QoS Considerations in Real-Time Scheduling*

A.W. Krings
Computer Science Dept.
University of Idaho
krings@cs.uidaho.edu

M.H. Azadmanesh
Computer Science Dept.
University of Nebraska at Omaha
azad@cmit.unomaha.edu

Abstract

This paper addresses real-time issues of scheduling tasks on multi-processors. A special hybrid task-model found in QoS applications is used that allows tasks to be preempted only at certain preemption points. These preemption points are defined for individual tasks by non-preemption intervals. 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 to address dynamic task inclusions, this model presents a low-overhead scheduling solution to hard real-time applications with QoS concerns.

1 Introduction

The use of multiprocessor systems in real-time applications has been motivated by increasing computational 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 may render the application useless. Some of these systems are operating in safety critical applications where missing a deadline may result in catastrophe, e.g., unacceptable cost in terms of live, environment, or finance.

Two basic scheduling disciplines exist, preemptive and non-preemptive. Preemptive scheduling generally offers a greater degree of flexibility and has potential for higher resource utilization. However, it proves to be hard, if not impossible, to fully predict the non-deterministic effects and overhead of context switching [2, 5, 6]. Non-preemptive scheduling on the other hand is more predictable. However, its main shortcoming is the lack of flexibility, inherently due to the inability to interrupt tasks once they start execution. Static priority list scheduling is a simple approach in non-preemptive scheduling in which tasks are ordered according to their priorities in a list. At run-time, this list is scanned and the first task that is ready is selected for execution. List scheduling has been implemented in projects like the Reliable Computing Platform (RCP) [2], the Multicomputer Architecture for Fault Tolerance (MAFT) [7], the Spring Kernel [17], or specific applications like turbojet engine control [15].

Some hard real-time systems have Quality of Service (QoS) requirements. Traditional QoS characteristics include fairness, delay, and/or delay jitter. However, for hard real-time systems QoS imposes the requirement that due to the hard deadlines no tasks may be late. Thus delay and delay jitter can be eliminated from consideration. This means that QoS may be seen as a strict quality improvement based on task response time and fairness. On the other hand, preemption offers potential for faster response and thus more flexibility in QoS. In order to inherent the advantages of both 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 [1] 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 [13]. 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 asynchronousy of autonomous input or sensor units [12]. It will be shown that list scheduling is susceptible to timing anomalies if one allows limited preemption.

Several stabilization methods exist that avoid tim-

*This work has been supported in part by the Idaho National Engineering Laboratories (INEEL) and the DoD Technical Support Working Group (TSWG).

ing anomalies. *A-priori stabilization* was introduced by Manacher [13]. Less restrictive *run-time stabilization* algorithms have been introduced in [9]. 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 [10]. The actual durations of tasks are often much smaller than the maximal durations, up to one order of magnitude [3]. As a result, the available slack-time increases as more and more tasks finish early. The flexibility of including dynamically arriving tasks or subgraphs of tasks at run-time has been addressed in [11]. 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 order to increase QoS. Our particular QoS focus is on improving the response time of tasks. We show that, like in non-preemptive systems, applications implementing limited preemption are also subject to timing anomalies and therefore need to address the issue of stabilization if applied to hard real-time systems. Section 2 will supply background information, definitions and an example of instability in the presence of limited preemption. In Section 3 the effect of limited preemption on dispatching is formalized and safe dispatching conditions are derived. These dispatching conditions are proven sufficient for safe task dispatching in Section 4. Finally Section 5 will conclude the paper with a summary.

2 Scheduling Environment

Tasks are the units of computations and they consist of sequentially executing code. The tasks may be run on any of M homogeneous processors. Each task T_i has an associated minimum and maximum computation 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 a directed acyclic precedence graph.

Tasks are assumed to execute non-preemptively during their corresponding *non-preemption intervals*. However, tasks may be preempted after each non-preemption interval. Therefore, associated with each task T_i is a non-preemption interval Δt_i. A task can only be preempted at integral multiples of Δt_i, i.e., during Δt_i task T_i is essentially non-preemptive. If $\Delta t_i = c_i^{max}$ the task is non-preemptive, whereas if $\Delta t_i = 0$ there are no restrictions on the preemption of T_i [1]. For the special case where all $\Delta t_i = 0$ no instabilities can arise.

In the presence of limited preemption it is necessary to consider the overhead resulting from context switching. A context switch can not only be seen as storing and setting up register contents and data structures, but must also include cost induced by memory management, e.g., a task may lose its cache image. We restrict our considerations to the impact of cache line misses and assume that main memory is large enough to not cause page faults. This practical consideration is to avoid large overhead since access outside of main memory, e.g., disk, may be orders of magnitude slower. A typical application meeting this assumption is a diskless embedded system. For each T_i we define $\mathcal{C}(T_i)$ as a function describing an upper bound on the cost associated with preempting T_i. It is clear that $\mathcal{C}(T_i)$ is a function of the size of the data set of T_i.

Defining non-preemption intervals for each individual T_i provides a very powerful feature for tuning the run-time behavior of the task system. To improve QoS this means that, under consideration of the cost, the response time to critical tasks can be improved by decreasing the non-preemption intervals of less critical tasks. This increases the probability that non-critical tasks are preempted in favor of executing a critical task. This will be shown in Section 3 where task preemptions may be the key for safe dispatching of high priority tasks.

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 [1]. 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 [4], 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 is important to see that the dispatcher is distinct from the scheduling algorithm. The scheduler is executed only once at design time for static workloads and invoked at run-time for dynamically arriving tasks in form of a feasibility check [11]. At run-time, the dispatcher simply arbitrates tasks.

2.1 Definitions

The following terms and definitions will be used throughout this paper and are partially restated from [11]:

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^{max} [13].

Non-Standard Scenario: a scenario in which each task T_i executes with $c_i^{min} \leq c_i \leq c_i^{max}$. However, at least one task T_j has duration c_j less than its maximum computation time c_j^{max}, i.e., $c_j < c_j^{max}$.

Standard Gantt Chart (SGC): the Gantt chart depicting the standard scenario. Task deadlines are the respective finishing times in the SGC, i.e., $d_i = f_i^{std}$. References with respect to the SGC are denoted by superscript *std* in the respective variable, e.g. s_i^{std} or f_i^{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 [13]. 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 in this list are the unstarted tasks. In the context of limited preemption, tasks that have been preempted can be seen as special "unstarted tasks" with computation times adjusted by the amount that has been executed before the last preemption, considering the preemption cost.

Stable Schedule: a schedule in which no scenario exists where the finishing time of any T_i in the NGC exceeds its completion time on the SGC. With non-standard computation times not known apriori, i.e., $c_i^{min} \leq c_i \leq c_i^{max}$, given any task T_i, the "deadline" for s_i is s_i^{std}, the starting time on the SGC, or the adjusted s_i^{std} resulting from a preemption. Thus, only if $s_i \leq s_i^{std}$ can $f_i \leq f_i^{std}$ be guaranteed.

Fan-out: occurs when a task causes the release of a second task executing in parallel with the first task. The task initiating the release is called a *forking task*. The release is called a *logical fork*, since it causes the occupancy of one more processor. Fan-out, caused by logical forks, is a necessary condition for instability to occur.

Fan-out task: a task descendent from an unfinished forking task, and it started executing on the SGC while another descendent of the same forking task was executing on another processor. An example of a fan-out task is the second child of a parent task, assuming this child started execution while the first child was executing. This can be observed in Figure 1, where parent T_2 has children T_4 and T_5. Task T_5 is a fan-out task, as can be verified on the SGC.

Let $\mathbf{T}_{<i}$ denote the set of all tasks which started before T_i on the SGC, i.e., $\mathbf{T}_{<i}$ is the set of tasks with indices less than i. Task set $\mathbf{T}_{\leq i}$ is defined as $\mathbf{T}_{<i} \cup \{T_i\}$. In a given scenario, a task T_v is *unstable* if and only if it is the lowest numbered task to start late, i.e., if (1) $s_v > s_v^{std}$, and (2) $s_i \leq s_i^{std}\ \forall\ T_i \in \mathbf{T}_{<v}$. The second condition is called the *On-Time Hypothesis* [4]. Task T_v is *vulnerable* to instability if there exists *any* scenario in which T_v is unstable.

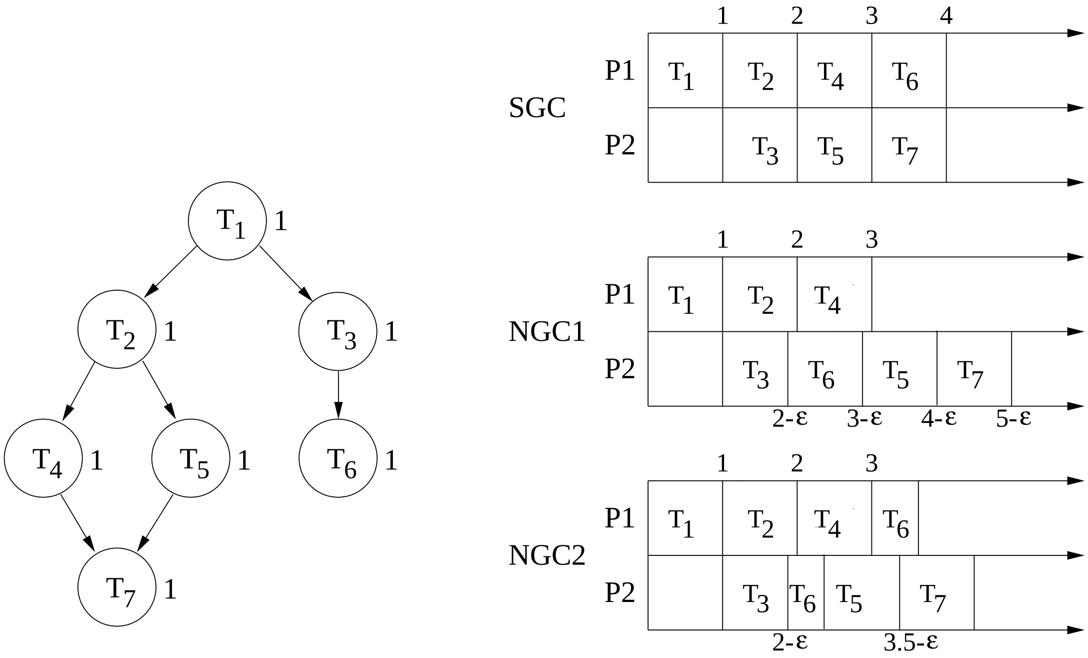
2.2 Instability and Limited Preemption

We want to demonstrate scheduling instability using two scenarios based on the precedence graph in Figure 1a. The precedence graph contains seven tasks, with maximum durations of unity, listed next to each vertex. Task priorities are defined in order of increasing start times on the dual-processor SGC in Figure 1b. During execution, the dispatcher scans the projective list and selects the first ready task for execution.

NGC1 of Figure 1b demonstrates scheduling instability for the non-preemptive case, i.e., $\Delta t_i = c_i^{std}$ for $1 \leq i \leq n$. On NGC1, T_3 is shortened by an arbitrarily small value ϵ. The shortened T_3 finished before T_2. Task T_6 was then able to claim the processor on which T_5 was executed on the SGC. As a result, both T_5 and T_7 started later on the NGC than they did on the SGC.

Now assume that each task T_i has a non-preemption interval equal to half its standard duration, i.e., $\Delta t_i = c_i^{std}/2 = 0.5$. Again T_3 is shortened by an arbitrarily small value ϵ and T_6 was then able to claim the processor as can be seen in NGC2 of Figure 1b. When T_4 and T_5 are released at time $f_2 = 2$ only T_4 can be started. Task T_6 can only be preempted after $\Delta t_i = 0.5$ time units. Consequently T_5 and T_7 start late at time $s_5 = 2.5 - \epsilon$ and $s_7 = 3.5 - \epsilon$. It should be noted that the second segment of T_6 executing on processor $P1$ incorporates the preemption cost $\mathcal{C}(T_6)$.

In order to avoid instabilities in non-preemptive priority list scheduling two basic stabilization methods have been proposed, i.e., *apriori* and *run-time* stabilization. In *apriori stabilization* methods, stabilization is achieved by fixing the task starting sequence or task starting times, or by introducing additional precedence constraints [7, 13, 14, 16]. Potentially poor utilization for the first and addition of many edges in the second case have motivated the development of less restrictive stabilization methods. *Run-time stabilization* is a less restrictive stabilization method, where the dispatcher



a) Precedence Graph b) T_3 shortened by ϵ on the NGCs

Figure 1: Example of Instability

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 [8, 9]. Limited preemptions to aid QoS 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 Δt_p. The cost for the preemption is $\mathcal{C}(T_p)$. 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 $\mathcal{C}(T_p)$, i.e., $c_p^{max(new)} = c_p^{max(old)} - c'_p + \mathcal{C}(T_p)$. If we assume that every task that is started is marked (or "shaded") on the SGC, then this is the amount of time that needs to be unmarked (or "un-shaded") on the SGC, i.e., its adjusted remaining execution time needs to be "re-inserted" into the SGC. To avoid re-scheduling[1] of the SGC, we assume that the re-insertion of T_p by the run-time dispatcher is such way that f_p^{std} is not changed by preemption. Thus, T_p is right-adjusted in its old SGC slot. Re-inserting tasks with adjusted execution times in the SGC can have as a consequence that task indices do not reflect their SGC starting order. In this case tasks need to be renumbered in order to maintain a projective list. For ease of discussion we assume that this adjustment is made upon task re-insertion and it will not be mentioned explicitly. An approach that does not require renumbering is to re-insert T_p at time s_p^{std} whenever possible. This way after re-insertion of T_p the standard starting times will remain unchanged. With respect to QoS both re-insertion approaches are justifiable.

3.2 Dispatching

Before describing the function of the run-time dispatcher a few definitions are needed. Let $\mathcal{U}(t)$ indicate the number of unmarked (unshaded) tasks in the SGC at time t. Furthermore, let $\mathcal{E}(t)$ be the number of tasks T_j that are currently executing (on a processor) for which $f_j^{max} > t$. Thus $\mathcal{E}(t)$ counts all tasks that could be still running at t.

In the following discussion, a task T_p is called a *preemptable task* if it has a non-preemption interval Δt_p which expires at or after the time of the current scan.

[1]Recall that scheduling is done once at design time.

We will reserve subscript p to indicate preemptable tasks.

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

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

2. At least one preemptable 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.

Let t_n be the time of the current scan (the time "now"). It will be shown that in the first case task T_i can be started if for the entire interval $[t_n, t_n + c_i^{max}]$ the number of unmarked tasks on the SGC plus the number of executing tasks is less than the number of processors M. Thus for any t in $[t_n, t_n + c_i^{max}]$ we need

$$\mathcal{U}(t) + \mathcal{E}(t) < M. \qquad (1)$$

In the second case we have to consider preemption. Therefore, if for any t in $[t_n, t_n + c_i^{max}]$ inequality 1 does not hold, we must attempt to reduce $\mathcal{E}(t)$ using preemptions. Let $\mathcal{P}(t)$ be the number of preemptions needed to satisfy inequality 1 at time t. Then $\mathcal{P}(t) = \mathcal{U}(t) + \mathcal{E}(t) - (M - 1)$ is the minimum number of preemptions needed at t. Note that $M - 1$ addresses the one processor needed to start T_i. If at any t in $[t_n, t_n + c_i^{max}]$ $\mathcal{P}(t)$ exceeds the number of preemptable tasks, then inequality 1 cannot be satisfied and, as will be shown, T_i cannot be safely started.

3.3 Safe Dispatching Conditions

The safe dispatching conditions presented here follow directly from the discussion of the two situations described in the previous subsection.

Dispatching Conditions: A ready task T_i can be safely started if for *each* T_j with s_j^{std} in $[t_n, t_n + c_i^{max}]$

1. $\mathcal{U}(s_j) + \mathcal{E}(s_j) < M$ or

2. $\mathcal{U}(s_j) + \mathcal{E}(s_j) \geq M$ and $\mathcal{P}(s_j)$ tasks can be preempted such that

   (a) $\forall\ T_p$ the adjusted remaining execution time plus $\mathcal{C}(T_p)$ does not exceed f_p^{std}, i.e., $s_p^{std} + c_p^{max} - c_p' + \mathcal{C}(T_p) \leq f_p^{std}$, where c_p' is the fraction of c_p^{max} that has already been executed,

   (b) T_p cannot have been mortgaged to justify safe starting of a task other than T_i.

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

The safe dispatching conditions can be used in an algorithmic fashion by the run-time dispatcher to justify the safe starting of the highest priority task T_i. If no processor is available to start T_i, preemption is considered. One question that might arise is which task should be preempted in case there are several preemptable tasks executing. Since actual task durations are not known apriori, i.e., they are between c_i^{min} and $c_i^{max} = c_i^{std}$, optimal selection of preemptable tasks may be difficult or impossible. However, with respect to QoS, preempting the lowest priority task would be a reasonable greedy approach. Another strategy would be to minimize the number of preemptions or to minimize the total preemption cost.

Many real-time systems allow dynamic changes in their workloads, e.g., new tasks may enter the system. Such systems typically consist of a static core and a dynamic workload. Priority list scheduling is traditionally a static approach. However, feasibility tests for dynamic task inclusion into the workload have been presented in [11]. These tests can be used to augment the safe dispatching conditions.

4 Proof of Correctness

It will be shown that the Dispatching 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 *usurper task*. Recall that $\mathbf{T}_{<v}$ denote the set of all tasks which started before T_v on the SGC. The *Leftover Set* $\mathbf{L}$ is defined as all tasks in $\mathbf{T}_{<v}$ which have not finished by s_x. Specifically, $\mathbf{L} = \{T_l \in \mathbf{T}_{<v} : f_l > s_x\}$.

Next, we want to define the *status* of a task set $\mathbf{T}$ to be the number of tasks in $\mathbf{T}$ executing, ready to execute, or waiting to be executed. Let $M_{\mathbf{T}}(t)$ denote the number of tasks in $\mathbf{T}$ running at time t, i.e., the number of processors occupied by $\mathbf{T}$ at time t. Furthermore, let $W_{\mathbf{T}}(t)$ be the number of tasks in $\mathbf{T}$ which are ready but waiting for a processor at time t. Lastly, let $R_{\mathbf{T}}(t) = M_{\mathbf{T}}(t) + W_{\mathbf{T}}(t)$, Thus, $R_{\mathbf{T}}(t)$ is the total number of tasks in $\mathbf{T}$ which are either ready or running at t. When we address R, M, and W with respect to the SGC, we use the notation $R_{\mathbf{T}}^{std}(t)$, $M_{\mathbf{T}}^{std}(t)$, and $W_{\mathbf{T}}^{std}(t)$.

The following two lemmas build the basis for our proof of correctness.

Lemma 1 *In the presence of limited preemptions a scenario can be unstable at T_v only if the number of processors available to* $\mathbf{L}$ *at* s_v^{std} *is less than the number of processors occupied by* $\mathbf{L}$ *at* s_v^{std} *on the SGC, i.e., only if* $M - M_X(s_v^{std}) < M_L^{std}(s_v^{std})$, *where* $\mathbf{X}$ *is the set of usurper tasks* T_x.

Proof: By the on-time hypothesis, the number of tasks in $\mathbf{L}$ which are ready or running at s_v^{std} on an NGC can not exceed the number which were ready or running at s_v^{std} on the SGC. This is not only true in the non-preemptive case but also in the limited preemption case, since adjusted tasks can be re-inserted after preemptions only into the original SGC slot. On the SGC, T_v was on time so that $W_L^{std}(s_v^{std}) = 0$. Therefore, $R_L(s_v^{std}) \leq R_L^{std}(s_v^{std}) = M_L^{std}(s_v^{std})$. Thus, on any NGC, $\mathbf{L}$ will require no more processors at s_v^{std} than it did on the SGC, so that T_v is unstable only if there are fewer processors available. The number of processors available to $\mathbf{L}$ at s_v^{std} is simply $M - M_X(s_v^{std})$. □

Lemma 2 *Assume that a usurper task* T_x *has started at time* s_x, *and define* $f_x^{max} = s_x + c_x^{max}$. *Then even in the presence of limited preemption no task* T_v *with* $s_v^{std} > f_x^{max}$ *can become unstable as a result of starting* T_x.

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

We are now ready to prove that the dispatching conditions presented in Subsection 3.3 are sufficient to avoid instability. To enhance readability, the conditions are restated in the following theorem.

Theorem 1 *A ready task* T_i *can be safely started if for* each T_j *with* s_j^{std} *in* $[t_n, t_n + c_i^{max}]$

1. $\mathcal{U}(s_j) + \mathcal{E}(s_j) < M$ *or*
2. $\mathcal{U}(s_j) + \mathcal{E}(s_j) \geq M$ *and* $\mathcal{P}(s_j)$ *tasks can be preempted such that*
   (a) $\forall\ T_p$ *the adjusted remaining execution time plus* $\mathcal{C}(T_p)$ *does not exceed* f_p^{std}, *i.e.,* $s_p^{std} + c_p^{max} - c_p' + \mathcal{C}(T_p) \leq f_p^{std}$, *where* c_p' *is the fraction of* c_p^{max} *that has already been executed,*
   (b) T_p *cannot have been mortgaged to justify safe starting of a task other than* T_i.

Proof: Let T_v be any unstarted task with standard starting time s_v^{std} in $[t_n, t_n + c_i^{max}]$. Tasks with standard starting times outside of this interval need not be considered by Lemma 2.

Condition 1: By Lemma 1 T_v can only be unstable if $M < M_X(s_v^{std}) + M_L^{std}(s_v^{std})$. This can only be the case if processor contention arises at s_v^{std}. However, since T_i is only started if $\mathcal{U}(s_v) + \mathcal{E}(s_v) \leq M - 1 < M$, this implies that there is at least one processor available at s_v^{std}. As a result processor contention is not possible and the instability condition of Lemma 1 cannot be met.

Condition 2: Since condition 1 cannot be met we must rely on $\mathcal{P}(s_v)$ preemptions. Thus $\mathcal{U}(s_v) + \mathcal{E}(s_v) - \mathcal{P}(s_v) \leq M - 1 < M$. However, similar to part 1, with the restrictions in condition 2, processor contention cannot occur and the instability condition of Lemma 1 can not be met. Condition 2(b) prevents incorrect counting, since a processor that is made available by preempting a task T_p can only be used by one T_i.

So far we have assumed that any scenario of the SGC is bounded by the original SGC, i.e., on the SGC no task is ever moved outside of the slot that it was originally assigned to. In the presence of preemption of tasks T_p and the associated re-insertion into the SGC we need to check if it is possible that an original SGC slot may be exceeded, i.e., if upon re-insertion a task T_p is not contained in its previous SGC slot. This is only possible if the remaining task execution time $c_p^{max} - c_p'$ plus the preemption overhead $\mathcal{C}(T_p)$ is greater than c_p^{max}. However, condition 2(a) prevents this. □

5 Summary

This paper addressed dispatching in real-time systems in which response time of tasks is a QoS concern. It subjects a task system represented by a task graph to the list scheduling paradigm under the consideration of limited preemption. Preemption points are defined for each task based on non-preemption intervals. After each expiration of a non-preemption interval tasks can be preempted in order to improve response time to higher priority tasks. It has been shown that similar to non-preemptive list scheduling, the inclusion of limited preemption leaves the task system vulnerable to timing anomalies, i.e., under this scheduling paradigm, unavoidable variations in task durations at run-time can result in instability when one or more tasks execute for less than their maximum duration.

Safe dispatching conditions have been presented that avoid these anomalies. The conditions can be used in an algorithmic fashion to dispatch tasks safely, or they can be used by system designers as the basis for deriving more efficient dispatching algorithms. A simple counting argument is used that insures availability of processors to each task in a timely fashion, while improving QoS for high priority tasks and guaranteeing scheduling stability.

References

[1] Bruno, J., E. Gabber, B. Ozden, and A. Silberschatz. "Move-To-Rear List Scheduling: a new scheduling algorithm for providing QoS guarantees", *Proceedings of the 5th ACM International Multimedia Conference*, Seattle, Washington, November 9-13, 1997, pp. 63-73.

[2] Butler, R.W., and B.L. DiVito, "Formal Design and Verification of a Reliable Computing Platform for Real-Time Control", *NASA Tech. Mem. 104196*, Phase 2 Results, Jan 1992.

[3] Carpenter, K., et al., "ARINC 659 Scheduling: Problem Definition", *Proc. IEEE Real-Time Systems Symposium*, 1994, pp. 165-169.

[4] Deogun, J.S., R.M. Kieckhafer, and A.W. Krings, "Stability and Performance of List Scheduling With External Process Delays, " *Real-Time Systems*, Vol. 15, No. 1, July 1998, pp. 5-39.

[5] Hwu, Wen-mei W., and T.M Conte, "The Susceptibility of Programs to Context Switching", *IEEE Transactions on Computers.*, Vol. 43, No. 9, Sep. 1994, pp. 994-1003.

[6] Jeffay, K., et al, "On Non-Preemptive Scheduling of Periodic and Sporadic Tasks", *IEEE Real-Time Systems Symposium*, 1991, pp. 129-139.

[7] Kieckhafer, R.M., et al, "The MAFT Architecture for Distributed Fault-Tolerance", *IEEE Trans. Computers*, V. C-37, No. 4, April, 1988, pp. 398-405.

[8] Krings, A.W., "Inherently Stable Priority List Scheduling in an Extended Scheduling Environment", *PhD Thesis*, Dept. of Comp. Sci., Univ. of Nebraska, Lincoln, 1993.

[9] Krings, A.W., R. Kieckhafer, and J. Deogun, "Inherently Stable Real-Time Priority List Dispatchers", *IEEE Parallel & Distributed Technology*, Winter 1994, pp. 49-59.

[10] Krings, A.W., and M.H. Azadmanesh, "Resource Reclaiming in Hard Real-Time Systems with Static and Dynamic Workloads", *Proc. Hawaii International Conference on System Sciences, HICSS-30*, IEEE Computer Society Press, Vol. I, 1997, pp. 616-625.

[11] Krings, A.W., and M.H. Azadmanesh, "Feasibility Tests for Stable Dynamic Scheduling", *Proc.* 10^{th} *International Conference on Parallel and Distributed Computing and Systems*, October 28-31, Las Vegas, Nevada, 1998, pp. 25-31.

[12] Sung-Soo Lim, et al., "An Accurate Worst Case Timing Analysis Technique for RISC Processors", *Proc. IEEE* 15^{th} *Real-Time Systems Symposium*, 1994, pp. 97-108.

[13] Manacher, G.K., "Production and Stabilization of Real-Time Task Schedules," *JACM*, Vol. 14, No. 3, July 1967, pp. 439-465.

[14] McElvaney, M., et. al., "Guaranteeing Task Deadlines for Fault-Tolerant Workloads with Conditional Branches", *J. of Real-Time Systems*, (3), 3, Sep. 1991, pp. 275-305.

[15] Shaffer, P.L., "A Multiprocessor Implementation of Real-Time Control for a Turbojet Engine", *IEEE Control Systems Magazine*, Vol. 10, No. 4, June 1990, pp. 38-42.

[16] Shen, C., et al, "Resource Reclaiming in Real-Time", *Proc. Sixth Real-Time Systems Symposium*, December 1990, pp. 41-50.

[17] Stankovic, J.A., and Ramamritham, K., "The Design of the Spring Kernel", *IEEE Proc. of the Real-Time Systems Symposium*, Dec 1987, pp. 371-382.