An Omniscient Scheduling Oracle for Systems with Harmonic Periods [*]

Alia Atlas and Azer Bestavros

Computer Science Department
Boston University
Boston, MA 02215
{akatlas, best}@cs.bu.edu

Abstract

Most real-time scheduling problems are known to be NP-complete. To enable accurate comparison between the schedules of heuristic algorithms and the optimal schedule, we introduce an omniscient oracle. This oracle provides schedules for periodic task sets with harmonic periods and variable resource requirements. Three different job value functions are described and implemented. Each corresponds to a different system goal.

The oracle is used to examine the performance of different on-line schedulers under varying loads, including overload. We have compared the oracle against Rate Monotonic Scheduling, Statistical Rate Monotonic Scheduling, and Slack Stealing Job Admission Control Scheduling. Consistently, the oracle provides an upper bound on performance for the metric under consideration.

1 Introduction

It has been shown that most real-time scheduling problems are NP-complete. For periodic task systems with constant resource requirements, two on-line algorithms — Earliest Deadline First (EDF) and Rate Monotonic Scheduling (RMS)[1] — provide optimal schedules for a system which is not in overload.

To consider the performance of EDF, RMS, and other scheduling algorithms, the goal of an overloaded system must be defined. Given an objective function to maximize, the performance of an on-line algorithm can be compared to a theoretical perfect knowledge optimal algorithm. In this paper, we present such an algorithm — an oracle. We used a dynamic programming approach to create an algorithm which runs in pseudo-polynomial time. This oracle works with task systems with harmonic periods.

An oracle is necessary to enable comparisons between the optimal solution and the solution of a feasible algorithm. A common approach to approximation algorithms is to devise a scheme where the closeness of the approximation can be analytically proven. However, in scheduling, an oracle is assumed to have perfect knowledge of all tasks and their jobs, past, current and future. This omniscience permits a better solution than is possible by an on-line algorithm, which does not possess all future knowledge. Rather than attempting to derive an omniscient approximation algorithm, we use an an omniscient oracle to unfairly compare against on-line scheduling algorithms.

[*]This work was partially supported by NSF research grant CCR-9706685.

This paper is organized as follows. In Section 2, we discuss related work. In Section 3, we provide the task and job models used by the oracle. In Section 4, we describe the pseudo-polynomial time algorithm which comprises the oracle. In Section 5, we present simulations comparing the oracle with Rate Monotonic Scheduling (RMS), Statistical Rate Monotonic Scheduling (SRMS), and Slack Stealing Job Admission Control Scheduling (SSJAC). Finally, in Section 6, we conclude with a summary of this work and directions for future research.

2 Related Work

Conventional scheduling theory concentrates on the problem of minimizing a specific objective function, given different task constraints. In [2], Lawler examined the problem of minimizing the number of late jobs, where each job has a release time, a deadline, and a weight or importance value. For the scheduling problem with preemption, Lawler presented an off-line polynomial-time algorithm for the case where all jobs are equally important. The same algorithm has pseudo-polynomial time complexity for differently weighted jobs. This scheduling problem has jobs with soft deadlines, where the work must be completed even after the job is late.

The problem of scheduling systems which cannot support any late jobs is considered by Earliest Deadline First (EDF) and Rate Monotonic Scheduling (RMS)[1]. Both are on-line algorithms which schedule periodic tasks. The resource requirement of a periodic task is assumed to be known and constant. With this information, both EDF and RMS provide feasibility tests [1, 3, 4] to guarantee that no deadlines are missed. EDF was proved to be optimal among any dynamic priority algorithms. Similarly, RMS was proved optimal among any fixed priority algorithms. However, the behavior of these algorithms in overload is not optimal.

Different system goals have been proposed for overloaded systems. If the system goal is to maximize the number of deadlines met, which is the completion count (CC), Baruah, Haritsa, and Sharma proved in [5] that on an arbitrary workload, an on-line algorithm can perform arbitrarily bad compared to a perfect knowledge optimal algorithm. For restricted workloads, however, the authors proved better results. If the system goal is to maximize the effective processor utilization (EPU), no on-line algorithm can attain a system value larger than one quarter of that obtained by the perfect knowledge optimal algorithm.

An overloaded system which includes optional work can first discard this work. This is the idea of incremental tasks [6], where each task has a mandatory part and an optional part. In [7], Koren and Shasha consider overloaded systems where some portion of the jobs can just be skipped. Associated with each periodic task is a skip value, s_i. One job every s_i jobs of the task τ_i can be skipped.

In [8, 9], we present Statistical Rate Monotonic Scheduling, a generalization of RMS which supports QoS specification and tasks with variable resource requirements. Due to the variability in requested utilization, SRMS must deal with the possibility of overload in any system. When a task attempts to exceed its guaranteed share of the resource, the jobs requesting the excess are considered optional and discarded. In this way, a statistical QoS guarantee can be provided for each task and overload is managed on an individual task basis.

In [10], we introduced Slack Stealing Job Admission Control (SSJAC) [10] to schedule firm deadline tasks with variable resource requirements. Slack stealing was used to admit or reject jobs. Associated with each task is a threshold. Jobs with resource requirements below the threshold were automatically admitted. Jobs with resource requirements above that threshold were considered for admittance based upon the slack in the system at their priority level.

If the task model does not include optional work, then the system goal is to ensure that all critical tasks meet their deadlines and that a minimum of non-critical tasks miss their deadlines. In [11], Marucheck and Strosnider provided a taxonomy of scheduling algorithms with varying overload-awareness and critical-awareness. In [12], Buttazzo, Spuri, and Sensini also presented a series of simulations comparing various scheduling algorithms in overload. They considered firm deadline aperiodic task loads, and compared dynamic priority algorithms.

3 Task and Job Models

In this section, we will describe the periodic task model which is used by on-line algorithms to enable comparison with the oracle. Then we will present the job model which the oracle, with omniscient knowledge, has available and uses. For both models, deadlines are assumed to be firm. If a job cannot complete by its deadline, no system value is ever gained or lost by working upon it.

Definition 1 *A periodic task, τ_i is a three-tuple, (P_i, $f_i(x)$, w_i), where P_i is the task's period, $f_i(x)$ is the probability density function (PDF) for the task's periodic resource utilization requirement, and w_i is the task's importance value.*

This task model is similar to the classical task model used for RMS and EDF [1]. A new job of the task is released at the start of the period and is due at the end of that period. There are no non-zero task phasings. The first job of all tasks is available at time zero.[1] The task periods of the system are harmonic.

Definition 2 *Harmonic Periods Assumption : $P_i < P_j \Longrightarrow P_i | P_j$.*

Definition 3 *A job, $\tau_{i,j}$, is a four-tuple, ($r_{i,j}$, $d_{i,j}$, $e_{i,j}$, $v_{i,j}$), where $r_{i,j}$ is the release and ready time, $d_{i,j}$ is the deadline, $e_{i,j}$ is the resource requirement, and $v_{i,j}$ is the value of completing the job by its deadline.*

The job model used by the oracle is derived from the task model used by other scheduling algorithms. The release times and deadlines of each job can be trivially computed from the task's period. The resource requirement, $e_{i,j}$, of each job is known by the oracle due to its omniscient. It remains to describe how the job value, $v_{i,j}$, is assigned.

3.1 Job Value Assignment

To enable the oracle to determine the system benefit of completing different jobs, each job must have a value, $v_{i,j}$, assigned to it. The larger the value, the higher the benefit of completing the given job. The system value only increases when a job is completed on-time; the completed job's value is added to the system. Due to our model of firm task deadlines, no value is gained by the system for late or incomplete jobs.

Depending upon the value assignments, different objective functions will be minimized. First, we will assume that all tasks have the same importance. In this case, there are three clear objective functions: completion count, job failure rate, and effective processor utilization.

[1]The lack of task phasings is necessary to simplify the problem of identifying overloaded time intervals. It also enables the oracle to solve seperately each interval corresponding to the longest period of any task in the system.

Completion count (CC)[5] minimizes the number of deadlines which are missed. Each job of every task is assigned an identical value. Jobs which require more resources will be preferentially rejected, because that rejection will free more resources than rejecting a job with fewer resource requirements. This strategy is extremely familiar from RMS. Using CC provides an optimal solution to the same scheduling problem as RMS, but with firm deadlines and the ability to pre-reject incompleted jobs.

Therefore, CC biases the algorithm towards short jobs. If all tasks are of equal importance, then this is not the proper objective function to minimize. For all tasks to be considered equal, over a given time interval, the sum of a task's jobs' values should be equal to those of any other task. The easiest time interval to consider is the least common multiple (LCM) of the tasks' periods. Therefore, each job should be assigned a value equal to the length of its period. The objective function thus optimized is the job failure rate.

Finally, the effective processor utilization (EPU) assigns to each job a value equal to its resource requirement. This function maximizes the useful utilization provided by the processor. The longer jobs will be favored by the scheduler, naturally.

None of these methods consider task importance. However, if the tasks have varying importance, this weight, w_i could be be used to modify the value given by an objective function as follows:

$$v_{i,j}^{weighted} = v_{i,j} * w_i$$

4 Oracle Algorithm

In this section, we present the algorithm used for the oracle. First, we discuss the overall organization of the algorithm. Then we describe how to detect overloaded intervals. Finally, we present a dynamic programming algorithm to eliminate overload in those intervals.

4.1 Structure of Optimal Algorithm

Earliest Deadline First (EDF) [1] is known to be optimal for systems which are never in overload. Therefore, an optimal algorithm must first detect and eliminate overload. Once the system is never overloaded, EDF can be used to schedule the remaining jobs. Thus, the algorithm described in this chapter focuses on detecting and eliminating overload.

As described in Section 3, we assume that the task system has harmonic periods. This makes overload detection trivial. It also decreases the size of the intervals which can independently scheduled. Each least common multiple of the task periods, P_{LCM}, can be independently scheduled, because no job will have a release time in one P_{LCM} and a deadline in the next. For systems with harmonic periods, P_{LCM} is equal to the longest period of the system's tasks.

The system can be scheduled in sections of length P_{LCM}. For each section, the algorithm must first determine the overloaded time intervals. An overloaded interval may be completely contained by another overloaded interval. Such intervals are identified as *nested* intervals. Each non-nested overloaded time interval needs to be examined and its load reduced.

Each such interval must have jobs accepted or rejected such that the utilization of the accepted jobs does not overload that interval or any nested overloaded intervals. This is similar to the 0-1 Knapsack problem, which either accepts an object or rejects it based upon its value and its weight. Both can be

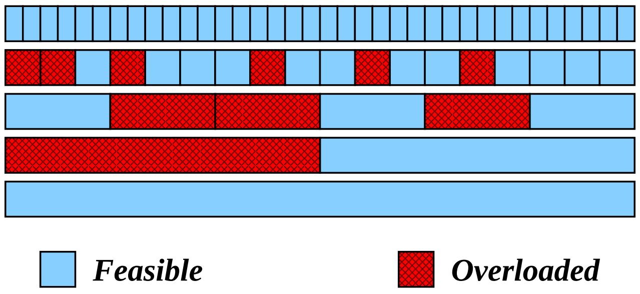

Figure 1: Example System with Overloaded Intervals

solved using dynamic programming. In the overload elimination problem, however, there is the added complication of ensuring that a solution does not cause nested intervals to remain overloaded.

Once this overload elimination has been done on all overloaded non-nested intervals, the list of accepted jobs is known. Those jobs are guaranteed to be schedulable using EDF. This completely solves the problem of creating an omniscient oracle for systems with harmonic periods. The complexity of the algorithm depends upon the detection and elimination of overloaded intervals. For determining the computational complexity of these algorithms, let N be the number of jobs to schedule in a given section of length P_{LCM}.

4.2 Overload Detection

In this section, we introduce the DETECT-OVERLOAD algorithm, which determines which intervals are overloaded. First, we consider under what circumstances a job requires resources during a given interval. Second, we limit the potential intervals to examine based upon the previous observations. Finally, we provide the algorithm and a brief analysis of its computational complexity.

Trivially, if a time interval contains both the release time, $r_{i,j}$, and the deadline, $d_{i,j}$, of a job, that job requires its full resource during that interval. That job is said to be *fully contained* in the interval. If a job is not fully contained in the interval, that job requires no resources during the interval. The resources required of a given interval are the sum of the resources requirements of all jobs fully contained in that interval. If the resource required of an interval are greater than available in that interval, then it is overloaded. Specifically, if the sum of the fully contained jobs' execution times is greater than the length of the interval, the interval is overloaded.

Next, the time intervals to be examined must be considered. The smallest interval to consider is P_1, the shortest period in the task system. No smaller interval could fully contain a job. The longest interval to consider is P_{LCM}, the longest period in the task system. All jobs to be scheduled in a given section will be fully contained in that P_{LCM}. Within those bounds, to be considered are each time interval which starts at the release time of a given job and ends at the deadline of the same job. Such an interval is the smallest interval which fully contains the given job, and, therefore, contains more work than any shorter interval. Any interval which does not meet the above criteria will simply be the union of other intervals, which have been considered. Therefore, it suffices to consider only the intervals corresponding to a job in the system:

$$TI = ([P_i * (j-1), P_i * j] \mid \forall i, j)$$

An example task system is shown in Figure 1. In this system, DETECT-OVERLOAD has been run, and the overloaded time intervals have been identified in red. Notice that the longest period is not overloaded, but without rejecting jobs, the system is still not feasible. The algorithm is shown in Figure 2. Definitions for the terms used are provided in Table 1. The time complexity of the DETECT-OVERLOAD algorithm is $\Theta(N^2)$. The space complexity is $\Theta(N)$.

JobList	List of jobs in the system with their specific information
release	Release time of the given job
deadline	Deadline time of given job
capacity	Time in the given job's period (deadline - release)
execTime	Execution time for given job
requestedTime	Execution time requested in this job's time
OverloadList	List of overloaded intervals
parent	Pointer to entry whose value was used to derive value

Table 1: Definitions of Terms used in Oracle Algorithms

4.3 Overload Elimination

From the algorithm described in the previous section, a list of the overloaded intervals can be obtained. Given those intervals, a list of the jobs in each interval can be trivially determined, using interval boundaries and job release times and deadlines. With this information, ELIMINATE-OVERLOAD determines which jobs to accept or reject in each overloaded interval, so as to produce a feasible job list with no overloaded time intervals.

Before ELIMINATE-OVERLOAD can be run, the overloaded intervals must be organized. If an overloaded interval is fully contained within another overloaded interval, then the former is associated with the latter and it is removed from consideration. At the end of this process, a list of independent overloaded intervals has been constructed. Associated with each independent overloaded interval is a list of those nested intervals which are overloaded. These lists can be trivially constructed based upon comparison of the intervals.

As can be seen in Figure 3, an example section of a five task system has been organized as discussed. Interval 1 fully contains 5 overloaded intervals. Interval 2 does not fully contain any other overloaded intervals. Interval 3 fully contains one other interval.

Dynamic programming is used to eliminate the overload from each independent overloaded interval. First, consider interval 2, as given in Figure 3, which does not fully contain any other overloaded interval. In such a case, the interval can be independently solved. Therefore, each job in the overloaded interval is

```
DETECT-OVERLOAD(JobList)
    OverloadList $\leftarrow$ NULL
    foreach Interval $\in$ JobList
    Interval.capacity $\leftarrow$ Interval.deadline - Interval.release
      Interval.requestedTime = 0
      foreach Job $\in$ JobList
        if Job.release $\geq$ Interval.release AND Job.deadline $\leq$ Interval.deadline
          Interval.requestedTime $\leftarrow$ Interval.requestedTime + Job.execTime
      if Interval.requestedTime $>$ Interval.capacity
      OverloadList $\leftarrow$ Interval::OverloadList
    return OverloadList
```

Figure 2: Algorithm to Detect Overloaded Time Intervals

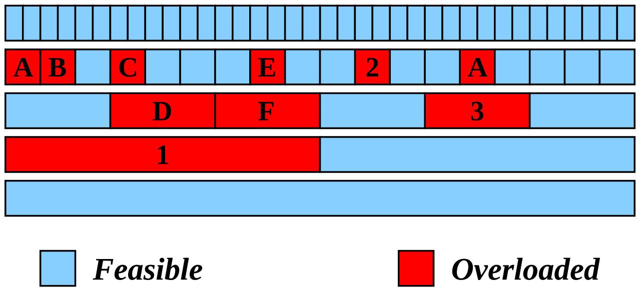

Figure 3: Example System with Overloaded Intervals Organized

not in any other overloaded interval.

To rephrase this independent problem, given a time capacity C and M jobs, $(v_{i,j}, e_{i,j})$, maximize the value of the jobs accepted, subject to the restraint that the summed resource requirement is not greater than the time capacity C. As can be clearly seen from this restating, this is identical to the 0-1 Knapsack problem, which can be solved by dynamic programming.

For dynamic programming, a table must be created and filled in incrementally. Each entry represents a solution of the specific subproblem. Thus a table of size $C+1$ by $M+1$ is created. The entry in $[c, m]$ is the solution for the problem with a capacity of c and only the first m jobs to be scheduled. The jobs can be ordered in any manner. The final solution is in $[C, M]$. The algorithm consists of a recurrence relationship, shown in Figurereffig:optRec1 to fill in the table entries. Additionally, the choice of how to fill in the entry must be recorded, as it determines whether the mth job should be accepted or rejected. The base cases where there are no jobs or there is no capacity are filled with a value of 0, and a DoJob of FALSE.

The recurrence in Figure 4 is not accurate if the interval fully contains any other overloaded intervals, as intervals 1 and 3 in Figure 3 do. The solution produced by the above recurrence is not constrained to remove the overload in the contained intervals. Therefore, it is necessary to consider those intervals.

```
if ((j < jobs[k].execTime) OR
   (table[j][k-1] ≥ table[j - jobs[k].execTime][k-1] + jobs[k].value))
    table[j][k].value ← table[j][k-1]
    table[j][k].DoJob ← FALSE
else
    table[j][k].value ← table[j - jobs[k].execTime][k-1] + jobs[k].value
    table[j][k].DoJob ← TRUE
```

Figure 4: Recurrence Without Nesting Intervals

Let there be T nested overloaded intervals in an independent interval, I. To eliminate overload, it is still necessary to accept and reject jobs so as to maximize the value gained. The time constraint is still C, the length of the interval I, but a solution yields no value if it overloads any of the T intervals fully contained therein. The solutions for those intervals do not need to be optimal; only the solution for I must be maximized. Thus, the requirement that fully contained intervals impose on the solution is that the solution not cause the nested intervals to be overloaded.

The modified recurrence relationship is presented in Figure 5. As discussed, the additional complexity results from the need to verify that no fully contained overloaded interval remains overloaded in the solution. As proven in Lemma 1, this recurrence suffices to properly eliminate overload.

Lemma 1 *Properly eliminating overloads in an interval implies that none of the fully contained subintervals is overloaded.*

Proof: Dynamic programming can be used to eliminate overloads in an interval. The recurrence to fill in the table is given in Figure 5. Using this recurrence, a subsolution is checked to determine if it overloads a fully contained subinterval; if so, that subsolution is rejected in favor of a previously acceptable subsolution. Thus, any subsolution which overloads a fully contained subinterval is rejected. Therefore, the final solution does not overload any fully contained subintervals.

■

Thus, the ELIMINATE-OVERLOAD algorithm consists of constructing a table and filling it in using the proper recurrence. The value in table entry (C, M) is the maximum which the system can gain. Of more interest, by following parent and DoJob, the jobs which have been admitted and rejected can be determined. For each table entry in the linked list pointed to by parent, (i, j), if DoJob is true, then job j should be admitted to the system. In this way, the list of admitted jobs can be constructed.

Once all independent overloaded intervals have had their overload eliminated, the lists of admitted jobs can be combined and submitted to an EDF scheduler to produce the actual schedule, if necessary. All admitted jobs are guaranteed to meet their deadlines.

The number of nested intervals is, in the very worst case, M, the number of jobs. Thus, the time complexity of this section, ELIMINATE-OVERLOAD, is $\Theta(C * M^3)$, where M is the number of jobs, and C is the length of the interval. The space complexity is $\Theta(C * M)$.

```
table[j]k].job = jobs[k]
if ((j < jobs[k].execTime) OR
  (table[j][k-1] $\geq$ table[j - jobs[k].execTime][k-1] + jobs[k].value))
   table[j][k].value $\leftarrow$ table[j][k-1]
   table[j][k].DoJob $\leftarrow$ FALSE
   table[j][k].parent $\leftarrow$ table[j][k-1]
   else
   table[j][k].value $\leftarrow$ table[j - jobs[k].execTime][k-1] + jobs[k].value
   table[j][k].DoJob $\leftarrow$ TRUE
   table[j][k].parent $\leftarrow$ table[j - jobs[k].execTime][k-1]
   foreach i $\in$ NestedIntervals
      JobPtr $\leftarrow$ table[j][k]
      spent $\leftarrow$ 0
      while (JobPtr $\neq$ NULL)
         if ((JobPtr->DoJob = TRUE) AND (JobPtr->release $\geq$ i.release)
           AND(JobPtr->deadline $\leq$ i.deadline))
              spent $\leftarrow$ spent + JobPtr->job->execTime
         JobPtr $\leftarrow$ JobPtr->parent
         if (spent > i.capacity)
         /* Interval overloaded, so clearly not optimal to include the kth job */
         table[j][k].value $\leftarrow$ table[j][k-1]
         table[j][k].DoJob $\leftarrow$ FALSE
         table[j][k].parent $\leftarrow$ table[j][k-1]
         break
```

Figure 5: Recurrence With Nesting Intervals

5 Simulation Experiments

In our experiments, we made a number of simplifying assumptions. These assumptions were necessary to allow for a more straightforward interpretation of the simulation results, by eliminating conditions or factors that are not of paramount interest to the subject matter of this paper (e.g. effects of task criticality). First, we assumed that all tasks demand the same average percentage utilization of the resource being managed. In other words, the ratio $\frac{E(e_{i,k})}{P_i}$ for all tasks is constant. Second, the probability distributions used to generate the resource requirements were of the same type[2] (but with different parameters) for each task in the system. Also, these distributions were truncated so that no infeasible jobs were submitted to the system. Third, we assumed that all tasks were of equal criticality/importance

To compare algorithms and discuss their characteristics, we define **job failure rate**. This metric corresponds to the oracle as OPT-T. This is plotted against the requested utilization of the system.

[2]We considered a variety of such distributions as will be evident later in this section.

Definition 4 *The job failure rate (JFR) is the average percentage of missed deadlines.*[3]

$$JFR = \frac{1}{n} * \sum_{i=1}^{n} \frac{\tau_i \ missed\ jobs}{\tau_i \ jobs}$$

Definition 5 *The requested utilization is the sum of all jobs' resource requirements divided by the time interval during which scheduling occurs.*

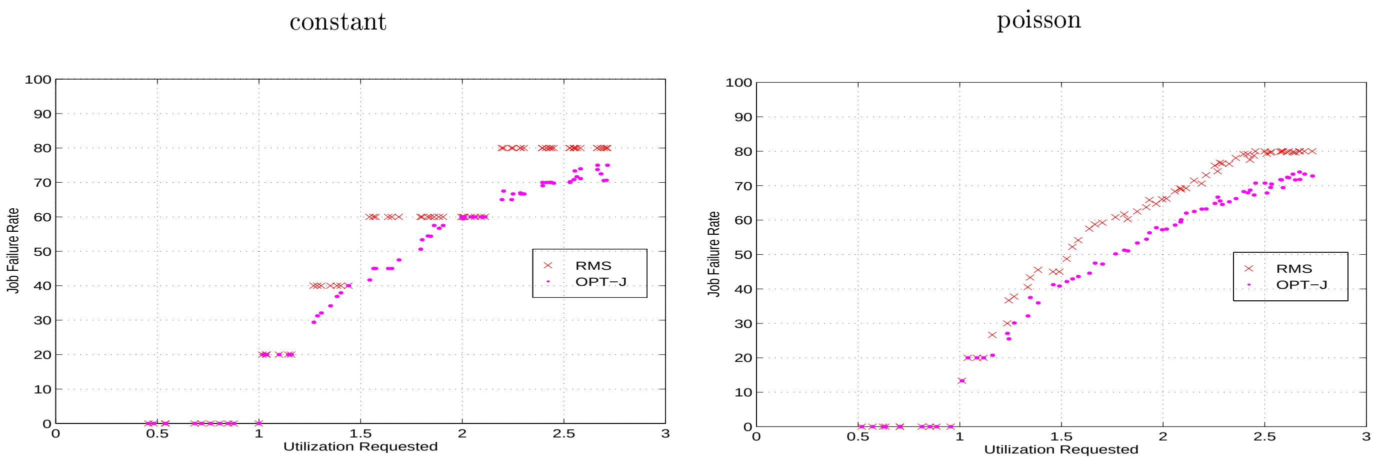

Figure 6: Job Failure Rate of RMS vs OPT-J

5.1 Simulation Experiments:

The set of simulation experiments which we conducted contained exactly five periodic tasks with harmonic periods. This permitted us to use the oracle to schedule these systems. The first period was fixed, and the remaining periods were chosen randomly, so that the ratio between adjacent periods was an *integer* uniformly distributed between two and four. For our experiments, we pre-determined the resource requirement of each job, so that all algorithms were run on the identical scheduling problem. While we ran sets of different random systems, the results presented below show one run of a given set of randomly generated systems and are representative.

Our experiments were run with different probability distributions used to generate the variable resource requests. We considered exponential, gamma, poisson, normal, uniform, and pareto distributions, as well as constant resource requirements, to determine if the gross behavior of the algorithms changed. We found that it did not. In this paper we restrict our presentations to the results obtained for constant resource requirements and for the gamma distribution. The former is the default assumption used in the classical task model.

We compare RMS against OPT-J. Both attempt to maximize the completion count. Figure 6 shows that OPT-J forms a clear performance upper bound for RMS, which is expected because OPT-J yields the

[3]This is the opposite of the **job completion rate** used in [11], which is the average percentage of met deadlines.

optimal schedule to maximize the completion count. RMS attempts to maximize the completion count by giving preference to tasks with shorter periods (i.e. those likely to contribute "more" to the completion count due to their frequent jobs).

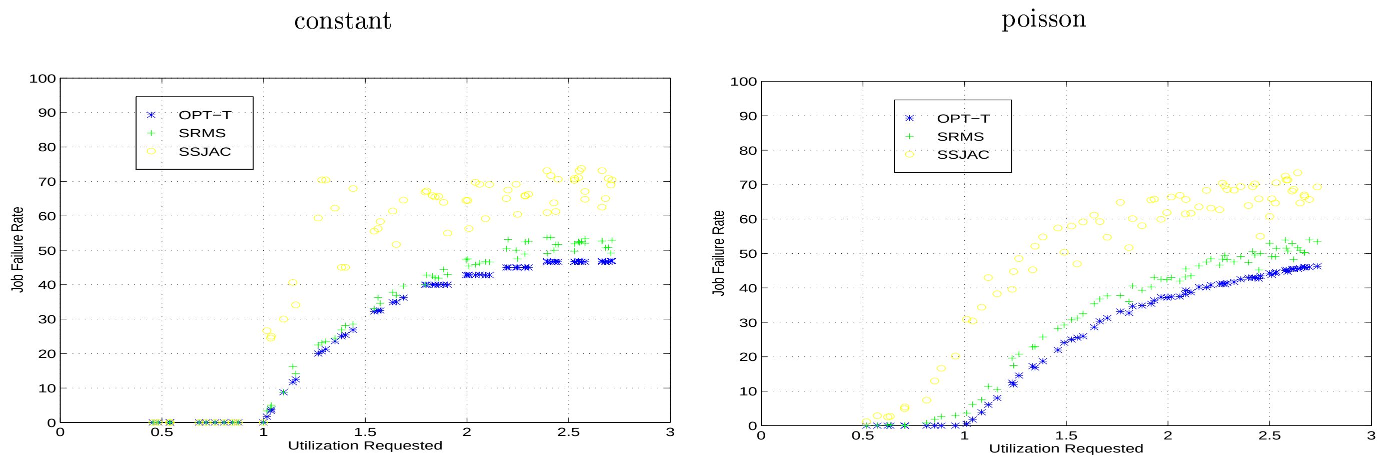

Figure 7: Job Failure Rate of SRMS and SSJAC vs OPT-T

OPT-T minimizes the job failure rate. As can be seen in Figure 7, OPT-T forms a clear performance upper bound for both SRMS and SSJAC. This is the anticipated results, because, with all tasks given the same percentage utilization (and requesting the same percentage utilization), both SRMS and SSJAC attempt to fairly distribute the resource among all tasks.

6 Conclusion

In this paper, we have introduced an omniscient oracle which can provide an optimal solution based upon arbitrary objective functions, as specified by assigning values to jobs. We considered three different functions: completion count, job failure rate, and effective processor utilization. This oracle is effective for task systems with harmonic periods.

The oracle first determines which time intervals are overloaded, using DETECT-OVERLOAD. Then, it organizes the intervals and jobs into lists. Any overloaded interval which is not fully contained in another overloaded interval is considered independent, and can be scheduled independently of all other overloaded intervals. ELIMINATE-OVERLOAD is used upon all such independent overloaded intervals to determine which jobs should be admitted to the system. Those admitted jobs are all guaranteed to meet their deadlines, if scheduled with EDF.

The clear drawback of this oracle is that it requires omniscience and pseudo-polynomial time and space. Therefore, it is useful as an benchmark for more practical algorithms. For example, theoretical results about the competitiveness of different algorithms do not yield useful information about an algorithm's performance on a given task set. Using this oracle, the best on-line algorithm to use for a given system can be determined. The flexibility of the oracle in accepting different job value functions permits systems with different system goals to use the oracle for design.

Future work includes the generalization of the oracle to arbitrary periods. To do so requires a solution to the following problem. Given a set of jobs and an interval in which all of those jobs can be at least partly scheduled, how should the work required by a job in that interval be determined? A job may have been partially scheduled in a previous interval, or may be scheduled in a future interval. EDF could be used to determine which jobs will have completed before the interval starts. Once it can be determined how to charge a job to an interval which doesn't fully contain it, the same dynamic programming algorithm can be used. There are extra complications, such as the fate of a job which was admitted in a given interval but rejected in another. In this case, the job is clearly rejected, but can another job now be accepted by the given interval?

Other than the tricky problem of generalizing the omniscient oracle to systems with arbitrary periods, future work remains in analyzing different job value functions. In this paper, we considered the completion count, the job failure rate, and the effective processor utilization as viable metrics. Consideration of weighted task sets remains to be examined and simulated.

References

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

[2] E. L. Lawler. A dynamic programming algorithm for preemptive scheduling on a single machine to minimize the number of late jobs. *Annals of Operations Research*, 26:125–133, 1990.

[3] John Lehoczky, Lui Sha, and Ye Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real-Time Systems Symposium*, pages 166–171, 1989.

[4] Ching-Chih Han and Hung ying Tyan. A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms. In *Real-Time Systems Symposium*, pages 36–45, 1997.

[5] Sanjoy Baruah, Jayant Haritsa, and Nitin Sharma. On line scheduling to maximize task completions. In *Real-Time Systems Symposium*, pages 228–237, December 1994. URL is http://www.emba.uvm.edu/ sanjoy/Papers/cc-jnl.ps.

[6] Jen-Yao Chung, Jane W. S. Liu, and Kwei-Jay Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, 39(9):1156–1174, September 1990.

[7] Gilad Koren and Dennis Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Real-Time Systems Symposium*, 1995.

[8] Alia K. Atlas and Azer Bestavros. Statistical rate monotonic scheduling. Technical Report BUCS-TR-98-010, Boston University, Computer Science Department, May 1998. Accepted for publication in IEEE Real-Time Systems Symposium, December 1998, Madrid, Spain.

[9] Alia K. Atlas and Azer Bestavros. Multiplexing vbr traffic flows with guaranteed application-level qos using statistical rate monotonic scheduling. Technical Report BUCS-TR-98-011, Boston University, Computer Science Department, 1998.

[10] Alia K. Atlas and Azer Bestavros. Slack stealing job admission control. Technical Report BUCS-TR-98-009, Boston University, Computer Science Department, 1998.

[11] M. Marucheck and J.K. Strosnider. An evaluation of the graceful degradation properties of real-time schedulers. In *The Twenty Fifth Annual International Symposium on Fault-Tolerant Computing*, June 1995.

[12] Giorgio Buttazzo, Marco Spuri, and Fabrizio Sensini. Value vs. deadline scheduling in overload conditions. In *Real-Time Systems Symposium*, pages 90–99, 1995.