Mixed-Criticality on Multicore (MC^2): A Status Report

Namhoon Kim, Jeremy P. Erickson, and James H. Anderson
Department of Computer Science, The University of North Carolina at Chapel Hill
{*namhoonk, jerickso, anderson*}@*cs.unc.edu*

***Abstract*—The MC^2 (mixed-criticality on multicore) framework has been proposed and implemented in LITMUS$^{\text{RT}}$, a real-time extension to Linux. The implemented MC^2 framework has been used in several research efforts pertaining to multiprocessor real-time systems. This paper describes the current status of work on MC^2. There are currently two MC^2 branches. We describe the features of each branch and report on current progress in unifying these branches.**

I. Introduction

Future embedded real-time systems are expected to require increased computational workloads and functionalities. Multicore platforms have the potential to meet these requirements by offering greater computational capabilities and advantages in size, weight, and power (SWaP). However, the introduction of multiple processing cores makes real-time resource allocation more difficult. To further complicate matters, many avionic and automotive embedded applications require tasks to be supported at different criticality levels, such as safety critical, mission critical, and best effort, on a single multicore system [1].

Because the failure of a safety critical task may cause a fatal failure of a system, such a task may be provisioned with a very pessimistic worst-case execution time (WCET). This can result in wasted computational capacity due to the difference between the predicted WCET and the actual execution time observed at run time. A technique to minimize this discrepancy has been proposed by Vestal [2]. He proposed the *multi-criticality (or mixed-criticality) task model*, which provides varying degrees of WCET assurance. Specifically, for low-criticality tasks, he proposed using less pessimistic WCETs for schedulability analysis, while for high-criticality tasks, he proposed using more pessimistic WCETs.

In the RTCA DO-178B and DO-178C software standards for avionics, criticality levels range from A (highest) to E (lowest) and are determined for a system component (e.g., a task) by examining the effects of failures. Mixed-criticality scheduling on multicore platforms was first considered by Anderson et al. [3]. They proposed operating-system (OS) infrastructure that allows mixed-criticality applications to be supported on a multicore platform, assuming the five criticality levels of DO 178B/C, while ensuring real-time correctness. In follow-up work, researchers at UNC Chapel Hill and Northrop Grumman Corp. proposed a mixed-criticality scheduling framework for multicore platforms, called MC^2 (mixed-criticality on multicore), and provided schedulability analysis results [4]. In MC^2, higher-criticality tasks are viewed as "slack generators" that use only a small fraction of their execution budget. Lower-criticality tasks execute using this slack. MC^2 also employs a two-level hierarchical scheduling approach, in which *containers* (also called *servers*) [3] are used to enable the temporal correctness of subsystems.

The first implementation of MC^2 was described by Herman et al. [1], who discussed design tradeoffs and evaluated the robustness of the implemented mixed-criticality scheduler with respect to breaches in execution-time assumptions. MC^2 is implemented within LITMUS$^{\text{RT}}$, a real-time extension of Linux that was designed to support real-time workloads on multicore platforms [5], [6], [13], [14].

In order to make safety-critical cyber-physical embedded systems more predictable, cache-management techniques were proposed by Ward et al. [7]. Specifically, they proposed two cache-management techniques, called *cache locking* and *cache scheduling*, and showed that the usage of such techniques can reduce WCETs in higher-criticality tasks. Ward et al. developed a branch of MC^2 in which these cache-management techniques are used, and presented experimental results on a multicore Tegra3 ARM machine.

As mentioned by Burns and Davis [8], a task may exceed its predicted level-l WCET. When such guarantees are violated, overload can occur. To provide guarantees in such overload situations, a recovery mechanism was proposed by Erickson et al. [9] that uses virtual time. This mechanism has been incorporated in a branch of MC^2 that employs a virtual timer [4]. This branch was used to obtain experimental results on an x86 machine. However, due to the different microarchitectures of MC^2 with cache management and with virtual time, these two branches of MC^2 have not yet been unified.

In this paper, we report on the current status of MC^2. In the rest of this paper, we provide relevant background (Sec. II), describe the current two branches of MC^2 (Sec. III), and then conclude (Sec. IV).

II. Background

In this section, we provide necessary background on multiprocessor real-time scheduling, LITMUS$^{\text{RT}}$, and the MC^2 architecture. We also briefly explain the container abstraction used for hierarchical scheduling and common MC^2 features.

A. Multiprocessor Real-time Scheduling

Task model. We assume that temporal constraints for tasks can be modeled by the implicit-deadline *periodic* or *sporadic task model*. Under the periodic task model, a system is comprised of a set of recurring tasks. Each such task τ_i releases a succession of *jobs*, denoted $\tau_{i,0}, \tau_{i,1}, \ldots$, and is defined by a *period*, p_i, and an *execution time*, e_i. Successive jobs of τ_i are released every p_i time units, starting at time 0, and a job released at time *t* must complete by its *deadline*, $t + p_i$. Under the sporadic task model, each task τ_i is specified by an execution cost, e_i, a *minimum separation* between successive job releases, p_i, and a *relative deadline*, d_i. Task τ_i's *utilization* is given by $u_i = e_i/p_i$. We sometimes assume a *harmonic task system* wherein all task periods are integer multiples of the smallest task period.

We assume a hardware platform with *m* processors. A task system is *schedulable* on such a platform under a given scheduling algorithm if no deadline constraint is violated. In a *hard real-time* (HRT) system, jobs must never miss their deadlines, while in a *soft real-time* (SRT) system, some deadline misses are tolerable. If a job $\tau_{i,j}$ released at $r_{i,j}$ completes execution at time *t*, then its *response time* is $t - r_{i,j}$ and its *tardiness* is $max\{0, t - d_{i,j}\}$. In the definition of SRT assumed in MC^2, tardiness is required to be bounded.

Partitioned and global scheduling. Under partitioned scheduling, tasks are statically assigned to processors and migration is not allowed, while under global scheduling, tasks may migrate across processors. Generally, partitioned scheduling is preferable in HRT systems, and global scheduling is preferable in SRT systems [10], [11]. Partitioned approaches have lower run-time overheads, but processing capacity may be wasted due to bin-packing problems. In contrast, global approaches eliminate bin-packing issues and are particularly effective in SRT systems where some deadline misses are allowed [12]. A drawback of global scheduling is increased OS overheads associated with contention of shared scheduler state.

B. LITMUS$^{\mathrm{RT}}$

MC^2 is implemented in LITMUS$^{\mathrm{RT}}$, an extension to the Linux kernel that supports real-time schedulers as plug-in components [13], [14]. LITMUS$^{\mathrm{RT}}$ was developed as an experimental platform for research on multiprocessor real-time scheduling and synchronization. Time-based events, such as job releases, are handled by Linux's high resolution timer application programming interface (hrtimer API) and scheduling events and synchronization requests are handled by plug-in event handlers. LITMUS$^{\mathrm{RT}}$ provides a very light-weight event tracing tool called *feather-trace* to record scheduling events and synchronization requests [15]. The partitioned *earliest-deadline-first* (P-EDF) and global EDF (G-EDF) schedulers have been implemented in LITMUS$^{\mathrm{RT}}$ previously [16]. As noted earlier, there are currently two branches of MC^2 implemented in LITMUS$^{\mathrm{RT}}$.

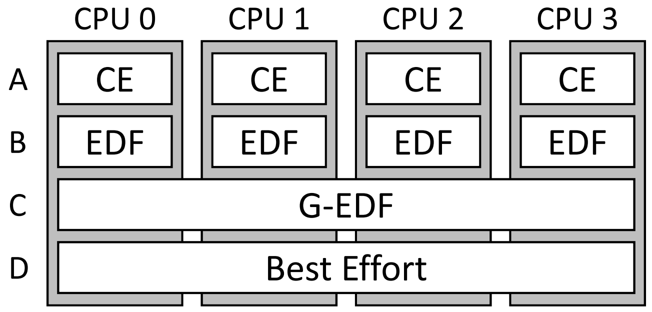

Fig. 1. Container allocation and the scheduler for each container under MC^2 on a four-processor system.

C. MC^2 Architecture

Vestal proposed a technique for eliminating under-utilization of processors due to very pessimistic WCET values [2]. He observed that the WCET values for higher-criticality tasks are needlessly pessimistic from the perspective of checking the schedulability of lower-criticality tasks. He proposed specifying per-criticality-level WCET values for each task. That is, each task τ_i has an execution time, $e_{i.l}$, for each criticality level *l* (depending on the scheduling scheme, it may not be necessary to specify an execution time for a task at higher criticality levels than its own). The level-*l* utilization of τ_i is defined as $u_{i.l} = e_{i.l}/p_i$. This has come to be known as the mixed-criticality task model. In the variant of MC^2 described here, four criticality levels are assumed, denoted A through D. MC^2 was designed with avionics workloads in mind and these workloads tend to be harmonic in nature [1].

Container abstraction. An essential part of mixed-criticality scheduling is that lower-criticality tasks should not affect higher-criticality tasks. This is very related to the concept of *temporal isolation*. Such isolation can be achieved by supporting a container (or server) abstraction within the OS. In mixed-criticality scheduling, a container is a group of tasks that is a isolated from the rest of the system [3]. MC^2 uses a two-level hierarchical scheduling approach. When the scheduler selects the next task to run on a processor, it first selects the highest-priority container among the containers that may execute tasks on that processor. Then, the scheduler selects the highest-priority task from the selected container, according to the associated scheduling algorithm of the container. The assumed containers and their associated scheduling algorithms are illustrated in Fig. 1, and explained below.

In MC^2 as proposed by Herman et al. [1], tasks are assumed to be periodic. Each level-*l* task τ_i is implemented as a single-task container within a container for its level. A task τ_i is assigned a *budget* equal to its execution time for its own level. The budget is consumed when the associated task executes and is replenished at time 0 and every p_i time units. Budget enforcement is enabled by default, but it can be disabled.

Level A. Level-A tasks are the highest-priority tasks in MC^2. They are statically assigned to processors and scheduled by a predefined dispatching table similar to the *cyclic executive* scheduling approach [17]. There are *m* level-A containers, one per processor. The schedulability analysis of level A is straightforward. Because level A is statically prioritized over all other levels, its schedulability is not affected by any other containers and is guaranteed at run time unless a level-A task τ_i exceeds it level-A budget, $e_{i.A}$.

If there are no level-A tasks to run on a processor at a given instant, then MC^2 considers level-B tasks. If a level-A task completes before its assigned level-A budget has been exhausted, then MC^2 allows a lower-criticality task to run for the duration of the remaining budget. This technique is known as *slack shifting* [1]. The completed job whose remaining budget is being consumed by a lower-criticality task becomes a *ghost job*. The ghost job completes when its remaining budget is equal to 0.

Level B. Similarly to level A, each processor has a level-B container. Level-B tasks are scheduled in EDF order. Optionally, *rate monotonic* (RM) scheduling can be used at level B. When no higher-criticality tasks are eligible to run on a processor, or when a level-A task is running as a ghost job, the scheduler selects the next job to run from the level-B container if such a job is available on that processor. It is required that the period of all level-B tasks is an integer multiple of the level-A *hyperperiod* (the least common multiple of level-A task periods) [4].

Level-B schedulability is achieved when the total level-B utilization of level-A and -B tasks on each processor is at most 1, since level-B scheduling across the system resembles the P-EDF scheduler and has similar theoretical properties. Level-B schedulability is guaranteed at run time unless some level-A or -B task exceeds its level-B execution time. Similarly to level-A jobs, a level-B job becomes a ghost job when it completes before exhausting its level-B budget; once it is a ghost job, its budget can be consumed by lower-priority tasks.

Level C. Level-C tasks are globally scheduled by the G-EDF algorithm. There is one global level-C container to which all level-C tasks are assigned. The G-EDF scheduler can be invoked on any processor whenever level-A or -B tasks are not eligible to run on that processor.

A level-C schedulability test is given in [4] assuming level-C execution times. Level-C schedulability is guaranteed at run time as long as no level-A, -B, or -C task exceeds it level-C execution time. Like higher-criticality levels, slack shifting is employed at level-C to allow level-D tasks to run.

Level D. Level-D tasks are scheduled on a best-effort basis. Such tasks are normal Linux tasks, which are not considered to be HRT or SRT tasks. Thus, there is no container for level-D tasks and no schedulability test is provided for this level. Level-D tasks can be scheduled by a stock Linux scheduler when there are no eligible real-time tasks to run.

Interrupt master. Dedicated interrupt handling, where all interrupts are directed to a designated processor called the *interrupt master* [16], can improve schedulability [1]. If an interrupt master is used, all release and timer events occur on the interrupt master. This enables budgeting for level-A and -B tasks on the other processors to be less pessimistic, but level-A and -B tasks on the interrupt master suffer from interrupt handing overheads. MC^2 supports using an interrupt master as an optional feature.

Timer merging. In a harmonic task system, multiple jobs are released frequently at the same time because the period of all tasks are integer multiples of the smallest period in the system. LITMUS$^{\mathrm{RT}}$ [5] uses a timer to release real-time jobs. In Linux, it is not guaranteed that all local timers start at the same time. Due to this local-timer error, tasks at levels B and C may have the same release time, yet their release timers may fire in reverse-criticality order. In this case, a level-C task is scheduled to run and then a level-B task is released and scheduled to run, which preempts the previously scheduled level-C task. To avoid this unnecessary preemption, an optional feature called *timer merging* was proposed by Herman et al. [1]. If enabled, release events within 1 μs of one another are merged using an $O(1)$ hash table operation. However, a global lock is required to merge all timer events across multiple processors. Thus, this feature can be enabled only when the interrupt master is enabled, which redirects all release events to a single processor.

Fine-grained locking. Each level-B and -C container has its own release queue and ready queue, and each level-A container has an associated dispatching table. Moreover, each processor has state indicating the currently scheduled task. The scheduler state data in MC^2 must be synchronized across processors to support MC^2's hierarchical scheduling approach. To access this state, spin locks are used to synchronize data structures on a per-container and per-processor basis [1]. The `rt_domain_t` data structure in LITMUS$^{\mathrm{RT}}$ is used to implement the ready and release queues needed to support containers. Fig. 2 provides an illustration.

If we do not carefully optimize synchronization, then MC^2 might suffer from significant overhead since the described state is accessed frequently. To mitigate this overhead, Herman et al. proposed a fine-grained state-locking mechanism [1]. This mechanism ensures two properties: (a) a processor lock can never be held for more than $O(1)$ time; and (b) container locks are never nested inside other container locks. Details are provided in [1].

III. Current MC^2 Branches

In this section, we discuss the two current branches of MC^2 implemented in LITMUS$^{\mathrm{RT}}$.

A. MC^2 with Virtual Time

In this subsection, we describe a branch of MC^2 in which virtual time is supported [9]. This version has been im-

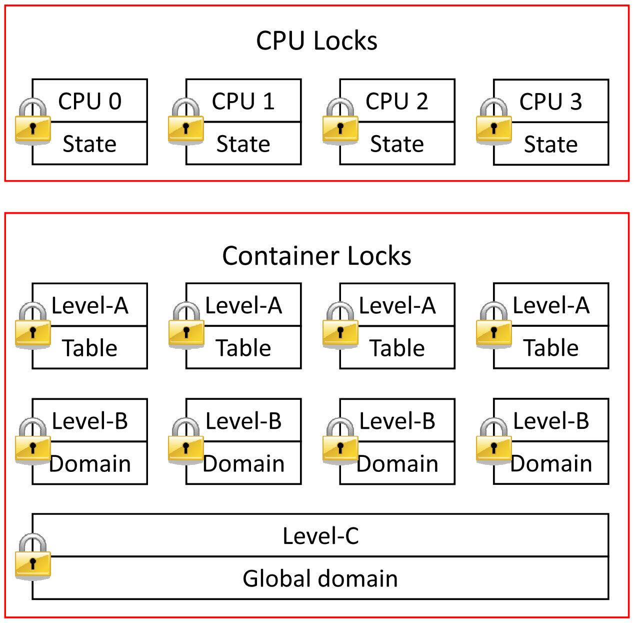

Fig. 2. Spin locks for containers and processors in the MC^2.

plemented based on the LITMUS$^{\mathrm{RT}}$ 2011.1 release (Linux 2.36.4). The usage of virtual time provides a mechanism to recover from overload of the level-C subsystem (described in more detail below), which can occur when any job at or above level C overruns its level-C WCET. This branch assumes a sporadic task model for level-C tasks since changes to the rate of virtual time alter the job-release frequencies at level C. In this branch, the interrupt master and timer merging are not supported.

Virtual time. Erickson et al. [9] modified MC^2 to support recovering from overload at level C. This can occur when any job at or above level-C overruns its level-C execution time.[1] In this situation, all successive job response times might increase permanently. Such ill effects can be dealt with by changing scheduling decisions until the overload situation has abated. In this MC^2 branch, such decisions are altered by using the idea of *virtual time* from Zhang [18] and Stoica et al. [19], where job releases are determined by a virtual clock that can change speeds with respect to the actual clock. Virtual time $v(t)$ is based on a global speed function $s(t)$. When a task overruns its level-C execution time and its response time exceeds a given tolerance value, the level-C scheduler slows down virtual time and reduces the job-release frequency at level C. The MC^2 branch with virtual time is comprised of a kernel component that manages virtual time and a userspace component that monitors job releases and completions. The kernel component controls job releases based on virtual time. The userspace component, called a *monitor* program, collects job-release and -completion information from the kernel to detect an overload situation or an idle instant. The monitor program is responsible for determining the virtual-clock speed. This virtual-clock mechanism affects only level-C tasks, not level-A or -B tasks. Detected idle instants are used to determine when recovery is completed, at which point virtual-time speed returns to actual-time speed. Experimental results show that the scheduling overheads and the execution time of the userspace monitor program are small. The introduction of virtual time increases scheduling time by about 40% on average and by 100% in the worst case. Each invocation of the monitor program completes in approximately 1 ms in the worst case [9].

[1]If budget enforcement of level-C tasks is disabled, a level-C job can overrun its level-C WCET. Even with budget enforcement, level-A and -B tasks can overrun their level-C WCETs.

GEL-v scheduling. In the MC^2 variant proposed by Mollison et al. [4], level-C tasks are scheduled by using G-EDF. As noted by Erickson et al. [20], other *G-EDF-like (GEL)* schedulers can provide better response-time bounds, so in this branch, arbitrary GEL schedulers are allowed at level C. Furthermore, since the virtual clock is used to manage job releases of level-C tasks, a modified version of GEL scheduling, called *GEL with virtual time* (GEL-v) scheduling, and a generalized version of the sporadic task model, called the *sporadic with virtual time and overload (SVO) model*, are used for level-C tasks. Under GEL-v scheduling, each job $\tau_{i,k}$ is prioritized on the basis of a virtual *priority point* (PP), and each task τ_i is characterized by a minimum separation time $T_i > 0$, and a relative PP $Y_i \geq 0$, both with respect to virtual time. At time 0, $s(t)$ is equal to 1, which means that actual time and virtual time progress at the same rate. However, when an overload is detected, the scheduler decreases $s(t)$, which reduces the progress of virtual time. As explained above, this slows down the rate of future job releases of level-C tasks, and creates extra slack to enable the system return to normal behavior.

B. MC^2 with Cache Management

Another MC^2 branch with cache management has been implemented by Ward et al. [7]. In this branch, several shared-cache management techniques have been implemented assuming a quad-core ARM machine. However, this MC^2 branch only supports the level-B and -C subsystems. The MC^2 branch with cache management uses *cache lockdown* mechanisms that requires hardware support. This is why this MC^2 branch works only for a specific ARM platform, which provides the needed cache lockdown instructions.

Cache management. Ward et al. [7] proposed a cache management technique that preallocates the dynamic memory a job uses before the job begins execution. *Page coloring* is used to allocate the memory pages required by a job. Under page coloring, a color is assigned to each page to control the mapping address of the page. The pages that have different colors map to different cache sets, so they cannot conflict with each other in the last level cache (LLC). This is used in conjunction with cache lockdown to prevent active pages for being evicted during the execution of the job. The cache is treated as a shared resource that can be either preemptive or non-preemptive, yielding two possible cache allocation policies: *cache locking* and *cache scheduling*. Under cache locking, the processors and the cache are not preemptible, while under cache scheduling, they are. In this MC^2 branch, cache management techniques are applied to level-B tasks. These cache-management techniques are not applied at level C, which is provisioned using less pessimistic WCETs. The MC^2 scheduler loads the memory pages of the next level-B job to execute into the shared LLC and flushes the pages used by the previous job. This results in additional scheduling overheads. However, it has been shown that cache management enables significant schedulability gains by reducing level-B WCETs.

Resource sharing. Cache locking ensures that the pages required by a job reside in the cache during the entire duration of its execution. This policy requires a multiprocessor real-time locking protocol: the cache is treated as a shared resource that has *k* replicas as given by the number of cache ways. The RNLP [21], which optimally supports the simultaneous locking of replicated resources, is used for this purpose. For example, if a job requires *r* pages with the same color, then it must lock *r* replicas of that color. Also, the job may require several colors simultaneously. To support these requirements, this MC^2 branch uses *dynamic group locking* as proposed by Ward et al. [22] in the context of the RNLP. Dynamic group locks allow a job to lock multiple resources with one lock request rather than requesting each resource individually in a nested fashion, which can increase system-call overhead and blocking times. The RNLP controls all colors and ways by using a FIFO queue for each way of each color. The maximum duration of blocking for all cache colors is $O(mr/k)$ where k is the number of ways available and r is the maximum number of ways per color requested by any job [21].

C. MC^2 in LITMUS$^{\mathrm{RT}}$ 2014.1

The previous two branches of MC^2 in LITMUS$^{\mathrm{RT}}$ have not been merged because they require different microarchitectures and the MC^2 patches are based on different Linux kernel versions (2.6.36 and 3.0.0). We are currently trying to unify the two branches with their features as a kernel configuration. This work is not finished at this time. We discuss some of the issues in unifying both branches in this section.

Container implementation. MC^2 ensures temporal isolation by supporting a container abstraction. However, LITMUS$^{\mathrm{RT}}$ currently does not support such an abstraction. The previous MC^2 implementation considers a real-time task as a container. Thus, the data structure `rt_param` in LITMUS$^{\mathrm{RT}}$ has extra variables to support container functions, such as replenishment and consumption. The MC^2 branch with virtual time uses `real_release` and `real_deadline` variables to keep track of a job's release and completion time, while the MC^2 branch with cache management uses another `rt_job` data structure, `user_job`. We need to merge these two different data types to support the container abstraction. This approach fulfills its requirements, but it is hard to trace the behaviors of both a container and each individual task in the container, and this implementation is not well-suited to job handling in LITMUS$^{\mathrm{RT}}$.

More fine-grained locking. As shown in Fig. 2, there is a lock for each domain. The domain structure at levels B and C includes release and ready queues. The scheduler is required to hold the ready-queue lock when a task is added to the release queue and vice versa. This domain locking at levels B and C should be more fine-grained. The locks at level A are fine-grained enough because the domain structure at level-A only has a dispatching table.

Cyclic executive scheduling table. Making a scheduling table for level-A tasks is quite complicated now. We currently use the Linux *proc* file system to construct a table, and we must change the LITMUS$^{\mathrm{RT}}$ scheduler plugin several times whenever changing a budget or adding a new task. We want to devise a more convenient way to manipulate the scheduling table. In both MC^2 branches, the scheduling table can be accessed by `read_proc_t` and `write_proc_t` function pointers. However, in Linux 3.10 (the base version of LITMUS$^{\mathrm{RT}}$ 2014.1), the structure `proc_dir_entry` does not have those function pointers anymore. We need to implement `proc_fops` operations to unify the two branches.

IV. Conclusion

In this paper, we have discussed the current status of work on MC^2, the first mixed-criticality scheduling framework implemented on multicore platforms. Due to the different microarchitectures and base kernel versions in LITMUS$^{\mathrm{RT}}$, two branches of MC^2 exist. We hope that the unified MC^2 we are constructing will provide more features and portability as a mixed-criticality research testbed.

In future work, we plan to extend MC^2 to allow tasks to acquire locks and have precedence constraints, in order to enable more realistic workloads. In addition, we hope to ease or remove the hardware dependency of the cache-management MC^2 branch. The cache-management MC^2 branch requires cache-lockdown instructions, which are not widely supported. We plan to investigate cache allocation mechanisms to remove the preloading and flushing of memory pages whenever a job is scheduled.

References

[1] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson. RTOS support for multicore mixed-criticality systems. In *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 197–208, April 2012.

[2] 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.

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

[4] M. Mollison, J. Erickson, J. Anderson, S. Baruah, and J. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Proceedings of the 7th IEEE International Conference on Embedded Software and Systems*, pages 1864–1871, June 2010.

[5] LITMUS$^{\mathrm{RT}}$ homepage. http://www.litmus-rt.org/.

[6] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina, Chapel Hill, 2011.

[7] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 157–167, July 2013.

[8] A. Burns and R. Davis. Mixed criticality systems - a review. http://www-users.cs.york.ac.uk/~burns/review.pdf, December 2013.

[9] J. Erickson, N. Kim, and J. Anderson. Recovering from overload in multicore mixed-criticality systems. In submission, 2014.

[10] 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.

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

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

[13] 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 Linux Workshop*, pages 107–123, November 2007.

[14] 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.

[15] B. Brandenburg and J. Anderson. Feather-trace: A light-weight event tracing toolkit. In *Proceedings of the 3rd International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 20–27, July 2007.

[16] B. Brandenburg and J. Anderson. On the implementation of global real-time schedulers. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 214–224, December 2009.

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

[18] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols*, pages 19–29, September 1990.

[19] I. Stoica, H. abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 288–299, December 1996.

[20] J. Erickson, J. Anderson, and B. Ward. Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling. *The Journal of Real-Time Systems*, 50(1):5–47, 2014.

[21] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 223–232, July 2012.

[22] B. Ward and J. Anderson. Fine-grained multiprocessor real-time locking with improved blocking. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, pages 67–76, October 2013.