Flexible Scheduling in Multimedia Kernels

Pierre G. Jansen, Hans Scholten
Dept. of Computer Science
University of Twente
Eindhoven, Netherlands
jansen,scholten@cs.utwente.nl

Emiel Wijgerink
Dept. of Math. and Comp. Science
Eindhoven University
Eindhoven, Netherlands
emiel@win.tue.nl

Abstract—Current Hard Real Time (HRT) kernels have their timely behaviour guaranteed on the cost of a rather restrictive use of the available resources. This makes current HRT scheduling techniques inadequate for use in a multimedia environment where we can make a considerable profit by a better and more flexible use of the resources. We will show that we can improve the flexibility and efficiency of multimedia kernels by combining Real Time Transactions with Deadline Inheritance policies under the Earliest Deadline First or the Deadline Monotonic rule. A currently running multimedia experiment shows a scheduling overhead of less than 0.5 percent. We also present the main ingredients for an open and adaptive Quality of Service (QoS) management. In particular a technique for rapid QoS schedulability analyses is offered. This technique can be applied to Hard Real Time analyses as well.

Keywords—Real Time kernel, transactions, deadline inheritance, multimedia, QoS, schedulability

I. INTRODUCTION

RECENT developments in the field of multimedia and communication architectures, open and exiting range of new applications. This is particularly true when combining them to new powerful systems that deal with vision, sound and control. In multimedia we have the possibility of manipulating sound and vision, robotics deals with control, while recent developments on computer and communication architectures open the possibility to distribute these functions. This opens a range of new architectures among which we find TV computers, Set Top Boxes and network computers. Typical applications for these architectures are shopping, games, (tele-) education, travel services, dating, (video-) conferencing etc. Moreover, multimedia can and will be used also in other than these typical applications. For instance in the process control environment we see the deployment of the combination of remote viewing, remote hearing and remote control, mainly composed from “of the shelf” components. Furthermore the deployment of Asynchronous Transfer Mode (ATM) network [10] allows for real time transmission of data. We may expect a Continuous Media (CM) stream to be transferred via ATM networks and to be (un)packed by the AAL5 interface [10]. We will refer this subject briefly in section VI.

Multimedia applications have in common that they require some degree of timeliness ranging from Soft Real Time (SRT) for most multimedia applications to Hard Real Time (HRT) for process control applications. Consequently this requires an operating system kernel that can handle timeliness, so we need a Real Time (RT) kernel that can give adequate support. There are already numerous kernels on the market. None of them is a standard or a de facto standard for RT purposes. The reason for this is that none of them is flexible enough to support a wide range of RT applications. The basic problem is twofold:

1. When used for multimedia tasks, scheduling of tasks and reservation of resources is far from optimal. Moving pre-calculated off-line scheduling to flexible online scheduling may cause phenomena like priority inversion, late reactions, and unnecessary reservation of resources.
2. When used for HRT tasks, these kernels are often inflexible. Current HRT tasks either require off-line reservation of resources or some other off-line guarantee for schedulability of tasks. New tasks may require a time consuming re-scheduling of all the existing tasks.

This paper investigates the possibilities of a multimedia scheduler that is flexible, fair and responsive enough to support multimedia continuous streams, and multimedia processes. We developed a technique for flexible and open QoS control. This technique allows for dynamic admission of new multimedia task and for QoS variation of running tasks. The main ingredient here is a flexible scheduling support and a rapid schedulability analyses both at runtime.

RT kernels have scheduling algorithms which can be classified as dynamic or static [4].

• A dynamic kernel uses little or no a-priori information about the arrival of tasks. Scheduling decisions are made at the moment a task arrives.
• A static kernel however uses a-priori information about periods, arrival times and resource usage and makes scheduling decisions off-line.

Static algorithms can be tuned optimal for static

1 The research is executed at the University of Twente in the Multimedia “Tukker” project.

2 This technique is sufficiently precise to be used for HRT purposes.
tasks, while dynamic algorithms offer more flexibility. Because of the need for flexibility in multimedia this paper concentrates on dynamic scheduling algorithms. They are based on variations of the Earliest Deadline First (EDF) rule. Periodicity of tasks is not of importance to EDF in the first place. However when it comes to QoS schedulability analyses we need to take into account period times and we will consider periodic EDF and Deadline Monotonic algorithms.

We first give an overview of the existing techniques in the dynamic kernels in section II. Our task model is described in section III, new scheduling protocols are presented in section IV, performance aspects are considered in section V, a brief overview of QoS management is given in section VI, and a method for QoS schedulability analyses is presented in section VII. Finally a prototype of a multimedia kernel is presented in section VIII.

II. EXISTING PREEMPTIVE SCHEDULING METHODS

In preemptive scheduling algorithms tasks are scheduled according to a priority. A task may preempt another task if it has a higher priority. In the following well known scheduling methods the priorities are determined as follows:

- Earliest Deadline First (EDF). Priority increases dynamically when the deadline comes closer.
- Rate Monotonic (RM). At runtime the highest priority is assigned to the task with the shortest period and the deadline is equal to the period.
- Deadline Monotonic (DM). At runtime the priority is assigned to the task with the shortest deadline, which is smaller than its period.

Note that RM is a special case of DM. Note also that EDF task do not have a fixed period necessarily. However if they have, it could open the possibility to derive the feasibility of a schedule for a set of periodic EDF tasks. We will consider this in more detail in section VII.

Without any precautions these scheduling methods may lead to the phenomenon of priority inversion. This happens when other resources than the processor come into view. Priority inversion could happen when a high priority task is blocked for a resource that is held by a low priority task. The latter may not proceed due to its low priority, consequently blocking the high priority task.

Depending on the synchronisation policy such as the Fixed Priority protocol, the Basic Inheritance protocol, the Priority Ceiling Protocol [14] or the Real Time Transaction (RTT) protocol [5] a dispatcher assigns tasks to the processor(s). The Fixed Priority protocol generally suffers from priority inversion, whereas the Basic Inheritance Pro-
tocol, the Priority Ceiling Protocol and the Real Time Transaction Protocol provide methods to bound the duration of priority inversion.

The Basic Inheritance (BI) protocol realises this by inheritance of priority. Low priority processes, owning shared resources that are also requested by high priority processes, inherit the high priority from the waiting processes. BI can limit priority inversion, however it can not avoid transitive waiting in a chain of processes, which are all waiting for the release of resources of their predecessors.

The Priority Ceiling protocol (PC) avoids priority inversion and also transitive waiting. The basic idea is to make way for high priority jobs, even if it is not certain that they will become active. The rule is that a medium priority job may not preempt a low priority job if the low priority job holds resources that could be claimed by a high priority job. The priority ceiling associated with a resource is the highest priority of a job that ever can claim this resource. PC is rather strict in preserving resources for processes of which it is not known whether they will become active. This prohibits preemption of a low priority job by a medium priority jobs, even when the high priority job seldom or never shows up.

The Real Time Transaction protocol (RTT) also avoids priority inversion and transitive waiting. It roughly works as follows:

When a transaction starts, it simultaneously acquires all resources it needs to complete the transaction. During the transaction resources can only be released. A transaction has completed when it has released all of them and becomes free running. A transaction is assigned a processor if it has the highest priority and when it can acquire all its requested resources. Priority inheritance is used when a high priority transaction is waiting for a low priority transaction in order to avoid preemption of the low priority transaction. In contrast to PC, RTT is more generous to the medium priority processes, which may preempt low priority processes.

We will now introduce a task model suited to continuous media and which supports our scheduling policy.

III. TASKS

This section discusses two protocols that are based on the Earliest Deadline First rule and which are enhanced by Deadline Inheritance refinements. The protocols have been designed in such a way that:

- They are fair in strategy. Tasks are served in accordance to the best knowledge of the system. A system can serve a task better if it has adequate information of the task.
• They are flexible with respect to the use of resources. Reservations are only done when necessary\(^3\).
• They have a low administration overhead, which is necessary in order to limit consuming scarce “real” processor time.

We now introduce our task model.

A. Task model

A non-periodic task is a sequence of “free running” or “resource using” transactions, similar to the Real Time Transaction Protocol [5]. A free running transaction is not subject to mutex scheduling constraints since it does not use shared resources. This makes it easy to schedule these transactions. (This is discussed in III-D with more detail). An invocation of a “resource using transaction” can only be run if it can acquire all its resources simultaneously. This condition guarantees that a transaction always runs to completion\(^4\). Unbounded priority inversion, transitive waiting and deadlock\(^5\) are impossible. A transaction may release its shared resources at any time. However, for the sake of simplicity, we assume that a transaction releases its resources when it runs to completion. Dealing with early release times is possible, however, a little more complicated and beyond the scope of this paper.

In fact a transaction is a subtask. From here on we are only referring to transactions and every time when we write task, or subtask we mean a transaction unless stated otherwise.

When we refer to periodic tasks we assume that they can be modeled as a single periodic transaction. Also here further refinements are possible. They are however beyond the scope of this paper. Whether tasks are periodic or not is not yet of interest but will become so when QoS schedulability has to be determined (see section VII).

B. Task states

A task may be in one of the following states: sleeping/pre-announced, ready, or flushed. The ready state is split up in new released, running or preempted. States and state transitions are shown in figure 1.

A task is put in the administration after the occurrence of an event from outside, e.g. an interrupt or receipt of data. The task is put in the sleeping administration where it is waiting for its release

\(^3\)The ceiling protocol is Static. It may keep a task preemptable if it has a resource in use that also would be used by a higher priority job, even if this job does not show up.

\(^4\)A consequence is that, waiting time for external events has to be bounded such that the transaction can finish before its deadline.

\(^5\)Deadlock is a special form of transitive waiting.

Fig. 1. Task invocation states and transitions

time, after which it leaves the administrations and it enters the ready queue. Pre-announcement is a special form of sleeping. The task is not yet active but the system takes already into account that the task will show up within a finite time. Pre-announcements are explained in detail in [6] and will not be considered in this paper.

After its release a task will go to the ready state and when it is terminated to the flushed state. In the ready state a task can be running when it has the processor or new-released/preempted when not. When a task is finished it is withdrawn from the administration.

In general the scheduling of tasks is subject to constraints. We will define the constraints and show that our proposed scheduling algorithms will obey them.

C. Timing attributes

The following task timing attributes of a transaction \(\tau\) are used for scheduling decisions:

• A release time \(r(\tau)\) is the absolute time from which a transaction may run,
• A deadline \(d(\tau)\) is the absolute time at which a transaction has to be completed.

and for the determination of the schedulability of a transaction set we have:

• A runtime interval\(^6\) \(C(\tau)\) is the maximum time a transaction takes to complete and
• A period \(T(\tau)\).

A RD-interval of a transaction \(\tau\) is the interval between release time and deadline: \(d(\tau) - r(\tau)\).

The set of resources which are in use by a transaction \(\tau\) is denoted by \(R(\tau)\).

\(^6\)Runtimes are mainly used in conjunction with QoS schedulability analyses for periodic processes. They are not needed by the scheduling algorithm.
D. Scheduling Constraints

The invocation of a transaction is subject to constraints resulting from the communications links, release times, deadlines, and resource requirements. Denote the release time and deadline of a transaction $\tau$ as $r(\tau)$ and $d(\tau)$ respectively. When a transaction $\tau_a$ is executed before $\tau_b$, there exists a precedence relation between them, denoted by $\tau_a < \tau_b$. The five types of scheduling constraints are:

**Precedence:** If $\tau_a$ must precede $\tau_b$ then there exist a precedence constraint between them and $\tau_a < \tau_b$ must be enforced.

**Release:** A transaction $\tau_a$ cannot start before its release time $r(\tau_a)$.

**Deadline:** A transaction $\tau_a$ is supposed to be finished before its deadline $d(\tau_a)$.

**Mutex:** If two transactions $\tau_a$ and $\tau_b$ require the same resource $(R(\tau_a) \cap R(\tau_b) \neq \emptyset)$, then they are not allowed to preempt each other.

**Equality:** A transaction with a deadline equal to a running transaction may not preempt the running transaction.

Note that the **mutex constraint** enforces mutual exclusion and that if a scheduling algorithm keeps to mutex constraints no additional synchronisation is required.

A mutex constraint can also be defined in terms of precedence constraints: the invocation of $\tau_a$ must precede the invocation of $\tau_b$ or vice versa. The invocations of both transactions have to execute in some sequential order that is not specified by the constraints. This can however only be used in a static environment where both transactions are known to exist and have known timing parameters.

Note that the following relations always must hold:

- $\tau_a < \tau_b$ implies $r(\tau_a) \leq r(\tau_b)$ (release time ordering)
- $\tau_a < \tau_b$ implies $d(\tau_a) \leq d(\tau_b)$ (deadline ordering) and
- $\tau_a < \tau_b$ implies $d(\tau_a) \leq r(\tau_b)$ (sequential ordering)

IV. THE PROPOSED SCHEDULING SCHEME

In this section we present the design of an integrated scheduling scheme. It basically uses the combination of RT transactions, the Earliest Deadline First (EDF) rule and deadline inheritance protocols. Before discussing the scheme we first present plain EDF rule and the deadline monotonic rule as an introduction and thereafter we discuss the inheritance protocols.

A. Non-inheritance protocols

In this section we will consider two non-inheritance protocols: (1) Earliest Deadline First (EDF) and (2) Deadline Monotonic (DM). The EDF strategy is bound to the following rules:

1. The invocations are subject to the release, mutex and equality constraints.
2. The algorithm executes one invocation when runnable invocations are available. The executing invocation has the minimum deadline of all runnable invocations, which is enforced by preemption if this were not the case.

We emphasize that the EDF algorithm maintains the precedence constraints according to given deadlines.

A direct consequence of our EDF scheduling strategy is the following: an invocation of a transaction $\tau_a$ can only preempt another invocation of a transaction $\tau_b$ if the RD-interval of $\tau_a$ is completely comprised by the RD-interval of $\tau_b$ (i.e., the invocation $\tau_a$ has a greater release time and a shorter deadline than the invocation of $\tau_b$).

Note that if the sizes of the two RD-intervals are equal, preemption is never possible. A special case is that two invocations of a single transaction can never preempt each other.

The DM protocol is similar to the EDF protocol. For DM however the meaning of deadline is different from the one used in EDF. While in EDF the priority of a transaction increases dynamically with time, the priority in DM is statically fixed. By giving the static meaning to deadline we can keep to the same text for DM as for EDF.

EDF and DM do not solve the problem of transitive waiting when shared resources come into play. The following two protocols will reduce the deadline inversion time by deadline inheritance.

B. Inheritance protocols

This subsection presents the Basic Deadline Inheritance (BDI) protocol. In combination with EDF these inheritance protocols maintain constraints as given in section III-D. The main differences with the original protocols are:

1. The original BDI is defined in terms of static priorities, while the new protocols are based on dynamic deadlines.
2. The new task invocations are treated as transactions, introduced in subsection III-A.

B.1 Basic Deadline Inheritance

**Definition 1 (BDI)** BDI is defined by the following rules for the set of runnable transactions:

1. According to EDF the runnable transaction with the shortest deadline is selected to run.
2. Deadline inheritance occurs at the release time of each invocation of a transaction, say \( \tau_a \).
3. Each preempted or running invocation, say \( \tau_b \), that has overlapping resource requirements \((R(\tau_a) \cap R(\tau_b) \neq \emptyset)\) is subject to the inheritance of the deadline of \( \tau_a \).
4. The invocation of \( \tau_b \) is assigned a deadline that is equal to the minimum of the current deadline of \( \tau_a \) and the inherited deadline of \( \tau_a \).

Note that a transaction has an original deadline. In first instance an invocation will run under this deadline unless it inherits a smaller one. The deadline under which an invocation runs determines its preemptability and can be indicated as its preemption deadline. It determines the order in which the invocations are scheduled by the EDF rule. In particular, a shorter preemption deadline reduces the preemptability of an invocation.

Figure 2 shows an example where two transactions require the same resource \( R \). The shortest deadline is shown by the darker shading. At time \( \epsilon_1 + r_e(\tau_1) \) the invocation of transaction \( \tau_2 \) inherits the deadline \( \epsilon_1 + d_e(\tau_1) \) from the invocation of \( \tau_1 \). This is the preemption deadline of \( \tau_2 \) and is shown by a dotted line on the time axis of \( \tau_2 \). Note that the release times \( r_e(\tau_i) \) and deadlines \( d_e(\tau_i) \) are relative to the respective event \( \epsilon_i \). However \( \epsilon_1 + r_e(\tau_1) \) and \( \epsilon_1 + d_e(\tau_1) \) are absolute values.

![Basic deadline inheritance](image)

Note that the combination of EDF and BDI maintains the mutex constraint by the equality constraint automatically: after the deadline inheritance, invocations that have overlapping resource requirements have equal preemption deadlines. This makes the mutex constraint redundant and the EDF/BDI algorithm straightforward and easy to implement: after deadline inheritance the only scheduling and preemption (selection) parameters are the deadlines. There are no additional synchronisation semaphores; inheritance and selection is all there is.

### B.2 Periodic Inheritance approaches

In this section we consider three periodic approaches for transactions with inheritance:

- **Periodic Deadline Inheritance (PDI)**. This approach is similar to BDI, however in PDI transactions are assumed to have a period.
- **Rate Monotonic Inheritance (RMI)**. The priority is assigned inversely proportional to the period which is also the deadline.
- **Deadline Monotonic Inheritance (DMI)**. The priority is assigned to the transaction with the shortest deadline. The latter is smaller or equal than the period.

As already stated in the introduction RM [11] is in fact a special case of DM [1] and consequently we mostly refer to DM. Both approaches are based on periodic tasks. A runtime preemptive scheduling mechanism is used. In RM \( \text{deadline} = \text{period} \) which is weakened in DM to \( \text{deadline} \leq \text{period} \). Moreover we allow \( \text{release-time} \neq 0 \) in DM. These weakenings would benefit the application designer for instance for modeling communication delays as "dead time" after the beginning or before the end of the period. Also sporadic events can be modeled in a more realistic way since their required response times are in general shorter than their worst case arrival period.

An additional assumption in the periodic approaches is that during a period only a single transaction is active. This assumption is not fundamental; it simplifies the model used for QoS feasibility analyses as given in section VII. We expect that the extension from a single transaction to multiple transactions is quite straightforward.

The most important statement of this section is that both PDI and DMI can be scheduled according to the BDI protocol definition from subsection IV-B.1. That PDI can be scheduled with BDI follows immediately from the fact that (1) BDI does not include any information about periods and (2) that the notion of deadline is the same in BDI and PDI.

Similar to the difference between EDF and DM we have a difference in the meaning of deadline between PDI and DMI. By giving the static meaning to deadline in DMI we can keep to the same text as for PDI in subsection IV-B.1. Note however that the implementation is not the same.

PDI and DMI both maintain the mutex constraint by implication from the inheritance rule and the equality constraint. PDI is more dynamic because of its dynamic deadline, hence more adaptive than DMI, which uses a static deadline.

### V. Performance of BDI

In this section we discuss the performance aspects of our kernel as flexibility, fairness, response and overhead. We will neither give a formal description of these aspects nor give a formal proof of our statements. Both are beyond the context of this paper. Below follows an informal justification of our claims about BDI/EDF.
BDI is fair

BDI is fair in the following sense: Due to the application of the EDF rule the scheduler will try to serve the transactions with the shortest deadline best. Resource needs and timing requirements of other transactions can and will be taken into account. By using priority inheritance the kernel avoids priority inversion to its best knowledge. Transitive waiting and consequently deadlocks are impossible due to the introduction of RT transactions.

BDI is flexible

This is mainly due to its dynamic nature. Scheduling is done at runtime and the scheduler adapts to new tasks without any additional effort in a natural way.

BDI is efficient

As already stated in the previous paragraph, this is mainly due to the nature of the RT transaction in combination with BDI/EDF/inheritance. Ordering of transactions, deadline inheritance and selection are all functions needed for scheduling. This combination guarantees mutual exclusion of shared resources. Consequently other synchronisation primitives are not needed anymore! The algorithms require ordering of sleeping - and ready transactions and in addition a $O(n)$ scan over ready transactions for determination of the inherited deadlines. Our first implementation, described in the following section, showed an overhead below 0.5 percent for the executed multimedia experiments.

VI. QoS management

In a multimedia kernel an application tries to establish a contract between several parties, such as a producer, a network and a consumer. The services that have to be supported are in the domain of video, sound, transport and task scheduling. QoS is measured with different parameters in each domain. For video these are for instance frame rates, window surface, resolution and number of colours. For sound the sample frequency, the jitter and the number of bits counts. The network is typically an Asynchronous Transfer Mode (ATM) network [10]. Such a network can give statistical guarantee of bandwidth and it can support RT behaviour. Processing power is supplied under the responsibility of a scheduler. The QoS parameters here are the period, run length, deadline, release time and resource usage. A difficult issue in QoS is that it is not always clear how to relate the QoS parameters of the different domains to each other.

Another possibility that we would like to support is to vary the requested QoS in order to adapt to the latest requirements of an end-user who would like to add, delete or change a multimedia application. In these cases a QoS manager has to arrange the new or changed setup by acting as a dealer for services.

Among others, the task of a QoS manager is to derive QoS specification for the scheduler. These specifications include process period, arrival time, release time, computation time and deadline, thereby taking into account the use of resources. With this information the scheduler could start a schedulability analysis. It might be that the QoS manager is satisfied with an average QoS schedulability analyses of the scheduler. However under circumstances a hard guarantee may be needed. In both cases a positive scheduling analysis from the scheduler to the QoS manager implies that a contract can be concluded.

In the following section we present a method for the determination of scheduling analyses for the DMI and PDI approach.

VII. QoS schedulability analyses

It would be attractive if we had a direct method for schedulability analyses of task sets when using PDI. Mok has shown that the problem of deciding schedulability of a set of periodic tasks with mutex constraints is NP-hard [12]. So we may expect that a direct tractable method could be hard to find. There are however arguments from which we may derive that indirect methods for schedulability analyses could work.

The first argument comes from from Liu and Layland [11]. They state that EDF is optimum in the sense that if a set of tasks can be scheduled by any algorithm, it can be scheduled by EDF. However resources and release times were not taken into account, so their argument can not be applied straightforwardly. The second argument comes from Jeffay. In [7] he states that non-preemptable EDF can schedule any schedulable task set with release times. These two arguments have invoked the following conjecture.

Conjecture

"PDI can schedule any set of transaction that can be scheduled by DMI"

The proof for the conjecture is beyond the scope of this paper. We claim to have found one, however not yet published.

A valid question is: "Why is PDI more interesting than DMI?". The answer is that we expect PDI to behave better on the average and that PDI
might be able to schedule sets that cannot be scheduled by DML. We expect this because, as already stated earlier in IV-B.2, PDI is less static, hence more dynamic and more adaptable than DML. It is also expected that PDI behaves better in case of near- or temporary overload situations.

In the following we present a rapid feasibility analysis for a set of transactions under DMI and according to our conjecture this set can also be scheduled by PDI.

For the QoS feasibility analyses we proceed as follows: We assume a DMI protocol with the following transaction parameters for transaction \( i \): period \( T_i \), run length \( C_i \), deadline \( d_i < T_i \), release time \( r_i \), and blocking time \( b_i \). Recall that blocking occurs if a higher priority transaction waits for a low priority transaction to complete in case of a shared resource. Blocking is limited by virtue of inheritance and by virtue of the nature of the transaction itself: transitive waiting is not possible: in the left hand side of figure 3 \( T_2 \) is supposed to wait for \( T_3 \). This is impossible since \( T_3 \) waits for \( T_1 \) or even preempts it.

![Fig. 3. Transitive and parallel waiting](image)

Note however that blocking in the form of parallel waiting is possible in case a high priority transaction needs resources which are in use by several low priority transactions. An example of transitive and parallel waiting is given in in the right hand side of figure 3. In the parallel waiting case \( T_1 \) is preemted by \( T_3 \). When \( T_3 \) shows up \( T_3 \) has to wait for the release of \( R_1 \) and \( R_2 \) and hence for \( T_1 \) and \( T_2 \).

Our test technique is based on the concept of a critical instant, first introduced by Liu and Layland [11] and further developed by Audsley et. al. [1]. A critical instant represents a time where all transactions are released simultaneously. This implies a worst case load demand for the processor. If all transactions can reach their deadlines from a critical instant, then they will always reach their deadlines. Note that if the release times \( \neq 0 \), the critical instant coincides with simultaneous release of all transactions.

We order \( m \) transactions \( \tau_i \) \( (1 \leq i \leq m) \) according to their priority: \( \tau_1 \) has the shortest deadline and the highest priority. Note that \( T_i > T_j \) is possible for \( i < j \). We will proceed with our test as follows: first we determine the feasibility of \( \tau_1 \). After having determined the feasibility of \( \tau_1 \ldots \tau_i \) we add \( \tau_{i+1} \) and determine the feasibility of \( \tau_1 \ldots \tau_{i+1} \). This process will be continued until infeasibility is found or \( \tau_1 \ldots \tau_m \) is feasible.

Example

First we introduce an example with four transactions \( \tau_1 \ldots \tau_4 \). All release times are 0 and there are no shared resources except the processor. \( \tau_1 \ldots \tau_3 \) are feasible with the following parameters: \( T_1 = 5, C_1 = 1; T_2 = 4, C_2 = 1; T_3 = 6, C_3 = 2 \). We will search for the smallest possible deadline of \( \tau_4 \) with \( C_4 = 1 \). In figure 4 the critical instant is at 0. We start \( \tau_4 \), simultaneously with \( \tau_1 \ldots \tau_3 \). Consequently the demand at \( t = 0 \) is \( C_1 + C_2 + C_3 + C_4 \) and will monotonically increase as shown by the stepped demand graph. The steps are due to the periodical addition of the load by \( \tau_1 \ldots \tau_3 \). The dotted line is the offer graph which shows the load demand a processor can solve by maximum utilisation. The first intersection point of demand and offer graph represents the time at which the processor could solve the load. This point is clearly the minimal deadline for \( \tau_4 \) and \( 10 \leq d_4 \leq 4 \) is a sufficient condition for \( \tau_1 \ldots \tau_4 \) to be feasible.

![Fig. 4. Feasibility of \( \tau_1 \ldots \tau_4 \)](image)
A. Release times

Next we introduce release times \( r_i \neq 0 \) and start again a feasibility test. The critical instant is now the instant at which all the transactions are released simultaneously at their release times. The periodic transactions will contribute in exactly the same way as if all release times were zero. Consequently the minimum deadline, which is the first intersection, is exactly at the same point as in case release times were zero. However this point is now relative to the release time and we have to correct for the deadline relative to the begin of the period. If in the previous example the release time of \( \tau_4 \) were \( r_4 \) then with \( 10 + r_4 \leq d_4 \leq T_4 \) the set \( \{\tau_1 \ldots \tau_4\} \) would be feasible.

The introduction of release times do not really change the nature of the feasibility tests. It only shifts the activity and the deadline to the right with the value of the release time.

B. Blocking

Shared resources may introduce blocking. By nature of the transactions and due to inheritance, a transaction can only be blocked by lower priority transactions. Transitive waiting is not possible, however parallel waiting may occur, as explained in section VII. We investigate the feasibility of a priority ordered set of transactions \( \Theta_i = \{\tau_1 \ldots \tau_m\} \) by iteration: after having determined the feasibility of \( \Theta_i^{-1}(1 \leq i \leq m) \) we add \( \tau_i \) to investigate the feasibility of \( \Theta_i \). During this investigation we need to take into account that \( \tau_i \) can be preempted by the higher priority transactions \( \tau \in \Theta_i^{-1} \). However, due to the use of shared resources, every transaction in \( \Theta_i \) may be blocked for already running lower priority transactions \( \tau \) in \( \Theta_i^{-1} \). In order to investigate the feasibility of the additional \( \tau_i \) we have to take any possible combination of these blockings into account.

Let \( R_i(1 \leq i \leq m) \) denote the set of shared resources used by \( \tau_i \) and \( R_i^m \) the set of resources that may cause blocking of \( \Theta_i \) by \( \Theta_i^{-1} \). Let \( R_i = (R_1 \cup \ldots \cup R_i) \) and \( R_i^m = (R_{i+1} \cup \ldots \cup R_m) \) then \( R_i^m = R_i^m \cap R_{i+1}^m \). Any transaction in \( \Theta_i^{-1} \) that uses resources from \( R_i^m \) may cause blocking. A possible blocking subset is denoted by \( \Theta \) with \( \Theta \subseteq \Theta_i^{-1} \). Let \( R_\Theta \) be the resource usage of \( \Theta \) with \( R_\Theta \subseteq R_i^m \). Transactions in \( \Theta \) are also subject to mutex constraints and any resource from \( R_\Theta \) may be used only once.\(^8\) Let \( C_\Theta \) be the blocking time caused by \( \Theta \). Consequently \( C_\Theta \) is the sum of blocking times of transactions in \( \Theta \). The maximum blocking time is derived from all \( \Theta \) that satisfy (1) thru (3):

\[ C_i^b = \max(C_\Theta: \Theta \subseteq \Theta_i^m \text{ and } R_\Theta \subseteq R_i^m \text{ and mutex(\Theta)}) \]

Determination of the maximum blocking time is in principle not a trivial task. In fact it is NP-hard. We may however expect a limited use of shared resources, making determination possible in a limited amount of time.

Note that we can make refinements to the computation of the maximum blocking time in the following way: We assumed blocking to last as long as the runtime of a transaction. Splitting up a single transaction in a sequence of multiple transactions during one period will introduce shorter transactions and consequently a smaller use of resources simultaneously. This will reduce the computed blocking time.

Example with resources

Before we start a feasibility test we determine \( C_i^b \) for all \( i \) according to the procedure in this section.

Suppose we would like to test the feasibility of \( \Theta_i \) while \( \Theta_{i-1} \) is feasible. If the maximum blocking time associated with \( \Theta_i \) is \( C_i^b \), then we need to add \( C_i^b \) to the load demand at the critical instant. Consequently the demand at the critical instant \( t = 0 \) is \( C_{i-1} + \ldots + C_i + C_i^b \). This lifts the demand graph by \( C_i^b \) and we can find the minimum possible deadline in the same way as in the previous example. Lifting the demand graph will shift the intersection point with the offer graph to the right, which increases the minimum possible deadline. Instead of lifting the demand graph we could also move the offer graph down by the same amount. This is shown in figure 5 for a feasibility test of \( \Theta_i \) from our first example. In this test we introduce the use of a resource \( R \) by \( \tau_3 \) and \( \tau_4 \). This causes a maximum blocking time \( C_3^b \) for the set \( \Theta_3 \) of \( C_4(=1) \). From figure 5 we conclude that \( 7 \leq d_3 \leq T_3 \) is a sufficient condition for the feasibility of \( \Theta_3 \).

C. The QoS feasibility machine

In figure 6 we show a parallel machine with \( m \) parallel (sub)machines \( M_1 \ldots M_m \) that determine the feasibility under DMI of the set of \( m \) transactions in \( \Theta_i^m \) ordered to priority. Each \( M_i \) represents \( \tau_i \) and is fed with \( \tau_i \)'s parameters \( T_i, C_i, d_i \) and \( R_i \).

Each machine \( M_i \) determines the blocking time
$C_i^b$ for $M_1 \ldots M_t$ from $M_{t+1} \ldots M_m$ conform the previous section by exchanging resource needs and runtime information with other machines.

All machines start simultaneously at $t = 0$. Each machine may have three colours; red white or blue. For $\tau_1 \ldots \tau_t$ red $M_i$ means non-feasibility detected, white $M_i$ means feasibility undetermined and blue $M_i$ means feasible. All machines start white.

Each $M_i$ sends at its own frequency $1/T_i$ its load demand $C_i$ to the demand line where it is added to the demand coming from the right. The result is sent to the left. All machines start adding load demand at $t=0$.

Each machine starts the feasibility test from a sign by its left neighbour by adding to its blocking time $C_i^b$ the value of the demand line $l_i$. This value $C_i^b + l_i$ is continuously compared with the time $t$. A machine stops when either (1) $C_i^b + l_i \leq t$ or (2) its deadline is reached ($d_i \leq t$ ). If (1) was the reason to stop, the machine is turned blue (feasible). If (2) was the reason the machine is turned red. If it is turned blue the right neighbour machine is informed of this and the neighbour may start its investigation. If it is turned red the set under investigation is not feasible.

When all machines are blue the set $\tau_1 \ldots \tau_m$ is feasible. Note that the time to complete the feasibility test is proportional to the longest deadline. How long it takes in “real time” depends on the time scale in which our parallel “QoS feasibility machine” can run.

Note that the machine in figure 6 does not take release times into account. According to section VII-A this could however be added easily.

![Fig. 5. Feasibility of $\tau_1 \ldots \tau_3$ under blocking](image)

![Fig. 6. The feasibility machine](image)

### VIII. Implementation and Tests

Scheduling strategies have been simulated in Python [13] first. Then they have been added to the Nemo kernel [4], where they run at the highest priority level. The original scheduler runs in the background.

Two applications have been written to test the scheduler and determine the administrations overhead.

The “oscilloscope” application consists of four tasks that draw horizontal lines on the screen. It displays when the tasks run and how they preempt each other.

The “jumping tea can” application presents four video streams of a bouncing tea can. The stream consists of 25 frames of 256x256 pixel in 256 colours at a rate of 25 frames per second. One task computes the frame number and the position and a second task draws the picture. Four additional empty dummy tasks that are activated every 50 ms increase the workload of the scheduler.

The system is running on a MIPS 3000. Tests have run under BDI. Each invocation requires two task switches (call and return). An empty invocation (two task switches) is timed 3-4 μs. The average computational load of the tasks was 90.31%.

The administration overhead is 0.45% with BDI. This overhead is relatively small due to the small average length of the ready queue (including the running invocation) in our experiments. No fine tuning has been performed. The QoS feasibility machine is currently under construction.

### IX. Conclusions

We have presented a new dynamic, real-time scheduling policy for multimedia purposes, called Periodic Deadline Inheritance. It has been shown that the policy is flexible, fair and efficient and is therefore in particular useful for an open and adaptive QoS management.
The basic ingredients of our policy are: (1) Real Time Transactions, (2) The Earliest Deadline First rule (3) and Deadline Inheritance. It is in particular this combination that (a) limits blocking time according to the best knowledge of the scheduler and (b) avoids transitive waiting for resources and consequently deadlock and (c) offers the flexibility, fairness and efficiency.

QoS schedulability guarantees are derived from the Deadline Monotonic policy. We have presented an QoS feasibility architecture with which “hard real time feasibility” of tasks can be assessed. Based on the prospect that worst case blocking can be determined within reasonable time, the QoS feasibility analyses time is proportional to the longest deadline. The ratio depends on the speed at which an architecture can run and is estimated to be far below 1.

The scheduling overhead of our runtime policy is low. This is due to the orthogonality of the ingredients which enable a systematic implementation. Among others, mutual exclusion is guaranteed by the aforementioned ingredients (1) to (3). No additional synchronisation primitives are needed. Experiments have measured a scheduling overhead of less than 0.5 percent. This is really a low price for the offered services.

Acknowledgments

Acknowledgments are due to Eelco Klaver, Erwin van Eijk, Berend Tel, Paul Sijben and Rob van de Wetering for their contributions during the Wednesday afternoon sessions, and to Pallapa Venkataram for comments on an early version of this manuscript.

References