In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS 2002)*, pages 315–326, Austin, TX, December 3-5 2002.

Scheduling Tasks with Mixed Preemption Relations for Robustness to Timing Faults

John Regehr
School of Computing, University of Utah
regehr@cs.utah.edu

Abstract

This paper introduces and shows how to schedule two novel scheduling abstractions that overcome limitations of existing work on preemption threshold scheduling. The abstractions are task clusters, groups of tasks that are mutually non-preemptible by design, and task barriers, which partition the task set into subsets that must be mapped to different threads. Barriers prevent the preemption threshold logic that runs multiple design-time tasks in the same run-time thread from violating architectural constraints, e.g. by merging an interrupt handler and a user-level thread.

We show that the preemption threshold logic for mapping tasks to as few threads as possible can rule out the schedules with the highest critical scaling factors — these schedules are the least likely to miss deadlines under timing faults. We have developed a framework for robust CPU scheduling and three novel algorithms: an optimal algorithm for maximizing the critical scaling factor of a task set under restricted conditions, a more generally applicable heuristic that finds schedules with approximately maximal critical scaling factors, and a heuristic search that jointly maximizes the critical scaling factor of computed schedules and minimizes the number of threads required to run a task set. We demonstrate that our techniques for robust scheduling are applicable in a wide variety of situations where static priority scheduling is used.

1. Introduction

To rapidly create reliable, reusable embedded and real-time systems software, it is important to begin with the right abstractions. This paper describes two novel abstractions that overcome limitations of existing work on preemption threshold scheduling [20, 26].

Saksena and Wang showed that task sets scheduled with preemption thresholds can have significant schedulability improvements over task sets using fixed priorities. They also showed that it is possible to transform a *design model* consisting of a feasible set of real-time tasks into a feasible *implementation model* containing (often significantly) fewer threads than the task set contains tasks. These implementation models use less memory and cause fewer context switches than the standard implementation model that maps each task to its own thread.

Although the introduction of non-preemptive scheduling is a useful performance optimization, the developers of real-time systems are still presented with a problem that is known to be very difficult: creating correct and predictable software in the presence of concurrent, synchronizing tasks. This paper introduces the *task cluster*, a collection of real-time tasks that are designed to be mutually non-preemptible. A task cluster could be used, for example, to embed a Click [15] graph in a real-time system. Click is an architecture for creating flexible routing software; in a distributed real-time system its components would have real-time requirements. Click gains significant ease of use through its restricted programming model, including the restriction that its components cannot preempt each other. Task clusters provide a natural way to express this requirement without forcing a system to be globally non-preemptive, which can result in a serious loss of effective processor utilization.

The preemption threshold model provides no way to express architectural constraints that rule out some implementation models. For example, it cannot ensure that a task representing an interrupt handler is not be mapped to the same implementation thread as a task representing user-level code. Our second new abstraction, the *task barrier*, provides first-class support for this type of constraint. Abstractly speaking, it might be useful to transparently migrate code into interrupt handlers when this does not impact schedulability, but in practice this is probably undesirable: the migrated code might attempt to access a blocking resource, crashing the system.

We describe task clusters and barriers in Section 2; in Section 3 we show how to find feasible schedules for task sets containing them, both for operating systems with support for preemption threshold scheduling and for systems that have purely static priorities at run time.

The potential to create implementation models with

many fewer threads than tasks begs the question: Is a schedule with fewer threads always better than a schedule with more threads? We found that if there is uncertainty about task worst-case execution times (WCETs), *the answer is no*. This is because the additional constraints on schedules with fewer threads can rule out the schedules with the highest *critical scaling factors* [16]. The critical scaling factor is the largest constant by which the execution time of each task can be multiplied without rendering a schedule infeasible. Intuitively, a schedule with critical scaling factor not much larger than one is "barely feasible" in the sense that a minor perturbation could cause it to miss deadlines — this may not be acceptable for important real-time systems.

We have developed a framework for reasoning about the robustness of task sets that are subject to timing faults, and we have developed new algorithms for finding highly robust schedules including a search algorithm for jointly minimizing the number of threads in an implementation model and maximizing the critical scaling factor among schedules mapping to a particular number of threads. This algorithm can permit system developers to make an informed tradeoff between memory use and robustness when picking an implementation model; it is described in Section 4. Furthermore, our algorithms for increasing the critical scaling factor have broader applicability than just to systems using preemption thresholds. For example, for randomly generated fully preemptible task sets with five members and an average utilization of 0.78, Audsley's optimal scheduling algorithm [2] creates schedules that, on average, permit a 10% increase in task execution time before deadlines start to be missed. One of our algorithms finds schedules that, on average, tolerate an 18% increase without missing any deadlines. Section 5 contains an evaluation of the new algorithms presented in this paper.

In Appendix A we correct an error in the existing response time analysis for task sets with preemption thresholds. We also extend the analysis to support tasks with release jitter.

2. Two New Scheduling Abstractions

This section formally defines *task clusters* and *task barriers*, but first provides an overview of the scheduling model and its notation, as well as some background on preemption threshold scheduling.

Our work begins with a standard task model: tasks are scheduled on a uniprocessor and have deadlines, periods, worst-case execution times, and jitter. Throughout this paper we assume that tasks are scheduled using fixed priorities (or almost-fixed priorities, since we make use of preemption thresholds). Furthermore, we always make a distinction between *tasks*, which are design-time entities with real-time requirements, and *threads*, which are run-time control flows scheduled by the RTOS (real-time operating system).

A real-time task set is $\mathcal{T} = \{\tau_0, ..., \tau_{n-1}\}$ where $\tau_i = (T_i, C_i, D_i, J_i)$. The elements of a task tuple respectively represent the period, worst-case execution time, deadline, and maximum release jitter. Jitter, as defined by Tindell [24, §4], "occurs when the worst-case time between successive releases of a task is shorter than the worst-case time between arrivals of a task." When there is no danger of ambiguity we write $\forall \tau_i$ rather than $\forall \tau_i \in \mathcal{T}$. A *schedule* for $\mathcal{T}$ is a set of priority and preemption threshold assignments $P = \{(P_0, \overline{P}_0), ..., (P_{n-1}, \overline{P}_{n-1})\}$. Zero is the highest priority, and throughout this paper we reverse the sense of the ordering relations for priorities so they have the intuitive meaning (e.g. $x > y$ means x has higher priority than y) rather than the numerical meaning. Priorities are assumed to be assigned uniquely, and so the following predicate must hold for a schedule to be valid:

$$\hat{U} \stackrel{\text{def}}{=} \forall \tau_i, \tau_j \, : \, i \neq j \, \Rightarrow \, P_i \neq P_j$$

Preemption thresholds are not assigned uniquely, but the preemption threshold of each task must be at least as high as its priority. Therefore, the following predicate is also true of valid schedules:

$$\hat{P} \stackrel{\text{def}}{=} \forall \tau_i \, : \, P_i \leq \overline{P}_i$$

Finally, for a given schedule, each task in a set has a worst-case response time R_i. Valid schedules must not permit a task to complete after its deadline:

$$\hat{S} \stackrel{\text{def}}{=} \forall \tau_i \, : \, R_i \leq D_i$$

2.1. Background: Preemption Thresholds

Preemption thresholds were introduced in ThreadX [9], a commercial RTOS, and were first studied academically by Saksena and Wang [20, 26] who developed a response time analysis and a number of useful associated algorithms.

The idea behind preemption thresholds is simple. Task instances compete for processor time based on their priorities, but a task instance that has started running may only be preempted by tasks with priorities higher than the running task's preemption threshold. Preemption threshold scheduling subsumes both preemptive and non-preemptive fixed priority scheduling: purely preemptive scheduling behavior is obtained when each task's preemption threshold is equal to its priority, and purely non-preemptive behavior is obtained when all preemption thresholds are set to the maximum priority. The dominance of preemption threshold scheduling is not merely theoretical; it has been shown to improve schedulability in practice [20]. Intuitively, the source of the improvement is the addition of a limited form of dynamic priorities: if rate-monotonic scheduling is viewed as a first-order approximation to optimal dynamic

priority scheduling, then preemption threshold scheduling can be viewed as a second-order approximation.

There is no known optimal algorithm for finding a feasible assignment of priorities and preemption thresholds that takes less than exponential time. However, efficient approximate algorithms exist. Once a feasible assignment of priorities and thresholds for a given task set is found, Saksena and Wang provide efficient algorithms for assigning *maximal preemption thresholds* (the largest threshold assignment for each task such that all tasks remain schedulable), and also for optimally dividing a task set into non-preemptible groups. Two tasks are mutually non-preemptible if the priority of the first is not higher than the preemption threshold of the second, and vice versa. A non-preemptible group is a collection of tasks within which each pair is non-preemptible; a non-preemptible group of tasks can be run in a single run-time thread.

An implementation model is a set of threads, each of which is responsible for running some non-empty set of tasks from $\mathcal{T}$. Let $M = \{M_0, ..., M_{m-1}\}$ be the sets of design-time tasks that map to each of the m implementation threads. Given a feasible schedule Saksena and Wang [20, Fig. 3] provide an algorithm that can be used to find a corresponding implementation model that is schedulable and satisfies the following predicate:

$$\hat{M} \stackrel{\text{def}}{=} \forall M_i \in M \ : \ \forall \tau_j, \tau_k \in M_i \ : \ P_j \leq \overline{P}_k$$

Saksena and Wang showed that for task sets with random attributes, the number of maximal non-preemptible groups increases much more slowly than the number of tasks: this has important implications for memory-limited embedded systems since each thread has significant memory overhead.

2.2. Task Clusters

A *task cluster* is a subset of a task set within which each pair of tasks must be mutually non-preemptible. Task clusters are different than Saksena and Wang's non-preemptible task groups: the latter are used as a performance optimization while the former are a first-class part of the programming model. In other words, task clusters are visible to, and can be specified by, real-time system developers.

Task sets containing clusters have the important benefit of often permitting the high resource utilizations associated with preemptive scheduling while also permitting the ease of programming that comes from non-preemptive scheduling. Furthermore, task clusters facilitate *synchronization elimination*: the removal of locks that have become superfluous because the resources they protect are only accessed by tasks within a single cluster. When synchronization elimination is applied retroactively, it is an optimization that does not provide software engineering benefits. Rather, it merely eliminates the CPU overhead of acquiring and releasing locks, and the memory overhead of functions supporting, e.g., the priority ceiling protocol. On the other hand, if synchronization elimination is applied at design time it potentially has enormous software engineering benefits: developers, who are often domain experts rather than skilled concurrent system programmers, can completely ignore the dangers of race conditions and deadlocks with respect to resources that are accessed within a single task cluster.

A task set is augmented with a set G of task clusters, where $G_i \subseteq \mathcal{T}$. Valid schedules for $\mathcal{T}$ must satisfy:

$$\hat{G} \stackrel{\text{def}}{=} \forall G_i \in G \ : \ \exists M_j \in M \ : \ G_i \subseteq M_j$$

Task clusters can have overlapping membership, and not every task need belong to a cluster. If a task cluster $G_i = \mathcal{T}$ exists, the only valid schedules will be fully non-preemptive.

2.3. Task Barriers

Priority and preemption relations are often hardwired into the design of a system. To support these relations we require an additional abstraction, the *task barrier*, which is the dual of the task cluster — it isolates groups of tasks that inherently run at different priorities, preventing the thread minimization logic from creating an impossible schedule.

A task set $\mathcal{T}$ is augmented with a set $X \subseteq \{0, ..., (n-1)\}$ of task barriers where valid schedules must satisfy:

$$\begin{aligned} \hat{X} \stackrel{\text{def}}{=} \ & \forall x \in X \ : \ \forall \tau_i \ : \\ & i > x \ \Rightarrow \ (P_i < x \wedge \overline{P}_i < x) \ \wedge \\ & i \leq x \ \Rightarrow \ (P_i \geq x \wedge \overline{P}_i \geq x) \end{aligned}$$

For example, a task barrier at y forces tasks $\tau_0..\tau_y$ to have both priority and threshold at least as high as y, while tasks $\tau_{y+1}..\tau_n$ must have priority and threshold lower than y. Clearly there can be no feasible schedule if there exists a task barrier that "splits" a task cluster.

We use task barriers to model the inherent relationships between implementation artifacts such as interrupts, bottom-half kernel routines, and ordinary threads. For example, consider a task set where tasks $\tau_0..\tau_3$ represent hardware interrupt handlers and $\tau_4..\tau_9$ represent standard tasks. In this case barriers $X = \{0, 1, 2, 3\}$ must exist to preserve the separate identities of the interrupt handlers.

2.4. Summary of the New Scheduling Model

We have introduced two new scheduling abstractions: the *task cluster*, which guarantees that a collection of tasks will be mutually non-preemptible in the implementation model, and the *task barrier*, which partitions the set of tasks into subsets that cannot be mapped to the same implementation thread.

```
PT-ANL (P_orig) {
  P_max = enforce_preds (P_orig)
  B_max = badness (P_max)
  while (max iterations not exceeded) {
    P_new = enforce_preds (permute (P_max))
    B_new = badness (P_new)
    if (B_new == 0) return P_new
    if (B_new ≤ B_max) {
      P_max = P_new
      B_max = B_new
    }
  }
  return FAILURE
}
```

Figure 1. PT-ANL schedules task sets containing task clusters and barriers

We define an overall schedulability function:

$$S(G, X, \mathcal{T}, P, M) \quad \stackrel{\text{def}}{=} \quad \hat{U} \wedge \hat{P} \wedge \hat{S} \wedge \hat{M} \wedge \hat{G} \wedge \hat{X}$$

In general, G, X, and $\mathcal{T}$ can be considered to be fixed for a given task set. On the other hand, P and M are derived terms and there may be many valid choices for them.

3. Scheduling with Task Clusters and Barriers

The previous section defined two new abstractions; in this section we present two complementary techniques for scheduling task sets containing them. Although both techniques make essential use of the response time analysis for preemption threshold scheduling [26], only one of them requires run-time support for preemption thresholds — the other permits threads to have strictly static priorities at run-time. In Section 5 we quantitatively compare the two approaches.

3.1. Targeting Systems with Run-Time Support for Preemption Thresholds

Task clusters and barriers can be scheduled on operating systems that support preemption thresholds using a technique similar to the one proposed by Saksena and Wang for the assignment of priorities and preemption thresholds [20]. Our algorithm, shown in Figure 1, greedily attempts to minimize the "badness" of a schedule using a randomized search through the space of possible priority and preemption threshold assignments. Our badness function is the same as Saksena and Wang's energy function [20, §4.3]: it is the sum of the lateness of each task where lateness is $\max(R_i - D_i, 0)$. The algorithm is finished when a schedule with badness zero is found, since this means that no task's response time is later than its deadline.

The *permute* function randomly either swaps the priorities of two tasks, or either increments or decrements the preemption threshold of a task. The *enforce_preds* function ensures that a schedule does not violate any of the predicates (other than $\hat{S}$) defined in the previous section. It does this, for example, first by noticing that $\hat{M}$ is violated, and second by appropriately adjusting the priority and/or preemption threshold assignments of the offending tasks. These adjustments are repeated until all predicates are satisfied; this is possible because we test for a conflict between predicates, e.g. a task cluster that is split by a barrier, before starting the randomized search.

For simplicity, the algorithms presented in this paper are randomized greedy algorithms. In practice, better results can often be obtained using simulated annealing. Converting a greedy search to one that uses simulated annealing is a straightforward matter of adding logic to probabilistically accept inferior solutions [18, §10.9].

3.2. Targeting Systems without Run-Time Support for Preemption Thresholds

A straightforward implementation of task clusters on a standard RTOS is to have each instance of a task belonging to a cluster acquire a lock associated with the cluster before performing any computation, and to release the lock just before terminating. If the lock implements the stack resource policy [3] or the priority ceiling protocol [21], then the lock protocols themselves introduce a form of dynamic priorities not unlike preemption thresholds — the difference being that the purpose of the priority change is to bound priority inversion and prevent deadlock, rather than to improve schedulability. As Gai et al. [10] have observed, there is considerable synergy between these synchronization protocols and preemption threshold scheduling. A lock-based implementation of task clusters, however, seems inelegant. It adds the time and space overhead of a lock, does not help minimize threads, and does not help support task barriers. Rather, we develop two solutions that fit into our existing framework; both perform better than the lock-based implementation, as we demonstrate in Section 5.

Let $\text{maxp}(M_i)$ denote the maximum of the highest priority or preemption threshold of any task in M_i. Similarly, let $\text{minp}(M_i)$ denote the minimum of the lowest priority or preemption threshold of any task in M_i. Define $\hat{F}$ as follows:

$$\begin{aligned} \hat{F} \quad \stackrel{\text{def}}{=} \quad & \forall M_i, M_j \in M \ : \\ & \text{maxp}(M_i) < \text{minp}(M_j) \ \vee \\ & \text{minp}(M_i) > \text{maxp}(M_j) \end{aligned}$$

This predicate ensures that the priorities and preemption thresholds of tasks mapped to each thread do not overlap the priorities and preemption thresholds of tasks mapped to

any other thread. Since there is no overlap any priority and preemption threshold in the range $\text{minp}(M_i)..\text{maxp}(M_i)$ can be chosen for thread i. By choosing the priority and threshold to be the same value we create a run-time schedule that is equivalent to purely preemptive thread scheduling — no preemption threshold support is required and a standard RTOS can be used. Furthermore, since only a single priority level is required for each thread, as opposed to the technique from the previous section that requires up to two priority levels per task, this technique is ideal for targeting a small RTOS that supports a limited number of priorities.

To satisfy $\hat{F}$ as well as the other predicates comprising the previously defined schedulability function S, we have developed a modified version of Audsley's optimal priority assignment algorithm for pure preemptive [2] and non-preemptive [12] scheduling. Audsley's algorithm reduces the space of priority assignments that must be searched from $n!$ to n^2 by exploiting the property that although the response time of a task depends on the set of tasks that has higher priority, it does not depend on the particular priority ordering among those tasks. The natural algorithm, then, is to find a task that is schedulable at the lowest priority, then the second-lowest priority, etc. Once a task is found to meet its deadline at a given priority, this property will not be broken by priority assignments made to tasks with higher priority.

To support task clusters and barriers within this framework we have designed a three-level hierarchical version of Audsley's algorithm, called **SP-3**, that operates as follows. At the outermost level the partitions created by task barriers are processed in order from lowest to highest priority. For example, a task set with 6 tasks and a barrier at 2 would be treated in two parts: first, tasks 3–5, and second, tasks 0–2. Within each partition task clusters are treated separately. For purposes of this algorithm we assume that each task belongs to a unique cluster: this can be easily accomplished by merging clusters that have tasks in common and by creating singleton clusters for tasks not initially belonging to a cluster. Task clusters within a partition are scheduled in a manner analogous to Audsley's algorithm for tasks. We try to schedule each cluster at the lowest priority in the partition; as priority assignments are found that meet the response time requirements of all tasks within the cluster, we progress to higher priorities. Finally, within a cluster, individual tasks are scheduled using the version of Audsley's algorithm that is optimal for non-preemptive scheduling. **SP-3** will find a feasible schedule if one exists that does not introduce any extra non-preemption beyond what is specified by the task clusters.

We have developed a second algorithm, **SP-ANL**, for scheduling task sets with clusters and barriers that, given enough time, outperforms **SP-3** in the sense that it finds feasible schedules more often. This performance is quantified in Section 5. **SP-ANL** is identical to **PT-ANL** (Figure 1) except that the *permute* function operates at a higher level. Instead of randomly permuting a priority or preemption threshold, it randomly either swaps the priorities of two tasks within a cluster, swaps the priority ordering of two entire clusters, or attempts to run two clusters in the same implementation thread. It is this final permutation that provides additional non-preemption beyond what is specified by task clusters, permitting **SP-ANL** to schedule more task sets than **SP-3**.

4. Robust Scheduling

A *timing fault* occurs when a task instance runs for too long, but eventually produces the correct result. Real-time systems that are robust with respect to timing faults are desirable for several reasons. First, analytic worst-case execution time (WCET) tools are not in widespread use, and it is not clear that tight bounds on WCET can be found for complex software running on aggressively designed processors. Second, even if accurate WCETs are available with respect to the CPU, it may difficult to ensure the absence of interference from bus contention, unexpected or too-frequent interrupts, or a processor that is forced to run in a low-power mode due to energy constraints. Finally, it is just sound engineering to avoid building systems that are sensitive to minor perturbations.

The rate monotonic algorithm, the deadline monotonic algorithm, and Audsley's priority assignment algorithm belong to the class of algorithms that we call FEAS-OPTIMAL: they are guaranteed to find, for different classes of task sets, a feasible schedule if any exist. In this section we define the ROBUST-OPTIMAL class of scheduling algorithms: they are guaranteed to produce a schedule that maximizes some robustness metric of interest.

4.1. A Framework for Robust Scheduling

A *transformation* Z is an arbitrary function from task sets to task sets. Transformations of interest will model a class of changes that should be "tolerated" by a task set. For example, $Z_J(\mathcal{T}, \Delta) \stackrel{\text{def}}{=} \{(T_i, C_i, D_i, \Delta \cdot J_i)\}$ is the transformation that models an increase in release jitter. Δ is a *scaling factor*. The *critical value* of Δ for a given priority assignment and transformation, denoted $\Delta^*(G, X, \mathcal{T}, Z, P, M)$, is the largest value of Δ such that the transformed task set remains schedulable:

$$\forall \Delta \in \mathbb{R} \ : \ S(G, X, Z(\mathcal{T}, \Delta), P, M) \ \Rightarrow \ \Delta \leq \Delta^*$$

Let $\mathcal{P}$ be the set of all possible priority and preemption threshold assignments for a task set. Note that the size of $\mathcal{P}$ can be large even for modestly sized task sets since it contains $n!n!$ elements. The *maximal critical value* of the

scaling factor, Δ^{**}, has the following property:

$$\forall P \in \mathcal{P} \ : \ \Delta^*(G, X, \mathcal{T}, Z, P, M) \leq \Delta^{**}(G, X, \mathcal{T}, Z)$$

The set of priority assignments of maximal robustness is P_{max} where:

$$P_{max} \subseteq \mathcal{P} : P \in P_{max} \ \Rightarrow \\ \Delta^*(G, X, \mathcal{T}, Z, P, M) = \Delta^{**}(G, X, \mathcal{T}, Z)$$

A ROBUST-OPTIMAL scheduling algorithm is one that can find a member of P_{max}.

We usually abbreviate $\Delta^*(G, X, \mathcal{T}, Z, P, M)$ as Δ^*; it is to be understood that Δ^* is a function of a task set, a transformation, and a schedule. Similarly, $\Delta^{**}(G, X, \mathcal{T}, Z)$ is a function of a transformation and a task set, including its associated clusters and barriers; we usually abbreviate it as Δ^{**}.

4.2. The Critical Scaling Factor

Throughout the rest of this paper we use a transformation Z_C that multiplies the WCET of each task in a set by the scaling factor: $Z_C(\mathcal{T}, \Delta) \stackrel{\text{def}}{=} \{(T_i, \Delta \cdot C_i, D_i, J_i)\}$. This is the transformation defined by Lehoczky et al. [16], but generalized slightly to support tasks with release jitter and arbitrary deadlines. Z_C models generic uncertainty about WCET and also uniform expansion of task run-times due to interference from memory cycle stealing or brief, unanticipated interrupts. A useful property of this transformation is that $S(G, X, Z_C(\mathcal{T}, \Delta), P, M)$ is monotonic in Δ and therefore Δ^* can be efficiently computed using a binary search.

For the remainder of this paper when we say that a task set is *robust*, we mean "robust with respect to uniform expansion in WCET." Also, we restrict the meaning of a ROBUST-OPTIMAL scheduling algorithm to be one that finds a schedule maximizing the scaling factor of Z_C.

Although our focus is on uniform expansion of task WCETs, the algorithms that we present are general and could easily support other transformations such as those that: scale only a single task or a subset of the tasks (this family of transformations is examined by Vestal [25]); reduce the period of a task representing a hardware interrupt whose minimum interarrival time is not precisely known; scale task execution times by a weighted factor reflecting the degree of uncertainty in WCET estimates; or, scale tasks with smaller run-times by a larger factor to model interference from a long-running, unanticipated interrupt handler.

4.3. A Simple Example

Consider the following task set:

$\tau_0 : C = 400; T, D = 1999; J = 0$
$\tau_1 : C = 400; T, D = 2000; J = 1200$

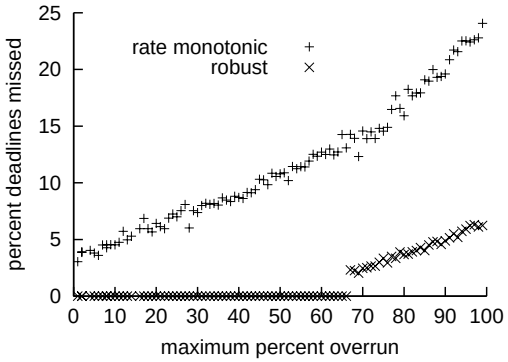

Figure 2. Comparing the behavior of two schedules in the presence of timing faults

There are only two possible fully preemptive schedules, and both of them are feasible. When scheduled using the rate-monotonic priority assignment, the worst-case response time of Task 1 is 400 and Task 2 is 2000. A little experimentation will show that if the WCET of either task is increased, the task set ceases to be schedulable. When the non-rate-monotonic priority assignment is used, the worst-case response times are 800 and 1600, respectively, and the WCET of both tasks can be scaled by 1.67 before the task set becomes infeasible. In other words, by avoiding the rate-monotonic priority assignment, we increase the critical scaling factor of the task set from approximately 1.0 to 1.67. Clearly the non-rate-monotonic priority assignment is preferable: a mispredicted worst-case execution time is far less likely to make it miss a deadline. This is demonstrated in Figure 2, which compares the propensity of the two schedules to miss deadlines under overload. Each data point was generated by simulating 50 million time units. A "maximum percent overload" of 25 means that the execution time of each task instance is uniformly distributed between the nominal WCET and 1.25 times the WCET.

4.4. Properties of Some FEAS-OPTIMAL Algorithms

Theorem 1. *For the class of task sets where the deadline monotonic (DM) algorithm is* FEAS-OPTIMAL *(i.e. fully preemptive scheduling, no release jitter, deadline not greater than period), it is also* ROBUST-OPTIMAL.

Proof. Let $\mathcal{T}$ be a member of the class of task sets for which DM is an optimal scheduling algorithm. Based on $\mathcal{T}$, define a set of scaled task sets $\mathcal{Z}_C \stackrel{\text{def}}{=} \{\forall \Delta \in \mathbb{R} : Z_C(\mathcal{T}, \Delta)\}$ that differ only in their WCETs. Let P_{DM} be the deadline monotonic schedule for $\mathcal{T}$. Since the deadline monotonic schedule for a task set is independent of the WCET of tasks in the set it follows that for each member of $\mathcal{Z}_C$, P_{DM} is a feasible schedule if any exist. Therefore, it is impossible that there exists a schedule $P_{max} \neq P_{DM}$ such that

$\Delta^*(P_{max}) > \Delta^*(P_{DM})$, since this would imply that there is a member of $\mathcal{Z}_C$ for which P_{DM} is infeasible, but a different schedule is feasible. □

Theorem 2. *Audsley's* FEAS-OPTIMAL *algorithm for priority assignment is not* ROBUST-OPTIMAL *for preemptive [2] or for non-preemptive [12] scheduling.*

Proof. In both versions of the algorithm, if all tests of task response time versus deadline succeed, then the first task in the set is assigned the lowest priority, the second task the second-lowest priority, etc. Therefore, we can feed tasks to the algorithm in such a way that a non-robust-optimal schedule is produced. For example, if τ_1 and then τ_0 from Section 4.3 were given to Audsley's algorithm, it would generate the rate-monotonic priority assignment that we know to not be ROBUST-OPTIMAL. It is straightforward to construct an analogous example for non-preemptive scheduling. □

4.5. Finding Robust Schedules

For classes of task sets that have an efficient FEAS-OPTIMAL scheduling algorithm and for transformations where the schedulability function is monotonic in the scaling factor, an efficient ROBUST-OPTIMAL algorithm can be created by invoking the FEAS-OPTIMAL algorithm in a binary search. This strategy can be used to maximize the critical scaling factor, for example, of a task set scheduled by either the preemptive or non-preemptive version of Audsley's algorithm for priority assignment. We call these algorithms **ROB-OPT**.

For classes of task sets that lack an efficient FEAS-OPTIMAL algorithm (e.g. task sets with preemption thresholds) or for transformations where schedulability is not monotonic in the scaling factor, we require an alternative to **ROB-OPT**. We have developed **ROB-ANL**, shown in Figure 3. It is a randomized heuristic search that can efficiently compute an approximate member of P_{max}. **ROB-ANL** is similar to **PT-ANL** (shown in Figure 1) except that (1) instead of minimizing the degree to which task response times exceed their deadlines, we maximize the critical scaling factor, and (2) in the version of the algorithm that uses simulated annealing we never accept an infeasible schedule, although we must sometimes accept a solution that has an inferior critical scaling factor.

An advantage of using a heuristic search is that the details of the parameter being optimized do not matter. For example, if the cost of acquiring and releasing locks were modeled in the schedulability function, then the heuristic would naturally attempt to merge synchronizing tasks since these schedules would have lower CPU overhead and consequently are good candidates for being highly robust. In the same vein, we would like to extend the response time analysis for preemption threshold scheduling to accurately model the costs of preemptive and non-preemptive context switches. This would cause the search heuristic to find schedules with low numbers of context switches, again because the reduced overhead would leave more room for timing faults. In summary, searching for robust schedules permits many schedule optimizations to be treated uniformly; we believe this is a significant advantage.

```
ROB-ANL (P_orig) {
  P_max = P_orig
  Δ_max = critical_scaling_factor (P_max)
  while (max iterations not exceeded) {
    P_new = permute (P_max)
    Δ_new = critical_scaling_factor (P_new)
    if (Δ_new ≥ Δ_max) {
      P_max = P_new
      Δ_max = Δ_new
    }
  }
  return P_max
}
```

Figure 3. ROB-ANL approximately maximizes the critical scaling factor of a task set

4.6. Maximizing the Critical Scaling Factor and Minimizing Implementation Threads

Minimizing the number of threads required to run a task set can conflict with maximizing robustness. To see this, notice that the fewer implementation threads required to run a schedule, the more constraints there are on the priority and preemption threshold assignments. Sometimes these constraints hurt schedulability because they rule out the most robust schedules. Instead of optimizing a composite value function, i.e. one based on some weighting of maximizing robustness and minimizing implementation threads, we believe that developers should be permitted to make an informed decision using a table that presents the largest critical scaling factor that could be achieved for each number of threads.

The algorithm for the joint minimization of implementation threads and maximization of critical scaling factor is **MIN-THR**; it appears in Figure 4. This algorithm uses a heuristic search to find a schedule mapping to as few implementation threads as possible. Whenever a schedule is found that maps to a number of threads that has not yet been seen, it forks off an optimization to attempt to find the schedule that maximizes the critical scaling factor over schedules mapping to that number of threads. In other words, it calls a slightly modified version of **ROB-ANL** (from Figure 3) that only accepts schedules that map to a particular number of threads.

```
MIN-THR (P_orig) {
  P_min = P_orig
  T_min = impl_threads (P_orig)
  while (max iterations not exceeded) {
    P_new = permute (P_min)
    T_new = find_impl_threads (T_new)
    if (not_yet_seen (T_new)) ROB-ANL-T (P_new, T_new)
    if (T_new ≤ T_min) {
      P_min = P_new
      T_min = T_new
    }
  }
}
```

Figure 4. MIN-THR approximately minimizes threads and maximizes robustness

5. Experimental Evaluation

This section provides a brief survey of the performance of the new techniques presented in this paper. Our procedure for generating random task sets is as follows, where all random numbers are taken from a uniform distribution. The period of each task is a random value between 1 and 1000 time units. The utilization is chosen by generating a random number in range 0.1–2.0 and dividing that number by the number of tasks in the set. (Scaling utilization by the inverse of the number of tasks is merely a heuristic to avoid generating too many infeasible task sets.) The deadline for each task is either set to be the same as the period or is an independently chosen random value between 1 and 1000, depending on the experiment. Tasks were assigned release jitter in some experiments; see below. Finally, any task set with utilization greater than one is immediately discarded.

5.1. Task Clusters and Barriers

In this section we compare the different algorithms that we have developed for finding feasible schedules for task sets containing task clusters and barriers.

We compare five algorithms for scheduling task clusters. The first is **NP-OPT**, the optimal algorithm for assigning priorities for fully non-preemptible scheduling [12]. Recall that in the presence of non-trivial task clusters a fully preemptive schedule is never valid (because members of a cluster must be mutually non-preemptible) while a fully non-preemptive schedule is always valid. The second algorithm is **SP-LOCK**, the strawman algorithm that we proposed in Section 3.2 for implementing task clusters by forcing tasks in the each cluster to always have a lock associated with the cluster. The third algorithm is **SP-3**, the hierarchical version of Audsley's algorithm for priority assignment, and the fourth is **SP-ANL**, the heuristic search for priority and preemption threshold assignments for task sets that are to have purely static priorities at run time (Section 3.2). Finally, the fifth algorithm is **PT-ANL**, the heuristic search for priority and preemption threshold assignments for task sets containing clusters and when preemption threshold support is available on the target RTOS.

n	NP-OPT	SP-LOCK	SP-3	SP-ANL	PT-ANL
5	34	65	73	88	100
10	35	49	61	77	100
15	25	48	53	63	100
20	29	41	49	58	100
25	41	39	43	47	100

Figure 5. Relative performance of algorithms for scheduling task clusters

n	SP-3	SP-ANL	PT-ANL
5	98	99	100
10	77	89	100
15	75	79	100
20	71	70	100
25	59	57	100

Figure 6. Relative performance of algorithms for scheduling task barriers

Figure 5 shows the results of an experiment where random task sets were passed to each of the five algorithms listed above. The experiment terminated when any algorithm successfully scheduled 100 task sets, and therefore the results are automatically normalized with respect to the best algorithm, which always has score 100. The experiment was repeated for task sets containing 5, 10, 15, 20, and 25 tasks. For every task set: there was a single task cluster containing between 2 and $n/2$ randomly selected tasks; each task's deadline was equal to its period; and, each task had a 50% chance of being assigned release jitter up to half its period.

Figure 6 shows the results of an experiment similar to the previous one, except that instead of containing a task cluster, each task set was assigned a single randomly placed task barrier. The algorithms tested were the same as in the previous experiment except that **NP-OPT** and **SP-LOCK** had to be dropped since they may produce invalid schedules for task sets containing task barriers.

These experiments show that **PT-ANL** consistently outperforms the other algorithms, and that the gap between it and the others increases for larger task sets. This can be taken as a corroboration of Saksena and Wang's results [20] about the practical dominance of preemption threshold scheduling over static priority scheduling. Of the

	Preemptive			Non-Preemptive		
n	**P-OPT**	**ROB-OPT**	% inc.	**NP-OPT**	**ROB-OPT**	% inc.
5	1.11	1.18	63%	1.09	1.16	84%
10	1.06	1.13	109%	1.05	1.12	131%
15	1.05	1.10	110%	1.04	1.10	138%
20	1.05	1.09	97%	1.04	1.09	136%
25	1.04	1.08	92%	1.04	1.08	108%

Figure 7. Improving the critical scaling factor

	no jitter		1 task w/J		50% tasks w/J	
n	T=D	T≠D	T=D	T≠D	T=D	T≠D
5	0%	62%	29%	61%	30%	63%
10	0%	101%	51%	93%	50%	109%
15	0%	100%	50%	87%	59%	110%
20	0%	100%	50%	100%	61%	97%
25	0%	81%	54%	77%	54%	92%

Figure 8. Headroom increases due to ROB-OPT

three algorithms that generate static-priority schedules, **SP-ANL**, the heuristic search, outperforms **SP-3**, although the gap narrows with increasing numbers of tasks. We believe that this is because the extra non-preemptibility available to **SP-ANL** becomes less valuable for larger task sets. Also, notice that in Figure 6 **SP-3** slightly outperforms **SP-ANL** for task sets with 20 and 25 members. We speculate that this happens because the size of the priority assignment space for large task sets overwhelms the search heuristic.

5.2. Improving the Robustness of Schedules

Figure 7 shows the increase in critical scaling factor that **ROB-OPT** can achieve relative to Audsley's FEAS-OPTIMAL algorithms for fully preemptive (**P-OPT**) and fully non-preemptive scheduling (**NP-OPT**). As before, task sets are randomly generated and have 5–25 members. Each task's deadline and period are unrelated and each task has a 50% chance of being assigned random release jitter up to half its period. Values in the table represent the median critical scaling factor over 500 feasible task sets, and *inc* indicates the percent increase in the distance of the critical scaling factor from 1.0 under optimization by **ROB-OPT**. For example, if the FEAS-OPTIMAL scheduling algorithm produces a schedule where $\Delta^* = 1.10$ and **ROB-OPT** produces a schedule that has $\Delta^* = 1.13$, then we say that we have increased the amount of "headroom" that the task has before missing deadlines by 30%.

Figure 8 shows another way to evaluate **ROB-OPT**'s ability to increase Δ^*. For task sets containing different numbers of tasks it shows the increase in headroom for task sets where (1) the deadline of each task is equal to its period, and (2) where period and deadline are unrelated. The other parameter that is adjusted is the amount of jitter: task sets either have no release jitter, a single task with jitter randomly distributed between zero and half its period, or each task has a 50% chance of being assigned jitter up to half its period. The failure to increase Δ^* for task sets without jitter and where T=D is a direct consequence of Theorem 1.

6. Related Work

Hybrid preemptive/non-preemptive schedulers are an old idea, and in fact they can be found in the kernel of almost every general-purpose operating system: interrupts are scheduled preemptively, bottom-half kernel routines are scheduled non-preemptively, and threads are scheduled preemptively. The real-time analysis of non-preemptive sections caused by critical regions [3, 21] is more recent. The real-time analysis of mixed preemption for its own sake was pioneered by Saksena and Wang [20, 26] and by Davis et al. [7]. Our work builds directly on Saksena and Wang's, adding several new capabilities.

Synchronization elimination has been addressed both by the real-time and programming language communities. For example, the Spring system [22] used static scheduling and was capable of recognizing situations where contention for a shared resource was impossible, in which case a lock was not used at run time. Aldrich et al. [1] show how to remove unnecessary synchronization operations from Java programs. The difference between previous work and the work presented in this paper is that synchronization elimination has until now been treated as a compile-time or run-time performance optimization. We believe that using task clusters to give the programmer explicit control over the elimination of synchronization between (logically) concurrent tasks can result in significant software engineering benefits in addition to the previously realized performance benefits.

Starting with Lehoczky et al. [16] a number of researchers have used the critical scaling factor as a metric for schedulability, including Katcher et al. [14], Vestal [25], Yerraballi et al. [27], and Punnekkat et al. [19]. However, as far as we know it has not been previously recognized that it is possible to search for schedules with higher critical scaling factors, and that these schedules are inherently preferable when there is generic uncertainty about task WCET.

Existing techniques for tolerating timing faults — task instances that run for too long but eventually produce a correct result — can be divided into those that change the task model from the developer's point of view and those that do not. A number of scheduling techniques for dealing with timing faults have been proposed that change the task model, including robust earliest deadline [4], time redundancy [6], rate adaptation [5], user-defined timing failure

handlers [23], and (m, k)-firm deadlines [13]. Our method for increasing robustness does not change the task model. It is complementary to, and can be used independently of or in combination with, essentially all of the other known techniques for dealing with timing faults in systems using static priority scheduling. Another technique that is transparent to developers is *isolation*- or *enforcement*-based scheduling [11, 17] where tasks are preempted when they exceed their execution time budgets. Although this technique cannot prevent missed deadlines it can isolate deadline misses to tasks that overrun.

Edgar and Burns [8] have developed a method for statistically estimating task WCET based on measurements. They also show how to statistically estimate the feasibility of a task set, but do not address the problem of finding highly or maximally robust schedules. Our work, on the other hand, directly addresses the problem of finding robust schedules, but permits the statistical nature of unreliable WCET estimates to remain implicit. It may be useful to integrate the two models.

7. Software

All numerical results in this paper were generated using SPAK, a static priority analysis kit that we have developed. SPAK is a collection of portable, efficient functions for creating and manipulating task sets, for analyzing their response times, and for simulating their execution. A variety of existing analyses with different tradeoffs between speed and generality are available, as is the corrected and extended preemption threshold analysis presented in Appendix A. SPAK is open source software and can be downloaded from `http://www.cs.utah.edu/~regehr/spak`.

8. Future Work

Currently, a task barrier is defined to split the task set into two parts based on task indices. This is useful when there are inherent priority relations between tasks, e.g. when some tasks model interrupt handlers. However, a more general abstraction is probably desirable — one that permits the specification of subsets of the task set that must be isolated from each other, e.g. by a CPU reservation, but between which there is no inherent priority ordering.

Although we currently do not use CPU reservations or any other kind of enforcement-based scheduling, in the future we plan to use them to create temporal partitions between task clusters. Partitions inside clusters probably do not make sense because clusters are internally non-preemptible, and because tasks in clusters are assumed to be part of a subsystem and therefore semantically related, reducing the utility of isolating them from each others' timing faults.

9. Conclusions

The paper has described a number of practical additions to existing work on fixed-priority real-time scheduling.

First, we have introduced two novel abstractions: task clusters and task barriers. Task clusters make non-preemptive scheduling into a first-class part of the real-time programming model. We claim that clusters provide significant software engineering benefits, such as the elimination of the possibility of race conditions and deadlocks within a cluster, as well as performance benefits due to reduced preemptions, reduced memory overhead for threads, and reduced lock acquisitions. These benefits are achieved without sacrificing the higher utilizations that can usually be achieved through preemptive scheduling. Task barriers restore an important advantage of static priority scheduling — support for integrated schedulability analysis of interrupts, kernel tasks, and user-level threads — to preemption threshold scheduling when the objective is to minimize the number of implementation threads onto which design tasks are mapped.

Second, we have developed three novel algorithms for finding feasible schedules for task sets containing clusters and barriers. The first targets systems with run-time support for preemption thresholds while the others permit thread priorities to be strictly static at run-time. By "compiling" task sets containing task clusters and barriers to target a static-priority environment, we have shown that while run-time support for preemption thresholds is often not necessary, the response time analysis for preemption thresholds is an important building block for real-time systems.

Third, we have characterized a framework within which it is possible to analyze the robustness of task sets under a given class of timing faults and we have developed two algorithms that can often find a schedule for a given task set that has a higher critical scaling factor than the schedule generated by the appropriate FEAS-OPTIMAL scheduling algorithm. This extra resilience to timing faults is essentially free: it is cheap at design time and imposes no cost at run-time.

Finally, we have corrected an error in the response time analysis for task sets with preemption thresholds.

Acknowledgments

The author would like to thank Luca Abeni, Eric Eide, Jay Lepreau, Rob Morelli, Alastair Reid, Manas Saksena, Jack Stankovic, and the reviewers for providing valuable feedback on drafts of this paper.

This work was supported, in part, by the National Science Foundation under award CCR-0209185 and by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory under agreements F30602-99-1-0503 and F33615-00-C-1696.

References

[1] Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, and Susan Eggers. Eliminating unnecessary synchronization from Java programs. In *Proc. of the Static Analysis Symp.*, Venezia, Italy, September 1999.

[2] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.

[3] Theodore P. Baker. A stack-based resource allocation policy for realtime processes. In *Proc. of the 11th IEEE Real-Time Systems Symp.*, pages 191–200, Lake Buena Vista, FL, December 1990.

[4] Giorgio Buttazzo and John A. Stankovic. RED: Robust earliest deadline scheduling. In *Proc. of the 3rd International Workshop on Responsive Computing Systems*, pages 100–111, Lincoln, NH, September 1993.

[5] Marco Caccamo, Giorgio Buttazzo, and Lui Sha. Handling execution overruns in hard real-time control systems. *IEEE Transactions on Computers*, 51(5), May 2002.

[6] Mario Caccamo and Giorgio Buttazzo. Optimal scheduling for fault-tolerant and firm real-time systems. In *Proc. of the 5th Intl. Workshop on Real-Time Computing Systems and Applications*, Hiroshima, Japan, October 1998.

[7] Robert Davis, Nick Merriam, and Nigel Tracey. How embedded applications using an RTOS can stay within on-chip memory limits. In *Proc. of the Work in Progress and Industrial Experience Sessions, 12th Euromicro Workshop on Real-Time Systems*, pages 43–50, Stockholm, Sweden, June 2000.

[8] Stewart Edgar and Alan Burns. Statistical analysis of WCET for scheduling. In *Proc. of the 22nd IEEE Real-Time Systems Symp.*, London, UK, December 2001.

[9] Express Logic Inc. ThreadX Technical Features, version 4. `http://www.expresslogic.com/txtech.html`.

[10] Paolo Gai, Giuseppe Lipari, and Marco di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proc. of the 22nd IEEE Real-Time Systems Symp.*, London, UK, December 2001.

[11] Mark K. Gardner and Jane W. S. Liu. Performance of algorithms for scheduling real-time systems with overrun and overload. In *Proc. of the 11th Euromicro Workshop on Real-Time Systems*, York, UK, June 1999.

[12] Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and non-preemptive real-time uni-processor scheduling. Technical Report 2966, INRIA, Rocquencourt, France, September 1996.

[13] Moncef Hamdaoui and Parameswaran Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451, December 1995.

[14] Daniel I. Katcher, Hiroshi Arakawa, and Jay K. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Transactions on Software Engineering*, 19(9):920–934, September 1993.

[15] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[16] John Lehoczky, Lui Sha, and Ye Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proc. of the 10th IEEE Real-Time Systems Symp.*, pages 166–171, Santa Monica, CA, December 1989.

[17] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves for multimedia operating systems. In *Proc. of the IEEE Intl. Conf. on Multimedia Computing and Systems*, May 1994.

[18] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992.

[19] Sasikumar Punnekkat, Rob Davis, and Alan Burns. Sensitivity analysis of real-time task sets. In *Proc. of the Asian Computing Science Conference*, pages 72–82, Kathmandu, Nepal, December 1997.

[20] Manas Saksena and Yun Wang. Scalable real-time system design using preemption thresholds. In *Proc. of the 21st IEEE Real-Time Systems Symp.*, Orlando, FL, November 2000.

[21] Lui Sha, Ragunathan Rajkumar, and John Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[22] John A. Stankovic, Krithi Ramamritham, Douglas Niehaus, Marty Humphrey, and Gary Wallace. The Spring system: Integrated support for complex real-time systems. *Real-Time Systems Journal*, 16(2/3):223–251, May 1999.

[23] David B. Stewart and Pradeep K. Khosla. Mechanisms for detecting and handling timing errors. *Communications of the ACM*, 40(1):87–93, January 1997.

[24] Ken Tindell, Alan Burns, and Andy J. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Real-Time Systems Journal*, 6(2):133–151, March 1994.

[25] Steve Vestal. Fixed-priority sensitivity analysis for linear compute time models. *IEEE Transactions on Software Engineering*, 20(4):308–317, April 1994.

[26] Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proc. of the 6th Intl. Workshop on Real-Time Computing Systems and Applications*, Hong Kong, December 1999.

[27] Ramesh Yerraballi, Ravi Mukkamala, Kurt Maly, and Hussein Abdel-Wahab. Issues in schedulability analysis of real-time systems. In *Proc. of the 7th Euromicro Workshop on Real-Time Systems*, Odense, Denmark, June 1995.

A. Correcting the Response Time Analysis for Preemption Threshold Scheduling

The original response time analysis for task sets scheduled using preemption thresholds [26] contains an error — it sometimes examines too few previous task invocations, resulting in the potential for underestimated response times.

Figure 9 shows the difference between the previous preemption threshold analysis and the corrected version presented in this section. The old analysis predicts that the task set is feasible, while the new analysis predicts that τ_1 may not meet its deadline. Figure 10 is a trace of a simulated execution of the task set. It proves the infeasibility of the task set by counterexample: τ_1 misses its second deadline.

The following response time analysis differs from the one presented by Wang and Saksena in two major ways. First, it has a different termination condition for the loop that takes previous invocations of a task into account when computing its response time. Second, it adds support for tasks with release jitter. We have also changed the notation to match that used in this paper.

The worst-case blocking time for a task is:

$$B_i = \max_{\forall j \,:\, \overline{P}_j \geq P_i > P_j} C_j$$

In other words, the worst-case blocking for τ_i happens when the task with the longest WCET that has lower priority and higher preemption threshold is dispatched infinitesimally earlier than τ_i is able to run.

$\mathcal{S}_i$, the worst-case start time of task i, is:

$$\mathcal{S}_i(q) = B_i + qC_i + \sum_{\forall j \,:\, P_j > P_i} \left(1 + \left\lfloor \frac{\mathcal{S}_i(q) + J_j}{T_j} \right\rfloor\right) C_j$$

Our only change to this equation is the addition of a term accounting for release jitter.

$\mathcal{F}_i$, the worst-case finish time of task i, is:

$$\mathcal{F}_i(q) = \mathcal{S}_i(q) + C_i + \sum_{\forall j \,:\, P_j > \overline{P}_i} \left(\left\lceil \frac{\mathcal{F}_i(q) + J_j}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{\mathcal{S}_i(q) + J_j}{T_j} \right\rfloor\right)\right) C_j$$

Again, we have only added the jitter terms.

The response time of task i is:

$$r_i = \max_{\forall q \,:\, 0 \leq q \leq Q} (\mathcal{F}_i(q) + J_i - qT_i)$$

Where Q is $\lfloor L_i / T_i \rfloor$. L_i is the longest level-i busy period for preemption threshold scheduling, and is:

$$L_i = B_i + \sum_{\forall j \,:\, P_j \geq P_i} \left\lceil \frac{L_i + J_j}{T_j} \right\rceil C_j$$

Task	C_i	T_i	D_i	J_i	P_i	$\overline{P}_i$	r_i^{old}	r_i^{new}
τ_0	40	70	70	0	0	0	60	60
τ_1	20	90	90	0	2	0	80	120
τ_2	20	100	100	0	1	0	80	80

Figure 9. Response times computed using the original and fixed analyses

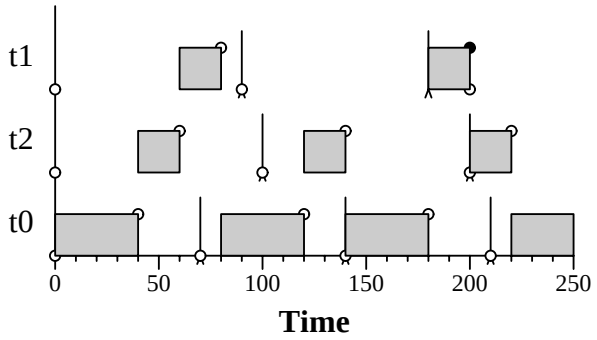

Figure 10. Simulated execution trace of the task set from Figure 9. The second instance of task 1 misses its deadline.

The computation of Q was adapted from George et al. [12], and is the core of the difference between our analysis and the previously published one, which iterated only until $q = m$ where $\mathcal{F}_i(m) \leq q \cdot T_i$. By working through the response time calculation for τ_1 in the example task set in Figure 9, this termination condition can be seen to be the source of the error.

Whenever a variable appears on both sides of the equation (i.e., $\mathcal{S}_i$, $\mathcal{F}_i$, and L_i) its value can be found by iterating until the value converges. Zero is a safe initial value for $\mathcal{S}_i$ and $\mathcal{F}_i$, but L_i needs to start at one.

Finally, we do not believe that the discrepancy between the old and new response time analyses affects any of the qualitative results reported by Saksena and Wang. For randomly generated task sets with 10 members and no release jitter the two analyses agree on the response times of all tasks about 99% of the time.