Jorvik: A Framework for Effective Scheduling

Alan Burns and Guillem Bernat
Real-Time Systems Research Group
Department of Computer Science
University of York
Heslington, York. Yo10 5DD
{burns,bernat}@cs.york.ac.uk

16, May 2001

Abstract

This paper presents a collection of mechanisms that together form a framework for the support of flexible scheduling. By choosing a mixture of established and novel techniques, a robust scheme is developed that can provide real support to a wide range of applications that mix hard and soft tasks, periodic and aperiodic load and complex intertask relationships. Key aspects of the framework are the low inherent implementational overheads - no worse that $O(\log n)$, and the fact that it can be implemented with commercial programming languages and operating system standards. A novel aspect of the framework is the use of a $O(\log n)$ algorithm for a threshold based acceptance test for firm tasks. Simulations are provided which illustrate the effectiveness of the proposed scheme. In general it performs as well as the Best-effort scheme which has considerable higher complexity and run-time overhead.

1 Introduction

Over the last decade a considerable volume of literature has been produced which addresses the issues and challenges of flexible scheduling. The focus of much of this research has been on how to effectively support a mixture of periodic and non-periodic, hard and non-hard activities. This work is seen as an enabling technology for a wide range of applications from multi-media to robust control. Notwithstanding the quality of individual contributions, it is unfortunately true that applications have not, in general, taken up these results. There are a number of reasons for this, including the normal inertia associated with technical change, but we would highlight the following.

- Inconsistency in the literature - no single technique has emerged as being widely applicable, indeed there is not even any consensus over the right simulation models to use for evaluation.

- Limitations of the scheduling models - too many models have unrealistic assumptions (such as ignoring run-time overheads).
- Difficulty of implementation - often the scheduling scheme will require operating primitives that are not available on any commercial platform.
- Lack of computational model - the scheduling results are not tied back to realistic models that applications can use to construct systems (for example how can an application exploit a model that has only hard periodic and independent Best-effort aperiodic tasks).
- Lack of computational algorithms - the seminal work on imprecise algorithms (and anytime algorithms) has not led to the development of application-specific instantiations of these generic schemes.

The focus of this paper is on the first four of these problems. The fifth will naturally be addressed once the others have been eliminated (or at least diminished in significance). Within the paper we will both distill the key findings in the literature, and report new results. Together this allows us to define a framework that will have wide applicability and utility. The framework is given the name *Jorvik*. Of course we do not argue that the framework will be effective in all situations. This is not the aim. Rather, we contend that it is possible to deliver, with today's technology, a general purpose platform that will enable flexible scheduling schemes to be exploited in a range of application domains.

The challenges of flexible scheduling are easy to (informally) define: hard deadlines must always be met, and any spare capacity (typically CPU resource) must be used to maximise, perhaps even optimally, the utility of the application. The 'extra' work that could be undertaken typically exceeds the spare capacity available, and hence the scheduler must decide which non-hard task to execute at any time. As processors become more complex and the analysis of worst-case execution time more pessimistic the amount of spare capacity will increase. Indeed the utilisation of the hard task will often be less than 50%. There is always pressure on applications to minimise their use of hard tasks and to exploit flexible scheduling to deliver dynamic behaviour. This is because of the inherent pessimism of the schedulability analysis of the hard tasks, mainly due to the overestimation of the worst case execution times. The framework presented in this paper will facilitate these application requirements.

From the available literature, and the points made earlier, it is clear that any framework must address the following key issues.

- Computational Model - how to structure application tasks so they can make effective use of flexible scheduling.
- Resource Management - how to ensure that hard tasks have their required resources (when they need them) and how to maximise the available resources for the non-hard tasks.
- Load Management - how to schedule the current load of non-hard tasks to maximise the total system utility.

Each of these issues will be addressed in separate sections of the paper. Performance evaluations will then be presented in Section 5 and conclusions in Section 6. However, before commencing on the detailed description of Jorvik it is necessary to identify the constraints on which the framework is built.

1.1 Framework Constraints

As identified above, a major inhibitor to the take-up of the results of flexible scheduling in the inaccessibility of the means of implementation. Jorvik will therefore only make use of standard OS provisions as found in the POSIX standard, and concurrent languages such as Ada and Java. This implies event and time triggered processors, priority based dispatching, dynamic priority scheduling, servers for resource management[1], alarms (temporal) and budgeting[2]. Although these restrictions may limit the expressive power of the framework it does mean that the approach can be used directly on today's OS technology.

The other major constraint concerns run-time overheads. Much published material is misleading as the overheads of the dynamic algorithms are ignored. Jorvik is based on a simple decision process, even when this produces theoretically sub-optimum performance. The reasons for this design constraint are:

- to reduce run-time overhead,
- to be able to measure/analyse the actual overhead,
- to be able to validate an implementation,
- to be implementable on the OS technology outlined above.

Our aim is to use algorithms which have, at worst, only $O(\log n)$ complexity, where n is the number of tasks.

Finally note that this paper is only concerned with single processor scheduling. The increased, and increasing, power of uniprocessors still makes them the main platform for building real-time systems.

2 Computational Model

Little of the published material on flexible scheduling identifies how an application can exploit any proposed scheme in order to build more effective systems. Indeed many papers deal solely with independent aperiodic activities that seem to arrive in the system in a completely random fashion. However, a real-time system is not a general purpose computing resource; it is a single system in support of a single application, albeit a potentially multi-faceted complex one. Once execution has commenced all potential computing activities are known. In this paper we are concerned only with

[1]This feature is not actually required by Jorvik but can be used as an alternative to dual-priority scheduling, see below.

[2]This feature is not widely available; its use and the consequences of it not being available will be discussed later.

activities that have associated deadlines, i.e. real-time jobs. Such jobs can be classified via a number of distinct characteristics:

criticality - jobs that must always complete by their deadline are known as *hard*, others are non-hard; jobs may have components that are of mixed criticality i.e. some hard, but not all.

regularity - time-triggerered jobs that have a regular cycle time are known as *periodic*, event-triggered jobs that have a minimum inter-event separation are *sporadic*, other event-triggered jobs are called *aperiodic* but even these have a bound on their arrival patterns (which may be expressed probabilistically).

predictability - jobs that are not, temporally, highly dependent on input data (or internal state) can have a known worst-case execution time (*WCET*), other jobs will have more dynamic behaviour, but they are never unpredictable - some characteristics of their execution will be known such as average-case execution time (*ACET*).

In order to statically guarantee a hard job (or hard component) it must be either periodic or sporadic and have a known WCET.

The complete set of jobs in an application will have a number of inter-job relationships including data sharing, producer/consumer interaction (this is more usual than client/server [36]) and various forms of precedence orderings. Various studies have show how scheduling parameters can be chosen to produce simple task relationships which nevertheless meet more complex application requirements [29, 30, 26, 10]. Of course a hard activity must never have a synchronous dependency on a non-hard job. Similarly a periodic activity cannot rely on the arrival of sporadic or aperiodic jobs.

In general, non-hard tasks are classified as either *firm* or *soft*. A firm task is one that should complete by its deadline - it is not critical but there is no value in it continuing to execute after its deadline. The definition of soft usually implies that the task has a deadline but that execution after the deadline is still useful. This is a very poorly defined notion. To exploit this property at run-time requires knowledge of how the significance of the task decays after the deadline. It is unclear that applications really possess this property or can represent the decay with any meaningful precision. Hence we conclude that the computational model should contain only hard and firm tasks. If a hard task misses a deadline it is an error condition. A firm task can either be aborted at its deadline or be allowed to complete. We have shown [3] that this computational model can be implemented with standard languages or OS facilities. We have also shown that it can be placed on a formal structure [6].

The restriction that a hard job must have a known WCET limits the functionality of the hard activity. To counter this, a generic structure can be used that implements a hard job as three tasks (two hard and one firm). Figure 1 illustrates this structure. The complete job can be either periodic or sporadic. The firm task F is released by the first hard task H^1; the deadline of F coincides with the release of H^2. In previous publications [5, 3] we have shown how this combination of tasks can implement a wide class of imprecise computations [51, 43, 47], imprecise query systems [15, 56], multiple method [27, 28], heuristic search [37, 45, 35] and anytime algorithms [25, 59, 49, 60]. For many of these applications the final hard task H^2 is omitted, and in some approaches there is more than one firm task executing in the intervening interval (or the same firm task executes a number of times before the deadline).

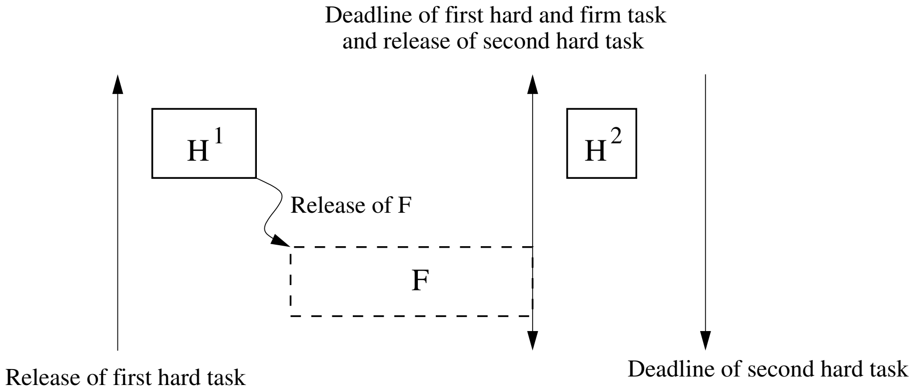

Figure 1: Hard Job as Hard and Firm Tasks

An application with a number of firm tasks, that may have dynamic execution times, cannot statically guarantee all computations. At run-time they may be an overload and a decision must be made as to which jobs to execute and which to pass over. To make this decision, some notion of *utility* must be associated with all firm tasks. The usual approach is to assign a 'value' to the task [44, 14, 9, 20, 32]; if it completes by its deadline it delivers this value to the application. In other work we have addressed the issue of assigning value to tasks [16]. We showed that the construction of an multi-criteria value system is non-trivial. Hence the use of complex time-value functions and very precise measurements of value are not usable or required by applications. We conclude that a relatively small range of values be used [46] and that within the computational model firm tasks have a static value assigned when they are released [33]. For example, if hard task H^1 releases firm task F it will inform the scheduler of the value of F if it completes by its deadline. Note also that the deadline is known statically and that at least the average execution time of F is known. This can be obtained during the testing phase.

Using this scheme for values requires that a classic anytime algorithm (as depicted in Figure 2) is implemented in one of two ways:

- if the underlying OS has a budgeting facility, then the firm task can be assigned a budget and hence its 'value' can be estimated. This is because a guarantee on the minimum amount of processing time a task may receive can be obtained, thus a minimum value is achieved.

- if the underlying OS does not have a budgeting facility then the firm task must structure itself as a series of invocations each with a similar execution time but decreasing value. A run-time decision can be made as to how many invocations to release at any one time [8].

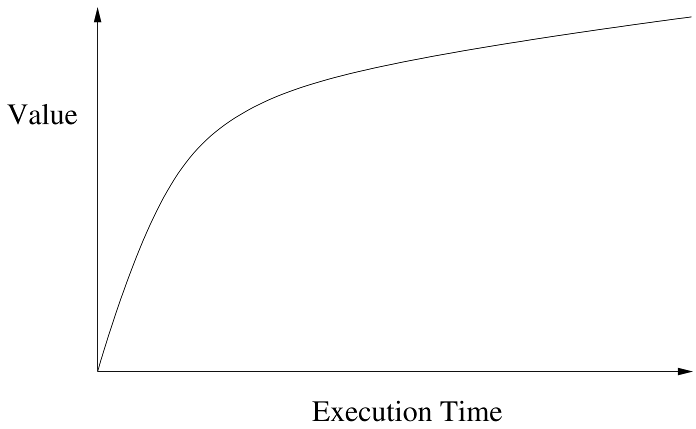

Figure 2: Time-Value Relation

The above discussion has concerned itself solely with firm tasks as the hard ones must be statically guaranteed. There are however some applications that would wish to program graceful degradation following partial failure. In these applications the hard tasks would also be assigned values - but far higher ones than the associated firm tasks - a mode change to 'degraded service' would cause all tasks to be scheduled according to their value (in that mode) [16].

The final topic to address in this section concerns run-time guarantees. Many papers require that all firm (or soft) tasks are given guarantees before they start executing, i.e. they never start unless they can be guaranteed to complete. One motivation for this is that a failure to commit can lead to the task moving to another part of the (distributed) system where more resources may be available. However our assessment of a number of application domains does not lead us to the view that run-time guarantees are essential. Hence we reject them for the following reasons:

- in single processor systems there is nowhere else for the rejected task to go, and the case for load balancing in real-time distributed systems has not been made in practice; there are significant overheads involved in managing global status information and performing task migration,
- it is easy to structure the work of the task as an atomic action [57] so that it has no effect unless committed before its deadline,
- in general, firm tasks do not have fully predictable execution times and hence a complete guarantee is either impossible or leads to very poor schedulability,
- although resource management and scheduling are considered in the next section, the general problem of knowing how much spare CPU resource there is prior to some deadline is computationally complex and cannot be done effectively without a high overhead run-time [22],

- as schedulers are not clairvoyant a ‘guarantee’ may need to be rescindered if the future changes dramatically - this undermines the whole notion of a guarantee.

Clearly it is wasteful to allocate processing cycles to a firm task that then misses its deadline (and hence has no value) but this is an issue for the scheduler, it is not a requirement on the computational model to support run-time guarantees on firm tasks.

In summary, the computational model of Jorvik allows an application to utilise periodic or sporadic hard tasks with known minimal inter-arrival intervals of T, deadline D and worst-case execution times C^w and firm tasks (periodic, sporadic or aperiodic) with bounded arrival characteristics, usually denoted with the period T or average interarrival time T^a, and known average computation time C^a; and characteristics of deadline D and value V - these being known either statically or when the task arrives in the system.

The estimated remaining computation time at a given time t is denoted by C^{left}, whereas D^{left} represents the effective deadline of a task, i.e. the time elapsed until the next deadline expires. Precedence relations between hard and firm tasks allow a range of application algorithms to be supported. Run-time guarantees of firm tasks are not part of the Jorvik framework.

The following notation is also used throughout the paper. We use $\mathcal{H}$ as the set of hard tasks, individual hard tasks are denoted with τ. $\mathcal{F}$ is the set of firm tasks, individual firm tasks are denoted with δ. The average utilisation of hard tasks, U_H^a is therefore:

$$U_H^a = \sum_{\tau_i \in \mathcal{H}} \frac{C_i^a}{T_i^a}$$

this value is static and it is computed off-line. We also use two measures which change dynamically, $\mathcal{Q}$ denotes the set of tasks currently in the queue (ready or running), and L_F is an estimate of the load of the tasks on the system at a particular time t.

3 Resource Management

The key resource of importance is, of course, the processor. The role of processor management (scheduling) is to ensure all hard tasks always meet their deadlines and firm tasks are given the maximum resource possible before their deadlines.

Table driven cyclic executives support hard tasks but lack the flexibility to support value-based scheduling. Although Fohler [26] has shown how some forms of slot movement can enhance the effectiveness of a purely static approach, the approach is not considered further here. Of the two more dynamic schedules, priority-based and deadline-based, Jorvik relies on preemptive fixed priority (PFP) dispatching. Although in theory earliest deadline first (EDF) yields higher utility we adopt PFP as:

- it is easy to isolate hard tasks and to ensure they meet their deadlines,

- PFP is supported by language frameworks (e.g. Ada) and most real-time operating systems and standards,

- there is a full framework of analysis available - including consideration of overheads [17] and jitter [4].

However, we will make use of EDF in an application-level scheduler, when considering the scheduling of the non-hard tasks (see next section).

With a fixed priority scheme the hard tasks are assigned priorities based upon their deadlines. If $D \leq T$ for all tasks then the *deadline monotonic priority ordering* is known to be optimal [42]. If $D > T$ for any task then other algorithms provide an optimal assignment of priority [7, 2].

For the firm tasks the simplest (and baseline) scheme would be to run them all at priorities less then any hard tasks (i.e. as background activities). Improved performance comes from running them at higher priorities - but never to the total exclusion of the hard tasks.

Since 1985 there has been a number of schemes defined for managing aperiodic work. Most of these take the form of servers which provide a limited but immediate quantity of resource that is replenished at defined intervals – example schemes are: Deferrable Server (DS) and Priority Exchange Server (PE) [41, 53], the Sporadic Server (SS) [54] and the Slack Stealing Algorithm (SA) [40, 48, 23]. The slack stealer is optimal but has very high overhead if the system contains sporadic tasks [23]. Interest in the Sporadic Server has increased recently as it is supported directly in the new POSIX standard (P1003.1d) [31]. However, as we showed at last year's RTSS [12], picking the parameters for quality of service of the different server mechanisms is non-trivial. Once overheads are taken into account DS is as effective as SS.

Notwithstanding this work on server technology, Davis [24] has shown that the dual-priority algorithm (DPS) out-performs all of these algorithms. Based on an original idea of Chetto and Chetto [21], DPS partitions the priority range into three groups. Hard tasks start executing in the bottom band but are transferred to the top band (at a known static time) so that they always meet their deadline. Non-hard tasks (the firm tasks of Jorvik) always run in the middle band; they can preempt the lower band but always make way for tasks in the upper set. DPS allows resource sharing and other forms of task interaction. Results[47, 11] have shown consistently that DPS has lower overheads but high performance than other schemes. Where jobs have the structure H^1FH^2 then we will show that the H^1 task must also be executed in the middle band for maximum effect.

To implement DPS requires a dynamic priority facility. This is available in most language provisions and the POSIX standard. Of all the server schemes only SS is available via standard (POSIX) implementations. Jorvik is based on dual-priority preemptive scheduling. However, the schemes defined in the following sections could be used with SS or DS. The provision of the resource is an orthogonal issue to its usage.

All the above schemes are known as *bandwidth preserving* as they attempt to preserve spare capacity if it not currently needed. However the slack stealing and server techniques are all *aggressive* in their use of slack as they assign it to firm tasks whenever such tasks are in the system. With the Jorvik computation model this is inappropriate as the execution of a hard task may lead to the release of a high value firm task. Hence the hard task may need to take priority over other existing firms tasks (i.e. non aggressive use of slack).

The DPS scheme can easily work in this non-aggressive way either by running such hard tasks in the top band only, or by entering the hard task in the EDF queue where it will then execute at an appropriate time or be promoted to the high band to ensure completion by its deadline. Simulation results show that the latter results in the highest performance. For the slack stealing and server techniques the less aggressive method is more difficult to achieve. We are currently investigating a modification of such techniques based a variation of priority inheritance.

Notwithstanding this need for non-aggressive slack usage, only a clairvoyant algorithm can know that slack should be preserved for a high value firm task that will arrive in the future. As a practical scheme cannot be clairvoyant, fully optimal performance is not possible. However during system development, when values are being assigned to firm tasks, it must be noted that low value tasks can always reduce the bandwidth available to high value tasks. And hence a system-level decision must be made as to whether to include such tasks at all. Again this points to the use of only a small range of values in the running system.

4 Scheduling Firm Tasks

Given some mechanism for managing spare processing capacity, the resulting spare resource time must be assigned efficiently to the incoming firm tasks. A number of results in the literature form a boundary to this problem:

- if there is no overload then EDF is an optimal scheme, but if there is an overload EDF performs poorly due to the known domino effect [19],
- with a pure value based scheme, scheduling according to V/C is the most effective [44].

The entity V/C, which we denote as W, gives the value that will accrue per unit of execution. It is optimal if a system is heavily loaded. These two observations have lead to a general scheduling structure which consists of two mechanisms. An *admissions policy* (that keeps the load from exceeding the capacity of the resource) and a *scheduling policy* (for the jobs that are admitted). If the admissions policy is effective then overloads are prevented and hence EDF is an effective scheduling policy. The difficulty with the admissions policy is that any near optimal scheme has very high overheads [52, 50, 34].

The scheme that Jorvik adopts is based on the following observations:

- it is wasteful to start a task that has little chance of completing and that would therefore be aborted (as zero value is obtained after the deadline). However, some level of aborts can be tolerated if it leads to better performance overall,
- in a preemptive system a firm task could arrive with a deadline less then the current running firm task and hence will preempt it. This adds complexity to application-level scheduling. In many cases the new task would still meet its deadline even if the current task is allowed to finish non-preemptively,

- in a busy system the state of the available resources as predicted when a task arrives is a poor indicator of the actual state when the task first gets to run,
- the 'optimal' best effort scheme [44] has extremely high overheads [55, 58] and is only effective if exact execution times are known and delivered at run-time.

The algorithmic behaviour of Jorvik is as follows.

1. When a firm task arrives it has known value V and average computation time C^a (see Computational Model).
2. Upon arrival the task is put into an EDF queue, and information on the current tasks in the system is updated.
3. When a task gets to the front of the queue for the first time, a decision is made to either accept it (execute it) or revoke it (see below).
4. If the task is revoked it is removed from the queue and never executes (for this invocation).
5. If the task is accepted it executes; a budget may be assigned and the task may be aborted at its deadline (see Computational Model).
6. When a firm task finishes (finishes normally, rejected or aborted) the information of current tasks in the systems is updated.

By postponing the decision to execute until this late stage, the data of relevance is as timely as possible; although it is still based on estimates of the resources needed by the tasks in the EDF queue.

As all firm tasks are placed in the EDF queue it will become overloaded. Jorvik implements what might be called a *lazy* admissions policy; tasks are only revoked immediately before they would otherwise start executing, rather than when they arrive.

The decision embodied in stage 3 has two components. One concerns the tasks themselves, the other the load in the EDF queue. We describe Jorvik as a framework rather than a single scheme as some aspects of its approach are not totally prescriptive. The first element of this decision is one such case. The application task can decide if there is sufficient time for the task to complete with a high enough probability (relative to the value that will accrue if the task is successful). Alternatively a simple algorithm can be used (this is the one employed in all simulation results given in Section 5). Assume a firm task δ_i is at the front of the EDF queue, let D_i^{left} be the time until the next deadline. Then the task is rejected if $C_i^a > D_i^{\text{left}}$; put simply, if there is enough time for the average case then it is not rejected. However, to account for the additional interference produced by hard tasks, the test $C_i^a > D_i^{\text{left}}(1 - U_H^a)$ is used. Simulation results shows that the best performance is achieved with the latter.

The second necessary check is the acceptance test.

4.1 Acceptance Test

The acceptance test (which in Jorvik concerns access to the processor not the EDF queue) is itself made up of two parts.

1. An estimate of the current load; is there an overload?
2. If there is an overload should the current task be accepted?

A key intuition behind both of these calculations comes from real-time queuing work of Lehoczky [38, 39]. If the number of tasks in the system is small then there is unlikely to be an overload. However, if the queue of tasks is long then it begins to behave as *heavy traffic* and estimates based on average values have increased validity.

Jorvik is characterised by the separation between the overload detection and the value-based acceptance test. The precision and complexity of both of them depends on the test used.

Clearly, in a system where all future arrival times of tasks are known (periodic) and real execution times are also known, an exact schedulability test for the overload and for the acceptance test is optimal. Independent of the high overhead that such an optimal test takes, the assumption of known arrival times and fixed execution times is too restrictive. Either only probabilistic arrival times are known or real execution times, although bounded, can not be predicted. As a result, exact schedulability tests are overpessimistic and naturally they lead to poor performance.

4.1.1 Overload

The notion of *load* is open to various interpretations, the basic one being the amount of work performed/requested over a period of time. Thus load is a measure that applies for a period of time. However, when we refer to an *overload* it means a situation, i.e. a point in time, in which the amount of work requested can not be finished by the associated deadlines. This has led to confusion in the literature where the terms have been interchanged.

The first component of the Jorvik approach is the determination of if there is an overload at time t. Several alternatives can be employed with different levels of accuracy and complexity.

The exact test is actually a schedulability test. Starting from time t when the acceptance test is performed, the response time of all tasks in the system is computed considering the interference of the other tasks and the interference of future arrivals. This is too complex and also ineffective for non-clairvoyant scenarios. Consider, for example, that the determination of whether there is an overload at time t depends on whether at a future time t' another firm task is accepted, which could depend on whether there was an overload in the first place. The overload test can be expressed in the following way:

$$\text{O1: Overload if } \exists \delta_i \in \mathcal{Q} \ : \ R_i > D_i^{\text{left}}$$

An approximation of this test is to consider the current load on the queue only and test each task for its schedulability, thus not considering future arrivals, i.e. there is an overload at time t if:

$$\text{O2: Overload if } \exists \delta_i \in \mathcal{Q} \; : \; \left(\sum_{\delta_j \in \mathcal{Q} \, \wedge \, D_j \leq D_i} C_j^{\text{left}} \right) > D_i^{\text{left}}$$

This is the test used by the RED algorithm[20]. The test has $O(n)$ complexity whenever the test is performed, all the tasks have to be scanned (this assumes that all tasks can be tested with a single scan of the active tasks), however this approach does not consider the interference from hard tasks.

There is even a simpler approach which is the one used in Jorvik. We can approximate the overload of firm tasks on the queue by:

$$L_F = \sum_{\delta_i \in \mathcal{Q}} \frac{C_i^a}{D_i^a}$$

This formulation is more pessimistic than $O2$ however it includes the future interference of recurring firm tasks. We use D_i^a instead of D_i^{left} to prevent using a value that changes dynamically. The overload test is, therefore, a simple comparison of whether the load of firm tasks is greater than that available load. Assuming that U_H^a is the load of the hard tasks (also computed statically), the overload test is:

$$\text{O3: Overload if } \; L_F > 1 - U_H^a$$

As each firm task, say δ_i, arrives and leaves (either by finishing or by rejection) the current firm task load in the EDF queue is updated:

$$L_F := L_F \pm \frac{C_i^a}{D_i^a}$$

Which is $O(1)$. Note C_i^a / D_i^a can be calculated off-line. Also, in an actual implementation scaling is performed so that integer arithmetic can be used throughout.

Note, other approaches for determination of the overload conditions can be also employed within the Jorvik framework.

4.1.2 Value-Based acceptance test

The operation of the Jorvik value-based acceptance test is an approximation of the behaviour of the Best-effort (BE) [44] queue reordering.

Best-effort queue reordering essentially undertakes the following process. When an overload is detected, it first removes all firm tasks from the queue, then it returns them one at a time in decreasing value-density order. After a task, say δ, is inserted, a schedulability test for each task in the queue is performed. If any task which was already accepted misses its deadline then task δ is dropped, otherwise δ is accepted. This process finishes when all tasks are considered in turn. The complexity of this acceptance test comes from the fact that if there are n tasks in the queue, in the worst case $\frac{n^2}{2}$ schedulability tests are performed. Several schedulability tests can be used, the simplest ones (and less effective) are of $O(n)$ and the more accurate ones, based on

i	$T = D$	C	V	V/C	C/T
F7	220	20	10	0.50	0.091
F6	650	80	25	0.31	0.123
F4	700	80	20	0.25	0.114
F2	180	50	10	0.20	0.278
F1	500	100	15	0.15	0.200
F0	300	80	10	0.13	0.267
F9	750	100	6	0.06	0.133
F8	240	100	5	0.05	0.417
F3	400	150	5	0.03	0.375
F5	450	100	1	0.01	0.222

Table 1: Example firm task set. Task are ordered in decreasing value-density

response time analysis, run in pseudo-polynomial time. In any case, the complexity of the whole test is very high.

There is another essential drawback, BE scheduling is only optimal if the information on the tasks is exact (i.e. actual execution times, and arrival times of future tasks are precisely known). In essence, it is optimal if it is clairvoyant. As we can not make this assumption, the BE test should be performed on either worst case values (leading to very pessimistic assumptions of the current load of the system) or on approximate values (leading to inaccurate estimates of the load). In a non-clairvoyant scenario the prediction embedded in the schedulability test is not accurate and this leads to a relative low value achieved. This is shown later with some simulation results.

Although one could build schedulability tests which deal with a non-clairvoyant schedule, the overhead penalty of this approach is still of the order of $O(n^2)$ times the complexity of the schedulability test, which is clearly unacceptable.

The key idea of the Jorvik approach is to approximate the operation of BE by determining a value-density threshold, denoted by $\mathcal{W}$. A task will be accepted only if its current value density is above the threshold. The threshold value changes dynamically and depends on the tasks in the EDF queue.

Table 1 shows a simple set of tasks to be used as an example. The tasks are sorted in decreasing value-density order. We assume that all tasks are firm, the description of the operation of Best-effort is more straightforward if we only consider firm tasks. Assume that all tasks are released together at $t = 0$ (the task set could also represent the pending computation time and pending deadlines at an arbitrary point in time t, but again, for simplicity we assume all tasks released at time $t = 0$).

We first consider the operation of the Best-effort algorithm. Starting with an empty ready queue, after each task, say δ_i is awakened, a schedulability test is performed to see whether any task already in the queue misses its deadline. All tasks are awoken at $t = 0$ and a schedulability test is performed. In that situation, for instance task $F6$ if it came in last would have a response time of $680 > 650$. This triggers the Best-effort queue reordering. All tasks are taken out of the queue, and inserted one by one in decreasing value-density order. All tasks are accepted, except task F3 (which when inserted makes task F9 miss its deadline). Note task F5 which has a lower value-

```
      Queue of firm tasks
        a - Task accepted
        X - Task rejected

     sorted in descending V/C
       1  2  3  4  5  6  7  8  9 10
     |--+--+--+--+--+--+--+--+XX+--|
 (a) |F7|F6|F4|F2|F1|F0|F9|F8|F3|F5|
     |--+--+--+--+--+--+--+--+XX+--|
```

Figure 3: Results of Best-effort acceptance test.

density than task F3 is accepted because of its lower computation time. After the queue reordering is performed, the tasks accepted are: F7, F6, F4, F2, F1, F0, F9, F8 and F5. Task F3 has been dropped. This is shown in figure 3.

Two ideas can be extracted from this example. The first one is that firm tasks with high value-density are the most likely to be accepted as the load of the high value-density tasks alone is low. Moreover, low value-density tasks are more likely to be rejected as the load of the already accepted tasks is very high. The idea of the threshold value is to approximate the operation of BE by finding a value-density threshold. However, the example also shows that a threshold calculation is sub-optimal. If we use as a threshold value $W_3 = 0.03$, we will reject task $F3$ and all other tasks of lower value-density. But this means also rejecting $F5$ which the Best-effort tests would accept. Anyway, the simulation results show that this suboptimal behavior is not very significant. In fact, the unpredictability of the computation times of the tasks is much more important and actually results in BE performing worse than the Jorvik approach.

The question is therefore how to compute, with a very low overhead, the threshold value. The idea is to find a value $\mathcal{W}$ so that the average load of the tasks of higher value-density than $\mathcal{W}$ is no higher than the maximum, i.e.:

$$\sum_{\delta_i \in \mathcal{Q} \wedge W_i > \mathcal{W}} \frac{C_i^a}{T_i^a} \leq 1 - U_H^a$$

Therefore, a task δ_i is accepted if its $W_i > \mathcal{W}$. This value can be easily found by first sorting the firm tasks in value-density order and then scanning them to find the task for which all tasks of higher value-density have a total utilisation of $1 - U_H^a$. However this approach has a $O(n^2 \log n)$ cost. We can do better than that if we use a data structure to keep the information required to do the test in an adequate form. Note that the load for the acceptance test is C^a/T which is a measure of the average load, whereas for overload computation we use C^a/D which is an approximation of the test $O2$[3]. As new tasks arrive and leave, the threshold also varies. Fortunately, we

[3]This problem is very similar to the n-median problem of p values, that is, finding the value which is greater than $n-1$ of the other components and and smaller than the $p-n$ other values. This problem was solved in the late sixties by Hoare. He presented an algorithm called FIND that has a $O(n)$ cost.

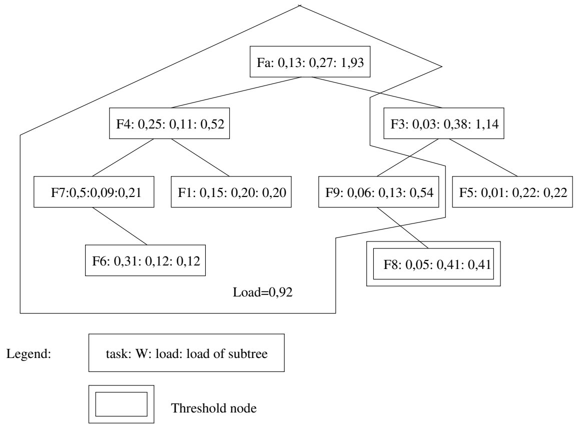

Figure 4: Value-density tree of tasks in table 1 at time $t = 300$ when task $F3$ gets its first tick

can determine the threshold value in $O(\log n)$. The algorithm uses a $O(n)$ size data structure (a balanced tree) that needs to be updated only on task arrival and leave, and which also has a $O(\log n)$ time cost.

Fortunately, it is not necessary to build the set every time. The data structure required is a balanced tree (for instance an AVL tree [1]) of $(W_i, C_i^a/T_i^a)$ values, where the elements on the tree are sorted in descending W_i values (i.e. a traversal of the tree in-order would find the tasks in the queue in descending value-density). Also, each node also holds the sum of the load of all the nodes of its subtree (including the load of node itself).

With such structure, finding the threshold value can be done in $O(\log n)$ time. The problem is to find the node $m = (W_i, C_i^a/T_i^a)$ such that the sum of the load of all nodes with higher value density (nodes to the left of m) is not greater than $1 - U_H^a$ but the load including m is greater than $1 - U_H^a$ The value-threshold is therefore the value of this node. With a search on the tree, therefore requiring $O(\log n)$ only. The structure needs to be updated only at every task arrival or exit. Clearly, such update can be performed in $O(\log n)$ time, even when rotations are required to keep the tree balanced.

Figure 4 shows the corresponding tree structure of the task set in table 1 at time $t = 300$ when task $F3$ gets its first tick and therefore when its acceptance test is performed. At this time, all tasks are in the EDF queue except task $F2$ which has just finished. The threshold node corresponds to task $F8$ as all tasks of higher value-density than $F8$ have a load of 0.92. The threshold value is $W_8 = 0.05$ and as $W_3 = 0.03 < 0.05$ task $F3$ is rejected.

The algorithm is as follows: Let $root$ be the root of the tree, and let for a node n, $n.left$ and $n.right$ be the pointers to, the left and right subtrees. Let also $n.W$, $n.\lambda$ be the value density and load values of the node and $n.sum\lambda$ be the sum of the load of all the nodes in the subtree whose root is n (including the load of the node n itself). The algorithm to find the threshold node, so that all nodes which higher value density have

id	T	D	C of H	C of F	V
X	60	60	3		
H1	120	120	[5,10,20]	[10,50,60]	10
H2	500	500	[5,10,20]	[20,80,100]	5
H3	3000	3000	[5,10,20]	[30,80,100]	19
H4	502	500	[5,10,20]	[20,80,100]	1
H5	700	700	[5,10,20]	[20,80,100]	23
H6	190	190	[5,10,20]	[10,50,60]	10
H7	200	200	[5,10,20]	[20,80,100]	5
H8	4000	4000	[5,10,20]	[30,80,100]	6
H9	400	400	[5,10,20]	[20,80,100]	9
Ha	600	600	[5,10,20]	[20,80,100]	15

Table 2: Task set TS1. Times expressed as fixed or with a bounded exponential distribution $[minimum, average, maximum]$

a load not greater than L is :

```
function threshold (L : load) return valuedensity is
 leftl : load; -- load of the nodes to the left of node n

 function sumload(n :node) return load is
 begin
    if n=null then return 0.0; else return n.sumλ; end if;
 end;

 function search (n : node) return valuedensity is
 begin
        if leftL + sumload(n.left) > L then
            return search(n.left);
        elsif leftL + sumload(n.left) + n.load > L then
            -- current node holds the threshold value
            return n.Wi;
        else -- traverse the right subtree
            leftL := leftL + sumload(n.left) + n.L;
            return (search (n.right));
        end if;
 end search;

begin -- of threshold
    if root.suml <= L then
        -- the total load is below the maximum. No threshold
        return (0.0);
    else
       leftL := 0.0;
       return (search (root));
    end if;
end threshold;
```

5 Evaluation

Extensive simulation analysis have been performed to evaluate the performance of the Jorvik approach and to determine the sensibility of the selection of parameters. The simulations investigate the following issues.

- Comparison of the performance of Jorvik against other firm load management techniques such as Best-Effort queue reordering and simple FIFO without acceptance tests.
- Determination of robustness of the performance achieved across a wide range of parameter configurations.

5.1 Simulation Test-bed

Two types of simulation studies have been performed. The first one simulates the same task set with varying execution times, and therefore different load. The second one simulates different task sets. All simulations shown correspond to 10 million ticks runs.

The first type of simulation uses the task set shown in Table 2. The task set is made up of 1 hard tasks and 10 HF tasks. Times are expressed either as constant or as a bounded exponential time distribution with the triple $[a, b, c]$ meaning an exponential distribution with a minimum value of a with average value b and cut at the maximum c; this allows a variability on the execution time of the task but also has a known WCET (other bounded time distributions have been used in the simulations leading to similar results). The average load of the hard tasks is 0.34. However, if we consider worst case values (minimum values for periods, and maximum values for computation times) the load rises up to 0.63. The average load of the complete task set is 2.23 and the worst case load of all the tasks is 2.96. In the simulations the load of the task set is adjusted in the range 0.5 to 3.2 by multiplying all computation times by an adequate factor α.

For the second type of experiments, 200 task sets have been generated randomly. Each task set has 10 transactions. The average load of the hard tasks ranges from 0.05 to 0.15. The total load of different task sets ranges from 0.62 to 2.77. The task sets have been generated as follows. Task periods follow a bounded exponential distribution $[a, b, c]$. Firstly average task period b has been randomly selected between 100 and 10000, minimum and maximum values have also been randomly selected in the range $[b/2, 2b]$. Deadlines have been set as the smallest interarrival time; then, computation times have been selected to achieve the desired utilisation. Task values have been generated randomly following a uniform distribution between 5 and 250. The dual priority mechanism requires two priorities for each hard task, the initial base priority and the promoted priority. The promoted priorities have been set up according to deadline monotonic priority ordering, worst case response times have been calculated and used to determine the priority promotion times ($Y_i = D_i - R_i$). Two alternatives for the assignment of the initial priority have been evaluated. The first one, called DP-below is the traditional dual priority assignment where hard tasks start executing at a priority level which is below the priority of the firm tasks, the second one, called DP-inside,

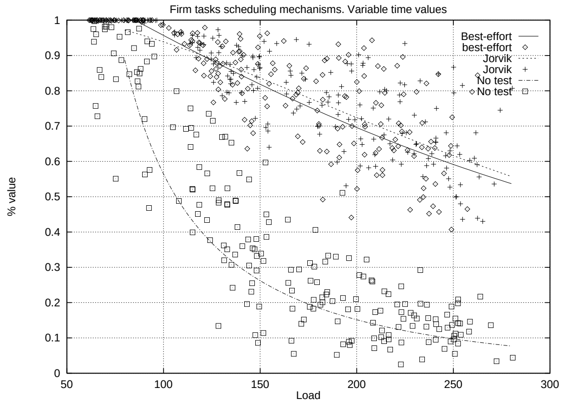

Figure 5: Percentage of total value achieved by the Jorvik scheduling framework, Best-effort queue reordering and a basic mechanism using random time distributions.

where hard tasks start at the same priority as all of the firm tasks, effectively being scheduled in EDF.

5.2 Experiments

5.2.1 Jorvik comparison

We have evaluated the performance of the Jorvik approach in comparison with the Best-effort queue reordering and against a basic mechanism for scheduling firm tasks in which no acceptance test is performed and where firm tasks are processed in FIFO order. This is shown in figure 5. The simulation presents the average value achieved (value achieved divided by total value submitted to the system). For the Jorvik approach, hard tasks base priorities are DP-inside, and the queue is preemptive. Each dot in the graph is the result of the simulation of a different task set. Both Jorvik and BE work very well in underload conditions and the performance decreases as the system gets overloaded. As expected, the basic mechanisms with no acceptance test performs very badly. Trend lines are also shown on the graph. In average the Jorvik approach is not only close to the Best-effort one, but superior by around 3%. This is due to the fact, as mentioned before, that Best-effort is a worst case test and performs less well under non clairvoyant scenarios. Jorvik achieves the same level of performance of the Best-effort at a fraction of its complexity (for instance, a simulation of one million ticks of the same task set on a desktop computer took 80 seconds for the best effort but only 4 seconds for the Jorvik framework).

Figure 6 shows the results of the same simulation but assuming that all the time distributions are known fixed values, i.e. fixed periods and known fixed computation times. In this situation, Best-effort is slightly better than Jorvik as the future is more predictable. Note, however, that the performance of Jorvik is very close to the performance achieved by Best-effort.

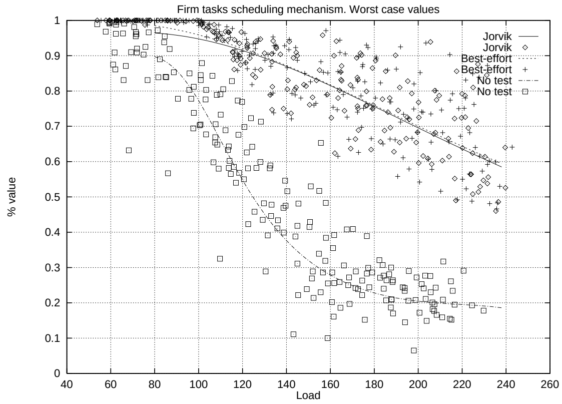

Figure 6: Percentage of total value achieved by the Jorvik scheduling framework, Best-effort queue reordering and a basic mechanism using fixed time distributions (worst case values).

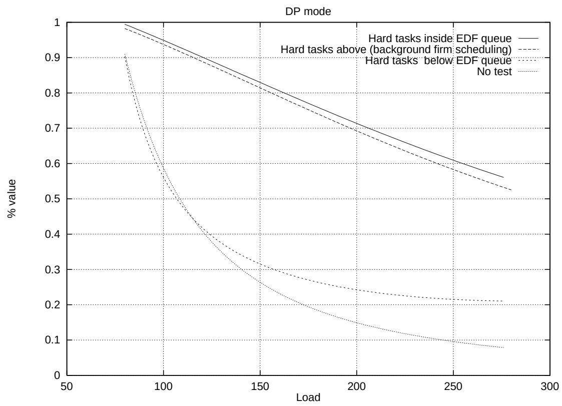

Figure 7: Trend line of average value achieved for different initial dual priority assignments.

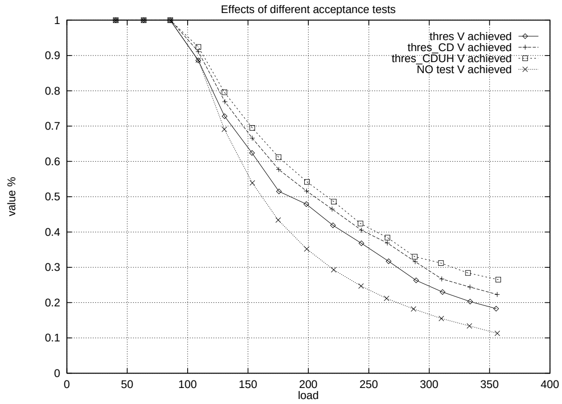

Figure 8: Percentage value achieved for different acceptance tests.

5.2.2 DP priority assignment

The second experiment, shown in figure 7, studies the effect of the different assignment of DP base priorities on the average value achieved (for clarity, we have omitted the plot of the particular task sets and only trend lines are displayed). The top line corresponds to DP-inside, the second one corresponds to starting the hard tasks above the EDF queue, therefore, effectively scheduling the firm tasks in background. The lower line corresponds to the traditional dual priority assignment, DP-below.

The first important results is that DP-below performs very badly. This is due to aggressive slack management. The analysis of the simulation log shows that most of the times a single firm task is running which prevents hard tasks from running and therefore releasing further firm tasks (with potential higher value). In essence, the performance is very close to background scheduling of firm tasks in FIFO order. This problem will occur in any aggressive slack method that gives preference to firm tasks over hard ones, so we expect aperiodic servers and slack stealing algorithms to have a poor performance. On the contrary, the assignment of the initial base priorities at the EDF queue ensures that as new hard tasks arrive they may run and therefore they may release their firm tasks, thus achieving a high value.

This problem will occur when scheduling HF tasks with any aggressive slack method (servers, slack stealing). To prevent it a similar solution as the one devised with DP-inside has to be employed. Unfortunately, this is not as easy as with dual priorities and modifications to the way slack management works should be addressed. This issue is currently under investigation, the solution proposed is called *value ceiling* and it is based on giving preference to hard tasks which can potentially release a firm task with a higher value than the ones currently in the queue[13].

5.2.3 Jorvik acceptance tests

The following experiment tests the performance of different acceptance tests. The graph shown in figure 8 shows four different acceptance tests: The first one, THRES, only performs the value-density threshold comparison; the second one, THRES-CD, tests both the value-density threshold and whether $C < D^{\text{left}}$; the third one, THRES-CDUH is THRES with the test $C < D^{\text{left}}(1 - U_H^a)$; the final graph performs no test. All simulations assume a DP-inside priority assignment for the hard tasks.

For very low hard utilisations, there is no essential difference between THRES-CD and THRES-CDUH, however for not so low hard load, the $C < D^{\text{left}}$ test is optimistic as it does not consider the potential interference from hard tasks until the deadline expires. Performing the threshold value test alone does not perform as well as before because of the fact that higher value density are allowed to run even when they do not have time to finish by their deadline.

6 Conclusion

Jorvik is a framework for scheduling mixed hard and firm tasks. It uses a slack management mechanism such as dual-priorities (or server mechanisms) for giving service to firm tasks.

Firm tasks are scheduled in EDF order and an acceptance test is performed when they receive their first tick. When a task δ_i gets its first tick then the following can happen:

- if $C_i^a > D_i^{\text{left}}$ then δ_i is rejected – or some other application specific test is substituted,
- if there is not an overload then the task is accepted,
- otherwise, if the value-density of the task is above the threshold the task is accepted,
- otherwise, it is rejected.

In any case, if the firm task is not finished by the deadline, it is aborted. The hard tasks are guaranteed to finish by the deadline and therefore no test is necessary.

The overload computation is $O(1)$, the value-density structure is updated twice for each task (once on arrival, and once on exit) with a $O(\log n)$ cost. The threshold computation is only performed under overload and when $C_i^a \leq D_i^{\text{left}}$ and also has a $O(\log n)$ cost. The additional data structures needed are $O(n)$ (basically a record for each task). All data is stored as integer values for which proper scaling has been performed so that all the implementation can be done with integer arithmetic only. The loss of precision due to this conversion is not significant.

Also note that the cost of handling an EDF queue is also $O(\log n)$ for insertions and deletions.

All the mechanisms used by Jorvik can be implemented with the features available in Ada 95 or by POSIX compliant systems on top of a priority scheduling kernel.

- Dual priority can be implemented with two threads per hard task, one base thread and another one which implements the priority promotion[3].

- EDF scheduling can be implemented at a library level using one special passive task which handles the necessary priority changes[18].
- The load management, load structure and acceptance test are implemented as passive objects and can be built without relying on underlying OS kernel structures.

Overall the Jorvik framework presents an effective means of supporting flexible scheduling and thereby facilitating the use of such techniques in a wide range of application areas.

References

[1] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Soviet Math. Doclady 3*, pages 1259–1263, 1962.

[2] N. C. Audsley. On Priority Assignment in Fixed Priority Scheduling. *Information Processing Letters*, to appear.

[3] N. C. Audsley, A. Burns, R.I. Davis, and A. J. Wellings. Integrating unbounded software components into hard real-time systems. In S. Natarajan, editor, *Imprecise and Approximate Computation*, pages 63–86. Kluwer Academic Publishers, 1995.

[4] N. C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[5] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Incorporating unbounded algorithms into predictable real-time systems. *Computer Systems Science and Engineering*, 8(2):80–89, 1993.

[6] N.C. Audsley, A. Burns, R.I. Davis, D.J. Scholefield, and A.J. Wellings. Integrating optional software components into hard real-time systems. *Software Engineering Journal*, 11(3):133–140, 1996.

[7] N.C. Audsley, K.W. Tindell, and A. Burns. The end of the road for static cyclic scheduling. In *Proceedings of 5th Euromicro Workshop on Real-Time Systems*, pages 36–41, Oulu, Finland, 1993.

[8] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Optimal reward-based scheduling of periodic real-time tasks. In *20th IEEE Real-Time Systems Symposium. Phoenix. USA*, December 1999.

[9] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of online real-time task scheduling. *Real-Time Systems*, 4(2):124–144, 1992.

[10] I. J. Bate and A. Burns. An approach to task attribute assignment for uniprocessor systems. In *Proceedings of the 11th Euromicro Workshop on Real-Time Systems*, pages 46–53, 1999.

[11] G. Bernat and A. Burns. Combining (n m)-hard deadlines with dual priority scheduling. In *Proceedings of 18th IEEE Real-Time systems symposium. San Francisco, CA*, December 1997.

[12] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *20th IEEE Real-Time Systems Symposium*, Phoenix. USA, December 1999.

[13] G. Bernat and A. Burns. Performance evaluation of the jorvik scheduling framework. Tech. Report YCS, Department of Computer Science. University of York, 2000.

[14] S. Biyabani, J.A. Stankovic, and K. Ramamritham. The integration of criticalness and deadlines in scheduling real-time tasks. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 152–169, 1988.

[15] P. Buneman, S.B. Davidson, and A. Watters. A semantics for complex objects and approximate queries. In *Proceedings of the 7th Symposium on the Principles of Database Systems*, pages 305–314, 1988.

[16] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Stringini. The meaning and role of value in scheduling flexible real-time systems. *Journal of Systems Architecture*, 46:305–325, 2000.

[17] A. Burns, K. Tindell, and A. J. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Trans. Software Enginering*, 21(5):475–480, 1995.

[18] A. Burns and A. J. Wellings. *Concurrency in Ada*. Cambridge University Press, 1995.

[19] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1995.

[20] G. Buttazzo and J. Stankovic. RED : A robust earliest deadline scheduling algorithm. In *Proceedings of the 3rd International Workshop on Responsive Computer Systems, Austin*, 1993.

[21] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, 1989.

[22] R.I. Davis. *On exploiting spare capacity in hard real-time systems*. PhD thesis, University of York, UK, 1995.

[23] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Proceedings Real-Time Systems Symposium*, 1993.

[24] R.I. Davis and A. J. Wellings. Dual priority scheduling. In *Proceedings Real-Time Systems Symposium*, pages 100–109, 1995.

[25] T.L. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54, 1988.

[26] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings of the 16th Real-Time Systems Symposium*, 1995.

[27] A. Garvey and Victor Lesser. Design-to-time real-time scheduling. *IEEE Transactions on Systems, Man and Cybernetics*, 23(6):1491–1502, 1993.

[28] A.J. Garvey and V.R. Lesser. A survey of research in deliberative real-time artificial intelligence. *Real-Time Systems*, 6(3):317–47, 1994.

[29] R. Gerber, S. Hong, and M. Sabsena. Guaranteeing end-to-end timing constraints by calibrating intermediate processes. *IEEE Real-Time Systems Symposium*, 1994.

[30] J.P. Gutierrez, J.G. Garcia, and M.G. Harbour. Optimized priority assignment for tasks and messages in distributed real-time systems. *IEEE Parallel and Distributed Systems*, pages 124–131, 1995.

[31] IEEE Std.1003.1d D12. Draft Information Technology - Portable Operating System Interface (POSIX): Part 1 : System Application program interface (API) [CLanguage]– Amendment d: Additional Realtime Extensions. [CLanguage], January 1999.

[32] E.D. Jensen, C.D. Locke, and H. Tokuda. A time driven scheduling model for real-time operating systems. In *Proceedings IEEE Real-Time Sytems Symposium*, pages 112–122, 1985.

[33] Chen K and P. Muhlethaler. A scheduling algorithm for tasks described by time value function. *Real-Time Systems*, 10(3):293–312, 1996.

[34] Y.S. Kim. An optimal scheduling algorithm for preemptive real-time tasks. *Information Processing Letters*, 50(1):43–48, 1994.

[35] S. Koenig and R. Simmons. The effect of representation and knowledge on goal-directed exploration with reinforcement learning algorithms. *Machine Learning Journal*, 22:227–250, 1996.

[36] H. Kopetz. The time-triggered model of computation. In *Proceedings 19th Real-Time Systems Symposium*, pages 168–177, 1998.

[37] R. Korf. Real-time heuristic search: First results. In *Proceedings of the AAAI*, pages 133–138, 1987.

[38] J.P. Lehoczky. Real-time queueing theory. In *Proceedings 17th Real-Time Systems Symposium*, pages 186–195, 1996.

[39] J.P. Lehoczky. Real-time queueing network theory. In *Proceedings 18th Real-Time Systems Symposium*, pages 58–67, 1997.

[40] J.P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks fixed-priority preemptive systems. In *Proceedings 13th Real-Time Systems Symposium*, pages 110–123, 1992.

[41] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environment. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[42] J. Y-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation (Netherlands)*, 2(4):237–250, Dec. 1982.

[43] J.W.S. Liu, K.J. Lin, W.K. Shih, A.C.S. Yu, J.Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, pages 58–68, 1991.

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

[45] S. Matsubara and T. Ishida. Real-time planning by interleaving real-time search with subgoaling. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pages 122–127, 1994.

[46] C. McElhone and A. Burns. Scheduling optional computations for adaptive real-time systems. *Journal of Systems Architectures*, 46:46–77, 2000.

[47] R. Oliveira and J. Fraga. Scheduling imprecise computation tasks with intra-task/ inter-task dependence. In *Proceedings of the 21st IFAC/IFIP Workshop on Real-Time Programming, WRTP'96*, pages 51–56, 1996.

[48] S. Ramos-Thuel and J.P. Lehoczky. On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems. In *Proceedings of 14th IEEE Real-Time Systems Symposium*, pages 160–171, December 1993.

[49] S.J. Russell and S. Zilberstein. Composinng real-time systems. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 212–217, 1991.

[50] K. Schwan and H. Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Transactions on Software Engineering*, 18(8):736–748, 1992.

[51] W. K. Shih, J. W. S. Liu, and J. Y. Chung. Algorithms for scheduling imprecise computations with timing constraints. In *Proc. IEEE Real-Time Systems Symposium*, 1989.

[52] M. Silly, H. Chetto, and N. Elyounsi. An optimal algorithm for guaranteeing sporadic tasks in hard real-time systems. In *Proceedings 2nd IEEE Symposium on Parallel and Distributed Systems*, pages 578–585, 1990.

[53] B. Sprunt, J. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Proceedings IEEE Real-Time Systems Symposium*, pages 251–258, 1988.

[54] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1:27–69, 1989.

[55] H. Tokuda, J.W. Wendorf, and H.Y. Wang. Implementation of a time-driven scheduler for real-time operating systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 271–280, 1987.

[56] S.V. Vrbsky and J.W.S. Liu. Approximate: A query processor that produces monotonically improving approximate answers. *IEEE Trans. on Knowledge and Data Engineering*, pages 1056–1068, 1993.

[57] A.J. Wellings and A. Burns. Implementing atomic actions in Ada 95. *IEEE Transactions on Software Engineering*, 23(2):107–123, 1996.

[58] J.W. Wendorf. Implementation and evaluation of a time-driven scheduling processor. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 172–180, 1988.

[59] S. Zilberstein. Resource-bounded sensing and planning in autonomous systems. *Autonomous Robots*, 3:31–48, 1996.

[60] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.