Stability and Performance of List Scheduling With External Process Delays

J.S. Deogun, R.M. Kieckhafer, and A.W. Krings

Department of Computer Science and Engineering
115 Ferguson Hall
University of Nebraska – Lincoln
Lincoln, NE 68588-0115

Keywords: Scheduling stability, General instability conditions, Non-preemptive scheduling, Multiprocessor anomalies, Priority list scheduling, Phantom Tasks.

Abstract

Non-preemptive static priority list scheduling is a simple, low-overhead approach to scheduling precedence-constrained tasks in real-time multiprocessor systems. However, it is vulnerable to anomalous timing behavior caused by variations in task durations. Specifically, *reducing* the duration of one task can *delay* the starting time of another task. This phenomenon, called *Scheduling Instability*, can make it difficult or impossible to guarantee real-time deadlines. Several heuristic solutions to handle scheduling instability have been reported.

This paper addresses three main limitations in the state of the art in schedule stabilization. First, each stabilization technique only applies to a narrowly defined class of systems. To alleviate this constraint, we present an *Extended Scheduling Model* encompassing a wide range of assumptions about the scheduling environment, and address the stability problem under this model. Second, existing stabilization methods are heuristics based on a partial understanding of the causes of instability. We therefore derive a set of *General Instability Conditions* which are both necessary and sufficient for instability to occur. Third, solutions to scheduling instability range from trivial constraints on the run-time dispatcher through complex transformations of the precedence graph. We present *scheduling simulation* results comparing the average performance of several inherently stable run-time dispatchers of widely varying levels of complexity. Results show that for representative real-time workloads, *simple low-overhead dispatchers* perform nearly as well as a complex "minimally stabilized" dispatcher. Thus, complex schedule stabilization methods may be unnecessary, or even detrimental, due to their high computational overhead.

Revised for: Real-Time Systems — June 18, 1996

Contents

List of Figures

List of Tables

1 Introduction

Non-preemptive static priority list scheduling is a relatively simple, low-overhead approach to scheduling computational tasks in real-time multiprocessor systems [Shirazi et al, 1990, Coffman and Gilbert, 1985, Kasahara and Narita, 1984]. With this type of scheduling, all tasks are pre-assigned to a single list with unique fixed priorities. When a processor becomes available, a run-time dispatcher scans the priority list in order of decreasing priority and begins execution of the first unstarted ready task on the idle processor. Individual tasks are part of a *task system* in which a *precedence graph* specifies precedence constraints between tasks. The precedence graph defines a partial order, in which vertices represent tasks and directed edges represent precedence constraints.

Non-preemptive list scheduling has proven desirable in several real-time systems because of its simplicity and low run-time overhead [Butler and Di Vito, 1992, Kieckhafer et al, 1988, Stancovic and Ramamithram, 1987]. However, it is vulnerable to anomalous timing behavior caused by variations in task durations. Specifically, *reducing* the duration of one or more tasks can cause the starting time of another task to be delayed [Hwang and Briggs, 1984]. This phenomenon, called *Scheduling Instability* [Manacher, 1967], can make it difficult or impossible to guarantee real-time deadlines. Unfortunately, variability in task duration is a natural occurrence caused by events such as cache misses, memory refresh cycles, DMA cycle stealing, or bus contention between processors.

Scheduling Stability is defined as the absence of instability; specifically, the schedule is stable if the latest starting time for each task occurs when all tasks run for their maximum duration. *Schedule Stabilization* is then the process of enforcing stability in the schedule. Manacher was among the first to investigate the problem of schedule stabilization in the late sixties [Manacher, 1967]. Recently, there has been renewed interest in this area because of applications to several real-world systems, most notably the Spring Kernel [Stancovic and Ramamithram, 1987] and the Multicomputer Architecture for Fault Tolerance (MAFT) [Kieckhafer et al, 1988]. Several heuristic solutions to scheduling instability have been developed, ranging from trivial constraints on the run-time dispatcher through complex transformations of the precedence graph [Hugue and Stotts, 1991, Manacher, 1967, McElvaney, 1988, Shen et al, 1990]. This paper addresses three main limitations in the state of the art in schedule stabilization.

First, existing approaches to stabilization lack generality, and thus have limited application to real-world systems. They are based on restrictive assumptions about the system environment, e.g. whether task-to-processor allocation is static or dynamic, whether there are communication delays between processors, or whether scheduling is affected by events or delays external to the processors. Thus, each stabilization technique only applies to a narrowly defined class of systems. To alleviate this constraint, we present an *Extended Scheduling Model* encompassing a wide range of assumptions about the system environment in which scheduling is to take place. Stabilization techniques developed under this model are thus applicable to a wider variety of systems. A specific system environment is then just a special case of the Extended Scheduling Model.

Second, existing stabilization methods are heuristics based on a partial understanding of the conditions necessary for instability to occur. We formalize the scheduling instability problem by deriving a set of *General Instability Conditions* necessary and sufficient for instability to occur, leading to a complete understanding of the problem. Based on these conditions, we present a dispatcher algorithm which is "minimally stabilized", i.e. it imposes the minimum constraints necessary to ensure stability.

Third, solutions to scheduling instability range from trivially simple to quite complex. The question is then, how do the average performance of the various stabilization heuristics compare with each other and with a minimally stabilized dispatcher? We present scheduling simulation results covering several run-time dispatchers of widely varying levels of complexity. All dispatchers tested are proven to be stable for all task systems. The simulated workloads include several real-world task systems. The simulation results show that for representative real-time workloads, simple, low overhead run-time dispatchers provide near-optimal average performance, in addition to guaranteed stability. Thus, complex approaches to schedule stabilization are unnecessary, or even detrimental, due to their high computational overhead.

In the next section we present some background material and introduce the scheduling stability problem. An Extended Scheduling Model is developed in Section 3. In Section 4, general instability conditions necessary and sufficient for instability to occur are derived. It is also demonstrated how these results can be applied to systems with different scheduling environments. In Section 5 the concept of *scan window dispatching* as a low-overhead approach to stabilization is introduced. In Section 6 we present simulation results showing that the simplest scan window dispatchers yield performance comparable to the most complex stabilization methods.

2 Background and Motivation

As stated above, non-preemptive priority list scheduling is a simple low-overhead approach to scheduling real time-tasks in multiprocessor systems. However, it is also vulnerable to several *multiprocessor anomalies* [Hwang and Briggs, 1984], which describe counter-intuitive or unstable scheduling behavior caused by variations in one or more system parameters. For example, the starting time of a particular task can be *delayed* by (1) increasing the number of processors, (2) relaxing one or more precedence constraints, (3) reducing the duration of one or more tasks.

This paper addresses the third anomaly, instability due to variations in task durations. The motivation for focusing on this anomaly is that it is the only one of the three likely to occur *as a result of run-time phenomena*. For example, one generally does not alter the precedence relationships between tasks in a running system. Similarly, one does not insert a processor into a running system without having previously examined the impact on scheduling. However, task duration can vary from one execution of a task to the next. These variations are stochastic, and generally unavoidable. A task may even have a minimum duration of zero (e.g. a task may sometimes be intentionally omitted based on run-time conditions). Simulating the schedule with all possible variations in all tasks is usually an intractable problem. It is therefore necessary to prove *a priori* that a schedule will remain stable under all permissible variations in task durations.

2.1 Definitions

For the purposes of this study, a *Task* is defined as an indivisible block of code which must be executed in its entirety on a single processor. Since we are dealing with real-time systems, it is assumed that the *maximum duration* of any task is known *a-priori*. In addition, a non-zero *minimum duration* may also be known. The actual duration of a particular execution of the task may vary randomly between these minimum and maximum limits.

A *Scenario* describes the schedule obtained with a particular set of task durations. The *Standard Scenario* is the scenario in which all tasks execute for their specified maximum durations. A Gantt chart depicting the standard scenario is called the *Standard Gantt Chart* (SGC) [Manacher, 1967]. In a *Non-Standard Scenario* at least one task runs for less than its maximum duration, producing a *Non-Standard Gantt Chart* (NGC).

Whenever a processor becomes idle, 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 system design time to assign task priorities and find a feasible schedule for the standard scenario.

A system is *stable* with respect to the SGC if there exists no scenario under which the completion time of any task on any NGC exceeds its completion time on the SGC[1] [Manacher, 1967]. If the SGC is feasible, then stability assures that *all* NGC's are feasible. Conversely, if the system is not stable, then one or more infeasible NGC's *may* exist. 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 Example of Instability

Figure 1 shows the precedence graph for an example nine-task system, with the maximum duration of each task listed next to its vertex. For this example, we assume that task priorities follow task numbering, with the lower numbered tasks receiving higher preference.

Insert Figure 1 here

Figure 2 illustrates an instance of scheduling instability. Gantt chart 2(a) is the dual-processor SGC for the task system of Figure 1. Gantt chart 2(b) is the NGC obtained when task T_3 is shortened by an arbitrarily small value $\epsilon > 0$. On the NGC, the shortened T_3 finished before T_2 causing T_6 to be ready before T_5. T_6 was thus able to "usurp" the processor on which T_5 was executed on the SGC. As a result, both T_5 and its child T_7 started later on the NGC than they did on the SGC.

Insert Figure 2 here

In Figure 1, priorities are defined in order of increasing start times on the SGC. A priority list with this property is called a *projective list*. Given any priority list and its corresponding SGC, a projective list can be obtained by scanning the SGC and reassigning the task priorities in order of increasing start time. There is no guarantee that a given scheduling algorithm will produce a projective list. However, it has been shown that in the standard scenario a projective list will always identically reproduce the SGC from which it was derived [Manacher, 1967]. This property allows us to convert any original priority list into a projective list without compromising the feasibility of the standard scenario.

In an early landmark study on instability, [Manacher, 1967] Manacher showed that when

[1] Manacher originally referred to this condition as "strongly stable".

scheduling tasks from a projective list, three conditions are necessary (though not always sufficient) to permit instability.

1. There is a *fork* in the precedence graph, i.e. there exists a task with more than one child [Manacher, 1967, Lemma 7]. In Figure 1, tasks T_1 and T_2 are both *forking tasks*.

2. At least two descendents of a forking task were executed in parallel on the SGC [Manacher, 1967, Lemma 7]. Figure 2(a) shows tasks T_4 and T_5 (both descended from T_2) running in parallel on the SGC.

3. A *Priority Inversion* occurs on an NGC, i.e. a lower priority task starts running before some higher priority task [Manacher, 1967, Corollary to Lemma 3]. The NGC of Figure 2(b) shows T_6 starting before T_4 and T_5. Hence, priority inversion has occurred.

2.3 Existing Stabilization Methods

Currently, several alternatives exist for stabilizing a task schedule.

Preemptive Scheduling – The instability problem occurs only in non-preemptive systems. If a higher priority task is allowed to preempt a lower priority task, then priority inversion is rectified at the time the higher priority task becomes ready to run. However, preemptive scheduling does have disadvantages which can make non-preemptive scheduling more attractive for certain applications.

- Preemptive scheduling can exhibit dramatically higher run-time overhead than non-preemptive scheduling [Chen et al, 1988, Krueger and Livny, 1988, Jeffay et al, 1991, Krueger and Chawla, 1991].

- In systems with cache memory, preemption can adversely affect task durations. When a previously preempted task is resumed its *cache image* is likely to have been overwritten, and must be rebuilt. Thus, the total duration of the task may be significantly longer than if it had been allowed to run to completion uninterrupted.

- It has been observed that "The overhead of preemptive algorithms is more difficult to characterize and predict"[Jeffay et al, 1991]. Because of this difficulty, at least one draft regulatory standard for safety-critical software prohibits the use of "interrupts," or "Multi-processing on a single processor," [Min. of Defence, 1989].

Since there are situations in which non-preemptive scheduling is the more attractive choice, it has been the subject of considerable research [Bettani et al, 1991, Hugue and Stotts, 1991, Hwang et al, 1989, Jeffay et al, 1991, McElvaney, 1988, Shirazi et al, 1990, Verhoosel et al, 1991]. In addition, several existing fault-tolerant real-time systems have chosen to use non-preemptive scheduling due to its predictability and low overhead [Butler and Di Vito, 1992, Kieckhafer et al, 1988, Pygott and Kershaw, 1990, Stancovic and Ramamithram, 1987]. Therefore, any stabilization method requiring preemption is considered beyond the scope of this study.

Fixed Starting Times – If each task is constrained to start at a fixed time, then priority inversion can not occur and stability is ensured. This constraint reduces the dispatcher to a static frame-based dispatcher. However, in some systems [Bettani et al, 1991, Jeffay et al, 1991, Kieckhafer, 1989], a task may have one of two levels of criticality: *Hard* tasks are critical to system safety and must complete execution by their specified deadlines, while *Soft* tasks are not critical and may be executed as time permits. If necessary, a soft task may even be aborted during execution with no adverse effect on system safety.

If the SGC is feasible for both hard and soft tasks, then a Fixed Starting Time dispatcher avoids the stability problem entirely. However, if the SGC is feasible for the hard tasks only, then some accomodation must be made for the soft tasks. One solution is to fix the starting times of only the hard tasks, allowing the soft tasks to exploit unused processor time between hard tasks. However, a soft task may be started only if it is *slack time safe* with respect to the first unstarted hard task, i.e. if the soft task is guaranteed to be finished before the starting time of the hard task. This constraint could cause fragmentation of processor time, and yield poor processor utilization.

This approach requires the run-time dispatcher to treat hard and soft tasks differently. It also requires a timer interrupt at the starting time of every hard task. These features add to dispatcher complexity and nullify the Fixed Starting Times dispatcher's primary advantage – simplicity.

Fixed Starting Sequence – Rather than fixing task starting times, this approach constrains 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.

With this approach, the highest priority unstarted task will start as soon as all of its parents are finished. This approach can reduce the processor time fragmentation which occurs with

Fixed Starting Times. Specifically, if actual task durations are less than the maximum durations, then the schedule will tend to contract toward the left hand side of the Gantt chart, leaving a contiguous block of processor time on the right hand side of the Gantt chart for soft tasks.

This approach allows more flexibility than the Fixed Starting Times approach, but still prevents priority inversion. A Fixed Starting Sequence dispatcher treats both hard and soft tasks the same, even if slack time reclaiming is employed. Thus, it can be simpler than a Fixed Starting Times dispatcher.

Manacher's Algorithm – A landmark stabilization algorithm was developed by Manacher [Manacher, 1967]. It was intended to permit greater flexibility than the previous two approaches. Manacher's stabilization algorithm alters the original precedence graph to produce a graph which is inherently stable. It first identifies all *co-running* task pairs on the SGC and their *nearest common ancestors*, if any. It then inserts new edges into the precedence graph between the nearest common ancestors and any tasks beyond the co-running pair which could act as usurpers. In the example of Figure 1, this algorithm installs a new edge from T_2 to T_6. This edge prohibits the priority inversion of T_6 about T_5, thus preventing the instability in Figure 2(b).

Manacher's stabilization algorithm permits tasks to execute as soon as they are ready. Thus, as in the Fixed Starting Sequence approach, there can be an accumulation of leftover processor time at the end of the schedule, allowing uniform treatment of hard and soft tasks. However, because Manacher's algorithm allows some "benign" priority inversions to occur, it offers the dispatcher greater flexibility than the Fixed Starting Sequence approach.

Manacher's algorithm has four main limitations. First, the algorithm applies only to systems with *dynamic* task-to-processor allocation, i.e. it assumes that all tasks are kept in a single priority list and are assigned to specific processors at run-time. Second, it does not account for the impact of communication overhead or other delays imposed from outside the processors (except for a restricted form of delayed arrival). Third, the algorithm performs *a priori* stabilization of the precedence graph based on the worst case conditions which *could* occur in *any* non-standard scenario. Thus, the algorithm over-constrains the dispatcher in scenarios which are not worst-case. Fourth, the addition of more edges to the precedence graph increases the run-time overhead required to resolve parent-child dependencies.

Shen's Dispatcher – In recent years, there has been a renewal of interest in schedule stabilization in the context of specific systems. One method was developed by Shen *et al.* for use in the Spring Kernel [Shen et al, 1990] in the context of resource reclaiming. This approach permits priority inversion to occur only under restricted conditions. Specifically, if task T_i started before task T_j on the SGC, then on an NGC T_j is allowed to start before T_i only if T_i and T_j ran concurrently on the SGC. This situation is called inversion without "passing"[Shen et al, 1990]. This constraint is more restrictive than Manacher's algorithm because it does not consider a fork as a prerequisite to instability. However, the approach imposes very little additional overhead on dispatcher.

Hugue's Algorithm – Another system-specific stabilization method was developed by Hugue [Hugue and Stotts, 1991, McElvaney, 1988] for the Multicomputer Architecture for Fault-Tolerance (MAFT) [Kieckhafer et al, 1988]. In particular, MAFT employs *static* task-to-processor allocation, i.e. tasks are assigned to processors *a priori*. MAFT also has considerable communication overhead associated with the dispatcher itself. Hence Manacher's algorithm does not apply.

Hugue's algorithm is a variation on Manacher's algorithm customized to the MAFT environment. However, the algorithm has some limitations. Although MAFT is capable of multiprocessing, Hugue's algorithm addresses only "unischedule" workloads, wherein multiple processors are used *only* for modular redundancy, not for parallel processing. Previous research does not address the problem of stabilizing true multiprocessor workloads in MAFT-like systems.

2.4 Objectives

Previous studies of Scheduling Stability have generally been limited to tightly defined application-specific scheduling environments. One problem with applying earlier results to specific systems is that the basic assumptions behind the solution method are usually not valid in the actual system at hand. Therefore, different *ad hoc* stabilization methods have been developed for different systems [Hugue and Stotts, 1991, McElvaney, 1988, Shen et al, 1990]. By contrast, this paper casts the stability problem in a general form that is application-independent. It was motivated by three basic objectives:

1. Develop a general *Extended Scheduling Model* to facilitate the investigation of stability in a variety of different scheduling environments. This model should be general enough to address the impact of processes and events that occur outside the system processors.

2. Derive a complete formalization of the instability problem permitting the development of a minimally stabilized dispatcher as well as lower cost heuristics. The focus of this formalism is a complete set of *General Instability Conditions* that are both necessary and sufficient to induce instability in any system covered by the Extended Scheduling Model.

3. Assess the relative performance of both simple and complex stabilization methods with respect to a minimally stabilized dispatcher. We can then determine whether it is worth the effort to employ complex dispatchers or whether simple dispatchers can approach the performance of the minimally stabilized dispatcher.

3 Extended Scheduling Model

The first objective of this study is to develop a general *Extended Scheduling Model* for describing the scheduling environment of a system. New stabilization methods developed under this model would be applicable to a variety of non-preemptive real-time systems. It would also allow existing stabilization methods to be analyzed under different sets of assumptions about the system.

3.1 General Requirements

Hard and Soft Tasks – It is not uncommon for a task system to contain both hard and soft tasks [Bettani et al, 1991, Jeffay et al, 1991, Kieckhafer, 1989]. The main scheduling objective is then to execute as many soft tasks as practicable within the constraint of guaranteeing the deadlines of all hard tasks. Support for this type of two-tiered scheduling has even been incorproated into the hardware of some real-time systems. For example, the VIPER–II microprocessor [Pygott and Kershaw, 1990] functions under the assumption that real-time tasks will be executed in a cyclic manner. For reasons of verifiability, the VIPER-II does not support preemptive scheduling. However, the architecture provides an end-of-cycle signal which aborts the currently running task and resets the dispatcher to the beginning of the task list. It is thus critical that all hard tasks be completed prior to the end-of-cycle. Similarly, the Multicomputer Architecture for Fault-Tolerance (MAFT) allows the workload to be partitioned into "mandatory" tasks and "optional" tasks [Kieckhafer, 1989]. The schedule is then constructed to ensure that all mandatory tasks are completed on time while optional tasks are executed as time permits.

Task-to-Processor Allocation – The allocation of tasks to specific processors can take two forms: *dynamic allocation*, in which tasks are allocated at run-time, and *static allocation*, in which each task is permanently assigned to a specific processor prior to run-time. With static allocation, the priority list is partitioned into separate priority lists for the individual processors. It can be shown that stabilization methods developed for one form of allocation do not necessarily work under the other form. To be completely general, the extended scheduling model must be able to represent both static and dynamic allocation.

External Delays – The release of a task may be delayed by an event or process external to the processors. For example, external timers may be used to enforce the delayed release of a task, or to implement cyclic scheduling. A task may also depend on the completion of a peripheral process or event such as an input/output operation. Furthermore, in many systems, communication delays between processors are not negligible. In such systems, a task is not released immediately after the completion of its last parent. Rather, release is delayed by interprocessor communication. Most stabilization methods do not address the impact of such external delays.

3.2 Phantom Tasks

The Extended Scheduling Model must be able to represent processes and delays occurring outside the system processors. In addition, the method used to represent these processes must be suitable for theoretical studies as well as scheduling simulations. We propose to *extend* the task system by introducing a construct called a *Phantom Task*. A Phantom Task is simply a task which takes time, but does not occupy a system processor. This model is equivalent to assuming that phantom tasks are executed on an infinite set of *phantom processors*, whereas real tasks are executed on the finite set of real system processors.

Phantom tasks are fully integrated into the precedence graph along with real tasks. Also, as with real tasks, the durations of phantom tasks may vary within pre-defined limits.

Example: Figure 3 shows a precedence graph containing two phantom tasks (T_0 and T_4) along with real tasks ($T_1 \ldots T_3$) and ($T_5 \ldots T_9$). Phantom task T_0 represents the time-delayed release of task T_3. Similarly, phantom task T_4 represents a delay between the completion of T_1 and the release of its child, T_6. T_4 could represent any of a number of real-world phenomena, such as an interprocessor communication delay or a peripheral process initiated by T_1 whose results are required by T_6.

Insert Figure 3 here

The incorporation of phantom tasks into a precedence graph reveals the impact of external delays on stability. Figure 4(a) shows the dual-processor SGC for the precedence graph of Figure 3 (phantom tasks are not shown on the Gantt chart because they do not occupy processors). Figure 4(b) shows the NGC obtained when task T_5 is shortened by an arbitrarily small value ϵ. This NGC shows tasks T_6 and T_8 starting later than on the SGC even though *there is no fork in the precedence graph.* The presence of phantom tasks thus violates the assumptions under which the necessary conditions listed in Subsection 2.2 are valid. Therefore, external delays can induce instability in a system which would otherwise be stable.

Insert Figure 4 here

4 General Instability Conditions

Previous studies of instability have limited application to real-world systems because the investigations were based on modeling the problem in restricted scheduling environments. In this section, we derive necessary and sufficient conditions for stability in the Extended Scheduling Model.

4.1 Definitions and Notation

We first introduce some useful definitions and notational conventions. In general, scalar values are indicated by symbols in normal font, while *sets* of tasks are indicated by bold-face font. Subscripts identify a specific task or set of tasks to which a symbol is relevant. Superscripts indicate a specific instance of a variable. In particular, the superscripts min, max, and std indicate minimum value, maximum value, and value in the standard scenario, respectively. A symbol with *no* superscript refers to a general non-standard scenario, or a scenario currently under consideration.

4.1.1 Time Variables

There are several important time variables associated with each task T_i.

c_i = Computation time (or duration) of task T_i. By definition, $c_i^{std} = c_i^{max}$. Any task-switching overhead incurred by the same processor that executed T_i is included in c_i. Thus, c_i, including overhead is contained within $[\, c_i^{min},\, c_i^{max}\,]$.

s_i = Starting time of task T_i. When addressing stability, the deadline for starting task T_i is s_i^{std}, its starting time on the SGC.

f_i = Finishing time of task T_i. On the SGC, all tasks run for their maximum durations. Thus the condition $s_i \leq s_i^{std}$ ensures that $f_i \leq f_i^{std}$.

r_i = Release time of task T_i. It is the time at which task T_i becomes *ready* for execution. Generally, release time is the time at which the last parent of task T_i finishes execution. If task T_i has no real or phantom parents, then $r_i = 0$.

4.1.2 Dispatcher Sequencing

For dynamic task-to-processor allocation, the run-time task dispatcher is modeled as a non-sharable procedure guarded by a mutual exclusion protocol. Therefore, only one processor can enter the dispatcher at any given time. A processor requests entry to the dispatcher upon completion of its current task. Once it has entered the dispatcher it performs four steps: (1) update the priority list, (2) launch any newly released phantom tasks, (3) select a new real task for itself, (4) exit the dispatcher. All four steps occur as an atomic action. If it finds no real tasks ready to run, the processor exits the dispatcher and immediately issues another request to enter.

We assume that requests to enter the dispatcher are serviced on a strict first-come-first-served basis. However, in the rare event that two requests from different processors arrive simultaneously (e.g. within the same clock cycle), a consistent tie-breaking scheme is needed. We follow Manacher's convention that simultaneous requests to enter the dispatcher are serviced in order of increasing processor number [Manacher, 1967].

The effect of this protocol is to impose a complete ordering on events which appear to be simultaneous on the Gantt chart. For example, assume that at some time processor 1 completes task T_i, while simultaneously processor 2 completes task T_j. Further assume that both requests to enter the dispatcher arrive simultaneously. Then, processor 1 (being the lower numbered processor) will enter the dispatcher first, imposing the ordering $f_i < f_j$. All subsequent events will then proceed as if task T_i actually did finish earlier than T_j.

4.1.3 Task Sets

It is assumed in the remainder of this paper that the priority list is a projective list. As shown in Subsection 2.2, this assumption does not interfere with the initial construction of the SGC. In the context of a projective list, several task *sets* have been defined relative to task T_i [Manacher, 1967].

$\mathbf{T}_{<i}$ = The set of all tasks, real or phantom, which started before task T_i on the SGC. By the definition of a projective list, $\mathbf{T}_{<i}$ contains all tasks T_j such that $j < i$.

$\mathbf{T}_{\leq i}$ = $\{\mathbf{T}_{<i} \bigcup \{T_i\}\}$.

Sets $\mathbf{T}_{>i}$ and $\mathbf{T}_{\geq i}$ are defined similarly.

4.1.4 Task Status

Consider a task set $\mathbf{T}_k$. For an M-processor system, the status of the task set $\mathbf{T}_k$ is reflected in the number of real tasks from $\mathbf{T}_k$ which are currently running or ready to run.

$M_k(t)$ = The number of *real* tasks in set $\mathbf{T}_k$ running at time t, or the number of processors occupied by set $\mathbf{T}_k$ at time t.

$W_k(t)$ = The number of *real* tasks in set $\mathbf{T}_k$ which are ready but waiting for a processor at time t.

$R_k(t)$ = $M_k(t) + W_k(t)$, or the total number of *real* tasks in set $\mathbf{T}_k$ which are either ready or running at time t.

Accordingly, $W_{\mathcal{T}}(t)$, $M_{\mathcal{T}}(t)$, and $R_{\mathcal{T}}(t)$, refer to the entire workload $\mathcal{T} = \{T_1, \ldots, T_N\}$.

4.1.5 Logical Fork

A *logical fork* is any event which causes an increase in $R_{\mathcal{T}}(t)$, the number of real tasks which are either ready or running at time t, i.e. any event which causes:

$$\lim_{\delta \to 0} R_{\mathcal{T}}(t - \delta) < R_{\mathcal{T}}(t).$$

Thus, values for $M_k(t)$, $W_k(t)$ and $R_k(t)$ reflect conditions immediately after the fork.

In the context of the Extended Scheduling Model, there are two types of logical fork: *real forks*, and *phantom forks*. A real fork occurs when the completion ofa real task, T_ϕ, causes *more than one* real task to be released at its finishing time f_ϕ. Task T_ϕ is called the *forking* *task*. Clearly, any real task with more than one real child is potentially a forking task, whereas a real task with only one real child can *not* be a forking task. A Phantom Fork occurs when the completion of a phantom task T_ϕ causes *one or more* real tasks to be released at its finishing time f_ϕ. Thus, any phantom task with at least one real child is potentially a forking task. The impact on the system is the same for both the real and phantom forks because both cause $R_{\mathcal{T}}(t)$ to increase at $t = f_\phi$.

Given a set of tasks $\mathbf{T}_k$, we say a forking task T_ϕ causes a fork *into* $\mathbf{T}_k$ if it causes $R_k(t)$ to increase at f_ϕ. If T_ϕ is a real task, then it has *at least two* real children in $\mathbf{T}_k$. Similarly, if T_ϕ is a phantom task, then it has *at least one* real child in $\mathbf{T}_k$.

Example: The precedence graph of Figure 1 contains only real tasks. In this graph, both T_1 and T_2 are potential forking tasks because each has more than one real child. Task T_2 is a forking task in all scenarios because it always releases two children, T_4, and T_5, every time it finishes execution. However, T_1 does not cause a fork in all scenarios. If, on some NGC, T_1 finishes later than T_0, then at time f_1, T_2 and T_3 are both released simultaneously, increasing $R_{\mathcal{T}}$ from 1 to 2. However, if T_0 finishes later than T_1, then no fork occurs because T_2 and T_3 are not released at the same time.

If Task T_1 causes a fork, then it forks into set $\mathbf{T}_{<5}$ because it has two real children (T_2 and T_3) which are both elements of $\mathbf{T}_{<5}$. By contrast T_2 does not fork into $\mathbf{T}_{<5}$ because only one of its children (T_4) is an element of $\mathbf{T}_{<5}$.

The precedence graph of Figure 3 contains no real forking tasks (Task T_1 is not a forking task because it has only one *real* child, T_3). However, phantom tasks T_0 and T_4 are both potentially forking phantom tasks because each has at least one *real* child. Phantom task T_4 is always a phantom fork because its completion always releases real task T_6. However, phantom task T_0 releases task T_3 *only* if it finishes *after* T_1. If T_0 finishes before T_1, then no phantom fork occurs.

4.1.6 Unstable Tasks

In any non-standard scenario, several tasks may start late. However, the lateness of one task may be the result of a chain reaction caused by the lateness of some lower numbered task. We define an *unstable task* as the task which starts this chain reaction. In a given scenario, task T_v is unstable only if it is the *lowest numbered task* to start late, i.e. if: (1) $s_v > s_v^{std}$, and (2) $s_i \leq s_i^{std}\ \forall\ T_i \in \mathbf{T}_{<v}$. Assumption (2) is called the *On-Time Hypothesis*. T_v is then *unstable in that scenario*, and the scenario is said to be *unstable at* T_v. Task T_v is *vulnerable*

to instability if there exists at least one scenario which is unstable at T_v. Conversely, an invulnerable task is guaranteed to be stable in all possible scenarios.

Example: In the NGC of Figure 2(b), two tasks, T_5 and T_7, started late. All tasks in $\mathbf{T}_{<5}$ were on-time. Thus, T_5 was the lowest-numbered late task, and therefore unstable. By contrast, T_7 was late because $T_5 \in \mathbf{T}_{<7}$ was late. Therefore, T_7 was not unstable.

4.1.7 Usurper Task

Manacher has shown [Manacher, 1967] that a task, T_v, can be unstable only if a priority inversion occurs *about* T_v, i.e. $\exists\ T_x \in \mathbf{T}_{>v}$ such that $s_x < s_v$. Instability at T_v occurs because T_x "usurped" a processor which T_v requires to start on time. T_x is thus called a *Usurper Task*. By this definition, only a *real* task can be a usurper task because phantom tasks do not utilize processors.

4.2 Task Starting Criteria

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, i.e. $W_{<v}(t) = 0$,

[S3] If T_v is real, then a processor is available (idle) at t, i.e. $M_{\mathcal{T}}(t) < M$.

If a task T_v is unstable, then one or more of the above conditions was not true for T_v until after s_v^{std}. As shown by following lemma, the on-time hypothesis guarantees that [S1] and [S2] are true for T_v no later than s_v^{std}.

Lemma 1: If all real and phantom tasks $T_i \in \mathbf{T}_{<v}$ are on-time (the on-time hypothesis), then [S1] and [S2] become true for task T_v no later than s_v^{std}.

Proof: T_v started precisely at s_v^{std} on the SGC. Therefore, all predecessors of T_v must have been completed no later than s_v^{std}. By the on-time hypothesis all predecessors of T_v are still completed no later than s_v^{std} on the NGC. Thus, at s_v^{std}, T_v is ready to run [S1]. Also, by the on-time hypothesis, all $T_i \in \mathbf{T}_{<v}$ have started no later than time s_v^{std}. Thus, $W_{<v}(s_v^{std}) = 0$, so T_v is the *first* ready task on the list [S2]. $\square$

Corollary L1.1: A phantom task can never be unstable.

Proof: According to Lemma 1, starting criteria [S1] and [S2] will be true no later than s_v^{std} for any potentially unstable task T_v. If T_v is a phantom task, then starting criterion [S3] is not required. Therefore, T_v will start at or before s_v^{std} unless some task in $\mathbf{T}_{<v}$ started late. Therefore, a phantom task can never be unstable. $\square$

4.3 Derivation of Instability Conditions

By Lemma 1, a vulnerable task is one that could be late even if [S1] and [S2] are satisfied at s_v^{std}. Thus, determining the stability of a real task T_v consists of defining *necessary and sufficient conditions under which [S3] is not true* for T_v by s_v^{std}.

4.3.1 Priority Inversion

Priority inversion has been previously identified as a key factor in scheduling instability [Manacher, 1967]. The following Lemma uses the three task starting criteria to identify features of a scenario which permit an inversion to occur.

Lemma 2: Let T_v and T_x be two real tasks such that $v < x$. If, in some scenario, the condition $(s_x < s_v)$ is true, then the following conditions must also be true:

1. $s_x < r_v$,
2. $W_{\leq v}(s_x) = 0$
3. $M_{\leq v}(s_x) < M$

Proof: From $v < x$ and the definition of a projective list it follows that $T_v \in \mathbf{T}_{<x}$, so that $\mathbf{T}_{\leq v} \subseteq \mathbf{T}_{<x}$. Assertion 1 follows from the fact that $s_x \geq r_v$ implies that [S2] can not be satisfied for T_x until T_v has started, because $T_v \in \mathbf{T}_{<x}$, which is a contradiction. For Assertion 2, it is clear that $W_{\leq v}(s_x) = 0$ is necessary because $\mathbf{T}_{\leq v} \subseteq \mathbf{T}_{<x}$, and because starting criterion [S2] requires $W_{<x}(s_x) = 0$. Assertion 3 follows by observing that $M_{\leq v}(t) \subset M_{\mathcal{T}}$, and therefore [S3] requires $M_{\leq v}(s_x) < M$ at s_x. $\square$

The following theorem restates that priority inversion is necessary for instability to occur. While this result is not new in the context of real tasks [Manacher, 1967, Lemma 2], it is re-derived here to verify that the presence of phantom tasks does not negate the necessity of this condition.

Theorem 1: Priority Inversion about real task T_v is necessary for T_v to be unstable, i.e. T_v can be unstable only if there exists a real task $T_x \in \mathbf{T}_{>v}$ such that $s_x < s_v$.

Proof: Define $\mathbf{C}_{<v}^{std}$ as the subset of all *real* tasks in $\mathbf{T}_{<v}$ which were still running at s_v^{std} on the SGC (i.e. $\mathbf{C}_{<v}^{std} = \left\{real\ T_c \in \mathbf{T}_{<v} : s_c^{std} < s_v^{std} < f_c^{std}\right\}$). Then, all tasks in $\mathbf{T}_{<v} \backslash \mathbf{C}_{<v}^{std}$ were completed at or before s_v^{std} on the SGC. The on-time hypothesis requires this condition to be true on the NGC also.

Assume there is no priority inversion about T_v in the current scenario, i.e. there is *no* real task $T_x \in \mathbf{T}_{>v}$ such that $s_x < s_v$. Then, by the on-time hypothesis, the only real tasks which can be running at s_v^{std} must be members of $\mathbf{C}_{<v}^{std}$. These tasks all ran concurrently with T_v on the SGC. Thus, given M processors, we know that $\left|\mathbf{C}_{<v}^{std}\right| \leq M-1$. Therefore, (assuming T_v did not start prior to s_v^{std}) there is at least one processor available for T_v at s_v^{std}. Hence [S3] is satisfied at s_v^{std}. According to Lemma 1, the on-time hypothesis guarantees that [S1] and [S2] are satisfied for T_v at or before s_v^{std}. Since [S1], [S2], and [S3] are all satisfied no later than s_v^{std}, $s_v \leq s_v^{std}$. Thus, T_v can not be the *first* task to start late in this scenario. $\square$

4.3.2 Logical Fork

The existence of a real fork in the precedence graph was previously shown to be a prerequisite to instability in the absence of phantom tasks [Manacher, 1967]. Here, that result is extended to show that phantom and real forks can be treated identically. Moreover, the problem of multiple usurper tasks is also addressed. It is also shown that any scenario unstable at task T_v must follow a specific sequence of the events prior to s_v^{std}.

Priority inversion is possible if one or more tasks in $\mathbf{T}_{<v}$ finishes before its standard finishing time. Consequently, all conditions of Lemma 2 become true for one or more $T_x \in \mathbf{T}_{>v}$ on the NGC, when they were not true on the SGC. This behavior depends upon:

T_Δ = The latest starting Usurper task. Thus s_Δ, the starting time of T_Δ, satisfies: $s_\Delta \geq s_x\ \forall\ T_x \in \mathbf{T}_{>v}$ for which $s_x < r_v$.

$\mathbf{L}$ = $\{T_l \in \mathbf{T}_{\leq v} : f_l > s_\Delta\}$. The *Leftover* set $\mathbf{L}$ contains all tasks, real and phantom, numbered $\leq v$ which are not yet finished at s_Δ. Thus, T_v is the highest numbered task in $\mathbf{L}$.

$\mathbf{X}$ $= \{T_x \in \mathbf{T}_{>v} : s_x < f_\phi < f_x\}$. Task set $\mathbf{X}$ is the set of usurper tasks with respect to f_ϕ, the time of the last logical fork to occur at or before r_v. Note that $\mathbf{X}$ may or may not contain task T_Δ, depending on whether or not $f_\Delta \leq f_\phi$.

Real tasks in $\mathbf{L}$ (including T_v) contend with each other for processors, given that tasks from $\mathbf{X}$ have already usurped one or more processors. The following theorem establishes the relationships between s_Δ, f_ϕ, and r_v which must occur in any scenario which is unstable at T_v.

Theorem 2: 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 $s_\Delta < f_\phi \leq r_v$.

Proof: From Theorem 1, priority inversion is necessary for instability. Therefore, assume that a priority inversion has occurred so that $s_\Delta < r_v$, but that there is no fork in the interval $(s_\Delta, r_v]$. For each task $T_l \in \mathbf{L}$ for which $s_\Delta < s_l$, Lemma 2 implies $s_\Delta < r_l$. Thus, by the no-fork assumption, any task in $\mathbf{L}$ which was not already running at s_Δ must be a descendent of one or more tasks in $\mathbf{L}$ which *were* running at s_Δ. Therefore, $R_{\leq v}(t) \leq R_{\leq v}(s_\Delta)$ for all t in $(s_\Delta, r_v]$. Thus, $W_{\leq v}(t) = 0$ for the *entire* interval $(s_\Delta, r_v]$. Let $T_p \in \mathbf{L}$ be the last parent of T_v to finish, so that $f_p = r_v$. Then:

1. By the on-time hypothesis: $f_p \leq f_p^{std}$. Thus $r_v \leq r_v^{std} \Rightarrow$ condition [S1] is true at r_v,
2. By the no-fork assumption: T_v is the *only* child of T_p. Since $W_{\leq v}(r_v) = 0$, T_v is at the head of the list as soon as it is released $\Rightarrow$ condition [S2] is true at r_v,
3. By the no-fork hypothesis, T_p must be a real task. Since $r_v = f_p$, the processor occupied by T_p becomes available at $r_v \Rightarrow$ condition [S3] is true at r_v.

Therefore, T_v starts at time r_v. Moreover, T_v is on-time because $s_v = r_v \leq r_v^{std} \leq s_v^{std}$. $\square$

4.3.3 Processor Contention

Theorems 1 and 2 define necessary conditions under which instability *may* occur. The following theorem defines a third condition under which instability *will* occur. It is based on an anaylsis of processor contention between real tasks in task sets $\mathbf{L}$ and $\mathbf{X}$.

Theorem 3: Assume that a priority inversion and logical fork into $\mathbf{L}$ have occurred such that $s_\Delta < f_\phi \leq r_v$, where f_ϕ is the time of the *last* fork into $\mathbf{L}$ to occur at or before r_v. Further assume that all $T_i \in \mathbf{T}_{<v}$ are on time (*on-time* hypothesis). Then, T_v is late *if and only if* $W_L(t) > 0$ for the entire interval $[f_\phi, s_v^{std}]$.

Proof: By the on-time hypothesis, $r_v \leq r_v^{std}$ so that [S1] is true for T_v no later than r_v^{std}. If $f_\phi < r_v$, then there are no more forks in the sub-interval $[f_\phi, r_v)$ because f_ϕ is defined as the time of the *last* fork before r_v. Thus, each task in **L** which has not started by f_ϕ must be the sole child of another real task in **L**. Therefore, if $W_L(t)$ is zero at any time during this interval, it *remains* zero at least until r_v. This sub-interval establishes the initial value of $W_L(t)$ for the sub-interval $[r_v, s_v^{std}]$. In sub-interval $[r_v, s_v^{std}]$, T_v is ready. T_v is also an element of **L**, so if $W_L(t) = 0$ at any instant in the interval, then T_v has already started. Conversely: if $W_L(t) > 0$ throughout $[r_v, s_v^{std}]$, then T_v could not start until after s_v^{std}. □

4.3.4 Instability Conditions

In a given scenario, the stability of task T_v is determined by the interactions of task sets **L** and **X**. Furthermore, T_v is by definition the largest numbered real task in **L**. Theorems 1 through 3 have shown that task T_v is vulnerable to instability if and only if there exists a scenario in which the following three *General Instability Conditions* are all true:

[I1] $\mathbf{X} \neq \emptyset$. Thus, 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* **L** such that $(s_x < f_\phi \leq r_v) \; \forall \; T_x \in \mathbf{X}$, where f_ϕ is the time of the last logical fork into **L** to occur at or before r_v,

[I3] $W_L(t) > 0, \; \forall \; t \in [f_\phi, s_v^{std}]$, i.e. there exists no time t in the interval $[f_\phi, s_v^{std}]$ at which $W_L(t) = 0$.

4.4 Applications

We have stated that the motivation for developing the Extended Scheduling Model was to provide a uniform basis for examining stability in a wide variety of scheduling environments. In this section, we show how several known scheduling environmments can be modeled as special cases of the Extended Scheduling Model. Then, the General Instability Conditions ([I1], [I2], and [I3]) can be applied to examine stability in each of these environments. In some cases, proof of stability or instability turns out to be trivial. In others, it is shown that previous, complex stabilization algorithms reduce to extremely simple algorithms.

4.4.1 Inherently Stable Precedence Graphs

General Instability Condition [I2] shows that if a precedence graph contains no logical forks, then it is inherently stable. For example, considerable work has been done on scheduling sets of *independent tasks*, i.e. $G = \{V, E\}$, where: $E = \{\emptyset\}$ (e.g. [Coffman and Gilbert, 1985, Liestman and Campbell, 1986, Leung, 1982, Ramamritham et al, 1989]).

As long as all tasks are real, such systems are inherently stable. However, recall that a phantom task with one or more real children is potentially a phantom fork. Thus, the presence of even one such phantom task makes the system potentially unstable. For example, consider a system of independent tasks with the *delayed arrival* of one or more real tasks. Under the Extended Scheduling Model, each delaying process is modeled as a phantom task with at least one real child. Thus, the introduction of delayed task arrivals causes a system of independent tasks to become potentially unstable.

4.4.2 Manacher's Scheduling Environment

The basic version of Manacher's stabilization algorithm covers a multiprocessor system with no phantom tasks. One extension of the algorithm allows "subsidiary start-times", or fixed delays in the initial release of tasks. Manacher handled these fixed delays by regarding "each subsidiary start-time as a null common ancestor task,"[Manacher, 1967, p. 447]. This is equivalent to treating the delaying process as a fixed-duration phantom task with no parents, and is thus a subset of the Extended Scheduling Model. Manacher's algorithm can be further adapted to cover the entire Extended Scheduling Model by treating *all* phantom tasks as "null common ancestors" of all their real descendents.

4.4.3 Modular Redundancy

Many fault-tolerant systems execute several copies of each task on different processors. Then, the results from all copies of the task are voted upon before being utilized by the children of the task. Voting serves to mask errors and to support the diagnosis of a faulty processor. However, voting can also destabilize the task system. For example, Figure 5(a) shows a simple, inherently stable 3 task precedence graph. Figure 5(b) shows the same graph modified for triple modular redundancy (where the subscripts A, B, and C indicate different copies of the same task). Note that three forks have been added to the task system. Thus, the effect of modular redundancy with voting was to destabilize an inherently stable task system.

Insert Figure 5 here

In Figure 5(b), voting was incorporated into task T_2. However, some systems such as MAFT employ separate hardware voters to speed up the voting process [Kieckhafer et al, 1988]. In such systems, voter operations must be treated as *phantom* tasks, as represented by phantom vertices V_A, V_B, and V_C in Figure 5(c). While separate voters may improve throughput, the introduction of phantom tasks aggravates the instability problem because every phantom task with a real child is potentially a phantom fork.

4.4.4 Uniprocessors

A uniprocessor is simply the limiting case of a multiprocessor, for which $M = 1$. It is thus the simplest possible application of the Extended Scheduling Model.

Without Phantom Tasks – A uniprocessor with no phantom tasks is inherently stable. This fact is easily proven by showing that priority inversion can not occur in this system. Specifically, let T_u be the lowest numbered unstarted (real) task on the list at the time a list scan begins. Thus, all tasks in $\mathbf{T}_{<u}$ have already started. At the time of the scan, the single processor has just become idle. Since all tasks are real, the existence of the idle processor implies that *all* previously started tasks must be finished. Therefore, all predecessors of T_u must be finished, so that T_u is ready to run. Since the first unstarted task on the list is always ready as soon as the processor becomes available, the tasks will always start in ascendign numerical order. Hence, priority inversion *can not occur*, and the system is inherently stable by condition [I1].

With Phantom Tasks – The introduction of phantom tasks violates the assumption that the first unstarted real task T_u is always ready at the time of a scan. Although the processor is idle, T_u may still be waiting for one or more *phantom* parents to finish. The dispatcher may thus scan beyond T_u and start a (real) usurper task T_x, where $x > u$. We may then observe the following:

1. If T_x is started, then s_x is the time of the scan. Thus, priority inversion has occurred, i.e. $s_x < s_u$, where $x > u$ so that condition [I1] is true.
2. With respect to T_u, the leftover set $\mathbf{L}$ contains only the unfinished phantom ancestors of T_u, and T_u itself. The fact that T_x started means that there is at least one unfinished phantom task which is a parent of T_u. Time f_ϕ, the time of the last fork into $\mathbf{L}$, occurs when the last phantom parent of T_u finishes. Thus, $s_x < f_\phi \leq s_u^{std}$ so that condition [I2] is true.
3. In a uniprocessor, instability condition [I3] (processor contention) reduces to starting condition [S3] (processor availiability). If the actual duration of task T_x is $c_x > s_u^{std} - s_x$, then T_x will still be occupying the processor at s_u^{std}. Hence, task T_u will be unable to

start on time. However, if $c_x \leq s_u^{std} - s_x$, then $f_x \leq s_u^{std}$. Thus, the processor is available no later than s_u^{std}, so that T_u can start on-time.

Thus, in a uniprocessor system with phantom tasks, the only allowable priority inversions are *slack-time safe* priority inversions. Thus, no stable dispatcher can exceed the limits of "Fixed Starting Sequence" dispatching with slack time reclaiming.

4.4.5 Static Task Allocation

Many real-time systems employ static task-to-processor allocation, i.e. tasks are assigned to processors at design time based on the SGC [Butler and Di Vito, 1992, Kieckhafer, 1989, Stancovic and Ramamithram, 1987]. Each processor then employs a separate, private priority list containing only those tasks which it executed on the SGC. However, a task may still have precedence relations with tasks allocated to other processors.

Static allocation is less flexible at run-time than dynamic allocation. However, it does have some advantages. First, it is simpler to implement, since it requires no shared dispatcher data structures. Second, it imposes less overhead since mutual exclusion is not needed on the dispatcher code or data structures. Third, full dynamic allocation can not be used in all systems. For example, under modular redundancy, different copies of the same task must be assigned to different processors. The question is then how static allocation affects stability.

In a system employing static allocation, stability can be determined by modeling each processor as a uniprocessor system. Then, for each processor, all tasks allocated to that processor are modeled as real tasks, while all tasks allocated to the other processors are modeled as phantom tasks. The stability of the schedule on each processor can then be examined individually, under the assumption that all phantom tasks are completed on-time. If the schedule of the processor in question is stable under this assumption, then this processor can not *initiate* instability, i.e. the *first* task to miss its deadline can not reside on this processor. If all processors individually satisfy this criterion, then there is no processor which can initiate instability. Hence, the entire system is stable.

This model for systems with static allocation exploits the conditions for guaranteeing uniprocessor stability from Subsection 4.4.4. Specifically, stability can be guaranteed only under Fixed Starting Sequence dispatching with optional slack-time reclaiming. This result has implications for several existing real-time architectures, including the Reliable Computing Platform (RCP) [Butler and Di Vito, 1992], and the Multicomputer Architecture for Fault-Tolerance (MAFT) [Kieckhafer, 1989].

RCP – The Reliable Computing Platform (RCP) is a recent design for a redundant uniprocessor to provide a "verified real-time computing platform" for a "wide variety of control system applications" [Butler and Di Vito, 1992]. Originally, RCP employed a frame-based deterministic schedule table to implement the Fixed Starting Times approach. It was later found that some variations can be permitted within a scheduling frame without compromising verification, as long as all tasks are completed by the end of the frame.

One long term goal of the RCP study is to determine whether "more sophisticated scheduling strategies could be shown to conform"[Butler and Di Vito, 1992]. However, the results of this subsection indicate that any scheduling strategy that goes beyond Fixed Starting Sequence with slack time reclaiming is vulnerable to instability.

MAFT – The Multicomputer Architecture for Fault-Tolerance (MAFT) employs static task-to-processor allocation [Kieckhafer, 1989]. MAFT employs a classical priority list dispatcher which scans the entire list, if necessary, to find a ready task. The system relies on Hugue's algorithm for off-line stabilization of the precedence graph [McElvaney, 1988, Hugue and Stotts, 1991].

The MAFT dispatcher is incapable of performing slack time calculations. Thus, the results of this subsection reveal that Fixed Starting Sequence is *the only stable dispatching method* available to MAFT. Careful examination of Hugue's algorithm reveals that it adds sufficient edges to the graph to prevent *all* priority inversions within the priority list of a given processor. The same effect could have been achieved by limiting the dispatcher to scan only the first unstarted task. Had these results been available at the time the MAFT dispatcher was designed, significant savings in run-time overhead and chip area could have been achieved. In addition, this result holds for true multiprocessing, whereas Hugue's method applies only to redundant "unischedule" workloads.

5 Run-Time Stabilization

The General Instability Conditions are necessary and sufficient to induce instability. These results can be applied to obtain new approaches to stabilization. We have examined the use of *Run-Time Stabilization*, in which the dispatcher enforces stability at run-time using the original precedence graph. Run-time stabilization has the advantage that it is not necessary to enforce stability in all possible scenarios. Rather, it only needs to enforce stability given the actual conditions at the time the list is scanned. If actual run-time conditions can be exploited, then run-time stabilization is less restrictive than *a priori* stabilization.

5.1 Scan Window Dispatching

When a processor finishes its current task, the traditional priority list dispatcher scans the entire task list, if necessary, to find a ready task. In the *Scan Window* approach, stability is enforced by limiting the number of real tasks scanned by the dispatcher such that no usurper task is ever started before a vulnerable task. The subset of *unstarted* real tasks scanned by the dispatcher are said to be in the Scan Window.

T_u = The first task in the scan window. T_u is always the lowest numbered unstarted real task on the priority list.

T_z = The last task in the scan window. The identity of T_z is determined by the scan window algorithm.

n_s = The number of real tasks in the set $\{T_u, \ldots, T_z\}$ which have already started execution.

n_p = The number of phantom tasks in the set $\{T_u, \ldots, T_z\}$.

D_u = The *Scan Depth*, or the number of unstarted real tasks in the scan window.

Based on the above definitions, the scan depth of a given execution of the dispatcher is:

$$D_u \quad = (z - u) + 1 - (n_p + n_s) \;\leq\; (z - u) + 1. \tag{1}$$

In (1) the scan depth is bounded by $(z-u)+1$. Furthermore, no dispatcher can scan beyond the end of the priority list. Thus, the scan depth is also bounded by $(N-u)+1$. To avoid visual clutter, this end-of-list constraint is not explicitly stated in the algorithm descriptions to follow. However, it is always presumed.

The General Instability Conditions provide the basis for determining the maximum safe scan depth. The problem of determining precisely when condition [I3] applies is NP-hard. Thus, a precise determination of the maximum safe scan depth is not practical in a real system. However, heuristic stability criteria derived from condition [I3] can be employed to implement low complexity stable dispatchers which may underestimate, but never overestimate, the safe scan depth.

5.1.1 Basic Algorithms

Four *basic* algorithms have been developed for limiting the dispatcher scan depth D_u. All four have been proven to produce inherently stable dispatchers [Krings and Kieckhafer, 1993]. The basic algorithms conservatively estimate the identity of the lowest numbered vulnerable task at the time of the scan. Each depth limiting algorithm treats one or more of the following tasks as potentially unstable, and inhibits priority inversions about that task.

1. T_u : the lowest numbered unstarted real task.
2. T_{u2} : the second unstarted real task on the list.
3. T_α : the lowest numbered unstarted real task that has an unfinished phantom parent.
4. T_β : the lowest numbered unstarted real task which is also the second real child of an unfinished forking task.
5. T_γ : the lowest numbered unstarted real task 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.

Task T_γ is called a *fan-out* task, because it represents an increase in the number of processors occupied by the descendents of a given parent. Task T_α is also a fan-out task, because it occupies one real processor, while its phantom ancestor occupied zero real processors. The *amount* of the fan-out is then the difference between the number of processors occupied by the descendents on the SGC and the number occupied by the common ancestor.

A scan window dispatcher limits the scan depth by defining the last task scanned, T_z such that potentially $z < N$. The four basic algorithms for limiting the scan depth are:

Algorithm 1: $z = u$ (Fixed Starting Sequence).
This scan window contains *only* task T_u, so $D_u = 1$.

Algorithm 2: $z = a \equiv \min(\alpha, u2))$.
This algorithm scans the first two unstarted real tasks unless $\alpha = u$. The largest possible scan window for this algorithm is thus $\{T_u, T_{u2}\}$, so $D_u \leq 2$.

Algorithm 3: $z = c \equiv \min(\alpha, \beta)$.
The largest possible scan window for this algorithm is $\{T_u, \ldots, T_c\}$, so $D_u \leq (c - u) + 1$.

Algorithm 4: $z = w \equiv \min(\alpha, \gamma)$.

The largest possible scan window for this algorithm is $\{T_u, \ldots, T_w\}$, so $D_u \leq (w - u) + 1$.

Example: To illustrate the operation of these dispatchers, consider again the precedence graph in Figure 1. In this example, instability is possible only if T_6 starts before T_5. None of these dispatchers allow that situation to occur. Algorithm 1, with its scan depth of 1, would not scan T_6 until T_5 has already started. Algorithm 2, with its maximum scan depth of 2, would not scan T_6 until after T_4 has started. By that time, T_5 will have already been released by its latest parent, T_2. T_5 is also the second child of forking task T_2. Thus, $T_\beta = T_5$, so that Algorithm 3 cannot scan beyond T_5 until T_2 is completed, making T_5 ready to run. Similarly, the SGC of Figure 2a shows that T_5 ran in parallel with T_4. Since both of these tasks have common ancestor T_2, $T_\gamma = T_5$. Thus, Algorithm 4 cannot scan beyond T_5 until T_2 is completed [2]. Thus, none of these dispatchers allow T_6 to start before T_5.

5.1.2 Augmented Algorithms

Let I be the number of idle processors at the time of the scan (including the processor which initiated the scan). It has been shown that the scan depth of each of the basic algorithms can be increased by the value $(I-1)$ [Krings and Kieckhafer, 1993]. These *augmented* algorithms are indicated by appending the letter "A" to the basic algorithm name. Table 1 summarizes the run-time complexities of the basic and augmented dispatcher algorithms for an M–processor, N–task system. The $\log(N)$ complexity is obtained by storing ready tasks in a heap data structure.

Insert Table 1 here

5.1.3 Slack-Time Reclaiming

Each of the Basic and Augmented dispatcher algorithms can be extended by including slack-time reclaiming. Specifically, a usurper task can be started either if it is inside the scan window, or if it is slack-time safe with respect to the first potentially vulnerable task recognized by the dispatcher.

[2] The stability criteria employed by Algorithm 4 are essentially the criteria of Manacher's algorithm, expanded to account for phantom tasks.

5.1.4 Minimally Stabilized Dispatcher

Krings [Krings, 1993] has developed a dispatcher algorithm which locates *all* unstarted ready tasks which can be started without enabling an unstable scenario. It performs a complete run-time evaluation of all three General Instability Conditions. This algorithm constructs an "NGC-tree" to perform an exhaustive search of all possible scenarios, given the scenario existing at the time of the scan. Since worst-case dispatcher overhead must be included in task durations when the SGC is constructed, the maximum overhead must be bounded and "small". Thus, due to its exponential complexity, the minimally stabilized algorithm can not be implemented in a real system. However, it has been used in simulation studies to evaluate the relative performance of the heuristic schedulers in Non-Standard Scenarios.

This algorithm imposes the minimum constraints on the dispatcher necessary to ensure stability. Thus, it never leaves a processor idle as long as there is *any* ready task which can not cause instability. This is not to imply that it produces an optimal schedule in any particular scenario. However, in performance simulations with a variety of precedence graphs, the minimally stabilized dispatcher always produced the best average processor utilization over a large number of non-standard scenarios.

6 Performance Simulations

Extensive scheduling simulations were performed to assess the relative performance of the scan window dispatchers presented above. The objective of the simulations was to investigate how well the various dispatchers utilized processor time in non-standard scenarios. Simulations were performed on a variety of representative task graphs, including several "Real-World" workloads for hard real-time systems in robotics and jet engine control. In addition, a number of precedence graphs were generated locally to represent task systems of varying size and structure. In some cases, phantom tasks were embedded in the graph to model external processes.

For each precedence graph, the SGC was generated assuming an unrestricted priority list dispatcher. In order to obtain relatively efficient SGC's, the *urgency strategy* described in [Manacher, 1967] was used to assign initial task priorities. The projective list was then extracted from the SGC. After the projective list was obtained, simulation trials were conducted to simulate non-standard scenarios. For each trial, the actual duration of each task was randomly generated within the range of its minimum and maximum durations. The resulting task system was then scheduled using each of the scan window dispatchers. For each precedence graph, 10,000 Non-Standard Scenarios were simulated on each of a 2, 4 and 8 processor system.

6.1 Performance Metrics

The primary metrics employed are:

$\overline{\mu}$ = Processor utilization averaged over all trial repetitions and dispatchers.

$\Delta\mu$ = The *maximum* difference in processor utilization observed between dispatchers for a given precedence graph.

$\overline{D}$ = The mean scan depth at which a ready task was found.

These metrics reflect how successful the dispatcher is at finding a safe ready task during a given scan, highlight the differences between dispatchers, and indicate a reason for the observed results. Other performance metrics were considered, including makespan and accumulated time leftover at the end of the schedule. However, simulation experiments showed these metrics to be redundant in that they closely followed the behavior of processor utilization.

6.2 Test Precedence Graphs

The precedence graphs used in simulation were partitioned into three categories, according to the applications they represent. These graphs are described briefly herein. For complete details and diagrams, the reader is referred to [Krings, 1993].

6.2.1 Generic Test Graphs

Several precedence graphs were generated locally to represent task systems of varying size and structure. These graphs were partitioned into four sub-groups.

Delayed Start Graphs represent systems in which the starting time of some initial real tasks is non-zero. In such graphs, all phantom tasks are initial tasks. Each child of a phantom task is then a real task with no real parents. On average, one-half of the initial real tasks were delayed by phantom parents.

Communication Delay Graphs model delays between the completion of a task and the release of its children. Each communication delay is modeled by a phantom task, which is the sole child of a real task. Any children of the real task then become children of the phantom communication task.

Modular Redundancy Graphs model fault-tolerant systems in which multiple copies of each task are executed concurrently on different processors. The results of the individual copies are then subjected to a voting process before any copies of any children are released. Some of the graphs incorporated phantom tasks to model separate hardware voters as previously illustrated in Figure 5(c).

Arbitrary Graphs represent a more general mixture of real and phantom tasks, comprising features of the other groups. These graphs were designed to provide structural variety rather than to model specific features of a system.

6.2.2 Pathological Graphs

Several precedence graphs were constructed with the intent of exacerbating performance differences between dispatchers. These graphs were designed to reward large scan depths by imposing large time penalties for dispatchers with inherently small scan depths. The results reported herein are for the graph which produced the largest value of $\Delta\mu$.

6.2.3 Real World Applications

While the above task systems are intended to be representative of real-time task systems, they are nonetheless artificial. To obtain data with real-world task systems, simulations were performed on the pecedence graphs for a turbojet engine controller and several robot-arm controllers as listed in Table 2.

Insert Table 2 here

6.3 Simulation Results – Uniform Distribution

Each precedence graph was subjected to 10,000 simulation trials. For each trial, the actual duration of each task was randomly generated using a uniform distribution over the interval between its minimum and maximum durations. The results of simulations are presented separately for each category of precedence graphs.

6.3.1 Generic Test Graphs

The performance of the generic test graphs is shown in Table 3, which lists average utilizations $\overline{\mu}$ for the minimally stabilized dispatcher, and the maximum difference $\Delta\mu$ to any other dispatcher. Averaged over all generic graphs, the 2 and 4 processor simulations showed relatively high processor utilizations, whereas for the 8 processor case lower values of $\overline{\mu}$ were achieved, due to the limited parallelism available in the graphs. The highest utilization was achieved for the graphs with the smallest number of phantoms, i.e. the delayed start graphs.

Insert Table 3 here

The most important result of Table 3 is the observation that values of $\Delta\mu$ are small, indicating there was very little difference in average processor utilization across the dispatchers. The reason for the nearly uniform performance of all dispatchers was observed in the low average scan depth $\overline{D}$ for all dispatchers. All basic and augmented dispatchers exhibited $\overline{D} < 2$ for all trials, while the minimally stabilized algorithm achieved a maximum value of $\overline{D} = 2.08$. Thus, even the least constrained dispatcher rarely scanned past the second unstarted real task (T_{u2}) to find a ready task. Hence, the deeper scan depth of the more complex dispatchers was seldom useful.

6.3.2 Pathological Graphs

The previous results showed nearly identical performance for all dispatchers over a variety of precedence graphs. Therefore, heuristic "Pathological" graphs were constructed to exacerbate the differences between dispatchers (i.e. to increase $\Delta\mu$). These graphs were designed to reward large scan depths, and to impose large time penalties on dispatchers with inherently small scan depths. The pathological graphs *without* phantom tasks yielded a worst-case $\Delta\mu = 18\%$, while the pathological graphs *with* phantom tasks yielded a worst-case $\Delta\mu = 12\%$. While, these values are larger than the $\Delta\mu$ values for the generic graphs, they are still quite acceptable, especially in light of the fact that the pathological graphs were contrived with the purpose of exacerbating $\Delta\mu$.

One reason for this relative uniformity in performance is that specific combinations of task durations are required to necessitate a large scan depth. The stochastic nature of task durations makes these combinations extremely rare. This assertion is supported by the fact that all dispatchers had an average scan depth of $\overline{D} \leq 2.2$.

6.3.3 Real-World Work Loads

Dispatcher performance was also simulated using the real workloads listed in Table 2. All work loads implement real tasks only, with the exception of the turbojet controller, which was also simulated in a MAFT-like dispatching environment, with phantom communication tasks inserted between each real task and its real children.

Processor utilization depends on c_i^{min}/c_i^{max}, the ratio of minimum duration to maximum duration for task T_i. This parameter is highly dependent on the system architecture. Simulations were performed for duration ratios in the range: $c_i^{min}/c_i^{max} = [0.1\ , \ldots ,\ 0.8]\,\forall\, T_i$. The lower value, 0.1, allows for large variations in message passing delays and for cache memories up to one order of magnitude faster than the main memory. In all cases, the lowest values of $\overline{\mu}$ and the largest values of $\Delta\mu$ were obtained when $c^{min}/c^{max} = 0.1$. Hence, these are the values reported herein.

Simulation results, shown in Table 4, reveal extremely high processor utilizations for the 2 and 4 processor systems. As before, the lower utilizations for the 8 processor case were due to the limited parallelism available in the precedence graphs. The $\Delta\mu$ column shows that the worst-case value of $\Delta\mu$ was 6.2%. Thus, the real-world workloads showed the same uniformity in relative dispatcher performance as the generic workloads.

Insert Table 4 here

6.3.4 Slack-Time Reclaiming

All precedence graphs were simulated to investigate the impact of adding slack-time reclaiming to the dispatcher algorithms. Table 5 summarizes the values of $\Delta\mu$ between all dispatchers with and without slack-time reclaiming. As can be seen, $\Delta\mu$ is significantly reduced for all non-pathological workloads when slack-time reclaiming was enabled. Most importantly, $\Delta\mu$ for the real-world workloads was reduced to approximately 1%. Slack-time reclaiming appears to be most effective for precedence graphs with long chains of dependent tasks. In graphs with this property, slack-time "compounds" or accumulates with the execution of each task in the chain.

Insert Table 5 here

6.4 Biased Duration Distributions

The preceding results were obtained with actual task durations uniformly distributed between their minimum and maximum values. In many systems, the maximum duration can occur only after an unlikely sequence of events (e.g. repeated cache misses). Thus, the average duration of a task may be *much less than* the maximum duration. Let $\overline{R}$ denote the normalized expected task duration within the minimum to maximum duration interval, i.e. $\overline{R} = 0.5$ for uniformly distributed durations. Further simulations were performed using distributions skewed toward the minimum duration ($\overline{R} = 0.2$, and $\overline{R} = 0.11$).

Simulations revealed that the *relative* performance of the dispatchers is quite insensitive to the distribution of task durations [Krings, 1993]. Non-pathological graphs showed a maximum decrease in $\overline{\mu}$ of 3%, with essentially no change in $\Delta\mu$ between distributions. The pathological graphs showed as much as a 16% reduction in $\overline{\mu}$, but again, there was no significant change in $\Delta\mu$.

7 Conclusions and Future Work

Non-premptive priority list scheduling is simple, easily implemented, and requires little run-time overhead. The potential for scheduling instability restricts its application in hard real-time systems. However, all deadlines are guaranteed in all possible scenarios, given two conditions: (1) a single feasible schedule is found for the standard scenario, (2) the system is proven to be stable. Previous studies of the instability problem have developed limited stabilization methods based on incomplete understanding of the problem in restricted scheduling environments. In contrast, this work makes four main contributions to the state of the art.

First, an *Extended Scheduling Model* was developed as a generalization of existing scheduling models. This model employs *phantom* tasks to represent external processes and events. The integration of phantom tasks into the precedence graph permits great flexibility in modeling features such as delayed task releases, non-transparent overhead, communication delays, and static task-to-processor allocation.

Second, a set of *General Instability Conditions* both necessary and sufficient to cause instability was derived. These conditions form a complete theoretical basis for studying the instability problem under a variety of scheduling environments. Their derivation also revealed the destabilizing effect of phantom tasks with real children. In addition, the General Instability Conditions permit stabilization methods to be derived for any scheduling environment within the Extended Scheduling Model.

Third, eight inherently stable low complexity *Scan Window Dispatchers* were presented. The most complex of these dispatchers had a run-time complexity of $O(\log\ N)$, where N is the number of tasks. This is the same complexity as that of the classical (unstabilized) priority list dispatcher. Some dispatchers had run-time complexities of $O(1)$ or $O(M)$, where M is the number of processors in the system. The complexities of these inherently stable dispatchers is thus less than that of the classical priority list dispatcher. In addition, a "Minimally Stabilized" dispatcher was developed, which determines precisely which usurper tasks can cause instability. While the exponential complexity of this dispatcher makes it impractical in real systems, it has proven useful as a benchmark for assessing relative performance of the heuristic dispatchers.

Fourth, performance simulations were conducted for the scan window dispatchers using representative real-time workloads. The objective was to determine how well the various dispatchers utilized processor time in non-standard scenarios. A dispatcher which maintains high processor utilization in non-standard scenarios would tend to leave a contiguous block of processor time available for soft tasks at the end of the scheduling cycle. Simulations have revealed that, for realistic workloads, scan-window dispatchers can maintain relatively high processor utilizations (up to the point where the number of processors exceeds the degree of parallelism available in the precedence graph). More importantly, the simulations showed only a few percent difference in processor utilization between dispatchers. This overall uniformity in performance indicates that the low overhead dispatchers are nearly as efficient as the most complex dispatchers. Hence, the most cost-effective dispatchers are likely to be the simplest ones (e.g. Algorithms 1, 1A, 2, and 2A).

One topic for continued research involves stabilty in resource-constrained systems. At this time, the Extended Scheduling Model does not deal with resources other than the processors. However, resource constraints between phantom tasks would violate some of the basic assumptions used to derive the General Instability Conditions. It will thus be necessary to rederive the General Instability Conditions under the resource constraint assumption.

Another topic currently being studied is the average performance of systems with static task-to-processor allocation relative to those with dynamic allocation. Static allocation reduces run-time message traffic and avoids contention for the dispatcher data structures. As shown in Subsection 4.4.5, static allocation makes a system unstable under all dispatchers except algorithm 1 (with optional slack-time reclaiming). Initial results from current research indicate very little difference in mean processor utilization between systems with static and dynamic allocation. If these results hold true, then static allocation offers a means of further reducing overhead without compromising average performance.

References

[Bettani et al, 1991] Bettani, R., D. Gillies, C.C. Han, K.J. Lin, J.W.S. Liu, and W.K. Shih, 1991. "Recent Results in Real-Time Scheduling", in *Foundations of Real Time Computing: Scheduling and Resource Management*, A. M. van Tilborg, and G.K. Koob (eds.), pp. 91–127, Kluwer Academic Publishers.

[Butler and Di Vito, 1992] Butler, R.W., and B.L. Di Vito, 1992. *Formal Design and Verification of a Reliable Computing Platform for Real-Time Control*, NASA Tech. Memo. 104196, NASA Langley Research Center.

[Chen et al, 1988] Cheng, S-C., J.A. Stankovic, and K. Ramamritham, 1988. "Scheduling Algorithms for Hard Real-Time Systems – A Brief Survey", in *IEEE Tutorial – Hard Real-Time Systems*, J.A. Stankovic and K. Ramamritham, (eds.) pp. 150–173.

[Coffman and Gilbert, 1985] Coffman, E.G.,and E.N. Gilbert, 1985. "On the Expected Relative Performance of List Scheduling", *Operations Research*, V. 33, No. 3, pp. 548–561.

[Hugue and Stotts, 1991] Hugue, M.C., and P.D. Stotts, 1991. "Guaranteeing Task Deadlines for Fault-Tolerant Workloads with Conditional Branches", *The Journal of Real-Time Systems*, Vol. 3, No. 3, pp. 275–305.

[Hwang and Briggs, 1984] Hwang, K. and F.A. Briggs, 1984. *Computer Architecture and Parallel Processing*, McGraw-Hill.

[Hwang et al, 1989] Hwang, J-J., Y-C. Chow, F.D. Anger, and C.Y. Lee, 1989. "Scheduling Precedence Graphs in Systems With Interprocessor Communication Times", *SIAM Journal of Computing*, V. 18, No. 2, PP. 244–257.

[Jeffay et al, 1991] Jeffay, K., D.F. Stanat, and C.U. Martel, 1991. "On Non-Preemptive Scheduling of Periodic and Sporadic Tasks", *Proc. 12th IEEE Real-Time Systems Symp.*, pp. 129–139.

[Kasahara and Narita, 1984] Kasahara, H., and S. Narita, 1984. "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing", *IEEE Trans. on Computers*, V. C-33, No. 11, pp. 1023–1029.

[Kasahara and Narita, 1985] Kasahara, H., and S. Narita, 1985. "Parallel Processing of Robot-Arm Control Computation on a Multimicroprocessor System", *IEEE Journal of Robotics and Automation*, 1(2), pp. 104-113.

[Kasahara, 1992] Kasahara, H., 1992. "Parallel Processing of Robot Control and Simulation", *Parallel Computation Systems for Robotics*, Edited by Fijany, A., and A. Bejczy, World Scientific Publishing Co. Ltd, pp. 77-93.

[Kieckhafer et al, 1988] Kieckhafer, R.M., C.J. Walter, A.M. Finn, and P.M. Thambidurai, 1988. "The MAFT Architecture for Distributed Fault-Tolerance", *IEEE Trans. Computers*, V. C-37, No. 4, pp. 398-405.

[Kieckhafer, 1989] Kieckhafer, R.M., 1989. "Fault-Tolerant Real-Time Task Scheduling in the MAFT Distributed System", *Proc. 22nd Hawaii Int'l. Conf. on System Sciences.*

[Krings and Kieckhafer, 1993] Krings, A.W., and R.M. Kieckhafer, 1993. "Inherently Stable Priority List Scheduling for Systems With External Delays", *Proc. 26th Hawaii Int'l. Conf. on System Sciences*.

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

[Krueger and Livny, 1988] Krueger, P. and M. Livny, 1988. "A Comparison of Preemptive and Non-Preemptive Load Distributing", *Proc. 8th Int'l Conf. on Distributed Computing Systems*, pp. 123–130.

[Krueger and Chawla, 1991] Krueger, P. and R. Chawla, 1991. "The Stealth Distributed Scheduler", *Proc. 11th Int'l Conf. on Distributed Computing Systems*, pp. 336–343.

[Leung, 1982] Leung, J. Y-T, 1982. "On Scheduling Independent Tasks with Restricted Execution Times", *Operations Research*, V. 30, No. 1, pp. 163–171.

[Liestman and Campbell, 1986] Liestman, A.L., and R.H. Campbell, 1986. "A Fault-Tolerant Scheduling Problem", *IEEE Trans. on Software Engineering*, V. SE-12, No. 11, pp. 1089–1095.

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

[McElvaney, 1988] McElvaney, M.C., 1988. "Guaranteeing Deadlines for MAFT Workloads", *Proc. 9th IEEE Real-Time Systems Symp.*.

[Min. of Defence, 1989] Ministry of Defence, United Kingdom, 1989. *Interim Defence Standard 00–55*.

[Pygott and Kershaw, 1990] Pygott, C. and J. Kershaw, 1990. "VIPER 2 and NODEN", *Proc. NASA Formal methods Workshop, 1990*, NASA Conf. Publication 10052.

[Ramamritham et al, 1989] Ramamritham, K, J.A. Stankovic, and P-F Shiah, 1989. "$O(n)$ Scheduling Algorithms for Real-Time Multiprocessor Systems", *Proc. 1989 Internations Conference on Parallel Processing*, pp. III-143 – III-152.

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

[Shen et al, 1990] Shen, C. et al, 1990. "Resource Reclaiming in Real-Time", *Proc. 6th IEEE Real-Time Systems Symp.*, pp. 41-50.

[Shirazi et al, 1990] Shirazi, B., M. Wang, and G. Pathak, 1990. "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling", *J. Parallel and Distributed Computing* V. 10, pp. 222–230, Academic Press.

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

[Verhoosel et al, 1991] Verhoosel, J.P.C., E.J. Luit, and D.K. Hammer, 1991. "A Static Scheduling Algorithm for Distributed Hard Real-Time Systems", *The Journal of Real-Time Systems*, V. 3, pp. 227–246.

Figures and Tables

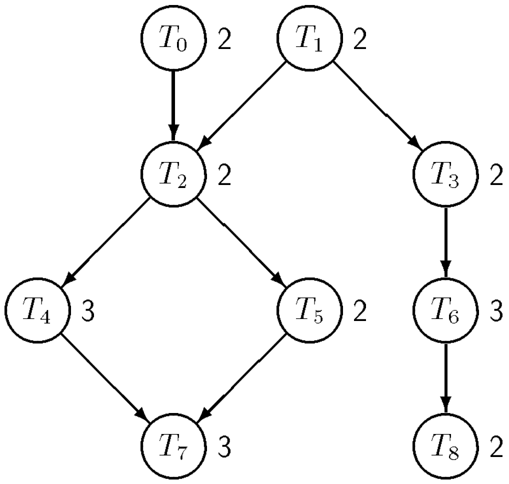
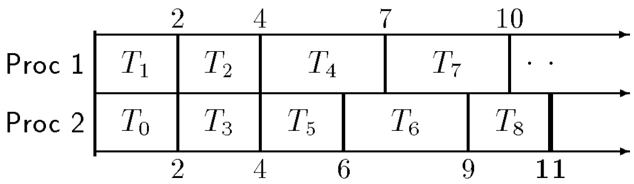
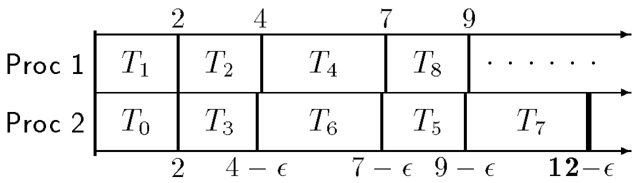


Figure 1: Example Precedence Graph



(a) Standard Gantt Chart – Maximum Durations.



(b) Non-standard Gantt Chart – T_3 shortened by ϵ.

Figure 2: Example of Instability

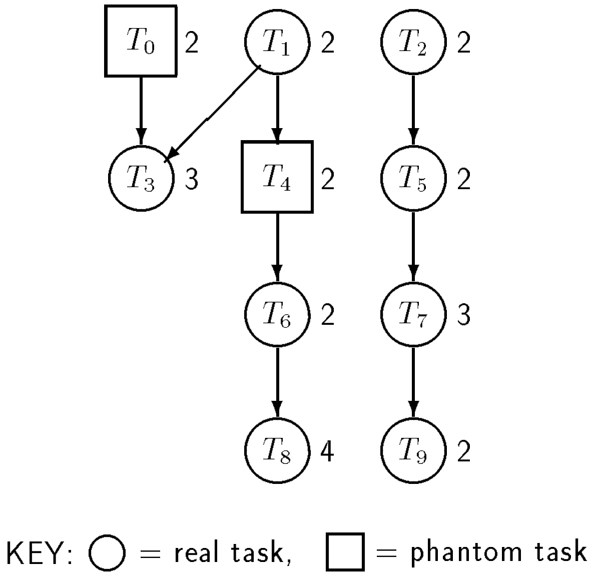
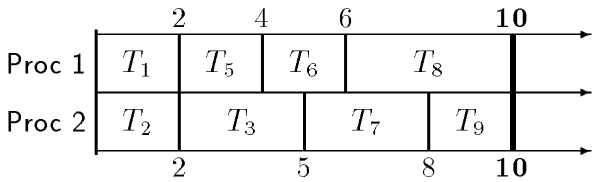
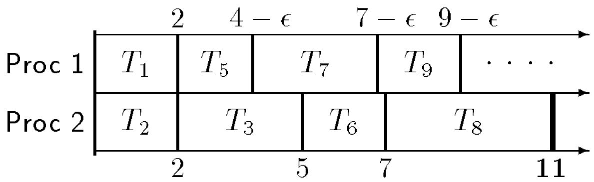


Figure 3: Example Precedence Graph With Phantom Tasks



(a) Standard Gantt Chart – Maximum Durations.



(b) Non-standard Gantt Chart – T_5 shortened by ϵ.

Figure 4: Example of Instability Caused by Phantom Tasks

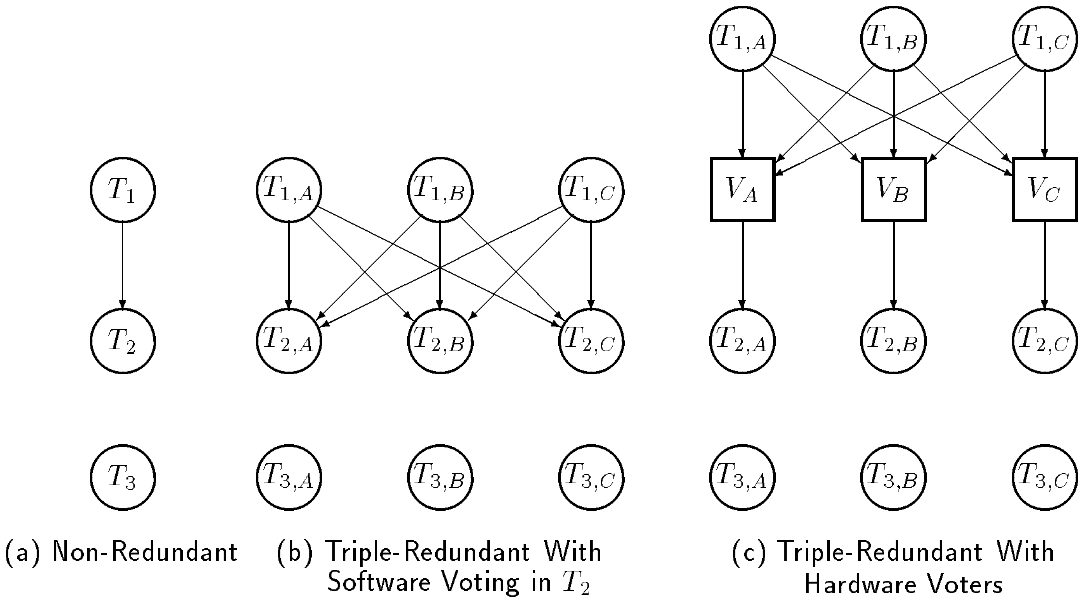


Figure 5: Modular Redundancy in a Precedence Graph

Table 1: Summary of Basic and Augmented Scan Window Dispatchers

Alg'm Name	Maximum Scan Depth		Run-Time Complex.
1	$D_u = 1$		$O(1)$
1A	$D_u = I$		$O(M)$
2	$D_u = (a - u) + 1$	where:	$O(1)$
2A	$D_u = (a - u) + I$	$a = \min(\alpha, u2)$	$O(M)$
3	$D_u = (c - u) + 1$	where:	$O(logN)$
3A	$D_u = (c - u) + I$	$c = \min(\alpha, \beta)$	$O(logN)$
4	$D_u = (w - u) + 1$	where:	$O(logN)$
4A	$D_u = (w - u) + I$	$w = \min(\alpha, \gamma)$	$O(logN)$

Table 2: Real Workloads

Name	Tasks	Application/Reference
Turbojet	64	Engine Control [Shaffer, 1990]
Robot 1	88	Newton-Euler [Kasahara and Narita, 1985]
Robot 2	103	Newton-Euler [Kasahara and Narita, 1985]
Robot 3	90	Newton-Euler [Kasahara and Narita, 1985]
Robot 4	200	Walker & Orin [Kasahara, 1992]

Table 3: Generic Workload Utilizations ($\overline{\mu}$, $\Delta\mu$) in %

Workload	M=2	M=4	M=8	$\Delta\mu$
Delayed Start	98	93	75	7.6
Communication Delay	96	86	65	4.1
Modular Redundancy	96	79	47	1.0
Arbitrary Graphs	97	80	44	3.3

Table 4: Real Workload Utilizations ($\overline{\mu}$, $\Delta\mu$) in %

Test graph	$\overline{\mu}$ $M = 2$	$\overline{\mu}$ $M = 4$	$\overline{\mu}$ $M = 8$	max $\Delta\mu$
Turbojet	99.7	97.1	60.0	6.1
Turbojet(MAFT)	99.4	91.1	54.0	5.7
Robot 1	99.2	89.9	50.0	3.7
Robot 2	99.9	82.3	42.5	5.4
Robot 3	99.8	93.5	54.4	6.2
Robot 4	97.6	92.8	58.1	6.2

Table 5: Impact of slack-time reclaiming on $\Delta\mu$ in %

Slack-time Reclaiming / Workload	disabled $\Delta\mu$	enabled $\Delta\mu$
Real Work-loads	6.2	1.1
Delayed Start	7.6	4.7
Communication Delay	4.1	1.7
Modular Redundancy	1.0	0.9
Arbitrary Graphs	3.3	1.0
Pathological Graphs	17.9	17.9