Mixed-Criticality Real-Time Scheduling for Multicore Systems

Malcolm S. Mollison*, Jeremy P. Erickson*,
James H. Anderson*, Sanjoy K. Baruah* and John A. Scoredos†

*University of North Carolina at Chapel Hill
{*mollison, jerickso, anderson, baruah*}*@cs.unc.edu*

†Northrop Grumman Corporation
john.scoredos@ngc.com

Abstract

Current hard real-time scheduling and analysis techniques are unable to efficiently utilize the computational bandwidth provided by multicore platforms. This is due to the large gap between worst-case execution time predictions used in schedulability analysis and actual execution times seen in practice. In this paper, we view this gap as "slack" that can be accounted for during schedulability analysis and reclaimed for less critical work. We use this technique to develop an architecture for scheduling mixed-criticality real-time workloads on multiprocessor platforms. Our architecture provides temporal isolation among tasks of different criticalities while allowing slack to be redistributed across criticality levels.

1. Introduction

In order to enable the next generation of *cyber-physical systems*, or systems that exhibit a high degree of interaction between computational and physical components, significant innovation in real-time scheduling and analysis will be necessary. Many future cyber-physical systems will exhibit two important characteristics that differentiate them from current systems.

First, they will make use of multicore platforms. The adoption of multicore platforms will be driven by the need for additional computational bandwidth, and by size, weight, and power (SWaP) concerns. Additionally, as major chip manufacturers switch to multicore designs in their product lines, reduced availability of single-core chips may be a factor.

Second, future systems will be supported by computational workloads that are both complex and of *mixed criticality*. Criticality is a designation of the level of assurance against failure needed for a task. A mixed-criticality task system is one in which the criticality levels demanded by tasks are diverse.

A good example of such a system is provided by the next-generation unmanned aerial vehicles (UAVs) currently under development in the aerospace industry [4]. Next-generation UAVs will have combat and reconnaissance capabilities surpassing those of currently deployed manned aircraft. Their workloads can be divided into three broad categories of tasks, each of which is likely to span criticality levels:

1) Tasks that perform "safety-critical" operations, such as adjusting flight surfaces to maintain aerial stability. Such tasks are already supported by embedded computers in existing airplanes (albeit with some limitations, discussed below).
2) Tasks that perform "mission-critical" operations, such as external communication and advanced decision-making. Such functions are performed by pilots in existing systems. An example would be intelligently responding to radar detection of an unfriendly aircraft.
3) Tasks that perform other, "background" work. Of particular note are tasks executing advanced planning algorithms that are not available in existing systems. An example would be finding an optimal route through an area under enemy radar surveillance. Oftentimes, these algorithms solve optimization problems, the results of which improve with time.

There is strong motivation for hosting the described workload on a multicore platform. Such a platform would produce significant SWaP improvements, which is of critical importance for UAVs. In addition, the alternative of using multiple single-core systems connected via a network would significantly limit computational throughput and increase costs.

Work supported by AT&T, IBM, Northrop Grumman, and Sun Corps.; NSF grants CNS 0834270 and CNS 0834132; ARO grant W911NF-09-1-0535; and AFOSR grant FA9550-09-1-0549.

Unfortunately, with existing real-time scheduling techniques, hosting mixed-criticality workloads on multicore platforms is not possible. This is due to two related problems: **(i)** underutilization caused by pessimism in worst-case execution time (WCET) analysis, especially for high-criticality tasks; and **(ii)** the need to temporally isolate tasks of differing criticality.

WCET analysis is needed to validate that tasks will complete by their specified deadlines. Even on single-core systems, WCET analysis is highly problematic. To compensate for this, pessimistic WCET estimates are used. These estimates increase in pessimism with increasing criticality level. This leads to an under-utilization of computing resources in practice and severely limits the computational workload that can be supported by a given piece of hardware. In order to make effective use of the computational bandwidth provided by multicore platforms, a scheduling methodology that makes use of this "slack" processing capability would be desirable.

Requirements for temporal isolation state that if a lower-criticality task can impact the scheduling of a higher-criticality task, it must be designed and certified at the criticality level of the higher-criticality task [15]. Doing so is often impractical: it exacerbates the WCET problem, raises costs significantly, and may require additional programming constraints to be placed on lower-criticality tasks, such as using only static loop bounds.

In this paper, we present an architecture for mixed-criticality scheduling on multicore platforms. We view higher-criticality tasks as "slack generators" that, in the common case, will use only a small fraction of the execution time budgeted for them. Lower-criticality tasks are then budgeted to run using this slack. Thus, computational hardware is used more efficiently, allowing more demanding task systems to be deployed. Our architecture preserves isolation among criticality levels from the perspective of temporal correctness.

The rest of this paper is organized as follows. In Sec. 2, we discuss relevant background material and related work. In Sec. 3, we present our architecture for mixed-criticality systems. In Sec. 4, we give correctness proofs for our architecture. In Sec. 5, we discuss future work. We conclude in Sec. 6.

2. Background

In the following subsections, we present relevant background information on multiprocessor real-time scheduling and mixed-criticality scheduling.

2.1. Multiprocessor Real-Time Scheduling

In this paper, we assume that temporal constraints for tasks can be modeled by the *periodic task model*. Under this model, each task T has an associated *period*, $T.p$, and WCET, $T.e$. Successive *jobs* of T are released every $T.p$ time units, starting at time 0, and a job released at time t must complete by its *deadline*, $t + T.p$. The *utilization*, or long-run processor share required by a task, is given by $T.u = T.e/T.p$. A *harmonic task system* is a periodic task system in which all task periods are integer multiples of the smallest task period. It is common for avionics workloads, among others, to be modeled as harmonic task systems.

A task system is *schedulable* if, given a scheduling algorithm and m processors, the algorithm can schedule tasks in such a way that all temporal constraints are met. For *hard real-time* tasks, jobs must never miss their deadlines, while for *soft real-time* tasks, some deadline misses are tolerable. Specifically, we require here that the tardiness of jobs of soft real-time tasks be bounded by a (reasonably small) constant.

Scheduling algorithms. Two scheduling approaches are common for multicore systems: *partitioning* and *global scheduling*. In a partitioning scheme, tasks are statically assigned to processors, and do not migrate. An example is *partitioned EDF* (P-EDF), which preemptively schedules jobs in earliest-deadline-first (EDF) order on each processor. In global scheduling, tasks may migrate across processors. An example is *global EDF* (G-EDF), which schedules jobs from a single deadline-ordered run queue.

P-EDF is often a good choice for hard real-time task systems [6]. One downside of P-EDF is that spare capacity on processors may be wasted due to bin-packing issues that arise when assigning tasks to processors.

In contrast, G-EDF is often a good choice for soft real-time task systems, because the bin-packing issues that arise under P-EDF are alleviated if bounded tardiness is permitted [9, 10]. G-EDF is particularly well suited for task systems that are provisioned on the basis of average execution time [11]. A downside of G-EDF is that contention for shared caches and busses increases, making WCET analysis more difficult.

In this paper, we make use of P-EDF, G-EDF, and a multiprocessor adaptation of the uniprocessor *cyclic executive* scheduling approach [3]. A cyclic executive is a simple real-time executive that dispatches tasks according to a table that is precomputed offline. Cyclic executives are often preferred for high-criticality real-time task systems because tasks are scheduled in a highly predictable way, which makes certification easier. Cyclic executives are best suited for scheduling harmonic task systems, since in that case, the size of the dispatching table can be pseudo-

	Crit.	$T.p$	$T.e_A$	$T.u_A$	$T.e_B$	$T.u_B$
T_1	A	10	5	0.5	3	0.3
T_2	A	20	10	0.5	6	0.3
T_3	B	20	–	–	4	0.2
T_4	B	40	–	–	8	0.2
Σ				1.0		1.0

Table 1: Multi-criticality task system for Example 1.

polynomially bounded.

Hierarchical scheduling. We use a two-level hierarchical scheduling approach in our mixed-criticality architecture. In hierarchical scheduling, special tasks known as *container tasks* are scheduled alongside normal tasks. Each container task schedules tasks from an associated *container* (also called a *server*). Hierarchical scheduling is typically used to allow the timing correctness of subsystems to be validated independently. Well-known hierarchical scheduling schemes include *constant bandwidth servers* [1], *total bandwidth servers* [13], and *resource kernels* [12].

2.2. Mixed-Criticality Scheduling

The conventional approach to scheduling high criticality tasks on uniprocessor systems is to use very pessimistic WCET values. Such a system may be fully utilized from a validation and certification perspective, i.e., at *design time*, but will be severely underutilized in practice, i.e., at runtime.

Vestal proposed a technique for accounting for this under-utilization and reclaiming it at design time for lower-criticality tasks [15]. In his work, it is assumed that tasks are scheduled using a static-priority, uniprocessor scheduling algorithm. In this class of scheduling algorithms, a task's schedulability is dependent on the WCET values of tasks of equal or higher priority. Vestal observed that, from the perspective of scheduling a less-critical task, the WCET values given for more-critical tasks are needlessly pessimistic. Thus, he proposed that schedulability tests for less critical tasks be altered to incorporate less pessimistic WCET values for more critical tasks. In light of this, multiple WCET values must be assigned to each task: one for its own criticality level, and one for each lower criticality level. Moreover, per-criticality-level schedulability tests must be used (L variants of an L-level system must be analyzed). The WCET value for task T at level X is denoted $T.e_X$ and the resulting utilization is denoted $T.u_X$. This has come to be known as the *multi-criticality task model*.

Example 1. Table 1 gives an example single-processor multi-criticality task system. T_1 and T_2 are highly-critical tasks (level A) that are assigned very pessimistic WCET values. If this degree of assurance were needed for the entire task system, no additional tasks could be included, because T_1 and T_2 would be assumed to fully utilize the processor. However, T_3 and T_4 are lower-criticality tasks (level B) that do not need such a high level of assurance. In fact, by assuming level-B execution costs and statically prioritizing tasks in the order T_1 (highest) to T_4 (lowest), all four tasks can be accommodated. Note that under this priority assignment, T_3 and T_4 can never impact the scheduling of T_1 and T_2.

Subsequent work by Baruah and Vestal [5] examined scheduling-theoretic issues that arise in the context of the multi-criticality task model. They proposed a new uniprocessor scheduling algorithm that dominates static-priority algorithms for multi-criticality task systems.

Anderson et al. [2] proposed a multi-criticality scheduling approach for multicore platforms that uses a two-level hierarchical scheduling framework in which containers provide isolation for tasks of different criticality levels. P-EDF is used as the intra-container scheduler. They also proposed the use of slack re-allocation techniques to redistribute unused processing capacity at higher criticality levels to lower criticality levels. However, the development of exact rules for slack redistribution was left as future work. This paper builds upon the foundation provided in [2] by proposing a framework in which different intra-container schedulers are used for tasks of different criticalities. In addition, slack redistribution rules are devised.

3. Mixed-Criticality Architecture

We seek to provide a scheduling architecture suitable for real-world mixed-criticality task systems scheduled on multicore platforms. While most prior theoretical research on mixed-criticality scheduling allows for an arbitrary number of criticality levels, in practice, the number of such levels is likely to be small. For example, RCTA standard DO-178B, which is used by the U.S. Federal Aviation Administration (FAA) for the certification of commercial airplanes, allows for five criticality levels, labeled A (most critical) through E (least critical). Our architecture assumes a similar five-level classification.

In our architecture, tasks at each criticality level are scheduled by different intra-container schedulers, and thus according to different scheduling policies. This allows the tasks of each criticality level to be scheduled in a way that is appropriate for that level. This scheme is described in more detail below, and illustrated in Fig. 1.

Level A. A table-driven scheduling approach, similar to that used in a real-time cyclic executive, is used to schedule level-A tasks. These tasks are statically assigned to processors, and a dispatching table for the tasks assigned

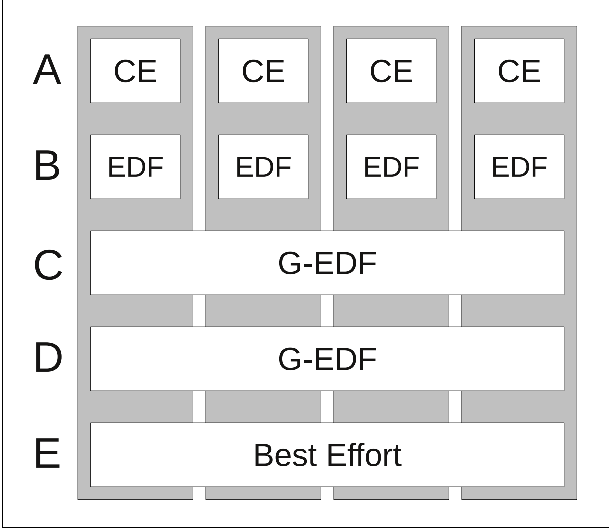


Figure 1: Container scheduling for a four-processor system. Here, CE stands for cyclic executive.

to each processor is used.[1] From a hierarchical scheduling perspective, the level-A tasks assigned to each processor form a container whose tasks are served by the table-driven scheduler. The table-driven scheduler is the highest-priority task in the system from the viewpoint of the top-level scheduler, so it always runs when a level-A task becomes eligible.

Rationale. Level-A tasks are the highest-criticality tasks that need to be scheduled. Table-driven scheduling is the de facto standard for scheduling high-criticality workloads.

Level B. Similarly to level A, each processor hosts a level-B container, and each level-B task is assigned to one of these containers. P-EDF is used at level B, so each level-B container is served by an EDF scheduler. Level B has the second-highest priority from the viewpoint of the top-level scheduler, so any eligible level-B task will execute if no level-A task is executing on the same processor. For each processor, we require that the periods of all level-B tasks be integer multiples of the level-A *hyperperiod* (the least common multiple of level-A task periods). Also, assuming level-B WCET values, the sum of the utilizations of all level-A and -B tasks must not exceed 1.0. As shown in Sec. 4, these conditions are sufficient for level-B schedulability, assuming no level-B WCET is exceeded (by tasks at level A or B).

Rationale. P-EDF is a simple and well-studied scheduling algorithm with relatively low overhead. Because task execution patterns are not constrained by a table, using EDF scheduling eases software development and allows a wider range of tasks systems to be scheduled. For these reasons, P-EDF is appropriate for scheduling hard real-time workloads that do not need to be table-driven.

Level C. Level-C tasks are globally scheduled using the G-EDF algorithm. All level-C tasks are grouped into the same container, which is served by all processors. G-EDF is invoked on any processor whenever level-C tasks are eligible but no higher-criticality tasks are eligible. In Sec. 4, we give a schedulability test for ensuring bounded tardiness for level-C tasks. The test uses level-C WCET values for tasks of criticality C and higher.

Rationale. In our architecture, level C provides a facility for supporting tasks for which a relatively small amount of tardiness is acceptable.

Level D. Level D provides an additional level of G-EDF scheduling and behaves in a similar fashion to level C. G-EDF is invoked on any processor to schedule a task from the level-D container whenever level-D tasks are eligible but no higher-criticality tasks are eligible.

Rationale. By providing an additional criticality level, level D enables further utilization of the system in practice, but with a less strong guarantee that deadlines will be met. Level D is furthermore distinguished from level C because its tardiness bounds are significantly less tight than those available for level C. This topic is discussed further in Sec. 4.

Level E. At level E, "best effort" jobs are scheduled by a server that is invoked whenever a processor would otherwise be idle. Level-E jobs are guaranteed to receive a long-term utilization equal to m minus the sum of the utilizations of all real-time tasks present in the system, based on level-E WCET values. Any non-real-time scheduling policy can be used at level E.

Rationale. Level-E WCET values are likely to be provisioned based on average-case behavior observed during testing. Thus, level E is suitable for long-running tasks that need to make a predictable amount of progress over time, and for short-running tasks for which a quick response time is desirable.

Slack shifting. We allow lower-criticality tasks to execute before eligible higher-criticality tasks, when doing so does not cause deadline misses or impact temporal isolation requirements. Each time a real-time job is released, it is allocated a *budget* equal to its WCET value for its own criticality level. As the job executes, the budget is depleted. If the job completes before its budget is exhausted, it becomes a *ghost job*. For scheduling purposes,

1. Existing approaches for creating dispatching tables may be used to ensure that level-A tasks will meet their deadlines [3].

ghost jobs are viewed identically to normal jobs. However, any intra-container scheduler that selects a ghost job to execute instead suspends until the next time that it (the scheduler) is invoked, which will occur when the ghost job exhausts its budget or when a higher-priority job at the same criticality level is released. This has the effect of shifting higher-criticality slack, which is consumed by lower-criticality tasks, earlier in the schedule. The budget of the ghost job continues to be depleted until work of equal or higher criticality commences execution on the same processor. When the budget of the ghost job is exhausted, it is removed from the system.

Rationale. Slack shifting allows tardy jobs at levels C and D to execute sooner than they otherwise would, thereby decreasing tardiness in the system. It also allows best-effort jobs at level E to run earlier than they would if no slack shifting occurred, improving response time.

Temporal isolation. Intra-container schedulers only suspend (i.e., donate slack) during times for which an already-completed task was provisioned to run, based on its WCET value for that criticality level. Thus, our slack shifting scheme does not violate temporal isolation requirements.

Example 2. Table 2 gives a five-criticality task system provisioned for our architecture. Fig. 2 gives an example schedule for this system. Note that in order to make our example more palatable, WCETs were artificially restricted to small values. In a real task system, the level-A WCET for a level-A task is likely to be orders of magnitude greater than its observed average execution time.

4. Schedulability

In this section, we prove that no level-A (level-B) task misses a deadline if all level-A (level-A and level-B) tasks execute for at most their level-A (level-B) WCETs. We establish similar results for levels C and D, except at those levels, only bounded tardiness is required, and level-C and -D WCETs (respectively) are assumed in the analysis. (No correctness proof is required for level E, as jobs at this level are scheduled on a best-effort basis.) Level-A correctness is straightforward: any correct cyclic executive schedule remains correct regardless of levels below it, since the level-A container (on each processor) is statically prioritized over all lower-criticality containers.

4.1. Level B

The analysis for level B uses level-B WCET values (for level-A and -B tasks), and assumes that those values are not exceeded at runtime.

Recall that we limited the total utilization of level-A and -B tasks on any processor to 1.0. Any periodic task system with utilization at most 1.0 is schedulable using EDF. Therefore, if EDF scheduling were used for all level-A and -B tasks, level-B tasks would never miss deadlines.

Let us denote the interval of time between the release and deadline of a job as the *window* of the job. Because of the requirement that level-B periods be integer multiples of the level-A hyperperiod, the window of each level-B job overlaps (without overhang) the window(s) of any level-A job(s) that may block it from executing. Thus, although a level-A job may execute at a different time in its window than it would under EDF (since table-driven scheduling is used), it cannot reduce the amount of processor time available to a level-B job within that level-B job's window, assuming level-B WCET values.

Therefore, no level-A job can cause a level-B job that completed by its deadline under full EDF scheduling to miss its deadline when level-A jobs are not EDF-scheduled. This means that level-B jobs will not miss their deadlines under our architecture (assuming level-B WCET values for level-A and -B tasks).

4.2. Levels C and D

At levels C and D, some of the time on each processor has already been consumed by levels A and B. Thus, the schedules at levels C and D can differ significantly from a typical G-EDF schedule in which all processors are available at all times. For example, in Fig. 2, over the interval $[2, 3)$ both processors are available, whereas over the interval $[3, 4)$ only the second processor is available, causing T_8 to migrate. Therefore, we must use analysis that takes into account restricted processor supplies. Such analysis, for G-EDF, has been given by Leontyev and Anderson [10].

Leontyev and Anderson considered a G-EDF-scheduled system where the supply on processor k is characterized by a *service function* [8]

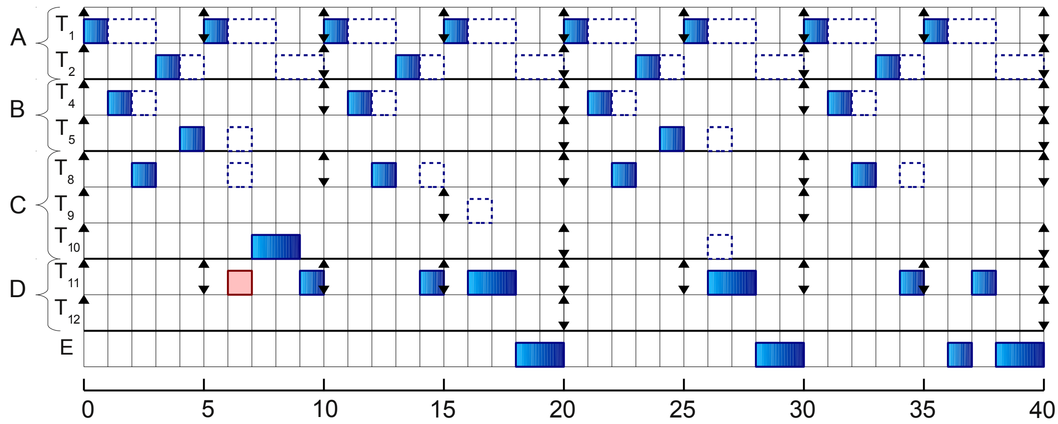
$$\beta_k(\Delta) = \max(0, \widehat{u_k} \cdot (\Delta - \sigma_k)) \tag{1}$$

where $\widehat{u_k} \in (0, 1]$ is the long-term available utilization on processor k and σ_k is a blocking term related to the longest duration of time when processor k can be unavailable. σ_k must be chosen such that $\beta_k(\Delta)$ lower bounds actual availability. Denote S as the task system under consideration. Denote U_S as the sum of the $m - 1$ largest $T_i.u$ values for $T_i \in S$, and F as the number of processors which may not be fully available. [10] demonstrates that tardiness is bounded assuming $\sum_{T_i \in S} T_i.u \leq \sum_{k=1}^{m} \widehat{u_k}$ and $\sum_{k=1}^{m} \widehat{u_k} - \max(F-1, 0) \cdot \max_{T_i \in S}(T_i.u) - U_S > 0$.

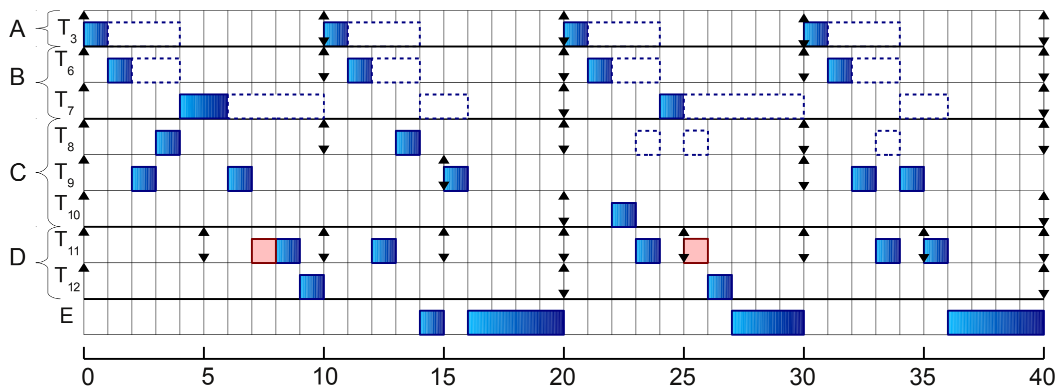
In our setting, choosing the proper parameters requires analyzing the particular workloads of higher criticality

	Crit.	CPU	$T.p$	$T.e_A$	$T.e_B$	$T.e_C$	$T.e_D$	$T.e_E$
T_1	A	1	5	3	2	1	1	1
T_2	A	1	10	4	2	2	2	1
T_3	A	2	10	4	3	2	1	1
T_4	B	1	10	–	2	2	1	1
T_5	B	1	20	–	2	1	1	1
T_6	B	2	10	–	3	2	1	1
T_7	B	2	20	–	8	3	2	2
T_8	C	Global	10	–	–	3	2	2
T_9	C	Global	15	–	–	2	2	2
T_{10}	C	Global	20	–	–	2	2	2
T_{11}	D	Global	5	–	–	–	2	2
T_{12}	D	Global	20	–	–	–	1	1

Table 2: Multi-criticality task system for Example 2.



(a) CPU 1



(b) CPU 2

Figure 2: Possible schedule for the task system in Example 2. Darker boxes indicate on-time task execution, while lighter boxes indicate tardy task execution. Empty boxes represent slack shifting. Up-arrows indicate releases, while down-arrows indicate deadlines.

levels, but making level-C or -D assumptions about worst-case execution times. To apply the above results to our architecture, we extend the notation introduced above, by allowing the criticality level L (C or D) to be specified, e.g., we use $\widehat{u_{Lk}}$ instead of $\widehat{u_k}$. Denote L as the set of all tasks running at level L. The result of [10] in our setting implies that tardiness at level L is bounded assuming

$$\sum_{T_i \in L} T_i.u_L \leq \sum_{k=1}^{m} \widehat{u_{Lk}}, \quad (2)$$

$$\sum_{k=1}^{m} \widehat{u_{Lk}} - (m-1) \cdot \max_{T_i \in L}(T_i.u_L) - U_L > 0, \quad (3)$$

and no task at any level exceeds its level-L WCET.

We first consider level C. Denote H_{Ck} as the set of all tasks on processor k above level C. Then $\widehat{u_{Ck}} = 1 - \sum_{T_i \in H_{Ck}} T_i.u_C$. To illustrate how level-C analysis is conducted, we consider the system in Example 2. To analyze this system, we use the $T_i.e_C$ column of Table 2. In particular,

$$\widehat{u_{C1}} = 1 - \left(\frac{1}{5} + \frac{2}{10} + \frac{2}{10} + \frac{1}{20}\right) = \frac{7}{20}$$

and

$$\widehat{u_{C2}} = 1 - \left(\frac{2}{10} + \frac{2}{10} + \frac{3}{20}\right) = \frac{9}{20}.$$

Also, substituting into (2),

$$\frac{3}{10} + \frac{2}{15} + \frac{2}{20} = \frac{8}{15} < \frac{16}{20},$$

and into (3),

$$\left(\frac{7}{20} + \frac{9}{20}\right) - 1 \cdot \left(\frac{3}{10}\right) - \frac{3}{10} = \frac{1}{5} > 0.$$

Thus, tardiness is bounded at level C for Example 2.

To compute actual tardiness bounds for level C, σ_{Ck} must be determined for each k. The processor-k schedule at levels A and B repeats at every integer multiple of the hyperperiod h of the set of tasks H_{Ck}. We refer to an interval $[nh, (n+1)h)$ for an arbitrary nonnegative integer n as a *cycle*. σ_{Ck} can be computed based on the observation that the upper bound for long-term utilization of tasks in H_{Ck} also upper bounds their utilization within a single cycle. Specifically,

$$\sigma_{Ck} \leq 2h \sum_{T_i \in H_{Ck}} T_i.u_C$$

Equality would be achieved in the case where all execution from H_{Ck} in one cycle occurred at its end and all execution from H_{Ck} in the next cycle occurred at its beginning. Further analysis allows tighter bounds for systems such as that given in Example 2, because the worst-case situation described here cannot happen, but such analysis is omitted due to space reasons. Using equality provides valid, albeit pessimistic, tardiness bounds. In our system, $\sigma_{C1} = 26$ and $\sigma_{C2} = 22$.

In the case of level D, σ_{Dk} and $\widehat{u_{Dk}}$ are problematic to compute. This is because level-C tasks may migrate, so accounting for the supply on each processor independently is not possible. Therefore, computing precise tardiness bounds is difficult. Nonetheless, from (2) and (3), establishing that tardiness is bounded at level D (whatever the bound) merely requires computing $\sum_{k=1}^{m} \widehat{u_{Dk}}$, which is equal to $m - \sum_{T_i \in H_D} T_i.u_D$, where H_D is the set of tasks on all processors above level D. Because no precise bound on tardiness is provided, only soft real-time tasks of low criticality should be scheduled at level D. In the case of the system in Example 2, we use the $T_i.e_D$ column of Table 2, and we have

$$\sum_{k=1}^{m} \widehat{u_{Dk}} = 2 - \left(\frac{1}{5} + \frac{1}{10} + \frac{1}{10} + \frac{1}{10} + \frac{1}{20} + \frac{1}{10} + \frac{2}{20} + \frac{2}{10} + \frac{2}{15} + \frac{2}{20}\right) = \frac{49}{60}.$$

Substituting into (2),

$$\frac{2}{5} + \frac{1}{20} = \frac{9}{20} < \frac{49}{60},$$

and into (3),

$$\frac{49}{60} - \frac{2}{5} - \frac{2}{5} = \frac{1}{60} > 0.$$

Thus, tardiness at level D is bounded.

5. Future Work

Additional work is needed before our architecture could be considered applicable for real workloads. Future research effort can be broken down into two related but distinct areas: implementation studies and enhancement of the architecture.

Implementation studies. We plan to implement our architecture in LITMUS^{RT} (**LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems) [7,14]. LITMUS^{RT} is a UNC-developed real-time operating system testbed based on the Linux kernel. Our implementation will be used for performance evaluations that will provide feedback on the practicality of our architecture. Collecting overhead measurements will be a significant portion of this work.

At a later date, we would like to investigate the possibility of implementing our architecture in an applicable commercial real-time operating system. Because Linux is not particularly well-suited for hard real-time tasks (largely due to sources of unpredictability within the kernel) [6], this would be a key step in demonstrating the feasibility

of our architecture for real-world use.

Architecture enhancement. A number of enhancements to our architecture would expand the range of real-time workloads that can be supported on multicore platforms. Resource sharing, particularly among tasks of different criticalities, is a prime example. Another area of interest is enabling adaptivity, which is likely to be important for future cyber-physical systems, which themselves must adapt to changing environments. For example, in a UAV, when previously-undetected enemy radar stations are located, it might be useful to increase the processor share of a route-planning task. Alternatively, if an enemy missile were to be detected, in order for evasive action or communication to be carried out rapidly, the ability to enact a *mode change* (in which a new set of tasks replaces those currently being scheduled) might be desirable.

Finally, we would like to obtain improved schedulability results for our architecture. For example, we would like to allow sporadic job releases (instead of requiring that jobs be released on period boundaries) and constrained deadlines (i.e., relative deadlines at most periods). We would also like to tighten the tardiness bounds for levels C and D.

6. Concluding Remarks

We have illustrated the challenges and potential rewards of enabling the deployment of mixed-criticality workloads on multicore platforms, and have specified an architecture that addresses those challenges. The fundamental insight exploited by our architecture is the notion of high-criticality tasks as "generators" of slack that can be reclaimed under the multi-criticality task model. Our architecture allows appropriate scheduling techniques to be employed for tasks of different criticalities, while preserving temporal isolation. Furthermore, our slack shifting technique alleviates much of the "unfairness" faced by lower-criticality tasks that would otherwise have to wait for more critical but less urgent higher-criticality work to be completed. Finally, we have outlined important areas of future research that are on the critical path to enabling the employment of our architecture by future cyber-physical systems.

References

[1] L. Abeni and G. C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 3–13, December 1998.

[2] J. H. Anderson, S. K. Baruah, and B. B. Brandenburg. Multicore operating-system support for mixed criticality. In *Proceedings of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, April 2009.

[3] T. P. Baker and A. C. Shaw. The cyclic executive model and ADA. *The Journal Of Real-Time Systems*, 1(1):7–25, 1989.

[4] J. Barhorst, T. Belote, P. Binns, J. Hoffman, J. Paunicka, P. Sarathy, J. Scoredos, et al. A research agenda for mixed criticality systems, 2009. Available online at http://www.cse.wustl.edu/˜cdgill/CPSWEEK09_MCAR/ as of March 5, 2010.

[5] S. K. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 147–155, July 2008.

[6] B. B. Brandenburg, J. M. Calandrino, and J. H. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *IEEE Real-Time Systems Symposium*, pages 157–169, 2008.

[7] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–126, December 2006.

[8] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proceedings of the conference on Automation and Test in Europe*, pages 10190–10195, March 2003.

[9] U. C. Devi and J. H. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. *The Journal of Real-Time Systems*, 38(2):133–189, 2008.

[10] H. Leontyev and J. H. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *The Journal of Real-Time Systems*, 44(1):26–71, February 2010.

[11] A. Mills and J. H. Anderson. A stochastic framework for multiprocessor soft real-time scheduling. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2010. To appear.

[12] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, pages 476–490, January 2001.

[13] M. Spuri and G. C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 2–11, December 1994.

[14] UNC Real-Time Group. LITMUS$^{\mathrm{RT}}$ project. http://www.cs.unc.edu/˜anderson/litmus-rt/.

[15] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 239–243, December 2007.