A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms

John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan,
James Anderson, and Sanjoy Baruah

Department of Computer Science, University of North Carolina at Chapel Hill

1 Introduction

Real-time multiprocessor systems are now commonplace. Designs range from single-chip architectures, with a modest number of processors, to large-scale signal-processing systems, such as synthetic-aperture radar systems. For uniprocessor systems, the problem of ensuring that deadline constraints are met has been widely studied: effective scheduling algorithms that take into account the many complexities that arise in real systems (*e.g.*, synchronization costs, system overheads, *etc.*) are well understood. In contrast, researchers are just beginning to understand the trade-offs that exist in multiprocessor systems. In this chapter, we analyze the trade-offs involved in scheduling independent, periodic real-time tasks on a multiprocessor.

Research on real-time scheduling has largely focused on the problem of scheduling of recurring processes, or *tasks*. The periodic task model of Liu and Layland is the simplest model of a recurring process [16, 17]. In this model, a *task* T is characterized by two parameters: a worst-case *execution requirement* e and a *period* p. Such a task is invoked at each nonnegative integer multiple of p. (Task invocations are also called *job releases* or *job arrivals*.) Each invocation requires at most e units of processor time and must complete its execution within p time units. (The latter requirement ensures that each job is completed before the next job is released.) A collection of periodic tasks is referred to as a *periodic task system* and is denoted τ.

We say that a task system τ is *schedulable* by an algorithm A if A ensures that the timing constraints of all tasks in τ are met. τ is said to be *feasible* under a class C of scheduling algorithms if τ is schedulable by some algorithm $A \in C$. An algorithm A is said to be *optimal* with respect to class C if $A \in C$ and A correctly schedules every task system that is feasible under C. When the class C is not specified, it should be assumed to include all possible scheduling algorithms.

Classification of scheduling approaches on multiprocessors. Traditionally, there have been two approaches for scheduling periodic task systems on multiprocessors: *partitioning* and *global scheduling*. In global scheduling, all eligible tasks are stored in a single priority-ordered queue; the global scheduler selects for execution the highest priority tasks from this queue. Unfortunately, using this approach with optimal uniprocessor scheduling algorithms, such as the rate-monotonic (RM) and earliest-deadline-first (EDF) algorithms, may result in arbitrarily low processor utilization in multiprocessor systems [11]. However, recent research on *proportionate fair* (Pfair) scheduling has shown considerable promise in that it has produced the only known optimal method for scheduling periodic tasks on multiprocessors [1, 3, 5, 19, 24].

In partitioning, each task is assigned to a single processor, on which each of its jobs will execute, and processors are scheduled independently. The main advantage of partitioning approaches is that they reduce a *multiprocessor* scheduling problem to a set of *uniprocessor* ones. Unfortunately, partitioning has two negative consequences. First, finding an optimal assignment of tasks to processors is a bin-packing problem, which is NP-hard in the strong sense. Thus, tasks are usually partitioned using non-optimal heuristics. Second, as shown later, task systems exist that are schedulable if and only if tasks are *not* partitioned. Still, partitioning approaches are widely used by system designers.

In addition to the above approaches, we consider a new "middle" approach in which each job is assigned to a single processor, while a task is allowed to migrate. In other words, inter-processor task migration is permitted only at job boundaries. We believe that migration is eschewed in the design of multiprocessor real-time systems because its true cost in terms of the final system produced is not well understood. As a step towards understanding this cost, we present a new taxonomy that ranks scheduling schemes along the following two dimensions:

1. **The complexity of the priority scheme.** Along this dimension, scheduling disciplines are categorized according to whether task priorities are **(i)** static, **(ii)** dynamic but fixed within a job, or **(iii)** fully dynamic. Common examples of each type include (i) RM [17], (ii) EDF [17], and (iii) least-laxity-first (LLF) [20] scheduling.

2. **The degree of migration allowed.** Along this dimension, disciplines are ranked as follows: **(i)** no migration (*i.e.*, task partitioning), **(ii)** migration allowed, but only at job boundaries (*i.e.*, dynamic partitioning at the job level), and **(iii)** unrestricted migration (*i.e.*, jobs are also allowed to migrate).

Because scheduling algorithms typically execute upon the same processor(s) as the task system being scheduled, it is important for such algorithms to be relatively simple and efficient. Most known real-time scheduling algorithms are work-conserving (see below) and operate as follows: at each instant, a *priority* is associated with each active job, and the highest-priority jobs that are eligible to execute are selected for execution upon the available processors. (A job is said to be *active* at time instant t in a given schedule if **(i)** it has arrived at or prior to time t; **(ii)** its deadline occurs after time t; and **(iii)** it has not yet completed execution.) In *work-conserving* algorithms, a processor is never left idle while an active job exists (unless migration constraints prevent the task from executing on the idle processor). Because the runtime overheads of such algorithms tend to be less than those of non-work-conserving algorithms, scheduling algorithms that make scheduling decisions on-line tend to be work-conserving. In this chapter, we limit our attention to work-conserving algorithms for this reason.[1]

To alleviate the runtime overhead associated with job scheduling (*e.g.*, the time required to compute job priorities, to preempt executing jobs, to migrate jobs, *etc.*), designers can place constraints upon the manner in which priorities are determined and on the amount of task migration. However, the impact of these restrictions on the schedulability of the system must also be considered. Hence, the effectiveness of a scheduling algorithm depends on not only its runtime overhead, but also its ability to schedule feasible task systems.

The primary motivation of this work is to provide a better understanding of the trade-offs involved when restricting the form of a system's scheduling algorithm. If an algorithm is to be restricted in one or both of the above-mentioned dimensions for the sake of reducing runtime overhead, then it would be helpful to know the impact of the restrictions on the schedulability of the task system. Such knowledge would serve as a guide to system designers for selecting an appropriate scheduling algorithm.

Overview. The rest of this chapter is organized as follows. Section 2 describes our taxonomy and some scheduling approaches based on this taxonomy. In Section 3, we compare the various classes of scheduling algorithms in the taxonomy. Section 4 presents new and known scheduling algorithms and feasibility tests for each of the defined categories. Section 5 summarizes our results.

[1]Pfair scheduling algorithms, mentioned earlier, that meet the Pfairness constraint as originally defined [5] are not work-conserving. However, work-conserving variants of these algorithms have been devised in recent work [1, 24].

2 Taxonomy of Scheduling Algorithms

In this section, we define our classification scheme. We assume that job preemption is permitted. We classify scheduling algorithms into three categories based upon the available degree of interprocessor migration. We also distinguish among three different categories of algorithms based upon the freedom with which priorities may be assigned. These two axes of classification are orthogonal to one another in the sense that restricting an algorithm along one axis does not restrict freedom along the other. Thus, there are $3 \times 3 = 9$ different classes of scheduling algorithms in this taxonomy.

Migration-based classification. Interprocessor migration has traditionally been forbidden in real-time systems for the following reasons:

- In many systems, the cost associated with each migration — *i.e.*, the cost of transferring a job's context from one processor to another — can be prohibitive.
- Until recently, traditional real-time scheduling theory lacked the techniques, tools, and results to permit a detailed analysis of systems that allow migration. Hence, partitioning has been the preferred approach due largely to the non-existence of viable alternative approaches.

Recent developments in computer architecture, including single-chip multiprocessors and very fast interconnection networks over small areas, have resulted in the first of these concerns becoming less of an issue. Thus, system designers need no longer rule out interprocessor migration solely due to implementation considerations, especially in tightly-coupled systems. (However, it may still be desirable to strict overhead in order to reduce runtime overhead.) In addition, results of recent experiments demonstrate that scheduling algorithms that allow migration are competitive in terms of schedulability with those that do not migrate, even after incorporating migration overheads [26]. This is due to the fact that systems exist that can be successfully scheduled only if interprocessor migration is allowed (refer to Lemmas 3 and 4 in Section 3).

In differentiating among multiprocessor scheduling algorithms according to the degree of migration allowed, we consider the following three categories:

1: No migration (partitioned) – In partitioned scheduling algorithms, the set of tasks is partitioned into as many disjoint subsets as there are processors available, and each such subset is associated with a unique processor. All jobs generated by the tasks in a subset must execute only upon the corresponding processor.

2: Restricted migration – In this category of scheduling algorithms, each job must execute entirely upon a single processor. However, different jobs of the same task may execute upon different processors. Thus, the runtime context of each job needs to be maintained upon only one processor; however, the task-level context may be migrated.

3: Full migration – No restrictions are placed upon interprocessor migration.

Priority-based classification. In differentiating among scheduling algorithms according to the complexity of the priority scheme, we again consider three categories.

1: Static priorities – A unique priority is associated with each task, and all jobs generated by a task have the priority associated with that task. Thus, if task T_1 has higher priority than task T_2, then whenever both have active jobs, T_1's job will have priority over T_2's job. An example of a scheduling algorithm in this class is the RM algorithm [17].

2: Job-level dynamic priorities – For every pair of jobs J_i and J_j, if J_i has higher priority than J_j at some instant in time, then J_i *always* has higher priority than J_j. An example of a scheduling algorithm that is in this class, but not the previous class, is EDF [10, 17].

3: Unrestricted dynamic priorities – No restrictions are placed on the priorities that may be assigned to jobs, and the relative priority of two jobs may change at any time. An example scheduling algorithm that is in this class, but not the previous two classes, is the LLF algorithm [20].

By definition, unrestricted dynamic-priority algorithms are a generalization of job-level dynamic-priority algorithms, which are in turn a generalization of static-priority algorithms. In uniprocessor scheduling, the distinction between job-level and unrestricted dynamic-priority algorithms is rarely emphasized because EDF, a job-level dynamic-priority algorithm, is optimal [17]. In the *multiprocessor* case, however, unrestricted dynamic-priority scheduling algorithms are strictly more powerful than job-level dynamic-priority algorithms, as we will see shortly.

By considering all pairs of restrictions on migrations and priorities, we can divide the design space into $3 \times 3 = 9$ classes of scheduling algorithms. Before discussing these nine classes further, we introduce some convenient notation.

Definition 1 *A scheduling algorithm is* $\boldsymbol{(x, y)}$**-restricted** *for* $x \in \{1, 2, 3\}$ *and* $y \in \{1, 2, 3\}$*, if it is in priority class* x *and migration class* y (*here,* x *and* y *correspond to the labels defined above*).

For example, a $(2, 1)$-restricted algorithm uses job-level dynamic priorities (*i.e.*, level-2 priorities) and partitioning (*i.e.*, level-1 migration), while a $(1, 3)$-restricted algorithm uses only static priorities (*i.e.*, level-1 priorities) but allows unrestricted migration (*i.e.*, level-3 migration). The nine categories of scheduling algorithms are summarized in Table 1. It is natural to associate classes of scheduling algorithms with the sets of task systems that they can schedule.

Definition 2 *An ordered pair denoted* $\boldsymbol{\langle x, y\rangle}$ *denotes the set of task systems are feasible under* (x, y)*-restricted scheduling.*

3: full migration	$(1,3)$-restricted	$(2,3)$-restricted	$(3,3)$-restricted
2: restricted migration	$(1,2)$-restricted	$(2,2)$-restricted	$(3,2)$-restricted
1: partitioned	$(1,1)$-restricted	$(2,1)$-restricted	$(3,1)$-restricted
	1: static	**2: job-level dynamic**	**3: unrestricted dynamic**

Table 1: A classification of algorithms for scheduling periodic task systems upon multiprocessor platforms. Priority-assignment constraints are on the x-axis, and migration constraints are on the y-axis. In general, increasing distance from the origin may imply greater generality.

Of these nine classes, $(1,1)$-, $(2,1)$-, and $(3,3)$-restricted algorithms have received the most attention. For example, $(1,1)$-restricted algorithms have been studied in [7, 11, 21, 22], while $(2,1)$-restricted algorithms (and equivalently, $(3,1)$-restricted algorithms) have been studied in [8, 9, 11]. The class of $(3,3)$-restricted algorithms has been studied in [1, 5, 16, 24]. In addition to these, $(1,3)$- and $(2,3)$-restricted algorithms were recently considered in [4] and [25], respectively.

3 Schedulability Relationships

We now consider the problem of establishing relationships among the various classes of scheduling algorithms in Table 1. (Later, in Section 4, we explore the design of efficient algorithms in each class and present corresponding feasibility results.)

As stated in Section 1, our goal is to study the trade-offs involved in using a particular class of scheduling algorithms. It is generally true that the runtime overhead is higher for more-general models than for less-general ones: the runtime overhead of a (w,x)-restricted algorithm is at most that of a (y,z)-restricted algorithm if $y \geq w \wedge z \geq x$. However, in terms of schedulability, the relationships are not as straightforward. There are three possible relationships between (w,x)- and

(y, z)-restricted scheduling classes, which we elaborate below. It is often the case that we discover some partial understanding of a relationship in one of the following two forms: $\langle w, x\rangle \subseteq \langle y, z\rangle$ and $\langle w, x\rangle \not\subseteq \langle y, z\rangle$, meaning "any task system in $\langle w, x\rangle$ is also in $\langle y, z\rangle$" and "there exists a task system that is in $\langle w, x\rangle$ but not in $\langle y, z\rangle$," respectively.

- The class of (w, x)-restricted algorithms is *strictly more powerful* than the class of (y, z)-restricted algorithms. That is, any task system that is feasible under the (y, z)-restricted class is also feasible under the (w, x)-restricted class. Further, there exists at least one task system that is feasible under the (w, x)-restricted class but *not* under the (y, z)-restricted class. Formally, $\langle y, z\rangle \subset \langle w, x\rangle$ (where $\subset$ means proper subset). Of course, $\langle y, z\rangle \subset \langle w, x\rangle$ is shown by proving that $\langle y, z\rangle \subseteq \langle w, x\rangle \wedge \langle w, x\rangle \not\subseteq \langle y, z\rangle$.

- The class of (w, x)-restricted algorithms and the class of (y, z)-restricted algorithms are *equivalent*. That is, a task system is feasible under the (w, x)-restricted class if and only if it is feasible under the (y, z)-restricted class. Formally, $\langle w, x\rangle = \langle y, z\rangle$, which is shown by proving that $\langle w, x\rangle \subseteq \langle y, z\rangle \wedge \langle y, z\rangle \subseteq \langle w, x\rangle$.

- The class of (w, x)-restricted algorithms and the class of (y, z)-restricted algorithms are *incomparable*. That is, there exists at least one task system that is feasible under the (w, x)-restricted class but not under the (y, z)-restricted class, and *vice versa*. Formally, $\langle w, x\rangle \otimes \langle y, z\rangle$, which is defined as $\langle w, x\rangle \not\subseteq \langle y, z\rangle \wedge \langle y, z\rangle \not\subseteq \langle w, x\rangle$.

These potential relationships are summarized in Table 2.

Among the nine classes of scheduling algorithms identified in Table 1, it is intuitively clear (and borne out by formal analysis) that the class of $(3, 3)$-restricted algorithms is the most general in the sense that any task system that is feasible under the (x, y)-restricted class is also feasible under

Notation	**Semantically**	**Proof Obligation**
$\langle w, x\rangle = \langle y, z\rangle$	(w, x)- and (y, z)-restricted classes are equivalent	$\langle w, x\rangle \subseteq \langle y, z\rangle \wedge$ $\langle y, z\rangle \subseteq (w, x)$
$\langle w, x\rangle \otimes \langle y, z\rangle$	(w, x)- and (y, z)-restricted classes are incomparable	$\langle w, x\rangle \not\subseteq \langle y, z\rangle \wedge$ $\langle y, z\rangle \not\subseteq \langle w, x\rangle$
$\langle w, x\rangle \subset \langle y, z\rangle$	(y, z)-restricted class dominates (w, x)-restricted class	$\langle w, x\rangle \subseteq \langle y, z\rangle \wedge$ $\langle y, z\rangle \not\subseteq \langle w, x\rangle$

Table 2: Possible relationships between the (w, x)- and (y, z)-restricted algorithm classes.

the $(3, 3)$-restricted class, for all x and y. Unfortunately, the runtime overhead of $(3, 3)$-restricted algorithms may prove unacceptably high for some applications, in terms of runtime complexity, preemption frequency, and migration frequency.

At first glance, it may seem that the class of $(1, 1)$-restricted algorithms is the least general class, in the sense that any task system that is feasible under the $(1, 1)$-restricted class is also feasible under the (x, y)-restricted class, for all x and y. However, Leung and Whitehead [15] have shown that $\langle 1, 1\rangle \otimes \langle 1, 3\rangle$. We have discovered that several other class pairs are similarly incomparable.

Some class relations are easily derived: since every static-priority algorithm is, by definition, a job-level dynamic-priority algorithm, and every job-level dynamic-priority algorithm is an unrestricted dynamic-priority algorithm, Theorem 1 (shown below) trivially holds.

Theorem 1 *The following relationships hold across the rows of Table 1.*

- $\langle 1, 1\rangle \subseteq \langle 2, 1\rangle \subseteq \langle 3, 1\rangle$
- $\langle 1, 2\rangle \subseteq \langle 2, 2\rangle \subseteq \langle 3, 2\rangle$
- $\langle 1, 3\rangle \subseteq \langle 2, 3\rangle \subseteq \langle 3, 3\rangle$

Similarly, as stated in the previous section, the optimality of EDF (a job-level dynamic algorithm) on uniprocessors implies the following relationship.

Theorem 2 $\langle 2,1\rangle = \langle 3,1\rangle$.

However, some of the relationships are not quite that straightforward to decipher, as the result of Leung and Whitehead [15] mentioned above and formally stated below in Theorem 3 shows.

Theorem 3 *The* $(1,1)$*-restricted and* $(1,3)$*-restricted classes are incomparable, i.e.,* $\langle 1,1\rangle \otimes \langle 1,3\rangle$.

Below is a list of task systems that will be used to further separate the algorithm classes. Each task is written as an ordered pair $(e,\ p)$, where e is its execution requirement and p is its period. The number of available processors is denoted by M.

$\mathcal{A}$**:** T_1 = (1,2), T_2 = (2,3), T_3 = (2,3); M=2

$\mathcal{B}$**:** T_1 = (2,3), T_2 = (2,3), T_3 = (2,3); M=2

$\mathcal{C}$**:** T_1 = (12,12), T_2 = (2,4), T_3 = (3,6); M=2

$\mathcal{D}$**:** T_1 = (3,6), T_2 = (3,6), T_3 = (6,7); M=2

$\mathcal{E}$**:** T_1 = (3,4), T_2 = (5,7), T_3 = (3,7); M=2

$\mathcal{F}$**:** T_1 = (4,6), T_2 = (7,12), T_3 = (4,12), T_4 = (10,24); M=2

$\mathcal{G}$**:** T_1 = (7,8), T_2 = (10,12), T_3 = (6,24); M=2

$\mathcal{H}$**:** T_1 = (4,6), T_2 = (4,6), T_3 = (2,3); M=2

$\mathcal{I}$**:** T_1 = (2,3), T_2 = (3,4), T_3 = (5,15), T_4 = (5,20); M=2

In several of the following lemmas, we make implicit use of Theorem 1 to establish certain results. (For instance, in Lemma 1 below, the implications in the statement of the Lemma follow directly from Theorem 1.)

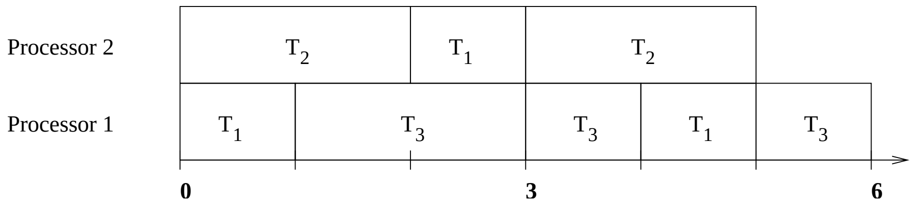


Figure 1: A $(1, 2)$-restricted schedule for task system $\mathcal{A}$.

Lemma 1 *$\mathcal{A} \in \langle 1, 2\rangle$ (which implies that $\mathcal{A} \in \langle 2, 2\rangle$ and $\mathcal{A} \in \langle 3, 2\rangle$).*

Proof: Consider the following $(1, 2)$-restricted algorithm: T_2 has higher priority than T_1, which has higher priority than T_3. Restricted migration is permitted, *i.e.*, each job must execute on only one processor, but different jobs may execute upon different processors. The resulting schedule, depicted in Figure 1, shows that $\mathcal{A} \in \langle 1, 2\rangle$. (Only the schedule in $[0, 6)$ is shown since 6 is the least common multiple (LCM) of all the task periods, and the schedule starts repeating after time 6.) ■

Lemma 2 *$\mathcal{A} \notin \langle 1, 1\rangle$ and $\mathcal{A} \notin \langle 2, 1\rangle$ and $\mathcal{A} \notin \langle 3, 1\rangle$.*

Proof: The tasks cannot be divided into two sets so that each set has utilization at most one. ■

Lemma 3 *$\mathcal{B} \in \langle 3, 3\rangle$.*

Proof: Figure 2 depicts a $(3, 3)$-restricted schedule. In this schedule, T_1 and T_2 execute over the interval $[0, 1)$, T_1 and T_3 execute over the interval $[1, 2)$, and T_2 and T_3 execute over the interval $[2, 3)$. Thus, over the interval $[0, 3)$, each task receives two units of processor time. ■

We now prove that task system $\mathcal{B}$ is only feasible under the $(3, 3)$-restricted class.

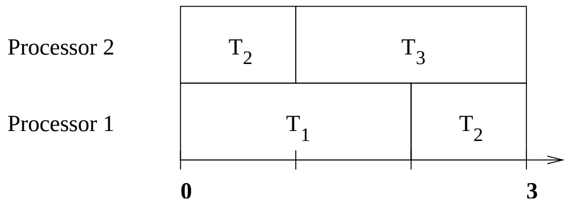


Figure 2: A $(3, 3)$-restricted schedule for task system $\mathcal{B}$.

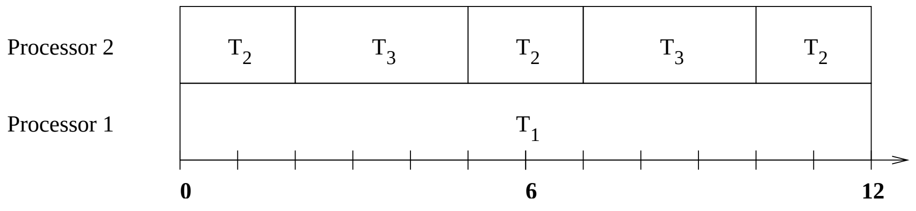


Figure 3: A $(2,1)$-restricted schedule (which is also $(2,2)$- and $(2,3)$-restricted) for task system $\mathcal{C}$.

Lemma 4 $((x \neq 3) \vee (y \neq 3)) \Rightarrow \mathcal{B} \notin \langle x, y \rangle$.

Proof: Consider the first job of each task in $\mathcal{B}$. If $((x \neq 3) \vee (y \neq 3))$, then either these jobs cannot migrate or their relative prioritization is fixed. Note that all three jobs are released at time 0 and must finish by time 3. If these jobs are not allowed to migrate, then two jobs must completely execute on one processor, which is not possible since each requires two units of processor time. Similarly, if the prioritization is fixed, then the lowest-priority job cannot start execution before time 2 and hence will miss its deadline at time 3. Thus, $\mathcal{B} \notin \langle x, y \rangle$. ■

Lemma 5 *$\mathcal{C} \in \langle 2, 1 \rangle$ (which implies that $\mathcal{C} \in \langle 3, 1 \rangle$), $\mathcal{C} \in \langle 2, 2 \rangle$ (which implies that $\mathcal{C} \in \langle 3, 2 \rangle$), and $\mathcal{C} \in \langle 2, 3 \rangle$ (which implies that $\mathcal{C} \in \langle 3, 3 \rangle$).*

Proof: The following algorithm correctly schedules $\mathcal{C}$: all jobs of task T_1 are given the highest priority, while jobs of T_2 and T_3 are prioritized using an EDF policy. Note that this is a job-level dynamic priority algorithm.

Since T_1 has a utilization of 1, it will execute solely on one of the processors. Thus, this algorithm will produce the same schedule as if the tasks were partitioned into the sets $\{T_1\}$ and $\{T_2, T_3\}$. Further, correctness follows from the optimality of EDF on uniprocessors: T_2 and T_3 can be correctly scheduled by EDF on a uniprocessor. Figure 3 shows the resulting schedule. ■

Lemma 6 *$\mathcal{C} \notin \langle 1, 1 \rangle$, $\mathcal{C} \notin \langle 1, 2 \rangle$, and $\mathcal{C} \notin \langle 1, 3 \rangle$*

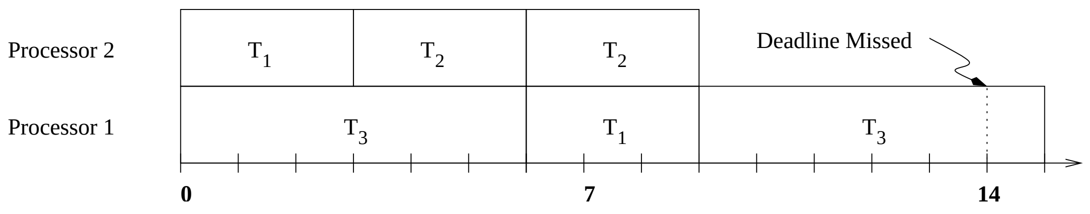


Figure 4: Deadline miss in a $(3, 2)$-restricted schedule for task system $\mathcal{D}$.

Proof: This task set is not feasible when using static priorities because no static-priority scheme can schedule the task system comprised of T_2 and T_3 upon a single processor. Clearly T_1 must be executed solely on one processor. Regardless of how T_2 and T_1 are statically prioritized, the lowest priority task will miss a deadline. ■

Lemma 7 $\mathcal{D} \in \langle 1, 1\rangle$ *(which implies that* $\mathcal{D} \in \langle 2, 1\rangle$ *and* $\mathcal{D} \in \langle 3, 1\rangle$*).*

Proof: The following algorithm correctly schedules $\mathcal{D}$: partition the tasks such that T_1 and T_2 are scheduled on one processor with T_1 getting higher priority, and T_3 is scheduled on the second processor. It is easy to see that all three tasks meet their deadlines. ■

Lemma 8 $\mathcal{D} \notin \langle 3, 2\rangle$ *(which implies that* $\mathcal{D} \notin \langle 2, 2\rangle$ *and* $\mathcal{D} \notin \langle 1, 2\rangle$*).*

Proof: Consider the three jobs that are released by $\mathcal{D}$ over the interval $[0, 6)$. Regardless of how these jobs are prioritized relative to each other, if deadlines are met, then a work-conserving algorithm will complete all three jobs by time 6. At this instant, tasks T_1 and T_2 each release a new job. *Any work conserving algorithm must now schedule* T_1*'s job on one processor and* T_2*'s job on the other processor*. Since a $(3, 2)$-restricted algorithm is not permitted to migrate jobs once they have commenced execution, these jobs must complete execution on the processors upon which they begin. When the second job of T_3 is released at time 7, it does not get scheduled until time 9. Thus, it will miss its deadline at time 14 as shown in Figure 4. ■

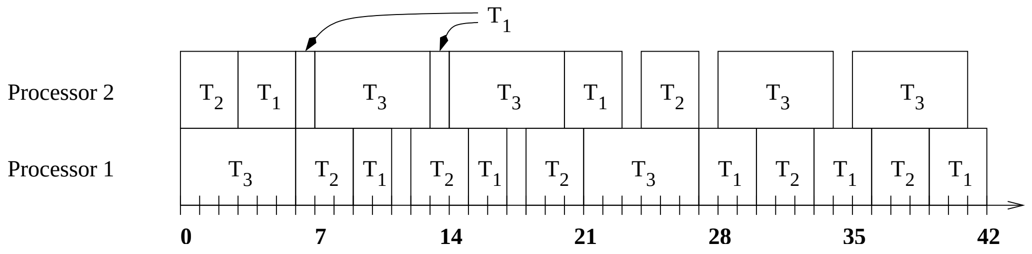


Figure 5: A $(1,3)$-restricted schedule for task system $\mathcal{D}$.

Lemma 9 $\mathcal{D} \in \langle 1,3\rangle$ (*which implies that* $\mathcal{D} \in \langle 2,3\rangle$ *and* $\mathcal{D} \in \langle 3,3\rangle$).

Proof: Consider an algorithm that assigns T_3 the highest priority, and T_1 the smallest priority. Over any interval $[6k, 6k+6)$, where k is any integer, T_3 cannot execute for more than 6 units. Since jobs may freely migrate between the processors, there are 6 consecutive units of processor time available for T_1 and T_2 to execute over every such interval; therefore, they will meet their deadlines. Figure 5 depicts the resulting schedule. ■

Lemma 10 $\mathcal{E} \in \langle 1,3\rangle$ (*which implies that* $\mathcal{E} \in \langle 2,3\rangle$ *and* $\mathcal{E} \in \langle 3,3\rangle$).

Proof: As shown in Figure 6, if the priority order (highest to lowest) is T_1, T_2, T_3, then all jobs complete by their deadlines when full migration is permitted. ■

Lemma 11 $\mathcal{E} \notin \langle 1,2\rangle$.

Proof: If only restricted migration is allowed, it may be verified that for each of the $3! = 6$

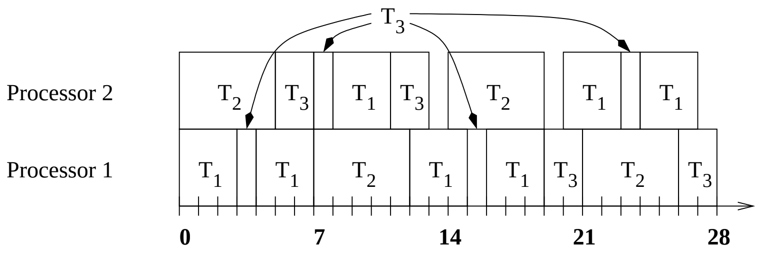


Figure 6: A $(1,3)$-restricted schedule for task system $\mathcal{E}$.

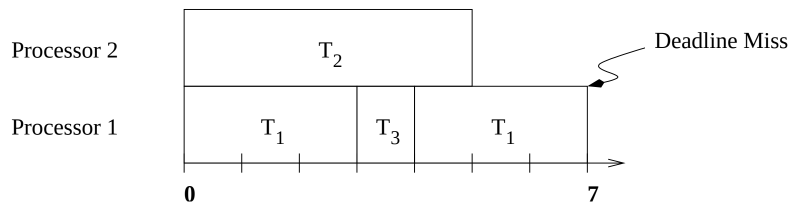


Figure 7: Deadline miss in a $(1, 2)$-restricted schedule for task system $\mathcal{E}$.

possible static-priority assignments, some task in $\mathcal{E}$ misses a deadline. Figure 7 illustrates the schedule assuming the priority order T_1, T_2, T_3. ■

The following two lemmas are somewhat counterintuitive in that they imply that *job-level migration sometimes is better than full migration* when static priorities are used.

Lemma 12 $\mathcal{F} \in \langle 1, 2\rangle$ (*which implies that* $\mathcal{F} \in \langle 2, 2\rangle$ *and* $\mathcal{F} \in \langle 3, 2\rangle$).

Proof: If the priority order (highest to lowest) is T_1, T_2, T_3, T_4 then all jobs will make their deadlines. The resulting schedule is depicted in Figure 8. A region of interest in this schedule occurs over the time interval $[7, 10)$ — since jobs cannot migrate, T_3 does not preempt T_4 during this interval despite having greater priority. This allows the job of T_4 to execute to completion. ■

Lemma 13 $\mathcal{F} \notin \langle 1, 3\rangle$.

Proof: We have verified that all $4! = 24$ possible priority assignments result in a deadline miss. Figure 9 illustrates the schedule assuming the priority order T_1, T_2, T_3, T_4. ■

Lemma 14 $\mathcal{G} \in \langle 1, 3\rangle$ (*which implies that* $\mathcal{G} \in \langle 2, 3\rangle$ *and* $\mathcal{G} \in \langle 3, 3\rangle$).

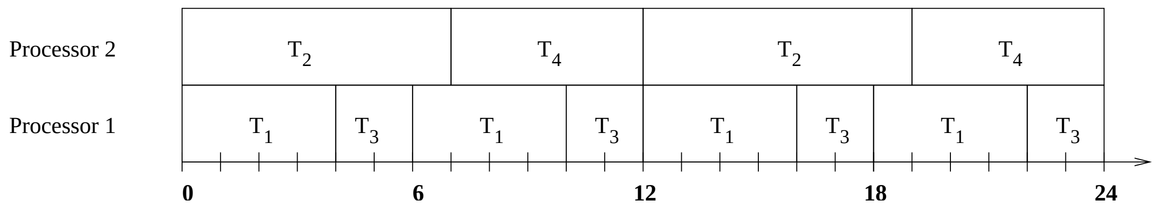


Figure 8: A (1,2)-restricted schedule for task system $\mathcal{F}$.

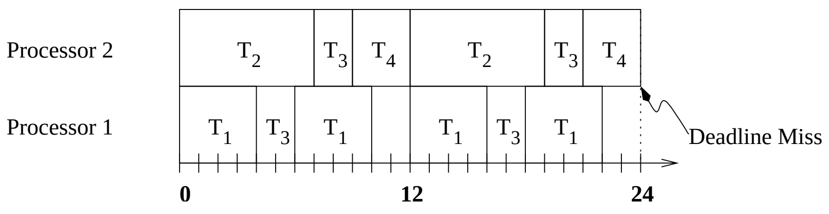


Figure 9: Deadline miss in a $(1, 3)$-restricted schedule for task system $\mathcal{F}$.

Proof: If the priority order (highest to lowest) is T_1, T_2, T_3, then all jobs will make their deadlines if full migration is allowed. Figure 10 shows the resulting schedule. ■

Lemma 15 $\mathcal{G} \notin \langle 3, 2 \rangle$ *(which implies that* $\mathcal{G} \notin \langle 2, 2 \rangle$ *and* $\mathcal{G} \notin \langle 1, 2 \rangle$*).*

Proof: Over the interval $[0, 24)$, the tasks in $\mathcal{G}$ release 6 jobs with execution requirements of 7, 7, 7, 10, 10, and 6, respectively. Since jobs cannot migrate, in order to complete all jobs before time 24, these jobs must be *partitioned* into two groups such that the sum of the execution requirements in each group does not exceed 24. The only such partition is into 7, 7, 10 and 6, 7, 10.

Consider the processor that must run jobs from the first group, which have execution requirements 7, 7, and 10, respectively. The job with execution requirement 10 executes either over the interval $[0, 12)$ or over $[12, 24)$. If the interval is $[12, 24)$, then only 7 units over the interval $[0, 8)$ can be utilized, since both jobs with execution requirement 7 belong to task T_1. Therefore, the processor must be idle for one slot, implying that one of the other two jobs misses its deadline. This is illustrated in Figure 11(a). On the other hand, if the interval is $[0, 12)$, then a job with

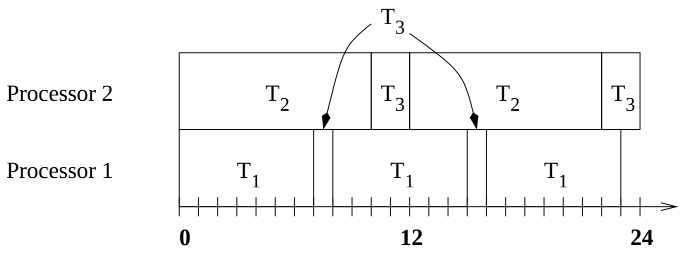
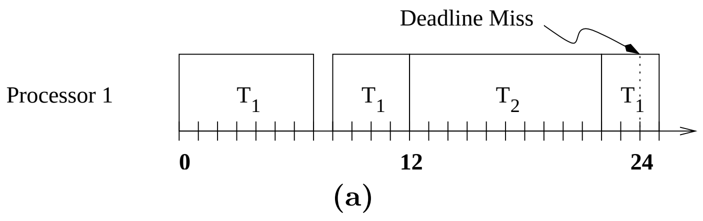


Figure 10: A $(1, 3)$-restricted schedule for task system $\mathcal{G}$.



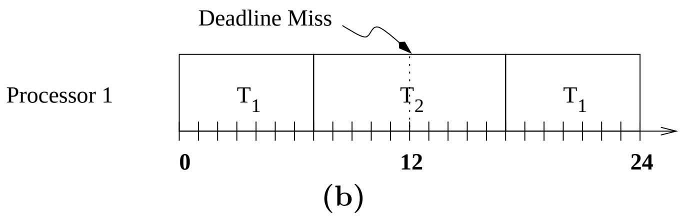


Figure 11: Deadline miss in a $(3, 2)$-restricted schedule for task system $\mathcal{G}$.

execution requirement 7 must execute in either $[0, 8)$ or $[8, 16)$. Regardless of which, the demand over $[0, 16)$ is 17, and thus, a deadline is missed at time 12 or 16, as illustrated in Figure 11(b). ■

Lemma 16 $\mathcal{H} \notin \langle 3, 1\rangle$ (*which implies that* $\mathcal{H} \notin \langle 2, 1\rangle$ *and* $\mathcal{H} \notin \langle 1, 1\rangle$).

Proof: $\mathcal{H}$ cannot be partitioned into two sets, each of which has utilization at most one. ■

Lemma 17 $\mathcal{H} \in \langle 3, 2\rangle$ *and* $\mathcal{H} \in \langle 2, 3\rangle$.

Proof: The schedule in Figure 12 shows that $\mathcal{H} \in \langle 3, 2\rangle$. This schedule can also be produced by the following priority order: T_1's job, T_3's first job, T_2's job, T_3's second job. Hence, $\mathcal{H} \in \langle 2, 3\rangle$. ■

Lemma 18 $\mathcal{I} \notin \langle 2, 3\rangle$.

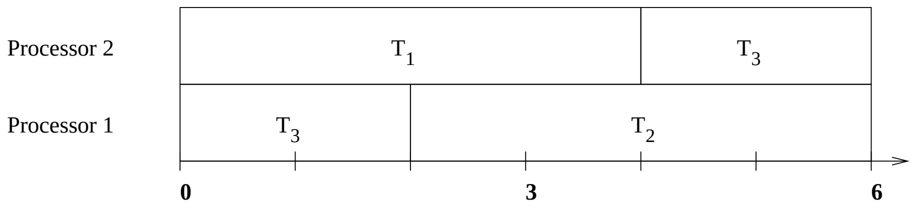


Figure 12: A (3,2)-restricted schedule for task system $\mathcal{H}$.

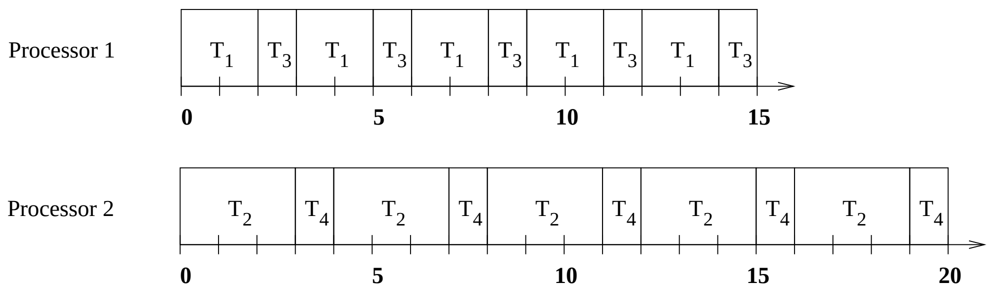


Figure 13: A $(1,1)$-restricted schedule for task system $\mathcal{I}$.

Proof: In the interval $[0,12)$, there are nine jobs released and hence, there are 9! possible priority assignments. We have verified through simulation that regardless of which priority assignment is chosen, either **(i)** a deadline is missed in $[0,12)$, or **(ii)** a processor is idle for at least one time unit over $[0,12)$. Since the total utilization equals the number of processors, (ii) implies that a deadline must be missed at some time instant after 12. ■

Lemma 19 $\mathcal{I} \in \langle 1,1\rangle$ (*which implies that* $\mathcal{I} \in \langle 2,1\rangle$ *and* $\mathcal{I} \in \langle 3,1\rangle$).

Proof: The following algorithm correctly schedules $\mathcal{I}$: partition the tasks into the sets $\{T_1, T_3\}$ and $\{T_2, T_4\}$. Each set can be correctly scheduled using the RM algorithm on a single processor as shown in Figure 13. Figure 13 shows the schedules on each processor up to the LCM of the periods of all the tasks assigned to that processor. ■

Lemma 20 $\mathcal{I} \in \langle 3,2\rangle$.

Proof: Consider the schedule suggested in the proof of Lemma 19. Such a schedule can be accomplished by an algorithm in $\langle 3,2\rangle$ simply by setting the appropriate jobs to the highest priority at each instant. (Note that this idea cannot be applied to the proof of Lemma 8 because it works only when there is no idle time in the schedule.) ■

Using the above lemmas, it is easy to derive many of the relationships among the classes of scheduling algorithms. These relationships are summarized in Table 3.

	$\langle 1,1\rangle$	$\langle 2,1\rangle$	$\langle 3,1\rangle$	$\langle 1,2\rangle$	$\langle 2,2\rangle$	$\langle 3,2\rangle$	$\langle 1,3\rangle$	$\langle 2,3\rangle$	$\langle 3,3\rangle$
$\langle 1,1\rangle$	$=$	$\subset$	$\subset$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\subset$
$\langle 2,1\rangle$	$\supset$	$=$	$=$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\subset$
$\langle 3,1\rangle$	$\supset$	$=$	$=$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\subset$
$\langle 1,2\rangle$	$\otimes$	$\otimes$	$\otimes$	$=$	$\subset$	$\subset$	$\otimes$	??	$\subset$
$\langle 2,2\rangle$	$\otimes$	$\otimes$	$\otimes$	$\supset$	$=$	$\subseteq$	$\otimes$	??	$\subset$
$\langle 3,2\rangle$	$\otimes$	$\otimes$	$\otimes$	$\supset$	$\supseteq$	$=$	$\otimes$	$\otimes$	$\subset$
$\langle 1,3\rangle$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$=$	$\subset$	$\subset$
$\langle 2,3\rangle$	$\otimes$	$\otimes$	$\otimes$	??	??	$\otimes$	$\supset$	$=$	$\subset$
$\langle 3,3\rangle$	$\supset$	$\supset$	$\supset$	$\supset$	$\supset$	$\supset$	$\supset$	$\supset$	$=$

Table 3: Relationships among the various classes.

Discussion. It is easy to see from Table 3 that several of the scheduling classes are incomparable. In particular, we can observe the following.

Observation 1 *Under static priorities, all three migration classes are incomparable.* ■

As can be seen from the table, there are two relationships that are still open.

4 Algorithm Design and Feasibility Analysis

In this section, we discuss feasibility analysis and on-line scheduling of periodic task systems. We will use the following notation: $\tau = \{T_1, T_2, \ldots, T_n\}$ denotes the system of periodic tasks to be scheduled on M processors. Let $U(T) = T.e/T.p$, where $T.e$ denotes T's execution requirement and $T.p$ denotes its period. Also, let $U(\tau) = \sum_{T \in \tau} U(T)$.

An (x, y)-restricted feasibility test accepts as input the specifications of τ and M, and determines whether some (x, y)-restricted algorithm can successfully schedule τ upon M processors. Such a test is *sufficient* if any task system satisfying it is guaranteed to be successfully scheduled by some (x, y)-restricted algorithm, and *exact* if it is both sufficient and *necessary* — *i.e.*, no task system failing the test can be scheduled by any (x, y)-restricted algorithm. Feasibility tests are often stated as utilization bounds; such a bound is the largest value $U_{\max}$ such that every task system τ satisfying $U(\tau) \le U_{\max}$ is guaranteed to be feasible.

For several of the classes identified in Table 1, exact feasibility analysis is provably intractable: a transformation from the 3-partition problem can be used to show that feasibility analysis is NP-hard in the strong sense. The class of $(3, 3)$-restricted algorithms is the most general class defined in Section 2. The following result is well-known, and immediately yields an efficient and exact $(3, 3)$-restricted feasibility test.

Theorem 4 *A periodic task system τ can be scheduled upon M processors using some $(3, 3)$-restricted algorithm if and only if $U(\tau) \le M$.*

$(3, 3)$-restricted scheduling algorithms have been the subject of several papers [1, 2, 3, 5, 12, 14]. For example, Leung [14] studied the use of global LLF scheduling on multiprocessors. Since LLF adopts a processor-sharing approach, ensuring that at most one task executes upon a processor at each instant in time may introduce an arbitrarily large number of preemptions and migrations. The Pfair scheduling approach, introduced by Baruah *et al.* [5] and extended by Anderson *et al.* [1, 2, 3, 12], reduces the number of preemptions and migrations by scheduling for discrete time units or "quanta." To summarize this work, the current state of the art concerning $(3, 3)$-restricted scheduling is as follows: there is no schedulability penalty (Theorem 4) and efficient runtime implementations are known, but the number of preemptions and interprocessor migrations may be high. Pfair scheduling is discussed in detail in the chapter titled "Fair Scheduling of Real-time Tasks on Multiprocessors" in this volume. We now briefly describe results relating to the remaining classes of scheduling algorithms.

4.1 Partitioning Approaches

There has been a considerable amount of research on $(x, 1)$-restricted scheduling algorithms, *i.e.*, partitioning approaches [7, 9, 11, 18, 22, 23]. Recall that, under partitioning, each task is assigned to a processor on which it will exclusively execute. Finding an optimal assignment of tasks to processors is equivalent to a bin-packing problem, which is known to be NP-hard in the strong sense. Several polynomial-time heuristics have been proposed for solving this problem. Examples include First Fit (FF) and Best Fit (BF). In FF, each task is assigned to the first (*i.e.*, lowest-indexed) processor that can accept it (based on the feasibility test corresponding to the uniprocessor scheduling algorithm being used). On the other hand, in BF, each task is assigned to a processor that **(i)** can accept the task, and **(ii)** will have minimal remaining spare capacity after its addition.

Surprisingly, the *worst-case achievable utilization* on M processors for all of the above-mentioned heuristics (and also for an optimal partitioning algorithm) is only $(M+1)/2$, even when an optimal uniprocessor scheduling algorithm such as EDF is used. In other words, there exist task systems with utilization slightly greater than $(M + 1)/2$ that cannot be correctly scheduled by any partitioning approach. To see why, note that $M + 1$ tasks, each with execution requirement $1 + \epsilon$ and period 2, cannot be partitioned on M processors, regardless of the partitioning heuristic and the scheduling algorithm.

Theorem 5 *No partitioned-based scheduling algorithm can successfully schedule all task systems τ with $U(\tau) \leq B$ on M processors, where $B > \frac{1}{2}(M + 1)$.*

Lopez *et al.* showed that EDF with FF (or BF) can successfully schedule any task system with utilization at most $(\beta M+1)/(\beta+1)$, where $\beta=\lfloor 1/\alpha \rfloor$ and α satisfies $\alpha \geq U(T)$ for all $T \in \tau$. A (2,1)-restricted, sufficient feasibility test immediately follows.

Theorem 6 *If $U(\tau) \leq (\beta M+1)/(\beta+1)$, where $\beta=\lfloor 1/\alpha \rfloor$ and α satisfies $\alpha \geq U(T)$ for all $T \in \tau$, then τ is feasible on M processors under the $(2,1)$-restricted class.*

We obtain the following result as a corollary of Theorem 6 by letting $\alpha = 1$ (and hence $\beta = 1$).

Corollary 1 *If $U(\tau) \leq \frac{1}{2}(M+1)$, then τ is feasible on M processors under the $(2,1)$-restricted class.*

Theorem 5 and Corollary 1 imply that using EDF with FF or BF is an optimal partitioning approach with respect to utilization bounds.

The worst-case achievable utilization is much smaller for RM-scheduled systems since RM is not an optimal uniprocessor scheduling algorithm. Let $U_{RM,FF}$ denote the worst-case achievable utilization under RM with FF (RM-FF). Oh and Baker proved the following bounds on $U_{RM,FF}$ [21]:

Theorem 7 $(\sqrt{2}-1) \times M \leq U_{RM,FF} \leq (M+1)/(1+2^{\frac{1}{M+1}})$.

Thus, task systems in which the total utilization does not exceed $(\sqrt{2}-1) \times M$ ($\approx 0.41 \times M$) are schedulable using RM-FF. Though this value is significantly small, RM is still popular because of its simplicity and predictability under overload. Several researchers have proposed partitioning heuristics that improve upon FF and BF. Oh and Son proposed an improved variant of the FF heuristic called First Fit Decreasing Utilization (FFDU) [22]. They showed that for RM-scheduled systems, the number of processors required by FFDU is at most 5/3 the optimal number of processors. (Dhall and Liu had shown earlier that the number of processors required by FF and BF is at most twice the optimal number [11].)

Burchard *et al.* proposed new sufficient feasibility tests for RM-scheduled uniprocessor systems that perform better when task periods satisfy certain relationships [7]. They also proposed new heuristics that try to assign tasks satisfying those relationships to the same processor, thus leading to better overall utilization. Lauzac *et al.* also proposed similar schedulability tests and heuristics, in which tasks are initially sorted in order of increasing periods [13]. One disadvantage of these heuristics is that they can lead to unacceptable overhead when used on-line due to the sorting overhead; when scheduling on-line, FF and BF are preferred.

4.2 Other Classes

An upper bound on the worst-case achievable utilization of any algorithm that is not $(3,3)$-restricted is easily obtained. Consider the same example used in the proof of Theorem 5: a system of $M+1$ tasks, each with a period of 2 and an execution requirement $(1+\epsilon)$, to be scheduled on M processors. Consider the set of jobs released by each task at time 0. If job migration is not allowed, then over the interval $[0,2)$, two of these jobs must be executed on the same processor, implying that one of them will miss its deadline. Further, this task system cannot be scheduled by a $(1,3)$- or $(2,3)$-restricted algorithm because when the $M+1$ jobs are released at time 0, the lowest-priority job will miss its deadline. As $\epsilon \to 0$, the utilization approaches $(M+1)/2$. Thus, by Theorem 5, we have the following.

Theorem 8 *Unless $x = 3$ and $y = 3$, no (x,y)-restricted algorithm can successfully schedule all task systems τ with $U(\tau) \leq B$ on M processors, where $B > \frac{1}{2}(M+1)$.*

We now present some results involving the other classes of scheduling algorithms.

(2,3)-restricted algorithms. These algorithms associate a fixed priority with each job but permit jobs to migrate among processors arbitrarily often. (Notice that global scheduling with EDF belongs to this class.) In these algorithms, a preemption, and hence a migration, can only be caused by a job release or (another) migration. Hence, the total number of preemptions and migrations can be bounded at an amortized number of one per job by designing the scheduling algorithm appropriately. Thus, while such algorithms do incur some preemption and migration overhead, this cost can be bounded, and may be acceptable for certain applications. Furthermore, some algorithms in this category, particularly EDF, have very efficient implementations.

As stated above, EDF is a $(2, 3)$-restricted scheduling algorithm. Although EDF is a very popular algorithm in uniprocessor real-time systems, studies (*e.g.*, [14, 20]) have suggested that it tends to miss many deadlines in the multiprocessor case. Srinivasan and Baruah [25] recently presented a new $(2, 3)$-restricted algorithm based upon EDF. In their algorithm, tasks that have utilizations at least $M/(2M-1)$ are statically assigned the highest priority in the system, while the remaining tasks are prioritized on an EDF basis. They proved that this new algorithm can schedule all task systems τ with $U(\tau) \leq \frac{M^2}{2M-1}$ (which simplifies to $(\frac{M}{2} + \frac{M}{4M-2})$) upon M processors. Thus, we have the following theorem.

Theorem 9 *A task system τ is feasible under the $(2, 3)$-restricted class if $U(\tau) \leq \frac{M^2}{2M-1}$.*

(1,3)-restricted algorithms. Andersson *et al.* [4] independently developed an algorithm similar to that of Srinivasan and Baruah, but based upon RM rather than EDF: tasks that have utilizations at least $M/(3M-2)$ are statically assigned the highest priority in the system, while the remaining tasks are prioritized on a RM basis. They proved that their algorithm can schedule all task systems τ with $U(\tau) \leq \frac{M^2}{3M-2}$ upon M processors. Hence, we have the following theorem.

Theorem 10 *A task system τ is feasible under the* $(1,3)$*-restricted class if* $U(\tau) \leq \frac{M^2}{3M-2}$.

It can be shown that if task periods are harmonic, then the schedule produced by RM is a valid EDF schedule. Therefore, we obtain the following result.

Theorem 11 *A task system τ, in which all task periods are harmonic, is feasible under the* $(1,3)$*-restricted class if* $U(\tau) \leq \frac{M^2}{2M-1}$.

(2,2)-restricted algorithms. These algorithms associate a fixed priority with each job, and restrict each job to execute exclusively on a single processor; however, different jobs of the same task may execute upon different processors. Such algorithms are particularly appropriate for scheduling task systems in which each job has a considerable amount of state (as a consequence, it is not desirable to migrate a job between processors), but not much state is carried over from one job to the next. Baruah and Carpenter [6] have designed a (2,2)-restricted algorithm which successfully schedules any periodic task system τ satisfying $U(\tau) \leq M - \alpha(M-1)$, where α is as defined earlier. Hence, we have the following result.

Theorem 12 *If* $U(\tau) \leq M - \alpha(M-1)$, *where α satisfies* $\alpha \geq U(T)$ *for all* $T \in \tau$, *then τ is feasible on M processors under the* $(2,2)$*-restricted class.*

The results in this section are summarized in Table 4. The exact utilization bound for $(3,3)$-restricted algorithms follows from Theorem 4. The bounds on worst-case achievable utilization for partitioned algorithms follow from Theorems 5–7. (The bounds for $(3,1)$-restricted algorithms follow from those for $(2,1)$-restricted algorithms because $\langle 3,1\rangle = \langle 2,1\rangle$. Refer to Table 3.) The upper bounds for the rest of the classes follow from Theorem 8. The lower bounds on worst-case achievable utilization for $(1,3)$-, $(2,3)$-, and $(2,2)$-restricted algorithms follow from Theorems 10, 9,

	1: static	**2: job-level dynamic**	**3: unrestricted dynamic**
3: full migration	$\frac{M^2}{3M-2} \le U \le \frac{M+1}{2}$	$\frac{M^2}{2M-1} \le U \le \frac{M+1}{2}$	$U = M$
2: restricted migration	$U \le \frac{M+1}{2}$	$M - \alpha(M-1) \le U \le \frac{M+1}{2}$	$M - \alpha(M-1) \le U \le \frac{M+1}{2}$
1: partitioned	$(\sqrt{2}-1)M \le U \le \frac{M+1}{1+2^{\frac{1}{M+1}}}$	$U = \frac{M+1}{2}$	$U = \frac{M+1}{2}$

Table 4: Known bounds on worst-case achievable utilization (denoted U) for the different classes of scheduling algorithms.

and 12, respectively. (The lower bound for $(3,2)$-restricted algorithms follows because $\langle 2,2\rangle \subseteq \langle 3,2\rangle$. Refer to Table 3.)

5 Summary

In this chapter, we presented a new taxonomy of scheduling algorithms for scheduling preemptive real-time tasks on multiprocessors. We described some new classes of scheduling algorithms and considered the relationship of these classes to the existing well-studied classes. We also described known scheduling algorithms that fall under these classes and presented sufficient feasibility conditions for these algorithms.

References

[1] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *Proceedings of the 12th Euromicro Conference on Real-time Systems*, pages 35–43, June 2000.

[2] J. Anderson and A. Srinivasan. Pfair scheduling: Beyond periodic task systems. In *Proceedings of the 7th International Conference on Real-time Computing Systems and Applications*, pages

297–306, Dec. 2000.

[3] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. In *Proceedings of the 13th Euromicro Conference on Real-time Systems*, pages 76–85, June 2001.

[4] B. Andersson, S. Baruah, and J. Jansson. Static-priority scheduling on multiprocessors. In *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pages 193–202, December 2001.

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

[6] S. Baruah and J. Carpenter. Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. To appear in *Proceedings of the Euromicro Conference on Real-time Systems*, 2003.

[7] A. Burchard, J. Liebeherr, Y. Oh, and S. Son. Assigning real-time tasks to homogeneous multiprocessor systems. *IEEE Transactions on Computers*, 44(12):1429–1442, December 1995.

[8] S. Davari and S. Dhall. On a real-time task allocation problem. In *Proceedings of the 19th Hawaii International Conference on System Science*, February 1986.

[9] S. Davari and S. Dhall. An on-line algorithm for real-time tasks allocation. In *Proceedings of the Seventh IEEE Real-time Systems Symposium*, pages 194–200, 1986.

[10] M. Dertouzos. Control robotics: The procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[11] S. Dhall and C. Liu. On a real-time scheduling problem. *Operations Research*, 26:127–140, 1978.

[12] P. Holman and J. Anderson. Guaranteeing Pfair supertasks by reweighting. In *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pages 203–212, December 2001.

[13] S. Lauzac, R. Melhem, and D. Mosse. An efficient RMS admission control and its application to multiprocessor scheduling. In *Proceedings of the 12th International Symposium on Parallel Processing*, pages 511–518, April 1998.

[14] J. Leung. A new algorithm for scheduling periodic real-time tasks. *Algorithmica*, 4:209–219, 1989.

[15] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

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

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

[18] J. Lopez, M. Garcia, J. Diaz, and D. Garcia. Worst-case utilization bound for edf scheduling on real-time multiprocessor systems. In *Proceedings of the 12th Euromicro Conference on Real-time Systems*, pages 25–33, June 2000.

[19] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Proceedings of the 20th IEEE Real-time Systems Symposium*, pages 294–303, Dec. 1999.

[20] A. Mok. *Fundamental Design Problems of Distributed Systems for Hard Real-time Environments.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1983.

[21] D. Oh and T. Baker. Utilization bounds for n-processor rate monotone scheduling with static processor assignment. *Real-time Systems*, 15(2):183–192, 1998.

[22] Y. Oh and S. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-time Systems*, 9(3):207–239, 1995.

[23] S. Saez, J. Vila, and A. Crespo. Using exact feasibility tests for allocating real-time tasks in multiprocessor systems. In *Proceedings of the Tenth Euromicro Workshop on Real-time Systems*, pages 53–60, June 1998.

[24] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 189–198, May 2002.

[25] A. Srinivasan and S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84(2):93–98, November 2002.

[26] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. In *Proceedings of the 11th International Workshop on Parallel and Distributed Real-time Systems*, April 2003.

In the parti-tioned scheduling, each real-time task is assigned to a dedicated processor. All instances from the same task will be executed solely on that particular processor. In global scheduling, all jobs rst enter a global queue, and thus each task can be potentially executed on any processor.Â [8] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, â€œA categorization of real-time multiprocessor scheduling problems and algorithms,â€ in Handbook on Scheduling Algorithms, Methods, and Models. Chapman Hall/CRC, Boca, 2004. [9] J. Anderson, V. Bud, and U. Devi, â€œAn edf-based scheduling algorithm for multiprocessor soft real-time systems,â€ in Real-Time Systems, 2005. Real-time multiprocessor systems are now commonplace. Designs range from single-chip architectures, with a modest number of processors, to large-scale signal-processing systems, such as synthetic-aperture radar systems. For uniprocessor systems, the problem of ensuring that deadline constraints are met has been widely studied: effective scheduling algorithms that take into account the many complexities that arise in real systems (e.g., synchronization costs, system overheads, etc.) are well understood. In contrast, researchers are just beginning to understand the trade-offs that exist in multiprocessor systems. In this chapter we analyze the trade-offs involved in scheduling independent, periodic real-time tasks on a multiprocessor. Real-Time Scheduling with Task Splitting on Multiprocessors. âˆ—. Shinpei Kato and Nobuyuki Yamasaki School of Science for Open and Environmental Systems. Keio University, Yokohama, Japan {shinpei,yamasaki}@ny.ics.keio.ac.jp. Abstract. This paper presents a real-time scheduling algorithm with high schedulability and few preemptions for multipro-cessor systems. The algorithm is based on an unorthodox method called portioned scheduling that assigns each task to a particular processor like partitioned scheduling but can split a task into two processors if there is not enough ca-pacity remaining on ... A Categorization of Real-time Multiprocessor. Scheduling Problems and Algorithms. John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. Department of Computer Science, University of North Carolina at Chapel Hill.Â Real-time multiprocessor systems are now commonplace. Designs range from single-chip archi-tectures, with a modest number of processors, to large-scale signal-processing systems, such as synthetic-aperture radar systems. For uniprocessor systems, the problem of ensuring that deadline constraints are met has been widely studied: eective scheduling algorithms that take into account the many complexities that arise in real systems (e.g., synchronization costs, system overheads, etc.) are well understood. In real-time multiprocessor systems there are two main is-sues that a scheduling algorithm must tackle. Where should a task be executed(allocation problem) and when(priority problem) [6]? The allocation problem divides into three dif-ferent approaches [9]. 1. No migration. Tasks are assigned to one processor and. A taskset is called feasible if there is an algorithm witch can schedule any combination of tasks out of this taskset [9]. For example showing feasibility for implicit-deadline peri-odic tasksets is quite is quite easy.Â A categorization of real-time multiprocessor scheduling problems and algorithms. Handbook on scheduling algorithms, methods, and models, pages 30â€“1, 2004. [7] S. Collette, L. Cucu, and J. Goossens.