Real Time Scheduling Theory: A Historical Perspective

Lui Sha (Editor),University of Illinois at Urbana Champaign
Tarek Abdelzaher, University of Virgina
Karl-Erik Årzén and Anton Cervin, Lund Institute of Technology
Theodore Baker, Florida State University
Alan Burns, University of York
Giorgio Buttazzo, University of Pavia
Marco Caccamo, University of Illinois at Urbana Champaign
John Lehoczky, Carnegie Mellon University
Aloysius K. Mok, University of Texas at Austin

Abstract. In this 25th year anniversary paper for the IEEE Real Time Systems Symposium, we review the key results in real time scheduling theory and the historical events that led to the establishment of the current real time computing infrastructure. We conclude this paper by looking at the challenges ahead of us.

Keywords: fixed priority scheduling, rate monotonic scheduling, dynamic priority scheduling, earliest deadline first, soft real time, hard real time, feedback, scheduling models, new challenges.

1. Introduction

A real-time system is one with explicit deterministic or probabilistic timing requirements. Historically, real-time systems were scheduled by cyclic executives, constructed in a rather *ad hoc* manner. During the 1970's and 1980's, there was a growing realization that this static approach to scheduling produced systems that were inflexible and difficult to maintain.

This understanding led to the launch in the 1980's, under the leadership of Andre van Tilborg, of a Real Time Systems Initiative by the United States Office of Naval Research. Many important early results reported here are the results of this research initiative. During this same period, a number of individuals and organizations cooperated to develop a new real-time computing infrastructure that consists of a fixed-priority scheduling theory, a suite of supporting hardware and software open standards, educational and professional training materials, and commercially available schedulability analysis tools. In the early 1990s a number of other initiatives were funded, including a series of influencial studies commissioned by the European Space Agency. Text books also began to appear, *e.g.*, Burns and Wellings [45].

The transition from cyclical executive based infrastructure to fixed-priority scheduling theory based infrastructure was ushered in by an integrated research and technology transition effort led by Lui Sha, John Lehoczky and John Goodenough at the Carnegie Mellon University and its Software Engineering Institute. This effort consisted of: 1) identifying model problems from challenges encountered in advanced system development; 2) creating theoretical results for the model problems; 3) providing consultation support to industry in the application of new results; 4) developing timely professional training materials, for example, the handbook for practitioners by Klein *et al* [111]; 5) revising open standards on real-time computing to support the application of the fixed-priority scheduling theory.

Building upon the seminal work of Liu and Layland [136], this integrated effort produced the main body of the fixed priority scheduling results reported here. In addition, there were notable and timely successes in the application of this theory to national high technology projects including a Global Positioning Satellite software upgrade [73] and the Space Station. In 1991 the US Department of Defense (DoD) Software Technology Strategy report called the development of generalized Rate Monotonic Scheduling theory a major payoff of DoD-sponsored research. The report stated that "system designers can use this theory to predict whether task deadlines will be met long before the costly implementation phase of a project begins. It also eases the process of making modifications to application software." In his October 1992 Speech, "Charting The Future: Challenges and Promises Ahead of Space Exploration", Aaron Cohen, Deputy Administrator of the US National Aeronautics and Space Administration (NASA), said, "through the development of Rate Monotonic Scheduling, we now have a system that will allow [International Space Station] Freedom's computers to budget their time, to choose between a variety of tasks, and decide not only which one to do first but how much time to spend in the process."

The successes in practice provided the momentum to revise the open systems standards and create a coherent set of hardware and software standards to support fixed-priority-theory-based real-time computing. Before these revisions, real-time computing standards were *ad hoc*. They suffered from priority inversion problems, and had an inadequate number of priority levels to support real-time applications of fixed-priority scheduling. A stanards-driven transformation of the real-time computing infrastructure started with solving the priority inversion problem in Ada [190], and in providing sufficient priority levels in Futurebus+ [195]. Today, all major open standards on real-time computing support fixed-priority scheduling.

Since this initiative, there has been an explosion of interest in real-time systems, and an explosion of research and publication on the analysis of real-time scheduling. In this paper, we briefly review five important areas of real-time scheduling theory: 1) fixed-priority scheduling, 2) dynamic-priority scheduling, 3) soft real-time applications, 4) feedback scheduling and 5) extended scheduling models. Finally, we take a look at some of the new challenges ahead of us.

2. Fixed-Priority Scheduling

Theodore Baker and Alan Burns

The notion of priority is commonly used to order access to the processor and other shared resources such as communication channels. In priority scheduling, each job is assigned a priority via some policy. Contention for resources is resolved in favor of the job with the higher priority that is ready to run. In discussions of fixed task priority scheduling we use the value one for the highest priority and larger integers for lower priorities, and number the tasks so that task τ_i has priority i.

When priorities are assigned to tasks, all the jobs of a task will have the same priority. Task-level priority assignments are also known as fixed-priority scheduling or as generalized rate-monotonic scheduling, because of historical reasons. A system may still be analyzed under the assumption of fixed priorities if the changes in priority only occur at major epochs, such as system "mode changes", or if the changes only apply to short time intervals, such as critical sections.

2.1. The Liu and Layland Analysis

In 1973, Liu and Layland published a paper on the scheduling of periodic tasks that is generally regarded as the foundational and most influential work in fixed-priority real-time scheduling theory [136]. They started with the following assumptions [136]:

(i) all tasks are periodic;

(ii) all tasks are released at the beginning of period and have a deadline equal to their period;

(iii) all tasks are independent, i.e., have no resource or precedence relationships;

(iv) all tasks have a fixed computation time, or at least a fixed upper bound on their computation times, which is less than or equal to their period;

(v) no task may voluntarily suspend itself;

(vi) all tasks are fully preemptible;

(vii) all overheads are assumed to be 0;

(viii) there is just one processor.

Under this model, a *periodic* task is time triggered, with a regular release time. The length of time between releases of successive jobs of task τ_i is a constant, T_i, which is called the *period* of the task. Each job has a deadline D_i time units after the release time. A task is said to have hard deadline if every job must meet its deadline. A task may also have an *offset,* from system start-up, for the first release of the task.

Feasibility analysis is used to predict temporal behavior via tests which determine whether the temporal constraints of tasks will be met at run time. Such analysis can be characterized by a number of factors including the constraints of the computational model (*e.g.*, uniprocessor, task independence, *etc.*) and the coverage of the feasibility tests. Sufficient and necessary tests are ideal, but for many computational models such tests are intractable. Indeed, the complexity of such tests is NP-hard for non-trivial computational models (an overview of complexity results is given by Garey and Johnson[81]). Sufficient but not necessary tests are generally less complex, but are pessimistic. Feasibility analysis is most successful in systems where the relative priorities of tasks (as in fixed priority scheduling), or at least jobs (as with EDF scheduling), does not vary. Ha and Liu [89] showed that if a system of independent preemptible jobs with fixed release times and fixed job priorities is feasible with worst-case execution times, it remains feasible when task execution times are shortened.

Liu and Layland's fundamental insight regarding the feasibility of fixed-priority task sets is known as the Critical Instant Theorem [136]. A *critical instant* for a task is a release time for which the response time is maximized (or exceeds the deadline, in the case where the system is overloaded enough that response times grow without bound). The theorem says that, for a set of periodic tasks with fixed priorities, a critical instant for a task occurs when it is invoked simultaneously with all higher priority tasks. The interval from zero to D_i is then one over which the demand of higher priority tasks $\tau_1 ... \tau_{i-1}$ is at a maximum, creating the hardest situation for τ_i to meet its deadline. This theorem has be proven to be robust. It remains true when many of the restrictive assumptions about periodic tasks listed above are relaxed.

Though further research would find more efficient techniques, the Critical Zone Theorem provided an immediately obvious necessary and sufficient test for feasibility, *i.e.*, to simulate the execution of the set of tasks, assuming they are all initially released together, up to the point that the lowest priority task either completes execution or misses its deadline. The task set is feasible if and only if all tasks have completed within their deadlines. The simulation only need consider points in time that correspond to task deadlines and release times. Since there are only $\lceil D_n/T_i \rceil$ such points for each task τ_i, the complexity of this simulation is $O((\sum_{i=1}^{n} D_n)/T_i)$.

Based on the concept of critical instant, both Serlin [76] and Liu and Layland [136] proved a sufficient utilization-based condition for feasibility of when a set of tasks assigned priorities according to a *rate-monotonic* (RM) policy, that is, when the task with the shortest period is given the highest priority, the task with the next shortest period the second highest priority, *etc.* The efficacy of RM priority assignment for periodic tasks had been suggested earlier [76, 226], but without analysis or proof. Liu and Layland proved that it is the optimal static task priority priority assignment, in the sense that if a task set can be scheduled with any priority assignment it is feasible with the RM assignment. Serlin, and Liu and Layland, also showed that with this policy scheduling a set of n periodic tasks is feasible if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$$

For example, a pair of tasks is feasible if their combined utilization is no greater than 82.84%. As n approaches infinity, the value of $n(2^{1/n} - 1)$ approaches $ln(2)$ (approximately 69.31%).

A common misconception is that with fixed-priority scheduling, and rate-monotonic scheduling in particular, it is not possible to guarantee the feasibility for any periodic task set with a processor utilization greater than $\ln 2$. This bound *is* tight in the sense that there exist some infeasible task sets with utilization arbitrarily close to $n(2^{1/n} - 1)$, but it is *only a sufficient* condition. That is, many task sets with utilization higher than Liu and Layland are still schedulable. Lehoczky, Sha and Ding [123] showed that the average real feasible utilization, for large randomly chosen tasks sets, is approximately 88%. The remaining cycles can be used by non-real-time tasks executing with background priorities. The desire for more precise fixed-priority schedulability tests, *i.e.*, conditions that are necessary as well as sufficient, led to the feasibility analysis which is described later in this section.

It is also important to note that high utilization can be guaranteed by appropriate choice of task periods. In particular, if task periods are *harmonic* (that is, each task period is an exact integer multiple of the next shorter period) then schedulability is guaranteed up to 100% utilization. This is often the case in practice in a number of application domains. Sha and Goodenough showed that by transforming periods to be nearly harmonic (with zero or small residues in the division of periods), the schedulability can be significantly improved [190]. For example, for two tasks with periods 10 and 15, the task with period 10 can be mapped into a new task with period 5 and half of the original execution time. The schedulability now becomes 100% because the residue in period division is now zero. This technique, called *period transformation*, can be done without changing the source code by means of using one of the fixed-priority aperiodic server scheduling techniques, such as the Sporadic Server.

2.2. Further Developments

The success of fixed-priority scheduling has come through the work of a large group of researchers, who extended Liu and Layland's original analysis in a number of ways. The full story spans a very large number of publications. However, taking a historical point of view [19, 190], one can recognize within all this research a few principal threads:

(i) *exact feasibility analysis* - necessary and sufficient feasibility tests (based upon calculation of the worst-case response time of a task) permitted higher utilization levels to be guaranteed, and led to further extensions of the theory, such as overrun sensitivity analysis and the analysis of tasks with start-time offsets;

(ii) *analysis of task interaction* - the introduction of the family of Priority Inheritance Protocols enabled potential blocking to be bounded and analyzed;

(iii) *inclusion of aperiodic tasks* - the introduction of aperiodic servers permitted aperiodic tasks to be accomodated within the strictly periodic task model;

(iv) *overload management* - techniques for handling variations in task execution times made it possible to relax the requirement of known worst-case execution times, and still ensure that a critical subset of the tasks will complete on time;

(v) *implementation simplifications* - the demonstration that only a small number of implementation priority levels is needed made it practical to apply fixed priority scheduling more widely, including in hardware buses;

(vi) *multiprocessors and distributed systems* - analysis techniques were adapted to systems with multiple processors.

In the following sections we discuss these key results, together with the subsequent threads of research leading from them.

2.3. Feasibility Analysis

The utilization bound feasibility test described above is simple, both in concept and computational complexity. Consequently, it is widely recognized and is frequently cited. However, it has some limitations:

(i) the feasibility condition is sufficient but not necessary (*i.e.*, pessimistic);

(ii) it imposes unrealistic constraints upon the timing characteristics of tasks (*i.e.*, $D_i = T_i$);

(iii) task priorities have to be assigned according to the rate-monotonic policy (if priorities are not assigned in this way then the test is insufficient).

During the mid 1980's, more complex feasibility tests were developed to address the above limitations.

In 1987 Lehoczky, Sha and Ding [123] abstracted the idea behind the Critical Zone simulation feasibility test for RM scheduling, by observing if a set of tasks is released together at time zero, the i highest priority task will complete its first execution within its deadline if there is a time $0 < t \leq T_i$ such that the demand on the processor, $W_i(t)$, of the i highest priority tasks is less than or equal to t; that is,

$$W_i(t) = \sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j \leq t \tag{1}$$

Since $\frac{t}{T_j}$ is strictly increasing except at the points where tasks are released, the only values of t that must be tested are the multiples of the task periods between zero and T_i. This test is also applicable to task sets with arbitrary fixed priority orderings, although it was not recognized until 1991 when Nassor and Bres [165] extended the approach to permit task deadlines to be less than their periods.

Concurrently, another group of researchers looked at the more general problem of determining the *worst-case response time* of a task, that is, the longest time between the arrival of a task and its subsequent completion. Once the worst-case response time of a task is known, the feasibility of the task can be checked by comparing its worst-case response time to its deadline.

Fixed-priority response-time analysis was initiated by Harter in 1984 [93, 94], who used a temporal logic proof system to derive the *Time Dilation Algorithm*. Equivalent analysis was developed independently by Joseph and Pandya [109] and Audsley *et al.* [20]. The algorithm computes the worst-case response time R_i of task τ_i as the least-fixed-point solution of the following recursive equation:

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2}$$

Harter, Joseph *et al.* and Audsley *et al.* observed that only a subset of the task release times in the interval between zero and T_i need to be examined for feasibility. That is, the above equation can be solved iteratively. Audsley *et al.* gave an algorithm for this in 1991 [20], with the formal mathematical representation stated in 1993 [22, 23].

In 1982, Leung [128] considered fixed-priority scheduling of sets of tasks that may have deadlines that are less than their periods, *i.e.*, $C_i \leq D_i \leq T_i$. He showed that the optimal policy for such systems, called *deadline monotonic* (DM) scheduling, is to assign tasks with shorter deadlines higher priorities than tasks with longer deadlines. RM and DM scheduling are the same when deadline equals period, and the proof of optimality follows the same reasoning. Since the response-time tests described above do not depend on any particular priority assignment, they can be applied to DM scheduling as well as RM scheduling.

Lehoczky [117] considered another relaxation of the Liu and Layland model, to permit a task to have a deadline greater than its period. For this case, he proposed two sufficient but not necessary feasibility tests. Both tests are based upon utilization, extending the $D_i = T_i$ utilization test of Liu and Layland. The tests restrict all tasks τ_i to have $D_i = kT_i$, where k is constant across all tasks. One test restricts k to be an integer, the other does not. In the early 1990s, Tindell [220] extended response time analysis, providing an exact test for tasks with arbitrary deadlines. Unlike the tests proposed by Lehoczky, the analysis given by Tindell places no restrictions upon the relative task periods.

A further relaxation of the Liu and Layland task model is to permit tasks to have specified offsets (phasing), in which case the tasks might never share a common release time and the worst-case situation described by Liu and Layland (when all tasks are released simultaneously) might never occur. Since the response time and utilization bound feasibility conditions rely on the assumption of worst-case task phasing, they may be overly pessimistic for such systems. For such systems the approach of Leung and Merrill [127] (*i.e.*, construction of a schedule for the LCM of task periods) can be used to determine feasibility. Alternatively, an approach that realizes a sufficient and necessary test can be developed (which is more efficient than evaluating the schedule) [18]. General offsets do however remain a problem to efficiently analyze (especially if sporadic tasks are contained in the system). Fortunately for many practical problems a system with offsets can be transformed into one without (for analysis purposes) [36].

Under the assumption of specified offsets the rate and deadline monotonic algorithms are no longer optimal. Leung [128] questioned whether there is a polynomial time algorithm to find the optimal priority assignment for such task sets. Later, Audsley [25, 18] showed that optimal priority assignment for this case can be achieved by examining a polynomial number of priority ordering over the task set (*i.e.*, not $n!$).

Although the forms of analysis outlined above are all efficient and can easily be incorporated into tools, researchers continue to work on the algorithms and produce even more efficient schemes [91, 203, 41]. For example, Bini, Buttazzo and Buttazzo [42] derived a new sufficient test having polynomial complexity, less pessimistic than the Liu and Layland one, proving that a set of n periodic tasks is feasible if

$$\prod_{i=3D1}^{n} \left(\frac{C_i}{T_i} + 1 \right) \leq 2$$

They showed that the test is tight and the improvement in terms of guarantee ratio tends to $\sqrt{2}$ for large n.

2.4. Task Interaction

The Liu and Layland model assumes that tasks are independent. In real systems that is not the case. There are resources, such as buffers and hardware devices, that tasks must access in a mutually exclusive manner. The mechanisms that enforce mutual exclusion necessarily sometimes cause a task to *block* until another task releases a resource. For a task system to be feasible, the duration of such blocking must be bounded. Given such a bound, the feasibility analysis can be refined to take the worst-case effect of blocking into account.

In a fixed-priority preemptive scheduling system, blocking is called *priority inversion* because it results in a violation of the priority scheduling rule: if the highest priority task is blocked, a lower priority task will execute. Sha observed that the duration of priority inversion due to blocking may be arbitrarily long. For example, a high-priority task may preempt a low priority task, the high-priority task may then be blocked when it attempts to lock a resource already locked by the low priority task, and then the low priority task may be repeatedly preempted by intermediate priority tasks, so that it is not able to release the resource and unblock the high-priority task. Sha, Rajkumar and Lehoczky introduced the family of *priority inheritance protocols* as a solution approach to the priority inversion problem [192, 196, 197, 178]. The *Priority Inheritance Protocol* (PIP) prescribes that if a higher priority task becomes blocked by a low priority task, the task that is causing the blocking should execute with a priority which is the maximum of its own nominal priority and the highest priority of the jobs that it is currently blocking.

However, the PIP does not prevent deadlocks. In addition, a job can be blocked multiple times. A solution is to add a new rule to the PIP: associate with each resource a priority, called the *priority ceiling* of the resource. The priority ceiling is an upper bound on the priority of any task that may lock the resource. A job may not enter its critical section unless its priority is higher than all the priority ceilings currently locked by other jobs. Under this ceiling rule, a job that may share resources currently locked by other tasks cannot enter its critical section. This prevents the deadlocks. In addition, under the priority ceiling protocol, a job can be blocked at most once. This notion of priority ceilings was also suggested earlier in the context of the Mesa programming language in 1980 [114]. It is the basis of several locking protocols, including the *Priority Ceiling Protocol* (PCP), [192, 196, 197, 178, 176], the *Stack Resource Protocol* (SRP) [27], the Ada 95 programming language ceiling locking protocol [29], and the POSIX mutex *PTHREAD_PRIO_PROTECT* protocol[64].

The feasibility analyses given in the previous section have been extended to account for the blocking that tasks may be subject to at run time. In the calculation of the feasibility of a task, its computation time is viewed as consisting of its worst-case execution time, the worst-case interference it get from higher priority tasks, and the worst-case time for which it may be blocked.

Permitting tasks to communicate and synchronize via shared resources is only one form of task interaction that is not allowed within the Liu and Layland model. Another is precedence constraints or relations between tasks. When two tasks have a precedence relationship between them, the successor task cannot commence execution before the predecessor task has completed. While the choice of task offsets and periods can enforce precedence relations implicitly, little work has appeared in the literature regarding the explicit analysis of precedence related tasks within fixed-priority systems. In 1991, Harbour *et al.* [92] considered tasks that are subdivided into precedence constrained parts, each with a separate priority. In 1992, Audsley *et al.* [21] considered priority assignment across precedence constrained tasks where the priority of a predecessor task is at least that of all its successors. Further, they also gave analysis for groups of tasks where each group comprises a set of tasks linked by a linear precedence constraint [25].

2.5. Aperiodic Tasks

In previous sections, we have discussed the scheduling of tasks with periodic releases and predictable execution times. However, many real-time systems also contain tasks whose processor demands do not fit that model. We use the term *non-periodic* for such tasks, whether they have significantly varying inter-release times, significantly varying execution times, no hard deadline, or some combination of these characteristics.

If the periodic task model is relaxed slightly, letting C_i be just the maximum execution time and T_i be just the minimum inter-release time, Liu and Layland's analysis and most of the refinements that followed it remain valid [137]. However, for non-periodic tasks with large variations in inter-release times and/or execution times, reserving enough processor capacity to guarantee feasibility under this model is impractical.

One simple way to handle non-periodic tasks is to assign them priority levels below those of tasks with hard deadlines, *i.e*, relegate them to background processing. However, if there is more than one such task and they have quality-of-service requirements, average response-time requirements, or average throughput requirements, background processing is likely to be unsatisfactory.

Non-periodic tasks with quality-of-service requirements may be run at a higher priority level, under the control of a pseudo hard real-time server task such as a polling server [191]. A polling server is a periodic task with a fixed priority level (possibly the highest) and a fixed execution capacity. The capacity of the server is calculated off line and is normally set to the highest level that permits the feasibility of the hard-deadline periodic task set to be guaranteed. At run time, the polling server is released periodically. Its capacity is used to service non-periodic tasks. Once this capacity has been exhausted, execution of the polling server is suspended until the server's next (periodic) release. Since the polling server behaves like a periodic task, the feasibility analysis techniques developed for periodic tasks can be applied.

A polling server can guarantee hard deadlines for sporadic tasks, by appropriate choice of period and server budget, and can guarantee a minimum rate of progress for long-running tasks. It also is a significant improvement over background processing for aperiodic tasks. However, if aperiodic jobs arrive in a large enough burst to exceed the capacity of the server, then some of them must wait until its next release, leading to potentially long response times. Conversely, if no jobs are ready when the server is released, the high-priority capacity reserved for it is wasted.

The Priority Exchange, Deferrable Server [217, 124], and Sporadic Server algorithms [206][1] are based on principles similar to those underlying the polling server. However, they reduce wasted processor capacity by preserving it if there are no jobs pending when the server is released. Due to this property, they are termed *bandwidth preserving algorithms*. The three algorithms differ in the ways in which the capacity of the server is preserved and replenished and in the feasibility analysis needed to determine their maximum capacity. In general, all three offer improved responsiveness over the polling approach. One variant of the Sporadic Server was adopted for the POSIX *SCHED_SPORADIC* scheduling policy[64].

Even these more complex server algorithms are unable to make full use of the slack time that may be present due to the often favorable (*i.e.*, not worst-case) phasing of periodic tasks. The Deferrable and Sporadic servers are also unable to reclaim spare capacity gained, when for example, other tasks require less than their worst-case execution time. This spare capacity can, however, be reclaimed by the Extended Priority Exchange algorithm [205]. Comparisons of these server schemes are provided by Bernat and Burns [39] who conclude that the best choice of server algorithm is application dependent.

The Slack Stealing algorithm overcomes these disadvantages [125, 183, 184]. It is optimal in the sense that it minimizes the response times of aperiodic tasks amongst all algorithms that allocated spare capacity immediately while meeting all hard task deadlines[2]. The Slack Stealer services aperiodic requests by making any spare processing time available as soon as possible. In doing so, it effectively steals slack from the hard-deadline periodic tasks. To achieve optimal aperiodic task scheduling, this static Slack Stealing algorithm relies on a table that stores the slack available on each release of each hard-deadline periodic task over the least common multiple (LCM) of task periods. Unfortunately, the need to map out the LCM restricts the applicability

[1] The original description of the Sporadic Server in [206] contained a defect, that led to several variations. See [137] for details.

[2] It was shown by Tia *et al.*[218] that there are situations in which it is better to preserve spare capacity rather than use it immediately; and as a consequence optimal performance is not attainable

of the Slack Stealer: slack can only be stolen from hard-deadline tasks which are strictly periodic and have no release jitter. Realistically, it is also limited to task sets with a manageably short LCM, and with very reliable worst-case execution time bounds.

Many of the limitations inherent in the optimal static Slack Stealing algorithm have been addressed by a dynamic variant developed in 1993 [65]. By computing the slack available at run time, this algorithm handles a wider range of scheduling problems, including task sets that contain hard-deadline sporadics and tasks that exhibit release jitter. However, there are problems with the practical implementation of both the static and dynamic versions of the optimal Slack Stealing algorithm due to their respective space and time complexities.

Another approach to supporting soft-deadline tasks within a system that contains hard deadlines is to assign two priorities to the hard-deadline tasks [46, 66]. When necessary, the task is promoted to the second, higher, priority to ensure that it will meet its deadline. In general, the soft tasks have deadlines below the second but above the initial deadline of these hard tasks.

Recently Abdelzaher *et al.* [3] have developed sufficient conditions for accepting firm deadline aperiodic work while guaranteeing that all deadlines of those tasks will be met. The admission algorithm requires knowledge of an arriving task's computation requirement and deadline. Given this knowledge, Abdelzaher derives a schedulability bound in the spirit of the Liu and Layland bound showing that if the workload level is kept below a certain synthetic utilization level, all admitted tasks will always meet their deadlines. Specifically, assume that each arriving aperiodic task has known computation time C and deadline D. At each instant of time, a *synthetic utilization* value, $U(t) = \sum \frac{C_i}{D_i}$, is computed where the sum extends over all tasks that have arrived and been accepted but whose deadlines have not yet expired. Abdelzaher *et al.* show that there is a fixed-priority policy that will meet all the deadlines of the accepted aperiodic tasks as long as the restriction $U(t) \leq 2 - \sqrt{2}$ is enforced. Arriving aperiodic tasks that would cause $U(t)$ to exceed this bound are not accepted for processing.

The admission algorithm has been tested for its performance and is shown to work well in overload conditions in that the system is able to achieve average utilization results near 100%, the maximum possible. The algorithm is especially attractive in that all aperiodic tasks accepted for processing are guaranteed to complete by their deadline. The algorithm has also been generalized to the multiprocessor case.

The results appear to be very promising, although the task acceptance algorithm does assume that the task computation times are known. This assumption may be reasonable for the case of web servers in which a certain file of known size is being requested, but it is less likely to be known with any precision in more general situations.

2.6. Overload Management

Analyses of schedulability must make some assumptions about workload, which is ordinarily characterized by the worst-case execution time and the minimum inter-release time of each task. However, real task execution times may vary widely, and the actual worst-case execution time of a task may be difficult or impossible to determine. Task release times may also vary from the ideal model. To cope with such variability, a system designer may try to: (a) *prevent* overloads, by making safe assumptions about workload, which may be even more pessimistic than the actual worst case; (b) *tolerate* overloads, by providing some reduced-but-acceptable level of service when the workload exceeds normal expectations.

With fixed-priority scheduling, an execution time overrun (provided it is not in a non-preemptible section) or early task release may only delay the execution of lower priority tasks. If it is known which tasks may be released early or run over their nominal worst-case execution times, then only those tasks and the tasks of lower priority need to be designed to tolerate and recover from overruns.

In many practical situations, it may be possible to distinguish critical and non-critical tasks. That is, the system may able to function tolerably for some period of time if all the critical tasks complete within their deadlines. Sha and Goodenough showed that it is easy to ensure a set of critical tasks' deadlines will be met when non-critical tasks overrun their nominal worst-case executions times, provided the critical tasks never overrun their own nominal worst-case execution times and are schedulable as a group under with those worst-case execution times [190]. The technique is to shorten the periods of the critical tasks (the period transformation mentioned in Section 2.1) so that the critical tasks all have higher rate monotonic priorities than the non-critical tasks.

More precise control over the effects of overload may be achieved in other ways. Servers can be used to isolate the cause of overload, or to provide resources for recovery [185]. A dual priority scheme can be used to assign higher priorities to the recovery code [131]. Another approach to overload is to define, if the application will allow, certain executions of each task to be 'skippable' [112, 38, 174, 40].

If one designs to prevent overloads, by sizing a system to function correctly under very conservative, safe, assumptions about workload, there will ordinarily be some spare capacity available at run time. This spare capacity becomes available for many reasons, including tasks completing in less than their worst-case execution time, sporadic tasks not arriving at their maximum rate and periodic tasks not arriving in worst-case phasing. One then has the problem of *resource recovery*, that is, finding a way to use the spare capacity to enhance the quality of hard real-time services.

One important approach to designing to make good use of spare capacity is the *imprecise computation* model. In a real-time system that supports imprecise computation [139, 126, 132, 138, 199, 140, 166], every task τ_i is decomposed into a *mandatory* subtask M_i and an *optional* subtask O_i. The mandatory subtask is the portion of the computation that must be done in order to produce a result of acceptable quality, whereas the optional subtask refines this result [200]. Both subtasks have the same arrival time a_i and the same deadline d_i as the original task τ_i; however, O_i becomes ready for execution when M_i is completed. If C_i is the worst-case computation time associated with the task, subtasks M_i and O_i have computation times m_i and o_i, such that $m_i + o_i = C_i$. In order to guarantee a minimum level of performance, M_i must be completed within its deadline, whereas O_i can be left incomplete, if necessary, at the expense of the quality of the result produced by the task. The concept of imprecise computation is useful for both fixed-priority and dynamic-priority scheduling.

For a set of periodic tasks, the problem of deciding the best level of quality compatible with a given load condition can be solved by associating each optional part of a task a reward function $R_i(\sigma_i)$, which indicates the reward accrued by the task when it receives σ_i units of service beyond its mandatory portion. This problem has been addressed by Aydin *et al.*, [26], who presented an optimal algorithm that maximizes the weighted average of the rewards over the task set under EDF scheduling. In 1993, it was shown how many of the above techniques could be supported within the framework of fixed-priority preemptive scheduling [22, 24]. However, to achieve the goal of maximizing system utility, some form of best effort scheduling [108, 141] of optional components is required. It remains an open research issue to fully integrate fixed-priority and best-effort scheduling.

2.7. Implementation Simplifications

While there can be an arbitrary large number of task periods, it is impractical to support an arbitrary large number of priority levels in system platforms, especially in hardware buses and communication channels. Lehoczky and Sha showed that when using a constant ratio grid to map periods to priority levels, having 256 priority levels is al-most as good as having an arbitrary large number of priority levels[122]. 256 priority levels can be neatly represented by a single byte. This trade-off analysis between priority levels and schedulability loss was used by many open standards in their design decisions on the number of priority levels, e.g., Futurebus+ [195].

2.8. Multiprocessor and Distributed Systems

The 1978 paper of Dhall and Liu [70] has been nearly as influential on multi-processor scheduling as that of Liu and Layland [136] was for single processor scheduling. There are two approaches to scheduling tasks on multiple processors: (1) partitioned, in which the each task is assigned to one processor, more or less statically; (2) global, in which tasks compete for the use of all processors. Dhall and Liu showed that the global application of rate-monotonic (RM) scheduling to m-processor scheduling cannot guarantee schedulability of a system of $m+1$ tasks for system utilizations above 1, whilst RM next-fit partitioning can guarantee schedulability of such task systems for system utilizations up to $m/(1+2^{1/3})$. For several years thereafter, the conventional wisdom was in favor of partitioned scheduling and research focused on partitioning algorithms.

The optimal partitioning of tasks among processors, a variant of the well-known Bin Packing problem, is known to be NP-complete [82], so the problem of finding optimal allocations is intractable for all but the simplest systems. Hence, heuristics and global optimization techniques, such as simulated annealing, have been used to find good sub-optimal static allocations [179, 219, 44, 170, 102]. The best understood heuristic is the Rate-Monotonic First-Fit (RMFF) heuristic. Dhall and Liu's [70] showed that this may require between 2.4 and 2.67 times as many processors as an optimal partition. Oh and Baker [169] showed that RMFF can schedule any system of periodic tasks with total utilization bounded by $m(2^{1/2}-1)$. Lopez, Diaz and Garcia [142] showed that RMFF can schedule any system of periodic tasks with utilization up to $(m+1)(2^{1/(m+1)}-1)$, a bound that appears to be tight.

In 2001, Andersson, Baruah, and Jonsson [15] showed that the utilization guarantee for any fixed-priority multiprocessor scheduling algorithm – partitioned or global, with any priority scheme – cannot be higher than $(m+1)/2$ for an m-processor platform.

Recent research has reopened the subject of global scheduling. shown that the "Dhall effect" is not a problem with global scheduling so much as it is a problem with high-utilization tasks. In 1997, Phillips, Stein, Torng, and Wein [172] studied the competitive performance of on-line multiprocessor scheduling algorithms, including fixed-priority scheduling, against optimal (but infeasible) clairvoyant algorithms. Among other things, they showed that if a set of tasks is feasible (schedulable by any means) on m processors of some given speed then the same task set is schedulable by preemptive RM scheduling on m processors that are faster by a factor of $(2 - \frac{1}{m})$. This was followed by a number of other results by other authors on schedulability of both identical and "uniform" (meaning differing only in speed) processors [15, 33, 78, 85].

The key seems to be be that global rate-monotonic scheduling works well for tasks with low individual utilizations. If λ is an upper bound on the individual utilization, the smaller is λ the larger is the worst-case guaranteeable system utilization. Andersson, Baruah, and Jonsson [15] showed that for $\lambda = m/(3m-2)$ a system utilization of at least $m^2/(3m-1)$ can be guaranteed. Baruah and Goossens [33] showed that for $\lambda = 1/3$ system utilization of at least $m/3$ can be guaranteed. Baker [28] showed that for any $\lambda \leq 1$, system utilization of $\frac{m}{2}(1-\lambda)+\lambda$ can be guaranteed.

Andersson, Baruah, and Jonsson [15] also showed how to extend these results to tasks with arbitrarily large individual utilizations. The scheduling algorithm called RM-US(θ) gives highest priority to tasks with utilizations above some threshold θ, and then assigns priority to the rest in RM order. This algorithm can guarantee a system utilization least $(m+1)/3$ for $\theta = 1/3$ [28]. Lundberg [152] has gone further, arguing that the guaranteed system utilization is maximized at $0.37482 \cdot m$ when $\theta = 0.37482$. Work on the analysis of RM-US continues, including its application to aperiodic servers.

Priority ceiling protocols, and synchronization protocols in general, are also significantly affected by parallel execution. Although a number of revised protocols have been developed [153, 176, 69, 177, 163] there is not as yet a general-purpose approach equivalent to the uniprocessor protocols. An alternative approach is to employ non-blocking or lock free protocols [202, 14].

Most research on fixed-priority scheduling has focused on processor scheduling, but there are other resources for which priority may be used to arbitrate between competing requests. For example Futurebus+ [194] and the CAN communication protocol use a non-preemptive fixed-priority scheme, and hence fixed-priority response-time analysis can be applied directly to real-time message passing [222, 43]. For a distributed system, all resources (processor and communication media) must be analyzed and then a complete end-to-end model of transactions passing through the many resources must be constructed. The use of a consistent priority scheduling policy for all resources significantly simplifies this problem and allows global real-time requirements to be addressed and verified [221, 87, 88, 86, 37, 115, 171, 28].

Despite the advances described above, feasibility analysis for parallel computation systems is still not as well understood as for the single processor case. It is likely that more of the advances made for single processor systems will be successfully generalized to parallel systems in coming years.

3. Dynamic Priority Scheduling

Giorgio Buttazzo and Marco Caccamo

With dynamic-priority scheduling, task priorities are assigned to individual jobs. One of the most used algorithms belonging to this class is the *Earliest Deadline First* (EDF) algorithm, according to which priorities assigned to tasks are inversely proportional to the absolute deadlines of the active jobs. The feasibility analysis of periodic task sets under EDF was first presented in 1973 by Liu and Layland [136], who showed that, under the same simplified assumptions used for rate-monotonic (RM) scheduling, a set of n periodic tasks is schedulable by the EDF algorithm, if and only if

$$\sum_{i=1}^{n} U_i \leq 1. \tag{3}$$

In 1974, Dertouzos [68] showed that EDF is optimal among all preemptive scheduling algorithms, in the sense that, if there exists a feasible schedule for a task set, then the schedule produced by EDF is also feasible. Later, Mok presented another optimal algorithm, *Least Laxity First* (LLF) [158], which assigns the processor to the active task with the smallest laxity[3]. Although both LLF and EDF are optimal algorithms, LLF has a larger overhead due to the higher number of context switches caused by laxity changes at run time. For this reason, most of the work done in the real-time research community concentrated on EDF to relax some simplistic assumption and extend the feasibility analysis to more general cases.

In this section we provide an overview of the key results related to EDF scheduling. We first present an analysis technique for verifying the feasibility of deadline-based schedules and briefly describe some efficient algorithms for aperiodic task scheduling and shared resource

[3] We recall that the laxity (or slack time) is the difference between the absolute deadline and the estimated worst-case finishing time.

management. Then, we describe some method for dealing with overload conditions and we conclude the section by presenting some open research issues.

3.1. Processor Demand Criterion

Under EDF, the analysis of periodic tasks with deadlines less than periods can be performed by the *processor demand* criterion, proposed in 1990 by Baruah, Rosier, and Howell [35].

In general, the processor demand in an interval $[t_1, t_2]$ is the amount of processing time $g(t_1, t_2)$ requested by those jobs activated in $[t_1, t_2]$ that must be completed in $[t_1, t_2]$. Hence, the feasibility of a task set is guaranteed if and only if *in any interval of time* the total processor demand does not exceed the available time, that is, if and only if

$$\forall t_1, t_2 \quad g(t_1, t_2) \; \leq \; (t_2 - t_1).$$

Baruah, Rosier, and Howell showed that a set of periodic tasks simultaneously activated at time $t = 0$ is schedulable by EDF if and only if $U < 1$ and

$$\forall L > 0 \quad \sum_{i=1}^{n} \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor C_i \; \leq \; L. \tag{4}$$

It can easily been shown that the points in which the test has to be performed correspond to those deadlines within the hyper-period H not exceeding the value

$$L^* \; = \; \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U}.$$

It follows that when deadlines are less than or equal to periods, the exact feasibility analysis of EDF is of pseudo-polynomial complexity, but when deadlines are equal to periods, the complexity is only $O(n)$.

3.2. Aperiodic task scheduling

The higher schedulability bound of dynamic scheduling schemes also allows achieving better responsiveness in aperiodic task handling. Several algorithms have been proposed in the real-time literature to handle aperiodic requests within periodic task systems scheduled by EDF [83, 207, 208]. Some of them are extensions of fixed-priority servers, whereas some were directly designed for EDF.

3.2.1. *The Total Bandwidth Server approach*

One of the most efficient techniques to safely schedule aperiodic requests under EDF is the Total Bandwidth Server (TBS) [207, 208], which assigns each aperiodic job a deadline in such a way that the overall aperiodic load never exceeds a specified maximum value U_s.

In particular, when the kth aperiodic request arrives at time $t = r_k$, it receives a deadline

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s},$$

where C_k is the execution time of the request and U_s is the server utilization factor (that is, its bandwidth). By definition $d_0 = 0$. Note that in the deadline assignment rule the bandwidth allocated to previous aperiodic requests is considered through the deadline d_{k-1}. Once the deadline is assigned, the request is inserted into the system ready queue and scheduled by EDF as any other periodic instance. As a consequence, the implementation overhead of this algorithm is practically negligible. The schedulability test for a set of periodic tasks scheduled by EDF in the presence of a TBS is given by the following theorem by Spuri and Buttazzo.

THEOREM 1. *Given a set of n periodic tasks with processor utilization U_p and a TBS with processor utilization U_s, the whole set is schedulable by EDF* if *and* only if

$$U_p + U_s \leq 1.$$

3.2.2. *Achieving optimality*

The deadline assigned by the TBS can be shortened to minimize the response time of aperiodic requests, still maintaining the periodic tasks schedulable. Buttazzo and Sensini [51] proved that setting the new deadline at the current estimated worst-case finishing time does not jeopardize schedulability.

The process of shortening the deadline can then be applied recursively to each new deadline, until no further improvement is possible, given that the schedulability of the periodic task set must be preserved. If d_k^s is the deadline assigned to the aperiodic request J_k at step s and f_k^s is the corresponding finishing time in the current EDF schedule (achieved with d_k^s), the new deadline d_k^{s+1} is set at time f_k^s. The algorithm stops either when $d_k^s = d_k^{s-1}$ or after a maximum number of steps defined by the system designer for bounding the complexity.

It is worth noticing that the overall complexity of the deadline assignment algorithm is $O(Nn)$, where N is the maximum number of steps performed by the algorithm to shorten the initial deadline assigned by the TBS. Finally, as far as the average case execution time of tasks is equal to the worst-case one, this method achieves optimality, yielding the minimum response time for each aperiodic task, as stated by the following theorem by Buttazzo and Sensini.

THEOREM 2. *Let σ be a feasible schedule produced by EDF for a task set $\mathcal{T}$ and let f_k be the finishing time of an aperiodic request J_k, scheduled in σ with deadline d_k. If $f_k = d_k$, then $f_k = f_k^*$, where f_k^* is the minimum finishing time achievable by any other feasible schedule.*

Although the TBS can efficiently handle bursty sequences of jobs, it cannot be used to serve jobs with variable or unknown execution times. In this case, a budget management mechanism is essential to prevent execution overruns to jeopardize the schedulability of hard tasks.

3.2.3. *The Constant Bandwidth Server*

The Constant Bandwidth Server (CBS) is a scheduling mechanism proposed by Abeni and Buttazzo [6, 8] to implement resource reservations in EDF based systems.

A CBS is characterized by a budget c_s, a dynamic server deadline d_s, and by an ordered pair (Q_s, T_s), where Q_s is the maximum budget and T_s is the period of the server. The ratio $U_s = Q_s/T_s$ is denoted as the server bandwidth.

Each job served by the CBS is assigned a suitable deadline equal to the current server deadline, computed to keep its demand within the reserved bandwidth. As the job executes, the budget c_s is decreased by the same amount and, every time $c_s = 0$, the server budget is recharged to the maximum value Q_s and the server deadline is postponed by a period T_s to reduce the interference on the other tasks. Note that by postponing the deadline, the task remains eligible for execution. In this way, the CBS behaves as a work-conserving algorithm, exploiting the available slack in an efficient way, providing better responsiveness as compared to non-work-conserving algorithms and to other reservation approaches that schedule the extra portions of jobs in background.

An important property is that, in any interval of time of length L, a CBS with bandwidth U_s will never demand more than U_sL, independently from the actual task requests. This property allows the CBS to be used as a bandwidth reservation strategy to allocate a fraction of the CPU time to soft tasks whose computation time cannot be easily bounded. The most important consequence of this result is that such tasks can be scheduled together with hard tasks without affecting the *a priori* guarantee, even in the case in which soft requests exceed the expected load.

3.3. Resource Sharing

Under EDF, several resource access protocols have been proposed to bound blocking due to mutual exclusion, such as Dynamic Priority Inheritance [216], Dynamic Priority Ceiling [62], Dynamic deadline Modification [104], and Stack Resource Policy [27]. The Stack Resource Policy is one of the most used method under EDF for its properties and efficiency, hence it will be briefly recalled in the next section.

3.3.1. *Stack Resource Policy*

The Stack Resource Policy (SRP) is a concurrency control protocol proposed by Baker [27] to bound the priority inversion phenomenon in static as well as dynamic-priority systems. In this work, Baker made the insightful observation that, under EDF, jobs with a long relative deadline can only delay, but cannot preempt jobs with a short relative deadline. A direct consequence of this observation is that a job cannot block another job with longer relative deadline. Thus, in the study of blocking under EDF, we only need to consider the case where jobs with longer relative deadlines block jobs with shorter relative deadlines.

This observation allows Baker to define preemption levels separately from priority levels under EDF, in a way that they are inversely proportional to relative deadlines. It follows that the semaphore preemption ceiling under SRP can be defined in the same way as the priority ceiling under PCP.

Hence, under SRP with EDF, each job of τ_i is assigned a priority p_i according to its absolute deadline d_i and a static *preemption level* π_i inversely proportional to its relative deadline. Each shared resource is assigned a ceiling which is the maximum preemption level of all the tasks that will lock this resource. Moreover, a *system ceiling* Π is defined as the highest ceiling of all resources currently locked.

Finally, SRP scheduling rule requires that

> *"a job J is not allowed to start executing until its priority is the highest among the active tasks and its preemption level is greater than the system ceiling* Π*".*

SRP guarantees that, once a job is started, it will never block until completion; it can only be preempted by higher priority tasks. SRP prevents deadlocks and a job can be blocked for at most the duration of one critical section.

Under the SRP there is no need to implement waiting queues. In fact, a task never blocks during execution: it simply cannot start executing if its preemption level is not high enough. As a consequence, the blocking time B_i considered in the schedulability analysis refers to the time for which task τ_i is kept in the ready queue by the preemption test.

The feasibility of a task set with resource constraints (when only periodic and sporadic tasks are considered), can be tested by the following sufficient condition [27]:

$$\forall i, \ \ 1 \leq i \leq n \quad \sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \ \leq \ 1, \tag{5}$$

which assumes that all the tasks are sorted by decreasing preemption levels, so that $\pi_i \geq \pi_j$ only if $i < j$.

As final remark, the Stack Resource Policy (SRP) allows tasks to share a common run-time stack, allowing large memory saving if there are many more tasks than relative priority levels.

Under EDF, resources can also be shared between hard tasks and soft tasks handled by an aperiodic server. Ghazalie and Baker [83] proposed to reserve an extra budget to the aperiodic server for synchronization purpose and used the utilization-based test [136] for verifying the feasibility of the schedule. Lipari and Buttazzo [135] extended the analysis to a Total Bandwidth Server. Caccamo and Sha [54] proposed another method for capacity-based servers, like the CBS, which also handles soft real-time requests with a variable or unknown execution behavior. The method is based on the concepts of *dynamic preemption levels* and allows resource sharing between hard and soft tasks without jeopardizing the hard tasks' guarantee.

3.4. Overload management

An overload condition is a critical situation in which the computational demand requested by the task set exceeds the time available on the processor, so that not all tasks can complete within their deadlines. When a task executes more than expected, it is said to overrun. This condition is usually transient and may occur either because jobs arrive more frequently than expected (*activation overrun*), or because computation times exceed their expected value (*execution overrun*). A permanent overload condition may occur in a periodic task system, when the total processor utilization exceeds one. This could happen after the activation of a new periodic task, or if some tasks increase their activation rate to react to some change in the environment. In such a situation, computational activities start to accumulate in system's queues and tasks response times tend to increase indefinitely.

Transient light overloads due to activation overruns can be safely handled by an aperiodic server that, in the case of a bursty arrival sequence, distributes the load more evenly according to the bandwidth allocated to it. In heavier load conditions admission control schemes may be necessary to keep the load below a desired threshold [213, 146].

Overloads due to execution overruns can be handled through a resource reservation approach [155, 156, 6]. The idea behind resource reservation is to allow each task to request a fraction of the available resources, just enough to satisfy its timing constrains. The kernel, however, must prevent each task to consume more than the requested amount to protect the other tasks in the systems (temporal protection). In this way, a task receiving a fraction U_i of the total processor bandwidth behaves as it were executing alone on a slower processor with a speed equal to U_i times the full speed. The advantage of this method is that each task can be guaranteed in isolation, independently of the behavior of the other tasks. A simple and effective mechanism for implementing temporal protection under EDF is the Constant Bandwidth Server (CBS) [6, 8]. To properly implement temporal protection, however, each task τ_i with variable computation time should be handled by a dedicated CBS with bandwidth U_{s_i}, so that it cannot interfere with the rest of the tasks for more than U_{s_i}.

Although resource reservation is essential for achieving predictability in the presence of tasks with variable execution times, the overall system's performance becomes quite dependent from a correct resource allocation. For example, if the CPU bandwidth allocated to a task is much less than its average requested value, the task may slow down too much, degrading the system's performance. On the other hand, if the allocated bandwidth is much greater than the actual needs, the system will run with low efficiency, wasting the available resources.

Two reclaiming techniques have been proposed to cope with an incorrect bandwidth assignment. The *CApacity SHaring* (CASH) algorithm [53] works in conjunction with the CBS. Its main idea can be summarized as following: 1) whenever a task completes its execution, the residual capacity (if any) is inserted with its deadline in a global queue of available capacities, the CASH queue, ordered by deadline; 2) whenever a new task instance is scheduled for execution, the server tries to use the residual capacities with deadlines less than or equal to the one assigned to the served instance; if these capacities are exhausted and the instance is not completed, the server starts using its own capacity. The main benefit of this reclaiming mechanism is to reduce the number of deadline shifts in the CBS, so enhancing aperiodic tasks responsiveness.

The *Greedy Reclamation of Unused Bandwidth* (GRUB) [135] algorithm is another server-based technique to reclaim unused processor bandwidth. According to the GRUB algorithm, each task is executed by a distinct server S_i, where each server is characterized by two parameters: a *processor share* U_i, and a *period* P_i. As main properties, GRUB is able to emulate a virtual processor of capacity U_i when executing a given job, and it uses a notion of virtual time to enforce isolation among different applications and to safely reclaim all unused capacities generated by periodic and aperiodic activities. Compared to CASH, the GRUB algorithm has more overhead due to its virtual time management but it achieves better performance in terms of aperiodic responsiveness.

Permanent overload conditions in periodic task systems can be handled using three basic approaches that allow keeping the load below a desired value. A first method reduces the total load by properly skipping (i.e., aborting) some jobs in the periodic tasks, in such a way that a minimum number of jobs per task are guaranteed to execute within their timing constraints. In a second approach, the load is reduced by enlarging task periods to suitable values, so that the total workload can be kept below a desired threshold. In the third approach, the load is reduced by decreasing the computational requirements of the tasks, trading quality of results with predictability.

3.4.1. *Job skipping*

Permitting skips in periodic tasks increases system flexibility, since it allows making a better use of resources and to schedule systems that would otherwise be overloaded. The *job skipping* model has been originally proposed by Koren and Shasha [112], who showed that making optimal use of skips is NP-hard and presented two algorithms that exploit skips to increase the feasible periodic load and schedule slightly overloaded systems.

According to the job skipping model, the maximum number of skips for each task is controlled by a specific parameter associated with the task. In particular, each periodic task τ_i is characterized by a worst-case computation time C_i, a period T_i, a relative deadline equal to its period, and a skip parameter S_i, $2 \leq S_i \leq \infty$, which gives the minimum distance between two consecutive skips. For example, if $S_i = 5$ the task can skip one instance every five. When $S_i = \infty$ no skips are allowed and τ_i is equivalent to a hard periodic task. The skip parameter can be viewed as a *quality-of-service* (QoS) metric (the higher S, the better the quality of service).

Using the terminology introduced by Koren and Shasha in [112], every instance of a periodic task with skips can be *red* or *blue*. A red instance must complete before its deadline; a blue instance can be aborted at any time. When a blue instance is aborted, we say that it was *skipped*. The fact that $S \geq 2$ implies that, if a blue instance is skipped, then the next $S-1$ instances must be red. On the other hand, if a blue instance completes successfully, the next task instance is also blue.

Koren and Shasha showed that the worst case for a periodic skippable task set occurs when all tasks are synchronously activated and the first $S_i - 1$ instances of every task τ_i are red (deeply-red condition). This means that, if a task set is schedulable under the deeply-red model, it is also schedulable without this assumption.

In the same work, Koren and Shasha proved the following theorem, which provides a sufficient condition for guaranteeing a set of skippable periodic tasks under EDF.

THEOREM 3. *A set of firm (i.e., skippable) periodic tasks is schedulable if*

$$\forall L \geq 0 \qquad \sum_{i=1}^{n} \left(\left\lfloor \frac{L}{T_i} \right\rfloor - \left\lfloor \frac{L}{T_i S_i} \right\rfloor \right) C_i \leq L \tag{6}$$

It is worth noting that, if skips are permitted in the periodic task set, the spare time saved by rejecting the blue instances can be reallocated for other purposes. In [52], Caccamo and Buttazzo generalized the results of Theorem 3 by identifying the amount of bandwidth saving achieved by skips to advance the execution of aperiodic tasks.

3.4.2. *Elastic scheduling*

In a periodic task set, processor utilization can also be changed by varying task periods. Whenever the total processor utilization is greater than one, the utilization of each task needs to be reduced so that the total utilization becomes equal to a desired value $U_d \leq 1$. This can be done as in a linear spring system, where springs are compressed by a force F (depending of their elasticity) up to a desired total length. To apply this method, each task needs to be specified with additional parameters, including a range of periods and an elastic coefficient, used to diversify compression among tasks, so that utilization reduction is higher for tasks with higher elasticity.

As shown in [50], in the absence of period constraints, the utilization U_i of each compressed task can be computed in order $O(n)$ (where n is the number of tasks), whereas in the presence of period constraints ($T_i \leq T_{i_{max}}$) the problem of finding the U_i values requires an iterative solution of $O(n^2)$ complexity.

The same algorithm can be used to reduce the periods when the overload is over, so adapting task rates to the current load condition to better exploit the computational resources.

Elastic scheduling has been also used in conjunction with feedback schemes to automatically adapt task rates to fully utilize the processor when task computation times are unknown [48]. Finally, elastic scheduling has been also extended to work with dumping coefficients

[49], whenever periods need to be smoothly adapted to satisfy some QoS requirements. Another method is imprecise computation that was discussed in Section 2.6.

3.5. Summary

Since the seminal work of Liu and Layland in 1973, many results has been provided by the real-time community to address most of the problems related to EDF-based real-time systems, including aperiodic task management, resource sharing, temporal isolation, resource reclaiming, overload management, and adaptive scheduling techniques. Combined with such advanced scheduling techniques, EDF has been shown to have many interesting properties that can be useful for developing efficient, predictable, as well as flexible real-time systems. In spite of these positive results, however, fixed priority scheduling is still the most adopted technique for implementing real-time applications.

One of the main reasons for the success of fixed priority scheduling is the simpler implementation, especially in commercial operating systems that do not provide direct support for explicit timing constraints and deadline-based scheduling. In fact, compared with fixed priority scheduling, the implementation of deadline-based scheduling requires some extra overhead for managing system queues and for updating the absolute deadlines at each job activation. In addition, to guarantee critical tasks' deadlines when the system is overloaded requires the use of servers, job skipping or imprecise computations. All of these have more overhead than the period transformation method under fixed priority scheduling. With the increase of computing power, however, such a difference may become unimportant in practice.

According to [47], another reason for the popularity of fixed priority scheduling may be due to theoretical misconceptions that penalized EDF. Despite its slightly larger run-time overhead, the use of dynamic scheduling would reduce the number of preemptions (hence context switches overhead), could have positive effects on jitter and overload conditions, would significantly enhance aperiodic responsiveness, would reduce input-output latency in periodic control tasks, and would allow a more efficient use of computational resources.

In the near future, one of the challenges for the real-time research community is to clear away all existing misconceptions about deadline-based scheduling, quantify the actual differences in overhead and quality of scheduling between fixed priority and deadline-based scheduling, and develop reference implementations to reduce the transfer delay from EDF theory to applications, making sure that the most important results quickly reach the industrial level, so that each problem can be solved with the most appropriate methodology.

4. Soft Real-Time Scheduling

John Lehoczky

4.1. The Soft Real-Time Case

Much of the early work on real-time scheduling focused on the hard deadline case in which tasks were assumed to have deterministic deadlines, and any missed task was considered to be a system failure. To build a predictable system that would have correct temporal behavior as well as correct logical behavior, it was necessary to impose a deterministic worst-case formulation. For example, it was necessary to assume that the jobs arriving from a task had deterministic minimum interarrival times, maximum computation times, and minimum deadlines. The worst-case formulation allowed for an upper bound on the workload during any interval of time, which, in turn, permitted an analysis of whether all deadlines of all tasks would always be met. The performance measure called *schedulable utilization* was developed, the largest level of utilization that could be achieved for a specific task set and still have all deadlines of all jobs be satisfied.

In traditional control and signal processing, the worst case and average execution times are often small. On the other hand, the average-case utilization can be much less than the worst-case utilization. An example of this occurs in video conferencing in which the bandwidth requirements for periodic transmission of frames or packets can vary widely. Figure 1 shows bandwidth utilization data from three different sections of a video conference. Figure 2 gives a histogram of the bandwidth usage over the course of a 60 minute video conference. The median bandwidth utilization is around 500 to 600 kbps; however, the requirements can reach values in excess of 5000 kbps when there is a scene change, action, or camera motion. If one wanted to guarantee adequate bandwidth for all frames, then one would have to treat this task as having 5000 kbps bandwidth requirements, a worst-case utilization nearly 10 times the average case utilization.

If a system is serving multiple tasks with highly variable computation requirements, and if one treats those tasks as having less their worst-case utilization, then it is possible that there could be a peak utilization time at which all tasks are simultaneously operating at their peak level. This would mean that some, if not all, of those frames would be lost due to inadequate bandwidth, and there would be a system failure according to the hard-deadline criterion. If the hard-deadline formulation is appropriate, then one must live with low levels of average-case utilization.

Fortunately, for many applications such as the video conference example, not all deadlines must be met, and some job loss is permissible. Often the actual requirements are for a certain quality of service (QoS), for example adequate audio and video quality for the user. Some frame or packet lateness (even loss) can be tolerated and may even go unnoticed by the user. When this is the case, the performance criterion should be altered.

There have been many formulations of the soft real-time scheduling problems. These include: 1) mixtures of hard-deadline periodics with aperiodic tasks having no deadlines (discussed in Section 2.5), 2) hard-deadline aperiodic tasks (discussed in Section 5), 3) the imprecise computation model (discussed in Section 2.6), 4) the extended periodic task model in which tasks have stochastic execution times and are guaranteed a minimum execution time [80], 5) stochastic rate-monotonic analysis in which aperiodic tasks have hard deadlines, stochastic execution times, and must meet quality-of-service requirements (discussed in Section 4.3.1), and 6) aperiodics with stochastic arrivals, execution times, and deadlines and a bound on the fraction of tasks that can miss their deadline (discussed in Section 4.4). Finally, models have been developed that assign to each task a utility value that is a function of the quality of service provided to that task. This is, in turn, related to the resources allocated to the task. One possible dimension of quality of service is the timeliness of the completion of the task. This is discussed in Section 4.2

One common formulation of the soft real-time scheduling problem assumes those tasks have deadlines and a lateness constraint. The lateness constraint can take many forms and can be a composite constraint. If we define $\alpha(x)$ to be the long run fraction of jobs that miss their deadline by more than x time units, then lateness constraints are typically of the form $\alpha(x) \leq \beta$. For example, for the *firm deadline case*, we require that $\alpha(0) \leq \beta$ for some value of β, *i.e.*, we limit the fraction of tasks that miss their deadline to some value β. Here missing a deadline by any amount is considered to be a failure, and the fraction of such cases must be bounded by some value β. The quantity $\alpha(-\infty)$ gives the fraction of tasks that are never completed (including those that were never accepted for processing). One could also specify a bound on this quantity. In general, one could require $\alpha(x_i) \leq \beta_i$, $1 \leq i \leq m$ for a set of time values $\{x_1, \cdots, x_m\}$ and constraint specifications $\{\beta_1, \cdots, \beta_m\}$.

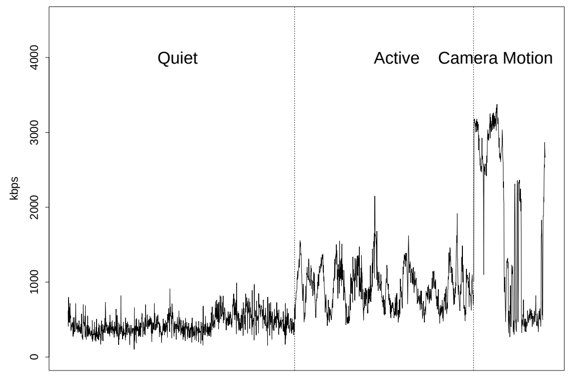

Figure 1. Video-Conference Bandwidth Usage

With such a lateness constraint specification, one can now take the stochastic nature of task arrivals and computation times into account. This leads to the concept of the *average-case schedulability*, the largest average workload that can be processed while still meeting the lateness constraint.

The soft real-time criterion opens up the possibility of having a predictable system with much higher levels of utilization than can be achieved under the hard real-time task formulation. Of course, this increased utilization level can be obtained only at the cost of either some missed deadlines or some jobs denied access to the system, or both.

4.2. Resource Allocation Among Soft Real-Time Tasks

Section 4.1 presented the example of a video-conference application and showed the large variation in bandwidth requirements associated with such applications. This application has processing, bandwidth, and timing requirements; however, it is possible to vary the task's resource requirements by varying the quality of service provided to it. One can select varying levels of video and audio quality and can permit a certain amount of packet lateness. Changes in these parameters can reduce the resource requirements at the cost of reduced quality of service.

Recently, new methodology has been introduced to capture the possibility of a trade-off between task resource requirements and the quality of service provided. This can be done in many contexts, for example

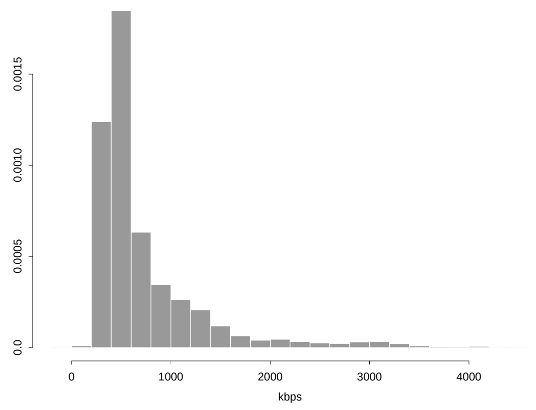

Figure 2. Video-Conference Bandwidth Usage Histogram

trading off task lateness against increased average-case schedulability or energy usage against audio or video quality.

One can also consider the semantic importance or utility of a particular aperiodic task. For example, in target tracking it is important to treat tracks associated with nearby hostile targets differently from distant tracks of friendly targets. This could be done by assigning different utility to tasks as a function of their semantic importance (*e.g.*, hostile/friendly or near/distant). One could also differentiate tasks by assigning them to different priority or service classes.

One can combine the resource-QoS trade-offs and the semantic importance approach through an appropriate utility function-based approach. For each task, one can associate a user-assigned utility function that captures the user benefit as a function of the set-points assigned to that application. One can separately determine the resources required to obtain each of the set-points. These two can be combined to obtain the utility as a function of the resources allocated to the task. If a utility function is defined for each task giving the utility derived as a function of the resources applied, then one can determine the resource allocations across all the tasks currently ready for processing to maximize the total system-wide utility. The QoS Resource Allocation Methodology (Q-RAM) [175] consists of algorithms designed to obtain near optimal resource allocations for a set of tasks to which resources must be assigned. Q-RAM allows for the possibility of multiple QoS dimensions including task timing requirements for real-time tasks.

4.3. Alternative Task Models

In Section 4.1 we noted that there are a variety of task models associated with soft real-time scheduling. Some of those models have already been discussed, under the topic of fixed-priority scheduling in Sections 2.5 and 2.6 and under the topic of dynamic-priority scheduling in Sections 3.2-3.2.3. There is also the extended periodic task model of Gardener**??**. In this section we consider another such model: the stochastic rate-monotonic scheduling model.

4.3.1. *Stochastic Rate-Monotonic Scheduling (SRMS)*

SRMS was developed by Atlas and Bestavros [17, 1] and is designed for use in systems in which periodic tasks have highly variable execution times, quality-of-service requirements, and firm rather than hard deadlines. SRMS is designed to maximize resource utilization and to minimize the use of resources on tasks that will miss their deadlines.

The SRMS workload model is based on periodic tasks with variable execution times. Each task τ_i, $1 \leq i \leq n$, is defined by a triplet $(P_i, f_i(x), Q_i)$, $1 \leq i \leq n$, where P_i is the task's period, $f_i(x)$ is a probability density function describing the task's resource utilization requirement, and Q_i is the task's requested QoS. The tasks are assumed to have firm deadlines meaning that the task must be completed by the deadline, which is assumed to be equal to the period; however, a certain fraction of tasks are permitted to miss their deadline. In particular, Q_i specifies an upper bound on the long-run fraction of jobs from τ_i that can miss their deadline. Given $f_i(x)$ and Q_i it is possible to determine the minimum resource allocation, a_i, for each of the jobs of τ_i that will ensure the QoS requirement is met.

Atlas and Bestavros [17] present an approach for harmonic task sets. They define the concept of a *superperiod*. Assume that the periodic tasks are arranged in order of decreasing priority (increasing period). Then the superperiod of τ_i is the period of the next lower priority task, P_{i+1}, with the superperiod of τ_n defined to be ∞. They take a_i to be the resource allocation to τ_i over its superperiod. In the case of a periodic task set, there will be a multiple number of jobs of τ_i that will arrive during the superperiod. Having a resource allocation for this set of jobs rather than for each individual job allows for smoothing of the utilization over a set of jobs and ultimately for more jobs being admitted and processed to completion within their deadline.

The authors define a set of tasks $\tau_1, \cdots, \tau_n$ to be schedulable under SRMS if τ_i is guaranteed to receive a_i units of resource over its superperiod, where a_i satisfies the QoS requirement Q_i for $1 \leq i \leq n$. Because rate-monotonic priorities are assumed and the task set is harmonic, the schedulability criterion is $\sum_{i=1}^{n} \frac{a_i}{P_i} \leq 1$. In [17], Atlas and Bestavros provide examples of the methodology, present comparisons with alternative scheduling methods, and give extensions of the basic SRMS methodology. The case of non-harmonic periods is addressed in [17].

4.4. The Fully Stochastic Case

We next turn to the case in which jobs from soft real-time tasks have stochastic interarrival times, computation times and deadlines. Assume all jobs are processed to completion and there is a constraint on the long-run fraction of jobs that miss their deadline. In this fully stochastic case, it is difficult to provide an exact analysis of the average-case schedulability. Recently, Lehoczky and coauthors [74, 113, 118, 120, 119, 121, 227] developed an approximation method to compute the fraction of late tasks for the heavy traffic case in which the system has high average-case utilization. The method is called Real-Time Queueing Theory (RTQT), because it is an extension of traditional queueing theory in that it takes the timing requirements of tasks explicitly into account, and its performance metric is the fraction of the offered load that completes within its deadline.

RTQT is developed under the assumption that Earliest Deadline First (EDF) is used as the scheduling algorithm. Interestingly, the results obtained for EDF systems can be used to provide results for FIFO, processor sharing, and fixed-priority schemes. RTQT was first introduced by Lehoczky [118]. The single queue case was put on a firm mathematical foundation in the paper by Doytchinov *et al.* [74]. The theory has also been extended to the case of open queueing networks with multiple independent traffic flows [113].

The analysis of real-time queues scheduled using EDF requires that deadline information be kept for every task. For RTQT, each task's lead time (time remaining until the task's deadline elapses) is kept. The addition of these lead-time variables results in a high dimensional state space. As a result, the analysis of the general problem is analytically intractable. Fortunately, excellent approximations are available in the heavy traffic case ($\rho \rightarrow 1$) where ρ is the traffic intensity, *i.e.*, the long-run task arrival rate times the long-run average task computation time. The heavy traffic case is the most important for real-time systems, since if one can accurately approximate the behavior of the system in this case, it will provide a bound for any lighter traffic case. Heavy traffic queueing theory is a well developed theory. RTQT adds the analysis of real-time tasks to this theory.

Traditional queueing theory studies the workload process, $\{W(t), t \geq 0\}$ or the queue length process, $\{Q(t), t \geq 0\}$. Here, $W(t)$ is equal to the sum of all remaining computation times of all tasks still in the system at time t. Similarly $Q(t)$ equals the number of tasks in the system at time t. The workload process decreases linearly at unit rate until the system is empty or until a new tasks arrives, at which time it jumps by an amount equal to the computational requirement of the arriving task.

Heavy traffic queueing theory considers a sequence of queueing systems indexed by some parameter $n \to \infty$. That is as n increases, the traffic intensity of the corresponding queueing system is assumed to increase to 1. For the n^{th} system assume that $\rho^{(n)} = 1 - \gamma/\sqrt{n}$ for some constant $\gamma > 0$. If one considers a scaled version of the workload and the queue length processes where time is scaled by n and the processes are scaled by $\sqrt{n}$, then these converge to drifted reflected Brownian motion processes. Specifically, if $W^{(n)}(t)$ and $Q^{(n)}(t)$ represent the workload and queue length of the n^{th} system at time t, then $\hat{W}^{(n)}(t) = W^{(n)}(nt)/\sqrt{n} \Rightarrow W^*$, and $\hat{Q}^{(n)}(t) = Q^{(n)}(nt)/\sqrt{n} \Rightarrow \mu W^*$, where W^* is a reflected Brownian motion with drift $-\gamma$ and some scale parameter θ, and $\Rightarrow$ symbolizes weak convergence.

Much is known about this limiting process, so many features of the system performance can be approximated using facts about Brownian motion. While we only describe the single node case, the ideas extend to networks with the Brownian motion process being replaced by a multi-dimension Brownian motion restricted to an orthant.

RTQT provides an approximation to the long-run fraction of tasks that are late. It also allows one to estimate the fraction of tasks with lead times less than x for any time x as a function of the current queue length. For example, if $x = 0$, this estimates the fraction of tasks in the queue that are late. These quantities can also be estimated given the current workload. The estimates depend upon the current workload and the scheduling policy.

Suppose the jobs arrive with stochastic deadlines drawn from a cumulative distribution function $G(x)$ and suppose at a moment in time the workload is given by $W(t)$. Let the *frontier*, $F(t)$, be defined by $W(t) = \int_{F(t)}^{\infty}(1 - G(u))du$. Then, if tasks are scheduled according to the EDF queue discipline, the fraction of tasks having current lead times greater than x is given by $\mu \int_{\max(F(t),x)}^{\infty}(1 - G(u))du$, where μ^{-1} is the mean task execution time.

When the frontier falls below 0, the tasks departing the queue have negative lead times, hence they are late. Since the frontier, $F(t)$, is determined by the equation $W(t) = \int_{F(t)}^{\infty}(1 - G(x))dx$, and lateness is associated with $F(t) \leq 0$, one can equivalently express the condition

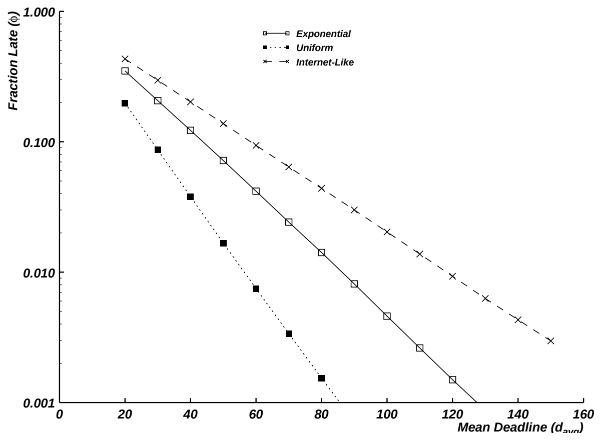

Figure 3. Task Lateness as a Function of Mean Deadline

for lateness in terms of the workload, $W(t) \geq \int_0^\infty (1 - G(x))dx = E(D)$, the mean task deadline.

It follows that RTQT predicts that under the EDF scheduling policy, arriving tasks will miss their deadline when the workload exceeds the mean of the deadline distribution. If we assume that the workload can be approximated by a drifted reflected Brownian motion process, then the stationary distribution of that process is given by an exponential distribution with some parameter θ involving the means and variances of the interarrival distribution and computation time distribution. The probability that such a Brownian motion process exceeds any level, L, is given by $e^{-\theta L}$. Consequently, for the single node case, RTQT predicts that $P(\text{Lateness}) = P(W(t) \geq E(D)) = e^{-\theta E(D)}$, or $\log(P(\text{Lateness})) = -\theta E(D)$. Hence the logarithm of the lateness probability is linear in the mean deadline. Figure 3 presents the results of a simulation study showing how lateness varies with the mean deadline. The different curves are results for different computational time probability distributions, ranging from low variability to moderate variability (exponential) to high variability (file sizes on the Internet). The linearity of the curves confirms the reasonableness of the RTQT prediction.

4.5. Future Research Involving Soft Real-Time Systems

While many advances have been made in real-time scheduling over the last 25 years, important research problems remain. In this section we highlight three broad research areas, each of which applies to most of the approaches discussed in this section: 1) the need to introduce practical system aspects into the soft real-time task models, 2) the need to develop new analytic methods to predict the performance of systems processing soft real-time tasks, 3) the need to introduce control mechanisms into the models to optimize the performance of the systems.

4.5.1. *Refinement of Models to Include Important System Effects*

One of the achievements of the last 25 years of scheduling theory was the evolution from a basic theory for idealized systems to a theory that included many important system effects such as blocking due to task synchronization, precedence constraints, or lack of preemption, mode changes, operating system overhead, and architectural details. At this point, much of the analytic work associated with soft real-time tasks has been done under idealized assumptions.

Consider, for example, the introduction of blocking effects. The early work on real-time scheduling assumed that all real-time tasks, both hard and soft, were independent. It will be necessary to develop new scheduling algorithms to handle the case in which all tasks may experience blocking effects such as task synchronization. As presently defined, aperiodic servers such as the sporadic server may become blocked when the task using its capacity becomes blocked. If the server keeps its capacity while it is blocked, then it can become more invasive than the sporadic server which may lead to missed periodic deadlines. If its capacity is lost due to blocking, then it will lead to longer aperiodic response times. New server algorithms may be needed and new analytic techniques must be developed. None of the other models described in this section gives a serious analysis of the problems that arise in estimating soft real-time task response in the presence of blocking effects.

The analyses of some models, such as the extended periodic task model of Gardener[80] and RTQT, use queueing theoretic methods to predict real-time performance. It will be challenging to extend these analyses to include some of the system effects described above. In addition, much research remains in the real-time network case. In the RTQT analysis of networks of real-time queues, the assumption is made that tasks have independent processing times at different nodes in the network. This assumption may be unrealistic as it seems more reasonable to assume that the computation times are positively correlated. For example, if the processing times are associated with the size of a packet or file, then as the file moves from node to node, if the computation requirements are large at one node for this task, they are likely to be large at another node. If there is such a positive correlation effect, it is likely that it would lead to poorer performance than would be predicted by independence formulation. Thus, new methods are need to handle the case in which dependencies of this sort occur.

4.5.2. *The Need for New Analytic Methods*

Real-time scheduling theory has been most fully developed for the case of predicting whether a set of hard-deadline periodic tasks will meet their deadlines given a certain set of assumptions about the task set and the system. Fewer analytic results are available to predict the performance of soft real-time tasks, either the fraction of those tasks that will meet their timing requirements if they have deadlines or the delay incurred by aperiodic tasks if they do not. For example, consider the case of aperiodic server algorithms designed to offer enhanced response to aperiodic tasks while still ensuring that the deadlines of hard-deadline periodic tasks are still met. Few analytic results are known for this case, even assuming that there are no blocking effects and all overhead is neglected. It may be possible to derive approximations in the heavy traffic case using some of the ideas exploited by RTQT.

On the other hand, while the RTQT approach does appear to be promising, it does have significant limitations. For example, for RTQT to offer accurate predictions of the lateness behavior of real-time tasks, it is necessary for the individual tasks to comprise a small part of the total workload. The situation in which the system is processing a relatively small number of relatively large tasks does not lend itself to the type of analysis used in RTQT. The theory offers accurate results when the queues are relatively long, and the tasks in the queues have relatively short execution times. RTQT takes advantage of the "averaging effect" of processing a large number of tasks. When a few tasks are involved, each must be individually considered to accurately determine lateness.

In general, each of the models and associated analytic techniques mentioned in this section must be generalized in order to realize the ultimate goal of building a comprehensive theory in which one can make accurate predictions about the real-time performance of a representative system carrying a fairly general workload.

4.5.3. *Introduction of Control Mechanisms*

An important area of research is the introduction of control mechanisms to real-time systems to improve real-time performance and to allow the system to adapt to changes in the environment, the workload, or to changes in the system architecture due to failures. The use of feedback control to improve the performance of soft real-time tasks is addressed in the next section of this article, so we make only a few comments here.

Generally, the performance criterion for soft real-time tasks is minimizing their delay, when they do not have explicit deadlines, or minimizing the fraction of tasks that do not meet their timing requirements (including those never admitted to the system). In much of the current work, scheduling and admission control policies are based on the assumption that the computation requirements of each task (their C) is known. Actually, there are only a few situations in which C is observable, for example when it is constant or when the task involves a file of known size. Usually, a task has a variable execution time that is not known. Consequently, admission control policies and scheduling policies based on knowledge of task execution times often cannot be accurately implemented. There is a need to develop new policies that do not depend on this knowledge.

In addition, it will be useful to consider scheduling policies other than fixed and dynamic-priority policies such as rate-monotonic and EDF. These policies are designed to optimally deal with hard-deadline periodic tasks, but other policies might be considered for handling the soft-deadline aperiodic tasks. For example, it is known in standard queueing contexts (*e.g.*, $M/M/1$) that the Shortest Remaining Processing Time (SRPT) policy gives the smallest mean response time of any policy. Of course, SRPT requires knowledge of the computation times, and this policy does not deal with the case in which there are hard-deadline tasks in the system as well. Nevertheless, alternative scheduling policies should be considered. Moreover, some consideration should be given to the development of admission control policies that will optimize system performance.

5. Feedback Scheduling

Tarek Abdelzaher, Karl-Eric Arzen and Anton Cervin

The increased complexity of embedded applications beyond what can be efficiently captured and analyzed by detailed low-level schedulability models calls for a fundamentally different approach to real-time system modeling that captures and controls *macroscopic* behavior. In this approach, models attempt to describe the aggregate behavior of a system directly in much the same way flow equations capture the aggregate behavior of a flow without synthesizing it from the dynamics of individual particles. It is observed that real-time performance of computing systems can generally be inferred from the behavior of resource queues (such as the scheduling queue, communication port queues, and semaphore queues). At a high level, a queue is an integrator of request flows, which can be modeled by a difference equation and is therefore amenable to control-theoretic analysis [77] offered by feedback control theory.

Feedback is a powerful concept that has played a pivotal role in the development of many areas of engineering. It has several advantages. Feedback can make a system robust towards external and internal disturbances and uncertainties. Using feedback, it is possible to create linear behavior out of non-linear components and to modify the dynamic behavior of a system. The advantage of using feedback in conjunction with real-time scheduling is that precise schedulability models are no longer needed. Instead, the system can adapt its load and resource allocation dynamically in a controlled and analyzable manner to achieve the desired temporal behavior. Feedback requires the use of actuators manipulated by a controller to act on the system and change its performance. In real-time computing, most actuators change either the enqueue or the dequeue rates of components (called *enqueue actuators* and *dequeue actuators* respectively), resembling valves in a flow control system.

Enqueue actuators might perform admission control or redirection of input requests to another server. Dequeue actuators manipulate the resources available to the server or change the resource requirements of requests (*e.g.*, by changing the quality of service). Eliminating the need for precise low-level models, and improving the ability to adapt to unpredictable load and resource fluctuations are the two main advantages of the feedback scheduling approach.

Initial efforts in applying feedback control to computing systems borrowed their theoretical foundations directly from prior control literature. Successful control schemes were developed for high-performance servers [71, 79, 151], multimedia streaming [164, 228], real-time databases [12], web storage systems [144], network routers [110, 63], active queue management schemes [98, 99], and processor architecture [204] to name a few. General-purpose resource management middleware frameworks, such as ControlWare [229] and IBM's AutoTune Agents [72], were developed that embody a control-theoretic paradigm for software performance tuning. Simple feedback controllers were also proposed as a way to manage software complexity [60] based on the assertion that a primitive but more reliable emergency backup entity could significantly improve the reliability of a complex primary system [189]. The proliferation of feedback control approaches for software performance management led to a textbook on the topic, where feedback control is explained by software examples [95]. A good survey of feedback-scheduler co-design is found in [16].

While simple application of control theory in scheduling has proved beneficial, much more work is needed to develop precise foundations for this approach. In the following subsections, we highlight some key challenges and current efforts to solve them in feedback control scheduling, then conclude with a discussion of open problems.

5.1. Delay Control

Consider the problem of controlling the delay of a server to adhere to a specification. Delay control is akin to level control in a water tank. The simplest delay control loop is to measure the current delay then adapt (*e.g.*, by changing the resource allocation) in the direction that causes the delay to approach its set point. Approximating this loop by a level control loop allows using textbook techniques to analyze its convergence properties and set controller parameters for the desired settling time.

Unlike a water tank, however, the delay control loop is event driven, since changes in measured response-time values and their averages occur only at discrete event times (such as the instances when requests are dequeued). Measurements of interest have an event-based statistical nature that depends in part on the statistics of the request arrival process. The underlying statistical relationships are known to be non-linear from queueing theory. Existing stochastic control techniques are inapplicable because they assume additive external noise that is derived from white noise using a linear filter, which is not the case with non-linear stochastic queueing dynamics. Moreover, requests in a queue may be scheduled using any of a wide range of scheduling policies that are difficult to model in a control loop. Hence, delay control opens interesting opportunities for integrating traditionally disjoint theoretic foundations such as control, queueing, and scheduling theory in order to understand and control software timing properties.

To accommodate the non-linearity offered by queueing dynamics, it was proposed in [193] to augment feedback control with prediction such that *future* delay fluctuations are anticipated and corrected before they occur. From queueing theory, we know that for an M/M/1 queue with arrival rate λ and service rate μ, the long-term average queueing time for the clients is $D = \frac{1}{\mu - \lambda}$. The above equation can be solved for μ that achieves a desired delay D. The resulting μ is enforced by the dequeue actuator. A linear controller was shown to be sufficient for eliminating any residual errors. This idea was extended to multiple classes in [149]. One problem with queueing-theoretic models, however, is that queueing theory describes relations between long-term averages only. In the short term, delay and rates may fluctuate. Unlike physical processes, statistical properties (such as probabilities) are not directly measurable. Too small a sample gives us inaccurate estimation of the statistics even if the traffic is stationary during measurement. Too large a sample size gives us inaccurate estimates when the arrival pattern changes.

An enhanced prediction scheme proposed in [96] computes the needed service rate from actual measurements of recent delay and queue length. In general, the integration of feedback control with model-based prediction derived from characteristic software performance models remains one of the most interesting problems in developing feedback scheduling algorithms.

5.2. Priority Scheduling

The most common performance metric addressed in previous feedback priority-based scheduling literature is the deadline miss ratio [147, 186]. It is desired to ensure that no deadlines are missed. The basic control loop architecture is the same as that of the basic delay control loop, except that the set point is defined on the deadline miss ratio.

Several challenges are immediately apparent. First, classical control theory does not have a notion of priorities. Hence, modeling the relation between delays (at a given priority level) and system inputs is not straightforward. Second, ideally, the desired miss ratio is zero. However, zero is also the saturation limit of the miss ratio sensor (since the miss ratio cannot be negative). Said differently, the deadline miss ratio does not provide a useful feedback signal when the system is underutilized. Control loops that operate at or near saturation points of their components are necessarily non-linear. To address this problem, control loops were proposed that measure both miss ratio and average system utilization [148, 145], quantifying both underutilization and overload. Adaptive control techniques have been proposed to adapt to miss ratio non-linearities [225]. The basic approach has also been extended to distributed systems [212].

Unfortunately, in the case of aperiodic tasks, average utilization (the average percentage of time a processor is busy) is not a good metric for predicting deadline misses. For example, two very short aperiodic tasks with unschedulable deadlines can arrive at an idle processor and miss their timing constraints while the average utilization could be arbitrarily small. The research question becomes, how to derive predictors that quantify the proximity of an aperiodic task system to the boundary of schedulability, and hence alert to possible deadline misses before they occur? A recent answer to this question lies in *synthetic* *utilization*[4] bounds [3]. It is shown in [3] that all aperiodically arriving tasks are guaranteed to meet their deadlines under an optimal fixed-priority policy as long as $U(t) \leq 2-\sqrt{2}$, regardless of the individual task arrival times, computation times, and deadlines. A control loop that keeps synthetic utilization just below this bound does not suffer the miss ratio saturation problem mentioned above and will ensure system timeliness. Excursions above the bound predict possible future deadline misses. Extensions of this predictive metric to non-independent tasks and resource pipelines are found in [2] and [5] respectively. The first practical implementation of a synthetic utilization control loop was described in [198] in the context of server delay control.

5.3. Multi-Class Resource Partitioning

The lack of priority support in control theory literature led to resource partitioning approaches that eliminate priority queues altogether from the feedback scheduling problem. These approaches commonly use a two-tier architecture for feedback-based delay guarantees. The top layer allocates a separate partition of the server resource to each traffic class together with its own FIFO queue as done, for example, in [173, 4, 129, 143]. If the underlying resource is indivisible, such as the CPU or a single-channel communication link, the bottom layer implements a logical resource partitioning abstraction to be used by the upper layer. For example, in [7] an adaptive reservation strategy is proposed for controlling the CPU bandwidth reserved to a task based on QoS requirements. A two-level feedback control is used to combine local application level mechanisms with a global system-level strategies. In [48], feedback is used in combination with elastic scheduling to estimate the actual load and adapt task periods to reach a desired utilization factor, without forcing the user to provide execution time bounds. The effect of the bottom-level scheduler must be accounted for in control loop design. For example, it was shown that a WFQ scheduler introduces a delay into the control loop since weight changes are applied only to timestamps of future arrivals and not requests already enqueued [230].

In general, the multi-class resource partitioning problem can be viewed as a constrained optimization problem where some problem-specific notion of utility is maximized by adapting per-class resource allocation subject to resource constraints. A non-linear formulation of this optimization problem based on queueing models is presented in [129], where a notion of web cluster utility is maximized by resource allocation. In [116], a linear formulation is presented, which uses opti-

[4] Also known as instantaneous utilization.

mal feedback control to achieve the best allocation of CPU shares to periodic tasks.

A robust (non-linear) H_∞ control model is derived in [31] for a single-server multi-class queueing system, where the objective is to determine which class (flow) to schedule next on the server resource. The optimality criteria of the resulting schedule, however, do not include delay metrics.

Mapping the original per-class performance specification to a feedback control problem is not always straightforward. Multiple mappings may be possible with complex trade-offs. For example, one very commonly proposed performance specification in multi-class systems is the weighted fairness guarantee. In this problem, the ratio of some performance metrics across different classes is specified instead of absolute values, creating coupling between per-class control loops. Formally, it is required that $\frac{D_{i+1}(k)}{D_i(k)} = \frac{c_{i+1}}{c_i}, i = 1, \cdots, N1$, where $D_i(k)$ is the average performance (*e.g.*, delay) of traffic class i at time k, and c_i is a constant per-class weighting factor representing the user's relative performance specifications. In [151, 150] a solution uses per-class control loops, where error $e_i(k)$ of $class_i$ at sampling time k is given by the expression $e_i(k) = \frac{c_i}{\sum_{j=1}^{N} c_j} - \frac{D_i(k)}{\sum_{j=1}^{N} D_j(k)}$. The controller computes a corresponding correction Δb_i to the current resource allocation b_i. In [149], the average of $\frac{D_1(k)}{c_1}$, $\frac{D_2(k)}{c_2}$, ..., $\frac{D_N(k)}{c_N}$ is taken as the common set point letting individual ratios converge to their average. In [143], the performance error of consecutive class pairs is defined as $e_i(k) = \frac{c_{i+1}}{c_i} - \frac{D_{i+1}(k)}{D_i(k)}$. While all three mappings allow loop convergence to the specification, the most appropriate mapping is not immediately obvious without a more fundamental analysis of the properties of the resulting control loops.

5.4. Feedback Scheduling of Control Systems

An important special case is where the computing system being controlled is itself a set of digital control loops. The objective for the feedback scheduler is to dynamically adjust the CPU utilization of the controller tasks. The feedback scheduling problem is stated as a optimization problem where the objective is to maximize the global control performance of the physical system subject to resource and schedulability constraints.

Feedback is needed for the scheduler because of the uncertainty associated with the WCET estimation. The actuators in the feedback scheduler loop modify the CPU utilization of individual controller tasks. The task period is a natural actuation knob. For a controller implemented on input-output form it is generally more difficult to change the sampling period than for a controller that is realized on state-space form. In certain cases it may be necessary to use a Kalman filter to estimate the values of the state at the new sampling instants. Controller parameters may need to be adjusted when the task period is changed. An alternative actuation knob is the execution time of the controller. This can be done using a multiple-versions approach or using an anytime approach, for example, in model-predictive controllers (MPC) in which an quadratic optimization problem is solved iteratively in every sample. The sensor is usually a measurement of the actual CPU utilization. In order to avoid control actions caused by outliers, a low-pass filter may be included to calculate the average utilization over a period of time. Such filters are an important source of loop dynamics. Different controller structures can be combined in feedback scheduling of control systems. A pure feedback scheme is reactive in the sense that the feedback scheduler will only remove a utilization error once it is already present. By combining the feedback with feedforward, a pro-active scheme is obtained (*e.g.*, [57]). The feedforward path is used to allow controller tasks to inform the scheduler that they are changing their desired amount of resources and to give the scheduler the possibility to compensate for this change before any overload has occurred. The feedforward path can be also be used for dynamic task admission.

5.4.1. *Performance Optimization*

The idea to treat the scheduling problem as a performance optimization problem was first put forth in [187]. The key observation is that a digital controller can give satisfactory performance within a range of sampling periods. Hence, the task period must not be treated as a given parameter. Instead, stating the performance of each controller as a function of its sampling rate, the objective is to maximize the overall performance subject to a schedulability constraint. If the performance functions are convex and decreasing functions of the task rates, this leads to a convex optimization problem.

Stating the scheduling problem as an optimization problem, a suitable performance metric must be used. In [187], the control performance is expressed by *cost functions*, commonly found in optimal control problems. Using a quadratic cost function, the performance of each controller is described by $J_i(T_i) = \int (x^T Q_1 x + u^T Q_2 u) dt$, where T_i is the task period, x is the plant state, u is the control signal, and Q_1 and Q_2 are weighting matrices. The optimization problem is stated as $\min_{T_1,\ldots T_n} \sum J_i(h)$ subject to a schedulability constraint. In [188] and [59], algorithms are proposed that optimizes the expected control performance under the constraint that the task set is schedulable.

The optimization problem can also be solved on line via feedback to the scheduler. For this purpose, a model of the scheduler is needed from a control perspective. The dynamics involved in feedback scheduling are often of low order or even purely static. The reason for this is obvious. If a task is given more or less CPU time the total utilization will change as soon as the next job of the task is started. Often the dynamics in the feedback loop comes from the filtering in the sensor. A consequence of this is that it is often enough with very simple control strategies in the feedback scheduler.

In [75, 57] it was shown that a simple linear rescaling of a set of nominal task periods in order to meet the utilization set-point is optimal with respect to the overall control performance under certain assumptions. It holds if the cost functions are quadratic, $J_i(h_i) = \alpha_i + \beta_i h_i^2$, or linear, $J_i(h_i) = \alpha_i + \gamma_i h_i$. The advantage of this is a simple and fast calculation that easily can be applied on line. The linear rescaling also has the advantage that it preserves rate-monotonic ordering of the tasks and, thus, avoids any changes in task priorities in the case that rate.monotonic fixed-priority scheduling is used. Linear or quadratic functions are quite good approximations of true cost functions in many cases. If the task set includes both tasks with quadratic cost functions and tasks with linear cost functions, the solution is not as simple, although it is still computable.

5.4.2. *Isolating the Scheduling and Control Subsystems*

A control task generally consists of three main operations: input data acquisition (sampling), control algorithm computation, and output signal transmission (actuation). The timing of the input and output actions is critical to the performance of the controller. Ideally, the sampling should be jitter-free, and there should be zero delay between reading the input and writing the output.

The seminal Liu and Layland paper [136] only concerns the scheduling of the control computations. It is assumed that the input and output actions are handled by external functions. The scheme introduces a delay of one period in all control loops closed over the computer. This delay must be accounted for in the control design.

In modern systems, the input and output actions are often triggered by software. If the operations are carried out within the task body, considerable amounts of jitter may result. From a control-theoretic perspective, it is useful to distinguish between *sampling jitter* (variation in the input instant) and *input-output jitter* (variation in the delay from input to output).

The timing parameters of distributed control loops are further discussed in [223]. From a design point of view, there is a fundamental trade-off between delay and jitter. Using hardware functions or dedicated, high-priority input and output tasks, it is possible to exchange jitter for delay. Both delay and jitter degrade the control performance, and it cannot be said which is worse is general. It is, however, easier to predict the performance degradation due to a constant delay. Furthermore, it is straightforward to account for the delay in the control design.

Eliminating the jitter greatly simplifies the performance optimization problem, since it creates a separation between the control and scheduling subsystems. This approach is taken in the Giotto programming language [97], I/O and communication are time-triggered and assumed to take zero time, while the computations in-between are assumed to be scheduled in real-time. drawback with the Giotto approach is that a minimum of one sample input-output delay is introduced in all control loops. This problem is remedied in the Control Server [56]. A control server creates the abstraction of a control task with a specified period and a fixed input-output delay shorter than the period. Individual tasks can be combined into more complex components without loss of their individual guaranteed fixed-delay properties. The single parameter linking the scheduling design and the controller design is the task utilization factor. The proposed server is an extension of the constant bandwidth server [6]. Where isolation is not achieved, controllers must tolerate or account for scheduling effects. There are two control design approaches to the problem of scheduling-induced jitter. The first approach is to design the controller to be robust against jitter. The second approach is to actively compensate for the timing variations in each period. Examples of the latter approach are found in [168, 154, 133]. It is also possible to reduce the delay and/or jitter by using more detailed scheduling models.

For instance, it is possible to treat the reading of the input, the control computation, and the writing of the output as subtasks. This approach has been investigated in [55, 30]. Jitterbug [134] is a tool for analytical evaluation of quadratic cost functions, taking sampling period, delay, and jitter into account.

5.5. Challenges

In summary, the application of feedback control to computing offers several important challenges for future research. Computing systems are generally event driven. Much of the classical feedback control theory assumes time-driven systems. Event arrival in computing systems is a stochastic process. It is often desired to control the properties of that stochastic process. Techniques such as stochastic control are powerful in minimizing the effect of additive noise but assume that the underlying (noise-free) system is both linear and deterministic. Neither of these assumptions holds in computing systems. The non-stationarity of the input process poses further challenges with respect to performance measurement. The optimal choice of the observation window size is complicated by trade-offs between measurement accuracy and system ability to adapt to changes in stochastic input patterns.

Modeling of priority-based systems for purposes of feedback control is another fundamental challenge. Additional research is needed to relate per-task metrics (such as meeting individual deadlines) to aggregate metrics (such as flavors of utilization) that abstract the notion of priorities away from the feedback controller. Mapping performance specifications into control loops remains an open problem. High-level specifications might entail complex relations between low-level system outputs that are the controllable parameters. Satisfying such complex relations may require extensions to the analysis and design of scheduler control loops. Architectures where computing systems control physical processes present two types of control loops that need to be jointly co-designed. One controls the allocation of computing system resources and one controls the actual external processes. The complex interactions between the two call for new foundations for scheduler design based on optimizing joint metrics that span both the computing and physical system domains.

While a plethora of individual feedback scheduling problems have been addressed, the main challenge remains in generalizing feedback control solutions to a comprehensive theory for performance management in software systems. This section illustrated some of the most important challenges and the current state of the art.

6. Further Extensions of Scheduling Models

Aloysius K. Mok

Real-time scheduling theory was originally developed for periodic tasks with hard deadlines. Over the years, substantial generalizations of this basic task set model have been considered. Other scheduling models for addressing application-specific timing constraints that cannot be directly modeled by Liu and Layland's basic task set model have also been studied by many researchers. We survey some of these contributions below.

The Multi-frame model [160] was invented to more accurately model periodic tasks whose execution time varies from period to period. An example of such applications is a video stream in a multimedia application where the size of a video frame varies according to some pattern that depends on the encoding scheme of the video stream. The simple periodic task model is too pessimistic because a conservative execution time specification must use the largest frame size. This problem is solved in the multi-frame model by allowing a sequence of numbers to be specified as the execution time in successive periods of a periodic task. The variation in the per-period execution time can be represented by a finite vector with the understanding that the execution time of successive periods is to be generated by repeating the finite vector *ad infinitum*. Schedulability bounds for the multi-frame model can be found in [160]. Further generalization of the multi-frame model is discussed in [32].

Although the periodic [136] and sporadic [158] task models have been the most widely studied notions of recurrent execution, event occurrences in reality often are neither periodic nor sporadic. For example, in an application that services packets arriving over a network, packet arrivals may be highly jittered. A scheduling scheme that assumes periodic or sporadic arrivals complicates the design of such an application. Rate-based scheduling schemes [105] are more seamlessly able to cope with jitter. In such schemes, there is no restriction on a task's instantaneous rate of execution, but an average rate *is* assumed. In multiprocessor systems, rate-based execution can be ensured by using scheduling algorithms that also ensure a property called *proportionate fairness* (Pfairness) [34]. Under Pfairness, each task is required to make progress so that its allocation error is always less than one quantum, where "error" is determined by comparing to an ideal fluid system.

In research on rate-based uniprocessor scheduling, Jeffay and colleagues [105, 107, 106] derived necessary and sufficient conditions for determining the feasibility of a rate-based task set and demonstrated that earliest-deadline-first scheduling is optimal for both preemptive and non-preemptive execution environments, as well as hybrid environments wherein tasks access shared resources. Baruah and colleagues originated work on Pfair scheduling and showed that Pfair scheduling algorithms can be used to optimally schedule periodic tasks on multiprocessors [34]. Srinivasan and Anderson extended this work by showing that sporadic and rate-based tasks can also be optimally scheduled [209]. Other extensions to early work on Pfair scheduling include techniques for reducing task migrations (*e.g.*, [157]), for ensuring synchronization requirements (*e.g.*, [100]), for supporting soft- and non-real-time components (*e.g.*, [210, 211]), for adaptively changing task share allocations at runtime (*e.g.*, [13]), and for reducing scheduling overheads (*e.g.*, [58, 101, 103]).

The Pinwheel model was invented to capture the requirements of sensor data acquisition tasks that must be executed sufficiently frequently so that the system can be assured that the sensor data is never too stale, *i.e.*, the sensor data value maintained by the system is always sufficiently close to the value of the real-world object being sensed. This requirement can be captured by representing sensor tasks as Pinwheel tasks. A set of n Pinwheel tasks is a multi-set of n integers A $= \{a_1, a_2, \ldots, a_n\}$. A successful schedule S is an infinite sequence "j_1, $j_2, \ldots$" over the index set $\{1, 2, \ldots, n\}$ such that any subsequence of a_i $(i \leq n)$ consecutive entries in S must contain at least one of the index i. Intuitively, this ensures that the i^{th} sensor task must be executed at least once in any interval of length a_i. The Pinwheel scheduling problem has been surveyed in [61]. Related work in distance-constrained scheduling can be found in [90].

Several other interesting variations on the periodic task model have been proposed, which cannot be decribed here due to space constraints. Examples include Gardener's *extended periodic task* model[80], and the *hyperperiodic task* model of Aggarwal and Chraibi[9, 10, 11].

A number of real-time data-flow models have also been considered by various authors to capture data/buffer-driven constraints. These models are designed to capture the requirements of applications where there are no explicitly specified deadlines but rather, deadlines are induced by the flow of data through a network of operators, and the processing of data must be completed by each operator before input/output buffer overflow occurs. The severity of the timing constraints depends on the availability of buffer space at the input/output edges of each operator. These applications are typical of signal processing and message-based transaction systems. Examples of work that treats real-time data-flow models are [159] and [84].

More recently, there has been substantial interest in investigating ways to extend real-time scheduling solutions to an *open system* environment. In an open environment, multiple real-time applications must share the same execution platform where each application may have its own set of real-time tasks specified in any of a number of real-time task scheduling models. The difficulty in real-time scheduling is compounded by the constraint that there may not be any single entity that has knowledge of all the scheduling models of the applications and their associated task parameters, and thus a global scheduler that schedules all the tasks in all the applications is not possible. An open system environment can be supported by a two-level scheduler. A resource-level scheduler is responsible for allocating time to the applications each of which claims a ”reserve”, and every reserve is to be allocated its budgeted time within a guaranteed latency bound. An application-level scheduler schedules tasks sharing a reserve according to the algorithm chosen for the tasks whenever their reserve is allocated time. The goal is for the two-level scheduling to provide the tasks sharing the reserve with timing isolation, protects them from ill effects of resource contention, enables the real-time performance of the tasks to be tuned and verified without regard to how tasks not sharing the reserve are scheduled. The open system problem was first treated in [67] where the Liu and Layland task model is assumed for all applications. An alternative approach has been proposed by Mok and Feng [161]. These authors regard the resource allocation problem as that of scheduling each individual application on a dedicated (virtual) processor but the dedicated processor may run at a non-constant speed. They introduced the concept of an RTVP (Real-Time Virtual Processor) which requires the speed variation of a virtual processor to be constrained by a jitter bound. RTVPs can be implemented by scheduling a physical processor to respect the jitter bounds of the RTVPs, each of which consumes a specified fraction of the physical processor. RTVP does not pose any restriction on the scheduling model that may be assumed by any application. An application-specific scheduler that meets the timing constraints of the tasks in the application on a dedicated (real) processor will also meet the same timing constraints when the application is run on an RTVP as long as the schedule generated by the application-specific scheduler has sufficient time slacks to tolerate the supply jitter of the RTVP on which the application runs.

A dual development is the notion of a composite task that modularizes a service supported by several tasks. The two key ideas are (1) to combine and abstract the collective real-time requirements of a component as a single real-time requirement, called *scheduling interface* and (2) to compose independently analyzed local timing properties under their local schedulers into a global timing property under a global scheduler. Shin and Lee [201] introduced an approach to use the standard Liu and Layland periodic model as a scheduling interface model to develop a compositional real-time scheduling framework. They addressed the scheduling interface derivation problem by abstracting a set of periodic tasks under EDF or RM scheduling as a periodic scheduling interface. Using this framework, each component exports its periodic scheduling interface to the system and the system treats the component as a single periodic task. Using the same technique as for the scheduling interface derivation problem, they also addressed the scheduling interface composition problem by composing a set of periodic interfaces under EDF or RM scheduling as a single periodic interface. They presented schedulability conditions to check exactly if a set of periodic tasks is schedulable under EDF and RM scheduling over the worst-case supply of a *periodic* resource (*i.e.*, a periodically partitioned resource).

In cases where the workload may not be known until run time, dynamic planning-based schedulers have been proposed that focus on dynamically performing feasibility checks on the fly. A task is *guaranteed* by constructing a plan for task execution whereby all guaranteed tasks meet their timing constraints. 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. To construct a schedule, a heuristic function, H, which synthesizes various characteristics of tasks affecting real-time scheduling decisions is used. 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 [181]. While the time complexity is proportional to n, the algorithm may be 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 algorithm was also extended to handle many different situations and types of requirements [233, 235, 234, 180, 182, 215]. This algorithm was implemented as part of the Spring kernel [214].

Unlike the scheduling of a processor, a network needs to be scheduled in a distributed fashion. Zhao *et al.* was among the first to show how to implement distributed deadline driven scheduling in a network setting [231, 232] During the 1990s, the switched networks (*e.g.*, ATM networks and switched Ethernets) became popular due to their capacity and flexibility. Li *et al.* showed how to guarantee the end-to-end delays in these types of wide area networks [130]. The methodology was based on network calculus techniques. Unlike traditional schedulability analysis techniques, this new method uses functions to model arrivals and services of real-time requests. The model is hence highly flexible and can work with many kinds of arrival patterns and service disciplines The resulting outcomes are highly effective and efficient admission control algorithms for modern switched networks. Further developments along this line can be found in [167].

Because of the large variety of timeliness requirements found in real-time applications, a worthy goal is to find canonical representations of timing constraints and to seek general solutions to the constraints. Towards this end, there has been some work in formalizing the general real-time scheduling problem as some sort of constraint satisfaction problems. However, due to the infinite horizon of real-time scheduling problems, the corresponding specification of the constraints involves not a finite but infinite sets of constraints and so the constraint representation language must be powerful enough to specify infinite constraint sets. In [162], a first-order logic RTL (Real-Time Logic) was used to specify timing constraints that are made up of the logical combination of simple timing constraints where a simple timing constraint specifies a distance relation between the occurrences of two events, such as the start and completion of a task execution. This language is sufficiently powerful to specify, for example, a wide variety of timing constraints that are typically imposed on cyclic-executive-type schedulers such as those found in the Boeing 777 AIMS (Air Information Management System). Solution techniques involving fast backtrack and incremental constraint satisfaction were introduced in [162]. Related work in composing real-time schedulers to combine cyclic executive and sporadic task models can be found in [224].

7. Challenges Ahead

Lui Sha

We have reviewed some of the major real time scheduling results in the past 25 years. Looking at the big picture, we have witnessed the changing trends in real-time systems. The system architecture has changed from a federated system architecture to an integrated system architecture and then to system of systems. Each shift has brought about enormous challenges to the available technological infrastructure.

In a federated architecture, a system is characterized by a collection of private hardware resources, a small number of high volume high variability sensor data streams on dedicated links, loosely coupled distributed actions, and hardware-based isolation and protection. Under this type of architecture, the task in managing shared resources is mainly how to handle periodic data flow driven by signal processing and control. The existing real-time computing infrastructure has served most practitioners well for this type of application.

In an integrated system architecture, sensors, communication channels and processors are extensively shared. The large number of possible configurations becomes a challenge for system architects. The current generation of schedulability analysis tools offer inadequate support for system architects. They must manually create the alternative options and then check the schedulability of each option. They would like to have tools to automatically search the design space and perform sensitivity analysis regarding the uncertainty of task parameters. During the system engineering phase, the values of task parameters are often educated guesses. Finally, from the perspective of runtime reconfiguration, dynamic priority scheduling theory has potential advantages. In addition to the potential of higher schedulability, the feasibility analysis of dynamic priority scheduling is faster. We are looking forward to the maturing and subsequent use of dynamic scheduling theory in practice.

In modern integrated systems, there are substantial high volume and high variability imaging data streams. Depending on the nature of applications, they have either hard or soft end-to-end deadlines. If the images are used for steering a vehicle, they will have a hard end-to-end deadline and tight jitter tolerance. Such hard-deadline streams pose challenges to traditional scheduling theory using worst-case assumptions. The large execution time variability causes inefficiency. There are many algorithms developed to capture the unused cycles to improve local aperiodic response times. How to effectively use such transient surpluses along a path to improve end-to-end responses requires more studies. The soft real-time image data streams also pose challenges to the statistical approach. As discussed in Section 4.1, the processing times of an imaging data stream at nodes on its path are positively correlated, not independent. In addition, on congested resources such as shared buses, an image data stream can consume a significant fraction of a shared resource, instead of consuming a small fraction of a shared resource that allows the use of the law of large numbers.

A system of systems is often a large distributed system, where keeping distributed views and actions timely and consistent is at the heart of collaborative actions. Ideally, we would like to keep distributed views, state transitions and actions consistency with each other. In business systems, the consistency of a distributed system is managed by atomic operations. Simply put, atomic operations wait for every working component to be ready and then commit the operations. However, this may not be viable for real-time systems. The train must leave the station without waiting for every passenger to board the train. However, those components that are left behind with outdated views and states must quickly catch up and re-synchronize themselves with the system. If more and more components fall out of synchronization, the system of systems will fall apart. How to handle the interactions between timing constraints, consistency requirements, and re-synchronization in a large distributed system is a challenge. As a networked embedded system of systems grows larger and the coordination becomes tighter, so will be the impact of this technological challenge. The re-synchronization loop is a form of feedback control. As discussed in Section 5, feedback is a powerful technique which has yet to be fully exploit in the control of the behavior of computing systems in the face of uncertainty.

Another characteristic of a system of systems is that a wide variety of real-time, fault tolerance and security protocols are used in different systems, because most of the systems of systems are integrated, not built from scratch. It is well known that perfectly fine medicines when taken alone can react badly when taking together. Priority inversion is an example of pathological interaction between independently developed synchronization protocol and priority scheduling protocol. This is not an easy problem to solve because the scope of modern technologies is so large and complex. To advance any area, one must specialize. As a result, we have specialized real-time, fault tolerance, security communities focusing on improving the results in one dimension with little attention on how separately developed protocols may interact. We need to create a forum for the co-development/integration of real-time, fault tolerant, security, communication and control protocols. Research is needed to formally verify that protocols do not invalidate each others' pre-conditions when they interact.

Looking ahead, much remains to be done in the creation of a new real-time computing infrastructure for modern real-time systems. It will be an exciting time!

Acknowledgements

The authors acknowledge the many valuable contributions to research on real-time systems appearing in hundreds of publications that we were unable to describe and cite here. It was agonizing that we had to delete as much 50% of the original drafts of some sections, due to space constraints.

References

1. A. Atlas, A. and Bestavros: 1998, 'Statistical Rate Monotonic Scheduling'. Technical Report BUCS-TR-98-010, Boston University.
2. Abdelzaher, T. and V. Sharma: 2003, 'A synthetic utilization bound for aperiodic tasks with resource requirements'. In: *5th Euromicro Conference on Real-Time System*. Porto, Portugal.

3. Abdelzaher, T., V. Sharma, and C. Lu: 2004a, 'A utilization bound for aperiodic tasks and priority driven scheduling'. *IEEE Trans. on Computers* **53**(3).
4. Abdelzaher, T., K. G. Shin, and N. Bhatti: 2002, 'Performance guarantees for Web server end-systems: a control-theoretical approach'. *IEEE Trans. on Parallel and Distributed Systems* **13**(1), 80–96.
5. Abdelzaher, T., G. Thaker, and P. Lardieri: 2004b, 'A feasible region for meeting aperiodic end-to-end deadlines in resource pipelines'. In: *IEEE International Conference on Distributed Computing Systems*. Tokyo, Japan.
6. Abeni, L. and G. Buttazzo: 1998, 'Integrating multimedia applications in hard real-time systems'. In: *Proc. 19th IEEE Real-Time Systems Symposium*. Madrid, Spain.
7. Abeni, L. and G. Buttazzo: 2001, 'Hierarchical QoS management for time sensitive applications'. In: *Proc. IEEE Real-Time Technology and Applications Symposium*. Taipei, Taiwan.
8. Abeni, L. and G. Buttazzo: 2004, 'Resource reservations in dynamic real-time systems'. *Real-Time Systems* **27**(2), 123–165.
9. Aggarwal, S. and C. Chraibi: 1993, 'On the Scheduling of Hyperperiodic Tasks'. In: *Proc. 5th Euro-Micro Workshop on Real-Time Systems*. Oulu, Finland, pp. 112–117.
10. Aggarwal, S. and C. Chraibi: 1995, 'Scheduling of Hyperperiodic Tasks in a Multiprocessor Environment'. In: *Proc. 2nd ISSAT Conference on Reliablity and Quality in Design*. Orlando, FL, USA.
11. Aggarwal, S. and C. Chraibi: 1996, 'On the Combined Scheduling of Hyperperiodic, Periodic, and Aperiodic Tasks'. In: *Proc. 1st Conference on Performability in Computing Systems*.
12. Amirijoo, M., J. Hansson, and S. H. Son: 2003, 'Specification and management of QoS in imprecise real-time databases'. In: *Proc. International Database Engineering and Applications Symposium*. pp. 192–201.
13. Anderson, J., A. Block, and A. Srinivasan: 2003, 'Quick-release Fair Scheduling'. In: *Proc. 24th IEEE Real-Time Systems Symposium*. pp. 130–141.
14. Anderson, J. H., S. Ramamurthy, and K. Jeffay: 1995, 'Real-Time computing with lock-free shared objects'. In: *Proc. 16th IEEE Real-Time Systems Symposium*. pp. 28–37.
15. Andersson, B., S. Baruah, and J. Jonsson: 2001, 'Static-priority scheduling on multiprocessors'. In: *Proc. 22nd IEEE Real-Time Systems Symposium*. London, UK, pp. 193–202.
16. Årzén, K. E., A. Cervin, J. Eker, and L. Sha: 2000, 'An introduction to control and real-time sceduling co-design'. In: *Conference on Decision and Control*. Sydney, Australia.
17. Atlas, A. K. and A. 'Bestavros: 1998, 'Statistical rate monotonic scheduling'. In: *Proc. 19th IEEE Real-Time Systems Symposium*. pp. 123–132.
18. Audsley, N. C.: 1994, 'Flexible scheduling for hard real-time systems'. D. Phil Thesis, Department of Computer Science, University of York.
19. Audsley, N. C., A. Burns, R. Davis, K. Tindell, and A. J. Wellings: 1995, 'Fixed priority preemptive scheduling: an historical perspective'. *Real Time Systems* **8**(3), 173–198.
20. Audsley, N. C., A. Burns, M. Richardson, and A. J. Wellings: 1991, 'Hard real-time scheduling: the deadline monotonic approach'. In: *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*. Atlanta, GA, USA, pp. 127–132.

21. Audsley, N. C., A. Burns, M. F. Richardson, and A. J. Wellings: 1992, 'Deadline monotonic scheduling theory'. In: *Proc. IFAC/IFIP Workshop on Real-Time Programming*. Bruges, Belgium, pp. 55–60.
22. Audsley, N. C., A. Burns, M. F. Richardson, and A. J. Wellings: 1993a, 'Incorporating unbounded algorithms into predictable real-time systems'. *Computer Systems Science and Engineering* **8**(2), 80–89.
23. Audsley, N. C., A. Burns, and A. J. Wellings: 1993b, 'Deadline monotonic scheduling theory and application'. *Control Engineering Practice* **1**(1), 71–78.
24. Audsley, N. C., R. I. Davis, and A. Burns: 1994, 'Mechanisms for enhancing the flexibility and utility of hard real-time systems'. In: *Proc. 15th IEEE Real-Time Systems Symposium*. pp. 12–21.
25. Audsley, N. C., K. Tindell, and A. Burns: 1993c, 'The end of the road for static cyclic scheduling'. In: *Proc. 5th Euromicro Workshop on Real-Time Systems*. Oulu, Finland, pp. 36–41.
26. Aydin, H., R. Melhem, D. Moss, and P. M. Alvarez: 2001, 'Optimal reward-based scheduling for periodic real-time tasks'. *IEEE Trans. on Computers* **50**(2), 111–130.
27. Baker, T. P.: 1991, 'Stack-based scheduling of real-time processes'. *Real-Time Systems* **3**(1), 67–100.
28. Baker, T. P.: 2003, 'Multiprocessor EDF and deadline monotonic schedulability analysis'. In: *Proc. 24th IEEE Real-Time Systems Symposium*. pp. 120–129.
29. Baker, T. P. and O. Pazy: 1991, 'Real-time features for Ada 9X'. In: *Proc. 12th IEEE Real-Time Systems Symposium*. pp. 172–180.
30. Balbastre, P., I. Ripoll, and A. Crespo: 2000, 'Control task delay reduction under static and dynamic scheduling policies'. In: *Proc. 7th International Conference on Real-Time Computing Systems and Applications*.
31. Ball, J. A., M. V. Day, and P. Kachroo: 1999, 'Robust feedback control of a single server queueing system'. *Mathematics of Control Signals and Systems* **12**(4), 307–345.
32. Baruah, S.: 2003, 'Dynamic- and static-priority scheduling of recurring real-time tasks'. *Real-Time Systems* **24**(1), 99–128.
33. Baruah, S. and J. Goossens: 2003, 'Rate-monotonic scheduling on uniform multiprocessors'. *IEEE Trans. on Computers* **52**(7), 966–970.
34. Baruah, S. K., N. Cohen, C. G. Plaxton, and D. Varvel: 1993, 'Proportionate progress: a Notion of Fairness in Resource Allocation'. In: *Proc. ACM Symposium on the Theory of Computing*. pp. 345–354.
35. Baruah, S. K., R. R. Howell, and L. E. Rosier: 1990, 'Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor'. *Real-Time Systems* **2**.
36. Bate, I. J. and A. Burns: 1997, 'Schedulability analysis of fixed priority real-time systems with offsets'. In: *Proc. 9th Euromicro Workshop on Real-Time Systems*. pp. 153–160.
37. Bate, I. J. and A. Burns: 1998, 'Investigating the pessimism in distributed systems timing analysis'. In: *Proc. 10th Euromicro Workshop on Real-Time Systems*. pp. 107–114.
38. Bernat, G. and A. Burns: 1997, 'Combining (n, m)-hard deadlines with dual priority scheduling'. In: *Proc. 18th IEEE Real-Time Systems Symposium*. pp. 46–57.
39. Bernat, G. and A. Burns: 1999, 'New results on fixed priority aperiodic servers'. In: *Proc. 20th IEEE Real-Time Systems Symposium*. pp. 68–78.

40. Bernat, G. and R. Cayssials: 2001, 'Guaranteed on-line weakly-hard real-time systems'. In: *Proc. 22nd IEEE Real-Time Systems Symposium.* pp. 25–35.
41. Bini, E. and G. C. Buttazzo: 2002, 'The space of rate montonic schedulability'. In: *Proc. 23rd IEEE Real-Time Systems Symposium.* pp. 169–178.
42. Bini, E., G. C. Buttazzo, and M. Giuseppe: 2003, 'Rate monotonic scheduling: the hyperbolic bound'. *IEEE Trans. on Computers* **52**(7), 933–942.
43. Broster, I., A. Burns, and G. Rodríguez-Navas: 2002, 'Probabilistic Analysis of CAN with Faults'. In: *Proc. 23rd Real-Time Systems Symposium.*
44. Burchard, A., J. Liebeherr, Y. Og, and S. H. Son: 1995, 'New strategies for assigning real-time tasks to multiprocessor systems'. *IEEE Trans. on Computers* **44**(12), 1429–1442.
45. Burns, A. and A. J. Wellings: 1990, *Real-Time Systems and Programming Languages, 1st Edition.* Addison-Wesley.
46. Burns, A. and A. J. Wellings: 1993, 'Dual priority assignment: a practical method of increasing processor utilisation'. In: *Proc. Fifth Euromicro Workshop on Real-Time Systems.* Oulu, Finland, pp. 48–53.
47. Buttazzo, G.: 2003, 'Rate Monotonic vs. EDF: Judgment Day'. In: *Proc. 3rd International Conference on Embedded Software.* Philadelphia, PA, pp. 67–83.
48. Buttazzo, G. and L. Abeni: 2002a, 'Adaptive workload management through elastic scheduling'. *Real-Time Systems* **23**(1-2), 7–24.
49. Buttazzo, G. and L. Abeni: 2002b, 'Smooth rate adaptation through impedance control'. In: *IEEE Proc. 14th Euromicro Conference on Real-Time Systems.* Vienna, Austria.
50. Buttazzo, G., G. Lipari, M. Caccamo, and L. Abeni: 2002, 'Elastic scheduling for flexible workload management'. *IEEE Trans. on Computers* **51**(3), 289–302.
51. Buttazzo, G. and F. Sensini: 1999, 'Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments'. *IEEE Trans. on Computers* **48**(10), 1035–1052.
52. Caccamo, M. and G. Buttazzo: 1997, 'Exploiting skips in periodic tasks for enhancing aperiodic responsiveness'. In: *Proc. IEEE 18th Real-Time Systems Symposium.* San Francisco, pp. 330–339.
53. Caccamo, M., G. Buttazzo, and L. Sha: 2000, 'Capacity sharing for overrun control'. In: *Proc. 21st IEEE Real-Time Systems Symposium.* Orlando, FL, USA.
54. Caccamo, M. and L. Sha: 2001, 'Aperiodic servers with resource constraints'. In: *Proc. 22nd IEEE Real-Time Systems Symposium.* London, UK.
55. Cervin, A.: 1999, 'Improved Scheduling of Control Tasks'. In: *Proc. 11th Euromicro Conference on Real-Time Systems.* York, UK, pp. 4–10.
56. Cervin, A. and J. Eker: 2003, 'The Control Server: a Computational Model for Real-Time Control Tasks'. In: *Proc. 15th Euromicro Conference on Real-Time Systems.* Porto, Portugal, pp. 113–120.
57. Cervin, A., J. Eker, B. Bernhardsson, and K. E. Årzén: 2002, 'Feedback-Feedforward Scheduling of Control Tasks'. *Real-Time Systems* **23**(1).
58. Chandra, A., M. Adler, and P. Shenoy: 2001, 'Deadline Fair Scheduling: Bridging the Theory and Practice of Proportionate-Fair Scheduling in Multiprocessor Servers'. In: *Proc. IEEE Real-Time Technology and Applications Symposium.* pp. 3–14.
59. Chandra, R., X. Liu, and L. Sha: 2003, 'On the scheduling of flexible and reliable real-time control systems'. *Real-Time Systems* **24**(2), 153–169.
60. Chandra, R. and L. Sha: 1999, 'On scheduling tasks in reliable real-time control systems'. In: *Proc. 20th IEEE Real-Time Systems Symposium.* pp. 164–175.

61. Chen, D. and A. K. Mok: 2004, ‘The pinwheel: a real-time scheduling problem’. *Handbook of Scheduling Algorithms, Models and Performance Analysis*.
62. Chen, M. and K. Lin: 1990, ‘Dynamic priority ceilings: a concurrency control protocol for real-time systems’. *Real-Time Systems* **2**.
63. Christin, N., J. Liebeherr, and T. Abdelzaher: 2002, ‘A quantitative assured forwarding service’. In: *Proc. IEEE Infocom*.
64. Committee, I. P. A. S.: 2003, *ISO/IEC 9945-2:2003(E), Information technology - Portable Operating System Interface (Committee:2003:III) - Part 2: System Interfaces*. IEEE Standards Association.
65. Davis, R. I., K. Tindell, and A. Burns: 1993, ‘Scheduling slack time in fixed priority preemptive systems’. In: *Proc. 14th IEEE Real-Time Systems Symposium*. pp. 222–231.
66. Davis, R. I. and A. J. Wellings: 1995, ‘Dual priority scheduling’. In: *Proc. 16th IEEE Real-Time Systems Symposium*. pp. 100–109.
67. Deng, Z. and J. Liu: 1997, ‘Scheduling real-time applications in an open environment’. In: *Proc. 18th IEEE Real-Time Systems Symposium*. pp. 308–319.
68. Dertouzos, M. L.: 1974, ‘Control robotics: the procedural control of physical processes’. *Information Processing* **74**.
69. Dertouzos, M. L. and A. K. Mok: 1989, ‘Multiprocessor on-line scheduling of hard real-time tasks’. *IEEE Trans. on Software Engineering* **15**(12), 1497–1506.
70. Dhall, S. K. and C. L. Liu: 1978, ‘On a real-time scheduling problem’. *Operations Research* **26**(1), 127–140.
71. Diao, Y., N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury: 2002, ‘MIMO control of an Apache Web server: modeling and controller design’. In: *American Control Conference*. Anchorage, Alaska.
72. Diao, Y., J. L. Hellerstein, S. Parekh, and J. P. Bigus: 2003, ‘Managing Web server performance with autotune agents’. *IBM Systems Journal* **42**(1), 136–149.
73. Doyle, L. and J. Elzey: 1994, ‘Successful use of rate monotonic theory on a formidable real time system’. In: *Proc. 11th IEEE Workshop on Real-Time Operating Systems and Software*.
74. Doytchinov:2001:RTQ, B., J. Lehoczky, and S. Shreve: 2001, ‘Real-time queues in heavy traffic with earliest-deadline-first queue discipline,’. *Annals of Applied Probability* **11**, 332–378.
75. Eker, J., P. Hagander, and K. E. Årzén: 2000, ‘A Feedback Scheduler for Real-Time Control Tasks’. *Control Engineering Practice* **8**(12), 1369–1378.
76. Fineberg, M. S. and O. Serlin: 1967, ‘Multiprogramming for hybrid computation’. *Proc. AFIPS Fall Joint Computing Conference* pp. 1–13.
77. Franklin, G. F., J. D. Powell, and A. Emami-Naeini: 1994, *Feedback Control of Dynamic Systems*. Addison-Wesley.
78. Funk, S., J. Goossens, and S. Baruah: 2001, ‘On-line scheduling on uniform multiprocessors’. In: *Proc. 22nd IEEE Real-Time Systems Symposium*. London, UK, pp. 183–192.
79. Gandhi, N., S. Parekh, J. Hellerstein, and D. M. Tilbury: 2001, ‘Feedback control of a Lotus Notes server: modeling and control design’. In: *American Control Conference*.
80. Gardener:1999:PAS, M. K.: 1999, ‘Probabilistic analysis and scheduling of critical soft real-time systems’. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.

81. Garey, M. R. and D. S. Johnson: 1975, 'Complexity Results for Multiprocessor Scheduling under Resource Constraint'. *SIAM J. Comput.* **4**, 397–411.
82. Garey, M. R. and D. S. Johnson: 1979, *Computers and Intractability: a Guide to the Theory of NP-Completeness*. New York: W. H. Freeman.
83. Ghazalie, T. M. and T. P. Baker: 1995, 'Aperiodic servers in a deadline scheduling environment'. *Real-Time Systems* **9**.
84. Goddard, S. and K. Jeffay: 2000, 'The synthesis of real-time systems from processing graphs'. In: *Proc. 5th IEEE International Symposium on High Assurance Systems Engineering*. pp. 177–186.
85. Goossens, J., S. Funk, and S. Baruah: 2003, 'Priority-driven scheduling of periodic task systems on multiprocessors'. *Real-Time Systems* **25**(2/3), 187–205.
86. Gutierrez, J. P., , and M. G. Harbour: 1998, 'Best-case analysis for improving the worst-case schedulability test for distributed hard real-time systems'. In: *10th Euromicro Workshop on Real-Time Systems*. pp. 35–44.
87. Gutierrez, J. P., J. G. Garcia, and M. G. Harbour: 1995, 'Optimized priority assignment for tasks and messages in distributed real-time systems'. *IEEE Parallel and Distributed Systems* pp. 124–131.
88. Gutierrez, J. P., J. G. Garcia, and M. G. Harbour: 1997, 'On the Schedulability analysis for distributed real-time systems'. In: *Proc. 9th Euromicro Workshop on Real-Time Systems*. pp. 136–143.
89. Ha, R. and J. W. S. Liu: 1993, 'Validating timing constraints in multiprocessor and distributed real-time systems'. In: *Proc. 14th IEEE International Conference on Distributed Computing Systems*. Poznan, Poland, pp. 162–171.
90. Han, C. and K. J. Lin: 1992, 'Scheduling distance-constrained real-time tasks'. In: *Proc. 13th IEEE Real-Time Systems Symposium*.
91. Han, C. and H. Tyan: 1997, 'A better polynomial-time scheduling test for real-time fixed-priority scheduling algorithms'. In: *Proc. 18th IEEE Real-Time Systems Symposium*. pp. 36–45.
92. Harbour, M. G., M. H. Klein, and J. P. Lehoczky: 1991, 'Fixed priority scheduling of periodic tasks with varying execution priority'. In: *Proc. 12th IEEE Real-Time Systems Symposium*.
93. Harter, P. K.: 1984, 'Response times in level structured systems'. Technical Report CU-CS-269-94, Department of Computer Science, University of Colorado, USA.
94. Harter, P. K.: 1987, 'Response times in level structured systems'. *ACM Trans. on Computer Systems* **5**(3), 232–248.
95. Hellerstein, J. L., Y. Diao, S. Parekh, and D. M. Tilbury: 2004, *Feedback Control of Computing Systems*. Wiley.
96. Henriksson, D., Y. Lu, and T. Abdelzaher: 2004, 'Improved prediction for Web server delay control'. In: *Proc. Euromicro Conference on Real-Time Systems*. Catania, Sicily, Italy.
97. Henzinger, T. A., B. Horowitz, and C. M. Kirsch: 2001, 'Giotto: a time-triggered language for embedded programming'. In: *Proc. First International Workshop on Embedded Software*.
98. Hollot, C. V., V. Misra, D. F. Towsley, and W. Gong: 2001a, 'A control theoretic analysis of RED'. In: *Proc. IEEE Infocom*. Anchorage, Alaska, pp. 1510–1519.
99. Hollot, C. V., V. Misra, D. F. Towsley, and W. Gong: 2001b, 'On designing improved controllers for AQM routers supporting TCP flows'. In: *Proc. IEEE Infocom*. pp. 1726–1734.

100. Holman, P. and J. Anderson: 2002, 'Locking in Pfair-scheduled Multiprocessor Systems'. In: *Proc. 23rd IEEE Real-Time Systems Symposium*. pp. 149–158.
101. Holman, P. and J. Anderson: 2004, 'Implementing Pfairness on a Symmetric Multiprocessor'. In: *Proc. 10th IEEE Real-Time and Embedded Technology and Applications Symposium*. pp. 544–553.
102. Hou, C.-J. and K. G. Shin: 1997, 'Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems'. *IEEE Trans. on Computers* **46**(12), 1338–1356.
103. Jackson, L. and G. Rouskas: 2003, 'Optimal quantization of periodic task requests on multiple identical processors'. *IEEE Transactions on Parallel and Distributed Systems* **14**(8), 795–806.
104. Jeffay, K.: 1992, 'Scheduling sporadic tasks with shared resources in hard-real-time systems'. In: *Proc. 13th IEEE Real-Time Systems Symposium*. Phoenix, AZ, USA.
105. Jeffay, K. and S. Goddard: 1999, 'A Theory of Rate-Based Execution'. In: *Proc. 20th IEEE Real-Time Systems Symposium*. pp. 304–314.
106. Jeffay, K. and S. Goddard: 2001, 'Rate-Based Resource Allocation Methods for Embedded Systems'. In: *Proc. EMSOFT 2001: The First International Workshop on Embedded Software*. pp. 204–222.
107. Jeffay, K. and G. Lamastra: 2000, 'A Comparative Study of the Realization of Rate-Based Computing Services in General Purpose Operating Systems'. In: *Proc. International Conference on Real-Time Computing Systems and Applications*. Cheju Island, South Korea, pp. 81–90.
108. Jensen, E. D., C. D. Locke, and H. Tokuda: 1985, 'A time driven scheduling model for real-time operating systems'. In: *Proc. 6th IEEE Real-Time Sytems Symposium*. pp. 112–122.
109. Joseph, M. and P. Pandya: 1986, 'Finding response times in a real-time system'. *BCS Computer Journal* **29**(5), 390–395.
110. Keshav, S.: 1993, 'A control-theoretic approach to flow control'. In: *Proc. Conference on Communications Architecture & Protocols*. pp. 3–15.
111. Klein, M., T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour: 1993, *A Practioner's Handbook for Real Time Analysis: a Guide to Rate Monotonic Analysis for Real Time Systems*. Boston-Dordrecht-London: Kluwer.
112. Koren, G. and D. Shasha: 1995, 'Skip-over: algorithms and complexity for overloaded systems that allow skips'. In: *Proc. IEEE Real Time System Symposium*. Pisa, pp. 110–117.
113. Kruk, L., J. Lehoczky, S. Shreve, and S.-N. Yeung: 2004, 'Earliest-deadline-first service in heavy-traffic acyclic networks'. *Annals of Applied Probability* **14**(3), 1306–1352.
114. Lampson, B. W. and D. Redell: 1980, 'Experience with Processes and Monitors in Mesa'. *CACM* **23**(2), 105–117.
115. Lauzac, S., R. Melkem, and D. Mosse: 1998, 'Comparison of Global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor'. In: *10th Euromicro Workshop on Real-Time Systems*. pp. 188–195.
116. Lawrence, D. A., G. Jianwei, S. Mehta, and L. R. Welch: 2001, 'Adaptive scheduling via feedback control for dynamic real-time systems'. In: *IEEE International Performance, Computing, and Communications Conference*. Pheonix, AZ, USA, pp. 373–378.
117. Lehoczky, J. P.: 1990, 'Fixed priority scheduling of periodic task sets with arbitrary deadlines'. In: *Proc. 11th IEEE Real-Time Systems Symposium*. pp. 201–209.

118. Lehoczky, J. P.: 1996, ‘Real-time queueing theory’. In: *Proc. IEEE Real-Time Systems Symposium.* pp. 186–195.
119. Lehoczky, J. P.: 1997a, ‘Real-time queueing network theory’. In: *Proc. IEEE Real-Time Systems Symposium.* pp. 58–67.
120. Lehoczky, J. P.: 1997b, ‘Using real-time queueing theory to control lateness in real-time systems’. *Performance Evaluation Review* pp. 158–168.
121. Lehoczky, J. P.: 1998, ‘Scheduling communication networks carrying real-time traffic’. In: *Proc. IEEE Symposium on Real-Time Systems.* pp. 470–479.
122. Lehoczky, J. P. and L. Sha: 1986, ‘Performance of real-time bus scheduling algorithms’. In: *Proc. 1986 ACM SIGMETRICS Joint International Conference on Computer Performance Modelling, Measurement and Evaluation.* pp. 44–53.
123. Lehoczky, J. P., L. Sha, and D. Y. Ding: 1989, ‘The rate monotonic scheduling algorithm: exact characterization and average case behavior’. In: *Proc. 10th IEEE Real-Time Systems Symposium.* pp. 166–171.
124. Lehoczky, J. P., L. Sha, and J. K. Stronsnider:1995:DSA: 1987, ‘Enhanced aperiodic responsiveness in a hard real-time environment’. In: *Proc. 8th IEEE Real-Time Systems Symposium.* pp. 261–270.
125. Lehoczky, J. P. and S. R. Thuel: 1992, ‘An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems’. In: *Proc. 13th IEEE Real-Time Systems Symposium.* Phoenix, AZ, USA, pp. 110–123.
126. Lesser, V. R., J. Pavlin, and E. Durfee: 1988, ‘Approximate processing in real-time problem solving’. In: *AI Magazine*, Vol. 9. pp. 49–61.
127. Leung, J. Y.-T. and M. L. Merrill: 1980, ‘A note on preemptive scheduling of periodic, real-time tasks’. *Information Processing Letters* **11**(3), 115–118.
128. Leung, J. Y. T. and J. Whitehead: 1982, ‘On the complexity of fixed-priority scheduling of periodic, real-time tasks’. *Performance Evaluation (Netherlands)* **2**(4), 237–250.
129. Levy, R., J. Nagarajarao, G. Pacifici, A. Spreitzer, A. Tantawi, and A. Youssef: 2003, ‘Performance management for cluster based Web services’. In: *IFIP/IEEE 8th International Symposium on Integrated Network Management.* pp. 247–261.
130. Li, C., R. Bettati, and W. Zhao: 1997, ‘Static priority scheduling for ATM networks’. In: *Proc. 18th IEEE Real-Time Systems Symposium.* San Francisco, CA, USA, pp. 264–273.
131. Lima, G. and A. Burns: 2003, ‘An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems’. *IEEE Trans. on Computer Systems.*
132. Lin, K. J., S. Natarajan, and J. W. S. Liu: 1987, ‘Concord: a system of imprecise computation’. In: *Proc. 1987 IEEE Compsac.*
133. Lincoln, B.: 2002, ‘Jitter Compensation in Digital Control Systems’. In: *Proc. 2002 American Control Conference.*
134. Lincoln, B. and A. Cervin: 2002, ‘Jitterbug: a tool for analysis of real-time control performance’. In: *Proc. 41st IEEE Conference on Decision and Control.* Las Vegas, NV.
135. Lipari, G. and S. K. Baruah: 2000, ‘Greedy reclaimation of unused bandwidth in constant bandwidth servers’. In: *Proc. 12th Euromicro Conference on Real-Time Systems.* Stokholm, Sweden.
136. Liu, C. L. and J. W. Layland: 1973, ‘Scheduling alghorithms for multiprogramming in a hard real-time environment’. *Journal of the ACM* **20**(1), 46–61.

137. Liu, J. W. S.: 2000, *Real-Time Systems*. Prentice-Hall.

138. Liu, J. W. S., K. J. Lin, and S. Natarajan: 1987, 'Scheduling real-time, periodic jobs using imprecise results'. In: *Proc. IEEE Real-Time System Symposium*. San Jose, CA, USA.

139. Liu, J. W. S., K. J. Lin, W. K. Shih, A. C. S. Yu, J. Y. Chung, and W. Zhao: 1991, 'Algorithms for scheduling imprecise computations'. *IEEE Computer* **24**(5), 58–68.

140. Liu, J. W. S., W. K. Shih, K. J. Lin, R. Bettati, and J. Y. Chung: 1994, 'Imprecise computations'. *Proc. IEEE* **82**(1), 83–94.

141. Locke, C. D.: 1986, 'Best-effort decision making for real-time scheduling'. CMU-CS-86-134 (PhD Thesis), Computer Science Department, CMU.

142. Lopez, J. M., J. L. Diaz, and D. F. Garcia: 2001, 'Minimum and maximum utilization bounds for multiprocessor RM scheduling'. In: *Proc. Euromicro Conference on Real-Time Systems*. Delft, Netherlands, pp. 67–75.

143. Lu, C., T. Abdelzaher, J. A. Stankovic, and S. Son: 2001a, 'A feedback control approach for guaranteeing relative delays in Web servers'. In: *IEEE Real-Time Technology and Applications Symposium*. TaiPei, Taiwan.

144. Lu, C., G. A. Alvarez, and J. Wilkes: 2002a, 'Aqueduct: online data migration with performance guarantees'. In: *USENIX Conference on File and Storage Technologies*. Monterey, CA.

145. Lu, C., J. A. Stankovic, T. Abdelzaher, and G. Tao: 2000, 'Performance specifications and metrics for adaptive real-time systems'. In: *Proc. 19th IEEE Real-Time Systems Symposium*. Orlando, FL.

146. Lu, C., J. A. Stankovic, G. Tao, and S. Son: 1999a, 'Design and evaluation of a feedback control EDF scheduling algorithm'. In: *Proc. 20th IEEE Real-Time Systems Symposium*. Phoenix, AZ, USA.

147. Lu, C., J. A. Stankovic, G. Tao, and S. Son: 1999b, 'Design and evaluation of a feedback control EDF scheduling algorithm'. In: *Proc. 20th IEEE Real-Time Systems Symposium*. Phoenix, AZ, USA.

148. Lu, C., J. A. Stankovic, G. Tao, and S. Son: 2002b, 'Feedback control real-time scheduling: framework, modeling, and algorithms'. *Real-Time Systems* **23**(1/2).

149. Lu, Y., T. Abdelzaher, C. Lu, L. Sha, and X. Liu: 2003, 'Feedback control with queueing-theoretic prediction for relative delay'. In: *IEEE Real-Time and Embedded Technology and Applications Symposium*.

150. Lu, Y., T. Abdelzaher, C. Lu, and G. Tao: 2002c, 'An adaptive control framework and its application to differentiated caching services'. In: *Proc. 10th International Workshop on Quality of Service*. Miami Beach, FL.

151. Lu, Y., A. Saxena, and T. Abdelzaher: 2001b, 'Differentiated caching services: a control-theoretical approach'. In: *Proc. 2001 International Conference on Distributed Computing Systems*. pp. 615–622.

152. Lundberg, L.: 2002, 'Analyzing Fixed-Priority Global Multiprocessor Scheduling'. In: *Proc. 8th IEEE Real-Time and Embedded Technology and Applications Symposium*. San Jose, CA, USA, pp. 145–153.

153. Markatos, E. P.: 1993, *Multiprocessor Synchronization Primitives with Priorities*, pp. 111–120. IEEE Computer Society. Yann Hang Lee and C. M. Krishna, Editor.

154. Marti, P., G. Fohler, K. Ramamritham, and J. M. Fuertes: 2001, 'Jitter Compensation for Real-Time Control Systems'. In: *Proc. 22nd IEEE Real-Time Systems Symposium*.

155. Mercer, C. W., S. Savage, and H. Tokuda: 1993, 'Processor Capacity Reserves: An Abstraction for Managing Processor Usage'. In: *WWOS*. 129-134.
156. Mercer, C. W., S. Savage, and H. Tokuda: 1994, 'Temporal Protection in Real-Time Operating Systems'. In: *Proc. 11th IEEE workshop on Real-Time Operating System and Software*. pp. 79–83.
157. Moir, M. and S. Ramamurthy: 1999, 'Pfair Scheduling of Fixed and Migrating Periodic Tasks on Multiple Resour ces'. In: *Proc. 20th IEEE Real-Time Systems Symposium*. pp. 294–303.
158. Mok, A. K.: 1983, 'Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment'. Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA.
159. Mok, A. K., P. Amerasinghe, M. Chen, S. Sutanthavibul, and K. Tantisirivat: 1987, 'Synthesis of a Real-Time Message Processing System with Data-driven Timing Constraints'. In: *Proc. 18th IEEE Real-Time Systems Symposium*. pp. 133–143.
160. Mok, A. K. and D. Chen: 1997, 'A Multiframe Model for Real-Time Tasks'. *IEEE Trans. on Software Engineering* **23**(10), 635–645.
161. Mok, A. K. and X. Feng: 2002, 'Real-Time Virtual Resource: a Timely Abstraction for Embedded Systems'. In: *The Second International Conference on Embedded Software, Lecture Notes in Computer Science, LNCS 2491*. pp. 182–196.
162. Mok, A. K., D. C. Tsou, and R. C. M. de Rooij: 1996, 'The MSP.RTL Real-Time Scheduler Synthesis Tool'. In: *Proc. 17th IEEE Real-Time Systems Symposium*. pp. 118–128.
163. Mueller, F.: 1999, 'Priority Inheritance and ceilings for Distributed Mutual Exclusion'. In: *Proc. 20th IEEE Real-Time Systems Symposium*. pp. 340–349.
164. Nahrstedt, K.: 1995, 'End-to-end QoS guarantees in networked multimedia system'. *ACM Computing Surveys* **27**(4).
165. Nassor, E. and G. Bres: 1991, 'Hard Real-Time Sporadic Task Scheduling for Fixed Priority Schedulers'. In: *Proc. International Workshop on Responsive Systems, Golfe-Juan, France*. pp. 44–47.
166. Natarajan, S. (ed.): 1995, *Imprecise and Approximate Computation*. Boston-Dordrecht-London: Kluwer.
167. Ng, J., S. Song, and W. Zhao: 1997, 'Integrated Delay Analysis of Regulated ATM Switch'. In: *Proc. 18th IEEE Real-Time Systems Symposium*. San Francisco, CA, USA, pp. 285–296.
168. Nilsson, J., B. Bernhardsson, and B. Wittenmark: 1998, 'Stochastic Analysis and Control of Real-Time Systems with Random Time Delays'. *Automatica* **34**(1), 57–64.
169. Oh, D. I. and T. P. Baker: 1998, 'Utilization Bounds for N-Processor Rate Monotone Scheduling with Stable Processor Assignment'. *Real Time Systems* **15**(2), 183–193.
170. Oh, Y. and S. H. Son: 1995, 'Allocating fixed-priority periodic tasks on Multiprocessors'. *Real-Time Systems* **9**, 207–239.
171. Palencia, J. C. and M. G. Harbour: 1998, 'Schedulability analysis for tasks with static and dynamic offsets'. In: *Proc. 19th IEEE Real-Time Systems Symposium*. pp. 26–37.
172. Phillips:1997:OTC, C. A., C. Stein, E. Torng, and J. Wein: 1997, 'Optimal time-critical scheduling via resource augmentation'. In: *Proc. 29th Annual ACM Symposium on Theory of Computing*. El Paso, Texas, pp. 140–149.

173. Pradhan, P., R. Tewari, S. Sahu, A. Chandra, and P. Shenoy: 2002, 'An Observation-based Approach Towards Self-Managing Web Servers'. In: *the Tenth International Workshop on Quality of Service*.

174. Quan, G. and X. Hu: 2000, 'Enhanced fixed-priority scheduling with (m,k)-Firm guarantees'. In: *Proc. 21st IEEE Real-Time Systems Symposium*. pp. 79–88.

175. Rajkumar, R., C. Lee, J. P. Lehoczky, and D. P. Siewiorek: 1998, 'Practical solutions for QoS-based Resource Allocation Problems'. In: *Proc. IEEE Symposium on Real-Time Systems*. pp. 296–306.

176. Rajkumar, R., L. Sha, and J. P. Lehoczky: 1988, 'Real-Time Synchronization Protocols for Multiprocessors'. In: *Proc. 9th IEEE Real-Time Systems Symposium*. pp. 259–269.

177. Rajkumar, R., L. Sha, and J. P. Lehoczky: 1990, 'Real-Time Synchronization Protocols for Shared Memory Multiprocessors'. In: *Proc. 10th International Conference on Distributed Computing*. pp. 116–125.

178. Rajkumar, R., L. Sha, J. P. Lehoczky, and K. Ramamritham: 1994, 'An Optimal Priority Inheritance Policy for Synchronization in Real-Time Systems'. In: S. H. Son (ed.): *Advances in Real-Time Systems*. Prentice-Hall, pp. 249–271.

179. Ramamritham, K.: 1990, 'Allocation and Scheduling of Complex Periodic Tasks'. In: *Proc. 10th IEEE International Conference on Distributed Computing Systems*. pp. 108–115.

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

181. Ramamritham, K., J. A. Stankovic, and P. Shiah: 1990, 'Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems'. *IEEE Trans. on Parallel and Distributed Systems*.

182. Ramamritham, K., J. A. Stankovic, and W. Zhao: 1989, 'Dynamic Task Scheduling in Distributed Hard Real-Time Systems'. *IEEE Trans. on Computers* **38**(8), 1110–23.

183. Ramos-Thuel, S. and J. P. Lehoczky: 1993, 'On-Line Scheduling of Hard deadline Aperiodic Tasks in Fixed-Priority Systems'. In: *Proc. 14th IEEE Real-Time Systems Symposium*. pp. 160–171.

184. Ramos-Thuel, S. and J. P. Lehoczky: 1994, 'Algorithms for scheduling hard aperiodic tasks in fixed priority systems using slack stealing'. In: *Proc. 15th IEEE Real-Time Systems Symposium*. San Juan, Puerto Rico, pp. 22–35.

185. Ramos-Thuel, S. and J. K. Stosnider: 1991, 'The transient server apporoach to scheduling time-critical recovery operations'. In: *Proc. 12th IEEE Real-Time Systems Symposium*. pp. 286–295.

186. Sahoo, D. R., S. Swaminathan, R. Al-Omari, M. V. Salapaka, G. Manimaran, and A. K. Somani: 2002, 'Feedback control for real-time scheduling'. In: *American Control Conference*. pp. 1254–1259.

187. Seto, D., J. P. Lehoczky, L. Sha, and K. G. Shin: 1996, 'On task schedulability in real-time control systems'. In: *Proc. 17th IEEE Real-Time Systems Symposium*. Washington, DC.

188. Seto, D., J. P. Lehoczky, L. Sha, and K. G. Shin: 2001, 'Trade-Off Analysis of Real-Time Control Performance and Schedulability'. *Real-Time Systems* **21**(3), 199–217.

189. Sha, L.: 2001, 'Using Simplicity to Control Complexity'. *IEEE Software* **18**(4).

190. Sha, L. and J. Goodenough: 1990, 'Real Scheduling Theory and Ada'. *IEEE Computer* **23**(4), 53–62.

191. Sha, L., J. P. Lehoczky, and R. Rajkumar: 1986, 'Solutions For Some Practical Problems in Prioritizing Preemptive Scheduling'. In: *Proc. 7th IEEE Real-Time Sytems Symposium*.
192. Sha, L., J. P. Lehoczky, and R. Rajkumar: 1987, 'Task scheduling in distributed real-time systems'. In: *Proc. IEEE Industrial Electronics Conference*.
193. Sha, L., X. Liu, Y. Lu, and T. Abdelzaher: 2002, 'Queuing Model Based Network Server Performance Control'. In: *IEEE Real-Time Systems Symposium*.
194. Sha, L. and R. Rajkumar: 1990, 'Real-Time scheduling support in Futurebus+'. In: *Proc. 11th IEEE Real-Time Systems Symposium*. pp. 331–340.
195. Sha, L., R. Rajkumar, and J. Lehoczky: 1991a, 'Real Time Computing with Futurebus+'. *IEEE Micro* **11**(3), 95–100.
196. Sha, L., R. Rajkumar, and J. P. Lehoczky: 1990, 'Priority inheritance protocols: an approach to real-time synchronisation'. *IEEE Trans. on Computers* **39**(9), 1175–1185.
197. Sha, L., R. Rajkumar, S. Son, and C.-H. Chang: 1991b, 'A Real-Time Locking Protocol'. *IEEE Trans. on Computers* **40**(7), 793–800.
198. Sharma, V., A. Thomas, T. Abdelzaher, and K. Skadron: 2003, 'Power-aware QoS Management in Web Servers'. In: *Real-Time Systems Symposium*. Cancun, Mexico.
199. Shih, W., W. S. Liu, and J. Chung: 1991, 'Algorithms for Scheduling Imprecise Computations with Timing Constraints'. *SIAM Journal of Computing* **20**(3), 537–552.
200. Shih, W., W. S. Liu, J. Chung, and D. W. Gillies: 1989, 'Scheduling Tasks with Ready Times and Deadlines to Minimize Average Error'. *Operating Systems Review* **23**(3).
201. Shin, I. and I. Lee: 2003, 'Periodic Resource Model for Compositional Real-Time Guarantees'. *IEEE Real-Time Systems Symposium*.
202. Simpson, H.: 1990, 'Four-Slot Fully Asynchronous Communication Mechanism'. *IEE Proc.* **137**(1), 17–30.
203. Sjodin, M. and H. Hansson: 1998, 'Improving Response-time analysis calculations'. In: *Proc. 19th IEEE Real-Time Systems Symposium*. pp. 399–408.
204. Skadron, K., T. Abdelzaher, and M. R. Stan: 2001, 'Control-Theoretic Techniques and Thermal-RC Modeling for Accurate and Localized Dynamic Thermal Management'. In: *International Symposium on High Performance Computer Architecture*.
205. Sprunt:1989:ATSa, B., J. Lehoczky, and L. Sha: 1988, 'Exploiting Unused Periodic Time For Aperiodic Service Using the Extended Priority Exchange Algorithm'. In: *Proc. 9th IEEE Real-Time Systems Symposium*. pp. 251–258.
206. Sprunt:1989:ATSa, B., L. Sha, and L. Lehoczky: 1989, 'Aperiodic task scheduling for hard real-time systems'. *Real-Time Systems* **1**(1), 27–60.
207. Spuri, M. and G. Buttazzo: 1994, 'Efficient aperiodic service under the earliest deadline scheduling'. In: *Proc. IEEE Real-Time Systems Symposium*.
208. Spuri, M. and G. Buttazzo: 1996, 'Scheduling aperiodic tasks in dynamic priority systems'. *Real-Time Systems* **10**(2), 179–210.
209. Srinivasan, A. and J. Anderson: 2002, 'Optimal Rate-based Scheduling on Multiprocessors'. In: *Proc. 34th ACM Symposium on Theory of Computing*. pp. 189–198.
210. Srinivasan, A. and J. Anderson: 2003, 'Efficient Scheduling of Soft Real-Time Applications on Multiprocessors'. In: *Proc. 15th Euromicro Conference on Real-Time Systems*. pp. 51–59.

211. Srinivasan, A., P. Holman, and J. Anderson: 2002, 'Integrating Aperiodic and Recurrent Tasks on Fair-scheduled Multiprocessors'. In: *Proceedings of the 14th Euromicro Conference on Real-Time Systems*. pp. 19–28.
212. Stankovic, J. A., T. He, T. F. Abdelzaher, M. Marley, G. Tao, S. H. Son, and C. Lu: 2001, 'Feedback control scheduling in distributed systems'. In: *Proc 22nd IEEE Real-Time Systems Symposium*. London, UK.
213. Stankovic, J. A., C. Lu, S. Son, and G. Tao: 1999, 'The case for feedback control real-time scheduling'. In: *IEEE Proc. 11th Euromicro Conference on Real-Time Systems*. York, UK.
214. Stankovic, J. A. and K. Ramamritham: 1991, 'The Spring kernel: a new paradigm for hard real-time operating systems'. *IEEE Software* **8**(3), 62–72.
215. Stankovic, J. A., K. Ramamritham, and S. Cheng: 1985, 'Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems'. *IEEE Trans. on Computers* **34**(12), 1130–43.
216. Stankovic, J. A., K. Ramamritham, M. Spuri, and G. Buttazzo: 1998, *Deadline scheduling for real-time systems*. Boston-Dordrecht-London: Kluwer.
217. Strosnider, J., J. P. Lehoczky, and L. Sha: 1995, 'The deferrable server algorithm for enhanced aperiodic responsiveness in real-time environments'. *IEEE Trans. on Computers* **44**(1), 73–91.
218. Tia, T. S., J. S. Liu, and M. Shankar: 1996, 'Algorithms and optimality of scheduling soft aperiodic requests in fixed-priority preemptive systems'. *Real-Time Systems* **10**(1), 23–43.
219. Tindell, K., A. Burns, and A. J. Wellings: 1992, 'Allocating real-time Tasks (an NP-hard problem made easy)'. *Real-Time Systems* **4**(2), 145–165.
220. Tindell, K., A. Burns, and A. J. Wellings: 1994a, 'An extendible approach for analysing fixed priority hard real-time tasks'. *Real-Time Systems* **6**(2), 133–151.
221. Tindell, K. and J. Clark: 1994, 'Holistic schedulability analysis for distributed hard real-time systems'. *Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)* **40**, 117–134.
222. Tindell, K., H. Hansson, and A. J. Wellings: 1994b, 'Analysing real-time communications: controller area network (CAN)'. In: *Proc. 15th IEEE Real-Time Systems Symposium*. pp. 259–265.
223. Tœrngren, M.: 1998, 'Fundamentals of implementing real-time control applications in distributed computer systems'. *Real-Time Systems* **14**(3), 219–250.
224. Wang, W., A. K. Mok, and G. Fohler: 2003, 'A hybrid proactive approach for integrating off-line and on-line real-time schedulers'. In: *The Third International Conference on Embedded Software*. pp. 356–372.
225. Wei, L. and H. Yu: 2003, 'Research on a soft real-time scheduling algorithm based on hybrid adaptive control architecture'. In: *American Control Conference*. Denver, CO, pp. 4022–4027.
226. Wyle, H. and G. J. Burnett: 1967, 'Management of periodic operations in a real-time computation system'. In: *Proc. AFIPS Fall Joint Computer Conference*. pp. 201–208.
227. Yeung, S.-N. and J. P. Lehoczky: 2001, 'End-to-end delay analysis for real-time networks'. In: *IEEE Real-Time Systems Symposium*. pp. 299–309.
228. Zhang, Q., W. Zhu, and Y.-Q. Zhang: 2001, 'Resource allocation for multimedia streaming over the Internet'. *IEEE Trans. on Multimedia* **3**(3), 339–355.

229. Zhang, R., C. Lu, T. Abdelzaher, and J. A. Stankovic: 2002, 'ControlWare: a middleware architecture for feedback control of software performance'. In: *Proc. 2002 International Conference on Distributed Computing Systems*. Vienna, Austria.
230. Zhang, R., S. Parekh, Y. Diao, M. Surendra, T. Abdelzaher, and J. A. Stankovic: 2004, 'Control of Weighted Fair Queueing: Modeling, Implementation and Experiences'. In: *(submitted to) Real-Time Systems Symposium*. Lisbon, Portugal.
231. Zhao, W. and K. Ramamritham: 1985, 'Distributed scheduling using bidding and focused addressing'. In: *Proc. 6th IEEE Real-Time Systems Symposium*. San Diego, CA, USA, pp. 103–111.
232. Zhao, W. and K. Ramamritham: 1986, 'A virtual time CSMA protocol for hard real-time communications'. In: *Proc. 7th IEEE Real-Time Systems Symposium*. New Orleans, Louisiana, USA, pp. 120–127.
233. Zhao, W. and K. Ramamritham: 1987, 'Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints'. *Journal of Systems and Software* pp. 195–205.
234. Zhao, W., K. Ramamritham, and J. A. Stankovic: 1987a, 'Preemptive scheduling under time and resource constraints'. *IEEE Trans. on Computers* **C-36**(8), 949–60.
235. Zhao, W., K. Ramamritham, and J. A. Stankovic: 1987b, 'Scheduling tasks with resource requirements in hard real-time systems'. *IEEE Trans. on Software Engineering* **SE-12**(5), 567–77.