Technical Report

Scheduling Arbitrary-Deadline Sporadic Task Systems on Multiprocessors

Björn Andersson, Konstantinos Bletsas and Sanjoy Baruah

HURRAY-TR-080501

Version: 0

Date: 09-09-2008

Scheduling Arbitrary-Deadline Sporadic Task Systems on Multiprocessors

Björn Andersson, Konstantinos Bletsas and Sanjoy Baruah

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: bandersson@dei.isep.ipp.pt, ksbs@isep.ipp.pt, baruah@cs.unc.edu

http://www.hurray.isep.ipp.pt

Abstract

A new algorithm is proposed for scheduling preemptiblearbitrary-deadline sporadic task systems upon multiprocessor platforms, with interprocessor migration permitted. This algorithmis based on a task-splitting approach --- while most tasks areentirely assigned to specific processors, a few tasks (fewer thanthe number of processors) may be split across two processors. This algorithm can be used for two distinct purposes: for actuallyscheduling specific sporadic task systems, and for feasibilityanalysis. Simulation-based evaluation indicates that this algorithm offers asignificant improvement on the ability to schedulearbitrary-deadline sporadic task systems as compared to thecontemporary state-of-art.With regard to feasibility analysis, the new algorithm is proved tooffer superior performance guarantees in comparison to prior feasibility tests.

Scheduling Arbitrary-Deadline Sporadic Tasks on Multiprocessors

Björn Andersson* Konstantinos Bletsas* Sanjoy Baruah†

Abstract

A new algorithm is proposed for scheduling preemptible arbitrary-deadline sporadic task systems upon multiprocessor platforms, with interprocessor migration permitted. This algorithm is based on a task-splitting approach — while most tasks are entirely assigned to specific processors, a few tasks (fewer than the number of processors) may be split across two processors. This algorithm can be used for two distinct purposes: for actually scheduling specific sporadic task systems, and for feasibility analysis. Simulation-based evaluation indicates that this algorithm offers a significant improvement on the ability to schedule arbitrary-deadline sporadic task systems as compared to the contemporary state-of-art. With regard to feasibility analysis, the new algorithm is proved to offer superior performance guarantees in comparison to prior feasibility tests.

1 Introduction

Consider the problem of preemptively scheduling n sporadically arriving tasks on m identical processors. A task τ_i is uniquely indexed in the range $1..n$ and a processor likewise in the range $1..m$. A task τ_i generates a (potentially infinite) sequence of jobs. The arrival times of these jobs cannot be controlled by the scheduling algorithm and are *a priori* unknown. We assume that the time between two successive jobs by the same task τ_i is at least T_i. Every job by τ_i requires at most C_i time units of execution over the next D_i time units after its arrival. We assume that T_i, D_i and C_i are real numbers and $0 \le C_i \le D_i$. A processor executes at most one job at a time and a job is not permitted to execute on multiple processors simultaneously.

A task set can, depending on its deadlines, be categorized as having *implicit deadlines*, *constrained deadlines* or *arbitrary deadlines*. A task set is said to be of implicit deadlines if $\forall i : D_i = T_i$. A task set is said to be of constrained deadlines if $\forall i : D_i \le T_i$. Otherwise, a task set is said to be of arbitrary deadlines. We consider the arbitrary-deadline model in this paper. In this model, a job may arrive although the deadline of a previously released job of the same task has not yet expired. Because of this, there may be instants where two jobs of the same task are ready for execution. We require that two jobs of the same task are not permitted to execute simultaneously. With this requirement, it follows that if a task τ_i has $C_i > T_i$ then it is impossible to schedule that task to meet deadlines. For this reason, we assume $C_i \le T_i$.

Algorithms for scheduling sporadic task systems on multiprocessors have traditionally been categorized as *partitioned* or *global*. Global scheduling stores tasks which have arrived but not finished execution in one queue, shared by all processors. At any moment, the m highest-priority tasks among those are selected for execution on the m processors. In contrast, partitioned scheduling algorithms partition the task set such that all tasks in a partition are assigned to the same processor. Tasks may not migrate from one processor to another. The multiprocessor scheduling problem is thus transformed to multiple uniprocessor scheduling problems. This simplifies scheduling and schedulability analysis as the wealth of results in uniprocessor scheduling can be reused. Partitioned scheduling algorithms suffer from an inherent performance limitation in that a task may fail to be assigned to any processor although the total available processing capacity across all processors is large. Global scheduling has the potential to rectify this performance limitation. In fact, there exist a family of algorithms called *pfair* [8, 1] which are able to schedule tasks to meet deadlines even when up to 100% of the processing capacity is requested.

*IPP-HURRAY! Research Group, Polytechnic Institute of Porto (ISEP-IPP), Rua Dr. António Bernardino de Almeida 431, 4200-072 Porto, Portugal – bandersson@dei.isep.ipp.pt, ksbs@isep.ipp.pt

†Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599, USA – baruah@cs.unc.edu

Unfortunately, these algorithms have two drawbacks; they are only designed for implicit deadlines. Also, all task parameters must be multiples of a time quantum and in every quantum a new task is selected for execution. Preemption counts can thus be high [11].

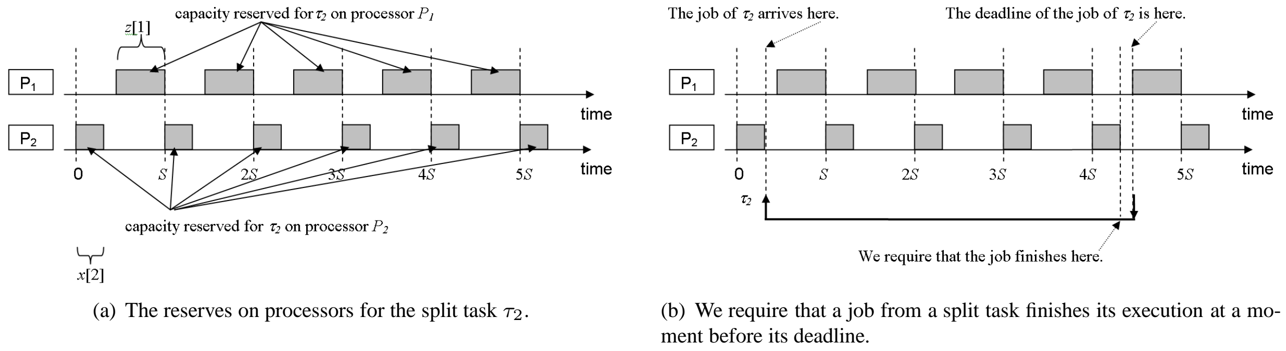
Recent advances in the real-time scheduling theory have however made a new class [2, 12, 10, 5, 3] of algorithms available with the purpose of offering the best of both global scheduling and partitioning. Tasks are assigned to processors in a way similar to partitioning but if the task cannot be assigned to a processor then the task is "split" into two pieces and these two "pieces" are scheduled by a uniprocessor scheduling algorithm on each processor. The splitting approach uses dispatchers on each processor which ensure that a split task never executes on two processors simultaneously.

The task splitting approach has been successfully, and widely, used in scheduling implicit-deadline systems. Anderson et al. [2] designed an algorithm which can miss deadlines but it offers the guarantee that even at high processor utilization, the finishing time is not too much later than its deadline and this amount is low and bounded. Kato et al. [12] designed an algorithm aiming to meet deadlines and it performed well in simulations. The task splitting approach has also been used [10, 5, 3] to design algorithms for both periodic and sporadic tasks such that if at most a certain fraction (greater than 50%) of the total processing capacity is requested then all deadlines are met. Despite these successes of the task splitting approach for scheduling implicit-deadline tasks, task splitting has not been used for scheduling arbitrary-deadline sporadic tasks. In fact, we are not aware of any previous work (with task splitting or some other approach) that aims for offering pre-run-time guarantees in scheduling arbitrary-deadline tasks on a multiprocessor with performance better than what partitioning schemes can offer.

Our algorithm. In this paper, we present a new algorithm for scheduling arbitrary-deadline sporadic tasks on a multiprocessor that is based on task splitting. We call this algorithm EDF-SS(DTMIN/δ), the name denoting **E**arliest-**D**eadline-**F**irst scheduling of non-split tasks with **S**plit tasks scheduled in **S**lots of duration DTMIN/δ. The symbol DTMIN denotes $\min(D_1,D_2,\ldots,D_n,T_1,T_2,\ldots,T_n)$, and δ is an integer parameter (≥ 1) that is selected by the designer. Thus for a given task system the value assigned to δ determines the scheduling slot size used. The smaller this slot size, the more frequently split tasks migrate between the processors that they utilize. As we will see, by choosing different values for δ the designer is able to trade off *schedulability* – the likelihood of a system being guaranteed schedulable – for a lower number of preemptions. More specifically, we will derive (Theorem 2) an upper bound on the number of preemptions per job in a schedule as a function of δ and the task parameters: the smaller the value of δ, the fewer the number of preemptions. We also quantify (in Section 2.3) the amount by which the execution requirements of jobs are "inflated" by our scheduling algorithm in order to guarantee that all deadlines are met: we will see that the larger the value of δ, the smaller the amount of such inflation. Thus a larger value of δ makes it more likely that a particular task system will be determined to be schedulable, but the generated schedule is likely to have a larger number of preemptions.

Using our algorithm. By choosing an appropriate value of δ, we can use our scheduling algorithm in two different ways. First it can be used by a designer to *schedule a specific task set*. For such a use, the run-time dispatching overhead is important. Our algorithm uses no global data-structures and is appropriate for that purpose. It is also important to maintain a low number of preemptions and hence the value of δ should not be too large. For example choosing δ such that $1 \leq \delta \leq 4$ seems reasonable. Baker [6] has conducted simulation experiments of randomly generated task sets of previously known algorithms for arbitrary-deadline sporadic multiprocessor scheduling and evaluated them for the context where pre-run-time guarantees are needed. Baker observed that generally speaking, partitioned algorithms tend to perform better than global scheduling algorithms perhaps because partitioned scheduling algorithms are based on uniprocessor schedulability tests whereas currently known global schedulability tests tend to be very pessimistic. The partitioning-based algorithm EDF-FFD [6] was found to be the champion among all algorithms studied. We compare our new algorithm EDF-SS(DTMIN/δ) with EDF-FFD in simulation experiments, with the same setup as Baker and we find that for every $\delta \geq 1$, EDF-SS(DTMIN/δ) offers a significant performance improvement.

A second use of our schedulability algorithm is in *feasibility analysis* — determining whether a given task system can be scheduled by a hypothetical optimal algorithm to always meet all deadlines. Currently no exact (necessary and sufficient) algorithms are known for performing feasibility analysis of arbitrary deadline sporadic task systems. In fact, all current feasibility tests are based on actually using a specific scheduling algorithm and performing schedulability analysis for it. As stated above, Baker [6] has observed that partitioning-based algorithms tend to perform better than global ones; by extending a previously-proposed partitioning-based scheduling algorithm [7] to allow for task-splitting, our algorithm, with $\delta \to \infty$, yields a sufficient feasibility test that is superior to previously-proposed sufficient tests.



(a) The reserves on processors for the split task τ_2.

(b) We require that a job from a split task finishes its execution at a moment before its deadline.

Figure 1. How to perform run-time dispatching of tasks that are assigned to two processors.

Organization. The remainder of this paper is organized as follows. Section 2 presents the theoretical basis of our new algorithm. Section 3 presents the new algorithm itself, and derives some of its properties while Section 4 discusses its use in feasibility testing. Section 5 presents the performance of the new algorithm obtained by running simulation experiments. Section 6 gives conclusions.

2 Conceptual foundations

Before describing the new algorithm, it is beneficial to get acquainted with the ideas behind its design and also certain concepts and results that will be used. Section 2.1 explains how task splitting is performed. Section 2.2 presents a dispatcher for use with task splitting. Section 2.3 performs dimensioning of the reserves used for execution of split tasks. Section 2.4 derives the amount of execution of split tasks. Section 2.5 presents a new schedulability analysis for real-time scheduling on a single processor.

2.1 Task Splitting

As mentioned in Section 1, partitioned scheduling offers the advantage that results from the vast body of uniprocessor scheduling theory can be reused to schedule tasks on a multiprocessor. The disadvantage of partitioned scheduling is also well-known: as tasks get assigned to processors, the remaining available capacity gets fragmented among the processors; it may then so happen that no individual fragment is large enough to accommodate an additional task (although the sum of these fragments would be sufficient for the additional task). *Task splitting* circumvents this problem by permitting that a task be split across multiple processors. And in fact this idea has been used by several researchers [2, 12, 10, 5, 3] for scheduling sporadic or periodic tasks with implicit deadlines. In this work, we apply task-splitting to scheduling sporadic tasks with arbitrary deadlines, by allowing an individual task to be split between two processors. Our approach is similar to previous work for implicit-deadline systems [3] but differs mainly (i) in the task assignment/splitting scheme used and (ii) in that, instead of a utilization-based schedulability test, a demand-based test is used, applicable to arbitrary-deadline systems.

Recall that our task model mandates that each task may be executing on at most one processor at each instant in time. Task splitting must therefore address two important challenges (i) create a dispatching algorithm for ensuring that two pieces of a task do not execute simultaneously and (ii) design a schedulability test for the dispatching algorithm. (Observe that because of (i), there is no need for the source code or the binary corresponding to a task to be restructured.)

2.2 Dispatching

Figure 1(a) provides further details on run-time dispatching. We subdivide the time into time slots of duration S=DTMIN/δ. The time slots on any two different processors are synchronized in time. On processor p, the beginning of a time slot is reserved for executing a split task τ_i shared with processor p-1; let $x[p]$ denote the duration of this reserve. Analogously, the final part of a time slot on processor p is reserved for executing a split task shared with processor p+1; let $z[p]$ denote the duration of this reserve.

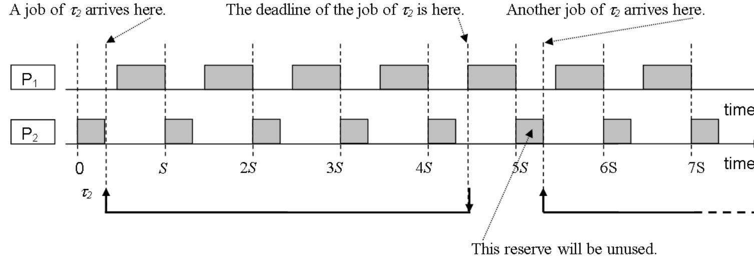


Figure 2. An example showing that at certain instants, a reserve may be unused.

The dispatching is simple. If processor p is in a reserve at time t and if the split task assigned that reserve has unfinished execution then the split task is executed in that reserve on processor p at time t. Otherwise, the non-split task, assigned to processor p, with the earliest deadline is selected for execution.

2.3 Dimensioning of the Reserves

A task that is split over processors $p-1$ and p executes during the reserves $z[p\text{-}1]$ and $x[p]$. For $\delta < \infty$, we will see that the total amount of these reserves must exceed the split task's execution requirement in order to guarantee that the split task meets its deadline. We now derive "safe" values for these reserves, which will ensure that all jobs of the split tasks are guaranteed to meet their deadlines. In deriving these values, the objective is to minimize the inflation – the amount by which the reserves exceed the execution requirement.

In order to meet deadlines we need to ensure that for each job $J_{i,k}$ of task τ_i it holds that whenever $J_{i,k}$ arrives, it completes C_i time units at most D_i time units after its arrival. For split tasks, we have chosen to impose an even stronger requirement; we require that within

$$\left\lfloor \frac{\min(D_i, T_i)}{S} \right\rfloor \cdot S \tag{1}$$

time units after its arrival, the job $J_{i,k}$ should complete C_i time units. Figure 1(b) illustrates this. Equation 1 implies that deadlines of split tasks are met and it also implies that a split task finishes before it arrives again. Observe that the duration in Equation 1 is an integer multiple of S. We know that for any time interval of duration S (which does not necessarily start at a slot boundary), it holds that the amount of execution available (in reserves) for a split task τ_i is $x[p]+z[p\text{-}1]$. Therefore we obtain that during a time interval of duration given by Equation 1, the amount of execution available for the split task τ_i assigned to processor p and processor p-1 is exactly

$$\left\lfloor \frac{\min(D_i, T_i)}{S} \right\rfloor \cdot (x[p] + z[p-1]) \tag{2}$$

In order to meet deadlines we should select $x[p]$ and $z[p\text{-}1]$ such that the expression in Equation 2 is at least C_i. We are interested also in using "small reserves" and for this reason we choose $x[p]$ and $z[p\text{-}1]$ such that it is equal to C_i. Hence we select $x[p]$ and $z[p\text{-}1]$ such that

$$x[p] + z[p-1] = \frac{C_i}{\left\lfloor \frac{\min(D_i, T_i)}{S} \right\rfloor} \tag{3}$$

Clearly, the choice by Equation 3, gives us that whenever a job of a split task τ_i is released, it finishes within at most $\lfloor \min(D_i, T_i)/S \rfloor \cdot S$ time units, as desired (see Equation 1 above). A property which we will find useful in the next section that follows from Equation 3 is that if a job released by a split task τ_i executes for C_i time units then it executes exactly

$$\left\lfloor \frac{\min(D_i, T_i)}{S} \right\rfloor \cdot x[p] \tag{4}$$

time units on processor p and

$$\left\lfloor \frac{\min(D_i, T_i)}{S} \right\rfloor \cdot z[p-1] \tag{5}$$

time units on processor p-1.

2.4 Execution by Split Tasks

In order to perform schedulability analysis (in Section 2.5 below), we must obtain an upper bound on the amount of execution that the split tasks perform on a processor p during any time interval of duration L. We now derive such an upper bound. This upper bound is given in Equation 8; the reader may wish to skip its derivation (i.e., the remainder of Section 2.4) at a first reading and return to it later.

It is easy to see that if jobs of a split task during this time interval execute for less than their maximum execution time then we can obtain another scenario with no less execution of split tasks on processor p during L, by letting jobs execute by their maximum amount[1]. Hence, we will assume that all jobs from split tasks execute for their maximum execution time. Let us define *slotexec* as:

$$\mathrm{slotexec}(t,r) = \left\lfloor \frac{t}{S} \right\rfloor \cdot r + \min(t - \left\lfloor \frac{t}{S} \right\rfloor \cdot S, r) \tag{6}$$

Slotexec gives us an upper bound on the amount of execution performed by a split task in a time interval of duration t, assuming that it executes in all of its reserves and assuming that each reserve assigned to that task has the duration r. This bound can be used to find an upper bound on the amount of time that split tasks execute on processor p.

But consider the example in Figure 2. It shows a task τ_2 split between processors 2 and 1. The task has $T_2 = 5 \cdot S$ and $D_2 = 4.2 \cdot S$. For the scenario in Figure 2, it follows that during $[4.4 \cdot S, 5.3 \cdot S)$, the reserve on processor 2 for task τ_2 is unused. For this reason, we need to calculate an upper bound on the amount of execution of split tasks by taking the parameters of the split tasks into account. We will do so now.

Let $\tau_{hi,p}$ denote the task that is split between processor p and processor p-1. Analogously, let $\tau_{lo,p}$ denote the task that is split between processor p and processor p+1.

With this definitions, one can show (see [4, Appendix A]) that the amount of execution of task $\tau_{hi,p}$ on processor p during a time interval of duration L is at most:

$$\begin{aligned} &\left\lfloor \frac{L+S-x[p]}{T_{hi,p}} \right\rfloor \cdot C_{hi,p} \cdot \frac{x[p]}{x[p]+z[p-1]} + \\ &\mathrm{slotexec}(\min(L+S-x[p] - \left\lfloor \frac{L+S-x[p]}{T_{hi,p}} \right\rfloor \cdot T_{hi,p}, \\ &\left\lfloor \frac{\min(D_{hi,p}, T_{hi,p})}{S} \right\rfloor \cdot S), x[p]) \end{aligned} \tag{7}$$

Intuitively, Equation 7 can be understood as follows. When finding the amount of execution of $\tau_{hi,p}$ during L, it is actually necessary to consider arrivals of $\tau_{hi,p}$ during $L+S-x[p]$ because the task $\tau_{hi,p}$ may arrive $S-x[p]$ time units before the beginning of the interval of duration L. The time interval of duration $L+S-x[p]$ can be subdivided into $\left\lfloor \frac{L+S-x[p]}{T_{hi,p}} \right\rfloor$ time intervals each one of duration $T_{hi,p}$ and one time interval of duration
$L+S-x[p] - \left\lfloor \frac{L+S-x[p]}{T_{hi,p}} \right\rfloor \cdot T_{hi,p}$.

Applying the reasoning of Equation 7 to $\tau_{lo,p}$ and adding them together gives us that the amount of execution of split tasks on processor p during a time interval of duration L is at most:

[1]For split task τ_i with $D_i \leq T_i$ it is easy to see this because increasing the execution time of a job of such a task does not affect the scheduling of any other jobs of split task τ_i. For the case of a task τ_i with $D_i > T_i$ it may not be so easy to see because one may think that two jobs of a task may be active simultaneously and then increasing the execution time of the former job may change the starting time of execution of the latter job. This cannot happen though because of our choice expressed in Equation 3.

$$\left\lfloor \frac{L+S-x[p]}{T_{hi,p}} \right\rfloor \cdot C_{hi,p} \cdot \frac{x[p]}{x[p]+z[p-1]} +$$
$$\text{slotexec}(\min(L+S-x[p] - \left\lfloor \frac{L+S-x[p]}{T_{hi,p}} \right\rfloor \cdot T_{hi,p},$$
$$\left\lfloor \frac{\min(D_{hi,p}, T_{hi,p})}{S} \right\rfloor \cdot S), x[p]) +$$
$$\left\lfloor \frac{L+S-z[p]}{T_{lo,p}} \right\rfloor \cdot C_{lo,p} \cdot \frac{z[p]}{x[p+1]+z[p]} +$$
$$\text{slotexec}(\min(L+S-z[p] - \left\lfloor \frac{L+S-z[p]}{T_{lo,p}} \right\rfloor \cdot T_{lo,p},$$
$$\left\lfloor \frac{\min(D_{lo,p}, T_{lo,p})}{S} \right\rfloor \cdot S), z[p]) \tag{8}$$

2.5 Schedulability Analysis

The processor demand [9] is a well-known concept for feasibility testing on a single processor. Intuitively, the processor demand represents the amount of execution that must necessarily be given to a task in order to meet its deadline. The processor demand of a task τ_j in a time interval of duration L is the maximum amount of execution of jobs released by τ_j such that these jobs (i) arrive no earlier than the beginning of the time interval and (ii) have deadlines no later than the finishing of the time interval. Let $\text{dbf}(\tau_j,L)$ denote the processor demand of task τ_j over a time interval of duration L. It is known [9] that:

$$\text{dbf}(\tau_j, L) = \max(0, \left\lfloor \frac{L-D_j}{T_j} \right\rfloor + 1) \cdot C_j \tag{9}$$

The processor demand of a task set is defined analogously. Let $\text{dbf}(\tau,L)$ denote the processor demand of the task set τ of a time interval of duration L. It is known that:

$$\text{dbf}(\tau, L) = \sum_{\tau_j \in \tau} \max(0, \left\lfloor \frac{L-D_j}{T_j} \right\rfloor + 1) \cdot C_j \tag{10}$$

One can show [9] that for EDF [13] scheduling on a single processor, it holds that all deadlines for tasks in τ are met if and only if $\forall L > 0$: $\text{dbf}(\tau_j,L) \leq L$. In fact, we only need to check this condition for the values of L where there is a non-negative integer k and a task τ_i such that $L{=}k \cdot T_i + D_i$ and L does not exceed $2 \cdot \text{lcm}(T_1,T_2,\ldots,T_n)$. This condition can be used as a schedulability test for partitioned scheduling where EDF is used on each processor. Recall however that the task splitting approach we use requires that split tasks are executed in special time intervals in the time slots and hence a new schedulability analysis must be developed. We know that if the processor demand of the non-split tasks during a time interval L (expressed by Equation 10) plus the amount of execution of the split tasks (expressed by Equation 8) does not exceed L then all deadlines are met. Let us define:

$$
\begin{aligned}
f(L) = \sum_{\tau_j \in \tau^p} \max(0, \left\lfloor \frac{L - D_j}{T_j} \right\rfloor + 1) \cdot C_j + \\
\min(L, \quad \left\lfloor \frac{L + S - x[p]}{T_{hi,p}} \right\rfloor \cdot C_{hi,p} \cdot \frac{x[p]}{x[p] + z[p-1]} + \\
\text{slotexec}(\min(L + S - x[p] - \left\lfloor \frac{L + S - x[p]}{T_{hi,p}} \right\rfloor \cdot T_{hi,p}, \\
\left\lfloor \frac{\min(D_{hi,p}, T_{hi,p})}{S} \right\rfloor \cdot S), x[p]) + \\
\left\lfloor \frac{L + S - z[p]}{T_{lo,p}} \right\rfloor \cdot C_{lo,p} \cdot \frac{z[p]}{x[p+1] + z[p]} + \\
\text{slotexec}(\min(L + S - z[p] - \left\lfloor \frac{L + S - z[p]}{T_{lo,p}} \right\rfloor \cdot T_{lo,p}, \\
\left\lfloor \frac{\min(D_{lo,p}, T_{lo,p})}{S} \right\rfloor \cdot S), z[p]) \quad)
\end{aligned}
\tag{11}
$$

Clearly this gives us that

$$
\begin{aligned}
&\text{if} \\
&\forall L, L = k \cdot T_i + D_i, L \leq 2 \cdot \text{lcm}(T_1, T_2, \ldots, T_n) : \\
&\qquad f(L) \leq L \\
&\qquad \text{then all deadlines are met}
\end{aligned}
\tag{12}
$$

2.6 Faster Schedulability Analysis

Consider the schedulability analysis in Equation 12. It can be seen that a large number of values of L must be explored. Our task splitting approach requires that several such schedulability tests are performed and consequently we are interested in creating a faster schedulability analysis. Let us define L_{lim} as follows:

$$
L_{lim} = \frac{(\sum_{\tau_j \in \tau^p} C_j) + 2 \cdot S + T_{hi,p} + T_{lo,p}}{1 - ((\sum_{\tau_j \in \tau^p} \frac{C_j}{T_j}) + \frac{C_{hi,p}}{T_{hi,p}} \cdot \frac{x[p]}{x[p]+z[p-1]} + \frac{C_{lo,p}}{T_{lo,p}} \cdot \frac{z[p]}{x[p+1]+z[p]})}
\tag{13}
$$

Let us consider the case that the denominator of the right-hand side Equation 13 is positive. Let DMAX denote $\max(D_1, D_2, ..., D_n)$. Then it holds (see Appendix B in Technical Report HURRAY-TR-080501 available at http://www.hurray.isep.ipp.pt) that $f(L) \leq L$ for all $L \geq L_{lim}$. This idea was known for pure EDF scheduling on a uniprocessor [9] but now we have transferred the result to a uniprocessor with reserves. Formally, this is stated as follows:

$$
\begin{aligned}
&\text{if} \\
&\qquad \text{the denominator of the right} - \text{hand side of} \\
&\qquad \text{Equation 13 is positive} \\
&\qquad \text{and} \\
&\qquad \forall L, L = k \cdot T_i + D_i, \\
&\qquad L < \min(2 \cdot lcm(T_1, T_2, \ldots, T_n), \max(\text{DMAX}, L_{lim})) : \\
&\qquad f(L) \leq L \\
&\qquad \text{then all deadlines are met}
\end{aligned} \tag{14}
$$

2.7 Design Consideration of Task Assignment

Previous work on task splitting [3] for implicit-deadline tasks was based on next-fit bin-packing. Unfortunately, such use for scheduling arbitrary-deadline tasks can lead to poor performance, as shown in Example 1.

Example 1. *Consider three tasks and two processors. The tasks are characterized as T_1=L^2,D_1=L,C_1=L $\cdot$ (2/3+ϵ), T_2=L^2,D_2=L, C_2=L $\cdot$ (2/3+ϵ), T_3=L^2,D_3=1, C_3=2/3+ϵ. Let L denote a positive number much larger than one. And let ϵ denote a positive number much smaller than one. Note that τ_1 and τ_2 are identical and they have large D_i. The task τ_3 has small D_i however.*

A next-fit bin-packing algorithm with task splitting would start considering τ_1 and it would be assigned to processor 1. The next task to be considered would be τ_2 and it would be attempted to be assigned to processor 1 but the schedulability test will fails so the task will be split between processor 1 and processor 2. The capacity of processor 1 assigned to τ_2 will be 1/3-ϵ and the capacity assigned to τ_2 will be 1/3+2ϵ. The next task to be considered would be τ_3 and it would be considered for processor 2. Unfortunately, it cannot be assigned there. The reason is that during a time window of duration 1, the reserves for τ_2 will be busy by execution of τ_2 and this does not give enough time to τ_3 to execute and hence it would miss a deadline. Intuitively, this can be understood as priority inversion, where τ_3 is ready for execution and it has the shortest deadline but still τ_2 executes and τ_2 has a much longer deadline.

It can be seen, from Example 1, that it is advantageous to assign tasks such that tasks with large deadlines are non-split and tasks with short deadlines are split. A bin-packing algorithm based on first-fit (boosted with a task splitting approach in the end) might achieve that. But unfortunately, another performance penalty would arise, as shown in Example 2.

Example 2. *Consider m+1 tasks and m processors. The tasks are characterized as T_i=1,D_i=1,C_i=2/3+ϵ. Let ϵ denote a positive number much smaller than one.*

A first-fit bin-packing algorithm with task splitting would start by assigning one task to each processor. Then it would consider task τ_{m+1} and find that it cannot be assigned to any processor. Therefore splitting is attempted; the task is split into two pieces but even in this way, it cannot be assigned to two processors. It is necessary to split the task into three or more pieces in order to meet deadlines. Such splitting makes run-time dispatching non-trivial; we note in particular that the approach sketched on Figure 1 cannot be used.

We have seen, from Example 1 and Example 2, that (i) a processor should be “filled” before new processors are considered and (ii) tasks that are split should preferably have a small D_i. This can be achieved as follows.

Initialize a variable p to one; it denotes the processor under consideration. For whichever processor p is in consideration, all tasks which are not yet assigned are scanned in order of decreasing deadline. If a task can be assigned (according to a schedulability test) to processor p then it is assigned there; if not, we consider if the next task can be assigned to processor p, subject to assignments already made (and so on).

After the scanning of all tasks has finished, it holds that of all the tasks that have still not been assigned to any processor, none can be assigned to processor p, subject to assignments already made. At this point, so as to utilize whatever capacity is still remaining on processor p, we thus have to “split” one of those tasks and assign a piece of it to the processor. We select for this purpose, among all tasks not yet assigned assigned, the task with the smallest D_i. It is split and assigned to processors p and $p + 1$, such that as much execution as possible is assigned to processor p.

After that, the variable p is incremented by one and the procedure is repeated. Note that in this way, we obtain the two properties (i) and (ii) above and we can therefore expect good performance.

```
1.  for each processor p do
2.    x[p] := 0
3.    z[p] := 0
4.    τ[p] := ∅
6.  end for
7.  for each processor p with index p < m do
8.    shared_task[p] := NULL
9.  end for
10. p := 1
11. unassigned := τ
12. while unassigned ≠ ∅ do
13.   for each task τ_i ∈ unassigned, considered in descending
       order of D_i do
14.     τ[p] := τ[p] ∪ { τ_i }
15.     if the test Equation 14 succeeds for processor p
         and x[p]+z[p]≤S then
16.       unassigned := unassigned \ { τ_i }
17.     else τ[p] := τ[p] \ { τ_i } end if
18.   end for
19.   if unassigned ≠ ∅ then
20.     if p ≤ m-1 then
21.       i := index of the task in unassigned with least D_i
22.       sumreserve := right-hand side of Equation 3
23.       if sumreserve > S then
24.         p := p + 1
25.       else
26.         p := p + 1
27.         assign values to x[p] and z[p-1] such that z[p-1] is
             maximized and the test Equation 14 succeeds for p-1
             and x[p-1]+z[p-1] ≤ S and Equation 3 is true for p
28.         shared_task[p-1] := τ_i
29.         unassigned := unassigned \ { τ_i }
30.       end if
31.     else declare FAILURE end if
32.   end if
33. end while
34. declare SUCCESS
```

Figure 3. An algorithm for assigning tasks to processors (and performing splitting if needed).

3 The New Scheduling Algorithm

Let us now specify detailed pseudo-code for the new scheduling algorithm. The scheduling algorithm is comprised of two algorithms (i) one algorithm for assigning (and splitting if needed) tasks to processor and (ii) an algorithm for dispatching the tasks.

Figure 3 shows the algorithm for assigning tasks to processors. The main idea is to assign as many tasks as possible to the current processor p. The lines 13-18 do that. The tasks are considered for assignment in the order largest-D_i-first and the decision on whether a task can be assigned to processor p is made based on whether Equation 14 can guarantee this assignment. Once no additional task can be assigned to processor p, the unassigned task with the least D_i is selected and it is split between processor p and processor p+1. The lines 26-29 do that. The splitting is done such that as much as possible of the currently considered task is assigned to processor p and as little as possible is assigned to processor p+1. Clearly, this leaves as much room as possible on processor p+1 for other tasks. Equation 14 is used for this splitting.

A variable `sumreserve` is calculated (on line 22). It represents the sum of the reserve of the split task τ_i on processor p and the reserve of the split task τ_i on processor p+1. In order to ensure that the split task τ_i does not execute on two processors simultaneously, we compare `sumreserve` to S, the slot size. If `sumreserve` is greater then splitting is not possible and the algorithm continues without splitting that task. If `sumreserve` does not exceed S then splitting is performed. As already mentioned, this splitting is performed on lines 26-29.

Line 27 states that a maximization problem should be solved. Note that the right-hand side of Equation 11 is non-decreasing with increasing $z[p\text{-}1]$ and hence it can be solved easily.[2]

Figure 4 shows the algorithm for dispatching. It computes a time t_0 which is the beginning of the time slot and a time t_1 which is the end of the time slot. Based on that, it computes timea and timeb which denote the end and the beginning respectively of the two reservations.

We will now present the correctness of the algorithm in Figure 4 and Figure 5. We will also prove a bound on the number of preemptions. In addition, we present equations for the case where $\delta \to \infty$.

Theorem 1. *If tasks are assigned according to the algorithm in Figure 3 and it declares SUCCESS and the algorithm in Figure 4 dispatches tasks at run-time then all deadlines are met.*

Proof. It follows from the schedulability analysis in Section 2.5 and Section 2.6. □

Let us now state a bound on the number of preemptions.

Theorem 2. *Assume that tasks are assigned according to the algorithm in Figure 3 and it declares SUCCESS and the algorithm in Figure 4 dispatches tasks at run-time. Let* npreempt*(t,p) denote an upper bound on the number of preemptions generated on processor p in a time interval of duration t. Let $\tau_{hi,p}$ denote the task that is split between processors p-1 and p. Analogously, let $\tau_{lo,p}$ denote the task that is split between processors p+1 and p. Let S denote the size of the time slots. Let* $\mathrm{jobs}(t,p)$ *denote the maximum number of jobs that are released during a time interval of duration t from either (i) non-split tasks assigned to processor p or (ii) the task split between processor p-1 and p or (iii) the task split between processor p+1 and p. We then have that:*

$$\begin{aligned}\mathrm{npreempt}(t,p) = & \quad \mathrm{jobs}(t,p) + 2 + \\ & \min(\lceil \tfrac{t}{S} \rceil, \mathrm{activeslots}(t,p)) \cdot 3\end{aligned} \tag{15}$$

where

$$\begin{aligned}\mathrm{activeslots}(t,p) = & \left\lceil \frac{t}{T_{hi,p}} \right\rceil \cdot \left\lceil \frac{\min(D_{hi,p}, T_{hi,p})}{S} \right\rceil + \\ & \left\lceil \frac{t}{T_{lo,p}} \right\rceil \cdot \left\lceil \frac{\min(D_{lo,p}, T_{lo,p})}{S} \right\rceil\end{aligned} \tag{16}$$

[2] A solution is as follows. Let LB denote a lower bound on $z[p\text{-}1]$ and let UB denote an upper bound on $z[p\text{-}1]$. We know that one choice is LB $=0$ and UB $=$ sumreserve so we start with that. Let `middle` denote (LB+UB)/2. We try $z[p\text{-}1]$:=middle and $x[p]$:=sumreserve-middle and apply the schedulability test. If it succeeds then we set LB := middle otherwise UB := middle. We repeat this procedure until LB and UB are close enough (10 iterations tend to be sufficiently good) and then compute $z[p\text{-}1]$:=LB and $x[p]$:=sumreserve-LB.

```
1.  when booting
2.    if another processor q has already performed do
        ”‘when booting”’ then
3.      tstart[p] := tstart[q]
4.    else
5.      tstart[p] := any value
6.    end if
7.    lo_active[p] := false
8.    hi_active[p] := false
9.  end when booting

10. when a job released from a task split between processor
      processor p-1 and p arrives do
11.   hi_active[p] := true
12.   set_timer_again( p)
13.   dispatch(p)
14. end when

15. when a job released from a task split between processor
      processor p+1 and p arrives do
16.   lo_active[p] := true
17.   set_timer_again( p)
18.   dispatch(p)
19. end when

20. when a job released from a task split between processor
      processor p-1 and p finishes do
21.   hi_active[p] := false
22.   set_timer_again( p)
23.   dispatch(p)
24. end when

25. when a job released from a task split between processor
      processor p+1 and p finishes do
26.   lo_active[p] := false
27.   set_timer_again( p)
28.   dispatch(p)
29. end when

30. procedure set_timer_again( p : integer) is
31.   if lo_active[p]=true or hi_active[p]=true then
32.     t := read_time
33.     t0 := tstart + ⌊ (t-tstart[p])/S ⌋ · S
34.     t1 := tstart + ⌊ (t-tstart[p])/S ⌋ · S + S
35.     timea := t0 + x
36.     timeb := t1 - y
37.   end if
38. end procedure

39. procedure dispatch( p : integer) is
40.   selected := NULL
41.   t := read_time
42.   if hi_active[p] and t0 ≤ t < timea then
43.     selected := task that is split between processor p-1
          and processor p
44.   end if
45.   if lo_active[p] and timeb ≤ t < t1 then
46.     selected := task that is split between processor p
          and processor p+1
47.   end if
48.   if selected=NULL and there is a non-split task assigned
48.     to processor p that is ready then
49.     selected := the non-split task assigned to
50.       processor p with the earliest deadline
50.   end if
51. end procedure
```

Figure 4. An algorithm for dispatching tasks.

Proof. Note that a preemption can only occur when either (i) a job arrives or (ii) a reserve begins or ends. The number of preemptions due to job arrivals is clearly at most $\mathrm{jobs}(t,p)$. If we consider a time interval $[t,t+S)$ then it holds that there can be at most three preemptions caused by the beginning or ending of reserves. We can therefore subdivide a time interval of duration L into time intervals of duration S, each with at most three time preemptions caused by the beginning or ending of reserves. And then add two preemptions because of the beginning and ending of the time interval of duration L. This reasoning gives us the theorem. □

Consider now the problem of feasibility test of a task set, that is to decide if it is possible to schedule the task set. For such a use, we recommend the use of $\delta \to \infty$ with our algorithm. Let us introduce the following auxiliary variables:

$$xf[p] = \frac{x[p]}{S} \tag{17}$$

and

$$zf[p] = \frac{z[p]}{S} \tag{18}$$

Applying $\delta \to \infty$, Equation 17, Equation 18 and Equation 6 on Equation 3, Equation 11, Equation 13 and Equation 14 yields that:

$$xf[p] + zf[p-1] = \frac{C_i}{\min(D_i, T_i)} \tag{19}$$

and

$$\begin{aligned} g(L) = \sum_{\tau_j \in \tau^p} \max(0, \left\lfloor \frac{L - D_j}{T_j} \right\rfloor + 1) \cdot C_j + \\ \min(L, \quad \left\lfloor \frac{L}{T_{hi,p}} \right\rfloor \cdot C_{hi,p} \cdot \frac{xf[p]}{xf[p] + zf[p-1]} + \\ \min(L - \left\lfloor \frac{L}{T_{hi,p}} \right\rfloor \cdot T_{hi,p}, \min(D_{hi,p}, T_{hi,p})) \cdot xf[p] + \\ \left\lfloor \frac{L}{T_{lo,p}} \right\rfloor \cdot C_{lo,p} \cdot \frac{zf[p]}{xf[p+1] + zf[p]} + \\ \min(L - \left\lfloor \frac{L}{T_{lo,p}} \right\rfloor \cdot T_{lo,p}, \min(D_{lo,p}, T_{lo,p})) \cdot zf[p] \quad) \end{aligned} \tag{20}$$

and

$$\begin{aligned} LG_{lim} = \\ \frac{(\sum_{\tau_j \in \tau^p} C_j) + T_{hi,p} + T_{lo,p}}{1 - ((\sum_{\tau_j \in \tau^p} \frac{C_j}{T_j}) + \frac{C_{hi,p}}{T_{hi,p}} \cdot \frac{x[p]}{x[p]+z[p-1]} + \frac{C_{lo,p}}{T_{lo,p}} \cdot \frac{z[p]}{x[p+1]+z[p]})} \end{aligned} \tag{21}$$

and

```
1.  for each processor p do
2.    xf[p] := 0
3.    zf[p] := 0
4.    τ[p] := ∅
6.  end for
7.  for each processor p with index p < m do
8.   shared_task[p] := NULL
9.  end for
10. p := 1
11. unassigned := τ
12. while unassigned ≠ ∅ do
13.   for each task τ_i ∈ unassigned, considered in descending
       order of D_i do
14.     τ[p] := τ[p] ∪ { τ_i }
15.     if the test Equation 22 succeeds for processor p
          and xf[p]+zf[p]≤1 then
16.       unassigned := unassigned \ { τ_i }
17.     else τ[p] := τ[p] \ { τ_i } end if
20.   end for
21.   if unassigned ≠ ∅ then
22.     if p ≤ m-1 then
23.       i := index of the task in unassigned with least D_i
24.       p := p + 1
25.       assign values to xf[p] and zf[p-1] such that zf[p-1] is
           maximized and the test Equation 22 succeeds for
           processor p-1 and xf[p-1]+zf[p-1] ≤ 1 and
           Equation 19 is true for p
26.       shared_task[p-1] := τ_i
27.       unassigned := unassigned \ { τ_i }
28.     else declare FAILURE end if
29.   end if
30. end while
31. declare SUCCESS
```

Figure 5. A feasibility test

$$\begin{aligned} &\text{if} \\ &\quad \text{the denominator of the right} - \text{hand side of} \\ &\quad \text{Equation 21 is positive} \\ &\quad \text{and} \\ &\quad \forall L, L = k \cdot T_i + D_i, \\ &\quad L < \min(2 \cdot \text{lcm}(T_1, T_2, \ldots, T_n), \max(\text{DMAX}, \text{LG}_{lim})) : \\ &\quad g(L) \leq L \\ &\quad \text{then all deadlines are met} \end{aligned} \tag{22}$$

Based on these equations we have a feasibility test as expressed by Figure 5.

4 Feasibility testing

A sporadic task system is said to be *feasible* upon a specified platform if every possible sequence of jobs that can be legally generated by the task system can be scheduled upon the platform to meet all deadlines. Exact (necessary and sufficient) tests are known for determining feasibility for sporadic task systems upon preemptive uniprocessors [9]. For global scheduling upon preemptive multiprocessors, however, no exact (or particularly good sufficient) feasibility tests are known; obtaining such feasibility tests is currently one of the most important open problems in multiprocessor real-time scheduling theory. In particular, it would be useful to have feasibility tests that both perform well in practice (as determined by extensive

simulations) and are able to provide concrete quantitative performance guarantees. The kind of quantitative performance guarantee we will look to provide is the processor speedup factor.

Definition 1 (Processor speedup factor)**.** *A feasibility test is said to have a processor speedup factor s if any task system deemed to not be feasible upon a particular platform by the test is guaranteed to actually not be feasible upon a platform in which each processor is $1/s$ times as fast.*

A sufficient feasibility test that may perform arbitrarily poorly has a processor speedup factor of infinity, while an exact feasibility test has a processor speedup factor of 1. Thus, the processor speedup factor of a sufficient feasibility test may be considered to be a quantitative measure of the amount by which the test is removed from being exact — the smaller the processor speedup factor, the closer to being an exact test and hence the better the test.

As stated in Section 1, current sufficient feasibility tests for sporadic task systems adopt the approach of performing schedulability analysis of a specific scheduling algorithm (since any schedulable task system is trivially feasible). Baker [6] has experimentally evaluated the schedulability tests for a large number of scheduling algorithms, and observed that partitioning-based schedulability tests tend to perform better than global ones. Our approach in this section is to take a partitioning algorithm that is able to make quantitative performance guarantees — the one proposed in [7] — and obtain a sufficient global feasibility test, which we call FEAS-SS, by extending it to incorporate task splitting. We will show that FEAS-SS is strictly superior to the test in [7] (in the sense that it is able to demonstrate feasibility for all task systems determined to be schedulable by the test of [7] while there are task systems determined to be feasible by FEAS-SS that the test of [7] will not determine to be schedulable) — hence, FEAS-SS tends to perform better in practice than the schedulability test of [7]. At the same time, since FEAS-SS dominates the schedulability test of [7] it is able to make the same quantitative performance guarantee as the test in [7].

The quantitative performance guarantee made by the schedulability test in [7] is as follows:

Theorem 3 (From [7])**.** *If an arbitrary sporadic task system is feasible on m identical processors each of a particular computing capacity, then the scheduling algorithm in [7] successfully partitions this system upon a platform comprised of m processors that are each $(4 - \frac{2}{m})$ times as fast.*

The scheduling algorithm in [7] has a run-time computational complexity of $O(n^2)$ where n is the number of tasks; Theorem 3 therefore yields a polynomial-time (an $O(n^2)$) sufficient feasibility test.

The partitioning algorithm of [7] considers tasks in non-decreasing order of relative deadline parameter, attempting to accommodate each task onto any processor upon which it "fits" according to a polynomial-time testable condition (see [7] for details). The algorithm succeeds if all tasks are assigned to processors, and returns failure if it fails to assign some task.

FEAS-SS extends the partitioning algorithm of [7] by not returning failure if some task is not assigned to a processor; instead, it attempts to split this task among two processors, using the test of Equation 12 to identify processors upon which to split. Since we are interested in *feasibility analysis*, the number of preemptions in the resulting schedule is not of significance to us, and we assign the parameter δ the value infinity. (It may be verified that this reduces the overhead – the size of the reserves over the minimum needed to accommodate the split tasks – to zero.) If FEAS-SS is able to successfully split this task, it proceeds with the next task, attempting to split this among two processors as well (ensuring that no processor gets assigned more than two split tasks), and so on until either all the remaining tasks are split in this manner or the test of Equation 12 reveals that some task cannot be split and assigned to processors.

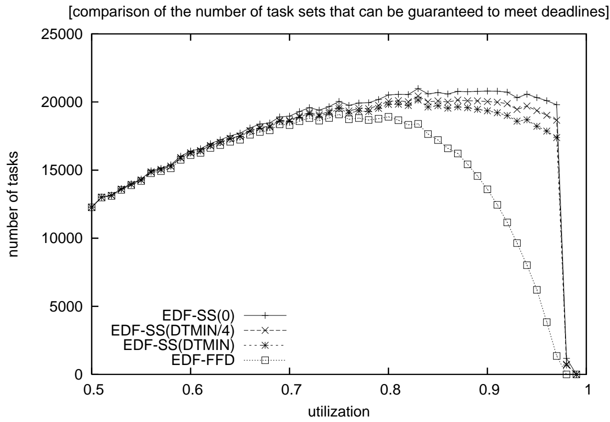
Evaluation. Baker has shown [6] that feasibility tests based on schedulability tests for partitioned scheduling tend to perform the best in practice; since FEAS-SS dominates the partitioning algorithm of [7], we expect it to perform at least as well as the algorithm of [7]. Furthermore, since FEAS-SS deems feasible all task systems that are successfully scheduled by the partitioning algorithm of [7], the following result trivially follows from Theorem 3:

Theorem 4. *The feasibility test FEAS-SS has a processor speedup factor $\leq (4 - \frac{2}{m})$ upon m-processor platforms.*

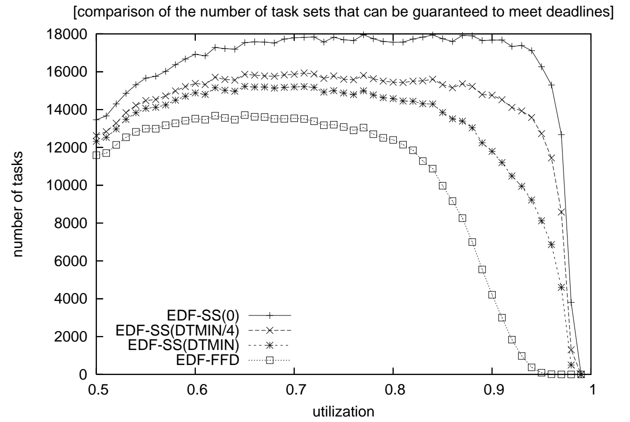
We have thus shown that FEAS-SS both performs well in practice, and is able to offer a non-trivial quantitative performance guarantee.

5 Evaluating the run-time algorithm

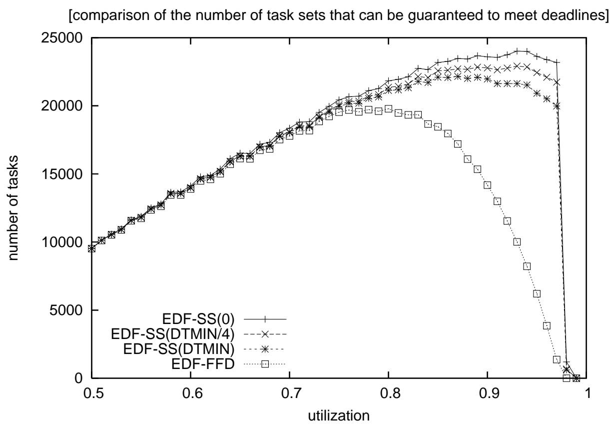
We saw above that the task-splitting approach yields a superior feasibility test, FEAS-SS. We now experimentally evaluate the performance of EDF-SS(DTMIN/δ) when used for scheduling systems. We claim that for the problem of scheduling



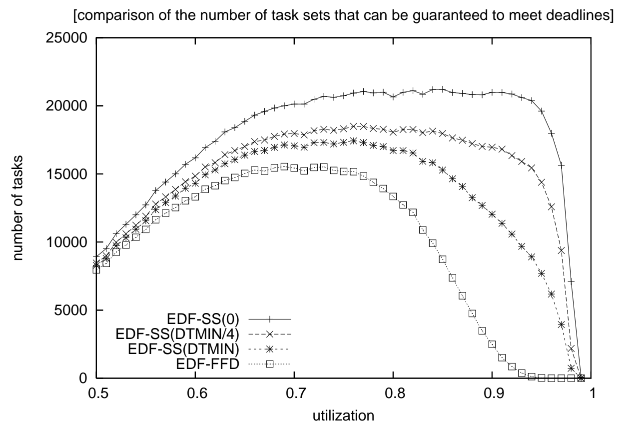
(a) bimodal; m=2



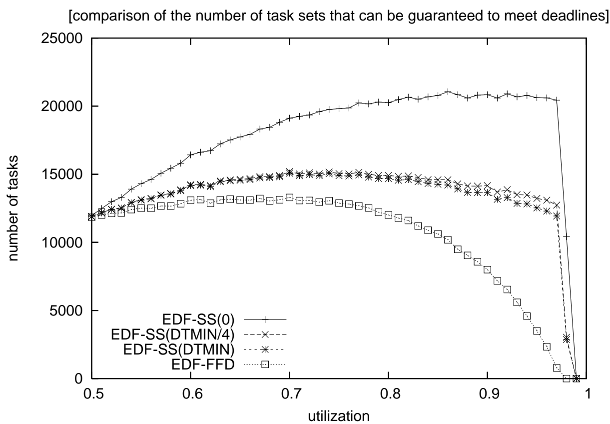
(b) bimodal; m=8



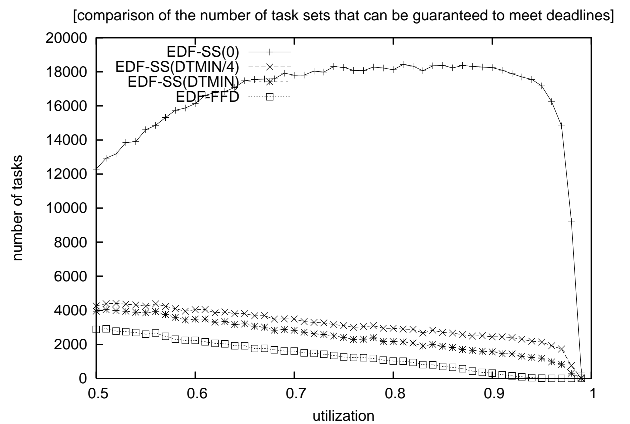
(c) uniform; m=2



(d) uniform; m=8



(e) exponential; m=2



(f) exponential; m=8

Figure 6. Results from simulation experiments. Two-processor and eight-processor systems are simulated, with task systems drawn from the bimodal, uniform, and exponential distributions (see the appendix for further detail.) Only the results for high-utilization systems ($\frac{1}{m}\sum\frac{C_i}{T_i} \geq 0.5$) are depicted. It is evident from these graphs that EDF-SS(DTMIN/δ) outperforms EDF-FFD for all the considered values of δ.

arbitrary-deadline sporadic tasks on a multiprocessor with pre-run-time guarantees, the new algorithm, EDF-SS(DTMIN/δ) offers a significant improvement versus the best previously known algorithm, EDF-FFD [6]. We substantiate this claim through a set of simulation experiments based on randomly generated task sets. The task sets were generated by a task generator, provided to us by Baker [3], which follows the spirit of the task generator in [6] but it is slightly different; details of the task-generation process are provided in the appendix.

Due to space limitations, only a few highlights of the simulation results are provided here; extensive description of the experimental results are available in [4, Appendix C].

Figure 6 shows, for arbitrary-deadline tasks, the number of tasks that could be given pre-run-time guarantees for our new algorithms and for EDF-FFD. It can be seen that all EDF-SS(x) algorithms outperform the best bin-packing algorithms. This figure focuses on task systems with utilization exceeding 50%. For task sets with utilization less than 50%, most task sets that were generated could be scheduled by all the algorithms. (Plots for task systems with other utilization ranges may be found in [4, Appendix C]).

6 Conclusions

We have explored the use of the *task splitting* approach for scheduling arbitrary-deadline sporadic task systems upon multiprocessor platforms. We have presented an algorithm for scheduling arbitrary-deadline sporadic tasks on a multiprocessor. The algorithm is configurable with a parameter δ, and can be used in two ways. For actual use in scheduling a system, choosing a small δ is recommended since this results in fewer preemptions. Our algorithm can also be used for feasibility testing and for this purpose the choice $\delta \rightarrow \infty$ is recommended.

We have evaluated both potential uses of our algorithm. When used for actually scheduling task systems, our new algorithm was shown, via extensive simulation experiments, to outperform previously known scheduling algorithms. These simulations were conducted using the same task-generator that has previously been used by other researchers, thereby providing additional confidence in the validity of our results.

When used for feasibility analysis, we have shown that our approach yields a sufficient feasibility test, FEAS-SS, with a processor speedup factor < 4; this is the same as the processor speedup factor of one of the best previously-known sufficient feasibility tests. We have also shown that FEAS-SS strictly dominates this prior test — all systems deemed feasible by the prior test are also deemed feasible by FEAS-SS, whereas there are systems determined to be feasible by FEAS-SS that the prior one fails to determine is feasible.

References

[1] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204, 2004.

[2] J. H. Anderson, V. Bud, and U. C. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on real-time systems*, pages 199–208, 2005.

[3] B. Andersson and K. Bletsas. Sporadic multiprocessor scheduling with few preemptions. In *Proceedings of the 20th Euromicro Conference on real-time systems*, 2008.

[4] B. Andersson, K. Bletsas, and S. Baruah. Scheduling arbitrary-deadline sporadic tasks on multiprocessors. Technical report, IPP-HURRAY Research Group. Institute Polytechnic Porto, HURRAY-TR-080501, available online at http://www.hurray.isep.ipp.pt/privfiles/HURRAY_TR_080501.pdf, September 2008.

[5] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *Proc. of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 322–334, 2006.

[6] T. P. Baker. Comparison of empirical success rates of global vs. partitioned fixed-priority EDF scheduling for hard real time. Technical Report TR-050601, Department of Computer Science, Florida State University, Tallahassee, available at http://www.cs.fsu.edu/research/reports/tr-050601.pdf, July 2005.

[7] S. Baruah and N. Fisher. The partitioned dynamic-priority scheduling of sporadic task systems. *Real-Time Systems: The International Journal of Time-Critical Computing*, 36(3):199–226, 2007.

[8] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, June 1996.

[9] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 182–190, 1990.

[3] We would like to acknowledge Professor Baker's assistance — he made his task-generator, implemented in Ada, available to us.

[10] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proc. of the 27th IEEE Real-Time Systems Symposium*, pages 101–110, 2006.

[11] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. In *Proc. of the 26th IEEE Real-Time Systems Symposium*, pages 330–341, 2005.

[12] S. Kato and N. Yamasaki. Real-time scheduling with task splitting on multiprocessors. In *Proceedings of the 13th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 441–450, 2007.

[13] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for the Computing Machinery*, 20:46–61, 1973.

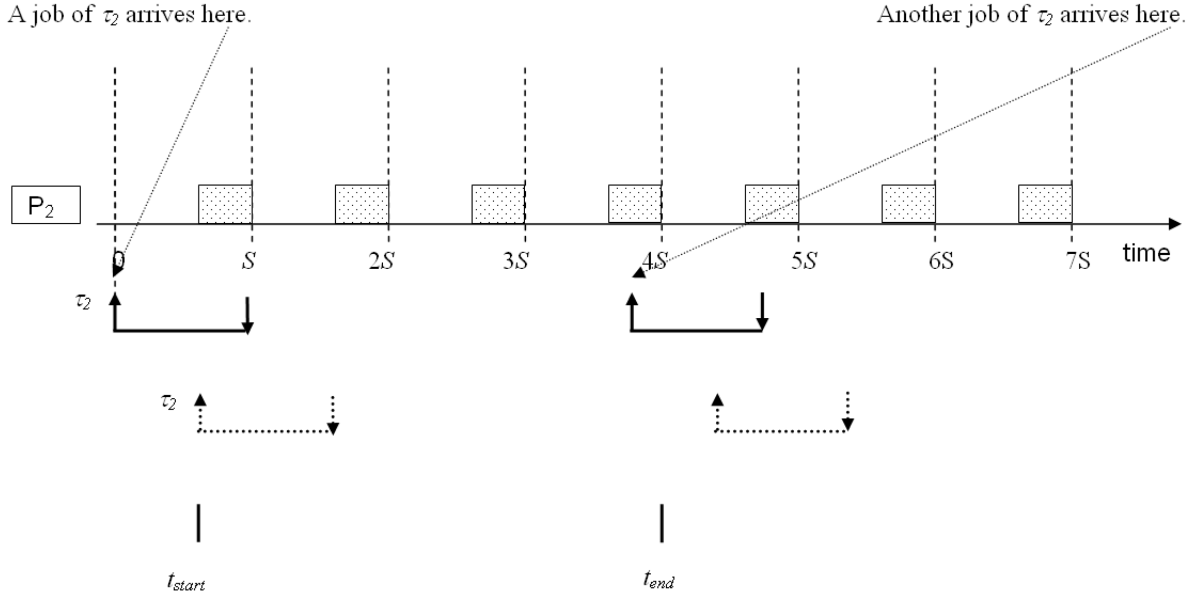


Figure 7. An example showing that it is challenging to find an upper bound on the amount of execution in reserves.

Appendix A: Execution by Split tasks

We are interested in finding an upper bound on the amount of execution that the split tasks perform on processor p during a time interval of duration L. It is easy to see that if the jobs during this time interval execute for less than its maximum execution time then we can obtain another scenario with no less execution of split tasks on processor p during L, by letting jobs execute by their maximum amount. (For split task τ_i with $D_i \leq T_i$ it is easy to see this because increasing the execution time of a job does not affect the scheduling of any other jobs of split task τ_i. For the case of a task τ_i with $D_i > T_i$ it may not be so easy to see because one may think that two jobs of a task may be active simultaneously and then increasing the execution time of the former job may change the starting time of execution of the latter job. This cannot happen though because of our choice expressed in Equation 3). For this reason, we will now assume that all jobs from split tasks execute by their maximum execution time. Let us define

$$\begin{aligned} \text{slotexec}(t, r) = & \lfloor \frac{t}{S} \rfloor \cdot r + \\ & \min(t - \lfloor \frac{t}{S} \rfloor \cdot S, r) \end{aligned} \tag{23}$$

Slotexec gives us an upper bound on the amount of execution performed by a task in a time interval of duration t, assuming that it executes in all of its reserves and assuming that each reserve assigned to that task has the duration r. This bound can be used to find an upper bound on the amount of time that split tasks execute on processor p. Many reserves are not used or not fully used though and for this reason, we need a bound that takes the parameters of the split tasks into account. We will derive such a bound now.

Let $\tau_{hi,p}$ denote the task that is split between processor p and processor p-1. Let us consider a time interval $[t_{start}, t_{end})$ of duration L. Our goal is to find an upper bound on the amount of execution by $\tau_{hi,p}$ during $[t_{start}, t_{end})$.

It is not obvious how to find the amount of execution. One may think that a split task arriving at time t_{start} would maximize the amount of execution of the split task during $[t_{start}, t_{end})$ because this is the case in many normal uniprocessor

scheduling problems, such as normal static-priority scheduling on a single processor. Consider Figure 7 however. It shows a task τ_2 and a processor P. To simplify the discussion, it shows only one reserve of the processor. The arrival times are shown as lines with arrows pointing upwards and the deadlines are shown as lines with arrows pointing downwards. It can be seen that two jobs of τ_2 can execute within the time interval shown in Figure 7. However if we would consider the case where the task τ_2 arrives at time t_{start} then only one of its job would execute in the time interval $[t_{start}, t_{end})$. Hence, care must be taken when finding an upper bound on the amount of execution time. In particular, we can observe that if a task arrives S-r time units before the considered time interval starts then it that task does not execute during those first S-r time units and before of this "'early arrival"' it is possible for the task to arrive again and hence it can execute more in the time interval considered.

We will find an upper bound on the amount of execution by $\tau_{hi,p}$ during $[t_{start}, t_{end})$.

Let us consider two cases.

1. $L \leq T_{hi,p}$

   Using Equation 23 we obtain that the amount of execution performed by $\tau_{hi,p}$ during this time interval is at most:

   $$\text{slotexec}(L, x[p]) \tag{24}$$

   another upper bound is:

   $$\text{slotexec}(\lfloor \frac{\min(D_{hi,p}, T_{hi,p})}{S} \rfloor \cdot S, x[p]) \tag{25}$$

   For the case $D_{hi,p} \leq T_{hi,p}$ it is obvious that Equation 25 holds. For the case $D_{hi,p} > T_{hi,p}$ may not be so obvious to see it. Its correctness follows though from the fact that we assign reserves according to Equation 3 and hence a job of $\tau_{hi,p}$ cannot finish more than $T_{hi,p}$ time units after its arrival even if $D_{hi,p} > T_{hi,p}$.

   Note that slotexec (defined in Equation 23) is non-decreasing with increasing t. So clearly we have an upper bound:

   $$\text{slotexec}(\min(L, \lfloor \frac{\min(D_{hi,p}, T_{hi,p})}{S} \rfloor \cdot S), x[p]) \tag{26}$$

2. $L > T_{hi,p}$

   We will now do a transformation of the jobs released by $\tau_{hi,p}$; the transformation has the following steps.

   (a) Let J denote the job with the earliest arrival time among the tasks that were released by $\tau_{hi,p}$ such that the arrival time is no earlier than t_{start}.

   (b) If no such J (as defined in (a)) exist and no job of $\tau_{hi,p}$ was released during $[t_{start}\text{-}T_{hi,p}, t_{start})$ then let $\tau_{hi,p}$ release a job at the time t_{start}. The job released will be referred to as J; it satisfies the definition of J in 1.

   (c) If no such J (as defined in (a)) exists and a job Q of $\tau_{hi,p}$ was released during $[t_{start}\text{-}T_{hi,p}, t_{start})$ then let $\tau_{hi,p}$ release a job at the the time $T_{hi,p}$ time units after the arrival of Q. The job released will be referred to as J; it satisfies the definition of J in 1.

   (d) Let S denote the set of jobs such that for each job Q in S it holds that (i) Q is released by $\tau_{hi,p}$ and (ii) the arrival time of Q is greater than the arrival time of J. For each job Q in S do

      i. Let P denote the job with the maximum arrival time among the jobs that satisfies (i) P is in S and (ii) the arrival time of P is less than the arrival time of Q.

      ii. Set the arrival time of Q to be equal to the arrival time of P plus $T_{hi,p}$.

   (e) Of all jobs from $\tau_{hi,p}$ that arrive no earlier than t_{start}, let K denote the job with the latest arrival.

   (f) Let AK denote the arrival time of job K.

(g) If $AK+T_{hi,p}<t_{end}$ then add new job arrivals of task $\tau_{hi,p}$ at time $AK+T_{hi,p}$, $AK+2\cdot T_{hi,p}$, $AK+3\cdot T_{hi,p}, \ldots$

This transformation (with step (a)-(g)) causes the amount of execution of split tasks in $[t_{start}, t_{end})$ to increase or stay unchanged. Let $\text{tbodystart}(\tau_{hi,p}, [t_{start}, t_{end}))$ denote the earliest time in $[t_{start}, t_{end})$ such that $\tau_{hi,p}$ arrives at that time. Let us consider two cases:

(a) There is no instant in $[t_{start}, \text{tbodystart}(\tau_{hi,p}, [t_{start}, t_{end})))$ such that $\tau_{hi,p}$ executes on processor p at that moment.

Then let us decrease the arrival time of all jobs from $\tau_{hi,p}$ by $\text{tbodystart}(\tau_{hi,p}, [t_{start}, t_{end}))$ - t_{start}. We end up with $\text{tbodystart}(\tau_{hi,p}, [t_{start}, t_{end})) = t_{start}$.

Note that the time interval of duration L can be subdivided into two time intervals, ΔV_1 and ΔV_2. One time interval, ΔV_1, starts at time t_{start} and ends

$$\lfloor \frac{L}{T_{hi,p}} \rfloor \cdot T_{hi,p} \tag{27}$$

time units later. Using Equation 27 and Equation 4 we obtain that the amount of execution performed by $\tau_{hi,p}$ during this time interval, ΔV_1, is at most:

$$\lfloor \frac{L}{T_{hi,p}} \rfloor \cdot C_{hi,p} \cdot \frac{x[p]}{x[p] + z[p-1]} \tag{28}$$

The other time interval, ΔV_2, ends at t_{end} and starts when the former time interval, ΔV_1, ends. The duration of this time interval, ΔV_2, is:

$$L - \lfloor \frac{L}{T_{hi,p}} \rfloor \cdot T_{hi,p} \tag{29}$$

Using Equation 23 we obtain that the amount of execution performed by $\tau_{hi,p}$ during this time interval is at most:

$$\text{slotexec}(L - \lfloor \frac{L}{T_{hi,p}} \rfloor \cdot T_{hi,p}, x[p]) \tag{30}$$

another upper bound is:

$$\text{slotexec}(\lfloor \frac{\min(D_{hi,p}, T_{hi,p})}{S} \rfloor \cdot S, x[p]) \tag{31}$$

Note that slotexec (defined in Equation 23) is non-decreasing with increasing t. So clearly we have an upper bound:

$$\text{slotexec}(\quad \min(L - \lfloor \frac{L}{T_{hi,p}} \rfloor \cdot T_{hi,p}, \lfloor \frac{\min(D_{hi,p}, T_{hi,p})}{S} \rfloor \cdot S) \quad , x[p] \quad) \tag{32}$$

This gives us that an upper bound on the amount of execution of $\tau_{hi,p}$ in a time interval of duration L is:

$$
\lfloor\frac{L}{T_{hi,p}}\rfloor \cdot C_{hi,p} \cdot \frac{x[p]}{x[p]+z[p-1]} +
$$
$$
\text{slotexec}(\quad \min(L - \lfloor\frac{L}{T_{hi,p}}\rfloor \cdot T_{hi,p},
$$
$$
\lfloor\frac{\min(D_{hi,p},T_{hi,p})}{S}\rfloor \cdot S) \quad , x[p] \quad)
$$

(33)

(b) There is an instant in [t_{start},tbodystart($\tau_{hi,p}$,[t_{start},t_{end}))) such that $\tau_{hi,p}$ executes on processor p at that moment.

Let us explore two subcases.

i. tbodystart($\tau_{hi,p}$,[t_{start},t_{end})) - $T_{hi,p} < t_{start}$-S
Then we can increase all arrival times of jobs released from $\tau_{hi,p}$ by S. This increases the amount of execution of split tasks on processor or it is unchanged. Repeat this argument as long as Case 2bi is true. Eventually, we end up in Case 2bii.

ii. tbodystart($\tau_{hi,p}$,[t_{start},t_{end})) - $T_{hi,p} \geq t_{start}$-S
Let us consider two further subcases:

A. There is no moment in [tbodystart($\tau_{hi,p}$,[t_{start},t_{end})) - $T_{hi,p}$,t_{start}) such that $\tau_{hi,p}$ executes on processor p at that moment.
Since there is no execution by $\tau_{hi,p}$ in [tbodystart($\tau_{hi,p}$,[t_{start},t_{end})) - $T_{hi,p}$,t_{start}), we can compute the amount of execution in [tbodystart($\tau_{hi,p}$,[t_{start},t_{end})) - $T_{hi,p}$,t_{end}) instead of during [t_{start}, t_{end}). The difference in duration between these intervals is at most S-$x[p]$. So we can get the amount of execution by considering a time interval of duration L+S-$x[p]$ and applying the expression in Case 2a.

B. There is a moment in [tbodystart($\tau_{hi,p}$,[t_{start},t_{end})) - $T_{hi,p}$,t_{start}) such that $\tau_{hi,p}$ executes on processor p at that moment.
Let EX denote the latest moment of execution in [tbodystart($\tau_{hi,p}$,[t_{start},t_{end})) - $T_{hi,p}$,t_{start}). We can increase all arrival times of jobs released from $\tau_{hi,p}$ by EX-(tbodystart($\tau_{hi,p}$,[t_{start},t_{end})) - $T_{hi,p}$). This transformation increases execution by $\tau_{hi,p}$ on processor p in [t_{start},t_{end}) or it is unchanged. This takes us to Case 2biiA.

It can be seen that Case 2biiA gives the maximum execution of $\tau_{hi,p}$ on processor p.

Let $\tau_{lo,p}$ denote the task that is split between processor p and processor p+1. Doing similar reasoning for $\tau_{lo,p}$ and adding them gives us that the amount of execution of split tasks on processor p during a time interval of duration L is at most:

$$
\lfloor\frac{L+S-x[p]}{T_{hi,p}}\rfloor \cdot C_{hi,p} \cdot \frac{x[p]}{x[p]+z[p-1]} +
$$
$$
\text{slotexec}(\min(L+S-x[p] - \lfloor\frac{L+S-x[p]}{T_{hi,p}}\rfloor \cdot T_{hi,p},
$$
$$
\lfloor\frac{\min(D_{hi,p},T_{hi,p})}{S}\rfloor \cdot S), x[p]) +
$$
$$
\lfloor\frac{L+S-z[p]}{T_{lo,p}}\rfloor \cdot C_{lo,p} \cdot \frac{z[p]}{x[p+1]+z[p]} +
$$
$$
\text{slotexec}(\min(L+S-z[p] - \lfloor\frac{L+S-z[p]}{T_{lo,p}}\rfloor \cdot T_{lo,p},
$$
$$
\lfloor\frac{\min(D_{lo,p},T_{lo,p})}{S}\rfloor \cdot S), z[p])
$$

(34)

Appendix B: Proof of Faster Schedulability Test

We will now prove that the speedup technique is safe. That is, it does not cause a deadline miss. Recall the definition of f. It is:

$$\begin{aligned} f(L) = & \sum_{\tau_j \in \tau^p} \max(0, \lfloor \frac{L - D_j}{T_j} \rfloor + 1) \cdot C_j + \\ \min(L, \quad & \lfloor \frac{L + S - x[p]}{T_{hi,p}} \rfloor \cdot C_{hi,p} \cdot \frac{x[p]}{x[p] + z[p-1]} + \\ & \text{slotexec}(\min(L + S - x[p] - \lfloor \frac{L + S - x[p]}{T_{hi,p}} \rfloor \cdot T_{hi,p}, \\ & \lfloor \frac{\min(D_{hi,p}, T_{hi,p})}{S} \rfloor \cdot S), x[p]) + \\ & \lfloor \frac{L + S - z[p]}{T_{lo,p}} \rfloor \cdot C_{lo,p} \cdot \frac{z[p]}{x[p+1] + z[p]} + \\ & \text{slotexec}(\min(L + S - z[p] - \lfloor \frac{L + S - z[p]}{T_{lo,p}} \rfloor \cdot T_{lo,p}, \\ & \lfloor \frac{\min(D_{lo,p}, T_{lo,p})}{S} \rfloor \cdot S), z[p]) \quad) \end{aligned} \tag{35}$$

Let DMAX denote $\max(D_1, D_2, \ldots, D_n)$. Let us assume that $L \geq \text{DMAX}$ and $\sum_{\tau_j \in \tau^p} \frac{C_j}{T_j} \leq 1$. We can then reason as follows:

We obtain that an upper bound on f is:

$$\begin{aligned} & \sum_{\tau_j \in \tau^p} (\frac{L - D_j}{T_j} + 1) \cdot C_j + \\ S + L \cdot & \frac{C_{hi,p}}{T_{hi,p}} \cdot \frac{x[p]}{x[p] + z[p-1]} + T_{hi,p} + \\ S + L \cdot & \frac{C_{lo,p}}{T_{lo,p}} \cdot \frac{z[p]}{x[p+1] + z[p]} + T_{lo,p} \end{aligned} \tag{36}$$

We can rewrite the upper bound. We obtain that an upper bound on f is:

$$\begin{aligned} L \cdot (\quad & (\sum_{\tau_j \in \tau^p} \frac{C_j}{T_j}) + \frac{C_{hi,p}}{T_{hi,p}} \cdot \frac{x[p]}{x[p] + z[p-1]} + \\ & \frac{C_{lo,p}}{T_{lo,p}} \cdot \frac{z[p]}{x[p+1] + z[p]} \quad) + \\ (\sum_{\tau_j \in \tau^p} & (C_j - C_j \cdot \frac{D_j}{T_j})) + 2 \cdot S + T_{hi,p} + T_{lo,p} \end{aligned} \tag{37}$$

We can do a simple relaxation. We obtain that an upper bound on f is:

$$L \cdot \left((\sum_{\tau_j \in \tau^p} \frac{C_j}{T_j}) + \frac{C_{hi,p}}{T_{hi,p}} \cdot \frac{x[p]}{x[p]+z[p-1]} + \frac{C_{lo,p}}{T_{lo,p}} \cdot \frac{z[p]}{x[p+1]+z[p]} \right) + (\sum_{\tau_j \in \tau^p} C_j) + 2 \cdot S + T_{hi,p} + T_{lo,p} \tag{38}$$

We can now see how these upper bounds are useful.

Lemma 1. *Assume that*

$$1 - ((\sum_{\tau_j \in \tau^p} \frac{C_j}{T_j}) + \frac{C_{hi,p}}{T_{hi,p}} \cdot \frac{x[p]}{x[p]+z[p-1]}) > 0 \tag{39}$$

Let L_{lim} be defined as:

$$L_{lim} = \frac{(\sum_{\tau_j \in \tau^p} C_j) + 2 \cdot S + T_{hi,p} + T_{lo,p}}{1 - ((\sum_{\tau_j \in \tau^p} \frac{C_j}{T_j}) + \frac{C_{hi,p}}{T_{hi,p}} \cdot \frac{x[p]}{x[p]+z[p-1]} + \frac{C_{lo,p}}{T_{lo,p}} \cdot \frac{z[p]}{x[p+1]+z[p]})} \tag{40}$$

We claim that for all L such that $L_{lim} \leq L$ and $\mathrm{DMAX} \leq L$, it holds that:

$$f(L) \leq L \tag{41}$$

Proof. From Inequality 39 we obtain that $\sum_{\tau_j \in \tau^p} \frac{C_j}{T_j} \leq 1$. From the fact that $L_{lim} \leq L$ we obtain:

$$\frac{(\sum_{\tau_j \in \tau^p} C_j) + 2 \cdot S + T_{hi,p} + T_{lo,p}}{1 - ((\sum_{\tau_j \in \tau^p} \frac{C_j}{T_j}) + \frac{C_{hi,p}}{T_{hi,p}} \cdot \frac{x[p]}{x[p]+z[p-1]} + \frac{C_{lo,p}}{T_{lo,p}} \cdot \frac{z[p]}{x[p+1]+z[p]})} \leq L \tag{42}$$

Rewriting yields:

$$(\sum_{\tau_j \in \tau^p} C_j) + 2 \cdot S + T_{hi,p} + T_{lo,p} \leq L \cdot \left(1 - ((\sum_{\tau_j \in \tau^p} \frac{C_j}{T_j}) + \frac{C_{hi,p}}{T_{hi,p}} \cdot \frac{x[p]}{x[p]+z[p-1]} + \frac{C_{lo,p}}{T_{lo,p}} \cdot \frac{z[p]}{x[p+1]+z[p]}) \right) \tag{43}$$

Further rewriting yields:

$$
\begin{aligned}
& (\sum_{\tau_j \in \tau^p} C_j) + 2 \cdot S + T_{hi,p} + T_{lo,p} + \\
L \cdot (\quad & (\sum_{\tau_j \in \tau^p} \frac{C_j}{T_j}) + \frac{C_{hi,p}}{T_{hi,p}} \cdot \frac{x[p]}{x[p] + z[p-1]} + \\
& \frac{C_{lo,p}}{T_{lo,p}} \cdot \frac{z[p]}{x[p+1] + z[p]} \quad) \\
& \leq L
\end{aligned}
\tag{44}
$$

Using Equation 38 gives us that:

$$
f(L) \leq L \tag{45}
$$

which states the lemma.

□

Recall that

$$
\begin{aligned}
& \text{if} \\
& \forall L, L = k \cdot T_i + D_i, L \leq 2 \cdot lcm(T_1, T_2, \ldots, T_n) : \\
& \qquad f(L) \leq L \\
& \text{then all deadlines are met}
\end{aligned}
\tag{46}
$$

Combining this with Lemma 1 gives us that:

$$
\begin{aligned}
& \text{if} \\
& \quad \text{the denominator of the right} - \text{hand side of} \\
& \quad \text{Equation 13 is positive} \\
& \quad \text{and} \\
& \quad \forall L, L = k \cdot T_i + D_i, \\
& L < \min(2 \cdot lcm(T_1, T_2, \ldots, T_n), \max(\text{DMAX}, L_{lim})) : \\
& \qquad f(L) \leq L \\
& \text{then all deadlines are met}
\end{aligned}
\tag{47}
$$

Appendix C: Performance Evaluation

We essentially follow the setup used by Baker [6], and generously provided to us (with minor differences) by him. The values of T_i are given as a uniformly distributed variable with minimum 1 and maximum 1000. Experiments are performed for m=2 processors or m=8 processors.

Execution times may be drawn from either a bimodal, uniform, or exponential distribution.

- Execution times given according to the bimodal distribution are as follows. Each task is categorized as "heavy"' or "light"' with 33% probability for the former and 67% probability for the latter. A heavy task has a utilization as given by a uniformly distributed random variable in [0.5,1). A light task has a utilization as given by a uniformly distributed random variable in [0,0.5). The execution time of a task C_i is chosen as T_i multiplied by the utilization of the task.

- Execution times chosen according to a uniform distribution are as follows. A task has a utilization as given by a uniformly distributed random variable in [0,1). The execution time of a task C_i is given as T_i multiplied by the utilization of the task.

- Execution times given according to an exponential distribution are as follows. A task has a utilization as given by a exponential distributed random variable with mean 0.3. The execution time of a task C_i is given as T_i multiplied by the utilization of the task. The mean value of this distribution in the code given by Baker was set to zero, so we set it arbitrarily to 0.3.

Experiments are performed for implicit deadlines (in Baker's setup this is called "periodic"), for constrained deadlines, for arbitrary deadlines (in Baker's setup this is called "unconstrained") and for arbitrary deadlines where the deadline of a task is significantly larger than its T_i (in Baker's setup this is called "superperiod")

- Deadlines given according to implicit deadlines are simply set as D_i=T_i for each task.

- Deadlines given according to constrained deadlines are set as follows. The deadline of a task τ_i is C_i + (a uniformly distributed variable in [0,1)) multiplied by (T_i-C_i+1).

- Deadlines given according to unconstrained deadlines are set as follows. The deadline of a task τ_i is C_i + (a uniformly distributed variable in [0,1)) multiplied by ($4 \cdot T_i$-C_i+1).

- Deadlines given according to arbitrary deadlines, superperiod are set as follows. When computing the deadline of a task, generate a uniformly distributed random variable and multiply that by four and take the floor of that. Then multiply it by T_i. This is D_i.

The intent of Baker's task set generator is to generate task sets that are not obviously infeasible. For this reason, task sets with $(1/m) \cdot \sum_{i=1}^{n} C_i/T_i > 1$ are immediately rejected by the task set generator and task sets where at least one task has $C_i > D_i$ or $C_i > T_i$ are rejected as well.

The intent of Baker's task set generator is also that task sets should not be easily scheduled by algorithms for implicit-deadline tasks. For this reason, task sets with $(1/m) \cdot \sum_{i=1}^{n} \frac{C_i}{min(D_i,T_i)} \leq 1$ are rejected as well.

We run several experiments. For each experiment, we used Baker's task set generator to generate 1000000 task sets and these task sets were written to a file. Our own tool read all task sets and applied algorithms on each task set. The algorithms considered were the following. EDF-SS(0), EDF-SS(DTMIN/4), EDF-SS(DTMIN) and EDF-FFD. In the case of implicit deadline systems, we also included the Ehd2-SIP algorithm.

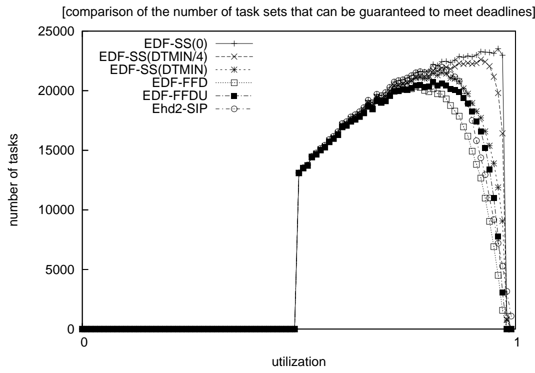
For each task set we calculated the utilization[4] and put it into a "bucket". There are 100 buckets; one for each percentage of utilization. For example, all task sets with utilization within [0.77,0.78) are put in one bucket. For each bucket we calculated the number of task sets that can be scheduled with each algorithm considered. These histograms are reported in subsequent subsections. Note that, some buckets have a large number of task sets generated and other buckets have a smaller number of task sets generated. Our simulation results should therefore not be used to asses how "success ratio" varies as certain parameters vary. Our simulation results can however be used to find out which algorithm performs best.

[4] The utilization of a task set is defined as $\frac{1}{m} \sum_{i=1}^{n} \frac{C_i}{T_i}$

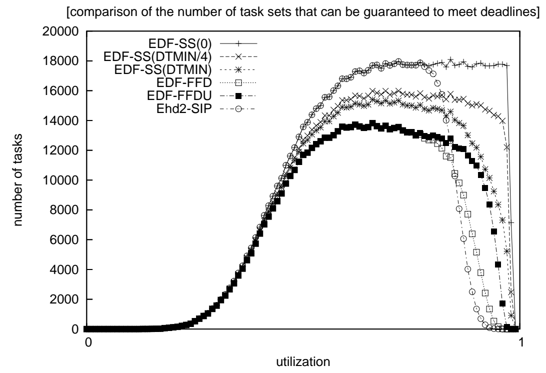
Experimental Results

The results from the bimodal distribution are shown in Figure 8-Figure 11. It can be seen that the new algorithm EDF-SS(x) performs significantly better than EDF-FFD.

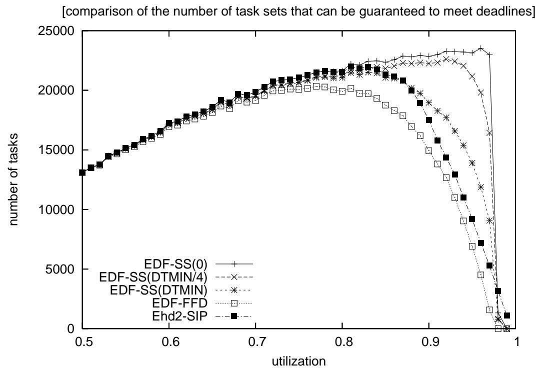
The results from the uniform distribution are shown in Figure 12-Figure 15. The results from the exponential distribution are shown in Figure 16-Figure 19. The conclusion is the same; the new algorithm EDF-SS(x) performs significantly better than EDF-FFD.



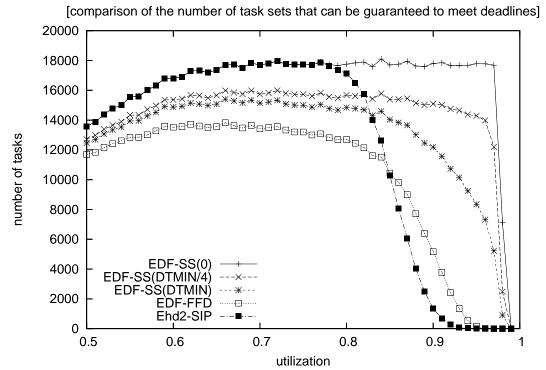
(a) m=2



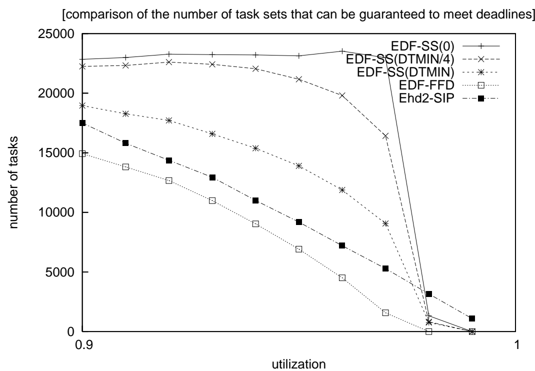
(b) m=8



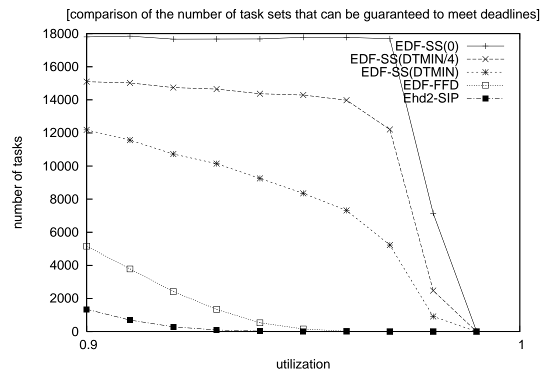
(c) m=2



(d) m=8

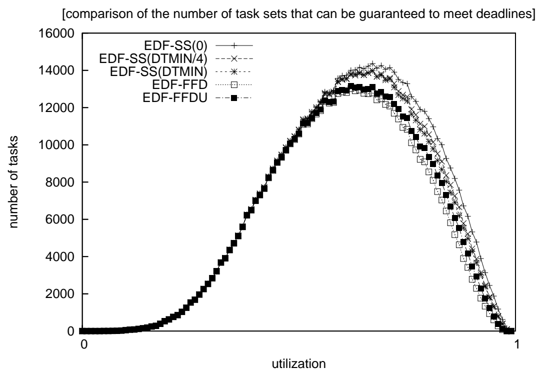


(e) m=2

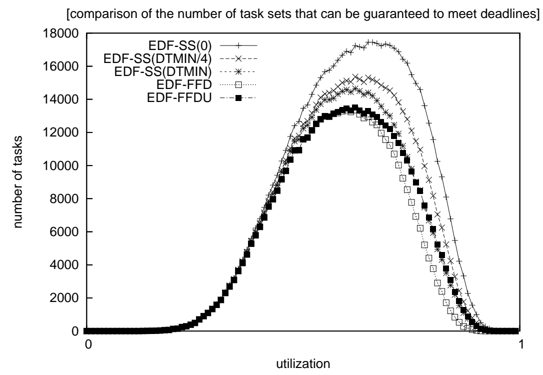


(f) m=8

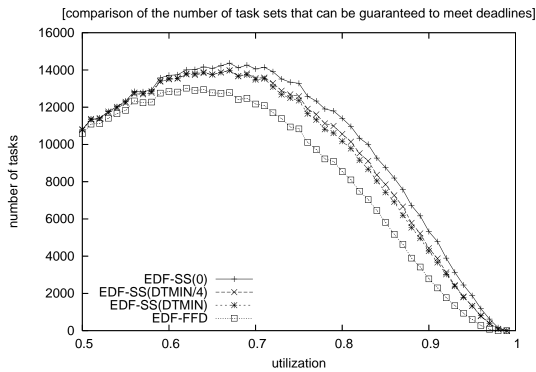
Figure 8. Results from simulation experiments with bimodal distribution and implicit deadlines. Figure (c)-(f) shows zoom-ed on figures.



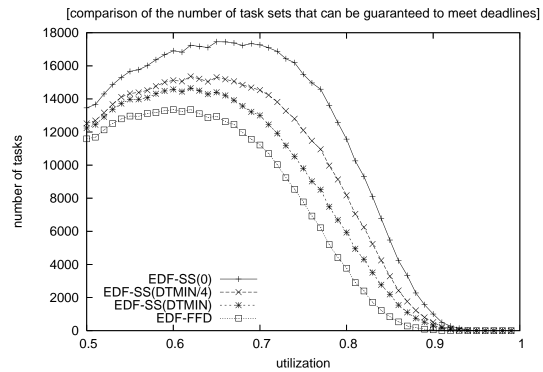
(a) m=2



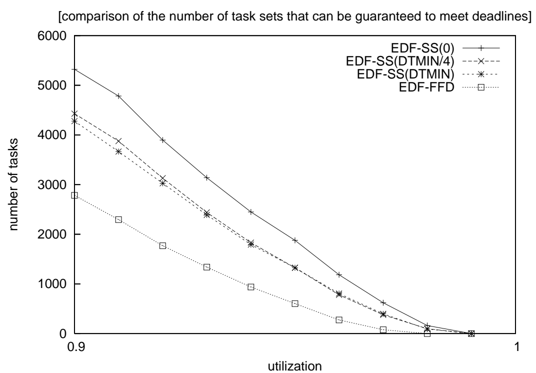
(b) m=8



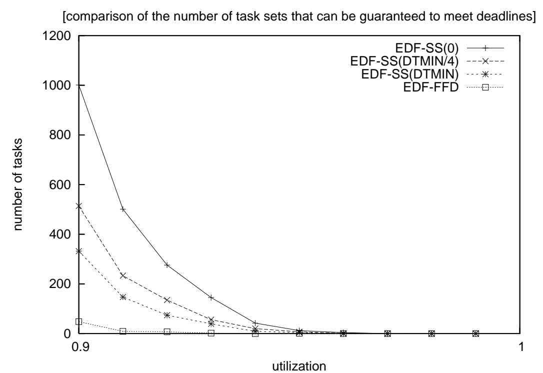
(c) m=2



(d) m=8

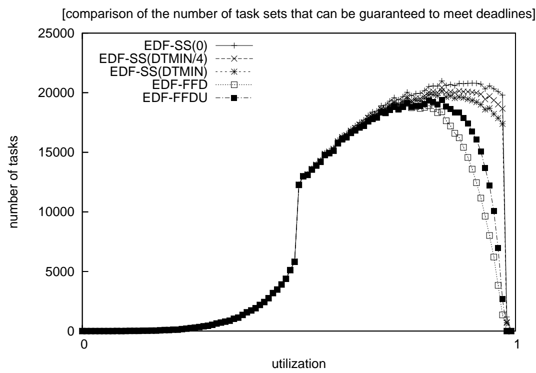


(e) m=2

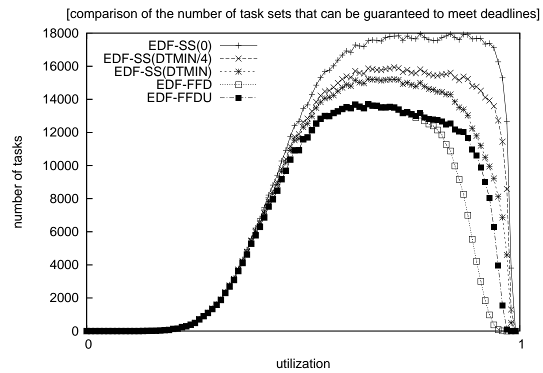


(f) m=8

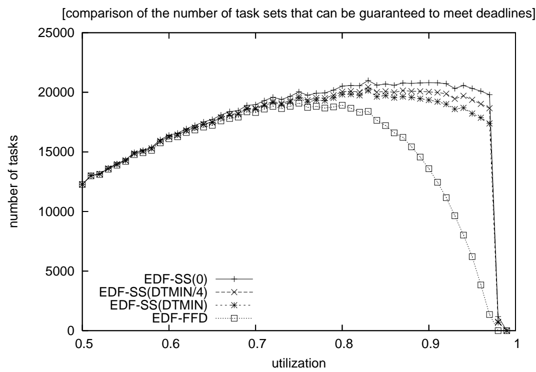
Figure 9. Results from simulation experiments with bimodal distribution and constrained deadlines. Figure (c)-(f) shows zoom-ed on figures.



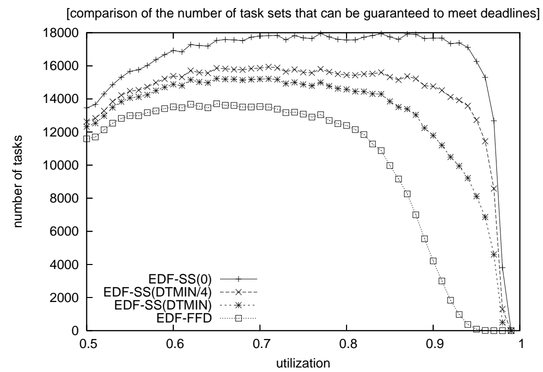
(a) m=2



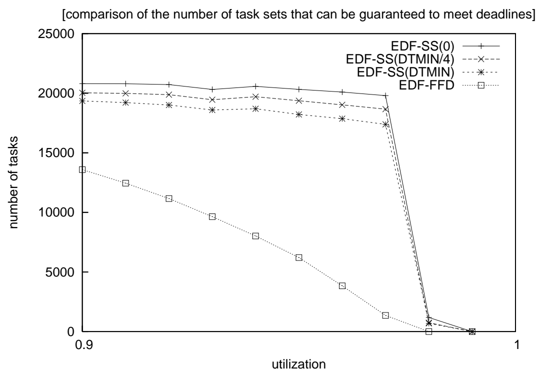
(b) m=8



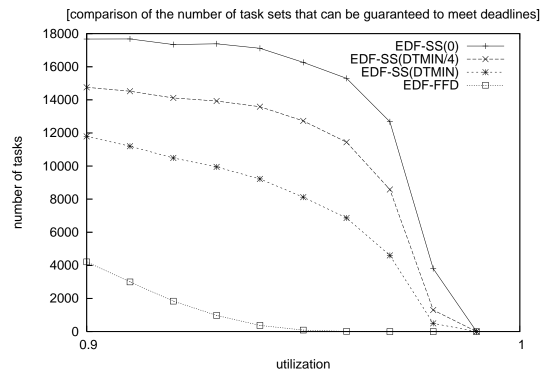
(c) m=2



(d) m=8

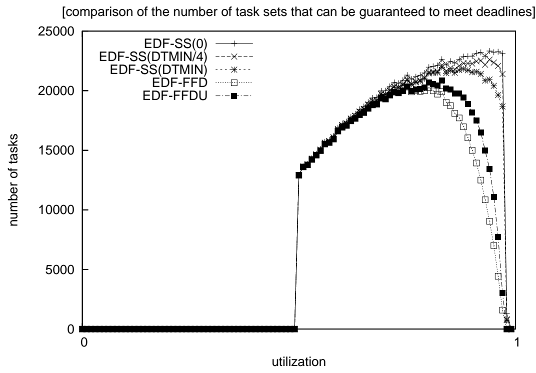


(e) m=2

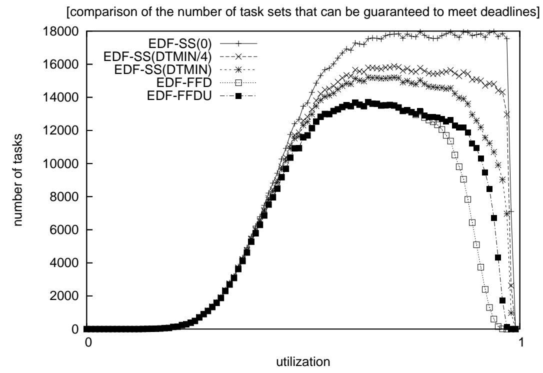


(f) m=8

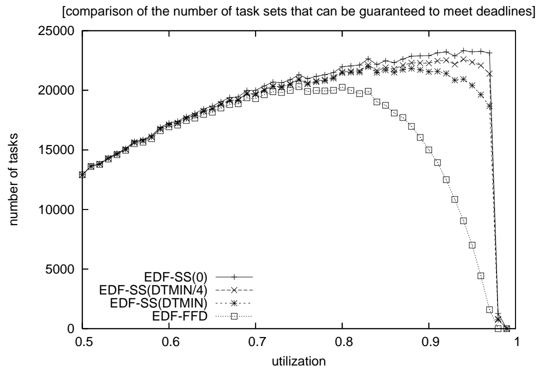
Figure 10. Results from simulation experiments with bimodal distribution and arbitrary deadlines. Figure (c)-(f) shows zoom-ed on figures.



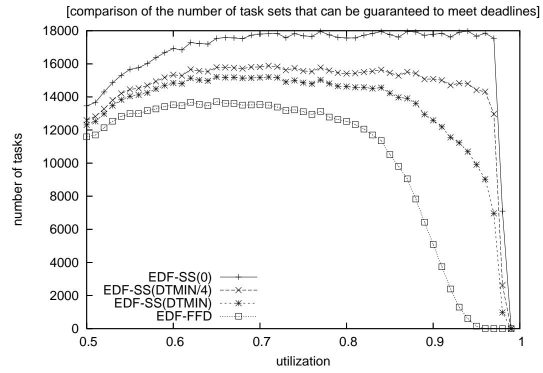
(a) m=2



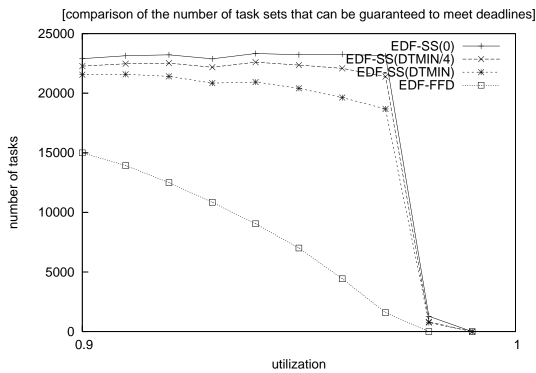
(b) m=8



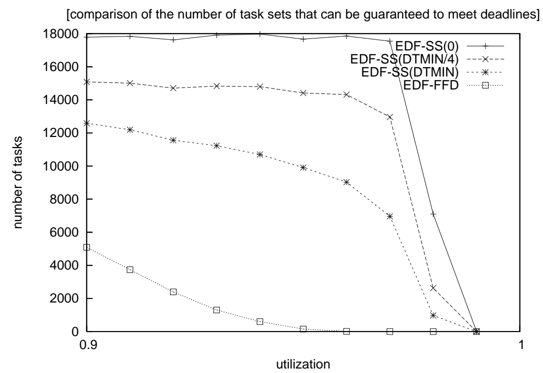
(c) m=2



(d) m=8

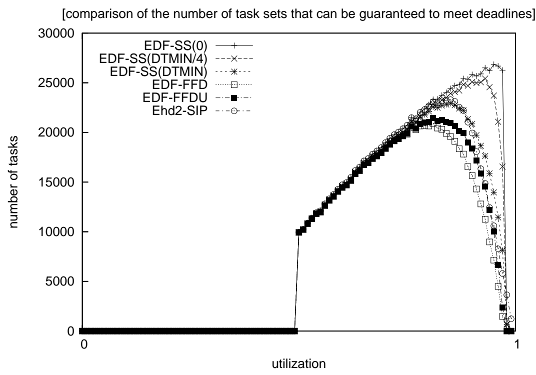


(e) m=2

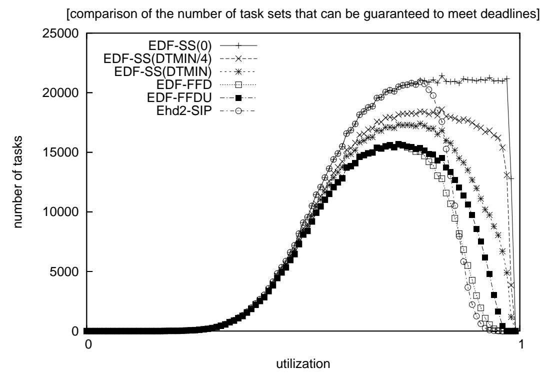


(f) m=8

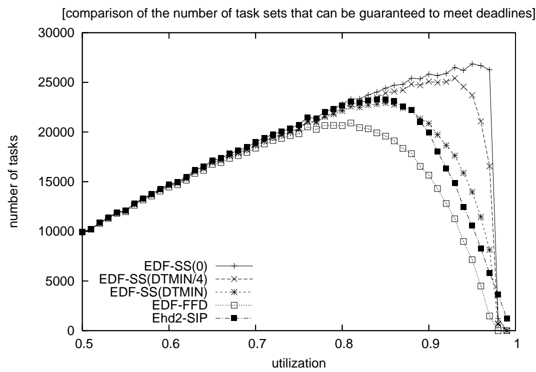
Figure 11. Results from simulation experiments with bimodal distribution and arbitrary deadlines (superperiod). Figure (c)-(f) shows zoom-ed on figures.



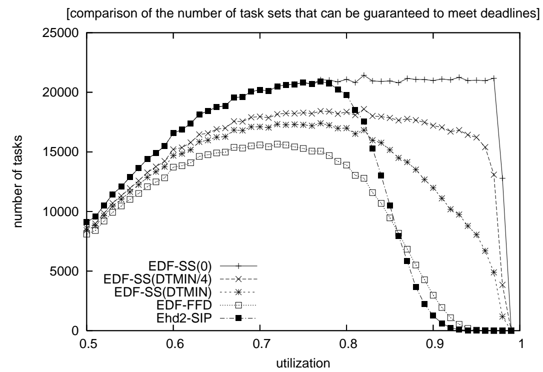
(a) m=2



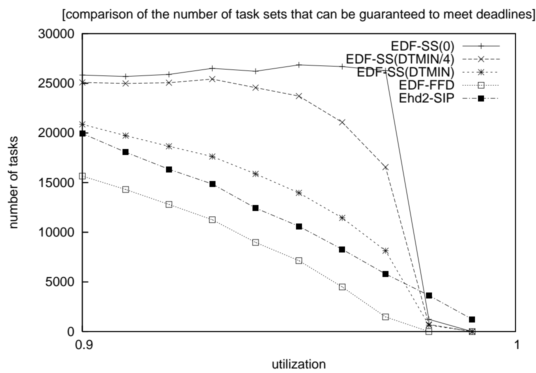
(b) m=8



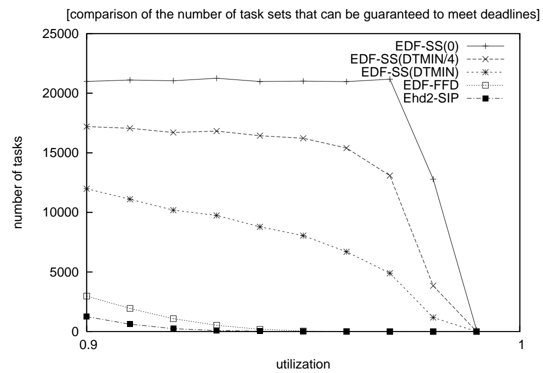
(c) m=2



(d) m=8

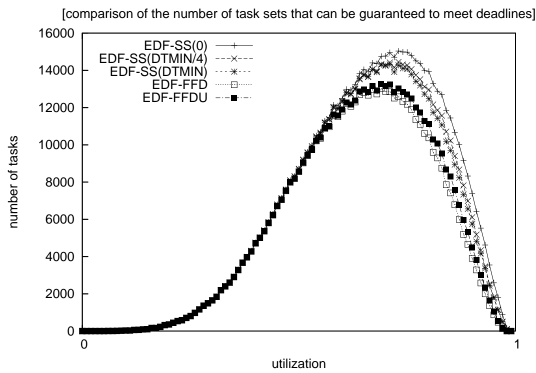


(e) m=2

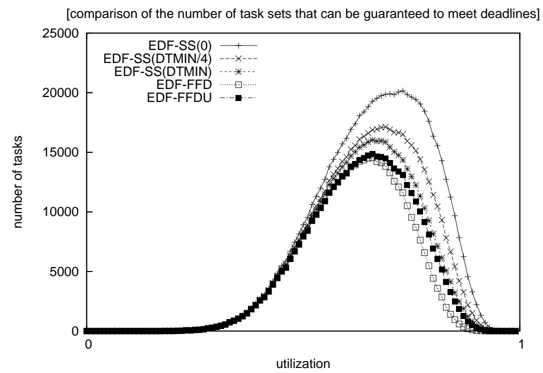


(f) m=8

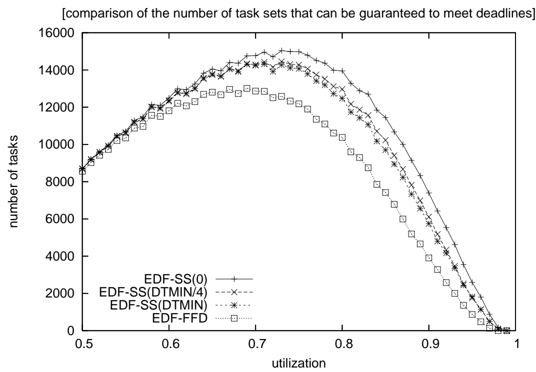
Figure 12. Results from simulation experiments with uniform distribution and implicit deadlines. Figure (c)-(f) shows zoom-ed on figures.



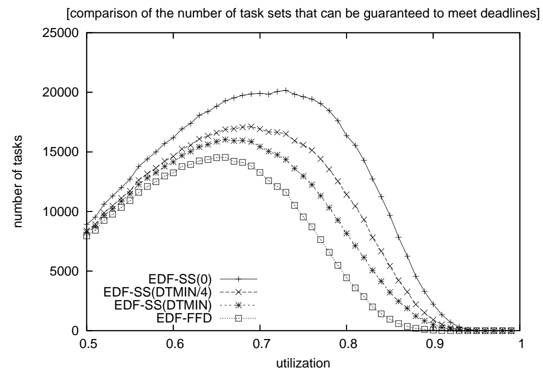
(a) m=2



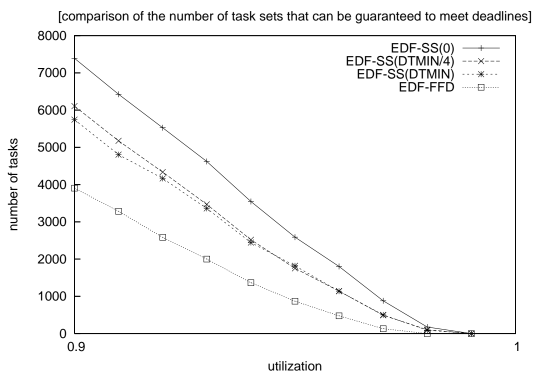
(b) m=8



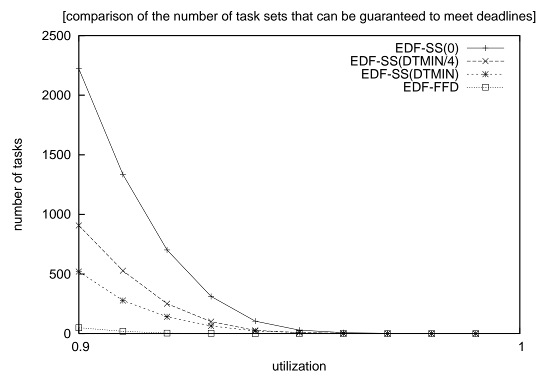
(c) m=2



(d) m=8

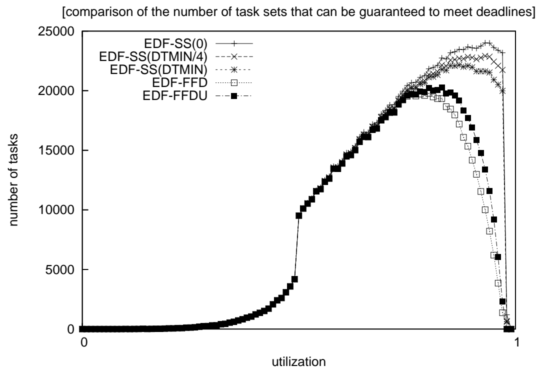


(e) m=2

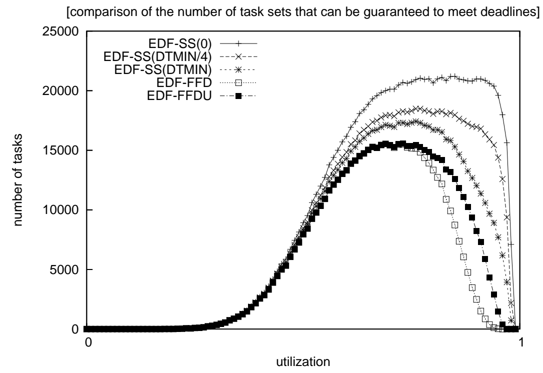


(f) m=8

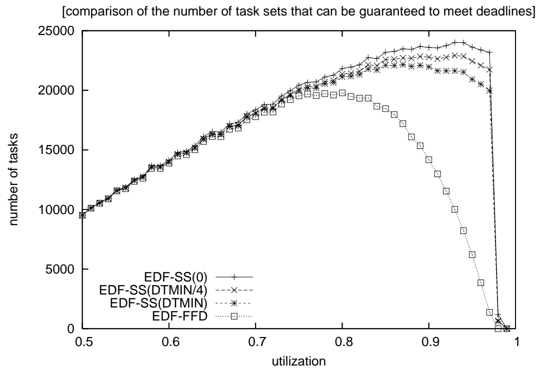
Figure 13. Results from simulation experiments with uniform distribution and constrained deadlines. Figure (c)-(f) shows zoom-ed on figures.



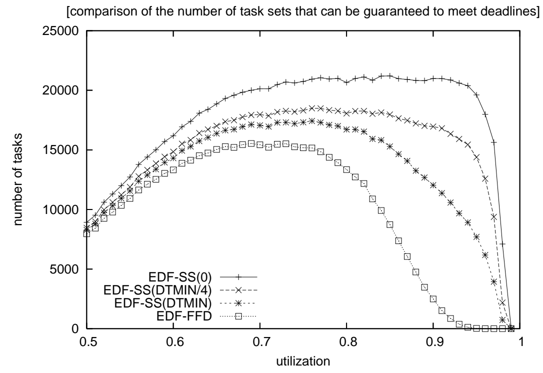
(a) m=2



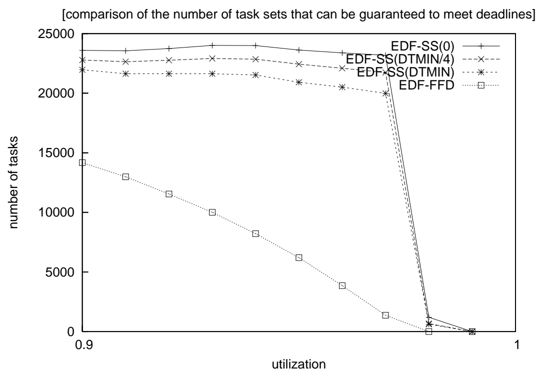
(b) m=8



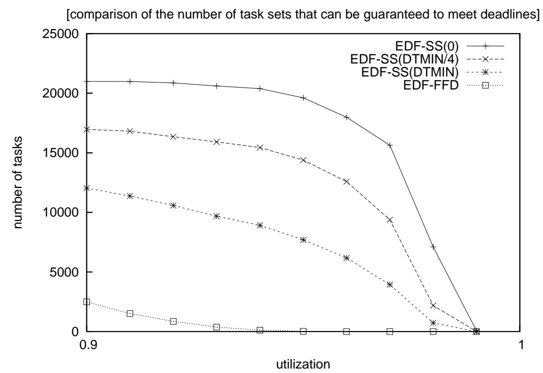
(c) m=2



(d) m=8

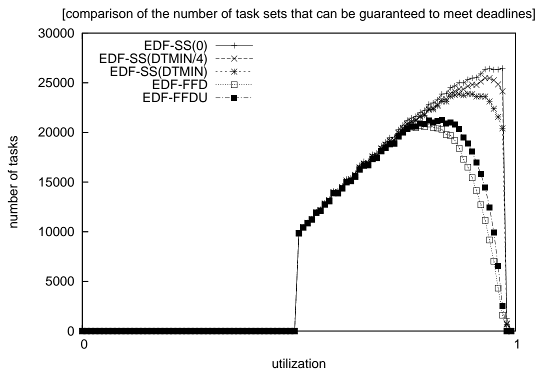


(e) m=2

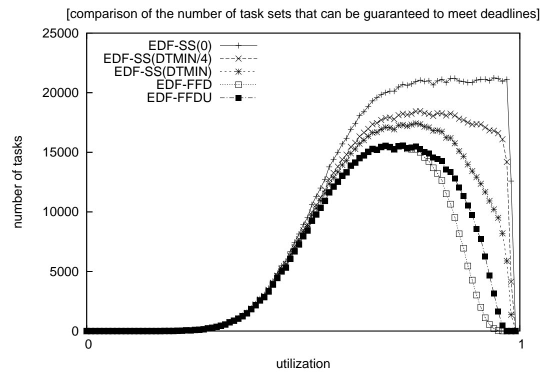


(f) m=8

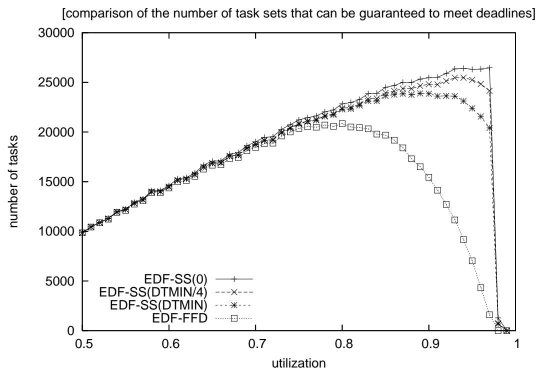
Figure 14. Results from simulation experiments with uniform distribution and arbitrary deadlines. Figure (c)-(f) shows zoom-ed on figures.



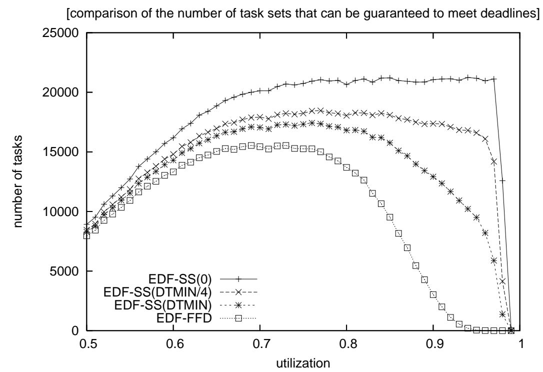
(a) m=2



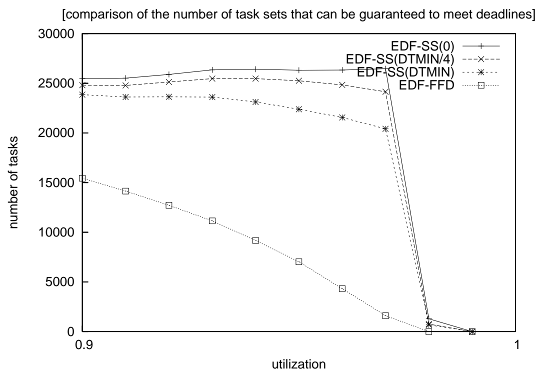
(b) m=8



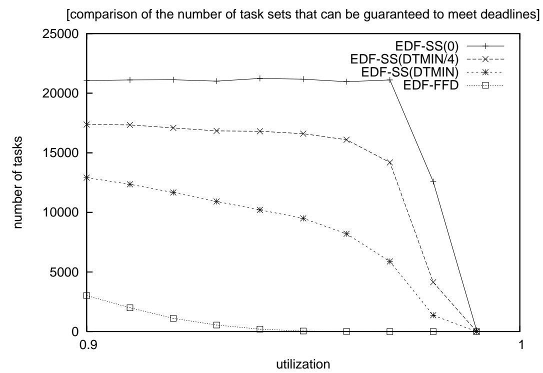
(c) m=2



(d) m=8

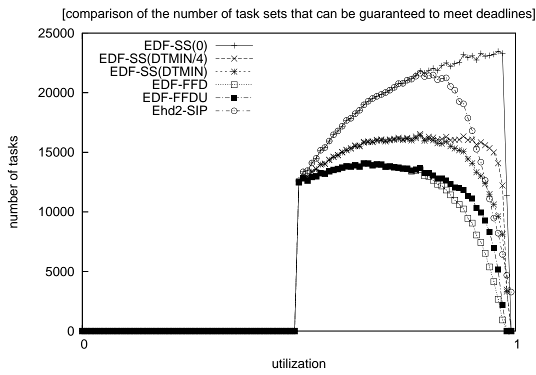


(e) m=2

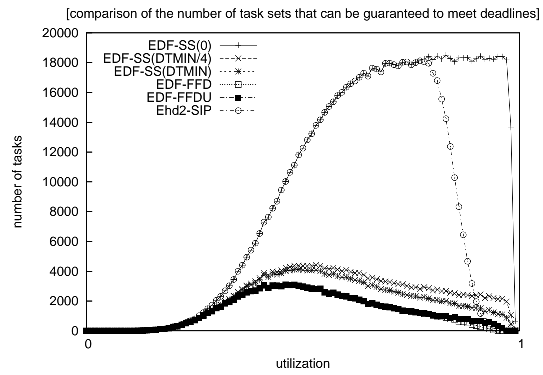


(f) m=8

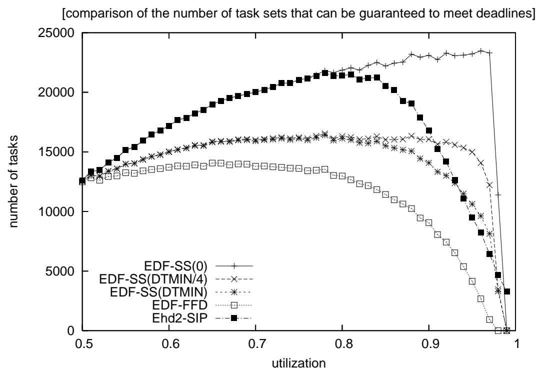
Figure 15. Results from simulation experiments with uniform distribution and arbitrary deadlines (superperiod). Figure (c)-(f) shows zoom-ed on figures.



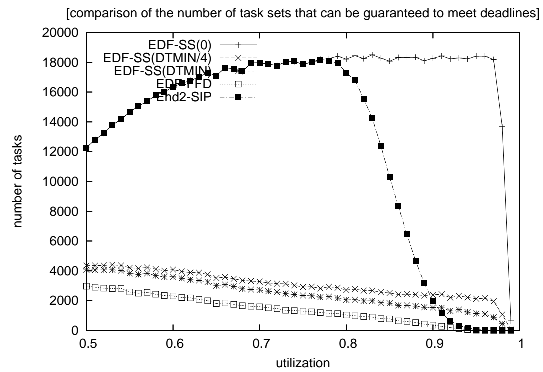
(a) m=2



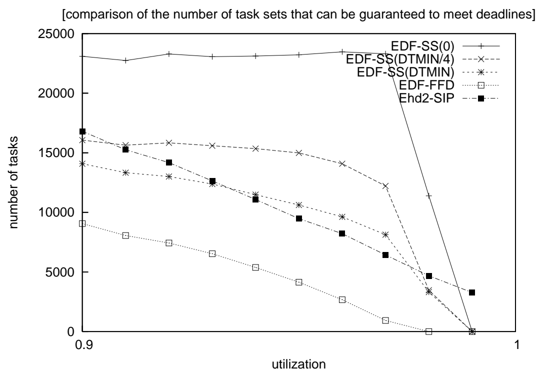
(b) m=8



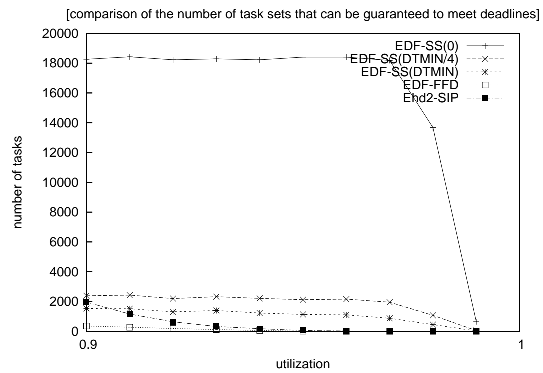
(c) m=2



(d) m=8

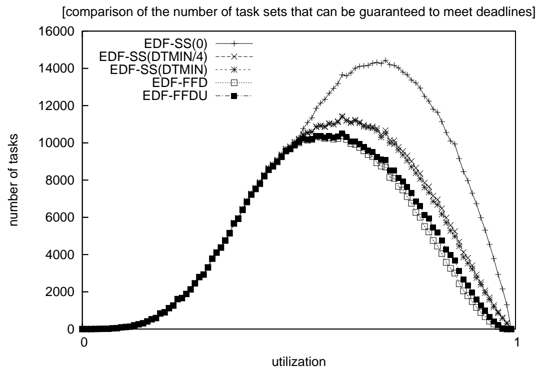


(e) m=2

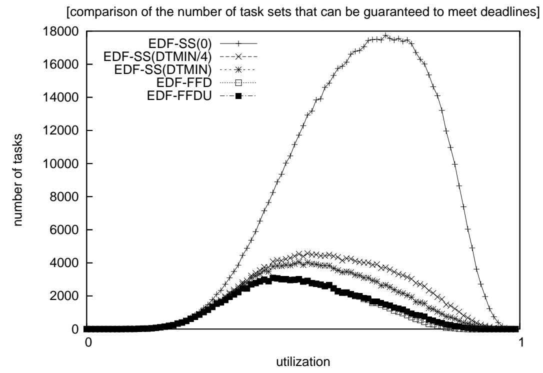


(f) m=8

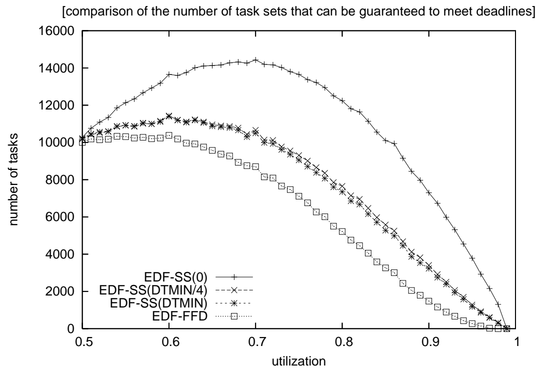
Figure 16. Results from simulation experiments with exponential distribution and implicit deadlines. Figure (c)-(f) shows zoom-ed on figures.



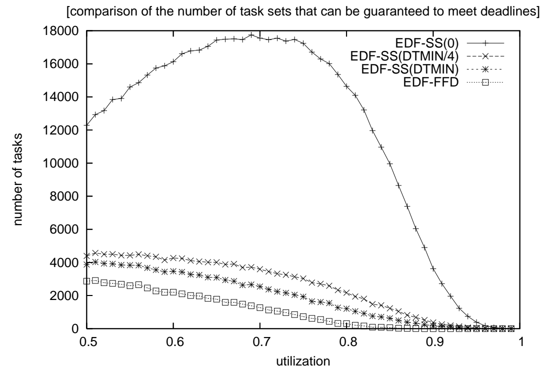
(a) m=2



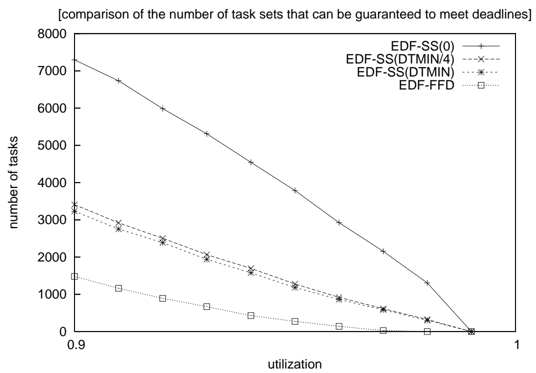
(b) m=8



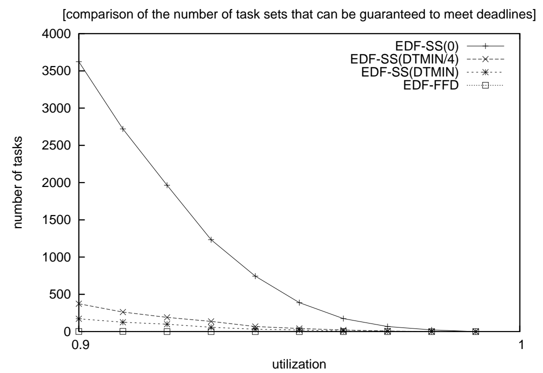
(c) m=2



(d) m=8

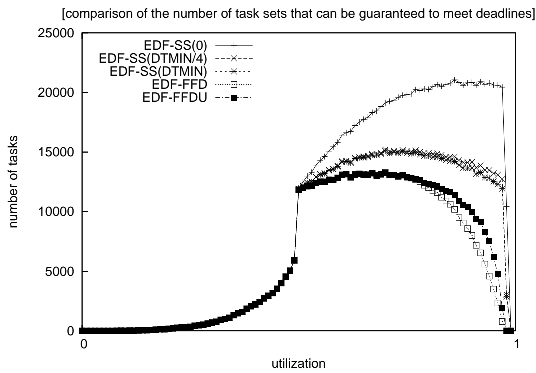


(e) m=2

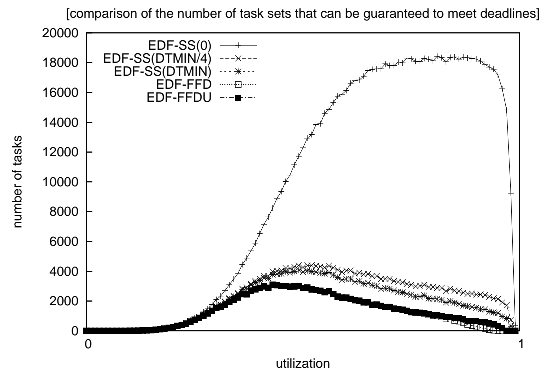


(f) m=8

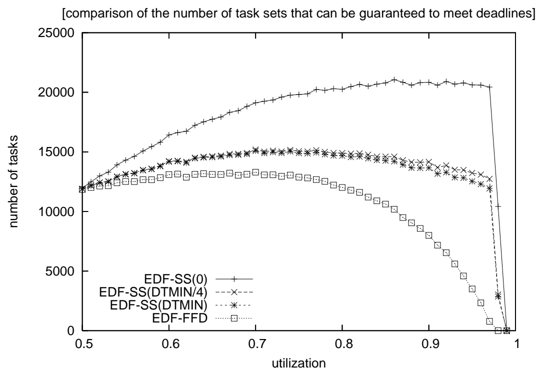
Figure 17. Results from simulation experiments with exponential distribution and constrained deadlines. Figure (c)-(f) shows zoom-ed on figures.



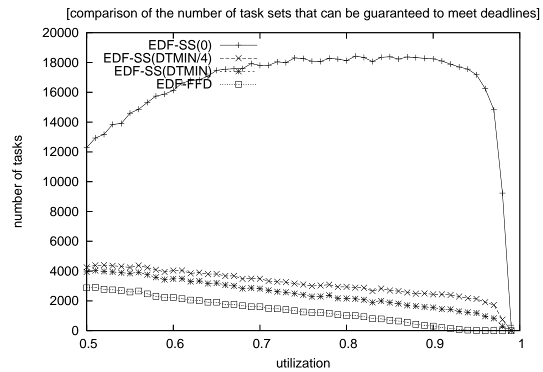
(a) m=2



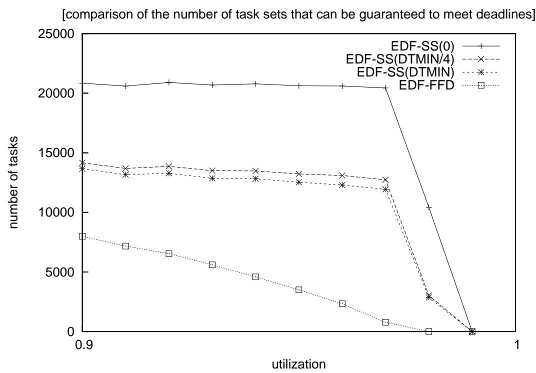
(b) m=8



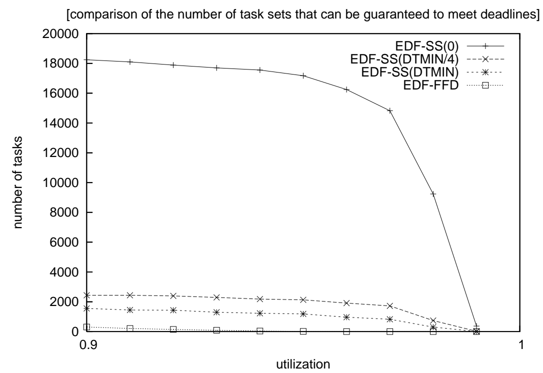
(c) m=2



(d) m=8

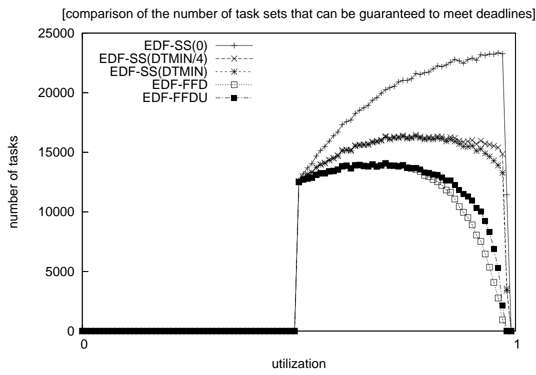


(e) m=2

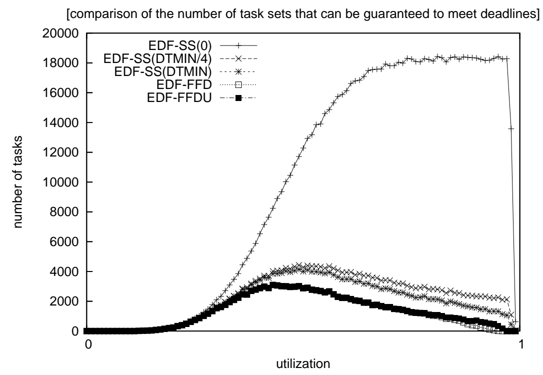


(f) m=8

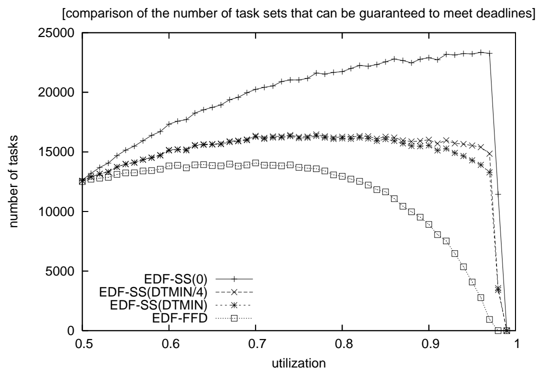
Figure 18. Results from simulation experiments with exponential distribution and arbitrary deadlines. Figure (c)-(f) shows zoom-ed on figures.



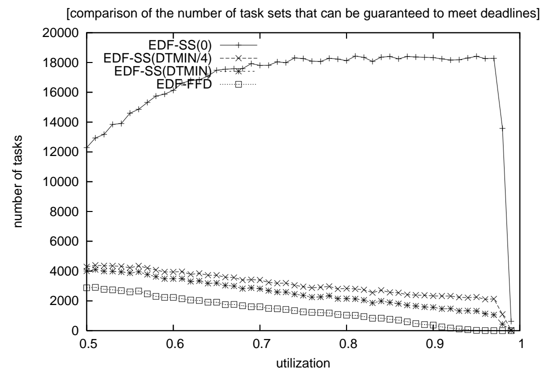
(a) m=2



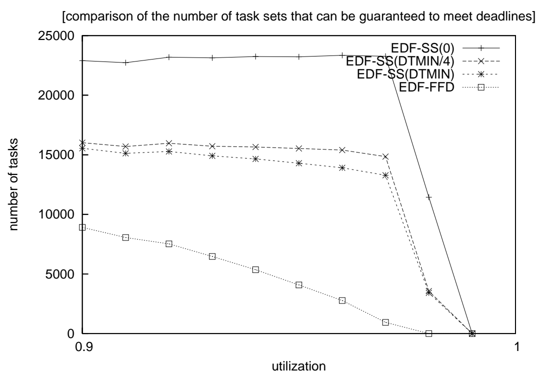
(b) m=8



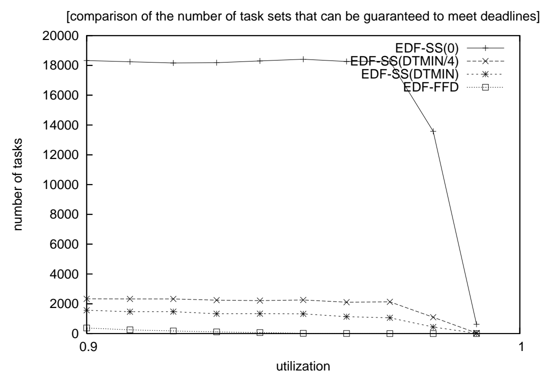
(c) m=2



(d) m=8



(e) m=2



(f) m=8

Figure 19. Results from simulation experiments with exponential distribution and arbitrary deadlines (superperiod). Figure (c)-(f) shows zoom-ed on figures.