Recovering from Overload in Multicore Mixed-Criticality Systems

Jeremy P. Erickson, Namhoon Kim, and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill*

Abstract

The multicore revolution is having limited impact on safety-critical cyber-physical systems. The key reason is the "one out of m" problem: certifying the real-time correctness of a system running on m cores can necessitate pessimistic analysis that easily negates the processing capacity of the "additional" $m - 1$ cores. In safety-critical domains such as avionics, this has led to the common practice of simply disabling all but one core. In this paper, the usage of mixed-criticality *(MC) scheduling and analysis techniques is considered to alleviate such analysis pessimism. Under MC analysis, a* single *system with components of different criticality levels is viewed as a set of* different *per-criticality-level systems. More optimistic analysis assumptions are made when certifying lower criticality levels. Unfortunately, this can lead to transient overloads at these levels, compromising real-time guarantees. This paper presents the first multicore MC framework that addresses this problem. This framework makes scheduling decisions in a virtual time domain that can be "stretched" until the effects of a transient overload have abated. Such effects dissipate more quickly if virtual time is "stretched" more aggressively, but this may reduce the quality of the work performed. This tradeoff is analyzed experimentally herein.*

1 Introduction

The computational capabilities enabled by multicore chips are being leveraged to realize a wealth of new products and services across many application domains. One domain, however, stands out as being largely unaffected: *safety-critical* cyber-physical embedded systems. In seeking to enable the usage of multicore platforms in such systems, one issue looms larger than any other, namely a problem we call the "the one out of m" problem: due to cross-core interactions that are hard to predict, certifying the real-time correctness of a system running on m cores can necessitate provisioning system components so conservatively, the processing capacity of the "additional" $m - 1$ cores is entirely negated. That is, only "one core's worth of work" can be certified even though m cores are available. In safety-critical domains such as avionics, this has led to the common practice of simply disabling all but one core (which is the simplest solution, if only one core's worth of work can be supported anyway).

Mixed-Criticality Systems. In many safety-critical applications, tasks are partitioned among different criticality levels, where higher criticality levels require more stringent certification. In work on uniprocessor real-time systems, Vestal suggested leveraging such certification stringency differences to achieve less pessimistic system provisionings [24]. Specifically, he observed that, from the perspective of certifying the real-time requirements of a less critical task, the execution times assumed of more critical tasks are needlessly pessimistic. Thus, he proposed that schedulability tests[1] for less critical tasks be altered to incorporate less pessimistic execution times for more critical tasks.

More formally, in a system with L criticality levels, *each* task has an *assumed worst-case execution time* (*AWCET*) specified at *every* level, and L system variants are analyzed: in the level-ℓ variant, the real-time requirements of all level-ℓ tasks are verified with level-ℓ AWCETs assumed for *all* tasks (at any level).[2] The degree of pessimism in determining AWCETs is level-dependent: if level ℓ is of higher criticality than level ℓ', then level-ℓ AWCETs will generally be greater than level-ℓ' AWCETs. For example, when certifying the system at the highest criticality level, *provably correct* tool-produced upper bounds on execution times might be assumed, while when certifying at a lower level, *observed* worst-case times from profiling might be assumed. The task model resulting from Vestal's work has come to be known as the *mixed-criticality* (*MC*) *task model*.

MC^2. In ongoing work with colleagues at Northrop Grumman Corp. (NGC), we have been extending work on scheduling and analyzing MC task systems to obtain an MC framework that can be practically applied on multicore machines [12, 18]; we see such work as a potential key step towards addressing the "one out of m" problem. The framework we have developed has come to be known as MC^2 (mixed-criticality on multicore). MC^2 was designed with future unmanned aerial vehicles (UAVs) as a primary motivating use case. Basic implementation issues were previously considered by Herman et. al [12].

MC^2 supports four criticality levels, denoted A (highest) through D (lowest), as illustrated in Fig. 1. Levels A and B are scheduled on a per-processor basis using table-driven and earliest-deadline-first (EDF) scheduling, respectively. Level C was proposed by Mollison et al. [18] to be scheduled using the global earliest-deadline-first (G-EDF) scheduler, which provides *bounded response times* but may

*Work supported by NSF grants CNS 1016954, CNS 1115284, CNS 1218693, and CNS 1239135.

[1] Such tests are used to verify real-time deadline requirements.

[2] We stress that a *single* system is analytically viewed as *L separate* and *different* systems. Such a notion can seem a bit strange a first.

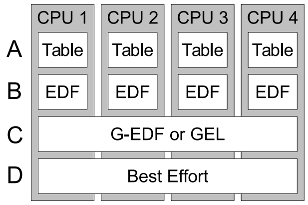


Figure 1: MC^2 architecture.

not meet all deadlines. Erickson et al. [9] demonstrated that the more general class of *G-EDF-like* (*GEL*) schedulers [16] can yield better response-time bounds. Therefore, we consider general GEL schedulers here. (These global schedulers are more fully described in Sec. 2.)

As noted by Burns and Davis [6], most proposed MC frameworks do not provide any guarantees for a given level ℓ if any job (i.e., task invocation) exceeds its level-ℓ AWCET. This assumption could be highly problematic in practice. For example, in a UAV, highly critical flight-control software might be supported at level A, and less critical mission management and planning software at level C. If a level-A flight-control job exceeds its level-C AWCET, then it would be very undesirable if no guarantees could be provided for level-C mission management tasks from that point forward.[3] In this paper, we present a method for providing guarantees in MC^2 in such situations. Specifically, we consider response-time behavior for tasks at level C in MC^2. (Tasks should be quite conservatively provisioned at levels A and B, thus breaches of level-A and -B AWCET assumptions should rarely, if ever, occur.)

Contributions. When any job at or above level C overruns its level-C AWCET, the system at level C may be *overloaded*, compromising level-C guarantees. Using the MC^2 framework, a task may have its per-job response times permanently increased as a result of even one overload event, and multiple overload events could cause such increases to build up over time. Examples of conditions that could cause this to happen are presented in Sec. 2. As a result, we must alter scheduling decisions to attempt to recover from transient overload conditions. In this paper, we propose a scheme that does so by scaling task inter-release times[4] and modifying scheduling priorities. We further discuss our implementation of this scheme, including both in-kernel and userspace components. We also provide experimental results based on our implementation to demonstrate that this scheme can effectively recover from overload.

Comparison to Related Work. Other techniques for managing overload have been provided in other settings, although most previously proposed techniques either focus exclusively on uniprocessors [2, 3, 7, 14, 17] or only provide heuristics without theoretical guarantees [11].

Our paper uses the idea of "virtual time" from Zhang [25] (as also used by Stoica et al. [21]), where job separation times are determined using a virtual clock that changes speeds with respect to the actual clock. In our work, we recover from overload by slowing down virtual time, effectively reducing the frequency of job releases. Unlike in [21], we never speed up virtual time relative to the normal underloaded system, so we avoid problems that have previously prevented virtual time from being used on a multiprocessor. To our knowledge, this work is the first to use virtual time in multiprocessor real-time scheduling.

Some past work on recovering from AWCET overruns in MC systems has used techniques similar to ours, albeit in the context of trying to meet all deadlines [13, 19, 20, 22, 23]. Our scheme is also similar to reweighting techniques that modify task parameters such as periods. A detailed survey of several such techniques is provided by Block [4].

Organization. In Sec. 2, we describe the task model used in prior work and show why overload can cause guarantees to be permanently violated. In Sec. 3, we describe our modified task model and scheduler, and discuss how it can be used to recover from overload. In Sec. 4, we describe our implementation, and in Sec. 5, we provide experimental evidence that our scheme is effective.

2 Original MC^2 And Overload

In this paper, we assume that time is continuous and we consider only the system at level C. In other words, we consider level-A and -B tasks as CPU time that is unavailable to level C, rather than as explicit tasks. We consider a system $\tau = \{\tau_0, \tau_1, \ldots, \tau_{n-1}\}$ of n level-C tasks running on m processors $P = \{P_0, P_0, \ldots, P_{m-1}\}$. Each invocation of τ_i is called a *job*, and the sequence of such jobs is denoted $\tau_{i,0}, \tau_{i,1}, \ldots$. The release time of $\tau_{i,k}$ is denoted as $r_{i,k}$. We assume that $\min_{\tau_i \in \tau} r_{i,0} = 0$. Each $\tau_{i,k}$ is prioritized on the basis of a *priority point* (*PP*), denoted $y_{i,k}$: a job with an earlier PP is prioritized over a job with a later PP. The time when $\tau_{i,k}$ actually completes is denoted $t^c_{i,k}$, and its *actual execution time* is denoted $e_{i,k}$. We define the *response time* $R_{i,k}$ of $\tau_{i,k}$ as $t^c_{i,k} - r_{i,k}$. We define a job $\tau_{i,k}$ as *pending* at time t if $r_{i,k} \leq t < t^c_{i,k}$.

We assume that τ is a sporadic task system (see below) and is scheduled via a *global scheduler*. A global scheduler schedules ready jobs from a single run queue[5] that is ordered by priority; a job is *ready* if it is pending and all prior jobs of the same task have completed. At any time, if j jobs are ready, then the $min(m, j)$ ready jobs of highest priority are scheduled; preemption and migration are allowed.

Under GEL scheduling and the conventional sporadic task model, each task τ_i is characterized by a per-job worst-case execution time (WCET) $C_i > 0$, a minimum separation $T_i > 0$ between releases, and a relative PP $Y_i \geq 0$; τ_i's *utilization* is given by C_i/T_i. The *utilization* of a task system is simply the sum of the utilizations of its tasks. Using

[3] At the industry session of RTAS 2014, several industry practitioners noted this as a practical concern that had not been adequately addressed in the literature on MC systems.

[4] That is, the time between subsequent job invocations

[5] That is, queue of available jobs

the above notation, the system is subject to the following constraints for every $\tau_{i,k}$:

$$e_{i,k} \leq C_i, \tag{1}$$

$$r_{i,k+1} \geq r_{i,k} + T_i, \tag{2}$$

$$y_{i,k} = r_{i,k} + Y_i. \tag{3}$$

In our work, we consider *assumed* WCETs due to the mixed-criticality analysis, as discussed in the introduction.

Prior work [15, 18] shows that bounded response times can be achieved for level-C tasks assuming certain constraints on system-wide and per-task utilizations. To illustrate this property, we depict in Fig. 2(a) a system that only has level-A and -C tasks, with one level-A task per CPU. For level-A tasks, we use the notation (T_i, C_i^C, C_i^A), where T_i is task τ_i's period, C_i^C is its level-C AWCET, and C_i^A is its level-A AWCET. For level-C tasks, we use the notation (T_i, Y_i, C_i). Observe that in Fig. 2(a), no job runs for longer than its level-C AWCET. Under this condition, response times can be bounded using techniques from prior work [15, 18]. In this paper, we typically concern ourselves with response times relative to a job's PP. Under the model we are defining in this section, such a response time can be converted to an absolute response time by adding Y_i. Observe that in Fig. 2(a) some jobs do complete after their PPs; this is allowed by our model. Similarly, some jobs complete after the release of their respective successor jobs.

The particular example in Fig. 2(a) fully utilizes all processors. In the situation depicted in Fig. 2(b), both level-A tasks released at time 12 run for their full level-A AWCETs. Therefore, from the perspective of level C, an overload occurs.[6] Because the system is fully utilized, there is no "slack" that allows for recovery from overload, and response times are permanently increased. In a system with large utilization, response times could take significant time to settle back to normal, even if they eventually will.

Another cause of overload is depicted in Fig. 3, where there is only a single level-C task. Observe that in Fig. 3(a) τ_1 executes except when both CPUs are occupied by level-A tasks. Therefore, when the overload occurs at time 12 in Fig. 3(b), τ_1 cannot recover despite the frequent presence of slack on the other CPU. This demonstrates that an overload can cause long-running problems due to a single task's utilization, not merely due to the total utilization of the system.

In the next section, we discuss our overload recovery.

3 Our Modifications

In order to recover from overload, it is necessary to effectively *reduce* task utilizations, to avoid the problems discussed in the previous section. In this paper, we propose to do so by using a notion of *virtual time* (as in [21]), as described in this section.

Our scheme involves a generalized 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*. Under the SVO model, we no longer assume a particular WCET (thus allowing overload). Therefore, (1) is no longer required to hold.[7] Under GEL-v scheduling and the SVO model, we also introduce the use of virtual time, and we define the minimum separation time and relative PP of a task with respect to virtual time after one of its job releases instead of actual time.[8] Virtual time affects *only* level C, not levels A and B. The use of virtual time will allow us to recover from overload. We now introduce our strategy using the example depicted in Fig. 2(c).

Once an overload occurs, the system can respond by altering virtual time for level C. Virtual time is based on a global speed function $s(t)$. During normal operation of the system, $s(t)$ is always 1. This means that actual time and virtual time progress at the same rate. However, after an overload occurs, the scheduler may choose to select $0 < s(t) < 1$, at which point virtual time progresses more slowly than actual time. In Fig. 2(c), the system chooses to use $s(t) = 0.5$ for $t \in [19, 29)$. As a result, virtual time progresses more slowly in this interval, and new releases of jobs are delayed. This allows the system to recover from the overload, so at actual time 29, $s(t)$ returns to 1. Observe that job response times are significantly increased after actual time 12 when the overload occurs, but after actual time 29, they are similar to before the overload. In fact, the arrival pattern of level A happens to result in better response times after recovery than before the overload, although this is not guaranteed under a sporadic release pattern.

An actual time t is converted to a virtual time using

$$v(t) \triangleq \int_0^t s(t)\, dt. \tag{4}$$

For example, in Fig. 2(c), $v(25) = \int_0^{25} s(t)\, dt = \int_0^{19} 1\, dt + \int_{19}^{25} 0.5\, dt = 19 + 3 = 22$. Unless otherwise noted, all instants herein (e.g., t, $r_{i,k}$, etc.) are specified in actual time, and all variables *except* T_i and Y_i (defined below) refer to quantities of actual time.

Under the SVO model, (2) generalizes to

$$v(r_{i,k+1}) \geq v(r_{i,k}) + T_i, \tag{5}$$

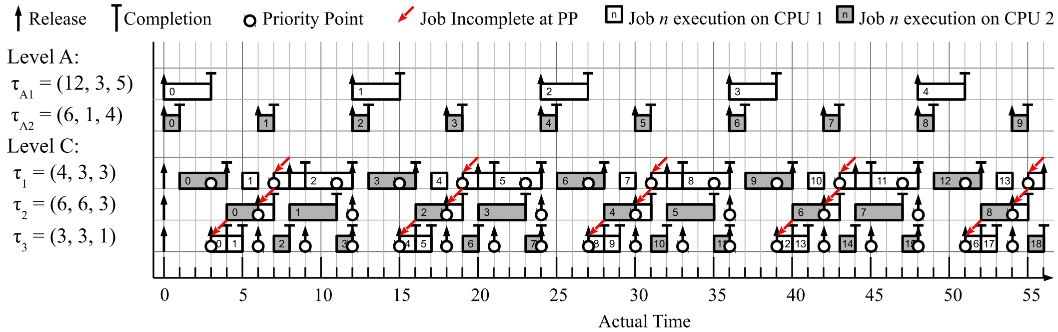
and under GEL-v scheduling, (3) generalizes to

$$v(y_{i,k}) = v(r_{i,k}) + Y_i. \tag{6}$$

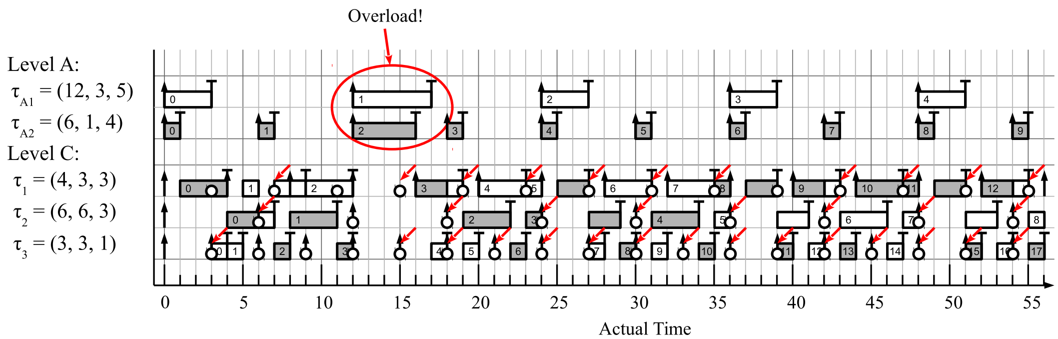
[6] A similar overload could occur if a level-C task exceeds its level-C AWCET. However, MC^2 optionally supports the use of execution budgets in order to prevent such an occurrence. While the use of execution budgets would prevent level-A and -B tasks from overrunning their level-A and -B AWCETs, respectively, they can still overrun their level-C AWCETs. Thus, we have chosen examples that provide overload even when execution budgets are used.

[7] As mentioned in Footnote 6, execution budgets can be used to restore this assumption at level C, in which case overloads can come only from levels A and B.

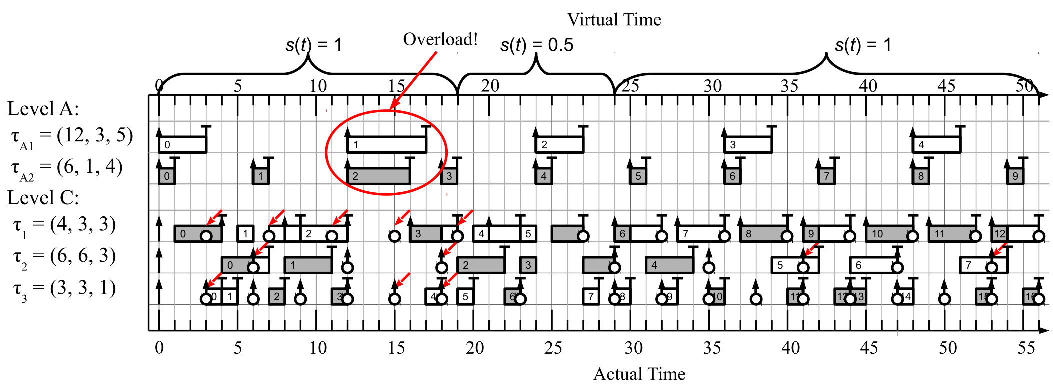
[8] If jobs of a task are released in response to external events, violations of the minimum separation constraint can be avoided by delaying the actual job release, setting a timer as if the job were released periodically.



(a) Example MC[2] schedule in the absence of overload, illustrating bounded response times.



(b) The same schedule in the presence of overload caused by level-A tasks running for their full level-A AWCETs. Notice that response times of level-C jobs settle into a pattern that is degraded compared to (a). For example, consider $\tau_{2,6}$, which is released at actual time 36. In (a) it completes at actual time 43 for a response time of 7, but in this schedule it does not complete until actual time 46, for a response time of 10.



(c) The same schedule in the presence of overload and the recovery techniques described in Sec. 3. Notice that response times of level-C jobs settle into a pattern that is more like (a) than (b). For example, consider again $\tau_{2,6}$, which now is not released until actual time 41 and completes at actual time 47 for a response time of 6. This is more similar to (a) than to (b).

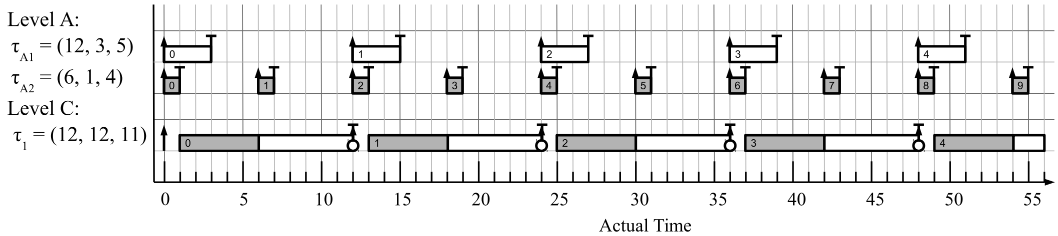
Figure 2: Example MC[2] task system, without and with overload.

For example, in Fig. 2(c), $\tau_{1,0}$ is released at actual time 0, has its PP three units of (both actual and virtual) time later at time 3, and $\tau_{1,1}$ can be released four units of (both actual and virtual) time later at time 4. However, $\tau_{1,5}$ of the same task is released at actual time 21, shortly after the virtual clock slows down. Thus, its PP is at actual time 27, which is three units of *virtual* time after its release, and the release of $\tau_{1,6}$ can be no sooner than actual time 29, which is four units of *virtual* time after the release of $\tau_{1,5}$. The execution time of $\tau_{1,5}$ is not affected by the slower virtual clock.

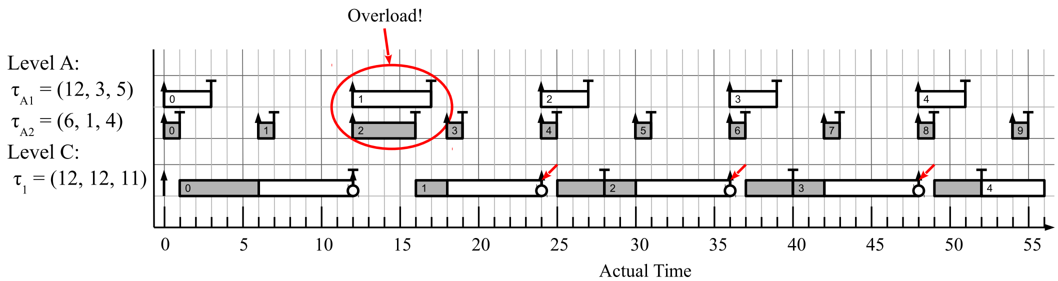
In a real system, unlike in our examples so far, level-C jobs will often run for less time than their respective level-C AWCETs. Therefore, it may be unnecessarily pessimistic to initiate overload response whenever a job overruns its level-C AWCET. Instead, we use the following definition.

Def. 1. τ_i has a nonnegative *response-time tolerance*, denoted ξ_i, relative to each job's PP. A task *meets* its response-time tolerance if $t^c_{i,k} \le y_{i,k} + \xi_i$, and *misses* it otherwise.

We slow down the virtual clock only after some job



(a) Example MC^2 schedule in the absence of overload, illustrating bounded response times.



(b) The same schedule in the presence of overload caused by level-A tasks running for their full level-A AWCETs.

Figure 3: Another example MC^2 task system, without and with overload. See Fig. 2 for key.

misses its response-time tolerance. Ideally, response-time tolerances should be determined based on analytical upper bounds of job response times, in order to guarantee that the virtual clock is never slowed down in the absence of overload. However, for illustration, in Fig. 2(c) we simply use a response-time tolerance of three for each task. Thus, we do not slow down virtual time until some job's completion time is greater than three units of actual time after its PP. At time 18, $\tau_{3,4}$ completes exactly three units after its PP, which is barely within its tolerance, so the virtual clock is not slowed down. However, at time 19, $\tau_{1,3}$ completes four units after its PP, which exceeds the response-time tolerance. Therefore, we slow down the virtual clock at time 19.

We will define *normal behavior* for a system as the situation in which all jobs meet their response-time tolerances. Recall that, as depicted in Fig. 2 above, a system with high utilization may not effectively be able to recover from overload, because there is no slack, and as depicted in Fig. 3 above, a system with a task of high utilization may not be able to effectively recover from overload. As we have just discussed, our technique creates extra slack both in a system-wide sense and in a per-task sense, solving both problems. Therefore, the system eventually returns to normal behavior. We denote the time required to do so as *dissipation time*.

In a technical report [8], we provide theoretical analysis of dissipation time. In that technical report, we first provide analytical upper bounds on response time that can safely be used as response-time tolerances. We also derive an upper bound on dissipation time, called a *dissipation bound*, with respect to these response-time tolerances. As discussed above, our technique causes the system to eventually return to normal behavior, so this bound exists. In this paper, rather than considering theoretical dissipation bounds, we focus on experimentally determining dissipation time at runtime.

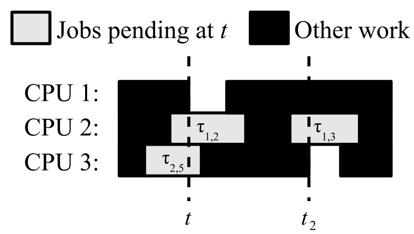


Figure 4: Illustration of "idle normal instant." If all jobs pending at t meet their response-time tolerances, then t is an idle normal instant. t_2 is referenced in Sec. 4.

In the analysis used in our technical report, demand for CPU time is considered beginning at an instant when some processor is idle. If all jobs pending at this time meet their response-time tolerances, then regardless of how that situation arose, the system has returned to normal behavior. Furthermore, the virtual clock can safely be returned to speed 1 after such an instant. Therefore, the system can detect such an instant to determine when to set the virtual-clock speed back to 1. We now define such an instant more formally.

Def. 2. Arbitrary time t is an *idle normal instant* if some processor is idle at t and all jobs pending at t meet their (normal) response-time tolerances.

Our method does not dictate a particular choice of $s(t)$, although in our experiments we consider several such values. Selecting a small value of $s(t)$ will result in a large short-term impact on level-C job releases, but the system will return to normal behavior quickly. For example, if $s(t)$ is assigned to be 0.1, then the system only allows a tenth as many level-C job releases as it normally does. Alternatively, selecting a large value of $s(t)$ will result in a lesser short-term impact on job releases, causing only minor delays, but the system will take a longer time in order to return to normal behavior. In our experimental comparison, we quantify these effects and point to proper design decisions.

As suggested by the analysis in our technical report [8], we determine when the system returns to normal behavior by detecting an idle normal instant. Therefore, we return the virtual clock to speed 1 after detecting such a t, which can only be determined when all jobs pending at t are complete. In Fig. 2(c), observe that only CPU 2 is executing work from actual time 28 to actual time 29. Thus, only τ_2 is pending throughout this interval, or CPU 1 would be executing work. Furthermore, $\tau_{2,4}$ is the only pending job of τ_2 at time 28. Observe that $\tau_{2,4}$ completes at its PP, and thus meets its response-time tolerance of three, at time 29. Therefore, time 28 is an idle normal instant. The system can determine this to be the case at time 29, when $\tau_{2,4}$ completes. Therefore, the virtual clock returns to speed 1 at time 29.

4 Implementation Description

We implemented our scheme by extending the existing MC^2 implementation that was described in [12]. That implementation is based on LITMUS$^{\mathrm{RT}}$ [1], a real-time extension to Linux originally developed at UNC. Source code for our implementation is also available at [1]. Our implementation consists of two components: the scheduler, which is part of the kernel, and a *monitor program*, which runs in userspace. The kernel reports job releases and job completions to the monitor program and provides a system call that the monitor program can use to change the speed of the virtual clock. The speed of the virtual clock does not change between these calls. The kernel is responsible for implementing virtual time, ensuring that the SVO model's minimum-separation constraints are respected, and making scheduling decisions according to GEL-v scheduling. The monitor program is responsible for determining when virtual-clock speed changes should occur. In an online appendix [10], we provide pseudocode for the functionality we changed.

Within the kernel, the primary change that we made compared to the prior MC^2 implementation was the use of virtual time at level C. No changes at levels A or B were required. Because the virtual-clock speed is constant between discrete changes, virtual time is a piecewise linear function of actual time, as depicted in Fig. 5(a), where t_s (speed change) is the latest speed change before arbitrary time t. The kernel can compute $v(t)$ by keeping track of t_s and $v(t_s)$. If the virtual-clock speed was changed to s at t_s, then $v(t) = v(t_s) + s \cdot (t - t_s)$.

At any time before a job's PP arrives, the kernel does not know whether the virtual-clock speed will change before that PP arrives. Thus, the kernel cannot compute $y_{i,k}$ upon the release of $\tau_{i,k}$. However, the scheduling priority of $\tau_{i,k}$ is simply the virtual time $v(y_{i,k})$. When $\tau_{i,k}$ is released at time $r_{i,k}$, the kernel can compute $v(r_{i,k})$. Then (6) can be used to compute $v(y_{i,k})$.

However, recall that the definition of "response-time tolerance" in Def. 1 is based on the actual time $y_{i,k}$. Therefore, it will generally be necessary for the kernel to determine $y_{i,k}$ and return it to the monitor program. It turns out that the kernel can perform this computation when it is already running for other reasons.

The precise time that $y_{i,k}$ is computed can be determined by considering three cases, as depicted in Fig. 5(b)–(d).

If $t^c_{i,k} \leq y_{i,k}$, as depicted in Fig. 5(b), then $\tau_{i,k}$ meets its response-time tolerance (which was defined in Def. 1 to be nonnegative) by definition. Therefore, it is sufficient to return a placeholder to the monitor program in this situation. It is not actually necessary to compute $y_{i,k}$.

If $t^c_{i,k} > y_{i,k}$ and the speed of the virtual clock changes at least once between $y_{i,k}$ and $t^c_{i,k}$, then this scenario is depicted in Fig. 5(c), where t_s now refers to the first virtual-clock speed change *after* $y_{i,k}$. In this case, $y_{i,k}$ is computed when the virtual-clock speed is changed at time t_s.

If $t^c_{i,k} > y_{i,k}$ and the speed of the virtual clock does not change between $y_{i,k}$ and $t^c_{i,k}$, then this scenario is depicted in Fig. 5(d). In this case, $y_{i,k}$ is computed when the job is completed.

In order to set the release timer for a level-C job, the kernel speculatively assumes that the virtual-clock speed will not change until the timer fires. Whenever the virtual-clock speed does change, the kernel updates all pending release timers based on the new speed.

The userspace monitor program has two functions. One is to detect that some job has missed its response-time tolerance and to react by changing the virtual-clock speed. The specific methods it uses to react are discussed below.

The second function of the userspace monitor program is to detect the earliest possible idle normal instant. We provide the following definition, which is closely related to the definition of "idle normal instant" in Def. 2.

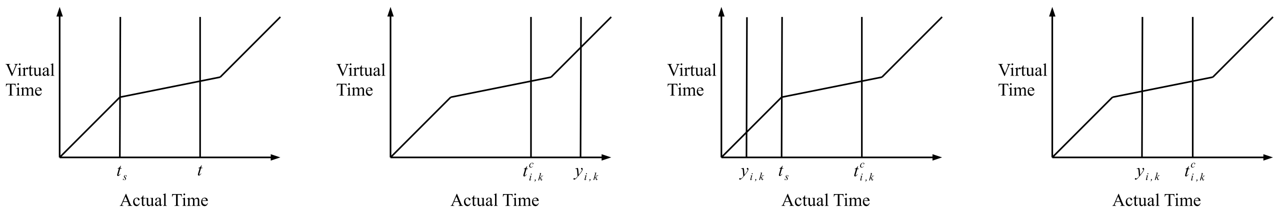
Def. 3. t is a *candidate idle instant* at time $t_2 \geq t$ if some processor is idle at t and any job pending at t either meets its response-time tolerance or is still pending at t_2.

In Fig. 4, t is a candidate idle instant at t_2 even if $\tau_{1,3}$ misses its response-time tolerance, as long as $\tau_{1,2}$ and $\tau_{2,5}$ meet their response-time tolerances.

The following theorem shows that we may consider only one candidate idle instant at any given time and still find the earliest idle normal instant. In Fig. 4, t_2 was selected as a time when a processor becomes idle, in order to illustrate this theorem.

Theorem 1. *If t is a candidate idle instant at t_2 and t_2 is an idle normal instant, then t is an idle normal instant.*

Proof. Because t is a candidate idle instant, by Def. 3, every job pending at t that is no longer pending at t_2 meets its response-time tolerance. Furthermore, because t_2 is an idle normal instant, by Def. 2, every job that is still pending at t_2 meets its response-time tolerance. Therefore, every job pending at t meets its response-time tolerance. Furthermore, because t is a candidate idle instant, by Def. 3, some processor is idle at t. Therefore, by Def. 2, t is an idle normal instant. □



(a) Example depicting how actual to virtual time conversion is done.

(b) Example depicting $y_{i,k}$ when $t^c_{i,k} \leq y_{i,k}$.

(c) Example depicting $y_{i,k}$ when $t^c_{i,k} > y_{i,k}$ and at least one speed change occured between $y_{i,k}$ and $t^c_{i,k}$.

(d) Example depicting $y_{i,k}$ when $t^c_{i,k} > y_{i,k}$ and no speed change occured between $y_{i,k}$ and $t^c_{i,k}$.

Figure 5: Examples illustrating virtual time computations in the kernel.

In order to detect an idle normal instant, we simply keep track of the earliest candidate idle instant and the set of incomplete jobs pending at that time, or the fact that there is no candidate idle instant. If all jobs that were pending at a particular candidate idle instant complete within their response-time tolerances, then that time was actually an idle instant. On the other hand, if some job $\tau_{i,k}$ pending at that time misses its response-time tolerance, then there can be no idle normal instant before $t^c_{i,k}$. Whenever a CPU becomes idle, if there is no current candidate idle instant, the current time is recorded as the candidate idle instant.

We refer to our first userspace monitor program as SIMPLE. It is given the response-time tolerances desired for the tasks and a virtual time speed $s(t)$ used for overload recovery. When a response-time tolerance miss is detected while the system is not in recovery mode, it simply slows down the virtual clock and starts looking for an idle instant.

We refer to our second userspace monitor program as ADAPTIVE. It allows a value of $s(t)$ to be determined at runtime, selecting a smaller value for a more significant response-time tolerance miss. This minimizes the impact on the system when only a minor response-time tolerance miss has occured, but provides a more drastic response when a larger miss has occured. The monitor accepts an *aggressiveness factor* a in addition to the set of response-time tolerances, providing additional tuning. Once a response-time tolerance violation is detected, the monitor maintains the invariant that $s(t) = a \cdot \min((Y_i + \xi_i)/R_{i,k})$, where the min is over all jobs with $t^c_{i,k}$ after recovery mode last started. Thus, it chooses the speed based on the largest observed response time since the virtual-time clock was last at normal speed.

5 Experiments

When a designer provisions an MC system, he or she should select level-C AWCETs that will be infrequently violated. Therefore, in the most common cases, overload conditions should be inherently transient, and it should be possible to return the system to normal operation relatively quickly. Therefore, our experiments consist of transient overloads rather than continuous overloads.

We ran experiments on a system with one quad-core 920-i7 CPU at 2.67 GHz, with 4GB of RAM. We generated 20 task sets, using a methodology similar to that described in [12], which used task systems designed to mimic avionics. We generated task systems where levels A and B each occupy 5% of the system's processor capacity and level C occupies 65% of the system's capacity, *assuming that all jobs at all levels execute for their level-C AWCETs.* As in [12], we assumed that each task's level-B AWCET is ten times its level-C AWCET, and that its level-A AWCET is twenty times its level-C AWCET.[9] The resulting percentages of the system occupied assuming different AWCETs is depicted in Fig. 6(a).

At levels A and B, we generated tasks on one CPU at a time, and at level C, we generated tasks for the entire system at once. In all cases, we used the system fractions in the third column of Fig. 6(a). Periods were selected as depicted in Fig. 6(c). We then selected, for each task, a utilization (at its own criticality level) uniformly from $(0.1, 0.4)$. This is the "uniform medium" distribution from prior work, e.g., [5]. The resulting distributions with AWCETs at different levels are depicted in Fig. 6(b). When a task would not fit within the capacity for its criticality level, its utilization was scaled down to fit. Each task was then assigned a level-C AWCET based on multiplying its level-C utilization by its period.

Y_i was selected for each level-C task using G-FL, which provides better response time bounds than G-EDF [9]. To determine response-time tolerances, we used the analytical bounds described in our technical report [8].

We tested the following overload scenarios:

- (SHORT) - All jobs at levels A, B, and C execute for their level-B AWCETs for 500 ms, and then execute for their level-C AWCETs afterward.
- (LONG) - All jobs at levels A, B, and C execute for their level-B AWCETs for 1 s, and then execute for their level-C AWCETs afterward.
- (DOUBLE) - All jobs at levels A, B, and C execute for their level-B AWCETs for 500 ms, execute for their level-C AWCETs for one second, execute for their

[9] In reality, level-A AWCETs likely need to be determined based on tools that do not currently exist, resulting in the need to guess the ratios between AWCETs.

	Level A	Level B	Level C
Level A	100%	50%	5%
Level B	0%	50%	5%
Level C	0%	0%	65%

(a) Percentage of the system occupied by the level specified in each row, when all jobs execute for the AWCET specified in the column.

	Level A	Level B	Level C
Level A	$(0.1, 0.4)$	$(0.05, 0.2)$	$(0.005, 0.02)$
Level B	$(0.2, 0.8)$	$(0.1, 0.4)$	$(0.01, 0.04)$
Level C	$(2.0, 8.0)$	$(1.0, 4.0)$	$(0.1, 0.4)$

(b) Uniform utilization distribution for a task at the level specified in each row, when measured using the AWCET specified in the column.

Level A	Randomly from $\{25 \text{ ms}, 50 \text{ ms}, 100 \text{ ms}\}$.
Level B	Random multiple of the largest level-A period on the same CPU, capped at 300 ms.
Level C	Random multiple of 5 ms between 10 ms and 100 ms, inclusive.

(c) Selection of periods for tasks at each level, which is independent of the level of the AWCETs.

Figure 6: Parameters used for experiments.

level-B AWCETs for another 500 ms, and then execute for their level-C AWCETs afterward.

As can be seen in Fig. 6(a), these are particularly pessimistic scenarios in which all CPUs are occupied by level-A and -B work for almost all of the time during the overload.

For each overload scenario, we used SIMPLE with $s(t)$ choices from 0.1 to 1 in increments of 0.1. *The choice of* $s(t) = 1$ *does not use our overload management techniques at all and provides a baseline for comparison.* We also used ADAPTIVE with a choices from 0.1 to 1.0 in increments of 0.1. We then recorded the minimum virtual-time speed (to analyze ADAPTIVE) and the dissipation time. We averaged each result over all twenty generated task sets.

In Fig. 7, we depict the average dissipation time using SIMPLE with respect to the choice of $s(t)$ during recovery. Additionally, we depict error bars for 95% confidence intervals. Under LONG, dissipation times are approximately twice as long as under SHORT, because overhead occurs for twice as long. Under DOUBLE, dissipation times are bigger than under SHORT for $s(t) \geq 0.9$, but nearly identical for smaller choices of $s(t)$. This occurs because dissipation time is measured from the end of the *second* (and final) interval during which overload occurs. For sufficiently small choices of $s(t)$, the system usually recovers completely before the second interval of overload starts, and that interval is the same length as in SHORT. In any case, a reduction of at least 50% of the dissipation time can be achieved with a choice of $s(t) = 0.6$, and with that choice, the dissipation time is less than twice the length of the interval during which overload occurs. Smaller choices of $s(t)$ have diminishing returns, with only a small improvement in dissipation time. Such a small improvement is likely outweighed by the larger impact on job releases from selecting a smaller $s(t)$. (Recall that the maximum number of job releases during an interval is proportional to $s(t)$.)

In Fig. 8, we depict the average dissipation time using ADAPTIVE with respect to the aggressiveness factor. As before, we depict error bars for 95% confidence intervals. There is significant variance in the initial choice of $s(t)$ by ADAPTIVE, depending on which level-C jobs complete first after the overload starts, resulting in the larger confidence intervals. This effect is particularly pronounced in the case of DOUBLE. By comparing Figs. 7 and 8, we see that ADAPTIVE significantly reduces the dependency of dissipation time on the length of the overload interval. Furthermore, dissipation times are often significantly smaller under ADAPTIVE than under SIMPLE.

However, in order to fully evaluate ADAPTIVE, we must consider the minimum $s(t)$ value it chooses. Fig. 9 depicts the average of this choice with respect to the aggressiveness value, in addition to 95% confidence intervals. Here, we see that ADAPTIVE achieves smaller dissipation times than SIMPLE by choosing significantly slower virtual-clock speeds. Thus, jobs are released at a drastically lower frequency during the recovery period. Therefore, under the highly pessimistic scenarios we considered, SIMPLE is a better choice than ADAPTIVE.

As discussed above, level-C tasks run very little during the overload, so jobs pending at the end of the overload dominate other jobs in producing the largest response times. Because ADAPTIVE usually results in complete recovery from overload before the second overload interval, this causes nearly identical minimum choices of $s(t)$ between SHORT and DOUBLE. Similarly, because the overload interval is twice as long under LONG than under SHORT, the minimum choice of $s(t)$ is about half under LONG compared to SHORT.

In summary, the best choice of monitor under the tested conditions was SIMPLE with $s(t) = 0.6$, but $s(t) = 0.8$ could be a good choice if it is preferable to have a smaller impact on new releases with a longer dissipation time.

In Fig. 10, we depict the execution time of job releases at level C using SIMPLE with $s(t) = 0.6$, with respect to the wall clock time. The pattern of job releases can be seen from where vertical lines appear. At time 1000, execution times of tasks are increased due to the overload. Therefore, the system enters recovery mode. In Fig. 10, intervals are marked at the top by $s(t)$ values. As can be seen in Fig. 10, level-C jobs are released less frequently in $[1000 \text{ ms}, 2100 \text{ ms}]$. At time 2100, an idle normal instant has been detected, so $s(t)$ is returned to 1, and the job release pattern returns to normal.

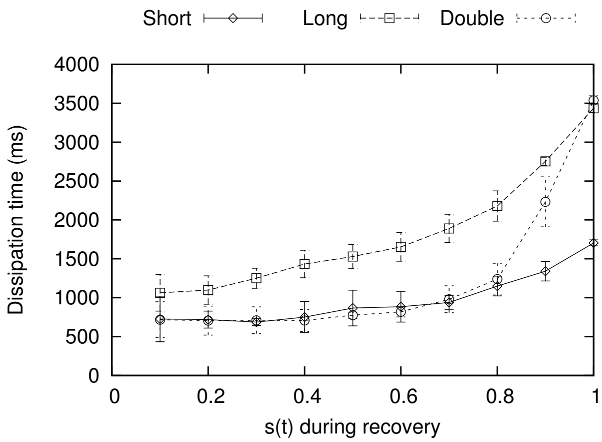


Figure 7: Dissipation time for SIMPLE

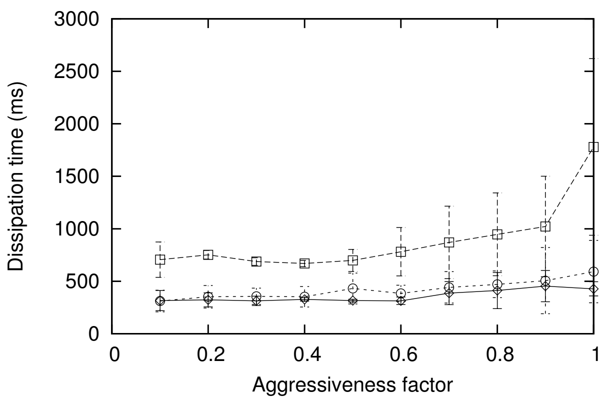


Figure 8: Dissipation time for ADAPTIVE

In Fig. 11, we depict response times (rather than execution times) with respect to wall clock time, with two different choices of $s(t)$. Once the overload begins at time 1000, response times are increased in both cases. By comparing parts (a) and (b), we see that our technique reduces the dissipation time and shrinks response times much more rapidly.

We also measured the same overheads considered in [12] both with and without our virtual time mechanism present, and considering both average and maximum observed overheads. For most overheads considered, there was no significant difference from the virtual time mechanism. However, there was variance in the scheduling overheads, as depicted in Fig. 12. For average-case overheads, the introduction of virtual time increased the scheduling time by about 40%, while for worst-case overheads, the introduction of virtual time approximately doubled the scheduling time. Because level C is SRT, the average-case overheads are more relevant, and the cost of adding the virtual time mechanism is small. Furthermore, the userspace monitor program had a CPU share of approximately 10%, less than a typical task.

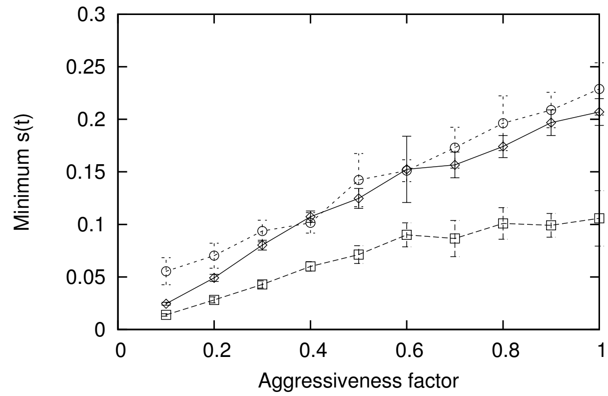


Figure 9: Minimum $s(t)$ for ADAPTIVE

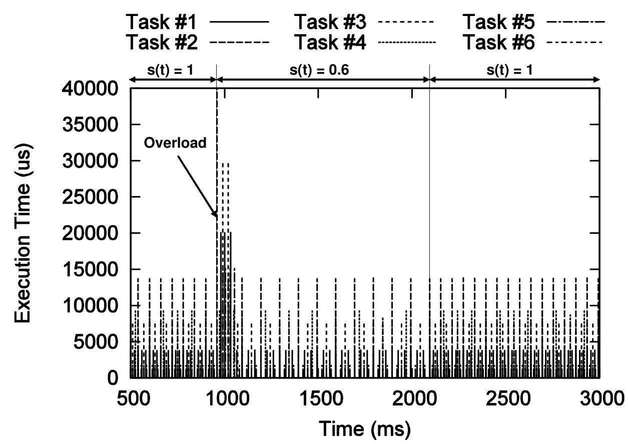


Figure 10: Allocations of tasks at level C for an example run

6 Conclusion

In this paper, we addressed the problem of scheduling under MC^2 when a transient overload occurs. We discussed the conditions that could cause an overload to result in a long-running increase in response-time bounds, and proposed a virtual-time mechanism to deal with these conditions.

We then presented an implementation of our mechanism and provided experiments to demonstrate that it can effectively provide recovery from unexpected overload scenarios. In our experiments, dissipation times could be brought within twice the length of a pessimistic overload scenario by only moderately affecting the time between job releases, and our scheme created little additional overhead.

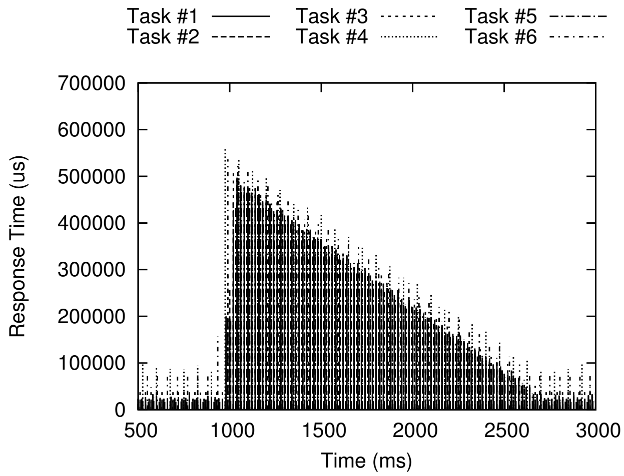
References

[1] LITMUS^{RT} home page. `http://litmus-rt.org`.

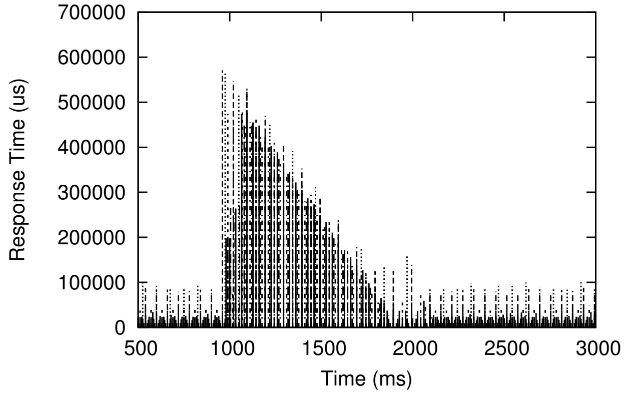
[2] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha. On-line scheduling in the presence of overload. In *FOCS*, pages 100–110, 1991.

[3] G. Beccari, S. Caselli, M. Reggiani, and F. Zanichelli. Rate modulation of soft real-time tasks in autonomous robot control systems. In *ECRTS*, pages 21–28, 1999.

[4] A. Block. *Adaptive Multiprocessor Real-Time Systems*. PhD



(a) Using $s(t) = 1.0$ (as if our techniques were not used)



(b) Using $s(t) = 0.6$

Figure 11: Response times for level-C jobs with respect to time

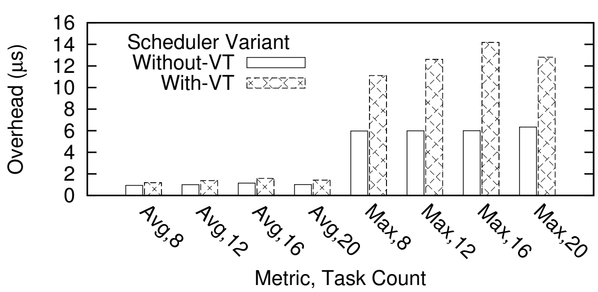


Figure 12: Scheduling overhead measurements

thesis, The University of North Carolina at Chapel Hill, 2008.

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

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

[7] G. Buttazzo and J. Stankovic. RED: Robust earliest deadline scheduling. In *3rd International Workshop on Responsive Computing Systems*, pages 100–111, 1993.

[8] J. Erickson and J. Anderson. Dissipation bounds: Recovering from overload in multicore mixed-criticality systems. Technical Report TR14-001, Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC, May 2014.

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

[10] J. Erickson, N. Kim, and J. Anderson. Appendix to Recovering from overload in multicore mixed-criticality systems. `http://cs.unc.edu/~anderson/papers.html`, October 2014.

[11] P. Garyali. On best-effort utility accrual real-time scheduling on multiprocessors. Master's thesis, The Virginia Polytechnic Institute and State University, 2010.

[12] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson. RTOS support for multicore mixed-criticality systems. In *RTAS*, pages 197–208, 2012.

[13] M. Jan, L. Zaourar, and M. Pitel. Maximizing the execution rate of low criticality tasks in mixed criticality systems. In *WMC, RTSS*, pages 43–48, 2013.

[14] G. Koren and D. Shasha. D^{over}: An optimal on-line scheduling algorithm for overloaded real-time systems. In *RTSS*, pages 290–299, 1992.

[15] H. Leontyev and J. H. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Sys.*, 44(1):26–71, 2010.

[16] H. Leontyev, S. Chakraborty, and J. Anderson. Multiprocessor extensions to real-time calculus. In *RTSS*, pages 410–421, 2009.

[17] C. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986.

[18] M. Mollison, J. Erickson, J. Anderson, S. Baruah, and J. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *ICESS*, pages 1864–1871, 2010.

[19] F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp. In *ECRTS*, pages 155–165, 2012.

[20] F. Santy, G. Raravi, G. Nelissen, V. Nélis, P. Kumar, J. Goossens, and E. Tovar. Two protocols to reduce the criticality level of multiprocessor mixed-criticality systems. In *RTNS*, pages 183–192, 2013.

[21] 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 *RTSS*, pages 288–299, 1996.

[22] H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *DATE*, pages 147–152, 2013.

[23] H. Su, D. Zhu, and D. Mosse. Scheduling algorithms for elastic mixed-criticality tasks in multicore systems. In *RTCSA*, 2013.

[24] 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. IEEE, December 2007.

[25] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching. In *SIGCOMM*, pages 19–29, 1990.

A Appendix: Psuedocode for Changed Kernel Functionality and Monitor Program

Function `initialize()`
- last_act := `now()`;
- last_virt := 0;
- speed := 1;

Function `act_to_virt(act)`
- **return** last_virt + (act − last_act) · speed;

Function `virt_to_act(virt)`
- **return** last_act + (virt − last_virt)/speed;

Function `schedule_pending_release`($\tau_{i,k}$, $v(r_{i,k})$)
- Set release timer to fire at `virt_to_act`($v(r_{i,k})$);

Function `job_release`($\tau_{i,k}$)
- $r_{i,k}$:= `now()`;
- $v(y_{i,k})$:= `act_to_virt`($r_{i,k}$) $+ Y_i$;
- $y_{i,k} := \perp$;

Function `job_complete`($\tau_{i,k}$)
- virt := `act_to_virt(now())`;
- **if** $y_{i,k} = \perp$ **and** $v(y_{i,k}) <$ virt **then**
  - $y_{i,k}$:= `virt_to_act`($v(y_{i,k})$);
- Report $\tau_{i,k}$, $r_{i,k}$, $y_{i,k}$, `now()`, and whether the level-C ready queue is empty to the monitor program;

Function `change_speed(new_speed)`
- act := `now()`;
- virt := `act_to_virt(act)`;
- **foreach** $\tau_{i,k}$ *such that* $y_{i,k} = \perp$ *and* $v(y_{i,k}) <$ virt **do**
  - $y_{i,k}$:= `virt_to_act`($v(y_{i,k})$);
- last_act := act;
- last_virt := virt;
- speed := new_speed;
- **foreach** $\tau_{i,k}$ *such that a pending release has been scheduled for virtual time* $v(r_{i,k})$ **do**
  - Reset release timer to fire at `virt_to_act`($v(r_{i,k})$);

Algorithm 1: In-kernel functionality used to handle virtual time.

Function `init_recovery(comp_time, queue_empty)`
- recovery_mode := **true**;
- **if** queue_empty **then**
  - idle_cand := comp_time;
  - pend_idle_cand := pend_now;
- **else**
  - idle_cand := $\perp$;
  - pend_idle_cand := $\{\}$;

Function `on_job_release`($\tau_{i,k}$)
- Add $\tau_{i,k}$ to pend_now;

Function `on_job_complete`($\tau_{i,k}$, $r_{i,k}$, $y_{i,k}$, comp_time, queue_empty)
- Remove $\tau_{i,k}$ from pend_now;
- **if** comp_time $- y_{i,k} > \xi_i$ **then**
  - `handle_miss`($\tau_{i,k}$, $r_{i,k}$, $y_{i,k}$, comp_time, queue_empty);
- **if** recovery_mode **and** idle_cand $\neq \perp$ **then**
  - **if** comp_time $- y_{i,k} > \xi_i$ **then**
    - idle_cand := $\perp$;
    - pend_idle_cand := $\{\}$;
  - **else**
    - Remove $\tau_{i,k}$ from pend_idle_cand;
- **if** recovery_mode **and** idle_cand $= \perp$ **and** queue_empty **then**
  - idle_cand := comp_time;
  - pend_idle_cand := pend_now;
- **if** recovery_mode **and** idle_cand $\neq \perp$ **and** pend_idle_cand $= \{\}$ **then**
  - `change_speed(1)`;
  - recovery_mode := **false**;

Algorithm 2: Userspace monitor algorithms common to SIMPLE and ADAPTIVE.

Function `handle_miss`($\tau_{i,k}$, $r_{i,k}$, $y_{i,k}$, comp_time, queue_empty)
- **if not** recovery_mode **then**
  - `change_speed`($s(t)$);
  - `init_recovery(comp_time, queue_empty)`;

Algorithm 3: Specific userspace implementation for SIMPLE.

Function `handle_miss`($\tau_{i,k}$, $r_{i,k}$, $y_{i,k}$, comp_time, queue_empty)
- **if not** recovery_mode **then**
  - current_speed := 1;
  - `init_recovery(comp_time, queue_empty)`;
- new_speed := $a \cdot (Y_i + \xi_i)/($comp_time $- r_{i,k})$;
- **if** new_speed < current_speed **then**
  - `change_speed(new_speed)`;
  - current_speed := new_speed;

Algorithm 4: Specific userspace implementation for ADAPTIVE.