RTOS Support for Multicore Mixed-Criticality Systems *

Jonathan L. Herman,† Christopher J. Kenna,† Malcolm S. Mollison,† James H. Anderson† and Daniel M. Johnson‡
†The University of North Carolina at Chapel Hill
‡Northrop Grumman Corp.

Abstract

Mixed-criticality scheduling algorithms, which attempt to reclaim system capacity lost to worst-case execution time pessimism, seem to hold great promise for multicore real-time systems, where such loss is particularly severe. However, the unique nature of these algorithms gives rise to a number of major challenges for the would-be implementer. This paper describes the first implementation of a mixed-criticality scheduling framework on a multicore system. We experimentally evaluate design tradeoffs that arise when seeking to isolate tasks of different criticalities and to maintain overheads commensurate with a standard RTOS. We also evaluate a key property needed for such a system to be practical: that the system be robust to breaches of the optimistic execution-time assumptions used in mixed-criticality analysis.

1 Introduction

In embedded real-time systems, it is commonly the case that the severity of failure is not the same for all tasks in the system. For example, the failure of one task may cause loss of life, while the failure of a different task may only cause degraded system performance. Such tasks are said to be of differing *criticalities*. Because a failure may have severe repercussions, the schedulability of such a system is conventionally assessed assuming very pessimistic worst-case execution times for highly critical tasks. Thus, the system may be fully utilized from a validation and certification perspective, *i.e.*, at *design time*, but be severely underutilized in practice, *i.e.*, at *run time*.

A technique for reclaiming this spare capacity has been proposed by Vestal [15]. He observed that, from the perspective of scheduling a less critical task, the execution times assumed of more critical tasks are needlessly pessimistic. Thus, he proposed that schedulability tests for less critical tasks be altered to incorporate less pessimistic execution times than those of the more critical tasks. More formally, in a system with L criticality levels, L system variants are analyzed: in the level-l variant, level-l execution times are assumed. The degree of pessimism in determining such execution times is level-dependent: if level l is of higher criticality than level l', then level-l execution times will be generally greater than level-l' execution times. The resulting task model has come to be known as the *mixed-criticality task model*.

The publication of [15] spurred additional work by other researchers on mixed-criticality scheduling, almost all of which has been directed at uniprocessor platforms (see [3] for relevant citations). In contrast to this uniprocessor-directed work, researchers at UNC Chapel Hill, in collaboration with colleagues at Northrop Grumman Corp. (NGC), have been working to determine whether mixed-criticality scheduling techniques can be practically applied on multicore platforms. This work has been motivated by the requirements of next-generation unmanned aerial vehicles (UAVs). In this context, there is a desire, primarily motivated by size, weight, and power (SWaP) concerns, to consolidate a large computational workload on a few multicore machines. Moreover, different criticality levels are intrinsic to such a workload. For example, tasks that are responsible for adjusting flight surfaces or responding to immediate threats are "safety critical"; tasks that are responsible for external communication and decision-making capabilities are "mission critical"; and (some) tasks that perform route mapping and surveillance may require significant computational capacity but do not have strict timing requirements and hence may be viewed as "best effort" (*i.e.*, non-real-time). Of course, the main challenge in this domain is to devise mixed-criticality scheduling (and ultimately synchronization) approaches for multicore platforms that are amenable to certification.

As a step towards addressing this challenge, the UNC/NGC team proposed a mixed-criticality scheduling framework for multicore platforms and provided corresponding schedulability analysis results [14]. This framework, referred to here as MC^2,[1] supports five criticality levels, denoted A (highest) through E (lowest). The choice of five levels was motivated by the five criticality levels found in the DO-178B standard for avionics. As explained in greater detail later, level-A and -B tasks in MC^2 are subject to hard deadlines and are scheduled via partitioning, while level-C and -D tasks are subject to bounded deadline tardiness and are scheduled globally. Level-E tasks are scheduled as best-effort tasks because DO-178B merely specifies that a failure at this level must not affect the operation of the aircraft. The schedulability analysis provided for MC^2 can be applied to validate the schedulability

*Work supported by NSF grants CNS 1016954 and CNS 1115284; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

[1] This notation, which stands for "mixed-criticality on multicore," was not used in [14]; we introduce it here for readability.

of the level-l system, where l ranges over A–D (as required in a mixed-criticality setting; note that level E is best effort, so it requires no schedulability analysis).

The schedulability analysis presented for MC^2 assumes an overhead-free task model. In practice, however, overheads can have a profound impact on schedulability. This has been demonstrated in prior schedulability studies in which overheads (but not criticalities) were considered (see [4] for relevant citations). Two sources of overhead are of concern: those introduced by the operating system (assuming, as we do here, that the scheduling framework is implemented in the OS instead of middleware) and cache-related costs due to preemptions and migrations. In a mixed-criticality setting, overheads should already be accounted for when determining per-criticality-level execution times. However, properly determining the impact of the OS-related overheads requires a working scheduler implementation.

A scheduler implementation is also needed to address issues of particular relevance to a mixed-criticality setting. For example, while MC^2 completely isolates level-A tasks from tasks at other levels *in theory*, OS activities (such as processing interrupts associated with lower-level work) can interfere with this sense of isolation. How should the OS be designed to minimize this interference?

Focus of this paper. There are two major contributions of this paper: the first is a discussion of design tradeoffs that affect mixed-criticality scheduling with a focus on reducing scheduler-induced overheads; the second is an evaluation of the robustness of the implemented mixed-criticality scheduler.

More concretely, we present an experimental evaluation of MC^2 motivated by the issues raised above as implemented within a UNC-produced real-time OS (RTOS) called LITMUS$^{\mathrm{RT}}$ [4]. This evaluation was conducted to assess different RTOS design tradeoffs that affect mixed-criticality scheduling and to determine how well the theoretical schedulability analysis of MC^2 carries over to practice. Regarding RTOS design choices, we sought to determine whether overheads in an RTOS that must manage multiple criticality levels can be made commensurate with overheads seen in RTOSs where criticalities do not arise. Furthermore, we sought to constrain and assess the impact of OS interference with respect to cross-level isolation guarantees.

With respect to schedulability, we also sought to assess the *robustness* of mixed-criticality analysis. As noted earlier, under this analysis, a system with L criticality levels is viewed as L different systems: when analyzing the system at level l, all tasks at all levels are assumed to execute for at most their level-l execution times. What happens in practice if the system functions as an "almost" level-l system, *i.e.*, level-l execution times are sometimes, but not often, exceeded? Is real-time correctness at level l completely compromised in this case? Or, does deviance from correct level-l behavior fall off more gradually as violations of level-l execution times become more common? Obviously, the latter would be preferred in practice. To determine the robustness of MC^2, we conducted a series of experiments in which, for each level l, deviance from level-l execution times is gradually increased.

To our knowledge, this is the first paper on multicore mixed-criticality scheduling to consider OS-related implementation issues. However, it is only a first step towards the practical deployment of a mixed-criticality multicore scheduler. For a mixed-criticality scheduler to be applied in practice, appropriate techniques must be used to determine per-level task execution times. Such techniques are *not* considered in this paper; rather, the focus of this paper is the implementation and evaluation of a mixed-criticality scheduler in a multicore system. We assume that tools and techniques exist to determine task execution times, and that the calculated times include any overheads incurred. We further assume that such tools properly account for relevant architectural features (*e.g.*, shared caches, if they exist) and software characteristics (*e.g.*, task working set sizes).

In the rest of this paper, we discuss needed background (Sec. 2), present our design and analyze the effects of implementation tradeoffs (Sec. 3), consider the issue of robustness (Sec. 4), and then conclude (Sec. 5).

2 Background

In this section, we first present necessary background on multiprocessor real-time scheduling. Then, we present an overview of MC^2. Finally, we give an overview of LITMUS$^{\mathrm{RT}}$, the RTOS underlying our implementation of MC^2.

2.1 Underlying Concepts

Task model. The MC^2 framework assumes that temporal constraints for tasks can be modeled by the *periodic mixed-criticality task model*. Under this model, each task T has an associated *period*, $T.p$, and execution time for each criticality level l, denoted $T.e_l$. (This value may be undefined for criticality levels higher than T's own criticality level.) 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 level-l *utilization*, or long-run processor share required by a task assuming a level-l execution time, is given by $T.u_l = T.e_l/T.p$.

Our focus is on the implementation of a mixed-criticality multicore scheduler; therefore, we assume that techniques exist for determining task execution times. The calculated execution times should account for variations and overheads caused by the OS, impacts due to a task's working set size, and architecture-specific issues such as cache line migrations.

Schedulability. 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* (HRT) tasks, jobs must never miss their deadlines, while for *soft real-time* (SRT) tasks, some deadline misses are tolerable. In the latter case, we require deadline tardiness to be provably bounded by a (reasonably

small) constant (*e.g.*, using analysis such as that found in [9]).

Hierarchical scheduling. MC^2 uses a two-level hierarchical scheduling approach. When the scheduler is invoked to select the next task to run on a processor, it first selects a subset of tasks, known as a *container* (in other literature, sometimes called a *server*). Second, the scheduler selects a task to execute from the chosen container, according to a scheduling algorithm associated with that particular container. In MC^2, such a scheme is used to treat differently tasks of different criticality. It also allows the temporal correctness of subsystems to be validated independently.

2.2 MC^2

MC^2 was designed assuming a modest core count (*e.g.*, $m \in \{2, \ldots, 8\}$), and we assume this as well throughout this paper.[2] Because the level-D tardiness bound of the original version of MC^2 in [14] is large, we instead implement a slight variation of MC^2 supporting four criticality levels (instead of the original five), labeled A through D. In this variant, the original level-D system is not implemented, and the best-effort level-E system is simply "renamed" to level D.[3] In the new system, A is the highest criticality, while D is the lowest criticality. Levels A and B each comprise m containers—that is, one per processor, per level. Levels C and D each comprise one container, shared among all m processors. This container allocation scheme is illustrated in Fig. 1.

In our implementation, there is implicitly an additional level of containment, as per-task budgets are enforced. Specifically, a level-l task T is assigned a *budget* (*i.e.*, an OS-enforced execution time) equal to its execution time at level l, $T.e_l$. In essence, T itself is implemented as a single-task container (within a container for its level) that receives a budget allocation of $T.e_l$ time units every $T.p$ time units. If an actual job of T has an execution time exceeding $T.e_l$, then several consecutive budget allocations will be required to service it. Hereafter, we use the term "container" only to refer to per-level containers. Note that, while budget enforcement is the default in our implementation, it can be disabled.

We now describe the two-level hierarchical scheduling scheme employed by MC^2.

Level A. Level-A tasks are statically prioritized above all other tasks in the system. They are scheduled according to a precomputed dispatching table, following the *cyclic executive* scheduling model [2]. The predictable and easy-to-analyze nature of this type of scheduler has led to its adoption as the de facto standard in industry for scheduling highly-critical workloads.

If no level-A task is eligible to run on a processor at a given instant, the scheduler instead considers level-B tasks. Furthermore, if a level-A task completes before its assigned level-A budget has been exhausted, MC^2 allows a lower-criticality task to run for the duration of the remaining budget. This technique is known as *slack shifting*.

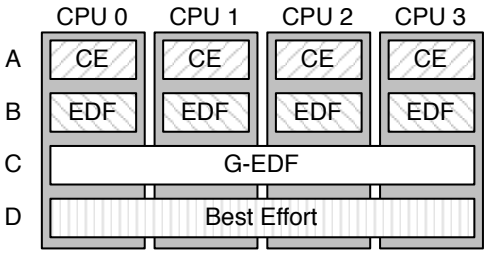


Figure 1: Container allocation under MC^2 on a four-processor system. "CE," "EDF," "G-EDF," and "Best Effort" indicate the scheduler used for each container (see text for details).

Slack shifting is a key optimization that allows lower-level work to safely execute earlier than it otherwise would. When slack shifting is in progress, the completed job, whose excess budget is being consumed by lower-level work, is known as a *ghost job*. From a schedulability perspective, slack shifting is transparent for tasks at the criticality level of the ghost job.

Level-A schedulability is achieved by applying existing techniques for constructing cyclic executive schedulers. In determining level-A schedulability, all tasks of lower criticality are ignored. Schedulability is guaranteed at runtime as long as no level-A task exceeds its level-A execution time. See [14] for restrictions on level-A task periods.

Level B. When no level-A tasks are eligible to run, or when a level-A task is "running" as a ghost job, the scheduler selects a level-B task (if one is eligible). Level-B tasks are scheduled in *earliest-deadline-first* (EDF) order, which is optimal on a uniprocessor. Because there is one level-B container per processor, level-B scheduling across the system resembles the *partitioned EDF* (P-EDF) scheduler, and has similar theoretical schedulability properties. P-EDF is a good candidate scheduler for HRT workloads that do not require the strict behavior provided by a table-driven cyclic executive.

Level-B schedulability is achieved when the level-B execution times of level-A and -B tasks on each processor do not exceed the total utilization of that processor. Schedulability is no longer guaranteed at runtime when some level-A or level-B task exceeds its level-B execution time. (In Sec. 4, we examine what happens to level-l tasks when execution times exceed level-l times.)

Similarly to level-A jobs, level-B jobs become ghost jobs when they complete before exhausting their level-B budget. In this case, or when no level-B job is eligible on a processor, a level-C job will be selected to run (if one is available).

Because EDF scheduling has not yet been widely accepted by the certification community for HRT tasks,[4] it is worth

[2] Multicore platforms are currently not used in avionics to host highly critical workloads. Enabling a platform with two to eight cores to be used would be a significant innovation.

[3] In the remainder of this paper, the use of "MC^2" to refer to the four- or five-level variant is context-dependent.

[4] The ARINC 653 specification for safety-critical RTOSs is a notable exception; it allows EDF scheduling as a second-level scheduler under a

	Crit.	$T.p$	$T.e_A$	$T.e_B$	$T.e_C$	$T.e_D$
T_1	A	5	3	2	1	1
T_2	A	10	4	2	2	2
T_3	B	10	–	2	2	1
T_4	B	20	–	2	1	1
T_5	C	10	–	–	2	2
T_6	C	20	–	–	2	2
T_7	D	5	–	–	–	2

Table 1: Example mixed-criticality task system.

noting that MC^2 could easily be modified to support *rate-monotonic* [12] scheduling in place of P-EDF. Besides being straightforward to implement, such a modification would not affect schedulability under the existing analysis, given restrictions on level-B task periods assumed in [14], where such analysis is presented. However, we chose to retain P-EDF at level B for the implementation described in this paper, in the hope that these restrictions will be lifted in a future extension to MC^2.

Level C. Unlike higher-criticality tasks, level-C tasks are not assigned to processors, but are instead scheduled globally across all processors. They are selected in EDF order; therefore, level-C scheduling resembles the *global EDF* (G-EDF) scheduler. Level-C tasks have only a SRT guarantee, namely, bounded deadline tardiness. G-EDF is known to be optimal with respect to ensuring bounded tardiness [9].

A schedulability test for level-C tasks is given in [14] assuming level-C task execution times. Schedulability is guaranteed at runtime as long as no level-A, -B, or -C task exceeds its level-C execution time. (Again, in Sec. 4, we examine what happens to level-l tasks when execution times do exceed level-l times.) As with higher levels, slack shifting is employed at level C (to allow level-D jobs to run earlier than they otherwise would).

Level D. Level-D tasks are scheduled on a best-effort basis. Thus, no schedulability test is provided. This level can be used for tasks that simply need to make a predictable amount of progress over time, and tasks that require a quick response time but are not considered HRT or SRT.

Example. Table 1 gives an example mixed-criticality task system, showing each task's period and execution times for different criticality levels. Fig. 2 shows how the system would be scheduled under MC^2 for the first 10 units of execution time. (Although this paper concerns multiprocessor systems, only a single processor is assumed in the example, in order to ease understanding.)

Note that the overall design of MC^2 was motivated by a desire to make intelligent tradeoffs among real-time schedulability (*i.e.*, highly utilizing the system), certification constraints, and engineering practice. More detail on these tradeoffs can be found in [14].

hierarchical scheduling paradigm [11].

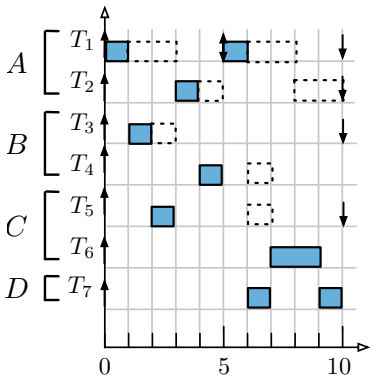


Figure 2: Possible MC^2 schedule for the task system in Table 1. Empty boxes represent ghost jobs. Up-arrows indicate releases, while down-arrows indicate deadlines.

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

We implemented MC^2 using LITMUS$^{\mathrm{RT}}$, an extension to the Linux kernel that supports real-time schedulers as event-driven plugins [4]. Two categories of events exist under LITMUS$^{\mathrm{RT}}$. Time-based events, such as job releases, are handled by plugin-defined interrupt routines triggered by Linux's high resolution timer (*hrtimer*) framework. Scheduling events, including job completions and synchronization requests, are also handled by plugin-defined event handlers.

Our plugin implementation provides event handlers for levels A through C. (Level-D tasks are relegated to the stock Linux scheduler.) The code for our plugin can be found on the LITMUS$^{\mathrm{RT}}$ homepage [1]. We do not elaborate on our event handlers, as cyclic executive schedulers have long been well-understood, and significant prior work has been done on the implementation of P-EDF and G-EDF schedulers in LITMUS$^{\mathrm{RT}}$ [6]. Instead, in Sec. 3, we focus on issues that arise from supporting all of these schedulers simultaneously in a hierarchical manner, including specific challenges that come with supporting the MC^2 framework.

3 Implementation Description and Evaluation

In this section, we explore the question of how best to support the MC^2 framework in an RTOS environment. For MC^2 to prove viable, two key RTOS overhead-related criteria must be met. First, it must be possible to bound overheads by relatively small constants. In the face of the complex state synchronization needed to support MC^2's hierarchical scheduling requirements, this is a serious challenge. Second, the overheads that *do* exist must be made to penalize higher-criticality tasks as little as possible (and, instead, penalize lower-criticality tasks). Otherwise, since they are provisioned in a pessimistic manner (such a provisioning would include overhead accounting), higher-criticality tasks could be adversely impacted.

The rest of this section is organized as follows. First, we discuss relevant overhead metrics. Then, we discuss four

specialized techniques used to meet the criteria outlined above. Finally, we present an evaluation of overheads in general and of our techniques in particular.

3.1 Overhead Metrics

We are concerned with two kinds of RTOS overheads: release overhead and scheduling overhead.

Release overhead is accrued when a LITMUS$^{\mathrm{RT}}$ release handler, triggered by the firing of a release timer, is executing. The release handler removes each task being released from the applicable release queue (*i.e.*, the release queue associated with the container to which that task belongs), and merges it into the applicable ready queue. (For level A, rather than accessing queues, the release handler references the applicable cyclic scheduling table.) The release handler also determines if each released task needs to be scheduled and, if so, notifies the affected processor, triggering it to begin executing a scheduling event.

Scheduling overhead is accrued when a scheduling handler is executing, triggered by either a task completion, or a notification of the need to reschedule. Unlike the release handler, the scheduling handler must execute on the processor that is being rescheduled. If a task is already running on the processor, the scheduling handler will preempt it and merge it into the applicable ready queue. (For level A, no requeuing is necessary.) If the preempted task is from level C, the scheduling handler will determine whether the task is of sufficient priority to begin executing on a remote processor. If so, the handler notifies that processor to reschedule. Finally, the scheduling handler removes the next task to run from the applicable ready queue (or references it in the table) and initiates a context switch to that task.

Fig. 3 (a) gives examples of these overheads. At time 3 task T_B releases a job. This causes the release handler to execute on P_2 (though it could have executed on any processor). The handler dequeues T_B from the release queue of P_2's level-B container, and enqueues it in the ready queue of the same container. Then, because level-B tasks are of higher priority than level-C tasks, it initiates a scheduling event on P_2, causing the scheduling handler to execute. The scheduling handler preempts T_C and enqueues it in the level-C ready queue (which is global), and, observing that T_C has sufficient priority to execute on P_1, notifies P_1 of the need to initiate a scheduling event. (This notification is accomplished using an inter-processor interrupt.) Finally, the handler dequeues T_B from the ready queue and initiates a context switch to it. After P_1 receives the notification, it, too, initiates a scheduling event, causing T_C to resume execution.

In provisioning the task system, the execution budget of T_B must be inflated to account for the overhead to release and schedule T_B, as well as any other release and scheduling events that can occur on P_2 while T_B is executing. Note that this overhead is increased by the need to service a level-C task from time 4 to time 5. This runs counter to the second criterion listed at the beginning of Sec. 3. In Sec. 3.2, we introduce and evaluate a technique to rectify this problem.

3.2 Specialized Techniques

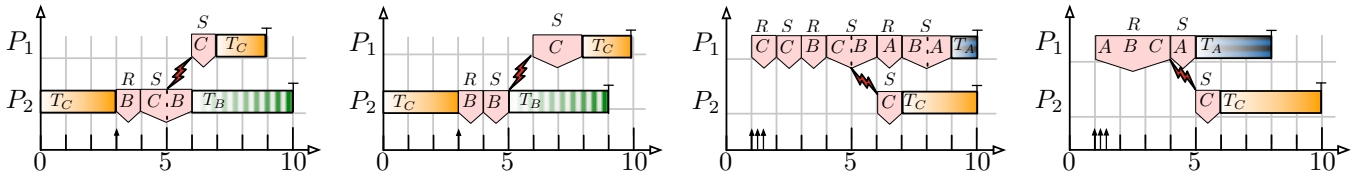
Fine-grained state locking. As noted at the beginning of Sec. 3, the scheduler state data that must be synchronized across processors for MC^2 is significant. Each container has an associated dispatch table (level A), or associated ready and release queues (levels B and C). Furthermore, each processor has state associated with it indicating the task currently scheduled to run. Spin locks are used to synchronize access to data structures on a per-container and per-processor basis. Fig. 4 gives an illustration.

The overall scheduling approach of MC^2 creates a high degree of contention for the described state. A naïve implementation that does not carefully optimize locking patterns would almost certainly suffer from significant overhead. In contrast, our implementation adopts the strategy of maintaining scheduler state locks that are as fine-grained as possible. Two important properties of our implementation that result from this strategy include: **(a)** processor locks are never held for more than $O(1)$ time; and **(b)** container locks are never nested inside other container locks.

Regarding (a): In order for an event handler to check for the need to initiate a local or remote scheduling event, it needs to compare the task running on the relevant processor to the highest-priority tasks in some container's ready queue. Obtaining that task typically requires $O(\log n)$ time, where n is the number of tasks in the ready queue. Intuitively, it would appear that the checking operation requires a task to hold both a processor lock and a container lock for $O(\log n)$ time. However, we employ a specialized strategy to avoid this penalty. In this strategy, processor locks are always nested inside container locks (not the other way around), and the results of all needed $O(\log n)$ operations are obtained and cached *before* the processor lock is acquired. Thus, the processor lock is never held for more than $O(1)$ time. While this strategy would not necessarily pay off in single-level global schedulers, we believe it is an important optimization for MC^2, because processor locks are especially highly contended (since multiple containers can "compete" for the same processor lock).

Regarding (b): A context switch that transitions between two different criticality levels requires obtaining locks for two containers. In our implementation, the first lock is dropped before the second is acquired. For example, in Fig. 3 (a), the scheduling handler that runs at time 4 must enqueue T_C into the level-C ready queue *and* dequeue T_B from P_2's level-B ready queue. In order to prevent holding both of these container locks at once, T_C is moved from P_2's lock-protected running task data structure to stack memory, instead of being immediately enqueued into the level-C ready queue. Only when the scheduling of T_B completes is T_C finally enqueued in the level-C ready queue.

A naïve implementation would be vulnerable to deadlock in two scenarios: when a processor must simultaneously ac-



(a) Example of overheads (b) Work redistribution technique (c) Pathological timer interrupts (d) Timer merging technique

Figure 3: RTOS overhead illustrations. Each task is subscripted according to its criticality level. Each "tab" (downward-pointing block) indicates overhead (either release overhead, denoted "R," or scheduling overhead, denoted "S"). Within a "tab," each letter indicates the container for which state data is being updated in a given interval. Inter-processor interrupts are denoted with a lightning bolt. Arrows indicate job releases, and "⊤" symbols indicate job completions.

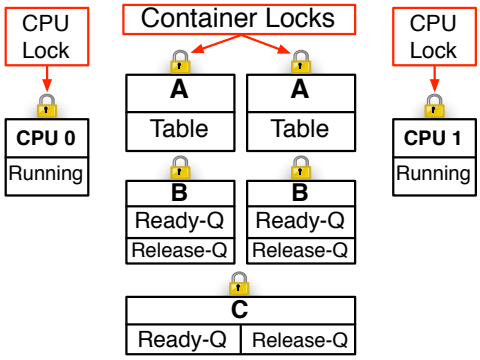


Figure 4: MC^2 scheduling state on a two-processor system.

cess both CPU and container state, and when a processor must simultaneously access state from two containers. Fortunately, under (a) and (b), our implementation avoids these scenarios. The fixed locking order given for (a) avoids the first deadlock source by preventing circular dependencies between container locks and CPU locks, *i.e.*, no processor can block on a CPU lock which is, in turn, held by a processor blocked on any container lock. Rule (b) avoids the second deadlock source by ensuring no container lock is ever held by a processor waiting on another container lock.

Note that the level-C container includes a data structure (not shown in Fig. 4) that allows event handlers to determine which processors (if any) should be preempted if higher-priority level-C work becomes available. In the case of Fig. 3 (a), this data structure needs to be updated immediately when T_C is preempted. However, updates to this data structure normally require the level-C lock, which has not yet been acquired by the scheduling handler. To work around this problem, our implementation uses a partially wait-free technique in which a consistency check must be performed by any handler that *does* acquire the level-C lock.

Interrupt master. It has been shown that redirecting all interrupts (such as timer interrupts) to a single processor, denoted here as the *interrupt master*, can significantly improve schedulability in single-level global schedulers [6]. Our implementation supports an interrupt master as an optional feature, allowing us to evaluate its effect in hierarchical scheduling and in MC^2 in particular (presented later in Sec. 3.3). When this feature is enabled, all release events and device interrupts occur on the interrupt master. This allows budgeting for level-A and -B tasks on other processors to be less pessimistic, as it is not necessary to account for release overhead suffered on behalf of other tasks. However, level-A and -B tasks on the interrupt master are penalized by this scheme. A real-world deployment may avoid allocating level-A and -B tasks to the interrupt master for this reason (our experiments take this approach).

Note that, by consolidating all device and timer interrupts onto a single CPU, an interrupt master can potentially increase the interrupt delays to which a single release event is exposed. A real-world implementation would need to address this issue if excessive device interrupt delays were possible. Specifically, the system would need to shield release events from the effects of these delays.

Timer merging. Recall that MC^2 was designed with avionics workloads in mind. Such workloads tend to be highly (if not entirely) *harmonic* in nature. Two tasks are harmonic with respect to one another when the period of one task evenly divides the period of the other. Under harmonic workloads, it will commonly be the case that several (perhaps many) jobs are released at approximately the same time.

Consider the pathological example given in Fig. 3 (c). At time 1, tasks of levels A, B, and C are released. Their release timers fire in reverse-priority order, causing the following unfortunate sequence of events. First, the level-C task is released; then it is scheduled to run. Then, the level-B task is released; this causes rescheduling for the level-C task, and the level-B task is scheduled to run. Finally, the level-A task is released; this causes the level-B task to be preempted, and the level-A task is scheduled to run.

Our implementation supports a feature, called *timer merging*, to rectify this situation. When this feature is enabled, release events that will occur within 1 μs of one another are merged using an $O(1)$ hash table operation. This results in the behavior illustrated in Fig. 3 (d). The merging algorithm does

not easily scale across multiple processors, as synchronization issues would arise that would require expensive global locks. Thus, in our implementation, the timer merging feature can only be used in conjunction with the interrupt master feature (where timers only fire on a single processor), and we evaluate the two features as a single unit (later in Sec. 3.3).

Work redistribution. Recall Fig. 3 (a), in which a scheduling event for task T_B must move task T_C to the level-C ready-queue (while holding the container C lock) before T_B can execute. In such a case, a higher-criticality task is penalized for this overhead, performing work and acquiring locks on behalf of a lower-criticality task. This runs counter to our stated goals. Thus, our implementation supports a feature, which we name *work redistribution*, to offload this work to the interrupt master. More specifically, when a higher-criticality task preempts a lower-criticality task, the lower-criticality task is placed on a special-purpose local queue, and a notification is sent to the interrupt master (via an inter-processor interrupt) to requeue the lower-criticality task in the applicable container. The redistributed work is then included in the scheduling overhead of a task on the interrupt master instead of the overhead of the higher-criticality task. This process is illustrated in Fig. 3 (b).

3.3 Overhead Measurements

We collected release and scheduling overhead samples by executing three system configurations under our implementation of MC^2. Our task systems were designed to mimic workloads that could be seen on avionics systems. Each level-A task was randomly assigned a period of 25 ms, 50 ms, or 100 ms, in accordance with common periods used in avionics applications [10, 13]. Level-B periods were randomly selected to be harmonic with respect to the level-A hyperperiod and limited to a maximum of 300 ms. Level-C periods were randomly selected from the range $[10, 100]$ ms and rounded to the nearest multiple of five. We purposely selected smaller periods to thoroughly test MC^2; shorter periods result in more scheduling decisions per unit of time and therefore increase overhead. Similarly, we rounded level-C periods to increase the probability of multiple scheduling decisions occurring simultaneously, further increasing overhead.

In current avionics systems, highly-critical tasks represent only a small portion of the overall workload (usually, at most 20% of the overall capacity[5]). Assuming this trend continues in future systems, we evaluated three different capacity configurations for the level-A, -B, and -C sub-systems. The following three-tuples represent the (A, B, C) capacity configurations we evaluated: (5%, 5%, 65%), (10%, 10%, 55%), and (20%, 20%, 35%). For example, the first configuration denotes that level-A, -B, and -C utilizations are upper bounded by $0.05m$, $0.05m$, and $0.65m$, respectively. Task utilizations were obtained by generating a single execution time per task.

We varied the number of tasks running, n, from 20 to 120 in steps of 20. For each n, we generated 10 task systems per capacity configuration, for a total of 30 task systems for each value of n. Our results (shown below) were obtained by averaging the results from all 30 task systems for each experimental configuration.

Each task consisted of an independent program periodically running and performing arithmetic calculations on a 32 kB per-task array for the amount of time given by the task's execution time. Level-A and -B tasks were not allocated to the interrupt master for reasons explained in Sec. 3.2. Additionally, each processor executed a background task that repeatedly accessed a large array to emulate bus contention and cache misses appearing in a heavily loaded system.

Three experimental runs were performed for each group of 30 task systems, each with a different configuration of the features described in Sec. 3.2. In the first run, only the fine-grained locking feature was enabled. The second run was similar to the first, with the addition of the interrupt master and timer merging features. The third run was similar to the second, with the addition of the work redistribution feature (*i.e.*, all features enabled). Other potential configurations were either not feasible, since certain features require other features to be enabled, or are omitted due to space constraints.

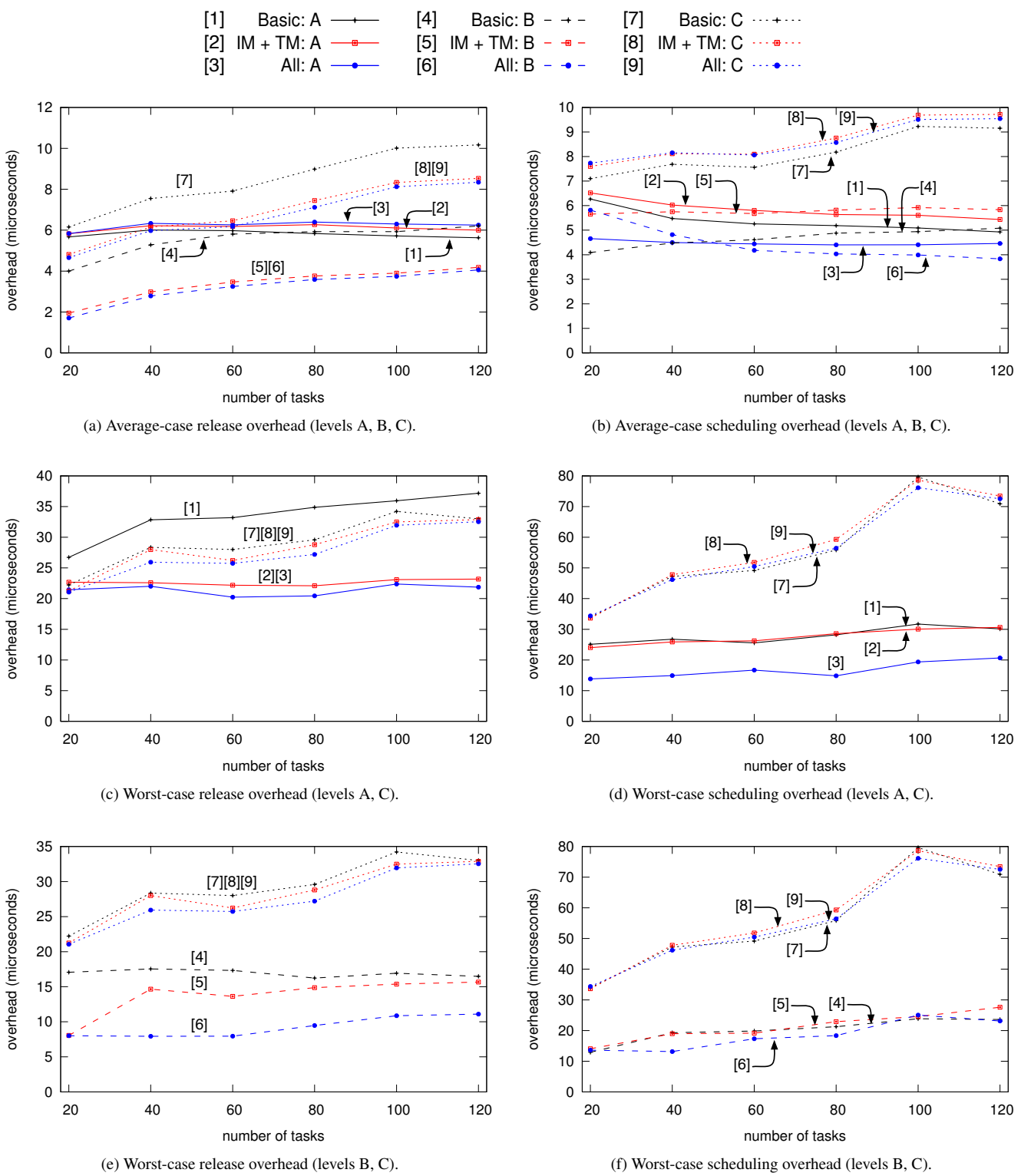
We performed our experiments on a six-core Intel Xeon processor chip running at 2.13 GHz. Overheads were collected using the Feather-Trace [5] tool, which imposes a small overhead of 61 instructions to collect a sample. In total, 338,899,980 release and scheduling overhead samples were recorded, consuming 5.05 GB of disk space.

Our measurements are plotted in Fig. 5. In this figure, the left-side insets give release overhead measurements and the right-side insets give scheduling overhead measurements. The "level-l overhead" is the overhead due to executing a release or scheduling handler for level-l. The top two graphs give average overheads for all levels, the middle two graphs give worst-case overheads for levels A and C, and the bottom two graphs give worst-case overheads for levels B and C. In the following paragraphs, we analyze the implications of the presented data for level-A, -B, and -C tasks separately. Following this, we discuss the effectiveness of our features with respect to the stated goals for our implementation mentioned at the beginning of Sec. 3.

Level A. In provisioning level-A tasks, the execution budget of each task will be pessimistically inflated to reflect uncertainty about the task's real execution cost *and* to reflect RTOS overheads. Thus, for level-A tasks, the worst-case overheads (in the middle two graphs of Fig. 5) are most relevant. Note that, in accounting for RTOS overheads, a level-A task T's execution budget must be inflated to account for its own scheduling and release overhead, as well as any release overhead incurred due to *other* tasks being released due to interrupts that occur on T's assigned processor.

Considering first the worst-case level-A scheduling overheads as shown in Fig. 5 (d), the value of our implementation techniques are readily apparent: enabling all of the proposed

[5]This estimate comes from private discussions with industry sources.



(a) Average-case release overhead (levels A, B, C).

(b) Average-case scheduling overhead (levels A, B, C).

(c) Worst-case release overhead (levels A, C).

(d) Worst-case scheduling overhead (levels A, C).

(e) Worst-case release overhead (levels B, C).

(f) Worst-case scheduling overhead (levels B, C).

Figure 5: Overhead measurements under three feature configurations, with trend lines added for clarity. "Basic" indicates the configuration with only the fine-grained locking optimizations. The "IM + TM" configuration adds to this the interrupt master and timer merging features. The "All" configuration additionally includes the work redistribution feature. Numeric labels ([1] through [9]) are printed near each curve for clarity. In a group of nearby curves, a set of labels printed horizontally indicates that the curves are ordered top-to-bottom when the labels are read left-to-right, *e.g.*, "[1][2][3]" indicates that curve [1] is above curve [2], and curve [2] is above curve [3]. Arrows pointing from labels to curves are provided when curves intersect or are very near one another.

features (*i.e.*, the curve "All (A)") reduces worst-case level-A scheduling overhead from approximately 25–32 μs to approximately 14–22 μs. The much lower average-case scheduling overheads in Fig. 5 (b) suggest that the worst-case values seen in Fig. 5 (d) occur rarely.

Understanding the implications of the release-overhead results for level A in Fig. 5 (c) is less straightforward. Under the "Basic" configuration (no release master), level-A tasks will be interrupted by the release handlers of level-B and -C tasks. Thus, level-A tasks must be penalized for their own release overhead (*i.e.*, "Basic: A" in Fig. 5 (c)), as well as some (potentially large) number of releases at any level (*e.g.*, "Basic: (B)"and "Basic: (C)" in Fig. 5 (c) and (e)). Fortunately, the situation *greatly* improves with the interrupt master enabled: level-A tasks *not* on the interrupt master are *never* penalized for lower-level task releases, because those releases occur on the interrupt master only. Thus, with all features enabled (*i.e.*, "All") such a task suffers only a single instance of scheduling and release overhead (both less than 22 μs).[6]

While the overheads for each event may seem like a small fraction of task execution time, in reality a single job's execution time must be inflated to account for a number of overhead sources. Consider a level-A task running in a system of 120 tasks. With no features enabled, the inputs to a (hypothetical) execution time analysis tool for this task would include a level-A scheduling cost of 32 μs and release event cost of 38 μs, a level-B release event cost of 17 μs, and a level-C release event cost of 34 μs, or 121 μs total. Even without additional pessimism added by an execution time analysis tool, this is 16% of 750 μs, one of the execution times used by level-A tasks in our system. With all features enabled, these inputs fall to a reduced level-A scheduling cost of 22 μs and release event cost of 22 μs, or 44 μs total (a 64% reduction).

Counterintuitively, some of the overhead curves trend slightly downward as the number of tasks in the system increases. This is most likely caused by decreased cache misses in the scheduler code due to more frequent scheduling and release overhead events [4]. This does not affect our conclusions and is not discussed further here.

Level B. Like level A, level B is HRT and thus worst-case overheads are most relevant. Also, like level A, a level-B task T's execution budget would be inflated to account for both a level-B schedule and release event as well as any release events for other levels that interrupt T's execution while it is running. With the interrupt master enabled, release events for other levels are not included, and level-B tasks only need to account for level-B release and scheduling overheads as shown in Fig. 5 (e) and Fig. 5 (f).

In Fig. 5 (e), we see that enabling all features results in a 30%–60% decrease in worst-case release overheads for level B. In Fig. 5 (f), level-B worst-case scheduling overheads are nearly unaffected ($n = 20$), improved by up to 35% ($n = \{40, 60, 80, 120\}$), or increased by only 3% ($n = 100$). Overall, we consider this an acceptable tradeoff.

Level C. Largely similar conclusions follow for level C, except that in this case, average-case overheads are more relevant (as level C is SRT). We need to consider average level-C scheduling overheads, as well as average release overheads for levels A, B, and C (recall that level-C tasks, being globally scheduled, are affected by all release overheads whether or not the interrupt master is enabled).

In Fig. 5 (a), we see that with our techniques enabled, average level-B release overhead is reduced by 2.2–2.3 μs (by 60% to 35%) and average level-C release overhead is reduced by 1.6–1.8 μs (by 25% to 18%). Level-A release overhead is increased slightly, by 0.1–0.8 μs (by 2% to 14%), likely due to increased overheads from timer merging. Finally, level-C scheduling overhead is increased by 0.4 μs , but this represents only 4% of the provisioned average scheduling cost. Given that level C is less critical and SRT, these improvements are too minor to significantly affect level-C performance.

Summary. Our analysis demonstrates that MC^2 can be supported in an RTOS-like environment (LITMUS$^{\mathrm{RT}}$) with relatively small overheads, in spite of the additional complexity introduced by needing to manage multiple criticality levels. Further, it shows that the proposed implementation features enable higher-criticality tasks to be largely shielded from overheads arising due to lower levels. Enabling all of these features reduces worst-case overheads at levels A and B while leaving level-C overheads essentially unaffected.

4 Robustness Evaluation

Mixed-criticality analysis allows a system designer to reclaim system capacity lost to execution-cost pessimism for highly-critical tasks. This capacity is reassigned to less-critical tasks, for which less pessimism is needed. In effect, the designer declares that she is willing to accept a greater risk of failure for lower-criticality tasks than higher-criticality tasks. In return for taking this risk, the system is more fully utilized.

It is presumed that the designer will provision execution times at each level so that the appropriate amount of risk is taken. However, in order to do so, she must know what happens when the "bet" she made does not pay off. In other words, what is the penalty to be paid when task failures cause *real* execution times to exceed the times assumed for some criticality level? If task performance degrades too abruptly, additional pessimism would have to be built into the system to compensate for this possibility. In the ideal case, the designer would like to see a graceful degradation of the performance of level-l tasks as level-l execution times are exceeded. Is this a realistic expectation?

In this section, we investigate this question in the context of our MC^2 implementation. Specifically, we give an MC^2 configuration and measure task behavior as execution times violate configuration assumptions. While we give what we

[6] In a real-world setting, the (relatively small) latency to send an interprocessor interrupt from the interrupt master to the task's assigned processor when it is released would also have to be considered.

consider to be a realistic MC^2 configuration, the configuration used is not as important as how task execution behavior degrades when configuration assumptions are violated.

Our MC^2 configuration is constructed as follows. Motivated by the characteristics of avionics systems, we assume that the level-A, -B, and -C subsystems consume 10%, 10%, and 55% of the system's capacity, similar to one of the configurations in Sec. 3. Further, we assume that level-C (resp., level-B) execution times are defined by profiling tasks and using observed average-case (resp., worst-case) values. We assume that level-A execution times are determined by a tool that adds additional pessimism. Based upon the differences between average- and worst-case observed overheads seen in Fig. 5, we assume that level-B execution times are ten times greater than level-C execution times.[7] Further, we assume that level-A execution times are twice level-B execution times. We have no way of justifying this choice, as timing-analysis tools for multicore systems currently do not exist.[8]

Our system models execution time assumption mismatches by having jobs determine *actual* execution times in one of two ways: executing for their average-case or drawing their execution time from a beta distribution modeling aberrant behavior. The beta distribution produces a value in the range (0,1), with a configurable mean and standard deviation. We configured our experiments such that distribution means of (arbitrarily close to) 1.0, 0.5, and 0.05 correspond to the assumed level-A, -B, and -C costs, respectively. Note that these values have the proper ratio mentioned above: the level-A cost is twice the level-B cost, which is in turn ten times the level-C cost. However, since a sample from the beta distribution returns a value in the range (0,1), the actual execution cost of a job is determined by using distribution samples to scale the level-A execution times. We initially configured the beta distribution used in experiments so that its mean was 0.05, *i.e.*, a level-C cost is obtained on average (as expected by the designer). We further configured the beta distribution's standard deviation so that the probability of obtaining a value larger than 0.5 is less than 1%, *i.e.*, the probability of exceeding the assumed level-B cost (which is an observed worst case) is low.

Modeling what happens when the designer's expectations are not met becomes a simple matter of "shifting" the parameters of the aberrant beta distribution so that its average progressively takes on values ranging from 0.05 to 1.0. For a fraction P of jobs, execution time is selected using the aberrant beta distribution average. This results in a sequence of progressively more difficult to schedule task systems. We generated two such sequences of task systems, a pathological one in which $P = 0.5$ (*i.e.*, a beta average higher than 0.05 was used with probability 0.5), and a more reasonable (*i.e.*, less pathological) one in which $P = 0.1$.

We executed these task systems on the same hardware platform considered in Sec. 3. In each experimental run, each task system was executed for one minute. Two metrics were recorded for criticality levels B and C.[9] The *deadline-miss ratio* is the fraction of all deadlines that were missed. The *average relative response time* is the average ratio of the response times of tasks to their periods. (This allows tardiness to be assessed in a unified way.) In total, 51,504,417 trace records were collected, consuming 1.15 GB of disk space.

In Fig. 6, the obtained results for these metrics are plotted. The insets in the left column show results for the $P = 0.5$ experiments; the insets in the right column show results for the $P = 0.1$ experiments. In each inset of this figure, the x-axis gives the assumed beta distribution average for each job with probability P. Thus, the task systems corresponding to $x = 0.05$ are those where (as expected by the designer) level-C execution costs occurred on average, $x = 0.5$ are those where level-B costs occurred on average with probability P, and $x = 1.0$ are those where level-A costs occurred on average with probability P.

Recall that our implementation of MC^2 supports optional budget enforcement, forcing tasks running at criticality level l to execute for no longer than their configured level-l execution budget. Our experimental data revealed that this feature has a significant effect on performance degradation when the system is under load. Thus, in Fig. 6, we plot curves for task systems both with and without budget enforcement enabled.

Level B. To understand the level-B system's resistance to failure, we must look at system behavior when the execution times of tasks exceed their level-B execution costs, *i.e.*, when x begins to exceed 0.5. As the level-B system is HRT, we are primarily concerned with the deadline miss ratio (insets (e) and (f) of Fig. 6).

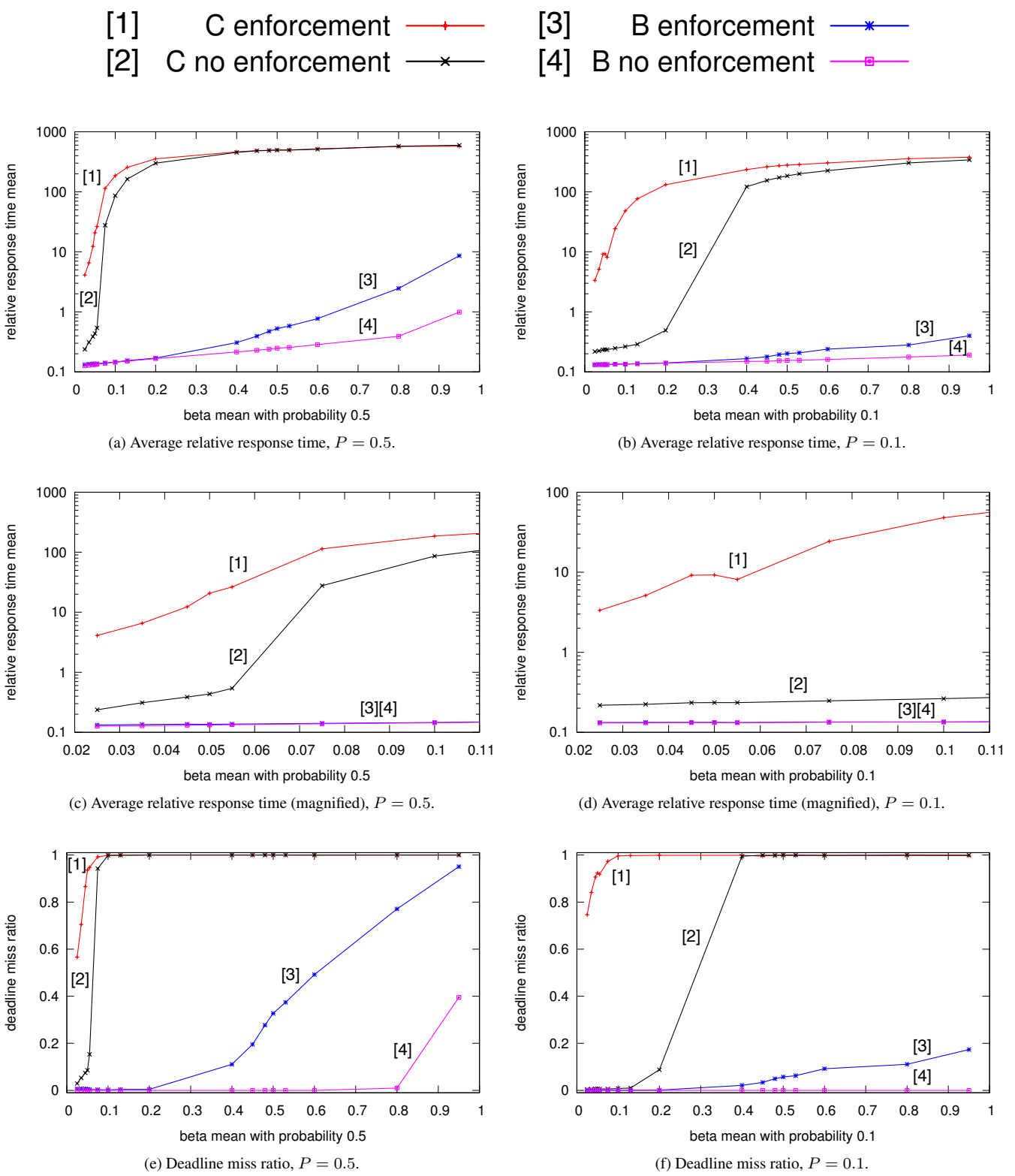
The level-B results illustrate the surprising flexibility shown by the system when budget enforcement is disabled. Without budget enforcement, overrunning tasks that would otherwise be forced to wait for their next release (and miss their deadlines) instead continue to run as the highest priority tasks in the system. This gives the system additional room to compensate for aberrant task execution behavior.

We can see this improvement in the $P = 0.5$ results of inset (e), where half of the jobs draw their execution times from the aberrant beta distribution. With budget enforcement, the deadline miss ratio rises to 10% at $x = 0.4$ and 50% at $x = 0.6$ (though in inset (a), we see that the average relative response time exceeds 1.0, *i.e.*, the average task is tardy, only for $x \geq 0.8$). However, without budget enforcement, we see that deadline misses begin to rise only after half of the jobs are executing for $x = 0.8$, on average (or 60% more than their level-B budget).

[7] We are not claiming that a ratio of ten is typical for an actual application, but merely explaining why we chose this value for levels B and C.

[8] Despite the availability of some such tools for uniprocessor systems, even today, in many avionics applications, worst-case execution-time estimates are often computed exactly as described here, *i.e.*, by multiplying an observed worst-case value by an arbitrary value.

[9] Level-A tasks never failed to execute within their cyclic executive scheduling windows in our system, and, thus, are not discussed in this section.



(a) Average relative response time, $P = 0.5$.

(b) Average relative response time, $P = 0.1$.

(c) Average relative response time (magnified), $P = 0.5$.

(d) Average relative response time (magnified), $P = 0.1$.

(e) Deadline miss ratio, $P = 0.5$.

(f) Deadline miss ratio, $P = 0.1$.

Figure 6: Scheduling metrics for our task system with different means for a "shifted" beta distribution, with trend lines included for clarity. Numeric labels ([1] through [4]) are printed near each curve for clarity. If two curves have nearly identical data points (*i.e.*, overlap), their labels are printed horizontally (*e.g.*, "[3][4]").

The system is even more flexible in the less pathological $P = 0.1$ case (inset (f)). With budget enforcement, the deadline miss ratio begins to rise as 10% of tasks approach their level-B execution times, though it never exceeds 0.2. Without budget enforcement, the deadline miss ratio hardly increases even when these tasks execute for their level-A execution time, on average. These results lead us to conclude that, without budget enforcement, the level-B system can maintain correctness in the presence of significant task execution failures before degrading entirely.

Level C. The analysis for level-C tasks is similar to level-B tasks, except that we are more concerned with relative response times, as level C is SRT and may be tardy by a bounded amount (*i.e.*, relative response times exceeding 1 are allowed). We focus on insets (c) and (d), which show relative response times as average task execution times drawn from the shifted distribution rise to 10% of our level-A execution budget, or twice the level-C execution budget (the data in these insets corresponds to that in (a) and (b) for $x \in (0, 0.1]$).

Level-C relative response times, like level-B deadline misses, are reduced with budget enforcement disabled. Thus, we focus on results without budget enforcement for level-C. In inset (d), where $P = 0.1$, level-C relative response times remain below 0.3 even when $x = 0.10$, or 10% of tasks are executing for an average of twice their level-C execution budget. With $P = 0.5$ in inset (c), performance degrades faster. Average relative response times reach 30.0 at $x = 0.075$, or when 50% of tasks are executing for 50% more than their level-C execution budget. After this, response times grow unboundedly. While level-C performance exhibits less flexibility in the $P = 0.5$ case, these high response times reflect both the pessimism of this experiment and the processor execution times devoted to higher-priority tasks to compensate.

5 Conclusion

In this paper, we have presented experimental results concerning the MC^2 mixed-criticality scheduling framework for multicore systems. To the best of our knowledge, this is the first paper on the implementation aspects of multicore mixed-criticality scheduling. Our research has shown that MC^2 can be implemented in a way that keeps RTOS-related overheads at acceptable levels, while largely shielding higher-criticality tasks from overheads created by lower-criticality tasks. It has also shown that MC^2 is quite robust with respect to mismatches in assumptions regarding execution-time estimates and actual execution times experienced at runtime.

In future work, we intend to extend MC^2 to also support task synchronization. In doing so, we hope to leverage recent work on asymptotically optimal real-time multiprocessor locking protocols [7, 8]. Such protocols will need to be adapted to minimize the impact lower-criticality tasks have on the blocking times experienced by higher-criticality tasks. In other future work, we hope to port our MC^2 design to an RTOS that is suitable for use in safety critical systems. While the open-source nature of LITMUS$^{\mathrm{RT}}$ makes it quite attractive for assessing various RTOS-related design alternatives, being based upon Linux, it is not a viable candidate for actual deployment in such systems. Although we believe that the conclusions reached in this paper are not Linux-specific, further experimentation with other RTOS choices, and even other hardware platforms, would strengthen these conclusions.

References

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

[2] T. Baker and A. Shaw. The cyclic executive model and ADA. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 120–129, December 1988.

[3] S. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 34–43, December 2011.

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

[5] B. Brandenburg and J. Anderson. Feather-trace: A light-weight event tracing toolkit. In *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 19–28, July 2007.

[6] 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.

[7] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 49–60, December 2010.

[8] B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *Proceedings of the International Conference on Embedded Software*, pages 69–78, October 2011.

[9] 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.

[10] C. Gill, R. Cytron, and D. Schmidt. Multi-paradigm scheduling for distributed real-time embedded computing. *Proceedings of the IEEE*, 91(1):183–197, 2003.

[11] J. Krodel and G. Romanski. Real-time operating systems and component integration considerations in integrated modular avionics systems report. Technical Report DOT/FAA/AR-07/39, U.S. Department of Transportation Federal Aviation Administration, August 2007.

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

[13] C. Locke, L. Lucas, and J. Goodenough. General avionics software specification. Technical Report CMU/SEI-90-TR-8, Carnegie Mellon University, December 1990.

[14] 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.

[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.