Restricting EDF migration on uniform multiprocessors *

Shelby Funk **Sanjoy K. Baruah**
The University of North Carolina
CB # 3175
Chapel Hill, NC 27599-3175
{shelby,baruah}@cs.unc.edu

Abstract

Restricted migration of periodic and sporadic tasks on uniform multiprocessors is considered. On multiprocessors, job migration can cause a prohibitively high overhead — particularly in real-time systems where accurate timing is essential. However, if periodic tasks do not maintain state between separate invocations, different jobs of the same task may be allowed to execute on different processors — i.e., *restricted migration* may be permitted. On uniform multiprocessors, each processor has an associated speed. A job executing on a processor of speed s for t units of time will perform $s \times t$ units of work. A utilization-based test for restricted migration on uniform multiprocessors is provided where each processor schedules jobs using the Earliest Deadline First (EDF) scheduling algorithm.

Keywords: hard real-time systems, periodic tasks, earliest deadline first, uniform multiprocessors, migration.

Please send correspondence to Shelby Funk
Phone (919) 962-1837
Fax (919) 962-1799

*Supported in part by the National Science Foundation (Grant Nos. CCR-0204312, CCR-0309825, and ITR-0082866).

1 Introduction

In hard real-time systems, correctness is determined not only by considering the output of the system, but also by considering its timing characteristics. Each job submitted to a real-time system has a deadline and it is imperative that execution completes by this deadline. On multiprocessor real-time systems, there are several different processors upon which jobs may execute. These processors may all be the same, in which case the multiprocessor is *identical*, or they may differ in some way. This paper considers real-time systems that use *uniform* multiprocessors. Each processor of a uniform multiprocessor has an associated speed. If a processor with speed s executes a job for t time units, then $s \times t$ units of work are performed.

Real-time systems often contain *periodic tasks*. These tasks generate jobs that are repeated at regular intervals of time. Each periodic task has an associated *utilization*, which is the proportion of processor capacity the task requires if it is executed on a unit-speed processor. A *periodic task set* is a real-time system that is comprised solely of periodic tasks. Task utilization can be used to devise a *schedulability test* which determines if all jobs can be guaranteed to meet their deadline when scheduled using a particular scheduling algorithm. For example, Liu and Layland [4] showed that on uniprocessors all jobs will meet their deadlines using the Earliest Deadline First (EDF) scheduling algorithm if the total utilization of all tasks is at most 1.

Some real-time systems may allow jobs to *migrate* between processors by permitting an executing job to be interrupted and to restart on a different processor. However, for some multiprocessors, the overhead incurred when a job migrates can be prohibitively high. In these systems, jobs should execute only on the processor upon which they began their execution. When considering periodic task sets, there are three possible levels of migration:

Full migration If a job is interrupted, it may resume execution on any processor.

Partitioning Each task is assigned to a single processor. All jobs of a given task must execute on the processor to which the task is assigned.

Restricted migration Each job must execute on only one processor. However, different jobs of the same periodic task can execute on different processors.

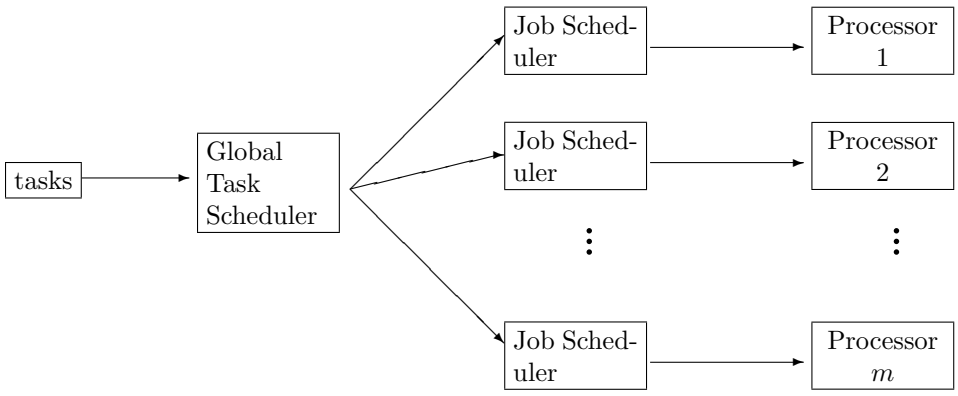

Figure 1: Scheduling tasks with restricted migration.

If a periodic task set is scheduled with full migration, a *global scheduler* must be employed. A global scheduler is a single scheduler that makes the scheduling decisions for all the processors of a multiprocessor system. On the other hand, a partitioned periodic task set may be scheduled using a *partitioned scheduler*. Partitioned schedulers have a separate scheduler for each processor. Each scheduler in a partitioned scheduler can employ uniprocessor scheduling techniques. Periodic tasks sets that are scheduled using restricted migration may use a combination of global and partitioned scheduling. When a periodic task generates a new job, the global scheduler assigns the job to a processor and the partitioned scheduler for the given processor determines when the job should execute. Figure 1 illustrates a restricted migration scheduling scheme.

This paper considers restricted migration scheduling of periodic task sets on uniform multiprocessors. In particular, we present a schedulability test for restricted migration scheduling of task sets using EDF on each processor. Restricted migration is most appropriate for tasks which have little or no state shared between successive jobs. In this case, the overhead of moving a task (but not a job) to a different processor is very low. While partitioning also reduces the migration overhead of a system, partitioned systems may be too restrictive. In particular, the partitions must be determined before the system begins execution. Thus, dynamic systems in which tasks may *join* or *leave* may not be appropriate for partitioned systems. Restricted migration systems can

be used to schedule these dynamic task systems provided that the system is always guaranteed to satisfy the schedulability test provided in this paper. Furthermore, partitioning may force the load on the processors to be highly imbalanced [2]. If different jobs may execute on different processors, the load on the processors may be more consistent. Thus, restricted migration adds a degree of *flexibility* to the system and allows for *load balancing*.

Restricted migration was considered for identical multiprocessors by Baruah and Carpenter [2]. To our knowledge, restricted migration has never been considered for uniform multiprocessors. This paper introduces a restricted migration schedulability test which can be used to guarantee deadlines will be met if the job scheduler on each processor is EDF [4, 3]. In addition, we introduce the concept of *semi-partitioning* and provide a schedulability test for semi-partitioning with restricted migration. Finally, we introduce *virtual processors*, which may be used to recapture spare capacity when scheduling with semi-partitioning and we provide a schedulability test for semi-partitioning with restricted migration using virtual processors.

The remainder of this paper is organized as follows. In Section 2, we introduce the model and definitions we will be using. Section 3 defines the restricted migration algorithm r-EDF in more detail and introduces a schedulability test for r-EDF. In addition, we define semi-partitioning in this section and introduce a schedulability test for semi-partitioned r-EDF. Section 4 introduces virtual processors and provides a schedulability test for semi-partitioned r-EDF with virtual processors. In addition, we describe the global scheduler r-SVP which schedules semi-partitioned systems with virtual processors. Finally, in Section 5 we conclude and discuss areas for further research.

2 Model and definitions

The notation $\pi = [s_1, s_2, \ldots, s_m]$ denotes the uniform multiprocessor with m processors of speeds $s_1, s_2, \ldots, s_m$ respectively, with $s_1 \geq s_2 \geq \cdots \geq s_m$. We use the notation $S(\pi)$ to denote the cumulative computing capacity of all of π's processors: $S(\pi) \stackrel{\text{def}}{=} \sum_{i=1}^{m} s_i$.

A real-time system is a set of **jobs** that are executed on a given processing platform. Each job is characterized by three parameters: its **arrival time**, a, its **execution requirement**, e, and its deadline d. The job $j = (a, e, d)$ must receive e units of execution in the time interval

$[a, d)$. A job is said to be **active** at any time t, where $a \leq t < d$. Note that a job is considered active until its deadline d even if the job's execution completes before d. Jobs are scheduled using a scheduling algorithm. If a set of jobs meets all its deadlines when scheduled using algorithm A on the processing platform π, the job set is said to be A**-schedulable** on π.

Periodic and sporadic task systems. A **periodic** real-time task $T = (e, p)$ [4, 5] is characterized by the two parameters *execution requirement* e and *period* p. A periodic task generates an infinite sequence of jobs, each with an execution requirement of at most e units. Successive jobs are generated p time units apart, and each job has a deadline p time units after its arrival time. A **sporadic** real-time task [6] is similar to a periodic task with the exception that p is the *minimum interarrival time* of successive jobs. Thus, if a job generated by the sporadic task $T = (e, p)$ arrives at time a, it has a worst-case execution time e and a deadline $p+a$. However, the next job generated by T may arrive at any time $t \geq a + p$. A **task system** $\tau = \{T_1, \ldots, T_n\}$ of tasks is comprised of several periodic or sporadic tasks, with each task T_i having execution requirement e_i and period p_i.

We refer to the quantity $u_i \stackrel{\text{def}}{=} e_i/p_i$ as the *utilization* of task T_i. For system τ, $U_{\mathsf{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{T_i \in \tau} u_i$ is referred to as the **cumulative utilization** (or simply utilization) of τ. Similarly, $U_{\mathsf{max}}(\tau) \stackrel{\text{def}}{=} \max_{T_i \in \tau} u_i$ is referred to as the **largest utilization** in τ, and $U_{\mathsf{min}}(\tau) \stackrel{\text{def}}{=} \min_{T_i \in \tau} u_i$ is referred to as the **smallest utilization** in τ.

We will refer to a (uniform multiprocessor, task system) ordered pair as a **uniform multiprocessor system**; when clear from context, we may simply refer to it as a **system**.

3 Restricting migration

In this section, we determine a schedulability test for restricted migration scheduling on uniform multiprocessors. Figure 1 illustrates a general restricted migration scheduler. In this algorithm, each job generated by a periodic or sporadic task is submitted to a global scheduler which determines which processor the job should be scheduled on. The global scheduler then submits the job to the scheduler associated with that processor. In this section, we assume that the job scheduler on each processor is using the preemptive EDF scheduling algorithm. We denote this algorithm $\mathsf{r\text{-}EDF}$. Thus, for $\mathsf{r\text{-}EDF}$ the job scheduler on each processor in Figure 1 is preemptive EDF.

By the optimality of preemptive EDF on uniprocessors [4], the total utilization of tasks scheduled on a processor of speed s must be less than or equal to s. Therefore, a job generated by a task with utilization u can be assigned to a processor of speed s whenever the active jobs assigned to the given processor are generated by tasks that have a total utilization no more than $(s-u)$. This reasoning is used to determine the following utilization-based r-EDF-schedulability test for uniform multiprocessor systems.

Theorem 1 Let π be any m-processor uniform multiprocessor and let τ be any task set such

$$U_{\mathsf{sum}}(\tau) \leq S(\pi) - (m-1)U_{\mathsf{max}}(\tau). \tag{1}$$

Then τ is r-EDF-schedulable on π.

Proof: We prove this theorem by contradiction. Let τ be any task set that is not r-EDF-schedulable on π. Assume that a job of the task T_i can not be feasibly scheduled on any processor of π and that U_{act} is the total utilization of active jobs at the time when T_i can not be scheduled. Since T_i can not fit on any processor, the slack of each processor of π is less than u_i. Therefore,

$$U_{\mathsf{act}} > \sum_{j=1}^{m}(s_j - u_i) = S(\pi) - m \cdot u_i.$$

Now consider U_{act}. Since T_i has a job that has just arrived, no other job generated by T_i can be active. On the other hand, every task of τ other than T_i may have an active job. Therefore,

$$U_{\mathsf{act}} \leq U_{\mathsf{sum}}(\tau) - u_i.$$

Combining these two inequalities gives

$$\begin{aligned} U_{\mathsf{sum}}(\tau) - u_i &> S(\pi) - m \cdot u_i \\ \Rightarrow U_{\mathsf{sum}}(\tau) &> S(\pi) - (m-1)u_i \\ \Rightarrow U_{\mathsf{sum}}(\tau) &> S(\pi) - (m-1)U_{\mathsf{max}}(\tau). \end{aligned}$$

Therefore, the theorem holds. ■

Notice that if $U_{\mathsf{max}}(\tau) > s_m$, it is actually beneficial to consider only a *subset* of the processors of π when applying this Theorem 1. In particular, if $s_k \geq U_{\mathsf{max}} > s_{k+1}$ for some $k < m$, then

$$S(\pi) - (m-1)U_{\mathsf{max}} < \sum_{i=1}^{k} s_i - (k-1)U_{\mathsf{max}}.$$

Therefore, it may be the case that the task set satisfies the test for $\pi' = [s_1, s_2, \ldots, s_k]$ but not for π. It is easy to see that adding processors would not cause an r-EDF-schedulable task set to become unschedulable. Since the above test applies for *sporadic* task sets, having a task T_i generate jobs that are scheduled on processors $s_{k+1}, s_{k+2}, \ldots, s_m$ does not affect the application of the test. From the perspective of the k fastest processors of π, it simply appears as though T_i has some long inter-arrival times for consecutive jobs. This property, called *robustness*, was proven to hold for preemptive EDF by Baruah [1].

Even in cases where $U_{\mathsf{max}} \leq s_m$, the bound determined in Theorem 1 may not be practical. For example, if τ consists of a few tasks with large utilization and several tasks with significantly smaller utilization, it may make more sense to reserve the fastest processors for the heaviest tasks and allow the lighter tasks to execute only on slower processors. A task set that fails Condition 1 may be become schedulable by using a minor modification of r-EDF which takes this observation into account. We call this modification *semi-partitioning*.

3.1 Semi-partitioning

Since Condition 1 may fail for systems where $U_{\mathsf{max}}(\tau)$ is significantly larger than $U_{\mathsf{min}}(\tau)$, we wish to consider a scheme whereby jobs generated by a given task may only be assigned to a subset of the processors of π. Thus, the system is somewhat partitioned because the jobs generated by a single task are only permitted to execute on a subset of the processors of π. However, this is not a fully partitioned system because each task may be assigned to several processors. In this modification of r-EDF, called r-EDF(k, ℓ), tasks are divided into two groups. The "heavy" group consists of the k tasks with the largest utilization and the "light" group consists of all the other tasks of τ. The heavy jobs are scheduled on the ℓ fastest processors of π and the light jobs are scheduled on the

remaining processors. We call this strategy **semi-partitioning** because the heavy group of tasks and the light group of tasks are executed on disjoint processors of π.

Example 1 Consider the uniform multiprocessor $\pi = [8, 3, 3]$ and the task set τ comprised of 21 tasks with $u_1 = 4$, $u_2 = u_3 = 1$, $u_4 = u_5 = \ldots = u_{11} = 0.5$, and $u_{12} = u_{13} = \ldots = u_{21} = 0.1$. Thus, $U_{\mathsf{sum}}(\tau) = 11$ and $U_{\mathsf{max}}(\tau) = 4$ and Condition 1 does not hold ($11 > 14 - 2 \cdot 4$) so this system may not be r-EDF-feasible. However, the system is r-EDF(3,1)-feasible since the utilization of the first 3 tasks is 6 so these jobs can be r-EDF scheduled on the fastest processor of π[1]. Also, applying Theorem 1 to $\pi' = [3, 3]$ and τ' with $U_{\mathsf{max}}(\tau') = 0.5$ and $U_{\mathsf{sum}}(\tau') = 5$, we see that $5 \leq 6 - 1 \cdot 0.5$, therefore the 18 light tasks of τ can be r-EDF scheduled on the two slowest processors of π.

Example 1 illustrates that a system which fails Condition 1 may be r-EDF(k, ℓ)-schedulable using semi-partitioning. Notice that r-EDF(k, ℓ)-schedulability can be tested by simply applying Condition 1 twice. The following theorem formalizes this observation.

Theorem 2 Let π be any uniform multiprocessor and let τ be any task set. Assume the n tasks of τ are indexed according to utilization, $u_i \geq u_{i+1}$ for all $1 \leq i < n$. If

$$\sum_{i=1}^{k} u_i \leq \sum_{j=1}^{\ell} s_j - (\ell - 1)u_1 \text{ and } \sum_{i=k+1}^{n} u_i \leq \sum_{j=\ell+1}^{m} s_j - (m - \ell - 1)u_{k+1}$$

then τ is r-EDF(k, ℓ)-schedulable on π.

If the two conditions in Theorem 2 are added together, the resulting condition is

$$U_{\mathsf{sum}}(\tau) \leq S(\pi) - (\ell - 1)U_{\mathsf{max}}(\tau) - (m - \ell - 1)u_{k+1}.$$

The right hand side of this equation is *never* smaller than the right hand side of Condition 1, and it may be significantly larger if $u_{k+1} << U_{\mathsf{max}}(\tau)$. Also, if the largest utilization is larger than the speed of one or more processors some of the tasks are *defacto* semi-partitioned even if it is scheduled using r-EDF as opposed to r-EDF(k, ℓ). For example, since the heaviest task of τ in Example 1 has utilization larger than the speed of the two slower processors of π, we can see that it can only be

[1] On a single processor r-EDF and EDF generate the same schedule.

executed on the fastest processor of π even if the system had not been semi-partitioned. Thus, task sets with widely divergent utilizations may benefit from semi-partitioning — particularly if $U_{\mathsf{max}}(\tau) > s_m$. These two observations lead us to consider two natural ways to determine how to semi-partition a system.

- If $U_{\mathsf{max}}(\tau) > s_m$, determine the index ℓ such that $s_\ell \geq U_{\mathsf{max}}(\tau) > s_{\ell+1}$ and determine the largest index k for which the first condition of Theorem 2 holds. Test if the second condition of Theorem 2 holds for the given k and ℓ.

- Otherwise, consider the tasks of τ. If $U_{\mathsf{max}}(\tau) >> U_{\mathsf{min}}(\tau)$, then semi-partitioning may be useful because of the widely divergent utilizations of τ. If there is some point where the utilization decreases significantly, let T_k be the heaviest task before this decrease. If there is no point at which the utilization decreases significantly, then let $k = \lfloor n/2 \rfloor$. Once k is chosen, determine the smallest index ℓ for which the first condition of Theorem 2 holds. Test if the the second condition of Theorem 2 holds for the given k and ℓ.

Of course, it is possible that the second condition does not hold. In this case, semi-partitioning can be done several times. The system can be semi-partitioned as many as $m-1$ times, where $m-1$ semi-partitions results in a fully partitioned system. The notation $\mathsf{r\text{-}EDF}(n_1, m_1; n_2, m_2; \ldots; n_k, m_k)$ denotes the $\mathsf{r\text{-}EDF}$ schedule with the following restrictions

$$\begin{gathered}
\text{Tasks } T_1, T_2, \ldots, T_{n_1} \text{ execute only on processors } s_1, s_2, \ldots, s_{m_1} \\
\text{Tasks } T_{n_1+1}, T_{n_1+2}, \ldots, T_{n_2} \text{ execute only on processors } s_{m_1+1}, s_{m_1+2}, \ldots, s_{m_2} \\
\vdots \\
\text{Tasks } T_{n_k+1}, T_{n_k+2}, \ldots, T_n \text{ execute only on processors } s_{m_k+1}, s_{m_k+2}, \ldots, s_m.
\end{gathered}$$

Thus, the test for $\mathsf{r\text{-}EDF}(n_1, m_1; n_2, m_2; \ldots; n_k, m_k)$-schedulability would involve $k+1$ applications of Theorem 1, each of which must be satisfied.

4 Virtual processors

Example 1 in the previous section illustrates that a system that does not satisfy Condition 1 can still be schedulable using a minor modification of $\mathsf{r\text{-}EDF}$. Notice that the first partition in this example

schedules the three heaviest tasks on a single processor of speed 8. Since the total utilization of these three tasks is 6, there are 2 units of spare capacity available which can be "loaned" to the system (π', τ') if necessary. For example, if 6 more tasks with execution utilization 0.1 were added to τ, $U_{\mathsf{sum}}(\tau')$ becomes 5.6 and Condition 1 fails on the system (π', τ') *unless* π' "borrows" of capacity from the faster partition. Thus the fastest processor of π would be divided into 2 *virtual processors* — one processor of speed 6 devoted to the heaviest tasks of τ and one processor of speed 2 devoted to τ'. Once the 2 units of capacity are borrowed, Condition 1 is satisfied since $5.6 \leq (6+2) - 2 \cdot 0.5$. (We now subtract $2 \cdot 0.5$ instead of $1 \cdot 0.5$ because $\pi' = [2, 3, 3]$.)

If capacity can be borrowed, we have more flexibility in determining the semi-partitioning points for π and τ (i.e., in determining k and ℓ). For example, since the heaviest task of τ in Example 1 can *only* be executed on the fastest processor of π and all the remaining tasks can be executed on any of the processors of π, a natural partitioning would be to only allow T_1 to be semi-partitioned onto the fastest processor and allow the remaining tasks to be semi-partitioned onto the slower two processors. Applying Condition 1 to this partition gives

$$4 \leq 8 - 0 \text{ and } 7 \leq 6 - 1.$$

The second test clearly fails. However, if the 4 units of spare capacity remaining from the first test are given to the second partition, we have

$$4 \leq 4 - 0 \text{ and } 7 \leq 10 - 2.$$

Therefore, this system can be scheduled with the given semi-partition provided the fastest processor is divided into two virtual processors. We use the notation $\mathsf{r\text{-}EDF}(k, \ell, c)$ to denote this type of semi-partitioning. Thus, the semi-partition with virtual processor discussed above is denoted $\mathsf{r\text{-}EDF}(1, 1, 4)$.

Definition 1 Let π be any uniform multiprocessor and let τ be any task set. Assume the n tasks of τ are indexed according to utilization, $u_i \geq u_{i+1}$ for all $1 \leq i < n$. Then $\mathsf{r\text{-}EDF}(k, \ell, c)$ indicates the scheduling algorithm in which tasks $T_1, T_2, \ldots, T_k$ are $\mathsf{r\text{-}EDF}$-scheduled on processors $s_1, s_2, \ldots, s_{\ell-1}, s_\ell - c$ and tasks $T_{k+1}, T_{k+2}, \ldots, T_n$ are $\mathsf{r\text{-}EDF}$-scheduled on processors $c, s_{\ell+1}, s_{\ell+2}, \ldots, s_m$.

Thus, the $\ell' th$ processor of π is divided into two virtual processors — one with capacity $(s_\ell - c)$ which is used by the first semi-partition and one with capacity c which is used in the second semi-partition.

Determining appropriate semi-partitions with virtual processors is similar to determining appropriate semi-partitions without virtual processors, though one step is added to the process. Once an appropriate k and ℓ have been determined check if the semi-partition is r-EDF(k, ℓ)-schedulable using Theorem 2. If this test fails, let

$$c = \sum_{j=1}^{\ell} s_j - (\ell - 1)u_1 - \sum_{i=1}^{k} u_i.$$

Then c is the spare capacity in the first semi-partition which can be loaned to the second semi-partition. Test if the system is r-EDF(k, ℓ, c)-schedulable by checking if (π', τ') is an r-EDF-schedulable system, where $\pi' = [c, s_{\ell+1} s_{\ell+2}, \ldots, s_m]$ and $\tau' = \{T_{k+1}, T_{k+2}, \ldots, T_n\}$.

Theorem 3 Let π be any uniform multiprocessor and let τ be any task set. Assume the n tasks of τ are indexed according to utilization, $u_i \geq u_{i+1}$ for all $1 \leq i < n$. If

$$\sum_{i=1}^{k} u_i \leq \sum_{j=1}^{\ell} s_j - (\ell - 1)u_1,$$
$$c = \sum_{j=1}^{\ell} s_j - (\ell - 1)u_1 - \sum_{i=1}^{k} u_i < s_\ell, \text{ and}$$
$$\sum_{i=k+1}^{n} u_i \leq \sum_{j=\ell+1}^{m} s_j + c - (m - \ell)u_{k+1}$$

then τ is r-EDF$(k, \ell$,c)-schedulable on π.

Just as semi-partitioning without virtual processors can be repeated several times, so can semi-partitioning with virtual processors. The main difference is that while $(m - 1)$ semi-partitions without virtual processors results in a fully partitioned system, the same does not hold for semi-partitioning with virtual processors.

4.1 Scheduling with virtual processors

The previous two sections introduces three variations of r-EDF — one with no partitioning, one with semi-partitioning and one with semi-partitioning and virtual processors. All these variations use the EDF scheduling algorithm on the processors. The difference between these variations of r-EDF is determined by the global scheduler they use to assign jobs to processors as the tasks generate them. In this section, we describe r-SVP — the restricted migration semi-partitioning global scheduler with virtual processors. The other variations of global schedulers can be determined by removing the implementation of the semi-partitioning and/or the virtual processor from the r-SVP scheduler.

The r-SVP scheduler maintains the mapping from semi-partitions to virtual processors, the mapping from virtual processors to actual system processors, and the current available capacity on each virtual processor. These parameters are maintained in the arrays SemiPartMap and ProcGap respectively. SemiPartMap is an $s \times 2$ array, where s is the number of semi-partitions in the system. SemiPartMap contains the minimum and maximum processor indices associated with each semi-partition, denoted SemiPartMap$(*, \min)$ and SemiPartMap$(*, \max)$, respectively. The indices point to the minimum and maximum indices in the array ProcGap that are associated with the semi-partitions. They are not the actual processor numbers associated with the semi-partition. SemiPartMap is not changed after initialization. ProcGap is a $v \times 2$ array, where v is the number of virtual processors in the system. ProcGap contains the available capacity, or *gap* for each virtual processor and the actual processor number associated with the virtual processor, denoted ProcGap(∗,gap) and ProcGap(∗,proc), respectively. ProcGap(∗,gap) is initialized with the gap on each virtual processor and ProcGap(∗,proc) with its system processor number. The gap is modified when jobs are assigned to the virtual processor and when their deadlines arrive. The system processor assignment never changes. The pseudo-code for r-SVP is shown in Figure 4.1.

When a job is generated by a task and submitted to the r-SVP global scheduler, it must provide three parameters: the semi-partition that the task belongs to, sp, the task's utilization, u_j, and the job's deadline d. r-SVP then assigns the job to the processor s_j in the semi-partition sp with the largest gap. ProcGap(s_j,gap) is decreased by u_j to reflect the arrival of the job. Also, an interrupt is set to increase ProcGap(s_j,gap) by u_j when the deadline is reached. Since r-SVP uses the values in ProcGap(s_j,gap) to determine the processor assignments, it is essential that the interrupts that

```
function r-SVP(job, sp, util, d)
    %Submit job with utilization = util and
    % deadline = d to some processor in
    % semi-partition sp
    vproc ← s_j such that ProcGap(s_j, gap) = max gap in sp
    if ProcGap(vproc,gap)< util
          Error
          exit
    proc ← ProcGap(vproc,proc)
    submit job to processor proc
    ProcGap(vproc,gap)← ProcGap(vproc,gap)−util
    set interrupt to increase ProcGap(vproc,gap) by util at d
```

occur at job deadlines have higher priority than the r-SVP algorithm. Otherwise, if a job arrives at another job's deadline, r-SVP may determine that the arriving job can not fit onto any processor.

Variations and optimizations. The version of r-SVP shown in Figure 4.1 can be varied in several ways — either to address different system requirements or to optimize the algorithm. For example, in Figure 4.1, r-SVP assigns jobs to the processor with the largest gap. However, any method may be used to select among the eligible processors (e.g., first fit, best fit, worst fit, random). The choice depends on the load-balancing goal for the given system.

Also, r-SVP can be optimized if it maintains only one entry in ProcGap for each processor in π (as opposed to one entry per *virtual* processor). This implementation would simplify the bookkeeping for shared processors, though the method of choosing among eligible processors would have to be modified slightly. For any processor s_i, only one semi-partition monitors the gap of its virtual processor on s_i — namely the first semi-partition for which $\mathsf{SemiPartMap}(sp, \min) = s_i$. If all other semi-partitions assign jobs to s_i only as a "last resort", then these semi-partitions will never use more capacity than is allocated to them. If some semi-partition were to use more than its allotted capacity on the virtual processor, then the semi-partition would require more capacity than is allotted to it. However, Theorem 1 proves that this can not happen. Of course, since there is no test to ensure that the job actually fits on the "last resort" processor, this optimization could only be used in cases where the application programmers are trusted to confirm the system is feasible before submitting the system.

5 Conclusion and future work

This paper considers EDF-scheduling of periodic and sporadic task sets on uniform multiprocessors where restricted migration is allowed. Migrating state from one processor to another may result in unacceptable overhead. Therefore, the scheme presented in this paper does not allow jobs to migrate between processors — once a task generates a job and that job is assigned to a specific processor, it can not move to any other processor. However, there are tasks that do not maintain state *between* jobs. These tasks would be appropriate candidates for scheduling with restricted migration.

We have discussed several approaches to restricted migration scheduling. First, we considered the case where a job can be scheduled on any processor with enough available capacity and we developed a utilization-based schedulability test for this type of task system. However, we showed that jobs with widely divergent task utilizations may fail this test. Due to this observation, we introduced the concept of *semi-partitioning* in which jobs generated by a given task may only be executed on a subset of all the processors in the system and we showed how to modify the schedulability test to account for this restriction. Finally, we developed a method for recapturing the capacity that may be wasted by the semi-partitioning scheme using *virtual processors* and we once again showed how to modify the schedulability test to account for these virtual processors.

This paper is a first look at restricted migration on uniform multiprocessors and there are many more interesting avenues of research on this topic. For example, we may want to consider task sets in which different tasks have different migration requirements — some tasks may be allowed to execute on a single processor, some may be allow restricted migration, and some may allow for full migration. Also, while we assumed that processors scheduled jobs using preemptive EDF, restricted migration may be used in conjunction with any job scheduling algorithm. It would be interesting to consider how the results in this paper would be affected if a different algorithm were used — particularly if the algorithm did not allow for preemption. Finally, if the task set being scheduled is dynamic, it may be interesting to consider the implications of allowing semi-partitions to be dynamic as well.

References

[1] Baruah, S. Robustness results concerning EDF scheduling upon uniform multiprocessors. In *Proceedings of the EuroMicro Conference on Real-Time Systems* (Vienna, Austria, June 2002), IEEE Computer Society Press, pp. 95–102.

[2] Baruah, S., and Carpenter, J. Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. In *Proceedings of the Euromicro Conference on Real-time Systems* (Porto, Portugal, 2003), IEEE Computer Society Press.

[3] Dertouzos, M. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress* (1974), pp. 807–813.

[4] Liu, C., and Layland, J. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 20*, 1 (1973), 46–61.

[5] Liu, C. L. Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary 37-60 II* (1969), 28–31.

[6] Mok, A. K. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983. Available as Technical Report No. MIT/LCS/TR-297.