Equivalence between schedule representations: theory and applications

Matthieu Lemerre[1] Vincent David[1] Christophe Aussaguès[1] Guy Vidal-Naquet[2]
CEA, LIST CEA, LIST CEA, LIST SUPELEC
BP94, F91191 Gif-Sur-Yvette CEDEX 91190 Gif-Sur-Yvette

Abstract

Multiprocessor scheduling problems are hard because of the numerous constraints on valid schedules to take into account. This paper presents new schedule representations in order to overcome these difficulties, by allowing processors to be fractionally allocated. We prove that these representations are equivalent to the standard representations when preemptive scheduling is allowed. This allows the creation of scheduling algorithms and the study of feasibility in the simpler representations. We apply this method throughout the paper.

Then, we use it to provide new simple solutions to the previously solved implicit-deadline periodic scheduling problem. We also tackle the more general problem of scheduling arbitrary time-triggered tasks, and thus in particular solve the open multiprocessor general periodic tasks scheduling problem. Contrary to previous solutions like the PFair class of algorithms, the proposed solution also works when processors have different speeds.

We complete the method by providing an online schedule transformation algorithm, that allows the efficient handling of both time-triggered and event-triggered tasks, as well as the creation of online rate-based scheduling algorithms on multiprocessors.

1. Introduction

Optimally scheduling real-time tasks on multiprocessor, i.e. constructing a schedule when one exists, or studying feasibility are hard problems that have solutions only for a few particular task systems. The fact that a job cannot be executed simultaneously by more than one processor is a complicating constraint on the placement of the jobs on the different processors. For instance, this is why online optimal scheduling of arbitrary job sets is impossible [11, 8].

To cope with this problem, researchers have built relatively complex solutions to solve particular problems; for instance the PFair class of algorithms [4] for implicit-deadline periodic tasks on multiple identical processors cuts jobs at every time quanta and dispatches them on the multiple processors.

The idea behind this algorithm is that the problem is simple if "in each unit of time a task may use a fraction f of a [processor], $0 \leq f \leq 1$: [one just has to] allocate a fraction $x.w$ of a [processor] to each task $[x]$ during each time unit"[4], where $x.w$ is the utilization of x, i.e. the "fraction of processor time spent in executing task $[x]$" [16]. This kind of ideal scheduling where tasks can receive only a fraction of the computing power of a processor is called *fluid scheduling*, and the PFair algorithms try to follow closely this ideal allocation for the implicit-deadline periodic task problem.

In this paper we prove formally that *preemptive scheduling is as powerful as fluid scheduling*, i.e. *any* scheduling problem is solvable using fluid scheduling if, and only if, it is also solvable using preemptive scheduling. Moreover, this property is true not only when time is modelled in a discrete way, but also in the more general continuous model of time; and we prove that it still works for *uniform* multiprocessors, i.e. processors of the same type that have different (but constant) speeds.

We derive from this property an efficient algorithm for solving the problem of scheduling a set of implicit-deadline periodic tasks in the continuous model of time for identical multiprocessors, for which we only gave a proof sketch [14] and which was independently solved by Cho *et al.*[6]. As this algorithm also works when processors have different speed we also solve the problem of scheduling implicit-deadline periodic tasks on uniform multiprocessors.

If the implicit-deadline periodic scheduling problem can be solved using fluid scheduling where each task has a *constant* fractional allocation of the CPUs, it is not the case for more complicated task systems, such as in the general periodic scheduling problem. We allow this fractional allocation to vary in two different ways: in the general *weighted scheduling representation* the fractions can vary at any time; in the more constrained *job-boundary weighted scheduling representation*, the fractions vary only when a job arrives or reaches its deadline. We prove that these representations are *equivalent* (i.e. equally powerful) and show how the latter can be used either to prove infeasibility or to find correct *weights*, i.e. the fraction of allocation of a processor, in more complicated task systems such as the general time-

[1]email: Firstname.Lastname@cea.fr
[2]email: Guy.Vidal-Naquet@supelec.fr

1080-1812/08 $25.00 © 2008 IEEE
DOI 10.1109/RTAS.2008.17

triggered task systems, as used in OASIS [2]. In particular this is also the first solution *of the problem of optimally scheduling general periodic task sets* (that may have deadlines different than their periods). Our task model is deliberately "simple", i.e. our tasks are fully preemptive and do not use any mutex nor semaphore ("shared resources"). Such task models could be studied in further articles. This does not mean that our paper has no practical applications: by an appropriate system design, e.g. using only wait-free buffered communications primitives and wait-free system services, one can eliminate any need for synchronization primitives in a real-time application and operating system; this has been done in OASIS, which has been deployed in safety-critical nuclear systems [7]. OASIS is a real-time programming paradigm with associated tools (dedicated operating system and offline tools for specific-language compilation, memory protection, feasibility proof and buffer sizing), which uses time as the sole mean of synchronization. There are many advantages in using such a simple task model: it not only makes schedulability analysis a lot easier and allows reaching full utilization of the processors, but also it avoids unnecessary thread commutations or waits between processors, all these problems being amplified on multiprocessor computers. This makes the OASIS system well suited for SMP computers.

In the last section, we provide and prove an algorithm for online schedule transformation for identical multiprocessors, i.e. that takes a set of weights expressed in the fluid scheduling model and transforms them efficiently into a concrete multiprocessor schedule at runtime, provided the next job boundary is known. This has several uses; in particular, it allows writing scheduling algorithms in the fluid scheduling model, but also safely using free CPUs to cope with tasks not expressed in this model (e.g. sporadic tasks).

Related work

The first multiprocessor scheduling problems that were addressed were the *optimal finish time* problems, in which one tries to minimize the total computation time of a batch of jobs. It is related but distinct to real-time scheduling; in the real-time context, we can perceive optimal finish time algorithms as the scheduling algorithms to use when full utilization of the processors is required and when the jobs' start times and deadlines are all equal. This problem is well known: in 1959, McNaughton [18] gave an optimal finish time algorithm for identical multiprocessor; later Horvath *et al.* [13] provided one for uniform multiprocessor, and Gonzalez and Sahni [9] gave a more efficient one which is now the most commonly used.

Then the problem of scheduling finite job sets with start time and deadline constraints was solved by Horn [12] (in the identical multiprocessor case) and Martel [17] (in the uniform case).

For infinite job sets, Hong and Leung [11] proved that online optimal scheduling on multiprocessor of arbitrary job sets is impossible, and Dertouzos and Mok showed [8] that a job sets can be optimally scheduled only if there is an *a priori* knowledge of their given start time, deadlines and execution time before execution. This explains why research on multiprocessor scheduling of infinite job sets has been mostly focused on periodic tasks, for which all these parameters are completely known.

Most known optimal real-time scheduling algorithms for multiprocessors try to follow the *fairness* ideal property [4], in which all jobs have a constant fractional allocation of a processor equal to their respective utilization. For instance, the PFair class of algorithm for implicit-deadline periodic tasks divides time in quanta and ensures that actual processor allocation for a task is always close to the ideal allocation at each quanta [4]. Zhu *et al.* realized [20] that the fairness property must only be applied at job boundaries (the BFair condition), and thus derived a more efficient algorithm that causes less preemptions.

Also, some sufficient or necessary bounds on the utilization of tasks using non optimal scheduling algorithms have been derived, e.g. for Global-EDF [19]. These bounds stress the considerable performance increases of using an optimal scheduling algorithm over a generic online one.

These sufficient bounds also give sufficient conditions for scheduling a set of tasks using an online algorithm, when coupled with feasibility analysis. Few methods of feasibility analysis for multiprocessors exist; the only other general one we are aware of is based on demand-bound functions [3]. This is an interesting abstraction which can be viewed as an approximation of our weighted schedules conditions (see Definition 7). Unfortunately, it leads to a test which is only sufficient, meaning in practice that the test works only for weakly constrained jobs sets.

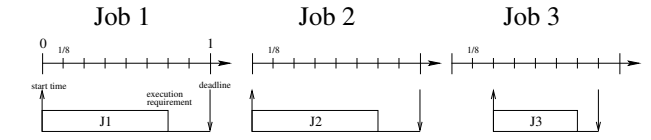
In Section 3, we show how we use our main theorem as a method for getting *exact* feasibility tests, which also give the exact maximum load of the system and allows exact CPU sizing. For that we use the method for exact feasibility analysis of time-triggered tasks for single processor presented by Aussaguès and David [2], whose extension has been the original motivation for the creation of this theory.

2. The schedule representations

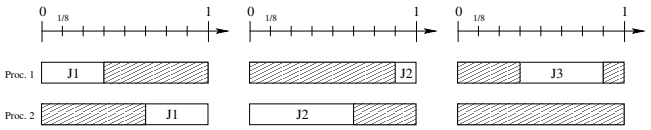
A schedule is usually represented as the mapping of a set of jobs to a set of processors; this mapping varies with time. This representation is important because it models the actual execution of the job set; this is why we name it the *concrete representation*.

However, this is not the only possible representation. For instance, a set of fixed priorities is a simple way to represent a schedule: from a set of jobs and their corresponding priorities, we can retrieve the concrete schedule. But this representation is limited, because there exists feasible concrete schedule that are not feasible in this representation [16].

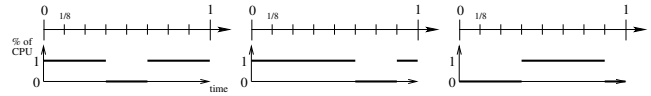
This paper focuses on three other schedule representations that are interesting because they are capable of representing *any* feasible schedule; and from any schedule in



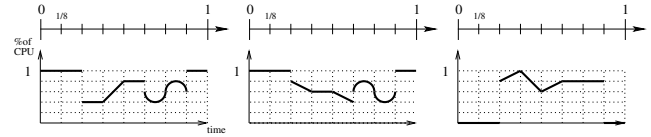
(a) Execution requirements; x axis represents time. For instance, Job 3 has to execute for $4/8$ between dates $2/8$ and $7/8$.



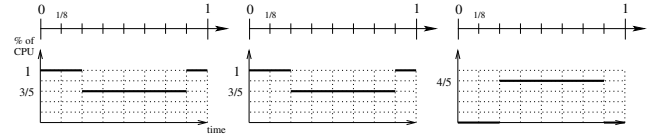
(b) Concrete representation: to each job, at each moment is associated the processor on which it runs (Proc. 1 or Proc. 2), or none.



(c) Anonymous representation: tells whether a job is elected at each moment, i.e. processor allocation is required to be 0 or 1.



(d) Weighted representation: represents the fraction of a processor allocated for a job at each moment. Weights may vary arbitrarily.



(e) Job-boundary weighted representation: the fraction of allocation may be fractional but may change only at job boundaries.

Figure 1. A job set and four correct schedules for it in the four representations. (Each row contains the complete schedule for a job set, and each column contains different equivalent schedules in different representations for the same job. The x axis represents time.)

these representations one can also extract a feasible concrete schedule. That is why these representations are said to be *equivalent*.

Instead of considering a schedule as a mapping on the processors, these three representations can be viewed as a function of time representing the rate of needed processor resource, with some constraints on the values of the function.

The *weighted representation* is the most general of them; the *job-boundary weighted representation* is a restriction of the weighted representation where weight changes can occur only at certain moments, and the *anonymous representation* is a restriction of the weighted representation where the weights must correspond to the processors speeds (fractional allocation is forbidden).

Figure 1 represents a task system and four correct equivalent schedules in the four representations.

2.1. Formal definitions

Definition of standard terms of scheduling are needed to formally specify the notion of scheduling representation:

Definition 1 (Job and job set)**.** A *job* J is characterized by three parameters:

- Its *start time* $J.s$
- Its *deadline* $J.d$, so that $J.s \leq J.d$
- Its *needed execution time*, $J.e$

A job is *active* at time t if $J.s \leq t \leq J.d$
A *job set* $\mathcal{J}$ is a countable set of jobs.

Definition 2 (Schedule representation)**.** To a given job set $\mathcal{J}$, a *schedule representation* associates two objects:

- The set of its *valid* schedules $\mathcal{S}_{\mathcal{J}}$
- An *execution function* $X_{\mathcal{J}} : \mathcal{S}_{\mathcal{J}} \times \mathcal{J} \times \mathbb{R}_+ \rightarrow \mathbb{R}_+$; such that for s and J fixed, $X_{\mathcal{J}}(s, J, t)$ is monotonically increasing.

  $X_{\mathcal{J}}(s, J, t)$ represents the time spent executing J at time t in the schedule s.

Note: When there is no ambiguity on $\mathcal{J}$, we simply note X for $X_{\mathcal{J}}$ and $\mathcal{S}$ for $\mathcal{S}_{\mathcal{J}}$.

Definition 3 (Correct execution)**.** A job J is *correctly scheduled* in a schedule s when it is scheduled for $J.e$ between $J.s$ and $J.d$, and is not scheduled otherwise; i.e. when the following conditions are fulfilled:

$$\begin{cases} \forall t \leq J.s, & X(s, J, t) = 0 \\ \forall t \geq J.d, & X(s, J, t) = J.e \end{cases}$$

A job set $\mathcal{J}$ is correctly scheduled by s when all its jobs are correctly scheduled.

A job set $\mathcal{J}$ is *feasible* in a schedule representation $(\mathcal{S}, \mathcal{X})$ when there exists a correct schedule in $\mathcal{S}_{\mathcal{J}}$.

Definition 4 (Weight function)**.** Let $J \in \mathcal{J}$ be a job and s a schedule for J. As $t \mapsto X(s, J, t)$ is monotonically increasing, it is differentiable, except on a countable set of discontinuities.

Its derivative is called the *weight function* of J and is noted $w_J(t)$.[1] w is the *set of weight functions* for s.

The intuitive notion of weight is the execution rate of a process. For a concrete execution on identical processors, $w_J(t)$ is 1 if the job is executing at time t and 0 otherwise.

[1] w_J should in fact be a distribution because of the possible discontinuity points of X. We ignore this problem because this paper only use schedule representations that have differentiable X functions.

Notes:
- For a given schedule, giving all the weight functions $w_J, J \in \mathcal{J}$ of a job set is equivalent to giving its execution function $X_{\mathcal{J}}$.
- When s is a schedule, w, the set of weight functions for s is a schedule by itself in a particular schedule representation; in this case $\mathcal{S}_{\mathcal{J}}$ is a subset of $\mathbb{R}_+^{\mathcal{J}}$ and:

$$X_{\mathcal{J}}(w, J, t) \quad = \quad \int_{\tau=0}^{t} w_J(\tau)\, \mathrm{d}\tau$$

w is called the *anonymization* of s.

The anonymization represents a loss of information, as it will be seen in the next section.

2.2. Important schedule representations

In this section we detail the four schedule representations previously introduced:

Definition 5 (Concrete schedule representation)**.** For a M-multiprocessor computer $\mathcal{C}$, a *concrete schedule* ($\mathcal{CS}$) is a function $s : \mathcal{J} \times \mathbb{R}_+ \to [\!|\, 0, M \,|\!]$.

Its meaning it that $s_J(t) = m$ if at time t, J is executing on m, and $s_J(t) = 0$ if J is not executing.

Moreover, s is *valid* iff a processor always executes at most one job, i.e.

$$\forall t, \forall m \in [\!|\, 1, M \,|\!], \quad |\{\, J : s_J(t) = m \,\}| \leq 1$$

The execution function X is defined by:

$$X(t) = \int_{\tau=0}^{t} \alpha(s_J(\tau))\, \mathrm{d}\tau$$

Where $\alpha(0) = 0$ and $\alpha(i) = \alpha_i \in \mathbb{R}_+^*$ is the speed of processor i. For identical multiprocessors, the standard convention is to take $\alpha_i = 1$.

Notes:
- Schedule representations are thus also parametrized by the CPU speed and number; by a slight abuse of notation this parametrization is made implicit.
- Another condition on s is that $\alpha \circ s$ is integrable. Given the other restrictions on s, this in fact means that s is piecewise constant.

A large part of the complexity of multiprocessor scheduling comes from this elementary definition. Checking that a schedule is valid or correct is not easy, so constructing one is even more difficult.

Definition 6 (Anonymous schedule representation)**.** An *anonymous schedule* ($\mathcal{AS}$) is the anonymization of a concrete schedule.

Notes:
- Thus it is a function $w : \mathcal{J} \times \mathbb{R}_+ \to \{\, 0, \alpha_1, \ldots, \alpha_M \,\}$.
- The validity conditions for an anonymous schedule, i.e. the conditions for a function $w : \mathcal{J} \times \mathbb{R}_+ \to \mathbb{R}_+$ to be an anonymous schedule in the general uniform processor case are complex. They are simple only in the identical multiprocessor case, in which they reduce to:

$$\begin{cases} \forall t, & \displaystyle\sum_{J \in \mathcal{J}} w_J(t) \;\leq\; M \\ \forall t, \forall J, & w_J(t) \in \{\, 0, 1 \,\} \end{cases}$$

Proof that these conditions are necessary and sufficient is simple. If given w fulfilling these conditions, it is easy to construct a concrete schedule s such that its anonymization is w: it suffices to arbitrarily choose the processors executing the elected jobs every time one of the w functions change.
- These validity conditions make the checking that an identical multiprocessor schedule is valid and correct easily computable. However, constructing one remains difficult.

This schedule representation is already much simpler to use than the standard, concrete schedule representation, and makes checking the validity and correctness of a schedule practically computable. These two representations illustrate the idea of what is *equivalence* between two representations (which is formally described in section 2.3): as there is a bijection between a schedule a in the anonymous representation and the set of concrete schedules whose anonymization is a, a job set is feasible in one representation if, and only if, it is also feasible in the other.

The next important representation is a "relaxed" version of the anonymous schedule. The following notation will be useful:

$\forall k \in \mathbb{N}$, we name (when defined) $\mathrm{Smax}_k(u)$ the sum of the k greatest elements of a countable set u, i.e. if we sort u such that $u_1 \geq u_2 \geq \cdots$, then

$$\mathrm{Smax}_k(u) = \sum_{i=1}^{k} u_i$$

Definition 7 (Weighted schedule representation)**.** A *weighted schedule* ($\mathcal{WS}$) of $\mathcal{C}$ is a function $w : \mathcal{J} \times \mathbb{R}_+ \to \mathbb{R}_+$, with the additional restrictions:

w is a valid weighted schedule iff, $\forall t$:

$$\begin{cases} \mathrm{Smax}_1(w(t)) & \leq & \mathrm{Smax}_1(\alpha) & (1) \\ \mathrm{Smax}_2(w(t)) & \leq & \mathrm{Smax}_2(\alpha) & (2) \\ \vdots & & & \vdots \\ \mathrm{Smax}_{M-1}(w(t)) & \leq & \mathrm{Smax}_{M-1}(\alpha) & (M-1) \\ \displaystyle\sum_{J \in \mathcal{J}} w_J(t) & \leq & (\alpha_1 + \ldots + \alpha_M) & (M) \end{cases}$$

In the identical multiprocessor case, these conditions reduce to:

$$\begin{cases} \forall t, & \displaystyle\sum_{J \in \mathcal{J}} w_J(t) \;\leq\; M \\ \forall t, \forall J, & w_J(t) \in [\, 0, 1 \,] \end{cases}$$

Note: We use $\mathrm{Smax}_k(w(t))$ which is not $\mathrm{Smax}_k(w)(t)$: the greatest weight functions can change at each t, as in e.g. jobs J_1 and J_2 Figure 1.

Intuitively, the first conditions state that the processors are sufficiently powerful for the most demanding jobs, while the last condition states that the overall system must be sufficiently powerful for all the jobs to run simultaneously.

Although the weighted representation is the most general, it is not convenient for numeric computation. The following one solves this problem:

Definition 8 (Job-boundary weighted schedule representation)**.** A job-boundary weighted schedule ($\mathcal{BWS}$) is a weighted schedule with the additional restriction that weights can change only at job boundaries, i.e.

$$\begin{array}{c} \forall \tau_1 < \tau_2, \forall J \in \mathcal{J}, \quad w_J(\tau_1) \neq w_J(\tau_2) \\ \Longleftrightarrow \\ \exists K \in \mathcal{J}, \quad \tau_1 \leq K.s \leq \tau_2 \quad \text{or} \quad \tau_1 \leq K.d \leq \tau_2 \end{array}$$

In the next sections, we will note $(t_n) = (t_0, t_1, \ldots)$ the ordered sequence of the start times and deadlines of all jobs.

2.3. Conversion between schedule representations and equivalence

The interest of having different "abstract" scheduling representations lies in the relations between them:

Definition 9 (Conversion function)**.** A *conversion function* between two schedule representations $\mathcal{S}_\mathcal{J}$ and $\mathcal{S'}_\mathcal{J}$ is a function $c : \mathcal{S}_\mathcal{J} \to \mathcal{S'}_\mathcal{J}$ such that if s is a correct schedule of $\mathcal{S}_\mathcal{J}$, then $c(s)$ is a correct schedule of $\mathcal{S'}_\mathcal{J}$.

Definition 10 (Equivalence between schedule representations)**.** If there exists $c : \mathcal{S}_\mathcal{J} \to \mathcal{S'}_\mathcal{J}$ and $c' : \mathcal{S'}_\mathcal{J} \to \mathcal{S}_\mathcal{J}$ two conversion functions, then $\mathcal{S}_\mathcal{J}$ and $\mathcal{S'}_\mathcal{J}$ are *equivalent*.

This relation is obviously an equivalence relation.

This notion of equivalence between schedule representations lies at the heart of our paper. Although some conversion functions between different schedule representations have already been used on some job sets (for instance in [17] for finite tasks), this is a new notion. Equivalence gives a necessary and sufficient condition of the existence of a correct schedule between schedule representations: if there is a correct schedule in $\mathcal{S}$, then there is a correct schedule in $\mathcal{S}'$; but *if there is no correct schedule in $\mathcal{S}$, then there is no correct schedule in $\mathcal{S}'$ either*. In other words:

Theorem 11. *Let $\mathcal{S}$ and $\mathcal{S}'$ be two equivalent schedule representations. A job set $\mathcal{J}$ is feasible in $\mathcal{S}$ if, and only if, it is feasible in $\mathcal{S}'$.*

Proof. Proof is simple: if $\mathcal{J}$ is feasible in $\mathcal{S}$, let s be a correct schedule and c a conversion function between $\mathcal{S}$ and $\mathcal{S}'$. $c(s)$ is a correct schedule in S', and $\mathcal{J}$ is feasible in $\mathcal{S}'$.

Similarly, if $\mathcal{J}$ is feasible in $\mathcal{S}'$ then it is feasible if $\mathcal{S}$. □

This theorem allows the notion of schedule representation to be used for feasibility analysis; because it may be simpler to prove that there is (or not) a schedule in one representation than in another one.

Lemma 12. *A schedule representation and its anonymization are equivalent.*

Proof. This is almost the definition of anonymization; the c function is the one that transforms a schedule into its anonymized schedule. A c' function is defined by taking one preimage for each anonymized schedule. □

Here comes the most important theorem of this paper:

Theorem 13. *The concrete, anonymous, weighted and job-boundary weighted schedule representations are all equivalent on a uniform multiprocessor.*

The proof is derived by finding conversion functions between all the schedule representations. As they can be composed, it is sufficient to find conversion functions between some of the representations.

Lemmas 14 and 15 give the necessary remaining conversion functions needed, then proof of the main theorem is provided.

Lemma 14. *The job-boundary weighted representation ($\mathcal{BWS}$) is equivalent to the weighted representation ($\mathcal{WS}$).*

Proof. As a job-boundary weighted schedule is a weighted schedule, the conversion function is simply the identity function in this case.

To convert a weighted schedule into a job-boundary weighted one, we convert each weight function into its mean value between two job boundaries:

Let (t_n) be the series of start times and deadlines of all the jobs in $\mathcal{J}$, i.e. between any t_n and t_{n+1} there are no new jobs nor any deadline (the set of active jobs is constant).

Let $J \in \mathcal{J}$, $t \in \mathbb{R}_+$ and n so that $t_n \leq t < t_{n+1}$.

Let

$$\langle w_J \rangle(t) \;=\; w_J^n \;=\; \frac{\displaystyle\int_{t_n}^{t_{n+1}} w_J(\tau)\,\mathrm{d}\tau}{t_{n+1} - t_n}$$

the mean value of $w_\mathcal{J}$ between t_n and t_{n+1}.

Proof that $(\langle w_J \rangle)_{J \in \mathcal{J}}$ is a correct job-boundary weighted schedule is in three steps:

1. It is a valid weighted schedule because it fulfills the validity inequalities of the weighted representation (see definition 7).

Let $n \in \mathbb{N}$; we note $\langle w_J \rangle$ the value of $\langle w_J \rangle(t)$ between t_n and t_{n+1}.

Inequality (M) is still satisfied:

$$\begin{array}{rcl} \displaystyle\sum_{J \in \mathcal{J}} w_J(t) & \leq & (\alpha_1 + \ldots + \alpha_M) \\ \Rightarrow \displaystyle\int_{t_n}^{t_{n+1}} \sum_{J \in \mathcal{J}} w_J(\tau)\,\mathrm{d}\tau & \leq & (t_{n+1} - t_n) \cdot (\alpha_1 + \ldots + \alpha_M) \end{array}$$

$$\Rightarrow^{\dagger} \sum_{J\in\mathcal{J}} \int_{t_n}^{t_{n+1}} w_J(\tau)\,\mathrm{d}\tau \quad\leq\quad (t_{n+1}-t_n)\cdot(\alpha_1+\ldots+\alpha_M)$$
$$\Rightarrow \sum_{J\in\mathcal{J}} \langle w_J\rangle \quad\leq\quad (\alpha_1+\ldots+\alpha_M)$$

$\forall i \in [\,|\,1, M-1\,|\,]$, inequality (i) are also still respected:
Indeed let $J_1, J_2, \ldots, J_M \in \mathcal{J}^M$ such that

$$\langle w_{J_1}\rangle \geq \langle w_{J_2}\rangle \geq \ldots \geq \langle w_{J_n}\rangle$$

for all t the sum of the weight of i jobs is lower than the weight of the i most demanding jobs: $\forall t$,

$$\begin{aligned} w_{J_1}(t)+\cdots+w_{J_i}(t) &\leq \mathrm{Smax}_i(w(t)) \\ &\leq \alpha_1+\cdots+\alpha_i \\ \Rightarrow \langle w_{J_1}\rangle+\cdots+\langle w_{J_n}\rangle &\leq \alpha_1+\cdots+\alpha_i \end{aligned}$$

So the new weight functions respect the weighted schedule validity conditions of definition 7.

2. It is a valid $\mathcal{BWS}$ schedule because the weights change only at job boundaries, by construction.

3. It is a correct schedule: As each job executes for the same amount of time with w or $\langle w\rangle$ in each interval $[t_n, t_{n+1}[$, we can prove by recurrence on n that the execution function has the same values on each job boundary:

$$\forall t_n, \forall J \in \mathcal{J}, \quad X(w,J,t_n) = X(\langle w\rangle, J, t_n) \qquad (1)$$

$\forall J$, we have $X(\langle w\rangle, J, t_0) = X(w, J, t_0) = 0$.
If we suppose that the relation (1) is true for n, then

$$\begin{aligned} X(\langle w\rangle, J, t_{n+1}) &= w_J^n * (t_{n+1}-t_n) + X(\langle w\rangle, J, t_n) \\ &= w_J^n * (t_{n+1}-t_n) + X(w, J, t_n) \\ &= X(w, J, t_{n+1}) \end{aligned}$$

We can then conclude that relation (1) is true for all $n \in \mathbb{N}$.

As w is a correct schedule by hypothesis, for each job J, $X(\langle w\rangle, J, J.s) = X(w, J, J.s) = 0$ and $X(\langle w\rangle, J, J.d) = X(w, J, J.d) = J.e$, almost fulfilling the correctness conditions of definition 3; the fact that $\langle w\rangle$ does not execute the job when it is not active is trivial. □

Thus weighted schedules aren't more "powerful" than job-boundary weighted ones; i.e. if there are no job boundary weighted schedule, then there are no weighted schedule at all. This simplifies searching for weighted schedules and proving infeasibility.

Lemma 15. *There exists a conversion function from the job boundary weighted schedule to the concrete schedule.*

Proof. Let (t_n) be the sequence of deadlines and start times of $\mathcal{J}$. As in the precedent proof, we build a new schedule s so that each job is executed for the same amount of time at each t_n in both schedules.

[†] by the dominated convergence theorem.

Let $n \in \mathbb{N}$. On the interval $[t_n, t_{n+1}[$, $w_J(t)$ is a constant that we name w_J^n.

We want to build s so that it schedules all jobs in $\mathcal{J}$ on the interval $[t_n, t_{n+1}[$ for an amount of $w_J^n * (t_{n+1}-t_n)$.

The theorem of Horvath *et al.* [13, Theorem 1] gives the optimal finish time of a set of jobs for uniform multiprocessor; by applying their formula in our case the value of this optimal finish time τ is found:

$$\tau \;=\; \left(t_{n+1}-t_n\right)\,\max\left[\left(\max_{1\leq i<M}\frac{\mathrm{Smax}_i(w^n)}{\mathrm{Smax}_i(\alpha)}\right),\ \frac{\sum_{J\in\mathcal{J}} w_J^n}{\sum_{k=1}^{M}\alpha_k}\right]$$

As τ is the *optimal* finish time, scheduling the jobs in $\mathcal{J}$ for the given amount of time is only possible if τ is smaller than the duration of the interval $[t_n, t_{n+1}[$.

$$\begin{aligned} & \tau \leq (t_{n+1}-t_n) \\ \iff\quad & \max\left[\left(\max_{1\leq i<M}\frac{\mathrm{Smax}_i(w^n)}{\mathrm{Smax}_i(\alpha)}\right),\ \frac{\sum_{J\in\mathcal{J}} w_J^n}{\sum_{k=1}^{M}\alpha_k}\right] \leq 1 \\ \iff\quad & \begin{cases} \mathrm{Smax}_1(w(t)) \leq \mathrm{Smax}_1(\alpha) \\ \mathrm{Smax}_2(w(t)) \leq \mathrm{Smax}_2(\alpha) \\ \qquad\vdots \\ \mathrm{Smax}_{M-1}(w(t)) \leq \mathrm{Smax}_{M-1}(\alpha) \\ \displaystyle\sum_{J\in\mathcal{J}} w_J(t) \leq (\alpha_1+\ldots+\alpha_M) \end{cases} \end{aligned}$$

These latter equations are the validity conditions we impose on the weighted schedule (definition 7); as these conditions are fulfilled by hypothesis, it is thus possible to construct s between t_n and t_{n+1}. To construct it, we can either use Horvath *et al.*'s optimal finish time algorithm or the Gonzalez and Sahni [9] one.

Hence, s so constructed is a valid concrete schedule by construction; the proof that it is also a correct schedule is similar to the proof that $\langle w\rangle$ is a correct schedule in Lemma 14 and, as such, is not reproduced here.

So s is a correct concrete schedule for $\mathcal{J}$.

□

Note: Proof of Lemma 15 is constructive; if at time t_n, the weights and t_{n+1} are both known, it is possible to incrementally build the concrete schedule from the weighted schedule. Section 4 covers this subject.

Proof (of the main theorem). Figure 2 exhibits all the conversion functions we have proved so far.

As conversion functions can be composed, it is clear that it is possible to convert a correct schedule in any representation into another; thus all the four schedule representations are equivalent. □

2.4. Summary

Thanks to Theorems 11 and 13, scheduling problems such as constructing an optimal schedule or performing feasibility analysis can be done on the simpler weighted or job-boundary weighted schedule representations instead of the standard concrete schedule representation.

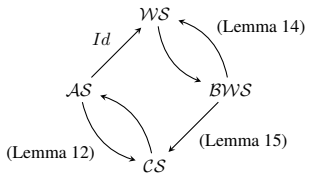


Figure 2. Conversion functions. (Id is the identity function; the other conversions are given in the different lemmas.)

The next section shows how this can be used to solve important practical problems.

3. Applications

The equivalence between the schedule representations greatly simplifies many scheduling problems; in this section we provide several important examples where equivalence between schedule representation is used both for feasibility analysis and to construct multiprocessor scheduling algorithms.

We call *task* an entity that can release jobs; this section shows how the task constraints on job releases can be used to compute correct weights and an optimal schedule.

3.1. Optimal schedule for periodic tasks

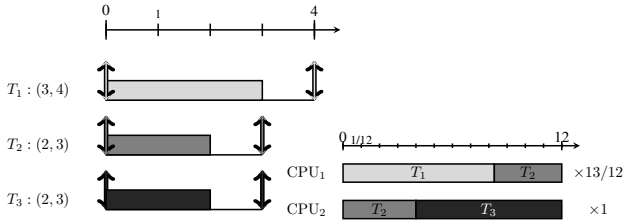
We consider the scheduling of a set of N implicit-deadline tasks on M processors, i.e. each task T releases jobs periodically every period $T.p$; each job has to execute for $T.e$ before the release of the next job.

We define the *utilization* of a job $J.u$ to be the share of the CPU that the job takes over its duration, i.e. $J.e/(J.d - J.s)$. For implicit-deadline periodic tasks, there is always one active job with constant utilization $T.u = T.e/T.p$.

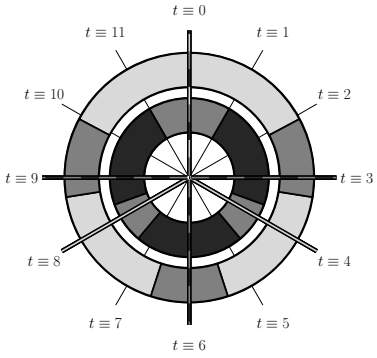
To construct a weighted schedule for this problem, it simply suffices to set the weight of each released job to its utilization. Such a schedule is called a *fair* schedule; thanks to equivalence between schedule representations, proving the feasibility is reduced to checking that the utilizations fulfill the equations of Definition 7, which can be made in $\mathcal{O}(N)$ time (N being the number of tasks).

The actual schedule can be built using the conversion function of Lemma 15; and the result is an optimal online scheduling algorithm which is absolutely fair at each boundary. Some remarks are worth noticing:

- The weight of each function is constant on each interval; only the duration of these intervals vary. This means that if the number of tasks does not change, the schedule conversion need to be done only once, and the same schedule can be repeated by scaling it to fit in each interval (figure 3(b)); this is why this method can be classified as "quasi-static" scheduling.
- To decrease the number of preemptions, it is possible to apply a "mirror symmetry" technique [14, 1] by using a reversed schedule every two intervals (e.g. $(t \equiv 0)$ is a symmetry axis in Figure 3(b)).
- This simple algorithm stands a qualitative comparison with PFair when comparing the number of preemptions: PFair has M preemptions per time quanta which is not as good as the given algorithm's N preemptions per interval except when N is big. However, when there is a large number N of small tasks (and total utilization is smaller than M), task migration is in general not needed and partitioning algorithms perform well. Other advantages of the algorithm is that it does not require integer period nor execution time, and that it works also on uniform multiprocessor.



(a) Execution requirements of the tasks, and their schedule (weights $3/4$, $2/3$ and $2/3$) on each interval for a uniform biprocessor of respective speed $13/12$ and 1.



(b) Complete schedule for the periodic tasks system of Figure 3(a). Job boundaries are at $t = 0, 3, 4, 6, 8, 9$. Interval $[0, 3]$ is three times the interval in $[8, 9]$; interval $[4, 6]$ is the symmetric of interval $[6, 8]$.

Figure 3. Scheduling three periodic tasks on 2 different processors.

These techniques are not reserved to periodic task systems, but can be used for every task system that has a constant utilization and for which the next job boundary is always known. For these tasks, fair scheduling is always optimal. However, when tasks have varying utilization it is in general not optimal; optimal scheduling for more general tasks are studied in section 3.3.

3.2. Optimal schedule for finite job sets

Although most real-time processing need recurring jobs, it is interesting to study finite job sets with real-time constraints because it is a good introduction for more compli-

cated tasks generating an infinite job set, presented in section 3.3.

Here by using the equivalence between the $\mathcal{BWS}$ representation and the concrete one, the problem of optimally scheduling finite job sets is transformed to the simpler one of finding a set of weights in each interval between job boundaries satisfying the conditions in Definition 7 and 8.

Identical multiprocessor case. The problem of finding an identical multiprocessor boundary weighted schedule for a job set $\mathcal{J}$ is eased by the fact that the validity and correctness conditions of a schedule w of $\mathcal{J}$ can be expressed by a set of linear equalities and inequalities. That way, the problem of finding a correct schedule (of finding whether one exists) is transformed into a simple linear programming problem, solvable using the simplex algorithm.

Indeed, the validity equations are the following ones:

$$\forall n, \quad \begin{cases} \sum_{J \in \mathcal{J}} w_J^n \ \leq \ M \\ \forall J, \quad 0 \leq w_J^n \leq 1 \end{cases}$$

The correctness equations are these ones (we note, when defined, $n(t)$ as being the n such that there is $t_n = t$) :

$$\forall J \in \mathcal{J}, \quad \begin{cases} \sum_{k=n(J.s)}^{n(J.d)-1} w_J^k * (t_{k+1} - t_k) = J.e \\ \forall k \text{ s.t. } k < n(J.s) \text{ or } k \geq n(J.d), \quad w_J^k = 0 \end{cases}$$

These correctness equations simply state that

$$\forall J \in \mathcal{J}, \quad \int_{J.s}^{J.d} w_J(t)\,\mathrm{d}t \ = \ J.e$$

They are rendered simpler because the w_n are constants on the intervals $[\,t_n, t_{n+1}\,]$.

Any solution (i.e. a set of w_J^n) satisfying these equations is a job-boundary weighted schedule of $\mathcal{J}$; and $\mathcal{J}$ is feasible *if, and only if*, the system of equalities and inequalities has a solution (i.e. it is an exact feasibility test).

For instance, the set of equations for the job set of Figure 1 would be:

$$\begin{array}{ll} w_1^0 * 2 + w_1^1 * 5 + w_1^2 * 1 = 6 & w_1^0 + w_2^0 \leq 2 \\ w_2^0 * 2 + w_2^1 * 5 + w_2^2 * 1 = 6 & w_1^1 + w_2^1 + w_3^1 \leq 2 \\ w_3^1 * 5 = 4 & w_1^2 + w_2^2 \leq 2 \end{array}$$

$$w_1^0, w_1^1, w_1^2, w_2^0, w_2^1, w_2^2, w_3^1 \in [0, 1]$$

Uniform multiprocessor case. The uniform multiprocessor case presents another difficulty, due to the non-linearity of the Smax_k functions of definition 7. The set of inequalities do not form a linear programming problem anymore.

This problem has already been studied by Martel [17] and solved using a specific network flow instead of linear programming. This technique offers a fast resolution and a small number of preemptions, and can in fact be used also in the case of identical multiprocessors [12].

This technique also acts as an exact feasibility test.

3.3. Optimal schedule for unpredictable tasks

The previous sections computed schedules of tasks for which the three parameters of all released jobs could be known before runtime. When this is not the case for a task set (like sporadic tasks), it may not be optimally schedulable [8]. However, it is still possible to use available partial information on the job sets to schedule it "at best", introducing the notion of *task-optimality*. To illustrate this problem, the OASIS [5] task model is used, because it allows the expression of varying temporal behaviours while retaining a finite set of possible behaviours (unlike sporadic tasks), allowing to infer much information more easily.

The OASIS model uses temporal automatons that can define the behaviour of any time-triggered task. For the sake of brevity only an informal description of OASIS automatons and their manipulation is given in this paper; complete coverage of the subject will be published in another one.

OASIS temporal automatons and their scheduling. An OASIS task is represented by a (potentially infinite) temporal automaton that releases jobs sequentially. Edges represent jobs and are labelled by their execution durations; nodes are job constraints and the label of the node pointed by an edge is the relative deadline of the job represented by the edge.

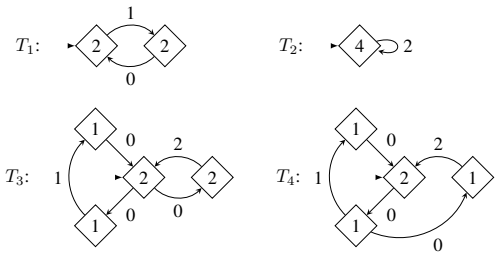


Figure 4. A set of temporal automatons.

For instance, in Figure 4, T_2 is an implicit-deadline periodic task of period 4 and T_1 is a constrained-deadline task of period 4 and relative deadline 2. T_3 and T_4 are more complicated tasks with varying temporal behaviour, but whose job releases (not counting jobs of duration 0) are the same; they differ by the moment of choice. T_3 chooses at the earliest moment and T_4 at the latest.

Scheduling OASIS tasks can be done as follows: from these automatons one can extract constraint equations (or flow network) similarly to the finite job set case (see Section 3.2)[2], using a synchronized product operator that combines the temporal automatons [2]. Solving these equations give a task-optimal multiprocessor boundary weighted schedule [14], which is convertible into a task-optimal multiprocessor concrete schedule. Note that the size of the synchronized product and thus the number of constraint equations is bounded in the worst case only by the gcm of the

[2] In fact, finite job sets are a special case of OASIS automatons.

sizes of the cycles of the original graphs. However, in practice as time is used for synchronization, there are many common factors between these cycles' size and the synchronized product does not grow that much.

Task- and job-feasibility. A task set is called *job-feasible* when every job set that can be released by the task set is schedulable. It is called *task-feasible* when the task set is schedulable. The difference is that task-feasibility takes into account the fact that job parameters are not known at the beginning of execution.

No such distinction exists on single processor computers because in this case there are optimal online scheduling algorithms for single processors such as EDF; also, no such distinction exists for periodic and finite tasks whose job parameters are all known before runtime. But as no optimal online scheduling algorithm exists on multiprocessor [11, 8] this distinction becomes important when scheduling more complex task sets with job releases only known at runtime.

Results. Our scheduling algorithm (offline weight finding plus weight conversion) is *task-optimal*: if a task set is task-feasible, it schedules it correctly. The weight-finding part of the algorithm alone is also an *exact task-feasibility test* (to our knowledge, this is the first time that a task-optimal algorithm or task-feasibility test is found for task sets that are not always job-feasible). In particular, as all job-feasible periodic tasks are also task-feasible (because all their job parameters are known at the beginning of execution), *we solve in particular the previously open problem of optimally scheduling general periodic tasks*.

Example. For instance, both task sets $\mathcal{T}_1 = (T_1, T_1, T_2, T_3, T_3)$ and $\mathcal{T}_2 = (T_1, T_1, T_2, T_4, T_4)$ are job-feasible on 2 processors; but only $\mathcal{T}_1$ is task-feasible (this is a transcription in the OASIS model of the Hong and Leung [11] proof that online scheduling of arbitrary tasks is impossible). Our method computes an optimal schedule for $\mathcal{T}_2$ and finds that $\mathcal{T}_1$ is schedulable when increasing each processor's speed by 11%. By comparison, using Global-EDF or Partitionned-EDF requires here an increase of 25% for both task sets; and proving that the system is schedulable with Global-EDF using Phillips *et al.*'s result [19] requires an increase of 50%.

So, despite the fact that optimal online scheduling is impossible [8, 11], it is still possible to schedule tasks at best using all available information; and this can represent a considerable gain over online algorithms.

4. Online schedule transformation

The previous section showed how transforming a schedule from the $\mathcal{BWS}$ representation to the concrete one can be used as a general method for constructing scheduling algorithms. Such transformations were done offline, which allowed using static or "quasi-static" scheduling algorithms.

Offline schedule transformation can be used when the set of changes of weight of the tasks is finite; in other cases when weights vary more dynamically it is necessary to perform schedule transformation online. For instance, the Least Laxity First algorithm for multiprocessor or CPU bandwidth-based systems [10] can both be implemented using this technique.

Other uses of online schedule transformation include saving memory space (weights takes less space to store than complete schedule) or coping with unpredictable jobs arrivals (e.g. to mix optimally-scheduled periodic tasks with sporadic tasks).

The basic online schedule transformation requires only to know the next job boundary; i.e., when t_n is reached, then t_{n+1} must be known. This is usually the case for time-triggered tasks like OASIS or periodic tasks. A simple algorithm, given in proof of Lemma 15, is to just apply the McNaughton [18] or Gonzalez and Sahni [9] algorithm for the sub-jobs between t_n and t_{n+1}.

4.1. Handling dynamic job arrival

The problem is that deciding the schedule for the interval renders it unmodifiable for the duration of this interval, and so unable to cope with unpredictable arrival of events.

The above basic online transformation algorithm, that decides the whole schedule for the interval $[t_n, t_{n+1}[$ at time t_n, is perfectly fine when the next job boundary is always known, for instance in a pure OASIS system or a set of periodic tasks. However, the current trend in the real-time industry is the integration of numerous tasks of different natures in a single system; for instance, OASIS or periodic tasks, event-triggered tasks and non real-time tasks.

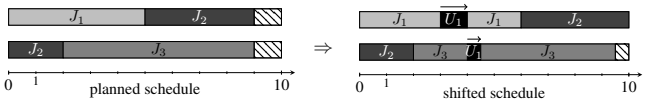
For such systems, the above solution has two drawbacks: first, as one cannot deal with unpredictable jobs (e.g. released by event-triggered tasks) before time t_{n+1}, these jobs are treated with a high latency; second, computational resources are lost when a job finishes earlier than expected, e.g. in the general case where the worst case execution time (WCET) is not reached.

To deal with these problems on identical processors, one can try to adapt the McNaughton algorithm to handle some dynamicity:

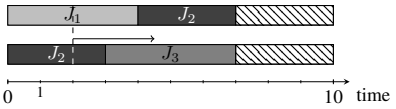
- The best way to cope with unpredictable events is to just "shift" schedules when such events appear. Long events have to be split and migrated between the different processors (thus causing $M - 1$ supplementary migrations). Figure 5(a) illustrates the method.

  Note that the shift must be done in the same processor order than used with the McNaughton algorithm to prevent simultaneous execution of the same job on different processors (Figure 5(b)).

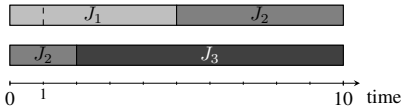
- However, handling early termination is impossible: Figure 5(c) shows why. The only simple possible adaptation would be to execute a completely unrelated job between the end of a job and its expected end, which is not a really efficient solution.



(a) Dealing with unpredictable job arrivals can be done by shifting the schedule on a processor to insert the unpredictable job when it arrives (here at time 3). When there is not enough room on it, migrate the unpredictable job on another processor.



(b) Shifting the schedule must be done in the same processor order than used when applying the McNaughton algorithm: here a job of length 2.5 arriving at time 2 must be placed on the upper processor, before another can be put on the lower one; else there would be a simultaneous execution conflict.



(c) Handling early job end is impossible: if at time 1 job J_1 finishes early, job J_2 cannot start because this would cause simultaneous execution on two processors

Figure 5. Adapting a McNaughton schedule to handle unpredictable job arrivals.

Thus, there is a need for a dynamic schedule transformation algorithm that can cope with these problems.

4.2. An incremental online schedule transformation algorithm

We propose Algorithm 1, that builds the schedule on the interval $[t_n, t_{n+1}[$ *incrementally*: it determines the next preemption or migration instant only at the previous one, which allows it handling dynamic jobs arrivals by changing the scheduling decision when these jobs arrive. It works by observing the biggest tasks and exclusively reserving processors to them when they need it, and is based on a previous proof of Lemma 14 we made [14].

Although it causes more preemptions $(2M-2)$ than the McNaughton algorithm $(M-1)$ in the maximum case, it never causes more migrations than it $(M-1)$. Nevertheless, when the system is lightly loaded, it causes few or no migration; whereas the McNaughton algorithm causes *on average* $M-1$ migrations.

Additional interesting properties of the algorithm is that the M biggest jobs are never migrated, which is interesting because they need a priori more cache and thus have a higher migration cost; the "setup time" at the beginning of the algorithm is $\mathcal{O}(M)$ instead of $\mathcal{O}(|\mathcal{J}|)$ (which is important because this setup time is lost for all processors for an online algorithm); and execution time of each `while` loop iteration is constant and in fact really small (the data structures can be arranged to be kept sorted).

Figure 6 provides an example of an execution of Algorithm 1 for 8 jobs on 4 identical processors.

Algorithm 1 Incrementally build a concrete schedule for a set of jobs J from their given weights w_J in the $\mathcal{BWS}$ representation, between t_n and t_{n+1} on M identical processors

Require: The job set $\mathcal{J}$ is feasible; t_n is the current date and t_{n+1} the next date when weight change

Ensure: At t_{n+1}, each job J will have been executed for $(t_{n+1}-t_n)*w_J$

$T \leftarrow t_{n+1}-t_n$
$\mathcal{P} \leftarrow \{1,2,\ldots,M\}$
$\mathcal{B} \leftarrow \mathcal{J}$
for all $J \in \mathcal{J}$ **do**
  $\tau_J \leftarrow w_J * T$
end for

while $\mathcal{B}$ contains jobs with non-null τ_J **do**
  $J_{\min} \leftarrow J \in \mathcal{B}$ that has the smallest non-null τ_J
  $J_{\max} \leftarrow J \in \mathcal{B}$ that has the biggest τ_J
  Place the $|\mathcal{P}|$ jobs in $\mathcal{B}$ with smallest non-null τ_J on processors of $\mathcal{P}$
  Place the $M-|\mathcal{P}|$ jobs of $\overline{\mathcal{B}}$ on processors of $\overline{\mathcal{P}}$
  $t \leftarrow \min(\tau_{J_{\min}}, T-\tau_{J_{\max}})$
  Execute the placed jobs for t
  for all $K \in \{\text{placed jobs}\}$ **do**
    $\tau_K \leftarrow \tau_K - t$
  end for
  $T \leftarrow T-t$
  if $t = \tau_{J_{\max}}$ **then**
    $\mathcal{P} \leftarrow \mathcal{P} \setminus \{\text{processor were was } J_{\min}\}$
    $\mathcal{B} \leftarrow \mathcal{B} \setminus \{J_{\max}\}$
  end if
end while
Place the $M-|\mathcal{P}|$ jobs of $\overline{\mathcal{B}}$ on processors of $\overline{\mathcal{P}}$
Execute the placed jobs for T

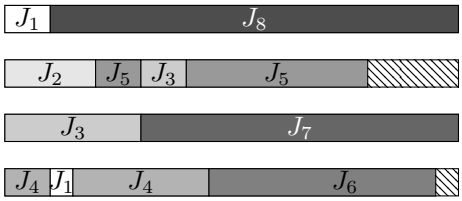


Figure 6. Typical run of Algorithm 1 on 4 identical processors. (Jobs are sorted by increasing durations, i.e. $J_1 \leq J_2 \leq \cdots \leq J_8$.)

Notes:
- For simplicity's sake, we assume that when no job is placed on a processor, this processor idles.
- Handling unpredictable job arrival or premature

end would require slight modifications of the algorithm, but they depends on what one wants to achieve (are the unpredictable jobs more critical than the others? do they have deadlines?), so the algorithm is given without them.

Proof of the algorithm and of its properties, as well as its real implementation, can be found in a technical report about this algorithm [15].

5. Conclusion

This paper proves that the intuitive notion of "fluid scheduling", in which jobs can execute at fractional rate, is no more powerful than preemptive scheduling, even on multiple processors that have different speeds (uniform multiprocessors). Explaining the precise meaning of this result and proving it require a formal framework named *equivalence between schedule representations*.

Using this framework, we are able to provide new simple, efficient solutions to some already solved but hard problems, like optimal scheduling of implicit-deadline periodic tasks [4] on identical multiprocessors; using the same technique we also solve the open problem of optimally scheduling these tasks on uniform multiprocessor.

The framework makes accessible the solving of new, harder problems, like optimally scheduling general periodic tasks on multiprocessors, or even any time-triggered task. As this latter problem cannot be optimally solved [11, 8], we distinguish two notions of optimal scheduling, the ideal *job-optimality* and the reachable *task-optimality*; and we briefly sketch how to task-optimally schedule time-triggered tasks using OASIS temporal automatons; a more detailed paper on this problem will follow.

Finally, we discuss on the interest of *online schedule transformation*, that allows expressing scheduling algorithm using weights or "rates" even on multiprocessor architectures. We then provide an efficient algorithm for online schedule transformation on identical multiprocessors that can handle event-triggered tasks.

As the framework greatly simplifies multiprocessor scheduling problems (feasibility analysis *and* construction of scheduling algorithms), numerous applications that need multiple resource preemptive scheduling can be built upon the framework even beyond multiprocessor scheduling, for instance in QoS in telecommunications.We are also implementing the task-optimal scheduling algorithm for OASIS, which will provide us with quantitative benchmarks on the performance increase compared to our current Global-EDF and partitioned-EDF solutions on an industrial case study. Further works will also cope with minimization of the number of preemptions and migrations caused by the algorithm.

References

[1] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *Proceedings IEEE (RTCSA'06)*, pages 322–334, 2006.

[2] C. Aussaguès and V. David. A method and a technique to model and ensure timeliness in safety critical real-time systems. In *Fourth IEEE International Conference on Engineering Complex Computer Systems (ICECCS'98)*, pages 2–13, Los Alamitos, CA, USA, 1998.

[3] S. Baruah and N. Fisher. The feasibility analysis of multiprocessor real-time systems. *ecrts*, 0:85–96, 2006.

[4] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. In *STOC '93: Proceedings of the 25th annual ACM symposium on Theory of computing*, pages 345–354, 1993.

[5] D. Chabrol, V. David, C. Aussagues, S. Louise, and F. Daumas. Deterministic distributed safety-critical real-time systems within the OASIS approach. In *17th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'05)*, November 2005.

[6] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 101–110, 2006.

[7] V. David, C. Aussaguès, S. Louise, P. Hilsenkopf, B. Ortolo, and C. Hessler. The OASIS based qualified display system. In *Proceedings ANS (NPIC&HMIT 2004)*, 2004.

[8] M. L. Dertouzos and A. K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, 1989.

[9] T. Gonzalez and S. Sahni. Preemptive scheduling of uniform processor systems. *J. ACM*, 25(1):92–101, 1978.

[10] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, 1996.

[11] K. Hong and J.-T. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41(10):1326–1331, 1992.

[12] W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.

[13] E. C. Horvath, S. Lam, and R. Sethi. A level algorithm for preemptive scheduling. *J. ACM*, 24(1):32–43, 1977.

[14] M. Lemerre, V. David, C. Aussaguès, and G. Vidal-Naquet. A new representation for the scheduling problem and its applications. In *Proceedings WiP IEEE RTSS'06*, pages 89–93, December 2006.

[15] M. Lemerre, V. David, C. Aussaguès, and G. Vidal-Naquet. *Online conversion of a weight-based schedule representation on multiple identical processors*. Technical Report, CEA LIST, 2008.

[16] 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.

[17] C. Martel. Preemptive scheduling with release times, deadlines, and due times. *J. ACM*, 29(3):812–829, 1982.

[18] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1):1–12, 1959.

[19] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. In *Proc. 29th Ann. ACM Symp. Theory of Computing*, pages 140–149. ACM, May 1997.

[20] D. Zhu, D. Mossé, and R. Melhem. Multiple-resource periodic scheduling problem: how much fairness is necessary? In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 142, 2003.