Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks

Theodore P. Baker*
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4530
e-mail: baker@cs.fsu.edu

Michele Cirinei
Scuola Superiore Sant'Anna
Pisa, Italy
e-mail: cirinei@gandalf.sssup.it

Abstract

This report describes a necessary and sufficient test for the schedulability of a set of sporadic hard-deadline tasks on a multiprocessor platform, using any of a variety of scheduling policies including global fixed task-priority and earliest-deadline-first (EDF). The contribution is to establish an upper bound on the computational complexity of this problem, for which no algorithm has yet been described. The compute time and storage complexity of the algorithm, which performs an exhaustive search of a very large state space, make it practical only for tasks sets with very small integer periods. However, as a research tool, it can provide a clearer picture than has been previously available of the real success rates of global preemptive priority scheduling policies and low-complexity sufficient tests of schedulability.

1 Introduction

This report describes a "brute force" algorithm for determining whether a hard-deadline sporadic task system will always be scheduled so as to meet all deadlines, for global preemptive priority scheduling policies on multiprocessor platforms. The algorithm is presented in a generic form, that can easily be applied to earliest-deadline-first, fixed-priority, least-laxity-first, or just about any other priority-based scheduling policy.

Symmetric multiprocessor platforms have long been used for high performance real-time systems. Recently, with the introduction of low-cost multi-core microprocessor chips, the range of potential embedded applications of this kind of architecture as expanded rapidly.

The historically dominant approach to scheduling real-time applications on a multiprocessor has been *partitioned*; that is, to assign each task (statically) to a processor, and then apply a single-processor scheduling technique on each processor. The alternative is *global* scheduling; that is, to maintain a single queue of ready jobs and assign jobs from that queue dynamically to processors. Despite greater implementation overhead, the global approach is conceptually appealing in several respects.

Several sufficient tests have been derived for the schedulability of a sporadic task set on a multiprocessor using a given scheduling policy, such as global preemptive scheduling based on fixed task priorities (FTP) or deadlines (EDF) [1, 2, 3, 5, 6, 7, 10, 13]. For example, it can be shown that a set of independent

*This material is based upon work supported in part by the National Science Foundation under Grant No. 0509131, and a DURIP grant from the Army Research Office.

periodic tasks with deadline equal to period will not miss any deadlines if it is scheduled by a global EDF policy on m processors, provided the sum of the processor utilizations does not exceed $(m-1)u_{\max}+u_{\max}$, where $u_{\max}$ is the maximum single-task processor utilization [13, 10].

One difficulty in evaluating and comparing the efficacy of such schedulability tests has been distinguishing the causes of failure. That is, when one of these schedulability tests is unable to verify that a particular task set is schedulable there are three possible explanations:

1. The problem is with the task set, which is *not feasible*, *i.e.*, not able to be scheduled by any policy.
2. The problem is with the scheduling policy. The task set is *not schedulable* by the given policy, even though the task set is feasible.
3. The problem is with the test, which is *not able to verify* the fact that the task set is schedulable by the given policy.

To the best of our knowledge there are no previously published accounts of algorithms that can distinguish the above three cases. The intent of this paper is to take one step toward closing this gap, by providing an algorithm that can distinguish case 2 from case 3.

The algorithm presented here is simple and obvious, based on modeling the arrival and scheduling processes of a sporadic task set as a finite-state system, and enumerating the reachable states. A task set is then schedulable if and only if no missed-deadline state is enumerated. Although the computational complexity of this state enumeration process is too high to be practical for most real task systems, it still interesting, for the following reasons:

1. Several publications have incorrectly asserted that this problem can be solved by a simpler algorithm, based on the presumption that the worst-case scenario occurs when all tasks have jobs that arrive periodically, starting at time zero.
2. To the best of our knowledge, no other correct algorithm for this problem has yet been described.
3. This algorithm has proven to be useful as a research tool, as a baseline for evaluating the degree to which faster, but only sufficient, tests of schedulability fail to identify schedulable task systems, and for discovering interesting small examples of schedulable and unschedulable task sets.
4. Exposure of this algorithm as the most efficient one known for the problem may stimulate research into improved algorithms.

Section 2 reviews the formal model of sporadic task systems, and what it means for a task system to be schedulable. Section 3 describes a general abstract model of system states, and Sections 4 and 5 explain how to compute the state transitions in the model. Section 6 shows how this model fits several well known global multiprocessor scheduling policies. Section 7 describes a generic brute-force schedulability testing algorithm, based on a combination of depth-first and breadth-first search of the abstract system state graph. Section 8 provides a coarse estimate of the worst-case time and storage requirements of the brute-force algorithm. Section 10 summarizes, and indicates the direction further research on this algorithm is headed.

2 Sporadic Task Scheduling

A *sporadic task* $\tau_i = (e_i, d_i, p_i)$ generates a potentially infinite sequence of *jobs*, characterized by a *maximum (worst case) compute time requirement* e_i, a maximum response time (relative deadline) d_i , and a *minimum inter-arrival time* (period) p_i. It is assumed that $e_i \leq \min(d_i, p_i)$, since otherwise a task would be trivially infeasible.

A sporadic task system τ is a set of sporadic tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$.

An *arrival time sequence* a_i for a sporadic task τ_i is a finite or infinite sequence of times $a_{i,1} < a_{i,2} < \cdots$ such that $a_{i,j+1} - a_{i,j} \geq p_i$, for $j = 1, 2, \ldots$. A *arrival time assignment* r for a task set is a mapping of arrival time sequences a_i to tasks τ_i, one for each of the tasks in τ. An arrival time assignment and a task set define a set of jobs.

An m-processor *schedule* for a set of jobs is a partial mapping of time instants and processors to jobs. It specifies the job, if any, that is scheduled on each processor at each time instant. For consistency, a schedule is required not to assign more than one processor to a job, and not to assign a processor to a job before the job's arrival time or after the job completes. For a job arriving at time a, the *accumulated compute time* at time b is the number of time units in the interval $[a, b)$ for which the job is assigned to a processor, and the *remaining compute time* is the difference between the total compute time and the accumulated compute time. A job is *backlogged* if it has nonzero remaining compute time. The *completion time* is the first instant at which the remaining compute time reaches zero. The *response time* of a job is the elapsed time between the job's arrival time and its completion time. A job misses its absolute deadline if the response time exceeds its relative deadline.

The *laxity* (sometimes also known as slack time) of a job at any instant in time prior to its absolute deadline is the amount of time that the job can wait, not executing, and still be able to complete by its deadline. At any time t, if job J has remaining compute time e and absolute deadline d, its laxity is $\ell^J(t) \stackrel{\text{def}}{=} d - e$.

The jobs of each task are required to be executed sequentially. That is the earliest start time of a job is the maximum of: its arrival time; the completion time of the preceding job of the same task. This earliest start time is also called the *ready time* of the job.

The decision algorithm described here is restricted to integer values for task periods, compute times, and deadlines. This is not a serious conceptual restriction, since in any actual system time is not infinitely divisible; the times of event occurrences and durations between them cannot be determined more precisely than one tick of the systems most precise clock. However, it is a practical restriction, since the complexity of the algorithm grows exponentially with the number of clock ticks in the task periods, compute times, and deadlines, as will be explained later.

The notation $[a, b)$ is used for time intervals, as a reminder that the interval includes all of the time unit starting at a but does not include the time unit starting at b.

These conventions allow avoid potential confusion around end-points and prevent impractical schedulability results that rely on being able to slice time at arbitrary points. They also permit exhaustive testing of schedulability, by considering all time values in a given range.

3 The Abstract State Model

Determining whether a sporadic task system τ can miss any deadlines if scheduled according to a given algorithm can be viewed as a reachability problem in a finite non-deterministic state transition graph. Given a start state in which no jobs have yet arrived, the problem is to determine whether the system can reach a state that represents a scheduling failure. A task set is schedulable if-and-only-if there is no sequence of valid transitions from the system start state to a failure state.

Ha and Liu [11] defined the concept of predictable scheduling policy, and showed that for for all preemptive global fixed-task-priority and fixed-job-priority scheduling policies are predictable. As a consequence, in considering such scheduling algorithms we need only consider the case where each job's compute time requirement is equal to the worst-case compute time requirement of its generating task.

In order to reduce the time and storage complexity of determining schedulability, it is desirable to express the system state as simply as possible, and to eliminate from consideration as many states as

possible.

Given a task system τ, an abstract system state S is an n-tuple of the form

$$((nat(S_1), rct(S_1)), \ldots, (nat(S_n), rct(S_n)))$$

The value $nat(S_i)$ denotes the earliest *next arrival time* for task τ_i, relative to the current instant, and the value $rct(S_i)$ denotes the *remaining compute time* of the job of τ_i that is currently contending for processor time.

If $rct(S_i)$ is zero there is no job of τ_i contending for processor time. That is, all the jobs of τ_i that have arrived so far have been completed. In this case the earliest time the next job of τ_i can arrive is $nat(S_i)$ time units from the present instant. That value cannot be negative, and it is zero only if a job of τ_i can arrive immediately.

If $rct(S_i)$ is positive, there is a job J of τ_i contending for processor time and J needs $rct(S_i)$ units of processor time to complete. In this case, $nat(S_i)$ is the offset in time, relative to the current instant, of earliest time that the next job of τ_i after J can arrive. If the value is negative, the earliest possible arrival time of the next job of τ_i after J is $nat(S_i)$ time units in the past.

It follows that the task abstract state determines the following information for each task in a system:

- Task τ_i has a ready job if-and-only-if $rct(S_i) > 0$.
- The remaining compute time of the current ready job of τ_i is $rct(S_i)$.
- The time to the arrival of the next job after the current ready job of τ_i is $nat(S_i)$.
- The *time to the next deadline* of the current ready job of task τ_i is

$$ttd(S_i) = nat(S_i) - (p_i - d_i) \tag{1}$$

- The *laxity* of the current ready job of τ_i is

$$laxity(S_i) = ttd(S_i) - rct(S_i) \tag{2}$$

There are two kinds of state transitions. The passage of one instant of time is modeled by a *clock-tick* state transition, which is deterministic. A job becoming ready is modeled by a *ready* state transition. Ready transitions are non-deterministic, because the sporadic task model allows a job to arrive any time $\geq$ one period after the arrival time of the preceding job in the same task.

An abstract system state is *reachable* if it is reachable from the start state via a finite sequence of clock tick and ready transitions.

The system *start state* is defined to be $((0,0), \ldots, (0,0))$. That is, the state in which there are no tasks contenting for processor time, and all tasks are eligible to arrive.

An abstract state is a *failure state* if there is some task τ_i for which $laxity(S_i) < 0$, that is, if the remaining time to deadline is less than the remaining compute time of the current ready job of τ_i.

It follows from the construction of the model that a task system is schedulable to meet deadlines if-and-only-if no failure state is reachable from the start state. This can be tested using any finite graph reachability algorithm.

4 Clock-Tick Transitions

The clock-tick successor of a state S depends on the set $run(S)$ of tasks that the scheduling policy chooses to execute in the next instant. It is assumed that $run(S)$ includes only tasks τ_i that need to

execute (that is, $rct(S_i) > 0$) and that the scheduling policy is work-conserving (that is, the number of tasks in $run(S)$ is the minimum of the number of available processors and the number of tasks that are competing for processor time.

The clock-tick successor $S' = Next(S)$ of any abstract system state S is then computable as follows:

$$rct(S'_i) \stackrel{\text{def}}{=} \begin{cases} rct(S_i) & \text{if } \tau_i \notin run(S) \\ rct(S_i) - 1 & \text{if } \tau_i \in run(S) \end{cases}$$

$$nat(S'_i) \stackrel{\text{def}}{=} \begin{cases} \max(0, nat(S_i) - 1) & \text{if } rct(S_i) = 0 \\ nat(S_i) - 1 & \text{if } rct(S_i) > 0 \end{cases}$$

The reasoning behind the computation of $rct(S'_i)$ is simple. S' represents a state one time unit further into the future than S, so when going from S to S' the remaining compute time is reduced by one time unit for those jobs that are scheduled to execute, and is unchanged for those jobs that are not scheduled to execute.

The reasoning behind the computation of $nat(S'_i)$ involves two cases, based on whether there is a job of τ_i contending for processor time in S:
(i) If $rct(S_i) = 0$ then there is no job of τ_i contending for processor time, and the earliest time a job of τ_i can arrive is $nat(S_i)$ time units after S. If $nat(S_i) \leq 0$ the inter-arrival constraint permitted a job of τ_i to arrive at the time of S, but it did not arrive and so can still arrive at the time of S'; therefore $nat(S'_i) = 0$. Otherwise, $nat(S_i) > 0$ and the time to next arrival in S' should be one time unit sooner than in S, so $nat(S'_i) = nat(S_i) - 1$.
(ii) If $rct(S_i) > 0$ then there is a job J of τ_i contending for processor time in S, and if $d_i > p_i$, there may be one or more other backlogged jobs of τ_i that have arrived but are waiting for J to complete. Let J' be the next job of τ_i to arrive after J. The earliest time that J' could arrive is $nat(S_i)$ (positive or negative) time units from the time of S . Since S' represents a state one time unit later than S, the adjustment for the passage of one unit of time is $nat(S'_i) = nat(S'_i) - 1$.

At first it might seem that the value of $nat(S'_i)$ could decrease without bound in case (ii) if $d_i > p_i$. That is not possible, so long as we stop as soon as we find a failure state. The deadline constraint provides a lower bound on how far negative $nat(S'_i)$ can go without a job missing a deadline. That is, if S is a non-failure state and J has non-zero remaining compute time then the current time is certainly before J's deadline, the deadline of J' is at least p_i units in the future, and J' cannot arrive earlier than $\min(d_i - p_i, 0) + 1$ time units before S'. Therefore, the number of reachable non-failure states is finite.

5 Ready Transitions

The event of a new job becoming ready is modeled by a *ready* state transition. A job of a sporadic task may become ready any time that the preceding job of τ_i has completed execution and at least p_i time has elapsed since the preceding job arrived. In other words, a ready transition from state S for task τ_i is possible if-and-only-if $rct(S_i) = 0$ and $nat(S_i) \leq 0$.

For each task τ_i and each state S such that $rct(S_i) = 0$ and $nat(S_i) \leq 0$, the states S' to which a τ_i*-ready transition* is possible are all those that differ from S only in the values of $rct(S'_i)$ and $nat(S'_i)$ and satisfy the following:

$$rct(S'_i) = e_i, \quad nat(S_i) + p_i \leq nat(S'_i) \leq p_i$$

The reasoning is that a new job J' of τ_i does not become ready until the current job J of τ_i has been completed (indicated by $rct(S_i) = 0$), and the exact arrival time of J' does not matter until that instant. Then, the arrival time of J' can be chosen (non-deterministically) to be any instant from the present

back to $nat(S'_i)$ clock ticks into the past. It follows that the earliest arrival time of the next job of τ_i after J', relative to the present, can be any value between $nat(S_i) + p_i$ and p_i.

Ready transitions are non-deterministic, and are viewed as taking no time. Therefore, several of them may take place between one time instant and the next. $Ready(S)$ is the set of all states that can be reached by a sequence of one or more ready transitions from state S.

6 Specific Scheduling Policies

The basic abstract state model described above contains enough information to support the computation of the $run(S)$ for several scheduling policies, including the following global multiprocessor scheduling policies:

Fixed task priority: Assuming the tasks are ordered by decreasing priority, choose the lowest-numbered tasks.

Shortest remaining-processing first: Choose tasks with the smallest nonzero values of $rct(S_i)$.

Earliest deadline first (EDF): Choose tasks with the shortest time to next deadline. The time to next deadline of each task can be computed using equation (1).

Least laxity first (LLF): Choose tasks with the smallest laxities. The laxity of each task can be computed using equation (2).

Throwforward:[12]: Choose up to m tasks by following algorithm: (i) Choose the task with shortest $ttd(S_i)$. Let $t \stackrel{\text{def}}{=} ttd(S_i)$ for this task. (ii) Choose the tasks with positive throwforward on the above task, where the throwforward $TF(S_i)$ of task τ_i in state S is defined as follows:

$$TF(S_i) \stackrel{\text{def}}{=} t - (ttd(S_i) - rct(S_i))$$

It is clear that some hybrids of the above including earliest-deadline zero-laxity (EDZL) [9], and LLREF [8][1], can also be accommodated. Some more algorithms can also be supported, by adding information to the state; for example, to resolve priority ties in favor of the task that last executed on a given processor, it is sufficient to add one bit per task, indicating whether the task executed in the preceding time instant.

7 Generic Algorithm

It is clear that the set of all states reachable from the start state can be enumerated by breadth-first or depth-first search. Suppose the state set *Known* starts out containing just the start state. The objective of the algorithm is to compute the closure of this set under all legal clock-tick and ready transitions, by iteratively *visiting* each member of the set, where visiting a state S involves: (a) determining the unique clock-tick successor, if there is one; (b) determining the set of states $Ready(S)$, including one state for each possible combination of new ready jobs. Each new state encountered in step (a) or step (b) is then added to the set of reachable states, *Known*. Thus, at any point in time the set *Known* includes both all the states that have been visited and some unvisited states whose predecessors have been visited.

To keep track of which states have been visited, the set *Unvisited* is introduced, which initially is the same as *Known*. When a new state is found, it is added to both *Known* and *Unvisited*. States to visit are chosen from *Unvisited*, and they are removed once they have been visited. Depending on whether *Unvisited* is organized as a stack or a queue, this results in either a depth-first or a breadth-first enumeration of reachable states. The algorithm terminates when either a failure state is found or there are no remaining states left in *Unvisited*.

[1]These algorithms are known to be predictable in the sense of Ha and Liu[11].

```
BRUTE(τ)
 1  S ← ((0,0), ..., (0,0))
 2  Unvisited ← {S} ∪ Ready(S)
 3  Known ← Unvisited
 4  while Unvisited ≠ ∅ do {
 5     choose S ∈ Unvisited
 6     Unvisited ← Unvisited −{S}
 7     repeat
 8        if ∃i  laxity(S, τ_i) < 0 then  return 0                ▷ failure
 9        S ← Next(S)
10        if S ∉ Known then  {
11           Known ← Known ∪ {S}
12           for S′ ∈ Ready(S) do
13              if S′ ∉ Known then  {
14                 Known ← Known ∪ {S′}
15                 Unvisited ← Unvisited ∪ {S′}
16              }
17        } else  S ← undefined
18     until S = undefined
19  }
20  return 1                                                        ▷ success
```

Figure 1. Pseudo-code for brute-force schedulability test.

The algorithm BRUTE, whose pseudo-code is given in Figure 1, differs from the above abstract description by taking the following two shortcuts.
(i) Since there is a most one state that can be reached by clock-tick transition from any given state, the step of inserting clock-tick successors into *Unvisited* is skipped, and the clock-tick successor is visited immediately after its clock-tick predecessor. In this way, the algorithm proceeds depth-first along the (deterministic) clock-tick transitions, and queues up in *Unvisited* all the unknown states that can be entered via (non-deterministic) ready transitions from states encountered along the depth-first path. When the algorithm reaches a state from which no further clock-tick transitions are possible, it backtracks to the next unvisited state.
(ii) Since the set $Ready(S)$ is closed under ready transitions, there is no need to enumerate further ready transitions from states in $Ready(S)$. Therefore, step (2) of visitation is skipped for states that are added via step (2).

Note that the chain of states S' enumerated by the depth-first repeat-until loop (7) is completely determined by the state S chosen at (5). There is a possibility that the chains of states for two different values of S may converge to a common state, after which the two chains will be the same. Such repetition of states entered on a scheduling transition will be detected at line (14), terminating the repeat-until loop early via (17) if S' is found in *Known*. This avoids re-traversal of convergent chains, and limits the aggregate number of iterations of the repeat-until loop (summed over all iterations of the while-loop) to one per state reachable by a scheduling transition.

This algorithm is just one of several state enumeration algorithms that would work. The combination of depth-first search for clock-tick transitions and breadth-first search for ready transitions was chosen

because it seemed to result in failures being detected earlier, on the average, than with pure breadth-first or depth-first search.

8 Computational Complexity

It is clear that algorithm *Brute* visits each node and edge of the state graph at most once, so the worst-case computation complexity can be bounded by counting the number of possible states.

Theorem 1 *The worst-case complexity of deciding schedulability on an abstract system-state graph for a task system τ of n tasks is $\mathcal{O}(N \cdot (1 + 2^n))$, and N is an upper bound on the number of system states, where*

$$N = \prod_{i=1}^{n} ((e_i + 1)(\min(0, d_i - p_i) + p_i + 1)) \tag{3}$$

proof:

Clearly, the range of values for $rct(S_i)$ is in the range $0 \ldots e_i$. From the definition of $nat(S_i)$ and the reasoning in the last paragraph of Section 4, it is clear that the range of values for $nat(S_i)$ is in the range $\min(0, d_i - p_i) + 1 \ldots p_i$. Therefore, an upper bound on the number of nodes in the state graph bounded by the value N given in (3).

The number of edges per node is at most $1 + 2^n$; that is, one for the clock-tick transition and at most 2^n for the various combinations of possible ready transitions. Therefore, the number of edges is grossly bounded by

$$E \leq N \times (1 + 2^n) \tag{4}$$

The theorem follows. □

Theorem 1 gives an upper bound on the number of iterations of the innermost loop of algorithm Brute. The primitive operations on the set *Known* can be implemented in average-case $\mathcal{O}(n)$ time (to compare the n elements of a state) using a hash table, and the primitive operations on the set *Unvisited* can be implemented in time $\mathcal{O}(n)$ (to copy new states) using a stack. It follows that the worst-case complexity of the algorithm is at most $\mathcal{O}(n \cdot N \cdot (1 + 2^n))$, where N is the upper bound on the number of states derived in Theorem 1.

Assuming that e_i and p_i fit into a standard-sized word, the storage size of one state is $\mathcal{O}(n)$ words, and so an upper bound on the storage complexity of the algorithm is $\mathcal{O}(n \cdot N)$,

These bounds grows very quickly, both with the number of tasks and the sizes of the task periods and deadlines. On the other hand, the bound is based on counting the entire domain of all possible states, and over-bounding the number of state transitions, whereas the algorithm considers only reachable states and terminates soon as it finds a failure state. Therefore, the actual numbers of states and transitions considered by the algorithm will be less than the upper bound.

9 Performance

We have implemented and tested algorithm Brute for all the specific scheduling policies mentioned as examples in Section 6. Because of the large storage and execution time requirements (especially storage), we have only been able to test it on small task sets. Nevertheless, the results are interesting. Due to the page limit of this conference, only the results of one experiment are provided here.

The experiments whose results are shown in Figures 2 through 5 are for global EDF scheduling of pseudo-randomly generated task sets, on two processors. The periods (p_i) are uniformly distributed in the range $1 \ldots p_{\max}$, for $p_{\max}$ in the range $3 \ldots 6$. (We ran out of memory for $p_{\max} = 6$.) The utilizations

($u_i = e_i/p_i$) are exponentially distributed with mean 0.35. The deadlines (d_i) are uniformly distributed in the range $[u_i p_i, p_i]$ (constrained deadlines), or the range $[u_i p_i, 4 * p_i]$.

Duplicate task sets were discarded, as were task sets that could be obtained from other task sets by re-ordering the tasks or scaling all the task parameters by an integer factor. In addition, task sets that were trivially schedulable by global EDF because $n \leq m$ or $\sum_{i=1}^{n} e_i / \min(d_i, p_i) \leq 1$, or obviously unschedulable by any algorithm because maxmin load $> m$ [4] were also discarded. The experiment was run on 100,000 task sets that passed this initial screening.

The following tests were applied to each task set:

- Quick EDF: The sufficient schedulability test for global EDF scheduling of Bertogna, Cirinei, and Lipari[6] and the density test $\sum_{i=1}^{N} \frac{c_i}{\min\{d_i, T_i\}} \leq m$, (also described in [6]) were computed. Since task sets that passed either of these quick tests are schedulable, they were not subjected to the brute force test.
- Brute EDF: Algorithm BRUTE for global EDF scheduling was applied.

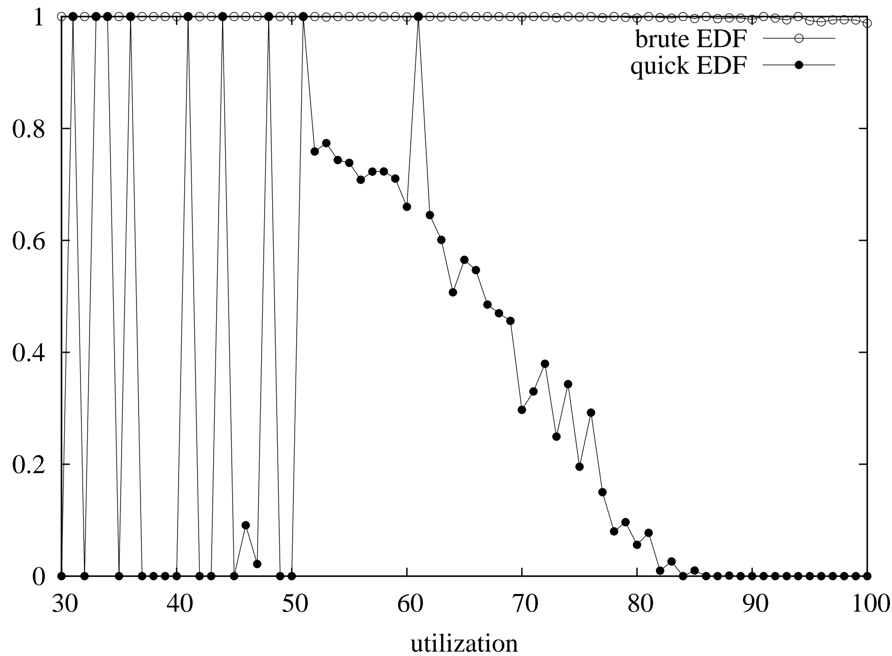

Figure 2. Success of quick versus brute-force tests for global EDF scheduling, for task sets with integer periods in the range $1 \dots 5$ and unconstrained deadlines.

Figure 2 is a normalized histogram in which the X axis corresponds to a total processor utilization value and the Y axis corresponds to the ratio of the number of task sets that could be shown to be schedulable to the number of task sets tested, for the utilization range $[X, X + 0.01)$. A Y-value of 1.0 indicates that all task sets were verified as being schedulable. The jaggedness of the graph is due primarily to the small range of integer values $(0 \dots 5)$ permitted for periods and execution times, which produces only a small number of possible utilization values and makes some values more probable than others. The jaggedness is further exaggerated by the small sample size.

It can be seen that the quick EDF tests are rather pessimistic, especially at higher utilization levels. The EDF performance, according to the brute force test, appears to be near 100 percent. That may

seem suspiciously good, considering that some of the task sets tested might be infeasible (not schedulable by EDF of any other algorithm. Part of the explanation is that rule for choosing deadlines is strongly biased toward post-period deadlines. Such task sets tend to be feasible, and to be EDF-schedulable. The performance of global EDF was much worse on other experiments, shown in Figure 3, where the task deadlines were constrained to be less than or equal to the periods.

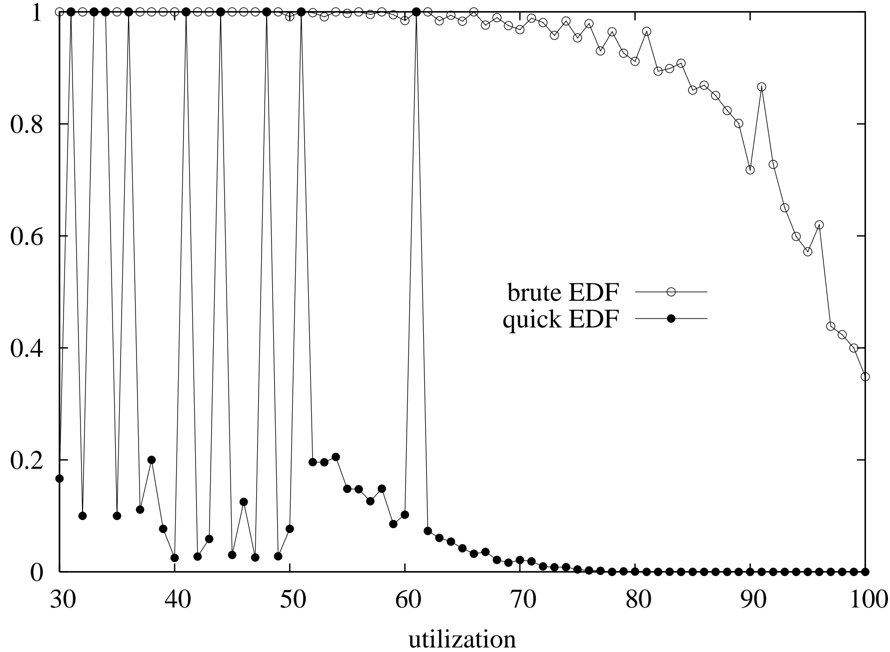

Figure 3. Success of quick versus brute-force tests for global EDF scheduling, for task sets with integer periods in the range $1 \ldots 5$ and constrained deadlines.

Part of the explanation also is that task sets with smaller integer periods tend to be more readily schedulable at high utilization levels than task sets with a larger range of periods. This is apparent in Figure 4, which shows the distribution of the number of states explored by the brute-force EDF algorithm for several values of $p_{\max}$.

Figure 5 shows the distribution of the number of states explored by the brute-force EDF algorithm This makes it clear why the experiments were limited to task sets with very small integer values for periods, deadlines, and execution times.

10 Conclusion

Schedulability of sporadic task systems on a set of identical processors can be decided in finite time for several global preemptive priority-based scheduling policies, using a generic brute-force enumerative algorithm. Gross upper bounds have been derived for the time and storage complexity of this approach.

Several prior publications have incorrectly asserted that schedulability of sporadic task systems under global EDF scheduling can be decided using a simpler algorithm, based on simulating execution when all tasks have jobs that arrive periodically, starting at time zero. To the best of our knowledge, ours is the first proposal of a correct algorithm for this problem. Moreover, it also applies to a variety of other priority-based global scheduling policies.

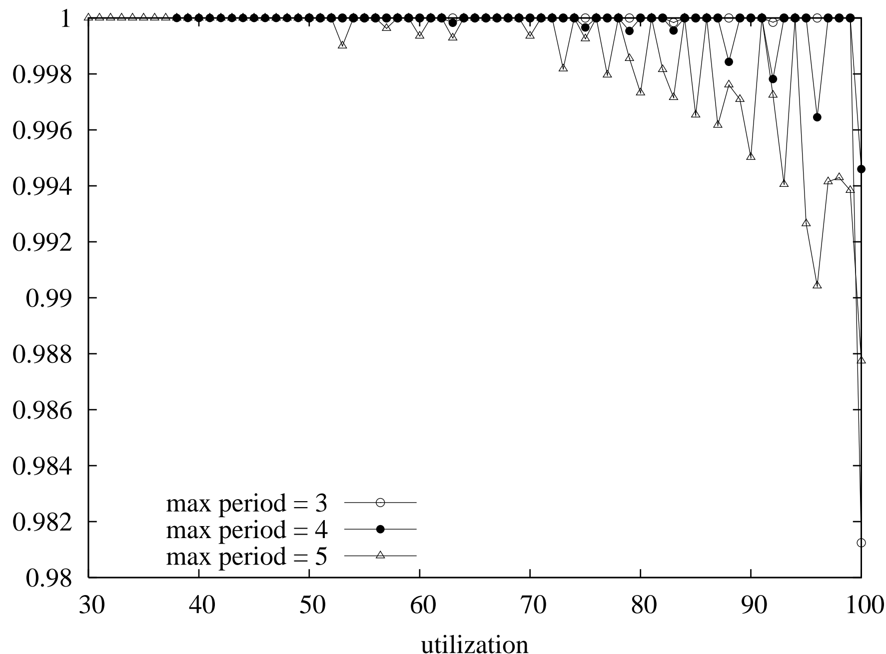

Figure 4. Success of brute-force test for global EDF scheduling with respect to time granularity

The algorithm has been implemented and tested on a variety of task sets, for several different scheduling policies. Exponential growth in the running time and storage, especially storage, limit the algorithm's application to small task sets.

Nevertheless, it has proven to be useful as a research tool, for finding examples of task sets that are schedulable by one method and not by another, and in providing insight into the degree to which more efficient but only sufficient tests of schedulability err in the direction of conservatism.

We hope that, by publishing this simple brute-force algorithm, we may establish a base-line and stimulate research into faster algorithms for this important class of scheduling problems.

References

[1] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Proc. 22nd IEEE Real-Time Systems Symposium*, pages 193–202, London, UK, Dec. 2001.

[2] T. P. Baker. An analysis of EDF scheduling on a multiprocessor. *IEEE Trans. on Parallel and Distributed Systems*, 15(8):760–768, Aug. 2005.

[3] T. P. Baker. An analysis of fixed-priority scheduling on a multiprocessor. *Real Time Systems*, 2005.

[4] T. P. Baker and M. Cirinei. A necessary and sometimes sufficient condition for the feasibility of sets of sporadic hard-deadline tasks. In *Proc. 27th IEEE Real-Time Systems Symposium*, Rio de Janeiro, Brazil, Dec. 2006.

[5] T. P. Baker, N. Fisher, and S. Baruah. Algorithms for determining the load of a sporadic task system. Technical Report TR-051201, Department of Computer Science, Florida State University, Tallahassee, FL, Dec. 2005.

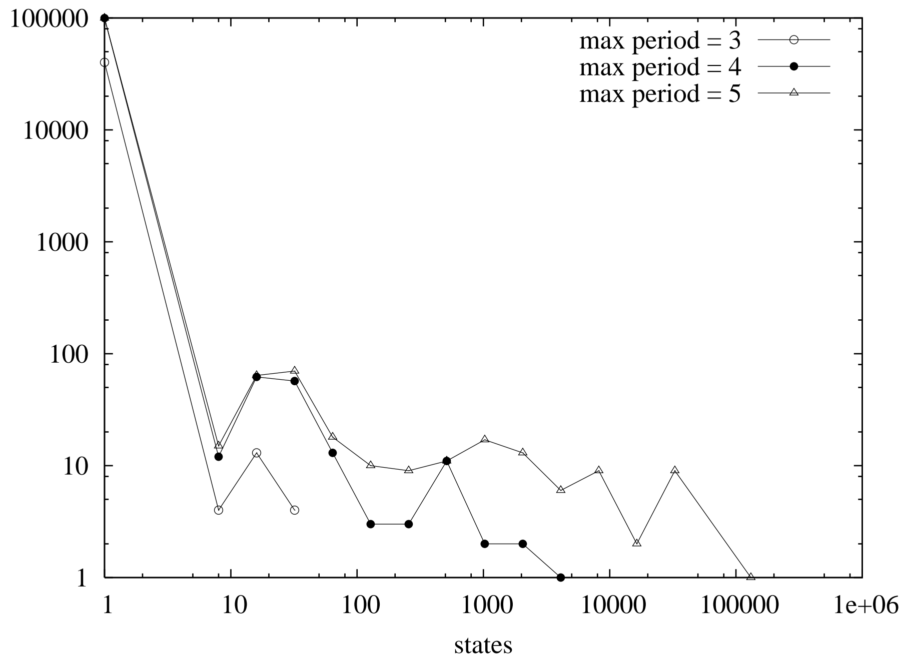

Figure 5. Number of states examined by brute-force EDF test

[6] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *Proc. 17th Euromicro Conference on Real-Time Systems*, pages 209–218, Palma de Mallorca, Spain, July 2005.

[7] M. Bertogna, M. Cirinei, and G. Lipari. New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors. In *Proc. 9th International Conf. on Principles of Distributed Systems*, Pisa, Italy, Dec. 2005.

[8] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proc. 27th IEEE International Real-Time Systems Symposium*, Rio de Janeiro, Brazil, Dec. 2006.

[9] S. Cho, S.-K. Lee, A. Han, and K.-J. Lin. Efficient real-time scheduling algorithms for multiprocessor systems. *IEICE Trans. Communications*, E85-B(12):2859–2867, Dec. 2002.

[10] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real Time Systems*, 25(2–3):187–205, Sept. 2003.

[11] R. Ha and J. W. S. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proc. 14th IEEE International Conf. Distributed Computing Systems*, pages 162–171, Poznan, Poland, June 1994. IEEE Computer Society.

[12] H. H. Johnson and M. S. Maddison. Deadline scheduling for a real-time multiprocessor. In *Proc. Eurocomp Conference*, pages 139–153, 1974.

[13] A. Srinivasan and S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84:93–98, 2002.