SOFTWARE VERIFICATION RESEARCH CENTRE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 02-19

Real-Time Scheduling Theory

C. J. Fidge

April 2002

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

Note: Most SVRC technical reports are available via anonymous ftp, from `svrc.it.uq.edu.au` in the directory `/pub/techreports`. Abstracts and compressed postscript files are available via `http://svrc.it.uq.edu.au`

Real-Time Scheduling Theory

C. J. Fidge
Software Verification Research Centre

Real-time, multi-tasking software, such as that used in embedded control systems, is notoriously difficult to develop and maintain. Scheduling theory offers a mathematically-sound way of predicting the timing behaviour of sets of communicating, concurrent tasks, but its principles are often unfamiliar to practising programmers. Here we survey the basic concepts of contemporary schedulability analysis, including preemptive and non-preemptive scheduling policies, shared resource locking protocols, and current directions in multi-processor scheduling theory.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.4.1 [**Operating Systems**]: Process Management—*Scheduling*

General Terms: Theory, Verification

Additional Key Words and Phrases: Scheduling theory, real-time programming

1. INTRODUCTION

Embedded computer systems now control critical functions in transport, emergency services, and defence applications. For instance, military Airborne Mission Computer Systems consist of a network of bus-connected processors interacting with numerous sensors and actuators. The complexity of these systems creates major difficulties for their development, certification, and maintenance [Kristenssen et al. 2001].

Real-time scheduling theory [Buttazzo 1997] offers a way of predicting the timing behaviour of complex multi-tasking computer software. It provides a number of 'schedulability tests' for proving whether a set of concurrent tasks will always meet their deadlines or not. Major improvements have been made to scheduling theory in recent years. The original Rate Monotonic Analysis [Liu and Layland 1973] and the newer Deadline Monotonic Analysis [Audsley et al. 1992] have both been absorbed into the general theory of fixed-priority scheduling [Audsley et al. 1995]. Ways of transferring scheduling theory from academia to industrial practice have also been investigated [Burns and Wellings 1995b].

Here we survey the basic principles of contemporary scheduling theory and illustrate them with worked examples. We consider both preemptive and non-preemptive scheduling policies, the impact of shared resource locking, and recent extensions to the theory to allow for multi-processor analysis. Finally, experiences in practical application of scheduling theory to complex systems are summarised.

Address: Software Verification Research Centre, The University of Queensland, Queensland 4072, Australia. E-mail: `cjf@svrc.uq.edu.au`

2. REAL-TIME COMPUTING

This section briefly reviews some of the general concerns associated with time-critical systems.

'Real time' computing is any situation where

- system correctness depends on the *time* at which results are produced, as well as their logical correctness, and
- the measure of system time is related to events in the *real* external environment [Buttazzo 1997, §1.2.1].

Such software is typically found in process control, railway switching, telecommunications, aerospace and military applications.

However, it is also important to recognise that 'real time' computing is

- *not* just 'fast' computing, since hardware upgrades will not necessarily solve all timing problems,
- *not* just a concern for good 'throughput' or 'average case' performance, since this cannot provide guarantees that *particular* deadlines will be met, and
- *not* just a matter of allowing for 'timeouts', which is mainly a requirement of fault-tolerant programming [Stankovic 1988].

Instead, typical characteristics of real-time systems are that [Buttazzo 1997, §1.2.2]

- they are *reactive* and *interact repeatedly* with their environment,
- they are usually components of some larger application and are thus *embedded* systems which communicate via *hardware interfaces*,
- they are trusted with important functions and thus must be *reliable* and *safe*,
- they must perform multiple actions simultaneously, and be capable of rapidly switching focus in response to events in the environment, and therefore involve a high degree of *concurrency*,
- they must compete for *shared resources*,
- their actions may be triggered *externally* by events in the environment or *internally* by the passage of time,
- they should be *stable* when overloaded by unexpected inputs from the environment, completing important activities in preference to less important ones [Buttazzo 1997, Ch. 8],
- satisfying all these requirements makes them both *large* and *complex*, and
- they are long-lived (often for decades in avionics applications) and thus must be *maintainable* and *extensible*.

Particular timing requirements can be categorised as either soft, firm or hard as shown in Figure 1 [Buttazzo 1997, p. 231]. *Soft* real-time calculations may still produce useful results even after their timing requirement is missed. For example, a program to calculate tomorrow's weather forecast may have a nominal deadline in the early evening. Missing this deadline is not catastrophic, but the value of the calculation decreases as the following day draws nearer, and the result becomes worthless if it is not completed until tomorrow lunchtime. Failure to meet a *firm* real-time requirement is wasteful but not harmful. For instance, a database query becomes valueless after the user's patience is exhausted and he moves on to some

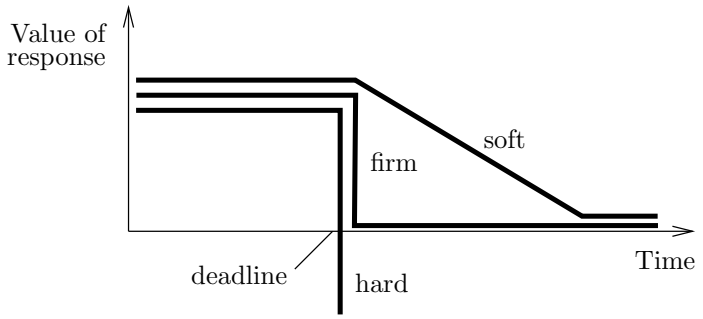

Fig. 1. Degrees of timeliness.

other activity, but producing a late result does no damage. However, failure to meet a *hard* real-time requirement is catastrophic. For instance, most process control applications require outputs to be produced within strict deadlines. A late result is not only worthless, but may have damaging effects. Scheduling theory is largely concerned with the ability to meet such *hard* real-time requirements, and we will confine our interest to them from now on.

Above all else, real-time systems must be *predictable* [Stankovic and Ramamritham 1990]. That is, the finishing times for computations must be guaranteed to always fit within an acceptable range. (This is only possible given known environmental assumptions about worst-case interrupt rates, etc [Buttazzo 1997, §1.3].) Importantly, however, the need for predictability should not be confused with a need for *determinism* [Kurki-Suonio 1994]. This is a stronger requirement in which the system's activity at any moment in time can be known in advance. Fortunately, determinism is not needed to construct predictable real-time systems [Stewart 2001].

3. REAL-TIME SCHEDULING PRINCIPLES

This section introduces the basic features of, and terminology for, real-time multitasking systems as viewed by scheduling theory.

3.1 Abstract Model of a Real-Time System

Scheduling theory assumes that a real-time system consists of the following components [Audsley et al. 1993].

- A set of computational *tasks* to be performed. Typically these are software 'processes' (subroutines with their own thread of control), but other programming constructs that may consume computing resources, such as interrupt handlers, are also modelled as tasks. Each task is assumed to consist of an infinite sequence of identical *invocations*.
- A run-time *scheduler* (or *dispatcher*, *kernel* or *operating system*) which controls which task is executing at any given moment.
- A set of *shared resources* used by the tasks. These may include shared software variables, both with and without mutual exclusion control, and shared hardware devices such as data buses. All communication and synchronisation between tasks is assumed to occur via shared resources.

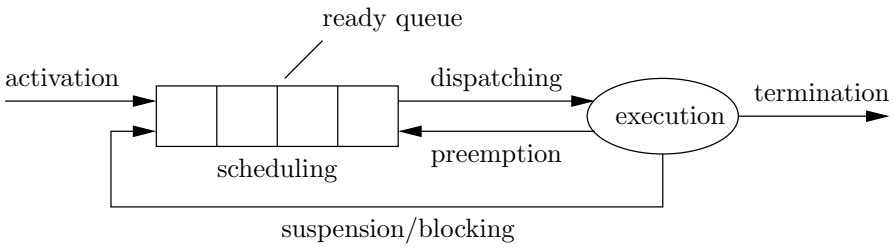

Fig. 2. Lifecycle of a task invocation.

3.2 Lifecycle of a Task Invocation

The lifecycle of each task invocation can be summarised using a conceptual *ready queue* of task invocations waiting to execute (Figure 2) [Taft and Duff 1997, §D.2]. When a task invocation is activated it is placed on the tail of the queue. The queue is ordered according to the particular run-time scheduling policy in force. (This conceptual 'queue' may actually be implemented by a list of queues, one per priority level [Taft and Duff 1997, §D.2.1], or some other such data structure.) The task invocation at the head of the queue, if any, is selected by the dispatcher for execution. The currently executing task invocation can stop executing in three ways.

(1) If it completes its computation it simply terminates.
(2) If the executing task invocation voluntarily suspends itself, e.g., by executing a 'delay' statement [Burns and Wellings 1990, §12.2], or it cannot proceed because its access to some shared resource is blocked [Buttazzo 1997, Ch. 7], it is placed on the tail of the ready queue, to be rescheduled [Taft and Duff 1997, §D.2.2]. (Scheduling theory usually assumes that individual task invocations do *not* voluntarily suspend themselves [Audsley et al. 1995, §2.1], although the parent task suspends itself *between* invocations, as explained in Section 3.4.)
(3) If a preemptive scheduling policy is in force, then the executing task invocation may be preempted by the activation of another task invocation with a higher priority. In this case the preempted task invocation is returned to the *head* of the ready queue [Taft and Duff 1997, §D.2.2]. This ensures that the preempted task invocation will be reinstated as soon as the preempting invocation terminates.

3.3 Types of Tasks

Scheduling theory usually assumes tasks are of three types, characterised by the arrival pattern of their individual invocations.

- *Periodic* tasks consist of an (infinite) sequence of (identical) invocations which arrive at fixed intervals [Buttazzo 1997, Ch. 4]. Their arrival pattern is thus time driven.
- *Aperiodic* tasks consist of a sequence of invocations which arrive randomly, usually in response to some external triggering event [Sprunt et al. 1989, p. 28]. Their arrival pattern is thus event driven.
- *Sporadic* tasks are a special case of aperiodic ones in which there is a known worst-case arrival rate for the task, i.e., they have a fixed minimum interarrival

```
1:  arrival : Time;
        ⋮
2:  arrival := Clock;                -- first arrival time
3:  loop                             -- loop forever
4:    delay until arrival;           -- suspend task until arrival time
5:    'action';                      -- code for task invocation
6:    -- deadline: arrival + D
7:    arrival := arrival + T;        -- determine next arrival time
8:  end loop;
```

Fig. 3. A typical periodic task expressed in Ada.

time [Sprunt et al. 1989, p. 28].

Since aperiodic tasks may arrive with an unknown rapidity, it is impossible to guarantee that any given task set can handle them quickly enough. Hard real-time scheduling theory therefore assumes that task sets consist of periodic and sporadic tasks only. (Some extensions to the theory also allow for hybrid forms of task such as 'sporadically periodic' ones [Audsley et al. 1993] [Tindell et al. 1994].)

Other computational activities are modelled using these task types. In particular, interrupt handlers are modelled as sporadic tasks, with the occurrence of the corresponding interrupt as the triggering event, and scheduler and communication overheads can be modelled explicitly as tasks. Furthermore, because schedulability analysis typically assumes a worst-case scenario in which sporadic tasks arrive as quickly as possible, i.e., once per their minimum interarrival time, the analysis effectively assumes that the whole task set consists of periodic tasks only.

Individual task invocations may be further categorised by their willingness to be preempted while executing [Buttazzo 1997, §2.3.1].

- *Non-preemptive* task invocations execute to completion without interruption once started.
- *Preemptive* task invocations can be temporarily preempted during their execution by the arrival of a higher-priority invocation. (Preemptive task invocations can temporarily prevent themselves from being preempted by locking a shared resource, as explained in Section 4.4.)

3.4 Typical Periodic Task Code

As a concrete illustration of a task, Figure 3 shows the programming language code that preemptive scheduling theory assumes is used to implement a typical periodic task [Stoyenko and Baker 1994, pp. 103–4]. It consists of an infinite loop (lines 3 to 8) where some 'action' is performed at each iteration (line 5). Each execution of this action represents one task invocation. The action itself is application-specific, but typically involves sampling an input, processing data, and/or writing an output.

The remaining statements are used to control the task's periodicity. A time-valued variable *arrival* is declared to hold the (earliest) time at which the next task invocation may begin (line 1). It is initialised with the time the task begins (line 2), or some specific absolute time. Within the loop, the start of each task invocation is then delayed until the time represented by *arrival* (line 4). Note

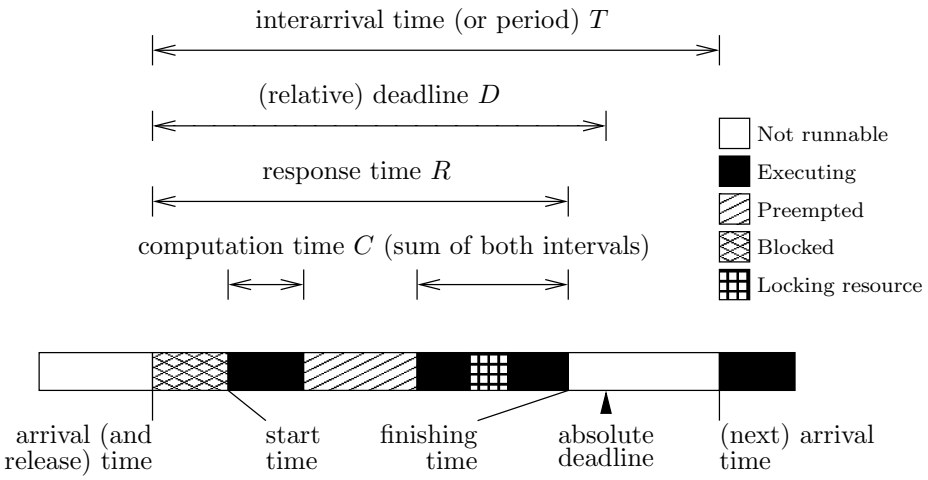

Fig. 4. Anatomy of a typical task invocation.

that the code uses an absolute-time '**delay until**' statement [Taft and Duff 1997, §9.6], rather than a relative-time 'delay' or 'wait' statement, to avoid cumulative drift between the arrival times [Blázquez et al. 1992]. After this, the action is free to occur (when the processor becomes available). Finally, variable *arrival* has the task's period T added to it to determine when the next invocation may begin (line 7).

The comment on line 6 documents the time by which the task invocation must finish, relative to its arrival time. Although the statement on line 4 controls the *earliest* time at which the task invocation may start, contemporary programming languages do not provide 'deadline' statements for controlling the *latest* time by which the invocation must finish [Burns and Wellings 1990, p. 342]. Nevertheless, such requirements should be documented in the code since they are needed to support schedulability analysis and other forms of real-time program verification [Fidge et al. 1999].

A sporadic task can be programmed similarly, except with some form of blocking input statement to await the occurrence of the triggering event, instead of the 'delay until' statement [Stoyenko and Baker 1994, p. 104].

3.5 Anatomy of a Task Invocation

At run time, each invocation of a task involves a number of possible states and significant timing points [Buttazzo 1997, §2.2.1], as illustrated by the example in Figure 4.

Initially the invocation is not ready to run. The earlist time at which it can notionally begin executing is its *arrival time* [Audsley et al. 1993, §2]. The invocation's *release time* is the moment when it is placed in the conceptual ready queue. In basic scheduling models an invocation's arrival time and release time are the same (but see Section 4.6). The duration between the arrival time and the time at which the next invocation of this task may arrive is the *minimum interarrival time*, T. For periodic tasks duration T is called the task's *period*.

Having arrived, the task invocation may start executing (when it reaches the head

of the conceptual ready queue). However, due to competition for the processor and other shared resources this may not happen immediately. In Figure 4 the invocation's *start time* is delayed because the task is initially *blocked* by a lower-priority task (Section 4.4).

Once started, the task invocation will execute until its computation is completed at its *finishing time*. A non-preemptible task invocation will execute without interruption. However, for a preemptible task, as shown in Figure 4, the task invocation may be temporarily *preempted* by a higher-priority invocation. The sum of the intervals during which the task invocation is executing, excluding blocking and preemption, is its *computation time*, C. While executing, the task may also lock shared resources (Section 4.4) which may impact upon the progress of other tasks.

As well as its period, the programmer must supply a *deadline*, D, for each task, within which all of its invocations must finish executing. The deadline is specified relative to the arrival time of the task invocation [Tindell et al. 1994, p. 149], and thus defines the corresponding *absolute deadline*. If the relative deadline for a periodic task is left unstated it is usually assumed to equal the task's period. The purpose of the deadline is to constrain the acceptable finishing times for the task invocation. The duration between the task invocation's arrival and finishing times (including intervals of blocking and preemption) is called the invocation's *response time*, R, which is required not to exceed the relative deadline D. Variability in the task's response time from one invocation to the next is known as (finishing time) *jitter* [Giering III and Baker 1994, p. 55].

3.6 A Taxonomy of Scheduling Algorithms

Numerous algorithms for scheduling real-time tasks exist. Broadly speaking, however, they can be categorised as follows [Buttazzo 1997, §2.3.1].

- *Static* scheduling algorithms require the programmer to define the entire schedule prior to execution. At run time this pre-determined schedule is then used to guide a simple task dispatcher. *Cyclic executives* are one way to program static task scheduling [Burns and Wellings 1990, pp. 352–354].
- *Dynamic* scheduling algorithms make decisions about which task to execute at run time, based on the priorities of the task invocations in the ready queue. They require a more complex run-time dispatcher or scheduler. Such algorithms can be further categorised into those based on fixed and changeable priorities.
  - *Fixed-priority* scheduling algorithms statically associate a priority with each task in advance. This can be done arbitrarily by the programmer, or according to some consistent policy. Two well-known policies for fixed priority assignment are *Deadline Monotonic Scheduling*, in which tasks with shorter deadlines are allocated higher priorities [Tindell 2000], and *Rate Monotonic Scheduling*, in which tasks with shorter periods are allocated higher priorities [Briand and Roy 1999].
  - *Dynamic-priority* scheduling algorithms determine the priorities of each task invocation at run time. Typically this requires a more complex run-time scheduler than fixed-priority scheduling. Two methods of dynamic priority assignment are *Earliest Deadline First*, in which the ready task invocation with the earlist upcoming deadline is given highest priority [Liu and Layland 1973], and *Least Laxity*, in which the ready task invocation

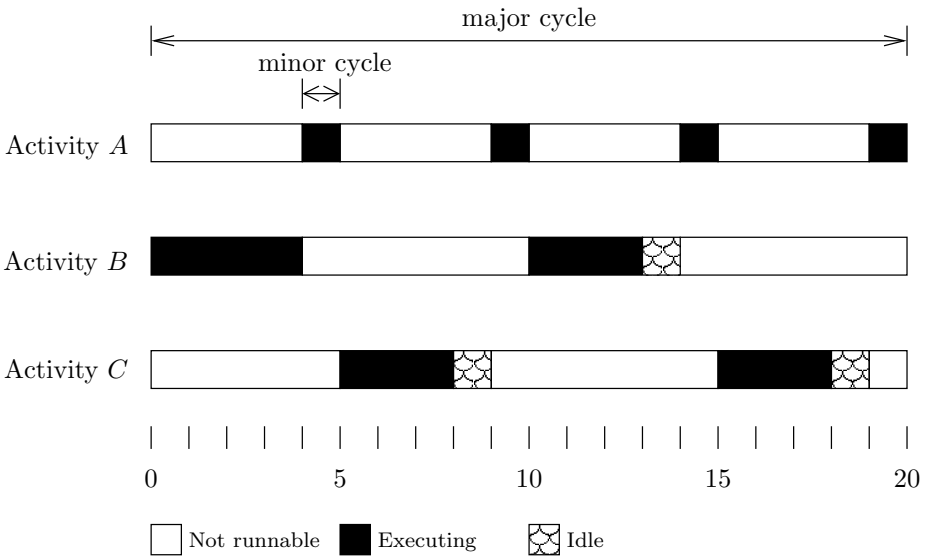

Fig. 5. A static schedule.

with the smallest difference between its upcoming deadline and (estimated) remaining computation time is given highest priority [Jones et al. 1996].

3.7 Task Scheduling Examples

To illustrate the difference between the scheduling policies mentioned in Section 3.6, this section presents four different ways of scheduling sets of computational requirements, and summarises their advantages and disadvantages.

3.7.1 *Static Scheduling.* Consider three distinct activities, A, B and C, which involve monitoring and responding to external events. Their rates of occurrence are determined by the need to respond to such events in a timely manner. In a 20 millisecond interval, activity A must occur four times and requires up to 1 millisecond of processor time at each occurrence. In the same interval activity B must occur once and requires 7 milliseconds of processor time, and activity C must occur twice and requires 3 milliseconds of processor time at each occurrence.

To meet these requirements using static scheduling, the programmer is obliged to allocate activities to each unit of processor time [Burns and Wellings 1990, pp. 352–354]. One such static schedule is shown in Figure 5. Having devised this schedule, the programmer would then encode it as a *cyclic executive* program which iterates every 20 milliseconds and in each iteration performs the three activities in the sequence shown in the figure [Kalinsky 2001] [Tindell 2000, p. 21] [Burns and Wellings 1990, p. 178]. In this case, the program has a *major cycle* of 20 milliseconds, after which the whole schedule repeats, and a *minor cycle* of 1 millsecond, into which each occurrence of an activity must fit [Locke 1992, §2].

Static scheduling is a simple approach that has a number of advantages.

+ It produces programs that are entirely deterministic. It is possible to know which task is executing at any given time.

+ It does not require a run-time operating system to schedule the activities [Locke 1992, §2.1]. The interleaving of activities is 'hardwired' into the application program's code.
+ For control-system applications it exhibits low *jitter*, i.e., the separate occurrences of each activity are always evenly spaced in time [Locke 1992, p. 43].

However, despite these advantages, static scheduling also has numerous drawbacks.

− Static schedules are rigid and difficult to construct [Kalinsky 2001]. They provide determinism (i.e., the ability to know at every instant which activity is executing) whereas only predictability (i.e., the ability to know that activities will finish their computations in time) is required in a real-time system [Locke 1992, p. 45].
− The approach supports periodic activities only. Responding to infrequent events must be achieved by 'polling' for their occurrence. Such programs can be computationally inefficient [Tindell 2000, p. 21]. For instance, activity A was scheduled frequently in Figure 5, on the assumption that it must respond rapidly to its triggering event. If, however, this event occurs infrequently, then most of A's occurrences will do nothing [Burns and Wellings 1990, p. 179].
− Another potential source of inefficiency is the need to fit all activities into common multiples of the major and minor cycles [Tindell 2000, p. 21]. In the schedule devised in Figure 5, some cycles were left idle because the activities did not fit neatly.
− If an activity does not fit exactly into the schedule, it may be necessary to rewrite its code to make it fit [Tindell 2000, p. 22]. In Figure 5, for instance, the occurrence of activity B, which required 7 milliseconds, was split into two parts of duration 4 and 3 milliseconds, respectively. The programmer must ensure that the activity's state is properly saved at the end of the first part and reloaded at the beginning of the second. Such code is awkward to develop and expensive to maintain—any changes to the schedule may mean rewriting each activity's code.
− The cyclic executive program which implements a static schedule is inherently monolithic. In it, code from unrelated activities is mixed together—the program's layout is unrelated to the application's structure. This makes the program difficult to design, prove correct, and maintain [Burns and Wellings 1990, p. 179]. (Although the concurrency constructs of a programming language like Ada can be used to build static schedules [Baker and Shaw 1989], this is still awkward and not an efficient use of these constructs.)
− Cyclic executive programs are unstable when an activity overruns its allotted processor time [Locke 1992, p. 42], e.g., due to the late arrival of some essential data from the environment. In this case, whichever activity appears next in the schedule will suffer the consequences of the overrun, regardless of its importance to the overall application. Thus, the failure of an activity of low importance can have a negative impact on a highly important one.

For these reasons, static scheduling is no longer advocated as a way of programming embedded real-time systems [Locke 1992]. Multi-tasking solutions instead support

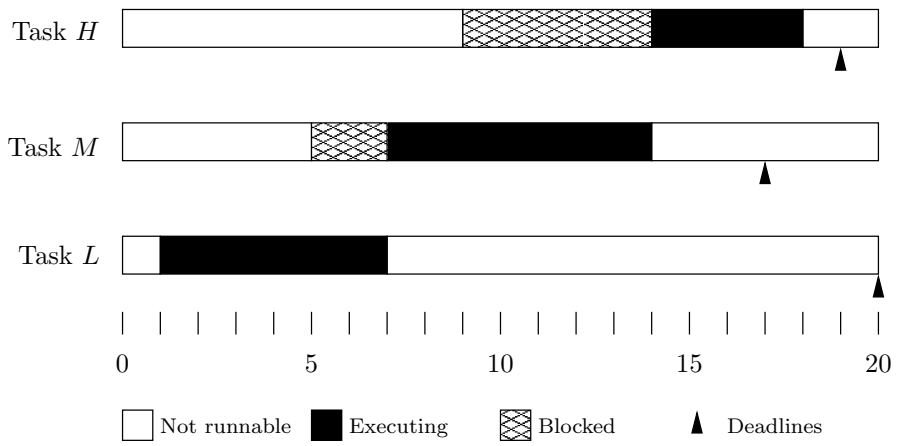

Fig. 6. Non-preemptive fixed-priority scheduling.

separation of activities into distinct tasks, which can be developed and maintained in isolation, while still supporting timing predictability of the whole task set, thanks to the schedulability testing principles described in Section 4 below.

3.7.2 *Non-Preemptive Fixed-Priority Scheduling.* Consider three distinct tasks, H, M, and L, with programmer-allocated priorities of high, medium and low, respectively. Following its arrival an invocation of task H requires up to 4 milliseconds of processor time and has a deadline of 10 milliseconds (relative to its arrival). Upon arrival an invocation of task M requires up to 7 milliseconds of processor time and has a deadline of 12 milliseconds. Task L's invocations require up to 6 milliseconds and have a deadline of 19 milliseconds. (This set of tasking requirements is not meant to be directly comparable to the set of activities in Section 3.7.1.) For the purposes of illustration, assume that an invocation of task L arrives at time 1, an invocation of task M arrives at time 5, and an invocation of task H arrives at time 9.

Under these cirumstances, Figure 6 shows how these three task invocations would be executed under a non-preemptive, fixed-priority scheduling policy. At time 1 task L's invocation arrives and begins executing. Task M's invocation arrives at time 5 but must wait until time 7, when task L's invocation finishes, before it can start. Similarly, although an invocation of task H arrives at time 9, it must wait until task M's invocation finishes at time 14 before it starts. Despite these delays, all three task invocations meet their respective deadlines.

Non-preemptive scheduling has several distinct advantages.

+ It offers a simple programming paradigm which can be implemented in a programming language such as Ada using a small number of basic tasking features [Burns and Wellings 1996b]. This is achieved as a form of coroutining, in which each task executes without interruption until it voluntarily suspends itself and returns control to the dispatcher.
+ Unlike static scheduling, a multi-tasking program allows individual tasks to be programmed separately, and the overall task set can be defined to mirror the structure of the particular application.

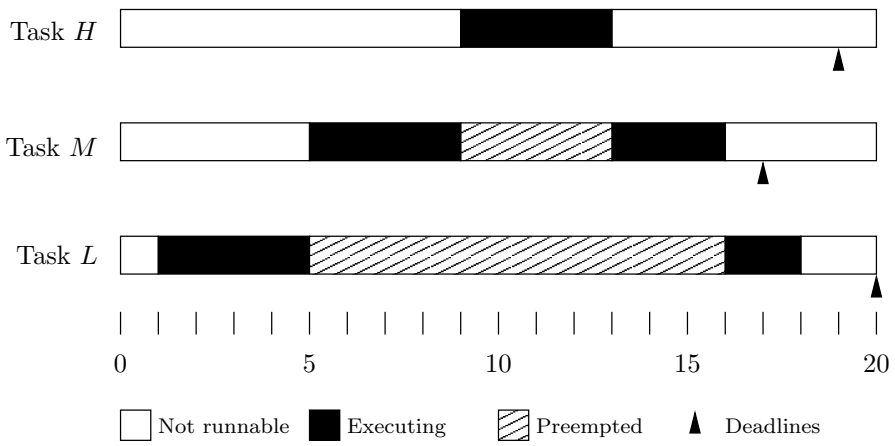

Fig. 7. Preemptive fixed-priority scheduling.

+ Task sets implemented under a fixed-priority scheduling policy can be formally analysed for schedulability (Section 4).
+ It has been suggested [Burns and Wellings 1996b, §2] that the simple behaviour of such systems is compatible with the Level A (uppermost) software requirements of the DO-178B standard for airborne systems certification [RTCA, Inc. 1992].

The principal disadvantages of non-preemptive scheduling are as follows.

− It is unresponsive to changes in the set of ready tasks or to inputs from the environment, due to the inability to stop a task invocation once it starts. This can unnecessarily delay high-priority tasks. In Figure 6, for instance, task H's invocation was forced to wait for a considerable interval following its arrival for the invocation of lower-priority task M to finish.
− More seriously, if a fault occurs, a wayward task invocation can monopolise the processor indefinitely. This also means that an overrunning low-priority task can delay a higher-priority one, so a non-preemptive schedule is unstable when overloaded.

3.7.3 *Preemptive Fixed-Priority Scheduling.* Consider the same task set described in Section 3.7.2. Figure 7 shows how it would behave under a preemptive, fixed-priority scheduling policy. Again an invocation of task L arrives at time 1 and begins executing immediately. However, when the invocation of task M arrives at time 5 this newly-arrived invocation has higher priority than the executing one so the run-time scheduler preempts task L's invocation and starts task M's. Similarly, when task H's invocation arrives at time 9 it preempts the invocation of task M. Once task H's invocation finishes, at time 13, the preempted invocation of task M can resume execution. Similarly, task L's invocation can continue to completion after task M's invocation finishes.

Preemptive scheduling offers several significant advantages.

+ It provides faster service to higher-priority tasks [Kalinsky 2001]. Compared to the nonpreemptive schedule in Figure 6, the preemptive schedule in Figure 7

significantly reduces the response time for task H's invocation (at the expense of the lower-priority task invocations). Preemptive scheduling thus respects the priority ordering defined by the programmer.

+ The application programmer does not need to do anything to achieve context switches. The run-time operating system determines when context switches must occur and effects them. Thus the application program code is simpler and easier to maintain.
+ Unlike static scheduling, there is no requirement in preemptive scheduling that task periods must be based on harmonic values [Tindell 2000]. Nor is there any required relationship between a task's timing characteristics and its priority. (Deadline and rate monotonic priority allocations can be used if such a relationship is desired, but this is not necessary [Audsley et al. 1995].) In fixed-priority preemptive scheduling the programmer is free to choose any combinations of task priorities, periods and deadlines.
+ Numerous commercial Real-Time Operating Systems are now available to support preemptive scheduling, many of which have been proven highly trustworthy and acceptably efficient [Embedded Systems Programming 2001, pp. 28–51].
+ Schedulability tests for proving that a given preemptive task set will always meet all its deadlines are now well established, as explained below (Section 4).
+ Under transient overloads, preemptively scheduled systems are stable. That is, the lowest priority task invocations will miss their deadlines, rather than higher-priority ones. In particular, both deadline and rate monotonic priority allocations for sets of independent tasks are stable when overloaded [Gomaa 1993, p. 124] [Audsley et al. 1992].

However, fixed-priority preemptive scheduling has some disadvantages for embedded control systems.

− A complex run-time operating system is required to support preemptive scheduling [Kalinsky 2001]. Context switching may take a significant time (this overhead was ignored in Figure 7), and a significant amount of memory may be required to store the state of preempted task invocations.
− Preemptive scheduling can introduce higher degrees of jitter than a static schedule. For instance, the finishing time of invocations of task L above, relative to its arrival time, can vary widely depending on whether the invocation is preempted or not. (Jitter can be minimised by appropriate structuring of the task set [Locke 1992, §3.2], or by introducing offsets [Bate and Burns 1999, §3.2], however.)
− It is not clear how a preemptively scheduled system can be certified safe using existing guidelines such as avionics standard DO-178B [RTCA, Inc. 1992]. Such standards often require exhaustive testing of all possible control-flow paths through a program. This requirement is inappropriate for a preemptive system in which control flow between tasks is determined dynamically at run time. (Instead, reliance should be put on scheduling theory to help prove timing correctness, rather than just testing.)

An additional disadvantage perceived by some programmers of critical systems is the fact that preemptive scheduling does not offer the same degree of determinism

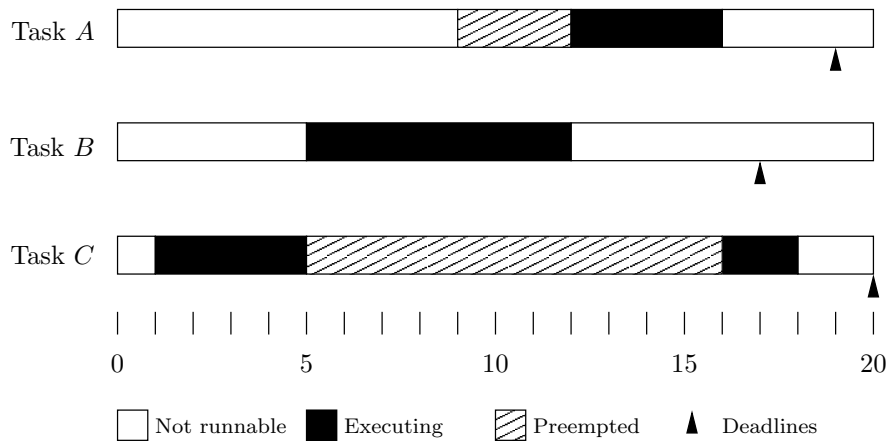

Fig. 8. Preemptive dynamic-priority scheduling.

as static scheduling. Nevertheless, determinism is not normally required in practice, only predictablility [Locke 1992, p. 45], and this can be guaranteed using scheduling theory.

3.7.4 *Preemptive Dynamic-Priority Scheduling.* Again, consider the same task set described in Section 3.7.2, except that here we label the tasks A, B and C, since they do not have statically-associated priorities. Figure 8 shows how these tasks would behave under a particular dynamic-priority scheduling policy, Earliest Deadline First. When task B's invocation arrives at time 5 the run-time scheduler compares its absolute deadline of 17 with the deadline of 20 for the currently-executing invocation of task C, and switches control to the invocation with the earlier deadline. When the invocation of task A arrives at time 9, its later deadline of 19 means that the invocation of task B is not preempted. At time 12 task B's invocation finishes and the run-time scheduler starts the invocation with the earliest deadline, in this case task A's.

The major advantage of dynamic-priority preemptive scheduling is as follows.

+ In theory, ignoring implementation overheads, it offers the best possible guarantee that tasks will meet their deadlines, since it always ensures that processing power is dedicated to the task invocation with the most urgent need [Liu and Layland 1973, p. 186] [Giering III and Baker 1994, p. 55].

However, dynamic-priority preemptive scheduling has the following disadvantages.

– It incurs even higher run-time overheads than fixed-priority scheduling, since the priorities of all tasks in the ready queue must be recalculated each time a task arrives or finishes.
– It is unstable under transient overload in the sense that it is difficult to predict which task will miss its deadline first.

These practical disadvantages tend to outweigh dynamic-priority scheduling's theoretical advantages [Liu and Layland 1973] [Jones et al. 1996] and it is thus less common in practice than fixed-priority scheduling.

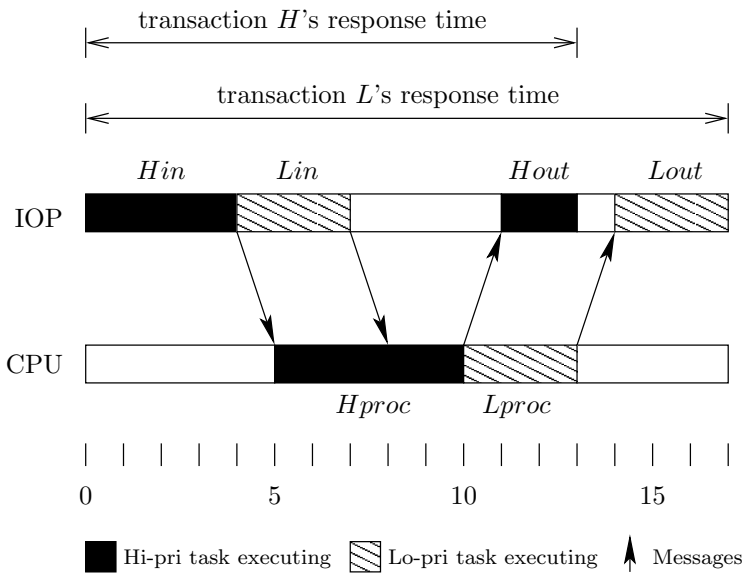

Fig. 9. Scheduling of multi-processor transactions.

3.8 Multi-Processor Scheduling

So far we have assumed that all tasks reside on the same processor. In multi-processor architectures and distributed systems, however, tasks may reside on different machines and communicate using asynchronous message passing or via data buses. This introduces several new features to the computational model [Grigg and Audsley 1999].

Figure 9 illustrates some of the features of a multi-processor, multi-tasking system. In this case we have shown timelines for two physical machines, an Input/Output Processor and a general-purpose Central Processing Unit. Six distinct tasks have been allocated to these two machines. These tasks are grouped to form two *transactions*, i.e., collections of related tasks. Here, both transactions have the same overall form. For instance, the high-priority transaction H consists of an input task Hin, a processing task $Hproc$, and an output task $Hout$. When an invocation of input task Hin finishes, it sends its data to the corresponding invocation of processing task $Hproc$ which, in turn, sends its results to output task $Hout$. The later invocations in this ordering cannot start executing until their immediate predecessor has finished. Such synchronisation requirements form *precedence constraints* between the tasks comprising a transaction.

Whole transactions can be viewed similarly to individual tasks. Like tasks they may have (overall) periods, (finishing time) jitter, (end-to-end) deadlines and (end-to-end) response times which must not exceed their deadlines [Grigg and Audsley 1999, §5.2]. In Figure 9, for instance, we assume that the two transactions both have overall periods of 20 milliseconds.

Figure 9 shows a particular behaviour of these two transactions. All six task invocations arrive simultaneously at time 0. However, only the invocation of task Hin starts executing at this time. It preempts any low-priority invocations on its

processor, and the other tasks in transaction H cannot begin until it has finished due to their precedence constraints. At time 4 Hin's invocation finishes and sends its data to the other processor. We assume in Figure 9 that all inter-processor communication takes 1 millisecond. This means that the invocation of task $HProc$ can start executing at time 5. Similarly, task $Hout$'s invocation starts executing at time 11, once it has received a message from $HProc$'s invocation.

Low-priority transaction L's behaviour in Figure 9 is similar to that of the high-priority one, except that it suffers interference from transaction H. For instance, the invocation of task Lin cannot begin until time 4, due to preemption from task Hin's invocation. Also, even though $Lproc$'s invocation has its precedence constraint satisfied at time 8, by the arrival of the message from Lin's invocation, it must still wait for the invocation of task $Hproc$ to finish. Thus schedulability analysis for tasks in multi-processor systems is especially complex because it depends not only on those tasks that reside on the same processor, but also on the arrival of messages from tasks on other processors.

The run-time sequencing of the nth and $(n+1)$th tasks in a transaction's precedence ordering can be enforced in three different ways. Recall that all task invocations comprising a particular transaction invocation are deemed to arrive simultaneously.

(1) If both tasks reside on the same processor then the $(n+1)$th task can simply be given a lower priority than the nth to achieve the necessary execution ordering.
(2) If the tasks reside on different processors then the ordering can be enforced by appropriate communication mechanisms. In a distributed system, message passing would be used and the nth task's invocation will finish by sending a message to the $(n+1)$th, and the $(n+1)$th task's invocation will begin with a (blocking) input action which awaits the message. In a multi-processor architecture, communication over a data bus or a system-wide interrupt can be used to signal completion of a task invocation between machines [Falardeau 1994, §4.1.3.1].
(3) Alternatively, precedence constraints across processors can be enforced by the use of *offsets* which introduce a constant shift of a task's arrival times from whole multiples of its period [Bate and Burns 1999]. The $(n+1)$th task invocation in a precedence ordering is given an offset at least as great as the worst-case response time of the nth. This not only ensures that the precedence ordering is respected, but it also allows the earlier task's invocation to communicate with the later one merely by writing to a shared data area, confident that the later task's invocation will not attempt to read the data too early. (However, using offsets is possible only if there is tight synchronisation of clocks on the different processors [Falardeau 1994, p. 68].)

Typically, therefore, the first task in a transaction will arrive periodically, whereas the starting times of later task invocations in the precedence ordering depend on the response times of earlier ones [Falardeau 1994, §3.4.3]. Unfortunately, this means that the (starting and finishing time) jitter of tasks [Grigg and Audsley 1999] is cumulative along the precedence ordering. Thus the response-time variability of successive invocations of task $Hout$ from Figure 9 could be very high. Since high jitter is usually unacceptable in control systems [Bate and Burns 1999], a way is needed to reduce it for multi-processor systems. This can usually be achieved by

Γ	The set of task identifiers	
n	The number of tasks in set Γ	
$hp(i)$	The set of tasks with higher priority than task i	
$hep(i)$	The set of tasks with higher or equal priority to task i	
$lp(i)$	The set of tasks with lower priority than task i	
$pre(i)$	The set of tasks preceding task i in a transaction	
$locks(k, i)$	The set of resources used by task k with ceiling priority i or higher	
T_i	Minimum interarrival time, or period	Given for each task i
D_i	Deadline (relative to arrival)	
C_i	Worst-case computation time	
B_i	Worst-case blocking time	
O_i	Offset from arrival time	
$c_{i,S}$	Worst-case computation time while locking resource S	
R_i	Worst-case response time relative to arrival	Calculated for each task i
r_i	Worst-case response time relative to release	
I_i	Worst-case interference	
J_i	Worst-case release jitter	
$W_i(t)$	Processor workload at time t	

Table 1. Symbols used in schedulability equations.

introducing offsets into the task set [Tindell and Clark 1994]. For instance, if we assume that Figure 9 is the worst-case behaviour for invocations of these tasks, then we could define an offset of 11 milliseconds for task $Hout$. Thus its first invocation would arrive at time 11, its second at time 31 and so on. This means that even if invocations of tasks Hin and $Hproc$ finish earlier than the times shown in Figure 9, the invocation of task $Hout$ will still wait until (at least) time 11 to start executing. Although this may mean that the processor may be idle more often than necessary, successive invocations of task $Hout$ will remain evenly spaced, as is usually required for input/output events in process control applications.

4. REAL-TIME SCHEDULABILITY TESTS

Although the computational model described above is in itself an aid to understanding the dynamic behaviour of a real-time multi-tasking system, the true value of scheduling theory is in its *schedulability tests* which accurately predict whether a task set will always meet its deadlines or not.

Dozens of schedulability tests have been developed for real-time multi-tasking systems [Fidge 1998] but, generally speaking, they are all variants of, or derived from, three basic approaches: processor *utilisation*, processor *workloads*, and *response time* analysis (Sections 4.1, 4.2 and 4.3, respectively). Furthermore, processor utilisation measurement is effectively obsolete, having been superseded by processor workload analysis, and workload analysis itself has proven to be equivalent to general response time analysis [Audsley et al. 1995, p. 182]. Therefore, although we briefly present processor utilisation and workload equations in Sections 4.1 and 4.2 below, due to their historical significance, the major principles of schedulability analysis are all covered by the response time tests detailed in Sections 4.3 to 4.8. The symbols used in the equations are shown in Table 1.

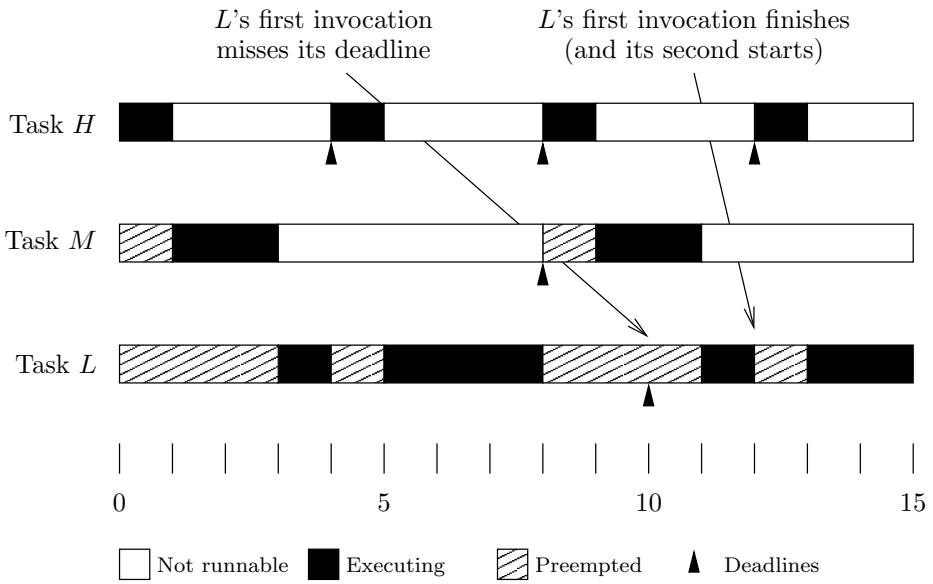

Fig. 10. Execution of a non-schedulable rate monotonic task set.

4.1 Processor Utilisation Analysis

Consider the following set of tasks. Their periods and worst-case computations are as shown, and their deadlines are assumed to equal their periods. A rate monotonic priority ordering has been assumed, i.e., the tasks with the shorter periods are given higher priority.

	T_i	C_i
Task H	4	1
Task M	8	2
Task L	10	5

The percentage of processor time required for each such task is simply its computation time divided by its period. It is therefore tempting to assume that if the sum of these processor utilisations does not exceed 100% then the task set is schedulable [Briand and Roy 1999, p. 23].

$$\sum_{1 \leqslant i \leqslant n} \frac{C_i}{T_i} \leqslant 1$$

Indeed, the above task set passes this 'test', and therefore seems schedulable.

$$\frac{1}{4} + \frac{2}{8} + \frac{5}{10} = 1$$

Sadly, however, as shown by Figure 10, this conclusion is wrong. Task L's first invocation misses its first deadline due to the preemptions it suffers from the higher-priority tasks. The proposed schedulability test is inadequate because it fails to properly account for the phasing of the task's periods.

To correct this, the following schedulability test [Liu and Layland 1973] was devised to account for any possible task phasing.

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

The term on the right is a *processor utilisation bound*, expressed in terms of the number n of tasks. For three tasks this bound is approximately 0.78. Since the total processor utilisation for our three tasks exceeds this, test 1 correctly determines that the task set above is not schedulable.

Unfortunately, schedulability test 1 has many limitations [Briand and Roy 1999, §2.3].

- It applies to preemptive task sets only.
- It applies to rate monotonic task sets only. Tasks whose importance is not proportional to their period cannot be accommodated.
- It applies to independent, non-interacting tasks only.
- All task deadlines are assumed to equal their periods. This is not acceptable for most control applications since it allows tasks to exhibit high jitter (Section 4.7).
- The processor utilisation bound rapidly converges to 0.69 as the number of tasks increases. For large task sets, this means that the test can confirm schedulability only for cases of low overall processor utilisation.

Therefore, despite its historical importance, this test has been superseded by the more general tests below.

4.2 Processor Workload Analysis

Schedulability test 1 is extremely pessimistic. (Although all schedulability tests err on the side of pessimism. It is better to reject a schedulable task set than to pass an unschedulable one.) To overcome this, the following equation instead calculates the total computational *workload* on the processor at some time t, due to invocations of tasks of priority equal to or higher than that of task i [Lehoczky et al. 1989].

$$W_i(t) = \sum_{j \in hep(i)} \left\lceil \frac{t}{T_j} \right\rceil C_j \tag{2}$$

For some task j, expression $\lceil t/T_j \rceil$ is the number of times this task can arrive in an interval of duration t. (The rounding-up operator $\lceil x \rceil$ returns the smallest integer not less than x.) Multiplying the number of arrivals by C_j thus defines the total computational requirement by all invocations of task j in this interval. The total workload at time t is then just the sum for all such tasks.

To determine whether task i is schedulable, the following test then checks that the computational workload associated with this task and those of higher priority is not always greater than 100% in an interval bounded by task i's period T_i [Lehoczky et al. 1989].

$$\min_{0 < t \leqslant T_i} \left(\frac{W_i(t)}{t} \right) \leqslant 1 \tag{3}$$

The test relies on the observation that the worst-case scenario for task i is at time 0, when it arrives simultaneously with all other tasks. If the first invocation of task i

can finish before its deadline at time T_i, then all subsequent invocations will also be schedulable. The test is computationally expensive due to the need to iterate over each time t, but the same result can be achieved by checking just those times at which higher-priority tasks arrive [Lehoczky et al. 1989].

To see if task L from Section 4.1 is schedulable, we can calculate the total workload for the three tasks at each time t, when a higher-priority task arrives during task L's period, using equation 2, and divide each result by t.

$$\frac{W_L(4)}{4} = \left(\left\lceil \frac{4}{10} \right\rceil \cdot 5 + \left\lceil \frac{4}{8} \right\rceil \cdot 2 + \left\lceil \frac{4}{4} \right\rceil \cdot 1 \right) /4 = 2$$

$$\frac{W_L(8)}{8} = \left(\left\lceil \frac{8}{10} \right\rceil \cdot 5 + \left\lceil \frac{8}{8} \right\rceil \cdot 2 + \left\lceil \frac{8}{4} \right\rceil \cdot 1 \right) /8 = \frac{9}{8}$$

$$\frac{W_L(10)}{10} = \left(\left\lceil \frac{10}{10} \right\rceil \cdot 5 + \left\lceil \frac{10}{8} \right\rceil \cdot 2 + \left\lceil \frac{10}{4} \right\rceil \cdot 1 \right) /10 = \frac{6}{5}$$

The minimum of these values is $\frac{9}{8}$, which exceeds 1, so test 3 confirms that the task set in Section 4.1 is unschedulable, because the processor is always overloaded.

Again, however, this test is limited to rate-monotonic, preemptive, periodic, non-interacting tasks, with deadlines equal to their periods. The response time tests used below overcome all of these restrictions [Audsley et al. 1995, p. 182] (as do further extensions to workload analysis [Briand and Roy 1999]).

4.3 Basic Response Time Analysis

Both the processor utilisation and workload tests described above suffer from numerous restrictions. By contrast, the response time test described below is applicable to any fixed priority, preemptive task set, and in subsequent sections it is extended for non-preemptive scheduling and interaction between tasks.

Since the goal of schedulability analysis is to show that each task invocation finishes before its deadline, the approach taken here is to simply calculate the worst case response time R_i for each invocation of a task i and compare it with the deadline D_i. Task i is schedulable if it passes the following trivial test [Joseph and Pandya 1986].

$$R_i \leqslant D_i \tag{4}$$

Of course, the challenge in using test 4 is to determine the response time. All of the equations below are founded on increasingly more sophisticated ways of determining this value.

Firstly, we consider the case of non-interacting tasks. In this situation the response time R_i for an invocation of task i is its own worst-case computation time C_i plus the interference I_i it suffers from higher-priority tasks.

$$R_i = C_i + I_i \tag{5}$$

(Recall that each C_i must include the run-time overheads associated with scheduling an invocation of task i. Also see Section 4.6.)

The interference term in equation 5 depends on how often higher-priority task invocations can preempt an invocation of task i. This can be determined, for each higher-priority task j, by dividing the interval of interest, R_i, by task j's period T_j,

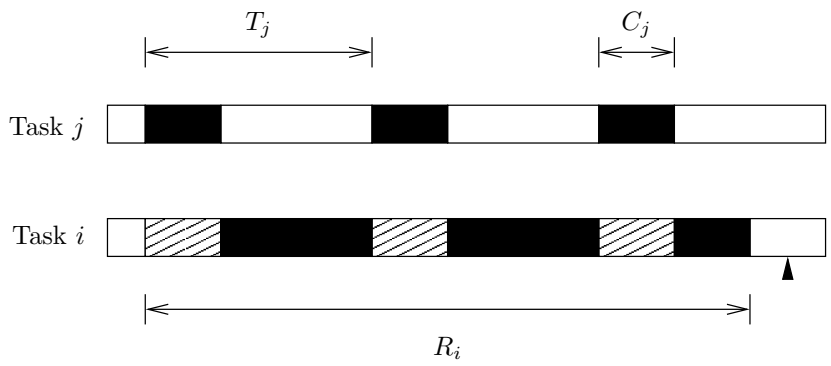

Fig. 11. Interference from higher-priority task invocations.

to obtain the number of arrivals of invocations of task j, and then multiplying this by task j's computation time C_j [Joseph and Pandya 1986]. For now we assume that all task priorities are unique.

$$I_i \;=\; \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{6}$$

For instance, Figure 11 shows three invocations of a high-priority task j preempting one invocation of lower-priority task i. Thus the total interference inflicted by task j on task i is 3 times C_j.

Combining equations 5 and 6 then gives us the total calculation for the worst-case response time of a task i in a set of preemptive, fixed-priority, non-interacting tasks [Joseph and Pandya 1986].

$$R_i \;=\; C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{7}$$

Note that this equation works for any fixed-priority task set, including those designed with either deadline or rate monotonic priority allocations, assuming unique priorities.

However, because the interference suffered by an invocation of task i increases its overall response time, and equation 6 calculates the total interference in terms of the response time, equation 7 is defined recursively. Term R_i appears on both sides of the equality. Fortunately, the equation can be solved iteratively [Audsley et al. 1993]. We begin with an initial approximation for R_i of 0. Then the $(x+1)$th approximation can be defined in terms of the xth one.

$$R_i^{x+1} \;=\; C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^x}{T_j} \right\rceil C_j$$

Performing this calculation will result in one of two outcomes. If R_i^x converges to a value no greater than D_i then we may conclude that task i is schedulable. Alternatively, if R_i^x grows to a value exceeding D_i then task i is not schedulable.

For example, consider the following task set.

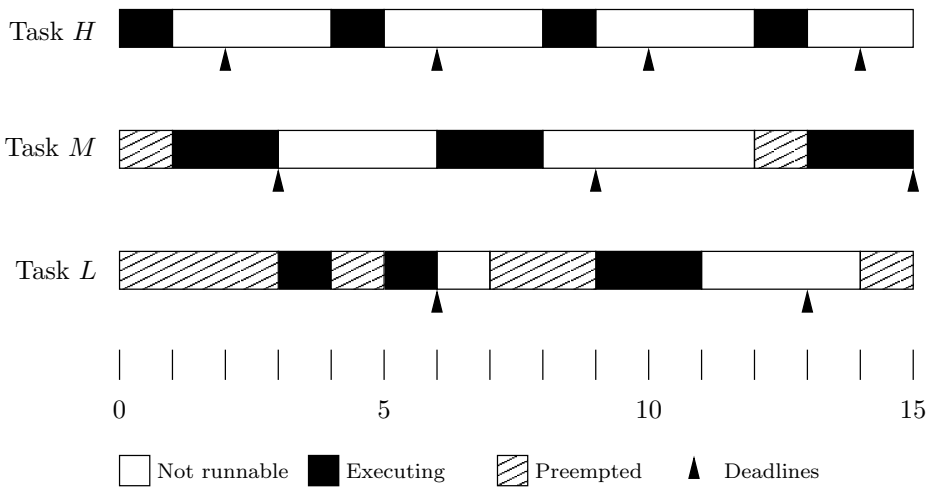

Fig. 12. Execution of the schedulable task set from Section 4.3.

	T_i	C_i	D_i
Task H	4	1	2
Task M	6	2	3
Task L	7	2	6

To show that this task set is schedulable we need to apply test 4 to each of the three tasks. The response time calculation for the low-priority task L converges as follows.

$$R_L^0 = 0$$

$$R_L^1 = C_L + \left\lceil \frac{R_L^0}{T_M} \right\rceil C_M + \left\lceil \frac{R_L^0}{T_H} \right\rceil C_H = 2 + \left\lceil \frac{0}{6} \right\rceil \cdot 2 + \left\lceil \frac{0}{4} \right\rceil \cdot 1 = 2$$

$$R_L^2 = 2 + \left\lceil \frac{2}{6} \right\rceil \cdot 2 + \left\lceil \frac{2}{4} \right\rceil \cdot 1 = 5$$

$$R_L^3 = 2 + \left\lceil \frac{5}{6} \right\rceil \cdot 2 + \left\lceil \frac{5}{4} \right\rceil \cdot 1 = 6$$

$$R_L^4 = 2 + \left\lceil \frac{6}{6} \right\rceil \cdot 2 + \left\lceil \frac{6}{4} \right\rceil \cdot 1 = 6$$

Since this calculated response time does not exceed deadline D_L, we may conclude that task L is schedulable. Indeed, the timeline in Figure 12 shows the worst-case behaviour of this task set, and the first invocation of task L exhibits the calculated response time. Similar calculations can be performed to show that both tasks M and H are also schedulable in this case.

4.4 Blocking on a Shared Resource

So far we have assumed that all tasks are independent, i.e., they do not interact apart from their competition for access to the processor. More realistically, tasks need to synchronise with one another and communicate data values. To accom-

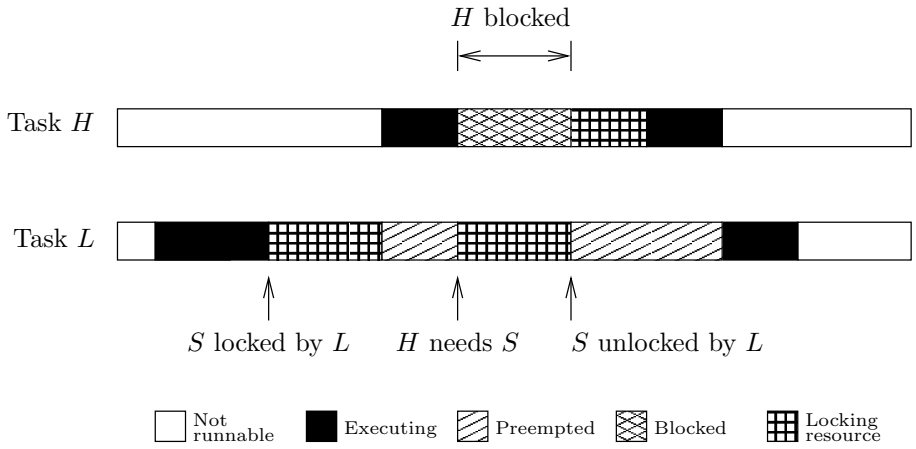

Fig. 13. Blocking by a lower-priority task.

modate this, scheduling theory's computational model assumes that tasks residing on the same processor interact by protected access to shared resources [Buttazzo 1997, §2.2.3]. (In Section 4.8 we consider message passing between tasks on different processors.) Typically, a shared resource can be accessed by only one task at a time. Any other task that wishes to access it at this time is *blocked*. Thus, by locking shared resources, low-priority task invocations can delay the progress of higher-priority ones.

For example, Figure 13 shows a situation where two tasks share the same resource S. An invocation of low-priority task L starts executing first and locks the resource. High-priority task H's invocation then arrives and preempts task L's invocation. However, after executing for a short time, task H's invocation also needs to access the shared resource. When it attempts to do so it finds that S is already locked by task L. At this point task H's invocation is blocked, i.e., it cannot execute any further, and control is returned to the low-priority task invocation. Task L's invocation then continues executing until it finishes using shared resource S and releases the lock. At this point task H's invocation is free to continue, so it immediately preempts task L's invocation and accesses S.

Therefore, blocking due to lower-priority tasks can be accommodated in schedulability test 4 simply by adding a term for task i's worst-case blocking time B_i to the response time calculation [Audsley et al. 1993, §3].

$$R_i \;=\; C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{8}$$

However, to know exactly what the worst-case blocking time will be for each task, we must look more closely at the way tasks may be blocked.

4.4.1 *Priority Inversion.* To know that a high-priority task invocation is schedulable, we must be able to put a bound on how long it will be blocked by lower-priority ones. However, this is not possible in general [Buttazzo 1997, §7.2].

For example, Figure 14 shows the behaviour of a task set in which tasks L and H both share resource S. At first, the behaviour is the same as in Figure 13.

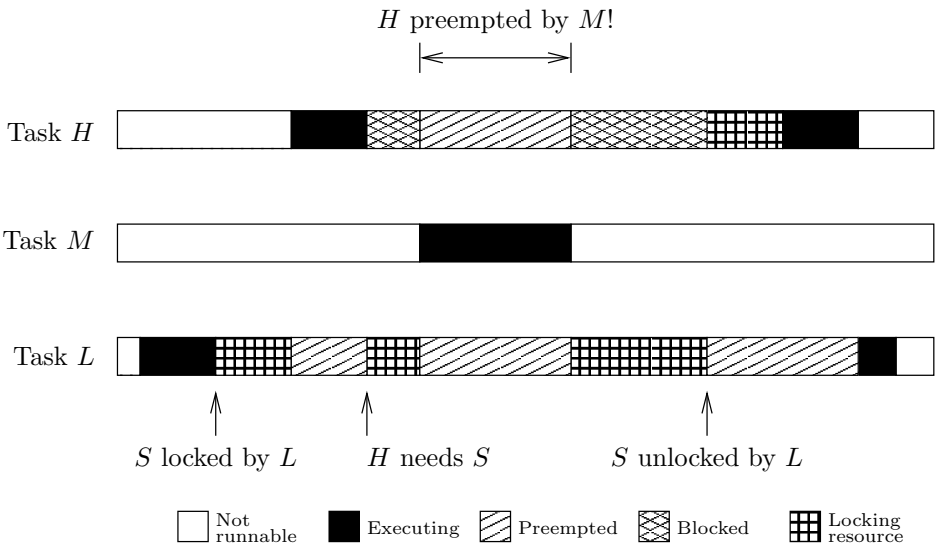

Fig. 14. Example of priority inversion.

However, after task H's invocation becomes blocked and control returns to task L's invocation, an invocation of medium-priority task M arrives. Since this newly-arrived invocation has a higher priority than the currently executing one, belonging to task L, task M's invocation can start executing immediately. Indeed, medium-priority invocations can execute indefinitely at this point, even though a high-priority task invocation is waiting. This phenomenon is known as *priority inversion* because its behaviour is the opposite of that intended by the programmer: medium-priority invocations are allowed to execute in preference to a high-priority one [Sha et al. 1990].

4.4.2 *Priority Inheritance Protocols.* A number of *resource locking protocols* have been proposed for bounding blocking times and avoiding priority inversion. For instance, a simple but potentially inefficient solution is to disallow preemption while a task invocation holds a lock [Buttazzo 1997, p. 185], and there also exist protocols suitable for dynamic priority scheduling [Baker 1990, §3.2] [Baker 1991]. For fixed-priority scheduling, the usual solution is some form of *priority inheritance protocol* [Sha et al. 1990, §III] [Chen and Lin 1990, p. 328].

The basic principle of the priority inheritance protocol is that while a task invocation locks a shared resource its priority is raised to equal that of all higher-priority blocked invocations. Thus, a task invocation's priority can change in 'fixed' priority scheduling! In this situation each task has a static *base priority* associated with it by the programmer, while a particular invocation of the task has a dynamic *active priority* determined by the locking protocol [Audsley et al. 1993, §2].

(Since task invocations inherit priorities under the priority inheritance protocol, several invocations may have the same active priority, even if their base priorities are unique. Therefore, it is also important to assume that there will be no unnecessary preemption—a task invocation must have a strictly higher priority to preempt another.)

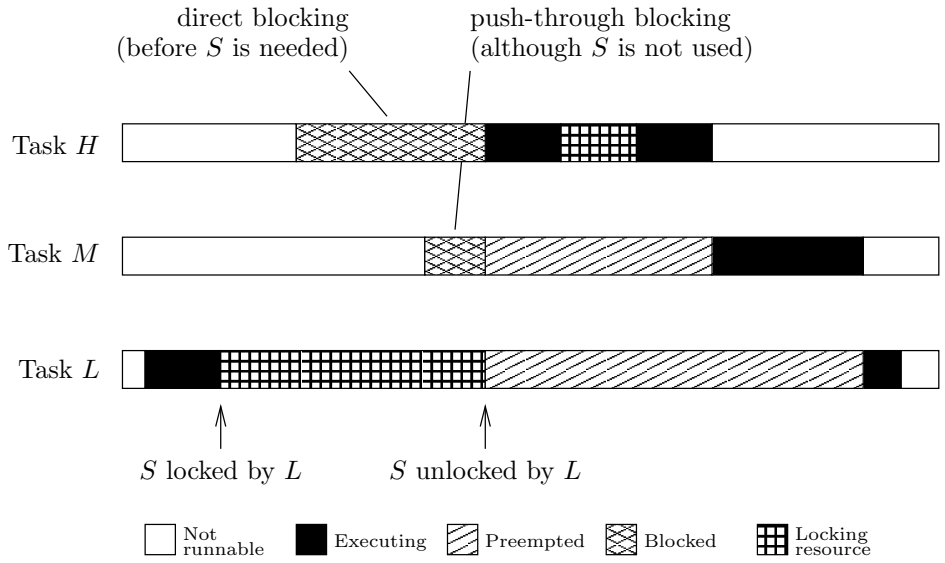

Fig. 15. Ceiling locking of a shared resource.

The advantage of the priority inheritance protocol is that when a low-priority task invocation is blocking a high-priority one, the low-priority invocation is allowed to execute with fewer preemptions, thus allowing control to go to the high-priority invocation more quickly. The priority inheritance protocol both prevents priority inversion and bounds blocking times, thus making interacting task sets analysable. A side-effect of the protocol, however, is that a task invocation may experience blocking on a shared resource even though it does not use the resource (see the example in Section 4.4.3 below).

4.4.3 *The Ceiling Locking Protocol.* In its basic form, as described above, the priority inheritance protocol can be expensive to implement because the run-time scheduler must keep track of which task invocations hold which locks. Fortunately, a slight variant of priority inheritance has proven to be almost as effective and dramatically cheaper to implement.

In the *ceiling locking protocol* each shared resource has a fixed *ceiling priority* which equals the highest base priority of any task that may access this resource [Stoyenko and Baker 1994, p. 104]. When a task invocation locks a shared resource, it immediately raises its active priority to equal this ceiling value. (In basic priority inheritance it would do this only when a higher-priority invocation became blocked on the resource.) This protocol effectively prevents any task invocation from starting to execute until all the shared resources it needs are free [Taft and Duff 1997, §D.2.1].

For example, Figure 15 shows how the task set from Figure 14 behaves under the ceiling locking protocol. The ceiling priority associated with shared resource S is 'H'. When task L's invocation locks the shared resource its (active) priority is immediately raised to equal that of task H. This means that task L's invocation executes without preemption until it releases the lock and its active priority drops back to its base level. By this time invocations of both tasks H and M have arrived

and the higher-priority one gets preference. The preemption of task H's invocation by that of task M seen in Figure 14 does not occur. The medium-priority task must wait for the high-priority one.

Two forms of blocking are evident in Figure 15. Task H's invocation suffers *direct blocking* because a lower-priority task has locked a shared resource it may need. (This occurs before the resource is actually required—in basic priority inheritance direct blocking occurs only when an attempt is made to access the locked resource. Although ceiling locking may cause tasks to be blocked in situations where they would not be under basic priority inheritance, the worst-case behaviour of the two protocols is the same.) Task M's invocation suffers *push-through blocking*, even though it does not use the shared resource, because task L's invocation inherits a higher priority while locking the resource shared with task H [Buttazzo 1997, p. 188].

Most importantly, the ceiling locking protocol has several desirable properties.

+ It is cheap to implement at run time. By raising priorities as soon as a resource is locked, whether a higher-priority task is trying to access it or not, the protocol avoids the need to make complex scheduling decisions on the fly.
+ The way that task priorities are manipulated ensures that a task invocation cannot start until all shared resources it *may* need are free. This means that no separate mutual exclusion mechanism, such as semaphores, is needed to lock shared resources [Tindell 2000]. The run-time scheduler implements both task scheduling and resource locking using the one mechanism.
+ Since a task invocation cannot start until all resources it may need are free, run-time deadlocks caused by circular dependencies on shared resources are impossible [Pilling et al. 1990].
+ Each task invocation can be blocked at most once, at its beginning, by a single lower-priority task. This provides the necessary bound on blocking times needed for analysing schedulability of interacting task sets [Audsley et al. 1995, p. 185].

For these reasons, ceiling locking is the default priority inheritance protocol in the Ada 95 real-time programming language [Taft and Duff 1997, §D.1, D.3].

4.4.4 *Response Time Test Incorporating Blocking Times.* With an understanding of the particular resource locking protocol used, the blocking term B_i needed for equation 8 can now be defined. With the ceiling locking protocol, B_i is the longest duration of a critical section executed by any lower-priority task that accesses a resource with ceiling value i or higher [Tindell 2000]. Let $locks(k, i)$ be the set of shared resources accessed by task k with ceiling priorities higher than or equal to task i's priority. Let $c_{k,S}$ be the worst-case computation time of task k while locking shared resource S.

$$B_i = \max_{\substack{k \in lp(i) \\ S \in locks(k,i)}} (c_{k,S}) \tag{9}$$

For example, consider the following task set. We saw in Section 4.3 that these three tasks are schedulable when they are independent. However, here we assume that tasks H and L both share a resource, and that an invocation of task L can

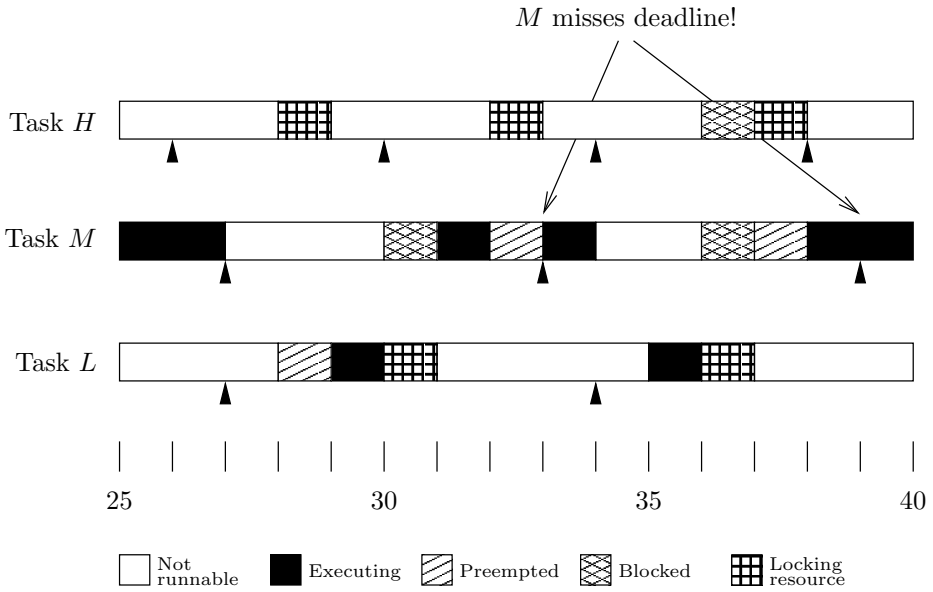

Fig. 16. Missed deadlines for the task set in Section 4.4.4 due to blocking from a lower-priority task.

lock the shared resource for up to 1 millisecond. The blocking time column was then calculated used equation 9.

	T_i	C_i	D_i	B_i
Task H	4	1	2	1
Task M	6	2	3	1
Task L	7	2	6	0

Applying test 4 using equations 8 and 9 then reveals that this task set is no longer schedulable due to the interactions between the tasks. For the medium priority task the calculated response time is as follows.

$$R_M^0 = 0$$

$$R_M^1 = C_M + B_M + \left\lceil \frac{R_M^0}{T_H} \right\rceil C_H = 2 + 1 + \left\lceil \frac{0}{4} \right\rceil \cdot 1 = 3$$

$$R_M^2 = 2 + 1 + \left\lceil \frac{3}{4} \right\rceil \cdot 1 = 4$$

$$R_M^3 = 2 + 1 + \left\lceil \frac{4}{4} \right\rceil \cdot 1 = 4$$

This exceeds its deadline as shown by the example in Figure 16.

Importantly, notice the time scale in Figure 16. Task M does not miss a deadline until time 33, when a particular invocation suffers *both* preemption from task H and blocking from task L. All previous invocations of task M meet their deadlines, including the first invocation when all three tasks arrive simultaneously. This reminds us that the behaviour of interacting tasks is much more subtle than independent

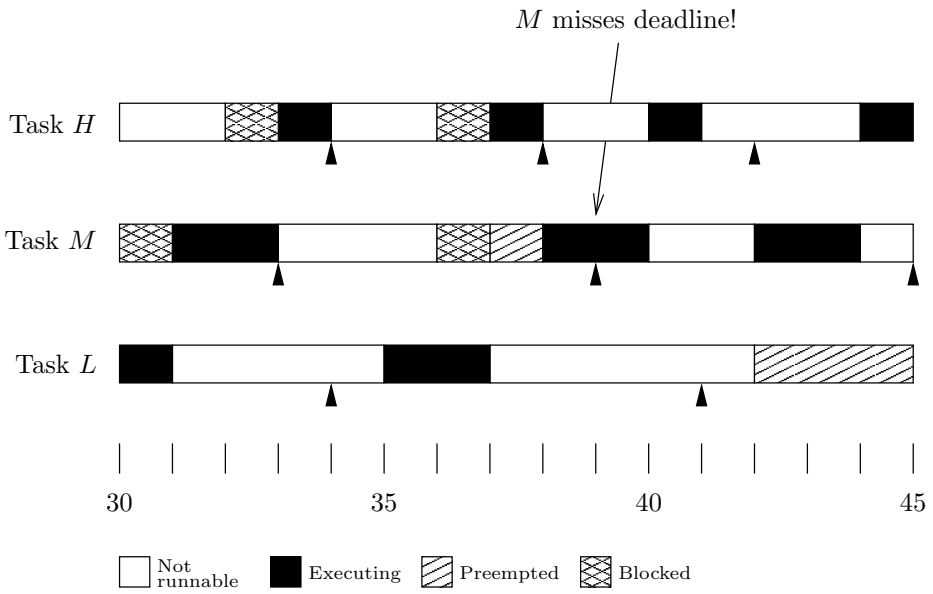

Fig. 17. A missed deadline for the task set in Section 4.5 due to non-preemptive scheduling.

ones.

4.5 Response Times for Non-Preemptive Scheduling

So far we have assumed that the scheduling policy is preemptive. Non-preemptive scheduling can be analysed as a special case, using the tests for preemptive scheduling, by assuming that all tasks share a single resource (the processor itself) and that they lock it for the entire duration of each invocation [Briand and Roy 1999, p. 19].

In this case the worst-case 'blocking' experienced by a task i is the largest computation time of any lower-priority task k. This represents the situation where an invocation of task k begins executing just before an invocation of task i. Since task k's invocation is not preemptible, task i's invocation must wait for it to finish. Thus the blocking term of equation 8 can be replaced with k's computation time as follows.

$$R_i \;=\; C_i + \max_{k \in lp(i)} C_k + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{10}$$

For example, consider the task set from Section 4.4.4. The blocking figures for tasks H and M are replaced with the worst-case computation times of lower-priority tasks. Both can be blocked by a non-preemptive invocation of low-priority task L with computation time 2.

	T_i	C_i	D_i	B_i
Task H	4	1	2	2
Task M	6	2	3	2
Task L	7	2	6	0

Thus the worst-case response time for task M under a non-preemptive scheduling policy is as follows, using equation 10.

$$R_M^0 = 0$$

$$R_M^1 = C_M + C_L + \left\lceil \frac{R_M^0}{T_H} \right\rceil C_H = 2 + 2 + \left\lceil \frac{0}{4} \right\rceil \cdot 1 = 4$$

$$R_M^2 = 2 + 2 + \left\lceil \frac{4}{4} \right\rceil \cdot 1 = 5$$

$$R_M^3 = 2 + 2 + \left\lceil \frac{5}{4} \right\rceil \cdot 1 = 6$$

$$R_M^4 = 2 + 2 + \left\lceil \frac{5}{4} \right\rceil \cdot 1 = 6$$

This exceeds task M's deadline of 3. Similar calculations show that the response time for task H equals 3 and that for task L equals 6. Thus, both the high and medium-priority tasks will miss deadlines under a non-preemptive scheduling policy. Figure 17 shows how an invocation of task M misses its deadline under non-preemptive scheduling.

A subtle point about equation 10 is that it assumes an invocation of low-priority task k can 'block' an invocation of higher-priority task i even if they arrive at the *same* time. This is a conservative assumption which models the fact that the run-time scheduler cannot react instantaneously to the arrival of task invocations. It may thus check the system clock fractionally before the arrival of an invocation of task i, and schedule task k instead [Burns and Wellings 1996b]. Having done so, the scheduler must wait for the non-preemptible invocation of task k to finish before it can schedule task i's invocation. (This danger is not a concern in preemptive scheduling because there the low-priority invocation would be immediately preempted once the higher-priority arrival was recognised.)

Unfortunately, however, equation 10 can be too naïve. Its interference term is unnecessarily pessimistic because it assumes that an invocation of task i can be preempted once started [Burns and Wellings 1996b, App. I]. The scenario shown in Figure 18(a) illustrates why the above calculation using equation 10 gave a response time of 6 for task M. In practice, though, the context switch at time 88 will never occur—once an invocation of task M has started executing, the newly-arrived invocation of task H cannot preempt it.

Therefore, a less pessimistic response time calculation can be used for non-preemptive scheduling [Burns and Wellings 1996b, App. I]. This is done in two parts. Firstly, task i's response time R_i in the non-preemptive case is defined to be its computation time C_i plus its release time r_i, i.e., the delay between its arrival and the time it can actually start executing.

$$R_i \;=\; r_i + C_i \tag{11}$$

Secondly, task i's release time is defined as its worst-case 'blocking' from a non-preemptible lower-priority invocation of some task k, plus the interference due to arrivals of each higher-priority task j. To be safely pessimistic in the latter case, it is assumed that each such task j will arrive at least once before the invocation of task i can get started, plus there may be more arrivals of task j in the interval

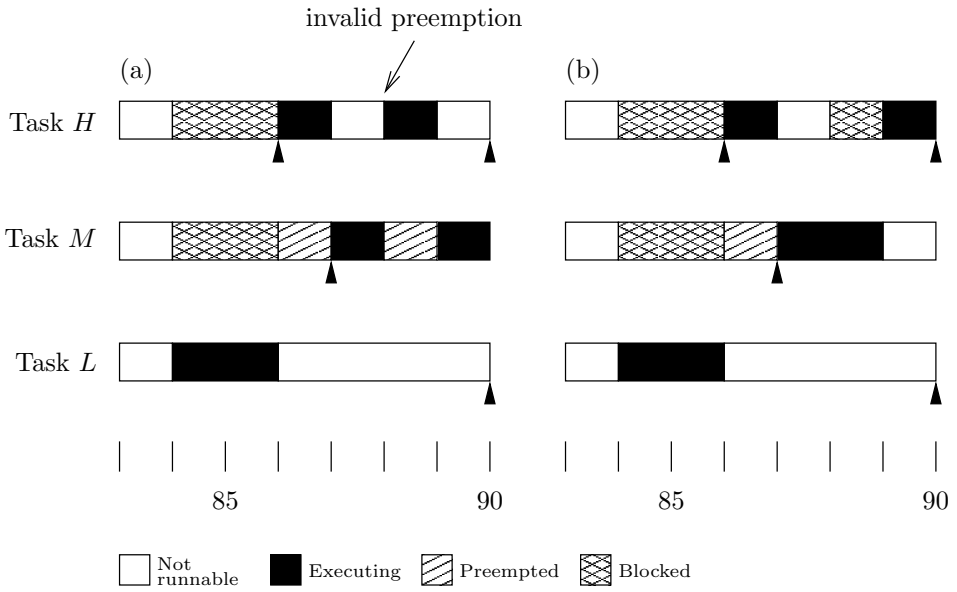

Fig. 18. (a) Incorrect modelling of interference for the task set from Section 4.5 in non-preemptive scheduling. (b) Actual worst-case behaviour for task M.

defined by r_i, depending on its period T_j.

$$r_i = \max_{k \in lp(i)} C_k + \sum_{j \in hp(i)} \left(\left\lfloor \frac{r_i}{T_j} \right\rfloor + 1 \right) C_j \tag{12}$$

(The rounding-down operator $\lfloor x \rfloor$ returns the largest integer not exceeding x.)

Calculating the response time for task M using equations 11 and 12 converges as follows.

$$\begin{aligned}
r_M^0 &= 0 & R_M^0 &= r_M^0 + C_M = 0 + 2 = 2 \\
r_M^1 &= C_L + \left(\left\lfloor \frac{r_M^0}{T_H} \right\rfloor + 1 \right) C_H & R_M^1 &= 3 + 2 = 5 \\
&= 2 + \left(\left\lfloor \frac{0}{4} \right\rfloor + 1 \right) \cdot 1 = 3 & & \\
r_M^2 &= 2 + \left(\left\lfloor \frac{3}{4} \right\rfloor + 1 \right) \cdot 1 = 3 & R_M^2 &= 3 + 2 = 5
\end{aligned}$$

This result is indeed the true worst-case response time for a non-preemptive invocation of task M as shown in Figure 18(b).

4.6 Incorporating Scheduling Overheads

So far we have ignored the overheads of the run-time scheduler, and have relied on them being incorporated in the worst-case computation time C_i for each task i. To justify this approach, Figure 19 shows explicit scheduler overheads, with the scheduler itself modelled as a fictitious task which is invoked whenever an application task's invocation arrives or finishes. An invocation of task L arrives at time 2

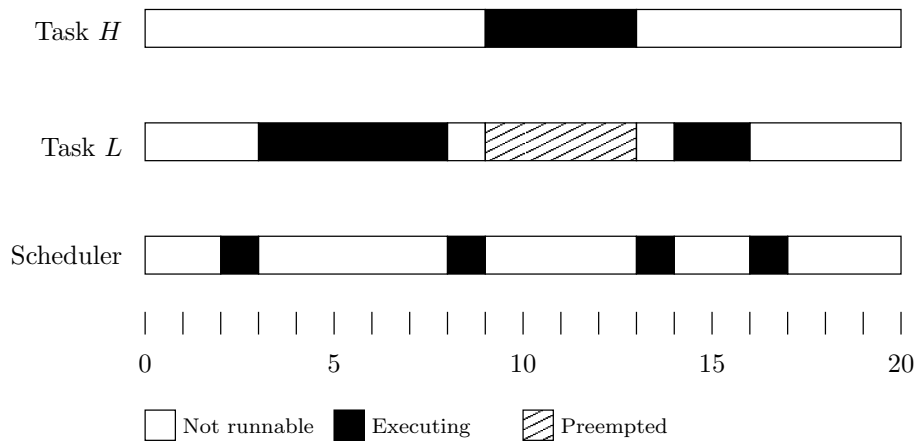

Fig. 19. Scheduler overheads associated with task invocations.

and needs 7 milliseconds of processor time (excluding scheduler overheads) and an invocation of task H arrives at time 8 and needs 4 milliseconds of processor time. Each invocation of the fictitious scheduler task consumes 1 millisecond. At time 2 the scheduler starts task L's invocation executing. Then at time 8 the scheduler awakes to start the newly-arrived invocation of task H, and put task L's preempted invocation back on the ready queue. At time 13 task H's invocation finishes its computation and the scheduler is invoked again to restore task L's invocation. This then finishes and the scheduler tidies-up at time 16. Thus the two scheduler invocations at times 2 and 16 can be associated with the invocation of task L, and the two scheduler invocations at times 8 and 13 can be associated with the invocation of task H. In this way, bounds can be defined on the scheduling overheads associated with each task invocation, thus simplifying schedulability analysis by effectively hiding scheduling overheads in each task's worst-case computation time.

Sometimes, however, it becomes necessary to more accurately characterise the scheduler's impact, especially when attempting to reduce pessimism in the calculations for systems with very high processor utilisation. In this case, significant context switching overheads can be explicitly separated from task computation times and incorporated as separate terms in the response time equation [Burns and Wellings 1995b, pp. 713–714].

Another practical scheduling issue is that the run-time operating system is often controlled by a kernel which is awoken periodically by a hardware interrupt. Such timer-driven scheduling overheads can be incorporated into the analysis as a fictitious periodic task [Burns and Wellings 1995b, pp. 716–717]. For instance, Figure 20 shows an application task A which has a period of 30 and requires 5 milliseconds of processor time at each invocation (excluding scheduler overheads). It is controlled by a timer-driven scheduler which is awoken by a clock interrupt every 4 milliseconds and takes 1 millisecond to check the ready queue and perform a context switch, if necessary. As shown, the invocation of task A suffers interference from the scheduler itself, which preempts the task at time 36, just to check if any scheduling actions are required (which they were not). Such interference can be incorporated easily into response time calculations using the approach illustrated

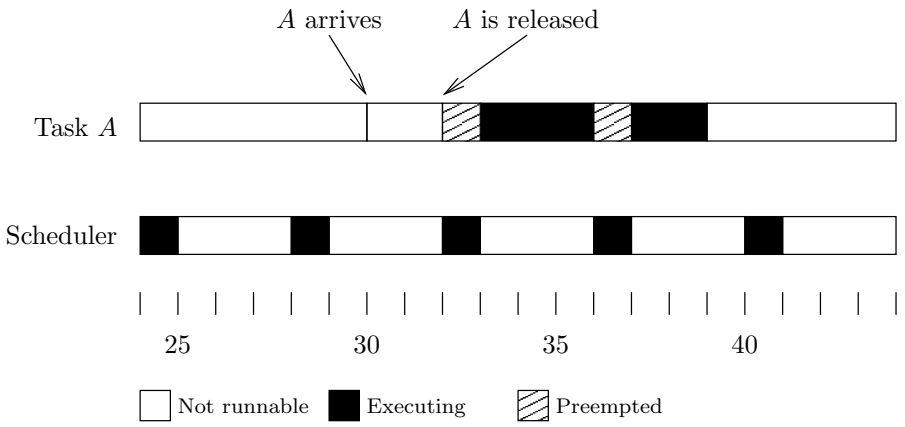

Fig. 20. Impact of a timer-driven scheduler on a task invocation.

by equation 6.

More seriously, however, note that task A's period is not a whole multiple of the scheduler's period. Thus, although an invocation of task A nominally arrives at time 30, the periodic scheduler does not awaken and recognise this until time 32. Thus the *arrival* of the task invocation, i.e., the earliest time at which it could start executing in theory, is separated from its *release* time, i.e., the earliest time at which the scheduler can actually start it executing. Such effects can be incorporated into schedulability analysis as a form of release jitter, as explained in Section 4.7 below.

Finally, the scheduling overheads may vary depending on whether a context switch is required or not. In Figure 20 it would be more realistic to assume that the scheduling overheads at time 32, when a new task invocation is started, would be greater than those at time 36, when the scheduler is invoked but takes no action. If necessary, such detailed complexities can be incorporated into schedulability analysis by appropriate extensions to the interference equations [Burns and Wellings 1995b, pp. 715–716].

4.7 Release Jitter

A task suffers from *release jitter* if the earliest time at which it can start executing is later than its nominal arrival time. In Section 4.6 we saw that this can occur when tasks are invoked by a timer-driven scheduler, and in Section 4.8 we will see that tasks with precedence constraints in multi-processor systems can be analysed using the principles of release jitter.

To see the effect of release jitter on schedulability, consider the following task set.

	T_i	C_i	D_i	J_i
Task H	12	3	8	4
Task L	16	6	10	0

Normally task L would be schedulable (with response time $R_L = 9$) but Figure 21 shows that this is not so when an invocation of task H has its release delayed. The invocation of task H nominally arrives at time 12 but, due to its release jitter, is not actually scheduled until time 16. This is the same time that an invocation of

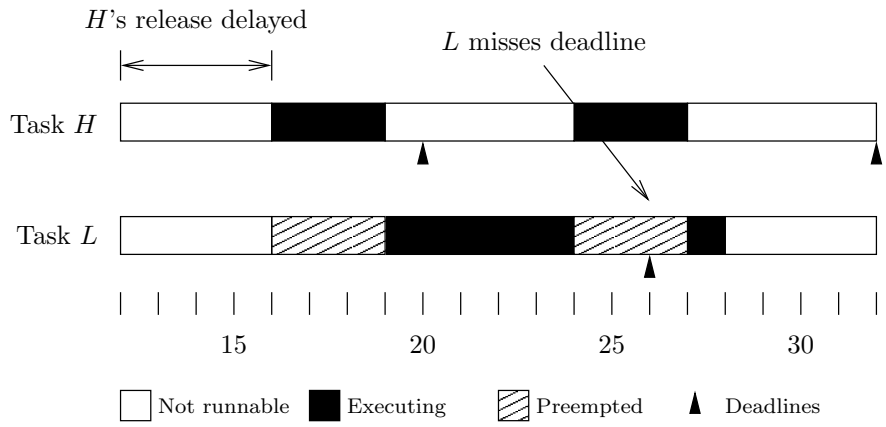

Fig. 21. A missed deadline for the task set in Section 4.7 due to release jitter.

task L arrives, and this is then immediately preempted by the delayed invocation of task H. Task L's invocation is also subsequently preempted by the next invocation of task H which is released, without any delay, at time 24. In effect, the two successive invocations of task H occur closer together than task H's period of 12, so the interference task H inflicts on this particular invocation of task L is worse than indicated by T_H, and task L's invocation misses its deadline.

To account for this, we need to introduce equations for calculating response times in the presence of release jitter [Audsley et al. 1993, §4]. To do so, we first calculate the worst-case response-time r_i of an invocation of task i measured from the time it is *released*.

$$r_i \;=\; C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i + J_j}{T_j} \right\rceil C_j \tag{13}$$

This is similar to equation 8 except that the interval of interest in the interference term is $r_i + J_j$. It thus incorporates the release jitter J_j of higher-priority task j to account for the way a delayed higher-priority task invocation can increase the interference on an invocation of task i.

Task i's worst-case response time R_i measured from the time it *arrives* is then merely its 'release-time' response time, plus its own worst case release jitter J_i.

$$R_i \;=\; r_i + J_i \tag{14}$$

As usual, the recursively-defined response time equation can be solved iteratively [Audsley et al. 1993, §4].

For the task set above, we can now calculate the response time of task L, in the

light of task H's release jitter, using equations 13 and 14 as follows.

$$r_L^0 = 0$$

$$r_L^1 = C_L + B_L + \left\lceil \frac{r_L^0 + J_H}{T_H} \right\rceil C_H = 6 + 0 + \left\lceil \frac{0+4}{12} \right\rceil \cdot 3 = 9$$

$$r_L^2 = 6 + \left\lceil \frac{9+4}{12} \right\rceil \cdot 3 = 12$$

$$r_L^3 = 6 + \left\lceil \frac{12+4}{12} \right\rceil \cdot 3 = 12$$

$$R_L = r_L^3 + J_L = 12 + 0 = 12$$

This calculated worst-case response time is exactly the result shown in Figure 21.

4.8 Multi-Processor Scheduling Theory

Applying scheduling theory to multi-processor transactions means that inter-processor communication overheads and precedence constraints must be considered during the analysis.

In the analysis model, the effect of asynchronous message-passing overheads can be handled by treating the network as a fictitious processor and the messages as fictitious tasks residing on this processor [Richard et al. 2001]. (Section 4.8.1 below illustrates this.) Alternatively, the effect of blocking forms of communication, such as accessing a shared data bus, can be modelled by adding a fictitious high-priority 'network task' to 'block' (preempt) tasks that must access the bus [Briand and Roy 1999, pp. 175–176].

Precedence constraints between tasks on different machines can be handled either by performing 'whole system' schedulability analysis (Section 4.8.1) or, if offsets are used to implement precedence constraints, by offset-based analysis (Section 4.8.2).

4.8.1 *Holistic Analysis.* It is tempting to think that schedulability analysis for a multi-processor system could be achieved merely by performing a separate analysis for each processor. However, the cross-processor impact of precedence constraints means that the problem is not partitionable in general [Grigg and Audsley 1999, §3.2]. Therefore, 'holistic' schedulability analysis aims to extend uni-processor analysis concepts for a whole multi-processor system [Tindell and Clark 1994].

Section 4.7 explained how tasks whose release is delayed can be analysed. In holistic schedulability analysis this capability is used to model the effects of precedence constraints. In Figure 9, for instance, high-priority transaction H can be modelled by assuming the three individual task invocations arrive simultaneously at time 0, but that task $Hproc$'s invocation suffers from release jitter while waiting for its message from task Hin's invocation, and similarly for the invocation of task $Hout$ while waiting for task $Hproc$'s invocation to finish. Thus the first task in a transaction is modelled as periodic, and the subsequent ones are treated as periodic with release jitter [Falardeau 1994, p. 46].

The xth approximation to task i's release (starting time) jitter is calculated in terms of its predecessor's response times as follows [Richard et al. 2001]. Let $pre(i)$ be the set of tasks that belong to the same transaction as task i and occur before

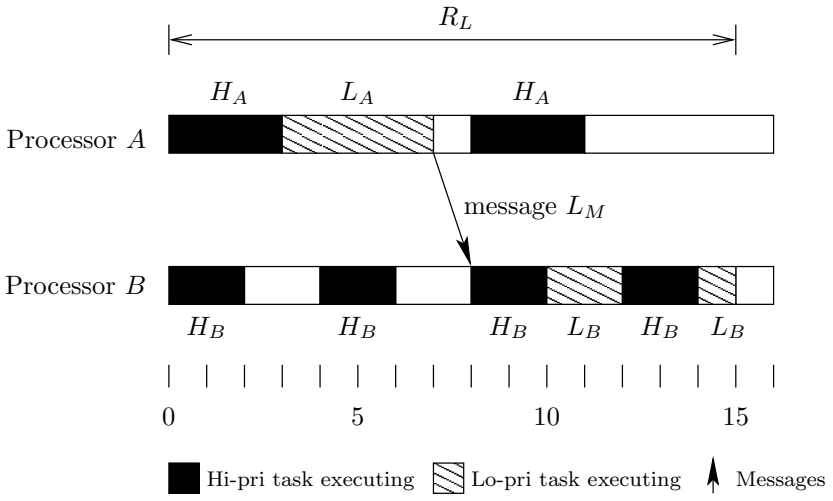

Fig. 22. Example of multi-processor transaction scheduling for the task set in Section 4.8.1.

task i in the precedence ordering.

$$J_i^x \;=\; \max_{j \in pre(i)} R_j^x \tag{15}$$

In other words, the earliest time at which task i's invocation can start executing is the latest time at which one of its predecessors can finish.

The xth approximation of task i's response time relative to its release time is then calculated via equation 13.

$$r_i^x \;=\; C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i^{x-1} + J_j^{x-1}}{T_j} \right\rceil C_j \tag{16}$$

As usual, we let $r_i^0 = 0$. In the case of a multi-processor system, set $hp(i)$ means those tasks with higher priority than task i that reside on the *same* processor as i.

Then the xth approximation of task i's response time relative to its arrival is defined as follows.

$$R_i^x \;=\; r_i^x + J_i^x \tag{17}$$

For example, consider the following multi-processor task set, illustrated in Figure 22. (For simplicity we have assumed that there is no intra-processor blocking, although this is added easily.)

	T_i	C_i	D_i	$pre(i)$
Task H_A	8	3	4	–
Task L_A	16	4	8	–
Task L_M	–	1	–	L_A
Task H_B	4	2	3	–
Task L_B	16	3	16	L_M

There are two processors, A and B. A high-priority task H_A resides on processor A and another (unrelated) high-priority task H_B resides on processor B. A

low-priority transaction L consists of two tasks, L_A and L_B, which reside on processors A and B, respectively. A precedence constraint requires that invocations of task L_B always occur after the corresponding invocation of task L_A. This is implemented by sending a message L_M from the first task to the second. In the worst case, each such message takes 1 millisecond to transmit. To account for this overhead, message L_M is represented in the task set as a fictitious task. This task resides on its own imaginary processor and fits into transaction L's precedence ordering between tasks L_A and L_B. (Since message-passing 'task' L_M has no competition for its 'processor' we have not bothered to specify its deadline or period. However, these features can be incorporated if inter-processor communication is via time-slots on a shared data bus [Tindell and Clark 1994, §3].)

The high-priority tasks' response times in this example are trivial, using equation 5, because they suffer from no blocking or interference.

$$R_{H_A} = C_{H_A} = 3$$
$$R_{H_B} = C_{H_B} = 2$$

To analyse transaction L, holistic analysis requires that we work our way forward along its precedence ordering [Burns and Wellings 1996a, p. 49]. The response time analysis for the first task is standard, using equation 7.

$$R^0_{L_A} = 0$$
$$R^1_{L_A} = C_{L_A} + \left\lceil \frac{R^0_{L_A}}{T_{H_A}} \right\rceil C_{H_A} = 4 + \left\lceil \frac{0}{8} \right\rceil \cdot 3 = 4$$
$$R^2_{L_A} = 4 + \left\lceil \frac{4}{8} \right\rceil \cdot 3 = 7$$
$$R^3_{L_A} = 4 + \left\lceil \frac{7}{8} \right\rceil \cdot 3 = 7$$

Also, this task does not suffer from release jitter, so $J_{L_A} = 0$.

The fictitious communications task L_M resides on its own imaginary processor and thus suffers from no blocking or interference. However, as per equation 15, it does have release jitter due to its dependence on task L_A.

$$J^0_{L_M} = 0$$
$$J^1_{L_M} = R^1_{L_A} = 4$$
$$J^2_{L_M} = R^2_{L_A} = 7$$
$$\vdots$$

Its response time, calculated using equations 16 and 17, is therefore affected only by this jitter and its own imaginary computation time (message propagation delay).

$$r^0_{L_M} = 0 \qquad R^0_{L_M} = r^0_{L_M} + J^0_{L_M} = 0$$
$$r^1_{L_M} = C_{L_M} = 1 \qquad R^1_{L_M} = r^1_{L_M} + J^1_{L_M} = 1 + 4 = 5$$
$$r^2_{L_M} = 1 \qquad R^2_{L_M} = 1 + 7 = 8$$
$$r^3_{L_M} = 1 \qquad R^3_{L_M} = 1 + 7 = 8$$

The final task in transaction L's precedence ordering suffers release jitter while waiting for message L_M to arrive.

$$\begin{aligned} J^0_{L_B} &= 0 \\ J^1_{L_B} &= R^1_{L_M} = 5 \\ J^2_{L_B} &= R^2_{L_M} = 8 \\ &\vdots \end{aligned}$$

It also suffers interference from the higher-priority task on its processor. Task H_B has no release jitter, so $J_{H_B} = 0$. Task L_B's response time is then calculated using equations 16 and 17.

$$\begin{aligned} r^0_{L_B} &= 0 & R^0_{L_B} &= r^0_{L_B} + J^0_{L_B} = 0 \\ r^1_{L_B} &= C_{L_B} + \left\lceil \frac{r^0_{L_B} + J^0_{H_B}}{T_{H_B}} \right\rceil C_{H_B} & R^1_{L_B} &= r^1_{L_B} + J^1_{L_B} = 3 + 5 = 8 \\ &= 3 + \left\lceil \frac{0+0}{4} \right\rceil \cdot 2 = 3 \\ r^2_{L_B} &= 3 + \left\lceil \frac{3+0}{4} \right\rceil \cdot 2 = 5 & R^2_{L_B} &= 5 + 8 = 13 \\ r^3_{L_B} &= 3 + \left\lceil \frac{5+0}{4} \right\rceil \cdot 2 = 7 & R^3_{L_B} &= 7 + 8 = 15 \\ r^4_{L_B} &= 3 + \left\lceil \frac{7+0}{4} \right\rceil \cdot 2 = 7 & R^4_{L_B} &= 7 + 8 = 15 \end{aligned}$$

This gives us the total worst-case response time for transaction L, as confirmed by the instance shown in Figure 22.

4.8.2 *Offset-Based Analysis.* In Section 3.8 we noted that precedence-constrained tasks can be implemented using offsets. The offsets are chosen so that the $(n+1)$th task in the precedence ordering of a transaction is given an offset at least as great as the worst-case response time of the nth task [Bate and Burns 1999]. A significant advantage of this is that schedulability analysis then becomes partitionable—uniprocessor analysis techniques can be applied independently to each machine [Falardeau 1994, p. 48].

However, analysing task sets that use offsets differs from the situation where tasks experience release jitter (Section 4.7). A task's worst-case release jitter J_i is an *upper* bound—an invocation of task i may have its release delayed by up to this duration. However, a task's offset O_i is a *lower* bound—every invocation of task i will have its release delayed by at least this duration.

In the case where each task i has an offset O_i, the schedulability test for task i is simply modified as follows [Burns and Wellings 1996a, p. 49].

$$O_i + R_i \leqslant D_i \tag{18}$$

Importantly, the calculation for response time R_i uses equation 8 and thus depends only on tasks that reside on the same processor as task i. Unlike the holistic approach (Section 4.8.1), an offset-based implementation allows each processor to

be analysed for schedulability separately from other processors. With offsets, the system is thus structured in a way that simplifies schedulability analysis. Also, use of offsets can reduce overall task jitter.

However, although safe, test 18 can be overly pessimistic. For instance, consider the following (uniprocessor) task set.

	T_i	C_i	D_i	O_i
Task H	4	2	3	0
Task L	4	2	3	2

Clearly this task set is schedulable. The first invocation of task H executes between times 0 to 2, the second between times 4 and 6, and so on, whereas the first invocation of task L executes between times 2 and 4, the second between times 6 and 8, etc. However, applying equation 8 to task L gives a response time of 4, exceeding its deadline, so the task is deemed unschedulable by test 18. (Keep in mind that task L's deadline is measured relative to its arrival, so the absolute deadline for its first invocation is 5, the second is 9, etc.) The problem is that the standard response time analysis defined by equation 8 fails to account for the different *phasing* of the two tasks, which in this case eliminated potential interference.

Developing less pessimistic schedulability tests for offset-based implementations is still a research topic. It can be approached either by devising modifications to the response time equation to define interference and blocking with respect to the periods and offsets of other tasks [Bate and Burns 1999]. Alternatively, similar results can be achieved using the standard response time equation by devising an imaginary task set for analysis which groups tasks by their periods and offsets [Bate 1999].

5. OTHER ISSUES

In this section we discuss two concerns for the practical application of scheduling theory.

5.1 Determining Worst-Case Execution Times

Scheduling theory assumes that the programmer can supply accurate estimates for the worst-case computation time C_i for each task i and the worst-case blocking time B_i due to tasks of lower priority. Both of these figures depend on the execution time of (sequential) code fragments from the task bodies. (With respect to Figure 3, the computation time for an invocation of this task is determined by how long it takes to perform one complete iteration of the loop, excluding any periods of suspension.) However, analysing program code for its worst-case execution time is a challenging problem in its own right. Moreover, it is important for the results to be as accurate as possible. Pessimistic estimates will mean that the schedulability tests may fail schedulable task sets, leading to unnecessary optimisation of the program. Even worse, overly optimistic estimates may mean that the tests pass unschedulable task sets, leading to missed deadlines at run time.

The most obvious way to determine the worst-case execution time for a task invocation is to simply execute the task code and measure how long it takes. However, this relies on the test successfully reproducing the worst-case behaviour of the task. Furthermore, such testing can be difficult for embedded control programs because

they must be executed in an appropriate run-time environment, which may require construction of a suitable test harness.

Therefore, worst-case execution time prediction has focussed on syntactic analysis of program code. Initially this was done for high-level language programs by defining a 'cost function' for each language construct and using it to sum the total execution-time cost associated with all control-flow paths through a task [Shaw 1989] [Puschner and Koza 1989]. This proved difficult to do accurately, however, without detailed knowledge of the compiler's assembler code generation strategy and optimisations.

Given the difficulties of analysing high-level language programs, recent research has concentrated on analysis of compiled machine code. However, even this has proven difficult. RISC processors are popular for programming real-time systems because of their high processing speeds [Williams 1991]. However, the very features they use to achieve high performance, such as instruction pipelining and data caching, introduce nondeterminism into the execution time of the code, and thus thwart attempts to accurately predict worst-case execution times. Therefore, much research has gone into explicitly allowing for such features in machine code analysis [Healy et al. 1995] [Lim et al. 1995] [Theiling et al. 2000] [Hur et al. 1995].

Furthermore, in both high and assembler-level timing prediction, a significant problem is that many syntactically-valid control-flow paths through program code are semantically 'dead' because they can never be followed at run time. Including such paths in execution time analysis may unnecessarily increase the pessimism of the estimate. Therefore, considerable effort has gone into algorithms [Park 1993] [Altenbernd 1996] [Ermedahl and Gustafsson 1997] [Lundqvist and Stenström 1999] and semantic definitions [Chapman et al. 1996] [Hayes et al. 2001] for identifying and excluding such dead paths.

5.2 Programming Guidelines

We have seen that scheduling theory assumes a simple 'computational model' of the task set being analysed. It is therefore important that the actual system is programmed in a way compatible with this model. The most important advance in this direction was the design of the Ada 95 programming language. Its revision of the original Ada 83 language specifically took recent progress in scheduling theory into account [Stoyenko and Baker 1994]. The resulting language therefore offers numerous constructs customised for programming multi-tasking, real-time programs in a way amenable to analysis using scheduling theory. In particular, Ada 95 features: 'delay until' statements for programming periodic tasks; 'protected objects' for mutually-exclusive access to shared resources; interrupt handlers implemented as (very) high-priority protected procedures; well-defined execution time overheads for all language constructs; and default implementations of fixed-priority, preemptive task scheduling and priority ceiling locking [Taft and Duff 1997].

However, the Ada language is also very rich and contains many features whose timing behaviour are hard to reason about. Therefore, an easily-analysable subset of the language, known as the 'Ravenscar Profile' [Burns et al. 1998] is currently being standardised.

Also, although Ada 95 provides strong support for uni-processor schedulability analysis, its support for multi-processor (distributed) real-time systems is not as well developed [Burns and Wellings 1995a, Ch. 14] and is still the subject of research

[García and Harbour 2001].

Ada 95 remains the best high-level language for programming analysable real-time systems. Attempts are currently underway to develop analysable versions of other programming languages, by borrowing concepts from Ada 95 [Brosgol and Dobbing 2001], but this work is still in its infancy.

6. APPLYING SCHEDULING THEORY IN PRACTICE

The examples above have shown that the calculations needed to assess schedulability are simple but tedious. Not surprisingly, therefore, there have been numerous tools developed to perform the necessary arithmetic [Briand and Roy 1999, App. D]. For instance, the MetaH programming environment performs basic workload calculations using equation 3 [Honeywell Technology Center 1998]. While they save some effort, such tools do not necessarily give the programmer any insight into the dynamic behaviour of the task set. This can be achieved by simulation tools which display (worst-case) timelines for a given task set [Audsley et al. 1994] [de Beer and Fidge 2000].

We have also seen that scheduling theory has been thoroughly explored in the academic literature and has progressed dramatically in recent years. However, the extent of its application in practice is harder to judge. For instance, avionics applications are often cited in the literature to motivate scheduling principles [Grigg and Audsley 1999] [Briand and Roy 1999], but the examples there are relatively simple, generalised case studies [Locke et al. 1990].

A more realistic application of scheduling theory was to the Olympus satellite's Attitude and Orbital Control System [Burns and Wellings 1995b]. The system was (re)engineered using fixed-priority preemptive scheduling, with deadline-monotonic priority allocation, and programmed in Ada. Worst-case computation times were predicted using an analysis tool integrated into the front-end of the Ada compiler. It was found that context switching times were significant in this system, so a variant of the response time equations was defined that made these overheads explicit. The task set also featured offsets, jitter and variable computation times that needed to be accounted for in order to produce acceptably accurate results. It was concluded that although scheduling theory provides 'a sound theoretical understanding of how to engineer real-time systems', in a complex application the schedulability equations must be customised to match the peculiarities of the particular system [Burns and Wellings 1995b].

In the avionics field, a detailed study was undertaken of how basic processor utilisation analysis could be applied to the Mission Computer Software for Canada's CF-188 fighter aircraft [Falardeau 1994]. The hardware architecture consists of two Mission Computers (providing redundancy for fault tolerance) connected by pairs of data buses [Falardeau 1994, §4.1]. Each Mission Computer comprises several processors, most notably a general purpose Central Processing Unit and a dedicated Input/Output Processor. The software architecture of the Operational Flight Program that resides on each of the Mission Computers is complex. From a scheduling viewpoint it consists of two types of transactions (confusingly called 'task rates' in the CF-188 documentation) [Falardeau 1994, §4.3.2]. Firstly, routine system functions are programmed as periodic transactions, with a rate monotonic priority assignment. A clock interrupt-driven kernel is used to schedule periodic transactions. Each such transaction consists of an input task, followed by a pro-

cessing task, and then an output task. The input and output tasks reside on the Input/Output Processor, and the processing task on the Central Processing Unit. The CPU initiates tasks on the IOP by sending queued (asynchronous) 'I/O request' messages over a bus, whereas the IOP uses an interrupt to signal the completion of an I/O task to the CPU. Secondly, sporadic actions which must respond quickly to external stimuli are implemented as high-priority interrupt-triggered transactions which preempt the currently executing transaction.

To analyse this complex system, extensive tables were drawn up to document the characteristics of transactions such as the precedence ordering of tasks, their periods, deadlines, priorities, worst-case computation times, release jitter characteristics, use of shared resources, and so on [Falardeau 1994, §4.4.2]. Sporadic transactions were modelled as periodic ones using their minimum interarrival rate as their period [Falardeau 1994, p. 75]. Overheads associated with the run-time kernel were explicitly considered [Falardeau 1994, p. 77]. Two analysis models were then applied. The first [Falardeau 1994, §4.4.4] attempted to partition the analysis by adding delays due to input and output activities into the computation time associated with each task and then applying the basic processor utilisation test (Section 4.1). However, the complex pattern of communication in the system made defining these delays accurately very difficult. The second approach [Falardeau 1994, §4.4.5] used a simple distributed system perspective [Burns and Wellings 1996a, p. 49] and attempted to determine the worst-case response times for each transaction by following the chain of precedence constraints and summing the task computation times and message propagation delays encountered. Most of the individual tasks comprising the transactions in the system suffered from considerable release jitter due to their precedence constraints. These values were felt to be difficult to calculate, so it was suggested that an offset-based model (Section 4.8.2) would be easier to analyse. Overall, it was concluded that neither approach seemed entirely satisfactory. It is therefore now interesting to ask whether a holistic analysis (Section 4.8.1), which was not available when this study was done, would have proved more successful.

Finally, apart from these technical challenges, a further practical issue for real-time programmers wanting to apply scheduling theory is that safety certification standards for such systems [RTCA, Inc. 1992] have yet to embrace modern multi-tasking programming principles, and therefore impose requirements that restrict the use of such technology [Burns and Wellings 1996b].

7. CONCLUSIONS

We have seen that real-time scheduling theory has now matured into a powerful and practical concept. It provides programmers with a predictive theory for assessing the capabilities of a multi-tasking system design before its construction is completed, and for identifying problems in existing systems. It is thus one aspect of 'software engineering' that genuinely has the characteristics of a true engineering discipline.

ACKNOWLEDGMENTS

I wish to thank David Tombs and Antonio Cerone for helping with Section 6, and Robert Colvin for reviewing a draft of this paper. This work was funded by

the Air Operations Division of the Defence Science and Technology Organisation. Particular thanks are due to Drs Zahid Qureshi, Paul Johnson and Thong Nguyen.

REFERENCES

Altenbernd, P. 1996. On the false path problem in hard real-time programs. In *Proc. 8th Euromicro Workshop on Real-Time Systems (WRTS)* (1996), pp. 102–107.

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

Audsley, N. C., Burns, A., Davis, R. I., Tindell, K. W., and Wellings, A. J. 1995. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems 8*, 173–198.

Audsley, N. C., Burns, A., Richardson, M. F., and Wellings, A. J. 1992. Deadline monotonic scheduling theory. In *Proceedings 18th IFAC/IFIP Workshop on Real-Time Programming (WRTP'92)* (June 1992).

Audsley, N. C., Burns, A., Richardson, M. F., and Wellings, A. J. 1994. STRESS: A simulator for hard real-time systems. *Software—Practice & Experience 24*, 6 (June), 543–564.

Baker, T. P. 1990. A stack-based resource allocation policy for realtime processes. In *Proc. Real-Time Systems Symposium* (Dec. 1990), pp. 191–200. IEEE Computer Society Press.

Baker, T. P. 1991. Stack-based scheduling of real-time processes. *Real Time Systems 3*, 1 (March), 67–99.

Baker, T. P. and Shaw, A. 1989. The cyclic executive model and Ada. *Journal of Real-Time Systems 1*, 1 (June), 7–26.

Bate, I. 1999. *Scheduling and Timing Analysis for Safety Critical Real-Time Systems*. Ph. D. thesis, Department of Computer Science, The University of York.

Bate, I. and Burns, A. 1999. A framework for scheduling and schedulability analysis for safety-critical embedded control systems. Draft, Department of Computer Science, The University of York.

Blázquez, V., Redondo, L., and Freniche, J. L. 1992. Experiences with delay until for avionics computers. *Ada Letters XII*, 1 (Jan/Feb), 65–72.

Briand, L. P. and Roy, D. M. 1999. *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach*. IEEE Computer Society.

Brosgol, B. and Dobbing, B. 2001. Can Java meet its real-time deadlines? In D. Craeynest and A. Stroheimer Eds., *Reliable Software Technologies (Proceedings of Ada-Europe 2001)*, Volume 2043 of *Lecture Notes in Computer Science* (2001), pp. 68–87. Springer-Verlag.

Burns, A., Dobbing, B., and Romanski, G. 1998. The Ravenscar tasking profile for high integrity real-time programs. In L. Asplund Ed., *Reliable Software Technologies—Ada-Europe '98*, Volume 1411 of *Lecture Notes in Computer Science* (1998), pp. 263–275. Springer-Verlag.

Burns, A. and Wellings, A. 1996a. Advanced fixed priority scheduling. In M. Joseph Ed., *Real-Time Systems: Specification, Verification and Analysis*, Chapter 3, pp. 32–65. Springer-Verlag.

Burns, A. and Wellings, A. J. 1990. *Real-Time Systems and Their Programming Languages*. Addison-Wesley.

Burns, A. and Wellings, A. J. 1995a. *Concurrency in Ada*. Cambridge University Press.

Burns, A. and Wellings, A. J. 1995b. Engineering a hard real-time system: From theory to practice. *Software–Practice & Experience 25*, 7 (July), 705–726.

Burns, A. and Wellings, A. J. 1996b. Simple Ada 95 tasking models for high integrity applications. Department of Computer Science, University of York.

Buttazzo, G. C. 1997. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer.

Chapman, R., Burns, A., and Wellings, A. 1996. Combining static worst-case timing analysis and program proof. *Real-Time Systems 11*, 145–171.

Chen, M.-I. and Lin, K.-J. 1990. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Journal of Real-Time Systems 2*, 4 (Nov.), 325–346.

de Beer, A. and Fidge, C. J. 2000. A simple multi-tasking simulator. Technical Report 00-28 (Aug.), Software Verification Reseach Centre, The University of Queensland.

Embedded Systems Programming. 2001. Buyer's guide 2002. Special Issue, Volume 14, Number 9.

Ermedahl, A. and Gustafsson, J. 1997. Deriving annotations for tight calculation of execution time. In C. Lengauer, M. Griebel, and S. Gorlatch Eds., *Euro-Par'97: Parallel Processing*, Volume 1300 of *Lecture Notes in Computer Science* (1997), pp. 1298–1307. Springer-Verlag.

Falardeau, J. D. G. 1994. Schedulability analysis in rate monotonic based systems with application to the CF-188. Master's thesis, Department of Electrical and Computer Engineering, Royal Military College of Canada.

Fidge, C. J. 1998. Real-time schedulability tests for preemptive multitasking. *Real-Time Systems 14*, 1 (Jan.), 61–93.

Fidge, C. J., Hayes, I. J., and Watson, G. 1999. The deadline command. *IEE Proceedings—Software 146*, 2 (April), 104–111.

García, J. J. G. and Harbour, M. G. 2001. Towards a real-time distributed systems annex in Ada. *Ada Letters XXI*, 1 (March), 62–66.

Giering III, E. and Baker, T. 1994. A tool for deterministic scheduling of real-time programs implemented as periodic ada tasks. *ACM Ada Letters XIV*, 54–73.

Gomaa, H. 1993. *Software Design Methods for Concurrent and Real-Time Systems*. Addison-Wesley.

Grigg, A. and Audsley, N. C. 1999. Towards a scheduling and timing analysis solution for integrated modular avionics systems. *Microprocessors and Microsystems 22*, 423–431.

Hayes, I. J., Fidge, C. J., and Lermer, K. 2001. Semantic characterisation of dead control-flow paths. *IEE Proceedings—Software 148*, 6 (Dec.), 175–186.

Healy, C. A., Whalley, D. B., and Harmon, M. G. 1995. Integrating the timing analysis of pipelining and instruction caching. In *Proc. 16th IEEE Real-Time Systems Symposium* (Dec. 1995), pp. 288–297. IEEE Computer Society Press.

Honeywell Technology Center. 1998. *MetaH User's Manual* (1.27 ed.). Honeywell Technology Center. `http://www.htc.honeywell.com/metah/prodinfo.html`.

Hur, Y., Bae, Y. H., Lim, S.-S., Rhee, B.-D., Min, S. L., Park, C. Y., Lee, M., Shin, H., and Kim, C. S. 1995. Worst case timing analysis of RISC processors: R3000/R3010 case study. In *Proc. 16th IEEE Real-Time Systems Symposium* (Dec. 1995), pp. 308–319. IEEE Computer Society Press.

Jones, M. B., Barrera III, J. S., Forin, A., Leach, P. J., Rosu, D., and Rosu, M.-C. 1996. An overview of the Rialto real-time architecture. In *Proc. Seventh ACM SIGOPS European Workshop* (Sept. 1996).

Joseph, M. and Pandya, P. 1986. Finding response times in a real-time system. *The Computer Journal 29*, 5, 390–395.

Kalinsky, D. 2001. Context switch. *Embedded Systems Programming 14*, 2 (Feb.), 94–105.

Kristenssen, L. M., Billington, J., and Qureshi, Z. H. 2001. Modelling military airborne mission systems for functional analysis. In *Proceedings of the 20th Digital Avionics Systems Conference (DASC)* (Oct. 2001).

Kurki-Suonio, R. 1994. Real time: Further misconceptions (or half-truths). *IEEE Computer*, 71–76.

Lehoczky, J. P., Sha, L., and Ding, Y. 1989. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proc. Real-Time Systems Symposium* (1989), pp. 166–171. IEEE Computer Society Press.

Lim, S.-S., Bae, Y. H., Jang, G. T., Rhee, B.-D., Min, S. L., Park, C. Y., Shin, H., Park, K., Moon, S.-M., and Kim, C. S. 1995. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering 21*, 7 (July), 593–604.

Liu, C. L. and Layland, J. W. 1973. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 20*, 1, 46–61.

Locke, C. D. 1992. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *The Journal of Real-Time Systems 4*, 37–53.

Locke, C. D., Vogel, D. R., Lucas, L., and Goodenough, J. B. 1990. Generic avionics software specification. Technical Report CMU/SEI-90-TR-8 (Dec.), Software Engineering Insititute, Carnegie Mellon University.

Lundqvist, T. and Stenström, P. 1999. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems 17*, 2/3 (Nov.), 183–207.

Park, C. Y. 1993. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems 5*, 31–62.

Pilling, M., Burns, A., and Raymond, K. 1990. Formal specifications and proofs of inheritance protocols for real-time scheduling. *Software Engineering Journal 5*, 5 (Sept.), 263–279.

Puschner, P. and Koza, C. 1989. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems 1*, 2 (Sept.), 159–176.

Richard, P., Cottet, F., and Richard, M. 2001. On-line scheduling of real-time distributed computers with complex communication constraints. In *Proceedings of the Seventh IEEE International Conference on Engineering of Complex Computer Systems* (2001), pp. 26–34. IEEE Computer Society.

RTCA, Inc. 1992. *Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Inc. Special Committee 167 Document No. RTCA/DO-178B.

Sha, L., Rajkumar, R., and Lehoczky, J. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers 39*, 9 (Sept.), 1175–1185.

Shaw, A. C. 1989. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering 15*, 7 (July), 875–889.

Sprunt, B., Sha, L., and Lehoczky, J. 1989. Aperiodic task scheduling for hard real-time systems. *Journal of Real-Time Systems 1*, 1 (June), 27–60.

Stankovic, J. A. 1988. Misconceptions about real-time computing. *IEEE Computer 21*, 10 (Oct.), 10–19.

Stankovic, J. A. and Ramamritham, K. 1990. Editorial: What is predictability for real-time systems. *Journal of Real-Time Systems 2*, 4 (Nov.), 247–254.

Stewart, D. B. 2001. Beginner's corner: Real time. *Embedded Systems Programming 14*, 12 (Nov.), 87–88.

Stoyenko, A. D. and Baker, T. P. 1994. Real-time schedulability-analyzable mechanisms in Ada 9X. *Proceedings of the IEEE 82*, 1 (Jan.), 95–107.

Taft, S. T. and Duff, R. A. Eds. 1997. *Ada 95 Reference Manual: Language and Standard Libraries*, Volume 1246 of *Lecture Notes in Computer Science*. Springer-Verlag.

Theiling, H., Ferdinand, C., and Wilhelm, R. 2000. Fast and precise WCET prediction by separated cache and path analyses. *The International Journal of Time-Critical Computing Systems 18*, 2/3 (May), 157–179.

Tindell, K. 2000. Deadline monotonic analysis. *Embedded Systems Programming 13*, 6 (June), 20–38.

Tindell, K., Burns, A., and Wellings, A. J. 1994. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems 6*, 133–151.

Tindell, K. and Clark, J. 1994. Holistic schedulability analysis for distributed hard real-time systems. *Euromicro Journal 40*, 117–134. Special Issue on Parallel Embedded Real-Time Systems.

Williams, T. 1991. Performance pushes RISC chips into real-time roles. *Computer Design*, 79–86.