Limited Preemption in Real-Time Scheduling

Krings, Axel W.; Azadmanesh, M.H.
Computación y Sistemas, vol. 4, núm. 2, octubre-diciembre, 2000, pp. 115-123
Instituto Politécnico Nacional
Distrito Federal, México

Available in: http://www.redalyc.org/articulo.oa?id=61540205
Limited Preemption in Real-Time Scheduling

Axel W. Krings¹ and M. H. Azadmanesh²
¹Computer Science Dept. University Idaho, Moscow, ID 83844-1010
²Dept. of Computer Science, University of Nebraska at Omaha, Omaha, NE 68182-0500
E-mail: krings@cs.uidaho.edu, azad@ahvaz.unomaha.edu

Article received on February 15, 2000, accepted on August 23, 2000

Abstract

It can be shown that priority list scheduling considering limited preemption is still subject to scheduling instabilities, where deadlines may be missed as the result of shortening run-time durations of one or more tasks. We present task starting conditions that avoid these instabilities at run-time. Together with existing methods addressing dynamic task inclusion, this model presents a low-overhead scheduling solution to hard real-time applications sensitive to task response times.

Keywords: Multi-processor scheduling, real-time dispatching, list scheduling, scheduling instabilities, limited preemption.

1 Introduction

The use of multiprocessor systems in real-time applications has been motivated by increasing component complexity or the need for functional fault tolerance through hardware redundancy. Many of these systems operate in hard real-time, where deadlines are associated with each task. Failure to meet the deadlines renders the application useless. Some of these systems are operating in safety critical applications where a missed deadline may result in catastrophe, e.g., utilize cost in terms of life, environment, or financial penalties.

Two basic scheduling disciplines exist, preemptive and non-preemptive. Preemptive scheduling offers a greater degree of flexibility and handling for higher resource utilization. However, it may be hard, if not impossible, to fully predict deterministic effects and overhead of containing (Butler, 1992; Hwu, 1994; Jeffay, 1991). Preemptive scheduling on the other hand is unpredictable. However, its main shortcoming is the lack of flexibility, inherently due to the inability to interrupt tasks once they start execution. Star list scheduling is a simple approach in non-preemptive scheduling in which tasks are ordered according to priorities in a list. At run-time, this list is scanned until the first task that is ready is selected for execution.

The advantage of list scheduling is its simplicity and the low run-time overhead. Therefore, it is suitable for real-time applications. As such, list scheduling has been implemented in projects like the Realtime Computing Platform (RCP) (Butler, 1992), the Multicomputer Architecture for Fault Tolerance (Kieckhafer, 1988), the Spring Kernel (Stankovic, 1990) for specific applications like turbojet engines.
Some hard real-time systems are sensitive to response time with respect to QoS (quality of service). Preemption offers potential for faster response to critical tasks of higher priority, by not allowing a lower priority task block the early execution of a higher priority task. Note that preemption is not necessary to achieve the design specifications with respect to QoS. The system is designed to meet the real-time requirements strictly non-preemptively. However, preemption can drastically improve response time of high priority tasks. In order to inherit the advantages of both non-preemptive and preemptive approaches, a hybrid scheduling approach is introduced that allows limited preemption. Limiting the ability to preempt tasks is not a new concept and has been discussed in the context of QoS in Bruno (1997), where Move-To-Rear List Scheduling was introduced to provide QoS guarantees.

A potential problem for hard real-time system using non-preemptive list scheduling is its vulnerability to tasking anomalies (Graham, 1969; Manacher, 1967). One anomaly is called timing anomaly. It implies that the reduction in task execution times of one or more tasks can cause deadlines to be missed. In general, task durations may not be assumed constant, because they are directly affected by, for example, memory management, communication overhead, channel contention, as well as asynchrony of autonomous input or sensor units (Lim, 1994). It will be shown that list scheduling is susceptible to timing anomalies even if one allows limited preemption.

Several stabilization methods exist that avoid timing anomalies. A-priori stabilization was introduced by Manacher (1967). Less restrictive run-time stabilization algorithms have been introduced in Krings (1994). The actual durations of tasks are often much smaller than the maximal durations, up to one order of magnitude (Carpenter, 1994). As a result, the available slack-time increases as more and more tasks finish early. A recent approach addressed higher flexibility of list scheduling by considering the inclusion of dynamically arriving tasks in addition to a static workload based on slack-time reclaiming (Krings, 1997). A more flexible approach of including dynamically arriving tasks or subgraphs of tasks at run-time has been addressed in Krings (1998). Thus solutions to stable scheduling and dispatching of workloads consisting of hard real-time workloads and dynamically arriving tasks are available for non-preemptive systems.

This paper focuses on list scheduling with limited preemption. In Section 3 the effect of limited preemption on dispatching is formalized and starting conditions are derived. These conditions are proven sufficient for safe task dispatching in Section 4. Finally Section 5 will conclude the paper and give summary.

2 Scheduling Environment

Tasks are the units of computations and are delimited by sequential execution code. The tasks are scheduled on any of $M$ homogeneous processors. Each task has an associated minimum and maximum execution time $c_{i}^{\min}$ and $c_{i}^{\max}$, release time $r_{i}$ at which the task becomes ready for execution, starting time $s_{i}$, finishing time $f_{i}$, and hard deadline $d_{i}$. Task dependencies are defined by a partial order, represented by an acyclic precedence graph.

Tasks are assumed to execute non-preemptively, i.e. one task is not allowed to preempt their corresponding non-preemption interval. However, tasks may be preempted after each non-preemption interval. Therefore, associated with each task are non-preemption intervals $\Delta t_{i}$. A task can only be preempted at integral multiples of $\Delta t_{i}$, i.e. during $T_{i}$ is essentially non-preemptive. If $\Delta t_{i} = c_{i}$ the task is non-preemptive, whereas if $\Delta t_{i} = 0$ there are no restrictions on the preemption of $T_{i}$ (Bruno et al.).

The special case where all $\Delta t_{i} = 0$ no issues arise.

In the presence of limited preemption it is important to consider the overhead resulting from context switching. A context switch cannot only be seen as the time spent setting up register contents and data in memory, but must also include cost induced by management, e.g. a task may lose its cache. This restrictive view restricts our considerations to the impact of context switches and assume that main memory is not fault prone. To avoid page faults. This practical view is to avoid large overhead since access to main memory, e.g. disk, may be orders of magnitude more expensive than access to on-chip memory. A typical application meeting this assumption is a simple, single-task, real-time system. Each task $T_{i}$ is defined as a function describing an upper bound on the load associated with preemption $T_{i}$. It is clear that the load is a function of the size of the data set of $T_{i}$.

Defining non-preemption intervals for each task $T_{i}$ provides a very powerful feature for tuning the non-preemption behavior of the task system. Under certain circumstances it may be desirable to have certain tasks that are more tolerant of preemption than other tasks. Therefore, one might wish to have the non-preemption intervals for certain tasks be larger than the minimum required. We will see in Section 3 how to derive these non-preemption intervals.

This paper focuses on list scheduling with limited preemptions. In Section 3 the effect of limited preemption on dispatching is formalized and starting conditions are derived. These conditions are proven sufficient for safe task dispatching in Section 4. Finally Section 5 will conclude the paper and give summary.
The task system described above is subjected to the list scheduling paradigm. Even though list scheduling has been traditionally used for non-preemptive scheduling, it has been adapted to consider limited preemption (Bruno, 1997). In priority list scheduling, whenever a processor becomes available, the run-time dispatcher scans the task list from left to right. The first unexecuted ready task encountered in the scan is assigned to the processor. We adopt the dispatching model presented in Deogun (1998), in which dispatching of a task is seen as an atomic operation that first updates the priority list, then selects a new task for execution, and finally exits the dispatcher. The dispatching model imposes a complete ordering on events which appear to be simultaneous on the Gantt chart by prioritizing requests according to processor indices. It should be pointed out that in our scheduling environment the traditional meaning of the terms “dispatching” and “scheduling” somewhat overlap, as will be elaborated on in Section 3.

2.1 Definitions

The following terms and definitions will be used throughout this paper and are partially restated from Krings (1998):

- **Scenario**: the schedule obtained by using a particular set of task durations.
- **Standard Scenario**: a scenario in which each task \( T_i \) uses the maximum computation time \( c_i^{\text{max}} \) (Manacher, 1967).
- **Non-Standard Scenario**: a scenario in which each task \( T_i \) executes with \( c_i^{\text{min}} \leq c_i \leq c_i^{\text{max}} \). However, at least one task \( T_j \) has duration \( c_j \) less than its maximum computation time \( c_j^{\text{max}} \), i.e. \( c_j < c_j^{\text{max}} \).
- **Standard Gantt Chart (SGC)**: the Gantt chart depicting the standard scenario. Task deadlines are the respective finishing times in the SGC, i.e. \( d_i = f_i^{\text{std}} \). References with respect to the SGC are denoted by superscript \( std \) in the respective variable, e.g. \( s_i^{\text{std}} \) or \( f_i^{\text{std}} \).
- **Non-Standard Gantt Chart (NGC)**: the Gantt chart resulting from a non-standard scenario.
- **Projective List**: the priority list from which the dispatcher selects tasks. This list is in one-to-one correspondence with the SGC, i.e. its tasks are ordered according to the time each task is picked up on the SGC (Manacher, 1967). Furthermore, without loss of generality, tasks are assumed to be ordered by increasing indices. Since the dispatcher traverses the projective list in search for a ready task, the only tasks of interest are those included in the current ready set. For task \( T_i \), we denote the earliest possible start time by \( s_i^{\text{std}} \), defined as:

\[
s_i^{\text{std}} = \max(d_j + c_j - \epsilon, j < i)
\]

where \( \epsilon > 0 \) is a sufficiently small value and \( d_j \) is the deadline of task \( T_j \). The finishing time is \( f_i^{\text{std}} = s_i^{\text{std}} + c_i \).

**Stable Schedule**: a schedule in which no scoff occurs after the finishing time of any \( T_i \) in the NGC is completed on the SGC. With non-standard computation times not known apriori, i.e. \( c_i^{\text{min}} \leq c_i \leq c_i^{\text{max}} \), given any task \( T_i \), the “deadline” for \( T_i \) is \( s_i^{\text{std}} \), the finishing time on the SGC, or the adjusted \( s_i^{\text{std}} \) resulting from a preemption. Thus, only if \( s_i \leq s_i^{\text{std}} \) can \( T_i \) be guaranteed.

Let \( T_{<i} \) denote the set of all tasks which come before \( T_i \) on the SGC, i.e. \( T_{<i} \) is the set of task indices less than \( i \). Task set \( T_{<i} \) is defined as:

\[
T_{<i} = \{ T_j | j < i \}
\]

In a given scenario, a task \( T_v \) is unstable if it is the lowest numbered task to start earlier than its immediate predecessor or if it is the second lowest numbered task to start earlier than its immediate predecessor. In either case, the task has been preempted more than once and is therefore considered on the dual-processor SGC in Figure 1b. Depending on the current state of the system, the dispatcher scans the projective list for the first ready task for execution.

NGC1 of Figure 1b demonstrates instability for the non-preemptive case, i.e. \( \Delta t_i \leq 0 \). On NGC1, \( T_3 \) is shortened by a sufficiently small value \( \epsilon \). The shortened \( T_3 \) finished first. A moment later, Task \( T_6 \) was then able to claim the processor. Task \( T_5 \) was executed on the SGC. As a result, but for the fact that \( T_7 \) started later on the NGC than they did on the SGC.

Now assume that each task \( T_i \) has a non-adjacent interval equal to half its standard duration, i.e. \( c_i^{\text{std}}/2 = 0.5 \). Again \( T_3 \) is shortened by a sufficiently small value \( \epsilon \) and \( T_6 \) was then able to claim the processor, as can be seen in NGC2 of Figure 1b. When tasks are released at time \( f_2 = 2 \) only \( T_4 \) can be started. Task \( T_5 \) can only be preempted after \( \Delta t_i = 0.5 \). Consequently \( T_5 \) and \( T_7 \) start late at time \( s_7 = 3.5 - \epsilon \). It should be noted that the segment of \( T_6 \) executing on processor \( P_1 \) is then not preempted by the cost \( C(T_6) \).

In order to avoid instabilities in non-preemptive list scheduling, two basic stabilizations have been proposed, i.e. *apriori* and *run-time* list scheduling.

2.2 Instability and Limited Preemption

We want to demonstrate scheduling instability in three scenarios based on the precedence graph in Figure 1b. The precedence graph contains seven tasks, \( T_1 \) through \( T_7 \), with the minimum durations of unity, and a maximum of five tasks. Priorities are defined in order of increasing distance on the dual-processor SGC in Figure 1b. Depending on the current state of the system, the dispatcher scans the projective list for the first ready task for execution.

NGC1 of Figure 1b demonstrates scheduling instability for the non-preemptive case, i.e. \( \Delta t_i \leq 0 \). On NGC1, \( T_3 \) is shortened by a sufficiently small value \( \epsilon \). The shortened \( T_3 \) finishes first. A moment later, Task \( T_6 \) is then able to claim the processor. Task \( T_5 \) is executed on the SGC. As a result, but for the fact that \( T_7 \) started later on the NGC than they did on the SGC.

Now assume that each task \( T_i \) has a non-adjacent interval equal to half its standard duration, i.e. \( c_i^{\text{std}}/2 = 0.5 \). Again \( T_3 \) is shortened by a sufficiently small value \( \epsilon \) and \( T_6 \) is then able to claim the processor, as can be seen in NGC2 of Figure 1b. When tasks are released at time \( f_2 = 2 \) only \( T_4 \) can be started. Task \( T_5 \) can only be preempted after \( \Delta t_i = 0.5 \). Consequently \( T_5 \) and \( T_7 \) start late at time \( s_7 = 3.5 - \epsilon \). It should be noted that the segment of \( T_6 \) executing on processor \( P_1 \) is then not preempted by the cost \( C(T_6) \).

In order to avoid instabilities in non-preemptive list scheduling, two basic stabilizations have been proposed, i.e. *apriori* and *run-time* list scheduling.
case have motivated the development of less restrictive stabilization methods. *Run-time stabilization* is a less restrictive stabilization method, where the dispatcher limits the depth of its scan into the task list in order to avoid instabilities. This approach takes advantage of information available at run-time (Krings, 1993; Krings, 1994). Limited preemptions to aid task response times are of course highly run-time dependent. Therefore run-time stabilization is the logical choice.

3 Stability and Limited Preemption

3.1 Limited Preemption

A task $T_p$ can only be preempted at multiples of its non-preemption interval $\Delta t_p$. The cost for the preemption is $C(T_p)$. Thus if $c_p$ is the amount of time that $T_p$ has executed at the time of the preemption, then the remaining execution time needs to be adjusted by $C(T_p)$, i.e. $c_p^{\text{max(new)}} = c_p^{\text{max(old)}} - c_p + C(T_p)$. If we assume that every task that is started or selected for dispatching is *marked* (or “shaded”) on the SGC, then $c_p^{\text{max(new)}}$ is the amount of time that needs to be *unmarked* (or “unshaded”) on the SGC upon preemption, i.e. its adjusted remaining execution time needs to be “re-inserted” into the SGC.

$T_p$ is in such a way that $f_{p}^{\text{std}}$ is not changed by preemption. Thus, $T_p$ is right-adjusted in its current SGC slot. Re-inserting tasks with adjusted execution times into the SGC can have as a consequence that task in the SGC that reflect their original SGC starting order are not in this case tasks need to be renumbered in order to maintain a projective list. For ease of discussion, it will not be mentioned explicitly. An adjustment does not require renumbering is to re-insert $s_p^{\text{std}}$ whenever possible. This way after re-insertion $T_p$ the standard starting times will remain unchanged.

Both re-insertion approaches are justifiable.

3.2 Dispatching

Before describing the function of the re-insertion, a few definitions are needed. Let $k$ be the number of unmarked (unshaded) tasks $T_j$ at time $t$. Thus $U(t)$ is the number of unmasked tasks $T_j$ at $t$, excluding the task currently considered by the dispatcher. Furthermore, let $E(t)$ be the number of tasks $T_j$ that are currently executing (on the SGC) and for which $f_{j}^{\text{max}} > t$. Thus $E(t)$ considers all tasks that could be still running.

In the following discussion, a task $T_p$ is called *peemptable task* if it has a non-preemption time.

We assume that the preemption cost is $c_p$. We want to guarantee progress in that a peemptable task will cause the adjusted execution time to exceed the standard execution time. We call this property *Preemption Hypothesis*. The motivation for the hypothesis is to ensure that tasks will not be re-inserted into its original SGC slot.

At this point in some discussion issues of scheduling and dispatching. By definition, the dispatcher is distinct from the scheduler. The scheduler is executed only once at design time to generate a SGC for all real-time requirements. The dispatcher, on the other hand, decides at run-time which tasks is to be scheduled.

In the context of this paper issues of preemption involves task re-insertion into the SGC, and we use a scheduler that reschedules the entire task set at run-time, we use the term dispatcher in a general sense, thus allowing it to re-insert tasks.

\[\text{a) Precedence Graph.} \quad \text{b) $T_5$ shortened by } \epsilon \text{ on the NGCs.}\]
which expires at or after the time of the current scan. We will reserve subscript \( p \) to indicate preemption tasks.

Assume \( T_i \) is the next ready task. The following two situations are possible:

1. No preemtable task with higher index (and thus lower priority) is executing, i.e. no \( T_p \) with \( p > i \) is executing.

2. At least one preemtable task is executing that has a higher index than \( i \), i.e. one or more \( T_p \) with \( p > i \) are executing and \( T_p \) has at least one preemption point that has not expired yet.

3. Similar to the previous situation at least one preemtable task is executing that has a higher index than \( i \), however all preemption points have expired.

Let \( t_n \) be the time of the current scan (the time “now”). It will be shown that in the first and third case task \( T_i \) can be started if for the entire interval \([t_n, t_n + c^{max}_i]\) the number of unstarted tasks on the SGC plus the number of executing tasks is less than the number of processors \( M \). Thus in order to start \( T_i \), for any \( t \) in \([t_n, t_n + c^{max}_i]\) we need

\[
U(t) + E(t) < M.
\]  

Note that the first and third situation are essentially the same in that task vulnerabilities need to be considered in the absence of preemption. Furthermore, note that by considering \( T_i \) for execution, it will affect \( U(t) \) since it is now considered to be marked during the condition check. For example, let’s assume an initial situation of the workload at time \( t = 0 \) in which there are \( M \) tasks scheduled on the SGC. At this time all tasks are unstarted and no task has yet been dispatched. Now \( T_i \) is picked up by the dispatcher, thus reducing \( U(0) \) to \( M - 1 \). With no task executing yet \( E(0) \) is still zero. Thus inequality (1) is satisfied since \( M - 1 + 0 < M \) and \( T_i \) can be started.

In the second case we may have to consider preemption. If for any \( t \) in \([t_n, t_n + c^{max}_i]\) inequality (1) does not hold, we must attempt to reduce \( E(t) \) using preemptions. Let \( P(t) \) be the number of preemptions needed to satisfy inequality (1) at time \( t \). Then

\[
P(t) = U(t) + E(t) - (M - 1)\]

is the number of preemptions needed at \( t \). Thus we have

### 3.3 Safe Task Starting Conditions

The safe task starting conditions presented here directly from the discussion of the three situations described in the previous subsection.

**Safe Task Starting Conditions:** A ready task \( T_j \) can be safely started if for each \( T_p \) with \( s^{std}_j \) in \([t_n, t_n + c^{max}_i]\),

1. \( U(s_j) + E(s_j) < M \) or
2. \( U(s_j) + E(s_j) \geq M \) and \( P(s_j) \) tasks can be preempted such that
   
   (a) \( \forall T_p \) re-insertion is feasible. Thus adjusted remaining execution time of \( T_p \) does not cause \( f^{std}_p \) to be exceeded,
   
   \[
c^{max}_p - c^{min}_p + C(T_p) \leq f^{std}_p, \text{ where } c^{min}_p \text{ is an } \text{ optimal } c^{max}_p \text{ that has already been computed}.
   
   (b) \( T_p \) cannot have been mortgaged to just prior to starting of a task other than \( T_i \).

Condition 1 covers the first and third situations described in the previous subsection. Condition 2 addresses preemption, which of course results in insertion into the SGC at the time of the preemption by the dispatcher. It should be noted that if task \( T_p \) is re-inserted using its adjusted run time it effectively an unstarted task. If the re-inserted \( T_p \) is subject to the new \( s^{std}_p \) to be in the interval \([t_n, t_n + c^{max}_i]\), \( T_p \) itself will be subjected to the two conditions.

Condition 2b prevents several \( T_i \) from using preemption of \( T_p \) to justify the availability of a processor. Mortgaging \( T_p \) can be easily realized by a set equal to the index of the task that relies on preemption, i.e. task \( T_i \).

The safe task starting conditions can be used in an algorithmic fashion by the run-time dispatcher. The safe starting of the highest priority task \( T_i \) is the processor is available to start \( T_i \), preemption is impossible. One question that might arise is which task to preempt in case there are several preemptable tasks executing. Since actual task durations are not a priori known, i.e. they are between \( c^{min}_i \) and \( c^{max}_i \), optimal selection of preemptable tasks may be or impossible. However, with respect to resource preemption the lowest priority task would be a simple strategy. Another option might be to attempt minimization of preemptions or total preemption cost.
Such systems may include dynamic tasks in addition to the static core workload. Priority list scheduling is traditionally a static approach. Feasibility tests for dynamic task inclusion into the workload have been presented in Krings (1998) for different dynamic task models. These tests can be adapted for the scheduling environment allowing limited preemption. We will demonstrate this by considering the simple special case of non-preemptive independent dynamic tasks, i.e., dynamic tasks \( T_i \) with \( \Delta t_i = c_i^{max} \). High priority dynamic tasks are allowed to enter the task system conditioned on a run-time feasibility test. This test essentially employs the safe task starting conditions and goes beyond traditional slack-time reclaiming.

In Subsection 3.2 and 3.3 a counting argument was stated that, in order to prevent instability, avoids processor contention. In other words, there cannot be any time during the execution of a task \( T_i \) in which the number of executing and unstarted tasks exceeds the number of processors. The same arguments can be made in order to justify inclusion of dynamic tasks. The result is a feasibility test that is based on the safe task starting conditions:

**Feasibility Conditions:** A dynamic ready task \( T_i \) with \( \Delta t_i = c_i^{max} \) can be safely started if the Safe Task Starting Conditions of Subsection 3.3 hold.

### 4 Proof of Correctness

It will be shown that the Safe Task Starting Conditions just described are sufficient to avoid instability, but first some definitions are needed.

Assume that a priority inversion occurs so that task \( T_x \) starts before \( T_v \), i.e., \( s_x < s_v \), where \( x > v \). Then \( T_x \) is called a usurer task. Recall that \( T_{<v} \) denote the set of all tasks which started before \( T_v \) on the SGC. The **Leftover Set** \( L \) is defined as all tasks in \( T_{<v} \) which have not finished by \( s_x \). Specifically, \( L = \{ T_i \in T_{<v} : f_i > s_x \} \).

Next, we want to define the status of a task set \( T \) to be the number of tasks in \( T \) executing, ready to execute, or waiting to be executed. Let \( M_T(t) \) denote the number of tasks in \( T \) running at time \( t \), i.e., the number of processors occupied by \( T \) at time \( t \). Furthermore, let \( W_T(t) \) be the number of tasks in \( T \) which are ready but waiting for a processor at time \( t \). Lastly, let \( R_T(t) = M_T(t) + W_T(t) \). Thus, \( R_T(t) \) is the total number of tasks in \( T \).

**Lemma 1** In the presence of limited preemption, a scenario can be unstable at \( T_v \) only if the number of processors available to \( L \) at \( s_v^{std} \) is less than the number of processors occupied by \( L \) at \( s_v^{std} \) on the SGC, i.e.,

\[
M - M_X(s_v^{std}) < M_L^{std}(s_v^{std}),
\]

where \( X \) is usurer tasks \( T_x \).

**Proof:** By the on-time hyposthesis, the number of processors in \( L \) which are ready or running at \( s_v^{std} \) cannot exceed the number which were running at \( s_v^{std} \) on the SGC. This is not only the non-preemptive case but also in the limited preemption case, since adjusted tasks can be re-inserted only into the original SGC at \( s_v^{std} \). Therefore, \( R_L(s_v^{std}) \leq R_L^{std}(s_v^{std}) = M_L^{std}(s_v^{std}) \). Thus, \( L \) will require no more processors available on the SGC, so that \( T_v \) is unstable only if fewer processors are available. The number of processors available to \( L \) at \( s_v^{std} \) is simply \( M - M_X(s_v^{std}) \).

**Lemma 2** Assume that a usurer task \( T_x \) at time \( s_x \), and define \( f_x^{max} = s_x + c_x^{max} \). In the presence of limited preemption now \( s_x^{std} > f_x^{max} \) can become unstable as a result of \( T_x \).

**Proof:** Assume \( T_v \) is any task with \( s_v^{std} < s_x \), i.e., \( T_v \) is any task whose standard starting time does not overlap time-wise with the execution on NGC. Recall that an unstable task by definition numbered task to start late. Then all \( T_i \in T_{<v} \) are on time. Lemma 1 states that \( T_v \) is unstable only if the number of processors at \( s_v^{std} \) is less than the number of processors by \( L \) at \( s_v^{std} \) on the SGC. If \( T_v \) had \( T_v \) would be stable and thus at \( s_v^{std} \), \( M - M_X(s_v^{std}) \geq M_L^{std}(s_v^{std}) \). However in any case of \( T_x \) we still have \( M - M_X(s_v^{std}) \geq M_L^{std}(s_v^{std}) \), \( f_x^{max} < s_v^{std} \), i.e., \( T_v \) has finished and the number of processors available cannot increase. Thus \( T_x \) has no impact of processors available to \( L \) at \( s_v^{std} \) and Lemma 1.

We are now ready to prove that the Safe Task Starting Conditions presented in Subsection 3.3 are sufficient to avoid instability. To enhance readability, they are restated in the following theorem.

**Theorem 1** A ready task \( T_i \) can be safely started at each \( T_i \) with \( s_i^{std} \in [t_i^1, t_i^2 + c_i^{max}] \).
of the task duration, i.e. \([t_n, t_n + c_i^{max}]\). This is captured in inequality (1) and inequality (2) which are rewritten as \(U(t) + \varepsilon(t) \leq M - 1\) and \(U(t) + \varepsilon(t) \leq M - 1\) respectively. Recall that the term \(M - 1\) represents the number of processors needed to start \(T_i\). With respect to the safe task starting conditions of Theorem 1, the processor is selected and therefore unmarked on the SGC as it is determined during the Safe Task Starting Condition check. The marking of \(T_i\) has no effect on the number of unstarted tasks \(U(t)\) overlapping with the SG execution of \(T_i\), in \([s_i^{std}, f_i^{std}]\). If the Safe Task Starting Condition does not hold, \(T_i\) is unmarked, reflecting that it cannot be started after all.

Now we assume that \(T_i\) is a dynamic task and enter the system conditioned on the feasibility set of \(T_i\) has no representation on the SGC and therefore no effect on \(U(t)\) for any \(t\), i.e. it does not reach during the check. Still, both conditions of the theorem ensure that one processor is available for causing contention for any task on the SGC. Hence, the Safe Task Starting Conditions still hold to ensure that \(T_i\) can be safely started if the conditions are met.

5 Summary

This paper addressed dispatching in real-time systems where task response time is a concern. It subsumes a system represented by a task graph to the limiting paradigm under the consideration of limiting conditions. Preemption points are defined for each interval, and the system is represented on non-preemption intervals. After each execution of a non-preemption interval, tasks can be pre-empted in order to improve response time to higher priority tasks. It has been shown that similar to non-preemption scheduling, the inclusion of limited preemption in the task system vulnerable to timing anomalies under this scheduling paradigm, unavoidable variations in task durations at run-time can result in instantaneous or no more tasks execute for less than their duration.

Safe task starting conditions have been derived that avoid these anomalies. The conditions are used in an algorithmic fashion to dispatch tasks or they can be used by task designers and schedulers for deriving more efficient dispatching or scheduling algorithms. The dispatcher can be augmented with the Feasibility Conditions to allow inclusion of dynamically arriving tasks.
References


Axel W. Krings received the Dipl.Ing. in Electrical Engineering from the FH-Aachen, Germany, in 1982 and his M.S. and Ph.D. degrees in Computer Science from the University of Nebraska - Lincoln, in 1991 and 1993, respectively. He is an assistant professor at the Department of Computer Science at the University of Idaho. His research interests include: Survivable and Fault-Tolerant Systems, Distributed Real-Time Systems, Data Communication, and Scheduling Theory. Dr. Krings is a member of the IEEE Computer and Reliability Society.

M.H. Azadmanesh (Azad) received the BS (1976) degree in Cost Accounting, and the MS (1982) and PhD (1993) degrees in Computer Science from Iowa State University and University of Nebraska-Lincoln respectively. He is currently an associate professor in the Department of Computer Science at the University of Nebraska-Omaha. His research interests include Fault-Tolerant and Survivable Systems, Distributed Agreement, Reliability Modeling, and Network Communication. Azadmanesh is a senior member of the IEEE.