The Performance of Inherently Stable Priority List Dispatchers in Hard Real-Time Systems

A.W. Krings
Dept. Comp. Sci.
Technical University of Clausthal
38678 Clausthal, Germany

R.M. Kieckhafer
Dept. Comp. Sci. and Eng.
University of Nebraska
Lincoln, NE 68588-0115

J.S. Deogun
Dept. Comp. Sci. and Eng.
University of Nebraska
Lincoln, NE 68588-0115

Abstract

This paper investigates low overhead solutions to the problem of scheduling instability in non-preemptive static priority list scheduling. Non-preemptive priority list scheduling is vulnerable to several *multiprocessor anomalies*. For example, in precedence constrained task systems, real-time deadlines can be missed due to a reduction in the duration of one or more tasks. A system displaying this behavior is called *unstable*. Several inherently stable run-time dispatchers of varying complexity are presented and their performance is investigated. These algorithms are less restrictive than previous stabilization methods and are based on an *Extended Scheduling model* that includes *phantom tasks* as a mechanism to model non-transparent overhead and events external to the processor.

The dispatchers presented range from the simplest priority sequence enforcing dispatcher, up to a minimally stable algorithm based on the conditions necessary and sufficient for instability to occur. Extensive simulation on a wide range of task systems, including real-world workloads, shows that very simple low-overhead dispatchers have near-optimal average performance. Thus, real-time system developers are supplied with simple run-time dispatching algorithms that make complicated stabilization methods unnecessary.

1 Introduction

Non-preemptive static priority list scheduling is a relatively simple, low-overhead approach to scheduling computational tasks in real-time multiprocessor systems. In 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 scheduling has proven desirable in several real-time systems, including the Spring Kernel [11], the Reliable Computing Platform (RCP) [1] and the Multicomputer Architecture for Fault Tolerance (MAFT) [5]. Simplicity and low run-time overhead are the main motivators for its use [1, 5]. 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 [2]. This phenomenon, called *Scheduling Instability*, can make it difficult or impossible to guarantee real-time deadlines [2, 8]. 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.

This paper addresses the relative performance of several dispatcher stabilization algorithms of widely varying complexity. Section 2 presents background information and describes the scheduling model employed in this study. Section 3 introduces Scan Window Dispatching and a new class of provably stable run-time dispatchers. These dispatchers are less restrictive than known stabilization algorithms, since they are derived from a set of General Instability Conditions which are both *necessary and sufficient* for instability to occur [6]. Section 4 describes the general performance testing procedures. Section 5 presents scheduling simulation data, comparing performance of the run-time dispatchers, including

a "minimally stabilized" dispatcher. Extensive simulations have shown that even simple, low overhead dispatchers perform remarkably well. This is an important result for developers of hard real-time systems, in that it suggests that "simple is better", i.e. one can implement fast low overhead dispatchers which guarantee stability, but still deliver near optimal performance.

2 Background and Motivation

Non-preemptive priority list scheduling is vulnerable to several *multiprocessor anomalies* [2], which describe counter-intuitive or unstable scheduling behavior caused by variations in system parameters. For example, deadlines can be missed due to (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 one 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. Simulating schedules with all possible variations in the durations of all tasks is in most cases an intractable problem. Therefore, task dispatching must be provably stable for 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 on a single processor. A task system, described by a directed acyclic precedence graph with N tasks, is to be scheduled on M processors. Durations for each

task T_i are specified by minimum and maximum values. For a given instance of a task, its actual duration varies randomly within these bounds.

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) [8]. 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 another ready task for execution 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* if there exists no scenario under which the completion time of *any* task on any NGC exceeds its completion time on the SGC [1] [8].

2.2 Example of Instability

Figure 1 shows the precedence graph for an example eight-task system, with the maximum duration of each task listed next to its vertex. Priorities are defined in order of increasing start times on the SGC.

Figure 2 demonstrates scheduling instability. Gantt chart (a) is the dual-processor SGC for the task system of Figure 1. Gantt chart (b) is the NGC obtained when task T_3 is shortened by an arbitrarily small value $\epsilon > 0$. On this 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.

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

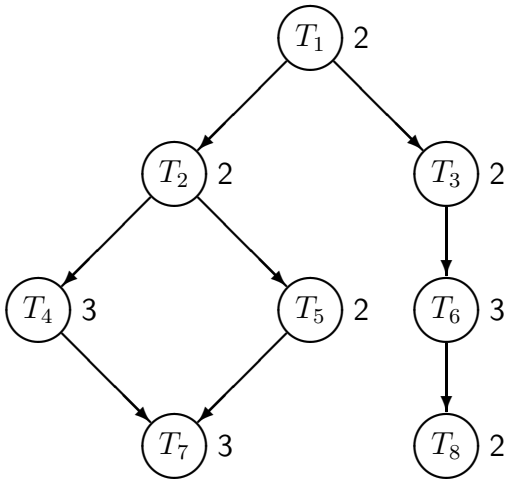


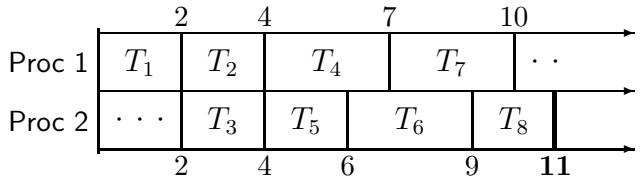
Figure 1: Example Precedence Graph

2.3 Stabilization Options

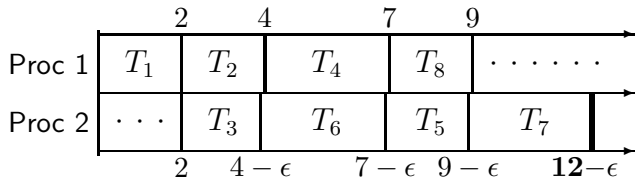
There are several approaches to stabilizing a task schedule. Two approaches, *fixing the task starting sequence* and *fixing task starting times* are among the most restrictive methods that will inhibit instability. The potential of these methods to cause poor processor utilization has motivated the development of several alternative stabilization methods.

Manacher's Algorithm One early algorithm for stabilizing a system of real tasks was developed by Manacher [8]. It was intended to permit greater flexibility than the two approaches stated above. Manacher's stabilization algorithm alters the original precedence graph, potentially adding many edges, to produce a graph which is inherently stable.

Hugue's Algorithm In recent years, stability has been studied in the context of specific real-time systems. In particular, Hugue [9] has developed a variation on Manacher's approach, customized to the scheduling environment of the Multicomputer Architecture for Fault-Tolerance (MAFT) [5].



(a) Standard Gantt Chart – Maximum Durations.



Non-standard Gantt Chart – T_3 shortened by ϵ.

Figure 2: Example of Instability

Run-Time Algorithms An alternative to the *a priori* stabilization methods above is *run-time* stabilization. In this approach, the dispatcher limits the depth of its scan into the task list in order to avoid instabilities, using information known at run-time. This less restrictive approach takes advantage of the general instability conditions described in [6], which are necessary and sufficient for instabilities to occur.

2.4 Extended Scheduling Model

The arrival and execution of tasks in a real-time system may be partially dependent upon processes and events occurring outside the processors themselves. To model such events, our scheduling model incorporates a concept called *Phantom Tasks*. These tasks take time, but do not occupy any processors. A critical feature of this model is that phantom tasks are fully integrated into the precedence graph along with real tasks. This model extension allows precedence constraints between real and phantom tasks. Some applications of phantom tasks include:

1. The arrival of a task may be delayed by an external timer. In the extended scheduling

model the timer process is represented by a phantom task. Then, in the precedence graph, the delayed real task is specified as a child of the phantom task.

2. Task A can initiate an external task B, (e.g. a DMA operation) which must complete before task C can proceed. This case can be modeled with a task chain, consisting of *real* task A, *phantom* task B and *real* task C.

3. In message-based systems, communication between processors can impose a delay between the completion of the parent task and the release of its children. In the interim, the processors are available for other unrelated tasks. These delays can be modeled by a phantom task inserted into the connecting edge of each parent-child pair.

3 Run-Time Stabilization

Manacher's algorithm modifies the precedence graph *a priori* to ensure stability in all possible scenarios. Run-time stabilization is less restrictive and leaves the original precedence graph unchanged. The dispatcher is modified to enforce stability, not necessarily considering all possible scenarios. Rather, it only needs to enforce stability given the actual scenario up to the current time.

This section presents several provably stable task dispatching algorithms that are based on restricting the number of tasks which the dispatcher may scan. Due to limited space, the stability proofs are omitted. The interested reader is referred to [7].

3.1 Scan Window Dispatching

When a processor finishes its current task, the traditional priority list dispatcher may scan the entire task list to find a ready task. Stability can be enforced at run-time if the number of tasks scanned is limited such that no usurper task is ever started before a *vulnerable task*. A task is called vulnerable to instability if there exists at least one scenario in which

it is the lowest numbered task to miss its deadline.

The subset of *unstarted* real tasks scanned by the dispatcher are said to be in the *Scan Window*. The first task in the scan window is always the first unstarted real task in the priority list. The number of unstarted real tasks scanned is called the *Scan Depth*, D_u, where T_u is the lowest numbered unstarted real task at the time of the scan.

The General Instability Conditions of [6] provide the basis for determining the maximum safe scan depth. Three tasks are important to the depth limiting algorithms to follow. These tasks are defined at the time the list is scanned.

T_α is the lowest numbered real task that has an unfinished phantom parent.

T_β is the lowest numbered real task which is also the second real child of an unfinished forking task.

T_γ is the lowest numbered real task such that:

1. T_γ is descended from an unfinished forking task,
2. On the SGC, T_γ started while another real descendant of the same forking task was running.

Example: Using the precedence graph of Figure 1 and the NGC of figure 2(b), consider the status of the list when it is scanned at time $t = 4-\epsilon$. At this time, the lowest numbered unstarted task is $T_u = T_4$. Further inspection shows that the only unfinished forking task at this time is T_2. Thus, tasks T_α, T_β, and T_γ are defined as follows.

T_α does not exist since there are no phantom tasks in this system. However, had T_6 been a phantom task instead of a real task, then the lowest numbered real task with an unfinished phantom parent would be $T_\alpha = T_8$.

T_β is the second real child of unfinished forking task T_2. Hence, $T_\beta = T_5$.

T_γ is also T_5 because it is the lowest numbered real task such that:

1. It is descended from unfinished forking task T_2,
2. On the SGC (Figure 2(a)), it started while T_4 (another descendent of T_2) was executing.

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 and the number of processors occupied by the common ancestor.

We have developed scan depth limiting algorithms which are partitioned into three classes, *Basic*, *Augmented* and *Frame-based* Algorithms [7]. No dispatcher can scan beyond the end of the priority list. Thus, the scan depth is always bounded by $D_u \leq (N - u) + 1$. To avoid visual clutter, this constraint is not explicitly stated, however, it is always presumed.

3.2 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 [7].

Algorithm 1: $D_u = 1$. This scan window contains *only* the first unstarted real task, $\{T_u\}$.

Algorithm 2: $D_u = (a - u) + 1$, where: $a = \min[\,\alpha, (u + 1)\,]$.
The *largest* scan window for this algorithm is $\left\{T_u, T_{(u+1)}\right\}$, so $D_u \leq 2$.

Algorithm 3: $D_u = (c - u) + 1$, where: $c = \min[\,\alpha, \beta\,]$.
The scan window is thus $\{T_u\ , \ldots,\ T_c\}$.

Algorithm 4: $D_u = (w - u) + 1$, where: $w = \min[\,\alpha, \gamma\,]$.
The scan window is thus $\{T_u\ , \ldots,\ T_w\}$.

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 was 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 the second child of forking task T_2. Therefore, 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, Algorithm 4 cannot scan beyond T_5 until T_2 is completed. Thus, none of these dispatchers allow T_6 to start before T_5.

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

3.3 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 extended by the value $(I-1)$ [7]. These *augmented* algorithms are indicated by appending the letter A to the basic algorithm name. The basic and corresponding augmented dispatcher algorithms, as well as their run-time and pre-processing complexities, are summarized in Table 1. In a list implementation, Algorithm 1 and 2 have constant complexity due to the fixed scan depth. This increases complexities for Algorithm 1A and 2A to $O(M)$ since the scan depth can increase to maximally M. The remaining algorithms have $O(N)$ due to the fact that worse case the full list is scanned. Heap implementations result in $O(logN)$, due to the reordering of the heap. For more information the reader is referred to [7].

[2]Manacher's algorithm prevents priority inversion with regard to the *first* task of a co-running pair, whereas Algorithm 4 prevents inversion with regard to the *second* task of the co-running pair.

Alg'm Name	Maximum Scan Depth	Run-Time Complex. linear	Run-Time Complex. heap	Pre-Proc. Complex.
1	$D_u = 1$	$O(1)$	$O(logN)$	None
1A	$D_u = I$	$O(M)$	$O(logN)$	None
2	$D_u = (a - u) + 1$ where $a =$	$O(1)$	$O(logN)$	$O(N)$
2A	$D_u = (a - u) + I$ $\min[\,\alpha, (u+1)\,]$	$O(M)$	$O(logN)$	$O(N)$
3	$D_u = (c - u) + 1$ where $c =$	$O(N)$	$O(logN)$	$O(N)$
3A	$D_u = (c - u) + I$ $\min[\,\alpha, \beta\,]$	$O(N)$	$O(logN)$	$O(N)$
4	$D_u = (w - u) + 1$ where $w =$	$O(N)$	$O(logN)$	$O(N^3)$
4A	$D_u = (w - u) + I$ $\min[\,\alpha, \gamma\,]$	$O(N)$	$O(logN)$	$O(N^3)$

Table 1: Summary of Basic and Augmented Scan Window Dispatchers

3.4 Frame Based Algorithms

Recall that a *fan-out task* is any real task with *at least one* of the following properties: (1) It is a child of a phantom task, (2) It is descended from a (real or phantom) forking task, and it started execution on the SGC while another descendent of the same forking task was executing on another processor. In Algorithm 4, the scan window was defined as $\{T_u, ..., T_w\}$, where $w = min[\alpha, \gamma]$. T_w is the first fan-out task, and on the SGC it causes a fan-out of 1 (note that T_w is a descendant of an unfinished parent). Assume that T_w is the only fan-out task in the workload. Then, if one wants to scan past T_w, stability is guaranteed only if there is at least one additional (reserved) idle processor that can *absorb* the possible fan-out of T_w. This argument led to augmented Algorithm 4A [7].

In the search for stabilization algorithms which are less constrained than the augmented algorithms, it is necessary to identify fan-out tasks beyond T_w. Let $T_{w1} = T_w$, then define $T_{wi} = min[T_\alpha, T_\gamma]$ ignoring all T_{wj}, $j < i$, as candidates for T_α and T_γ. The result is the complete list of *all* fan-out tasks. All T_{wi} are called **basic fan-out tasks**, in that each T_{wi}

can cause a fan-out of 1. In the examples of the following sections, it will be shown that some fan-out tasks have special properties. For ease of notation T_w will be used from now on to denote any arbitrary basic fan-out task T_{wi}.

Recall that Algorithm 4A scanned $I-1$ tasks beyond T_{w1}. It can be shown [7], that one can extend the idea of Algorithm 4A to span the scan window $I-1$ fan-out tasks beyond T_{w1}, i.e. up to the I^{th} fan-out task T_{wI}. This algorithm is called the **W-Algorithm**. Its complexity stays the same as Algorithm 4/4A, i.e. $O(N)$, since a scan up to T_{wI} might at worse require a full scan.

3.4.1 Effective Fan-out

The W-Algorithm is based on the assumption that every basic fan-out task T_w needs to be considered when determining the safe scan depth. However certain basic fan-out tasks cannot cause instability.

Let s_w^{std} denote the start time of T_w in the standard scenario, i.e. on the SGC. Define $\mathcal{F}(T_w)$ as a function that indicates how many basic fan-out tasks with indices less than or equal w are executing at s_w^{std}. Then a task T_w is said to have an *effective fan-out* of $\mathcal{F}(T_w)$. The example in Figure 3 shows parent tasks T_{p1} and T_{p2} which have 2 and 1 basic fan-out tasks as children, respectively. In the corresponding SGC shown in Figure 4, where shaded areas indicate that the task is a fan-out task, $\mathcal{F}(T_{w1}) = 1$, $\mathcal{F}(T_{w2}) = 2$ and $\mathcal{F}(T_{w1'}) = 3$.

3.4.2 Overlapping Fan-out Tasks:

If there are k basic fan-out tasks executing on the SGC at some time t, then the effective fan-out of the k *overlapping* fan-out tasks is k. In the SGC of Figure 4 one can see that at time $t_1 = s_{w1'}^{std}$ the effective fan-out of the three basic fan-out tasks is 3, as reflected by $T_{w1'}$ with $\mathcal{F}(T_{w1'}) = 3$.

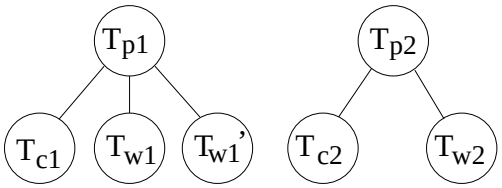


Figure 3: Subgraph of workload

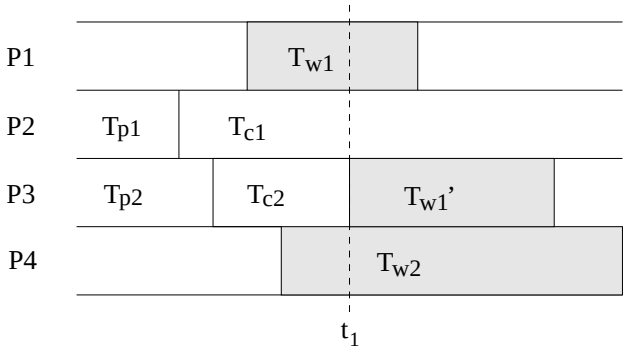


Figure 4: SGC overlapping fan-out tasks

3.4.3 Non-Overlapping Fan-out Tasks:

Not every basic fan-out task contributes to an increase in the effective fan-out. Assume that several basic fan-out tasks exist such that their executions do not overlap on the SGC. It can be shown that these *non-overlapping* basic fan-out tasks can collectively contribute only to an effective fan-out of 1. Figures 5 and 6 show the subgraph and the SGC of a system with two non-overlapping basic fan-out tasks. Fan-out $\mathcal{F}(T_{w1}) = 1$ and $\mathcal{F}(T_{w2}) = 1$ and the effective fan-out of $\{T_{w1}, T_{w2}\}$ at any time is 1 and not 2, as expected by the W-Algorithm.

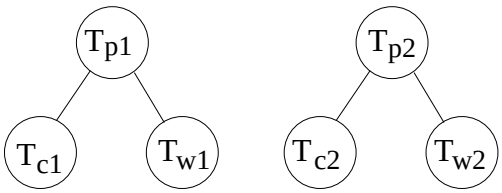


Figure 5: Subgraph of workload

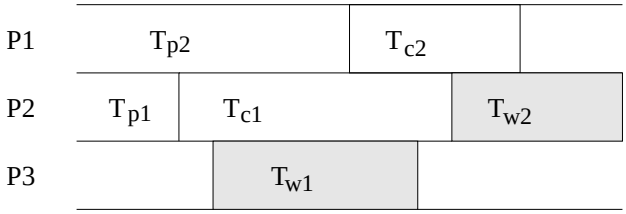


Figure 6: SGC non-overlapping fan-out tasks

3.4.4 Effective Fan-out Tasks:

Let T_{ei} denote the *lowest numbered* basic fan-out task with $\mathcal{F}(T_w) = i$. Then T_{ei} is called an *effective fan-out task*. T_{e1} is the first effective fan-out task ($\mathcal{F}(T_{e1}) = 1$), T_{e2} the second and so forth. By definition $T_{e1} = T_{w1}$. Every effective fan-out task is also a basic fan-out task, but the reverse is not necessarily true. It should be noted that T_{ei} is not necessarily the only fan-out task with a fan-out of i, but it is the *first*.

3.4.5 Scan Frames

The priority list can now be partitioned starting with the first unstarted task T_u. The general priority list at the time of the scan is

$$PL = (T_{e0}, ..., T_{e1}, ..., T_{e2}, ..., T_{ek}, ...).$$

Task $T_{e0} = T_u$ if[3] $\mathcal{F}(T_u) = 0$, otherwise T_{e0} does not exist and the list starts with T_{e1}. T_{ek} is the last effective fan-out task, and its effective fan-out is bounded by M, the number of processors in the system. Positioned between T_{ei} and $T_{e(i+1)}$ are any number of tasks T_j with effective fan-outs $0 \leq \mathcal{F}(T_j) \leq i$. These tasks, including T_{ei}, are called the **Scan-Frame** of T_{ei} and are denoted by $\mathbf{\Delta}_{ei}$. Thus $\mathbf{\Delta}_{ei}$ is the set $\{T_{ei}, ..., T_{e(i+1)-1}\}$. The definition of scan-frames is with respect to the current scan. Whereas the W-Algorithm takes the first I basic fan-out tasks under consideration, one can modify this algorithm (see subsection 3.4.7) considering effective fan-out tasks T_{ei} only.

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

3.4.6 Impact of Usurper Task Durations

Assume T_x is the next task checked for safe starting. It can be shown that the only scan frames that need to be investigated are those which contain tasks T_v, whose SGC starting time s_v^{std} overlap timewise with the execution of usurper task T_x on the NGC, assuming T_x were started [7]. This means that scan frames $\mathbf{\Delta}_{ej}$ with s_{ej}^{std} greater than the maximal finishing time of T_x cannot be vulnerable to instability caused by starting T_x. Thus, a scan window algorithm has to reserve processors only for the frames whose associated T_{ei} starts at or before the maximum finishing time of T_x, since safety of the succeeding frames follows.

3.4.7 The E-Algorithm

As a result of the previous discussion the following E-Algorithm can be specified:

1. Find the first ready task T_x and determine its scan-frame k'.
2. Find the last task T_v with index $v < x$ whose standard starting time overlaps the hypothetical execution of T_x and find its scan frame k.
3. T_x can be safely started if k idle processors can be reserved for tasks from $\{\Delta_{e0}, \ \ldots, \ \Delta_{ek}\}$.

3.4.8 The F-Algorithm

The *fan-in* of a task sub-graph is defined to be the event that causes at least one processor to permanently become idle, with respect to tasks *in that sub-graph*. If one extends the E-Algorithm to include the impact of fan-ins, one derives a run-time implementation of the General Instability Conditions. This *F-Algorithm* is then minimally stable in the sense that no processor is left idle as long as there exists any ready task which can be started safely [7]. However, the F-Algorithm has exponential complexity due to the need of generating an NGC-Tree to investigate all possible fan-in scenarios.

3.4.9 Slack-Time Reclaiming

Assume the dispatcher scans the list at time t. The slack-time of a task T_i is then $s_i^{std} - t$,the time remaining until the starting deadline of T_i. At time t, starting a usurper task $T_x \in \mathbf{T}_{>i}$, can not cause T_i to be unstable as long as its maximum duration $c_x^{std} \leq s_i^{std} - t$. If this inequality is true, then it is said that task T_x is *slack-time safe* with respect to T_i.

Each of the dispatcher algorithms above can be extended by including slack-time reclaiming[4]. 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 vulnerable task recognized by the dispatcher. Slack time reclaiming can cause the dispatcher to scan *all* ready tasks to find a slack-time safe task. Thus, given N tasks, slack time reclaiming adds $O(N)$ to the complexity of each algorithm, regardless of whether the ready tasks are maintained in a heap.

4 Performance Testing

Extensive scheduling simulations were performed to assess the performance of the dispatchers presented. The objective of performance testing of the dispatchers was (1) to investigate how well the low overhead dispatchers performed and (2) to investigate the change in performance as dispatcher complexities increase.

Simulations were performed on a variety of representative task graphs, including several "Real World" workloads for hard real-time systems from the areas of 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.

Execution of each precedence graph was simulated using each of the scan window dispatchers. In order to obtain relatively efficient SGC's, the *urgency strategy* described in [8] was

[4]Slack time reclaiming is implicit in the F-Algorithm, but may be added as an option to each of the others.

used to assign initial task priorities.

For each simulation trial, the actual duration of each task was randomly generated using a uniform distribution over the interval between its minimum and maximum durations. The resulting task system was then scheduled using each of the scan window dispatchers. Each simulation trial was repeated 10,000 times with different durations on a 2, 4 and 8 processor system, respectively.

4.1 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 average scan depth at which a ready task was found.

4.2 Test Precedence Graphs

The different graphs tested were partitioned into three categories, according to the applications they represent. The test graphs are described briefly herein. For complete details, the reader is referred to [7].

4.2.1 Generic Test Graphs

Different graph structures were generated locally to represent real task systems of varying size and structure.

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. In addition, each child of a phantom task is 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, considering parameter passing. Such delays occur in loosely coupled systems, wherein updating of the dispatcher's data structures is subject to message passing delays [5]. Each 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 new phantom 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 systems employ separate hardware voters in order to free the processors from voting [5]. Voting processes must then be modeled as phantom tasks.

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.

4.2.2 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.

4.2.3 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 five precedence graphs shown in Table 2. Workload "Turbojet" is the precedence graph for Shaffer's turbojet engine controller program, originally adapted to a four-processor system [10]. Workloads "Robot 1" through "Robot 3" implement the Newton-Euler algorithm for control of the Stanford Manipulator arm [3], while workload "Robot 4" implements the Walker and Orin algorithm, also for the Stanford Manipulator [4].

In addition to representing real-world controllers, these graphs offer substantial variety in structure and size, and have the advantage of being available in the open literature. Since space limitations prohibit complete descriptions of the graphs here, The interested reader is referred to [7] for more detail.

Name	Tasks	Application/Reference
Turbojet	64	Engine Control [10]
Robot 1	88	Newton-Euler [3]
Robot 2	103	Newton-Euler [3]
Robot 3	90	Newton-Euler [3]
Robot 4	200	Walker & Orin [4]

Table 2: Real Workloads

5 Simulation Results

The results of simulations are presented separately for each category of precedence graphs. Simulations were first performed without slack-time reclaiming (except for the F algorithm, in which slack-time reclaiming is implicit). The impact of enabling slack-time reclaiming is discussed in subsection 5.5

5.1 Generic Test Graphs

The performance of the generic test graphs is shown in Table 3, where the average utilizations $\overline{\mu}$ for the minimally stable F-Algorithm, and the maximum difference $\Delta\mu$ to any other dispatcher are given. Averaged over all generic graphs, the 2 and 4 processor simulations showed relatively high processor utilizations, whereas for the 8 processor case lower $\overline{\mu}$ were achieved, due to the limited concurrency of *real* tasks available in the graphs. The highest utilization was achieved for the graphs with the smallest number of phantoms, i.e. the delayed start graphs.

The most important result is the small values of $\Delta\mu$ observed, indicating there was very

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 3: Utilization of generic workloads in %

little difference in average processor utilization across the dispatchers. Furthermore, the simulations showed nearly identical performance within the set of basic algorithms, (1, 2, 3 and 4), the augmented algorithms, (1A, 2A, 3A and 4A), and the frame based algorithms, (W, E and F). The reason for the nearly uniform performance of all dispatchers was observed in the low average scan depth $\overline{D}$, i.e. $\overline{D} < 2$ for all basic and augmented dispatchers.

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: $c_i^{min}/c_i^{max} = [0.1\ , \ldots ,\ 0.8]\ , \forall T_i in \mathcal{T}$. 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 in Table 3.

For the F-Algorithm on average 98%, 87% and 65% of all tasks were selected at a scan depth of unity, with corresponding average scan-depths of 1.06, 1.29 and 2.08, in the 2, 4 and 8 processor systems respectively. Thus, the more complex dispatchers derived little benefit from a deeper scan depth.

5.2 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. These graphs were designed to reward large

scan depths, and to impose large time penalties on dispatchers with inherently small scan depths. The graph structure *without phantom tasks* which gave the largest performance differences is shown in Figure 7a. For each pathological graph, the ratio c_i^{min}/c_i^{max} was individually selected for each task T_i to exacerbate $\Delta\mu$. Pathological graphs were constructed

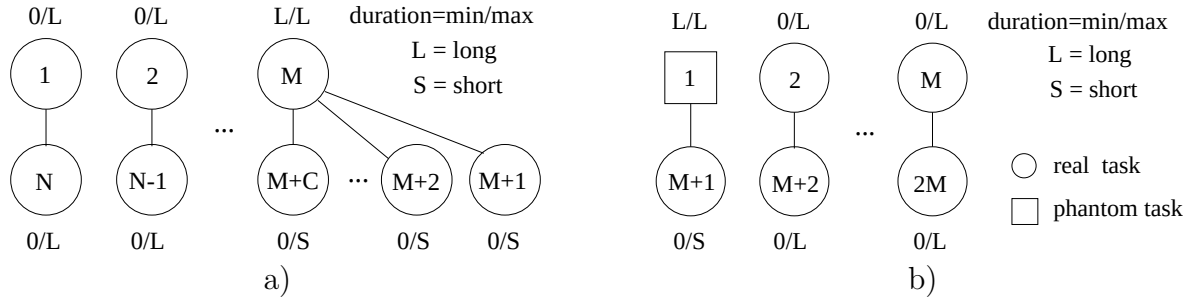
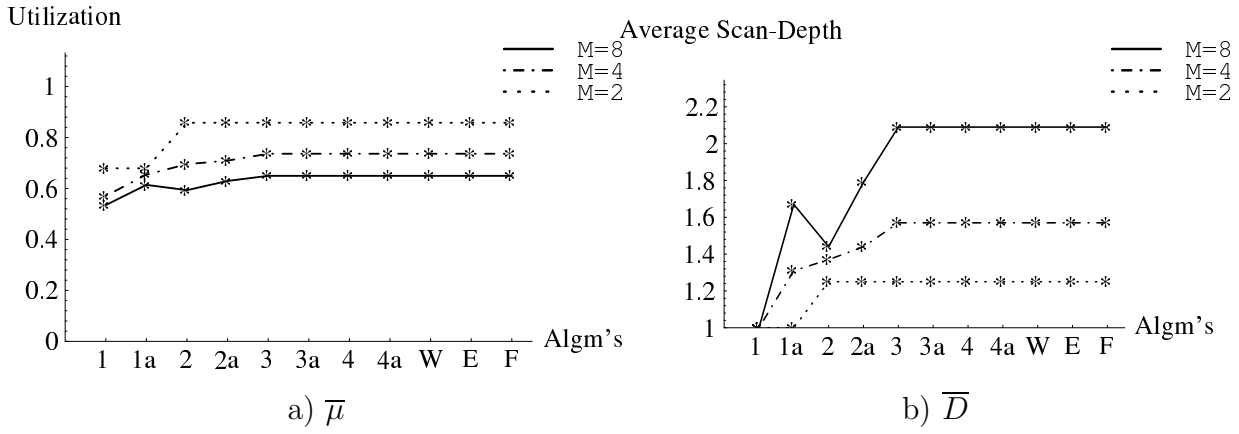


Figure 7: Pathological Graphs

for $M = 2$, 4, and 8, respectively. Utilization and average scan depth for all dispatchers is shown in Figure 8a and 8b. The effect of forcing deep scans is evidenced by the worst case values of $\overline{D} > 2$. However, the largest value of $\Delta\mu$ was still less than 18%. One reason for the uniformity in performance is that specific combinations of task durations are needed to induce poor performance. The stochastic nature of task durations makes these combinations extremely rare.

Figure 8a shows that performance generally improves with increasing complexity of the dispatcher. However, the major differences are due to Algorithms 1, 1A and 2. Algorithm 2A consistently produces results very close to the F-Algorithm.

Figure 7b shows the particular pathological graph *with phantom tasks* that had the worst performance. Figure 9a shows significantly lower average utilizations than for the previous graphs, because the phantom task causes the scan window to stall. Again, the performance of the augmented and frame based algorithms was uniformly better than that of the basic algorithms. However, the worst case value of $\Delta\mu$ was only 12%. The phantom task stalling the window causes the basic algorithms to limit their window to unity, whereas the dispatchers taking advantage of extra idle processors increase their window size up to



a) $\overline{\mu}$ b) $\overline{D}$

Figure 8: $\overline{\mu}$, $\overline{D}$ for Pathological Graphs Without Phantom Tasks

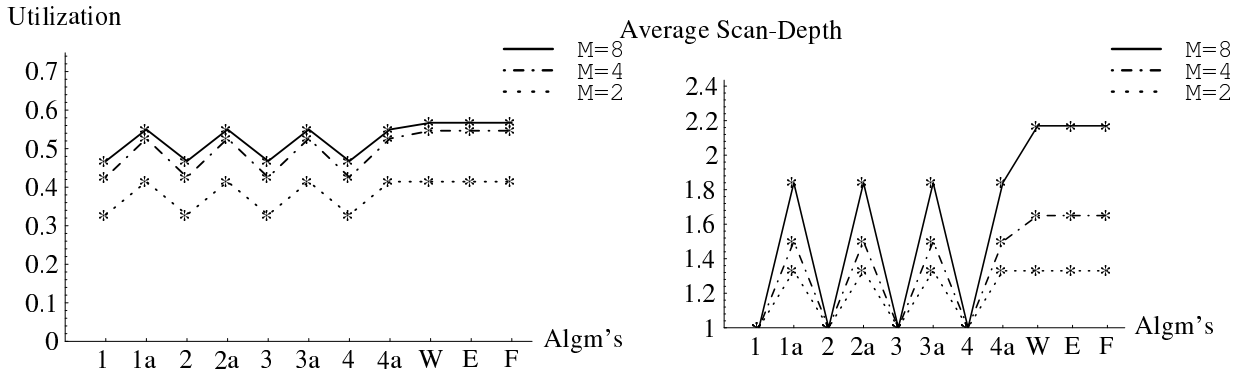
$\overline{D} = 2.17$. The average scan depth for all dispatchers is shown in Figure 9b. The phantom tasks prohibit $\overline{D}$ from going beyond 1 for all basic dispatchers, thus reducing them to algorithm 1.

5.3 Real-World Work Loads

Next, the real workloads listed in table 2 were simulated. All work loads implement real tasks only, with the exception of the Turbojet, which was also simulated in a MAFT-like dispatching environment, with phantom communication tasks added between real tasks.

The average utilization $\overline{\mu}$ for all scan algorithms was about equal for the 2 processor case, with maximum differences in utilization of less that 2%. The maximum differences in utilization for the 4 and 8 processor case increased slightly, but remained below 6.2%. The results are shown in table 4, where column $\Delta\mu$ A-F indicates the maximum difference between augmented dispatchers and the F-Algorithm.

As with the generic graphs, the ratio c_i^{min}/c_i^{max} was varied from 0.1 through 0.8 to allow for a variety of system architectures. Again, the value $c_i^{min}/c_i^{max} = 0.1$ consistently yielded worst case values for $\overline{\mu}$ and $\Delta\mu$. Hence, these are the values reported in Table 4. The lower utilizations for the 8 processor case were due to lack of available parallelism in the graphs.



a) $\overline{\mu}$ b) $\overline{D}$

Figure 9: $\overline{\mu}$, $\overline{D}$ for Pathological Graphs With Phantom Tasks

Test graph	$M = 2$	$M = 4$	$M = 8$	$\Delta\mu$	$\Delta\mu$ A-F
Turbojet	99.7	97.1	60.0	6.1	1.78
Turbojet(MAFT)	99.4	91.1	54.0	5.7	1.50
Robot 1	99.2	89.9	50.0	3.7	0.44
Robot 2	99.9	82.3	42.5	5.4	1.55
Robot 3	99.8	93.5	54.4	6.2	1.85
Robot 4	97.6	92.8	58.1	6.2	3.75

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

This conclusion was drawn from the observation that in less than 0.01% of the unsuccessful scans, did ready tasks exist beyond the safe scan window.

5.3.1 Performance of Manacher's Algorithm

Manacher's Algorithm implements edge stabilization by inserting extra edges in the graph to enforce stability a priori. The significant increase in the edge count for the real work loads is shown in table 5, with highest increase of 59%. Comparing the simulation results of window dispatchers with Manacher's algorithm showed that Algorithm 4 and Manacher's algorithm performed nearly identically.

Test graph	Original # edges	% increase by Mancher's Alg'm $M = 2$	$M = 4$	$M = 8$
Turbojet	102	21	12	39
Robot 1	130	18	32	59
Robot 2	149	43	50	57
Robot 3	125	29	34	24
Robot 4	339	35	40	39

Table 5: Edges added by Manacher's Algorithm

5.4 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. $\overline{R} = 0.5$ implies uniformly distributed durations. Further simulations were performed using distributions skewed toward the minimum duration ($\overline{R} = 0.2$, and $\overline{R} = 0.11$). These simulations revealed that the *relative* performance of the dispatchers is quite insensitive to the distribution of task durations [7].

5.5 Impact of Slack-Time Reclaiming

All previous simulations were performed without slack-time reclaiming (except for the F algorithm in which slack-time reclaiming is implicit). All simulation trials were then repeated with slack-time reclaiming enabled. Table 6 shows the maximum difference in utilizations between all dispatchers without and with slack-time reclaiming for different $\overline{R}$.

As can be seen, the difference in processor utilizations between dispatchers, $\Delta\mu$, was drastically reduced for all of the non-pathological workloads when slack-time reclaiming was enabled. Slack-time reclaiming is most effective when minimum and maximum task durations vary greatly. Furthermore, for graphs with many levels, slack-time tends to accu-

mulate with each task within a chain. In the simulations of all algorithms implementing

Slack-time Reclaiming	disabled	enabled		
Workload	$\overline{R} = 0.5$	$\overline{R} = 0.5$	$\overline{R} = 0.2$	$\overline{R} = 0.11$
Real Work-loads	6.2	1.1	0.8	0.5
Delayed Start	7.6	4.7	3.1	2.6
Communication Delay	4.1	1.7	1.5	1.3
Modular Redundancy	1.0	0.9	0.7	0.7
Arbitrary Graphs	3.3	1.0	1.0	0.6
Pathological Graphs	17.9	17.9	14.2	6.8

Table 6: Impact of slack-time reclaiming on $\Delta\mu$ in %

slack-time reclaiming, nearly identical results were achieved. This indicates that for work-loads with large differences in minimum and maximum task durations, enough slack-time can be accumulated to compensate for the different window sizes of the dispatchers. However, this performance is achieved by increasing the complexity of the basic and augmented algorithms to $O(N)$.

6 Summary

The objective of this paper was to examine schedule stabilization algorithms for non-preemptive static priority list dispatchers. Several provably stable run-time dispatchers of widely varying complexity have been presented. These algorithms are less restrictive than a-priori stabilization methods. They are based on an extended task model, featuring phantom tasks to model events like delayed task releases, non-transparent overhead, communication delays and static task-to-processor allocation.

The concept of scan window dispatching was introduced, leading to dispatching algorithms reaching from the most limited scan window (Algorithm 1), up to the run-time implementation of the necessary and sufficient General Instability Conditions of [6] (F-Algorithm).

Performance of the dispatchers was simulated for a variety of precedence graphs, including

several real world applications. Simulations showed that the simple dispatchers performed remarkably well, even though graph structures could be constructed that amplify the advantages enjoyed by the more complex dispatchers. The largest deviation in processor utilizations observed with a pathological precedence graph less than 18%. With real-world precedence graphs, this difference never exceeded 6.2%. The use of slack-time reclaiming reduced these performance differences even further.

The results of the simulations support an important thesis for developers of hard real-time systems, in that there is no need to implement complex algorithms for non-preemptive real-time task dispatching. Extremely simple, low-overhead task dispatchers exist which are guaranteed to be stable, and yet perform nearly as well as the most complex dispatchers. Thus, it is only necessary for the designer to find a feasible schedule in the standard scenario. The stability of all non-standard scenarios is then guaranteed by the dispatcher. Current research focuses on whether these results still hold for systems with static task to processor allocation.

References

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

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

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

[4] Kasahara, H., "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, 1992.

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

[6] Kieckhafer, R.M., and J.S. Deogun, " On the Stability of List Scheduling in Real-Time Multiprocessor Systems", Univ. of Nebraska – Lincoln, Dept. of Comp. Sci. and Eng., Report Series #99, Feb 1990.

[7] 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, 1993.

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

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

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

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