Scheduling Algorithms and Operating Systems Support for Real-Time Systems [1]

Krithi Ramamritham and *John A. Stankovic*

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract

This paper summarizes the state of the real-time field in the areas of scheduling and operating system kernels. Given the vast amount of work that has been done by both the operations research and computer science communities in the scheduling area, we discuss four paradigms underlying the scheduling approaches and present several exemplars of each. The four paradigms are: static table-driven scheduling, static priority preemptive scheduling, dynamic planning-based scheduling, and dynamic best-effort scheduling. In the operating system context, we argue that most of the proprietary commercial kernels as well as real-time extensions to timesharing operating system kernels do not fit the needs of predictable real-time systems. We discuss several research kernels that are currently being built to explicitly meet the needs of real-time applications.

[1] This material is based upon work supported by the National Science Foundation under grants CDA-8922572 and IRI 9208920, and by the Office of Naval Research under grant N00014-92-J-1048.

Contents

1 Introduction

Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. Examples of this type of real–time system are command and control systems, process control systems, flight control systems, the space shuttle avionics system, future systems such as the space station, space-based defense systems such as SDI, and large command and control systems. A majority of today's systems assume that much of this knowledge is available *a priori*, and hence are based on *static* designs which contribute to their high cost and inflexibility. The next generation hard real–time systems must be designed to be *dynamic*, *predictable*, and *flexible*.

When activities have timing constraints, as is typical of real-time computing systems, scheduling these activities to meet their timing constraints is one major problem that comes to mind. However, as we show in Section 2 of this paper, in spite of an extensive literature on scheduling, scheduling algorithms that are of practical value for real-time computing, ones that take real-world considerations into account, have only begun to appear. Given the vast amount of work that has been done by both the operations research and computer science communities in the scheduling area, it is impossible to do an exhaustive survey of the field. Instead, we discuss four paradigms underlying the scheduling approaches and discuss several exemplars of each. The four paradigms are: static table-driven scheduling, static priority preemptive scheduling, dynamic planning-based scheduling, and dynamic best-effort scheduling. Because of their increasing importance we also discuss the impact of quality-timeliness trade-offs, fault tolerance constraints, and resource reclaiming on scheduling.

Clearly, a real-time operating system must be able to perform integrated CPU scheduling and resource allocation so that collections of cooperating tasks can obtain the resources they need, at the right time, in order to meet timing constraints. In addition to proper scheduling algorithms, predictability requires bounded operating system primitives. Using the current operating system paradigm of allowing arbitrary waits for resources or events, or treating a task as a *random process* will not be feasible in the future to meet the more complicated set of requirements. It is also important to avoid having to rewrite the operating system for each application area. In Section 3 we elaborate on these issues and discuss operating systems under three broad categories: proprietary commercial kernels, real-time extensions to timesharing operating system kernels, and research kernels.

2 Real-Time Scheduling

Scheduling involves the allocation of resources and time to tasks in such a way that certain performance requirements are met. Scheduling has been perhaps the most widely researched topic within real-time systems. This is due to the belief that the basic problem in real-time systems is to make sure that tasks meet their time constraints. Scheduling is also a well-structured and conceptually-demanding problem. Given the resulting enormous amount of literature available on scheduling, any survey paper can only scratch the surface. On the other hand, just giving a list of algorithms is not useful. Hence, we have categorized the state of the art into a set of paradigmatic approaches to scheduling and present instances of the algorithms that fit the different paradigms.

This section is structured as follows. Since a scheduling algorithm is typically geared to meet a certain performance requirement, we first discuss, in Section 2.1, the different metrics that have been used in real-time systems. Based on these metrics and also on whether an algorithm is used on-line or off-line, different approaches or paradigms for scheduling have been used in the literature. Four main paradigms are introduced in Section 2.2. Section 2.3 discusses different examples of scheduling algorithms that conform to these four paradigms. In section 2.4 we discuss three additional important scheduling topics: scheduling with quality-timeliness trade-offs, scheduling with fault tolerance constraints, and resource reclaiming.

2.1 Performance Metrics in Real-Time Systems

The metrics that guide scheduling decisions depend on the application areas. The need to minimize the schedule length pervades static non-real-time systems and minimizing response times and increasing the throughput are the primary metrics in dynamic non-real-time systems. However, in both static and dynamic real-time systems, the main goal is to achieve timeliness. This introduces quite different metrics for the real-time case.

The variety of metrics that have been suggested for real-time systems is indicative of the different types of real-time systems that exist in the real world as well as the types of requirements imposed on them. This sometimes makes it hard to compare different scheduling algorithms. Another difficulty arises from the fact that different types of task characteristics occur in practice. Tasks can be associated with computation times, resource requirements, importance levels (sometimes also called priorities or criticalness), precedence relationships, communication requirements, and of course, timing constraints. If a task is periodic, its period becomes important; if it is aperiodic, its deadline becomes important. A periodic task may have a deadline by which it must be completed. This deadline may or may not be equal to the period. Both periodic and aperiodic tasks may have start time constraints.

Let us consider some of the performance metrics. In the static case, an off-line schedule is to be found that meets all deadlines. If many such schedules exist, a secondary metric, such as maximizing the average *earliness* is used to choose one among them. If no such schedule exists, one which minimizes the average *tardiness* may be chosen. In dynamic real-time systems, since, in general, it cannot be *a priori* guaranteed that all deadlines will be met, maximizing the number of arrivals that meet their deadlines is often used as a metric.

An issue related to metrics is the level of predictability afforded by a particular scheduling approach. That is, using a particular approach how well can we predict that the tasks will meet their deadlines? We will comment on this as we examine the different scheduling paradigms in the next subsection.

It should be mentioned that two different research communities have examined scheduling problems from their own perspectives. Scheduling in the Operations Research (OR) community has focussed on job-shop and flow-shop problems, with and without deadlines. For instance, manpower scheduling, project scheduling, and scheduling of machines are some of the topics studied in OR. The types of resources assumed by OR researchers (machines, factory cells, etc.) and how jobs use those resources (e.g., a job may be required to use every machine in some specified order) are quite different from those assumed by Computer Science researchers (CPU cycles, memory, etc. and where jobs typically use only a single machine). Also, activities on a factory floor typically have larger time granularities than those studied by computer scientists. Some of the metrics of interest to the OR community are: minimizing maximum cost, minimizing the sum of completion times, minimizing schedule length, minimizing tardiness, and minimizing the number of tardy jobs. Also, OR techniques are geared towards static (off-line) techniques whereas those developed in computer science focus more on dynamic techniques. In spite of these differences, the abstract problems studied by the two communities have a large commonality. In this paper, however, we examine scheduling problems mainly from the perspective of computer science. For an OR view of the problem we refer the reader to [29, 7, 12, 20, 34].

2.2 Scheduling Paradigms

As was mentioned in the previous section, predictability is one of the primary issues in real-time systems. Schedulability analysis or feasibility checking of the tasks of a real-time system has to be done to predict whether the tasks will meet their timing constraints. Several scheduling paradigms emerge, depending on (a) whether a system performs schedulability analysis, (b) if it does, whether it is done statically or dynamically, and (c) whether the result of the analysis itself produces a schedule or plan according to which tasks are dispatched at run-time. Based on this we can identify the following classes of algorithms:

- *Static table-driven approaches*: These perform static schedulability analysis and the resulting schedule (or table, as it is usually called) is used at run-time to decide *when* a task must begin execution.

- *Static priority-driven preemptive approaches*: These perform static schedulability analysis but unlike in the previous approach, no explicit schedule is constructed. At run-time, tasks are executed highest-priority-first.

- *Dynamic planning-based approaches*: Unlike the previous two approaches, feasibility is checked at run-time, i.e., a dynamically arriving task is accepted for execution only if it found feasible. (Such a task is said to be *guaranteed* to meet its time constraints.) One of the results of the feasibility analysis is a schedule or plan that is used to decide when a task can begin execution.

- *Dynamic best-effort approaches*: Here no feasibility checking is done. The system tries to do its best to meet deadlines. But since no guarantees are provided, a task may be aborted during its execution.

It must be pointed out that even though we have identified these four categories for ease of discussion, some scheduling techniques possess characteristics that span multiple paradigms. Now we briefly elaborate on each of these categories.

Static table-driven approaches are applicable to tasks that are periodic (or have been transformed into periodic tasks by well known techniques). Given task characteristics, a table is constructed, using one of many possible techniques (e.g., using various search heuristics), that identifies the start and completion times of each task and tasks are dispatched according to this table. This is a highly predictable approach but, is highly inflexible since any change to the tasks and their characteristics may require a complete overhaul of the table.

The approach traditionally used in non real-time systems is the *priority-based preemptive scheduling* approach. Here, tasks have priorities that may be statically or dynamically assigned and at any time, the task with the highest priority executes. It is the latter requirement that necessitates preemption: if a low priority task is in execution and a higher priority task arrives, the former is preempted and the processor is given to the new arrival. If priorities are assigned systematically in such a way that timing constraints can be taken into account, then the resulting scheduler can also be used for real-time systems. For example, using the rate-monotonic approach [36], utilization bounds can be derived such that if a set of tasks do not exceed the bound, they can be scheduled without missing any deadlines using such a *static priority-driven preemptive scheduler*.

Cyclic scheduling, used in many large-scale dynamic real-time systems [8], is a combination of both table-driven scheduling and priority scheduling. Here, tasks are assigned one of a set of harmonic periods. Within each period, tasks are dispatched according to a table that just lists the order in which the tasks execute. It is slightly more flexible than the table-driven approach because no start times are specified and it is amenable to a priori bound analysis – if maximum requirements of tasks in each cycle are known beforehand. However, pessimistic assumptions are necessary for determining these requirements. In many actual applications, rather than making worse-case assumptions, confidence in a cyclic schedule is obtained by very elaborate and extensive simulations of typical scenarios. This approach is both error-prone and expensive [40].

The *dynamic planning-based approaches* provide the flexibility of dynamic approaches with some of the predictability of approaches that check for feasibility. Here, after a task arrives, but before its execution begins, an attempt is made to create a schedule that contains the previously guaranteed tasks as well as the new arrival. If the attempt fails and if the attempt is made sufficiently ahead of the deadline, time is available to take alternative actions. This approach provides for predictability with respect to individual arrivals.

In contrast, if a purely priority-driven preemptive approach is used, say, by using task deadlines as priorities, and without any planning, a task could be preempted any time during its execution. In this case, until the deadline arrives, or until the task finishes, whichever comes first, we do not know whether a timing constraint will be met. This is the major disadvantage of the *dynamic best-effort approaches*. If, however, we can analyze the worst case performance characteristics of such a scheduler, then perhaps it can be recognized and avoided. Such worst-case analyses are in their infancy, being applicable to tasks with very simple characteristics [4].

2.3 Scheduling Algorithms For These Paradigms

The variety of performance metrics, scheduling approaches, and types of processing resources used by tasks imply a wide variety of scheduling algorithms. Before we delve into the algorithms, we note that most instances of the scheduling problem for hard real–time systems are computationally intractable. Here is a summary of the results discussed in [17, 18]. The problem of scheduling tasks with unit computation times and arbitrary precedence relationships on two processors and one resource is NP-complete; A polynomial time algorithm exists when the precedence relation is empty but arbitrary number of resources are present. But, for three processors and one resource, even with an empty precedence relationship, the problem is NP-complete. The generalized versions of the above problem are NP-complete even though for a limited number of cases polynomial time solutions exist. In summary, resource-constrained scheduling is an NP-complete problem, and the presence of precedence constraints exacerbates the problem. Hence, as we shall see in the rest of this section, many authors have examined the use of heuristics and approximation algorithms to deal with tasks that have complex requirements, including resource requirements and precedence constraints.

2.3.1 Static Table-Driven Scheduling

These approaches are motivated by the fact that resources needed to meet the deadlines of safety-critical tasks must be preallocated so that they can be guaranteed *a priori*. These tasks are usually statically scheduled such that their deadlines will be met even under worst-case conditions. For obvious reasons, these tasks are assumed to be periodic. (If they are not, assuming worst-case interarrival times, they can be converted into periodic arrivals.) For periodic tasks, there exists a feasible schedule if and only if there exists a feasible schedule for the LCM (the least common multiple) of the periods [30]. Given a set of periodic tasks, a typical algorithm that deals with multiprocessors or a distributed system attempts to assign subtasks of the tasks to processors or sites in the system and to construct a schedule of length LCM of the the task periods. At run-time, the set of tasks are repeatedly executed according to this schedule every LCM units of time.

If the tasks have simple characteristics, then a table can be constructed using the earliest-deadline-first (EDF), or the shortest-period-first technique. However, besides periodicity constraints, tasks may have resource requirements and can possess precedence, exclusion, communication, as well as replication constraints. In these cases, given the NP-completeness of the resulting scheduling problem, heuristics are resorted to [33]. Most of the algorithms adopt aspects from the branch-and-bound discipline in searching for a feasible schedule.

For instance, Xu and Parnas [73] have examined this scheduling problem for a task model where tasks are divided into subtasks and exclusion and precedence relations are specified among subtasks. It is applicable to multiprocessor systems. If an exclusion relation exists between two subtasks s_1 and s_2 then s_1's execution cannot be interrupted by s_2 and vice versa. Exclusion relations can be used to model resource access conflicts. The algorithm uses a branch and bound technique where an initial schedule that is based on ordering the tasks according to their deadlines is modified at each step to reduce the maximum lateness of the tasks. If a schedule is found with a maximum lateness that is zero or negative, then the schedule meets all the deadlines. If a feasible schedule is not found then the algorithm at least derives a schedule with the smallest maximum lateness.

The algorithm described in [47] considers communication and replication constraints and is applicable to distributed systems. It clusters subtasks of tasks based on the amount of communication involved between a pair of communicating subtasks and the computation time of the subtasks. Clustered subtasks are assigned to the same site, thereby eliminating the communication costs involved. A feasible schedule is determined by using a heuristic search technique that takes into account the various task characteristics, in particular, subtask computation times, communication costs, deadlines, and precedence constraints. Communication (between subtasks) on the communication channels in the system is also scheduled. This algorithm is designed for tasks whose subtasks may have to be executed on different sites, to cater, for example, to subtasks of a task having replication requirements. Further, the total computational requirements of subtasks of a task may be such that a single site may not be able to execute all of them within the period of the task. However, by distributing the subtasks, in particular, by exploiting the parallelism within a task, it may be possible to meet the periodicity requirements. Also, all resources needed by all the subtasks may not be available on any one site. The work described in [43] is also applicable to tasks with precedence and communication constraints and is a pure branch and bound search. Unlike in [47], all the subtasks of a task are scheduled to execute on the same site.

The primary criterion in the static scheduling of periodic tasks is *predictability*, i.e., determining a feasible schedule wherein all tasks meet their timing requirements, precedence constraints, etc. Under static allocation and scheduling, exactly when and where instances of a task will execute are fixed. But, if both periodic tasks as well as non-periodic tasks exist in a system, it will be advantageous to make some provision, during static scheduling, to cater to the needs of dynamic arrivals. For instance, some leeway could be provided such that the static schedules can be dynamically modified for better non-periodic task schedulability while retaining the feasibility of the critical task set. A scheme to achieve this is discussed in [51].

2.3.2 Priority-driven Preemptive Scheduling

Priority-driven preemptive-scheduling is the one used in most time-sharing systems. In non real-time systems, the priority of a job changes depending on whether it is CPU-bound or I/O-bound. In real-time systems, priority assignment is related to the time constraints associated with a job or task and this assignment can be either static or dynamic.

Liu and Layland [36] were perhaps the first to formally study priority-driven algorithms. They focussed on the problem of scheduling periodic tasks on a single processor and proposed two preemptive algorithms. The first algorithm, called the *Rate-Monotonic* (RM) algorithm assigns static priorities to tasks based on their periods. It assigns higher priorities to tasks with shorter periods. They showed that this scheme is optimal among static-priority schemes. This assignment is intuitively easy to understand. Liu and Layland also analyzed Earliest-Deadline-First (EDF), a dynamic priority assignment algorithm: The closer a task's deadline, the higher its priority. This again is an intuitive priority assignment policy.

Static priorities are attractive because a task's priority is assigned once it arrives and does not have to be reevaluated as time progresses. The RM priority assignment policy is applicable to periodic tasks. A dynamic priority assignment policy, however, can be applied to both periodic and aperiodic tasks. In contrast with static priorities, a task's dynamic priority may change when a new task, say with an earlier deadline, arrives. This makes the use of dynamic priorities more expensive in terms of run-time overheads.

The advantage of either of these two priority assignment policies is that, for periodic tasks, schedulability bounds, on resource utilization by the tasks, exist. In the case of the RM policy, a set of n tasks can be scheduled to meet its periodicity constraints on a uniprocessor provided the processor's utilization is no greater than $\ln 2$ for large n. Better bounds based on more exact characterization of the RM policy can be found in [32]. If the periods are harmonics of the smallest period, the bound is 1.00. In the case of EDF, the bound is always 1.00.

In addition to EDF, a task's laxity (given by the amount of time one can wait and still meet its deadline) can be used as its dynamic priority. This leads to the Least-Laxity-First algorithm. In fact, any function of the task's parameters can be used to assign priorities. We will see examples of these in Section 2.3.3.

Even though the RM policy has been in use by NASA in its software for the Apollo space missions, [36] is the first publication that gave a formal characterization and analysis of the RM policy. Then, after a long hiatus, it was picked up again by Sha, Lehoczky, and their colleagues, as well as Baker, among others, and extended in a variety of ways to deal with shared resources, aperiodic tasks, tasks with different importance levels, and mode changes. These are discussed in detail in [56].

Although feasibility checking or schedulability analysis is made easier by preemptive scheduling, (in terms of theoretical optimality and complexity), the checking generally either ignores the dispatching cost, or assumes it is a negligibly small constant. However, in an actual system, dispatching is more complicated involving preemption, context switching and readying the preempted task for future resumption. The dispatcher must also incorporate timer interrupts. The complexity of the dispatching process under nonpreemptive scheduling depends on whether the tasks are independent and whether there are resource constraints. The planning-based scheduling algorithms discussed next typically use nonpreemptive scheduling, partly motivated by the goal of reducing unnecessary preemptions.

2.3.3 Dynamic Planning-based Scheduling

Dynamic planning-based schedulers focus on dynamically performing feasibility checks. A task is *guaranteed* by constructing a plan for task execution whereby all guaranteed tasks meet their timing constraints. A task is guaranteed subject to a set of assumptions, for example, about its worst case execution time and resource needs, and the nature of faults in the system. If these assumptions hold, once a task is guaranteed it *will* meet its timing requirements. A guarantee algorithm must consider many issues including worst case execution times, resource requirements, timing constraints, the presence of periodic tasks, preemptable tasks, precedence constraints (which is used to handle task groups), multiple importance levels for tasks, and fault tolerance requirements. In a distributed system, when a task arrives at a site, the scheduler at that site attempts to guarantee that the task will complete execution before its deadline, on that site. If the attempt fails, the scheduling components on individual sites cooperate to determine which other site in the system has sufficient resource surplus to guarantee the task.

An algorithm to guarantee non-preemptable tasks arriving at a site given their arrival time, deadline or period, worst case computation time, and resource requirements is described in [49]. A task uses a resource either in shared mode or in exclusive mode and holds a requested resource as long as it executes. Using heuristics, a full feasible schedule for a set of tasks is constructed in the following way. Starting at the root of the search tree which is an empty schedule the algorithm tries to extend the schedule (with one more task) by moving to one of the vertices at the next level in the search tree until a full feasible schedule is derived. To this end, a heuristic function, H, which synthesizes various characteristics of tasks affecting real-time scheduling decisions is used to actively direct the scheduling to a plausible path. H is applied to at most k tasks that remain to be scheduled at each level of search. The task with the smallest value of function H is selected to extend the current partial schedule. If a partial schedule is found to be infeasible, it is possible to backtrack and then continue the search. If the value of k is constant (and in practice, k will be small when compared to the task set size n), the complexity is linearly proportional to n, the size of the task set [49]. While the complexity is proportional to n, the algorithm is programmed so that it incurs a fixed worst case cost by limiting the number of H function evaluations permitted in any one invocation of the algorithm. The paper also discusses how to choose k.

Dynamic algorithms that do not *a priori* know the arrival times of tasks cannot guarantee optimal performance [15]. But one dynamic algorithm can be considered better than another, if given a number of task sets for which feasible schedules exists, the former is able to find feasible schedules for more task sets than the latter. Extensive simulation studies of the algorithm show that a heuristic that combines deadline and resource requirement information works very well (see also [74, 75]) according to this performance criterion. Hence this algorithm has been implemented as part of the Spring kernel [63]. In [72], another dimension of the heuristic algorithm, namely, the bound on the length of the schedule compared to an algorithm that minimizes schedule length is derived.

With regard to cooperation between processing elements, several schemes have been reported in the literature [50, 24, 6]. We now discuss details of the four algorithms evaluated in [50]. They differ in the way a site treats a task that cannot be guaranteed locally: In the *random scheduling algorithm*, the task is sent to a randomly selected site; In the *focussed addressing algorithm*, the task is sent to a site that is estimated to have sufficient surplus resources and time to complete the task before its deadline; In the *bidding algorithm*, the task is sent to a site based on the bids received for the task from sites in the system; and, in the *flexible algorithm*, the task is sent to a site based on a technique that combines *bidding* and *focussed addressing*. These algorithms are compared, via simulations, relative to each other as well as with respect to two baselines. The first baseline is the non-cooperative algorithm where a task that cannot be guaranteed locally is not sent to any other site. The second is an (ideal) algorithm that behaves exactly like the bidding algorithm, but incurs no communication overheads. The fact that distributed scheduling improves the performance of a hard real-time system is attested by the better performance of the flexible algorithm compared to the non-cooperative baseline under all load distributions. The performance of the flexible algorithm is better than both the focussed addressing and bidding algorithms. However, the performance difference between the bidding algorithm and the flexible algorithm under small communication delays is negligible. The same can be said about the performance difference between the focussed addressing algorithm and the flexible algorithm under large communication delays. The random algorithm performs quite well compared to the flexible algorithm, especially when system load is low as well as when system load is high and the load is unevenly distributed. Under moderate loads, its performance falls short by a few percentage points which may be significant in a hard real-time system. Overall, the studies show that no algorithm outperforms all others in all system states. Though the flexible algorithm performs better than the rest in most cases, it is more complex than the other algorithms. Other details of the distributed scheduling algorithms can be found in [46, 50, 64]. The stability of these algorithms is discussed in [59].

2.3.4 Dynamic Best-Effort Scheduling

Best-effort scheduling is the approach used by many real-time systems deployed today. In such systems, a priority value is computed for each task based on the task's characteristics and the system schedules tasks according to their priority. Confidence is gained in the system via extensive simulations, in conjunction with recoding the tasks and priority adjustment.

Often used real-time scheduling algorithms, such as, earliest deadline first and least laxity first have optimal behavior as long as no overloads occur. However, experiments reported in [39] show that extreme performance degradation is encountered under overloads. But, since dynamic algorithms must perform well under varying loading conditions, the next task to execute or to discard in the case of an overload must be chosen carefully. The best-effort approach proposed in [39] tries to maximize the sum of the *values* of the tasks completed under overload condition where a task's value to the system depends on when it completes execution. Priority-driven preemptive scheduling is employed. Many different types of value functions are examined in [39], including shortest processing time first, earliest deadline first, least laxity first, first come first served, an algorithm that randomly chooses the next task to execute, as well as one that fixes a task's priority to be its highest possible value. In addition to the standard highest-priority-first scheduling algorithm, an algorithm which discards tasks with low value density, i.e., value per unit computation time, when an overload is considered likely, is also evaluated. As expected, the new algorithm improves performance under overloads. Dealing with overheads, in general, is a complex problem and solutions are still in their infancy [4, 5].

Clearly, the biggest disadvantage of dynamic best-effort algorithms lie in their lack of predictability and their suboptimality. A dynamic scheduling algorithm is said to be optimal if it always produces a feasible schedule whenever a clairvoyant algorithm, i.e., a static scheduling algorithm with complete prior knowledge of the tasks, can do so. For most real-world circumstances, optimal dynamic algorithms do not exist [15, 23, 9, 42]. However, recognizing that it will be useful to quantify the worst-case behavior of the dynamic algorithms, recently, there has been a surge of activity in this area. The results of this work can be useful in handling overloads effectively.

For example, [4] analyzes such bounds for the problem of preemptively scheduling sporadic task requests in both uniprocessor and multiprocessor environments. In the model considered, if a task is successfully scheduled to completion, a value equal to the task's execution time is imparted to the system; otherwise no value is obtained. It is proved that no dynamic scheduling algorithm can guarantee a cumulative value greater than a fourth of the value obtainable by a clairvoyant algorithm. (In fact, for the algorithm in [39], this ratio can be as low as zero.) Furthermore, the paper presents a dynamic scheduling algorithm TD_1 with this behavior, thus showing the bound to be tight. The paper also quantifies the relationship between the the amount of overloading permitted and the bound. Generalization of these results to two processors gives an upper bound of 1/2 which is tight in some very special cases. These results are just the beginning and have to be elaborated to apply to more interesting and useful situations in order for dynamic best-effort approaches to be employed by real-time systems that must be predictable.

2.4 Other Important Scheduling Issues

In this section we discuss two issues that are important in any real real-time system. They are: supporting fault tolerance, and improving performance by utilizing time left unused when tasks do not use all the time earmarked for them. A third issue concerns scheduling *imprecise computations*, computations in which a trade-off between the solution quality and timeliness can be achieved. Since a detailed discussion of imprecise computations appears in [38] we do not discuss it here.

2.4.1 Scheduling with Fault Tolerance Constraints

In this section, we examine some of the scheduling algorithms that explicitly take fault tolerance into account.

In [35], Liestman and Campbell propose a deadline mechanism that can guarantee that a primary task will make its deadline if there is no failure, and that an alternative task (of less precision) will run by the deadline if there is a failure. If the primary task executes then it is not necessary to run the alternative task and the time set aside for the alternative is reused. The paper deals with periodic tasks only and allows all tasks to be preempted. It is possible to precompute a tree of schedules (and back–up schedules) where the tree can be encoded within an efficient table-driven scheduler.

Krishna and Shin continue with this theme in [28]. Specifically, they want to be able to quickly switch to a new task schedule upon failure, where that new schedule has been precomputed. Off–line they use a dynamic programming algorithm to compute contingency schedules which are embedded within the primary schedule. In this approach they are able to ensure that hard deadlines are met in the face of some maximum number of failures. The embedded contingency schedules are not used unless there is a failure. However, the contingency schedules do represent a latent demand for processing time, thereby lowering the efficiency of the primary schedules to some extent. This is the price paid for having very little on–line processing time available to respond to failures. This paper also assumes that there is a need to conserve memory so that at most one contingency schedule per processor can be stored. In many of today's real–time systems memory constraints are still bottlenecks and therefore need to be accounted for in a manner illustrated in this paper. This paper also considers periodic tasks, but in contrast with [35] it does not suggest running some restricted and less accurate version of the task.

Approaches for fault tolerance, such as these two papers represent, are valuable for static, embedded computer systems where fault tolerance is extremely important and deadlines are very tight. In such cases, processor utilization is not important, rather guaranteeing the primary and contingency schedules are important. However, these static approaches are not suitable for many next generation real-time systems which must provide for predictability while reacting to the dynamics of the environment. Also, techniques are required that can trade-off fault-tolerance for timeliness, if an application allows such trade-offs, to handle overloads. For example, it is possible to combine the use of dynamic planning-based schedulers, to provide predictability, with the notion of imprecise computations, to effect the trade-offs. This brings to bear the power of the two complementary approaches to provide adaptive fault tolerance by focusing on the specific interaction between fault tolerance and scheduling. It allows the system to dynamically adapt the fault tolerance requirements of processes. Planning permits the *forecasting of timing errors*, supports graceful degradation, and allows dynamic trade-off analysis involving levels of redundancy and value accrued to the system.

2.4.2 Scheduling With Resource Reclaiming

The variance in tasks' execution times may result in some tasks completing earlier than expected by the scheduler. The task dispatcher can try to reclaim the time left by such early completion and utilize that to execute other tasks. Clearly, non real-time tasks can be executed in the idle time slots. But, more valuable will be an approach that improves the guaranteeability of tasks that have time constraints. Several issues must be considered to achieve this. When the actual computation time of a task differs from its worst case computation time in a nonpreemptive multiprocessor schedule with resource constraints, run time anomalies [19] may occur. These anomalies may cause some of the scheduled tasks to miss their deadlines. In particular, one cannot simply use any greedy or work-conserving dispatcher, one that will never leave a processor idle if there is a dispatchable task. For tasks with precedence constraints, Manacher [41] proposed an algorithm to avoid these anomalies by imposing additional precedence constraints on tasks to preserve the order of tasks which can run in parallel. Manacher's work was motivated by a need to make sure that the processors that execute task replicas (for fault tolerance) follow a consistent schedule even when tasks finish early. This is termed the stabilization problem.

Reclaiming unused time to improve the schedulability of dynamically arriving tasks is the motivation behind the work in [57]. Resource reclaiming algorithms used in systems that do dynamic planning-based scheduling must be *correct*, i.e, must maintain the feasibility of guaranteed tasks; must be *inexpensive*, i.e., the overhead cost of a resource reclaiming algorithm should be very low compared to tasks' computation times since a resource reclaiming algorithm is invoked whenever a task finishes; must have *bounded complexity*, in particular, it should be *independent* of the number of tasks in the schedule, so that its cost can be incorporated into the worst case computation time of a task; and must be *effective*, i.e, it should improve the performance of the system.

In [57] two resource reclaiming algorithms, are presented: Basic Reclaiming and Reclaiming with Early Start. These two algorithms employ strategies that are a form of dynamic *local optimization* of a feasible multiprocessor schedule. Both of these algorithms have *bounded* time complexity although Reclaiming with Early Start is more expensive to run than Basic Reclaiming. Simulation results demonstrate that these simple local optimizations can be very effective in improving the system performance in a dynamic real-time system and that resource reclaiming can compensate for the performance loss due to the worst case assumptions about the computation times of real-time tasks.

3 Real-Time Operating Systems

Real-time operating systems are an integral part of real-time systems. Not surprisingly, four main functional areas that they support are process management and synchronization, memory management, interprocess communication, and I/O. However, the manner in which they support these areas differs from conventional operating systems as will be discussed in this section. In particular, real-time operating systems stress predictability and include features to support real-time constraints. Three general categories of real-time operating systems exist: small, proprietary kernels (commercially available as well as *homegrown* kernels), real-time extensions to commercial timesharing operating systems such as UNIX, and research kernels. In this section we will survey these three main categories of real-time operating systems.

3.1 Small, Fast, Proprietary Kernels

The small, fast, proprietary kernels come in two varieties: *homegrown*[2] and commercial offerings[3]. Both varieties are often used for small embedded systems when very fast and highly predictable execution must be guaranteed. The homegrown kernels are usually highly specialized to the application. The cost of uniquely developing and maintaining a homegrown kernel, as well as the increasing quality of the commercial offerings is significantly reducing the practice of generating homegrown kernels. For both varieties of proprietary kernels, to achieve speed and predictability, the kernels are stripped down and optimized versions of timesharing operating systems. To reduce the run-time overheads incurred by the kernel and to make the system *fast*, the kernel

- has a fast context switch,
- has a small size (with its associated minimal functionality),
- responds to external interrupts quickly (sometimes with a guaranteed maximum latency to post an event but, generally, no guarantee is given as to when processing of the event will be completed; this later guarantee can *sometimes* be computed if priorities are chosen correctly),
- minimizes intervals during which interrupts are disabled,
- provides fixed or variable sized partitions for memory management (i.e., no virtual memory) as well as the ability to lock code and data in memory, and
- provides special sequential files that can accumulate data at a fast rate.

To deal with timing requirements, the kernel

- provides bounded execution time for most primitives,
- maintains a real-time clock,
- provides a priority scheduling mechanism,
- provides for special alarms and timeouts,

[2] Examples include [1, 22].

[3] Examples of commercials kernels include QNX, PDOS, pSOS, VCOS, VRTX32, and VxWorks.

- supports real-time queuing disciplines such as earliest deadline first and primitives for jamming a message into the front of a queue, and
- provides primitives to delay processing by a fixed amount of time and to suspend/resume execution.

In general, the kernels also perform multi-tasking and inter-task communication and synchronization via standard, well-known primitives such as mailboxes, events, signals, and semaphores. While all these latter features are designed to be fast, fast is a relative term and not sufficient when dealing with real–time constraints. Nevertheless, many real-time system designers use these features as a basis upon which to build real–time systems. This has been effective in small embedded applications such as instrumentation, communication front ends, intelligent peripherals and many areas of process control. Since these applications are simple it is relatively easy to show that all timing constraints are met. Consequently, the kernels provide exactly what is needed. However, as applications become more complex it becomes more and more difficult to *craft a solution* based on priority-driven scheduling where all timing, computation time, resource, precedence, and value requirements are mapped to a single priority for each task. In these situations demonstrating predictability becomes very difficult. For example, a task may block when it attempts to access a semaphore, new tasks may be dynamically invoked at higher priorities, messages may not be available when a task begins execution, events may be posted very quickly but there may be no guarantee that the processing required to respond to the event will execute in time, etc. Given this large amount of asynchrony, concurrency, and blocking, the unfortunate implementor is required to assign the proper priorities that ensures the system always meets all of its deadlines. Because of these reasons, some researchers believe that current kernel features provide almost no direct support for solving the difficult timing problems, and would rather see more sophisticated kernels that directly address timing and fault tolerance constraints.

Recently there have been efforts to produce seamless real-time kernels that scale from the small, proprietary kernels to large kernels that support the full POSIX/UNIX interfaces. The idea is to let the user select trade-offs in size, performance and functionality depending on the application. The lowest level of support is being called a nanokernel or alternatively a microkernel.

3.2 Real-Time Extensions to Commercial Operating Systems

A second approach to real-time operating systems is the extension of commercial products, e.g., extending UNIX to RT-UNIX [16], or POSIX to RT-POSIX, or MACH to RT-MACH [69], or CHORUS to a real-time version [10]. The real-time version of commercial operating systems are generally slower and less predictable than the proprietary kernels, but have greater functionality and better software development environments – very important considerations in many applications. Another significant advantage is that they are based on a set of familiar interfaces (standards) that facilitate portability. For

UNIX, since many variations of UNIX have evolved, a new standards effort, called POSIX, has defined a common set of user level interfaces for operating systems. In particular, the POSIX P.1003.4 subcommittee is defining standards for real-time operating systems. To date, the effort has focused on eleven important real-time related functions: timers, priority scheduling, shared memory, real-time files, semaphores, interprocess communication, asynchronous event notification, process memory locking, asynchronous I/O, synchronous I/O, and threads.

Various problems exist when attempting to convert a non real-time operating system to a real-time version. These problems can exist both at the system interface as well as in the implementation. For example, in UNIX interface problems exist in process scheduling due to the **nice** and **setpriority** primitives and its round robin scheduling policy. In addition, the timer facilities are too coarse, memory management (of some versions) contains no method for locking pages into memory, and interprocess communication facilities don't support fast and predictable communication. The implementation problems include intolerable overhead, excessive latency in responding to interrupts, partly but very importantly, due to the non-preemptability of the kernel, and internal queues are FIFO. These and other problems can and have been solved to result in a real-time operating system that is used for both real-time and non real-time processing. However, because the underlying paradigm of timesharing systems still exists users must be careful not to use certain non real-time features that might insidiously impact the real-time tasks. For example, in [16] they list over 60 RT-UNIX system calls that are not recommended to be used when running a real-time application. This is very disturbing because in converting from UNIX to RT-UNIX the following aspects were changed: scheduling, interrupt handling, IPC, the file system, I/O support, how the user controls resource use, timer facilities, memory management and the basic synchronization assumptions of the kernel. The juxtaposition of changing almost everything and then ending up with over 60 system calls that should *still* not be used, should lead us to question whether extending a commercial timesharing OS is the correct approach. We believe that it is not the correct approach because too many basic and inappropriate underlying assumptions still exist. This includes:

- optimizing for the average case (rather than worst case),
- assigning resources on demand,
- ignoring most if not all semantic information about the application, and
- independent CPU scheduling and resource allocation possibly causing unbounded blocking.

On the other hand, the trend to begin with a completely new implementation of UNIX based on microkernels may reduce or eliminate some of the above problems. Consider several more detailed examples from MACH.

MACH is heavily based on lazy evaluation, meaning that you never do anything until it is really needed. One example of this strategy is copy-on-write. Here either a message or part of an address space is not actually copied at the message send time or at address space create time, respectively, but delayed until that message (memory) is actually accessed. On the average this provides excellent performance. The problem is that large amounts of execution time may be required at the *wrong* time to finally perform the copy, causing a task to miss a deadline. Basically, it cannot be predicted as to when slowdowns will occur. Can all forms of lazy evaluation be eliminated to push MACH towards predictability? Yes, but it is difficult because of the overpowering integration of this philosophy in the kernel. Virtual memory is another problem. It is possible to lock pages in memory to remove some of the unpredictability (except, it is nontrivial to decide when to lock and unlock, accounting for the cost of the lock and unlock, and ensuring that the pages are locked in time). Does locking pages, by itself, make the virtual memory part of the system predictable? What about unpredictabilities due to the memory map tables (lookup and maintenance), the MMU TLB entries (present or not), hash table entries used for quick lookup (access time in the table), and indirect problems such as how by locking many pages we might effect the performance of both real-time and non real-time threads needing pages now being drawn from a smaller pool? Valuable real-time features that were added to MACH include real-time threads, real-time synchronization primitives, support for priority inheritance, and real-time scheduling, but all of these are still embedded in a timesharing paradigm.

Another fundamental problem with the timesharing paradigm is that these operating systems want to remove control over resources from the application. Such operating systems consider it their prerogative to decide who should get resources for the best average case performance. For example, a multi-level feedback queue will modify the user specified priorities to balance I/O and CPU performance. After a real-time application designer goes through torture to map all the complexities of his application into a set of priorities, if the system adjusts these priorities, then the analysis and evaluation were for naught. Allowing fixed priorities or another real-time scheduling algorithm helps, but insidious interactions from the non real-time threads, through their resource use and scheduling policy, might slow down the real-time tasks (in some unanticipated way).

Given all these problems for RT-UNIX or RT-MACH can they be used in real-time applications? Yes, certainly for real-time applications where missing a deadline has *no* severe consequences, they can be used. If deadlines must be guaranteed to be met, these operating systems can still be used *if* the designers can hand craft a set of priorities that will always work. For example, given 5 independent periodic tasks with certain periods and deadlines, running only these at fixed priorities on these operating systems can easily be shown to work (however, it would be just as easy to use the proprietary kernels). As we add aperiodics, interrupts from the environment, shared data structures, precedence constraints between tasks, non real-time background processing, etc. assigning priorities such that it will always work becomes difficult and the designer is still not certain that lurking problems don't exist due to the underlying timesharing design. Such an approach typically has very high cost and is very difficult to maintain.

3.3 Research Operating Systems

While many real-time applications will continue to be constructed with proprietary real-time kernels and with extensions to commercial timesharing operating systems, as discussed above, significant problems still exist. In particular, the proprietary kernels have difficulty when scaling to large applications, and the timesharing extensions emphasize speed rather than predictability, thereby perpetuating the myth that real-time computing is fast computing [61]. Trends in the current research in real-time operating systems include:

- identifying that new approaches are needed which challenge the basic assumptions made by timesharing operating systems and developing those new paradigms,
- developing real-time process models
  - some systems use the standard process model both to program with and at execution time,
  - some systems use the process model to program with but translate into a different run time model to help support predictability and on-line guarantees,
  - some systems use real-time threads,
- developing real-time synchronization primitives such as those that support priority inheritance,
- developing solutions that facilitate timing analysis of both the initial system and upon modifications (the real-time scheduling algorithms play a large role here),
- strongly emphasizing predictability not only of the kernel but also providing good support for application level predictability,
- retaining significant amounts of application semantics at run time,
- developing support for fault tolerance,
- investigating object oriented approaches,
- providing support for multiprocessor and distributed real-time systems including end-to-end timing constraints, and
- attempting to define a real-time micro-kernel.

In the remainder of this section we will briefly survey several research projects to provide a brief idea about the scope and type of research that is ongoing. The projects chosen here are representative of a much wider set of work in the field.

3.3.1 MARS

The MARS kernel [13, 27] offers support for controlling a distributed application based entirely on time events (rather than asynchronous events) from the environment. Emphasis is placed on an *a priori* static analysis to demonstrate that all the timing requirements are met. An important feature of this system is that flow control on the maximum number of events that the system handles is automatic and this fact contributes to the predictability analysis. This system is based on a paradigm, i.e., the time driven model, that is different than what is found in timesharing systems. The scheduling approach is static table-driven. Support for distributed real-time systems includes a hardware based clock synchronization algorithm and a TDMA-like protocol to guarantee timely message delivery.

3.3.2 SPRING

The Spring kernel [63] contains real-time support for multiprocessors and distributed systems. A novel aspect of the kernel is the dynamic planning-based scheduling of tasks that arrive dynamically. This takes tasks' time and resource constraints into account and *avoids* the need to *a priori* compute worst case blocking times. Safety-critical tasks are dealt with via static table-driven scheduling. The kernel also embodies a reflective architecture that retains a significant amount of application semantics at run time. This approach provides a high degree of flexibility and graceful degradation. These planning and application semantic features are integrated to provide direct support for achieving both application and system level predictability. The kernel also uses global replicated memory to achieve predictable distributed communication. The abstractions provided by the Kernel include guarantee, reservation, planning, and end-to-end timing support. Spring, like MARS, presents a new paradigm for real-time operating systems, but unlike MARS (to date), it strives for a more flexible combination of off-line and on-line techniques.

3.3.3 MARUTI

The MARUTI system [21] focuses on support for dynamic on-line guarantees that tasks will make their deadlines and on fault tolerance. It is object based and supports distributed systems. Each object has service access points which are the operations (services) that the object provides. Information about objects such as their computation times and deadlines are retained with the objects to be used by the dynamic planning-based scheduler. When an object is invoked the scheduler determines if the object can be guaranteed to meet its timing constraint. If so, the schedule for it is added to a calendar that represents the deterministic manner in which the object will execute and all resources the object will require are reserved. MARUTI has been designed in a top down fashion with a goal of demonstrating principles. As such, the actual implementation is high level and runs on top of UNIX. An implementation in native mode is underway.

3.3.4 ARTS

The ARTS kernel [68] provides a distributed real-time computing environment that works in conjunction with the static priority-driven preemptive scheduling paradigm. The kernel supports the notion of real-time objects and real-time threads. Each real-time object is time encapsulated. This is enforced by a time fence mechanism which provides a run time check that ensures that the slack time is greater than the worst case execution time for an object invocation about to be performed. If it is, the operation proceeds, else it is aborted. Each real-time thread can have a value function, timing constraints, worst case execution time, phase, and delay value associated with it. Communication (object invocation) proceeds in a request-accept-reply fashion, but does not address deadlines for messages. A real-time transport protocol has been developed, but is not yet included in the ARTS kernel. The ARTS kernel is also tied to various tools that *a priori* analyze the system wide schedulability of the system.

3.3.5 CHAOS

The CHAOS system [53] represents an object based approach to real-time kernels. This approach allows easy creation of a family of kernels, each tailored to a specific hardware or application. This is important because real-time applications vary widely in their requirements and it would be beneficial to have one basic paradigm for a wide range of applications. The family of kernels is based on a core that supports a real-time threads package. This core is the machine dependent part. Virtual memory regions, synchronization primitives, classes, objects, and invocations all comprise additional support provided in each kernel. One of the investigated scheduling approaches is guarantee-oriented, employing a variation of the preemptive deadline-first scheduling algorithm for its feasibility checking [6]. Unlike the scheduling approach used in Spring in which both timing and functionality of a task are guaranteed, here, it is verified that a set of tasks can meet their deadline requirements based on optimistic assumptions about resource availability, for instance. Thus, depending on blocking for resources, a task may not achieve its desired functionality, even though it will meet its timing constraint.

3.3.6 HARTOS

The Hexagonal Architecture for Real-Time Systems (HARTS) consists of multiple sites connected by a hexagonal mesh network. Each site may be a uniprocessor or multiprocessor and contains an intelligent network processor. The intelligent network processor handles much of the low level communication functions. An experimental operating system called HARTOS [26] is a distributed real-time kernel running on HARTS. On each site HARTOS runs in conjunction with the commercial uniprocessor OS, pSOS, so, by itself, is not a full operating system. Rather, HARTOS focuses on interprocess communication, thereby providing some support for distributed real-time systems. In particular, HARTOS supports message send and receive, non-queued event signals, reliable streams, and message scheduling that provides a best-effort approach in delivering a message by its deadline. Support for fault tolerant routing, clock synchronization, and for replicated processes are planned for the future.

3.3.7 DARK

Ada is mandated to be used in embedded real-time systems for many DoD projects. The Distributed Ada Real-Time Kernel (DARK) [71] has been developed to provide support for execution of Ada applications in a distributed real-time environment. The kernel supports both Ada tasks and kernel processes which are outside of the Ada run time environment. For real-time control, the application programmer, writing in Ada, deals directly with kernel processes and the kernel's scheduler by appropriate declarations. The scheduler is based on the dynamic best-effort paradigm, where a simple highest priority first scheduler is used. DARK also implements layers 2 through 4 of the standard ISO reference model to support distributed communication. There are no special time related services provided in the interprocess communication implementation. The goal of DARK was to provide a near-term option of how to use Ada in a distributed real-time system.

4 Summary

This paper presents a categorized summary of work in the areas of scheduling and operating systems for real-time applications. In particular, four scheduling paradigms were identified: static table-driven scheduling, static priority preemptive scheduling, dynamic planning-based scheduling, and dynamic best-effort scheduling. Real-time operating systems were categorized into three classes: small, proprietary kernels, real-time extensions to commercial operating systems, and research kernels. Rather than being exhaustive, we have provided specific examples from each of the categories. Exciting developments and serious limitations of the current work both in scheduling and operating systems was also noted. Important interactions between scheduling algorithm development and operating systems exist. For example, whereas scheduling is an integral part of any real-time operating system, barring a few exceptions, most scheduling work has ignored the overheads involved in scheduling. As we saw, for predictability, it is essential to account for all the overheads involved. This is another area where there are new challenges.

It is also important to point out that in several real-world applications, there exists end-to-end timing constraints with respect to computations that span many processing sites. Allocation and scheduling the communication as well as processing resources in an integrated fashion still remains a problem awaiting efficient and flexible solutions. Furthermore, many applications with end-to-end constrains have prababilistic requirements. For example, in telephone switching, it is required to establish x percentage of the connections within y amount of time. Schemes to meet such performance requirements and methodical approaches for showing that the requirements will be met are also worthy of further exploration.

References

[1] L. Alger and J. Lala. Real–Time Operating System For A Nuclear Power Plant Computer. *Proc. Real–Time Systems Symposium*, December 1986.

[2] T. Baker. Stack-Based Scheduling of Real-Time Processes. *Real-Time Systems* 3(1):67-100, March 1991.

[3] J. A. Bannister and K. S. Trivedi. Task Allocation in Fault-tolerant Distributed Systems. *Acta Informatica*, pp. 261-81, Springer-Verlag, 1983.

[4] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, F. Wang. On the Competitiveness of On-Line Real-Time Scheduling. *Proc of the Real-Time Systems Symposium*, December 1991.

[5] S. Biyabani, J.A. Stankovic and K. Ramamritham. The Integration of Deadline and Criticalness in Hard Real-Time Scheduling. *Proc. Real-Time Systems Symposium*, December 1988.

[6] B. A. Blake and Karsten Schwan. Experimental Evaluation of a Real-Time Scheduler for a Multiprocessor System. *IEEE Transactions on Software Engineering*, 17(1), January 1991.

[7] J. Blazewicz, W. Cellary, R. Slowinski and J. Weglarz. Scheduling under Resource Constraints – Deterministic Models Annals of Operations Research. J.C. Baltzer AG Scientific Publishing Company, 1986.

[8] G.D. Carlow. Architecture of the Space Shuttle Primary Avionics Software System. *CACM* 27(9), September 1984.

[9] H. Chetto and M. Chetto. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transactions on Software Engineering*, 15(10), October 1989.

[10] Chorus Kernel v3 r4.0 Programmer's Reference Manual. Technical Report CS/TR-91-71, Chorus systemes, 91/09.

[11] W. Chu and L. Lan. Task Allocation and Precedence Relations for Distributed Real-Time Systems. *IEEE Transactions on Computers*, C-36(6), June 1987.

[12] E.G. Coffman (ed.). *Computer and Job/Shop Scheduling Theory*, Wiley, New York, 1976.

[13] A. Damm, J. Reisinger, W. Schnakel, and H. Kopetz. The Real-Time Operating System of Mars. *Operating Systems Review*, pp. 141-157, July 1989.

[14] S. Davari and S. K. Dhall. An On Line Algorithm for Real-Time Tasks Allocation. In *Proc. Real-Time Systems Symposium*, Computer Society Press, Washington, DC, December 1986.

[15] M. L. Dertouzos and A. K-L. Mok. Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks. *IEEE Transactions on Software Engineering*, 15(12), December 1989.

[16] Furht, B., D. Grostick, D. Gluch, G. Rabbat, J. Parker, and M. Roberts. *Real-Time Unix Systems.* Kluwer Academic Publishers, Norwell, Massachusetts, 1991.

[17] M.R. Garey and D.S. Johnson. Complexity Results for Multiprocessor Scheduling Under Resource Constraints. *SIAM Journal of Computing* 4, 1975.,

[18] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[19] R.L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.*, 17(2), March 1969.

[20] R.L. Graham, E. L. Lawler, J.K. Lenstra and A.H.G.R. Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics* 5, 1979.

[21] O. Gudmundsson, D. Mose, K. Ko, A. Agrawala, and S. Tripathi. MARUTI, An Environment for Hard Real-Time Applications. in *Mission Critical Operating Systems*, edited by A. Agrawala, K. Gordon and P. Hwang, IOS Press, 1992.

[22] V. P. Holmes, D. Harris, K. Piorkowski, and G. Davidson. Hawk: An Operating System Kernel for a Real–Time Embedded Multiprocessor. Sandia National Labs Report, 1987.

[23] K.S. Hong and J.Y-T. Leung. On-Line Scheduling of Real-Time Tasks. *Proc. Real-Time Systems Symposium*, December 1988.

[24] C-J. Hou and K. G. Shin, Load Sharing with Considerations of Future Arrivals in Heterogeneous Distributed Real-Time Systems. *Proc. Real-Time Systems Symposium*, pp. 94-103, December 1991.

[25] D. Jensen. The Kernel Computational Model of the Alpha Real-Time Distributed Operating System. in *Mission Critical Operating Systems*, edited by A. Agrawala, K. Gordon and P. Hwang, IOS Press, 1992.

[26] D. Kandlur, D. Kiskis, and K. Shin. A Real-Time Operating System for HARTS. in *Mission Critical Operating Systems*, edited by A. Agrawala, K. Gordon and P. Hwang, IOS Press, 1992.

[27] H. Kopetz, A. Demm, C. Koza and M. Mulozzani. Distributed Fault Tolerant Real-Time Systems: The Mars Approach. *IEEE Micro*, pp. 25-40, 1989.

[28] C.M. Krishna and K.G. Shin. On Scheduling Tasks with a Quick Recovery from Failure. *IEEE Transactions on Computers*, C-35(5):448-55, May 1986.

[29] E. Lawler. Recent Results in the Theory of Machine Scheduling. *Mathematical Programming: The State of the Art*, A. Bachem et al (eds), Springer-Verlag, New York, 1983.

[30] E. L. Lawler and C. U. Martel. Scheduling Periodically Occurring Tasks on Multiple Processors. Information Processing Letters, 12(1), February 1981.

[31] J.P. Lehoczky, L. Sha and J. Strosnider. Enhancing Aperiodic Responsiveness in a Hard Real-Time Environment *Proc. Real-Time Systems Symposium* 1987.

[32] J.P. Lehoczky, L. Sha and Y. Ding. The Rate Monotone Scheduling Algorithm: Exact Characterization and Average Case Behavior. *Proc. IEEE Real-Time Systems Symposium*, pp. 166-71, December 1989.

[33] D.W. Leinbaugh. Guaranteed Response Time in a Hard Real-Time Environment. *IEEE Transactions on Software Engineering*, SE-6, January 1980.

[34] J. K Lenstra, A. H. G.R. Kan and P. Bruchker. Complexity of Machine Scheduling Problems. *Annals of Discrete Mathematics*, North–Holland, New York, 1977.

[35] A.L. Liestman and R.H. Campbell. A Fault Tolerant Scheduling Problem. *IEEE Transactions on Software Engineering*, SE-12(11):1089-95, November 1986.

[36] C.L. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46-61, 1973.

[37] J.W.S. Liu, K. Lin, W. Shih, A. Yu, J. Chung and W. Zhao. Algorithms for Scheduling Imprecise Calculations. *IEEE Computer*, 24(5):58–68, May 1991.

[38] J.W.S. Liu *et. al.*, Imprecise Computations. *Proceedings of the IEEE*, (this issue).

[39] C.D. Locke. Best-Effort Decision Making for Real-Time Scheduling. *Ph.D. Thesis* Carnegie Mellon University, Pittsburgh, PA., May 1985.

[40] C. D. Locke. Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives. *Real- Time Systems*, Kluwer Academic Publishers, 4(1), March 1992.

[41] G. K. Manacher. Production and Stabilization of Real-Time Task Schedules. *Journal of the ACM*, 14(3), 1967.

[42] A. K. Mok. Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment. *Ph.D Dissertation*, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1983.

[43] D.T.Peng and K.G.Shin. Static Allocation of Periodic Tasks with Precedence Constraints in Distributed Real-time Systems. *Proc. of the Intl Conf. on Distributed Computing*, 190-198, June 1989.

[44] R. Rajkumar, L. Sha and J. Lehockzy. Real-Time Synchronization Protocols for Multiprocessors *Proc. Real-Time Systems Symposium*, 1988.

[45] R. Rajkumar, L. Sha, J.P. Lehoczky and K. Ramamritham. An Optimal Priority Inheritance Protocol for Real-Time Synchronization, submitted for publication, April 1991.

[46] K. Ramamritham and J. A. Stankovic. Dynamic Task Scheduling in Distributed Hard Real-Time Systems. *IEEE Software*, 1(3):65-75, July 1984.

[47] K. Ramamritham. Allocation and Scheduling of Complex Periodic Tasks. *10th Intl Conf. on Distributed Computing Systems*, Paris, France, June 1990.

[48] K. Ramamritham and J.A. Stankovic. Scheduling Strategies Adopted in Spring: An Overview. *Foundations of Real-Time Computing: Scheduling and Resource Management*, edited by Andre van Tilborg and Gary Koob.

[49] K. Ramamritham, J.A. Stankovic and P. Shiah. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184-94, April 1990.

[50] K. Ramamritham, J.A. Stankovic and W. Zhao. Distributed Scheduling of Tasks with Deadlines and Resource Requirements. *IEEE Transactions on Computers*, 38(8):1110-23, August 1989.

[51] Ramamritham, K., G. Fohler and J. M. Adan, Issues in the Static Allocation and Scheduling of Complex Periodic Tasks, *10th IEEE Workshop on Real-Time Operating Systems and Software*, May 1993.

[52] J. Ready, VRTX: A Real–Time Operating System for Embedded Microprocessor Applications, *IEEE Micro*, pp. 8-17, August 1986.

[53] K. Schwan, A. Geith, and H. Zhou. From $Chaos^{base}$ to $Chaos^{arc}$: A Family of Real-Time Kernels. *Proc. Real–Time Systems Symposium*, pp. 82-91, December 1990.

[54] L. Sha, R. Rajkumar and J. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(3):1175-85, 1990.

[55] L. Sha, R. Rajkumar, J. Lehoczky and K. Ramamritham. Mode Change Protocols for Priority-Driven Preemptive Scheduling. *Real-Time Systems* 1(3):243-64, December 1989.

[56] L. Sha *et al*, Generalized Rate-monotonic Scheduling Theory: A Framework for Developing Real-time Systems. *Proceedings of the IEEE*, (this issue).

[57] C. Shen, K. Ramamritham and J.A. Stankovic. Resource Reclaiming in Multiprocessor Real-Time Systems. (to appear in) *Transactions on Parallel and Distributed Systems*, 1993.

[58] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Real-Time Systems*, 1(1):27–60, 1989.

[59] J.A. Stankovic. Stability and Distributed Scheduling Algorithms. *IEEE Transactions on Software Engineering*, SE-11(10):1141-52, 1985.

[60] J.A. Stankovic. Decentralized Decision Making for Task Allocation in a Hard Real-Time System. *IEEE Transactions on Computers*, March 1989.

[61] J. A. Stankovic, Misconceptions About Real–Time Computing. *IEEE Computer*, Vol. 21, No. 10, October 1988.

[62] J. A. Stankovic, and K. Ramamritham. The Spring Kernel: A New Paradigm for Real–Time Operating Systems, *ACM Operating Systems Review*, Vol. 23, No. 3, pp. 54-71, July, 1989.

[63] J.A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Hard Real-Time Operating Systems. *IEEE Software*, 8(3):62-72, May 1991.

[64] J.A. Stankovic, K. Ramamritham and S. Cheng. Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems. *IEEE Transactions on Computers*, C-34(12):1130-43, 1985.

[65] J. A. Stankovic. SpringNet: A Scalable Architecture For High Performance, Predictable, Distributed, Real-Time Computing. Univ. of Massachusetts, Technical Report, 91-74, October 1991.

[66] J. A. Stankovic, F. Wang, and K. Ramamritham, Dynamic Scheduling to Support Adaptive Fault Tolerance in Real-Time Systems, submitted for publication, December 1992.

[67] SYSTRAN Corporation, Scramnet Network Reference Manual, Dayton, Ohio, 45432.

[68] H. Tokuda, and C. Mercer. ARTS: A Distributed Real–Time Kernel. *ACM Operating Systems Review*, Vol. 23, No. 3, July, 1989.

[69] H. Tokuda, T. Nakajima and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. *Proc. Usenix Mach Workshop*, October 1990.

[70] H. Tokuda, J. Wendorf and H. Wang. Implementation of a Time Driven Scheduler for Real-Time Operating Systems. *Proc. Real-Time Systems Symposium*, Computer Society Press, Washington, DC, December 1987.

[71] R. van Scoy, J. Bamberger, and R. Firth. An Overview of DARK. in *Mission Critical Operating Systems*, edited by A. Agrawala, K. Gordon and P. Hwang, IOS Press, 1992.

[72] F. Wang, K. Ramamritham and J.A. Stankovic. Bounds on the Schedule Length for Some Heuristic Scheduling Algorithms for Hard Real-Time Systems. *Proc. Real-Time Systems Symposium*, 1992.

[73] J. Xu and L. Parnas. Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations. *IEEE Transactions on Software Engineering*, pp. 360-69, March 1990.

[74] W. Zhao and K. Ramamritham. Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints. *Journal of Systems and Software* 7:195-205, 1987.

[75] W. Zhao, K. Ramamritham and J.A. Stankovic. Scheduling Tasks with Resource Requirements in Hard Real-Time Systems. *IEEE Transactions on Software Engineering*, SE-12(5):567-77, 1987.

[76] W. Zhao, K. Ramamritham and J.A. Stankovic. Preemptive Scheduling Under Time and Resource Constraints. *IEEE Transactions on Computers* C-36(8):949-60, August 1987.