Resource Reclaiming in Hard Real-Time Systems with Static and Dynamic Workloads

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 resource reclaiming in the context of non-preemptive priority list scheduling for hard real-time systems. Such scheduling is inherently susceptible to multiprocessor timing anomalies. We present low overhead run-time stabilization methods for a general tasking model that allows phantom tasks as a mechanism to model processor external events.

A family of scheduling algorithms is defined, that guarantees run-time stabilization for systems consisting of tasks with hard and soft deadlines. The later, i.e. soft tasks, may arrive dynamically. Stabilization is addressed in the context of dynamic and static task to processor allocation. Previous stabilization methods, focused on apriori stabilization for static workloads with dynamic task to processor allocation, thus cannot support this general scheduling model.

By taking advantage of run-time information, the stabilization algorithms use the scan-window approach to prevent instability from occurring. Mechanisms are introduced that explicitly control the run-time behavior of tasks with hard deadlines. As a consequence, processor resources become available that can be used to improve processor utilization and response time of soft tasks. The resulting scan algorithms are intended for real world applications where low run-time overhead and a realistic task model are needed.

1 Introduction

The increasing computational complexity of many control applications promotes the use of multiprocessor environments. The application tasks may be distributed over multiple processors to take advantage of parallelism of the workload, or as a redundancy issue for fault-tolerant reasons. Many applications operate in hard real-time safety critical environments, where the missing of deadlines could have catastrophic results, i.e. loss of human lives, environmental damage or unacceptable cost. The algorithms scheduling the workload, typically represented by a task graph, must be provably correct and free of side effects.

Most scheduling methods can be partitioned into preemptive and non-preemptive approaches. Whereas preemptive scheduling generally offers a greater degree of flexibility and has potential for higher resource utilization, it proves to be hard, if not impossible, to fully predict the non-deterministic effects and overhead of context switching [1, 8, 9].

Non-preemptive static priority list scheduling has been selected as a simple, low-overhead approach in systems like the Reliable Computing Platform (RCP) [1], the Multicomputer Architecture for Fault Tolerance (MAFT) [10], the Spring Kernel [23], or specific applications like turbojet engine control [20]. However, list scheduling is vulnerable to timing anomalies [7], where the reduction in task execution times of one or more tasks can cause deadlines to be missed. Task durations may generally not 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 [17]. Several approaches have been proposed to avoid timing anomalies. Whereas *apriori* stabilization is based on Manacher's algorithm [18] which only supports static workloads, a family of *run-time stabilization* algorithms has been introduced [12, 14] that is less restrictive than previously known stabilization methods.

The individual task execution times may vary between minimum and maximum values. Even though worst case task duration must be considered, the *mean* run-time durations are often one order of magnitude lower [2]. At run-time, this effect is generally compounding. The resulting available resources are the subject of many recent studies on resource reclaiming. Slack-time is used in preemptive and non-preemptive environments to increase fault-tolerance [6], to minimize the response time of soft workloads [16], or to jointly schedule hybrid workloads consisting of combinations of periodic, aperiodic, sporadic or adaptive tasks with hard and soft deadlines [3, 4, 5, 22, 24].

This paper focuses on resource reclaiming in the context of run-time stabilization for systems employing workloads consisting of a static hard real-time core workload, supplemented by dynamically arriving tasks. The results are presented for dynamic task to processor allocation and are expanded for the more restrictive case of static task allocation. A framework is developed that allows stable dispatching of hybrid workloads consisting of static tasks with *hard* deadlines and dynamically arriving tasks with *soft* deadlines. The set of *hard tasks* typically constitutes the control workload of the real-time application, whereas *soft tasks* are non-critical, possibly dynamically arriving tasks with soft deadlines. Mixed task scheduling models have been the subject of many algorithms inspired by slack stealing [16], including [3, 4, 5, 24]. For non-preemptive applications, reclaiming has been considered in dynamic multiprocessor applications [22] in conjunction with the Spring Project [23], where reclaiming is considered in the context of priority inversions called *non-passing* and *passing*. Our approach has no restrictions on priority inversions and is purely run-time dependent. Furthermore, in contrast to [22] it allows reclaiming with constant overhead in site of extended passing.

Some applications may not benefit from early finishing times of the hard workload and prioritize early starting times of soft tasks instead. The problem of minimizing the response time of soft aperiodic tasks, while guaranteeing deadlines of hard workloads, has been explored for preemptive scheduling in [16]. Mechanisms based on special phantom tasks are presented to artificially delay and control the run-time behavior of hard tasks. The resulting idle times are then reclaimed by allowing safe priority inversions and slack-time reclaiming. Several provably stable run-time dispatching algorithms are presented allowing resource reclaiming of varying complexity. Even though derived for dynamic task allocation, they illustrate strict limitations for reclaiming in static allocation models. A task graph transformation based on phantom tasks is used to allow distributed static dispatching.

This paper is a continuum to the recent results in run-time stabilization and presents a comprehensive framework which is based on the list scheduling paradigm. It is more general than traditional list schedulers and thus opens up a larger application domain. Specifically, it allows run-time stabilization in hard real-time systems implementing (1) static and dynamic workloads, (2) dynamic or static task to processor allocation, and (3) mechanisms to implement *slack-time tuning*, i.e. to control the degree of resource reclaiming. Motivation for slack-time reclaiming is to increase processor utilization and to increase response time to the dynamic workload. Section 2 describes the basic task model and gives preliminary definitions. Section 3 addresses basic run-time stabilization issues and mechanisms to selectively force slack-time. In Section 4, safe run-time dispatching algorithms for dynamic task allocation are presented. Section 5 describes the resource reclaiming mechanisms in the context of dynamic task allocation. Section 6 addresses dispatching with static task allocation and its limitations. Finally, Section 7 concludes the paper with a summary and remarks on future work.

2 Preliminaries

2.1 Task Model

A task T is the basic unit of computation, consisting of a set of sequentially executed instructions. Associated with each task T_i is a maximum and minimum 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, and finishing time f_i. Dependencies among tasks are defined by a partial order, resulting in a directed acyclic precedence graph. Tasks are assumed to be executed on M homogeneous processors.

In order to model events external to a processor, non-transparent overhead or task synchronization, special tasks called *phantom tasks* are defined. These tasks are fully incorporated into the precedence graph. Although they consume time, unlike *real tasks*, they consume no resources. As a result, phantom tasks may *always* be started upon becoming released.

The workload is partitioned into a static workload called the *hard task set*, $\mathbf{T}^H = \{T_1, ..., T_{N'}\}$, and dynamically arriving tasks called *soft task set*, $\mathbf{T}^S = \{T_{N'+1}, ..., T_N\}$, where N' is the number of hard tasks and N is the total number of tasks in the system at the present time t_n, i.e. $N = |\mathbf{T}^H| + |\mathbf{T}^S|$. Whereas N' is fixed at design time, N is run-time dependent. The hard task set typically constitutes the control workload of the hard real-time application. Consequently, tasks in $\mathbf{T}^H$ have *hard deadlines*. All information about hard tasks is known apriori, except for the actual task durations, which are in the interval $[c_i^{min}, c_i^{max}]$.

Soft tasks on the other hand are considered non-critical. They are defined at design time or may enter the system at run-time. To reduce penalties associated with missing soft deadlines it is desirable to minimize their response time [16].

2.2 Terminology

The philosophy of the algorithms described in this paper is adopted from *fixed priority list scheduling*, where 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. Whereas the scheduler is executed only once at design time, the dispatcher arbitrates tasks during run-time.

A *Scenario* describes the schedule obtained by using a particular set of task durations. The *Standard Scenario* is the scenario in which each T_i uses the maximum computation time c_i^{max} [18]. The Gantt chart depicting the standard scenario is called the *Standard Gantt Chart* (SGC). Task deadlines of hard tasks are their respective finishing times in the SGC. It should be noted that the SGC is some feasible schedule, defined at design time. Optimality is desirable, but not necessary.

In a *Non-Standard Scenario*, tasks T_i from $\mathbf{T}^H \cup \mathbf{T}^S$ execute with $c_i^{min} \leq c_i \leq c_i^{max}$. However, at least one task $T_j \in \mathbf{T}^H$ has duration c_j with less than its maximum computation time c_j^{max}, i.e. $c_j < c_j^{max}$. The resulting Gantt chart is called *Non-Standard Gantt Chart* (NGC).

The dispatcher selects tasks from a list called *projective list*. This list is in one-to-one correspondence with the SGC, i.e. its tasks represent the task starting sequence of the SGC [18]. Let $\mathcal{L}^H$ and SGC^H denote the projective list component of hard tasks $\mathbf{T}^H$ and the SGC defined by $\mathbf{T}^H$ respectively. Conversely, let $\mathcal{L}^S$ and SGC^S denote the priority list component of soft tasks $\mathbf{T}^S$ and the portion of the SGC defined by $\mathbf{T}^S$ respectively. Thus the SGC consists of SGC^H followed by SGC^S.

In contrast to $\mathcal{L}^H$, list $\mathcal{L}^S$ is dynamic since $\mathbf{T}^S$ is a dynamic task set. As a result, SGC^S bares no real meaning by the definition of dynamic soft tasks, i.e. $\mathbf{T}^S$ is indeterminate and must be assumed to be purely run-time dependent. Therefore, SGC^S may be seen as *any* arbitrary schedule of the tasks in $\mathbf{T}^S$ at the time of the scan. This assumption is consistent with the definition of soft tasks, whose deadlines are non-critical. Soft tasks are inserted into $\mathcal{L}^S$ according to their priority upon arrival into the system. If they all arrive at time $t = 0$, the hybrid workload becomes pure static.

Let $\mathcal{L}^E$ denote the *Extended Priority List* consisting of $\mathcal{L}^H$ concatenated with $\mathcal{L}^S$, i.e. $\mathcal{L}^E = \mathcal{L}^H \circ \mathcal{L}^S$. Thus the dispatcher scans a *single* list that first contains all hard tasks followed by soft tasks.

A scenario is *stable* if there exists no scenario in which the finishing time of any $T_i \in \mathbf{T}^H$ 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 $T_i \in \mathbf{T}^H$, the latest safe starting time s_i is s_i^{std}, the starting time on the SGC as denoted by superscript *std*. Thus, if $s_i \leq s_i^{std}$, then $f_i \leq f_i^{std}$. Since stability of a task T_i is meaningful only if $T_i \in \mathbf{T}^H$, when addressing stability issues, the inclusion of T_i in the set of hard tasks will be assumed and will not be explicitly stated in the remainder of the paper.

Several task sets will be used throughout the paper. Let $\mathbf{T}_{<i}$ denote the set of all tasks with indices less than i. Thus for any T_i in $\mathbf{T}^H$, this means that $\mathbf{T}_{<i}$ is the set of tasks which started before T_i on the SGC. In a projective list, $\mathbf{T}_{<i}$ contains all tasks T_j such that $j < i$. $\mathbf{T}_{\leq i}$ 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.

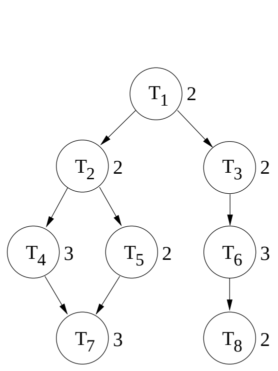
2.3 Instability

An example of instability is shown in Figure 1. The precedence graph contains eight 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 and 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.

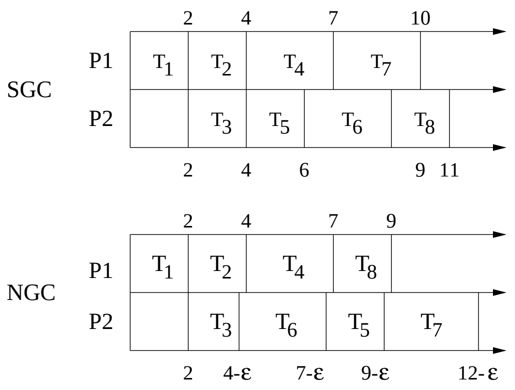
To avoid instabilities two stabilization methods have been proposed that can be partitioned into *apriori* and *run-time* stabilization.

1. In *Apriori Stabilization* methods, stabilization is achieved by (1) restricting the dispatcher, i.e. fixing the task starting sequence or task starting times, or by (2) modifying the task graph by introducing additional precedence constraints [10, 18, 19, 21]. Potentially poor utilization for the first and addition of many edges in the second case have motivated the development of less restrictive stabilization methods.

2. *Run-Time Stabilization* is a less restrictive stabilization method, where the dispatcher limits the



a) Precedence Graph



b) T_3 shortened by ϵ on the NGC

Figure 1: Example of Instability

depth of its scan into the task list in order to avoid instabilities. This approach takes advantage of information available at run-time [13, 14].

3 Run-Time Stabilization & Slack-Time Tuning

The first step in the chain of events possibly leading to instability is a *priority inversion* by some task T_x, such that $s_x < s_i$ for some T_i with $i < x$ [11]. Task T_x is said to *usurp*. It is the responsibility of the run-time stabilization algorithm to only allow dispatching of such usurper tasks that cannot induce instability.

3.1 The Scan Window

When a processor finishes its current task, the traditional priority list dispatcher starts at the head of the list, scanning the list until it finds a ready task, if one exists. The scan window approach restricts the scan, by limiting the scan depth to the size of the window. The *scan window* $\mathbf{\Sigma}$ is this subset of *unstarted* real tasks scanned by the dispatcher, i.e. $\mathbf{\Sigma} = \{T_u, ..., T_l\}$ where T_u and T_l are the first and last real tasks visible to the dispatcher. If the number of tasks scanned is limited such that no usurper task is ever started before a vulnerable task, stability can be enforced at run-time.

3.2 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. The initiating release is called a *logical fork*, since either case causes the occupancy of one more processor, 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. Several tasks and task sets with different fan-out properties are defined. They build the foundation of the run-time dispatching algorithms described in Section 4.2.

3.2.1 Fan-out Tasks

A *fan-out task* is defined as a real task with *at least* one of the following properties: (1) it is a child of an unfinished phantom task, (2) 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.

Let T_w be the first fan-out task and T_u the first task in the window. On the SGC, T_w was the task that occupied an additional processor, i.e. T_w caused a fan-out of 1. It will be shown that one can safely scan up to T_w. This defines a scan window $\mathbf{\Sigma} = \{T_u, ..., T_{w-1}\}$. Assume that T_w is the only fan-out task in the workload. If one wants to scan past T_w, stability is guaranteed[1] only provided there is at least one idle processor that can be reserved to *absorb* the possible fan-out of T_w. In general, to scan past T_w, one needs to identify additional fan-out tasks in $\mathbf{T}_{>w}$. Let $T_{w1} = T_w$, and define T_{wi} as the i^{th} fan-out task. All T_{wi} are called *basic fan-out tasks*, in that each T_{wi} can cause a fan-out of 1. Note that phantom tasks cannot be fan-out tasks since they do not consume resources.

[1]At this point we ignore issues of slack-time reclaiming.

3.2.2 Effective Fan-out

Let $\mathcal{F}(T_w)$ be a function that indicates how many basic fan-out tasks with indices less than or equal w are overlapping on the SGC at standard starting time s_w^{std}. Thus $\mathcal{F}(T_w)$ is the cardinality of set $\{T_i : T_i \in \mathbf{T}_{\leq w}$, T_i *is a fan-out task, and* $s_i^{std} \leq s_w^{std} < f_i^{std}\}$. A task T_w is said to have an *effective fan-out* of $\mathcal{F}(T_w)$. Since soft tasks are non-critical and dynamic, their fan-outs bare no real meaning and are defined as zero.

Not every basic fan-out task contributes to an increase in the effective fan-out. Assume that several basic fan-out tasks exist such that their executions do not overlap on the SGC, i.e. $f_{wi}^{std} < s_{wj}^{std}$ for any T_{wi} and T_{wj}, with $i < j$. It can be shown [13] that these *non-overlapping* basic fan-out tasks can collectively contribute only to an effective fan-out of 1. Figure 2 shows the subgraph and the SGC of a system with two non-overlapping basic fan-out tasks T_{w1} and T_{w2}. Fan-out $\mathcal{F}(T_{w1}) = 1$ and $\mathcal{F}(T_{w2}) = 1$, but the effective fan-out of $\{T_{w1}, T_{w2}\}$ at any time is 1. Let T_{ei} denote the *lowest numbered* basic fan-out task with $\mathcal{F}(T_w) = i$. Then T_{ei} is called an *effective fan-out task*. T_{ei} is thus the first task executing *in parallel* with $i-1$ other fan-out tasks from $\mathbf{T}_{<i}$. T_{e1} is the first effective fan-out task ($\mathcal{F}(T_{e1}) = 1$), T_{e2} is the second ($\mathcal{F}(T_{e2}) = 2$), and so forth. By the definition of fan-out, $T_{e1} = T_{w1}$. Every effective fan-out task is also a basic fan-out task, but the reverse is not necessarily true. It should be noted that T_{ei} is not necessarily the only fan-out task with a fan-out of i, but it is the *first*. As a convention, task subscripts starting with letter e will be reserved for effective fan-out tasks.

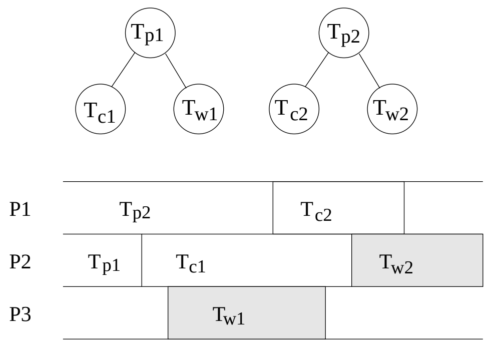


Figure 2: Non-Overlapping Fan-out Tasks T_{w1} and T_{w2}

3.2.3 Scan Frames

The priority list can be partitioned starting with the first unstarted task. The general priority list at the time of the scan is $(T_{e0}, ..., T_{e1}, ..., T_{e2}, ..., T_{ek}, ...)$. Task $T_{e0} = T_u$ if[2] $\mathcal{F}(T_u) = 0$, otherwise T_{e0} does not exist and the list starts with T_{e1}. T_{ek} is the last effective fan-out task. Positioned between T_{ei} and $T_{e(i+1)}$ are any number of tasks T_j with effective fan-outs $0 \leq \mathcal{F}(T_j) \leq i$. These tasks, including T_{ei}, are called the *Scan-Frame* of T_{ei} and are denoted by $\mathbf{\Delta}_{ei}$. Thus frame $\mathbf{\Delta}_{ei}$ is the set $\{T_{ei}, ..., T_{e(i+1)-1}\}$. It will be shown that with the knowledge of idle processors I at the time of the scan, the scan window $\mathbf{\Sigma}$ can be expressed as the sequence of frames $\mathbf{\Delta}_{e0}, \mathbf{\Delta}_{e1}, ..., \mathbf{\Delta}_{I-1}$.

3.3 Forcing Slack-Time

The *slack-time* of a task T_i is the maximal time the starting of task T_i may be postponed without causing T_i to miss its deadline. If the dispatcher should give preference to soft tasks, yet guarantee all deadlines of tasks in $\mathbf{T}^H$, early release of hard tasks has to be discouraged. By artificially delaying hard tasks, possibly to their standard starting times, the slack-time of the delayed task becomes available for so-called *slack-time reclaiming.*

Phantom tasks called *delay enforcement tasks* (det), can serve as delay mechanisms. Thus each hard task T_i to be delayed will be assigned a phantom task $T_{det(i)}$ which is inserted as an immediate parent into the precedence graph as shown in Figure 3. Assuming all $T_{det(i)}$ are released at time $t = 0$, the maximum duration of $T_{det(i)}$ can be set as late as $c_{det(i)}^{max} = s_i^{std}$. Generally,

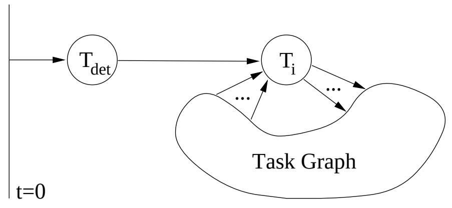


Figure 3: Delay Enforcement Task $T_{det(i)}$

it will be desirable to assign delay enforcement tasks to few selected tasks of $\mathbf{T}^H$, in an attempt to reduce the number of $T_{det(i)}$'s. Benefits of early finishing hard tasks and response to soft tasks are in competition. This eliminates the overly restricted case of strictly enforcing standard starting times of all hard tasks from consideration. Thus an application can control the selection of hard tasks T_i to be delayed, through appropriate minimum and maximum durations $c_{det(i)}^{min}$ and $c_{det(i)}^{max}$ of the respective delay enforcement tasks $T_{det(i)}$.

[2] $\mathcal{F}(T_u) \neq 0$ if and only if T_u is descended from a phantom task.

This customization of selectively forcing slack-time is called *slack-time tuning*.

It should be noted that delay enforcement tasks are phantom tasks. As such they do not consume any processor resources but only time. From the dispatcher's point of view, they induce no overhead other than maintaining additional precedence constraints in the dispatcher's data structure. This increase in precedence constraints, due to delay enforcement phantom tasks, should not be confused with additional edges in apriori stabilization. Apriori stabilization is not capable of dealing with the concept of phantom tasks as a general mechanism to model external events. Furthermore, scenarios may occur at run-time where no ready task exists, but where the termination of a $T_{det(i)}$ would release a safe task T_i. In this case, it is advantageous to terminate $T_{det(i)}$ to increase processor utilization.

3.4 Reclaiming Philosophy

At run-time the dispatcher selects tasks from the extended priority list based on the scan window defined on the current scan. Since deadlines must be guaranteed for hard tasks, one could argue that soft tasks should be dispatched solely based on slack-time reclaiming. However, this approach is quite restrictive, since it does not take under consideration the issues of slack-time tuning. Furthermore, the selection of the tasks, which will be the basis for slack-time reclaiming, is not trivial and algorithm dependent, i.e. dispatching a task based on the slack-time of the first hard task in the window can become overly conservative. Two types of resource reclaiming are considered.

1. Processor resources are reclaimed by allowing safe priority inversions. The effects of this implicit algorithmic reclaiming method increase with the sophistication of the scan window selection, i.e. algorithms resulting in a larger safe scan window have higher probabilities of reaching a safe ready task.

2. If no task can be found in the scan window, reclaiming can be used strictly based on slack-time of the task defining the size of the window.

4 Dynamic Run-Time Dispatching

Window dispatching algorithms are first shown for dynamic task to processor allocation. In this allocation scheme, tasks can be executed on any idle processor. Static task allocation is discussed later in Section 6.

4.1 Dispatcher Task Selection

When using the extended priority list $\mathcal{L}^E$, the position of the window with respect to the junction of $\mathcal{L}^H$ and $\mathcal{L}^S$ is important. Assume a task finishes and the dispatcher scans $\mathcal{L}^E$ via the scan window $\mathbf{\Sigma}$ defined by the number of processors that can be reserved for fan-out tasks. Let $\mathbf{\Delta}_{el}$ denote the last scan frame of $\mathbf{\Sigma}$ to include tasks from $\mathcal{L}^H$. Furthermore, let T_δ denote the last unstarted real task in $\mathcal{L}^H$. At the time of the scan several window scenarios can occur which will be the basis for the dynamic dispatching algorithms:

1. T_δ does not exist. This is the case when $\mathcal{L}^H$ is empty.

2. $T_\delta \notin \mathbf{\Delta}_{el}$. Tasks from $\mathbf{T}^S$ are not visible to the dispatcher.

3. $T_\delta \in \mathbf{\Delta}_{el}$. The window includes $\mathcal{L}^S$.

4.2 Dynamic Dispatch Algorithms

The following four tasks, defined at the time the list is scanned, are important in the statement of run-time dispatchers.

1. T_α: the lowest numbered real task in $\mathbf{T}^H$ that has an unfinished phantom parent.

2. T_β: the lowest numbered real task in $\mathbf{T}^H$ which is also the second real child of an unfinished forking task.

3. T_γ: the lowest numbered real task in $\mathbf{T}^H$ such that
   (a) T_γ is descended from an unfinished forking task,
   (b) on the SGC, T_γ started while another real descendent of the same forking task was running.

4. T_δ: (as defined previously) the highest numbered unstarted real task in $\mathbf{T}^H$.

Let I be the number of idle processors at the time of the scan (including the processor which initiated the scan). Table 1 shows run-time dispatching algorithms, where T_u denotes the first unstarted real task in $\mathbf{\Sigma}$. The key element in the algorithms is the position of T_δ and the sentry tasks $T_{u+I-1}, T_{a+I-1}, T_{c+I-1}, T_{w+I-1}, T_{wI}$, and T_{eI}.

The most general algorithm is Algorithm 6. It is based on the observation that given I idle processors, one can *unconditionally* select tasks from the first I scan frames [15], i.e. $\mathbf{\Sigma} = \mathbf{\Delta}_{e0}, ..., \mathbf{\Delta}_{e(I-1)}$. If T_δ is beyond these frames ("if-clause" of Alg'm 6), then the

Alg'm	if [a]	then	else
1	$\delta \geq u + I - 1$	$\mathbf{\Sigma} = \{T_u, ..., T_{u+I-1}\}$	$\mathbf{\Sigma} = \mathcal{L}^E$
2	$\delta \geq a + I - 1$, where $a = \min[\alpha, (u+1)]$	$\mathbf{\Sigma} = \{T_u, ..., T_{a+I-1}\}$	$\mathbf{\Sigma} = \mathcal{L}^E$
3	$\delta \geq c + I - 1$, where $c = \min[\alpha, \beta]$	$\mathbf{\Sigma} = \{T_u, ..., T_{c+I-1}\}$	$\mathbf{\Sigma} = \mathcal{L}^E$
4	$\delta \geq w + I - 1$, where $w = \min[\alpha, \gamma]$	$\mathbf{\Sigma} = \{T_u, ..., T_{w+I-1}\}$	$\mathbf{\Sigma} = \mathcal{L}^E$
5	$\delta \geq wI$	$\mathbf{\Sigma} = \{T_u, ..., T_{wI-1}\}$	$\mathbf{\Sigma} = \mathcal{L}^E$
6	$\delta \geq eI$	$\mathbf{\Sigma} = \{T_u, ..., T_{eI-1}\}$	$\mathbf{\Sigma} = \mathcal{L}^E$

[a] If the sentry tasks in the algorithms do not exist, a full scan is allowed.

Table 1: Hybrid Scan Window Dispatchers

I frames define the window $\mathbf{\Sigma} = \{T_u, ..., T_{eI-1}\}$. However, if T_δ is in the I frames ("else-clause"), then a full scan is safe, i.e. $\mathbf{\Sigma} = \mathcal{L}^E$. Algorithm 1 through 5 are special cases and result in subsets of the window defined by Algorithm 6. For a better understanding of the motivation of the other algorithms, the reader is referred to the derivation of dispatching algorithms in the context of a static hard task system [14]. Due to space restrictions stability proofs of the algorithms are not shown but can be found in [15]. Issues of complexity are discussed in [13, 15]. However, it should be noted that run-time complexities range from $O(M)$ to $O(N)$ for linear and $O(logN)$ for heap implementations, where M and N are the number of processors and tasks respectively. Furthermore, by bounding the size of the window by a constant c, linear dispatchers achieve $O(1)$.

5 Reclaiming Beyond the Window

The window size of the previous algorithms is based on the number of idle processors, and the window consists of *consecutive* tasks of $\mathcal{L}^E$ and is *unconditionally* safe. However, it will be shown that with the knowledge of the maximum duration of a usurper task conditional dispatching is possible. But first some preliminary lemmas are needed.

Assume that a priority inversion occurs so that a task T_x from $\mathbf{T}_{>v}$ starts before T_v, i.e. $s_x < s_v$ and $x > v$. The *Leftover Set* $\mathbf{L}$ consists of all tasks in $\mathbf{T}_{\leq v}$ which have not finished by s_x. Specifically, $\mathbf{L} = \{T_l \in \mathbf{T}_{\leq v} : f_l > s_x\}$. Recall that M is the number of processors.

Lemma 1 *Let $M_X(s_v^{std})$ and $M_L^{std}(s_v^{std})$ be the number of processors occupied by usurper tasks and set* $\mathbf{L}$ *at s_v^{std} respectively. 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 $M - M_X(s_v^{std}) < M_L^{std}(s_v^{std})$.*

Proof: See [11, Corollary 3.4]. □

The next lemma shows that all tasks in scan frame $\mathbf{\Delta}_{e0}$ are *unconditionally* stable and are thus not affected by *any* usurper task. It should be pointed out that invulnerability of $\mathbf{\Delta}_{e0}$ does not necessarily imply stability for subsequent frames.

Lemma 2 *Let T_v be any task in $\mathbf{\Delta}_{e0}$ at the time of the list scan. Then T_v is invulnerable to instability.*

Proof: See [15, Lemma 3]. □

5.1 Impact of Usurper Task Durations

Assume an algorithm scanning $\mathbf{\Sigma} = \mathbf{\Delta}_{e0}, \mathbf{\Delta}_{e1}, ..., \mathbf{\Delta}_{e(I-1)}$ cannot find a ready task, but a ready task T_x exits beyond $\mathbf{\Delta}_{e(I-1)}$. It will be shown that the only scan frames that need to be investigated for vulnerability are those which contain tasks T_v whose SGC^H starting time s_v^{std} overlap time-wise with the execution of T_x on the NGC, assuming T_x were started. This means that scan frames $\mathbf{\Delta}_{ej}$ with s_{ej}^{std} greater than the maximal finishing time of T_x need not be considered.

Lemma 3 *Assume a usurper task T_x has started at time s_x, and define $f_x^{max} = s_x + c_x^{max}$. Then no task $T_v \in \mathbf{T}^H$ with $s_v^{std} > f_x^{max}$ can become unstable as a result of starting T_x.*

Proof: Assume T_v is any task in $\mathcal{L}^H$ with $s_v^{std} > f_x^{max}$, i.e. T_v is any hard 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. $T_v \in \mathbf{T}^S$ need not be considered by the definition of soft tasks. □

The implication of Lemma 3 is that a soft task T_s may be started based on its maximum execution time. Then any task T_x may be safely started if each scan frame overlapping with the maximal execution of T_x is guaranteed its reserved processors.

5.2 Slack-Time Reclaiming

The *slack-time* of a task T_v, denoted by $\mathcal{S}(T_v)$, is defined at t_n as $\mathcal{S}(T_v) = s_v^{std} - t_n$. Thus it is the maximal time the starting of task T_v may be postponed without causing T_v to miss its deadline. The starting of a task $T_x \in \mathbf{T}_{>v}$ is said to be *slack-time safe* with respect to T_v if $c_x^{max} < \mathcal{S}(T_v)$. In general, by Lemma 3, any task T_x from $\mathbf{T}_{>v}$ can be started if $c_x^{std} < \mathcal{S}(T_v)$. If a scan window algorithm fails to find a ready task in its maximal window, the slack time argument can be used to start a task T_x from beyond the window. The reclaimable slack-time available, however, depends upon the task T_v on which it is defined. Intuition might prompt us to use the slack-time of the first task in the scan window, or perhaps the first fan-out task T_{w1}. However, since slack-time is defined by the SGC starting times, i.e. $s_v^{std} - t_n$, it is desirable to find the T_v with the largest numbered index, thus resulting in the largest slack-time.

Finding T_v follows from a simple observation about scan frames, as will be shown using Algorithm 6. Assume that Algorithm 6 could not find a ready task in its maximal unconditional window $\mathbf{\Sigma} = \mathbf{\Delta}_{e0}, ..., \mathbf{\Delta}_{e(I-1)}$, and $T_\delta \notin \mathbf{\Sigma}$. Then the first task beyond the window vulnerable to instability is T_{eI}. A usurper task $T_x \in \mathbf{T}_{\geq eI}$ can be started if $c_x^{std} < \mathcal{S}(T_{eI})$. T_{eI} is the largest numbered task on which slack time reclaiming can be based. The following theorem generalizes slack-time reclaiming.

Theorem 1 *Assume T_δ is not in the safe window $\mathbf{\Sigma} = \mathbf{\Delta}_{e0}, ..., \mathbf{\Delta}_{e(i-1)}$ and no ready task exists in $\mathbf{\Sigma}$. Then a task T_x from beyond the window can be safely started if $c_x^{std} < \mathcal{S}(T_{ei})$.*

Proof: Assume priority inversion by T_x with $c_x^{std} < \mathcal{S}(T_{ei})$ has occurred. Let T_v be any task in $\mathbf{T}_{<x} \cap \mathbf{T}^H$. Tasks in $\mathbf{T}^S$ need not be considered by the definition of soft tasks. No T_v in $\mathbf{T}_{<ei}$ can be unstable, since the window is safe by assumption. Therefore let T_v be in $\mathbf{T}_{\geq ei} \cap \mathbf{T}^H$. Since f_x^{max} is less than s_{ei}^{std}, T_x must finish before any T_v started on the SGC. Then stability of all T_v in $\mathbf{T}_{<x}$ follows immediately from Lemma 3, which states that no task T_v with $s_v^{std} > f_x^{max}$ can be unstable. □

The next theorem shows that, based on a single sentry task, all window dispatchers presented so far can be extended to include slack-time safe ready tasks beyond the scan window.

Theorem 2 *If scan window algorithms 1 though 6 fail to find a ready task in their maximal unconditional scan window with $T_\delta \notin \mathbf{\Sigma}$, then a ready task T_x from beyond the scan window can be safely started if $c_x^{std} < \mathcal{S}(T_s)$. The sentry tasks T_s for each algorithm are:*

Algorithm	*1*	*2*	*3*
Sentry Task	$T_{u+(I-1)}$	$T_{a+(I-1)}$	$T_{c+(I-1)}$
Algorithm	*4*	*5*	*6*
Sentry Task	$T_{w+(I-1)}$	T_{wI}	T_{eI}

Proof: The condition $T_\delta \notin \mathbf{\Sigma}$ exists, since otherwise $\mathbf{\Sigma} = \mathcal{L}^E$ and with a full scan no ready tasks would be in the system. For Algorithm 6, the first task beyond the maximal window is T_{eI}. Then starting a usurper T_x with $c_x^{std} < \mathcal{S}(T_{eI})$ is safe by Theorem 1. Now make the following observations [15] about the relative values of u, a, c, w, wI and eI.

$$u \leq a \leq c \leq w \leq wI \leq eI$$

$$u+(I-1) \leq a+(I-1) \leq c+(I-1) \leq w+(I-1) \leq wI \leq eI.$$

Thus the scan windows of all algorithms are subsets of the scan window of Algorithm 6, which is stable. □

6 Static Run-Time Dispatching

In the previous section task to processor allocation was considered dynamic. However, some hard real-time environments do not allow such allocation, and tasks are allocated statically to processors at design time. The motivation for this can be the use of heterogeneous processors, i.e. special purpose processors, or the need for keeping data local to avoid the overhead involved in migrating possibly large sets of data to other processors.

In static allocation it is assumed that each processor is assigned a set of tasks, i.e. the tasks associated with the processors on the SGC^H. Furthermore, dynamically arriving tasks may be assigned to specific processors at tun-time. Each processor P_i maintains its own local extended priority list $\mathcal{L}_i^E$, composed of $\mathcal{L}_i^H$ and $\mathcal{L}_i^S$, which is scanned upon becoming idle.

The workload needs to be statically assigned to processors. Let $\mathbf{G}(\mathbf{T}, \mathbf{E})$ denote the task graph of the global workload with task set $\mathbf{T}$ and edge set $\mathbf{E}$ defined by the precedence constraints. Local workloads $\mathbf{G}_i$ at Processor P_i are derived as follows:

1. Generate $\mathbf{G}_i$ from $\mathbf{G}(\mathbf{T}, \mathbf{E})$ by substituting each real task T_j in $\mathbf{T}$ not assigned to processor P_i by a phantom task T_j^p with equal task duration.

2. Any T_j^p not affecting the local workload can be eliminated, by recursively removing all tasks T_j^p which are leafs, i.e. T_j^p with no descendents.

Figure 4 demonstrates the procedure using the sample workload of Figure 1, assuming that $\{T_1, T_2, T_4, T_7\}$ and $\{T_3, T_5, T_6, T_8\}$ are assigned to processor P_1 and P_2 respectively. $\mathbf{G}_1$ and $\mathbf{G}_2$ result from procedure step 1, where T_j^p are shaded. Task reductions of step 2 are indicated by dotted lines. The implications of the trans-

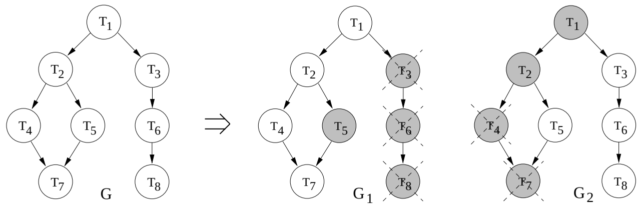


Figure 4: Task Graph Transformation

formation is that when a processor P_i becomes idle, the first task in its associated list $\mathcal{L}_i^E$ is either ready or depending on an unfinished phantom task. Thus for static task allocation, all dispatching attempts degenerate to Algorithm 1 using $I = 1$, since no reservable processors exist. Any resource reclaiming must be strictly based on the slack-time of T_u. The result is formalized in Theorem 3.

Theorem 3 *In static task allocation all dispatching attempts degenerate to Algorithm 1, with $I = 1$.*

Proof: First consider the case $\delta \geq u$, i.e. $T_u \in \mathbf{T}^H$. If T_u is released it can be dispatched. Therefore assume T_u is not released. T_u must be depending on an unfinished task not local to the processor represented by a phantom task, or a local phantom task. Thus by definition, T_u is a fan-out task, i.e. $T_u = T_w$. With no reservable processors this is the scenario of Algorithm 4 with $I = 1$, i.e. $\mathbf{\Sigma} = \{T_u, ..., T_{w+I-1}\} = \{T_u\}$, which under the scenario is indistinguishable from Algorithm 1. The case $\delta < u$ implies that T_δ does not exist, i.e. only unstarted soft tasks remain, which by definition have soft deadlines. $\square$

Any dispatching from beyond the window must be based on the slack-time of T_u, i.e. a task T_x in $\mathbf{T}_{>u}$ can be safely started only if $s_x^{max} < \mathcal{S}(T_u)$.

7 Summary

This paper investigated resource reclaiming in hard real-time environments comprising of a hard workload supplemented by soft workloads. Whereas the first set is static and has hard deadlines, the second set is dynamic with soft deadlines. The goal was to allow soft tasks to execute with higher priority, as long as hard task deadlines are guaranteed. This makes it possible to decrease response time of soft tasks.

The scheduling environment is based on low overhead static non-preemptive list scheduling. However, a scan window approach was introduced that expands the scheduling model, allowing more flexibility in executing soft tasks, i.e. the resulting dispatchers allow dynamic task arrival of soft tasks.

An extended task model included phantom tasks as a mechanism for controlling run-time behaviors. Under this model delay enforcement phantom tasks were defined that allow for the controlled delay of selected hard tasks, thus implicitly controlling their release times and thereby forcing slack-time. To implement effective resource reclaiming, the dispatchers allow safe priority inversions and slack-time reclaiming based on sentry tasks specified for each algorithm. With respect to dynamic task to processor allocation, the result was a spectrum of provably stable dispatching algorithms with varying run-time complexities. Dispatching using static task allocation was reduced to a special case of dynamic dispatching in which all efforts degraded to Algorithm 1.

Current research focuses on expanding the task system to include coupled tasks. Such systems are more complex, since in addition to stabilization the issue of run-time infeasibility needs to be addressed. However, more theoretical groundwork is needed, as coupling constraints need to be considered in the development of conditions necessary and sufficient for instability to occur.

References

[1] Butler, R.W., and B.L. DiVito, "Formal Design and Verification of a Reliable Computing Platform

for Real-Time Control", *NASA Tech. Memorandum 104196*, Phase 2 Results, Jan 1992.

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

[3] Davis. R.I., K.W. Tinedell, and A. Burns, "Scheduling Slack Time in Fixed Priority Preemptive Systems", *IEEE Real-Time Systems Symposium*, pp. 222-231, 1993.

[4] Davis. R.I., and A. Wellings, "Dual Priority Scheduling", *IEEE* 16^{th} *Real-Time Systems Symposium*, pp. 100-109, 1995.

[5] Gerhard Fohler, "Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems", *IEEE* 16^{th} *Real-Time Systems Symposium*, pp. 152,161, 1995.

[6] Ghosh, Sunondo, et al., "Enhancing Real-Time Schedules to Tolerate Transient Faults", *Proc. IEEE* 16^{th} *Real-Time Systems Symposium*, pp. 120-129, 1995.

[7] Graham R.L., "Bounds on Multiprocessor Timing Anomalies", *SIAM J. Appl. Math.*, Vol. 17, No. 2, pp. 416-429, Mar 1969.

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

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

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

[11] Kieckhafer R.M, J.S. Deogun and A.W. Krings, "The Performance of Inherently Stable Multiprocessor List Schedulers", Univ. of Nebraska – Lincoln, Dept. of Comp. Sci., Report Series UNL–CSE–92–009, May 1992.

[12] Krings, A.W., and R.M. Kieckhafer, "Inherently Stable Priority List Scheduling in Systems with External Delays ", *Proc. Twenty-Sixth Annual Hawaii International Conference on System Sciences*, Vol. 2, pp. 622-631, 1993.

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

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

[15] Krings, A.W., and M.H. Azadmanesh, *Hybrid Dispatching for Hard Real-Time Systems with Fixed and Sporadic Workloads*, Univ. of Idaho, Dept. of Comp. Sci., Report CS–96–01, Feb. 1996.

[16] Lehoczky, J.P., and Sandra Ramos-Thuel, "An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems", *IEEE Real-Time Systems Symposium*, pp. 110-123, 1992.

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

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

[19] McElvaney, M.C., et. al., "Guaranteeing Task Deadlines for Fault-Tolerant Workloads with Conditional Branches", *Journal of Real-Time Systems*, Vol. 3, No. 3, Sep 1991.

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

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

[22] Shen, C., et al, "Resource Reclaiming in Multiprocessor Real-Time Systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 4, pp. 382-397, Apr 1993.

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

[24] Thuel, S.R., and J.P. Lehoczky, "Algorithms for Scheduling Hard Aperiodic Tasks in Fixed-Priority Systems using Slack Stealing", *IEEE Real-Time Systems Symposium*, pp. 22-33, 1994.