Multicore Operating-System Support for Mixed Criticality*

James H. Anderson, Sanjoy K. Baruah, and Björn B. Brandenburg
The University of North Carolina at Chapel Hill

Abstract

Ongoing research is discussed on the development of operating-system support for enabling mixed-criticality workloads to be supported on multicore platforms. This work is motivated by avionics systems in which such workloads occur. In the mixed-criticality workload model that is considered, task execution costs may be determined using more-stringent methods at high criticality levels, and less-stringent methods at low criticality levels. The main focus of this research effort is devising mechanisms for providing "temporal isolation" across criticality levels: lower levels should not adversely "interfere" with higher levels.

1 Introduction

The evolution of computational frameworks for avionics systems is being impacted by two trends. First, such frameworks are becoming increasingly complex. While this is generally true, a particularly good case in point is next-generation unmanned systems, where complex decision-making software must be used to perform pilot functions. Second, there is increasing pressure to host such complex systems using hardware platforms that require less weight and volume.

These trends are occurring at a time when multicore chip designs have replaced single-core designs in the product lines of most chip manufacturers. This development is profound, as it means that multiprocessors are now a "common-case" platform that will be used across a wide range of application domains. In the avionics domain, multicore platforms offer the potential of enabling greater computational capabilities in less space. Unfortunately, currently-used avionics operating systems do not provide sufficient functionality to allow applications of varying criticalities to be co-hosted on a multicore platform. In this paper, we discuss ongoing research that is being conducted to address this issue.

*Work supported by IBM, Intel, and Sun Corps.; NSF grants CNS 0834270, CNS 0834132, and CNS 0615197; and ARO grant W911NF-06-1-0425.

Different criticality levels provide different levels of assurance against failure. In current avionics designs, highly-critical software is kept physically separate from less-critical software. Moreover, no currently deployed aircraft uses multicore processors to host highly-critical tasks (more precisely, if multicore processors are used, and if such a processor hosts highly-critical applications, then all but one of its cores are turned off). These design decisions are largely driven by certification issues. For example, certifying highly-critical components becomes easier if potentially adverse interactions among components executing on different cores through shared hardware such as caches are simply "defined" not to occur. Unfortunately, hosting an overall workload as described here clearly wastes processing resources.

In this paper, we propose operating-system infrastructure that allows applications of different criticalities to be co-hosted on a multicore platform. Our specific focus is ensuring real-time correctness. We assume that criticalities are defined as in the RTCA DO-178B software standard, which specifies five criticality levels, as listed in Table 1 (a more thorough discussion of criticalities is given later). We further assume that the workload to be supported is specified as a collection of periodic tasks with harmonic periods.

The OS development platform used in our work is a UNC-produced open-source system called LITMUS$^{\mathrm{RT}}$ (**LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems) [7, 9, 12, 14, 15, 26]. LITMUS$^{\mathrm{RT}}$is an extension of Linux (currently, version 2.6.24) that allows different (multiprocessor) real-time scheduling algorithms to be linked at runtime as plug-in components. Relevant details regarding LITMUS$^{\mathrm{RT}}$, as well as our rationale in choosing it as a development platform, are given later. The main focus of this paper is to report on the status of our implementation efforts in extending LITMUS$^{\mathrm{RT}}$ to support multi-criticality workloads on multicore platforms. We also more broadly discuss some of the challenges inherent in supporting such workloads.

The rest of this paper is organized as follows. In Sec. 2, we provide relevant background material and discuss related work. In Sec. 3, we describe the LITMUS$^{\mathrm{RT}}$-based implementation efforts mentioned above. In Sec. 4, we

Level	Failure Condition	Interpretation
A	Catastrophic	Failure may cause a crash.
B	Hazardous	Failure has a large negative impact on safety or performance, or reduces the ability of the crew to operate the plane due to physical distress or a higher workload, or causes serious or fatal injuries among the passengers.
C	Major	Failure is significant, but has a lesser impact than a Hazardous failure (for example, leads to passenger discomfort rather than injuries).
D	Minor	Failure is noticeable, but has a lesser impact than a Major failure (for example, causing passenger inconvenience or a routine flight plan change).
E	No Effect	Failure has no impact on safety, aircraft operation, or crew workload.

Table 1: DO-178B is a software development process standard, *Software Considerations in Airborne Systems and Equipment Certification*, published by RTCA, Inc. The United States Federal Aviation Authority (FAA) accepts the use of DO-178B as a means of certifying software in avionics applications. RTCA DO-178B assigns criticality levels to tasks categorized by effects on commercial aircraft.

discuss a number of future extensions to the system being built. We conclude in Sec. 5.

2 Background

In the following subsections, we present relevant background information on multiprocessor real-time scheduling, multi-criticality scheduling, and scheduling approaches for ensuring temporal isolation. We also present a brief overview of the LITMUS$^{\mathrm{RT}}$ system being used in our research.

2.1 Multiprocessor Real-Time Scheduling

When designing a real-time application, the goal is to produce a *schedulable* system, *i.e.*, one whose timing constraints can be guaranteed. Timing constraints are usually expressed using deadlines. In this paper, we consider a well-studied task model in which such notions arise, namely the *periodic task model* [20]. In this model, the scheduled system is comprised of a collection of recurring sequential tasks. Each such task T releases a succession of *jobs* and is defined by specifying a *period* $T.p$ and an *execution cost* $T.e$. Successive jobs of T are released every $T.p$ time units, starting at time 0, and one that is released at time t has a *deadline* at time $t + T.p$. Also, each such job executes for at most $T.e$ time units and its required processing time should be allocated between its release and deadline. In this paper, we make the simplifying assumption that task periods are *harmonic*. This means that, for any two tasks T and U, if $T.p < U.p$, then $U.p$ is an integer multiple of $T.p$. In avionics systems, task periods are often harmonic.

Real-time schedulability. In the periodic model, the definition of the term "schedulable" depends on whether deadlines are hard or soft. In a *hard real-time* (*HRT*) system, deadlines can never be missed, while in a *soft real-time* (*SRT*) system, some misses are tolerable. SRT schedulability can be defined in different ways; we assume here that a SRT system is schedulable if deadline tardiness is bounded (by a reasonably small constant). In determining schedulability (hard or soft), the processor share required by each task T is of importance. The share required by T is given by the quantity $T.u = T.e/T.p$, which is called its *utilization*.

With regard to scheduling, two kinds of algorithms actually are of interest: algorithms used within an OS to make scheduling decisions (of course) and algorithms used to test schedulability (*i.e.*, timing correctness). The latter algorithm is necessarily dependent on the former. In the multiprocessor case, a scheduling algorithm may follow a *partitioning* or *global scheduling* approach (or some combination of the two): under partitioning, tasks are statically assigned to processors, while under global scheduling, they are scheduled from a single run queue and may migrate among processors. An example of a partitioning algorithm is *partitioned EDF* (P-EDF), which uses the uniprocessor *earliest-deadline-first* (EDF) algorithm as the per-processor scheduler. An example of a global algorithm is *global EDF* (G-EDF), under which tasks are EDF-scheduled using a single priority queue. In this paper, P-EDF and EDF are considered extensively. In addition, we consider the *cyclic executive* approach, wherein scheduling decisions are statically pre-determined offline and specified in a dispatching table [2]. If task periods are harmonic, then the required dispatching table requires pseudo-polynomial space. In avionics and other safety-critical domains, the cyclic executive approach is often preferred, because dispatching tables (once constructed) are relatively easy to certify as correct. In the case of cyclic executives, we assume partitioned scheduling (one dispatching table per processor).

Global vs. partitioned scheduling. Generally speaking, partitioning algorithms are preferable for HRT workloads, and global algorithms are preferable for SRT workloads. If all deadlines are hard, or if partitioning or static-priority scheduling is used, then restrictive caps on overall utilization (of up to *half* the system's processing capacity) must be enforced to ensure that real-time constraints are met [16].[1] Because such caps are unavoidable in the HRT case, partitioning approaches tend to be better because they have lower run-time overheads than global approaches [13, 15]. In contrast, a wide variety of dynamic-priority, global algorithms, including G-EDF, are capable of ensuring bounded deadline tardiness on an m-processor platform for any periodic task system with total utilization at most m [17, 19].

2.2 Multi-Criticality Scheduling

In this paper, we consider the problem of scheduling tasks of different criticalities (in our case, as specified by the RTCA DO-178B standard summarized in Table 1) on a multicore platform. This problem has been considered before in the context of uniprocessor systems. The conventional approach to implementing such a system has been to assign greater priority to higher-criticality tasks. However, Vestal observed that such an approach results in severe resource under-utilization, and in fact showed that existing scheduling theory cannot adequately address multiple criticality requirements [27]. He proposed the *multi-criticality task model*, under which multiple worst-case execution time (WCET) estimates, each made at a different level of assurance (*i.e.*, with a different degree of confidence), may be specified for each task. Since higher criticality levels require greater levels of assurance and hence more over-provisioning of computing capacity, the WCET estimate at higher criticality levels is typically far more pessimistic (*i.e.*, greater) than the WCET estimate for the same task at lower criticality levels (and this is even more likely on a multicore platform, where interactions among cores through shared hardware are difficult to predict). In determining whether tasks of a particular criticality meet their timing constraints under a specified scheduling discipline, the WCET estimates of all tasks that are used are those determined at the level of assurance corresponding to this given criticality level. Since such validation is done at system *design time*, this implies that the excess capacity that must be over-provisioned for high-criticality tasks gets "reclaimed" at system design time itself, and can be used for guaranteeing deadline compliance of lower-criticality tasks. The multi-criticality task model is remarkably powerful and expressive. For instance, in order to obtain a WCET bound for a task at a higher level of assurance (*i.e.*, at a degree of confidence corresponding to a higher criticality), it is typically necessary to limit the kinds of programming constructs that may be used in that task (loops, for example, may be required to have static bounds). However, low-criticality tasks should not be subject to these same stringent programming restrictions. By allowing the corresponding (high-assurance) WCETs for these low-criticality tasks to be set to infinity, the model does not forbid low-criticality tasks from using these "unsafe" constructs.

In other work on this topic, Baruah and Vestal proposed new algorithms for scheduling systems of multi-criticality tasks on preemptive uniprocessors [6]. These algorithms exploit knowledge of WCET estimates at different assurance levels to maximize processor utilization. Baruah and Vestal proved that their algorithms are capable of successfully scheduling a larger class of task systems than is the case with prior algorithms.

2.3 Ensuring Temporal Isolation

A key requirement in multi-criticality scheduling is that tasks of lower criticality should not adversely affect tasks of higher criticality. This requirement is very related to a concept called *temporal isolation*. In a system that provides temporal isolation, the timing requirements of different components or sub-systems can be validated *independently*. Temporal isolation can be ensured by supporting within the OS a *container* abstraction. For the purposes of this paper, a container is a task group that is managed in a way that isolates the grouped tasks from the rest of the system. In the real-time-systems community, this notion of a "container" is more commonly called a "server." Server-based abstractions were first considered in the context of uniprocessor systems to allow the timing correctness of subsystems to be validated independently. Each such subsystem (a set of tasks) is assigned to a "server task." A two-level scheduling approach is then used, where at the top level, server tasks are scheduled, and at the next level, the servers themselves schedule their constituent tasks. A server is provisioned by assigning it a designated utilization, or *resource budget*. Some of the more well-known server schemes that function in this way include *constant bandwidth servers* [1], *total bandwidth servers* [24], and *resource kernels* [22].

Uniprocessor systems have been the main focus of server-related research. However, resource kernels have

[1] A category of global algorithms exists called *Pfair algorithms* [3, 25] for which this statement is not true. Pfair algorithms are capable of scheduling any m-processor HRT system if total utilization is at most m. However, under Pfair algorithms, tasks are preempted and potentially migrated frequently, so runtime overheads can be high.

been implemented on multiprocessor platforms, and a few other multiprocessor schemes have been proposed as well [4, 5, 21, 24]. To the best of our knowledge, multi-criticality scheduling (our focus) has not been considered before in work on server-based approaches.

2.4 LITMUS$^{\mathrm{RT}}$

As noted earlier, the LITMUS$^{\mathrm{RT}}$ system being used in our research is an extension of Linux 2.6.24 that allows different (multiprocessor) real-time scheduling algorithms to be linked at runtime as plug-in components. Several such algorithms have been implemented, including the P-EDF and G-EDF algorithms considered earlier. To the best of our knowledge, LITMUS$^{\mathrm{RT}}$ is the only published system wherein global real-time scheduling algorithms (a major focus our work) are implemented in a real OS.

As its name suggests, LITMUS$^{\mathrm{RT}}$was developed as a *testbed* for experimentally evaluating real-time resource-allocation policies. In this capacity, LITMUS$^{\mathrm{RT}}$ has proved to be very valuable: it has been used to explore fundamental implementation-oriented tradeoffs involving multiprocessor real-time scheduling algorithms [13, 15] and synchronization methods [8, 11, 10, 14], to evaluate scheduling policies that can adapt to workload changes [7] or support systems with mixed HRT, SRT, and non-real-time components [9], and to examine scalability issues on large (by today's standards) multicore platforms [13].

Due to various sources of unpredictability within Linux (such as interrupt handlers and priority inversions within the kernel), it is not possible to support true HRT execution in Linux,[2] and hence in LITMUS$^{\mathrm{RT}}$. Given these limitations, LITMUS$^{\mathrm{RT}}$would not be a viable OS for domains like avionics, where safety-critical components exist. However, the primary goal of our current research is to explore various implementation-related tradeoffs that arise when supporting multi-criticality workloads, and LITMUS$^{\mathrm{RT}}$is more than adequate for this purpose. Additionally, LITMUS$^{\mathrm{RT}}$provides several scheduling-related debugging and tracing tools that greatly aid development efforts. Once we have developed a good understanding of implementation-oriented tradeoffs that impact multi-criticality scheduling, we hope to port the scheduling methods we produce to an avionics-suitable real-time operating system.

[2]By "Linux," we mean modified versions of the stock Linux kernel with improved real-time capability, not paravirtualized variants such as RTLinux[28] or L^{4}Linux[18], where real-time tasks are not actually Linux tasks. Stronger notions of HRT can be provided in such systems, at the expense of a more restricted and less familiar development environment.

3 Proposed Architecture

We plan to support harmonic multi-criticality workloads with a two-level hierarchical scheduling framework. The key idea is to encapsulate, on each core, each of the five criticality classes in separate containers (C_A–C_E), and to use intra-container schedulers appropriate for each level (*e.g.*, cyclic executives are a likely choice for C_A and C_B, whereas G-EDF may be a favorable choice for C_D). Temporal isolation is achieved by statically assigning higher scheduling priority to higher criticality classes (*i.e.*, C_A has the highest priority). While simple in concept, some complications (discussed next) arise, which differentiate the proposed architecture from prior work on hierarchical scheduling in real-time systems.

Container parameters. For each container C (five per core), we need to specify an *allocation window* $C.p$ and a *budget* $C.b$: the budget is consumed when client tasks execute and is replenished at time zero and every $C.p$ time units thereafter. A major complication in most container schemes is the need to retain budget when tasks are inactive in order to handle job arrivals between replenishment points. However, since task periods are harmonic, this problem can be avoided by choosing $C.p = \min\{T.p\}$ for all containers—by definition, jobs only arrive at times that are multiples of $C.p$ and budget retention becomes unnecessary.

Choosing a budget is more complicated. Obviously, $C.b$ must match or exceed the execution requirement of all clients during $C.p$. However, recall that in a multi-critically workload, each task T has an execution requirement and utilization at every criticality level ($T.e_A$–$T.e_E$ and $T.u_A$–$T.u_E$). Hence, we specify for each container five budgets $C.b_A$–$C.b_E$ that are consumed *in parallel*, where each budget corresponds to the execution requirements of its clients at the respective criticality level. Separate per-level budgets are required for analysis purposes, but it may be beneficial to track them at runtime, too, as discussed below.

Example. Consider the periodic, harmonic multi-criticality workload specified in Table 2 consisting of ten tasks (two per criticality level). For the sake of simplicity, we consider only one partition (*i.e.*, one processor) in this example and assume that P-EDF is used as the intra-container scheduler for each level. The minimum period (and hence the allocation window $C.p$) is ten time units. Based on the execution requirements of the tasks at each level, container budgets have been derived as given in Table 3.

Note that T_1 and T_2 alone fully utilize the processor assuming level-A execution time requirements. Hence, without the proposed architecture, only those two tasks could be

	Crit.	$T.p$	$T.e_A$	$T.u_A$	$T.e_B$	$T.u_B$	$T.e_C$	$T.u_C$	$T.e_D$	$T.u_D$	$T.e_E$	$T.u_E$
T_1	A	10	5	0.5	3	0.3	2	0.2	1	0.1	1	0.1
T_2	A	20	10	0.5	6	0.3	4	0.2	3	0.15	2	0.1
T_3	B	40	–	–	8	0.2	2	0.05	2	0.05	2	0.05
T_4	B	20	–	–	4	0.2	3	0.15	2	0.1	2	0.1
T_5	C	10	–	–	–	–	2	0.2	1	0.1	1	0.1
T_6	C	20	–	–	–	–	4	0.2	2	0.1	1	0.05
T_7	D	20	–	–	–	–	–	–	4	0.2	2	0.1
T_8	D	10	–	–	–	–	–	–	2	0.2	1	0.1
T_9	E	20	–	–	–	–	–	–	–	–	4	0.2
T_{10}	E	40	–	–	–	–	–	–	–	–	4	0.1
Σ				1.0		1.0		1.0		1.0		1.0

Table 2: Parameters of the task set shown in Fig. 1. Note that each task has a maximum execution time specified for its own and lesser criticality levels. Execution time parameters for higher criticality levels are not required due to temporal isolation. Note also that the processor is fully utilized at each criticality level—capacity lost due to pessimistic worst-case execution time estimates at high criticality levels has been reclaimed at lower criticality levels.

provisioned on the processor. However, if temporal isolation is guaranteed, then each criticality level's schedulability can be tested individually. In this case, all ten tasks can be provisioned since the total utilization is less than one for each criticality level, as explained in detail below. Intra-container schedules illustrating that all deadlines are met at each criticality level if no overruns occur at higher levels are shown in Fig. 1. Note that, even though all deadlines are met in the depicted schedules, schedulability is actually a rather deep issue, as discussed in detail below.

In inset (a), level-A temporal correctness is shown: since C_A has the highest (static) priority, lower-criticality work cannot interfere with T_1 and T_2, which fully utilize the processor and meet all of their deadlines.

Level-B correctness is shown in inset (b). As can be seen in Table 3, C_A is expected to require at most six time units in every allocation window assuming level-B execution requirements. This leaves four time units every ten time units for level-B work to complete, which suffices for T_3 and T_4 to meet all their deadlines. However, if C_A exceeds its level-B estimate in any window (*i.e.*, C_A requires more than $C_A.b_B$ time units of service), then T_3 and T_4 may (of course) incur deadline misses.

The level-C intra-container schedule is shown in inset (c). Assuming level-C estimates, C_A and C_B require are total of six time units per allocation window (see Tabel 3), which, similarly to C_B's schedule, leaves four time units per allocation window for level-C work. Note that $T_5.p = 10$, hence a job of T_5 has a deadline at every allocation window boundary. In order for T_5 to meet all of its deadlines, each job of T_6 has to execute in two allocation windows—as it does, assuming P-EDF scheduling and the absence of higher-level overruns.

C_D's and C_E's intra-container schedules are similar to those just discussed: assuming level-D (level-E) execution requirements for higher-criticality containers in each window, there is enough left-over capacity such that all level-D (level-E) tasks meet all of their deadlines. Note that the intra-container schedules depicted in insets (a)–(e) cannot occur simultaneously—none of the schedules leaves sufficient capacity to accommodate all lower-criticality work. This corresponds to the multi-criticality notion of correctness: each schedule depicts the *expected* worst case according to the corresponding level's execution time analysis method (assuming higher-criticality containers do not overrun).

Schedulability. Since all deadlines were met in the above example, it is tempting to assume that the described system is indeed HRT schedulable, *i.e.*, that each level-X task *always* meets all of its deadlines if each task T of equal or higher criticality executes for at most $T.e_X$ time units. However, this is unfortunately not the case. In fact, in our example, HRT correctness can only be guaranteed for levels A and B, and only SRT correctness at lower criticalities.

This is shown in Fig. 2, which depicts a schedule of all tasks of criticality C and higher (T_1–T_6) across four allocation windows. Since, in the depicted schedule, each task T's jobs require exactly $T.e_C$ execution time, each level-C job should meet all of its deadlines (corresponding to inset (c) in Fig. 1). This, however, is not the case: T_5 misses a deadline both at times 10 and 30, and T_6 misses a deadline at time 20.

Why are deadlines met in Fig. 1(c), and yet some missed in Fig. 2? The reason is that, in Fig. 2, both C_A (*i.e.*, T_1 and T_2) and C_B (*i.e.*, T_3 and T_4) overran their level-C budget in the first and third allocation windows, hence level-C jobs were starved in these windows. In contrast, in Fig. 1(c), it is assumed that the higher-criticality containers never overrun their level-C budget. Unfortunately, this requires "clairvoy-

	$C.b_A$	$C.u_A$	$C.b_B$	$C.u_B$	$C.b_C$	$C.u_C$	$C.b_D$	$C.u_D$	$C.b_E$	$C.u_E$
C_A	10	1.0	6	0.6	4	0.4	2.5	0.25	2	0.2
C_B	–	–	4	0.4	2	0.2	1.5	0.15	1.5	0.15
C_C	–	–	–	–	4	0.4	2	0.2	1.5	0.15
C_D	–	–	–	–	–	–	4	0.4	2	0.2
C_E	–	–	–	–	–	–	–	–	3	0.3

Table 3: Container parameters for the example shown in Fig. 1. The window size $C.p$ of the task set is $\min\{T.p\} = 10$.

ant" scheduling: in order to (safely) restrict C_A and C_B to their level-C budgets in the first window, it must be known that all the jobs of T_1–T_4 do not exceed their level-C execution time estimate *before* they complete.

The fundamental problem exposed by this example is that a level-X job T_h can excessively delay a lower-criticality level-Y job T_l with $T_l.p < T_h.p$ even if T_h executes for at most $T_h.e_Y$ time units because higher-criticality work is "shifted" from allocation windows past T_l's deadline to allocation windows prior to T_l's deadline—in effect, T_h uses a share of the processor greater than $T_h.u_Y$ while T_l must complete.

Note that the problem does not occur if periods are non-decreasing across criticality levels: if P-EDF is used as the intra-container scheduler for level X, then all level-X tasks will meet their deadlines if, on each processor P, the sum of the level-X utilizations of all tasks of criticality X and all higher-criticality containers does not exceed one and the shortest period of any level-X task assigned to P is at least as long as the longest period of any higher-criticality task assigned to P. Hence, in our example, all deadlines will be met in C_B because

$$T_3.u_b + T_4.u_b + C_A.u_B \leq 1$$

and

$$\min(T_3.p, T_4.p) \geq \max(T_1.p, T_2.p).$$

The classic EDF utilization bound [20] applies in the case of non-decreasing deadlines (across criticality levels) even though higher-criticality containers have a statically higher priority because the resulting schedule is *always* a valid EDF schedule, too. This is because, by assumption, any higher-criticality job that causes delays has a deadline no later than the earliest deadline of any job in the container under analysis, and EDF allows for arbitrary tie breaking (in the resulting schedule, ties are broken in favor of the containers in order of decreasing criticality). This analysis method was used before—and is explained in detail—in [9].

If periods increase across criticality levels (*e.g.*, levels C–E in Table 2), then there two options to transform the task set. Either lower-criticality work with short periods can be promoted to a sufficiently high criticality, or higher-criticality jobs with long periods can be split into multiple sub-jobs with short periods. The former may be not a viable option for verification reasons—*e.g.*, if unsafe constructs such as unbounded loops are used in a level-E task, then it likely cannot be promoted to level-A criticality, even if it has the minimum period in the task set. The latter approach has the downside that it requires tasks to be modified, and that analyzing sub-jobs may yield more pessimistic execution time requirements than analyzing complete jobs would have. However, we note that loss of cache affinity between sub-jobs is likely not a problem at high criticality levels since caches are assumed to be ineffective in most existing timing-analysis approaches appropriate for such levels. We are currently investigating these schedulability issues in greater depth.

If G-EDF is used as an intra-container scheduler, then timing correctness can be established using existing reduced-supply schedulability tests. Due to space constraints, we omit further details, and refer the reader to [19, 23] instead.

Overruns. Budget overruns, by design, may occur in multi-criticality systems since execution requirements are estimated using less-pessimistic methods at lower criticality levels, and may even be empirically-determined average execution costs at level-E criticality.

For example, in a schedulable multi-criticality system, all level-D tasks are guaranteed to meet their deadlines if *no* task (of any criticality) overruns its level-D execution time estimate. However, it is reasonable to expect that tasks of criticality C and higher will (occasionally) overrun their level-D estimate. In fact, if tasks are provisioned on an average-case basis at lower criticality levels, then such tasks may often overrun. Hence, multi-criticality systems are likely to benefit from the use of *slack scheduling* methods, which attempt to ameliorate the negative impact of overruns by re-distributing any unused capacity at runtime. In our setting, slack is generated at a given criticality level X when containers of higher criticality than X require less than their level-X execution time estimate, *i.e.*, level-X slack is available if the processing of level-X jobs can be safely postponed because higher-criticality work required less capacity than expected. Reassigning slack to lower-criticality work, however, requires accurate tracking of budgets at *all* criticality levels, as the following example

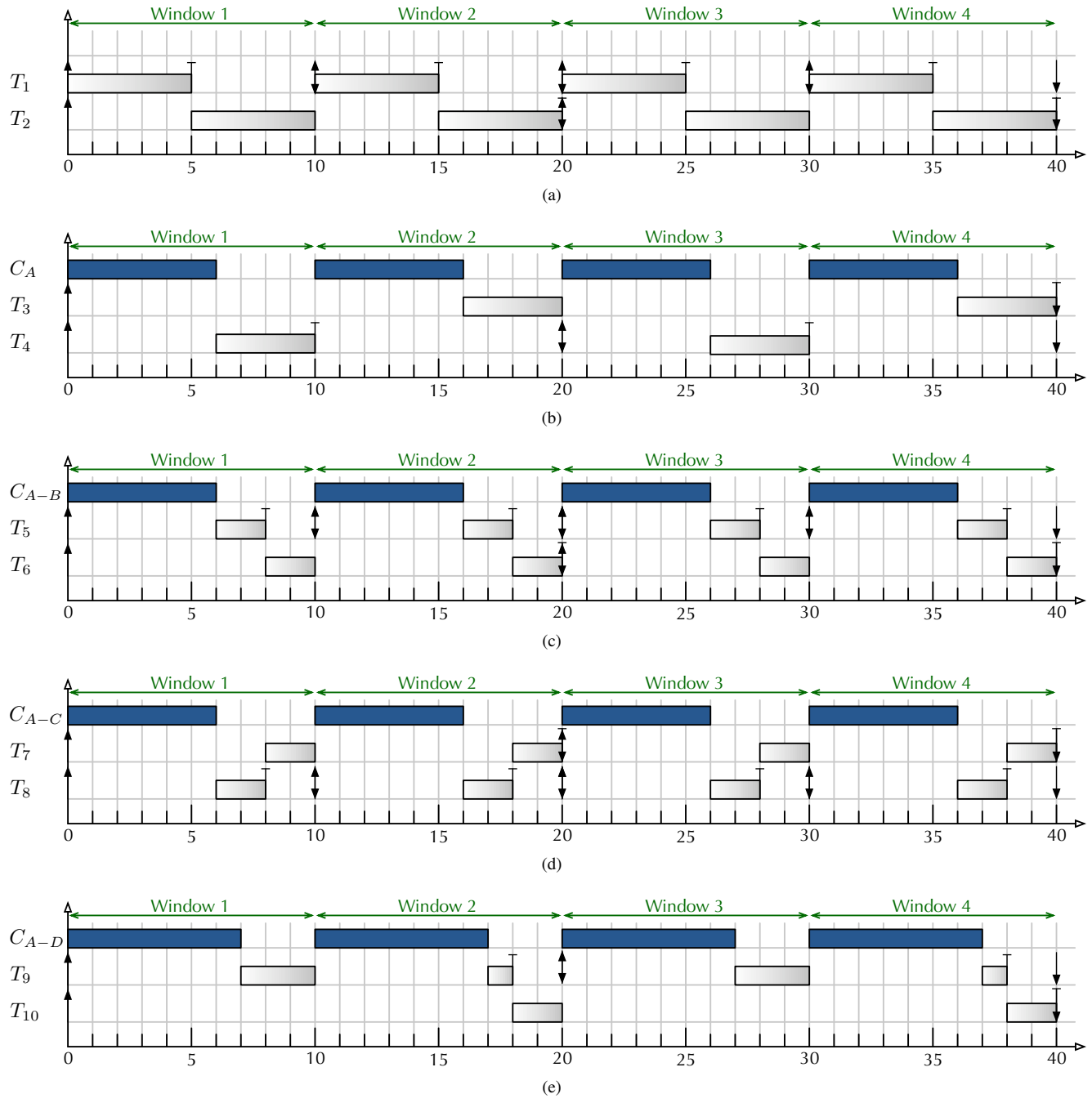

Figure 1: Schedules showing four allocation windows for **(a)** level A, **(b)** level B, **(c)** level C, **(d)** level D, and **(e)** level E criticality assuming that higher-criticality containers do not overrun their respective budgets. For example, inset (b) depicts the schedule of both level-B tasks (T_3, T_4) that results when each such task T requires only $T.e_B$ execution time and the level-A container executes for $C_A.b_B$ time units per allocation window. Higher-priority containers (if any) execute at the beginning of each allocation window. The schedule repeats after four windows since $\max\{T.p\} = 40$. (Up-arrows indicate job releases, down-arrows indicate deadlines, and T-shaped arrows indicate job completions.)

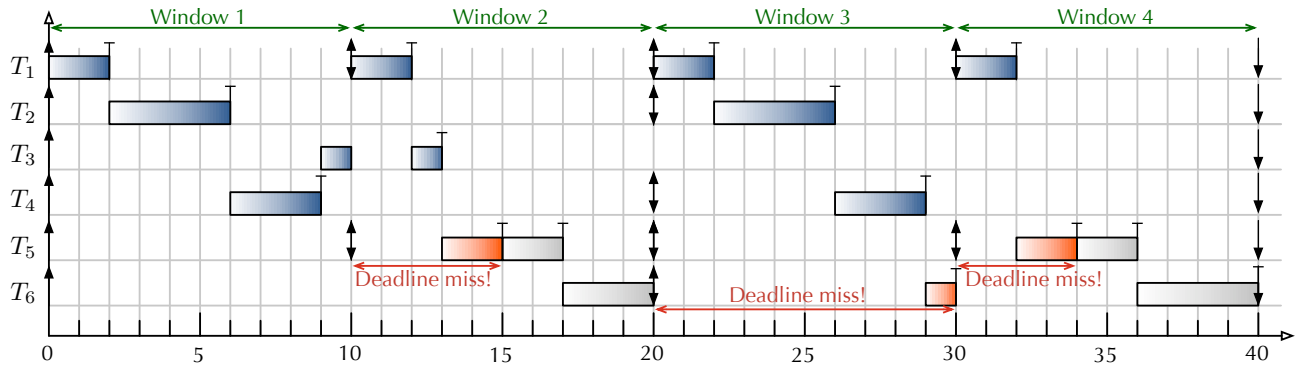

Figure 2: An example schedule corresponding to inset (c) of Fig. 1 that demonstrates that both level-C tasks (of the task system given in Tabel 2) can incur deadline misses even if each task T of criticality C and higher requires at most $T.e_C$ time units. Note that maximum tardiness is bounded since the schedule repeats after four windows. Hence, SRT correctness can be guaranteed for level C.

illustrates.

Example. Fig. 3 depicts three examples of slack scheduling corresponding to inset (c) of Fig. 1. Inset (a) shows how slack scheduling can be used to reduce the negative impact of budget overruns. In this case, C_A required six time units in each of the first two allocation windows, which exceeds the level-C estimate of $C_A.b_C = 4$, but not the level-B estimate of $C_A.b_B = 6$. Consequently, the level-B container executes as expected, but T_6, a level-C task, misses a deadline at time 20. However, C_A underruns its level-B budget at time 21 by finishing five time units early. This underrun creates dynamic slack that is reassigned to T_6, which causes it to finish earlier than it would have otherwise. T_6's deadline miss does not imply that the system behaved incorrectly because C_A exceeded its level-C estimate.

Inset (b) shows how slack scheduling can undermine temporal isolation if budgets are not tracked carefully. In this case, C_A and C_B *never* exceed their level-C budgets—hence, all level-C tasks should meet their deadlines. Unfortunately, T_6 misses a deadline again at time 20. This is because three units of slack were assigned to a level-D job at time three. While there were indeed three units of level-B slack available, there was only *one* unit of level-C slack available. This example shows that multiple budgets must be maintained at runtime.

Finally, inset (c) shows the same scenario as in inset (b) with correct budget tracking. In this case, only one time unit can be reassigned to level-D work at time three. Further allocations are made at times 23 and 33 as more level-C slack becomes available.

Implementation considerations. There are three major implementation questions that will have to be addressed: **(i)** how to realize containers; **(ii)** how to implement cyclic executives; and **(iii)** how (and if) to implement slack scheduling. We discuss these issues from a LITMUS$^{\mathrm{RT}}$-centric perspective next and consider supporting avionics-capable RTOSs in Sec. 4.

Regarding (i), current versions of Linux provide a full-fledged container abstraction called *cgroups*. With cgroups, tasks can be organized into an arbitrarily-deep hierarchy of containers. However, cgroups lack support for several features vital to multi-criticality systems. First, Linux only supports static-priority intra-container schedulers for real-time tasks[3] and has only rudimentary support for tracking container budgets. Neither tracking budgets at multiple criticality levels nor redistributing unused capacity at runtime is supported. Hence, we have two avenues for realizing containers: either we establish a notion of a container besides cgroups that is specific to LITMUS$^{\mathrm{RT}}$ and real-time tasks (as done previously to a limited exent in [9]), or we extend cgroups to support pluggable intra-container schedulers and accurate budget tracking and slack redistribution. While extending cgroups is certainly the much more general solution, it remains to be seen whether cgroups can be used for multi-criticality workloads with acceptable overheads—since our proposed architecture has only modest requirements in terms of container features (*e.g.*, no nesting, no migrations of containers), it may be preferable to use a special-purpose implementation in practice. Also, when ported to an avionics RTOS, Linux capabilities may be irrelevant.

Concerning (ii), cyclic executives are commonly used in embedded systems with a single, shared address space. In such systems, a cyclic executive can be trivially implemented as a "master loop" that repeatedly selects and executes procedures that represent jobs based on a predetermined and hard-coded dispatch table. This approach, however, does not work without modification in an OS with address space protection such as LITMUS$^{\mathrm{RT}}$. The tradi-

[3] A second scheduler called *CFS* is available to non-real-time tasks.

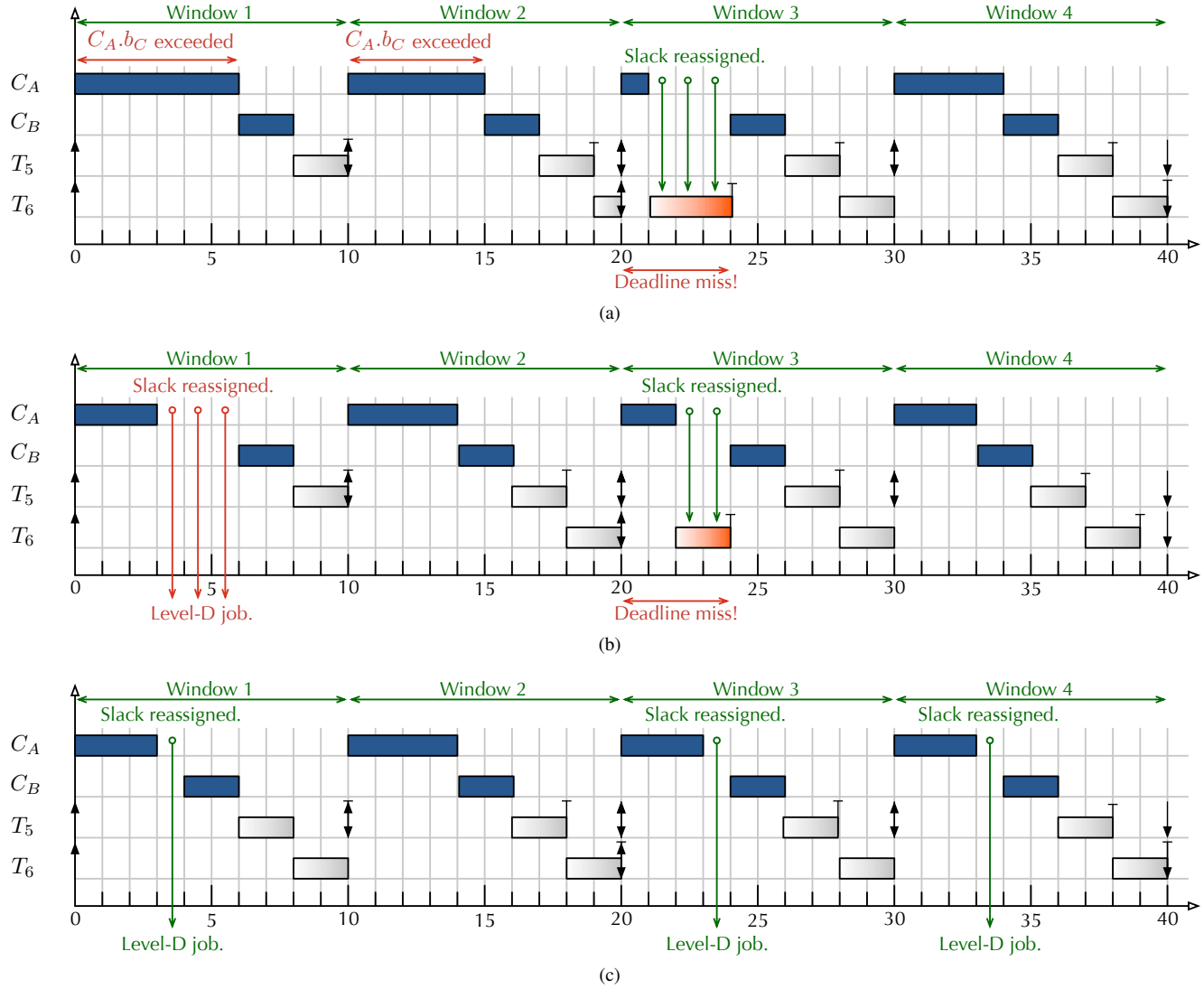

Figure 3: Three illustrations of slack scheduling corresponding to inset (c) of Fig. 1. **(a)** Example of correct slack scheduling: C_A exceeds the level-C execution time estimate $C_A.b_C = 4$ twice, which causes T_6 to miss a deadline at time 20. Dynamic slack is reassigned to complete the tardy job at times 21–24 to expedite its completion. **(b)** Example of incorrect slack scheduling: T_6 misses a deadline even though neither $C_A.b_C$ nor $C_B.b_C$ is exceeded. The deadline is missed because there was only one time unit of level-C slack available at time three, but three time units were reassigned to level-D work. Consequently, temporal isolation was violated as level-D work interfered with the level-C task T_6. **(c)** Example of correct slack scheduling: The same scenario as in (b), but only one time unit (the amount of level-C slack available) is reassigned at time three. Hence, temporal isolation is maintained and no deadline is missed.

tional way of implementing cyclic executives could be employed in LITMUS$^{\mathrm{RT}}$ (and other systems with separate address spaces) by realizing all real-time tasks of the same criticality in one binary, *i.e.*, the problem is trivial if only one task is presented to the OS. However, it is very desirable to make use of separate address spaces if available because the verification of logical correctness becomes much easier in this case (due to a reduction in size of the state space that must be considered), and any run-time errors are likely less catastrophic (*e.g.*, the chances of aircraft survival are probably gravely worse if all level-A tasks fail simultaneously than if only a single level-A task were to fail). Further, if cyclic executive activity is hidden from the OS, then slack scheduling becomes much more difficult. Hence, we would like to support cross-address-space cyclic executives at the OS level in LITMUS$^{\mathrm{RT}}$.

We already have gathered some experience with (iii). In prior work, slack scheduling has been shown to greatly reduce response times of best-effort jobs in practice [9]. For the reasons outlined above, we believe that slack scheduling could also have a great impact in multi-criticality systems. However, several complications arise. Due to the requirement to track multiple budgets, slack scheduling is likely to become much more difficult and hence time consuming. Runtime overheads will thus likely be non-negligible and could even negate possible performance gains. Further, it is unclear at which granularity slack should be tracked—if the granularity is too low, then updating budget-tracking data structures could likely become a bottleneck in global schedulers due to contention. These issues can only be investigated by benchmarking an actual (prototype) implementation on real hardware.

4 Research Directions

While the proposed architecture is simple in structure, it raises a number of interesting research questions.

For example, while cyclic executives are likely the only viable choice for scheduling level-A (and maybe even level-B) tasks due to verification reasons, more flexible choices are likely preferable for less-critical levels. At the same time, overhead considerations (*e.g.*, kernel instruction and data cache footprints, kernel size, *etc.*) may place strict limits on scheduler implementations. Which scheduling algorithms are the "best" candidates, both in terms of overheads and the ease of verification, for typical avionics workloads? Is it feasible to support pluggable intra-container schedulers in a real-world system, and are the gains worth the added implementation complexity?

While multiprocessor real-time scheduling algorithms have received much attention in isolation, an important question raised by multi-criticality workloads is how scheduling algorithms at each level affect each other. For example, if G-EDF is employed for both levels C and D on all cores, then it is not clear how to optimally allocate container budgets. Should capacity requirements be spread out among cores (*e.g.*, to improve parallelism), or should some cores be allocated exclusively to level-D processing and others to level-C processing (*e.g.*, to reduce tardiness)?

Obviously, another major open question is how to address overruns at each level. For example, if a level-A task overruns such that neither all level-B nor all level-C tasks can complete by their respective deadlines, is it preferable to abort level-C tasks and have level-B tasks finish on time or is it better to accept tardiness for both level-B and level-C tasks? How can slack scheduling be used to reduce the impact of overruns?

Future extensions. Once we have a working prototype implementation based on LITMUS$^{\mathrm{RT}}$, we plan to extend our research into multi-criticality systems in three directions.

First, we would like to investigate the feasibility of integrating our approach with an ARINC-653-compliant[4] RTOS in joint work with industry colleagues. We expect this study both to serve as a "reality check" for the proposed architecture and our implementation techniques and to yield recommendations for the development of future standards for the implementation of avionics software on multicore platforms.

Second, we intend to relax some of the requirements imposed on real-time tasks. Specifically, we plan to support synchronization to enable resource-sharing among tasks. Some of the questions that will arise in this context are how critical sections can be integrated with the multi-criticality model (sharing resources across criticality boundaries is especially troublesome) and how blocking analysis impacts temporal isolation guarantees. Further, if critical sections are long or allocation windows short, then the issue of cross-allocation-window critical sections arises. This will likely require the development of a more refined method of choosing allocation windows.

Finally, we would like to extend the proposed architecture to support non-harmonic, sporadic, and aperiodic workloads. This will require the development of new budget-retaining techniques and will impact the choice of allocation windows.

5 Concluding Remarks

We have illustrated the importance of supporting multi-criticality workloads on multicore platforms in the context of avionics and the RTCA DO-178B software standard. As a first step, we have proposed an architecture to support the special case of periodic task systems with harmonic periods and outlined some of the development challenges inherent in the proposed container-based system in the context of LITMUS$^{\mathrm{RT}}$, a UNC-produced real-time extension of Linux. Finally, we have illustrated numerous avenues for future research based on the proposed architecture and future exentensions.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th*

[4] The *Avionics Application Standard Software Interface* (ARINC-653) standard specifies isolation requirements for avionics platforms.

IEEE Real-Time Systems Symposium, pages 3–13, December 1998.

[2] T. Baker. The cyclic executive model and ADA. *The Journal of Real-Time Systems*, 1:120–129, 1989.

[3] S. Baruah, J. Gehrke, and C.G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 280–288, April 1995.

[4] S. Baruah, J. Goossens, and G. Lipari. Implementing constant-bandwidth servers upon multiprocessor platforms. In *Proceedings of the IEEE International Real-Time and Embedded Technology and Applications Symposium*, pages 154–163, September 2002.

[5] S. Baruah and G. Lipari. A multiprocessor implementation of the total bandwidth server. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, April 2004.

[6] S. 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.

[7] A. Block, B. Brandenburg, J. Anderson, and S. Quint. An adaptive framework for multiprocessor real-time systems. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 23–33, July 2008.

[8] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 71–80. IEEE, August 2007.

[9] B. Brandenburg and J. Anderson. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 61–70, July 2007.

[10] B. Brandenburg and J. Anderson. A comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\mathrm{RT}}$. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 105–124, December 2008.

[11] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, M-PCP, D-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 185–194, August 2008.

[12] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A status report. In *Proceedings of the 9th Real-Time Workshop*, pages 107–123. Real-Time Linux Foundation, November 2007.

[13] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169, December 2008.

[14] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353, April 2008.

[15] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123, December 2006.

[16] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In Joseph Y. Leung, editor, *Handbook on Scheduling Algorithms, Methods, and Models*, pages 30.1–30.19. Chapman Hall/CRC, Boca Raton, Florida, 2004.

[17] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 330–341, December 2005.

[18] H. Härtig, M. Hohmuth, and J. Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems*, September 1998.

[19] H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 413–422, 2007.

[20] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30:46–61, January 1973.

[21] R. Pellizzoni and M. Caccamo. The M-CASH resource reclaiming algorithm for identical multiprocessor platforms. Technical Report UIUCDCS-R-2006-2703, University of Illinois at Urbana-Champaign, March 2006.

[22] 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.

[23] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 181–190, July 2008.

[24] M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 228–237, December 1994.

[25] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094–1117, September 2006.

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

[27] 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, 2007.

[28] V. Yodaiken and M. Barabanov. A real-time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, January 1997.