Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks
TR-061001

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

(preliminary report)

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 execution time and storage complexity of the the algorithm, which performs an exhaustive search of a very large state space, make it practical only for tasks sets with with very small integer periods. However, as a research tool, it can provide a clearer picture than has been available heretofor 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 aplied

*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.

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, 4, 5, 6, 9, 13]. For example, it can be shown that a set of independent 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, 9].

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 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 execution states. Section 4 shows how this model fits several well known global multiprocessor scheduling policies. Section 5 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 6 provides a coarse estimate of the worst-case time and storage requirements of the brute-force algorithm. Section 7 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) execution 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. A 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 execution time. A job is *backlogged* if it has nonzero remaining execution 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 time any time t, if job J has remaining execution 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, execution 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 lengths of the task periods, 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, 10] defined the concept of predictable scheduling policy. Consider any two sets $\mathcal{J}$ and $\mathcal{J}'$ of jobs that only differ in the execution times of the jobs, with the execution times of jobs in $\mathcal{J}'$ being less than or equal to the execution times of the corresponding jobs in $\mathcal{J}$. A scheduling policy is defined to be *predictable* (with respect to completion times) if, in scheduling such $\mathcal{J}$ and $\mathcal{J}'$ (separately), the completion time of each job in $\mathcal{J}'$ is always no later then the completion time of the corresponding job in $\mathcal{J}$. That is, with a predictable scheduling policy it is sufficient, for the purpose of bounding the worst-case response time of a task or proving schedulability of a task set, to

look just at the jobs of each task whose actual execution times are equal to the task's worst-case execution time.

Not every scheduling policy is predictable in this sense. There are well-known examples of scheduling anomalies, in which shortening the execution time of one or more jobs in a task set that is schedulable by a given algorithm can make the task set un-schedulable. In particular, such anomalies have been observed for non-preemptive priority-driven scheduling policies on multiprocessors. Ha and Liu proved that all preemptable 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 execution requirement is equal to the worst-case execution 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 execution.

If $rct(S_i)$ is zero there is no job of τ_i contending for execution. 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 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 execution 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 $nat(S)$ 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 execution 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

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

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 $nat(S_i) \leq 0$. The *τ_i-ready successor state* S' of S is the same as S except that $rct(S'_i) = e_i$ and $nat(S'_i) = nat(S_i) + p_i$.

Ready transistions 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 ready transistions from state S.

The passage of one instant of time is modeled by a *clock-tick* state transition, which is deterministic. The clock-tick successor of a state S depends on the set $run(S)$ of tasks that the scheduling policy choses 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 maximum of the number of available processors and the number of tasks that need to execute.

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

For each $\tau_i \in run(S)$, $rct(S'_i) = rct(S_i) - 1$, and for each $\tau_i \notin run(S)$, $rct(S'_i) = rct(S_i)$.

For each τ_i such that $rct(S_i) = 0$

1. If $nat(S_i) = 0$ then $nat(S'_i) = 0$,
2. If $nat(S_i) > 0$ then $nat(S'_i) =$

$$\max\left(nat(S_i) - 1, \min(0, p_i - d_i)\right) \tag{3}$$

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 $nat(S'_i)$ starts out equally simple, but then is refined to avoid large negative values when $d_i > p_i$. We consider two cases, based on whether there is a job of τ_i contending for execution in S.

If $rct(S_i) = 0$ then there is no job of τ_i contending for execution, and the earliest time a job of τ_i can arrive is $nat(S_i)$ time units after S. If $nat(S_i) = 0$, that means the inter-arrival constraint permitted a job of τ_i to arrive at the time of S, but since it did not arrive then it can still arrive at the time of S', and so $nat(S'_i) = 0$. Otherwise, $nat(S_i) > 0$ and the time to next arrival in S' should be one time unit sooner than in S, so $nat(S'_i) = nat(S_i) - 1$.

If $rct(S_i) > 0$ then there is a job J of τ_i contending for execution 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. Assuming $nat(S_i)$ is correct, 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 normal adjustment for the passage of one unit of time would be $nat(S'_i) = nat(S'_i) - 1$.

However, the deadline constraint provides a lower bound on how far negative this value can go without a job missing a deadline. That is, S is a non-failure state and J has non-zero remaining execution 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'. Equation (3) follows.

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

The system *start state* is defined to be $(0, \ldots, 0)$. That is, the state in which there are no tasks contenting for execution, 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 execution 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 a failure state is reachable from the start state. This can be tested using any finite graph reachability algorithm.

4 Specific Scheduling Policies

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

Fixed task priority: Choose the lowest-numbered tasks with $rct(S_i) > 0$.

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, from those with $rct(S_i) > 0$. The time to next deadline of each task can be computed by subtracting the slack time from the shortest permitted time to next arrival; that is,

$$time\_to\_deadline(S_i) = nat(S_i) - (p_i - d_i)$$

Least laxity first (LLF): Choose tasks with the smallest laxities from those $rct(S_i) > 0$. The laxity of of each task can be computed by subtracting the remaining execution time from the time to next deadline, as follows:

$$laxity(S_i) = ttd(S_i) - rct(S_i)$$

Throwforward: [12]: Choose up to m tasks by following algorithm:

(1) Choose the task with shortest $ttd(S_i)$ from the tasks with $rct(S_i) > 0$. Let $t \stackrel{\text{def}}{=} ttd(S_i)$ for this task.

(2) 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 hybrids of the above, such as earliest-deadline zero-laxity (EDZL) [8], and LLREF [7], can also be accomodated. 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.

5 Generic Algorithm

The algorithm BRUTE, whose pseudo-code is given in Figure 1, uses a combination of depth-first and breadth-first search. The state set *Known* starts out containing just the start state, and is successively augmented with states that have been verified to be reachable from the start state. The set *Unvisited* contains the known states that are entered on ready transitions and have not yet been visited, where visiting a state entails enumerating all possible transitions from it. 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 dept-first path. When the algorithm reaches a state from which no further clock-tick transitions are possible, it backtracks to the next unvisited state. The algorithm terminates when either a failure state is found or there are no remaining states left in *Unvisited*.

Note that the chain of states S' enumerated by the dept-first while loop (7) is completely determined by the state S chosen at (r). 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 8, terminating the inner while-loop early at (18) if S' is found in *Known*. This avoids re-traversal of convergent chains, and limits the aggregate number of iterations of the inner while-loop (summed over all iterations of the outer while-loop) to one per state reachable by a scheduling transition.

6 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 possible states.

```
BRUTE(τ)
 1  Unvisited ← {((0,0,0),...,(0,0,0))}
 2  Known ← ∅
 3  while Unvisited ≠ ∅ do
 4    choose S ∈ Unvisited
 5    Unvisited ← Unvisited −{S}
 6    S′ ← Next(S)
 7    while S′ is defined do
 8      if ∃i  laxity(S′, τ_i) < 0 then
 9        return 0                                   ▷ failure
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        S′ ← Next(S)
17      else
18        S′ ← undefined
19  return 1                                         ▷ success
```

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

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*

$$N = \prod_{i=1}^{n} ((e_i + 1)(\min(0, d_i - p_i) + p_i + 1)) \tag{4}$$

proof:

Clearly, the range of values for $rct(S_i)$ is in the range $0 \ldots e_i$, and from (3) 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 (4).

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

$$E \leq N \times (1 + 2^n) \tag{5}$$

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.

7 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.

This generic algorithm has been implemented and tested on a variety of task sets, for the specific scheduling policies mentioned as examples above. The results of those experiments will be presented in an extende version of this report, to appear in the future.

In practice, the memory needed by BRUTE to keep track of previously visited states turned out to be a more serious constraint than the execution time. This constraint became apparent when the algorithm was coded and tested. It typically ran out of virtual memory after a few minutes of execution. Moreover, as the memory requirement grew so did the real execution time. That is, when *Known* becomes large enough to exceed the available real memory the operating system starts swapping virtual memory between real memory and backing store. This swapping activity causes the running time of the algorithm to climb much more steeply once the number of states exceeds what can be stored in real memory. Therefore, several techniques were tried to reduce the number of states that need to be stored. Those techniques, and how they affected the performance of the algorithm will be discussed in the full version of this report.

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, 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] R. Ha. *Validating timing constraints in multiprocessor and distributed systems*. PhD thesis, University of Illinois, Dept. of Computer Science, Urbana-Champaign, IL, 1995.

[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.