International Journal of Foundations of Computer Science

REAL-TIME DISPATCHING: SCHEDULING STABILITY AND PRECEDENCE

AXEL W. KRINGS
Department of Computer Science, University of Idaho
Moscow, Idaho 83844-1010, USA

and

MOSHE DROR
MIS Department, University of Arizona
Tucson, Arizona 85721, USA

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

This paper introduces a new graph theoretical concept called strong precedence which is used to address the problem of scheduling instability in non-preemptive static list scheduling. Scheduling instability occurs when a reduction in task duration of one or more tasks causes other tasks to miss their deadline. This problem has been addressed in the past by introducing additional precedence constraints into the precedence graph representing the workload, or by limiting the depth the dispatchers scan at run-time. We present an alternative stabilization approach based on the concept of strong and weak precedence. By defining a strong precedence relation on selected subgraphs, the workload becomes inherently stable without requiring the introduction of new edges into the graph.

Keywords: Strong precedence, multi-processor scheduling, dispatching, timing anomalies.

1. Introduction

In real-time multiprocessor systems, non-preemptive static priority list scheduling represents a low-overhead scheduling approach for precedence-constrained tasks [1, 10, 15, 17]. However, it is well documented that such scheduling is vulnerable to anomalous timing behavior due to run-time variations in task durations. Specifically, in priority list scheduling, reduction in the duration of one or more tasks at run-time might delay other tasks starting times. This *Scheduling Instability* phenomenon makes it difficult to guarantee real-time task deadlines. In this context, the causes of instability and performance of list scheduling have been carefully examined in the work by Deogun, Kieckhafer, and Krings [3]. In their work, they derive a set of *General Instability Conditions* which are both necessary and sufficient for instability to occur. In addition, they compare the average performance of a number of inherently stable run-time dispatchers with widely varying levels of complexity.

The literature on scheduling with focus on the instability phenomenon suggests a variety of heuristic techniques which modify the list's priority scheduling rules, task precedence structure, and the system environment to facilitate stability and assure completion deadlines. The solutions to scheduling instability range from trivial limiting rules on the scheduling options for run-time dispatchers through non-trivial transformations of the precedence graph. In this work we examine the notion of task precedence and its relation to scheduling stability. We study the past instability solutions as they relate to a new and more general notion of precedence distinction called *Strong Precedence* and *Weak Precedence*.

This paper presents solutions to the instability problem based on the notion of strong and weak precedence. Section 2 introduces the instability problem and examines existing stabilization methods. In Section 3 strong and weak precedence are defined. Next, the impact of fan-out on instability is discussed in Section 4 as a basis to finding a solution. A stabilization method based on strong precedence subgraphs, an example, as well as the proof of stability is given in Section 5. Finally, Section 6 concludes the paper with a summary.

2. Stability and List Scheduling

Scheduling Stability in task processing systems is defined as the absence of instability; specifically, a task schedule is stable if the scheduling procedure guarantees that no task will be scheduled so its completion time (for all possible run-time task durations) might exceed its deadline. *Schedule Stabilization* is the procedure of enforcing stability of a scheduling system.

As stated above, non-preemptive priority list scheduling is a simple low-overhead approach to scheduling real-time tasks in multiprocessor systems. It was noted long ago [9] that non-preemptive priority list scheduling is vulnerable to a number of anomalies which describe counter-intuitive scheduling behavior. For instance, task completion can be delayed when: (1) reducing the duration of one or more tasks, (2) increasing the number of processors in the system, or (3) relaxing the precedence of the task graph representing the workload.

The main focus of the work in [3, 11, 12] has been on addressing the potential scheduling anomaly due to run-time reduction in task durations. The motivation was that this so-called timing anomaly is unavoidable and is likely to be of concern in real-time and fault-tolerant multicomputer applications. The other anomalies can be prevented, e.g., by respectively fixing the number of processors and the precedence graph at design time. Task duration can vary from one execution of the task to the next. These variations are event driven, stochastic, and generally unavoidable. Therefore, it is necessary to secure a priori that a schedule will remain stable under all potential variations in task durations. In addition, one might want to add stability assurance for the case where the number of processors in the system can change when adding processing flexibility in a multicomputer architecture. We will return to this stability issue at a later point in this work.

Throughout this paper, it is assumed that the maximum duration of each task is known a priori. In addition, the minimum duration of each task may also be known and we do not exclude the situation that the minimum duration be zero for some tasks. Thus, either a task may sometimes be intentionally omitted based on run-time conditions, or a particular realization of task duration might be considered zero. The actual duration of a particular execution of the task may vary randomly between a minimum and maximum value.

The inclusion of zero duration tasks requires some explanation, since such tasks are not covered by the work in [3, 11, 12]. However, in order to be able to use the theory derived it is important to justify that this modification has no impact, since zero duration tasks can be modeled with tasks of duration ϵ. The problem of zero task durations can be seen from a dispatcher's point of view. There, a task with zero duration is still maintained in the data structure of the run-time system. Thus, it requires a finite amount of time to update the data structure at run-time. However, run-time system overhead is modeled as part of the task durations, which effectively changes zero duration tasks into tasks with finite duration ϵ. Another way of justifying zero duration tasks is modeling them as tasks with finite durations ϵ, where ϵ is chosen much smaller than the clock rate of the system. Then due to discretization, the task duration is effectively zero.

2.1. Definitions

Below we essentially restate the basic definitions from [3], extending them somewhat for our broader context.

A *Scenario* describes the schedule obtained with a particular set of task durations for a given set of processors. The *Standard Scenario* is the scenario in which, for a given set of processors, all tasks execute for their specified maximum durations. A Gantt chart depicting the standard scenario is called the *Standard Gantt Chart* (SGC). In a *Non-Standard Scenario* at least one task runs for less time than its maximum duration, and/or one or more additional processors have been added to the system. The resulting Gantt chart is called a *Non-Standard Gantt Chart* (NGC).

Whenever a processor becomes idle or becomes available, the run-time dispatcher selects the next task to be executed on that processor. The dispatcher is thus distinct from the scheduling algorithm, which is executed at the system design time to assign task priorities and find a feasible schedule for the standard scenario.

A system is *stable* if there is no scenario under which the completion time of any task on any NGC exceeds its completion time on the SGC. If the SGC is feasible, then stability implies that all NGC's are feasible. Conversely, if the system is not stable, then one or more infeasible NGC's may result. Thus, to guarantee all deadlines in all possible scenarios, it is only necessary to: (1) find one feasible schedule for the standard scenario, (2) prove that the system is stable.

2.2. Existing Stabilization Methods

2.2.1. Fixed Starting Times

If each task is constrained to start at a fixed time, then priority inversion can not occur and stability is assured [18]. This type of scheduling stability assurance can be very expensive in terms of poor processor utilization. The effect can be significant, since in real-time applications the actual task durations are often much smaller than the maximum durations, e.g., up to an order of magnitude [2].

2.2.2. Fixed Starting Sequence

Rather than fixing task starting times, this approach constraints the starting sequence on an NGC to be the same as on the SGC. The method is implemented by prohibiting the dispatcher from scanning beyond the first unstarted task on the priority list [12, 16].

With this approach, the highest priority unstarted task will start as soon as all of its parents are finished. If the actual task durations are less than the maximum durations, then the schedule will tend to contract towards the left hand side of the Gantt chart. It allows more flexibility than the Fixed Starting Times approach, but still prevents priority inversion which might improve processor utilization without causing instability.

2.2.3. Manacher's Algorithm

Manacher's stabilization algorithm [14] implements a priori stabilization. It was developed to permit greater flexibility than the Fixed Starting Times and Fixed Starting Sequence algorithms and alters the original precedence graph to produce a graph which is inherently stable. The algorithm identifies all *co-running* task pairs on the SGC and their *nearest common ancestors*, if any. It then inserts new edges (precedence) into the precedence graph between the nearest common ancestors and any tasks beyond the co-running pair which could act as usurpers. Manacher's stabilization algorithm permits tasks to execute as soon as they are ready. Since it allows some priority inversions to occur, it offers greater flexibility than the two approaches above.

In [3], the Scheduling Stability issue has been cast in a general, application-independent setting. Their objective was to develop a general *Extended Scheduling Model* to derive a complete formalization of the instability problem. General instability conditions were presented that are both necessary and sufficient to induce instability and to assess the relative performance of stabilization methods.

This study takes as its point of departure the work presented in [3] and examines the scheduling stability implications of the recently developed separate concepts of *strong* and *weak* precedence relation for scheduling in chains and trees. We extend the notion of strong and weak precedence relation to more general graphs (not just chains and trees) with special focus on strong and weak precedence subgraph structures to assure scheduling stability.

3. Strong and Weak Precedence Relations

The objective of this paper is to examine *strong* and *weak* precedence implications for stability. Before stating what "strong" and "weak" precedence relations are some definitions are necessary.

A precedence relation $\prec$ defined on a set of tasks $\mathbf{T} = \{T_1, \cdots, T_n\}$ to be examined for a processing order and timing on a set of processors $\mathbf{M} = \{M_1, \cdots, M_m\}$, corresponds to an irreflexive partial order (a binary relation) defined on $\mathbf{T}$. $T_i \prec T_j$ implies that (i) task T_i has to be completed before the processing of task T_j can begin on any processor, or (ii) the processing of task T_i on any given processor has to be completed before the processing of task T_j can begin on the same processor. In this paper we examine only case (i) and restrict the notation of $\prec$ to immediate precedence and not one obtained by applying the transitive property of $\prec$. If we represent each element (task) in $\mathbf{T}$ by a node, then $\prec$ can be represented by directed edges between tasks with precedence and we obtain a corresponding directed graph.

We start first with the "oldest" instance of defining strong and weak precedence as it was presented for the case of chained tasks [8]. Consider K chains $C = \{C_k\}_{k=1}^{K}$, where $C_k = \{T_{k,i}\}_{i=1}^{m_k}$, and the tasks in each C_k are linearly ordered by precedence, i.e., $T_{k,1} \prec T_{k,2} \prec \cdots \prec T_{k,m_k}$, with $\sum_{k=1}^{K} m_k = n$. The chains are to be scheduled on m parallel processors $M_1, \ldots, M_m$ to optimize a given objective function. Each processor can handle at most one task at a time. We focus on two modes the task chains can be scheduled with: The first is called *strong chain* mode, because for any two tasks T_a and T_b, if T_a immediately precedes T_b in a chain, then no other task is scheduled between the completion time of T_a and the start time of T_b. The second called *weak chain* mode is one in which the above scheduling condition is allowed to be violated as long as the schedule still conforms to the $\prec$ relation. The weak chain precedence corresponds to the traditional way of addressing precedence relations.

We note that this distinction of chained tasks into "strong" and "weak" chains has algorithmic and complexity implications and occurs in a number of real-life settings [4, 5, 6]). Still, from the Scheduling Stability perspective, the chained task relation (precedence) does not produce timing anomalies in scheduling. Chain precedence does not allow for priority inversion, a necessary condition for instability, as will be shown in subsection 4.4.

Expanding the precedence relations in scheduling from chains into tree (in-tree and out-tree) precedence is not a straight-forward extension from chain precedence. In fact, the discussion is restricted to "proper" trees, i.e., there are nodes in the tree with in-degree (out-degree) equal to or greater than two.

A graph is a *tree* if it is connected and the number of edges is $n - 1$, where n is the number of nodes. An *in-tree* is a tree such that there is at most one edge emanating from each node and an *out-tree* is a tree with at most one edge entering a node.

Definition (of a *strong in-tree*): Given the task set $\mathbf{T}$ and in-tree precedence relation $\prec$ defined on $\mathbf{T}$, we say that $\prec$ is a *strong in-tree* precedence if, for any given task T_j, the processing of the tasks in the subset $\mathbf{T}(j) = \{T_i | T_i \in \mathbf{T}, T_i \prec T_j\}$

has to be scheduled successively in some order either before or after the start of any task $T_k \notin T(j)$, and has to be completed before the start of the task T_j. The emphasis here is on the successive processing of all the tasks with $\prec$ relation to T_j. This implies that any task T_k with no relation $\prec$ to T_j can not be processed some time in between the processing assignment of the tasks in $\mathbf{T}(j)$. Otherwise, the in-tree $\prec$ defines a *weak in-tree.*

The *strong out-tree* together with the *weak out-tree* are defined similarly for the subset of tasks in $\mathbf{T}$ which task T_j precedes, i.e., their processing is scheduled successively in the case of a strong out-tree, or could be interrupted by other unrelated tasks in the case of a weak out-tree.

3.1. Strong and Weak Precedence Subgraph

Now we introduce for the first time the notion of a *strong precedence subgraph* and *weak precedence subgraph.* The set of tasks is represented by the nodes of a precedence graph G.

Definition: Given a general precedence graph of tasks $G = (V, E)$, where V is the set of tasks and E the set of edges, and subgraphs $G^i = (V^i, E^i)$, $i = 1, \ldots, K$, where $V^i \subseteq V$ and $E^i \subseteq E$, we say that the precedence subgraph $G^i = (V^i, E^i)$ represents a *strong precedence subgraph* if all the tasks in G^i have to be scheduled in succession (respecting the precedence of the subgraph) before any task not in G^i can be scheduled. In addition, the schedule has to be consistent with respect to the precedence in G. Thus no two strong precedence subgraphs $G^i, G^j, j \neq i$ with edges $(a, b), (c, d) \in E \setminus \{E^i \cup E^j\}$, such that $a, d \in V^i, b, c \in V^j$, are allowed. Lastly, the definition of strong precedence is recursive, i.e., strong precedence subgraphs G^i may in turn contain strong precedence subgraphs G^j.

A graph (or a subgraph) without strong precedence subgraphs is defined as a *weak precedence graph (subgraph).*

In general, a precedence graph G might have possibly nested strong precedence subgraphs and regular (weak) precedence subgraphs.

4. Fan-out and Instability

4.1. Phantom Tasks

Sometimes the need arises to model events external to the system, non-transparent overhead, communication delay or delayed task release. The Extended Scheduling Model in [3] introduces a mechanism called *Phantom Task* for such a purpose. There is a processing time associated with a Phantom Task, just as is the case with a regular task, but unlike a regular task a phantom task does not occupy any system resources such as a processor. This is equivalent to assuming that for phantom tasks we have an infinite number of imaginary dedicated processors, i.e., *phantom processors.* Phantom tasks are fully integrated into the precedence graph together with real tasks and similarly their durations are stochastic within pre-defined bounds. However, phantom tasks do not appear on the SGC, or for that matter on any

NGC, because they do not occupy any real processors.

For a task T_i we let c_i denote the computation time, i.e., the task duration, r_i the release time, i.e., the time at which the last parent of T_i finishes executing, s_i the starting time, and lastly f_i the finishing time. When these times are with reference to the SGC, superscript *std* is used, e.g., r_i^{std}, s_i^{std} or f_i^{std}. Similarly superscripts *min* and *max* denote minimal and maximal values. By definition $c_i^{std} = c_i^{max}$ and c_i is in the interval $[c_i^{min}, c_i^{std}]$.

We assume the dynamic task-to-processor allocation model. The dispatcher is thought of as a non-sharable procedure guarded by a mutual exclusion protocol. The result of this protocol is a complete ordering on events which appear to be simultaneous on the Gantt chart. Thus, without loss of generality (see [12]), it is assumed that the priority list is a projective list.

4.2. Logical Fork

At the heart of the instability issue are forks. Consider a task set $\mathbf{T}_k$. The status of the task set $\mathbf{T}_k$ is reflected by the number of real tasks from $\mathbf{T}_k$ which are currently running or ready to run. Note that, in contrast to [3], we do not fix the number of processors in the system a priori.

Let $M_k(t)$ be the number of real tasks in $\mathbf{T}_k$ running at time t. Thus $M_k(t)$ is also the number of processors occupied by $\mathbf{T}_k$ at time t. Denote by $W_k(t)$ the number of real tasks in $\mathbf{T}_k$ which are ready but waiting for a processor at time t. Thus, $R_k(t) = M_k(t) + W_k(t)$ is the total number of real tasks in $\mathbf{T}_k$ which are either ready or running at time t.

Let $\mathcal{T} = \{T_1, \ldots, T_N\}$ be the set of all tasks. A *logical fork* is any event which triggers an increase in $R_{\mathcal{T}}(t)$ at time t, i.e., the values $M_k(t), W_k(t)$, and $R_k(t)$ reflect the changed task conditions immediately after the fork.

4.3. Unstable Tasks

Given a non-standard scenario with a scheduling anomaly, several tasks may start late relative to their starting time in the SGC. However, the lateness of one task may be the result of a chain reaction caused by the lateness of some lower numbered task. An *unstable task* is defined as the task which starts this lateness chain reaction. Then, in a given scenario, task T_v is unstable if it is the lowest numbered task to start late. Task T_v is *vulnerable* to instability if there exists at least one scenario in which T_v is unstable.

Following [14], given an initial fixed number of processors, let $\mathbf{T}_{<i}$ denote the set of all tasks, real or phantom, which started before task T_i on the SGC. Given a priority list for this SGC, $\mathbf{T}_{<i}$ contains all tasks T_j with $j < i$. Let $\mathbf{T}_{\leq i} = \{\mathbf{T}_{<i} \cup \{T_i\}\}$. Sets $\mathbf{T}_{>i}$ and $\mathbf{T}_{\geq i}$ are defined similarly.

Manacher has shown that a task T_v can be unstable only if a priority inversion occurs about T_v. Thus, there exists a task $T_x \in \mathbf{T}_{>v}$ such that $s_x < r_v$. Instability occurs because T_x "usurped" a processor which T_v requires to start on time. Task T_x is thus called a *Usurper Task*. Phantom tasks, by their very definition, cannot be usurper tasks.

4.4. Instability Conditions

As indicated in [3], according to the list scheduling paradigm, the start time for task T_v is the earliest time t for which all of the following criteria are satisfied:

S1 All parents of T_v have finished so that T_v is ready,
S2 There are no lower-numbered ready tasks on the list at t (i.e., $W_{<v}(t) = 0$),
S3 If T_v is real, then a processor is available (idle) at t (i.e., $M_{\mathcal{T}}(t) < M$).

For an unstable task T_v one or more of the above conditions became true only after s_v^{std} under the run-time conditions. It is interesting to note that a phantom task can never be unstable. Furthermore, following Deogun et al. [3] a vulnerable task could be late even if $[S1]$ and $[S2]$ hold at s_v^{std}. Thus, determining the stability of task T_v consists of stating the necessary and sufficient conditions under which $[S3]$ is not true for T_v at time s_v^{std}.

It has been established in [14] and again in [3] that priority inversion of a real task is a necessary condition for instability to occur. It also has been shown that priority inversion can occur only after a real or logical fork in the precedence graph.

Priority inversion is possible if one or more tasks in $\mathbf{T}_{<v}$ finish before their standard finishing time. Consequently, $s_x < r_v$ for one or more ready tasks $T_x \in \mathbf{T}_{>v}$ if for such T_x we have $W_{\leq v}(s_x) = 0$ and $M_{\leq v}(s_x) < M$. Again borrowing the notation from [3] we let $T_\triangle$ denote the latest starting usurper task with respect to unstable task T_v. Thus, the starting time of $T_\triangle$, i.e., $s_\triangle$, satisfies $s_\triangle \geq s_x$, $\forall T_x \in \mathbf{T}_{>v}$ for which $s_x < r_v$. Let $\mathbf{L}$ be the set of all tasks in $\mathbf{T}_{\leq v}$ which have not been completed by s_x, i.e., $\mathbf{L} = \{T_l \in \mathbf{T}_{\leq v} : f_l > s_\triangle\}$. The set $\mathbf{L}$ is called the *Leftover set* and T_v is the highest numbered task in $\mathbf{L}$. Thus, T_v can be unstable only if a priority inversion and a logical fork into $\mathbf{L}$ have occurred with $s_\triangle < r_v$. It is further proven in [3, Theorem 3] that instability *will* occur if there still are real tasks in $\mathbf{L}$ of a lower number than v which need to be scheduled in the entire interval from the last fork occurrence in $\mathbf{L}$ until standard time s_v^{std}. This leads to instability if and only if three *General Instability Conditions* are true:

I1 There exists at least one real task $T_x \in \mathbf{T}_{>v}$ such that $s_x < s_v$,
I2 There must be a logical fork into $\mathbf{L}$ such that for all tasks in $\mathbf{X} = \{T_x \in \mathbf{T}_{>v} : s_x < f_\xi < f_x\}$, where f_ξ is the time the last logical fork into $\mathbf{L}$ to occur at or before r_v,
I3 $W_L(t) > 0$, $\forall t \in [f_\xi, s_v^{std}]$.

4.5. Fan-out and Immediate-fan-out

The result of a logical fork is a *fan-out.* A 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.

A *fan-out task* was defined in [11] as (1) a real child of an unfinished phantom task, or (2) a descendent from an unfinished forking task that started executing on the SGC while another descendent of the same forking task was executing on another processor. We are interested in the *first* fan-out task of each forking task. Thus we define an *immediate-fan-out task* as a real task with *at least* one of the following properties:

(i) it is the *first* child of a phantom task,
(ii) it is a descendent from a forking task, and it is the *first* task that started executing on the SGC while another descendent of the same forking task was executing on another processor.

Thus there exists at most one immediate-fan-out task for a given forking task.

To demonstrate the concept of fan-out tasks consider Figure 1 which depicts a subgraph of a workload. Parent task T_p has three descendents, T_{c1}, T_{c2} and T_{c3}. Both shaded tasks T_{c2} and T_{c3} are fan-out tasks, since they start executing in parallel with T_{c1}. Task T_{c1} can be thought of as inheriting the processor from its parent T_p. However, only T_{c2} is an immediate-fan-out task, since it is the fan-out task that started first. In the case of equal starting times, the fan-out tasks with lowest index is selected as the immediate-fan-out task.

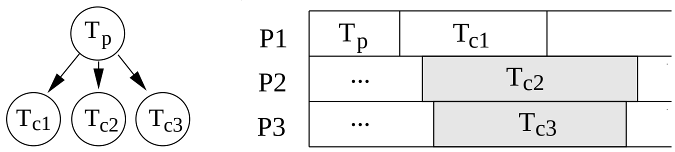


Fig. 1. Example of fan-out with real tasks

Figure 2 elaborates on fan-out of phantom tasks. The figure shows that a phantom task can be thought of as a phantom T_p running on a phantom processor P^p, i.e., an imaginary processor. Upon finishing, T_p releases a phantom child $T_{p'}$ and the shaded real task T_α, which are dispatched on phantom processor P^p and real processor $P1$ respectively. Thus, property (i) of the fan-out definition above is essentially the same as property (ii), except that the existence of the phantom child $T_{p'}$ causing the fan-out of T_α might not be obvious.

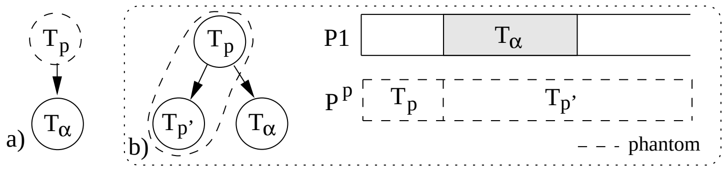


Fig. 2. Example of fan-out with phantom tasks

Immediate-fan-out tasks will build the basis for defining subsets of the workload with strong precedence constraints in order to avoid scheduling instability.

5. Strong Subgraph Precedence and Stability

The main result of this section is, that, if we selectively impose strong subgraph precedence on the precedence graph of tasks (including the phantom tasks), then we can eliminate the scheduling stability issue completely. One only has to impose the strong precedence subgraph relation based on immediate-fan-out tasks. For this purpose, all immediate-fan-out tasks T_{fi} are identified on the SGC. Next, strong precedence is defined on subgraphs G^i for each T_{fi} such that all tasks in $\mathbf{T}_{\leq fi}$ are in G^i. Thus, immediate-fan-out task T_{fi} is the task in G^i with latest standard starting time s_{fi}^{std}. Even though there might be other tasks T_j in G^i with $s_j^{std} = s_{fi}^{std}$, T_{fi} is the task with largest index, i.e., $j < fi$. The resulting structure of nested subgraphs is shown in Figure 3, where the immediate-fan-out tasks defining each strong precedence subgraph are identified. It can be seen that G^i contains subgraph G^{i-1}, and G^i is properly contained in G^{i+1}. The nested structure of the subgraphs is indicated by coloring, where darker shading indicates membership to more strong precedence subgraphs. Note that basing subgraph G^i on immediate-fan-out task T_{fi} does not imply that T_{fi} is the only fan-out task in $G^i \setminus G^{i-1}$. There might be others, i.e., $T_{f(i-1)}$ is the *first* but not necessarily the *only* fan-out task as a consequence of the same logical fork. Furthermore, by the very definition of G^i based on T_{fi}, inconsistency according to the definition of strong precedence subgraphs on page 6 cannot occur. The reason is that, given two strong subgraphs G^i and G^j where G^i is a proper subgraph of G_j, all tasks in G_i started before any tasks in $G^j \setminus G^i$ on the SGC.

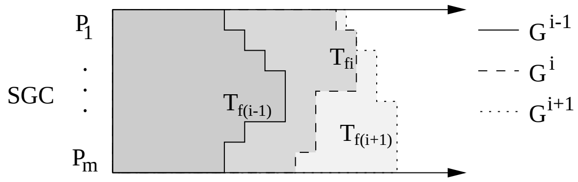


Fig. 3. Nested Strong Precedence Subgraphs

The following lemma is repeated from [3].

Lemma 1 *Given two tasks T_v and T_Δ where $v < \Delta$, T_v can be unstable only if a priority inversion and a logical fork* into $\mathbf{L}$ *have occurred such that $s_\Delta < f_\xi \leq r_v$.*

Proof: See [3, Theorem 2]. □

Lemma 1 *Given two subgraphs G^{i-1} and G^i based on immediate-fan-out tasks $T_{f(i-1)}$ and T_{fi} respectively, then there can be at most one fan-out task in $G^i \setminus G^{i-1}$ at the time $T_{f(i-1)}$ is started. This fan-out task, if it exists, must be T_{fi}.*

Proof: Since by definition G^{i-1} is properly contained in G^i, due to strong precedence all tasks in G^{i-1} have to be started first. Therefore, assume that all tasks in G^{i-1} have been started. Thus the next task considered for starting must be in G^i.

Assume, by contradiction, that there exists another fan-out task T_w in G^i, where T_w precedes T_{fi} in the priority list. By the definition of a fan-out task, T_w must have an unfinished forking task as an ancestor. All immediate-fan-out tasks up to $T_{f(i-1)}$ have started, implying that their respective forking parent must have finished. However, this means that there cannot be any more fan-out defined on these forking tasks either. For T_w this means that either one of the executing tasks from G^{i-1} or a task in $G^i \setminus G^{i-1}$ is its forking task. This qualifies T_w as an immediate-fan-out task. However, since T_w precedes T_{fi} in the priority list this contradicts the assumption that T_{fi} is the immediate-fan-out task defining G^i, i.e., T_w has a lower index. Thus T_{fi} is the only fan-out task in G^i at the time of the scan, unless its forking parent has finished, in which case there is no fan-out task left in G^i. Note that there cannot be other fan-out tasks defined on the forking parent of T_{fi} in $G^i \setminus G^{i-1}$, since by definition T_{fi} is the last task in G^i and those other fan-out tasks would thus be in G^{i+1}. □

Theorem 1 *If for each immediate-fan-out task T_{fi} a strong subgraph precedence is defined for subgraph $G^i = (V^i, E^i)$, where $V^i = \mathbf{T}_{\leq fi}$, no scheduling instability can occur.*

Proof: The workload consists of nested subgraphs with strong precedence constraints, $G^1, ..., G^l$, where T_{fl} is the last immediate-fan-out task. For each $G^i = (V^i, E^i)$, $1 < i \leq l$ we have $V^{i-1} \subset V^i$ and $E^{i-1} \subseteq E^i$. The only tasks with weak precedence are those in $\mathbf{T}_{>fl}$.

Assume that all tasks of G^{i-1} were on-time and the next task to be started is in G^i. Thus, due to strong precedence, the only unstarted tasks in G^i are in $G^i \setminus G^{i-1}$. By the definition of an immediate-fan-out task and as a consequence of Lemma 1, T_{fi} must be the lowest numbered fan-out task, provided it still has an *unfinished* forking parent. Otherwise, the fork has already occurred and T_{fi} is no longer a fan-out task.

Since T_{fi} is the highest numbered task in G^i, and due to strong precedence no task in $V \setminus V^i$ can be dispatched, there can be no priority inversion about T_{fi}. Thus T_{fi} is stable by the General Instability Condition [I1]. Therefore, instability is possible only if there is at least one unstarted task in $\mathbf{T}_{<fi}$.

Let T_v be any unstarted task in $\mathbf{T}_{<fi}$ at the time of dispatching. To enable instability at T_v, assume that a priority inversion about T_v by task $T_\triangle \in \{T_{(v+1)}, \ldots, T_{fi}\}$ occurs as a result of the current scan. Then, $s_\triangle$ is the current time. Note that T_{fi} is in the set, since its forking parent might have finished early.

Recall that $\mathbf{L}$ is the set of all tasks in $\mathbf{T}_{\leq v}$ which have not been completed by $s_\triangle$. By Lemma 1, T_v is stable unless $\mathbf{L}$ contains a fan-out task descended from an *unfinished* forking task at time $s_\triangle$. In the absence of a fan-out task in G^i there can be no forking task in $\mathbf{L}$, and instability cannot occur. Otherwise, by Lemma 1 there is exactly one fan-out task, i.e., T_{fi}. However, for any $T_v \in \mathbf{T}_{<fi}$, $T_{fi} \notin \mathbf{L}$ and again no instability can occur. □

So far we have not addressed the impact of the number of processors at run-time.

Corollary 1.1 *If for each immediate-fan-out task T_{fi} strong subgraph precedence is defined for subgraph $G^i = (V^i, E^i)$, where $V^i = \mathbf{T}_{\leq fi}$, an increase in the number of processors M cannot cause scheduling instability.*

Proof: Theorem 1 represents the special case where M is fixed. In the absence of a logical fork instability cannot occur by general instability condition [**I2**]. Thus assume the existence of logical forks.

Assume that all tasks of G^{i-1} were on-time and the next task to be started is in G^i. Since T_{fi} is the highest numbered task in G^i, there cannot be a priority inversion about T_{fi} and T_{fi} is thereby invulnerable according to general instability condition [**I1**]. Therefore, let T_v be any unstarted task in $\mathbf{T}_{<fi}$ at the time of dispatching. To enable instability at T_v, assume that a priority inversion about T_v by task $T_\triangle \in \{T_{(v+1)}, \ldots, T_{fi}\}$ occurs as a result of the current scan.

Showing that increasing the number of processors cannot cause instability is equivalent to showing that no processor constriction for tasks in the leftover set $\mathbf{L}$ can occur. By [11, Lemma 2] T_v cannot be unstable if the number of processors available to $\mathbf{L}$ at s_v^{std} is greater than or equal to the number of processors occupied by $\mathbf{L}$ at s_v^{std} on the SGC, i.e. if $M - M_X(s_v^{std}) \geq M_L^{std}(s_v^{std})$ where X is the set of all usurper tasks. This was true in Theorem 1. However, increasing the number of processors to $M' > M$ cannot change this inequality, since $M' - M_X(s_v^{std}) > M - M_X(s_v^{std}) \geq M_L^{std}(s_v^{std})$ and thus T_v cannot be unstable. □

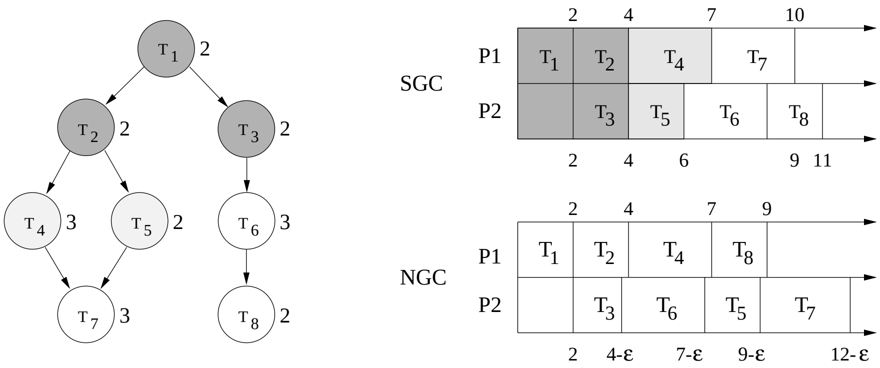
5.1. Stabilization Example

The stabilization method based on strong precedence subgraphs will be demonstrated for the eight task workload specified in Figure 4a, where task durations are indicated next to the vertices. The task priorities are defined by the SGC starting sequence in Figure 4b, reflected by task indices. Ignoring the coloring of the graph and Gantt chart, we can see instability in the NGC of Figure 4b. Specifically, given the SGC, if one shortens the execution time of T_3 by some arbitrary amount ϵ, then instability can be observed. As can be seen in the NGC, T_6 was able to usurp, and as a result T_5 and T_7 miss their deadlines defined by the SGC.

Defining strong subgraph precedence for appropriate subgraphs the problem can be prevented. The immediate-fan-out tasks in the SGC are T_3 and T_5, because they represent the first tasks exhibiting fan-out with their respective forking parents T_1 and T_2. Now one can define a strong precedence subgraph G^1 on the task set $V^1 = \mathbf{T}_{\leq 3} = \{T_1, T_2, T_3\}$, and G^2 on $V^2 = \mathbf{T}_{\leq 5} = \{T_1, T_2, T_3, T_4, T_5\}$. This can be seen on the SGC and the precedence graph of Figure 4, where strong precedence causes shading of the tasks involved. Note that T_1, T_2 and T_3 are shaded darker, since they are colored twice due to their membership in G^1 and G^2. Tasks T_4 and T_5 are shaded lighter, since they only belong to G^2 but not to G^1. At runtime, the tasks of the deepest nested and therefore darkest colored strong subgraph begin execution. Only after all tasks in G^1 have been started can tasks in G^2 be considered. However, this excludes T_6, the task that caused instability on the NGC of Figure 4b.

5.2. Comparing Approaches

The proposed solution to the instability problem based on strong precedence is an apriori stabilization method. The reasoning in Manacher's algorithm [14], which is also an apriori approach differs in two assumptions. Firstly, Manacher's algorithm cannot deal with the concept of phantom tasks. Secondly, Manacher's algorithm does not differentiate between fan-out tasks of specific forking tasks. As a result, multiple edges might be introduced to avoid instability in site of a full scan, i.e., there are no limitations into how deep the dispatcher scans into the priority list. By



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

Fig. 4. Example of Instability and Strong Precedence Stabilization

defining immediate-fan-out tasks, our approach needs to consider only the first fan-out task of a forking task. As such, by defining the strong precedence it adopts the limited scan depth method found in run-time stabilization. However, unlike in run-time stabilization, the degree of fan-out does not change as forking tasks finish. As a result, we are not burdened with re-defining fan-out tasks dynamically at run-time. On the other hand, this approach, like Manacher's, does not take advantage of run-time behavior and its effectiveness can vary depending on the workload. It should be noted, however, that resource utilization can always be improved at run-time using slack-time reclaiming. The slack-time of a task T_i is defined as the maximum time the starting of T_i can be delayed, i.e., the difference between the time of the scan and the standard starting time s_i^{std}. General techniques for resource reclaiming in the context of run-time stabilization have been presented in [13]. These basic concepts can be easily applied to the research presented here.

6. Summary

This paper presents a solution to the instability problem in non-preemptive static priority list scheduling. The strong precedence relation is introduced for subgraphs. By selectively defining strong precedence on portions of the task graph, instability can be avoided. Strong precedence on subgraphs are defined based on immediate-fan-out tasks which can be identified at system design time. Specifically, by selecting the strong subgraphs to be all tasks with indices less than or equal to the immediate-fan-out tasks, a structure of nested strong precedence subgraphs arises that is inherently stable.

This approach not only solves the instability problem with respect to timing anomalies, but also allows the number of processors to become dynamic, so that adding processors to the system cannot cause instability.

While strong subgraph precedence can be used to solve the instability problem, it represents a restriction at run-time since it limits the dispatcher to dispatching tasks within currently processed strong subgraphs. As a result, lower processor utilization may be possible, which in turn permits the use of slack-time reclaiming to overcome this possible inefficiency.

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, pp. 165-169.
3. Deogun, J.S., Kieckhafer, R.M., and Krings, A.W., "Stability and Performance of List Scheduling With External Process Delays," *Real-Time Systems*, Vol. 15, No. 1, July 1998, pp. 5-39.
4. Dror, M., "Chains and Trees: 'Strong' - 'Weak' order in job scheduling," *ORDER*, Vol. 14, No. 3, 1997-1998, pp. 211-228.
5. Dror, M., Kubiak, W., and Dell'Olmo, P., " 'Strong' - 'Weak' chain constrained scheduling," *Ricerca Operativa*, Vol. 27, 1998, pp. 35-49.
6. Dror, M., Kubiak, W., and Dell'Olmo, P., "Scheduling chains to minimize mean flow time," *Information Processing Letters*, 71, 1997, pp. 297-301.
7. Dror, M., Kubiak, W., and Leung, J.Y.-T., "Tree precedence in scheduling: The strong-weak distinction," 1997, (submitted for publication).
8. Dror, M., Leung, J.Y.-T., and Lin, C-H., "Uniform and identical machines with unit jobs and chain precedence," 1997, (submitted for publication).
9. Graham R.L., "Bounds on Multiprocessor Timing Anomalies," *SIAM J. Appl. Math.*, Vol. 17, No. 2, Mar. 1969, pp. 416-429.
10. Kieckhafer, R.M., et al, "The MAFT Architecture for Distributed Fault-Tolerance," *IEEE Trans. Computers*, Vol. C-37, No. 4, April, 1988, pp. 398-405.
11. Krings, A.W., "Inherently Stable Priority List Scheduling in an Extended Scheduling Environment," *PhD Thesis*, Dept. of Comp. Sci., Univ. of Nebraska, Lincoln, 1993.
12. Krings, A.W., R. Kieckhafer, and J. Deogun, "Inherently Stable Real-Time Priority List Dispatchers," *IEEE Parallel & Distributed Technology*, Winter 1994, pp. 49-59.
13. Krings, A.W., and M.H. Azadmanesh, "Resource Reclaiming in Hard Real-Time Systems with Static and Dynamic Workloads," *Proc. 30^{th} Hawaii International Conference on System Science*, IEEE Computer Society Press, Vol. I, 1997, pp. 616-625.
14. Manacher, G.K., "Production and Stabilization of Real-Time Task Schedules," *JACM*, Vol. 14, No. 3, July 1967, pp. 439-465.
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., K. Ramamritham and J.A. Stankovic, "Resource Reclaiming in Real-Time", *Proc. Sixth Real-Time Systems Symposium*, Dec 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.
18. Wensley, J.H, et al, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proc. of the IEEE*, 66 (11) Oct. 1978, pp. 1240-1255.