DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling

Greg Levin[†], Shelby Funk[‡], Caitlin Sadowski[†], Ian Pye[†], Scott Brandt[†]
[†]Computer Science Department
University of California, Santa Cruz
{glevin, supertri, ipye, sbrandt}@soe.ucsc.edu
[‡]Department of Computer Science
University of Georgia, Athens, GA
shelby@cs.uga.edu

Abstract

We consider the problem of optimal real-time scheduling of periodic and sporadic tasks for identical multiprocessors. A number of recent papers have used the notions of fluid scheduling *and* deadline partitioning *to guarantee optimality and improve performance. In this paper, we develop a unifying theory with the* DP-FAIR *scheduling policy and examine how it overcomes problems faced by greedy scheduling algorithms. We then present a simple* DP-FAIR *scheduling algorithm,* DP-WRAP*, which serves as a least common ancestor to many recent algorithms. We also show how to extend* DP-FAIR *to the scheduling of sporadic tasks with arbitrary deadlines.*

1. Introduction

Multiprocessor systems are becoming commonplace as more computers, even desktops, have multiple cores. Many implications of using a multiprocessor system, including scheduling issues, are still not well understood. Multiprocessor scheduling is particularly difficult in the presence of hard real-time constraints. Real-time scheduling algorithms that are known to perform very well on uniprocessor systems, such as Earliest Deadline First (EDF) [21], do not perform as well on multiprocessors.

Broadly, there are two types of multiprocessor scheduling algorithms: *global* and *partitioned*. Global algorithms use a single scheduler for all processors and allow tasks to migrate between processors. Partitioned algorithms start by partitioning tasks among processors; scheduling is then handled by simpler uniprocessor algorithms, and no migration is allowed. Partitioned approaches are easy to implement, as they reduce multiprocessor scheduling to uniprocessor scheduling. However, they are not optimal, in the sense that they can fail to schedule theoretically feasible task sets[1].

[1]In fact, examples may be constructed where partitioned schedulers fail to successfully schedule tasks sets that only require $(50 + \epsilon)\%$ of processor capacity [7, 22].

In 1996, Baruah et al. [5] introduced the PFAIR algorithm, the first optimal multiprocessor scheduler for periodic tasks. By migrating tasks between processors, PFAIR can successfully schedule any task set whose utilization does not exceed processor capacity. More recently, a number of papers have exploited *deadline partitioning* (subdividing time into slices where all tasks have the same deadline) to achieve optimality while greatly reducing the number of required context switches and process migrations [3, 9, 26]. These and other algorithms, while superficially different, have achieved optimality by expanding on the core idea of tracking the *fluid schedule*, or average rate curve. A better understanding of their shared traits will aid in the ability to understand, compare, and contrast these algorithms, to consolidate their insights, and to point towards new avenues of study.

The contributions of this papers are:

- We explore the difficulties of optimal multiprocessor scheduling and the failure of greedy algorithms.
- We give a simple set of guidelines, called DP-FAIR, for designing optimal schedulers for periodic task sets.
- We describe the DP-FAIR scheduling algorithm DP-WRAP (similar to EKG [3] and BF [26]), the simplest optimal scheduler to date, with reasonably good migration bounds and limited computational overhead.
- We demonstrate the flexibility of the DP-FAIR guidelines and the DP-WRAP algorithm by extending them to handle sporadic task sets with arbitrary deadlines.

The remainder of this paper is organized as follows. Section 2 formalizes the problem under consideration and provides relevant definitions. Section 3 examines why greedy scheduling algorithms (like EDF) tend to fail in a multiprocessor environment. Section 4 presents the DP-FAIR scheduling principles and proves their correctness; it also presents the simple DP-WRAP scheduling algorithm. Section 5 extends the DP-FAIR conditions to handle sporadic tasks with arbitrary deadlines. Finally, Section 6 gives a brief survey of recent schedulers in the context of DP-FAIR.

1068-3070/10 $26.00 © 2010 IEEE
DOI 10.1109/ECRTS.2010.34

m	number of processors
n	number of tasks
τ	set of tasks $\{T_1, \ldots, T_n\}$
T_i	i^{th} task
p_i	period (minimum interarrival time) of T_i
e_i	workload of each job of T_i
D_i	time between arrival and deadline of T_i
$a_{i,h}$	arrival time of h^{th} job of T_i
δ_i	$e_i / \min\{p_i, D_i\}$ (density of T_i)
$\Delta(\tau)$	$\sum_i \delta_i$ (total density of τ)
$S(\tau)$	$m - \Delta(\tau)$ (total slack of τ)
σ_j	j^{th} time slice, time interval = $[t_{j-1}, t_j)$
t_j	j^{th} system deadline (end time of σ_j)
L_j	$t_j - t_{j-1}$ (length of σ_j)
$l_{i,t}$	local execution remaining for T_i at t
$r_{i,t}$	$l_{i,t}/(t_j - t)$ (local utilization of T_i at t)
$\mathcal{L}_t$	total local execution at t
R_t	total local utilization at t
$c_{i,t}$	local capacity remaining for T_i at t
$\alpha_{i,j}(t)$	time T_i has been active in σ_j as of t
$f_{i,j}(t)$	time T_i has freed slack in σ_j as of t
$w_{i,j}(t)$	work executed by T_i in σ_j as of t
$F_j(t)$	$\sum_i \delta_i f_{i,j}(t)$ (total slack freed in σ_j as of t)
$I_j(t)$	total idle time in σ_j as of t

Figure 1. Summary of Notation
The first group of symbols defines a task set, the second group is used in Section 4, and the third in Section 5.

2. Background

This paper considers the scheduling of n periodic or sporadic tasks on a system of m identical processors. Without loss of generality, we assume the speed of each processor is 1, i.e., each processor performs one unit of work per unit of time. Each task will invoke a series of jobs, and each job requires a certain amount of work be performed before its deadline. Given a collection of periodic or sporadic tasks, the basic problem is to find a *schedule* to specify which task (if any) runs on each processor at any given instant, with the restriction that no task can run on multiple processors at the same time. We allow tasks to be preempted or to migrate between processors at any time.

A task $T_i = (p_i, e_i, D_i)$ is a process that invokes a sequence of jobs $\{T_{i,h}\}_{h \geq 1}$. Each job $T_{i,h}$ arrives at time $a_{i,h}$, and has an execution requirement e_i and a deadline at $a_{i,h} + D_i$. Thus $T_{i,h}$ must be allowed to execute for e_i time units during the interval $[a_{i,h}, a_{i,h} + D_i)$. The minimum time between the arrivals of the jobs of T_i is p_i. If $D_i = p_i$, we refer to this as an *implicit deadline* and, dropping the implicit D_i, use the abbreviated notation $T_i = (p_i, e_i)$; otherwise, we say the task has an *arbitrary deadline*. If T_i is a periodic task, then it invokes its first job at time $t = 0$ and all its remaining jobs are invoked exactly p_i time units apart, i.e., $a_{i,h} = (h-1)p_i$ for all h. If T_i is a sporadic task, then it invokes its first job at any time $t \geq 0$ and the remaining jobs are invoked no less than p_i time units apart, i.e., $a_{i,1} \geq 0$, and $a_{i,h} \geq a_{i,h-1} + p_i$ for all $h > 1$. We let $\tau = \{T_1, T_2, \ldots, T_n\}$ denote a set of n periodic or sporadic tasks.

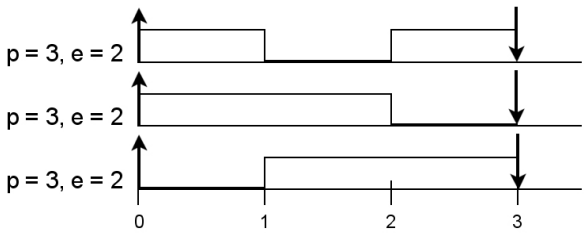

Figure 2. Simple Scheduling Problem
Three tasks, each with a rate of $2/3$, can run successfully on two processors with migration.

One important parameter used to describe a task T_i is its *utilization* $u_i = e_i/p_i$. For periodic tasks, the utilization measures the proportion of time a task executes on average. For sporadic tasks, the utilization measures the "worst-case average", i.e., the average proportion of required computing time assuming a worst case sequence of arrivals ($a_{i,j} = a_{i,j-1} + p_i$). The total utilization of task set τ, denoted $U(\tau)$, is the sum of the individual utilizations:

$$U(\tau) = \sum_{i=1}^{n} u_i \ .$$

When deadlines are not equal to periods, we instead use the task's *density*, $\delta_i = e_i / \min\{p_i, D_i\}$, and we let $\Delta(\tau)$ denote the total density of the task set τ. Notice that if $D_i = p_i$ then $u_i = \delta_i$. For the remainder of the paper, we will use δ and Δ for consistency, even when discussing utilization for tasks with implicit deadlines.

A *valid* schedule is one where all jobs meet their deadlines. We say that a set of tasks is *feasible* if some valid schedule exists, and a scheduling algorithm is *optimal* if it can successfully schedule any feasible task set. A simple example depicted in Figure 2 demonstrates a set of 3 tasks that can be successfully scheduled on two processors only when one of them divides its time between both CPUs.

Not all valid schedules are equally good. In order to reduce overhead, scheduling algorithms must have short execution times and also try to minimize other costs, such as those associated with context switches and migrations. Although highly system-dependent, task migrations generally take longer than context switches, sometimes prohibitively longer. Because global scheduling algorithms migrate tasks and also tend to be complex (and, therefore, have long

run times), partitioned schemes are preferred in practice. Newer multiprocessor architectures, such as multicore processors, have significantly reduced the migration overhead. The preference for partitioned scheduling may no longer be necessary on these types of multiprocessors.

If we assume preemptions and migrations can occur instantly, then in theory it does not matter *which* processor is hosting a given task, only which tasks are running at a given time. This assumption can lead to clearer scheduling descriptions (e.g., Figures 2 & 4). In fact, some recent algorithms give no explicit prescription for how to assign tasks to processors [9, 13].

For now, we will focus on periodic task sets with implicit deadlines, and save the more general cases for Section 5. We will assume no scheduling overhead[2], so that we should have enough CPU time to complete all jobs (i.e., the task set is feasible) provided:

(i) Total task workload doesn't exceed total CPU capacity ($\Delta(\tau) \leq m$).

(ii) No task's workload exceeds its period or deadline ($\delta_i \leq 1 \ \ \forall i$).

(iii) Process migration is allowed.

Given unlimited context switching and migration, it is not hard to see that any task set satisfying these conditions will be feasible. This fact is just an extension of the uniprocessor case presented by Liu and Layland [21]. Imagine that we can reschedule our jobs after each ϵ of time. As $\epsilon \to 0$, we can turn each task T_i on or off sufficiently often so that it appears to be constantly running on only a fraction δ_i of a processor. In the limit, each job executes at exactly its necessary rate and, when all rates sum to no more than m, all jobs finish on time. Srinivasan et al. [25] refer to this as a *fluid* scheduling model. Figure 3 shows the fluid and actual scheduling of a task.

Determining the feasibility of a periodic task set with implicit deadlines is easy; much more challenging is actually *finding* a valid schedule that minimizes context switches and migrations. This has been the goal of recent papers in this problem domain, and is our primary interest. To motivate our approach, we explore the shortcomings of greedy schedulers.

3. Greedy Schedulers

An attractive (and common [7]) first approach to scheduling is to try to find a simple greedy solution. Greedy algorithms are straightforward to explain, prove and implement. They often attempt to encapsulate the criticality

[2] Alternatively, we can assume the overhead costs are included in the tasks' execution requirements. This is a valid assumption if the worst-case number of preemptions and migrations can be determined in advance, which is often the case. However, this implementation could lead to very pessimistic worst case execution times.

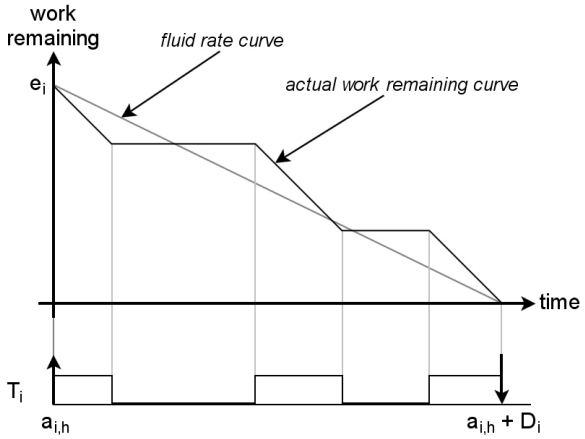

Figure 3. Fluid versus Practical Schedules

(likeliness of a missed deadline) of a job into a single quantity and then use that to greedily schedule jobs.

Two common greedy algorithms for uniprocessor scheduling are Earliest Deadline First (EDF) [21] and Least Laxity First (LLF) [24]. As their names imply, these algorithms give execution priority to the job with the earliest deadline or least laxity, respectively. The *laxity* of a job is the difference between its time and work remaining until its deadline. Although LLF has a higher scheduling overhead, both algorithms are optimal on a single processor; unfortunately, neither is optimal on a multiprocessor ([18], Figure 4a).

The LLF scheduler is based partially on the observation that a schedule has become infeasible if any job ever has negative laxity (more work than time remaining). Clearly, any job whose laxity has reached zero must be immediately activated and run continuously until its deadline in order to complete its work on time. This observation leads us to our second consideration when designing a greedy algorithm. The "greedy" part tells us *which* tasks to schedule, but we must also specify *when* the algorithm will do the scheduling. That is, at what times should re-sorting and the application of the greedy preference occur? Three standard scheduling events are:

RELEASE: A task is at the beginning of its period;

WORK COMPLETE: A task has finished its work for its current period, and must be turned off;

ZERO LAXITY: A task has no remaining laxity for its current period, and must be turned on.

Any simple greedy algorithm must specify its greedy sort key and which scheduling events it will observe. For example, when ZERO LAXITY events are added to EDF, the result is a hybrid scheduler known as EDZL [10]. While this provides an improvement over standard EDF for multiprocessors, EDZL is still not optimal.

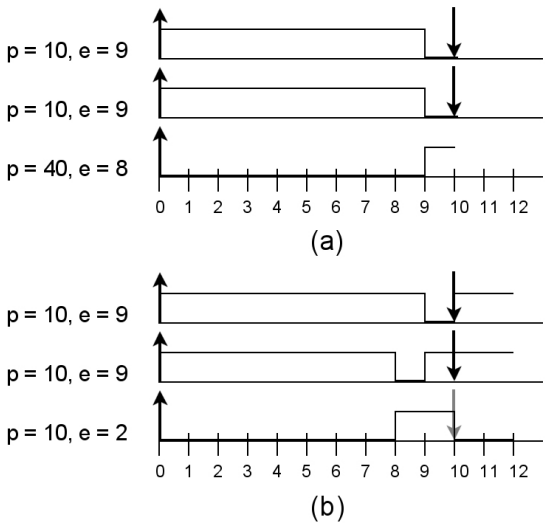

Figure 4. Greedy Counter-example
Task set which confounds known greedy schedulers using common events. (a) shows an incorrect greedy scheduling, while (b) shows a feasible proportional scheduling.

3.1. Why Greedy Schedulers Fail

To improve upon greedy schedulers, it is necessary to understand why they fail. Consider the feasible task set

$$\{\ T_1 = (10, 9),\ T_2 = (10, 9),\ T_3 = (40, 8)\ \}$$

shown in Figure 4(a); LLF cannot successfully schedule this job set on two processors. T_1 and T_2 each have laxity of 1, and so are prioritized and run to completion while T_3's laxity drops from 32 to 23. When T_1 and T_2 finish at $t = 9$, T_3 is the only task with work remaining. One processor is idle until T_1 and T_2 are re-released at $t = 10$.

In fact, it seems implausible that *any* greedy scheduler would choose to activate T_3 before $t = 9$. Even at $t = 8$, T_1 and T_2 have earlier deadlines, lower laxity, higher total and remaining utilizations, etc. It is difficult to envision an intelligent criterion which would prefer T_3. And yet, if T_3 is not activated by $t = 8$, a deadline will eventually be missed[3]. The following theorem generalizes this observation.

Theorem 1 *When the total utilization of a periodic task set is equal to the number of processors, then no feasible schedule can allow any processor to remain idle for any length of time.*

Proof. Given tasks $T_1, \ldots, T_n$ on m processors where the rates sum to m. In a feasible schedule, task T_i, at the end of h periods, must have done work equal to $he_i = h(p_i\delta_i) = (hp_i)\delta_i = a_{h+1}\delta_i$, where a_{h+1} is the ending time of the h^{th} period. Let t' be the first positive time at which all tasks reach a deadline simultaneously (i.e., the least common multiple of their periods). Then the total work done by all tasks by time t' must be $\sum_i t'\delta_i = t' \sum_i \delta_i = t'm$. This much work can be accomplished by time t' only if all processors are running continuously until this time. Any idle time implies less than $t'm$ total work is done, and some deadline will be missed. □

Clearly something critical is happening at time $t = 8$ in our example, but no obvious scheduling event occurs here, and no reasonable greedy sort would respond correctly. Only by considering T_1 and T_2 *as a set* can we see that they leave two units of idle time to be filled by time $t = 10$, and that one job cannot fill this time on two processors simultaneously. When greedy algorithms fail, it is usually for lack of some global knowledge. In our case, such knowledge is implicitly provided by a single over-constraint.

4. Deadline Partitioning and DP-FAIR

To date, the only known solutions to the "global knowledge" problem are variations on *proportional fairness*. By over-constraining our scheduling requirements, proportional fairness forces tasks to march in step with their fluid rate curves more precisely than is theoretically necessary. Suppose we modify our previous example to

$$\{\ T_1 = (10, 9),\ T_2 = (10, 9),\ T_3 = (10, 2)\ \}\ .$$

All we have done is to impose the additional requirement that task T_3 complete a proportional share of its work every time the other tasks hit their deadlines. Suddenly we have a ZERO LAXITY event at time 8. T_3 will then be switched on; T_1 and T_2 will each run for one of the remaining two time units on the other processor, and a feasible schedule results (see Figure 4b). In this example, where the third period was a multiple of the first two, it is easy to reformulate the problem in this way. When we have numerous jobs with disparate periods, the question of when to force jobs to hit their proportional rate quotas is not immediately obvious.

The first solution to this problem was the PFAIR scheduling scheme [5]. PFAIR creates a scheduling event and recomputes the set of running tasks at every multiple of a discrete time quantum. The notion of proportional fairness used is very strict, requiring the actual work completed by a task to be within 1 unit of its fluid rate curve at each time quantum. The result of this policy is a large number of scheduling calculations and context switches, with correspondingly high overhead. Intuitively, it seems unnecessary to adhere so closely to the fluid schedule: performance could be improved by a more judicious choice of scheduling events.

[3] Between $t = 0$ and $t = 40$, $(2 \times 4 \times 9) + 8 = 80$ units of work must be done, though the two processors can only accomplish 79 if one is idle between $t = 9$ and 10.

4.1. DP-FAIR Conditions for Periodic Tasks

To motivate our next step, recall the following result by Hong and Leung [15].

Theorem 2 *No optimal on-line scheduler can exist for a set of jobs* with two or more distinct deadlines *on any m-processor system, where $m > 1$.* $\Diamond$

Note that Theorem 2 does not apply when all deadlines are equal. In fact, Hong and Leung also present the RESCHEDULE algorithm, which they prove is optimal when all jobs have the same deadline, even when some arrival times are unknown. Their RESCHEDULE algorithm is similar in design to our own DP-WRAP algorithm in Section 4.2. Let us first consider the benefit of *forcing* all jobs to have the same deadline.

Deadline partitioning (DP) is the technique of partitioning time into *slices*, demarcated by all the deadlines of all tasks in the system. Within each slice, all jobs are allocated a workload for the time slice and these workloads share the same deadline. While a number of recent algorithms [3, 9, 26] have used deadline partitioning, there has not previously been a unifying theory for why this technique is so effective. With the DP-FAIR conditions presented below, we provide such a theory.

There are two aspects to deadline partitioning: *allocating* the workloads for all tasks for each time slice, and *scheduling* within a time slice. We say that an algorithm using this approach is DP-CORRECT if (i) the time slice scheduler will execute all jobs' allocated workload by the end of the time slice whenever it is possible to do so, and (ii) jobs are allocated workloads for each slice so that it is possible to complete this work within the slice, and completion of these workloads causes all tasks' actual deadlines to be met. In other words, any DP-CORRECT scheduler is optimal.

Before we proceed, we will require some additional notation. We let $t_0 = 0$ and $t_1, t_2, \ldots$ denote the distinct deadlines of all tasks in τ, where $t_j < t_{j+1}$ for all $j \geq 0$. Then the j^{th} time slice, denoted σ_j, is $[t_{j-1}, t_j)$, and has length $L_j = t_j - t_{j-1}$. Unless otherwise noted, we only consider one time slice σ_j at a time. As general conventions, when time t is a parameter, we will subscript (e.g., X_t) to refer to "remaining X", and parenthesize (e.g., $X(t)$) to refer to "X so far". Subscript h will index the h^{th} job in a task, i will represent task T_i, and j is for time slice σ_j.

We analyze schedules by considering execution during σ_j, drawing heavily on notation from the LLREF scheduling algorithm [9]. The *local execution remaining* of a task T_i at time t, denoted $\ell_{i,t}$, is the amount of time that T_i must execute before the next time slice boundary, i.e., between times t and t_j. A task's *local utilization* $r_{i,t} = \ell_{i,t}/(t_j - t)$ is the proportion of time between t and t_j that T_i must spend executing. We let $\mathcal{L}_t$ and R_t denote a task set's summed local remaining execution and utilization, respectively, at time t.

The scheduling process is most easily understood when the task set has full utilization, i.e., $\Delta(\tau) = m$. Since we do not generally expect full utilization, one or more dummy tasks may be introduced to make up the difference. With this intent, we define the *slack* of a task set to be $S(\tau) = m - \Delta(\tau)$. Consider a time slice of length 10 on 2 processors, and a task set with $\Delta(\tau) = 1.5$. The system has the capacity to do 20 units of work, but with only 15 units of work to be done, 5 units of idle time must appear somewhere within the slice. While common sense might dictate that an algorithm should always be doing work if there is work to be done, and that the idle time should therefore come at the end of the slice, this is an unnecessary over-constraint on algorithm design. By viewing slack as a dummy job and idle time as a necessary activity, we provide maximum freedom in scheduling the time slice.

Note that our description of idle time as a capacity-consuming resource, and our attempts to provide maximum flexibility in scheduling it, are not just for convenience. While our model treats it as dead processor time, that "dead time" can actually be employed for a number of purposes, including load balancing [4], improving performance [3, 19], creating a work-conserving scheduler [12, 13], or running non-real-time tasks in a hybrid system [20].

We now propose a minimally restrictive set of scheduling rules, DP-FAIR, which ensure that an algorithm is DP-CORRECT and provide substantial latitude for algorithm design. DP-FAIR Allocation for periodic task sets with implicit deadlines is quite simple: ensure that all tasks hit their fluid rate curves at the end of each slice by assigning each task a workload proportional to its utilization; that is, task T_i is assigned workload $\ell_{i,t_{j-1}} = \delta_i \times L_j$ for time slice σ_j. With these allocations in mind, we are ready to formulate our DP-FAIR Scheduling conditions. $F_j(t)$ in RULE 3 is a *freed slack* term that will be used in Section 5, but for now is just zero.

Definition 1 (DP-FAIR Scheduling for time slices) *A slice-scheduling algorithm is* DP-FAIR *if it schedules jobs within a time slice σ_j according to the following rules:*

RULE 1: *Always run a job with zero local laxity;*

RULE 2: *Never run a job with no remaining local work;*

RULE 3: *Do not voluntarily allow more than $(S(\tau) \times L_j) + F_j(t)$ units of idle time to occur in σ_j before time t.* $\Diamond$

We now prove that any DP-FAIR scheduler is optimal via a pair of Lemmas.

Lemma 3 *If tasks τ are scheduled within a time slice according to* DP-FAIR*, and $R_t \leq m$ at all times $t \in \sigma_j$, then all tasks in τ will meet their local deadlines at the end of the slice.*

Proof. A task can only miss its (local) deadline if it achieves negative (local) laxity. However, by RULE 1, any job that hits zero laxity will be run to completion on some processor. The only way this scheme can fail to finish all jobs' local workloads on time is if more than m jobs simultaneously have zero laxity, so that one of them cannot be run. Since a zero laxity job has $r_{i,t} = 1$, we would have $R_t \geq m + 1$, contradicting our assumption. □

Lemma 4 *If a set τ of periodic tasks with implicit deadlines is scheduled in σ_j using any* DP-FAIR *algorithm, then $R_t \leq m$ will hold at all times $t \in \sigma_j$.*

Proof. Let us introduce the dummy job T_{n+1} representing idle time, and give it utilization $S(\tau)$. Note that T_{n+1}'s utilization can be larger than 1, since this one "job" is allowed to run on multiple processors at once. (To stay closer to our task model, we could introduce $S(\tau)/\epsilon$ jobs, each with utilization ϵ, and let $\epsilon \to 0$. For this proof, this is an unnecessary complication.) We let $\ell_{n+1,t}$ be the portion of the total $S(\tau) \times L_j$ idle time in σ_j not yet used up as of time t, and let

$$\mathcal{L}_t = \sum_{i=1}^{n} \ell_{i,t} \quad \text{and} \quad \mathcal{L}'_t = \sum_{i=1}^{n+1} \ell_{i,t}$$

be the work remaining at time t in our original and extended task sets, respectively. Now, the m processors are consuming the workload from $T_1, \ldots, T_{n+1}$ at a rate of m per time unit, so of the mL_j units of work and idle time that needed to be consumed at the beginning of σ_j, $m(t_j - t_{j-1}) - m(t - t_{j-1}) = m(t_j - t)$ remain at time t, i.e., $\mathcal{L}'_t = m(t_j - t)$. Then

$$R_t = \sum_{i=1}^{n} \frac{\ell_{i,t}}{t_j - t} \leq \frac{1}{t_j - t} \sum_{i=1}^{n+1} \ell_{i,t} = \frac{\mathcal{L}'_t}{t_j - t} = m \ ,$$

as desired. □

Theorem 5 *Any* DP-FAIR *scheduling algorithm for periodic task sets with implicit deadlines is optimal.*

Proof. Lemmas 3 and 4 show that all tasks will meet all local deadlines at the end of time slices by following DP-FAIR's rules; that is, each job's work completed will match its fluid rate curve at *every* system deadline, including its own. Since any T_i's fluid rate curve is zero at its own deadlines, it follows that T_i will meet its deadlines. This holds for all jobs from all tasks, so any DP-FAIR algorithm is optimal. □

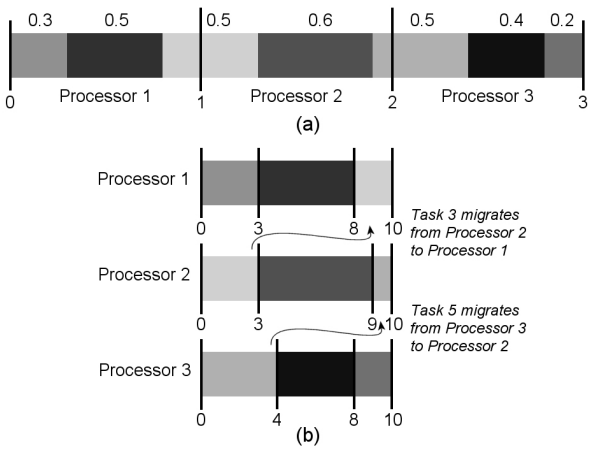

Figure 5. The DP-WRAP Algorithm
(a) Seven tasks with utilizations shown above. These are lined up in arbitrary order, then split at length 1 intervals. **(b)** Each processor runs its task set over a length 10 time slice. Jobs sliced in (a) are seen migrating in (b).

RULES 1-3 of Definition 1 are about as simple a set of criteria as one could hope for. In essence,

"If a job needs to be started now in order to finish on time, then start it. If a job finishes, then stop it. Don't allow idle time in excess of the task set's slack."

These three rules, although an overconstraint when applied at every slice boundary, are obviously necessary to keep tasks hitting their fluid rate curves. Yet, when we require proportional workloads be completed at all system deadlines (DP-FAIR Allocation), they are also *sufficient*. As these rules are so simple, they leave plenty of room to design scheduling algorithms that attempt to reduce the number of context switches and task migrations or address variants of the basic problem model.

4.2. The DP-WRAP Algorithm for Periodic Tasks

We now present our DP-WRAP algorithm. DP-WRAP is a simplification of EKG [3], and is perhaps the simplest possible DP-FAIR scheduler. The algorithm may be visualized as follows. To schedule jobs in σ_j, make a "block" of length δ_i for each T_i, and line these blocks up along a number line (in any order), starting at zero. Their total length will be no more than m. Split this stack of blocks into length 1 chunks at $1, 2, \ldots, m-1$, and assign each chunk to its own processor. Each length 1 chunk of tasks represents the scheduling of tasks on the respective processor; tasks which are sliced in two migrate between their two processors (this task-to-processor scheme is essentially McNaughton's wrap around algorithm [23]). See Figure 5 for an illustration with 7 tasks and 3 processors. To find the actual timing points of context switches (at local WORK

COMPLETE events) within any σ_j, multiply each length 1 segment by L_j.

It is immediately clear from this description that all three DP-FAIR scheduling rules are satisfied. Tasks which migrate are run at the beginning of the slice on one processor, and at the end on the other. So long as such a task has utilization no more than 1 (which is required for *any* feasible schedule), its running times on the two processors will not overlap. We now have the straightforward DP-WRAP scheduling algorithm: compute the context switch times indicated in the diagram (partial sums of task utilizations), reduce modulo 1 for each processor, and multiply times by L_j. Except for this last multiplication, all calculations can be done once as a preprocessing step, so long as the task set is static. Note that there is *no* computational overhead at secondary events: here, a "scheduling event" (which in many algorithms requires iterating through all jobs, performing various calculations, or even sorting them) is merely following a predetermined instruction to replace one task with another on one processor; no "decisions" are made.

Notice that, in general, there will be $m-1$ tasks which are required to migrate. Further, if we repeat a predetermined ordering for each time slice, each of these $m-1$ tasks will migrate *twice* per window: once in the middle, and again at the end, when it moves back to its starting processor. We can cut this number of migrations in half simply by reversing (*mirroring* [3]) the ordering of tasks on each processor in odd-numbered slices. Looking at the example in Figure 5, task 3 runs for the first 0.3 of the window on processor 2, then for the last 0.2 on processor 1. If we reverse the ordering within each processor for the next slice, then task 3 will *start* on processor 1 (for 0.2) and then *finish* on processor 2 (for 0.3).

Theorem 6 *The* DP-WRAP *scheduling algorithm with mirroring in odd slices will produce at most $n-1$ context switches and $m-1$ migrations per slice.*

Proof. With mirroring, context switches and migrations only occur in the middle of a slice, never at the end. In the worst case, every job except the first causes a context switch when it is started, resulting is $n-1$ context switches per slice. There are $m-1$ tasks which migrate once each per slice. □

Various heuristics could be added to improve DP-WRAP's performance in terms of context switches and migrations (e.g., EKG). Instead, we present DP-WRAP in its simplest form to demonstrate how the DP-FAIR scheduling rules can lead to a minimal optimal algorithm, which is both easy to describe and implement, and which requires little computational overhead.

5. DP-FAIR Conditions for Sporadic Tasks and (Un)constrained Deadlines

Due to their simplicity, the DP-FAIR scheduling rules may be extended to various generalizations of the scheduling problem without excess complications. In this section, we will see how to expand DP-FAIR to handle tasks with sporadic job arrivals. We will also consider *constrained* deadlines, where $D_i \leq p_i$ and, consequently, $\delta_i = e_i/D_i$; the case of *arbitrary* deadlines (additionally allowing $D_i > p_i$) is addressed in Section 5.2. We will give more detailed rules for how to allocate workloads within a time slice in these cases; the rules for scheduling in a time slice remain the same, except that we now use the $F_j(t)$ term in RULE 3.

We maintain the (sufficient but no longer necessary) requirements that $\Delta(\tau) \leq m$ and $\delta_i \leq 1\ \forall i$. Because there exist feasible task sets with sporadic arrivals where these conditions are violated[4], and our extended DP-FAIR rules do not handle these cases, DP-FAIR algorithms are no longer optimal. In fact, it has recently been shown that there *can be* no optimal algorithm for sporadic task sets [11]. Thus we will limit ourselves to showing that DP-FAIR algorithms are optimal on task sets with $\Delta(\tau) \leq m$ and $\delta_i \leq 1\ \forall i$.

In the domain of sporadic tasks and constrained deadlines, a task may not use all the capacity reserved for it. Since $\delta_i = e_i/D_i$ when $D_i < p_i$, the time between deadline and next period represents unused capacity. The same is true for the late time between earliest possible and actual arrivals for a sporadic task. During this time (i.e., between $a_{i,h-1} + D_i$ and $a_{i,h}$), we say that a task is *freeing slack* (or *inactive*); a task is *active* between times $a_{i,h}$ and $a_{i,h} + D_i$ (even if it has no work remaining). Thus, for each task, time is partitioned into slack freeing and active periods. Because $\Delta(\tau) \leq m$, T_i "owns" a portion δ_i of the system's total capacity m, even during times when the task is inactive. For this reason, we still attach a task's freed slack to it for accounting purposes, even though this slack goes into the system's general pool of idle processor time.

Similarly to how $\ell_{i,t}$ represents local execution time remaining, we will let $c_{i,t}$ represent *local capacity* remaining for task T_i at time t. Local execution is only consumed (at a rate of 1) when the task is executing; local capacity is consumed either by the task executing (at a rate of 1) or freeing slack (at a rate of δ_i). We define $\alpha_{i,j}(t)$ and $f_{i,j}(t)$ to be the amounts of time that T_i has been active or freeing slack, respectively, during slice σ_j as of time t. We use $f_{i,j}$ and $\alpha_{i,j}$ as shorthands for $f_{i,j}(t_j)$ and $\alpha_{i,j}(t_j)$.

In time slice σ_j, T_i will be allotted a total of $\delta_i \times \alpha_{i,j}$ local execution time (although this must be allocated dy-

[4]For example, $T_1 = (2, 1, 1)$ and $T_2 = (2, 1, 2)$ on $m = 1$ processor is feasible, but has $\Delta(\tau) = 1.5$.

namically as new jobs arrive), and fixed local capacity

$$c_{i,t_{j-1}} \;=\; \delta_i \times L_j \;=\; \delta_i(\alpha_{i,j} + f_{i,j})$$

Finally, we define the *freed slack in σ_j as of time t* to be

$$F_j(t) = \sum_{i=1}^{n} (\delta_i \times f_{i,j}(t)) \quad .$$

We now present two rules for work allocation in our new problem domain.

Definition 2 (DP-FAIR Allocation for sporadic tasks and constrained deadlines) *An algorithm has* DP-FAIR Allocation *if, for every time slice σ_j, local execution is allocated according to the following rules:*

RULE 4: *Initialize $\ell_{i,t_{j-1}}$ to 0. At the beginning time t' of any active time segment for T_i in σ_j (either $t' = t_{j-1}$ or $a_{i,h}$) that ends at time $t'' = \min\{a_{i,h} + D_i, t_j\}$, increment $\ell_{i,t}$ by $\delta_i(t'' - t')$.*

RULE 5: *If a task T_i arrives and has a deadline within the same time slice σ_j, split the remainder of σ_j into two secondary slices σ_j^1 and σ_j^2 so that T_i's deadline coincides with the end of σ_j^1. Divide remaining local execution (and capacity) of all jobs (as well as slack allotment for* RULE 3*) in proportion to the lengths of σ_j^1 and σ_j^2. This rule may be invoked repeatedly / recursively by multiple T_i within σ_j.* $\Diamond$

Since we require $D_i \leq p_i$, an active period for T_i can only end at a task deadline, not the end of a period. Since RULE 5 creates a new slice whenever a deadline appears within an existing slice, all deadlines form the end of some slice. Thus, once a job is active within some slice, it cannot become inactive before the end of that slice. Now, $\delta_i \times \alpha_{i,j}$ work is required of T_i in σ_j, so $\ell_{i,t}$ is incremented by $\delta_i \times \alpha_{i,j}$ whenever T_i becomes active in σ_j (which will be at t_{j-1} if the task starts σ_j with work remaining). Since $c_{i,t_{j-1}} = \delta_i(\alpha_{i,j} + f_{i,j})$, we will have $c_{i,t} > \ell_{i,t}$ so long as T_i is freeing slack, and $c_{i,t} = \ell_{i,t}$ once T_i becomes active.

Lemma 3 from the previous section makes no assumption about deadlines or periodicity, and so is still valid in this extended problem domain. Thus, to prove the correctness of these new DP-FAIR conditions, it only remains to show that $R_t \leq m$ for all $t \in \sigma_j$, and that RULE 5 suffices to meet deadlines introduced in the middle of σ_j.

Lemma 7 *A* DP-FAIR *algorithm cannot cause more than $(S(\tau) \times L_j) + F_j(t)$ units of idle time in slice σ_j prior to time t.*

Proof. Since RULE 3 prohibits *voluntary* idle time in excess of this amount and $F_j(t)$ is a non-decreasing function, we only need to prove that mandatory idle time (when we have fewer jobs with work remaining than processors) cannot force this limit to be broken. Let $I_j(t)$ be the amount of idle time as of time t during slice σ_j. For the sake of contradiction, let t' be the first failure point in σ_j. Since I_j and F_j are continuous functions of t, this means that $I_j(t') = (S(\tau) \times L_j) + F_j(t')$ and $I_j(t' + \epsilon) > (S(\tau) \times L_j) + F_j(t' + \epsilon)$ for all sufficiently small $\epsilon > 0$.

Since tasks can't switch from active to inactive in the middle of a slice, if a task has no work to do at time t', it is either because it has not yet become active, or because it has finished its entire workload for the current slice. We can therefore partition τ into three sets at time t': let A be the set of active tasks with work remaining, B be the set of unarrived (slack freeing) tasks, and C be the set of tasks that have arrived and completed their allotted work for σ_j. For convenience, we will let $\Delta_X = \sum_{i \in X} \delta_i$ for $X \in \{A, B, C\}$.

Based on our definition of local capacity, any task T_i should account for $\delta_i L_j$ processor time during σ_j with a combination of work done and idle time from slack freed. At time t', all freed slack has been consumed as idle time, so tasks in B have used exactly their allotment of processor time so far. C tasks, on the other hand, have already used *all* of their allotted time, having freed their slack (if any) and finished their workloads. That is, they have consumed $\Delta_C L_j$ processor time, and are $\Delta_C(t_j - t')$ *ahead* of their fair share at time t'. Similarly, the static slack pool $S(\tau)L_j$ is already consumed, and so is $S(\tau)(t_j - t')$ ahead of its proportional allotment at time t'. This means that tasks in A must be collectively $(\Delta_C + S(\tau))(t_j - t')$ units *behind* on their use of processor time. If they were keeping up, they would have $\Delta_A(t_j - t')$ work remaining, so as it is they must have exactly

$$\Delta_A(t_j - t') + (\Delta_C + S(\tau))(t_j - t') \;=\; (m - \Delta_B)(t_j - t')$$

work remaining, since $\Delta_A + \Delta_B + \Delta_C + S(\tau) = m$.

Given our definition of t', RULE 3 tells us that we cannot choose to idle processors at time t'. If $|A| \geq m$, then we can run m tasks at time t'. $I_j(t)$ will not immediately increase, contradicting our definition of t'. Thus, we must have $|A| < m$. By RULE 1, we know that each job in A, if left to run on its own processor, will finish its work on time. Thus A can't have more than $|A|(t_j - t')$ work remaining. From above,

$$(m - \Delta_B)(t_j - t') \leq |A|(t_j - t') \;\;\Rightarrow\;\; \Delta_B \geq m - |A| \quad .$$

Tasks in B are freeing slack at a rate of Δ_B at time t'; the system is only adding idle time at a rate of $m - |A|$. Then $F_j(t)$ is growing faster than $I_j(t)$ at time t', and $I_j(t)$ cannot immediately exceed $S(\tau)L_j + F_j(t)$, again contradicting our definition of t'. Since there can be no first point t' of failure, Lemma 7 holds for the duration of σ_j. $\square$

Lemma 8 *If a set τ of sporadic tasks with constrained deadlines is scheduled in σ_j using any* DP-FAIR *algorithm, then $R_t \leq m$ will hold at all times $t \in \sigma_j$.*

Proof. As of time $t \in \sigma_j$, the system has consumed $m(t - t_{j-1})$ capacity, either by executing jobs, or by idling. If it has idled for $I_j(t)$ time units by time t then Lemma 7 gives $I_j(t) \leq S(\tau)L_j + F_j(t)$. If we let $w_{i,j}(t)$ be the work executed on task T_i during σ_j as of time t, then we have

$$m(t - t_{j-1}) \;=\; I_j(t) + \sum_{i=1}^{n} w_{i,j}(t) \;\;, \tag{1}$$

and

$$\begin{aligned} c_{i,t} \;&=\; c_{i,t_{j-1}} - w_{i,j}(t) - \delta_i f_{i,j}(t) \\ &=\; \delta_i L_j - w_{i,j}(t) - \delta_i f_{i,j}(t) \;\;. \end{aligned} \tag{2}$$

Recalling that $r_{i,t}(t_j - t) = \ell_{i,t} \leq c_{i,t}$,

$$\begin{aligned} R_t(t_j - t) \;&=\; \sum_{i=1}^{n} \ell_{i,t} \;\leq\; \sum_{i=1}^{n} c_{i,t} \\ &=\; \sum_{i=1}^{n} (\delta_i L_j - w_{i,j}(t) - \delta_i f_{i,j}(t)) && \text{by (2)} \\ &=\; \Delta(\tau) L_j - \sum_{i=1}^{n} w_{i,j}(t) - F_j(t) \\ &\leq\; (m - S(\tau)) L_j - \sum_{i=1}^{n} w_{i,j}(t) + (S(\tau)L_j - I_j(t)) \\ &=\; mL_j - (m(t - t_{j-1})) && \text{by (1)} \\ &=\; m(t_j - t) \end{aligned}$$

and we see that $R_t \leq m$, as desired. □

Theorem 9 *Any* DP-FAIR *scheduling algorithm is optimal for sporadic task sets with constrained deadlines where $\Delta(\tau) \leq m$ and $\delta_i \leq 1 \; \forall i$.*

Proof. As in Theorem 5, all tasks finishing their local workloads at the end of time slices ensures that they hit their fluid rate curves at their deadlines, i.e., they don't miss their deadlines. The only remaining questions are whether a task which arrives and has its deadline within a slice will meet this deadline, and whether RULE 5 will interfere with other tasks completing their workloads.

If task T_i has an arrival at time $t' = a_{i,h}$ in σ_j, then $m(t' - t_{j-1})$ work and idle time have been consumed thus far in the slice, and the remaining capacity of $m(t_j - t')$ is exactly enough to complete each task's allotment of $\delta_i L_j$ plus consume the $S(\tau)L_j$ static slack. If we create subslices $\sigma_j^1 = [t', t'')$ and $\sigma_j^2 = [t'', t_j)$, where $t'' = a_{i,h} + D_i$, and divide remaining work for each task and idle time between these subslices proportionally to their lengths, then each subslice will have been given a work/slack load exactly equal to its capacity. Lemmas 3 and 8 prove that, by following a DP-FAIR slice scheduling policy, these workloads will be successfully completed. Any other task $T_{i'} \neq T_i$ has its remaining work divided between σ_j^1 and σ_j^2, and so it is finished by the end of σ_j, as it requires.

As for T_i, since it has been freeing slack prior to t', it has exactly $\delta_i(t_j - t')$ capacity reserved for the remainder of σ_j. T_i claims the proper proportion $\delta_i(t'' - t') = \delta_i D_i = e_i$ of this capacity for execution in σ_j^1, and gets exactly enough work done to meet its deadline. □

5.1. Modifying DP-WRAP

Modifying DP-WRAP to handle arrivals within a time slice is fairly straightforward. If a task T_i generates a job at time t' within time slice σ_j and $t' + D_i \geq t_j$, then we allocate execution time $\ell_{i,t'} = \delta_i(t_j - t')$, as per RULE 4. This execution is wrapped onto the end of the existing schedule without otherwise impacting the schedule for σ_j.

If $t' + D_i < t_j$, then we need to split the remainder of σ_j into two subslices σ_j^1 and σ_j^2 according to RULE 5. T_i's workload is given entirely to σ_j^1. The remaining workloads of all other tasks are divided proportionally between σ_j^1 and σ_j^2. Each subslice is scheduled as described in Section 4.2.

5.2. Arbitrary Deadlines

Let us now consider the problem where deadlines can be larger than periods via the following example.

Example Consider the periodic task set τ on $m = 2$ processors where $T_1 = (6, 4)$ and $T_2 = T_3 = T_4 = T_5 = (3, 1, 6)$. Since $\Delta(\tau) = 4/6 + 4(1/3) = 2$, and $4 + 4 \times 2 \times 1 = 12$ units of work must be done by time 6 in order to meet our time slice deadlines, the system can allow no idle time. However, if we run T_2 then T_3 to completion on the first processor and T_4 then T_5 on the second, then at time 2, tasks $T_2, \ldots, T_5$ are out of work. Only at this point is T_1 forced to run by zero laxity. Until more work arrives for $T_2, \ldots, T_5$ at time 3, the other processor sits idle, implying eventual failure by Theorem 1. ◊

Allowing deadlines longer than periods breaks the "global knowledge" granted by giving all tasks the same deadlines. Fortunately, the problem is easily solved. If we are given a task where $D_i > p_i$, we simply impose an artificial deadline of $D_i' = p_i$. This doesn't increase the task's density δ_i, and if the artificial deadline is met, the real one will certainly be also. However, these artificial deadlines might force unnecessary slice boundaries. In the absence of artificial deadlines, if a task were to finish its workload in some slice prior to its deadline, then that period of the task wouldn't create any slice boundary. Increasing the number of time slices, in turn, incurs additional overhead from the added context switches and migrations.

5.3. Some Simplifications

RULE 5's time slice splitting could be very complicated, particularly if it is done recursively for several tasks within a single time slice. We can avoid ever having to split time slices in this manner by ensuring time slices are never longer than the minimum deadline. Like our solution for arbitrary deadlines, this simplifies scheduling but increases context switches and migrations.

We could also simplify RULE 3 by replacing it with sufficiently strong heuristics. A simple one is "*Never allow a processor to idle if there are tasks waiting to execute*." A somewhat less restrictive rule is "*At all times t, at least $\lceil R_t \rceil$ tasks are executing jobs*." These rules would be easier to implement in practice, and also satisfy RULE 3.

6. Related Work

We now examine some recent algorithms in the context of DP-FAIR. Unless otherwise noted, the following algorithms only address periodic task sets with implicit deadlines. PFAIR [5] was the first optimal multiprocessor scheduler. It uses a very strict notion of proportional fairness to target fluid rate curves at every multiple of a discrete time quantum, and incurs a large overhead in context switches and migrations. The 2003 BF Algorithm [26] appears to be the first use of deadline partitioning for real-time scheduling. BF modifies PFAIR, and is still quantum-based. Because of the resultant integer rounding, workload assignments aren't quite DP-FAIR, but the scheme is DP-CORRECT and closely resembles DP-WRAP. It also matches our early insight [6] that fluid rate targets are only important at deadlines, not at every time quantum.

The 2006 LLREF [9] and EKG [3] algorithms were the first optimal schedulers that were not quantum-based, and most subsequent work has been an extension of one of these two models. LLREF is a strictly DP-FAIR algorithm, but does unnecessary work: at each local ZERO LAXITY or WORK COMPLETE event, it resorts all jobs, and executes those m with least laxity. However, it does introduce the "T-L Plane" visualization, which can be very instructive in thinking about slice scheduling.

EKG is very similar to our DP-WRAP for periodic tasks, but with two improvements. First, the non-migrating tasks assigned to a given processor are scheduled with uniprocessor EDF instead of McNaughton's wrap around algorithm [23]. This reduces context switching since some of a processor's tasks may not run during a slice, but adds some computational complexity. While this means that work allocation in time slices is not DP-FAIR, it is easy to verify that EDF will correctly schedule these non-migrating tasks. EKG's other improvement only applies to task sets with $\Delta(\tau) < m$. If τ has enough slack, it allows the "end" segments of some processors to be left idle, instead of partially assigning a task which will wrap and migrate. This allows subsets of processors to be scheduled independently, meaning that any task's deadline will only impose slice overhead on the tasks in its processor subgroup. This may be controlled with a tunable parameter k, which gives partitioned EDF in one extreme ($k = 1$), and an optimal scheduler almost identical to DP-WRAP in the other ($k = m$).

Following these two works, numerous other algorithms have appeared that expand upon them, either to provide improvements or to address variants of the basic periodic scheduling problem. Andersson et al. [1, 2] provide a pair of EKG variant algorithms for handling sporadic tasks and arbitrary deadlines, but use fixed width (instead of deadline bounded) time slices. The Ehd2-SIP [16] and EDDP [17] algorithms are also similar to EKG, but sacrifice optimality in favor of improved general performance. While they fail to schedule some feasible task sets, they have a high success rate until $\Delta(\tau)$ reaches the 80-90% range.

Subsequent improvements to LLREF have all done away with some of that algorithm's unnecessary scheduling overhead. Funaoka et al. present the E-TNPA [13] and TRPA [12] algorithms which, for task sets with $\Delta(\tau) < m$, fill the idle time in a slice with work from future slices; that is, they are *work conserving* (they never allow an idle processor when there's a job available to run on it). Thus, only their slice scheduling (not their allocations) are DP-FAIR. Like LLREF, neither gives a prescription for assigning tasks to processors, so it is difficult to gauge their real overheads. Chen et al. [8] extend the T-L Plane model to handle the extended problem of *uniform multiprocessors* (where processors run at different speeds, but treat all tasks uniformly). Based on this, they develop PCG, the first optimal scheduler for uniform multiprocessors. Funk et al. [14] extend LLREF with LRE-TL to handle sporadic as well as periodic tasks, and remove the sorting overhead from each scheduler invocation. They also extend their work to uniform multiprocessors.

6.1. Comparisons With DP-WRAP

Because DP-WRAP is designed to be simple and instructive, not optimized for performance, we have not undertaken extensive side-by-side comparisons with existing algorithms. Early simulations show that DP-WRAP causes about 1/3 as many context switches and migrations as LLREF. However, other papers have noted LLREF's inefficiencies [13, 14], and we would expect BF and EKG to show improvements comparable to DP-WRAP. We expect EKG (with appropriately tuned k parameter) to outperform DP-WRAP and BF on task sets with $\Delta(\tau) < m$. Additional simulation comparisons may be performed as necessary to test future refinements to DP-WRAP.

In terms of algorithmic complexity, DP-WRAP has the clear advantage. It does $O(n)$ work at the beginning of each

slice to determine switching and migration times, and then each event just requires a constant time lookup. For periodic task sets with implicit deadlines, every slice is equivalent. Thus, the only work needed at the beginning of a slice is multiplying the reusable schedule by the length of the slice, giving minimal overhead. Scheduling complexity per slice for LLREF and LRE-TL are $O(n^2)$ and $O(n \log n)$, respectively. EKG also has a worst-case $O(n \log n)$ per slice complexity due to its EDF subroutine, but is more efficient in practice. BF is $O(n)$ per slice, (like DP-WRAP, BF does its slice scheduling up front), but each slice is scheduled differently, and the complexity due to time quantum rounding is high.

7. Conclusion

There have been a number of recent advances in scheduling algorithms for periodic task sets in hard real-time, multiprocessor environments. A recognition of their shared traits and insights, and an underlying theory to explain their success, were previously missing from the literature. This paper provides such a theory. We started by examining the inherent problems of older, greedy approaches and used this to motivate the simple DP-FAIR conditions for optimal scheduling of periodic tasks. We demonstrated the power and flexibility of DP-FAIR by describing the simplest optimal scheduler to date, DP-WRAP, and by extending the DP-FAIR rules to sporadic tasks with arbitrary deadlines. We hope that our model aids in understanding past work, and contributes to the direction of future research.

References

[1] B. Andersson and K. Bletsas. Sporadic Multiprocessor Scheduling with Few Preemptions. *Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.

[2] B. Andersson, K. Bletsas, and S. K. Baruah. Scheduling Arbitrary Deadline Sporadic Task Systems on Multiprocessors. *IEEE Real-Time Systems Symposium (RTSS)*, 2008.

[3] B. Andersson and E. Tovar. Multiprocessor Scheduling with Few Preemptions. *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2006.

[4] S. K. Baruah and J. Carpenter. Multiprocessor Fixed-Priority Scheduling with Restricted Interprocessor Migrations. *Journal of Embedded Computing*, 1(2):169–178, 2004.

[5] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. Varvel. Proportionate Progress: A Notion of Fairness in Resource Allocation. *Algorithmica*, 15(6):600–625, 1996.

[6] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time, and Non-Real-Time Processes. *IEEE Real-Time Systems Symposium (RTSS)*, 2003.

[7] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. K. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*, pages 30.1–30.19. Chapman Hall/CRC, 2004.

[8] S.-Y. Chen and C.-W. Hsueh. Optimal Dynamic-priority Real-Time Scheduling Algorithms for Uniform Multiprocessors. *IEEE Real-Time Systems Symposium (RTSS)*, 2008.

[9] H. Cho, B. Ravindran, and E. Jensen. An Optimal Real-Time Scheduling Algorithm for Multiprocessors. *IEEE Real-Time Systems Symposium (RTSS)*, 2006.

[10] S.-K. Cho, S. Lee, A. Han, and K.-J. Lin. Efficient Real-Time Scheduling Algorithms for Multiprocessor Systems. *IEICE Transactions on Communications*, E85-B(12):2859–2867, 2002.

[11] N. Fisher, J. Goossens, and S. Baruah. Optimal Online Multiprocessor Scheduling of Sporadic Real-Time Tasks is Impossible. *Real-Time Systems, to appear*, 2010.

[12] K. Funaoka, S. Kato, and N. Yamasaki. New Abstraction for Optimal Real-Time Scheduling on Multiprocessors. *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.

[13] K. Funaoka, S. Kato, and N. Yamasaki. Work-Conserving Optimal Real-Time Scheduling on Multiprocessors. *Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.

[14] S. Funk and V. Nadadur. LRE-TL: An Optimal Multiprocessor Algorithm for Sporadic Task Sets. *Conference on Real-Time and Network Systems (RTNS)*, 2009.

[15] K. S. Hong and J. Y.-T. Leung. On-Line Scheduling of Real-Time Tasks. *IEEE Transactions on Computers*, 41:1326–1331, 1992.

[16] S. Kato and N. Yamasaki. Real-Time Scheduling with Task Splitting on Multiprocessors. *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2007.

[17] S. Kato and N. Yamasaki. Portioned EDF-based Scheduling on Multiprocessors. *ACM International Conference on Embedded Software (EMSOFT)*, 2008.

[18] J. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4(1):209–219, 1989.

[19] C. Lin and S. A. Brandt. Improving Soft Real-Time Performance Through Better Slack Management. *IEEE Real-Time Systems Symposium (RTSS)*, 2005.

[20] C. Lin, T. Kaldewey, A. Povzner, and S. A. Brandt. Diverse Soft Real-Time Processing in an Integrated System. *IEEE Real-Time Systems Symposium (RTSS)*, 2006.

[21] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[22] J. M. López, M. Garcia, J. L. Diaz, and D. F. Garcia. Worst-case Utilization Bound for EDF Scheduling on Real-Time Multiprocessor Systems. *Euromicro Conference on Real-Time Systems (ECRTS)*, 2000.

[23] R. McNaughton. Scheduling with Deadlines and Loss Functions. *Machine Science*, 6(1):1–12, October 1959.

[24] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Massachusetts Institute of Technology, 1983.

[25] A. Srinivasan, P. Holman, J. H. Anderson, and S. K. Baruah. The Case for Fair Multiprocessor Scheduling. *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2003.

[26] D. Zhu, D. Mossé, and R. Melhem. Multiple-Resource Periodic Scheduling Problem: how much fairness in necessary? *IEEE Real-Time Systems Symposium (RTSS)*, 2003.