Relaxing the Synchronous Approach for Mixed-Criticality Systems

Eugene Yip, Matthew M Y Kuo, Partha S Roop
Department of ECE, University of Auckland, New Zealand
{eyip002, mkuo005}@aucklanduni.ac.nz, p.roop@auckland.ac.nz

David Broman
UC Berkeley, USA and Linköping University, Sweden
davbr@berkeley.edu

***Abstract*—Synchronous languages are widely used to design safety-critical embedded systems. These languages are based on the synchrony hypothesis, asserting that all tasks must complete instantaneously at each logical time step. This assertion is, however, unsuitable for the design of mixed-criticality systems, where some tasks can tolerate missed deadlines. This paper proposes a novel extension to the synchronous approach for supporting three levels of task criticality: life, mission, and non-critical. We achieve this by relaxing the synchrony hypothesis to allow tasks that can tolerate bounded or unbounded deadline misses. We address the issue of task communication between multi-rate, mixed-criticality tasks, and propose a deterministic lossless communication model. To maximize system utilization, we present a hybrid static and dynamic scheduling approach that executes schedulable tasks during slack time. Extensive benchmarking shows that our approach can schedule up to 15% more task sets and achieve an average of 5.38% better system utilization than the Early-Release EDF (ER-EDF) approach. Tasks are scheduled fairer under our approach and achieve consistently higher execution frequencies, but require more preemptions.**

I. Introduction

Safety-critical embedded systems [1] are continuously superseded by more complex designs that must be certified against stringent safety-standards [2], [3]. Moreover, safety-measures must be incorporated to isolate and mitigate errors or faults that can develop at runtime [1]. *Mixed-criticality* systems emerge when tasks with diverse levels of importance or *criticality* [1], [4] are integrated together [5]. With respect to timing, high criticality tasks are hard real-time because they cannot miss any deadlines, whereas lower criticality tasks are soft real-time because they can tolerate missed deadlines. Determining efficient schedules for mixed-criticality tasks has recently become an important research topic. Since the introduction and formalization of the problem [4], there have been several studies on uni-processor [6]–[9] and multi-processor [10]–[12] systems.

Safety-critical systems are typically periodic hard real-time systems and can be modeled directly with synchronous languages [13], [14]. Synchronous languages are based on sound mathematical semantics, which facilitates system verification by formal methods [13] and generation of correct-by-construction implementations [15], [16]. The core of any synchronous language is the *synchrony hypothesis* [13], asserting that a system *reacts* instantaneously to all its environmental inputs. A reaction consists of reading inputs from the environment, computing the system's next state, and emitting outputs to the environment. All concurrent tasks (or threads, in the parlance of synchronous languages) in the system react in lock-step with respect to a *logical clock*. An implementation satisfies the synchrony hypothesis if the *Worst-Case Execution Time* (WCET) [17] of all reactions do not exceed the system's specified period [18]. Unfortunately, this requirement precludes the use of tasks with soft real-time deadlines in a synchronous program. Thus, several significant limitations arise in the modeling of mixed-criticality systems with synchronous languages:

- Only tasks with statically computable WCETs, necessary to validate the synchrony hypothesis, can be included in the system. Tasks with unknown WCETs (e.g., too complex to analyze) are excluded.
- The synchrony hypothesis requires all tasks to be hard real-time. Thus, there is no advantage in prioritizing the execution of high criticality tasks over lower criticality tasks.
- To satisfy the synchrony hypothesis, the implementation must provide enough resources to ensure that the system's estimated WCET does not exceed the specified period. If the assumptions used for the WCET estimation are unlikely to occur, then the provisioned resources will be under-utilized at runtime.

The work of Baruah [8] is the first attempt at generating efficient (static) schedules for multi-rate, mixed-criticality, synchronous programs on uni-processor systems. Baruah applies the result from [19] to the synchronous setting. The assumption is made that the WCET estimate of a task becomes more pessimistic as its level of timing assurance increases. Tasks are statically scheduled to execute for up to their WCET as estimated at the lowest level of timing assurance. When any high criticality task exceeds its scheduled time, all the low criticality tasks are immediately discarded. This frees up the system to meet the deadlines of just the high criticality tasks, but causes low criticality tasks to execute sporadically. This is undesirable for control-related tasks as sporadic delays are introduced and can cause system instability [14]. To mitigate this problem, Su et al. [9] propose an *Elastic Mixed-Criticality*

This work was supported in part by the RIPPES INRIA International Lab, the University of Auckland PReSS account, the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by IBM and United Technologies), the Swedish Research Council (#623-2011-955), and the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley (supported by the National Science Foundation, NSF awards #0720882 (CSR-EHS: PRET), #1035672 (CPS: Medium: Timing Centric Software), and #0931843 (ActionWebs), the Naval Research Laboratory (NRL #N0013-12-1-G015), and the following companies: Bosch, National Instruments, and Toyota).

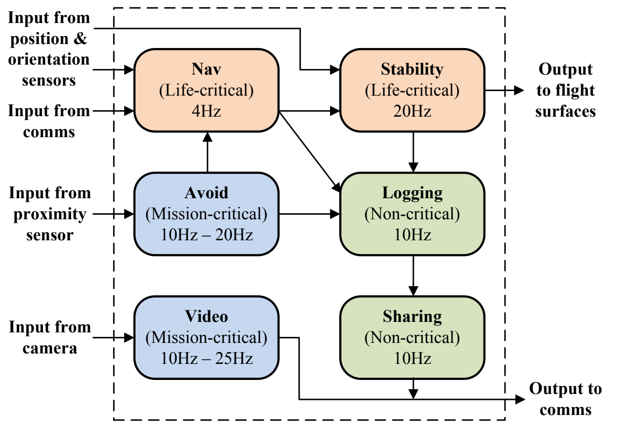
task model and an *Early-Release EDF* scheduling algorithm for uni-processors to guarantee minimum *service levels* for low criticality tasks. Every low criticality task has a *maximum* period and a set of shorter *desired* periods. At runtime, the low criticality tasks are released according to their maximum period. If there is enough slack in the system, then the tasks are released at one of their earliest desired periods. This approach was later extended to multi-processors [11].

Contributions. We propose a novel extension to multi-rate synchronous languages that allows the modeling of mixed-criticality systems. The synchrony hypothesis is relaxed to allow the specification of tasks with soft real-time deadlines. Instead of requiring tasks to execute at constant frequencies, low criticality tasks can execute within a specified range of frequencies. We also propose a multi-processor scheduling method for the proposed multi-rate, mixed criticality, synchronous task model. Tasks are statically scheduled on processors such that their *minimum* execution frequencies are met. Slack in the static schedule is consumed by increasing the scheduled time of the low criticality tasks, thereby increasing their execution frequency. Additional slack can develop at runtime when tasks execute for less than their WCET. The additional slack is used to reschedule low criticality tasks to further increase their execution frequency. We claim that this is the first multi-rate, mixed criticality framework for synchronous languages on multi-processors. In summary, our main contributions are:

- Relaxing the synchrony hypothesis to capture the timing requirements of mixed-criticality tasks. We offer frequency-based parameters for specifying different levels of criticality. The parameters are tightly coupled to embedded multi-rate applications by relating task frequency bounds to task criticality. (Section II).
- The relaxation of the synchrony hypothesis to support mixed-criticality violates the synchronous model of communication. To address this, we use a simple *lossless buffering* approach with bounded queue sizes. In contrast to closely related formalisms, such as *Synchronous Data Flow* [20], tasks in our approach can produce dynamically varying number data items, but only within statically known bounds. (Section III).
- We devise a multi-processor (static and dynamic) scheduling method that tries to maximize system utilization by distributing slack time proportionally across all tasks. (Section IV).
- We evaluate our proposed scheduling approach extensively against the ER-EDF approach [11]. Benchmarking results show that our proposed approach can schedule up to 15% more task sets and achieve consistently higher system utilization (up to 98.5%) than ER-EDF. Moreover, tasks achieve higher execution frequencies and share the slack more fairly than ER-EDF. (Section V).

II. Multi-Rate, Mixed-Criticality, Synchronous Task Model

We contend that the proposed task model is applicable to a wide range of cyber-physical systems, such as *Unmanned Aerial Vehicles* (UAVs) [21], biomedical devices [22], and automotive systems [23]. As a motivating example, we describe the design of a UAV inspired by the Paparazzi project [24]. A UAV is a remotely controlled aerial vehicle commonly used in surveillance operations. Figure 1 illustrates the functionality of a UAV as a block diagram of tasks. The *Nav* task localizes the UAV using onboard sensors, updates the flight path, and sends the desired position to the *Stability* task. The *Stability* task controls the flight surfaces to ensure stable flight to the desired position. Extremely low jitter is required for stable flight. The UAV has the following useful but less critical features. The *Video* task streams a video of the UAV's flight from an onboard camera to allow users to fly the UAV from the UAV's point of view. The higher the frame rate, the better the flying experience. The *Avoid* task uses onboard sensors to detect obstacles around the UAV and sends collision avoidance data to the *Nav* task. Thus, the more frequent obstacles are checked for, the faster the UAV can safely travel at. Less critical features that a UAV can have include a data logging facility to log important flight events (*Logging*) and to share obstacle and localization data with nearby UAVs (*Sharing*). Because the UAV combines tasks of different criticalities, it is an excellent example of a mixed-criticality system.



Input from position & orientation sensors
Input from comms
Input from proximity sensor
Input from camera
Nav (Life-critical) 4Hz
Stability (Life-critical) 20Hz
Avoid (Mission-critical) 10Hz – 20Hz
Logging (Non-critical) 10Hz
Video (Mission-critical) 10Hz – 25Hz
Sharing (Non-critical) 10Hz
Output to flight surfaces
Output to comms

Fig. 1. Functional block diagram of a UAV with criticality levels and frequency bounds.

In our task model, a synchronous program is a set of tasks $\tau \in \Gamma$ that are released together when the program starts executing. Without loss of generality, we assume tasks that do not create new tasks at runtime. The programmer assigns a criticality ζ_τ to each task as either $life$, $mission$, or $non\text{-}critical$. That is, $\zeta_\tau \in \{life,\ mission,\ non\text{-}critical\}$. Life critical tasks are released periodically and adhere to the synchrony hypothesis. Thus, life critical tasks must complete their computation before their next release time (a hard real-time deadline). For example, if a life critical task with period p_τ is released at time r_τ, then its deadline (and next release) is at time $r_\tau + p_\tau$. We relax the synchrony hypothesis for mission critical tasks in that *bounded* deadline misses are tolerated. For example, if a mission critical task misses its deadline of time $r_\tau + p_\tau^{min}$, then it cannot miss a relaxed deadline of time $r_\tau + p_\tau^{max}$, where $p_\tau^{min} < p_\tau^{max}$. We relax the synchrony hypothesis completely for non-critical tasks by removing the notion of deadlines. For example, if a non-critical task with period p is released at time r_τ and completes its computation at time $r_\tau + q$, then its next release is $Max\{r_\tau + p_\tau,\ r_\tau + q\}$.

To better understand the execution rates of the tasks, their periods can be converted into frequencies using the equation $f = 1/p$. The minimum and maximum release frequencies (f_τ^{min} and f_τ^{max}) depend on the assigned criticality ζ_τ. For life critical tasks, $f_\tau^{min} = f_\tau^{max}$. For mission critical tasks, $f_\tau^{min} < f_\tau^{max}$. For non-critical tasks, only f_τ^{max} is specified and is treated as a goal frequency. All specified frequencies must be greater than 0. Any implementation must ensure that all life critical tasks execute at their f_τ^{max} and that mission critical tasks execute within their f_τ^{min} and f_τ^{max}. For non-critical tasks, while the implementation tries to meet their f_τ^{max} (goal frequency), no guarantees are provided, i.e., these tasks may execute at a lower frequency compared to their goal frequency. WCET analysis [17] can be used to estimate an upper bound on a task's maximal computation time, c_τ. For non-critical tasks, c_τ is left unspecified. In Appendix A-A, we discuss how more levels of criticality can be supported by the task model.

We now motivate the criticality levels assigned to the tasks in Figure 1. When assigning a criticality, we assume that life critical tasks must meet specific hard deadlines, whereas mission critical tasks can improve their quality of service by executing more frequently. The *Nav* and *Stability* tasks are life critical because they are required for safe operation. The *Video* and *Avoid* tasks are mission critical because high frequencies are desirable for the video frame rate and checking of obstacles, but their lowest tolerable frequency is $10Hz$. The *Logging* and *Sharing* tasks are non-critical because they are not necessary for the correct operation of the UAV.

III. Communication

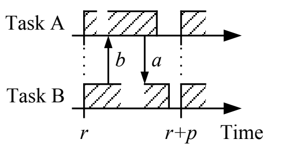
In this section, we define the communication model for the proposed multi-rate, mixed-criticality, synchronous task model.

A. Synchronous Communication

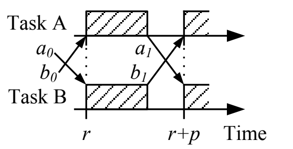
The synchrony hypothesis requires all communication to complete in the same period they were started in, i.e., before the tasks are released again. This is shown in Figure 2a for a single-rate synchronous program with two tasks `A` and `B` that communicate using the variables `a` and `b`. The tasks must be scheduled such that data is always sent before it is received. This data-dependency limits task schedulability and the ability to execute tasks in parallel. A way to relax data-dependencies is to delay the receiving of data by one period. When tasks are released, they receive the data sent from the previous period. This delayed semantics is illustrated in Figure 2b and is supported by some synchronous languages (e.g., the `pre` operator of *Esterel* [25] and *Lustre* [26], *Prelude*'s `fby` operator [27], *ForeC*'s `shared` variable semantics [28], and *SL*'s delayed semantics [29]). In Figure 2b, variables `a` and `b` are assumed to be declared with an initial value. At time r, both tasks receive the initial values a_0 and b_0. At time $r + p$, both tasks receive the values a_1 and b_1 sent from the previous period. For the remainder of the paper, we assume that all task communications are delayed.

B. Multi-Rate Communication

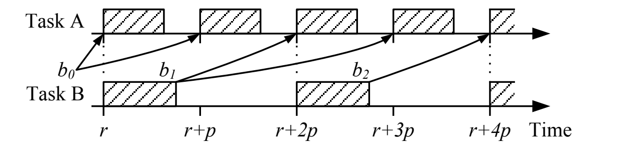
For multi-rate synchronous programs, tasks may need to communicate with others at different frequencies. A common approach is to *oversample* data sent from lower frequency tasks and to *undersample* data sent from higher frequency tasks [27], [30], [31]. Figure 2c shows oversampling, where task `A` receives the last value sent from task `B` multiple times, $\{b_0, b_0, b_1, b_1, b_2, \dots\}$. Figure 2d shows undersampling, where task `B` only receives the last value sent from task `A`, $\{a_0, a_2, a_4, \dots\}$. The main disadvantage with undersampling is the loss of data (e.g., $\{a_1, a_3, \dots\}$) which may be unacceptable when, e.g., commands are sent between tasks.



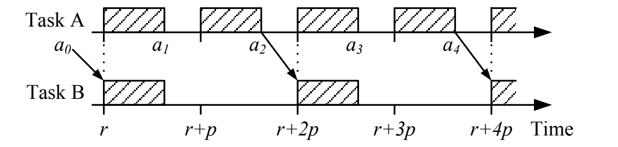
(a) Instantaneous communication between tasks A and B.



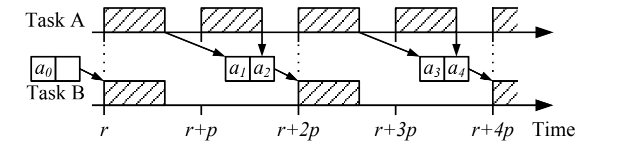
(b) Delayed communication between tasks A and B.



(c) Oversampling: A lower frequency task (task B) communicating to a higher frequency task (task A).



(d) Undersampling: A higher frequency task (task A) communicating to a lower frequency task (task B).



(e) Lossless buffering: A higher frequency task (task A) communicating to a lower frequency task (task B).

Fig. 2. Task communication models.

To overcome the problem with undersampling, we propose a simple approach of *lossless buffering*. The idea of lossless buffers for multi-rate systems is not new. For instance, in *Synchronous Data Flow* (SDF) [20], bounded lossless buffers can be achieved by restricting tasks (typically called nodes or actors) to consume and produce fixed, statically known, number of data items. By contrast, in our methodology, mission-critical tasks can produce dynamically varying number data items; only the upper and lower bounds of rates are statically known.

The suggested approach of lossless buffers is simple: all data sent from a higher frequency task to a lower frequency task is buffered in a *First In, First Out* (FIFO) buffer. When the lower frequency task is released, it consumes all the data in the buffer and then clears the buffer. This is shown in Figure 2e, where the buffer begins with the initial value a_0 for variable

a. At time r, the buffer is consumed and cleared. Between the time interval $[r, r+2p]$, the data sent from task A is buffered. At time $r+2p$, task B is released so it consumes and clears the buffer. When task B executes, it has the values a_1 and a_2 for variable a. The programmer is free to use the values in any way they like. For example, if data loss is undesirable, then the task can process the "backlog" of values. As another example, a task may only use the latest value for its computation. In this case, the communication behaves like the undersampled method shown in Figure 2d. We observe two key properties of the proposed lossless buffering approach:

Observation 1. *The maximum buffer size needed for lossless communication between two life or mission critical tasks is bounded: Let τ be a life or mission critical task that communicates to another life or mission critical task τ'. Let τ be the higher frequency task. The maximum buffer size needed for lossless communication is equal to the maximum number of times that τ can send data between each release of τ'. This occurs when τ is released at its maximum frequency (f_τ^{max}) and when τ' is released at its minimum frequency ($f_{\tau'}^{min}$). Thus, the maximum buffer size can be calculated as the ratio of both frequencies:*

$$\textit{Maximum buffer size} = \left\lceil \frac{f_\tau^{max}}{f_{\tau'}^{min}} \right\rceil \tag{1}$$

Observation 2. *By using FIFO buffers, the (untimed) sequence of data sent from the higher frequency task is always received in the same sequence by the lower frequency task.*

C. Mixed-Criticality Communication Model

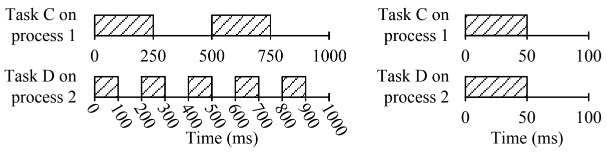
We now define the communication model for mixed-criticality tasks. All communication between tasks of the same frequency are simply delayed (Figure 2b). All communication from lower to higher frequency tasks are oversampled (Figure 2c). All communication from higher frequency tasks to lower frequency life and mission critical tasks use lossless buffering (Figure 2e). All communication from higher frequency tasks to lower frequency non-critical tasks are undersampled (Figure 2d). Lossless buffering is not used because the maximum time between each release of a non-critical task is unbounded. Hence, an infinite number of data may have to be buffered. This communication model can be implemented as high-level library functions or as a new data type in an existing synchronous language.

IV. Task Scheduling

In this section, we describe a scheduling algorithm for the proposed multi-rate, mixed-criticality, synchronous task model. Each task $\tau \in \Gamma$ is scheduled on a processor $n \in N$. A processor can execute instructions from multiple tasks by *preemptively context-switching* between the tasks at scheduled times. Note that the scheduling algorithm is applicable to other types of parallel architectures, such as multi-core and multi-threaded processors [32], [33].

A. Schedulability

For a given set of tasks and available processors, we need to determine if the tasks are schedulable. First, similar to the definitions presented by the ER-EDF approach [11], we use the following notations:



(a) Hyper period approach. (b) Base period approach.

Fig. 3. Static schedules for two life critical tasks on their processor.

$$u_\tau^{min} = c_\tau \cdot f_\tau^{min} \tag{2}$$
$$u_\tau^{max} = c_\tau \cdot f_\tau^{max} \tag{3}$$
$$U_\Gamma(\textit{life}) = \sum_{\tau \in \Gamma,\ \zeta_\tau = \textit{life}} (u_\tau^{min}) \tag{4}$$
$$U_\Gamma^{min}(\textit{mission}) = \sum_{\tau \in \Gamma,\ \zeta_\tau = \textit{mission}} (u_\tau^{min}) \tag{5}$$

where, u_τ^{min} and u_τ^{max} is task τ's minimum and maximum *utilization*, respectively. A task's utilization is the ratio of its computation time c_τ compared to its period. Note that $u_\tau^{min} = u_\tau^{max}$ for life critical tasks only. $U_\Gamma(\textit{life})$ is the total u_τ^{min} of all life critical tasks in Γ. $U_\Gamma^{min}(\textit{mission})$ is the total u_τ^{min} of all mission critical tasks in Γ. Let $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \cdots \cup \Gamma_x$, where Γ_n is the set of tasks allocated to processor n.

Definition 1. (Schedulability) *A task set Γ is schedulable over a set of (homogeneous) processors N if:*

$$\forall n \in N: \ U_{\Gamma_n}(\textit{life}) + U_{\Gamma_n}^{min}(\textit{mission}) \leq 1 \tag{6}$$

The schedulability condition represented by equation (6) only considers life and mission critical tasks. No guarantees are made for the execution of non-critical tasks.

B. Scheduling Approaches

Traditional multi-rate, synchronous tasks can be scheduled dynamically [14], [16], [27] or statically [8], [14]. To guarantee schedulability, the tasks are scheduled for their maximum computation time c_τ. In dynamic scheduling, the multi-rate tasks are scheduled using the *Rate-Monotonic* or *Earliest Deadline First* (EDF) algorithms. In static scheduling, each task is statically assigned to a processor and allocated a fixed amount of time to execute. A static schedule is created such that it can be repeated indefinitely to meet the timing constraints of all tasks. The length of the schedule (makespan) is usually the *hyper* period or *base* period of all the task periods. The hyper and base periods are computed as the *Least Common Multiple* (LCM) and *Greatest Common Divisor* (GCD) of all task periods, respectively. A limitation is that the task periods must be rational numbers in order to compute the LCM or GCD. Figure 3 compares the hyper and base period approaches for scheduling the same multi-rate tasks, C and D, on separate processors. Tasks C and D have a release frequency of $2Hz$ and $5Hz$, and a maximum computation time of 250 *milliseconds* (ms) and $100ms$, respectively. The hyper period is $LCM\{500ms, 200ms\} = 1{,}000ms$ and the base period is $GCD\{500ms, 200ms\} = 100ms$.

C. Hyper and Base Period Approaches

The hyper period approach [8], [14] constructs a schedule for the shortest time interval in which all tasks can meet their timing constraints. In Figure 3a, it can be verified that tasks C and D can continue to execute at $2Hz$ and $5Hz$, respectively, when the schedule is repeated indefinitely. However, very long schedules are generated when the hyper period is much larger than the task periods. Longer schedules typically require more memory to store. The base period approach [14] creates shorter schedules by allocating a portion of the task's computation time over each base period. Thus, the execution of a task is split over an integral number of base periods. The release of a task is decided at runtime at the start of each base period. A task is released if it has completed its computation and its minimum period p_τ^{min} has elapsed since its last release. Note that all tasks get released in the program's first base period. In Figure 3b, it can be verified that tasks C and D can complete their computations in 5 and 2 base periods, respectively, which equates to $2Hz$ and $5Hz$. However, preemptive scheduling is required to split the execution of each task and the number of preemptions increases with the number of tasks.

Although tasks are scheduled for their maximum computation time c_τ, their actual computation time may be shorter. Thus, slack can develop at runtime and lead to an under-utilized system. However, the slack can be used to help mission and non-critical tasks execute at a faster frequency and improve system utilization. This insight was used in the Early-Release EDF approach [9], [11] where each (low criticality) task has a set of recurring times specifying when it can be released earlier than usual. The elastic approach could be applied to the hyper period approach but determining the early release times is non-trivial. This is because slack can develop at anytime between the statically scheduled task release times.

We take a simpler approach to reclaiming slack. Specifically, we use the base period approach to statically schedule all the life and mission critical tasks. During runtime, if a task completes before its allocated time expires, then the next task in the static schedule is executed. Thus, slack will always accumulate at the end of each base period and the slack can be used to execute the non-critical tasks. In the following, we describe the proposed scheduling and some heuristics to further improve system utilization.

D. Static Scheduling Algorithm

The algorithm for obtaining a static schedule is given in Figure 4. Lines 1-3 computes the base period p_b of the tasks. For the UAV example, $p_b = GCD\{\frac{1}{4}, \frac{1}{10}, \frac{1}{20}, \frac{1}{25}\} = \frac{1}{100} = 10ms$. For each life and mission critical task, we need to determine what portion of their computation time can be allocated in each base period (lines 5-9). The minimum and maximum portion, t_τ^{min} and t_τ^{max} respectively, depends on the task's minimum and maximum utilization. Table I exemplifies the calculation of t_τ^{min} and t_τ^{max} for the life and mission critical tasks of the UAV example. The function GENSCHEDULE (described in Section IV-E) is then used to find a feasible task-to-processor allocation (line 10). GENSCHEDULE returns a static schedule $s_n \in S$ for each processor n. Each static schedule s_n is a queue of allocated tasks and their execution times. The static scheduling algorithm

```
Input: $\Gamma$, $N$                                  ▷ Set of tasks, and set of processors.
Output: $S$                                 ▷ Set of static schedules for all processors.

 1: $P^{min} := \{1/f_\tau^{max} \mid \tau \in \Gamma,\ \zeta_\tau \neq \textit{non-critical}\}$      ▷ Min periods.
 2: $P^{max} := \{1/f_\tau^{min} \mid \tau \in \Gamma\}$                    ▷ Max periods.
 3: $p_b := GCD(P^{min} \cup P^{max})$                          ▷ Base period.
 4: $T := \emptyset$            ▷ Set of min and max allocation times for each task.
 5: for all $\tau \in \Gamma,\ \zeta_\tau \neq \textit{non-critical}$ do    ▷ Life & mission critical tasks.
 6:     $t_\tau^{min} := p_b \cdot u_\tau^{min}$       ▷ Min time needed during each base period.
 7:     $t_\tau^{max} := p_b \cdot u_\tau^{max}$      ▷ Max time needed during each base period.
 8:     $T := T \cup \{t_\tau^{min},\ t_\tau^{max}\}$
 9: end for
10: $S :=$ GENSCHEDULE$(p_b,\ N,\ T)$                ▷ Generate static schedules.
11: return $S$
```

Fig. 4. Obtaining a static schedule for the base period approach.

can be extended to heterogeneous processors by considering each task's computation time on each heterogeneous processor. This would require $n \in N$ to be an additional parameter to c_τ and its derived values, such as u_τ^{min}, u_τ^{max}, t_τ^{min}, and t_τ^{max}.

E. GENSCHEDULE

We use *Integer Linear Programming* (ILP) to find a feasible task-to-processor allocation and static schedule for each processor. The ILP formulation requires the following inputs: p_b, the base period of the tasks; $n \in N$, the set of available processors; and $t_\tau^{min}, t_\tau^{max} \in T$, the set of minimum and maximum execution times to allocate. ILP requires all constraints to use integral coefficients. Thus, p_b and all times in T need to be integers. The objective function for the ILP problem is to maximize the *utilization* U of all the processors:

$$\text{Maximize}:\ U = \sum_{n \in N} u_n \tag{7}$$

where, u_n is the utilization of processor n for one base period. Here, we define utilization as the time processor n spends executing its tasks:

$$u_n = \beta + \sum_{\tau \in \Gamma} \left(a_\tau^n \cdot \left(t_\tau^{min} + \alpha \right) \right) \tag{8}$$

$$u_n \leq p_b \tag{9}$$

where, α is the cost of preempting a task and β is the cost of resolving delayed communication in each base period. We can estimate α by analyzing the WCET needed to preempt a task and to schedule the next task. Similarly, β can be estimated conservatively by analyzing the WCET needed to resolve the delayed communication of all tasks[1]. The a_τ^n is a Boolean indicating whether or not task τ is allocated to processor n:

$$a_\tau^n = \begin{cases} 1 & \text{if task } \tau \text{ is allocated to process } n \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

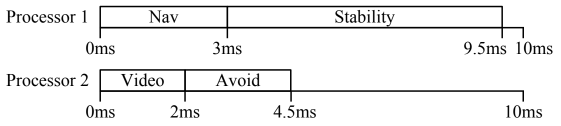
We constrain the allocation of a task to exactly one processor:

$$\forall \tau \in \Gamma:\ \sum_{n \in N} a_\tau^n = 1 \tag{11}$$

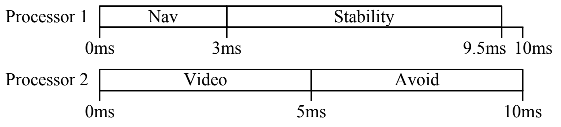
[1]Note that a task's delayed communication only occurs when it finishes its computation and that tasks can finish in different base periods. Thus, the time spent in each base period to resolve delayed communication can vary. Since all base periods will use the same static schedule, β needs to be a safe upper bound on the time spent resolving delayed communication.

TABLE I. CALCULATED PARAMETERS FOR THE LIFE AND MISSION CRITICAL TASKS OF THE UAV EXAMPLE. $p_b = 10ms$.

	Given parameters				Calculated parameters			
τ	ζ_τ	c_τ	f_τ^{min}	f_τ^{max}	u_τ^{min}	u_τ^{max}	t_τ^{min}	t_τ^{max}
Nav	*life*	$75ms$	$4Hz$	$4Hz$	0.3	0.3	$3ms$	$3ms$
Stability	*life*	$32.5ms$	$20Hz$	$20Hz$	0.65	0.65	$6.5ms$	$6.5ms$
Video	*mission*	$20ms$	$10Hz$	$25Hz$	0.2	0.5	$2ms$	$5ms$
Avoid	*mission*	$25ms$	$10Hz$	$20Hz$	0.25	0.5	$2.5ms$	$5ms$

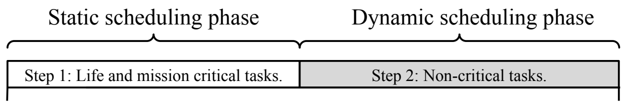


(a) Minimum execution times that can be allocated to the tasks. The overall system utilization is 70%.



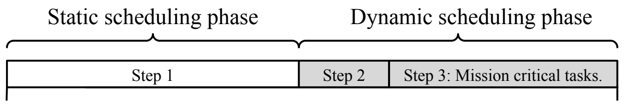
(b) Maximum execution times that can be allocated to the tasks. The overall system utilization is 97.5%.

Fig. 5. Static schedules for the UAV example.

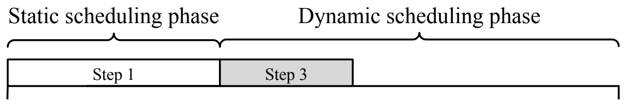


Static scheduling phase
Dynamic scheduling phase
Step 1: Life and mission critical tasks.
Step 2: Non-critical tasks.

(a) Base period for scenario 1.



(b) Base period for scenario 2.



(c) Base period for scenario 3.

Fig. 6. Runtime traces of three different base periods to show the scheduling decisions that a processor can make.

If equation (9) is satisfied for all processors, then the tasks are schedulable (Definition 1). Once a solution is found for the objective function, a static schedule s_n can be constructed for each processor n as a queue of allocated tasks and execution times. Let s_n be represented as a partial function defined only for tasks τ with $a_\tau^n = 1$:

$$\forall \tau \in \Gamma, a_\tau^n = 1 : \; s_n(\tau) = t_\tau^{min}$$

Since tasks communicate using delayed semantics (Section III-C), the task ordering in each s_n can be arbitrary. Let S contain all the static schedules such that $\forall n \in N, s_n \in S$. Possible static schedules for the UAV example over two processors are depicted in Figure 5a. The scheduling of non-critical tasks (e.g., *Logging* and *Sharing*) is considered later in Section IV-F.

Note that only the minimum task execution times (t_τ^{min}) were used in equation (8) to compute processor utilization. If slack exists on a processor, i.e., $u_n < p_b$, then extra execution time can be allocated to the mission critical tasks. This would allow the mission critical tasks to complete their computations earlier and be released more frequently (towards their f_τ^{max}). The maximum extra execution time that a mission critical task can make use of is bounded by $x_\tau^{max} = (t_\tau^{max} - t_\tau^{min})$. In the following, we extend the ILP formulation to utilize the slack on each processor. Let x_τ^n denote the extra execution time that can be allocated to task τ on processor n:

$$0 \le x_\tau^n \le (a_\tau^n \cdot x_\tau^{max}) \tag{12}$$

The a_τ^n variable ensures that task τ only receives extra execution time on its allocated processor. We update equation (8) with the following equation to include the extra execution time:

$$u_n = \beta + \sum_{\tau \in \Gamma} \left(a_\tau^n \cdot \left(t_\tau^{min} + \alpha \right) + x_\tau^n \right) \tag{13}$$

We now add the extra execution times into the static schedules:

$$\forall \tau \in \Gamma, a_\tau^n = 1 : \; s_n(\tau) = t_\tau^{min} + x_\tau^n$$

Based on equations (12) and (13), the slack is allocated arbitrarily. This can be *unfair* if slack is only allocated to a few mission critical tasks. Fairness [34] has many definitions and can be measured by various metrics, depending on the application at hand. Thus, fairness can be decided manually by the programmer or automatically by a chosen metric. As an example, we describe a simple fairness constraint based on proportionate fairness [35] for homogeneous processors. The constraint will need to be reformulated for heterogeneous processors. Using x_τ^{max} as the metric, tasks with larger x_τ^{max} are allocated more slack than tasks with smaller x_τ^{max}. We capture this for any two tasks, τ and τ', with the inequality $(x_\tau^{max}/x_{\tau'}^{max}) \le (x_\tau^n/x_{\tau'}^{n'})$, where $x_\tau^{max} \ge x_{\tau'}^{max}$, and n and n' are the processors that tasks τ and τ', respectively, are allocated to (i.e., a_τ^n = 1 and $a_{\tau'}^{n'}$ = 1). Rearranging to remove the divisions, we have the following constraint:

$$\forall n, n' \in N, \forall \tau, \tau' \in \Gamma, \; x_\tau^{max} \ge x_{\tau'}^{max}, a_\tau^n = 1, a_{\tau'}^{n'} = 1 : \\ x_{\tau'}^{n'} \cdot x_\tau^{max} \le x_\tau^n \cdot x_{\tau'}^{max} \tag{14}$$

Since equation (14) is generated for pairs of tasks, it will not scale for large task sets. Thus, fairness (equation (14)) can be omitted for shorter ILP solving time. For the UAV example, new static schedules with extra execution times are depicted in Figure 5b.

F. Runtime Scheduling

Figure 6a is a runtime trace of the steps (decisions) that a processor takes to schedule tasks in each base period. Each step is drawn as a block and labeled in chronological order. The first step is the static scheduling phase where life and mission critical tasks are executed for up to their statically allocated times (computed in Section IV-E). Note that a processor can finish its static scheduling phase earlier in some base periods if its tasks complete their computation without using all of their allocated time. After the processor finishes the static scheduling phase, it will use the rest of the available time in the base period (called slack) for the dynamic scheduling phase. Mission critical tasks are statically scheduled to meet their minimum execution frequencies, but these frequencies can be improved by executing the tasks in the dynamic scheduling phase. Non-critical tasks, however, are not statically scheduled so their only opportunity to execute is in the dynamic scheduling phase. Thus, the processor's second step is to execute non-critical tasks until they complete their computation or until the base period expires. We allow non-critical tasks to be executed on any processor. In some base periods, only a fraction of the slack may be needed to complete the execution of all non-critical tasks. In this case, shown in Figure 6b, the processor's third step is to execute mission critical tasks in the remaining slack. For the dynamic scheduling phase only, we allow mission critical tasks to be executed on any processor. Observe that the duration of each step in static and dynamic scheduling phases depends on the actual execution time of the tasks. As an example, Figure 6c shows the trace of a processor's base period where only mission critical tasks can be executed because all the life and non-critical tasks have already completed their computations in a prior base period.

G. Heuristics

This section describes some heuristics for improving the use of slack in the dynamic scheduling phase. For the second scheduling step, to ensure all non-critical tasks get equal opportunities to execute, non-critical tasks that have received the least amount of cumulative slack are executed first. For the third step, to maintain proportionate progress [35] among the mission critical tasks, the mission critical tasks with the least improvement in execution frequency (compared to their maximum possible improvement) are executed first. A task's improvement in execution frequency is measured by:

$$f_\tau^{improve} = \frac{f_\tau^{avg} - f_\tau^{min}}{f_\tau^{max} - f_\tau^{min}} \tag{15}$$

where, f_τ^{avg} is task τ's average execution frequency at runtime. Thus, tasks with lower $f_\tau^{improve}$ values are executed first. If there are no more non-critical or mission critical tasks for a processor to execute in the dynamic scheduling phase, then any slack that still exists will be forfeited. Rather than forfeit the slack, it can be *shifted* to a future base period by using the slack to execute life critical tasks.

V. Performance Evaluation

We evaluate the performance characteristics of the proposed scheduling approach (Section IV) against the *Early-Release Earliest Deadline First* (**ER-EDF**) approach [11]. ER-EDF only supports *high* and *low* critical tasks, but can be mapped to our life and mission critical tasks. ER-EDF does not support our notion of non-critical tasks. Thus, we refer to ER-EDF's high and low critical tasks as simply life and mission critical tasks. In ER-EDF, all tasks are released on their statically allocated processor. A mission critical task can be released earlier at user-defined times if there is enough slack on a processor. For early releases only, mission critical tasks can migrate to other processors to use their slack. In [11], ER-EDF showed best performance when tasks were (statically) allocated to processors using the *First-Fit Decreasing-Criticality* heuristic. We refer to this as the **ER-EDF-FF** approach. The tasks are first sorted by minimum task utilization (u_τ^{min}, equation (2)) in descending order. Then, using First-Fit, all the life critical tasks are allocated before the mission critical tasks. We also compare against the traditional **EDF** approach.

We follow the simulation-based evaluation approach of ER-EDF by Su *et al.* [11] and describe how we generate our task parameters. A task's maximum utilization u_τ^{max} is generated uniformly between 0.05 and 0.5, or 5% and 50%. The system's base frequency f_b, equal to $1/p_b$, is generated uniformly between $100Hz$ and $1000Hz$. A task's f_τ^{min} and f_τ^{max} values are random divisors of f_b. The generated u_τ^{max}, f_τ^{min}, and f_τ^{max} values are inputs to our static scheduling algorithm (Section IV-D). For ER-EDF, the early release points of a mission critical task are generated, starting at its minimum period ($1/f_\tau^{max}$), in increments of one base period until its maximum period ($1/f_\tau^{min}$) is reached. Only life and mission critical tasks are generated for each task set and the proportion of generated life critical tasks is denoted by $prop(\mathit{life})$. Each task set Γ is generated according to a *normalized system utilization* [11], defined as $\frac{U^{max}}{N}$, where

$$U^{max} = Max\{U_\Gamma(\mathit{life}),\ U_\Gamma^{min}(\mathit{life}) + U_\Gamma^{min}(\mathit{mission})\} \tag{16}$$

and N is the total number of processors in the system. $U_\Gamma(\mathit{life})$ and $U_\Gamma^{min}(\mathit{mission})$ are defined by equations (4) and (5) respectively. $U_\Gamma^{min}(\mathit{life})$ is the total utilization of all life critical tasks if they had shorter computation times and, thus, smaller utilization u_τ^{min}:

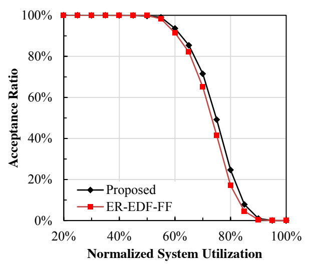
$$U_\Gamma^{min}(\mathit{life}) = \sum_{\tau \in \Gamma, \zeta_\tau = \mathit{life}} u_\tau^{min} \tag{17}$$

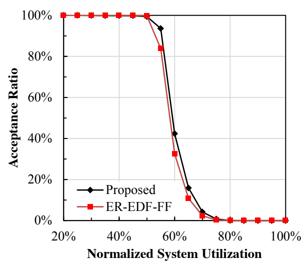
where, u_τ^{min} is generated uniformly between $u_\tau/8$ and u_τ. We do not consider the effects of scheduling overheads in our comparisons. The proposed scheduling approach uses the heuristics detailed in Section IV-G.

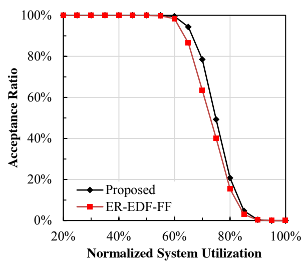
A. Schedulability

We evaluate task schedulability for the proposed and ER-EDF-FF approaches. The performance metric we use is the *acceptance ratio* [11], defined as the proportion of generated task sets that are schedulable. We use `Gurobi`[2] (version 5.6) to solve the ILP constraints of the proposed static scheduling approach. On average, 118.9 constraints are generated per task set and `Gurobi` could take more than a day to find a solution for a large task set. If the fairness constraint (equation (14)) is omitted, then only an average of 66.5 constraints are generated per task set, requiring on average less than a minute to solve. As a compromise, we generate ILP constraints with fairness and allow `Gurobi` one minute to find a (possibly suboptimal) solution before declaring the task set unschedulable. Figures 7a-7c show the acceptance ratio under varying normalized system utilization and $prop(\mathit{life})$. $10,000$ task sets were generated for each data point and attempted to be scheduled on a system with four (homogenous) processors. The acceptance ratio decreases with increasing normalized

[2]http://www.gurobi.com


(a) $prop(life) = 20\%,\ N = 4.$


(b) $prop(life) = 50\%,\ N = 4.$


(c) $prop(life) = 80\%,\ N = 4.$

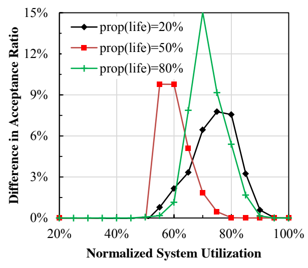
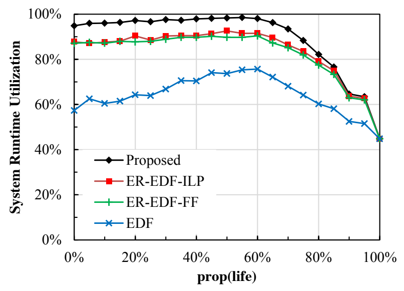
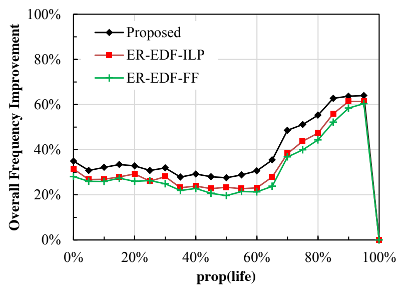

(d) Difference in acceptance ratio (Proposed minus ER-EDF-FF).

Fig. 7. The acceptance ratio as the normalized system utilization and $prop(life)$ are varied.


(a) System runtime utilization, $N = 4$.


(b) Overall frequency improvement, $N = 4$.

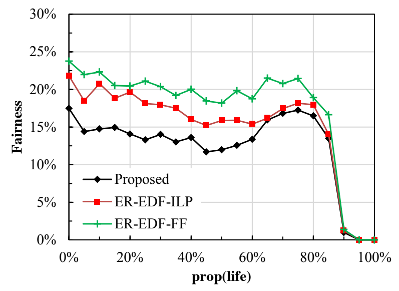

(c) Fairness, $N = 4$.

Fig. 8. The system runtime utilization, overall frequency improvement, and fairness as $prop(life)$ is varied. Normalized system utilization = 50%.

system utilization because equation (16), used to generate task sets, is more optimistic than the test for task schedulability (Definition 1). Hence, as the normalized system utilization increases, more unschedulable task sets are generated. When $prop(life) = 50\%$, the greatest proportion of unschedulable task sets is generated, causing both approaches to reject the most task sets. As expected, the First-Fit heuristic used by ER-EDF-FF rejects more task sets than the proposed approach. This is because the proposed approach uses ILP to find better task-to-processor allocations. Figure 7d highlights this by showing the result of subtracting the acceptance ratio of ER-EDF-FF from the proposed approach. The proposed approach can accept up to 15% more task sets than ER-EDF-FF.

B. *Effects of Varying $prop(life)$*

We evaluate the effects of varying $prop(life)$ on the system runtime utilization and the execution frequency of mission critical tasks. We execute each task set for a minimum of $1,000$ base periods. Each time a task is released, its actual computation time is chosen uniformly between $0.8 \cdot c_\tau$ and c_τ. The normalized system utilization is set to 50% because schedulable task sets are harder to generate at higher utilizations. For a fairer comparison, we allow ER-EDF to use the task-to-processor allocations found by the proposed approach (Section IV-E) and call this the **ER-EDF-ILP** approach.

Figure 8a shows the system runtime utilization under varying $prop(life)$ for 4 (homogeneous) processors. 100 task sets were generated for each data point. Figure 9a in Appendix A gives similar results for 8 processors. System runtime utilization is the average proportion of time that the system spends executing tasks. The EDF result shows the system runtime utilization without releasing tasks early and peaks at around $prop(life) = 50\%$. The system runtime utilization of all approaches converge to the normalized system utilization of 50% with increasing $prop(life)$. This is because fewer mission critical tasks are generated to use the slack. The average (geometric mean) system runtime utilization of the proposed approach is 5.38% better than ER-EDF-ILP. For the proposed approach, slack always accumulates at the end of each base period and is easier for tasks to access than in ER-EDF.

Figure 8b shows the overall improvement in execution frequency for all mission critical tasks under varying $prop(life)$ for 4 processors. Figure 9b in Appendix A gives similar results for 8 processors. The overall improvement is defined as:

$$f^{improve}_{mission} = \frac{\sum_{\tau\in\Gamma,\zeta_\tau=mission}(f^{avg}_\tau - f^{min}_\tau)}{\sum_{\tau\in\Gamma,\zeta_\tau=mission}(f^{max}_\tau - f^{min}_\tau)} \quad (18)$$

where, f^{avg}_τ is the average execution frequency recorded for task τ. A higher value of $f^{improve}_{mission}$ means a better overall improvement. The results show that $f^{improve}_{mission}$ increases until $prop(life) = 100\%$, when no mission critical tasks are generated. The value of $f^{improve}_{mission}$ decreases slightly at around $prop(life) = 50\%$ because the system runtime utilization, without releasing tasks early, is already near its peak. This was illustrated by the EDF result in Figure 8a. The average (geometric mean) improvement in execution frequency for the proposed approach is 5.91% better than ER-EDF-ILP.

Figure 8c shows how *fairly* the execution frequencies of the mission critical tasks are improved by as $prop(life)$ is varied for 4 processors. Figure 9c in Appendix A gives similar results for 8 processors. We define fairness as how well all mission critical tasks can improve their execution frequency by the same proportion:

$$fairness = \frac{\sum_{\tau \in \Gamma, \zeta_\tau = mission} \left| f^{avg\_improve} - f_\tau^{improve} \right|}{Number\ of\ mission\ critical\ tasks} \tag{19}$$

where, $f_\tau^{improve}$ was already defined by equation (15) in Section IV-G, and

$$f^{avg\_improve} = \frac{\sum_{\tau \in \Gamma, \zeta_\tau = mission} (f_\tau^{improve})}{Number\ of\ mission\ critical\ tasks} \tag{20}$$

is the average overall frequency improvement of all mission critical tasks. A completely fair improvement in execution frequencies results in $fairness = 0\%$, while a completely unfair improvement results in $fairness = 50\%$. As expected the proposed approach is fairer than the other approaches when $prop(life)$ is between 0% to 60%. Fairness can be improved by selecting better fairness constraints and heuristics. All approaches become unfair when $prop(life)$ is between 60% and 90%. This is because some task sets have fewer mission critical tasks than available processors and the fairness heuristics (Section IV-G) are less effective on such task sets. All approaches are very fair when $prop(life)$ is around 100% because less than two mission critical tasks are generated in each task set.

C. Summary and Discussions

The benchmarking showed that our proposed static scheduling algorithm accepts a greater range of task sets than ER-EDF-FF. This is because ILP is used to find better task-to-processor allocations than the first-fit heuristic used by ER-EDF-FF. Our proposed scheduling approach was shown to be much fairer than ER-EDF while achieving much higher system utilization and execution frequencies. We also investigated the impact that non-critical tasks have on the performance of mission critical tasks and give results in Appendix A-B. Although `Gurobi` did not scale well in finding globally optimal solutions, in Appendix A-C we show that `Gurobi` can find locally optimal solutions quickly, much like a heuristic. Results concerning the response time of tasks and the number of task preemptions can be found in Appendix A-D.

VI. Related Work

Mixed-critical systems can be modeled by periodic task sets scheduled by a *Real-Time Operating System* (RTOS) [36]. In one of the first works on mixed-criticality scheduling [4], Vestal assumes that WCET estimates for tasks become more pessimistic as its criticality (level of timing assurance) increases. Static [8], [19] and dynamic [4], [6], [7], [10], [12], [37], [38] scheduling strategies have been proposed to ensure that the highest criticality tasks always meet their deadlines. Many of the scheduling strategies are based on EDF because it provides optimal scheduling on preemptive uni-processors. Tasks are scheduled to execute for up to their *low criticality* WCET, estimated at a low level of timing assurance. When a high criticality task executes beyond its low criticality WCET, all the low criticality tasks are immediately discarded. This frees up the system to meet the deadlines of just the high criticality tasks, but causes lower criticality tasks to execute sporadically. Thus, a single higher criticality task can prevent all the lower criticality tasks from executing.

To alleviate the sporadic execution of low criticality tasks, Su *et al.* [9], [11] propose an *Early-Release EDF* (ER-EDF) scheduling algorithm that guarantees a minimum execution time for all low criticality tasks. The scheduling modifies the task period of low criticality tasks according to the processor's runtime utilization. Each low critical task has a set of (recurring) early-release times specifying when they can be released earlier than usual when enough slack can be reclaimed. The results show improved task schedulability and processor utilization, but no method for automatically generating early-release times that make more efficient use of slack was proposed. In comparison, our proposed scheduling approach does not need early release times to be specified, but achieves superior runtime utilization, task frequency improvements, and fairness. Moreover, our task model supports more levels of task criticalities than ER-EDF: Life, mission and non-critical.

Closely related is the *Zero-Slack QoS-based Resource Allocation Model* (ZS-QRAM [37]) that aims to maximize system utilization under all operating conditions. ZS-QRAM subsumes earlier work on *Zero-Slack* (ZS [6], [38], [39]) by supporting the scheduling of criticality and utility-based tasks. The scheduling of criticality-based tasks takes precedence over utility-based tasks to ensure critical tasks always meet their deadlines. Utility-based tasks use *marginal utility* to quantify the benefits of allocating extra resources to each task. The initial allocation of resources to each task is performed iteratively, guided by marginal utility, until all resources are exhausted. At runtime, if a utility-based task executes for longer than its *nominal* execution time, then tasks of lower utility are degraded (by selecting a longer task period). Our proposed approach maximizes system utilization in a complementary manner by increasing a task's resource usage beyond its static allocation when resources become available, rather than reducing a task's resources usage from its initial allocation when resources become scarce. Marginal utility can also be used in our approach as a scheduling heuristic.

Tasks with different criticality levels may also be scheduled at the hardware level, instead of only by software. Recently, Zimmer *et al.* [33] proposed a processor platform called *FlexPRET*, where tasks with different criticality levels are scheduled in hardware using a thread-interleaved pipeline. Our approach can be useful even for such a processor, when the number of tasks are more than the number of hardware threads. Moreover, we also believe that the approach described in this paper may be a useful component in a *Precision Timed* (PRET) infrastructure [40].

Synchronous languages are also used to model safety-critical systems [13] and multi-rate synchronous languages (e.g., multiclock *Esterel* [25], *SCADE* [41], and *Prelude* [27]) can model tasks with different execution frequencies. Related languages include *Giotto* [42] and *Simulink* The synchrony hypothesis provides a convenient abstraction that separates the physical time of the execution platform from the logical time of the program. However, synchronous languages cannot faithfully model mixed-criticality tasks. In Giotto, only the timeliness of task synchronization (called jitter) can be specified. The scheduling of multi-rate, mixed-criticality, synchronous task sets has been attempted [8] for uni-processors. However, the execution of low criticality tasks is not guaranteed. Baruah [43] presents a multi-processor scheduling approach for mixed-criticality synchronous tasks, but only for single-rate tasks.

The multi-rate work by Goddard and Jeffay [44] demonstrates the management of buffer sizes in programs defined as *Processing Graphs* mapped to the *Rate-Based Execution* [45] task model. Processing Graphs are similar to SDF [20] and task execution rates depend on the (statically defined) task communication rates. Buffer sizes are bounded by the amount of data that can be produced before it can be consumed. Since data is made available as soon as it is produced, the bound also depends on the task scheduling order. In contrast, our proposed lossless buffering decouples the task execution frequencies from their communication frequencies, allowing (mission critical) tasks to vary their execution frequency depending on the processor utilization. By using delayed semantics, the bounding of each buffer is unaffected by task scheduling order.

The TDMA work by Pop *et al.* [46] integrates the scheduling of mixed *Time Triggered* (TT) and *Event Triggered* (ET) task sets with separate dynamic and static slots for communication. Steiner [47] integrates time-triggered network traffic with non-time triggered traffic. Although the work by both Pop *et al.* and Steiner address a different problem (scheduling over communication networks), there are similarities in that they statically allocate time-triggered traffic. A major difference is that all our tasks are implicitly time triggered (none are event triggered) and that the periods of the time triggered tasks can be assigned bounds. Our current approach does not yet consider distributed embedded systems with a shared communication network; we consider it as future work to integrate our approach with reliable network protocols.

VII. Conclusion

In this paper, we introduced two new levels of task criticality (mission and non-critical) to the synchronous model of computation. This was achieved by relaxing the synchronous task model to support tasks with bounded (mission critical) or unbounded (non-critical) tolerances to deadline misses. A lossless communication model was proposed to allow life and mission critical tasks to communicate deterministically at variable frequencies. We developed a static scheduling approach for life and mission critical tasks that maximizes system utilization. To further improve runtime utilization, tasks are dynamically scheduled during the slack whenever possible. Extensive empirical benchmarking showed that our proposed scheduling approach accepted more task sets and scheduled tasks more fairly than the ER-EDF approach [11] without sacrificing system runtime utilization. Our approach requires, however, more preemptions than ER-EDF. For future work, we plan to implement and study the effect of preemption and scheduling overhead in more detail. We also plan to extend the definition of task criticality to consider energy use and to develop improved fairness heuristics.

Appendix A

A. Extension to Additional Levels of Criticality

Certification standards typically define more than three levels of criticality. For example, the *DO-178B standard* [3] defines 5 levels, labeled *A* to *E*. Although our mixed-criticality task model has only dealt with three levels of criticality, it can be extended to 5 levels in the following manner. Life critical tasks correspond to the highest level *A*, while non-critical tasks correspond to the lowest level *E*. Mission critical tasks correspond to the intermediate levels *B*, *C*, and *D*. We can use the variability in execution frequency of a mission critical task to decide whether it is level *B*, *C*, or *D*. E.g., defining variability as $v = f_{\tau}^{min}/f_{\tau}^{max}$, a mission critical task is level *B* if $0.66 < v < 1$, level *C* if $0.33 < v \leq 0.66$, or level *D* if $0 < v \leq 0.33$. Alternatively, task criticality can also be defined in terms of functional correctness and fault tolerance, e.g., as for Pellizzoni *et al.* [22] in their pacemaker design. By combining our timing-centric task model with a function-centric task model, more criticality levels can be supported.

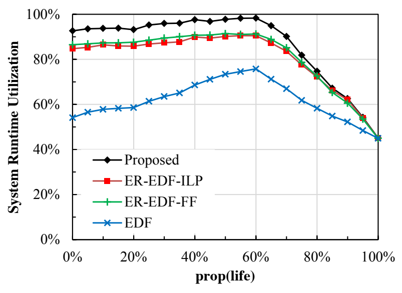
B. Effects of Non-Critical Tasks

We repeat the evaluation described in Section V-B except we introduce non-critical tasks into the generated task sets. Let $prop(mission)$ and $prop(non\text{-}critical)$ denote the proportion of mission and non-critical tasks generated in each task set. For the evaluation, we vary $prop(non\text{-}critical)$ from 10% to 90%. Equal proportions of life and mission critical tasks form the remaining tasks, i.e., $prop(life) = prop(mission)$. The normalized system utilization is left at 50% and we do not change how the system utilization (equation (16)) is calculated. Thus, as $prop(non\text{-}critical)$ increases from 10% to 90%, we generate a relatively constant number of life and mission critical tasks but an increasing number of non-critical tasks.

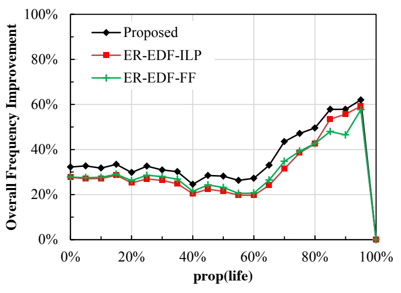
Figure 10a shows that the system runtime utilization reaches 100% quickly with increasing $prop(non\text{-}critical)$. This is because more non-critical tasks are generated to use the slack. Figures 10b and 10c show the values of $f_{mission}^{improve}$ and fairness for the mission critical tasks. Both $f_{mission}^{improve}$ and fairness are relatively constant as $prop(non\text{-}critical)$ increases. This is because of two factors. First, mission critical tasks are rarely scheduled in the dynamic scheduling phase because non-critical tasks have higher priority (Section IV-G). Second, the static scheduling algorithm (Section IV-E) allocates slack to the mission critical tasks in a proportionately fair manner. Hence, the $f_{mission}^{improve}$ and fairness values shown in Figures 10b and 10c are mostly due to the static schedule. Note that when $prop(non\text{-}critical)$ is 10%, the system runtime utilization is less than 100%. This means all the mission and non-critical tasks get scheduled during the dynamic scheduling phase. As a result, $f_{mission}^{improve}$ is at the highest (19.8%) when $prop(non\text{-}critical)$ is 10%.

C. ILP Scalability

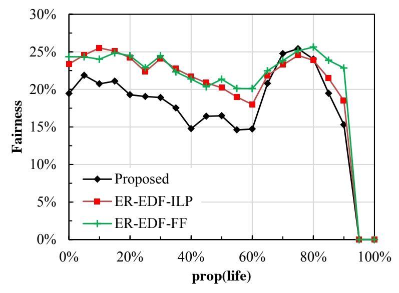
`Gurobi` solves ILP problems iteratively by finding better locally optimal solutions until the globally optimal solution is found. It is possible to run `Gurobi` until it finds the first locally optimal solution for a substantially shorter solving time. Running `Gurobi` in this mode is similar to using a heuristic because the first locally optimal solution can be treated as an approximation to the final globally optimal solution. To demonstrate this approach, we used `Gurobi` to solve the ILP constraints of Section IV-E (including fairness) on 250 randomly generated task sets, each containing 2 to 50 tasks. Figure 11 shows that `Gurobi` finds locally optimal solutions for all task sets in under 1 second, but increasingly more time is required to find the globally optimal solutions.



(a) System runtime utilization, $N = 8$.

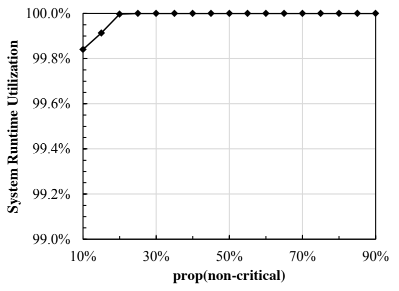


(b) Overall frequency improvement, $N = 8$.

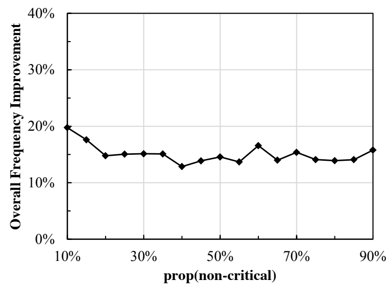


(c) Fairness, $N = 8$.

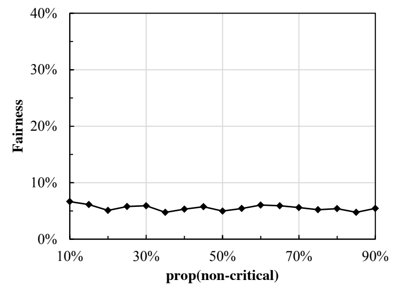
Fig. 9. The system runtime utilization, overall frequency improvement, and fairness as $prop(life)$ is varied. Normalized system utilization = 50%.



(a) System runtime utilization, $N = 4$.



(b) Overall frequency improvement, $N = 4$.



(c) Fairness, $N = 4$

Fig. 10. The system runtime utilization, overall frequency improvement, and fairness as $prop(non\text{-}critical)$ is varied. Normalized system utilization = 50%.

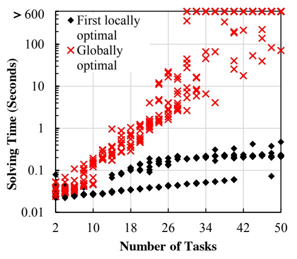


Fig. 11. Times for Gurobi to find the first locally optimal and the globally optimal solution. Note that the time scale is logarithmic.

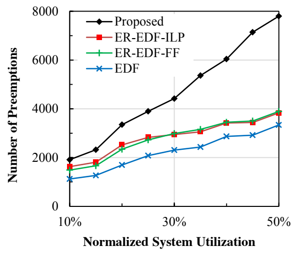


Fig. 12. Number of preemptions as normalized system utilization is varied. $prop(life) = 50\%$ and $N = 4$.

D. Response Times and Preemptions

The use of delayed communication in our approach means that the outputs computed by a task are only made available at the end of the task's period. Thus, with respect to the timing of outputs, a task's response time correlates to its execution frequency. For a life critical task, its response time is constant because its minimum and maximum frequencies, f_τ^{min} and f_τ^{max}, are equal. For a mission critical task, its execution frequency is statically guaranteed to meet f_τ^{min} but can improve towards f_τ^{max} at runtime. Hence, the response time is bounded by f_τ^{min} and f_τ^{max}. Figures 8b and 9b demonstrate the ability of mission critical tasks to improve their execution frequency. For a non-critical task, it only tries to meet its response time because it only has a goal frequency.

The cost of preempting tasks cannot be ignored when implementing a system. Thus, to gauge these costs, we repeat the evaluation described in Section V-B, but vary the normalized system utilization with $prop(life) = 50\%$, and record the average number of preemptions that occur on each processor. A preemption is recorded whenever a task is interrupted to allow other tasks or the scheduler to execute. Figure 12 shows that the proposed approach requires nearly twice the number preemptions than ER-EDF. Thus, in return for achieving higher system utilization and frequency improvements, the proposed approach may incur higher preemption penalties than ER-EDF. However, the preemption cost in terms of execution time depends on how efficiently both approaches are implemented.

References

[1] W. R. Dunn, "Designing Safety-Critical Computer Systems," *Computer*, vol. 36, no. 11, pp. 40 – 46, 2003.

[2] International Electrotechnical Commission, "IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems," Apr. 2010.

[3] Radio Technical Commission for Aeronautics, "Software Considerations in Airborne Systems and Equipment Certification," Apr. 1992.

[4] S. Vestal, "Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance," in *28th Real-Time Systems Symposium (RTSS)*, 2007, pp. 239 – 243.

[5] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. M. Burguiere, J. Reineke, B. Triquet, and R. Wilhelm, "Predictability Considerations in the Design of Multi-Core Embedded Systems," *Embedded Real Time Software and Systems (ERTS)*, 2010.

[6] D. de Niz, K. Lakshmanan, and R. Rajkumar, "On the Scheduling of Mixed-Criticality Real-Time Task Sets," in *30th Real-Time Systems Symposium (RTSS)*, 2009, pp. 291 – 300.

[7] P. Ekberg and W. Yi, "Bounding and Shaping the Demand of Mixed-Criticality Sporadic Tasks," in *Real-Time Systems (ECRTS), 24th Euromicro Conference on*, 2012, pp. 135–144.

[8] S. K. Baruah, "Semantics-Preserving Implementation of Multirate Mixed-Criticality Synchronous Programs," in *20th International Conference on Real-Time and Network Systems RTNS*, 2012, pp. 11 – 19.

[9] H. Su and D. Zhu, "An Elastic Mixed-Criticality Task Model and its Scheduling Algorithm," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 147 – 152.

[10] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, "Mixed-Criticality Real-Time Scheduling for Multicore Systems," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, 2010, pp. 1864 – 1871.

[11] H. Su, D. Zhu, and D. Mosse, "Scheduling Algorithms for Elastic Mixed-Criticality Tasks in Multicore Systems," in *19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2013.

[12] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin, "Mixed-Criticality Scheduling on Multiprocessors," *Real-Time Systems*, pp. 1 – 36, 2013.

[13] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The Synchronous Languages 12 Years Later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64 – 83, Jan. 2003.

[14] P. Caspi and O. Maler, "From Control Loops to Real-Time Programs," in *Handbook of Networked and Embedded Control Systems*, ser. Control Engineering, D. Hristu-Varsakelis and W. S. Levine, Eds. Birkhauser Boston, 2005, pp. 395 – 418.

[15] S. A. Edwards and J. Zeng, "Code Generation in the Columbia Esterel Compiler," *EURASIP Journal on Embedded Systems*, vol. 2007, 2007.

[16] M. D. Natale and H. Zeng, "Task Implementation of Synchronous Finite State Machines," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012, pp. 206 – 211.

[17] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 1 – 53, 2008.

[18] M. Boldt, C. Traulsen, and R. von Hanxleden, "Worst Case Reaction Time Analysis of Concurrent Reactive Programs," *Electronic Notes in Theoretical Computer Science*, vol. 203, no. 4, 2008.

[19] S. Baruah and G. Fohler, "Certification-Cognizant Time-Triggered Scheduling of Mixed-Criticality Systems," in *32nd Real-Time Systems Symposium (RTSS)*, 2011, pp. 3 – 12.

[20] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235 – 1245, 1987.

[21] G. S. C. Avellar, G. D. Thums, R. Lima, P. Iscold, L. A. B. Torres, and G. A. S. Pereira, "On the Development of a Small Hand-Held Multi-UAV Platform for Surveillance and Monitoring," in *Unmanned Aircraft Systems (ICUAS), 2013 International Conference on*, 2013, pp. 405 – 412.

[22] R. Pellizzoni, P. Meredith, M.-Y. Nam, M. Sun, M. Caccamo, and L. Sha, "Handling Mixed-Criticality in SoC-Based Real-Time Embedded Systems," in *Proceedings of the seventh ACM International Conference on Embedded Software*, ser. EMSOFT '09. ACM, 2009, pp. 235 – 244.

[23] D. Goswami, M. Lukasiewycz, R. Schneider, and S. Chakraborty, "Time-Triggered Implementations of Mixed-Criticality Automotive Software," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, 2012, pp. 1227 – 1232.

[24] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. D. Michiel, "PapaBench: A Free Real-Time Benchmark," in *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[25] Esterel Technologies, *The Esterel V7 Reference Manual*, 7th ed., 2005.

[26] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305 – 1320, 1991.

[27] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens, "Multi-task Implementation of Multi-periodic Synchronous Programs," *Discrete Event Dynamic Systems*, vol. 21, no. 3, pp. 307 – 338, 2011.

[28] E. Yip, P. S. Roop, M. Biglari-Abhari, and A. Girault, "Programming and Timing Analysis of Parallel Programs on Multicores," in *13th International Conference on Application of Concurrency to System Design (ACSD)*, Jul. 2013.

[29] F. Boussinot and R. D. Simone, "The SL Synchronous Language," *Software Engineering, IEEE Transactions on*, vol. 22, no. 4, pp. 256 – 266, 1996.

[30] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis, "Semantics-Preserving Multitask Implementation of Synchronous Programs," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 2, pp. 1 – 40, Jan. 2008.

[31] G. Wang, M. D. Natale, and A. Sangiovanni-Vincentelli, "Improving the Size of Communication Buffers in Synchronous Models With Time Constraints," *Industrial Informatics, IEEE Transactions on*, vol. 5, no. 3, pp. 229 – 240, 2009.

[32] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee, "A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance," in *Proceedings of the 30th IEEE International Conference on Computer Design (ICCD 2012)*. IEEE, 2012, pp. 87–93.

[33] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "FlexPRET: A Processor Platform for Mixed-Criticality Systems," in *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS)*. IEEE, 2014.

[34] T. Lan, D. Kao, M. Chiang, and A. Sabharwal, "An Axiomatic Theory of Fairness in Network Resource Allocation," in *INFOCOM, 2010 Proceedings IEEE*, 2010, pp. 1 – 9.

[35] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel, "Proportionate Progress: A Notion of Fairness in Resource Allocation," *Algorithmica*, vol. 15, no. 6, pp. 600 – 625, 1996.

[36] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson, "RTOS Support for Multicore Mixed-Criticality Systems," in *18th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2012, pp. 197 – 208.

[37] D. de Niz, L. Wrage, N. Storer, A. Rowe, , and R. Rajkumar, "On Resource Overbooking in an Unmanned Aerial Vehicle," in *Cyber-Physical Systems (ICCPS), 2012 IEEE/ACM Third International Conference on*, 2012, pp. 97 – 106.

[38] K. Lakshmanan, D. D. Niz, R. Rajkumar, and G. Moreno, "Overload Provisioning in Mixed-criticality Cyber-physical Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 11, no. 4, pp. 1 – 24, Jan. 2013.

[39] H.-M. Huang, C. Gill, and C. Lu, "Implementation and Evaluation of Mixed-Criticality Scheduling Approaches for Periodic Tasks," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE 18th*, 2012, pp. 23 – 32.

[40] D. Broman, M. Zimmer, Y. Kim, H. Kim, J. Cai, A. Shrivastava, S. A. Edwards, and E. A. Lee, "Precision Timed Infrastructure: Design Challenges," in *Proceedings of the Electronic System Level Synthesis Conference (ESLsyn)*. IEEE, May 2013.

[41] Esterel Technologies. The SCADE Product Family. http://www.esterel-technologies.com/. [Last accessed 12 Oct 2013].

[42] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A Time-Triggered Language for Embedded Programming," in *Embedded Software*, ser. Lecture Notes in Computer Science, T. Henzinger and C. Kirsch, Eds., 2001, vol. 2211, pp. 166 – 184.

[43] S. Baruah, "Implementing Mixed-Criticality Synchronous Reactive Systems Upon Multiprocessor Platforms," The University of North Carolina at Chapel Hill, Tech. Rep., 2013, http://www.cs.unc.edu/∼baruah/Submitted/2013-SubmitECRTS.pdf.

[44] S. Goddard and K. Jeffay, "Managing Latency and Buffer Requirements in Processing Graph Chains," *The Computer Journal*, vol. 44, no. 6, pp. 486 – 503, 2001.

[45] K. Jeffay and S. Goddard, "Rate-Based Resource Allocation Models for Embedded Systems," in *Embedded Software*, ser. Lecture Notes in Computer Science, T. A. Henzinger and C. M. Kirsch, Eds. Springer Berlin Heidelberg, 2001, vol. 2211, pp. 204 – 222.

[46] T. Pop, P. Eles, and Z. Peng, "Holistic Scheduling and Analysis of Mixed Time/Event-Triggered Distributed Embedded Systems," in *Hardware/Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on*, 2002, pp. 187–192.

[47] W. Steiner, "Synthesis of Static Communication Schedules for Mixed-Criticality Systems," in *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on*, 2011, pp. 11–18.