Mixed-Criticality Processing Pipelines

Dionisio de Niz[1], Bjorn Andersson[1], Hyoseung Kim[2], Mark Klein[1], Linh Thi Xuan Phan[3], and Raj Rajkumar[1]
Carnegie Mellon University[1] U.C. Riverside[2] University of Pennsylvania[3]

***Abstract*—While a number of schemes exist for mixed-criticality scheduling in a single processor setting, no solution exists to cover the industry need for end-to-end scheduling across multiple processors in a pipeline. In this paper, we present an end-to-end zero-slack rate-monotonic scheme (ZSRM) based on real-time pipelines, called the *ZSRM pipeline* scheduler, that addresses this need. Under ZSRM, each task is associated with a parameter called *zero-slack instant*, and whenever a higher-criticality job has not finished at its zero-slack instant relative to its arrival time, all jobs of lower criticality are suspended to meet the deadline of the higher-criticality job. We develop a new schedulability test and algorithm for computing the zero-slack instants of tasks scheduled across a pipeline.**

I. Introduction

Mixed-criticality (MC) scheduling provides a way to handle variable execution times for tasks with different criticalities. In this setting new guarantees are offered to ensure that, if deadlines are missed due to overload, these deadlines are missed in the reverse order of criticality. However, such scheduling has not been considered for processing pipelines, which are common in real systems. Thus, the goal of this paper is to develop MC scheduling for processing pipelines.

The research community has developed solutions for processing pipelines without considering MC. Holistic schedulability analysis [10] is an approach which considers the behavior of a job across multiple resources and does not require end-to-end deadline decomposition. Unfortunately, this approach considers that for each stage, a task may experience the worst-case interference from higher-priority tasks; thus, it can be very pessimistic for long pipelines. To address this issue, Jayachandran and Abdelzaher [7] developed a real-time pipeline scheduling approach where a task is composed of a sequence of stages that run on different processors and their schedulability test exploits the fact that a low-priority task running in a stage can execute in parallel with a high-priority task running in the next stage of the pipeline. This reduces the interference that the low-priority task suffers when running in the next stage. Other works consider a task as a sequence of segments with potential suspension between these segments [9] but only considers soft deadlines.

The research community has developed solutions for MC scheduling but without pipelines [1], [6], [8], [2] — see also [3] for an excellent survey. Zero-Slack Rate-Monotonic (ZSRM) [5] is one solution where a zero-slack timer is started upon the arrival of each job and all lower-criticality jobs are suspended if the timer elapses and the corresponding job has not finished. Tasks have a nominal and an overload worst-case execution time and are guaranteed to execute for its overload execution time if no higher-criticality tasks execute for more than their nominal. The work that is closest to our goal is [4] which offers MC scheduling of multihop traffic on NoC. While it is possible to think of a message flow as a task and the links of a flow's route as processing resources, [4] does not solve our problem. This is because (i) we are interested in allowing a job to have different execution time on different stages (the transmission time of a message in [4] is the same on all links) and (ii) we are interested in preemptive scheduling (a flit in [4] is non-preemptive).

In this paper, we present the *ZSRM pipeline* scheduler — the first solution for MC scheduling on processing pipelines. We do so by leveraging the work by Jayachandran and Abdelzaher [7] to extend the Zero-Slack Rate-Monotonic (ZSRM) scheduler [5] to pipelines in order to support MC tasksets with end-to-end deadlines in an efficient way.

II. System Model

We define a system $\mathcal{S} = (\Gamma, \Pi = \{\pi_1, \ldots, \pi_N\})$, where Γ is a taskset and Π is the set of N processors in the pipeline with π_j denoting the j^{th} processor of this pipeline. Each processor of the pipeline represents a unique stage. Hence, we use the terms *processors* and *stages* interchangeably in the rest of the paper. A task $\tau_i \in \Gamma$ is characterized as $(T_i, D_i, \zeta_i, \{C_{i,j}\}, \{C^o_{i,j}\})$ where T_i is the period of τ_i, D_i is the relative deadline ($D_i \leq T_i$), ζ_i is the criticality of τ_i, and $C_{i,j}$ and $C^o_{i,j}$ are, respectively, the worst-case *nominal* and *overload* execution times of any job of τ_i in stage j ($C_{i,j} \leq C^o_{i,j}$). Task priorities and criticalities do not change at runtime. We let ρ_i denote the priority of τ_i. Tasks have unique priorities but not unique criticalities.

We use $C_{i,\max 1}$ and $C_{i,\max 2}$ to denote the largest and second largest stage execution times of τ_i among all of its stages, respectively. The terms C_i and C^o_i denote the sum of all per-stage nominal and overload execution times of τ_i. Hence, $C_i = \sum_{k=1}^{N} C_{i,k}$ and $C^o_i = \sum_{k=1}^{N} C^o_{i,k}$. We also use the following notation to describe tasks that interfere with τ_i in an MC system:

- H_i^{hc}: tasks with higher priority and higher criticality than τ_i
- H_i^{sc}: tasks with higher priority and same criticality as τ_i
- H_i^{lc}: tasks with higher priority and lower criticality than τ_i
- L_i^{hc}: tasks with lower priority and higher criticality than τ_i

Overload condition. Although our model captures the worst-case nominal and overload execution times per stage, we test the overload condition in a global fashion. Specifically, we say that a job of task τ_i overloads when/if its accumulated execution time across its stages exceeds $\sum_{k=1}^{N} C_{i,k}$. This approach can yield better schedulability compared with an alternative approach where the overload condition is checked on a per-stage basis (i.e., a job of τ_i is overloaded if its per-stage execution time in any stage k exceeds $C_{i,k}$).

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

[Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

DM-0003976

III. Background

Jayachandran and Abdelzaher [7] presented a method for computing an upper bound on the delay that a job can experience across all stages in a pipeline. In their model a task τ_i has one worst-case execution time $C_{i,j}$ per stage j, executed periodically with a period T_i and deadline D_i. For this model the authors built a synthetic task model to calculate the worst-case end-to-end response time of a task executing in a pipeline.

In their response-time test, a pipeline taskset is transformed into an equivalent single-stage taskset with the following two types of synthetic execution times: the one given by Eq. (1) for τ_i's own execution time, and the other one given by Eq. (2) for the execution time of higher-priority tasks. Then, the worst-case response time of τ_i (denoted R_i) is computed as the smallest fixed point of Eq. (4).

$$C_e^*(i) = \sum_{j|\rho_j \geq \rho_i} C_{j,\max1} + \sum_{s=1}^{N-1} \max_{j|\rho_j \geq \rho_i} (C_{j,s}) \tag{1}$$

$$C_j^* = C_{j,\max1} + C_{j,\max2} \tag{2}$$

$$R_i^{(0)} = C_e^*(i) \tag{3}$$

$$R_i^{(k)} = C_e^*(i) + \sum_{j|\rho_j > \rho_i} \left\lceil \frac{R_i^{(k-1)}}{T_j} \right\rceil C_j^* \tag{4}$$

Zero-Slack Rate-Monotonic (ZSRM) scheduling [5] is a MC scheduler developed for a non-pipeline system, i.e., the number of stages $N = 1$. Under ZSRM, a task has two parameters C_i and C_i^o for nominal and overload WCET and two execution modes: *normal* and *critical*. When the job of a task τ_i arrives, it starts executing in normal mode, where the Deadline-Monotonic policy is used for scheduling. However, towards the end of its execution, the job may switch to the critical mode if it reaches its *zero-slack instant*. The zero-slack instant of τ_i (Z_i) is an offline bound on the last instant when it is possible to stop lower-criticality tasks and still complete C_i^o units of execution before its deadline. The ZSRM runtime, in fact, suspends the low-criticality tasks whenever Z_i is reached (the calculation of Z_i will be discussed shortly in Section IV). ZSRM guarantees that a task τ_i can execute for C_i^o before its deadlines if no task τ_j with $\zeta_j > \zeta_i$ executes for more than its nominal execution time C_j. To achieve this guarantee, an enforcement mechanism must ensure that, if a task τ_i runs for more than C_i, the execution of any lower-criticality task τ_k suspended or arrived during τ_i's job execution in critical mode is deferred until τ_k's current period elapses.

IV. Zero-Slack Instant Calculation Reformulation

This section presents the calculation of zero-slack instants under ZSRM. The calculation is based on the response-time test given in [5], but reformulated to simplify the reuse of the pipeline schedulability test from [7]. We calculate the zero-slack instant Z_i for each task τ_i in the taskset such that with the resulting zero-slack instants, this taskset is schedulable. During this calculation, temporal bounds are calculated on the execution time that each task τ_i can perform in normal mode, C_i^n, and in critical mode, C_i^c. We start by iterating over all tasks in the taskset in descending order of criticality and for each task τ_i, compute Z_i as follows. At first, set $Z_i = 0$, $C_i^c = C_i^o$, and $C_i^n = 0$, i.e., all of its execution is done in its critical mode assuming that it executes for its overload execution time C_i^o. Then, compute the response time of τ_i in its critical mode, R_i^c, using Eq. (5) which can be solved with fixed-point iteration.

$$R_i^c = C_i^c + \sum_{j \in H_i^{hc}} \left\lceil \frac{R_i^c}{T_j} \right\rceil C_j + \sum_{j \in L_i^{hc}} \left\lceil \frac{R_i^c}{T_j} \right\rceil (C_j - C_j^n) + \sum_{j \in H_i^{sc}} \left\lceil \frac{R_i^c}{T_j} \right\rceil C_j^o \tag{5}$$

Note that, as mentioned in the previous section, the guarantee offered to task τ_i assumes different execution times of interfering tasks depending on their criticality. In particular, a task τ_j with higher criticality and higher priority than τ_i is assumed to run for at most C_j, given that if it executes beyond C_j, the execution of τ_i may be deferred (to let τ_j complete). A task τ_j with higher criticality and lower priority than τ_i is assumed to execute for up to $C_j - C_j^n$, given that τ_j's execution in its normal mode does not interfere with τ_i. A task τ_j with the same criticality as and higher priority than τ_i is assumed to execute for up to C_j^o. These are reflected in Eq. (5).

$$Z_i = D_i - R_i^c \tag{6}$$

$$I_i = \sum_{j \in H_i^{hc}} \left\lceil \frac{Z_i}{T_j} \right\rceil C_j + \sum_{j \in H_i^{lc} \cup H_i^{sc}} \left\lceil \frac{Z_i}{T_j} \right\rceil C_j^o \tag{7}$$

$$S_i^n = \max(0, Z_i - I_i - C_i^n) \tag{8}$$

$$C_i^c = \max(0, C_i^c - S_i^n) \tag{9}$$

$$C_i^n = \min(C_i^o, C_i^n + S_i^n) \tag{10}$$

Then, Eq. (6) calculates how much it is possible to increase Z_i without missing the deadline. This is done by subtracting the critical mode response time from the deadline. Z_i also indicates the elapsed time of τ_i in its normal mode (from 0 to Z_i). Hence, it is used to calculate the interference from higher-priority tasks in the normal mode, I_i, in Eq. (7).

Using the calculated interference I_i, Eq. (8) calculates the slack in normal mode, S_i^n, by subtracting from the length of normal mode Z_i the execution time in normal mode C_i^n and the interference I_i. Then, we move computation from critical mode to normal mode by subtracting the slack S_i^n from the critical mode execution time C_i^c and adding it to the normal mode execution time C_i^n, as shown in Eqs. (9) and (10), respectively. With these new execution times, a new response time, slack, and execution times are calculated again until convergence is achieved obtaining the final Z_i of task τ_i.

The resulting Z_i of a task τ_i can be either (i) $Z_i < 0$, meaning the taskset is unschedulable, (ii) $0 \leq Z_i < D_i$, meaning τ_i could execute partially in critical mode, or (iii) $Z_i = D_i$, meaning τ_i will not execute in critical mode.

Let us now discuss the reasoning behind this procedure. First, note that in Eq. (5), there is no summation over tasks in H_i^{lc}; the reason for this is because when a job executes in its critical mode, it is not impacted by the execution of lower-criticality tasks.

Second, the computation of R_i^c (expressed by Eq. (5)) does not depend on the Z values of lower criticality tasks. The computation of I_i (expressed by Eq. (7)) depends on lower-criticality tasks but only on their T and C parameters; not on their Z parameter. Hence, the calculation of Z_i of a task does not depend on the values of Z of lower criticality tasks. However, in Eq. (5), the computation of R_i^c depends on C_j^n (where τ_j is a higher criticality task), which in turn depends on Z_j. In conclusion, the calculation of Z of a task depends on Z of higher-criticality tasks but not on Z of lower-criticality tasks. It is for this reason that we calculate Z of tasks in decreasing order of criticality.

Third, in Eq. (7) there is no summation over tasks in L_i^{hc}. This is due to the following reason. Consider two tasks τ_i and τ_j with $\tau_j \in L_i^{hc}$. Because of our assumption of priorities being assigned according to deadline monotonic, it holds that $D_i \leq D_j$ and because of our assumption of constrained-deadlines, it holds that $D_j \leq T_j$. Combining them yields $D_i \leq T_j$ and hence, there can be at most one job of τ_j whose execution impacts a job of τ_i. We can change the arrival time of this single job of τ_j so that the arrival time of τ_j plus Z_j is equal to the arrival time of τ_i plus Z_i, and after this change, the response time of the job of τ_i remains unchanged or increases. Hence, from the perspective of computing the worst-case response time of τ_i, it is sufficient to consider that τ_j only performs execution in the critical mode of τ_i. As a result, we consider L_i^{hc} only in Eq. 5 and not in Eq. 7.

V. ZSRM Pipeline Scheduling

ZSRM pipeline scheduling has (just like its uniprocessor counterpart) an offline Z calculation and an online enforcement mechanism. We first discuss the zero-slack calculation.

A. Overview of Zero-Slack Pipeline Scheduling

In broad terms, to schedule a ZSRM pipeline taskset we calculate the end-to-end zero-slack instant for each of the tasks in decreasing order of criticality. To calculate the zero-slack instant of a task τ_i we follow the same strategy as in Section III creating an interfering taskset for τ_i's normal mode and another for its critical mode. This is done by extending the task transformation mentioned in Section III by taking into account the different interfering execution times of the tasks that depend on their relative criticalities. With these two tasksets, we initially assume that a task τ_i starts executing all of its overload execution time in critical mode and calculate the first zero-slack instant. We also calculate the zero-slack stage, which is the first stage where the zero-slack instant can occur. This allows us to separate the stages into normal stages and critical stages, reflecting the execution mode of the task. Then, compute the slack in normal mode and move part of the computation from critical mode to the slack in normal mode. Next, recalculate the zero-slack instant with the new computation in critical mode. Repeat these steps until the zero-slack instant converges.

Before providing the details of the zero slack calculation, we first need to model the interference that a task $\tau_j \in L_i^{hc}$ can cause on τ_i when τ_j reaches its zero-slack instant Z_j and executes in critical mode. In order to do that let us denote the first stage where Z_j can happen as σ_j^z (to be explained in Eq. (21)). Now, because τ_j does not interfere with τ_i in any stage $\pi_j | j < \sigma_j^z$ we need to model it as a new arrival at stage $\pi_{\sigma_j^z}$.

Every time we calculate the zero-slack instant of τ_i (explained below) along with the zero-slack instant stage σ_i^z we check if $\sigma_i^z = \sigma_j^z$, i.e., if Z_i and Z_j occur in the same stage. If that is the case, we align Z_j with Z_i to ensure that we capture the worst-case phasing of τ_j's interference on τ_i as explained in Section IV. However, if $\sigma_i^z \neq \sigma_j^z$ we align Z_j at the beginning of the τ_i's stage where it occurs.

B. Revisiting Single Processor Analysis

Recall the zero-slack computation procedure for a single processor system described in Section IV. We now discuss how to transition this analysis to pipelines. For this discussion we use the following notation. Γ_i^c is the interfering taskset in τ_i's critical mode; formally $\Gamma_i^c = L_i^{hc} \cup H_i^{hc} \cup H_i^{sc}$. Γ_i^n is the interfering taskset in τ_i's nominal mode; formally $\Gamma_i^n = H_i^{hc} \cup H_i^{lc} \cup H_i^{sc}$. We also define the execution variable of an interfering task as $C_j^{e_i}$ as follows

$$C_j^{e_i} = \begin{cases} C_j & \text{if } \tau_j \in H_i^{hc} \\ (C_j - C_j^n) & \text{if } \tau_j \in L_i^{hc} \\ C_j^o & \text{if } \tau_j \in H_i^{lc} \cup H_i^{sc} \end{cases} \quad (11)$$

We use Eq. (11) to rewrite the response time of τ_i in critical mode from Eq. (5) as presented in Eq. (12) and the interference in normal mode from Eq. (7) as presented in Eq. (13).

$$R_i^c = C_i^c + \sum_{j \in \Gamma_i^c} \left\lceil \frac{R_i^c}{T_j} \right\rceil C_j^{e_i} \quad (12) \qquad I_i = \sum_{j \in \Gamma_i^n} \left\lceil \frac{Z_i}{T_j} \right\rceil C_j^{e_i} \quad (13)$$

We will now discuss how to adapt Eqs. (12) and (13) to processing pipelines. In order to determine $C_j^{e_i}$ we need to take into account whether an interfering task τ_j overloads or not according to its criticality and in which stages. In particular, note that the pipeline response time calculation in [7] uses the maximum (and second maximum) execution times across all stages and all tasks. In particular, we will develop new equations for the pipeline case where a new $C_j^{e_i}$ term for each stage s ($C_{j,s}^{e_i}$) will be used determining if it overloads or not in stage s. These will be used in Eq. (14). The calculation of $C_{j,s}^{e_i}$ is necessary because if we were to select an execution time of say $C_{j,s}$ instead of $C_{j,s}^o$ in a stage s for an interfering task τ_j this may not get selected by the max function of Eq. (14). This is a problem because for interfering tasks τ_j with higher-criticality than τ_i we must only consider its end-to-end nominal execution time C_j allowing it to overload in any single stage (according to our end-to-end overload semantics). We discuss this in the next section.

C. Taskset Tranformations

Let us now discuss how to create MC pipelined tasksets. Specifically, we create two tasksets: one for the normal execution mode and the other for critical mode. In these tasksets we will select the computation time for each task τ_i for which we will calculate its zero-slack instant and its interfering tasks τ_j's. Then, we start by assuming that all the computation of τ_i occurs in critical mode. Hence, we initialize the set of critical stages as $\Pi_i^c = \{\pi_j | 1 \leq j \leq N\}$. Note that, even though we can calculate the minimum zero-slack stage, we assume that it starts in the first stage to start with all the computation in critical mode.

1) Taskset Transformation for Critical Mode: For the taskset in critical mode we first initialize the execution time in critical mode of each task τ_i with $C_e^c(i) = C_e^K(i)$, where $C_e^K(i)$ is defined as in Eq. (14).

$$C_e^K(i) = \sum_{\tau_j \in \Gamma_i^c \cup \{\tau_i\}} C_{j,\max1}^{c_i} + \sum_{\pi_{i,s} \in \Pi_i^c \setminus \{\pi_N\}} \max_{\tau_j \in \Gamma_i^c \cup \{\tau_i\}} C_{j,s}^{e_i} \tag{14}$$

We identify $C_{j,\max1}^{c_i}$ and $C_{j,\max2}^{c_i}$ as the largest and the second largest overload execution time of τ_j's stages and refer to their respective stages as $\pi_{j,\max1}$ and $\pi_{j,\max2}$. $C_{j,s}^{e_i}$ is calculated by solving

$$\text{maximize} \sum_{s \in \Pi_i^c \setminus \{\pi_N\}} \max_{j \in \Gamma_i^c \cup \{\tau_i\}} \{x_{j,s}\} \text{ subject to}$$

$$\forall \langle j, s\rangle \text{ s. t. } (j \in \Gamma_i^c \cup \{\tau_i\}) \wedge (s \in \Pi_i^c), x_{j,s} \leq C_{j,s}^o$$

$$\forall j \text{ s. t. } (j \in \Gamma_i^c \cup \{\tau_i\}) \wedge (\zeta_j > \zeta_i), \sum_{s \in \Pi_i^c} x_{j,s} \leq \sum_{s \in \Pi_i^c} C_{j,s}$$

where $x_{j,s}$ are non-negative real numbers. This can be formulated as a Mixed-Integer Linear Program; a problem for which a large number of efficient solvers are available (we use Gurobi). It is worth noting that for tasks in L_i^{hc} the search for the maximums will be limited to $\Pi_j^{c,e} = \Pi_j^c \cap \Pi_i^c$, i.e., stages where both τ_i and τ_j run in critical mode.

Note that Eq. (14) is equivalent to the original Eq. (1) in the basic pipeline scheme but limited to the execution within the set of critical stages (denoted by Π_i^c).

Then, the execution times of the interfering tasks are initialized with $C_j^{*c_i} = C_{j,\max1}^{c_i} + C_{j,\max2}^{c_i}$ and the response time with $R_i^c = C_e^c(i)$ finding the final value of the response time with the fixed-point computation of Eq. (15).

$$R_i^c = C_e^c(i) + \sum_{j \in \Gamma_i^c} \left\lceil \frac{R_i^c}{T_j} \right\rceil C_j^{*c_i} \tag{15}$$

2) Taskset Transformation for Normal Mode: The nominal execution time for task τ_i is calculated by first identifying the largest and second largest execution times of the nominal mode as $C_{j,\max1}^{n_i}$ and $C_{j,\max2}^{n_i}$ respectively among the nominal stages Π_i^n. The stage set Π_i^n is initialized to $\emptyset$ in the first iteration.

Next, the execution time of the preempting tasks is calculated as $C_j^{*n_i} = C_{j,\max1}^{n_i} + C_{j,\max2}^{n_i}$. $C_e^n(i)$ is initialized to zero to allow it to grow as the zero-slack instant moves towards the end of the period.

D. Pipeline Zero-Slack Calculation

With the response time equation for critical mode and the tasksets for the normal and critical modes we can then construct the zero-slack instant calculation in the pipeline and solve Eq. (16) once we find the response time in critical mode. Then, we calculate the interference before the zero slack instant (normal mode) with Eq. (17). Next, we calculate the slack in normal mode with Eq. (18) followed by the update to the computation for each of the modes (Eqs. (19), (20)). To update Π_i^c we first update σ_i^z with Eq. (21) and then we update the critical stage set with $\Pi_i^c = \{\pi_j | \sigma_i^z \leq j \leq N\}$ and the nominal stage set with $\Pi_i^n = \{\pi_j | 1 \leq j < \sigma_i^z\}$. We repeated until the zero-slack instants converge. The zero-slack calculation is applied to each task in decreasing order of criticality to guarantee convergence.

$$Z_i = D_i - R_i^c \tag{16}$$

$$I_i = \sum_{j \in \Gamma_i^n} \left\lceil \frac{Z_i}{T_j} \right\rceil C_j^{*n_i} \tag{17}$$

$$S_i^n = \max(0, Z_i - I_i - C_e^n(i)) \tag{18}$$

$$C_e^c(i) = \max(0, C_e^c(i) - S_i^n) \tag{19}$$

$$C_e^n(i) = \min(C_i^o, C_e^n(i) + S_i^n) \tag{20}$$

$$\sigma_i^z = \min_{1 \leq j \leq N} \{j | \sum_{s=1}^{j} C_{i,s}^o > C_e^n(i)\} \tag{21}$$

The run-time enforcement of pipeline ZSRM works as described in the end of Section III but where C_i is defined as in Section II.

VI. Conclusions

We presented the first MC scheduling scheme for processing pipelines which we call ZSRM pipeline scheduling. This is based on two previous results (i) schedulability analysis and zero-slack configuration for ZSRM scheduling and (ii) an efficient method for analyzing delays in pipelines which avoids inefficiencies that previous methods suffered from when analyzing long pipelines.

References

[1] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 2014.

[2] S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *RTSS*, 2011.

[3] A. Burns and R. Davis. Mixed criticality systems - a review. http://www-users.cs.york.ac.uk/burns/review.pdf.

[4] A. Burns, J. Harbin, and L.S. Indrusiak. A wormhole NoC protocol for mixed criticality systems. In *RTSS*, 2014.

[5] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *RTSS*, 2009.

[6] H. Huang-Ming, C. Gill, and C. Lu. Implementation and evaluation of mixed-criticality scheduling approaches for periodic tasks. In *RTAS*, 2012.

[7] P. Jayachandran and T. F. Abdelzaher. A delay composition theorem for real-time pipelines. In *ECRTS*, 2007.

[8] H. Li and S. Baruah. Outstanding paper award: Global mixed-criticality scheduling on multiprocessors. In *ECRTS*, 2012.

[9] C. Liu and J. H. Anderson. Scheduling suspendable, pipelined tasks with non-preemptive sections in soft real-time multiprocessor systems. In *RTAS*, 2010.

[10] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 1994.