Improved Feasibility Tests for Asynchronous Real-Time Periodic Task Sets*

Daniel Jelkmann
OFFIS Institute for Information Technology, Oldenburg, Germany
Daniel.Jelkmann@offis.de

Karsten Albers, Frank Slomka
Department of Embedded / Real-Time Systems, University of Ulm, Germany
{Karsten.Albers, Frank.Slomka}@uni-ulm.de

Abstract

The feasibility test for synchronous task systems scheduled by EDF is a well known problem in the literature. However, the analysis of task systems with deadlines shorter than the periods of the tasks has a high runtime complexity. This problem increases if we consider asynchronous task sets. In this paper, the runtime complexity of Pellizzoni's approach to the analysis of asynchronous task sets scheduled by EDF is analyzed. In our experiments, we can show that most of the runtime problem results by mapping the problem to the synchronous case and solving this by using the demand bound feasibility test. To overcome this, we combine Pellizzoni's algorithm with a new approximation approach. Using the superposition algorithm as feasibility test reduces the runtime of the original algorithm given by Pellizzoni in orders of magnitude.

1. Introduction

The correctness of real-time systems depends apart from the functionality on the temporal behavior of the system. Usually, real-time systems are defined by several tasks, which are running on the given system resources. Among other parameters, each task is characterized by a deadline at which the task's execution must be completed. In order to guarantee the temporal correctness of a real-time system, it must be verified that all deadlines are met. An algorithm, which examines exactly this, is called a feasibility test.
A feasibility test, which determines whether all deadlines are met in a general real-time system, has exponential complexity [7]. In special cases the test is of less complexity. For example Baruah et al. presented in [3] their processor demand test. The test is necessary and sufficient for *synchronous* systems and has a pseudo-polynomial complexity. In such synchronous systems, all tasks are released simultaneously and request their first execution at the same time (i.e. $t = 0$). If this constraint does not hold, the system is called *asynchronous*.

In this paper, we present an improved analysis for asynchronous systems. We developed new algorithms by combining the test proposed by R. Pellizzoni [8, 9] and the superposition approach [1, 2].
The task model used in this paper is introduced in section 2. Section 3 gives an introduction to the existing feasibility tests and algorithms further needed in this paper. In section 4, we show the evaluation results of Pellizzoni's algorithm and analyze how much time the different parts of the algorithm will need. We will combine the described tests and present the new algorithms in section 5. In section 6 the new algorithms are evaluated and compared to Pellizzoni's original algorithm, followed by a conclusion in section 7.

The research described has been supported in part by the Deutsche Forschungsgemeinschaft under grants SL 47/1-1 and SL 47/2-1.

2. Task model

We consider asynchronous real-time systems characterized by periodic tasks. A task is defined by a tuple (ϕ_i, C_i, D_i, T_i), where C_i is the worst-case execution time of task τ_i, D_i the relative deadline and T_i the period. ϕ_i is called the offset of the task and indicates when the task gets ready for the first time. The task starts its first execution at time ϕ_i and periodically gets ready all T_i time units. So the task requests its k-th execution at time $t_{i,k} = \phi_i + (k-1)T_i$ with $k \in \{1, 2, 3, \dots\}$. These times are called the release times. Each release time is assigned an absolute deadline, $d_{i,k} = t_{i,k} + D_i$, at which the task's execution must be completed. A system characterized by such tasks is called *asynchronous*, since the tasks have offsets and are not ready simultaneously. In a *synchronous* system, all offsets are zero and all tasks are released at time $t = 0$.

In this work we analyze the feasibility of real-time systems with a single processor using preemptive *Earliest Deadline First (EDF)* scheduling. EDF is optimal for such systems [6], since EDF provides a valid schedule, if a valid schedule exists for the given task set. So a valid schedule can be constructed with the EDF scheduling algorithm, after the feasibility of a task set is proven.

We assume the following definitions: $U = \sum_{i=1}^{N} \frac{C_i}{T_i}$ is the *total utilization* of the task set, $gcd(T_i, T_j)$ is the *greatest common divisor* of T_i and T_j, $lcm(T_i, T_j)$ is the *least common multiple* of T_i and T_j, and $H = lcm(T_1, \dots, T_N)$ is the *hyperperiod* of the task set. We use the short term *ratio* for the ratio between the largest and the smallest period in a task set, and the term *gap* for the difference between deadline and period.

3. Background and related work

In [3] Baruah et al. present a necessary and sufficient feasibility test for synchronous systems with pseudo-polynomial complexity. The test computes the amount of time needed for the successful execution of all tasks and compares it to the available processor time. Since the test calculates the demand requested by the tasks, the test is called processor demand test. The criterion for feasibility is given by $\forall I > 0 : D_b(I) \leq I$, where $D_b(I)$ is the demand requested by all tasks in the interval I. The value of the demand bound function $D_b(I)$ changes at the absolute deadlines of the tasks, therefore exactly these deadlines must be checked during the test. There are known different test borders where the test can be terminated. One possible test border is the calculation of the busy period (see e.g. [10]). Baruah et al. give in their work [3] the formula $I_{max} = \frac{U}{U-1} \cdot \max_{1 \leq i \leq N} (T_i - D_i)$ as a test border, where U is the total utilization of the task set and N is the number of tasks. The runtime of the algorithm depends not only on the utilization but also on the ratio of the different periods and deadlines in the task set. If the task sets contain tasks with small periods and tasks with large periods, the runtime can become quite large [1, 2].

In order to reduce the number of test points and to accelerate the test, different approximation algorithms were developed. One of them is the superposition approach introduced in [1]. The test is based on the processor demand test, but it does not compute the exact demand bound function. Instead, the algorithm analyzes the demand bound function for each task separately. After examining k test points, these functions are approximated by a straight line. k is a parameter of the algorithm and affects the runtime and also the error, which occurs due to the approximation. The total demand bound function is the superposition of the demand bound functions of all tasks (see figure 1). A detailed description of the algorithm can be found in [1]. The Superposition algorithm has a polynomial complexity and is only sufficient due to the approximation error. The error is limited by $\frac{1}{k}$, where k is the number of test points before the

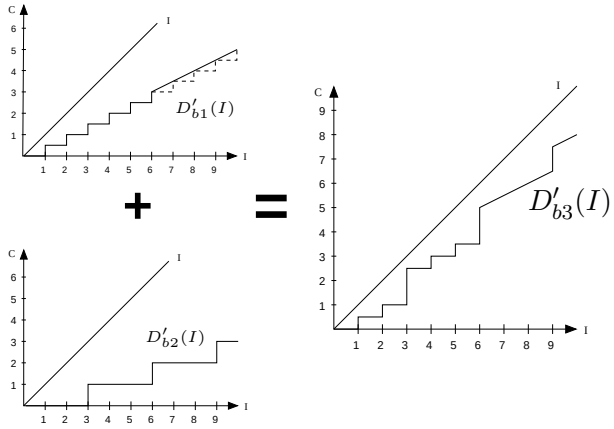

Figure 1. Approximation of the total demand bound function by superposition

approximation begins.

In [2], two improved versions of the Superposition algorithm are presented: the *Superposition Dynamic Error* and *Superposition All Approximated*. Both algorithms use a dynamic error which is adapted during the test if necessary. The Superposition Dynamic Error begins with just a few test points (maybe only one test point) and increases this number, if the approximated demand bound function exceeds the available processor time. In this case, the approximation (and thus the error) is eliminated by increasing the number of test points so that the exact demand bound function is calculated. If the exact function exceeds the available processor time, the test terminates with a negative feasibility result. Otherwise, the analysis is continued with the increased number of test points.

The Superposition All Approximated algorithm reduces the average number of analyzed test points even further by approximating the demand bound function as much as possible. Only if the test fails for an interval, the approximation is cancelled gradually. The tasks, whose approximation was cancelled, are immediately approximated again in the next test interval.

Since both algorithms cancel the approximation completely if necessary and return only a negative result if the exact demand bound function exceeds the available processor time, both tests are necessary and sufficient for synchronous systems (like the processor demand test). The complexity of both algorithms is pseudo-polynomial, since in the worst case the approximations are cancelled and the same number of test points as with the usual processor demand test are analyzed. Pseudocode and an evaluation of the algorithms can be found in [2].

In general, the analysis of asynchronous systems is more complex than the analysis of synchronous systems. As shown in [7, 4], a necessary and sufficient test for asynchronous systems has an exponential complexity, while for synchronous task sets there are known pseudo-polynomial tests. The reason for the higher complexity of the analysis of asynchronous systems is that the time interval with the highest demand is unknown. Since an exponential test is practically not useable for a rising number of tasks, other test algorithms are needed.

Such an algorithm for the analysis of *asynchronous* systems was presented by Rodolfo Pellizzoni [8, 9]. The algorithm is only sufficient and has a pseudo-polynomial complexity. The idea of the algorithm is to analyze the tasks' periods and offsets, in order to determine a set of critical arrival patterns. For each of these patterns, the algorithm generates a new task set. The feasibility of these task sets is analyzed with the processor demand test. These new task sets are equal to the original task set, only the tasks' offsets are changed in such manner that the critical arrival pattern starts at the beginning of the task set. In other words, the critical time interval begins at time $t = 0$. This is the reason why the processor demand can be terminated after the worst case interval has been analyzed. To determine the length of this interval, Pellizzoni's algorithm

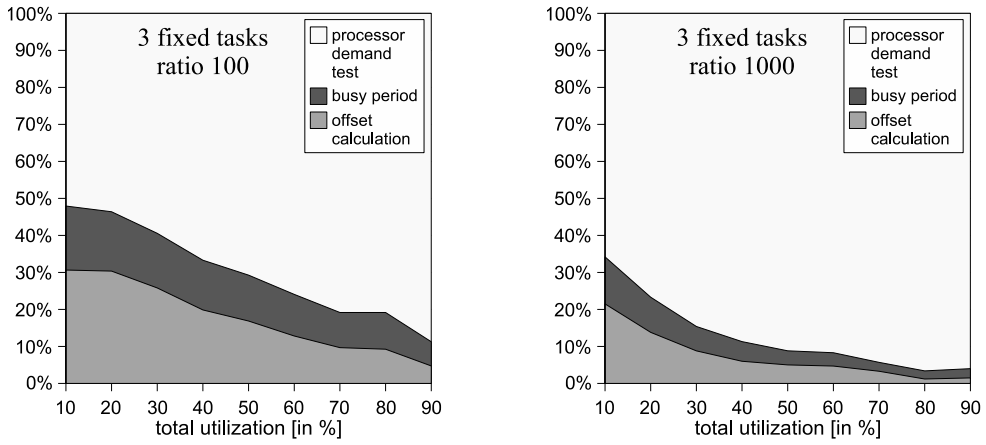

Figure 2. Distribution of the computation time by percentage as a function of the utilization for Pellizzoni's algorithm with three fixed tasks and a ratio of 100 and 1000 (10 tasks per task set, $D \in \{0.3T, 0.8T\}$)

uses the busy period approach [10]. The algorithm does not analyze all critical patterns in detail which can occur in the task set. Such an analysis would result in exponential complexity. Instead the algorithm uses a parameter which specifies how detailed the arrival patterns of the tasks are examined. The parameter indicates the number of fixed tasks. The algorithm analyzes all constellations, in which these n fixed tasks are activated. For the remaining (nonfixed) tasks, the algorithm sets the offsets as the minimum distance to the fixed tasks. Because of this, the task sets generated by the algorithm may be stricter than the original task set. This is the reason why Pellizzoni's test is only sufficient and not necessary. We will refer to these task sets as *strict task sets*.

4. Evaluation of Pellizzoni's algorithm

We generated several test runs with random task sets to analyze the distribution of the computation time of Pellizzoni's algorithm. We tested the task sets for feasibility with Pellizzoni's algorithm and measured the percentage of time needed for performing the processor demand test and for calculating the busy period and the offsets of the strict task sets. As a result it turned out that most of the algorithm's runtime is needed for the processor demand test. Figure 2 shows an example for the distribution of the computation time by percentage as a function of the utilization.

The diagrams show that more than 50% of the computation time is consumed by the processor demand test. For high utilizations this percentage increases to more than 90% of the total computation time. Especially for task sets with a high utilization and high ratios, for which Pellizzoni's feasibility test requires a lot of time, most of the computation time is needed for the processor demand test. The test runs shown in figure 2 were computed using three fixed tasks, but the distribution looks very similar for another number of fixed tasks.

Our approach to improve Pellizzoni's algorithm is to combine it with the Superposition test. Since the Superposition test accelerates the processor demand test, the combination of the algorithms is very promising. The concept of the combination is to replace the processor demand test which is used in Pellizzoni's algorithm by the efficient Superposition test. The Superposition test greatly reduces the number of analyzed test points in comparison to the original processor demand test, so the combination of both algorithms leads to an improved algorithm for the analysis of asynchronous systems.

5. Acceleration of Pellizzoni's algorithm

We combined Pellizzoni's algorithm with the three versions of the Superposition test, which are described in section 3. The computation of the offsets for the tasks as well as the computation of the busy period is taken from Pellizzoni's algorithm without changes. The feasibility of the task sets is analyzed with the Superposition test.

Pellizzoni and the simple Superposition test

The algorithm resulting from combination of Pellizzoni's algorithm with the simple Superposition test has two parameters which both affect the error and the runtime of the algorithm. One parameter is from Pellizzoni's algorithm and determines the number of fixed tasks. The other one is from the Superposition test and specifies the number of test points which are analyzed before the approximation of the demand bound function begins.
Figure 3 shows the pseudocode of the new algorithm. The shown code uses two fixed tasks.

```
1  U = \sum_i C_i/T_i
2  IF U > 1 => return not feasible;
3  for each i_1 = 1 ... N
4   \phi'_{i_1} = 0
5   for each i_2 = 1 ... N, i_2 \neq i_1
6    for each k_1 = 0 ... T_{i_2}/gcd(T_{i_1},T_{i_2}) - 1
7     \phi'_{i_2} = \lceil (\phi_{i_1}+k_1 T_{i_1}-\phi_{i_2})/T_{i_2} \rceil T_{i_2} - (\phi_{i_1}+k_1 T_{i_1}-\phi_{i_2})
8     for each j = 1 ... N, j \neq i_3, i_2, i_1
9      \phi'_{i_3} = \lceil (\phi_{i_1}+k_1 T_{i_1}-\phi_j)/(gcd(T_j, lcm(T_{i_1},T_{i_2}))) \rceil · gcd(T_j, lcm(T_{i_1},T_{i_2})) - (\phi_{i_1}+k_1 T_{i_1}-\phi_j)
10    next j
11    // calculate max testinterval
12    IF U = 1 THEN
13     BP = hyperperiod;
14    ELSE
15     BP = C_{i_1};
16     while BP changes
17      BP = \sum_{p=1}^{N} \lceil (BP-\phi'_{i_p})/T_p \rceil · C_p
18     repeat
19    // superposition test
20    testlist = { }; D'_b = 0; U_ready = 0; I_old = 0;
21    \forall j \in S : testlist.add( \phi'_{i_j} + D_j , j );
22    WHILE (testlist \neq { })
23     j = testlist.getNextDemand();
24     I_act = testlist.intervalForDemand( j );
25     IF (I_act > BP) break;
26     D'_b = D'_b + C_j + (I_act - I_old) · U_ready
27     IF (D'_b > I_act) return not feasible;
28     IF (I_act < (M-1) · T_j + D_j + \phi'_{i_j})
29      testlist.add( I_act + T_j , j );
30     ELSE
31      U_ready = U_ready + C_j/T_j;
32     I_old = I_act;
33    END WHILE
34   next k_1
35  next i_2
36 next i_1
37 return feasible
```

Figure 3. Pseudocode of the Pellizzoni+Superposition test with 2 fixed tasks

According to Pellizzoni's algorithm, the computation of the offsets must be adapted for another number of fixed tasks. N is the number of tasks in the task set and M (line 28) is the number of test points for the Superposition test. The offset of the task τ_j in the original task set is ϕ_{i_j} and the computed offset of the strict task set is ϕ'_{i_j}.
First, the total utilization of the task set is computed in line 1. If it is larger than 1, the task set is not feasible and the test ends. The remaining pseudocode essentially consists of two parts. Lines 3 to 18 and lines 34 to 37 are from Pellizzoni's algorithm and implement the computation of the offsets and the busy period (compare to [8]). The remaining lines 19 to 33 implement the feasibility test by the simple Superposition algorithm. Since the test is used to analyze asynchronous task sets, small changes in comparison to the original Superposition test for synchronous tasks are necessary. In lines 21 and 28, we added the task's offset (compare to [1]). The test is terminated when the test interval gets larger than the busy period (line 25).
The Superposition algorithm analyzes M test points for each task. Since the remaining test points are approximated and thus an error occurs, the test does not recognize all feasible task sets as feasible. Because of this the simple Superposition test is only sufficient in the synchronous case. This means for our new test that the feasibility rate (task sets recognized as feasible compared to the feasible task sets) is smaller than with Pellizzoni's original algorithm. The error $\varepsilon = \frac{1}{M}$ (and consequently the feasibility rate) depends on the number of test points used. For an infinite number of test points ($M \to \infty$) is the new algorithm identical to Pellizzoni's test, since the tasks are never approximated and thus no error can occur.

```
1  // superposition test
2  testlist = { }; ApproxList = { };
3  D'_b = 0; U_ready = 0; I_old = 0;
4  ∀j ∈ S : testlist.add( φ'_ij + D_j , j );
5  WHILE (testlist ≠ { })
6     j = testlist.getNextDemand();
7     I_act = testlist.intervalForDemand( j );
8     IF (I_act > BP) break;
9     D'_b = D'_b + C_j + (I_act − I_old) · U_ready
10    WHILE (D'_b > I_act)
11       IF (ApproxList = { }) return not
      feasible;
12       increase level;
13       ∀r ∈ S_rev :
14          ApproxList.remove( r );
15          U_ready = U_ready − C_r/T_r;
16          D'_b = D'_b − app(I_act, r);
17          testlist.add( NextInt( I_act, r ) , r );
18    END WHILE
19    IF (I_act < Testborder( j ))
20       testlist.add( I_act + T_j , j );
21    ELSE
22       U_ready = U_ready + C_j/T_j;
23       ApproxList.add( j );
24    I_old = I_act;
25 END WHILE
```

Figure 4. Pseudocode of the feasibility test of the Pellizzoni+Superposition Dynamic Error algorithm

```
1  // superposition test
2  testlist = { }; ApproxList = { };
3  D'_b = 0; U_ready = 0; I_old = 0;
4  ∀j ∈ S : testlist.add( φ'_ij + D_j , j );
5  WHILE (testlist ≠ { })
6     j = testlist.getNextDemand();
7     I_act = testlist.intervalForDemand( j );
8     IF (I_act > BP) break;
9     D'_b = D'_b + C_j + (I_act − I_old) · U_ready
10    WHILE (D'_b > I_act)
11       IF (ApproxList = { }) return not
      feasible;
12       r = ApproxList.
      getAndRemoveFirstTask();
13       U_ready = U_ready − C_r/T_r;
14       D'_b = D'_b − app(I_act, r);
15       testlist.add( NextInt( I_act, r ) , r );
16    END WHILE
17    U_ready = U_ready + C_j/T_j;
18    ApproxList.add( j );
19    I_old = I_act;
20 END WHILE
```

Figure 5. Pseudocode of the feasibility test of the Pellizzoni+Superposition All Approximated algorithm

Pellizzoni and Superposition Dynamic Error

The combination of Pellizzoni's algorithm with the Superposition Dynamic Error leads to a new algorithm, which can be directly compared to Pellizzoni's original test. The algorithm has one parameter which specifies the number of fixed tasks. The Superposition Dynamic Error does not use a fixed number of test points. Instead the algorithm adjusts the number of test points (and also the error) at runtime. If necessary, the approximations are completely canceled. Because of this, the new algorithm recognizes exactly the same task sets as feasible as Pellizzoni's original algorithm.

The pseudocode for the Pellizzoni+Superposition Dynamic Error is similar to the listing in figure 3. The code for the computation of the busy period and the offsets is identical (like in Pellizzoni's algorithm), only the simple Superposition test in lines 19 to 33 changes. In figure 4 the feasibility test for the new Pellizzoni+Superposition Dynamic Error is given, which replaces lines 19 to 33 of figure 3.

Testborder(j) in line 37 returns the size of the interval at which the approximation of the task τ_j begins. In comparison to the synchronous case, the function must be extended by the offset, so that the function returns $(k-1) \cdot T_j + D_j + \phi'_{i_j}$, where k is the number of test points (which may be increased in line 30). To cancel the approximation of a task, the algorithm uses the functions *app(I, r)* and *NextInt(I, r)* (lines 34 and 35). *app(I, r)* returns the approximation error for task τ_r in the interval I and *NextInt(I, r)* supplies the next test interval greater than I for the task. The functions result in:

$$app(I,r) = \left(\frac{I - D_r - \phi'_{i_r}}{T_r} - \left\lfloor \frac{I - D_r - \phi'_{i_r}}{T_r} \right\rfloor\right) C_r \tag{1}$$

$$NextInt(I,r) = \left(\left\lfloor \frac{I - D_r - \phi'_{i_r}}{T_r} \right\rfloor\right) T_r + D_r + \phi'_{i_r} \tag{2}$$

Pellizzoni and Superposition All Approximated

The Superposition All Approximated reduces the number of examined test points compared to Superposition Dynamic Error even further, so that the combination with Pellizzoni's algorithm seems to be promising. The resulting algorithm recognizes the same task sets as feasible as Pellizzoni's original algorithm, since the Superposition All Approximated is exact and has no error.

In figure 5, the pseudocode for the feasibility test of the Pellizzoni+Superposition All Approximated algorithm is shown. The computation of the busy period and the offsets is identical to the code given in figure 3. In order to obtain the whole algorithm, lines 19 to 33 in figure 3 must be replaced by the code from figure 5. For the functions *app(I, r)* and *NextInt(I, r)*, the formulas 1 and 2 for the asynchronous case are used as given above.

5.1. Complexity

The complexity of the simple Superposition algorithm is polynomial [1]. This does not apply to the Superposition Dynamic Error and the Superposition All Approximated, since in the worst case both algorithms cancel the approximations. Thus, they have the same pseudo-polynomial complexity as the processor demand test.

All three new algorithms resulted from the combination of the superposition approach with Pellizzoni's algorithm and use Pellizzoni's way to compute the offsets for the strict task sets. The offset computation has already a pseudo-polynomial complexity, since the complexity of this part depends both on the number of tasks and on the task parameters. The busy period, which is computed for every analyzed task set, has also a pseudo-polynomial complexity for $U < 1$ [10]. Thus it follows that all of the three new algorithms have a pseudo-polynomial complexity (as Pellizzoni's original algorithm).

6. Evaluation of the new algorithms

In this section we evaluate the new algorithms and compare them to Pellizzoni's original test. The random generation follows the uniform distribution proposed by Bini [5]. All algorithms test the feasibility of the same task sets, so that we can compare the results directly in order to determine the advantages of the new algorithms. We carried out several test runs and examined them under different aspects. The most important results are presented below. The following tests were performed on a AMD Athlon PC (1.9 GHz, 512MB RAM) on SuSE Linux 9.2.

Comparison of the algorithms

To compare the performance of the algorithms, we generated task sets with 8 tasks and a ratio of 50. The gap of the tasks amounted to 20% and 70% ($D \in \{0.3T, 0.8T\}$) and the total utilization ranged from 1% to 99% (in steps of two percent). We have chosen these values to cover the whole test range extensively. Pellizzoni also used gaps between 20% and 70% in his work ([8], section 3.5). We compare Pellizzoni's original test and the three new algorithms, each with three fixed tasks, using different approximation levels and 20,000 task sets totally. The analysis with the Pellizzoni+Superposition algorithm was done for 2, 6, 10 and 19 test points for each task, which corresponds to an exactness ($= 100\% -$ error) of 50%, 85%, 90% and 95%.
The diagram in figure 6 shows the average number of test points as a function of the utilization. The number of test points is well below the one's of Pellizzoni's algorithm. It is remarkable that for Pellizzoni's algorithm and the Pellizzoni+Superposition variants, the number of test points clearly increases with rising utilization. In contrast, the effort for the advanced algorithms seem to be independent from the utilization. The values of all algorithms in figure 6 drop for high utilizations close to 100% as many infeasible task sets are generated then.
For the same test run, the average number of analyzed test points, the runtime and the feasibility rate of the different algorithms are shown in table 1. Both new exact algorithms (Dynamic Error and All Approximated) were capable to do the feasibility test by analyzing only 20% of the test points Pellizzoni's algorithm needed (in average), while recognizing the same task sets as feasible. It is noticeable that the average runtime of the algorithms does not differ so much as one would expect due to the differences between the number of analyzed test points (i.e 20%

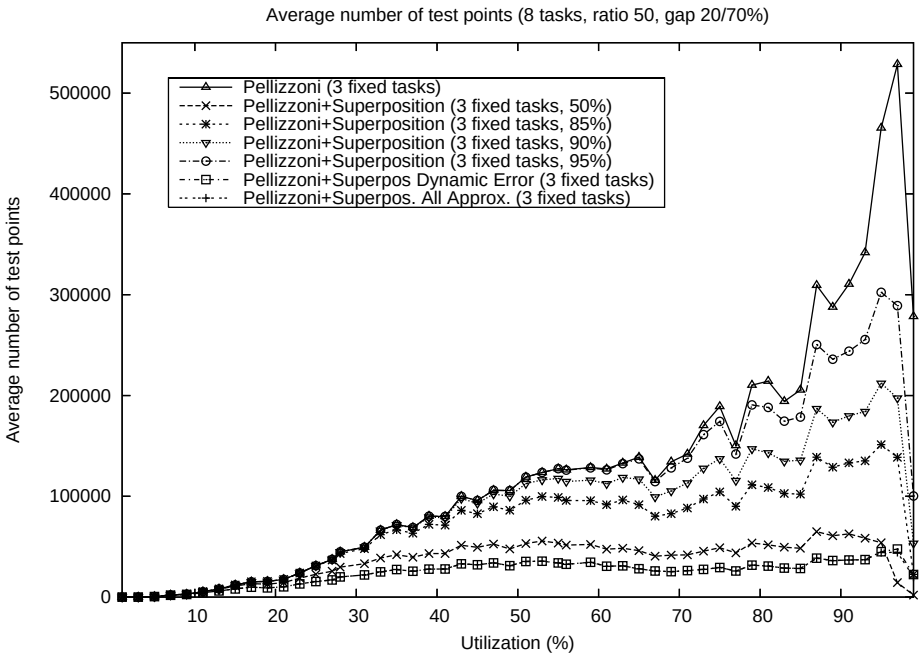

Figure 6. Average number of test points for Pellizzoni's original test and the new algorithms

Algorithm	% of feasible task sets	Average run-time (in ms)	Average number of test points
Pellizzoni	78.52	363.2	126,255.8
Pellizzoni+Superpos. (50%)	76.89	258.6	35,990.7
Pellizzoni+Superpos. (85%)	77.99	314.9	70,494.0
Pellizzoni+Superpos. (90%)	78.21	341.5	87,930.8
Pellizzoni+Superpos. (95%)	78.37	369.3	106,576.6
Pell.+Superpos. Dynamic Error	78.52	245.0	24,205.0
Pell.+Superpos. All Approx.	78.52	242.7	24,056.6

Table 1. Percentage of feasible task sets, runtime and number of analyzed test points for 3 fixed tasks (8 tasks, ratio 50, gap 20%/70%)

of Pellizzoni's runtime for the two new exact algorithms). The explanation is that in the new algorithms, only the feasibility test is accelerated, but the computation of the offsets and the busy period needs the same time as in Pellizzoni's original test. The distribution of the runtime is shown in figure 7 for the All Approximated algorithm. In compare to figure 2, we notice that the distribution is more or less independent of the utilization.

Performance for higher ratios

Table 2 shows the average runtimes and the number of analyzed test points for different ratios. The task sets consisted of five tasks and the algorithms used one fixed task. The utilization ranged from 1% to 99% and the gap was 20% and 70% of the task's period. As ratios we have chosen the powers of ten between $1{,}000$ and $10{,}000{,}000$, in order to measure the development of the runtime for strongly increasing ratios. While for the new algorithms the number of test points increases from ratio $1{,}000$ to a ratio of ten million approximately by factor 1.27, for Pellizzoni's algorithm this factor is greater than $8{,}000$. We suggest to use the values of the runtime of the new algorithms in table 2 with some caution. These values are small and therefore influence by other processes on the system cannot be excluded. The increase of the average number of test points for rising ratios results from the fact that Pellizzoni's algorithm analyzes the different arrival patterns of the tasks. For a rising ratio between the largest and the smallest period in the task set, also the number of critical arrival patterns increases. Since the algorithm generates for every critical pattern a new task set which will be analyzed, also the number of examined task sets grows with rising ratio. It is not possible to predict how strong the rise of the number of test points and also the runtime will be generally, since it depends among other aspects on the number of tasks in the task set and the number of fixed tasks.

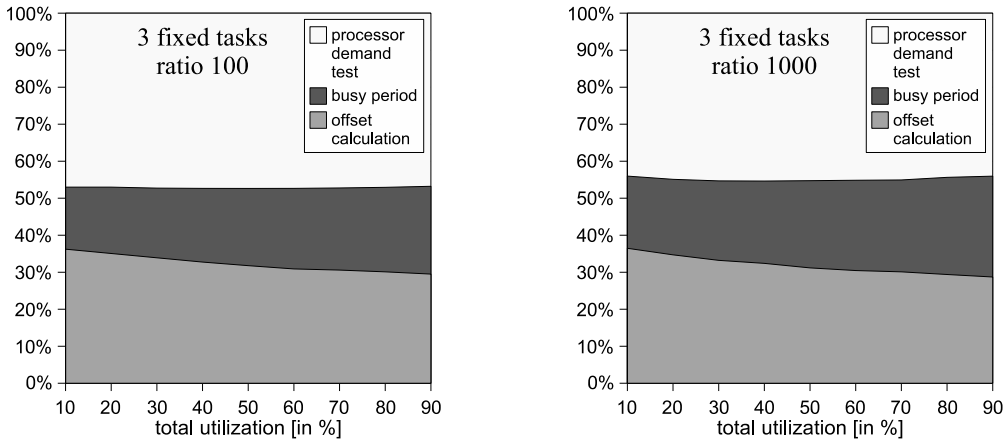

Figure 7. Distribution by percentage of the runtime for the Pellizzoni+Superposition All Approximated algorithm with three fixed tasks (10 tasks, $D \in \{0.3T, 0.8T\}$)

Ratio	Pellizzoni		Pellizzoni+Superpos. Dynamic Error		Pellizzoni+Superpos. All Approximated	
	runtime	test points	runtime	test points	runtime	test points
1.000	1.13 ms	764	0.21 ms	9.29	0.24 ms	9.15
10.000	8.66 ms	6,859	0.22 ms	10.41	0.20 ms	10.24
100.000	78.90 ms	64,733	0.25 ms	11.02	0.24 ms	10.92
1.000.000	743.62 ms	619,681	0.25 ms	11.36	0.21 ms	11.28
10.000.000	7,100.78 ms	6,247,543	0.27 ms	11.79	0.25 ms	11.66

Table 2. Average runtime and number of analyzed test points depending on the ratio for one fixed task (5 tasks, gap 20%/70%, utilization 1-99%)

7. Conclusion

In this paper, we evaluated Pellizzoni's algorithms to find out which part consumes most of the algorithm's runtime. The results in section 4 have shown that between 50% and 90% of the total computation time (depending on the task set) is needed for performing the processor demand test.

We accelerated Pellizzoni's algorithm by combining the superposition test and Pellizzoni's approach. The result of this combination are three new algorithms for the analysis of asynchronous systems. We described the new algorithms in detail in section 5 and also provided pseudo-codes. By evaluating the new algorithms in section 6, we found that they allow enormous speed advantages in comparison to Pellizzoni's original algorithm. The new algorithms especially accelerate the analysis of task sets with a high utilization and a high ratio, whose feasibility test needs a relative long time using Pellizzoni's original algorithm. The amount of the speed advantage of these algorithms depend on the task sets which are examined and cannot be given in general.

Two of the new algorithms, the Pellizzoni+Superposition Dynamic Error and the Pellizzoni+ Superposition All Approximated, represent a clear improvement of Pellizzoni's original test. Both algorithms recognize the same task sets as feasible as Pellizzoni's algorithm and permit due to the approximations a clear reduction of the runtime.

References

[1] K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. Euromicro Conference on Real-Time Systems (ECRTS 04), pages 187-195, June 2004.

[2] K. Albers and F. Slomka. Effi cient feasibility analysis for real-time systems with EDF scheduling. Design, Automation and Test in Europe (DATE 05), pages 492-497, March 2005.

[3] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *IEEE Real-Time Systems Symposium*, pages 182–190, 1990.

[4] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, 1990.

[5] E. Bini and G. C. Buttazzo. Biasing effects in schedulability measures. In *IEEE Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 196–203, June 2004.

[6] M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974.

[7] J. Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980.

[8] R. Pellizzoni. Effi cient feasibility analysis of real-time asynchronous task sets. Master's thesis, Università di Pisa and Scuola Superiore S. Anna, Pisa, Italy, 2004.

[9] R. Pellizzoni and G. Lipari. A new suffi cient feasibility test for asynchronous real-time periodic task sets. Euromicro Conference on Real-Time Systems (ECRTS 04), Catania (Italy), June 2004.

[10] M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report RR-2772, 1996.