FEASIBILITY TESTS FOR STABLE DYNAMIC SCHEDULING

A.W. KRINGS
Computer Science Dept.
University of Idaho
Moscow, ID 83844–1010, USA

M.H. AZADMANESH
Computer Science Dept.
University of Nebraska at Omaha
Omaha, NE 68182–0500, USA

Abstract – A dynamic scheduling approach is introduced that takes advantage of the simplicity of non-preemptive priority list scheduling, yet allows the inclusion of dynamically arriving tasks into the hard real-time task system. In list scheduling, so-called timing anomalies can cause deadlines to be missed if one or more tasks finish early. Apriori and run-time stabilization algorithms have been developed that can avoid these instabilities. However, these approaches assume static task priorities. Any arrival of tasks in addition to the static workload has been dealt with using slack-time reclaiming arguments. We introduce feasibility tests that allow tasks and subgraphs to be inserted into the priority list at run-time, modifying the priority list. The result is a scheduling environment where task priorities are no longer fixed, yet stability of the dispatcher is guaranteed.

Index Terms – real-time systems, non-preemptive scheduling, feasibility test, list scheduling.

1 INTRODUCTION

Multiprocessor systems are used in real-time applications in order to deal with increasing computational workloads or to achieve 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. Moreover, in safety critical applications missing deadlines may lead to catastrophe, i.e., loss of live, environmental damage or unacceptable cost. It is therefore of great importance to have a predictable and provably stable scheduling environment. Furthermore, due to the real-time requirements, only low-overhead approaches are of interest.

Scheduling disciplines can be partitioned into preemptive and non-preemptive approaches. 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 [1, 3, 4]. Non-preemptive scheduling is more restrictive but also more predictable. Static priority list scheduling is a simple approach in non-preemptive scheduling in which tasks are ordered according to their priorities in a list. The priorities are fixed, e.g. they are defined according to a feasible schedule at the time of system design. Then 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) [1], the Multicomputer Architecture for Fault Tolerance (MAFT) [5], the Spring Kernel [15], or specific applications like turbojet engine control [13].

Non-preemptive list scheduling poses a problem for critical real-time applications in that it is vulnerable to tasking anomalies [11]. An unavoidable anomaly, called timing anomalies, implies that the reduction in task execution times of one or more tasks can cause deadlines to be missed. Task durations may not generally be assumed constant, as they are directly affected by memory management, communication overhead, channel contention, as well as asynchronousy of autonomous input or sensor units [10]. Several approaches have been proposed to avoid timing anomalies. Whereas *apriori* stabilization is based on Manacher's algorithm [11], a family of *run-time stabilization* algorithms has been introduced [7, 8] that is less restrictive than previously known stabilization methods. One limitation of stable list scheduling described in all previous citations is that the workload is assumed to be static, i.e., the task system is defined apriori and task priorities are fixed. A recent approach considered the inclusion of dynamically arriving tasks in addition to a static workload based on slack-time reclaiming [9]. However, task priorities remained static. Slack-time compounds generally fast since actual task durations are often much smaller than the maximal durations, up to one order of magnitude [2].

Static priority list scheduling has the limiting characteristic that the complete workload must be known apriori. However, this limitation can be overcome if the scheduling environment allows for dynamically arriving tasks. Therefore, this paper considers scheduling and dispatching of workloads consisting of a hard real-time (core) workload, supplemented by dynamically arriving tasks. Of course, the set of core tasks or dynamically arriving tasks could be empty, i.e., there could be only dynamic tasks or all tasks are core tasks. We introduce scheduling feasibility checks that allow tasks and subgraphs to be included into the hard workload at run-time without destabilizing the workload. Whereas our approach also takes advantage of slack-time arguments, it extends the reclaiming philosophy by allowing tasks to be inserted into the task systems at *any* points in time and not only at the time of the scan. This is fundamentally different from previous approaches in that a task or subgraph passing the feasibility test will generally change priorities of the hard real-time workload, i.e., task priorities become dynamic. Tasks failing the feasibility check can not be inserted and can not have guaranteed deadlines. However, they can be appended to the workload for possible dispatching based on slack-time reclaiming [9]. Then, by definition, the rejected tasks are said to have *soft* deadlines. Section 2 will supply background information, definitions and an example of instability. In Section 3, four feasibility tests are presented. These tests are proven to be incapable of destabilizing the workload in Section 4. Finally Section 5 will conclude the paper with a summary.

2 BACKGROUND

2.1 Task System

The basic units of computation are tasks. Tasks consist of sequentially executing code. They may be run on any of M homogeneous processors, and may not be preempted. Each task T_i has an associated minimum and maximum computation time c_i^{max} and c_i^{min}, release time r_i at which the task becomes *ready* for execution, starting time s_i, finishing time f_i, and deadline d_i. Task dependencies are defined by a partial order, represented by a directed acyclic precedence graph.

Special tasks called *phantom tasks* have been used to model events external to a processor, such as delayed task release, non-transparent overhead or task synchronization [6, 7, 9]. These tasks are fully incorporated into the precedence graph. Although they consume time, unlike *real tasks*, i.e., tasks in the regular sense, they consume no resources. As a result, phantom tasks may *always* be started upon becoming released. Phantom tasks will be the mechanism for controlling the insertion of dynamic tasks into the workload.

The work presented here is based on the list scheduling paradigm. In *priority list scheduling*, whenever a processor becomes available, the run-time *dispatcher* scans the task list from left to right, and the first unexecuted ready task encountered in the scan is assigned to the processor. The dispatcher is distinct from the scheduling algorithm. The scheduler is executed only once at design time for the core workload and invoked at run-time for dynamically arriving tasks in form of a feasibility check. At run-time, the dispatcher simply arbitrates tasks.

2.2 Definitions

The following terms and definitions will be used throughout this paper:

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} [11].

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}. It should be noted that, unlike previous research, the SGC is not considered static anymore, since it will be modified by dynamic tasks which pass the feasibility test.

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 [11]. Furthermore, without loss of generality, tasks are assumed to be ordered by increasing[1] 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.

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},

[1] The projective list reflects the task starting sequence on the SGC at any given time. In general, the inclusion of a dynamic task into the workload will require renumbering all tasks following it in the list. Thus is is advantages to select data structures such that task priorities are relative rather than absolute, e.g. using pointers rather than indexed arrays.

the starting time on the SGC. Thus, if $s_i \leq s_i^{std}$, then $f_i \leq f_i^{std}$.

Fan-out: occurs when a phantom task releases a real task, or when a real task causes the release of a second real task executing in parallel with the first real task. The task initiating the release is called a *forking task.* The release is called a *logical fork*, since either case causes the occupancy of one more processors, i.e., from 0 to 1 and from 1 to 2 processors for phantom and real forks respectively. Fan-out, caused by logical forks, is a necessary condition for instability to occur.

Fan-out task: a real task with *at least* one of the following properties: (1) it is 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, (2) it is a descendent of an unfinished phantom task.

Several task sets will be used throughout the paper. 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. $\mathbf{T}_{\leq i}$ is defined as $\mathbf{T}_{<i} \cup \{T_i\}$. Sets $\mathbf{T}_{>i}$ and $\mathbf{T}_{\geq i}$ are symmetric to $\mathbf{T}_{<i}$ and $\mathbf{T}_{\leq i}$ respectively.

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., $s_v > s_v^{std}$, and $s_i \leq s_i^{std}\ \forall\ T_i \in \mathbf{T}_{<v}$. Task T_v is *vulnerable* to instability if there exists *any* scenario in which T_v is unstable.

2.3 Instability

An example of instability is shown in Figure 1. The precedence graph contains seven hard tasks, with maximum durations 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. Scheduling instability can be observed on the NGC, where 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.

To avoid instabilities two stabilization methods have been proposed that can be partitioned into *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 [5, 11, 12, 14]. 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 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 [7, 8].

3 FEASIBILITY CONDITIONS

Feasibility conditions indicate whether tasks can be inserted into the SGC at run-time without destabilizing the workload. We consider the following cases of dynamic task insertion: (1) independent tasks, (2) dependent tasks, i.e., tasks that have predecessors in the current SGC, (3) subgraphs with no external dependencies, (4) subgraphs with external dependencies. For all cases two considerations must be fulfilled. Firstly, there must be idle processing time available, and secondly this idle processing time must not be mortgaged to justify stability of the current scenario.

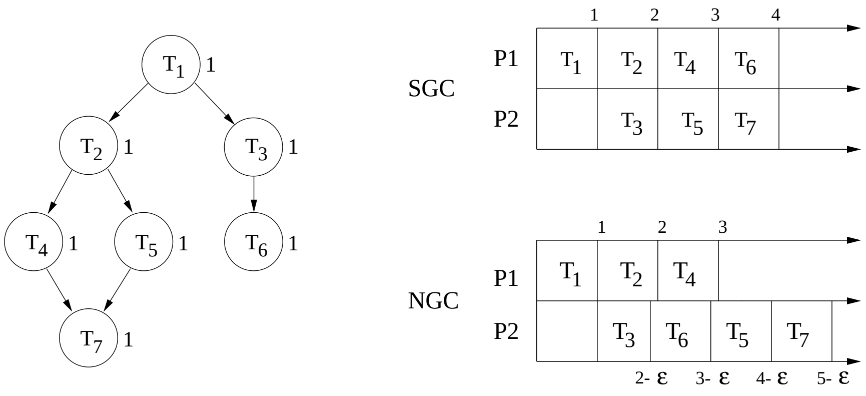
It should be noted that the feasibility conditions are not a mechanism to generate a schedule that is inherently invulnerable to instability, but they guarantee that the schedule does not become destabilized through the insertion of dynamic tasks. It is up to the run-time dispatcher to avoid instability. Such dispatcher can be found in [8, 9].

3.1 Feasibility Interval and Idle-Chains

Assume that all tasks that have been started are marked on the SGC. This includes tasks already finished. The marking can be visualized by shading the slots the tasks occupied on the SGC. If one also shades processor idle times, i.e., the slots that are not occupied by any tasks, then any shaded area on the SGC can be potentially used to justify the insertion of dynamic tasks. Continuous shaded slots will be referred to as *idle-slots*, denoted by Ψ.

A dynamically arriving task T_d is assumed to have a hard deadline d_d. However, d_d can be only guaranteed if T_i can be executed somewhere in the interval $[s_d^{min}, d_d]$, where s_d^{min} refers to the earliest possible starting time. This interval is called the *feasibility interval* Φ_d, i.e., $\Phi_d = [s_d^{min}, d_d]$. Φ_d implies that T_d must be started in the interval $[s_d^{min}, d_d - c_d^{max}]$. It should be noted that if no restrictions exist on s_d^{min}, starting time s_d^{min} can be set to the current time t_n. As will be shown, one of the conditions necessary for scheduling feasibility of T_d is that there exists an idle-slot on the SGC for the duration of c_d^{max} in the feasibility interval Φ_d.

The part of Φ_d of length c_d^{max} that is actually reserved to schedule T_d will be called the *selected fea-*



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

Figure 1: EXAMPLE OF INSTABILITY

sibility interval, denoted by $\hat{\Phi}_d$. Thus Φ_d indicates the feasible time interval for scheduling T_d and $\hat{\Phi}_d$ is the interval within Φ_d that is actually selected, given that $\hat{\Phi}_d$ is covered by an idle-slot Ψ, i.e., $\hat{\Phi}_d$ is in $\Phi_d \cap \Psi$.

During run-time, as tasks get started, idle-slots may be scattered over the SGC. It is possible to defragment idle-slots, i.e., combining smaller idle-slots to larger ones. Figure 2 shows the concept of idle-slot defragmentation. Here the idle times on processors P_i and P_j are manipulated by exchanging all tasks allocated on both processors following their respective idle-slots. This is possible since a homogeneous processor environment and dynamic task to processor allocation are assumed. Thus a larger idle-slot is generated on processor P_i in SGC'. As a consequence, in order to investigate if there exists an idle-slot for T_d in Φ_d it suffices to look at chains of idle-slots of length c_d^{max}. An *idle-chain* is a sequence of either overlapping or bordering idle-slots, regardless of the processor on which the idle-slot is located. Idle-chains can be transformed into idle-slots of equal length by applying the transformation of figure 2 repeatedly for each idle-slot of the chain. Figure 2 shows an idle-chain spanning from f_a^{std} to s_s^{std} in SGC and the defragmented idle-slot on P_i in SGC'. Since idle-slots and idle-chains are logically equivalent, i.e., every idle-chain can be transformed into an idle-slot and an idle-slot is by definition a trivial idle-chain, we will use both terms interchangingly and Ψ will denote both idle-slots and idle-chains. To distinguish between idle-slots, subscripts will be used. Thus Ψ_i will denote the i^{th} idle-slot.

3.2 Feasibility of independent tasks

We first discuss independent dynamic tasks, i.e., those that are not dependent on any task of the SGC. A task T_x can be admitted into the SGC if the following conditions are satisfied. Firstly, there exists $\hat{\Phi}_x$ in Φ_x, i.e., there is an idle-chain of length greater than or equal to c_x^{max} in the feasibility interval $\Phi_x = [s_x^{min}, d_x]$. Secondly, for each fan-out task T_w whose standard starting time is in the selected feasibility interval $\hat{\Phi}_x$, the number of processors assigned to unmarked tasks plus the number of processors occupied by currently executing tasks T_i with f_i^{max} in $\hat{\Phi}_x$ is less than or equal to $M-1$ at s_w^{std}. The use of only fan-out tasks T_w, rather than all tasks, in the second condition might not be intuitive, and will be justified in Section 4. It might help at this time to think of all tasks and not just fan-out tasks. Also, note that so far nothing is said about the relative position of $\hat{\Phi}_x$ within $\Phi_x \cap \Psi$. This issue will be addressed in Subsection 3.6.

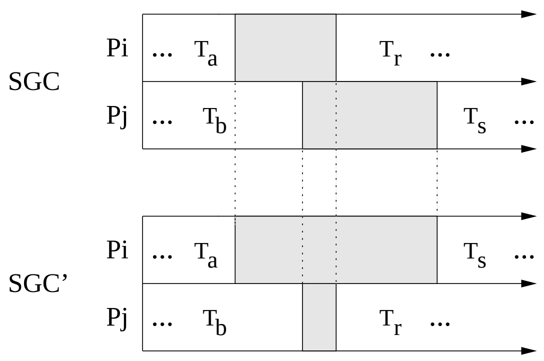


Figure 2: IDLE-SLOT DEFRAGMENTATION

Let $\mathcal{U}(s_w^{std})$ indicate the number of unmarked tasks in the SGC at s_w^{std}, and $\mathcal{E}(s_w^{std})$ the number of tasks T_i that are currently executing (on a processor) for which $f_i^{max} > s_w^{std}$.

FT-1: A task T_x can be admitted into the SGC if:

1. $\exists\hat{\Phi}_x$ such that d_x is satisfied, and
2. for every fan-out task T_w with s_w^{std} in $\hat{\Phi}_x$: $\mathcal{U}(s_w^{std}) + \mathcal{E}(s_w^{std}) \leq M - 1$.

3.3 Feasibility of dependent tasks

We now consider dynamic tasks that are dependent on one or more tasks, i.e., they have predecessors in the SGC. Let $min(\hat{\Phi})$ denote the starting time of the time interval $\hat{\Phi}$.

FT-2: A task T_x can be admitted into the SGC if

1. the following hold:
   (a) $\exists\hat{\Phi}_x$ such that d_x is satisfied, and
   (b) $f_i^{std} \leq min(\hat{\Phi}_x)$ $\forall$ unstarted predecessors T_i of T_x, and
   (c) $f_i^{max} \leq min(\hat{\Phi}_x)$ $\forall$ executing predecessors T_i of T_x
2. and, for every fan-out task T_w with s_w^{std} in $\hat{\Phi}_x$: $\mathcal{U}(s_w^{std}) + \mathcal{E}(s_w^{std}) \leq M - 1$.

3.4 Feasibility of subgraphs with internal dependencies only

Assume that a subgraph $\mathbf{G}_d = (\mathbf{T}_d, \prec_d)$ is to be scheduled, where $\mathbf{T}_d$ is the set of dynamic tasks and $\prec_d$ the partial order defined on the tasks in $\mathbf{T}_d$. Note that there are no dependencies among tasks in $\mathbf{G}_d$ and tasks of the SGC. Therefore $\mathbf{G}_d$ is called a subgraph with *internal dependencies* only. Furthermore, we assume that $\prec_d$ defines a *strong* precedence, i.e., for each task $T_i \in \mathbf{T}_d$ starting time s_i is explicitly defined. The motivation for using strong precedence is that we want to focus on stability issues rather than on how to optimize the insertion into the schedule, i.e., strong precedence simplifies the feasibility tests. However, if $\mathbf{G}_d$ passes the feasibility test, the strong precedence can be relaxed to normal precedence. With respect to practical considerations, the strong precedence of $\mathbf{G}_d$ would be derived from a tentative schedule of $\mathbf{G}_d$. The choice for selected feasibility intervals $\hat{\Phi}$ are fixed in Φ. If the maximal number of tasks of $\mathbf{G}_d$ executing in parallel is a, then the tasks in $\mathbf{T}_d$ will require a maximum of a parallel idle-chains. Again, note that this is a result of the strong precedence assumption.

Let $\mathcal{O}(s_w^{std})$ be the number of selected feasibility intervals $\hat{\Phi}_i$ overlapping at time s_w^{std}.

FT-3: A subgraph $\mathbf{G}_d = (\mathbf{T}_d, \prec_d)$ with internal dependencies only can be admitted into the SGC if:

1. $\forall T_x \in \mathbf{T}_d$, $\exists\hat{\Phi}_x$ such that d_x and $\prec_d$ are satisfied, and
2. for every fan-out task T_w with s_w^{std} in $\hat{\Phi}_x$: $\mathcal{U}(s_w^{std}) + \mathcal{E}(s_w^{std}) \leq M - \mathcal{O}(s_w^{std})$.

Thus, for each fan-out task T_w whose standard starting time overlaps with any $\hat{\Phi}_i$, at s_w^{std}, the number of processors assigned to unmarked tasks plus the number of processors occupied by currently executing tasks T_i with f_i^{max} beyond s_w^{std} is less than or equal to M minus the number of selected feasibility intervals $\hat{\Phi}_i$ overlapping at s_w^{std}.

3.5 Feasibility of subgraphs with external dependencies

This is the most restrictive feasibility test and differs from the previous test only in that now precedence constraints between tasks in $\mathbf{G}_d$ and unfinished tasks, i.e., currently executing or unstarted tasks, need to be considered.

FT-4: A subgraph $\mathbf{G}_d(\mathbf{T}_d, \prec_d)$ with internal and external dependencies can be admitted into the SGC if

1. the following hold:
   (a) $\forall T_x \in \mathbf{T}_d$, $\exists\hat{\Phi}_x$ such that d_x and $\prec_d$ are satisfied, and
   (b) $f_i^{std} \leq min(\hat{\Phi}_x)$ $\forall$ unstarted predecessors T_i of T_x, and
   (c) $f_i^{max} \leq min(\hat{\Phi}_x)$ $\forall$ executing predecessors T_i of T_x
2. and, for every fan-out task T_w with s_w^{std} in $\hat{\Phi}_x$: $\mathcal{U}(s_w^{std}) + \mathcal{E}(s_w^{std}) \leq M - \mathcal{O}(s_w^{std})$.

3.6 Intentional Idle Times

One inherent property of list scheduling is that there cannot be any intentional idle times. With respect to a dynamically scheduled task T_d admitted with FT-2, this presents no problem if $\hat{\Phi}_d$ is selected from the left of Ψ, i.e., T_d is to be inserted immediately following its predecessor or another task on the SGC. However, in the general case intentional idle times must be avoided if one wants to apply the theory of timing anomalies. Phantom tasks provide a mechanism to deal with "intentional idle times" by modeling them as delayed releases. Thus if T_d is to be admitted into the SGC at $\hat{\Phi}$ and intentional idle times would arise, a phantom task T_{pd} called *delay enforcement phantom* can be inserted into the precedence graph as a predecessor of T_d. The subscript pd in T_{pd} indicates that it is a phantom task dedicated to T_d. The starting time of T_{pd} is set to $s_{pd}^{std} = t_n$, the current time, and its duration to $c_{pd}^{std} = s_d^{std}$, the beginning of $\hat{\Phi}$. During run-time, delaying a task artificially has of course no benefit. Therefore, if the dispatcher scans up to task T_i that has T_{pi} as predecessor, T_{pi} can be prematurely finished in order to start T_i.

4 PROOF OF CORRECTNESS

The first step in proving that the insertion of dynamic tasks that pass feasibility tests FT-1 through FT-4 cannot destabilize the schedule is in narrowing down the interval that must be considered for investigation. The following lemma, restated from [9], shows that only tasks whose standard starting times overlap with the execution of usurper task T_x on the NGC need to be considered as potentially vulnerable tasks in order to guarantee stability.

Lemma 1 *Assume that a usurper task T_x has started at time s_x, and define $f_x^{max} = s_x + c_x^{std}$. No task T_v with $s_v^{std} > f_x^{max}$ can become unstable as a result of starting T_x.*

Proof: See [9, Lemma 3]. □

With respect to the feasibility tests, this means that we only have to consider the selected feasibility intervals $\hat{\Phi}$ in our investigations. In order to prove the feasibility tests we will focus on FT-2, and FT-4, i.e., the other tests represent special cases.

Theorem 1 *Tasks admitted by feasibility test FT-2 cannot destabilize the workload.*

Proof: FT-2 condition $1a$ guarantees that there is a feasible idle-slot available for T_x. This must be, since by the definition of $\hat{\Phi}_x$ an idle-slot Ψ_x must overlap with Φ_x for the whole interval $\hat{\Phi}_x$. Conditions $1b$ and $1c$ are needed to enforce the precedence constraints. Therefore, assume that conditions 1 hold.

In order to prove the necessity of FT-2 condition 2, we first show that only tasks in $\hat{\Phi}_x$ need to be considered. For a given real task T_v, three General Instability Conditions (GIC1 - GIC3) have been derived that are both necessary and sufficient for T_v to be unstable [6]. GIC1 indicates that priority inversion is a necessary condition for instability to occur (see also [11]). However, for T_v with $s_v^{std} < s_x^{std}$ the inclusion of T_x does not constitute a priority inversion[2] and invulnerability of T_v follows from GIC1.

For tasks with starting times after the interval $\hat{\Phi}_x$, i.e., for T_v with $s_v^{std} > s_x^{std} + c_x^{max}$, the inclusion of T_x does constitute a priority inversion. However, the effect of a usurper task is limited to those tasks overlapping on the SGC with the execution of the usurper T_x. Invulnerability of tasks T_v with standard starting times beyond $\hat{\Phi}_x$ follows from Lemma 1.

It remains to be shown that tasks T_v with starting times in $\hat{\Phi}_x$ are invulnerable if FT-2 condition 2 holds. Consider all tasks T_v with standard starting times s_v^{std} in $\hat{\Phi}_x$. Now assume that condition 2 holds. Define f_ϕ as the time of the last fork into $\mathbf{T}_{\leq v}$ to occur at or before ready time r_v. General Instability Condition GIC3 states that for T_v to be unstable, there must exist *no* time in the interval $[f_\phi, s_v^{std}]$ at which all ready real tasks in $\mathbf{T}_{\leq v}$ are running. Now consider individual forking tasks. Finishing of a forking task T_ϕ causes the fan-out for the respective fan-out tasks T_{wi}. However, since $\mathcal{U}(s_w^{std}) + \mathcal{E}(s_w^{std}) \leq M - 1$, at each s_{wi}^{std}, there is a processor available for each unstarted task, independent of currently executing tasks. Thus at time s_w^{std} latest, a processor is available for each task in $\mathbf{T}_{\leq w}$ and GIC3 cannot hold, which implies invulnerability due to condition 2. Tasks $T_{v'}$ with $s_{v'}^{std} \neq s_{wi}^{std}$, for any T_{wi} with s_{wi}^{std} in $\hat{\Phi}_x$, need not be considered, since no new fan-out is introduced, and tasks in $\mathbf{T}_{<v'}$ are safe by assumption. □

Theorem 2 *A subgraph $\mathbf{G}_d = (\mathbf{T}_d, \prec_d)$ admitted by feasibility test FT-4 cannot destabilize the workload.*

Proof: FT-4 condition $1a$ guarantees that there is an idle-slot available for every $T_x \in \mathbf{T}_d$. Conditions $1b$ and $1c$ are needed to guarantee the precedence constraints of T_x. Therefore, assume that conditions $1a$, $1b$ and $1c$ are satisfied.

To differentiate between different usurper tasks T_x let T_{xl} denote the l^{th} task in $\mathbf{T}_d$. Now consider any selected feasibility interval $\hat{\Phi}_{xl}$ and let $\mathbf{T}^{\hat{\Phi}_{xl}}$ be the set of tasks with standard starting times in $\hat{\Phi}_{xl}$. Again, as in the proof of Theorem 1, the argument of General Instability Condition GIC3 is used, which implies that instability is *only* possible if processor contention occurs. However, from FT-4 condition 2 we have $\mathcal{U}(s_w^{std}) + \mathcal{E}(s_w^{std}) + \mathcal{O}(s_w^{std}) \leq M$, and for each fan-out task $T_{wi} \in \mathbf{T}^{\hat{\Phi}_{xl}}$, at s_{wi}^{std} there is a processor available for each unstarted task. This is independent of currently executing tasks and $\mathcal{O}(s_w^{std})$ processors claimed for selected feasibility intervals. Thus at time s_w^{std} latest, a processor is available for each task in $\mathbf{T}_{\leq w}$ and GIC3 cannot hold.

Analogously to the proof of Theorem 1, tasks outside $\mathbf{T}^{\hat{\Phi}_{xl}}$ need not be considered since either they do not experience the necessary priority inversion by GIC1, or cannot become instable by Lemma 1. □

[2] It should be noted that, upon arrival, every dynamic task has lower priority than any tasks in the SGC.

The proof for FT-1 and FT-3 follow directly from Theorem 1 and Theorem 2 respectively, since FT-1 and FT-3 represent special cases of FT-2 and FT-4, where no external precedence constraints exist.

5 CONCLUSIONS

This paper addressed the scheduling of dynamically arriving tasks and subgraphs in the context of non-preemptive priority list scheduling. 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 durations. Therefore either the task system or the dispatcher must enforce that instability cannot arise. In the presence of dynamic tasks, the use of inherently stable run-time dispatchers is required. Whereas it is up to the run-time dispatcher to enforce stability of the workload, dynamic tasks can only be permitted into the schedule if they don't destabilize the workload. For this purpose four feasibility tests were introduced. They allowed independent and dependent tasks, as well as subgraphs with internal dependencies and subgraphs with internal and external dependencies to be included into the schedule. The feasibility tests take into consideration the slots in the standard Gantt chart left vacant when tasks are started, as well as processor idle times. Thus they take advantage of accumulative available processing time due to early finishing tasks. Dynamic tasks are scheduled into the workload if there exist idle-slots in the standard Gantt chart, and the idle-slots were not mortgaged by the run-time dispatcher to justify the starting of lower priority tasks in order to improve processor utilization, prior to the arrival of the dynamic task.

The feasibility tests bridge the gap of inflexibility normally associated with priority list scheduling. The tests open up the use of this simple scheduling paradigm for applications where the whole schedule might not be known up front, or where the workload is run-time dependent.

References

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

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

[3] Hwu, Wen-mei W., and T.M Conte, The Susceptibility of Programs to Context Switching, *IEEE Transactions on Computers.*, 43(9), Sep. 1994, 994-1003.

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

[5] Kieckhafer, R.M., et al, The MAFT Architecture for Distributed Fault-Tolerance, *IEEE Trans. Computers*, C-37(4), April, 1988, 398-405.

[6] Kieckhafer, R.M., and J.S. Deogun, On the Stability of List Scheduling in Real-Time Multiprocessor Systems, *Report #99*, UNL, Dept. of Comp. Sci., Feb 1990.

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

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

[9] Krings, A.W., and M.H. Azadmanesh, Resource Reclaiming in Hard Real-Time Systems with Static and Dynamic Workloads, *Proc. HICSS-30*, Vol. I, 1997, 616-625.

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

[11] Manacher, G.K., Production and Stabilization of Real-Time Task Schedules, *JACM*, 14(3), July 1967, 439-465.

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

[13] Shaffer, P.L., A Multiprocessor Implementation of Real-Time Control for a Turbojet Engine, *IEEE Control Systems Magazine*, 10(4), June 1990, 38-42.

[14] Shen, C., et al, Resource Reclaiming in Real-Time, *Proc. Sixth Real-Time Systems Symposium*, Dec. 1990, 41-50.

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