Efficient Exact Schedulability Tests for Fixed Priority Real-Time Systems

Robert Davis, Attila Zabos, and Alan Burns, Senior M*ember, IEEE*

Abstract— *Efficient exact schedulability tests are required both for on-line admission of applications to dynamic systems and as an integral part of design tools for complex distributed real-time systems. This paper addresses performance issues with exact Response Time Analysis (RTA) for fixed priority pre-emptive systems. Initial values are introduced that improve the efficiency of the standard RTA algorithm (i) when exact response times are required, and (ii) when only exact schedulability need be determined. The paper also explores modifications to the standard RTA algorithm, including; the use of a response time upper bound to determine when exact analysis is needed, incremental computation aimed at faster convergence, and checking tasks in reverse priority order to identify unschedulable tasksets early. The various initial values and algorithm implementations are compared by means of experiments on a PC recording the number of iterations required, and execution time measurements on a real-time embedded microprocessor. Recommendations are provided for engineers tasked with the problem of implementing exact schedulability tests, as part of on-line acceptance tests and spare capacity allocation algorithms, or as part of off-line system design tools.*

Index Terms— Multiprocessing / multiprogramming / multitasking, Real-time systems and embedded systems, Scheduling.

1 Introduction

FIXED priority pre-emptive scheduling is widely used in real-time embedded systems, and is supported by the majority of commercial real-time operating systems.

In the context of fixed priority pre-emptive systems, schedulability analysis is used to determine if a set of tasks can be guaranteed to always meet their deadlines at run-time.

A schedulability test is referred to as *sufficient* if all tasksets deemed to be schedulable by the test are in fact schedulable. Similarly, a schedulability test is referred to as *necessary* if all tasksets deemed to be unschedulable by the test are in fact unschedulable. Schedulability tests that are both *sufficient* and *necessary* are referred to as *exact*.

In this paper, we are concerned with exact schedulability tests for fixed priority pre-emptive systems. Although these tests are known to be pseudo-polynomial in complexity [5], [6], [11], the scale of many commercial systems is such that exact tests can be used.

Efficient, exact schedulability tests are required for, 1) admission of applications to dynamic systems at run-time, 2) design of complex real-time systems, where schedulability analysis forms part of some higher level process of system optimisation. Reducing the execution times of exact schedulability tests is an important consideration in these practical applications.

We can classify the requirements for exact schedulability tests as follows: *Boolean schedulability tests*: only a Boolean answer, either schedulable or not schedulable is required. *Response time tests*: in the case of a schedulable system, the exact worst-case response time of each task is required.

For on-line admission tests, a Boolean schedulability test is often sufficient; however, for use off-line, as part of a system design tool, response time tests are typically required. For example, in a distributed system based on Controller Area Network (CAN), the response times of tasks that read sensor data and then output information on CAN affect the release jitter of messages sent on the bus [25]. Knowing exact worst-case response times for the tasks makes possible accurate analysis of message worst-case response times, and hence derivation of exact end-to-end response times from input event to output response.

1.1 Motivation

The motivation for this research comes from the Frescor project [23]. The Frescor scheduling framework supports the execution of multiple applications on a single processor. Each application is executed within its associated periodic server which has a capacity, period, and deadline. The servers run under a fixed priority pre-emptive scheduler. Determining server schedulability is an analogous problem to computing the schedulability of a set of periodic/sporadic tasks.

Applications can be added to a Frescor system at run-time. Before a new application can be added, the admission test needs to check that all the existing servers remain schedulable, and that the additional server supporting the new application is also schedulable. Once an application has been admitted to the system, the scheduling framework must determine the amount of spare capacity to allocate to each of those applications requesting additional capacity. The spare capacity allocation algorithm makes multiple calls to a schedulability test to determine the feasibility of the system with respect to different allocations of spare capacity.

• *R.I. Davis, A. Zabos, and A. Burns are with the Department of Computer Science at the University of York, England. Email: {robdavis, attila, burns}@cs.york.ac.uk.*

To achieve the best possible performance in terms of the applications that can be admitted, and the spare capacity that can be allocated, it is therefore desirable to use an exact schedulability test. As the schedulability test must be carried out on-line, and completed before a new application can start, it is important that the schedulability test is as efficient as possible. This desire to provide an efficient and effective schedulability test for use in the Frescor scheduling framework motivates our research.

As the research presented in this paper is applicable to the widely used fixed priority pre-emptive tasking model, in the remainder of the paper, we will use the term *task* to mean the schedulable entity of interest, which in the case of Frescor may, in fact, be a server or virtual resource.

1.2 Related research

Research into schedulability tests for fixed priority pre-emptive systems effectively began in 1967, when Fineberg and Serlin [1] considered priority assignment for two tasks. They noted that if the task with the shorter period is assigned the higher priority, then the least upper bound on the schedulable utilisation is $2(\sqrt{2}-1)$ or 82.8%. This result was generalised by both Serlin [2] in 1972 and Liu and Layland [3] in 1973, both of whom showed that for *synchronous* tasks (that share a common release time), that comply with a restrictive system model, and that have deadlines equal to their periods $(D_i = T_i)$, then *rate monotonic*[1] priority ordering (RMPO) is optimal. Liu and Layland [3] provided the following sufficient schedulability test for tasks compliant with their model, and with priorities assigned according to RMPO:

$$\sum_{i=1..n} U_i \le n(2^{1/n}-1) \quad (1)$$

Where $U_i = C_i / T_i$ is the utilisation of task τ_i, C_i is an upper bound on the execution time of τ_i, and n is the number of tasks.

In 1982, Leung and Whitehead showed that *deadline monotonic*[2] priority ordering (DMPO) [4] is optimal for tasks with deadlines less than or equal to their periods $(D_i \le T_i)$. Exact response time tests were introduced by Joseph and Pandya [5] in 1986, and Audsley et al. [11] in 1993. An exact Boolean schedulability test was introduced by Lehoczky et al. [6] in 1989. Both forms of exact test have been extended to cater for cases where tasks access mutually exclusive shared resources according to mechanisms such as the Stack Resource Policy [9] and the Priority Ceiling Protocol (PCP) [7]. Further work on schedulability tests for fixed priority systems has lifted many of the earlier restrictions, providing exact tests for tasks with offset release times [10], arbitrary deadlines $(D_i > T_i)$ [8], [12], and non-pre-emptive sections [21]; these extensions are however beyond the scope of this paper.

Improvements to the performance of exact response time tests effectively began with Audsley [22] in 1993, who provided an initial value, for use in the recurrence relation used to compute task worst-case response times, that was based on the response time of the next higher priority task.

In 1998 Sjodin and Hansson [13] extended Audsley's work, by accounting for blocking factors in the initial value calculation. They also introduced a closed form lower bound on the response time that could be used as an effective initial value. Sjodin and Hansson showed that these initial values lead to fewer iterations of the recurrence relation and quantified the improvements in algorithm performance.

In 2003, Bril et al. [14] considered online response time calculations using similar initial values to those introduced by Audsley, and Sjodin and Hansson.

The initial values used by Audsley [22], Sjodin and Hansson [13], and Bril et al. [14] are all lower bounds on the worst-case response time, thus exact worst-case response times can be found starting from these values.

In 2007, Lu et al. [17] introduced two new "deadline dependent" initial values, which can be used to determine exact schedulability, but cannot in general be used to find exact worst-case response times. Lu et al. showed that significant efficiency gains are possible using these new initial values combined with previous ones.

Previous research by Lu et al. in 2006 [16] sought to improve the performance of response time analysis by partitioning higher priority tasks into two sets. Interference from one set of tasks was then treated as consuming execution time according to their utilisation, leaving a fraction of the processor available for computation due to the remaining tasks. This approach reduced the number of iterations of the algorithm required for convergence; however, this came at the expense of requiring the use of floating point types. On most hardware platforms, the use of floating point is considerably slower than integer arithmetic, even when floating point hardware is available. For example, on the PowerPC (MPC555) microprocessor, code for the standard response time test is approximately 2.5 times slower using floating point rather than integer types. This difference in efficiency effectively negates the apparent speed ups reported in [16].

Related work by Bini and Buttazzo introduced the Hyperplanes Exact Test (HET) [19] in 2004. The Hyperplanes Exact Test provides a means of improving the efficiency of the exact schedulability test formulated by Lehoczky et al. in [6], via a reduction in the number of points in time at which the workload needs to be evaluated. Recent research by the authors [28], reproduced in the appendix, shows that contrary to the findings in [19], the Hyperplanes Exact Test is not in general as computationally efficient as the exact Response Time Analysis tests discussed in this paper.

The research presented in this paper builds upon the work of Sjodin and Hansson [13], and Bril et al. [14]. It takes the concept of task partitioning introduced by Lu et al. [16] and uses it to form a new series of initial values that can be used in exact response time tests. The research also builds upon the work of Lu et al. [17] providing two improved initial values for use in exact Boolean schedulability tests.

[1] RMPO assigns priorities in order of task periods, such that the task with shortest period is given the highest priority.

[2] DMPO assigns priorities in order of task deadlines, such that the task with the shortest deadline is given the highest priority.

1.3 Organisation

Section 2 gives the system model, terminology and notation used in the rest of the paper, along with a recapitulation of the standard Response Time Analysis (RTA) recurrence relation. Section 3 introduces a new series of initial values, the largest of which can be used to compute exact worst-case response times. Section 4 presents two improved initial values that can be used in Boolean schedulability tests. Section 5 discusses improvements to the schedulability test algorithm, including an incremental approach, the use of a response time upper bound ensuring that exact schedulability computation is only performed when necessary, and reversing the order in which task schedulability is checked with the aim of identifying unschedulable tasks more quickly. Section 6 outlines an empirical investigation into schedulability test efficiency. This is complemented by Section 7 which provides execution time measurements from an implementation of the tests on an embedded microprocessor. Section 8 gives our recommendations to engineers tasked with implementing exact schedulability tests. Finally, Section 9 concludes with a summary of the main contributions of this paper, and an outline of areas for future research.

2 System model and basic analysis

2.1 Terminology and notation

In this paper, we are interested in providing efficient, exact schedulability tests for applications executing under a fixed priority pre-emptive scheduler on a single processor. The application is assumed to comprise a static set of n tasks ($\tau_1..\tau_n$), each assigned a unique priority i, from 1 to n (where n is the lowest priority).

We use the notation *hp(i)* and *lp(i)* to mean the set of tasks with priorities higher than i, and the set of tasks with priorities lower than i respectively. Similarly, we use the notation *hep(i)* and *lep(i)* to mean the set of tasks with priorities higher than or equal to i, and lower than or equal to i respectively.

Application tasks may arrive either *periodically* at fixed intervals of time, or *sporadically* after some minimum inter-arrival time has elapsed. Each task τ_i, is characterised by: its relative *deadline* D_i, *worst-case execution time* C_i, minimum inter-arrival time or *period* T_i, and *release jitter* J_i, defined as the maximum time between the task arriving and it being released (ready to execute). It is assumed that once a task starts to execute it will never voluntarily suspend itself.

Tasks may access shared resources in mutual exclusion according to the Stack Resource Policy (SRP) [9]. A task at priority i may be blocked by lower priority tasks, as a result of the operation of the SRP, for at most B_i, referred to as the *blocking time*.

A task's *worst-case response time* R_i, is the longest time from the task becoming ready to execute to it completing execution. A task is referred to as *schedulable* if its worst-case response time is less than or equal to its deadline less release jitter ($R_i \le D_i - J_i$). A system is referred to as schedulable if all its tasks are schedulable.

We assume that task deadlines are less than or equal to their periods $D_i \le T_i$, and without loss of generality that task priorities are in *deadline minus jitter monotonic*[3] (D-JMPO) priority order [20].

2.2 Busy periods and idle instants

The concept of a *busy period*, introduced by Lehoczky in [8], is fundamental in analysing worst-case response times. The following concepts are used in the analysis presented in this paper.

A *priority level-i idle instant* is defined as a time instant t at which there are no tasks of priority i or higher awaiting execution that became ready to execute strictly before time t.

A *priority level-i busy period* is defined as follows:

1. It starts at a priority level-i idle instant t^s, when a task of priority i or higher becomes ready to execute.
2. It is a contiguous interval of time during which any task of priority lower than i is unable to start executing.
3. It ends at the first priority level-i idle instant t^e, following t^s.

A *critical instant* [3], for task τ_i, is defined as a time at which task τ_i becomes ready to execute, and is then subject to the maximum possible delay, i.e. its worst-case response time, before completing execution. For tasks complying with the system model outlined above, a critical instant occurs when task τ_i becomes ready to execute simultaneously with all tasks of higher priority, and subsequent invocations of these higher priority tasks become ready as soon as possible. Further, immediately before task τ_i is released, a lower priority tasks locks a resource with a ceiling priority of i or higher, resulting in the maximum blocking time B_i. For this system model, the worst-case response time of task τ_i is equivalent to the length of the longest priority level-i busy period.

2.3 Basic response time analysis

Response time analysis [5], [11], [12] calculates the length of the longest priority level-i busy period and hence the worst-case response time of task τ_i, using the following equation.

$$R_i = B_i + C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \qquad (2)$$

Note that the worst-case response time R_i appears on both the left and right hand side of Equation (2). As the right hand side is a monotonically non-decreasing function of R_i, the equation can be solved using the following recurrence relation:

$$r_i^{n+1} = B_i + C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i^n + J_j}{T_j} \right\rceil C_j \qquad (3)$$

Iteration starts with an initial value r_i^0, typically $r_i^0 = B_i + C_i$, and ends when either $r_i^{n+1} = r_i^n$ in which case the worst-case response time R_i, is given by r_i^{n+1} or

[3] D-JMPO assigns priorities in order of deadline minus jitter, such that the task with the smallest value of D_i-J_i is given the highest priority.

when $r_i^{n+1} > D_i - J_i$ in which case the task is unschedulable.

In general, Equation (2) may have a number of different solutions, the smallest of which corresponds to the worst-case response time R_i. The recurrence relation is guaranteed to converge provided that the taskset utilisation is <1. It is guaranteed to converge on the smallest solution R_i, if and only if the initial value r_i^0 is less than or equal to R_i, thus any initial value $\leq R_i$ will suffice to determine the exact value of R_i. Further, for any two initial values r^A and r^B where $r^A \leq r^B \leq R_i$, then the number of iterations N^A, of the recurrence relation required to converge on the solution R_i from initial value r^A is at least as great as the number of iterations N^B, required when starting from initial value r^B. Stated otherwise, the largest possible initial value $\leq R_i$ will result in the least number of iterations and hence the fastest possible convergence.

2.4 Performance metrics

A number of different metrics could be used to explore the performance of the recurrence relation:

1. Number of i*terations* of the recurrence relation required for convergence.
2. Total number of *ceiling operations* required for convergence.
3. *Execution time* of a specific implementation on a particular microprocessor.

In their experiments, Sjodin and Hansson [13], and Lu et al. [17] used the number of iterations of the recurrence relation as a performance metric. By contrast, Bril et al. [14] used the number of ceiling operations. We argue that the latter is a better metric, as each iteration requires a variable number of ceiling operations dependent on the priority of the task. Thus using iterations as a measure could potential skew the results, if for example, a particular approach required less iterations for high priority tasks, but more for those of low priority.

In our empirical investigations, in Section 6, we use the number of ceiling operations as a performance metric, and as a simple proxy for the later execution time measurements made in Section 7.

3 Initial values for exact response time tests

In this section, we consider initial values for exact response time tests.

3.1 Previous work

In 1993, in chapter 4 of his thesis [22], Audsley showed that, for systems of independent tasks, with task schedulability tested in priority order, $R_{i-1} + C_i$ is an effective initial value.

In 1998, Sjodin and Hansson [13] extended this lower bound on R_i to account for blocking:

$$R_i^{LB} = R_{i-1} - B_{i-1} + B_i + C_i \tag{4}$$

By approximating the ceiling function in the recurrence relation by a division operation, Sjodin and Hansson [13] also introduced the following closed form lower bound:

$$R_i^{LB} = \frac{B_i + C_i + \sum_{\forall j \in hp(i)} J_j U_j}{1 - \sum_{\forall j \in hp(i)} U_j} \tag{5}$$

In [14], Bril et al. derived essentially the same lower bounds for a simple scheduling model, assuming no jitter or blocking:

$$R_i^{LB} = R_{i-1} + C_i \tag{6}$$

$$R_i^{LB} = \frac{C_i}{1 - \sum_{\forall j \in hp(i)} U_j} \tag{7}$$

We note that using the initial values given by Equations (4) and (6) requires that task response times are determined in priority order, highest priority first.

3.2 New initial values

We now introduce a series of new initial values, the maximum of which can be used to provide a later starting point, reducing the number of iterations required by the recurrence relation.

For each priority level *i*, there are *i* initial values in the series. To form each new initial value, identified by the index *k* (*k*=1…*i*), we partition the set of tasks of higher than or equal priority to *i*, i.e. *hep(i)*, into two sets: *hp(k)* and $lep(k) \cap hep(i)$.

Following the approach of Lu et al. in [16], we consider the tasks in *hp(k)* as taking a proportion of the available processing time α_k where:

$$\alpha_k = \sum_{\forall j \in hp(k)} U_j \tag{8}$$

Thus only a fraction of the processing time $1 - \alpha_k$ remains available to accommodate the remaining task load. Given that $R_i \geq R_{i-1}$, the contribution of each task in *hep(i)* to the total task load in R_i is at least $I_j(R_{i-1})$, where $I_j(R_{i-1})$ is the worst-case interference due to task $\tau_j \in hp(i)$ occurring during the response time of task τ_{i-1}.

$$I_j(R_{i-1}) = \left\lceil \frac{R_{i-1} + J_j}{T_j} \right\rceil C_j \tag{9}$$

We note that as the response time of task τ_i is only computed if task τ_{i-1} is schedulable, $I_{i-1}(R_{i-1}) = C_{i-1}$.

Using this information, we compose a series of *i* lower bounds on R_i corresponding to each priority *k* from 1 to *i*.

$$R_i^{LB}(k) = \frac{B_i + C_i + \sum_{\forall j \in lep(k) \cap hp(i)} I_j(R_{i-1}) + \sum_{\forall j \in hp(k)} J_j U_j}{1 - \sum_{\forall j \in hp(k)} U_j} \tag{10}$$

The largest such bound is given by:

$$R_i^{LB} = \max_{\forall k = 1..i} (R_i^{LB}(k)) \tag{11}$$

We note that this set of lower bounds has as its first and last members, the two initial values proposed in [13] and [14]; and hence subsumes and dominates the bounds given by Equations (4) to (7).

For k = 1:

$$R_i^{LB}(1) = B_i + C_i + \sum_{\forall j \in hp(i)} I_j(R_{i-1})$$
$$= B_i + C_i + C_{i-1} + \sum_{\forall j \in hep(i)} I_j(R_{i-1})$$
$$= B_i + C_i + R_{i-1} - B_{i-1} \quad (12)$$

which is equivalent to Equations (4) and (6).

For $k = i$:

$$R_i^{LB}(i) = \frac{B_i + C_i + \sum_{\forall j \in hp(i)} J_j U_j}{1 - \sum_{\forall j \in hp(i)} U_j} \quad (13)$$

which is equivalent to Equations (5) and (7).

We observe that the new initial value given by Equation (11) is not optimal, as larger values are possible, up to R_i, which still result in an exact response time test.

3.3 Examples

We now show using a simple example that the lower bound calculated via Equation (11) can be greater than the lower bounds computed via Equations (4) to (7). The example is based on the task parameters given in Table 1 below. The overall utilisation of this taskset is 92.5%. The tasks are assumed to be independent, and have zero release jitter. Note, the final column in the table, headed U*, is the cumulative utilisation for all higher priority tasks.

TABLE 1: TASK PARAMETERS

Priority	C	D	T	U	U*
1	5	10	10	0.5	0
2	25	100	100	0.25	0.5
3	25	200	200	0.125	0.75
4	30	1000	1200	0.025	0.875
5	30	1200	1200	0.025	0.9

Our example considers the lower bounds (initial values) for the calculation of R_5. Assuming that response times are calculated in priority order, we have the following information available from the calculation of R_4: $R_4 = 360$, comprising $I_1(R_4) = 180$, $I_2(R_4) = 100$, $I_3(R_4) = 50$, and $I_4(R_4) = C_4 = 30$.

The series of lower bounds $R_5^{LB}(k)$ are therefore:

$R_5^{LB}(0) = (I_1 + I_2 + I_3 + I_4 + C_5)/1 = 390$
$R_5^{LB}(1) = (I_2 + I_3 + I_4 + C_5)/0.5 = 420$
$R_5^{LB}(2) = (I_3 + I_4 + C_5)/0.25 = 440$
$R_5^{LB}(3) = (I_4 + C_5)/0.125 = 480$
$R_5^{LB}(4) = C_5/0.1 = 300$

The largest such bound, $R_5^{LB}(3)$, is 480. This is a significant improvement on the previous bounds of 390 and 300 given by Sjodin and Hansson [13], and Bril et al. [14]. In this example, $R_5 = 570$, which takes 7 iterations to calculate starting with an initial value of 480, or 10 iterations starting with an initial value of 390.

4 INITIAL VALUES FOR EXACT BOOLEAN SCHEDULABILITY TESTS

In this section, we consider initial values for exact Boolean schedulability tests.

4.1 Previous work

In 2007, Lu et al. [17] introduced two "deadline dependent" initial values $D_i/2$ and $D_i - D_{i-1}$ that can be used as a starting point for the recurrence relation. Unlike all of the initial values discussed in Section 3, these initial values are not necessarily lower bounds on R_i and so do not guarantee that the recurrence relation will converge on the first solution (i.e. the exact worst-case response time). However, in their analysis Lu et al. showed that using these initial values the recurrence relation is guaranteed to converge on some upper bound R_i^{UB}, such that $R_i \le R_i^{UB} \le D_i$, if τ_i is in fact schedulable. Hence these initial values can be used to implement an exact Boolean schedulability test, but not an exact response time test.

4.2 New initial values

We now build upon the work of Lu et al. [17], extending their initial values so that they are applicable to the more general case of systems with blocking and release jitter. We then derive improved initial values that dominate those introduced by Lu et al.

Theorem 1. *The initial value $(D_i - J_i) - (D_{i-1} - J_{i-1})$ guarantees that the recurrence relation will converge on a solution R_i^{UB}, where $R_i \le R_i^{UB} \le D_i - J_i$ if τ_i is in fact schedulable.*

Proof. We assume that task schedulability is determined in priority order and therefore that τ_{i-1} is schedulable. As τ_{i-1} is schedulable, there must be at least one priority level *i*-1 idle instant in any interval of length R_{i-1} or greater (such as $D_{i-1} - J_{i-1}$).

To prove the theorem, there are two cases to consider:

1. τ_i is schedulable with $R_i \ge (D_i - J_i) - (D_{i-1} - J_{i-1})$. In this case, the initial value $(D_i - J_i) - (D_{i-1} - J_{i-1})$ is less than the first solution to the recurrence relation, and so the equation is guaranteed to converge on R_i.
2. τ_i is schedulable with $R_i < (D_i - J_i) - (D_{i-1} - J_{i-1})$. In this case, as τ_i has completed execution by $(D_i - J_i) - (D_{i-1} - J_{i-1})$, only tasks in the set *hp(i)* are able to execute after this point. As the longest priority level-(*i*-1) busy period is known to be of length $R_{i-1} \le D_{i-1} - J_{i-1}$, a further idle instant must occur by $(D_i - J_i) - (D_{i-1} - J_{i-1}) + R_{i-1} \le D_i - J_i$. The recurrence relation is therefore guaranteed to converge on some value R_i^{UB}, where $R_i \le R_i^{UB} \le D_i - J_i$ □

We now improve on the initial value given by Theorem 1.

Theorem 2. *The initial value $(D_i - J_i) - R_{i-1}^{UB}$, where R_{i-1}^{UB} is an upper bound on the response time of τ_{i-1}, and τ_{i-1} is known to be schedulable, so $R_{i-1}^{UB} \le (D_{i-1} - J_{i-1})$, guarantees that the recurrence relation will converge on a solution R_i^{UB}, where $R_i \le R_i^{UB} \le D_i - J_i$ if τ_i is in fact schedulable.*

Proof. Follows directly from the proof of Theorem 1 □

As these methods do not guarantee to find exact response times, we use an upper bound R_{i-1}^{UB} in Theorem 2. If an exact response time is known for τ_{i-1}, then this provides the best possible upper bound.

Assuming that task schedulability is determined in priority order, then an appropriate value for R_{i-1}^{UB} is found as a result of computing the schedulability of τ_{i-1}; hence, no additional computation is needed to determine the upper bounds, their computation naturally forms part of the overall schedulability test.

Theorem 3. *The starting value* $(D_i - J_i)/2$ *guarantees that the recurrence relation will converge on a solution* R_i^{UB}, *where* $R_i \le R_i^{UB} \le D_i - J_i$ *if* τ_i *is in fact schedulable.*

Proof. There are two cases to consider:

1. τ_i is schedulable with $R_i \ge (D_i - J_i)/2$. In this case, the initial value $(D_i - J_i)/2$ is less than the first solution of the recurrence relation, and so the equation is guaranteed to converge on R_i.
2. τ_i is schedulable with $R_i < (D_i - J_i)/2$. In this case, as τ_i has completed execution by $(D_i - J_i)/2$, only tasks in the set *hp(i)* are able to execute after this point. As the longest priority level-(*i*-1) busy period is known to be of length $R_{i-1} \le R_i \le (D_i - J_i)/2$, a further idle instant must occur by $D_i - J_i$. The recurrence relation is therefore guaranteed to converge on some value R_i^{UB}, where $R_i \le R_i^{UB} \le D_i - J_i$ □

We now improve on the initial value given by Theorem 3.

Theorem 4. *The initial value* $(D_i - J_i + C_i + B_i)/2$ *guarantees that the recurrence relation will converge on a solution* R_i^{UB}, *where* $R_i \le R_i^{UB} \le D_i - J_i$ *if* τ_i *is in fact schedulable.*

Proof. There are again two cases to consider:

1. τ_i is schedulable with $R_i \ge (D_i - J_i + C_i + B_i)/2$. In this case, the initial value $(D_i - J_i + C_i + B_i)/2$ is less than the first solution to the recurrence relation, and so the equation is guaranteed to converge on R_i
2. τ_i is schedulable with $R_i < (D_i - J_i + C_i + B_i)/2$. In this case, the interference due to higher priority tasks in R_i is at most $(D_i - J_i - C_i - B_i)/2$, the additional time being accounted for by blocking B_i and the execution time C_i of task τ_i itself. Hence the longest possible priority level-(*i*-1) busy period comprising only execution of tasks in the set *hp(i)* is of length $(D_i - J_i - C_i - B_i)/2$. This means that there must exist a priority level-*i* idle instant in the interval between $(D_i - J_i + C_i + B_i)/2$ and $(D_i - J_i + C_i + B_i)/2 + (D_i - J_i - C_i - B_i)/2 = D_i - J_i$. The recurrence relation is therefore guaranteed to converge on some value R_i^{UB}, where $R_i \le R_i^{UB} \le D_i - J_i$ □

We note that as the initial values given by Theorems 1 to 4 may be larger than the exact worst-case response time R_i, these initial values cannot be used to calculate exact worst-case response times, only to provide exact Boolean schedulability tests.

Using the initial values given by Theorems 1 to 4, the next value generated by the recurrence relation may, in some cases, be smaller than the initial value. If so, iteration can be terminated immediately, as the task is then known to be schedulable, with the value computed on the 1st iteration providing an upper bound R_i^{UB} on the worst-case response time.

We observe that the initial value given by Theorem 4 can be considered optimal in the sense that the initial value is tight; any increase in this value could in the general case result in the schedulability test ceasing to be exact. The initial value given by Theorem 2, is similarly optimal provided that $R_i^{UB} = R_i$; however, larger, more pessimistic values of R_i^{UB} make the bound given by Theorem 2 less precise.

4.3 Example

In this section, we use the simple example taskset described in Table 2 to illustrate the operation of the initial values given by Theorems 2 and 4.

Here we use the maximum of the two initial values: (i) $(D_i - J_i) - R_{i-1}^{UB}$ and (ii) $(D_i - J_i + C_i + B_i)/2$ as a starting point for the recurrence relation. These values are shown in Table 2, along with the computed upper bound R_i^{UB} on the worst-case response time of each task. These upper bounds were calculated in priority order. Note that the exact worst-case response times are not calculated by this method; however, they are shown for comparison purposes in the final column of the table.

TABLE 2: INITIAL VALUES AND UPPER BOUND

				Initial values			
Pri	**C**	**D**	**T**	**(i)**	**(ii)**	R_i^{UB}	R_i
1	5	10	10	-	-	5	5
2	100	800	800	795	450	500	200
3	200	1000	1000	500	600	600	600

We note that the large initial value of 795 for τ_2 resulted in the recurrence relation terminating on its first iteration, giving an upper bound R_2^{UB} of 500. In the case of τ_3, the initial value of 600 also enabled task schedulability to be determined in a single iteration, compared to 5 iterations using the initial values given in [17].

5 SCHEDULABILITY TEST EFFICIENCY

In this section, we outline three other methods of improving the efficiency of exact schedulability tests based on Response Time Analysis, aside from using appropriate initial values:

1. Using a sufficient schedulability test to quickly determine, on a task-by-task basis, if an exact schedulability calculation is required. This approach is only applicable to Boolean schedulability tests where exact response times are not required.
2. Alternative implementations of the recurrence relation. This approach is applicable to both Boolean schedulability tests and response time tests.
3. Checking task schedulability in reverse priority order. This approach aims to identify unschedulable tasks early, reducing the amount of computation required when a taskset is unschedulable. This approach is applicable to both exact Boolean schedulability tests and exact response time tests.

5.1 Sufficent schedulability test

The use of a suitable sufficient schedulability test on a task-by-task basis can in theory improve the efficiency of

an exact Boolean schedulability test. If a task is schedulable according to the sufficient test, then exact schedulability does not need to be calculated, short circuiting a significant amount of computation. Note this approach is not appropriate if exact response times are required.

A number of simple sufficient schedulable tests have appeared in the literature. These include the Utilisation Bound [3], the Hyperbolic Bound [26], the Utilisation RBound [27], and the response time upper bound [18]. The complexity of applying these tests to n tasks is O(n) – if applied at each priority level, O(n), O($n$log$n$), and O(n) respectively.

In [18], Bini and Baruah introduced the response time upper bound, and compared its performance to that of the Utilisation Bound, the Hyperbolic Bound, and the Utilisation RBound. They showed that the performance of the response time upper bound was superior to the other sufficient tests for n>10 and also when the task period dispersion, given by $\max_i(T_i / T_{i-1})$ was greater than 2.

As we are interested in improving the performance of an exact Boolean schedulability test in those cases where it can require a large number of iterations, (typically large n and a wide range of task periods), then we use the response time upper bound, introduced by Bini and Baruah and reproduced in Equation (14) below, as our sufficient schedulability test of choice.

$$R_i^{ub} = \frac{C_i + \sum_{\forall j \in hp(i)} C_j (1 - U_j)}{1 - \sum_{\forall j \in hp(i)} U_j} \qquad (14)$$

It is interesting to note that, although using Equation (14) alone as a sufficient schedulability test results in poor performance for tasksets with high utilisation, there are still a significant number of individual tasks that are schedulable according to Equation (14), even if the taskset as a whole is not.

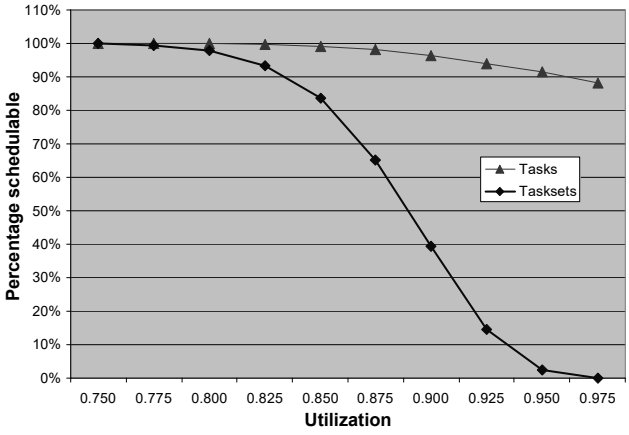

Fig. 1: Percentage of tasks and tasksets that were deemed to be schedulable by the response time upper bound, Equation (14).

This is illustrated by Fig. 1 which shows the percentage of tasksets, all of which are schedulable according to an exact test, that are deemed schedulable using a sufficient test based on Equation (14). Whilst performance of the sufficient test rapidly tails off above 80% utilisation, the number of individual tasks deemed schedulable remains high (over 85% at 97.5% utilisation). Note this data is based on averages over 10,000 tasksets for each utilisation level, with each taskset comprising 24 tasks and having a range of task periods spanning 4 orders of magnitude. See Section 6 for further details of the tasksets used.

5.2 Algorithm implementations

A standard implementation of the recurrence relation given by Equation (3) is shown in Fig. 2 below. This C code fragment computes the response time of the task at priority i.

We note that in the standard implementation, although the computed response time may effectively increase each time line 7 is executed, these increases are not reflected in the value of the variable `rprev` passed to the `ceiling()` function until the next iteration of the `while` loop. With this in mind, the number of iterations of the `for` loop needed for convergence can be reduced by the alternative implementation shown in Fig. 3.

```
1   rprev = 0;
2   r = initial_value();
3   while((r > rprev) && (r <= tasks[i].D)) {
4     rprev = r;
5     r = tasks[i].C;
6     for(j = 0; j < i; j++) {
7       r+=ceiling(rprev,tasks[j].T)*tasks[j].C;
8     }
9   }
```

Fig. 2: Standard implementation of the RTA algorithm.

The alternative implementation (Fig. 3) records in the variable `inter[j]`, the amount of interference due to task j, that has already been accounted for in the response time of task i. This facilitates incremental increases to `r` on each iteration of the second `for` loop (lines 10-14).

```
1   rprev = initial_value();
2   r = tasks[i].C;
3   for(j = 0; j < i; j++) {
4     inter[j] = ceiling(rprev,tasks[j].T)
5                 * tasks[j].C;
6     r += inter[j];
7   }
8   while((r > rprev) && (r <= tasks[i].D)) {
9     rprev = r;
10    for(j = 0; j < i; j++) {
11      tmp = ceiling(r,tasks[j].T)* tasks[j].C;
12      r += (tmp - inter[j]);
13      inter[j] = tmp;
14    }
15  }
```

Fig. 3: Incremental implementation of the RTA algorithm.

Note, in both implementations, it is assumed that the tasks are in priority order. Blocking factors and jitter terms are omitted for the sake of simplicity; however these can easily be included in either method.

5.3 Order in which tasks are examined

It can be argued that when a schedulability test is used as an online acceptance test, it does not matter how long the schedulability test takes to determine the schedulability of *unschedulable* tasksets. The schedulability test can always be suspended if it is taking too long, and the taskset

deemed unschedulable (which is correct). What does matter is how long the test takes to determine the schedulability of *schedulable* tasksets. In this case suspending computation before an answer is available would mean wrongly classifying the taskset as unschedulable. As the schedulability of all tasks needs to be checked before a taskset can be shown to be schedulable, the amount of computation required by the schedulability test for *schedulable* tasksets is effectively independent of the order in which the tasks are examined. The initial value calculations may however require a particular ordering.

An alternative use of exact schedulability tests is as part of a design time tool, or online spare capacity allocation algorithm. In these cases, a binary search may be used to determine the maximum/minimum values of C_i and T_i that can be supported for each task or server. In this case, it is reasonable to expect approximately 50%, of the parameter sets put forward for testing to be unschedulable. Further, it is important to determine both schedulability and unschedulability efficiently, so that the higher level algorithm can make rapid progress towards its goal. In this case, checking schedulability in reverse priority order may be more effective. This is because lower priority tasks are more likely to be unschedulable than those of higher priority. Once a single task has been shown to be unschedulable then further computation can be abandoned, as the taskset as a whole is unschedulable.

Checking task schedulability in reverse priority order is only possible using some of the initial values discussed. The initial values given by Equations (5) and (7), and Theorems 1, 3 and 4, do not depend on the order in which task schedulability is checked, and so may be used when schedulability is determined in reverse priority order. By contrast, the initial values given by Equations (4), (6), and (11), and Theorem 2 rely on knowing the response time, or an upper bound on the response time, of the next highest priority task. These initial values cannot be used when task schedulability is checked in reverse priority order.

As a simple example of the effectiveness of testing task schedulability in reverse priority order, consider the taskset described in Table 1, but with modified deadlines $D_4 = 400$ and $D_5 = 550$, so that task τ_5 is unschedulable. Using the default initial value, determining that the taskset is unschedulable takes 48 ceiling operations in reverse priority order, compared with 107 ceiling operations in forward priority order.

6 Empirical Investigation

In this section, we describe an empirical investigation into the effectiveness of using the initial values introduced in Sections 3 and 4, and the sufficient test, algorithm implementations, and task orderings discussed in Section 5.

For ease of reference, Table 3 provides a summary of the initial values and algorithm options used in our experiments. In the remainder of this section and in Section 7, we refer to the initial values and algorithm options used by these numbers, thus #1 refers to the standard algorithm implementation given in Fig. 2, using the default initial value C_i, whilst #9 refers to using a sufficient test based on Equation (14) to determine when exact analysis is required, and then an exact test using the standard algorithm implementation, with the initial value given by the maximum of Equation (7), Theorem 2 and Theorem 4. The rationale for combining these three values is that larger initial values are more effective. As none of the three values dominates the others, taking the maximum provides the largest initial value that can be computed with low overhead, and without needing to know the exact response time of the next highest priority task.

TABLE 3: RTA OPTIONS: INITIAL VALUES AND ALGORITHMS

	Initial Value	Algorithm
#1	Default (C_i)	Standard algorithm (see Fig. 2).
#2	Equation (7)	
#3	Equation (6)	
#4	Max of Eqs. (6) and (7)	
#5	Equation (11)	
#6	Theorem 1	
#7	Theorem 2	
#8	Theorem 4	
#9	Max of Equation (7) and Theorems 2 and 4	Sufficient test then standard algorithm.
#10	Default (C_i)	Incremental algorithm (see Fig. 3).
#11	Equation (11)	

The experiments described in this section were performed on a PC, enabling results to be obtained for large numbers of randomly generated tasksets. In Section 7, we complement this data with execution time measurements recorded on an embedded microprocessor.

The task parameters used in our experiments were randomly generated as follows: Of the n tasks in each taskset, n/M tasks were assigned to each of the M order of magnitude ranges used (e.g. 1000-10000, 10000-100000, 100000-1000000, 1000000-10000000 etc). Task periods were then determined according to a uniform random distribution, from the assigned range. This was done both to replicate the type of period distributions found in commercial real-time systems (by varying M from 2 to 6), and also to enable an investigation into how the efficiency of response time analysis depends on the overall range of task periods. In all cases, task deadlines were set equal to their periods, and blocking and jitter were set equal to zero[4].

For each utilisation level studied, the UUniFast algorithm [15] was used to determine individual task utilisations U_i, and hence task execution times, $C_i = U_i T_i$, given the previously selected task periods. 100,000 tasksets were generated in all, 10,000 for each utilisation level.

We used the number of ceiling operations as a performance metric to compare the different approaches and as a simple proxy for schedulability test execution time.

The results of Experiments 1-3 below record the number of ceiling operations required to determine the schedulability of *schedulable* tasksets. This avoids skewing the data, due to the significant numbers of unschedulable

[4] Increased values of blocking and release jitter only decrease the interval between the initial values (e.g. Equations (5) & (10), Theorem 4 etc.) and the termination condition $r_i \geq D_i - J_i$, and hence tend to reduce the number of iterations required for convergence compared to an equivalent task parameter set with blocking and release jitter equal to zero.

tasksets that occur at high levels of utilisation. Experiment 4 examines this effect and shows how the time taken to determine the schedulability of unschedulable tasksets depends strongly on the order in which tasks are checked.

6.1 Experiment 1

This experiment investigated the efficiency of exact schedulability tests used to determine the feasibility of tasksets comprising 24 tasks with periods spanning 4 orders of magnitude, and overall utilisations varying from 75% to 97.5%.

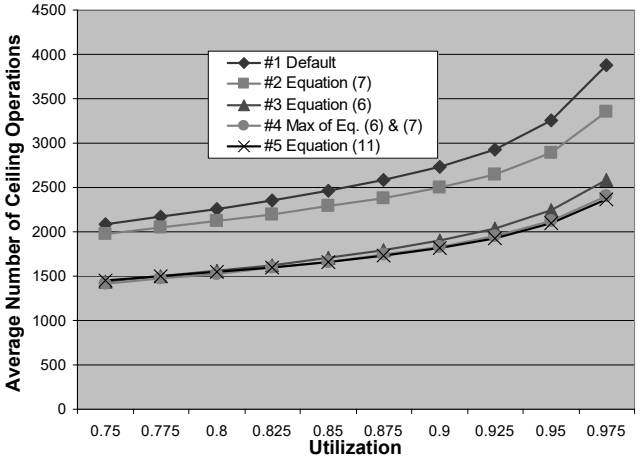

Fig. 4: Average number of "ceiling operations" required by exact response time tests v. taskset utilisation. Data is for the standard RTA algorithm, and the initial values from Section 3.

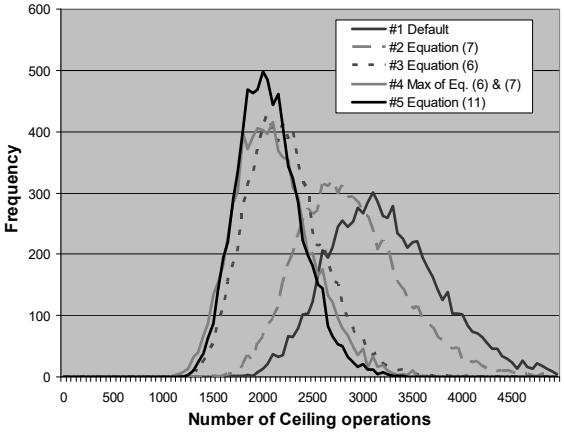

Fig. 5: Frequency distribution of the number of "ceiling operations" required by exact response time tests using the initial values from Section 3. Data is for 10,000 tasksets with 95% utilisation.

Fig. 4 shows that the most efficient initial values to use are #5, given by Equation (11), and #4, given by the maximum of Equations (6) and (7). In fact, when initial value #5 is used, the schedulability test itself requires significantly fewer ceiling operations; however, once the additional $n(n\text{-}1)/2$ ceiling operations involved in computing the initial value itself are taken into account (as it has been done in Fig. 4), performance is reduced to a similar level to that obtained using initial value #4.

Fig. 5 illustrates the frequency distribution of the number of ceiling operations required by the schedulability test for each of the 10,000 tasksets with 95% utilisation. Fig. 5 shows that using initial values #4 and #5 result in, lower maximums, narrower frequency distributions, and smaller averages than using the default initial value #1.

Fig. 6 is similar to Fig. 4; however, it shows the average number of ceiling operations required using the initial values described in Section 4. Fig. 6 shows that using the two new initial values #7 and #8 given by Theorems 2 and 4 results in significantly improved performance. For utilisation levels from 75-85%, the algorithm requires just 1 or 2 iterations of the `while` loop on average. This is because in many cases, the value computed on the 1st iteration is less than the initial value resulting in an immediate exit.

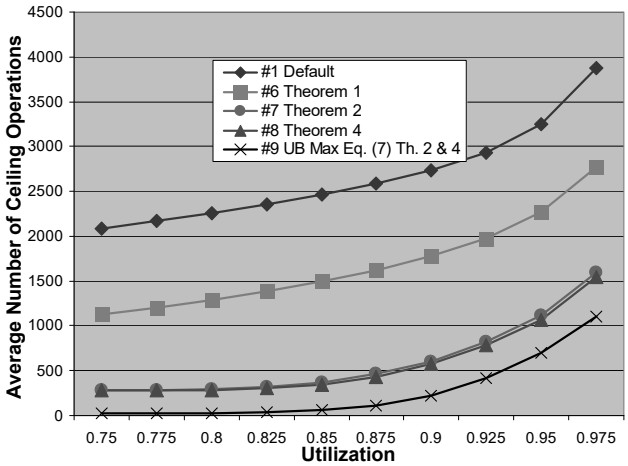

Fig. 6: Average number of "ceiling operations" required by exact Boolean schedulability tests v. taskset utilisation. Data is for the standard RTA algorithm, and the initial values from Section 4.

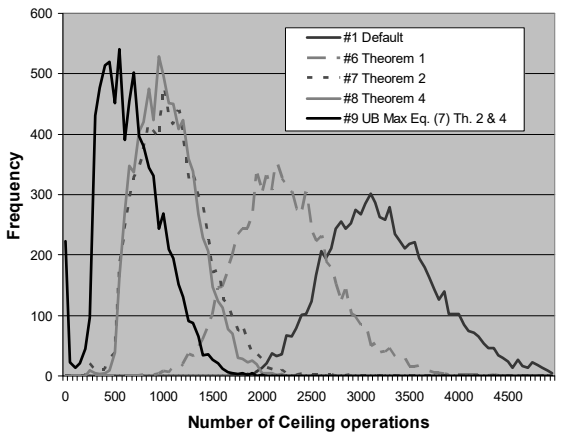

Fig. 7: Frequency distribution of the number of "ceiling operations" required by exact Boolean schedulability tests using the initial values from Section 4. Data is for 10,000 tasksets with 95% utilisation.

Fig. 6 shows that using the response time upper bound combined with the initial values from Theorems 2 and 4 (method #9) results in excellent performance. Using this approach, it is very rare that the exact schedulability computation is required for utilisation levels below about 85%. (Recall that Fig. 1 shows that on average, at 97.5% utilisation, only 12% of the tasks require an exact schedulability computation).

Fig. 7 illustrates the frequency distribution of the number of ceiling operations required by the schedulability test for each of the 10,000 tasksets with 95% utilisation. This graph shows that method #9, results in a significantly lower maximum number of ceiling operations, narrower distribution, and smaller average than the default initial value #1. We note that 194 tasksets were schedulable using the sufficient test alone. This accounts for the initial peak in line #9.

Fig. 8 compares the standard and incremental algorithm implementations described in Section 5.2 for the default initial value C_i and for the initial value given by Equation (11). This graph shows that the alternative, incremental implementation converges significantly faster (see lines #10 and #11 on the graph compared with #1 and #5 respectively); however, there is more computation on each inner loop iteration of the incremental implementation. We return to this point in Section 7.

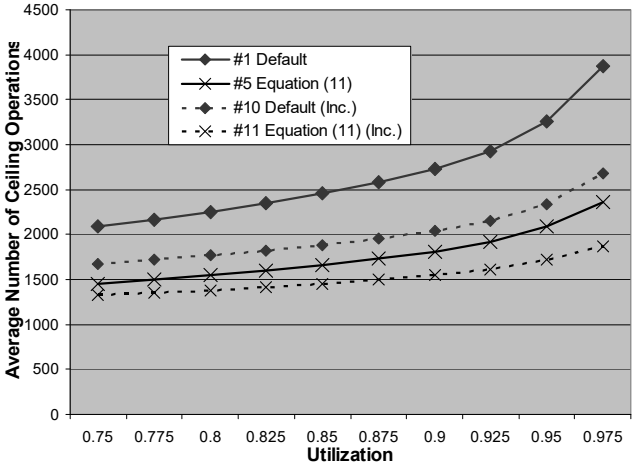

Fig. 8: Comparison between the standard and incremental implementations of the RTA algorithm.

6.2 Experiment 2

This experiment was similar to Experiment 1, only instead of varying the utilisation of the tasksets, overall utilisation was fixed at 95%, and the range of task periods was varied from 2 to 6 orders of magnitude.

Fig. 9 illustrates that for all of the initial values discussed in Section 3 (that can be used to compute exact response times), the average number of ceiling operations required increases approximately linearly with the number of orders of magnitude spanning task periods. We note that the increase is slower for the new initial value #5 given by Equation (11). The extra cost of determining this initial value results in lower performance for smaller ranges of task periods (<4 orders of magnitude), but is justified for larger ranges of task periods (>4 orders of magnitude) where it provides better performance than initial value #4.

Fig. 10 illustrates that for the initial values discussed in Section 4 (that can be used to compute exact schedulability but not exact response times, i.e. #6, #7 and #8); the average number of ceiling operations required remains approximately constant, irrespective of the range of tasks periods. This is an interesting result as it shows that these initial values are particularly useful in reducing the execution time of the schedulability test in just those cases where the task parameters (range of task periods and computation time values) tend to increase the number of iterations required.

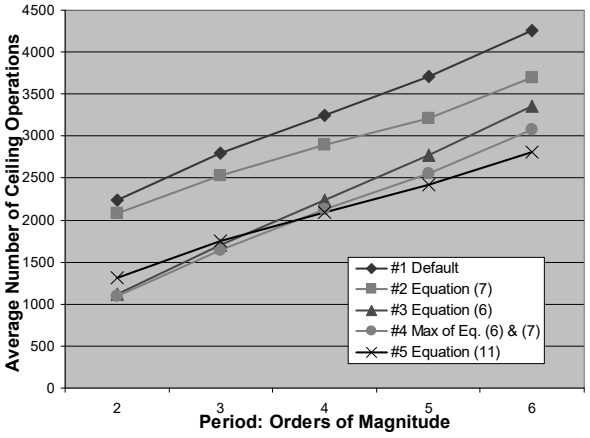

Fig. 9: Average number of "ceiling operations" required by exact response time tests v. the number of orders of magnitude range of task periods, for tasksets with 95% utilisation. Data is for the standard RTA algorithm, and the initial values from Section 3.

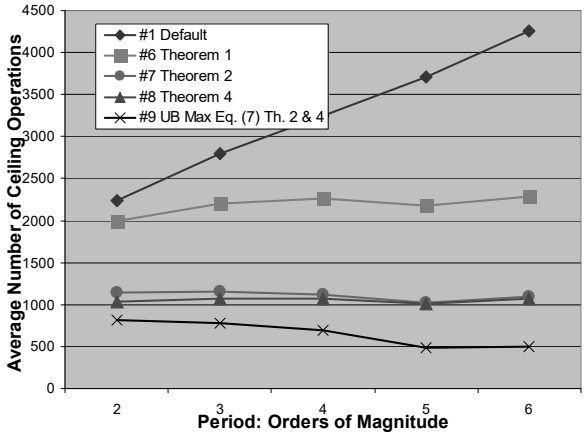

Fig. 10: Average number of "ceiling operations" required by exact Boolean schedulability tests v. the number of orders of magnitude range of task periods, for tasksets with 95% utilisation. Data is for the standard RTA algorithm, and the initial values from Section 4.

Using the response time upper bound combined with the new initial values given by Theorems 2 and 4 (method #9) is highly effective. In particular, the upper bound becomes more accurate as the range of task periods increases, resulting in a decrease in the number of times that an exact schedulability computation is required, and consequently a decrease in the average number of ceiling operations required.

6.3 Experiment 3

In this experiment, we varied the number of tasks from 8 to 256, with the range of task periods fixed at 4 orders of magnitude, and the taskset utilisation fixed at 95%. We found that the average number of ceiling operations increases roughly as the square of the number of tasks.

Fig. 11 shows how the average number of ceiling op-

erations required for each of the initial values given in Section 3 relates to the average number of ceiling operations required when starting with the default initial value. We note that as the number of tasks increases, the utilisation and thus execution time of the individual tasks becomes smaller. This tends to make initial value #3 more accurate and initial value #2 less accurate. Initial value #5 performs progressively better for a larger number of tasks. This is because for large n, typically at least one of the n potential initial values generated by Equation (11) is a close approximation to the exact response time.

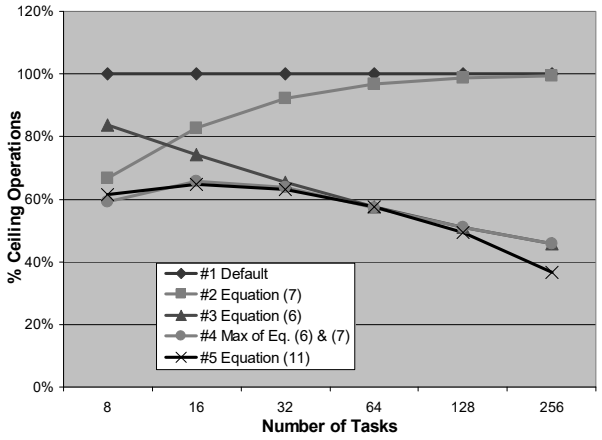

Fig. 11: Performance of exact response time tests relative to the default approach v. taskset cardinality, for tasksets with 95% utilisation. Data is for the standard RTA algorithm and the initial values from Section 3.

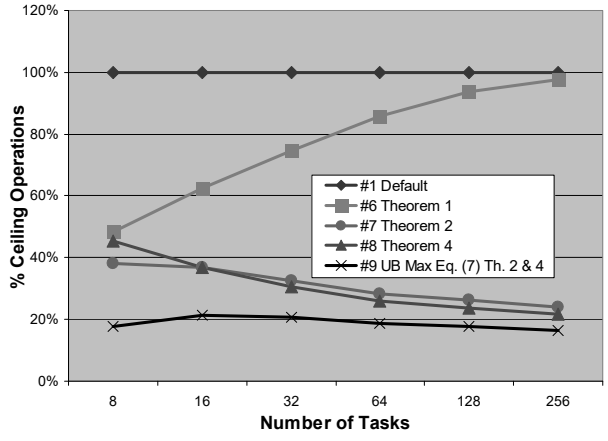

Fig. 12: Performance of exact Boolean schedulability tests relative to the default approach v. taskset cardinality, for tasksets with 95% utilisation. Data is for the standard RTA algorithm and the initial values from Section 4.

Fig. 12 shows how the average number of ceiling operations required for each of the initial values given in Section 4 relates to the average number of ceiling operations required when starting with the default initial value. We note that as the number of tasks increases, the differences between the deadlines of two tasks with adjacent priorities becomes progressively smaller. Thus initial value #6 based on deadline difference performs poorly with an increasing number of tasks. By contrast, the fact that individual task execution times tend to decrease with an increasing number of tasks (for the same overall utilisation and period distribution) means that the other initial values work progressively better for larger numbers of tasks. Again, method #9 is highly effective, reducing the number of ceiling operations required to approx. 20% of those required by the default approach.

6.4 Experiment 4

In this experiment, we examined the hypothesis that checking tasks in reverse priority order helps identify unschedulable tasks early and thus decreases the average number of ceiling operations required by the schedulability test to identify unschedulable tasksets.

This experiment used the same tasksets as Experiment 1. Recall that these tasksets comprised 24 tasks with periods spanning 4 orders of magnitude, for total taskset utilisations varying from 75% to 97.5%.

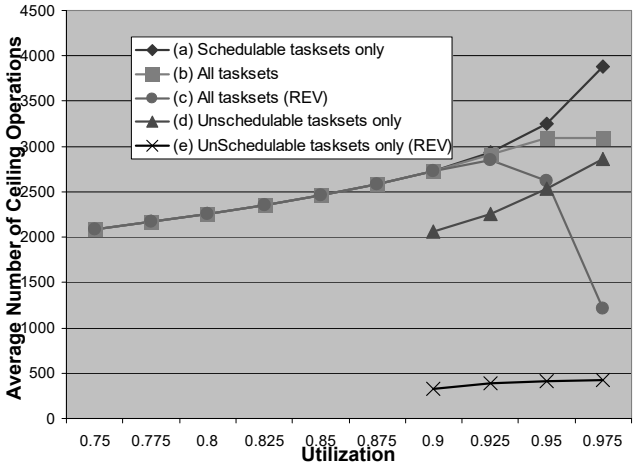

Fig. 13: Average number of "ceiling operations" required by exact response time tests v. taskset utilisation. Data is for the standard RTA algorithm (forward and reverse priority order) using the default initial value.

Fig. 13 shows the average number of ceiling operations required to determine schedulability and exact response times using the default initial value, for (a) schedulable tasksets only, (b) all tasksets, schedulable and unschedulable, (c) all tasksets – checking schedulability in reverse priority order, (d) unschedulable tasksets only, and (e) unschedulable tasksets only – checking schedulability in reverse priority order.

It is clear from the graph that for the tasksets studied; checking schedulability in reverse priority order identified unschedulable tasksets more efficiently, reducing the average number of ceiling operations required.

Out of the 10,000 tasksets at each utilisation level, the percentage that were unschedulable was zero up to 85% utilisation, then 0.01%, 0.2%, 3.3%, 26.5% and 77.4% at utilisation levels of 87.5%, 90%, 92.5%, 95%, and 97.5% respectively. This is why Fig. 13 shows data for unschedulable tasksets only from 90% utilisation upwards. Partly, this is because the tasks all had deadlines equal to their periods ($D = T$). For systems with $D < T$, unschedulability is to be expected at lower utilisation levels.

It is clear from Fig. 13 that at very high utilisation levels (>95%), where there is a high probability that a taskset

will be unschedulable, it is worthwhile checking task schedulability in reverse priority order. This is however only possible for those initial values that are not dependent on knowing the response time of the next highest priority task i.e. initial values based on Equation (7) and Theorems 1 and 4. Checking tasks in reverse priority order enables the schedulability test to terminate early when a low priority task is found to be unschedulable. At lower utilisation levels; however, it is appropriate to use other more effective initial values.

6.5 Experiment 5

Using the default initial value, a pathological set of task parameters (i.e. high priority tasks with very short periods and close to 100% utilisation, combined with low priority tasks with very long periods/deadlines) can lead to the schedulability test requiring a very large number of ceiling operations. The initial value given by Equation (7) is however highly effective in these cases, producing a value close to the exact worst-case response time, hence methods #4, #5 and #9 do not result in a large number of ceiling operations in these cases.

To obtain an indication of the maximum number of ceiling operations that could reasonably be required by schedulability tests applied in a practical application of this work, we performed the following experiment. We generated 1,000,000 tasksets, each comprising 24 tasks with 99% overall utilisation, and task periods spanning 6 orders of magnitude. The maximum number of ceiling operations required by methods #4, #5 and #9 were 11959, 9926, and 7860 respectively. Whilst it is possible to exceed these values with carefully crafted examples, we conclude that it is unlikely that such tasksets will arise frequently in real systems.

7 Execution Time Analysis

In the previous section, we evaluated the efficiency of the exact RTA schedulability test using various initial values, by counting the number of ceiling operations required for convergence. In this section, we look in more detail at the execution time of the algorithms, by running them on an embedded microprocessor.

Our test environment comprised a 32-bit PowerPC (MPC555) development board, clocked at 40 MHz (20 MHz timer/counter clock), with 4 Mbytes of external SDRAM. Execution time information was obtained via the RapiTime worst-case execution time analysis toolset [24]. RapiTime was used to automatically instrument the code, capture timing traces, and produce a report of function execution times. The schedulability test algorithms were coded in C, and compiled using the GNU C compiler, using optimisation level 2 (option –O2).

Using the embedded environment, it was only possible to carry out a limited number of experiments. We therefore confined our investigation to 3 specific tasksets, with utilisations of 75%, 85% and 95% respectively. The tasksets selected comprised 24 tasks, with periods distributed across a range spanning 4 orders of magnitude. These tasksets were worst-case, in the sense that of all 10,000 tasksets generated for each utilisation level, the ones selected required the largest number of ceiling operations to determine schedulability, starting with the default initial value. In this case, the numbers of ceiling operations required were 2956, 3959, and 6324 respectively.

For the taskset with 95% utilisation, Table 4 records: the number of ceiling operations required, the execution time in clock cycles to determine each initial value, the execution time of the schedulability test (not including calculation of the initial value), the overall execution time of the schedulability test, and finally the percentage execution time with respect to the default approach.

Note that in the last row of the table, the data should be interpreted as follows: the response time upper bound (sufficient test) was computed for all 24 tasks, this took 3051 clock cycles in total (including looping over the tasks), as the 22 highest priority tasks were shown to be schedulable by this sufficient test, initial values were only computed for the two lowest priority tasks, this took 615 clock cycles. Finally, exact analysis of these two tasks took 722 ceiling operations, corresponding to 18516 clock cycles. The overall execution time of the schedulability test was therefore 22182 clock cycles, some 13.5% of the time for the default approach.

TABLE 4: EXECUTION TIME MEASUREMENTS

	#	Counter Clock Cycles			
Initial Value	Ceil. Ops.	Initial value	Sched. test	Total	%
#1 Default	6324	1371	163186	164557	100.0
#2 Equation (7)	5724	4390	147751	152141	92.5
#3 Equation (6)	3867	1688	100086	101774	61.9
#4 Max Eqs. (6) & (7)	3611	5179	93541	98720	60.0
#5 Equation (11)	3094	24548	80275	104823	63.7
#6 Theorem 1	4357	1688	112494	114182	69.4
#7 Theorem 2	1609	1688	41962	43650	26.5
#8 Theorem 4	1573	1587	41010	42597	25.9
#9 Max of Eq. (7), Theorems 2 and 4	722	3051 + 615	18516	22182	13.5

The data in Table 4 shows for the initial values discussed in Section 3 (i.e. #1 to #5), the schedulability test itself is faster using initial value #5; however, when the overheads of computing this initial value are included, its performance is slightly worse than that achieved by using initial value #4.

The data in Table 4 also shows that the new deadline dependent initial values #7 and #8 given by Theorems 2 and 4 represent a significant reduction in execution time compared with previous approaches. Combining these in method #9 reduces the overall execution time of the schedulability test by a factor of 7.5 with respect to the default approach.

We also recorded a set of execution time measurements for the incremental schedulability test implementation described in Fig. 3 in Section 5.2. For the 95% utilisation taskset, we found that the number of inner loop iterations was reduced to between 68% and 81% of the number required by the standard implementation, dependent

on the initial value used. Despite this, the overall execution time was between 104% and 121% of the times recorded for the standard implementation. On this particular microprocessor, the extra overheads in the inner loop outweighed the reduction in the number of loop iterations. The execution time for one inner loop iteration was 26 clock cycles for the standard implementation and 39 clock cycles for the incremental approach. We note that this finding is representative only of the specific microprocessor/compiler combination used. We expect that for some microprocessor/compiler combinations the incremental approach could be more efficient.

8 Recommendations

In this section, we provide recommendations to engineers tasked with the problem of implementing exact schedulability tests as the basis of an online acceptance test, as part of an online spare capacity allocation algorithm or, as part of an offline design-time tool.

The most important consideration when choosing how best to implement an exact schedulability test is whether exact response times are required, or if a simple Boolean (schedulable/unschedulable) result will suffice.

Recommendation 1: If exact response times are not required, then we recommend the use of an appropriate sufficient test, such as the response time upper bound given by Equation (14), to determine whether to perform exact schedulability analysis on a task-by-task basis.

Rationale: Our experiments showed that even for tasksets with a very high utilisation, a significant number of individual tasks were schedulable according to the response time upper bound [18], and thus exact schedulability analysis for these tasks was unnecessary. Using this sufficient test to determine when exact analysis was required resulted in a significant improvement in efficiency.

Recommendation 2: If exact response times are not required, then we recommend using an initial value corresponding to the maximum of the values given by Equation (7) and Theorems 2 and 4, as a starting point for the recurrence relation (Equation (2)).

Rationale: Whilst this initial value does not guarantee that the recurrence relation will determine the exact worst-case response time, it does result in an exact schedulability test. Further, the number of operations required for convergence was found to be significantly lower using this initial value than using others that lead to exact response times.

Recommendation 3: If exact response times are required, then we suggest using the initial value given by Equation (11), or alternatively the initial value given by the maximum of Equations (6) and (7)[5].

Rationale: Whilst the overheads of computing the initial value given by Equation (11) mean that it can typically be expected to provide broadly similar overall performance to that achieved starting with the maximum of Equations (6) and (7), in the circumstances where the recurrence relation tends to require a large number of ceiling operations for convergence (.i.e. large numbers of tasks, and/or a wide spread of task periods), then this initial value results in superior performance.

Recommendation 4: If the schedulability test is used as an online admission test, then we recommend checking task schedulability in priority order.

Rationale: When an exact schedulability test is used as an online admission test, then what is important is how long it takes to determine that a schedulable taskset is in fact schedulable. The time taken to determine that an unschedulable taskset is in fact unschedulable is of little consequence, for the reasons discussed in section 5.3. Checking task schedulability in priority order enables the most effective initial values to be used.

Recommendation 5: If the schedulability test is used as part of an online spare capacity allocation algorithm, or as part of a design-time tool, then we recommend considering checking task schedulability in reverse priority order.

Rationale: In these cases, a significant proportion (perhaps 50%) of the task parameter sets considered are likely to represent unschedulable tasksets. Here efficiency can be improved by recognizing unschedulable tasksets as soon as possible. This is best done by starting at the lowest priority. We note; however, that checking task schedulability in reverse priority order effectively precludes the use of certain initial values, in particular, those given by Theorem 2, and Equations (4), (6) and (11).

Recommendation 6: Although the alternative, incremental implementation of the recurrence relation given in Fig. 3 could possibly be more efficient for some compiler/microprocessor combinations, we recommend using the standard implementation given in Fig. 2.

Rationale: The standard implementation is easier to code, takes up less code space, and is likely to be faster on more advanced processors due to the fact that it does not need to write to memory on each iteration of the inner loop.

We note that the analysis presented in this paper requires that task periods and computation times are bounded. If this is not the case, then the methods presented can equally well be used as part of a system sensitivity analysis that determines the extent to which task execution times can increase before the system becomes unschedulable. In cases where other aspects of the system are not well known, for example the amount of interference from interrupts, then the analysis presented in this paper could be used to improve the efficiency of a robust priority assignment method [29].

9 Summary and Conclusions

In this paper, we examined how the efficiency of schedulability tests based on the Response Time Analysis recurrence relation can be improved via:

1. The use of appropriate initial values.
2. Using a sufficient test to determine when exact schedulability calculations are required.
3. Using an incremental algorithm implementation.
4. Examining task schedulability in reverse priority order.

We demonstrated the effectiveness of these approaches

[5] We note that in systems with non-zero blocking and/or jitter, Equations (4) and (5) should be used in preference to Equations (6) and (7).

via empirical investigations on both a PC and on a real-time embedded micro-processor; a PowerPC MPC555. We then used these results to make a series of recommendations to engineers tasked with implementing exact schedulability tests as part of on-line admission tests, on-line spare capacity allocation algorithms, and off-line, design-time optimization tools.

9.1 Contribution

The main contributions of this paper are as follows:

1. Introducing a new family of initial values that can, in some cases, be used to improve schedulability test performance, when it is necessary to calculate the exact response time of each task.
2. Deriving improvements to the "deadline dependent" initial values introduced in [17], that are effective in increasing algorithm performance when an exact schedulability test is required, but upper bounds on response times will suffice (i.e. when exact response times are not required).
3. Extending the initial values introduced in [17] to account for blocking factors and release jitter. Removing these limitations makes it possible, for the first time, to use these initial values in the analysis of real-world systems.
4. Illustrating the efficiency improvements made possible by using the response time upper bound from [18], to determine, on a task-by-task basis, if an exact schedulability calculation is required.

9.2 Conclusion and future work

The research presented in this paper shows that our approach of using the response time upper bound to determine when to compute exact schedulability, and new initial values as an advanced starting point significantly reduces the execution time of exact schedulability tests based on Response Time Analysis. We intend to implement an on-line schedulability test, and spare capacity allocation algorithm based on this research, as part of the Frescor scheduling framework [23].

ACKNOWLEDGEMENTS

This work was funded in part by the EU Frescor project.

APPENDIX: COMPARISON WITH HYPERPLANES EXACT TEST

In this appendix, we make some basic comparisons between the Response Time Analysis (RTA) methods described in this paper and the Hyperplanes Exact Test (HET) described in [19].

Hyperplanes Exact Test

In [6], Lehoczky et al. showed that for tasks that comply with the Liu and Layland [3] system model, exact schedulability of a task τ_i can be determined by inspecting the workload at all points S_i, corresponding to the releases of higher priority tasks between 0 and T_i.

$$S_i = \{kT_j : j = 1..i, k = 1..\left\lfloor \frac{T_i}{T_j} \right\rfloor\} \tag{15}$$

Thus τ_i is schedulable if and only if:

$$L_i = \min_{t \in S_i} \frac{\sum_{j=1..i} \left\lceil \frac{t}{T_j} \right\rceil C_i}{t} \leq 1 \tag{16}$$

In essence, the Hyperplanes Exact Test works by reducing the number of points in S_i that need to be checked. In [19], Bini and Buttazzo showed that the only points that need to be checked, to determine the schedulability of a task τ_i with $D_i \leq T_i$, are those in the set $P_{i-1}(D_i)$, where the set of points $P_j(t)$ is recursively defined as follows: $P_0(t) = \{t\}$ and

$$P_j(t) = P_{j-1}\left(\left\lfloor \frac{t}{T_j} \right\rfloor T_j\right) \cup P_{j-1}(t) \tag{17}$$

The interested reader is referred to [19] for further details of the Hyperplanes Exact Test, including a worked example of its operation.

Performance of RTA and HET schedulability tests

In [19], Bini and Buttazzo provided evidence showing that the Hyperplanes Exact Test outperforms Response Time Analysis based schedulability tests, starting with the default initial value or the initial values given by Sjodin and Hansson [13] – see figures 6 and 7 in [19] for further details. We were therefore initially surprised to observe how poorly the HET algorithm performed on our randomly generated tasksets. For 10,000 tasksets, each comprising 24 tasks with an overall utilisation of 95%, and a range of task periods spanning 4 orders of magnitude, the HET algorithm required on average 23,365 ceiling operations, compared with 3,253 ceiling operations for the RTA algorithm, assuming the default initial value. Further investigation, as to why these results differ so widely from those reported in [19], revealed that the HET algorithm is *extremely* sensitive to the distribution of task periods; something that is not immediately apparent from the results presented in [19].

In the experiments reported in [19], task periods were chosen from the range [1, 1,000,000] according to a uniform distribution, and the results averaged across 10^8 tasksets. This effectively meant that only tasksets with 1 order of magnitude range of task periods were properly represented in the data. This can be seen by noting that the probability of choosing a task period of less than 10,000 from the range [1, 1,000,000] is 1% when a uniform distribution is used, similarly, the probability of choosing a task period of less than 1000 is just 0.1%.

TABLE 5: HET V. RTA ALGORITHMS, NUMBER OF CEILING OPERATIONS

	Orders of magnitude spanning task periods					
Algorithm	**1**	**2**	**3**	**4**	**5**	**6**
RTA	1247	1652	2050	2462	2847	3297
HET	380	1326	5642	23014	81636	197642
HET/RTA	0.3	0.8	2.75	9.35	28.7	60.0

Table 5 shows how the average number of ceiling op-

erations required by both the HET algorithm and the RTA algorithm (starting with the default initial value), varied with the number of orders of magnitude spanning task periods, for 10,000 tasksets, each comprising 24 tasks with an overall utilisation of 85%. Here $24/M$ task periods were chosen according to a uniform distribution from each of the M order of magnitude ranges used (i.e. 1000-10000, 10000-100000, 100000-1000000 etc). Note, we used 85% utilisation tasksets, as at higher utilisations too few tasksets were generated that were schedulable with 1 order of magnitude range of task periods.

For tasksets with a range of task periods amounting to 1 order of magnitude, the HET algorithm performs well, outperforming the default RTA approach, at least in terms of the average number of ceiling operations required. This confirms the results published in [19]. However, the execution requirements of the HET algorithm grow exponentially with an increasing range of task periods, so that for the sample tasksets with periods spanning 6 orders of magnitude, the number of ceiling operations required by the HET algorithm is on average 60 times that of the RTA approach. This behaviour, with respect to the range of task periods, is inherent in the HET algorithm, which in the worst-case can require schedulability to be checked for 2^n points to determine the schedulability of task τ_n.

```
Uint32 WorkLoad(Int32 i, Uint32 b)
{
  Uint32 f,c, branch0, branch1;

  if(i<0) return 0;
  if(b <= gLastPhi[i])  return gLastWL[i];
  f = b / tasks[i].T;
  c = ceiling(b, tasks[i].T);
  branch0 = b-(f*(tasks[i].T-tasks[i].C))
        + WorkLoad(i-1,f*tasks[i].T);
  branch1 = (c * tasks[i].C)+ WorkLoad(i-1,b);
  gLastPhi[i] = b;
  gLastWL[i] = min(branch0,branch1);
  return gLastWL[i];
}
```

Fig. 14: Workload Function of the HET algorithm (implemented using 32-bit unsigned (Uint32) and 32-bit signed (Int32) integer types).

We note that Table 5 actually overstates the performance of the HET algorithm. In the HET algorithm, each "ceiling operation" corresponds to a call[6] of the recursive WorkLoad() function given in Fig. 14 (see [19] for further details of this function, which is the core component of the HET algorithm). The code for the WorkLoad() function is more complex and takes longer to execute than the code associated with the inner loop of the RTA algorithm, line 7 in Fig. 2 (also counted as a "ceiling operation").

To examine the actual execution time of the HET algorithm, we implemented it in C, compiled it using the GNU C compiler (using the –O2 option), and ran it on the MPC555 microprocessor (also used for the execution time measurements reported in Section 7). To obtain the best possible performance from the HET algorithm, we made some simple improvements to the algorithm as presented in [19]; avoiding the use of floating point arithmetic, and instead using 32-bit integers for task parameters and computed values. The basic code for the Workload function is given in Fig. 15. Note that the `ceiling()` and `min()` functions were implemented as Macros, so as to avoid function call overheads. We subsequently also removed the short circuit returns from the original implementation, instead coding the algorithm so that the final 'leaf' calls to the Workload() function were not required. This reduced the overall number of calls to the Workload() function by just over a factor of 2, significantly reducing the overall execution time.

Execution time measurements were taken using the RapiTime worst-case execution time analysis toolset [24].

Applied to the same 95% utilisation taskset referred to in Section 7, the HET algorithm required 19181 ceiling operations to determine that the taskset was schedulable, corresponding to 3,196,748 clock cycles, some 19 times longer than the RTA algorithm using the default initial value, and some 144 times longer than the most efficient approach (#9) reported in Section 7.

The average time for each "ceiling operation" in the HET algorithm, (effectively corresponding to a call to the Workload() function) was 167 clock cycles (down from 228 clock cycles with the short circuit returns present); by comparison, the RTA algorithm required on average just 26 clock cycles for each iteration of its inner loop (i.e. 26 clock cycles per "ceiling operation"). From this data, we infer that the apparent performance advantage of the HET algorithm for small ranges of task periods is illusionary. Correcting the figures from Table 5 to account for the differences in execution times for each iteration of the two algorithms, results in the data in Table 6 and Fig. 15.

TABLE 6: HET V. RTA ALGORITHMS, EXECUTION TIME IN COUNTER CLOCK CYCLES X 1000

	Orders of magnitude spanning task periods					
Algorithm	**1**	**2**	**3**	**4**	**5**	**6**
RTA	32.4	43.0	53.3	64.0	74.0	85.7
HET	63.1	221.5	942	3843	13633	33006
HET/RTA	1.95	5.2	17.7	60	184	385

The final row in Table 6 records the factor by which the execution time of the HET algorithm exceeds that of the RTA algorithm. We infer from this data, that in practice, the RTA algorithm generally outperforms the HET algorithm, and by some significant margin in the case of tasksets with a broad spread of task periods.

[6] Note, we only counted calls that did not exit via the short circuit returns at the start of the Workload() function as "ceiling operations".

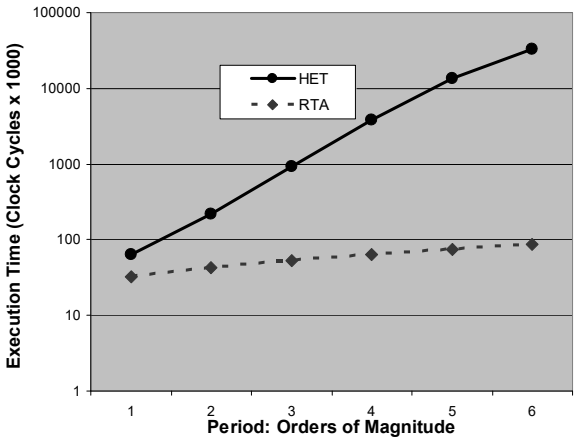

Fig. 15: Average execution time required by the HET and RTA exact schedulability tests v. number of orders of magnitude range of task periods, for tasksets with 85% utilisation. Note both x- and y-axes are logarithmic scales.

It is still possible that the HET algorithm may outperform the RTA algorithm for some tasksets comprising small numbers of tasks (so the number of scheduling points inspected by the HET algorithm is small), and with a small range of task periods. However, we argue that improving upon the RTA algorithm under these conditions is of little practical value as the RTA test has a sufficiently short execution time in this domain (low number of tasks, small spread of task periods) anyway.

REFERENCES

[1] M.S. Fineberg and O. Serlin, "Multiprogramming for Hybrid Computation". In *proceedings AFIPS Fall Joint Computing Conference*, pp. 1-13, 1967.
[2] O. Serlin, "Scheduling of Time Critical Processes". In *proceedings AFIPS Spring Computing Conference*, pp 925-932, 1972.
[3] C.L. Liu and J.W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-time Environment", *Journal of the ACM*, 20(1): 46-61, January 1973.
[4] J.Y.-T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic Real-time Tasks". *Performance Evaluation*, 2(4): 237-250, December 1982.
[5] M. Joseph and P.K. Pandya. "Finding Response Times in a Real-time System". *The Computer Journal*, 29(5):390–395, October 1986.
[6] J. P. Lehoczky, L. Sha, and Y. Ding, "The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior". In *proceedings of the 10th IEEE Real-Time Systems Symposium*, pp. 166–172, 1989.
[7] L. Sha, R. Rajkumar, and J.P. Lehoczky. "Priority Inheritance Protocols: An Approach to Real-time Synchronization". *IEEE Transactions on Computers*, 39(9): 1175-1185, 1990.
[8] J.P. Lehoczky. "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines". In *Proceedings 11th IEEE Real-Time Systems Symposium*, pp. 201–209, IEEE Computer Society Press, December 1990.
[9] T.P. Baker. "Stack-based Scheduling of Real-Time Processes." *Real-Time Systems Journal* (3)1, pp. 67-100, 1991.
[10] K.W. Tindell. "Using Offset Information to Analyse Static Priority Pre-Emptively Scheduled Task Sets". *Technical Report YCS-92-182*. Dept. of Computer Science, University of York, UK, 1992.
[11] N.C. Audsley, A. Burns, M. Richardson, K.W. Tindell, and A.J. Wellings. "Applying New Scheduling Theory to Static Priority Preemptive Scheduling". *Software Engineering Journal*, 8(5):284-292, September 1993.
[12] K.W. Tindell, A. Burns, and A.J. Wellings. "An Extendible Approach for Analyzing Fixed Priority Hard Real-time Tasks". *Real-Time Systems*. Volume 6, Number 2, pp133-151 March 1994.
[13] M. Sjodin and H. Hansson. "Improved Response Time Analysis Calculations". In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 399–408, Madrid, Spain, December 1998.
[14] R.J. Bril, W.F.J. Verhaegh, and E-J.D. Pol. "Initial Values for Online Response Time Calculations". In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 13–22, Porto, Portugal, July 2003.
[15] E. Bini and G.C. Buttazzo. "Measuring the Performance of Schedulability tests". *Real-Time Systems*, 30(1–2):129–154,May 2005.
[16] W-C. Lu, J-W. Hsieh, W-K. Shih, and T-W. Kuo. "A Faster Exact Schedulability Analysis for Fixed-priority Scheduling". *Journal of Systems and Software*, Vol. 79, Issue 12, pp. 1744-1753, December 2006.
[17] W-C. Lu, K-J. Lin, H-W. Wei1, and W-K. Shih. "Period-Dependent Initial Values for Exact Schedulability Test of Rate Monotonic Systems". In *Proceedings Parallel and Distributed Processing Symposium 2007*, IPDPS 2007. Long Beach, USA, pp. 1-8, March 2007.
[18] E. Bini and S.K. Baruah. "Efficient Computation of Response Time Bounds under Fixed-priority Scheduling". In *Proceedings of the 15th conference on Real-Time and Network Systems*, pp. 95–104, Nancy, France, March 2007.
[19] E. Bini and G.C. Buttazzo. "Schedulability Analysis of Periodic Fixed Priority Systems". *IEEE Transactions on Computers*, 53(11):1462–1473, November 2004.
[20] A. Zuhily and A. Burns "Optimality of (D-J)-monotonic Priority Assignment". *Information Processing Letters*. Number 103, pp. 247-250, April 2007.
[21] R.J. Bril, J.J. Lukkien, and W.F.J. Verhaegh. "Worst-Case Response Time Analysis of Real-Time Tasks under Fixed-Priority Scheduling with Deferred Preemption Revisited". In *Proceedings of the 19th Euromicro Conference on Real-Time Systems ECRTS*. pp. 269-279. July 2007.
[22] N.C. Audsley, "Flexible Scheduling of Hard Real-Time Systems". *PhD Thesis*. Dept. of Computer Science, University of York, UK, 1993.
[23] http://www.frescor.org/ 2007.
[24] "RapiTime White Paper: Worst-case Execution Time Analysis" available from www.rapitasystems.com 2007.
[25] R.I. Davis, A. Burns, R.J. Bril, and J.J. Lukkien. "Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised". *Real-Time Systems*, Volume 35, Number 3, pp 239-272. April 2007.
[26] E. Bini, G.C. Buttazzo, and G.M. Buttazzo. "Rate Monotonic Scheduling: The Hyperbolic Bound". *IEEE Transactions on Computers*, 52(7):933–942, July 2003.
[27] S. Lauzac, R. Melhem, and D. Mosse. "An Improved Rate-monotonic Admission Control and its Applications". *IEEE Transactions on Computers*, 52(3):337–350, March 2003.
[28] R.I. Davis, A. Zabos, and A. Burns, "Efficient Exact Schedulability Tests for Fixed Priority Pre-emptive Systems" *Technical Report YCS-418-2007*. Dept. of Computer Science, University of York, UK, 2007.
[29] R.I. Davis and A. Burns. "Robust Priority Assignment for Fixed Priority Real-Time Systems" *In proceedings IEEE Real-Time Systems Symposium* pp. 1-12, 2007.

BIOGRAPHIES

Robert I. Davis received a DPhil in Computer Science from the University of York in 1995. Since then he has founded three start-up companies, all of which have succeeded in transferring real-time systems research into commercial product. Currently, Robert is a Senior Research Fellow in the Real-Time Systems Research Group at the University of York, and a Director of Rapita Systems Ltd. His research interests include scheduling algorithms and schedulability analysis for real-time systems.

Attila Zabos received the BSc degree in computer science from the University of Applied Sciences in Darmstadt, Germany, in 2001. He has been working for more than 5 years in the industry on the development and documentation of embedded real-time and safety-critical systems

before he return to research. Currently he is a PhD student with the Department of Computer Science at the University of York. His main research interests include flexible real-time systems, resource management and mode-changes.

Professor **Alan Burns** co-leads the Real-Time Systems Research Group at the University of York. His research interests cover a number of aspects of real-time systems including the assessment of languages for use in the real-time domain, distributed operating systems, the formal specification of scheduling algorithms and implementation strategies, and the design of dependable user interfaces to real-time applications. He has authored/co-authored over 400 papers and 15 books, with a large proportion of systems and programming languages. Professor Burns has been actively involved in the creation of the Ravenscar Profile, a subset of Ada's tasking model, designed to enable the analysis of high integrity real-time programs and their timing properties.