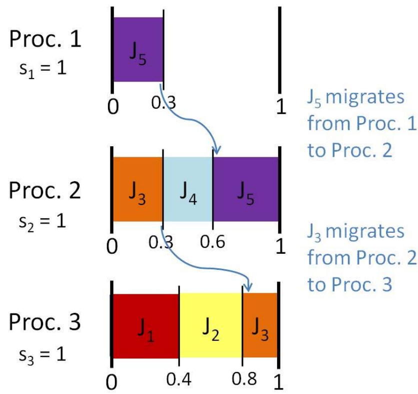

Fig. 2. The schedule constructed by Wrap-Around-MC under normal mode in Example 1.

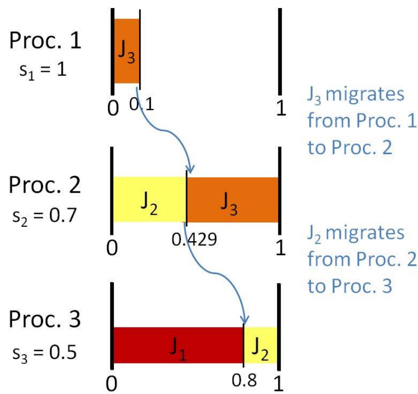

Fig. 3. The schedule constructed by Wrap-Around-MC under a given degraded mode in Example 1.

■

Theorem 1: Algorithm Wrap-Around-MC (in addition to the linear program construction) is an *optimal* correct scheduling strategy for the preemptive multiprocessor scheduling of a collection of independent MC jobs.

Proof: By *optimal*, we mean that if there exists a correct scheduling strategy (Definition 2 above) for an instance $\mathcal{I}$, then our scheduling strategy will succeed. From the definition, the obligation is to show that Wrap-Around-MC is able to correctly schedule any instance that can be correctly scheduled by any non-clairvoyant algorithm.

All inequalities defined in the linear program (1) – (6) have been shown to be necessary conditions. The optimality will come from the necessity of them – whenever Wrap-Around-MC returns fail, there must be some violations to the conditions, and thus no other non-clairvoyant algorithm can schedule this instance correctly.

What remains to be proved is this: given any solution to the LP, Algorithm Wrap-Around-MC will construct a correct scheduling strategy, so that these conditions are also sufficient.

We now show that parallel execution does not occur. In degraded mode, each processor remains a minimum execution speed of at least s. Since HI-criticality fractions are upper bounded by the same value (s), it is guaranteed that any HI-criticality job will not require a total execution time exceeding Δ. Thus with "wrap-around", the migrating jobs will not have any overlapping execution upon two different processors. Similar argument can be made regarding the LO-criticality jobs according to constraints (8) and normal mode processing speeds.

As far as each quantum follows Algorithm Wrap-Around-MC, lag of all jobs remains zero under normal mode, while lag of HI-criticality ones remains zero under degraded mode as well. From the definition of lag, we have shown that the conditions in LP are sufficient for the given algorithm to construct a *correct* schedule, and thus can conclude *optimality* of our algorithm.

■

The *optimality* of the algorithm tells us: (i) if all processors run in normal speed, all jobs will meet their deadlines; and (ii) if some (maybe all) processors run no slower than degraded speed s, HI-criticality jobs will meet their deadlines. We have not talked about how do deal with possible idleness during execution. Idleness is a critical issue in multi-processor platforms, and is difficult to treat optimally in varying-speed systems. The following item may be added into Algorithm Wrap-Around-MC:

- Whenever some processor idles (which indicates this processor will remain idle for the rest of this quantum), execute the LO-criticality job with earliest deadline that is assigned to next interval. If there is no LO-criticality job *active*, execute HI-criticality ones with similar attributes. Update the assigned value to further intervals by reducing the finished amount at the end of each interval.

Note that this item has nothing to do about *optimality* of the algorithm; i.e., leaving any processor idle as it is according to Algorithm Wrap-Around-MC will still result in correctness.

IV. Discussion

In this section, some further discussions about the problem and also the algorithm will be provided.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on April 28,2023 at 17:42:16 UTC from IEEE Xplore. Restrictions apply.

A. Processor-Sharing

We first show that processor-sharing is *necessary* in order to ensure that any instance for which the LP generates a solution can be scheduled during run-time.

In the following example, the mixed-criticality instance is compromised of three jobs and two processors. We will show that although the Linear Program has a feasible solution for this instance,, there does not exist a feasible schedule for this job set without processor-sharing.

Example 2: Consider three independent jobs $J_1 = J_2 = \{0, 1, 2, \text{HI}\}, J_3 = \{0, 2, 2, \text{LO}\}$, to be scheduled on a platform of two varying-speed processors with degraded speed 0.5. Since all jobs share the same scheduling window, there is only one interval $I_i = [0, 2)$, and the LP has the solution $x_{11} = x_{21} = 1$ and $x_{31} = 2$.

Wrap-Around-MC will execute this set of jobs easily by combining the HI-criticality ones together and executing them on one processor while the system remains normal. Job J_3 can be dropped whenever the system begins to suffer from degradation. Here processor-sharing gives us the ability to execute any fraction of a job within a short enough quantum (with length Δ).

Under the case where processor-sharing is forbidden, we can still assume processors do *not* change their speeds during each quantum. The only difference is that we can no longer assign a fraction of capacity to each quantum; one certain job needs to be assigned to a given processor within each quantum. We will show that no matter how small Δ is, there does not exist a feasible schedule for this job set (without processor-sharing).

Consider two possible decisions at time $t = 0$ (for the next quantum) – we may either assign both two processors the HI-criticality jobs, or allocate the LO-criticality job to one of them.

The first choice is certainly not correct in the case both processors never degrades. To make sure the LO-criticality job with utilization of 1 meets its deadline, it needs to be executing for the whole interval. However, the LO-criticality job will not start to execute until $t = \Delta$ under this decision. Since a job cannot be executed on both processors in parallel, remaining capacity $2 - \Delta$ on either processor is not enough for the LO-criticality job to meet its deadline.

For the second choice, consider the case that both processors degrade into 0.5-speed at instant $t = \Delta$. There remains a HI-criticality job (assumed to be J_2, without loss of generality) which requires an execution of 1 time unit within the interval. However each processor has a remaining capacity of $(2-\Delta) \cdot 0.5$, which is smaller than 1. Since a job cannot be executed on both processors in parallel, the remaining capacity on either processor is not enough for J_2 to be finished on time. The wasted Δ capacity (used for executing the LO-criticality job) of the system is crucial (and unavoidable).

■

The problem without processor-sharing is that we can no longer guarantee that upon any instant that the system may degrade, we will execute a fraction of 0.5 to both HI-criticality jobs on one processor. The lag of some HI-criticality job may be negative, which means the constructed schedule is "left behind" when compared to the ideal case. The key assumption in processor-sharing is that processor speed will not change throughout each quantum. This gives us the ability to execute each job a proper length which leaves a zero lag after each period of length Δ.

B. Weak Degradation

So far we have focused upon a rather restrictive model that places a relatively *strong* requirement on system behavior during degraded mode: all processors must execute at a minimum speed of s. The requirement is *strong* since we eliminate the case when only a few among m processors are not functional, while most ones execute at full speed – the whole system may still be able to ensure a cumulative speed of $s \cdot m$.

We now introduce a somewhat different definition of system degradation described as follows.

Definition 5 (weak degraded mode): A system with m processors is said to be in *weak degraded mode* at a given instant if the processing speeds $\{s_i\}$ of all processors satisfy:

$$\sum_{j=1}^{m} s_i \geq s \cdot m. \tag{10}$$

The requirement is considered *weak* because if the m processors are executing with an average speed no slower than s, correctness must be guaranteed for HI-criticality jobs. Now it includes the annoying case that several processors may run at a very low (but not zero) speed, and they need to be well utilized for some heavy load instances.

The following simple example shows how Algorithm Wrap-Around-MC will fail in weak degraded mode for a *feasible* job set.

Example 3: Consider two jobs $J_1 = J_2 = \{0, 0.5, 1, \text{HI}\}$, to be scheduled on a varying-speed platform of two processors with degraded speed 0.5. Since both jobs share the same scheduling window, solution $x_{11} = x_{12} = 0.5$ to the LP is trivial.

Now consider the case if at the very beginning Processor 1 degrades into speed 0.75, while Processor 2 degrades into speed 0.25. Although the system is no longer in degraded mode; it still satisfies the weak degradation definition. Figure 4 compares the incorrect result by Wrap-Around-MC (where the dotted box marks the parallel execution period) and a possible correct scheduling strategy.

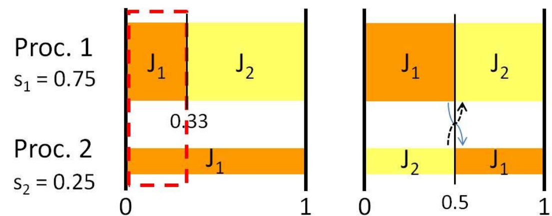

Fig. 4. The incorrect schedule constructed by Wrap-Around-MC (left), and a feasible one (right) under weak degraded mode in Example 3.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on April 28,2023 at 17:42:16 UTC from IEEE Xplore. Restrictions apply.

■

This example shows that wrap-around is no longer optimal under weak degraded mode. Additional "slices" inside each quantum need to be made, so that jobs will migrate and get rid of parallel execution. In general, for optimal scheduling on this kind of heterogenous system, studies have been made, and the current state of art suggests the adaptation of the Level Algorithm [9].

The Level Algorithm creates a significantly large number ($O(m^2)$) of preemptions and migrations for each short period (quantum), in order to fully utilize all slow processors with jobs that need to execute for a considerable duration during this quantum without running into the parallel execution problem. With the help of (the optimal) Level Algorithm, we can extend Algorithm Wrap-Around-MC as follows to correctly deal with systems in weak degraded mode.

Algorithm Level-MC(Δ, **f**)

- At the beginning of each quantum (with length Δ), order both the processor speeds $s_1, ..., s_n$ and the assigned fractions $f_1, ..., f_n$ in decreasing order.
- **If** the system is in normal mode, "wrap-around" all jobs.
- **Elseif** the system is in degraded mode, "wrap-around" HI-criticality jobs.
- **Elseif** the system is in weak degraded mode, apply the Level Algorithm to HI-criticality jobs.
- During run-time, in both the normal and the degraded modes, jobs are assigned the priority order same as the assignment order in the steps above, and are executed on their allocated processors. In weak degraded mode, priorities of jobs are not fixed, and the detailed schedule is given by the Level Algorithm.

The following example illustrates how Algorithm Level-MC works under weak degraded mode.

Example 4: Consider four HI-criticality jobs $J_1 = \{0, 0.2, 1, \text{HI}\}, J_2 = \{0, 0.25, 1, \text{HI}\}, J_3 = \{0, 0.4, 1, \text{HI}\}, J_4 = \{0, 0.5, 1, \text{LO}\}$, to be scheduled on a platform of three varying-speed processors with a minimum weak degraded speed threshold of 0.5. Consider the weak degraded case where three processors run at speeds of $0.3, 0.4$, and 0.8, respectively (the average speed of the system is 0.5).

Since all jobs share the same scheduling window, there is only one interval $I_i = [0, 2)$, and the LP has the solution $x_{11} = 0.2, x_{21} = 0.25, x_{31} = 0.4$, and $x_{41} = 0.5$. Figure 5 shows how Level-MC would schedule these jobs under such weak degraded mode for the next quantum. Without loss of generality, we assume unit length for each quantum; i.e., $\Delta = 1$. A shorter quantum length would result in repeating of a shrunk version of the same schedule pattern.

In the schedule shown in Figure 5, jobs are *jointly* executing on more than one processor during some intervals; e.g., jobs J_1 and J_2 during interval $[0.133, 0.5)$, jobs J_3 and J_4 during interval $[0.25, 0.5)$, and all jobs during interval $[0.5, 0.9)$. The Level Algorithm designs the schedule in a way that capacity as well as execution speeds are evenly divided (shared) by combined jobs. The intuition is that since a heavy job executes

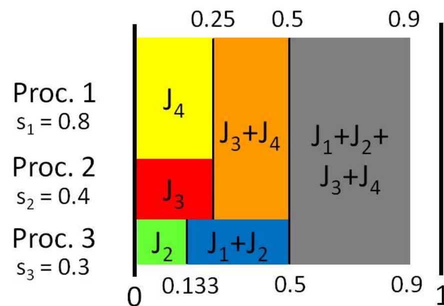

Fig. 5. The schedule constructed by Level-MC under weak degraded mode in Example 4.

on a high-speed processor, there may be an instant that two (or more) jobs have the same amount left (to be executed). For example, in the schedule given by Figure 5, both J_1 and J_2 require 0.2 time units of further execution at time $t = 0.133$. From then on, they should execute at the same speed, and thus are *joined* by the Level Algorithm.

To *jointly* execute n jobs on m processors, where $n \geq m$, the Level Algorithm divides the period into n equal subperiods, and makes the assignment that each processor executes (and only executes) each job for one subperiod. Figure 6 expresses the schedule for all the jobs by this divide-and-assign scheme.

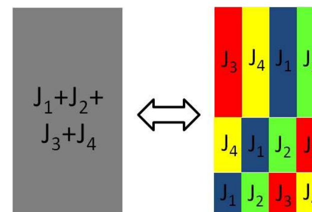

Fig. 6. Joint execution of all jobs on the system by Level Algorithm during $[0.5, 0.9)$ of Example 4.

■

Theorem 2: Algorithm Level-MC (in addition to the linear program construction) is an *optimal* correct scheduling strategy for the preemptive multiprocessor scheduling of collections of independent MC jobs.

Proof: Similar to the proof in Theorem 1, we only need to show that the weak degraded condition is also *sufficient* for Level-MC to construct a correct schedule.

According to [9], the Level Algorithm will always return a feasible schedule if the following m constraints hold (assume both $\{f_i\}$ and $\{s_j\}$ are in *decreasing* order):

$$\sum_{j=1}^{i} f_j \leq \sum_{j=1}^{i} s_j, \forall i, 1 \leq i \leq m-1; \tag{11}$$

$$\sum_{j=1}^{n_h} f_j \leq \sum_{j=1}^{m} s_j. \tag{12}$$

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on April 28,2023 at 17:42:16 UTC from IEEE Xplore. Restrictions apply.

From Inequality (7), we have $f_j \leq s$, for any j. Since $\{s_j\}$ are ordered in decreasing order, from the property of "average", we know that $s \cdot i \leq \sum_{j=1}^{i} s_j$ holds true for any i. Putting these together, we have Inequality (11). Inequality (12) follows directly from the capacity constraint (5).

As a consequence, under such a processor-sharing protocol, the Level Algorithm returns a feasible schedule within each quantum (a small enough interval of length Δ). Here feasibility indicates that no job gets executed simultaneously on more than one processor, and all jobs receive their designated amounts by the end of the quantum. As the system continues to run quantum by quantum, the HI-criticality amounts are guaranteed to be finished by their assigned fractions (with zero lag). This indicates all HI-criticality jobs will meet their deadlines when the system is in weak degraded mode.

Correctness in both normal mode and degraded mode (each with a processing speed no less than s) follows from Theorem 1 since no change has been made from Algorithm Wrap-Around-MC for these cases.

We have shown that Inequalities (1) – (6) are *sufficient* for Algorithm Level-MC to construct a *correct* schedule. Since it has been shown that these conditions are also necessary, we can conclude the *optimality* of the algorithm. ■

Dropping LO-criticality jobs. Under the definition of *correctness*, the two algorithms proposed so far drop all LO-criticality jobs whenever degradation occurs (even to only one of the processors). One can certainly argue that such sacrifice may not be necessary.

Inequalities (11) and (12) can also be applied to all jobs (instead of only HI-criticality ones) to check the feasibility of the current system (described by processing speeds). The following item can be added into Algorithms Wrap-Around-MC and Level-MC to further improve them by not dropping the LO-criticality jobs in some of the degraded cases:

- If the system is in degraded (or weak degraded) mode, check feasibility conditions for all jobs; i.e., Inequalities (11) and (12). If they hold, apply the Level Algorithm to all jobs; else follow the previous protocols to the HI-criticality jobs only, and suspend the LO-criticality ones.

However, whether *optimality* can be proved under such protocol remains unknown; i.e., if our algorithm drops any LO-criticality job under certain degradation condition(s), is it necessarily the case that other algorithm(s) must drop some LO-criticality job(s) to guarantee correctness?

V. Context & Conclusions

We have recently begun studying the scheduling of mixed-criticality systems upon platforms that may suffer degradations in performance during run-time. We expect that these processors are likely to execute at unit speed (or faster) during run-time, and in some cases they execute slower, but each at a speed no slower than some specified threshold s. Upon such platforms, the scheduling objective is to ensure that all jobs complete in a timely manner under normal circumstances, while simultaneously ensuring that more critical jobs complete in a timely manner under degraded conditions.

In this paper, we expand our previous investigations upon uniprocessor platforms into the scheduling of mixed-criticality jobs upon multicore systems. Upon such varying-speed multiprocessors, a *correct* and *optimal* scheduling algorithm named Wrap-Around-MC has been proposed. During execution, our algorithm mimics a processor-sharing scheme, which requires sufficient preemption and migration (assumed to be zero cost).

We also initiate some studies on a *weaker* definition of degradation to multicore systems, where processors on average execute at a minimum speed of s. The Level Algorithm [9] is executed within each quantum to maintain the optimality, which results in much more (unavoidable) preemption and migration during execution. It would be interesting and important to derive algorithms for scheduling mixed-criticality varying-speed systems with limited (such as bounding their numbers) or restricted (such as only allowing them at job boundaries) preemption and migration.

So far we only considered collections of independent jobs. We are certainly going to make an effort to better understand the scheduling of sporadic task set in the future work.

References

[1] Theodore Baker and Sanjoy Baruah. Sustainable multiprocessor scheduling of sporadic task systems. In *Proceedings of the EuroMicro Conference on Real-Time Systems*. IEEE Computer Society Press, 2008.

[2] S. Baruah and et al. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012. IEEE Computer Society.

[3] Sanjoy Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transactions on Computers*, 53(6), 2004.

[4] Sanjoy Baruah, Bipasa Chattopadhyay, Haohan Li, and Insik Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 50(1):142–177, 2014.

[5] Sanjoy Baruah and Zhishan Guo. Mixed-criticality scheduling upon varying-speed processors. In *Proceedings of the 34th IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society Press, 2013.

[6] David Bull and et al. A power-efficient 32b arm isa processor using timing-error detection and correction for transient- error tolerance and adaptation to pvt variation. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 284–285, 2010.

[7] Alan Burns and Robert Davis. Mixed-criticality systems: A review. http://www-users.cs.york.ac.uk/~burns/review.pdf, 2013.

[8] Zhishan Guo and Sanjoy Baruah. Mixed-criticality scheduling upon unmonitored unreliable processors. In *Proceedings of the IEEE Symposium on Industrial Embedded Systems (SIES)*. IEEE Computer Society Press, 2013.

[9] E.C. Horvath, S. Lam, and R. Sethi. A level algorithm for preemptive scheduling. *Journal of the ACM*, 24(1):32–43, 1977.

[10] N. Karmakar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[11] L.G. Khachiyan. A polynomial algorithm in linear programming. *Dokklady Akademiia Nauk SSSR*, 244:1093–1096, 1979.

[12] Risat Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012.

[13] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the Real-Time Systems Symposium*, pages 239–243, Tucson, AZ, December 2007. IEEE Computer Society Press.

[14] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, Pisa (Italy), 2012. IEEE Computer Society Press.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on April 28,2023 at 17:42:16 UTC from IEEE Xplore. Restrictions apply.