Object Transportation System Using LEGO Mindstorms RCX

CSC 714: Fall 2011

Term project report

Vishnu Dasa Harish Babu - vdasaha

David Boyuka – daboyuka

http://www4.ncsu.edu/~vdasaha/rtcs.htm

Project Overview:

In this project we have constructed an object transportation system using the LEGO Mindstorms RCX. The objects (AA batteries) will be input to the system by hand, being placed into “input bins” which are located around the track. The aim is to deliver these objects to a common “output bin” with a given delivery guarantees. The input bins and output bin are collectively place at regular intervals around the track. The objects to be delivered are divided into “classes” based on their input bin. The number and order of input/output bins is known a priori.

The system features an RCX-powered vehicle capable of moving in either direction around this ring, picking up objects at input bins and storing them for transport, and dropping off all stored objects at the output bin. It incorporates sensors to perform these tasks effectively. We have used brickOS [1,2,3,5] as our software platform and used C as our coding language.

In order to meet the real-time constraints we impose, we use an algorithm based on non-preemptive EDF with TBS. The application of this algorithm to our system has some nuance, described subsequently. Additionally, we present two new techniques, “TBS WCET reduction” and “preemptive EDF emulation” to improve the performance and feasibility bounds of the basic algorithm.

Real-time Goals and Challenges:

We wish to make two basic real-time guarantees in our object delivery system. First, we will guarantee each input bin is checked according to a periodic task model (that is, checked exactly once per periodic time window), with each input bin potentially having a different period. Secondly, whenever an object is picked up, and if the system decides to accept it for guaranteed delivery, it is guaranteed to be delivered by a relative deadline specific to each input bin.

In order to model this, we consider the system to be a set of tasks, allowing us to apply various real-time techniques/algorithms to meet these guarantees. However, the tasks in our system are not entirely traditional, where the “execution time” is typically CPU usage. In our system, a task includes both the movement of the train itself as well as the time to perform the bin check/delivery function. Hence, here the WCET corresponds to the maximum possible transit time plus the time required for checking/dump at a bin.

We model input bin checking as periodic tasks, with a period corresponding to the guarantee period. We pick up any objects that may be present at the bin at check time. When an object is picked up, we then release a sporadic task for the delivery of the object.

There are complications, however. The following issues make it difficult for us to use standard real time techniques:

1. The time required to move to an input/output bin is variable is dependent on the RCX's current location. An overall worst-case bound on this time is very loose.
2. Modeling pickup and delivery as simple tasks can be very inefficient, as visiting bins in arbitrary order can lead to "excessive seeking" (i.e. wasted movement)

3. Our system is inherently non-preemptive. The task of checking a bin or delivering objects includes the transportation time and time for the atomic action of checking/dumping, none of which can be suspended or resumed later.

Our Solution:

As a solution to all the above problems, we have developed an algorithm based on non-preemptive EDF (Earliest Deadline First) along with TBS (Total Bandwidth Server) for handling sporadic tasks. Additionally, we also develop two algorithmic extensions that improve base performance. As stated before, we treat the checking of the input bins as periodic tasks, and an object delivery as a sporadic task that is released when an object is discovered at an input bin. There are three important points to this construction. First, the WCETs are based on maximum possible travel distance in the system (i.e. halfway around the track). Second, since all objects are delivered at once, we combine multiple object deliveries into a single sporadic task, which is added by the first object picked up. Lastly, when an object is picked up, it is accepted with either "guaranteed" or "best-effort" delivery. It is "guaranteed" when the TBS deadline is at most the required deadline and "best-effort" otherwise. In both cases, the task is added to the TBS and the object is delivered, but only "guaranteed" objects are guaranteed to meet their deadlines.

The first extension to this basic scheme, which we call "TBS WCET reduction", allows us to give better delivery guarantees for objects. We do this by calculating tighter WCET bounds for delivery on a case-by-case basis using the train's current location. The basic idea is to compute the WCET as if we were to go immediately to deliver the object from our current location. This can give a lower WCET than assuming maximum distance. Then, we compute the new TBS deadline given this WCET, and compare it to other job deadlines in the system. If other jobs might intervene and execute first, we find the worst-case distance from the output bin in which we could end up, and then iteratively recompute the WCET, TBS deadline and set of intervening jobs until it converges. The actual algorithm is as follows:

1. Let $dist_{current}$ be the distance from the current bin to the output bin, let t_{move} be the (worst-case) time to travel between adjacent bins, let t_{check} be the (worst-case) time to service a bin, and let D_{last} be the previous TBS deadline.
2. Tentative WCET as $WCET_{tent} = (dist_{current}) \cdot (t_{move}) + (t_{check})$
3. Tentative TBS deadline is calculated as $D_{tent} = \max(D_{last}, now) + \frac{WCET_{tent}}{U_s}$
4. Let J be the set of released jobs which have deadlines at most D_{tent}
5. Let $dist_{wc}$ be the largest distance from any bin corresponding to a job in J to the output bin.

6. If $dist_{wc} \leq dist_{current}$, terminate and use D_{tent} as the next TBS deadline.
7. Otherwise, let $dist_{current} := dist_{wc}$ and go to step 2.

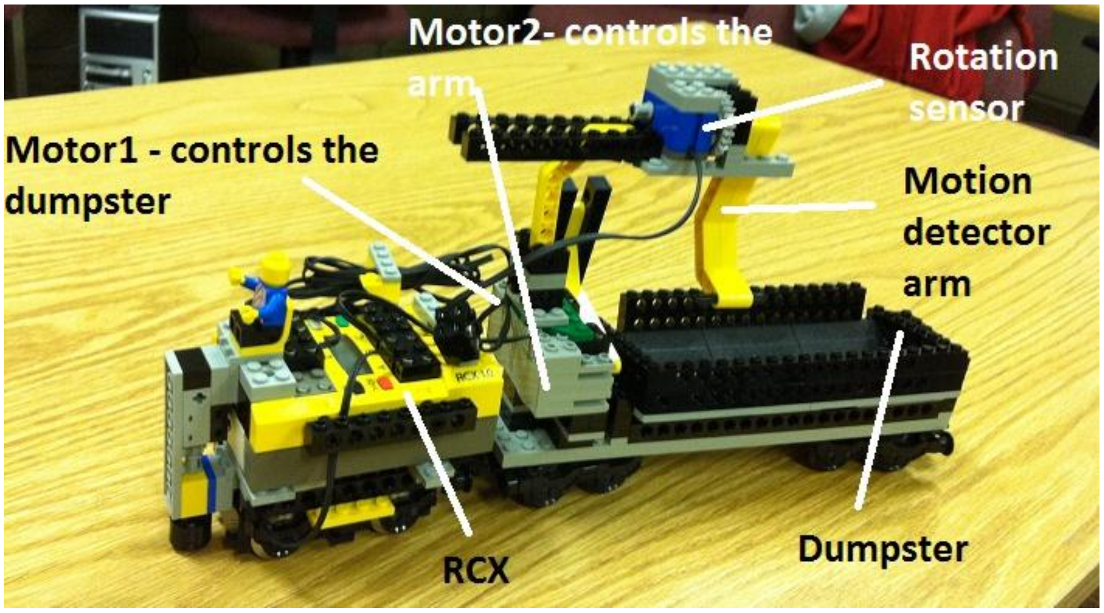
This algorithm results in guaranteed delivery more often than traditional TBS by offering lower WCETs, and therefore earlier TBS deadlines, without jeopardizing the correctness of the system (since the WCETs calculated are still guaranteed; they are just tighter than before).

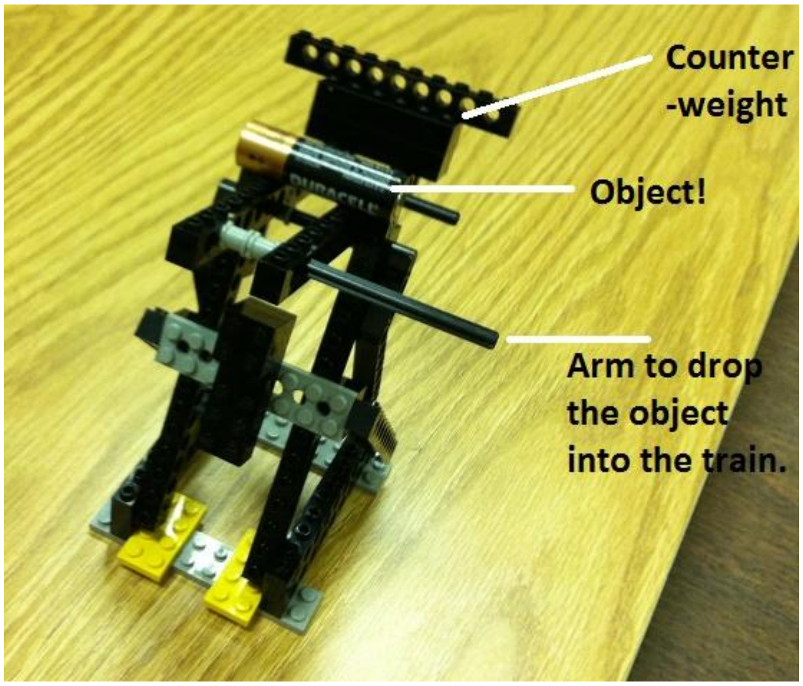
The second extension, which we call "preemptive EDF emulation", deals with the inherent non-preemptive limitation of the system. It is well known that, unlike preemptive EDF, non-preemptive EDF is non-optimal. Unfortunately, lack of preemption is a limitation of the physical system and cannot be avoided. However, we present a modification of the non-preemptive EDF algorithm that achieves *the same utilization bounds* as preemptive EDF (i.e. optimal). This normally impossible feat is only feasible due to two facts specific to our system. Firstly, the WCET of every periodic task is identical. Secondly, since the bin-checking guarantee we make only requires that we *check* the bin in each period, we can "cheat" and begin moving and checking a bin *before the periodic task is actually released*, so long as we *continue checking* until the instant of release. Given these two facts, we slightly modify non-preemptive EDF so that, in addition to tasks that have already been released, it also considers tasks that *will be released within at most one WCET* (using the deadline it *would* be assigned at that point). Additionally, the physical actuation code ensures that each bin check continues until at least the task's release time. This simple modification actually provides optimal scheduling (i.e. feasible whenever $U \leq 1$). The proof is included in Appendix A.

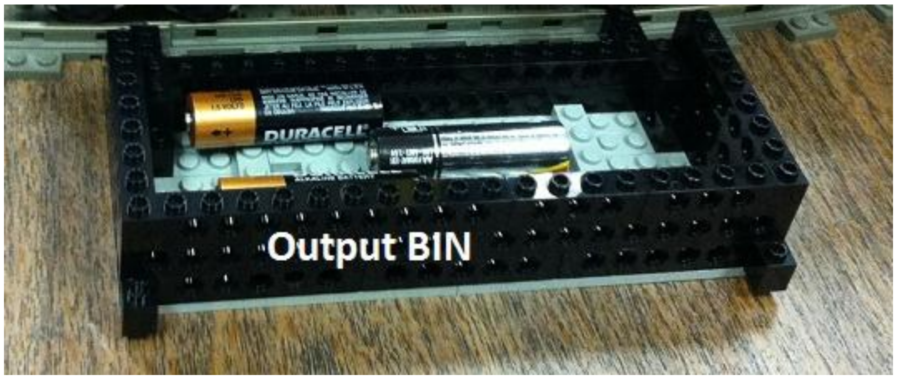
Physical Construction:

We have combined the pieces from a LEGO Mindstorms kit and a LEGO Train set [4], which has allowed us to mount the RCX on a track. This greatly simplifies the ring movement. We use a touch sensor to detect the arrival of the train at the input/output bins using small markers on the tracks. When the train arrives at a station, the train is programmed to align itself with the input bin/output bin properly. We use a rotation sensor attached to an object detection arm to detect when objects are received. An object falling into the back of the train causes the detector arm to move, which we detect. Since the rotation sensor is not very sensitive (it only detects 22.5 degree rotations), we have used gearing to significantly increase the effect of the arm on the sensor, and therefore the accuracy. Care has been taken to calibrate all sensors in order to filter out unwanted stimuli.

The RCX uses an engine block that came with the train kit to move forward and backward on the track. It also uses two motors to drive its mechanisms. The first motor drives the dumpster mounted on the train, which unloads objects at the output bin. The second controls the arm that is used to dump the objects from the input bins into the dumpster on the train.


Motor2- controls the arm
Rotation sensor
Motor1 - controls the dumpster
Motion detector arm
RCX
Dumpster


Counter -weight
Object!
Arm to drop the object into the train.


Output BIN

Test Scenario/Cases:

We have designed several test cases, included in the project archive, to display the benefits and limitations of the techniques we have developed. Below is an overview of each test, describing what it is designed to show and what the expected behavior is.

Test 1: The input bins have moderate periods, and all bins have objects waiting at the start. Due to a tight delivery deadline at the first bin checked, the first delivery cannot be guaranteed. Subsequent objects which might have otherwise been guaranteed are not due to a backlog that accumulates in the TBS. Thus, all deliveries are best effort in this test. However, it turns out that all are in fact delivered by their required deadline.

Test 2: Same as Test 1, but the first bin does not have an object. This prevents the TBS backlog, and allows the other two objects guaranteed delivery.

Test 3: Same as Test 1, but now we apply TBS WCET reduction. This reduces the TBS backlog, and provides all three objects guaranteed delivery, whereas before none achieved this.

Test 4: Same as Test 3, but now the second input bin (farthest from the output bin) has a reduced delivery deadline. Now two bins have the same delivery deadline, and neither has a TBS backlog when it is visited. However, the closer bin has its object guaranteed, whereas the farther does not, underscoring how TBS WCET reduction incorporating distance.

Test 5: Here, one input bin has a very short period, and accounts for very high utilization (over 80%), whereas the other two bins have very long periods (less than 8% utilization each). Total utilization is 100%. In this case, non-preemptive EDF fails because at one point it proceeds to service a bin distant from the short-period bin, which is released very soon after the train leaves. This requires almost an entire trip around the track, plus two check times (at the distant bin and the short-period bin), missing the deadline. Note this is particularly bad because the utilization analysis assumes WCETs based on maximum travel distance, but most travel is not this far, so actual utilization is significantly less than 100% (around 75 or 80%).

Test 6: Same as Test 5, but with preemptive EDF emulation enabled. Now the provably system meets all its deadlines. It accomplishes this by checking the short-period bin twice in a row at the beginning, which gives it more slack time for the rest of the schedule (since the second check can be completed right at release time). This demonstrates the increased set of feasible configurations this technique allows.

Our Code:

Our code is organized into three main components. The “core” component, contained in “train.c”, constitutes the first component of the project. It includes basic motor actuation and sensor code, and provides the main loop of the program. Invoked from this file is the “scheduler” contained in “train_edf_sched.c”, which is second component. The scheduler is completely decoupled from the physical control of the system via a small API. The scheduler is notified when objects are picked up or dropped off, and is consulted at each station to request the next destination. This information is then used in the main component to direct the actual movement. The third component is the “problem definition” file, which includes the real-time parameters of the system, including input bin periods and delivery deadlines, WCETs for movement and bin servicing, and scheduler parameters such as TBS size and whether TBS WCET reduction and/or preemptive EDF emulation are enabled.

Open Issues:

While the techniques we apply provide us with provable correctness bounds, the bounds given are fairly loose due to the WCET assumption of maximum travel distance. This becomes especially important as more input bins are added (since the minimum travel time could be very small compared to maximum).

A more effective scheduling algorithm would incorporate movement planning to get better bounds on WCET using actual distances. Such an algorithm would probably reduce the system to a graph, where vertices are bins and edges are WCETs for moving to and servicing each station. The scheduler then must plot a walk through the graph that meets all scheduling constraints. A brute force walk enumeration algorithm does not seem feasible, as the permutations explode exponentially. There is also no clear stopping point: even the hyperperiod is not safe, since you may start the next hyperperiod at a new location, changing the schedule, and sporadic tasks also complicate any hope for a static solution.

In terms of outside applications, it is possible that a solution to this problem might be applicable to the area of disk scheduling. Disks only services blocks (bins) in one direction, and rotational/seek distances (travel times) are not uniform as in our case, but if a generalized algorithm were found which functions on a distance graph, it might be effective.

In more mundane terms, the physical robot construction could also be improved. It might be interesting to allow input bins to be checked *without also retrieving the objects*. This would open more options for responding to delivery acceptance/rejection. Having multiple output bins where different classes of objects could be delivered would also be possible if the object dump bin could dump only some objects, opening other problems/algorithmic possibilities.

Tasks Completed:

1	Learn brickOS, run example programs for practice	Common
2	Look at brickOS kernel and determine where we will need to make modifications	Removed
3	Come up with some strategies for meeting deadlines more efficiently than an extreme-worst-case bound approach (factoring in current position, etc.)	Common
4	Setup the tracks and the layout	Vishnu
5	Build input/output bins	Common
6	Construct the basic delivery vehicle (with pickup/dropoff mechanisms)	Vishnu
7	Add bin station sensor to vehicle (for sensing when we arrive)	David
8	Add object pickup sensor (for sensing if objects are present in a bin)	David
9	Code the motion routines (vehicle movement, detecting current position)	Common
10	Code the loading/unloading routines for moving objects	Vishnu
11	Code the real-time scheduler to control the above two modules	David
12	Come up with (physical) test cases, both feasible and infeasible	David
13	Run the tests (will record video and analyze later, as testing is time-sensitive)	Common

References:

[1] http://brickos.sourceforge.net/documents.htm
[2] http://freshmeat.net/projects/brickos/
[3] http://did.mat.uni-bayreuth.de/~matthias/veranstaltungen/ws2004/mindstorms/doc/brickos-howto.html
[4] http://shop.lego.com/en-US/Cargo-Train-7939
[5] "Introduction to the legOS kernel"

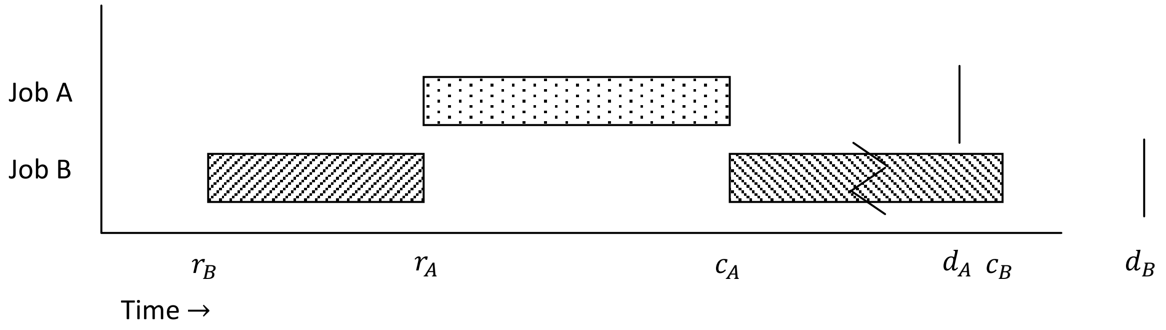
Appendix A: Preemptive EDF emulation optimality

We only present the proof of sufficiency here ($U \leq 1 \rightarrow feasible$), as the necessity proof is trivial and uninteresting. We prove sufficiency under preemptive EDF emulation (PEDFE) in a similar manner as with preemptive EDF (PEDF) itself. Given some system with $U \leq 1$, apply preemptive EDF scheduling to produce a feasible schedule. We will show this schedule can be transformed into a PEDFE schedule without missing any deadlines. We consider first only periodic tasks, and then we augment the proof with our single sporadic task.

Recall three critical facts about our system:

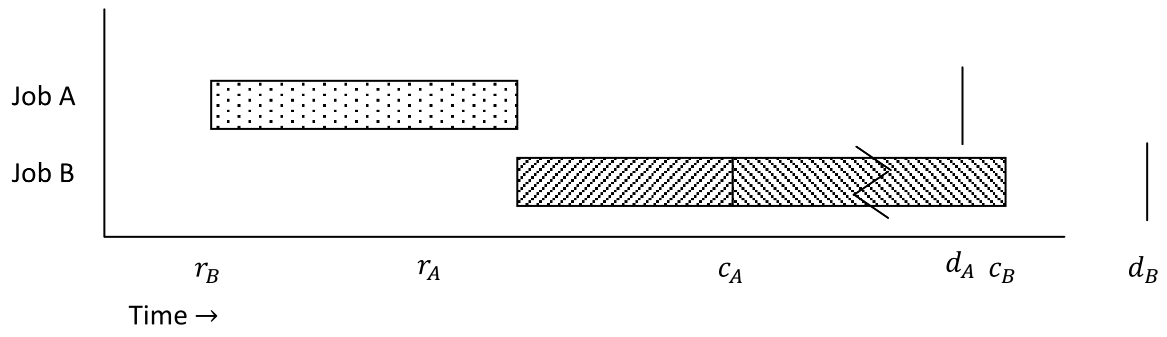
1. The WCETs for all periodic tasks are equal
2. We are allowed to execute a job before it is released, so long as it completes at the moment of release at the earliest
3. We can extend the execution time of a job up to its WCET if we so desire (by sustaining the bin check)

Since both PEDF and PEDFE schedule with earliest deadline first, the only place where a PEDF schedule violates PEDFE protocols is when a preemption occurs. Therefore, locate the first point in the PEDF schedule such that some job A preempts some job B, and job A is never preempted (at least one such configuration exists if any preemptions occur). This is visualized in the figure below. Note that we only consider the first such preemption; B might be preempted again later. We will remove the first such preemption, and then repeat this procedure until no preemptions remain.



The first preemption produced by a PEDF schedule

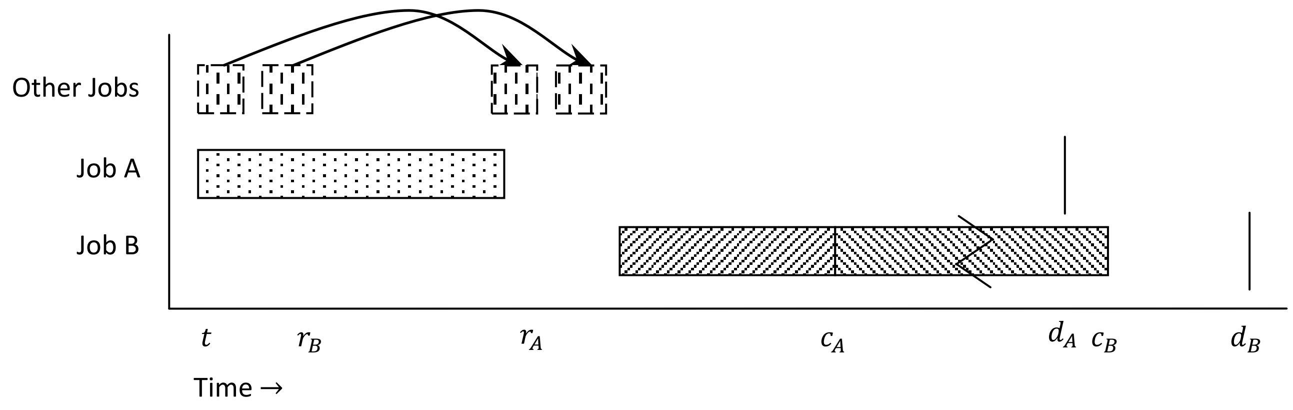
Here we have r_A, r_B as the job release times, c_A, c_B as job completion times, and d_A, d_B as job deadlines. Since this is a PEDF schedule, we know that $r_A < r_B$ (since otherwise A would simply block B, not preempt), $c_A < c_B$ (since A will not return control to B until it is finished), and $d_B < d_A$ (since B preempted A). Since B completes after A, we know that $r_A - r_B \leq e_B = e_A$ (this last equality since all WCETs are equal). Thus the beginning un-preempted segment of B is shorter than the WCET of A. This means that we can apply the following transformation in our system:



The same schedule segment, now transformed

This transformation is feasible under our system: job A starts at time r_B, and since we have proved that, $r_A - r_B \leq e_A$, we know that A completes on or after its release time r_A. Note that if A's actual execution time were less than $r_A - r_B$, we could artificially extend it to its full WCET, rerun PEDF, and start this process over again; this process is terminal since in the extreme all jobs will be extended to their WCET.

This transformed schedule is almost in PEDFE form, since both A and B will be considered for execution at time r_B, since $r_A - r_B \leq e_A$ (recall that PEDFE considers tasks that are at most 1 WCET before their release). However, it is possible that PEDFE would actually schedule A *even earlier than* r_B, shifting A's execution (and also B's) earlier to t (see figure below). In this case, either this shift interval $[t, r_B)$ covers only idle time, or some jobs execute *entirely within the shift interval*. Firstly, no job intersecting the shift interval can complete after r_B (since it would have been preempted, and we would have considered it first instead of jobs A and B). Secondly, no such a job can begin execution before t (because if it did, it would have a non-preemptive hold on the system, and A could not have been released).



The same schedule segment, showing the possibility of PEDFE back-shifting job A

In either of these cases, all idle time or short jobs can easily be moved to later in the schedule. In particular, the deadlines of the short jobs must be after d_A, so they can safely be placed after A's execution (because A met its deadline before it was backshifted), and possibly after B's execution if their deadlines are even later. Idle time can, of course, be shifted anywhere, as it has no deadline.This schedule segment is now in PEDFE form: job A executes completed before job B, such that at scheduling point t, A is chosen over B even though it has yet to be released, since its deadline is before B's.

This process eliminates this preemption, and if applied iteratively, will eliminate all preemptions in the schedule. There may still be places where PEDFE reorders tasks relative to PEDF, or shifts some tasks earlier, covering idle time with its "early execution". These can be resolved with simple task shifting/reordering, however, and since PEDFE only reorders tasks by putting earlier deadline tasks first, no deadlines will be missed (since such tasks still met their deadline even though they were placed later, so any forward-displaced tasks would meet their later deadlines).

Thus, PEDFE produces a feasible schedule wherever PEDF produces one, and therefore PEDFE also has $U \leq 1$ as a sufficient condition for schedulability.