UNIVERSITY OF CALIFORNIA

SANTA CRUZ

OLD AND NEW APPROACHES TO OPTIMAL REAL-TIME MULTIPROCESSOR SCHEDULING

A dissertation submitted in partial satisfaction
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Gregory M. Levin

September 2013

The Dissertation of
Gregory M. Levin
is approved:

Professor Scott Brandt,
Chair

Professor Carlos Maltzahn

Professor David Helmbold

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Old and New Approaches to Optimal
Real-Time Multiprocessor Scheduling

by

Gregory M. Levin

We consider the problem of scheduling a collection of processes, or tasks, on a multiprocessor platform. The tasks in question have *real-time* computing requirements, meaning that they have frequent processing deadlines that must be met. We are interested in *optimal* scheduling algorithms, which find a correct schedule for a set of tasks whenever it is possible to do so.

Recent work in the field has used the notions of *fluid scheduling* and *deadline partitioning* to guarantee optimality and improve performance. In the first part of this dissertation, we develop a unifying theory of existing approaches with the DP-Fair scheduling policy, and show how it easily proves and explains existing approaches. 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.

While easy to understand and implement, DP-Fair algorithms have the drawback of incurring a significant overhead in preemptions and migrations. In the second part of this dissertation, we present RUN, the first optimal algorithm which does not rely on a DP-Fair approach. Instead, RUN reduces the multiprocessor problem to a series of much easier uniprocessor problems. RUN's average preemptions per job never exceeded 3 in any simulation, and has a provable upper bound of 4 for most task sets. It also reduces to Partitioned EDF whenever a proper task-to-processor partitioning is found, and significantly outperforms all existing optimal algorithms.

This work is dedicated

to the best parents I know.

(Those would be mine, in case you were wondering.)

Academic Acknowledgments

This work is the result of a number of collaborations, and would not exist without the help of all these fine people. First and foremost is my advisor Scott Brandt. From its beginning as a class project right through its defense, this work was made possible by his support, encouragement, and insight. He has been a pleasure to work with both personally and professionally, and even more importantly, he took me on two free trips to Europe!

Thanks to Caitlin Sadowski and Ian Pye for their early collaborations on this work, back when it was just a class project. Caitlin in particular got the project rolling, provided many key insights, and kept things moving through all our work on DP-Fair.

Thanks to Shelby Funk, an unintended coauthor. Unbeknownst to us, Shelby was making the same discoveries as us at the same time. After we submitted what were essentially two copies of the same paper to the same conference, we got together to combine these two papers into one superior one. She brought new ideas to the project, as well as invaluable wisdom and experience. She also forever improved my PowerPoint presentations with a few insightful comments.

Thanks to Paul Regnier, George Lima, and Ernesto Massa from Brazil. Prior to their insights and innovations, scheduling duality was just a curiosity. They turned it into something fantastic, and allowed me to join them in sharing it with everyone else.

Finally, thanks to Carlos Maltzahn and David Helmbold for sitting on my defense committee, and shepherding this work through its final stages.

Personal Acknowledgments

First and foremost, thanks to my family, for everything: to Mom, for her love and support; to Keith, for being more awesome than can reasonably be expected of a little brother; and to Dad, who we lost along the way.

Enormous love and thanks to Caitlin, Ian, and Jaeheon, for being my Santa Cruz family for the last seven years; for cooking me hundreds of meals; for keeping me company through work, play and travel; and for making me happy and at home every single time I'm around them.

Thanks to Renee N for friendship, intellectual geekery, and for frequent encouragement and prodding to get this thing done. Thanks to David, Gillian, and the GNoD group for fun, movies, and games. Thanks to Alicia and Brian for a great place to live, great food, and great company. Thanks to Krista for always being there, and for just generally being Krista. Love and thanks to Cj, for all her love and support, and for starting me off on this endeavor. And thanks to Avani for the push I needed at the beginning.

Thanks to old teachers John Jackson, Donna Williams, Art Benjamin, and Ed Scheinerman, who got me from grade school through the first Ph.D. with their wisdom, support, and enthusiasm for mathematics. Great teachers make all the difference, and they were the best.

And finally, thanks to Renee H for love, support, travel, adventure, games, pottery, inspiration, chaos, and other wonderful things too numerous to mention. And for being my best friend for the last five years.

Chapter 1

Introduction

This dissertation is focused on pushing the theoretical state of the art in optimal real-time multiprocessor scheduling. First, we will understand existing solutions to the problem by describing a simple theory that clarifies and unifies known optimal algorithms. We will then present a brand new approach, which defies this theory and greatly outperforms all previous approaches.

1.1 Motivation

Multiprocessor systems are becoming more and more common, as not just personal computers but even tablets, smart phones, and small embedded systems are being produced with multiple cores. Many of the implications of a multiprocessor system, including scheduling issues, are still not well understood. Multiprocessor scheduling (the scheduling of various tasks on multiple processors) is particularly difficult in the presence of real-time constraints (when tasks have recurring execution deadlines that must be met). Real-time scheduling algorithms that are known to perform very well on uniprocessor systems, such as **E**arliest **D**eadline **F**irst (EDF) [33], 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 of the processors and allow tasks to migrate between processors. Partitioned algorithms initially partition tasks among processors, and then schedule each processor with a simpler uniprocessor scheduling algorithm; task migration is not allowed. *Partitioned EDF* is a simple example: after assigning tasks to processors, it simply runs uniprocessor EDF independently on each processor. Most previous algorithms (*e.g.,* [8, 29]) prefer this partitioned approach, since uniprocessor scheduling is much easier to implement. However, partitioned algorithms are not optimal, in the sense that they can fail to schedule theoretically feasible task sets. Some task sets simply cannot be divided among the available processors,

even though they are within the system's available capacity when migration is allowed. And as the partitioning of tasks among processors is essentially the NP-Complete bin packing problem, finding a viable partition can be challenging. In the worst case, examples may be constructed where partitioned schedulers fail to successfully schedule tasks sets that only require $(50 + \epsilon)\%$ of processor capacity [8, 35].

Given the large potential for inefficiencies in partitioned approaches, there has been much interest in recent years in global schedulers, and in particular in optimal scheduling algorithms. In 1996, Baruah *et al.* [4] introduced the PFAIR algorithm, the first optimal multiprocessor scheduler for periodic real-time tasks. PFAIR uses *proportional fairness* to ensure that tasks receive processor time roughly proportional to their rates, not just at their deadlines, but at all times. By migrating tasks between processors, PFAIR can successfully schedule any task set whose execution requirement does not exceed processor capacity.

More recently, a number of algorithms 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, 10, 47, 48]. These algorithms enforce proportional fairness only at these shared deadlines, greatly reducing the overhead of preemptions and migrations. Subsequent papers have expanded on these basic models. All these algorithms, while superficially different, have achieved optimality by tracking the *fluid schedule* (average rate curve) of each task. Until now, there has been little apparent recognition of the power and simplicity of the theory that underlies their successes. The first part of this work explores the commonality of these various optimal algorithms, and provides a unifying theory of their behavior. The second part will go beyond this theory, and introduce a much more efficient optimal algorithm that is not tied to this old paradigm.

1.2 Organization

This dissertation is divided into four chapters. This first chapter provides some background and motivation on the problem at hand, and introduces the problem model and notation.

The second chapter details DP-Fair, a simple, elegant theory that encompasses and explains all previous solutions to the scheduling problem at hand. Section 2.1 motivates our approach by examining the shortcomings of greedy approaches to this problem. Section 2.2 introduces the theory of DP-Fair scheduling for periodic task sets. DP-Fair consists of three simple, almost obvious rules which guarantee the correctness of any scheduling algorithm which adheres to them. It also describes the DP-Wrap algorithm, the simplest optimal scheduling algorithm to date. Section 2.3 extends our DP-Fair rules and the DP-Wrap algorithm to sporadic tasks with arbitrary deadlines, and the even more general problem model of unrelated, aperiodic jobs. Section 2.4 surveys previous multiprocessor scheduling algorithms, and shows how they may be easily described within the context of the DP-Fair scheduling rules.

The third chapter introduces RUN, the first optimal multiprocessor scheduling algorithm which does not adhere to the DP-Fair scheduling rules. By placing much smaller overconstraints on schedules, it incurs only about a fifth as many preemptions and migrations as other optimal algorithms. Section 3.1 introduces *scheduling duality*, and shows how this provides a different approach to overcoming the flaws of greedy schedulers. Section 3.2 provides some needed additional notation and system modeling. Section 3.3 introduces *severs*, which aggregate and schedule multiple tasks as one. Section 3.4 describes the off-line process whereby the RUN algorithm reduces the multiprocessor scheduling problem to a set of much simpler uniprocessor scheduling problems. Section 3.5 describes how RUN then translates the on-line schedules for these uniprocessor systems into a schedule for the original multiprocessor system. Section 3.6

provides some theoretical bounds on the performance of the RUN algorithm, and also the results of side-by-side simulation comparisons with other optimal algorithms.

The fourth chapter considers possible directions for future work, and summarizes our contributions. Appendix A provides a table summarizing the notation used throughout this dissertation. Appendix B discusses several bin packing algorithms that may be used as subroutines of RUN. Appendix C provides a proof that the problem of finding a feasible schedule with fewest migrations is NP-Complete.

1.3 Background and Notation

We consider the scheduling of n periodic [33] or sporadic [14, 15] 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. All work needing to be executed will be part of some job. Formally, a *real-time job* (henceforth, just "job") is a finite sequence of instructions which become available for execution at some particular time, and which must be completed by some subsequent time. Since jobs generally represent small amounts of work needed by some larger process, we collect a related sequence of jobs into a *task*. As an example, a DVD player program might have a playback task, and 24 times per second, that task will release a job which requires some amount of processor time to decode compressed data and translate it into one frame on the screen. Any job which misses its deadline results in a frame being displayed late or not at all.

Given a collection of such 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 once. We assume no dependence between tasks (the order of execution of tasks is unimportant). We allow *context switches* (replacing one executing task with another on some processor) and *migrations* (moving a

task from one processor to another) at any time. We say that a task is *preempted* if it is replaced on its processor by another task before the work of its current job is complete. Note that any preemption is also a context switch, but not vice versa.

Even though computers operate on discrete clock cycles and atomic CPU instructions, we will generally model time as continuous. For convenience, however, in our examples and our simulation code, we take all job release times, workloads, and deadlines to be integers. This makes examples easier to read, and our code easier to describe and maintain. Our continuous model of time permits us to measure processor time as intervals instead of counting clock ticks, and we allow context switches and migrations to occur at any time, not just at integral times.

We wish to treat all jobs of a task as identical, but in practice, workloads may vary from one job to the next. Further, we may not know beforehand the exact execution requirement of each job. Instead, we will just assume that we know the *worst case execution time* (WCET), or upper bound, of a job. By assuming a single WCET for all jobs from a task, we may treat all jobs of a task as identical. If a job falls short of its WCET estimate, we may switch to a different task sooner, or simply idle its processor to fill in the excess time. We will henceforth simply assume that all jobs do work exactly equal to their WCET. Now let us be more precise about describing our tasks.

Definition 1.1 (Task)**.** A *task* $\tau_i = (p_i, c_i, \delta_i)$ is a process that invokes an infinite sequence of identical jobs $\{J_{i,h}\}_{h \geqslant 1}$. A task is characterized by three quantities: its *period* p_i, which represents the minimum interarrival time of consecutive jobs; its *work* c_i, which is the worst case execution time of each job; and its *duration* δ_i, which is the length of time between a job's release and its deadline. □

We say that a job $J_{i,h}$ of τ_i *arrives* or is *released* (we use the terms interchangeably) when it becomes available for execution. We denote the arrival time of $J_{i,h}$ by $a_{i,h}$, so that its deadline occurs at time $a_{i,h} + \delta_i$. Thus $J_{i,h}$ must be allowed to execute

for c_i time units during the interval $[a_{i,h}, a_{i,h} + \delta_i)$. If $\delta_i = p_i$, we refer to this as an *implicit deadline* and, dropping the implicit δ_i, use the abbreviated notation $\tau_i = (p_i, c_i)$; otherwise, we say the task has an *arbitrary deadline*. If τ_i is a *periodic task*, then its first job arrives at time $t = 0$ and all its remaining jobs arrive exactly p_i time units apart, *i.e.,* $a_{i,h} = (h-1)p_i$ for all h. If τ_i is a *sporadic task*, then its first job may arrive at any time $t \geqslant 0$ and the remaining jobs arrive no less than p_i time units apart, *i.e.,* $a_{i,1} \geqslant 0$, and $a_{i,h} \geqslant a_{i,h-1} + p_i$ for all $h > 1$. We let $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ denote a set of n periodic or sporadic tasks.

One important characteristic of a task τ_i is its *rate* ρ_i, which for tasks with implicit deadlines is $\rho_i = c_i/p_i$ (sometimes referred to as its *utilization* or *density*). For periodic tasks, the rate measures the proportion of time a task executes on average. For sporadic tasks, the rate measures the "worst-case average", *i.e.,* the average proportion of required computing time assuming a soon-as-possible sequence of arrivals ($a_{i,h} = a_{i,h-1} + p_i$). When deadlines are not equal to periods, we define the task's rate to be $\rho_i = c_i / \min\{p_i, \delta_i\}$. Notice that if $\delta_i = p_i$ then this matches the definition for implicit deadlines. The total rate of task set $\mathcal{T}$, denoted $\rho(\mathcal{T})$, is the sum of the individual rates:

$$\rho(\mathcal{T}) = \sum_{i=1}^{n} \rho_i \quad .$$

In general, the rate is the proportion of one processor that we will need to allocate to a task. The *utilization* of a task set on a system is the fraction of processing resources which it requires, *i.e.,* $\rho(\mathcal{T})/m$. We will generally be interested in task sets with 100% utilization, that is, $\rho(\mathcal{T}) = m$.

Formally, a *schedule* is a function which, at all non-negative times, assigns to each processor zero or one tasks[1]. We will only concern ourselves with *legal* schedules,

[1]In Chapter 3, we will present a slightly different formal definition of "schedule", where task-to-processor assignment is not specified. There we will only specify *which* tasks are running at any given time, and then deal with task-to-processor assignment separately. This is only a matter of convenience; these differences are minor, and easy to reconcile formally if need be.

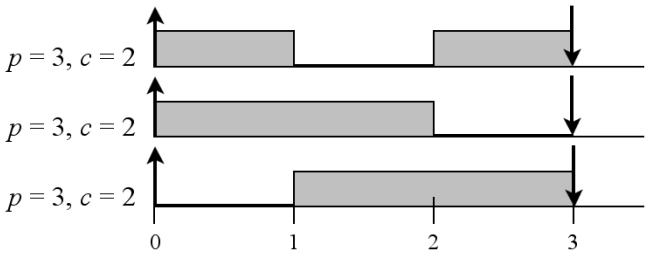


Figure 1.1: **Simple Scheduling Problem**
Three tasks, each with a rate of 2/3, can run successfully on two processors with migration. Up arrows indicate arrivals; down arrows indicate deadlines.

in which no task is assigned to more than one processor at a time, and where tasks are only assigned when they have some outstanding job with work remaining. Further, we are only interested in *valid* schedules, which are legal and in which 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 finds a valid schedule for every feasible task set. A simple example depicted in Figure 1.1 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. We say that a scheduling algorithm is *on-line* if the arrival times of jobs are not known ahead of time. This distinction is relevant only when we consider sporadic tasks (by definition, the future behavior of periodic tasks is known in advance).

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. Any system will require some amount of time for a processor to change the context (memory, cache, etc.) from one task to another, or to move the context of a task from one processor to another. Ultimately, we will use the number of migrations and context switches observed as the metric with which we compare various scheduling algorithms. It is also instructive to compare the number of *preemptions* suffered; a context switch when a job's work is complete is a necessary part of scheduling, whereas the preemption of an uncompleted

job is strictly a scheduler-based decision. Consequently, preemption counts can provide a better picture of how much extra overhead the scheduler itself is adding. Although highly system-dependent, task migrations generally take longer than context switches (sometimes prohibitively longer), but both operations consume system time. Because global scheduling algorithms migrate tasks and also tend be complex (and, therefore, have long run times), partitioned schemes are preferred in practice. Newer multiprocessor architectures, such as multicore processors, have significantly reduced migration overhead. The preference for partitioned scheduling may no longer be necessary in these environments.

Initially, we will focus on periodic tasks with implicit deadlines, and save the more general cases for Section 2.3. Ironically, although our primary goal is to minimize context switches and migrations, we will follow the conventions of other recent papers in the field and assume that these operations are "free", *i.e.,* that they occur instantly[2]. Under this assumption, 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 ($\rho(\mathcal{T}) \leqslant m$),

(ii) No task's workload exceeds its period or deadline ($\rho_i \leqslant 1 \ \ \forall i$), and

(iii) Process migration is allowed.

Given unlimited context switching and migration, it is not hard to see that these constraints are sufficient in our theoretical model. In fact, this is just an extension of the uniprocessor case presented by Liu and Layland [33]. Imagine that we can reschedule our jobs after each ϵ of time. As $\epsilon \to 0$, we can turn each task τ_i on or off sufficiently often so that it appears to be running continuously on a fraction ρ_i of a processor. In the limit, each job executes at exactly its necessary rate and, when all

[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 context switches and migrations can be determined in advance, which is often the case. However, it could lead to very pessimistic worst case execution times.

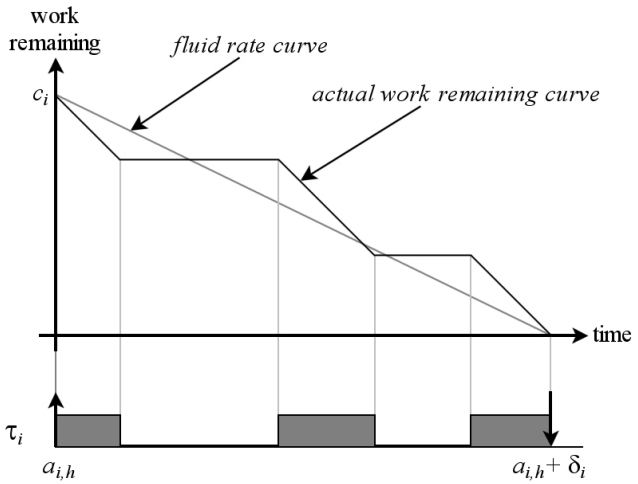


Figure 1.2: **Fluid versus Actual Schedules**
The fluid rate curve decreases continuously at a rate of ρ_i, while the actual work remaining curve only decreases (at a rate of 1) while the task is executing.

rates sum to no more than m, all jobs finish on time. Srinivasan *et al.* [45] refer to this continuous fractional execution as the *fluid* scheduling model. Figure 1.2 shows the the fluid and actual scheduling of one job from task τ_i. The vertical axis shows work remaining, which starts at c_i when the job is released, and decreases to 0 by the deadline. The fluid rate curve decreases continuously at a rate of ρ_i; the actual work remaining curve decreases at a rate of one when τ_i is executing, and is horizontal when τ_i is idle.

Finally, if context switches and migrations are free, 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 1.1 & 2.1). In fact, some recent algorithms give no explicit prescription for how to assign tasks to processors [10, 21].

Determining the feasibility of a periodic task set 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. Unfortunately, finding a valid schedule with the fewest possible migrations is NP-Complete (see Appendix C), so we will have to content ourselves with heuristics and comparisons to other efforts. Understanding existing schedulers, and finding more efficient ones, is the primary focus of this work.

1.4 Contributions

Prior to the work in this dissertation, the existing optimal real-time multiprocessor scheduling algorithms were seen as diverse and complex. There was no apparent discussion of their similarity, and the proofs of their correctness were long and involved. Except for the original algorithm PFAIR, which is highly over-constrained, optimal algorithms all rely on deadline partitioning (or sharing), but otherwise appear to have substantially different approaches. Our introduction of DP-FAIR theory greatly simplifies this field of work. It provides three obviously necessary scheduling rules that, when used with deadline partitioning, also prove sufficient to guarantee optimality. It turns out that all previous optimal deadline partitioning algorithms were following these rules implicitly without acknowledging them (or deviating slightly in ways that clearly didn't break their correctness). Further, the proof of DP-FAIR requires only half a page in the original publication [30], making the theory of optimal scheduling much more accessible. We also present the DP-WRAP algorithm, which is the simplest optimal algorithm to date, and which serves as a sort of least common ancestor, structurally speaking, to previous algorithms. The end result is a new unifying theory that easily proves the correctness of previous algorithms, and along with comparisons to DP-WRAP, greatly simplifies their understanding. Additionally, this simple new approach to optimal scheduling provides a light framework on which to build scheduling solutions to more general problem domains.

Unfortunately, deadline partitioning requires certain overconstraints on the

schedule which lead to considerable inefficiencies. However, prior to the work in this dissertation, no other approach to optimal scheduling was known. Our RUN algorithm is the first optimal algorithm not bound by the constraints of deadline partitioning, and represents a new direction in the theory of optimal scheduling. Further, because it is not bound by these usual constraints, it is, on average, five times more efficient that any previous approach. It also reduces naturally to a partitioned scheduling approach on any task set for which partitioned scheduling is viable, making it a single ideal solution to optimal scheduling in all cases.

Chapter 2

DP-Fair: Understanding the Past

2.1 What's Wrong with Greedy Schedulers?

2.1.1 Greedy Scheduling Algorithms

In order to motivate our approach in this section, we will examine the shortcomings of greedy schedulers, which are preferred for uniprocessors and partitioned algorithms. Greedy schedulers are an attractive and common first approach to scheduling [8] because they are straightforward to explain and implement. They often attempt to encapsulate the criticality (likeliness of a missed deadline) of a job into a single characteristic and then use that to greedily schedule jobs.

Several successful greedy algorithms have been found for uniprocessor scheduling. Two of the earliest are from a seminal paper by Liu *et al.* [33]. The first is Rate Monotonic (RM) scheduling, which statically sorts jobs by their rates, and guarantees feasible schedules for up to $\sim 70\%$ processor utilization. 100% utilization in guaranteed by their second algorithm, Earliest Deadline First (EDF). Here, whenever jobs are completed or introduced, the job with the earliest deadline is selected to run. While EDF is optimal on a uniprocessor, it is suboptimal in a multiprocessor environment [16]. A task set where EDF fails to find a feasible schedule is depicted in Figure 1.1 (the necessary rescheduling at time $t = 1$ does not correspond to the introduction or completion of any job).

Another well studied greedy scheduling algorithm which is optimal on one processor is preemptive **L**east **L**axity **F**irst (LLF), initially introduced as the least slack algorithm [40]. LLF always runs the job with least *laxity*: time remaining until the next deadline minus time required for the remaining workload (*i.e.,* allowable idle time). Since LLF requires laxity to be recomputed at every clock tick, it has a large overhead, and causes numerous context switches when two jobs have the same laxity. MLLF [42] reduces scheduling events and context switches, but is still more complicated than EDF and provides no obvious advantage. Although neither LLF nor MLLF are optimal in a

multiprocessor setting [29], LLF can find a feasible schedule in some cases where EDF cannot. For the task set in figure 1.1, EDF will run two jobs to completion, leaving one job only one time unit to complete its two units of work. LLF, on the other hand, will interrupt one of the first jobs when the idle job reaches zero laxity at time $t = 1$, producing the schedule shown.

The LLF scheduler is based partially on the observation that a schedule has become infeasible if any job ever has negative laxity in its current period (a job has 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 leads us to our second consideration when designing a greedy algorithm. The "greedy" part tells us *which* task(s) to schedule, but we must also specify *when* we will do the scheduling. That is, at what times should we re-sort and apply our greedy preference? We have a list of three *standard scheduling events*:

Job Release:

A task has reached the beginning of its period, and a new job is available

Work Complete:

A task has finished the work of its current job, 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 will specify a greedy sort key and which scheduling events it will observe. For example, adding Zero Laxity events to EDF gives a hybrid scheduler known as EDZL [11]. While this provides an improvement over standard EDF for multiprocessors, EDZL is still not optimal.

2.1.2 Greedy Algorithms Are Shortsighted

To motivate our solutions to the optimal scheduling problem, we will examine why greedy scheduling algorithms fail. In general, when a greedy algorithm fails to provide an optimal solution, it's because the algorithm proceeds by making some locally optimal (greedy) decision without global knowledge, or concern for the "big picture." Consider the following:

Example 2.1. Let

$$\mathcal{T} \;=\; \{\, \tau_1 = (10, 9),\ \tau_2 = (10, 9),\ \tau_3 = (20, 4) \,\}$$

be the set of periodic tasks shown in Figure 2.1, where $\tau_i = (\textit{period, work})$, and $m = \rho(\mathcal{T}) = 9/10 + 9/10 + 4/20 = 2$ is the number of processors. It seems inevitable that any sensible greedy scheduling policy would choose to initially schedule tasks τ_1 and τ_2 (which have earlier deadlines, lower laxities, higher rates, etc.) over task τ_3 (which has huge laxity, low rate, and a long deadline). As we shall see in a moment, something critical then happens at time $t = 8$, but even then, any common greedy scheme would still prefer τ_1 and τ_2 over τ_3 (they still have earlier deadlines, lower laxities, and higher "remaining rates"- 1/2 vs. 4/12). However, if these tasks are allowed to run to completion at $t = 9$, then during the $[9, 10)$ time interval, only τ_3 has work remaining, and so only it is scheduled; the other processor sits idle. Let's suppose we are scheduling with LLF, as shown in Figure 2.1. When new jobs of τ_1 and τ_2 arrive at $t = 10$, they are switched on, and execute until τ_3 causes a zero-laxity event at $t = 17$. τ_1 is switched off, but is switched on at $t = 18$ when it reaches zero laxity. So τ_2 is switched off, but it also reaches zero laxity at $t = 19$. At this point, we have three tasks with zero laxity, and only two processors, so a deadline will be missed. The cause of this missed deadline is the idle time suffered during the $[9, 10)$ time interval. At time $t = 10$, $(9+9+3) = 21$ units of work remain until the shared deadline at $t = 20$, but only 20 units of processor

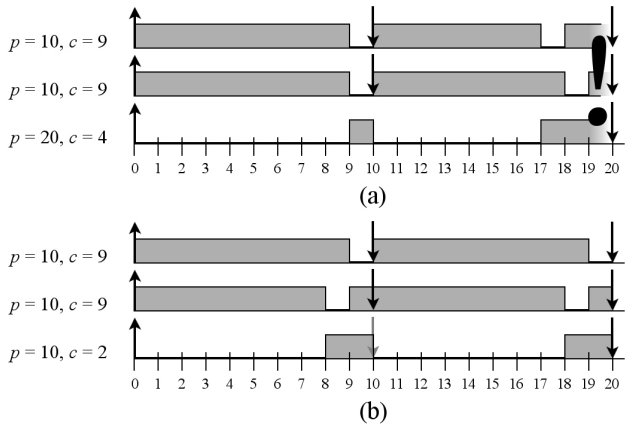


Figure 2.1: **Greedy Counter-example**
Task set which confounds known greedy schedulers using common events. (a) shows an incorrect greedy schedule, while (b) shows a feasible proportional schedule.

time are available. In a periodic system with 100% utilization, any idle time eventually forces a deadline miss, as the next theorem shows. □

Theorem 2.1. When the total rate of a periodic task set is equal to the number of processors, then no feasible schedule can contain any idle processor time.

Proof. Given tasks $\tau_1, \ldots, \tau_n$ on m processors where the rates sum to m. In a feasible schedule, task τ_i, at the end of h periods, must have done work equal to $hc_i = h(p_i\,\rho_i) = (hp_i)\,\rho_i = a_{h+1}\,\rho_i$, 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 $\sum_i t'\,\rho_i = t' \sum_i \rho_i = t'm$. 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 can be done, and some deadline will be missed. □

We can now see where LLF failed on our previous example. In their initial period, τ_1 and τ_2 each require 1 unit of idle time. Because there is only one other task in the system to fill it, these two blocks of idle time cannot be allowed to occur

simultaneously. τ_3 must be started no later than $t = 8$ in order to fill up the idle blocks of both τ_1 and τ_2. This can be seen by looking at the *collective* idle time remaining for τ_1 and τ_2: with 2 units of idle time needing to be filled and only one other task to fill them, clearly one of these two tasks must begin idling at $t = 8$. But the greedy models discussed so far are inadequate for detecting this: no practical greedy criteria would select τ_3 over τ_1 or τ_2 to run at $t = 8$, and even if it would, none of our three standard scheduling events occur at this time to force a rescheduling. In fact, *no* event set that considers only individual tasks would include $t = 8$; to see this event, we must consider both τ_1 and τ_2. But this example constitutes a very simple task set. In task sets with many tasks and highly varying periods, we might need to consider many or all possible subsets of tasks in order to reliably detect all such "collective idle time" events. This might well lead to an NP-Hard search space, and is clearly impractical. As is usually the case when a greedy approach fails, the failure is due to a lack "global knowledge": not looking further into the future, or not considering the combined behavior of multiple items. So how are we to overcome these shortcomings, and develop an efficient algorithm that is more "far-sighted"? The first solution to this problem is surprisingly simple, and is achieved by a specific overconstraining of our system.

2.2 Deadline Partitioning and DP-Fair

Prior to our work presented in Chapter 3, the only known solutions to the "global knowledge" problem were 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 Example 2.1 to

$$\{\ \tau_1 = (10, 9),\ \tau_2 = (10, 9),\ \tau_3 = (10, 2)\ \}\ .$$

All we have done is to impose the additional requirement that task τ_3 complete a proportional share of its work every time the other tasks reach their deadlines. Suddenly τ_3 has an observable ZERO LAXITY event at time 8: τ_3 cannot remain idle any longer if it is to complete its work on time. So naturally τ_3 is switched on; τ_1 and τ_2 will each run for one of the remaining two time units on the other processor, and a feasible schedule results (see Figure 2.1(b)). 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 [4]. 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.

2.2.1 DP-FAIR Conditions for Periodic Tasks

To motivate our next step, we recall the following result by Hong and Leung [25].

Theorem 2.2. No optimal on-line scheduler can exist for a set of (unrelated, aperiodic) jobs *with two or more distinct deadlines* on any m-processor system, where $m > 1$. □

Note that Theorem 2.2 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 jobs arrive within the scheduling

interval and their arrival time is unknown. Their RESCHEDULE algorithm, like our own DP-WRAP algorithm (Section 2.2.2) and several other schedulers, is based on McNaughton's wrap-around rule [38]. 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, 10, 47] have used deadline partitioning, there has not previously been a unifying theory for why this technique is so effective. With the DP-FAIR (**D**eadline **P**artitioning with **Fair**ness) 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 $\mathcal{T}$, where $t_j < t_{j+1}$ for all $j \geqslant 0$. Then the j^{th} time slice, denoted $\mathbb{S}_j$, 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 $\mathbb{S}_j$ 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 τ_i, and j is for time slice $\mathbb{S}_j$. Appendix A contains a summary of all notation used throughout this work.

We analyze schedules by considering execution during $\mathbb{S}_j$, drawing heavily on

notation from the LLREF scheduling algorithm [10]. The *local execution remaining* of a task τ_i at time t, denoted $e_{i,t}$, is the amount of time that τ_i must execute before the next time slice boundary, *i.e.,* between times t and t_j. A task's *local rate* $r_{i,t} = e_{i,t}/(t_j - t)$ is the proportion of time between t and t_j that τ_i must spend executing. We let E_t and R_t denote a task set's summed local remaining execution and rate, respectively, at time t.

The scheduling process is most easily understood when the task set has full utilization, *i.e.,* $\rho(\mathcal{T}) = m$. 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 $S(\mathcal{T}) = m - \rho(\mathcal{T})$. Consider a time slice of length 10 on 2 processors, and a task set with $\rho(\mathcal{T}) = 1.5$. 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 one or more dummy jobs 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 [7], improving performance [3, 31], creating a work-conserving scheduler [20, 21], or running non-real-time tasks in a hybrid system [32].

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 rate; that is, task τ_i is assigned

workload $e_{i,t_{j-1}} = \rho_i \times L_j$ for time slice $\mathbb{S}_j$. 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 2.3.1, but for now is just zero.

Definition 2.1 (DP-FAIR Scheduling for Time Slices)**.** A slice-scheduling algorithm is DP-FAIR if it schedules jobs within a time slice $\mathbb{S}_j$ 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 $(S(\mathcal{T}) \times L_j) + F_j(t)$ units of idle time to occur in $\mathbb{S}_j$ before time t. □

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

Lemma 2.3. If tasks $\mathcal{T}$ are scheduled within $\mathbb{S}_j$ according to DP-FAIR, and $R_t \leqslant m$ at all times $t \in \mathbb{S}_j$, then all tasks in $\mathcal{T}$ will meet their local deadlines at the end of $\mathbb{S}_j$.

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 $R_t \geqslant m + 1$, contradicting our assumption that $R_t \leqslant m$. □

Lemma 2.4. If a feasible set $\mathcal{T}$ of periodic tasks with implicit deadlines is scheduled in $\mathbb{S}_j$ using any DP-FAIR algorithm, then $R_t \leqslant m$ will hold at all times $t \in \mathbb{S}_j$.

Proof. Let us introduce the dummy job τ_{n+1} representing idle time, and give it rate $S(\mathcal{T})$ (which can be larger than 1, since this one "job" is allowed to run on multiple processors at once [3]). We let $e_{n+1,t}$ be the portion of the total $S(\mathcal{T}) \times L_j$ idle time in

[3] Alternately, imagine $S(\mathcal{T})/\epsilon$ jobs, each with rate ϵ, as $\epsilon \to 0$. These jobs would fill up the required $S(\mathcal{T})$ idle time on any processors, could be run in parallel, and would not, individually, need to run on more than one processor at a time.

$\mathbb{S}_j$ not yet used up by time t, and let

$$E_t \;=\; \sum_{i=1}^{n} e_{i,t} \quad \text{and} \quad E'_t \;=\; \sum_{i=1}^{n+1} e_{i,t}$$

be the work remaining at time t in our original and extended task sets, respectively. Now, the m processors are consuming the workload from $\tau_1, \ldots, \tau_{n+1}$ 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 $\mathbb{S}_j$, $m(t_j - t_{j-1}) - m(t - t_{j-1}) = m(t_j - t)$ remain at time t, *i.e.*, $E'_t = m(t_j - t)$. Then

$$R_t \;=\; \sum_{i=1}^{n} \frac{e_{i,t}}{t_j - t} \;\leqslant\; \frac{1}{t_j - t} \sum_{i=1}^{n+1} e_{i,t} \;=\; \frac{E'_t}{t_j - t} \;=\; m \;\;,$$

as desired. □

Theorem 2.5. Any DP-Fair scheduling algorithm for periodic task sets with implicit deadlines is optimal.

Proof. For any feasible task set $\mathcal{T}$, Lemmas 2.3 and 2.4 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 τ_i's fluid rate curve is zero at its own deadlines, it follows that τ_i will meet its deadlines. This holds for all jobs from all tasks in $\mathcal{T}$. Thus any DP-Fair algorithm will correctly schedule any feasible task set, and so is optimal. □

Rules 1 - 3 of Definition 2.1 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.

Requiring every task to hit its fluid rate curve at every system deadline (*i.e.,* DP-Fair Allocation) is clearly an overconstraint. However, given this objective, these three rules are obviously necessary. What is surprising, given their simplicity, is that 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.

2.2.2 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 $\mathbb{S}_j$, make a "block" of length ρ_i for each τ_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 [38]). See Figure 2.2 for an illustration with 7 tasks and 3 processors. To find the actual timing points of context switches within any $\mathbb{S}_j$, 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 rate 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 rates), 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

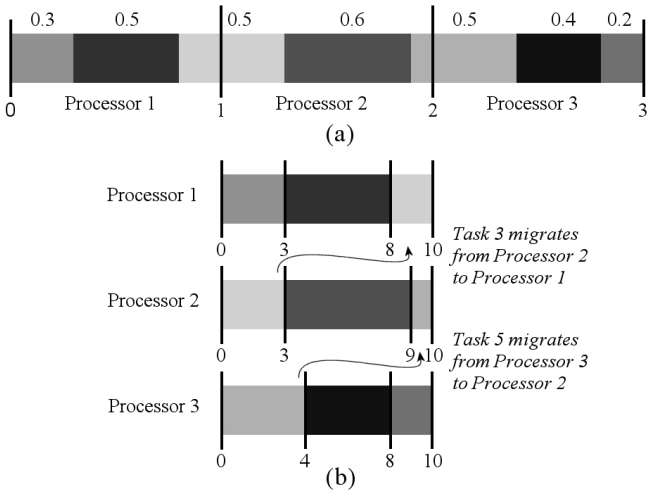


Figure 2.2: **The DP-WRAP Algorithm**
(a) Seven tasks with rates shown above. These are lined up in arbitrary order, then split at length 1 intervals.
(b) Each processor runs its task set over a length 10 time slice. Jobs sliced in (a) are seen migrating in (b).

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 at run time.

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 slice: 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 2.2, task 3 runs for the first 0.3 of the slice 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 2.6. The DP-Wrap scheduling algorithm with mirroring in odd slices will produce at most $n-1$ context switches, $n-1$ preemptions, 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. At least one of these jobs will have a deadline at the end of the slice, and so the context switch which turns this job off would not be counted as a preemption. However, if (i) this job is migrating, and its first segment is turned off before migrating to complete on another processor, or (ii) this job is the last job on the last processor, and so is not context switched off, then this soonest-deadline job will not provide a non-preemptive context switch. So while, on average, the number of preemptions will be slightly less than the number of context switches, in any given slice there may still be $n-1$ preemptions. Finally, there are (at most) $m-1$ tasks which migrate once each per slice (or fewer, if some task fully fills a processor and does not need to wrap onto the next processor). □

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.

2.3 Extended Problem Domains and Algorithmic Modifications

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 and deadline lengths that differ from periods. We will even show how DP-Fair algorithms can schedule a randomly arriving sequence of unrelated jobs. We also discuss various modifications to the DP-Wrap algorithm.

2.3.1 DP-Fair for Sporadic Tasks and Constrained Deadlines

We begin by expanding DP-Fair to handle tasks with sporadic job arrivals. We will also consider *constrained* deadlines, where $\delta_i \leqslant p_i$ and, consequently, rate $\rho_i = c_i/\delta_i$; the case of *arbitrary* deadlines (additionally allowing $\delta_i > p_i$) is addressed in Section 2.3.4. Note that constrained deadlines may be viewed as a special case of sporadic arrivals, so we needn't treat them differently. That is, if a task with a constrained deadline has an *expected* arrival at some release time after its deadline, this can be accommodated by any mechanism which handles the *unexpected* arrival of a late sporadic task at some point after its deadline. To deal with this new gap between deadline and arrival, we will give more detailed rules for how to allocate workloads within a time slice in these cases. We will also need to be more careful about how and when we delineate our time slices. The DP-Fair rules of Definition 2.1 for scheduling within 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 $\rho(\mathcal{T}) \leqslant m$ and $\rho_i \leqslant 1\ \forall i$. These conditions are necessary for sporadic task sets with implicit deadlines in the sense that we must be able to accommodate all possible arrival patterns that do not violate the minimum inter-arrival time – including the worst case scenario when all arrivals occur as early as allowable (*i.e.,* all tasks behave like implicit deadline periodic tasks). However, it is possible to have a sporadic system with $\rho(\mathcal{T}) > m$ that is still schedulable if enough jobs are sufficiently late, so the constraint is not necessary for all specific instances of sporadic arrivals. Similarly, it is not difficult to construct task sets with constrained deadlines that violate the condition that $\rho(\mathcal{T}) \leqslant m$, and

yet may be feasibly scheduled. For example, if $\tau_1 = (2, 1, 1)$ and $\tau_2 = (2, 1, 2)$ then $\rho_1 = 1$ and $\rho_2 = 0.5$, giving $\rho(\mathcal{T}) = 1.5$. Even so, this task set is feasible on $m = 1$ processor because τ_2 can wait for 1 time unit whenever both tasks have a job arrive simultaneously. The $\rho_i \leqslant 1$ constraint, on the other hand, must always hold if task parallelism is to be avoided. Since our extended DP-Fair rules do not handle these cases, DP-Fair algorithms are no longer optimal for constrained deadline systems (that is, they cannot schedule all feasible task sets). In fact, it has recently been shown that there *can be* no optimal algorithm for sporadic task sets when deadlines can be smaller than periods [19]. Thus we will limit ourselves to showing that DP-Fair algorithms are optimal on task sets with $\rho(\mathcal{T}) \leqslant m$ and $\rho_i \leqslant 1 \ \forall i$.

With periodic arrivals and implicit deadlines, we know at the outset exactly when all deadlines/arrivals will occur, and so have available a complete map of all future time slices. With the sporadic arrival of a task with a short deadline in the middle of a long time slice, we might have an unexpected deadline appear in the middle of the current slice. When this happens, we will need to subdivide our time slice into secondary *subslices*. Suppose τ_i has no available job at the beginning of slice $\mathbb{S}_j = [t_{j-1}, t_j)$, and that it's next job arrives at time t with a deadline of $t+\delta_i$, so that $t_{j-1} < t < t+\delta_i < t_j$. In order to ensure that τ_i meets its deadline at $t + \delta_i$, we will split the remainder of $\mathbb{S}_j$ into subslices $\mathbb{S}_j^1 = [t, t + \delta_i)$ and $\mathbb{S}_j^2 = [t + \delta_i, t_j)$. We refer to this as a *splitting arrival*. It is possible that another job with a short deadline might arrive and further subdivide a subslice. A top-level time slice is a *primary slice*; no other time slice was active when it began, and so it is not contained within any other slice. For the moment, we will only prove results about primary slices; we will still allow sporadic arrivals in the middle of time slices, but we will assume that an arrived job's deadline does not fall before the end of the current slice. We will deal with the issue of subslices in Section 2.3.2.

Another complication created by sporadic tasks and constrained deadlines is failure of tasks to use all the execution capacity reserved for them. For example, since

$\rho_i = c_i/\delta_i$ when $\delta_i < p_i$, the time between deadline and next period represents unused capacity. The same is true for the delay between earliest possible and actual arrivals for a sporadic task. During this time (*i.e.,* between $a_{i,h-1} + \delta_i$ and $a_{i,h}$), the processor capacity reserved for the task is going unused, and so is going into the system's pool of available idle time, or "slack." In the time between a task's deadline and its next job arrival, we say that the task is *freeing slack* (or *inactive*); a task is *active* between times $a_{i,h}$ and $a_{i,h} + \delta_i$ (even if it has no work remaining). Thus, for each task, time is partitioned into slack-freeing and active periods. Because $\rho(\mathcal{T}) \leqslant m$, τ_i "owns" a portion ρ_i of the system's total capacity m, even during times when the task is inactive. For this reason, we sometimes 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.

Let us be more precise about our accounting of how a task's work and freed slack consume its allocated capacity. Similarly to how $e_{i,t}$ represents local execution time remaining, we will let $\kappa_{i,t}$ represent *local capacity* remaining for task τ_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 ρ_i). We define $\alpha_{i,j}(t)$ and $f_{i,j}(t)$ to be the amounts of time that τ_i has been active or freeing slack, respectively, during slice $\mathbb{S}_j$ as of time t. We use $\alpha_{i,j}$ and $f_{i,j}$ as shorthands for $\alpha_{i,j}(t_j)$ and $f_{i,j}(t_j)$, the total active and inactive times of τ_i in $\mathbb{S}_j$, respectively.

In time slice $\mathbb{S}_j$, τ_i will ultimately be allotted a total of $\rho_i \times \alpha_{i,j}$ local execution time (although some of this may be allocated dynamically mid-slice if a new job arrives). τ_i will also free $\rho_i \times f_{i,j}$ slack during this time slice. Thus, in $\mathbb{S}_j$ of length L_j, τ_i is allotted total capacity

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

as expected. Since $\rho_i \times \alpha_{i,j}$ is added to $e_{i,t}$ over the course of $\mathbb{S}_j$, and $\kappa_{i,t}$ and $e_{i,t}$ both

decrease at a rate of 1 while τ_i is executing, we always have $e_{i,t} \leqslant \kappa_{i,t}$. Example 2.2 illustrates the progression of $e_{i,t}$ and $\kappa_{i,t}$, as well as the subdividing of time slices. Finally, we define the *total freed slack in* $\mathbb{S}_j$ *as of time* t to be

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

This is the same freed slack term seen in RULE 3.

We now formalize the rules for time slice boundaries and work allocation in our new problem domain.

Definition 2.2 (DP-FAIR Allocation for Extended Task Model)**.** An algorithm has *DP-FAIR Allocation* if, for every time slice $\mathbb{S} = [t_1, t_2)$, end time t_2 and local execution times are determined according to the following rules:

RULE 4: At the beginning of any time slice, we compute its ending to be the minimum of all future deadlines among all active tasks. That is, when $\mathbb{S}$ begins at time t_1, we set $\mathbb{S} = [t_1, t_2)$, where

$$t_2 = \min_{\tau_i \in A(t)} \{d_i \mid d_i \text{ is the next deadline of } \tau_i \text{ after } t\},$$

and $A(t)$ is the set of active tasks (with or without work remaining) at time t. If no tasks are active at t_1, we begin the next slice as soon as one arrives.

RULE 5: For each task τ_i, if it is active at t_1, assign it a local execution requirement of $e_{i,t_1} = \rho_i(t_2 - t_1)$. If τ_i is inactive at t_1, $e_{i,t_1} = 0$; if it arrives (becomes active) at time $a_{i,h}$ in the middle of $\mathbb{S}$ (with a deadline $a_{i,h} + \delta_i \geqslant t_2$), assign it $e_{i,a_{i,h}} = \rho_i(t_2 - a_{i,h})$. □

Because of RULE 4, no deadline can occur within a time slice. If a job arrives and also has a deadline within the current slice, a new subslice is created which ends at this job's deadline. If a job arrives during the current slice with a deadline after the

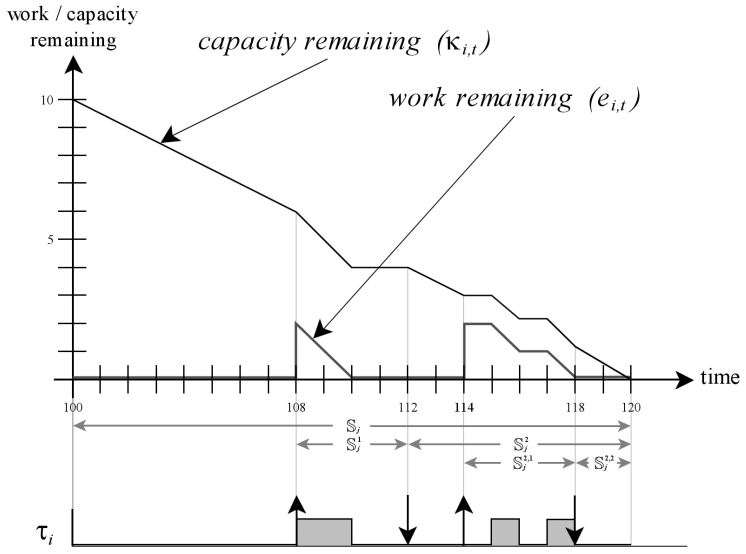


Figure 2.3: **Multiple Nested Subslices**
τ_i has two arrivals within the primary time slice $\mathbb{S}_j$. The first divides the remainder of $\mathbb{S}_j$ into $\mathbb{S}_j^1$ and $\mathbb{S}_j^2$, and then the second divides the remainder of $\mathbb{S}_j^2$ into $\mathbb{S}_j^{2,1}$ and $\mathbb{S}_j^{2,2}$.

end of that slice, no subslices are created; RULE 4 will account for that deadline at the appropriate future time. Hence, every deadline ends some time slice, and consequently, a task can never become inactive *during* a time slice. That is, any task will (i) remain active for an entire slice, (ii) become active during the slice, and remain active until the end, or (iii) remain inactive for the entire slice.

The following example traces local work and capacity over a primary time slice, and also illustrates how multiple arrivals from a single task within a time slice can create nested subslices.

Example 2.2. To illustrate the above ideas, suppose that the sporadic task $\tau_i = (4, 2)$ is inactive, and late in its next arrival at the beginning of time slice $\mathbb{S}_j = [100, 120)$. Since $\rho_i = 0.5$, its initial capacity is $\kappa_{i,100} = \rho_i \times L_j = 10$. Since it is inactive, $e_{i,100} = 0$. See Figure 2.3 for reference.

Suppose it has a job arrive at time $t = 108$, with a deadline at $t = 112$. As it is freeing slack in the interval $[100, 108)$, $\kappa_{i,108} = 10 - 0.5(8) = 6$. Since the job has

workload 2, $e_{i,108} = 2$. The remainder of $\mathbb{S}_j$ is partitioned into subslices $\mathbb{S}_j^1 = [108, 112)$ and $\mathbb{S}_j^2 = [112, 120)$. Suppose τ_i immediately executes this workload to completion by time $t = 110$. Then $\kappa_{i,110} = 6 - 2 = 4$, and $e_{i,110} = 0$. These are unchanged when $\mathbb{S}_j^1$ ends at $t = 112$, and τ_i resumes freeing slack.

Suppose the next job of τ_i arrives at $t_2 = 114$. Now $\kappa_{i,114} = 4 - 0.5(114 - 112) = 3$, and $e_{i,114}$ is set back up to 2. Further, $\mathbb{S}_j^2$ is recursively subdivided into $\mathbb{S}_j^{2,1} = [114, 118)$ and $\mathbb{S}_j^{2,2} = [118, 120)$. τ_i will do 2 units of work during $\mathbb{S}_j^{2,1}$, so we'll have $\kappa_{i,118} = 3 - 2 = 1$, and $e_{i,118} = 0$. If τ_i remains inactive during $\mathbb{S}_j^{2,2}$, then $\kappa_{i,t}$ will drop at a rate of 0.5 from 1 to 0 by time 120.

Now suppose that another job of τ_i were to arrive at $t = 119$. No further subslices would be created, because it's deadline would be at time 123, outside of the current slice. Instead, when slice $\mathbb{S}_{j+1}$ starts at time 120, the end of this next slice can be no later than 123 due to this deadline. At this new arrival at $t = 119$, τ_i would be assigned a local workload of $e_{i,119} = \rho_i(120 - 119) = 0.5$ by RULE 5. We will subsequently prove that the idle time reserved by RULE 3 will be sufficient to handle this new additional workload. □

Lemmas 2.7 and 2.8 below establish some properties of primary time slice $\mathbb{S}_j$. In their proofs, we will assume we have no splitting arrivals during $\mathbb{S}_j$. In Section 2.3.2, we will show that these results also apply to subslices.

Lemma 2.7. A DP-FAIR algorithm cannot cause more than $(S(\mathcal{T}) \times L_j) + F_j(t)$ units of idle time in time slice $\mathbb{S}_j$ 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 $\mathbb{S}_j = [t_{j-1}, t_j)$. For the sake of contradiction, let t' be the first failure point in $\mathbb{S}_j$. Since I_j and F_j are

continuous functions of t, this means that $I_j(t') = (S(\mathcal{T}) \times L_j) + F_j(t')$ and $I_j(t' + \epsilon) > (S(\mathcal{T}) \times L_j) + F_j(t' + \epsilon)$ for all sufficiently small $\epsilon > 0$.

Since no task which is active at time t' can become inactive before the end of $\mathbb{S}_j$, 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 $\mathcal{T}$ into three sets based on task status 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 active tasks that have completed their allotted work for $\mathbb{S}_j$. More specifically,

- $A = \{\tau_i \mid e_{i,t'} > 0\}$
- $B = \{\tau_i \mid \tau_i \text{ is inactive}\}$, and
- $C = \{\tau_i \mid \tau_i \text{ is active and } e_{i,t'} = 0\}$.

For convenience, we will let $\rho_X = \rho(X) = \sum_{i \in X} \rho_i$ for $X \in \{A, B, C\}$.

Based on our definition of local capacity, any task τ_i should account for $\rho_i \, L_j$ processor time during $\mathbb{S}_j$ 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 B's allotment of processor time has been used up exactly. 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 $\rho_C \, L_j$ processor time, and are $\rho_C(t_j - t')$ *ahead* of their fair share at time t'. Similarly, the static slack pool $S(\mathcal{T})L_j$ is already consumed, and so is $S(\mathcal{T})(t_j - t')$ ahead of its proportional allotment at time t'. This means that tasks in A must be collectively $(\rho_C + S(\mathcal{T}))(t_j - t')$ units *behind* on their use of processor time. If they were keeping up with their fluid rate curves, they would have $\rho_A(t_j - t')$ work remaining, so as it is they must have exactly

$$\rho_A(t_j - t') + (\rho_C + S(\mathcal{T}))(t_j - t') \;=\; (m - \rho_B)(t_j - t')$$

work remaining, since $\rho_A + \rho_B + \rho_C + S(\mathcal{T}) = m$.

Given our definition of t', RULE 3 tells us that we cannot choose to idle processors at time t'. If $|A| \geqslant m$, 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,

$$(m - \rho_B)(t_j - t') \leqslant |A|(t_j - t') \quad \Rightarrow \quad \rho_B \geqslant m - |A| \ .$$

Tasks in B are freeing slack at a rate of ρ_B at time t'; the system is only adding idle time at a rate of $m - |A|$. Then $F_j(t)$ is growing at least as fast as $I_j(t)$ at time t', and $I_j(t)$ cannot immediately exceed $S(\mathcal{T})L_j + F_j(t)$, again contradicting our choice of t'. There can be no first failure time t', so our inequality holds for all $t \in \mathbb{S}_j$ by contradiction. □

Lemma 2.3 from the previous section makes no assumption about implicit deadlines or periodicity, and so is still valid in this extended problem domain. Once we have shown that $R_t \leqslant m$ for all t in a time slice, and extended this result to subslices, our desired conclusion will soon follow.

Lemma 2.8. If a set $\mathcal{T}$ of sporadic tasks with constrained deadlines is scheduled using any DP-FAIR algorithm, then at any time t, $R_t \leqslant m$ will hold for local remaining rate R_t in time slice $\mathbb{S}_j$.

Proof. As of time $t \in \mathbb{S}_j = [t_{j-1}, t_j)$, 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 2.7 gives $I_j(t) \leqslant S(\mathcal{T})L_j + F_j(t)$. If we let $w_{i,j}(t)$ be the work executed on task τ_i during $\mathbb{S}_j$ as of time t, then we have

$$m(t - t_{j-1}) \quad = \quad I_j(t) + \sum_{i=1}^{n} w_{i,j}(t) \ , \tag{2.1}$$

and

$$\begin{aligned} \kappa_{i,t} &= \kappa_{i,t_{j-1}} - w_{i,j}(t) - \rho_i \, f_{i,j}(t) \\ &= \rho_i \, L_j - w_{i,j}(t) - \rho_i \, f_{i,j}(t) \ . \end{aligned} \tag{2.2}$$

Recalling that $r_{i,t}(t_j - t) = e_{i,t} \leqslant \kappa_{i,t}$,

$$\begin{aligned} R_t(t_j - t) &= \sum_{i=1}^{n} e_{i,t} \\ &\leqslant \sum_{i=1}^{n} \kappa_{i,t} \\ &= \sum_{i=1}^{n} (\rho_i \, L_j - w_{i,j}(t) - \rho_i \, f_{i,j}(t)) \qquad \text{by (2.2)} \\ &= \rho(\mathcal{T}) L_j - \sum_{i=1}^{n} w_{i,j}(t) - F_j(t) \\ &\leqslant (m - S(\mathcal{T}))L_j - \sum_{i=1}^{n} w_{i,j}(t) + (S(\mathcal{T})L_j - I_j(t)) \\ &= mL_j - (m(t - t_{j-1})) \qquad \text{by (2.1)} \\ &= m(t_j - t) \end{aligned} \tag{2.3}$$

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

Note that Lemma 2.7 and Inequality 2.3 hold in the current slice even at the moment of a new job arrival. That is, $R_t \leqslant m$ will hold even if all inactive jobs were to suddenly arrive with active periods lasting the remainder of the slice. The portion of Inequality 2.3 which says "$\sum_i \kappa_{i,t} \leqslant m(t_j - t)$" can be interpreted as "Even in the event of the simultaneous worst case releases of all inactive tasks, the system has still reserved enough capacity to schedule them all."

2.3.2 Splitting Arrivals and Subslices

Lemmas 2.7 and 2.8 were derived only for primary (top level) time slices, and under the assumption of no splitting arrivals. We will now show how to distribute workloads in a subslice so that these results apply there as well. The basic idea is simple: we distribute remaining workloads proportionately between the two new subslices. However, in order to establish the correctness of this, we must proceed cautiously. In brief, our prior results show that the system is feasible at the moment of a splitting arrival; if we take the remaining workloads for the slice, and then pretend that we have a new system of n tasks with these workloads and a common deadline at the end of the current slice, then this virtual system may be treated as a top level slice, and the above results will still apply. Throughout, we will assume that for primary slice $\mathbb{S}_j = [t_{j-1}, t_j)$, task τ_i is inactive at t_{j-1}, and has an arrival at time $t \in \mathbb{S}_j$ which splits the remainder of $\mathbb{S}_j$ into subslices $\mathbb{S}_j^1 = [t, t + \delta_i)$ and $\mathbb{S}_j^2 = [t + \delta_i, t_j)$. In order to establish the validity of this temporary virtual system, we need the following Lemma.

Lemma 2.9. When a splitting arrival occurs in a primary slice $\mathbb{S}_j$, the system remains feasible; that is, a valid schedule still exists for the system at that moment.

Proof. Suppose the arrival of τ_i splits $\mathbb{S}_j$ as described above. Now imagine that instead of τ_i, we see the arrival of τ_i', with the same rate ρ_i but a deadline at t_j instead of $t + \delta_i$. This job would have work equal to $\rho_i(t_j - t)$, and as described in the proof of Lemma 2.8, the system will have reserved this much capacity for τ_i''s execution as of time t. Lemma 2.8 would apply, and Lemma 2.3 would tell us that all local deadlines may be met, indicating a feasible system. For any feasible schedule executed during the unsplit interval $[t, t_j)$, we could instead run two proportional copies of that schedule during the intervals $[t, t + \delta_i)$ and $[t + \delta_i, t_j)$; the same amount of total work would be done by each task during $[t, t_j)$. In particular, τ_i' would do $\rho_i((t + \delta_i) - t)$ work during the first interval, and $\rho_i(t_j - (t + \delta_i))$ work during the second. This provides us with

a feasible schedule for our original system with τ_i and its splitting arrival: the new job of τ_i requires $\rho_i((t + \delta_i) - t)$ units of work during the first interval (the duration of its job, or $\mathbb{S}_j^1$), and during the second interval $\mathbb{S}_j^2$ we may idle a processor in place of the execution of τ_i'. In this way, all local deadlines may be met, and thus the splitting arrival does not cause the system to become infeasible. □

Since the system is feasible at the instant t of a splitting arrival, we may take a snapshot of the remaining local workloads at this moment, and proceed as if we have a fresh new system defined by these workloads, with one deadline at the splitting point $t + \delta_i$, and all other deadlines at t_j, the end of the primary slice. We collectively refer to these temporarily altered tasks and and this time interval $\mathbb{S}_j^D = \mathbb{S}_j^1 \cup \mathbb{S}_j^2$ as a *slice domain*. Note that within the perspective of this domain there are no splitting arrivals: τ_i is arrived at the domain's start time t, and its deadline is known. As per RULE 4, at time t we establish one slice $\mathbb{S}_j^1$ lasting until the first deadline at $t + \delta_i$, and at that time create a second slice $\mathbb{S}_j^2$ lasting until the next deadline at t_j. As Lemma 2.9 establishes that the virtual system of this domain is feasible, we may apply Lemmas 2.7 and 2.8 to deduce that our DP-FAIR scheduling policies are correct within this domain. We formalize this procedure below.

Definition 2.3 (DP-FAIR Time Slice Splitting)**.** An algorithm has *DP-FAIR Slice Splitting* if slice-splitting arrivals are handled according to the following rules:

RULE 6: Suppose a job of task τ_i arrives at time t and has a deadline at $t + \delta_i$, and the current time slice $\mathbb{S}_j = [t_{j-1}, t_j)$ at t is such that $t_{j-1} < t < t + \delta_i < t_j$; that is, τ_i arrives and has its next deadline all within $\mathbb{S}_j$. Then we split the remainder of $\mathbb{S}_j$ into two *subslices* $\mathbb{S}_j^1 = [t, t + \delta_i)$ and $\mathbb{S}_j^2 = [t + \delta_i, t_j)$. This rule may be invoked repeatedly / recursively by multiple jobs arriving within a primary slice.

RULE 7: At the moment of a splitting arrival, we establish a virtual system, or *slice domain*, which we use to schedule our actual system for the remainder of $\mathbb{S}_j$, *i.e.,*

during the interval $\mathbb{S}_j^D = [t, t_j)$. WLOG, suppose that τ_1 is the splitting task. Our virtual task set is $\mathcal{T}' = \{\tau'_1, \ldots, \tau'_n\}$, has sporadic arrivals and implicit deadlines, and releases a job for τ'_i at time t if τ_i is active at this time. The tasks are defined by $\tau'_1 = (\delta_1, \rho_1\, \delta_1)$ and $\tau'_i = (t_j - t, e_{i,t})$ for $i = 2, \ldots, n$. That is, τ'_1 has a deadline at $t + \delta_i$ and the usual workload, and each other τ'_i which was active at t has its next deadline at t_j and a workload equal to the local remaining execution of τ_i at time t. During $\mathbb{S}_j^D$, the virtual system is scheduled according to RULES 1 - 7, and that schedule is applied to the original corresponding tasks in $\mathcal{T}$. □

In short, when a splitting arrival occurs, we schedule the remainder of the current time slice as if it were a virgin system with no history, using remaining local workloads and deadlines. Note that RULE 5 is still used to assign local executions within the domain's subslices, but the "rate" ρ_i referred to within RULE 5 is actually the rate ρ'_i of the temporary domain task τ'_i, so that $\rho'_1 = \rho_1$ and $\rho'_i = e_{i,t}/(t_j - t)$ for $i = 2, \ldots, n$.

Lemma 2.10. If a set $\mathcal{T}$ of sporadic tasks with constrained deadlines is scheduled using any DP-FAIR algorithm, then local execution deadlines will be met for all tasks within all primary slices and subslices.

Proof. As described in RULE 7, the scheduling of a subslice is equivalent to scheduling a primary slice in a different, but still feasible, system with no splitting arrivals. Then our previous Lemmas 2.7 and 2.8 apply to this other virtual system. Should another splitting arrival occur within the slice domain, we will merely apply RULE 6 and RULE 7 recursively, and create another virtual system within the domain. Because the domain is mimicking a top level system, the creation of a subdomain is equally valid. Within any such domain, Lemma 2.8 tells us that $R_t \leqslant m$ at all times, and so Lemma 2.3 indicates that all local deadlines will be met. □

Since we have shown that Lemmas 2.7 and 2.8 hold even in the presence of

splitting arrivals and subslices, we may now prove our principal result. The correctness of this approach follows easily from our preceding discussion.

Theorem 2.11. Any DP-Fair scheduling algorithm is optimal for sporadic task sets with constrained deadlines where $\rho(\mathcal{T}) \leqslant m$ and $\rho_i \leqslant 1 \ \forall i$.

Proof. Unlike Theorem 2.5, a task finishing its local workload at the end of some time slice does *not* ensure that it hits its fluid rate curve at that point, if that time slice is a subslice. Let's suppose that the splitting arrival of τ_1 at time t subdivides $\mathbb{S}_j = [t_{j-1}, t_j)$ in the usual way. τ_2 may be below its fluid rate curve when τ_1 arrives (perhaps τ_2 has yet to execute during $\mathbb{S}_j$). Since it's remaining local execution $e_{2,t}$ is divided proportionally between $\mathbb{S}_j^1 = [t, t + \delta_1)$ and $\mathbb{S}_j^2 = [t + \delta_1, t_j)$, τ_2 will still be below its fluid rate curve at the end of the subslice $\mathbb{S}_j^1$. However, this is not a problem, because τ_2 cannot have a deadline at the end of $\mathbb{S}_j^1$. Because τ_2 was not a splitting arrival (at least at the level of the slice domain containing $\mathbb{S}_j$), it's earliest possible deadline is t_j. And because the work remaining for τ_2 (namely, $e_{2,t}$) is divided between $\mathbb{S}_j^1$ and $\mathbb{S}_j^2$, and Lemma 2.10 tells us that local workloads are completed by their deadlines in these subslices, we know that the total remaining workload $e_{2,t}$ of τ_2 will be completed by the end of $\mathbb{S}_j^2$, which is also the end of $\mathbb{S}_j$. As for τ_1, it's local workload within $\mathbb{S}_j^1$ is its entire workload (since it's deadline is at the end of $\mathbb{S}_j^1$), and so Lemma 2.10 tells us that it will complete its splitting job on time.

Thus we see that, while every task might not match its fluid rate curve at the end of every slice, every task *will* match its fluid rate curve at the end of any slice whose end might coincide with the task's deadline. That is to say, all tasks match their fluid rate curves at their own deadlines, and thus meet their deadlines. The system is therefore correctly scheduled by our DP-Fair rules. □

2.3.3 Modifying DP-Wrap

Modifying DP-Wrap to handle sporadic arrivals is fairly straightforward. The first difference is that slice schedules are no longer practically identical, differing only in length and mirroring. Because the set of active jobs will change from slice to slice, DP-Wrap will need to recompute the schedule of each slice on-line, rather than relying on a look-up table which was computed off-line. This is still a very simple process: local workloads for the new time slice are computed, lined up as blocks, and divided onto processors at intervals equal to the length of this slice, much as described in Section 2.2.2.

The other modification from the periodic case is the handling of jobs which arrive in the middle of the current slice. A non-splitting arrival is assigned a local workload as described in Rule 5, and is merely wrapped onto the end of the existing stack-of-blocks schedule. In the event of a splitting arrival, we subdivide the remainder of the current slice and assign it a slice domain, as described in Rule 6 and Rule 7. Once we have used remaining local executions to create workloads for the temporary tasks of the slice domain (Rule 7), scheduling the two subslices of the domain is done exactly as DP-Wrap would schedule any other time slice. Any subsequent splitting arrival is handled by creating additional subdomains. Mirroring is still used when the set of active tasks does not change between subsequent slices.

2.3.4 Arbitrary Deadlines

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

Example 2.3. Consider the periodic task set $\mathcal{T}$ on $m = 2$ processors where $\tau_1 = (6, 4)$ and $\tau_2 = \tau_3 = \tau_4 = \tau_5 = (3, 1, 6)$. Since $\rho(\mathcal{T}) = 4/6 + 4(1/3) = 2$, 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. The first deadline of this system is at time 6. Figure 2.4

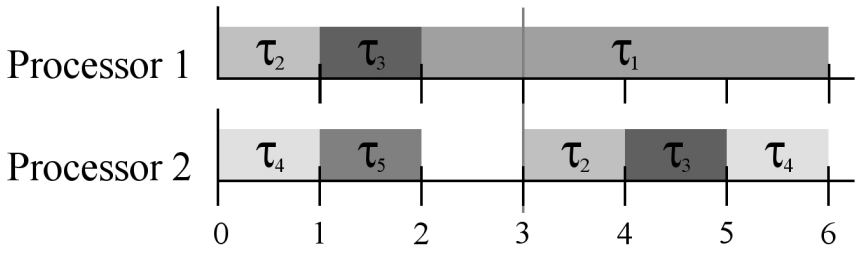


Figure 2.4: **Difficulties with $\delta > p$:**
With deadlines longer than periods, the DP-Fair rules can result in forced idle time.

illustrates the schedule of this task set during the interval [0,6) if we run τ_2 then τ_3 to completion on the first processor and τ_4 then τ_5 on the second. At time 2, tasks $\tau_2, \ldots, \tau_5$ are out of work. Only at this point is τ_1 forced to run by zero laxity. Until more work arrives for $\tau_2, \ldots, \tau_5$ at time 3, the other processor sits idle. Theorem 2.1 does not apply here because these tasks do not all have implicit deadlines, and in fact the task set is still feasible at time 6. Nonetheless, this unforced idle time shows how longer deadlines disrupt DP-Fair's ability to fully utilize all processors when $\rho(\mathcal{T}) = m$. □

Allowing deadlines longer than periods breaks the "global knowledge" granted by giving all tasks the same deadlines. There is a critical arrival event at $t = 3$, but because we haven't ended a time slice there, τ_1 doesn't "know" about it, and doesn't know to complete a proportional amount of its workload by that time. Fortunately, the problem is easily solved by adding a deadline at that arrival time. Specifically, if we are given a task where $\delta_i > p_i$, we simply impose an artificial deadline of $\delta'_i = p_i$. This doesn't increase the task's rate ρ_i, and if the artificial deadline is met, the the real one will certainly be also.

Unfortunately, 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 a slice boundary. Increasing the number of time slices, in turn, incurs additional overhead from the

added context switches and migrations. So while the artificial deadline solution is simple and effective, it may not be the best possible way to deal with arbitrary deadlines. Funk *et al.* [23] discuss some techniques for removing unnecessary time slice boundaries.

2.3.5 Admission Control of Aperiodic Jobs

Our approach to scheduling sporadic tasks with constrained deadlines can actually be extended to a much more general job model. To see how, consider the set B of unarrived jobs (as described in the proof of Lemma 2.7). Until one arrives, they may be treated as one undifferentiated mass, freeing slack at a rate of ρ_B, and holding enough capacity in reserve to finish all their remaining workloads should they all arrive at once. In this light, it does not actually matter what their periods or individual workloads are; the system will behave the same until some job in B arrives, regardless of what those jobs are. The only exception is when one of those unarrived jobs is forcing the very next slice boundary, which can happen when some task with $\delta_i < p_i$ has had its deadline in a previous slice, and its next arrival is at the end of the current slice. Even then, we could opt to ignore that next arrival (slice boundary) until the job actually arrives, treat it as a sporadic arrival instead of a constrained deadline, and then deal with it according to the usual rules. In other words, at any given time, we can construct our schedule based only on active jobs, so long as we're holding enough capacity in reserve to deal with the arrival of any set of jobs with total rate ρ_B.

With this in mind, we realize that the DP-Fair strategy for sporadic tasks with constrained deadlines will work just as well for independent, *aperiodic jobs* with or without deadlines. We first consider the jobs with deadlines. Consider a sequence of such jobs $J_1, J_2, \ldots$, where $J_i = (a_i, c_i, d_i)$ arrives at time a_i, and must complete a workload c_i before a deadline at time d_i. We will assign such a job a rate of $c_i/(d_i - a_i)$. These jobs are *not* recurring or in any way related to each other, and unlike the periodic and sporadic task model, we have *no* idea what jobs may arrive or when. So long as

we impose the restriction that at no time do we allow jobs with summed rate in excess of m into the system, we may schedule these jobs using the DP-Fair rules. Thus, we have a simple mechanism for admission control of aperiodic tasks with deadlines when using the DP-Fair strategy.

Now consider a job $J_i = (a_i, c_i)$ that has no deadline. If there is no urgency to it, we may simply execute it during otherwise idle processor time. If J_i is of higher priority, or if a number of these deadline-free jobs are backing up, we can assign it an artificial deadline d_i' so that it may reserve some amount of processor time. J_i's rate would be $c_i/(d_i' - a_i')$, where a_i' is the time at which the system assigns the artificial deadline, and d_i' is made sufficiently large so that J_i's new rate doesn't cause the active job set to exceed the total rate limit of m.

The allocation of time slices and the arrival of new jobs are handled according to Rules 4 - 7. At the end of any time slice, the next time slice is determined by Rule 4, and local executions are determined as in Rule 5. Should a new job arrive whose deadline falls within the current slice, subslices are formed as in Rule 6, and remaining local executions are subdivided according to Rule 7. Scheduling within a slice must simply obey Rules 1 - 3, as usual. Any scheduling algorithm for independent aperiodic jobs that follows Rules 1 - 7 is said to be DP-Fair. The DP-Wrap algorithm for aperiodic jobs simply uses the usual stack-and-slice scheduling for each time slice, and uses Rules 4 - 7 to determine slice lengths and workloads, and to deal with splitting arrivals.

Theorem 2.12. Given a set of independent, randomly arriving jobs $\{J_i\}_{i \geqslant 1}$, any DP-Fair algorithm will schedule it successfully so long as, at any time t, the set of active jobs A_t at time t satisfies (i) $\rho_i \leqslant 1$ for all $J_i \in A_t$, and (ii) $\sum_{J_i \in A_t} \rho_i \leqslant m$. Further, the DP-Wrap algorithm is DP-Fair.

Proof. As previously discussed, this more generalized job model does not change how

jobs are scheduled within a time slice. Consequently, it is still true that an algorithm following the DP-Fair rules will result in any job J_i's actual work curve matching its fluid rate curve at the end of (i) any primary (top level) time slice that started with J_i active, and (ii) any subslice that was created by the arrival of J_i. These necessarily include the end of the time slice which corresponds to J_i's deadline, guaranteeing that its workload will be completed on time. Lemma 2.10 still guarantees the successful completion of jobs who's entire span is within a single slice. The DP-Wrap algorithm does not behave any differently within slices than it did in previous cases, and so still follows the DP-Fair rules. □

Note that our condition that $\sum_{J_i \in A_t} \rho_i \leqslant m$ at *all* times is overly restrictive. By shifting workloads between slices, it would be fairly easy to deal with some situations where this condition is temporarily violated. For example, jobs could get ahead of their fluid rate curves while $\sum_{J_i \in A_t} \rho_i < m$, freeing additional capacity to deal with an added task that temporarily brought $\sum_{J_i \in A_t} \rho_i > m$. Also, there is no reason that we cannot schedule systems comprised of periodic, sporadic and aperiodic task sets using this strategy. This further illustrates the inherent flexibility of the DP-Fair approach.

2.3.6 Some Simplifications

Rule 6'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 earliest possible next deadline. That is, at the beginning of each time slice, set the end of the slice to be the soonest of all deadlines of all arrived jobs and also of the unarrived jobs, were they to arrive at that moment. Like our solution for arbitrary deadlines, this simplifies scheduling but increases context switches and migrations by creating slice boundaries that aren't always necessary.

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 yet are easily verified to satisfy RULE 3.

2.4 Survey of Deadline Partitioning Algorithms

We now explain and analyze a number of recent papers in the context of DP-FAIR. Unless otherwise noted, the following algorithms only address periodic task sets with implicit deadlines. The 1996 PFAIR algorithm [4] was the first optimal multiprocessor scheduler. It enforces a very strict notion of *proportional fairness* by ensuring that each task's work completed is within 1 of its fluid rate curve at *every* multiple of some discrete time quantum. It is computationally complex, and the frequent context switching and migrating would cause a high overhead in practice.

The first algorithm to use the technique of deadline partitioning seems to be the 2003 Boundary Fair (BF) algorithm [47]. It is a variation on PFAIR, with the same quantum-based timing, but with one key improvement. It makes the observation that it is only necessary for tasks to agree with their fluid rate curves at their deadlines. Consequently, it only enforces proportional fairness at the deadlines of tasks. Because of the resultant integer rounding, workload assignments aren't quite DP-FAIR, but the scheme is DP-CORRECT, and closely resembles DP-WRAP. However, because BF is also based on discrete units of work and time, there is still substantial complexity and overhead in dealing with round-off issues in amounts of allocated work.

Subsequent algorithms have all been based on continuous time, and benefited from the resulting simplicity. The discrete time model is somewhat more realistic, as processors operate on discrete clock cycles. However, as job workloads will generally be

much larger than this minimal time quantum, continuous time should generally be a good approximation, and small roundoff errors can be dealt with in any number of ways. Most recent papers in the field seem to be derived from one of two optimal algorithms, EKG [3] and LLREF [10], that appeared in the latter half of 2006. While later papers tackle related problem domains or present sub-optimal variants, none make substantial theoretical progress on the basic problem of optimal scheduling. We will examine these first two papers in some detail, and then survey others that have followed.

2.4.1 Improvements with EKG

Like DP-Wrap, the EKG ("**E**DF with task splitting and **k** processors in a **g**roup") algorithm is a variation of McNaughton's wrap around algorithm [38], but with two significant improvements. First, we observe that the $n - m + 1$ non-migrating tasks actually form a partitioned task set; that is, each one is permanently assigned to a single processor. On any processor, the interval between the execution of the migrating tasks at the beginning and end of a time slice consists of that processor running jobs assigned exclusively to it. Instead of assigning proportioned workloads to these tasks, EKG schedules them with EDF. In Figure 2.2, for example, we might view Processor 1 as a uniprocessor system with 80% capacity, and two tasks with rates of 0.3 and 0.5. In scheduling these two tasks, we essentially ignore all other tasks in the system, and run the simple, uniprocessor-optimal EDF algorithm. In this way, all tasks except the migrators may be "decoupled" from the proportional deadline scheme, allowing them to complete with fewer context switches. This method is known as *task splitting* or *semi-partitioning*. The $m - 1$ migrating tasks are still tightly coupled. The task split between Processors 1 and 2 must be scheduled in sync with the one on 2 and 3, which must be synced with the task split on 3 and 4, etc. The deadlines of all tasks, migrating and partitioned, impose slice boundaries.

EKG's second advantage over the basic DP-Wrap is its clever use of slack in the task set when $\rho(\mathcal{T}) < m$. The algorithm divides and decouples processors into groups of k, each of which operates independently from other groups. The extra space at the end of a group is not filled in with a partial task, but instead is left idle to consume the slack in $S(\mathcal{T})$. Because no task wraps onto the next processor, tasks before and after this gap needn't be synchronized. Each processor group may then be scheduled independently. Since a processor group only needs to observe the deadlines of its own tasks, these tasks are subjected to fewer time slices, and consequently, fewer context switches and migrations. When $k = m$, we have optimality; when $k = 2$, we can guarantee only 66% utilization, but with significantly fewer context switches and migrations.

2.4.2 The T-L Plane Visualization

The *Time and Local Execution Time Plane* ("T-L Plane") model of Cho *et al.* [10] provides a convenient means of visualizing a time slice $\mathbb{S} = [t_0, t_f)$. The T-L Plane is represented as an isosceles right triangle, with time on the horizontal axis, and work remaining on the vertical axis. As time moves forward, a task's work remaining curve will not follow its fluid rate curve, but instead will be in one of two modes:

- If it is running, it will have a slope of -1, since both axes are on the same scale
- If it is idle, it will have a slope of 0

A task's work remaining curve will alternate between these two modes as it is turned on and off. (Figure 1.2 gives a simple picture of the progression of one job through this plane, drawn next to its fluid rate curve.) By time t_f, all curves must have height 0 (*i.e.,* tasks have completed their work for this slice.) These are reset to heights proportional to their rates for the beginning of the next slice. A Zero Laxity event is an inactive job (horizontal curve) hitting the hypotenuse; a Work Complete event

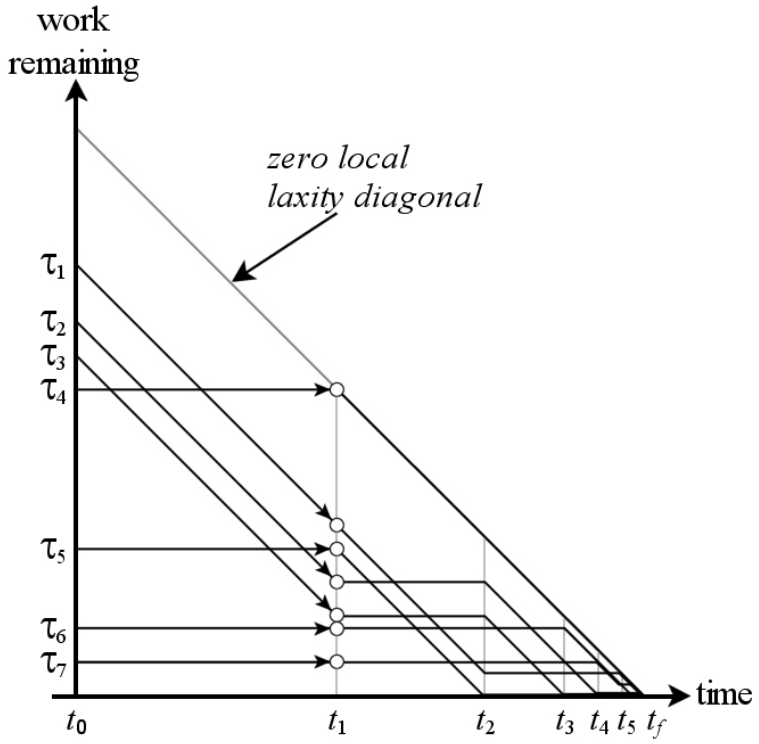


Figure 2.5: **Work Remaining Curves in a T-L Plane**
Seven tasks on three processors. At each secondary Zero Laxity or Work Complete event, the three jobs with most work remaining are executed.

is an active job (slope $= -1$) hitting the horizontal axis. A sample T-L Plane with executing jobs can be seen in Figure 2.5, with events indicated by time t_i and vertical lines.

The LLREF ("**L**argest **L**ocal **R**emaining **E**xecution **F**irst") scheduler is based on the T-L Plane visualization. At t_0 and at each secondary event, LLREF sorts the jobs by work remaining, and activates the m highest jobs. LLREF is a greedy scheduler, but it is also optimal because it adheres to the DP-Fair rules. Its greedy behavior is actually unnecessary, creates a higher computational overhead than EKG, and causes more context switches than are required to be DP-Fair. Also, this policy gives no prescription for the assignment of jobs to processors; it does not address the issue of task migration at all.

2.4.3 Subsequent Work

A number of subsequent algorithms have expanded on ideas introduced by EKG and LLREF. They do not represent improvements in optimal scheduling, per se. Instead, they reduce context switches and migrations for sub-100% utilization task sets, and address variants of our basic scheduling problem. There are two main classifications: those that follow EKG and use task-splitting, and those that use variants of LLREF's T-L Plane model.

Andersson and Bletsas [1] propose an EKG variant for dealing with sporadic task sets. Time slices are bounded by small, fixed width intervals rather than task deadlines. Like EKG, a tunable parameter (corresponding to time slice widths) is available, and can bring schedulable utilization arbitrarily close to 100% at the cost of more context switches and migrations. A sequel to this paper introduces the EDF-SS(DTMIN/ρ) algorithm [2], which extends the problem to tasks with arbitrary deadlines. It also uses fixed-width time slices, but attempts to split tasks with the smallest minimum deadlines.

The Ehd2-SIP algorithm [26] is also similar to EKG, but sacrifices optimality for improved general performance. It starts with a general wrap around task-to-processor assignment, but with tasks stacked in increasing period order. Then, rather than utilizing any notion of fluid scheduling, it uses EDF to schedule tasks on each processor, subject to the following exception: the "right-hand" half of a split task always has highest priority on its processor, unless it's "left-hand" other half is already running on the adjacent processor. This scheme only allows for 50% processor utilization in the worst case, but generally has high scheduling success until total utilization reaches the 80-90% range. These success rates are higher than strict partitioning EDF algorithms, although Ehd2-SIP suffers from more preemptions; conversely, EKG tends to have a higher success rate at the cost of more preemptions. The sequel to Ehd2-SIP is EDDP [27], which uses EKG's scheme of only partially filling processor capacity. It

also schedules each processor with EDF, but removes priority for the right half of a split task, and artificially adjusts deadlines to improve schedulability. Its worst case utilization improves to 65%, but performance is otherwise similar.

The E-TNPA algorithm [21] extends the T-L Plane/LLREF algorithm with two major improvements. First, when the total utilization is under 100%, it runs an excess time apportionment algorithm at the beginning of each slice, and distributes unused CPU time among tasks in the form of increased workloads within the slice. Once this is done for a new slice, it runs as if it were a normal T-L Plane, but with different rates. In this way, E-TNPA is *work-conserving*, that is, it never idles a processor when any idle task has unfinished work in its current period. The second improvement is the realization that the sorting of tasks by laxity at each scheduling event within the slice is unnecessary. Instead, the paper merely claims that tasks can instead be ordered based on the needs of the application/environment. This still requires scheduling invocations at every Zero Laxity and Work Complete secondary event. The authors provided a modified approach to their work-conserving scheduler with TRPA [20]. Instead of apportioning free time at the beginning of a slice, they now allow tasks to run arbitrarily (subject to the zero laxity rule) until such time as remaining required work in the slice to meet all fluid schedule goals equals the remaining processor capacity in the slice. Both E-TPNA and TRPA mimic the T-L Plane/LLREF system when given 100% utilization task sets, and like LLREF, they only prescribe *which* tasks should be running at a given time. With no scheme for assigning tasks to processors, it is difficult to gage the potential overhead of migrations.

Two other papers extend LLREF into different problem variants. Chen *et al.* [9] extend the T-L Plane model to uniform multiprocessors (processors have different speeds, but treat all tasks equivalently) with their T-L$_{er}$ Plane model. Added to the T-L Plane are lines representing remaining capacity in the plane for each processor. A

new "line hitting" event is added, where a task's token intersects a processor's execution capacity line. At this point, the task is assigned to the processor for the rest of the time slice, and henceforth ignored. Funk *et al.* [22] present the LRE-TL algorithm, which modifies LLREF with various overhead reductions, as well as extends it to both the uniform multiprocessor and sporadic task problem variants. LRE-TL also does away with the sorting of events by laxity at each secondary event within the T-L Plane. It uses a pair of heaps to order the running and idle tasks and determine when the next secondary event will occur. This reduces the computational overhead of processing at each secondary event from $O(n)$ to $O(\lg n)$.

Finally, in a recent work [41], a new algorithm called U-EDF (**U**nfair scheduling algorithm based on **EDF**) has been proposed. U-EDF starts with a DP-Fair algorithm, but relaxes the proportional fairness assumption in order to decrease the need for preemptions and migrations. While not proven to be optimal, U-EDF correctly scheduled more than thousand randomly generated task sets. In all those experiments, U-EDF significantly reduced the average number of preemptions and migrations per job when compared with existing optimal algorithms.

2.4.4 A Note on Performance

Because DP-Wrap is designed to be simple and instructive, not optimized for performance, we do not expect it to outperform more optimized and complex optimal algorithms. The RUN algorithm presented in Chapter 3 *does* significantly outperform all prior optimal algorithms, and so extensive simulation results are presented there, some of which compare the performance of DP-Wrap with EKG and LLREF. In brief, because LLREF does a significant amount of unnecessary preempting, DP-Wrap incurs about a third as many context switches and migrations as LLREF[4]. At 100%

[4]In actuality, as LLREF does not assign jobs to processors, we had to devise a greedy method for this in an attempt to measure its migration performance.

processor utilization, DP-WRAP and EKG have exactly the same number of migrations, as they migrate exactly the same tasks at the same time. EKG does only about 3/4 as many context switches, as it is running uniprocessor EDF on the non-migrating tasks on each processor. As processor utilization decreases, EKG's performance relative to DP-WRAP improves, as EKG's processor grouping heuristic is designed to take advantage of task set slack.

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, so 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 will also have 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, it does its slice scheduling up front), but each slice is scheduled differently, and the complexity due to time quantum rounding is high.

2.5 Conclusions

Our work on DP-FAIR algorithms was not meant to improve upon existing optimal algorithms in terms of performance. Rather, we sought to improve understanding of existing solutions, and find common features of these solutions to use as a framework for future developments. With our DP-FAIR scheduling theory and rules, we provide extremely simple, almost obvious, scheduling guidelines which are nonetheless sufficient to guarantee optimality. Because our theory easily explains the correctness of all previous competitive algorithms, we have managed to unify all prior work on this problem into one simple framework. We have also provided DP-WRAP as a minimal yet competitive

example of how to translate this theory into a workable algorithm.

Unfortunately, the imposition of all system deadlines upon all tasks under DP-Fair creates a large overhead on the system. Between its release and deadline, a long job might be divided into a dozen or more time slices, each of which requires turning the job on and off to meet its intermediate proportional work requirements. These overconstraints on the system impose a large number of potentially unnecessary context switches and migrations. Ultimately, in order to make substantial improvements in efficiency, we had to abandon the constraints of DP-Fair and move in another direction. In the next chapter, we will see a brand new approach to optimal scheduling that provides a substantial reduction in operational overhead.

Chapter 3

RUN : A Peek at the Future

3.1 Introduction

In this chapter, we introduce RUN, the first optimal scheduling algorithm that is not based on the DP-Fair rules. While these rules make optimal scheduling very simple, they do impose a large number of job deadlines on the system, and subsequently result in a large number of unnecessary context switches and migrations. Our new RUN algorithm uses the novel construction of *dual scheduling* to achieve optimal results with far fewer overconstraints, and results in (on average) 80% fewer context switches and migrations than known DP-Fair algorithms. To motivate dual scheduling, we will once again use greedy schedulers as an inspiration.

3.1.1 Scheduling Duality

Recall Example 2.1 from Section 2.1.2:

$$\mathcal{T} \;=\; \{\, \tau_1 = (10, 9),\ \tau_2 = (10, 9),\ \tau_3 = (20, 4) \,\} \;\; .$$

In Figure 2.1 and the subsequent discussion, we saw that no greedy approach could schedule this task set because no greedy algorithm would recognize the collective idle time requirement of τ_1 and τ_2, or recognize the "joint idle time" event that occurs at $t = 8$. However, that is not strictly true; it *is* possible for a greedy scheduler to detect the required event at $t = 8$ if we schedule the tasks' idle time rather than their work time using the *dual* of this task set.

Suppose we wish to schedule a set $\mathcal{T}$ of n tasks that fully utilizes m processors, *i.e.,* $\rho(\mathcal{T}) = m$. As noted in Theorem 2.1, no processor idle time is allowed in a valid schedule, so at each time instance, exactly m tasks must be executing. Since $n - m$ tasks must always be idle, determining which tasks will be idle is just as important as (and is, in fact, fully equivalent to) determining which tasks will be executing. The parallel system for scheduling idle time instead of execution time is known as the *dual*

system. We construct this dual system by creating n *dual tasks* which represent the idle time requirements of the original tasks, and scheduling them on $n - m$ *dual processors*.

For the task set $\mathcal{T}$ from Example 2.1, for instance, since $\tau_1 = (10, 9)$ must be idle for exactly 1 out of every 10 time units, it's dual task will be $\tau_1^* = (10, 1)$. In this way we construct the dual task set

$$\mathcal{T}^* \;=\; \{\, \tau_1^* = (10,1),\ \tau_2^* = (10,1),\ \tau_3^* = (20,16) \,\}\,.$$

These dual tasks have total rate $\rho(\mathcal{T}^*) = 1/10 + 1/10 + 16/20 = 1$, and so may be correctly scheduled on a single processor via EDF (recall that EDF is optimal on uniprocessor systems). Figure 3.1(a) shows the EDF schedule of $\mathcal{T}^*$. Dual tasks τ_1^* and τ_2^* share the earliest deadline of 10, and have 1 unit of work in their first period, so these are scheduled during intervals $[0, 1)$ and $[1, 2)$, respectively. At $t = 2$, only τ_3^* with a deadline of 20 has work remaining, and so it is scheduled until τ_1^* and τ_2^* release new jobs at $t = 10$. As all tasks now share the deadline of 20, we allow τ_3^* to continue executing, so as to avoid a pointless context switch (ties in EDF may be broken arbitrarily). When it finishes at $t = 18$, τ_1^* and τ_2^* may successively execute the 1 unit of work of their second jobs.

Because $\mathcal{T}^*$ is the dual of $\mathcal{T}$, Figure 3.1(a) shows the scheduling of the *idle time* of the tasks of $\mathcal{T}$. We may easily translate this into a schedule for $\mathcal{T}$ by idling each τ_i whenever τ_i^* is executing, and by executing τ_i whenever τ_i^* is idle. The resultant schedule of $\mathcal{T}$ is shown in Figure 3.1(b). Because EDF produced a valid schedule for $\mathcal{T}^*$, we now also have a valid schedule for $\mathcal{T}$.

We now formalize the dual system. Given the problem of scheduling a full utilization task set

$$\mathcal{T} \;=\; \{\, \tau_1 = (p_1, c_1),\ \ldots,\ \tau_n = (p_n, c_n) \,\}$$

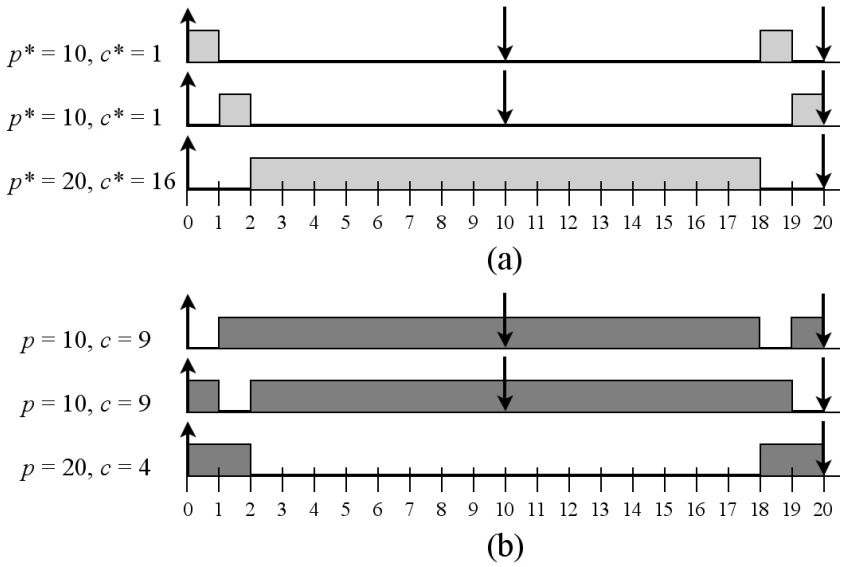


Figure 3.1: **A Schedule and Its Dual**
(a) shows the dual task set $\mathcal{T}^*$ scheduled using EDF. (b) shows the schedule of $\mathcal{T}$ that is inferred from the dual schedule.

on m processors (which we henceforth refer to as the *primal system*), the *dual scheduling problem* requires scheduling

$$\mathcal{T}^* \;=\; \{\, \tau_1^* = (p_1, p_1 - c_1),\; \ldots,\; \tau_n^* = (p_n, p_n - c_n) \,\}$$

on $n - m$ processors. Given any schedule (valid or otherwise) of $\tau_1, \ldots, \tau_n$, the *dual schedule* of $\tau_1^*, \ldots, \tau_n^*$ is the schedule that has τ_i^* executing precisely when τ_i is idle, and vice versa. Consequently, a Work Complete event in the primal is a Zero Laxity event in the dual, and vice versa. We easily verify that

$$\rho(\mathcal{T}^*) \;=\; \sum_{i=1}^{n} \frac{p_i - c_i}{p_i} \;=\; \sum_{i=1}^{n} \left(1 - \frac{c_i}{p_i}\right) \;=\; n - \sum_{i=1}^{n} \frac{c_i}{p_i} \;=\; n - \rho(\mathcal{T}) \;=\; n - m\,,$$

so that $\mathcal{T}^*$ is feasible on $n - m$ processors. Finally, note that the dual of a dual is just the primal.

As observed in our example, duality yields a simple solution to the multipro-

cessor scheduling problem when we have one more task than processor.

Lemma 3.1. Any scheduling problem with m processors and $m + 1$ tasks (where the total rate of the tasks is m) may be scheduled by applying EDF to the uniprocessor dual.

Proof. This follows directly from the above discussion and the fact that EDF is optimal for uniprocessor scheduling. □

From the primal's view, the above translates into the following scheduling policy:

1. Schedule all tasks *except* the one with the earliest deadline
2. The only preemptive scheduling events needed are ZERO LAXITY and JOB RELEASE times

It seems strange from the primal's point of view, but we will never need to explicitly schedule WORK COMPLETE events, any more than we need to schedule ZERO LAXITY events for EDF on a uniprocessor; if the task set is feasible, these will attend to themselves.

3.1.2 RUN Overview

Unfortunately, $n > m + 1$ in most task systems, so by itself Lemma 3.1 is mostly a curiosity. However, since the general dual contains $n - m$ processors, reducing the number of tasks in the primal will reduce the number of dual processors. We can accomplish this by packing multiple tasks together into a single aggregate task, or *server*. Once we have a dual with fewer processors, we can also consolidate the dual tasks, compute the dual of the dual, and obtain even fewer processors. This packing/dual operation is known as a *reduction*. A sequence of reductions will result in a hierarchy of virtual systems, which terminates in one or more uniprocessor systems.

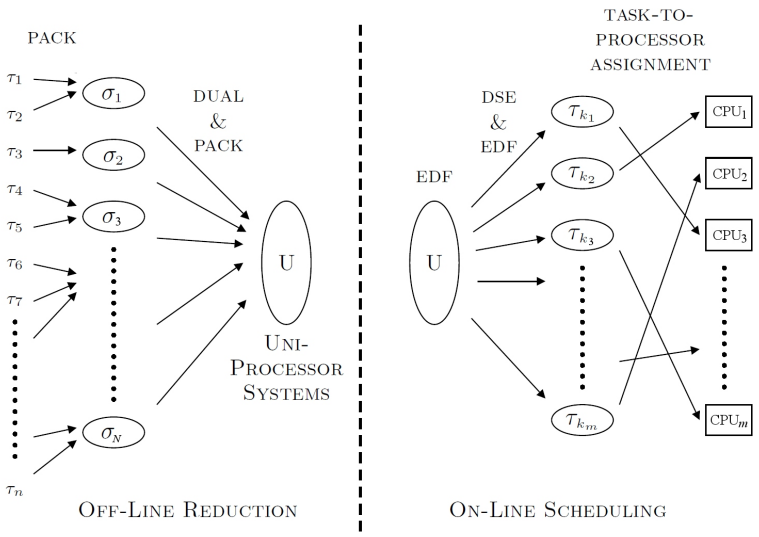


Figure 3.2: **RUN Global Scheduling Approach**
The RUN off-line reduction combines the PACK and DUAL operations. The RUN on-line scheduling uses the DSE rule, the EDF algorithm for servers, and a job-to-processor assignment scheme.

This reduction process may be carried out off-line before scheduling begins. Once we have our reduction sequence, the uniprocessor system(s) may be scheduled on-line using EDF. From this a schedule may be derived for the original m processor system. This is the RUN (**R**eduction to **UN**iprocessor) algorithm in a nutshell, and is summarized in Figure 3.2.

The RUN algorithm represents a significant advancement for both theory and scheduling efficiency. The first contributions are the theoretical building blocks of *Dual Scheduling Equivalence* (DSE) and *limited proportional fairness*. DSE uses the dual system to derive feasible schedules for the primal system, and represents a new way of finding feasible schedules. While DP-FAIR algorithms enforce proportional fairness on every task at every deadline, RUN only requires that the tasks on a server *collectively* receive their fair share of processor time at the deadlines *of that server*. By placing weaker over-constraints on the system, we can reduce the overhead of excess context switches and migrations.

From these components, we derive the RUN scheduling algorithm. RUN presents the following advantages:

- Through a sequence of off-line reduction operations, RUN allows the complicated multiprocessor scheduling problem to be solved by the much simpler application of EDF to uniprocessor systems.
- RUN significantly outperforms existing optimal algorithms in terms of preemptions and migrations. Run has a theoretical upper bound of $O(\log m)$ average preemptions per job on m processors, and an observed average of less than 3 preemptions per job in each of our simulations.
- RUN reduces naturally to the simple and efficient Partitioned EDF on any system where Partitioned EDF would find a correct schedule.

3.2 Additional Modeling and Notation

In order to accommodate duality and servers, we will need to expand and alter some of our notation and system modeling from Chapter 2. Duality requires 100% utilization, so that at all times, m tasks are executing and $n - m$ tasks are idle. As a result, we will not consider sporadic tasks or arbitrary deadlines in this chapter, as both of these lead to idle processors. However, we will extend our task model to a generalization of periodic tasks.

3.2.1 Fixed-Rate Tasks

When we combine multiple tasks into a server, that server will have all the deadlines of the component tasks, and a rate equal to the summed rates of those components. Since the component tasks generally won't share deadlines, the server won't be periodic, and its jobs won't be identical. We will use $J.a$, $J.d$, and $J.c$ to refer to a

job J's arrival time, deadline, and workload (WCET), respectively. Thus J is required to execute for $J.c$ time during the interval $[J.a, J.d)$.

The server itself will be described by its rate and arrival times. Since servers are just virtual tasks which release a sequence of jobs in a virtual system, the concepts of *task* and *server* are largely interchangeable. We will examine servers in further detail in in Section 3.3, but we introduce our generalized task model here:

Definition 3.1 (Fixed-Rate Task)**.** Let $\rho \leqslant 1$ be a positive real number and A a countably infinite set of non-negative integers which includes zero. The *fixed-rate task* τ with rate ρ and arrival times A, denoted $\tau{:}(\rho, A)$, releases an infinite sequence of jobs satisfying the following properties:

1. A job of τ arrives at time t if and only if $t \in A$
2. A job J of τ arriving at time $J.a$ has deadline $J.d = \min\{t \in A \mid t > J.a\}$
3. The workload of job J is $J.c = \rho(J.d - J.a)$ □

Periodic tasks are just a special case of fixed-rate tasks. Given a periodic task $\tau = (p, c)$, we may represent this as a fixed rate task $\tau{:}(\rho, A)$, where $\rho = c/p$ and $A = \{kp \mid k \in \mathbb{N}\}$. As all tasks (and servers) in this chapter are fixed-rate tasks, we shall henceforth simply refer to them as "tasks".

Since we will have tasks and servers at multiple virtual levels, we may no longer simply index all tasks with $i = 1, \ldots, n$. Consequently, we denote the rate and arrival set of a task τ with $\rho(\tau)$ and $A(\tau)$, respectively. As implied by item (2) of Definition 3.1, we assume the implicit deadline model, *i.e.,* that the deadline of τ's current job is equal to the arrival time of its next job. Consequently, $A(\tau) \setminus \{0\}$ is the set of τ's deadlines, and each task has exactly one active job at any time.

3.2.2 Schedules

We will now modify our definition of "schedule" slightly. Traditionally, as in Section 1.3, a schedule specifies which tasks are running *on which processors* at all times. However, dual scheduling is only concerned with *which tasks* are running (or idle) at any give time, and not with processor assignment. Consequently, we will exclude processor assignment from our definition of "schedule" in this chapter. Once we've determined *which* tasks are running, determining processor assignment is relatively easy, and will be addressed in Section 3.6.1.

Definition 3.2 (Schedule)**.** Consider a set of jobs $\mathcal{J}$ (typically generated by a set of tasks) on a platform of m identical processors. Let $e_{J,t}$ denote the work remaining for job J at time t. A (legal) *schedule* Σ is a function from the set of all non-negative times t onto the power set of $\mathcal{J}$ such that (i) $|\Sigma(t)| \leqslant m$ for all t, and (ii) if $J \in \Sigma(t)$, then $J.a \leqslant t$ and $e_{J,t} > 0$. Thus $\Sigma(t)$ represents the set of jobs executing at time t. □

Condition (ii) requires that we only execute jobs which have arrived and have work remaining. Note also that $\Sigma(t)$ is a *set* of jobs, meaning that no job is selected multiple times in a single time instant, *i.e.,* no job can execute simultaneously on different processors. These two facts imply that the above definition is, in fact, for *legal* schedules (see Section 1.3). As before, a *valid schedule* is one in which all deadlines are met.

3.2.3 Fully Utilized System

A system of m processors is *fully utilized* by a task set $\mathcal{T}$ provided that (i) the total rate $\rho(\mathcal{T})$ of $\mathcal{T}$ is equal to m; (ii) all jobs always require their full WCET times; and (iii) all tasks release a job a time zero. Henceforth, we will only consider fully utilized systems. We will observe, however, that this assumption does not actually restrict our task model.

First, if the total rate of $\mathcal{T}$ is less than m, idle (dummy) tasks may be inserted as needed to make up the difference. In fact, the careful placement of these dummy tasks may significantly improve system performance by allowing us to partially or fully partition our task set onto fixed processors (see Section 3.6.1).

Second, assume that a job J has a WCET estimate of $J.c$, but that it completes after consuming only $c' < J.c$ units of processor time. In such a case, the system can easily simulate $J.c - c'$ of J's execution by blocking a processor accordingly. We may thus assume that a job's WCET estimate is always correct, and that each job J of τ executes for exactly $J.c = \rho(\tau)(J.d - J.a)$ time during the interval $[J.a, J.d)$.

Third, suppose that some task τ has its initial job arrive at some time $s > 0$, and that s is known at the outset. We may then add a dummy job J_0 with arrival time 0, deadline s, and execution time $J_0.c = \rho(\tau)s$.

So without loss of generality, we henceforth assume that all systems are fully utilized. One consequence of this is that, in any valid schedule Σ on m identical processors, we must have $|\Sigma(t)| = m$ for all times t.

3.2.4 A Simple Example

Before we formally define servers, or the DUAL or PACK operations, we will give a simple example of the RUN algorithm, to provide a context for understanding the subsequent exposition. Figure 3.3 illustrates all the involved steps. Let us start with a set of five periodic tasks:

$$\mathcal{T} = \{\, \tau_1 = (10, 2),\ \tau_2 = (15, 9),\ \tau_3 = (20, 6),\ \tau_4 = (15, 6),\ \tau_5 = (30, 15) \,\}$$

Since sum of the rates is $\rho(\mathcal{T}) = 2/10 + 9/15 + 6/20 + 6/15 + 15/30 = 2$, we wish to schedule $\mathcal{T}$ on $m = 2$ processors (Figure 3.3(a)).

Our first step is to pack tasks with lower rates together into aggregate fixed-

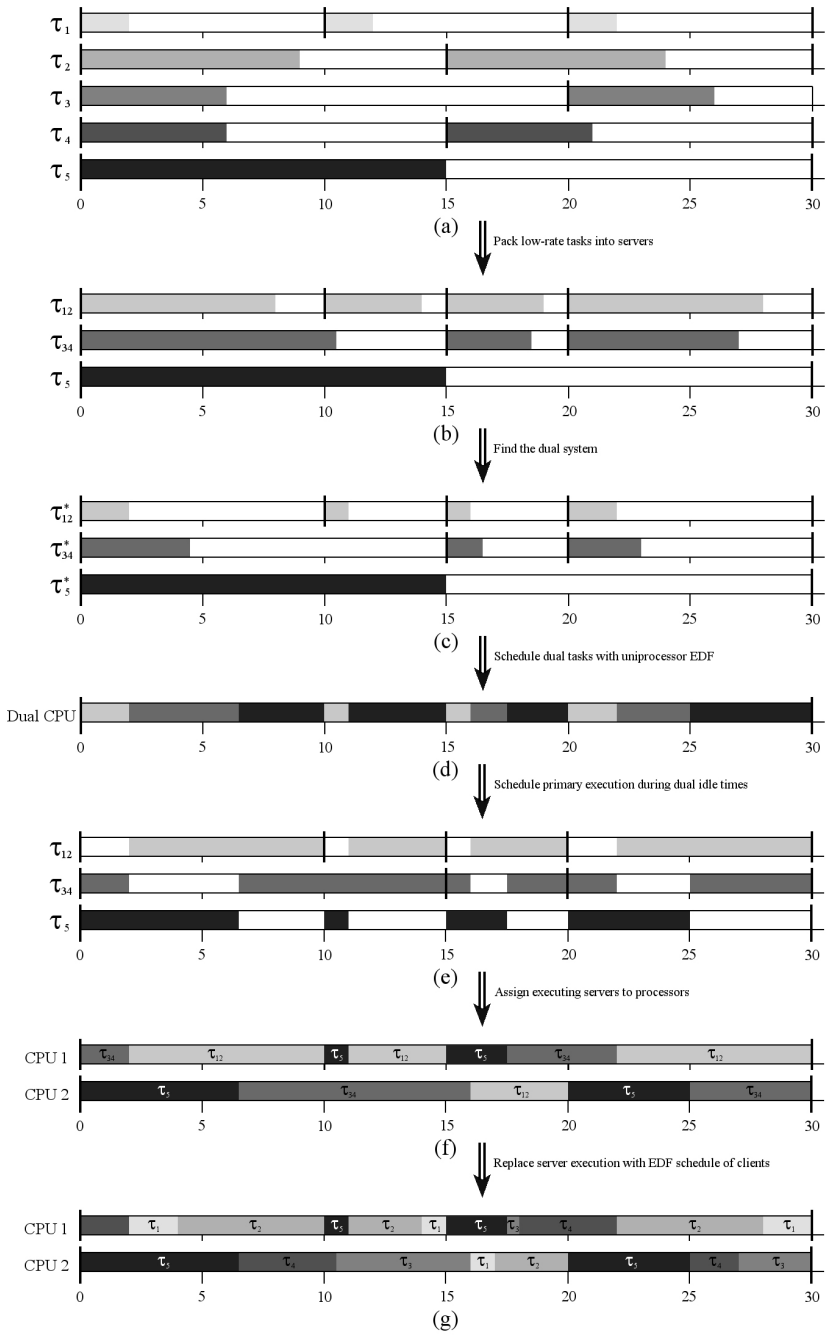


Figure 3.3: **A Simple Preview of RUN**
This demonstrates a simple execution of RUN on 5 tasks and 2 processors.

rate tasks (servers). We will pack τ_1 and τ_2 together into τ_{12} with rate $2/10+9/15 = 4/5$, and deadlines at all multiples of 10 or 15. Similarly, we pack τ_3 and τ_4 together into τ_{34}, giving us our new packed task set

$$\mathcal{T}_P = \{\ \tau_{12}{:}(4/5, 10\mathbb{N} \cup 15\mathbb{N}),\ \ \tau_{34}{:}(7/10, 15\mathbb{N} \cup 20\mathbb{N}),\ \ \tau_5{:}(1/2, 30\mathbb{N})\ \} \quad ,$$

recalling that $\tau_i{:}(\rho_i, A_i)$ is a fixed-rate task with rate ρ_i and arrival times A_i (Figure 3.3(b)).

Our next step is to find the dual of this packed set of tasks (Figure 3.3(c)):

$$\mathcal{T}_P^* = \{\ \tau_{12}^*{:}(1/5, 10\mathbb{N} \cup 15\mathbb{N}),\ \ \tau_{34}^*{:}(3/10, 15\mathbb{N} \cup 20\mathbb{N}),\ \ \tau_5^*{:}(1/2, 30\mathbb{N})\ \} \quad .$$

Recall that a dual task has the same deadlines and complimentary rate to its primal task. Since the dual rates sum to one, we may schedule $\mathcal{T}_P^*$ on a single (virtual dual) processor using EDF. τ_{12}^* has the earliest deadline at $t = 10$, so we run it to completion at time $t = 2$. τ_{34}^* has the next deadline at $t = 15$, so we execute its $3/10 \times 15 = 4.5$ units of work from $t = 2$ to $t = 6.5$. And so forth. The complete EDF schedule for $\mathcal{T}_P^*$ through $t = 30$ is shown in Figure 3.3(d).

From this EDF dual schedule of $\mathcal{T}_P^*$, we may infer a schedule for $\mathcal{T}_P$. Specifically, τ_i will execute precisely when τ_i^* is idle, and vice versa. This will tell us *when* tasks in $\mathcal{T}_P$ must execute (Figure 3.3(e)); assigning tasks to processors is a simple matter, and is shown in Figure 3.3(f), but we'll save the details for Section 3.6.1.

Finally, since τ_{12} is just a server (placeholder) for the execution of its clients τ_1 and τ_2, we replace the execution of τ_{12} with the EDF scheduling of τ_1 and τ_2, and similarly for τ_{34}, τ_3, and τ_4. The final RUN schedule is shown in Figure 3.3(g). □

With this simple example as motivation, we proceed with the formal definitions of servers, duality, packing, and reductions that define the RUN algorithm.

3.3 Servers

RUN's reduction from multiprocessor to uniprocessor systems is enabled by aggregating tasks into *servers*. We treat servers as tasks with a sequence of jobs, but they are not actual tasks in the system; each server is a proxy for a collection of *client* tasks. In any instant when a server is running, its allocated processor time is actually being used by one of its clients. A server's clients are scheduled via some internal scheduling mechanism.

Since we treat servers as tasks, the rate of a server can never be greater than one; consequently, this section focuses only on uniprocessor systems. We precisely define the concept of servers (Section 3.3.1) and show how they correctly schedule the client tasks associated with them (Section 3.3.2). We return to multiprocessors in the following section.

3.3.1 Server model and notations

A server for a set of tasks is defined as follows:

Definition 3.3 (Server/Client)**.** Let $\mathcal{T}$ be a set of tasks with total rate given by $\rho(\mathcal{T}) = \sum_{\tau \in \mathcal{T}} \rho(\tau) \leqslant 1$. A *server* σ for $\mathcal{T}$, denoted $\text{ser}(\mathcal{T})$, is a virtual task with rate $\rho(\mathcal{T})$, arrival times $A(\sigma) = \cup_{\tau \in \mathcal{T}} A(\tau)$, and some internal scheduling policy to schedule the tasks in $\mathcal{T}$. $\mathcal{T}$ is the set of σ's *clients*, and is denoted $\text{cli}(\sigma)$. □

We refer to a job of any client of σ as a *client job* of σ. If σ is a server and Γ a set of servers, then $\text{ser}(\text{cli}(\sigma)) = \sigma$ and $\text{cli}(\text{ser}(\Gamma)) = \Gamma$.

As we will see in Section 3.4.2, the packing of clients into servers is done off-line prior to execution, and remains static during on-line scheduling. It is therefore unambiguous to define the rate $\rho(\sigma)$ of server σ to be $\rho(\text{cli}(\sigma))$. Since servers are themselves tasks, we may also speak of a server for a set of servers. And since a server may contain only a single client task, the concepts are largely interchangeable.

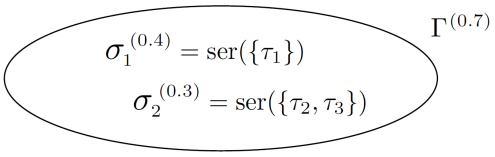


Figure 3.4: **A Two-Server Set**
A set of two servers for three tasks. The notation $X^{(\mu)}$ indicates that $\rho(X) = \mu$.

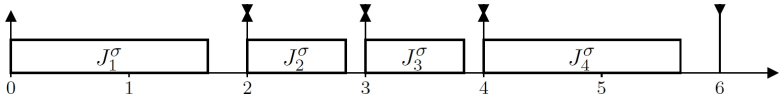
By Definition 3.3, the execution requirement of a server σ in any interval $[a_h, a_{h+1})$ equals $\rho(\sigma)(a_{h+1} - a_h)$, where a_h and a_{h+1} are consecutive arrival times in $A(\sigma)$. Then the workload for job J of server σ with $J.a = a_h$ and $J.d = a_{h+1}$ equals $J.c = e_{J,J.a} = \rho(\sigma)(J.d - J.a)$, just as with a "real" job. However, since a server σ is a just proxy for its clients, the jobs of σ are just budgets of processor time allocated to σ so that its clients may execute. These *budget jobs* of σ may be viewed as σ simply replenishing its budget for each interval $[a_h, a_{h+1})$. Similarly, if J is the current job of σ at time t, then $e_{J,t}$ is just the budget of σ remaining at time t.

As an example, consider Figure 3.4, where Γ is a set comprised of the two servers $\sigma_1 = \text{ser}(\{\tau_1\})$ and $\sigma_2 = \text{ser}(\{\tau_2, \tau_3\})$ for the tasks τ_1, and τ_2 and τ_3, respectively. If $\rho(\tau_1) = 0.4$, $\rho(\tau_2) = 0.2$ and $\rho(\tau_3) = 0.1$, then $\rho(\sigma_1) = 0.4$ and $\rho(\sigma_2) = 0.3$. Also, if $\sigma = \text{ser}(\Gamma)$ is the server in charge of scheduling σ_1 and σ_2, then $\Gamma = \text{cli}(\sigma) = \{\sigma_1, \sigma_2\}$ and $\rho(\sigma) = 0.7$.

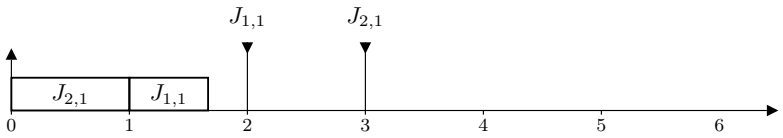
Task sets and servers with summed rates of one are a key part of RUN's construction. We therefore define *unit sets* and *unit servers*, both of which can be feasibly scheduled on one processor.

Definition 3.4 (Unit Set/Unit Server)**.** A set Γ of servers is a *unit set* if $\rho(\Gamma) = 1$. The server $\text{ser}(\Gamma)$ for a unit set Γ is a *unit server*. □

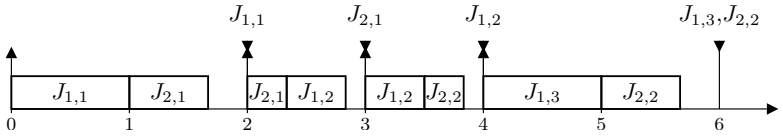
We say that a server meets its deadlines if all of its budget jobs meet theirs. Even if this is the case, the server must employ an appropriate scheduling policy to ensure that its clients also meet their deadlines.



(a) Valid schedule of the budget jobs of σ. J_h^σ is the h^{th} job of σ.



(b) Job $J_{1,1}$ misses its deadline when σ prioritizes job $J_{2,1}$ of τ_2.



(c) A valid schedule of jobs of τ_1 and τ_2 by σ.

Figure 3.5: **Client Deadline Misses in a Valid Server Schedule**
Schedule of τ_1:$(1/2, 2\mathbb{N})$ and τ_2:$(1/3, 3\mathbb{N})$ by a single server σ on a dedicated processor, where $A(\sigma) = \{2, 3, 4, 6, \ldots\}$ and $\rho(\sigma) = 5/6$. σ meets its deadlines (a), but it's client may either miss (b) or meet (c) their deadlines, depending on how they are scheduled by σ.

For example, consider two periodic tasks τ_1:$(1/2, 2\mathbb{N})$ and τ_2:$(1/3, 3\mathbb{N})$ (rates are 1/2 and 1/3, and periods are 2 and 3, respectively, and initial arrival times are zero). Consider a server σ scheduling these two tasks on a dedicated processor. We have $A(\sigma) = \{0, 2, 3, 4, 6, \ldots\}$ and $\rho(\sigma) = 5/6$. σ's first job J_1^σ will arrive at $J_1^\sigma.a = 0$, and will have deadline $J_1^\sigma.d = 2$ and workload $e_{J_1^\sigma,0} = \rho(\sigma)(2 - 0) = 5/3$, *i.e.,* σ has a budget of 5/3 for the interval $[0, 2)$.

Figure 3.5(a) shows a valid schedule for the first four budget jobs of σ. But suppose that σ employs a scheduling policy for its clients where τ_2 is given priority over τ_1 (see Figure 3.5(b), where $J_{i,j}$ represents the j^{th} job of τ_i). Then $J_{2,1}$ will consume one unit of time before $J_{1,1}$ begins its execution. The remaining server budget $e_{J_1^\sigma,1} = 2/3$ will be insufficient to complete $J_{1,1}$'s workload of 1 by its deadline at time 2. We see that a server meeting its deadlines does not insure that its clients will meet theirs.

If, on the other hand, σ's scheduling policy had prioritized τ_1 at time zero, this deadline miss would be avoided. This is the case with the optimal EDF scheduling policy, which is shown in Figure 3.5(c).

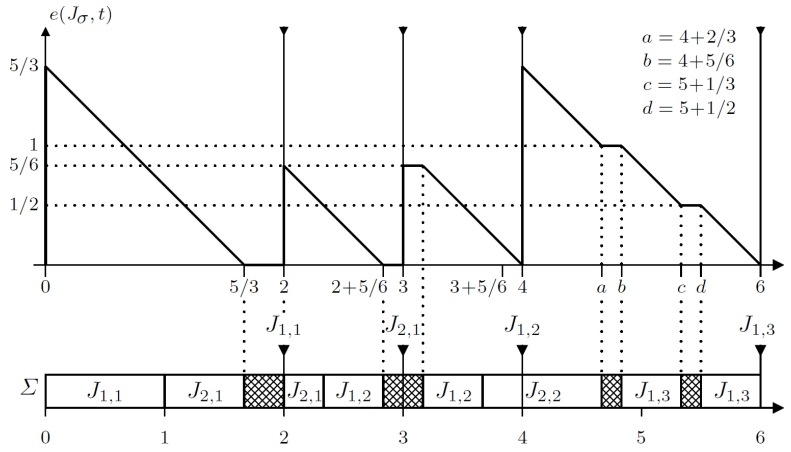


Figure 3.6: **Server Budget and Client Jobs**
$\text{cli}(\sigma) = \{\tau_1{:}(1/2, 2\mathbb{N}),\ \tau_2{:}(1/3, 3\mathbb{N})\}$ and $\rho(\sigma) = 5/6$. The graph shows budget (work) remaining in four consecutive server jobs; below is the schedule of client jobs. Crosshatched regions represents execution of external jobs.

3.3.2 EDF Server

We will henceforth use EDF as our servers' scheduling policy, as it is optimal, simple, and efficient.

Rule 1 (EDF Server)**.** Servers in RUN will be *EDF servers*, which schedule their client jobs with EDF.

As an illustrative example, consider a set of two periodic tasks $\mathcal{T} = \{\tau_1{:}(0.4, 2\mathbb{N}),\ \tau_2{:}(0.2, 3\mathbb{N})\}$. Since $\rho(\mathcal{T}) = 0.6 \leqslant 1$, we can define an EDF server σ to schedule $\mathcal{T}$ such that $\text{cli}(\sigma) = \mathcal{T}$ and $\rho(\sigma) = 0.6$. Figure 3.6 shows both the evolution of $e_{J_\sigma,t}$ during interval $[0, 6)$ and the schedule Σ of $\mathcal{T}$ by σ on a single processor. In this figure, i_j represents the j-th job of τ_i. During intervals $[1\frac{2}{3}, 2)$, $[2\frac{5}{6}, 3\frac{1}{6})$, $[4\frac{2}{3}, 4\frac{5}{6})$ and $[5\frac{1}{3}, 5\frac{1}{2})$, the execution of σ is replaced with execution of external events represented by crosshatched regions.

Note that a unit EDF server σ has rate $\rho(\sigma) = 1$ and must execute continuously in order to meet its clients' deadlines. Deadlines of σ have no effect, since budgets are replenished (a new budget job arrives) the instant they are depleted (at the old budget

job's deadline).

Theorem 3.2. The EDF server $\sigma = \text{ser}(\Gamma)$ of a set of servers Γ produces a valid schedule of Γ when $\rho(\Gamma) \leqslant 1$ and all jobs of σ meet their deadlines.

Proof. By treating the servers in Γ as tasks, we can apply well known results for scheduling task systems. For convenience, we assume that σ executes on a single processor; this need not be the case in general, so long as σ does not execute on multiple processors in parallel.

Recall from Definition 3.3 that $\rho(\Gamma) = \sum_{\sigma_i \in \Gamma} \rho(\sigma_i)$. We first prove the theorem for $\rho(\Gamma) = 1$, then use this result for the case of $\rho(\Gamma) < 1$.

Case $\rho(\Gamma) = 1$ **:** Let $\eta_\Gamma(t, t')$ be the execution demand within a time interval $[t, t')$, where $t < t'$. This demand gives the sum of all execution requests (*i.e.,* jobs) that arrive no earlier than t and have deadlines no later than t'. By Definition 3.3, this quantity is bounded above by

$$\eta_\Gamma(t, t') \leqslant (t' - t) \sum_{\sigma_i \in \Gamma} \rho(\sigma_i) = t' - t \tag{3.1}$$

It is known [5,6] that there is no valid schedule for Γ if and only if there is some interval $[t, t')$ such that $\eta_\Gamma(t, t') > t' - t$. Since Equation 3.1 implies that this cannot happen, some valid schedule for Γ must exist. Because σ schedules Γ using EDF and EDF is optimal [5, 33], σ must produce a valid schedule.

Case $\rho(\Gamma) < 1$ **:** In order to use the result for case $\rho(\Gamma) = 1$, we introduce a slack-filling task τ', as illustrated in Figure 3.6, where $A(\tau') = A(\sigma)$ and $\rho(\tau') = 1 - \rho(\sigma)$. We let $\Gamma' = \Gamma \cup \{\tau'\}$, and let σ' be an EDF server for Γ'. Since $\rho(\Gamma') = 1$, σ' produces a valid schedule for Γ'.

Now consider the scheduling window $W_J = [J.a, J.d]$ for some budget job J of

σ. Since $A(\tau') = A(\sigma)$, τ' also has a job J' where $J'.a = J.a$ and $J'.d = J.d$. Since σ' produces a valid schedule, τ' and σ do exactly $\rho(\tau')(J.d - J.a)$ and $\rho(\sigma)(J.d - J.a)$ units of work, respectively, during W_J. Since there are no deadlines or arrivals between $J.a$ and $J.d$, the workload of τ' may be arbitrarily rearranged or subdivided within the interval W_J without compromising the validity of the schedule. We may do this within all scheduling windows of σ so as to reproduce *any* schedule of σ where it meets its deadlines. Finally, since σ and σ' both schedule tasks in Γ with EDF, σ will produce the same *valid* schedule for Γ as σ', giving our desired result. □

As noted above, a server and its clients may migrate between processors, as long as no more than one client executes at a time. This will allow us to schedule multiple servers on a multiprocessor platform.

3.3.3 Benefits of the RUN Server Model

Compared to the proportional fairness in DP-Fair approaches to optimal scheduling, the *limited proportional fairness* required by RUN's server mechanism enforces a much weaker notion of fairness. In DP-Fair algorithms, every task must do its proportional share of work between all system deadlines. The budget jobs of a server, on the other hand, only require that its clients *collectively* receive their proportional share of processor time between every *client* deadline. For example, if the three clients of a server have summed rate of 0.8, then proper scheduling of the servers' jobs ensures that these three clients share 80% of a processor's capacity between any two of their deadlines. The servers' allocation of its budget to its clients need not be proportionally fair at any particular point, so long as its internal scheduling mechanism (*e.g.,* EDF) ensures that its clients meet their deadlines. Nonetheless, according to Theorem 3.2, this limited proportional fairness is sufficient to guarantee the correct scheduling of a server's clients. Because this approach applies much weaker over-constraints to the sys-

tem than traditional proportional fairness, it incurs a significantly lower overhead of preemptions and migrations for optimal scheduling.

Unlike periodic tasks, fixed-rate tasks do not, and need not, make all of their arrival times known at the outset. It is sufficient that they have implicit deadlines, *i.e.,* that there are neither gaps nor overlaps between a task's consecutive jobs. In order for a server to set the deadline for its next budget job, it only needs to know the next deadline of each of its clients. This is also sufficient for an EDF server to make its scheduling selections from among its clients. Thus, unlike periodic tasks, which implicitly provide *all* their arrival times at the outset, fixed-rate tasks need only make their *next* arrival time known in order to be scheduled on-line by an EDF server.

3.4 RUN Off-Line Reduction

In this section, we describe the DUAL and PACK operations, which are used iteratively in an off-line procedure to reduce a multiprocessor task system to a collection of uniprocessor systems. The DUAL operation is just the formal mechanism for expressing the previously introduced dual system. Recall that, for a system with n tasks and m processors, the dual system will require $n - m$ processors. The PACK operation aggregates multiple low-rate tasks into a single high-rate task (server). This packing decreases n, and consequently, decreases the number of processors in the dual system. Since the dual rate of τ is computed as $\rho(\tau^*) = 1 - \rho(\tau)$, this also means that the duals of these packed servers will generally have low rates, and may themselves be packed into even fewer servers. Given this synergy, we compose the two operations into a single REDUCE operation, which, when applied iteratively, eventually reduces our original system down to one or more uniprocessor systems. In Section 3.5 we will show how the EDF schedules for these uniprocessor systems may be transformed back into a schedule for the original multiprocessor system.

3.4.1 DUAL Operation

The introduction of duality in Section 3.1.1 shows a special case wherein $m+1$ tasks are to be scheduled on m processors. In such a case, the dual task set has an total rate of one, and may therefore be scheduled on a single processor. The primal schedule is then easily inferred, as shown in Figure 3.1. We will now formalize these results, and extend them to general systems of fixed-rate tasks (servers).

Definition 3.5 (Dual Server)**.** The *dual server* σ^* of a server σ is a server with the same deadlines as σ and with rate $\rho(\sigma^*)$ equal to $1 - \rho(\sigma)$. If Γ is a set of servers, then its dual set Γ^* is the set of dual servers to those in Γ, *i.e.,* $\sigma \in \Gamma$ if and only if $\sigma^* \in \Gamma^*$. □

The dual of a unit server (which has rate $\rho(\sigma) = 1$ and must execute continuously in order to meet its clients' deadlines) is a *null server*, which has rate $\rho(\sigma^*) = 0$ and never executes.

Definition 3.6 (Dual Schedule)**.** Let Γ be a set of servers and Γ^* be its dual set. Two schedules Σ of Γ and Σ^* of Γ^* are duals if, for all times t and all $\sigma \in \Gamma$, $\sigma \in \Sigma(t)$ if and only if $\sigma^* \notin \Sigma^*(t)$; that is, σ executes exactly when σ^* is idle, and vice versa. □

σ, Γ, and Σ are referred to as *primal* relative to their duals σ^*, Γ^*, and Σ^*. As with any good notion of "duality", we find that the dual of the dual is just the primal, *i.e.,* $(\sigma^*)^* = \sigma$, $(\Gamma^*)^* = \Gamma$ and $(\Sigma^*)^* = \Sigma$. In fact, it is this property of dual schedules that motivated the avoidance of task-to-processor assignment in Definition 3.2. Here we are only concerned with *which* tasks (or servers) are executing at any given time; task-to-processor assignment may be handled as a separate step (see Section 3.6.1).

We now establish Dual Scheduling Equivalence (DSE) which states that the schedule of a primal set of servers is correct precisely when its dual schedule is correct.

Theorem 3.3 (Dual Scheduling Equivalence)**.** Let Γ be a set of $n = m + k$ servers with $k \geqslant 1$ and $\rho(\Gamma) = m$, an integer. For a schedule Σ of Γ on m processors, let Σ^* and Γ^* be their respective duals. Then $\rho(\Gamma^*) = k$, and so Γ^* is feasible on k processors. Further, Σ is valid if and only if Σ^* is valid.

Proof. First,

$$\rho(\Gamma^*) = \sum_{\sigma^* \in \Gamma^*} \rho(\sigma^*) = \sum_{\sigma \in \Gamma} (1 - \rho(\sigma)) = n - \rho(\Gamma) = k \ ,$$

so k processors are sufficient to schedule Γ^*. Next, we prove that if Σ is valid for Γ then Definition 3.2 implies that Σ^* is valid for Γ^*.

Because Σ is a valid schedule on m processors and we assume full utilization, Theorem 2.1 indicates that Σ always executes m distinct tasks. The remaining $k = n - m$ tasks are idle in Σ, and so are exactly the tasks executing in Σ^*. Hence Σ^* is always executing exactly k distinct tasks on its k (virtual dual) processors. Since Σ is valid, any job J of server $\sigma \in \Gamma$ does exactly $J.c = \rho(\sigma)(J.d - J.a)$ units of work between its arrival $J.a$ and its deadline $J.d$. During this same time, σ^* has a matching job J^* where $J^*.a = J.a$, $J^*.d = J.d$, and

$$\begin{aligned} J^*.c &= \rho(\sigma^*)(J^*.d - J^*.a) \\ &= (1 - \rho(\sigma))(J.d - J.a) \\ &= (J.d - J.a) - J.c \end{aligned}$$

That is, J^*'s execution requirement during the interval $[J.d, J.a)$ is exactly the length of time that J must be idle. Thus, as J executes for $J.c$ during this interval in Σ, J^* executes for $J^*.c$ in Σ^*. Consequently, J^* satisfies condition (ii) of Definition 3.2 and also meets its deadline. Since this holds for all jobs of all dual servers, Σ^* is a valid schedule for Γ^*.

The converse also follows from the above argument, since $(\Sigma^*)^* = \Sigma$. $\square$

Once again, see Figure 3.1 for a simple illustration. We now summarize this dual scheduling rule for future reference.

Rule 2 (Dual Server)**.** At any time, execute in Σ the servers of Γ whose dual servers are not executing in Σ^*, and vice versa.

Finally, we define the DUAL operation $\mathcal{D}$ from a set of servers Γ to its dual set Γ^* as the bijection which associates a server σ with its dual server σ^*, *i.e.,* $\mathcal{D}(\sigma) = \sigma^*$. We adopt the usual notational convention for the image of a subset. That is, if $f : A \to B$ is a function from A to B and $A' \subseteq A$, we understand $f(A')$ to mean $\{f(a) \mid a \in A'\}$. For example, $\mathcal{D}(\Gamma) = \{\sigma^* \mid \sigma \in \Gamma\} = \Gamma^*$.

Note that Theorem 3.3 does not provide any particular rules for generating a schedule; it merely establishes the equivalence of scheduling n tasks on m processors with scheduling their dual tasks on $n - m$ virtual processors. This can be advantageous when $n - m < m$, so that the number of processors is reduced in the dual system, as seen in Figure 3.1. The PACK operation ensures this desirable outcome.

3.4.2 PACK Operation

We cannot generally expect to find $n - m < m$. Consider the example of a set $\mathcal{T}$ of 5 tasks, each with rate 2/5. Here, $n = 5$, $m = \rho(\mathcal{T}) = 2$, and $n - m = 3 > 2$. The dual tasks in $\mathcal{T}^*$ have rates of 3/5, and will require 3 processors to schedule. Here, the DUAL operation has made the system larger, not smaller.

However, suppose we combine two pairs of tasks in $\mathcal{T}$ into two new tasks (servers), each with rates of $2/5 + 2/5 = 4/5$. Then the dual of this new 3 task set has rates 1/5, 1/5, and 3/5, and may be scheduled on a single processor. This is because this dual system requires $n - m$ processors, and we have just reduced n by two without changing m. This is the essence of the PACK operation.

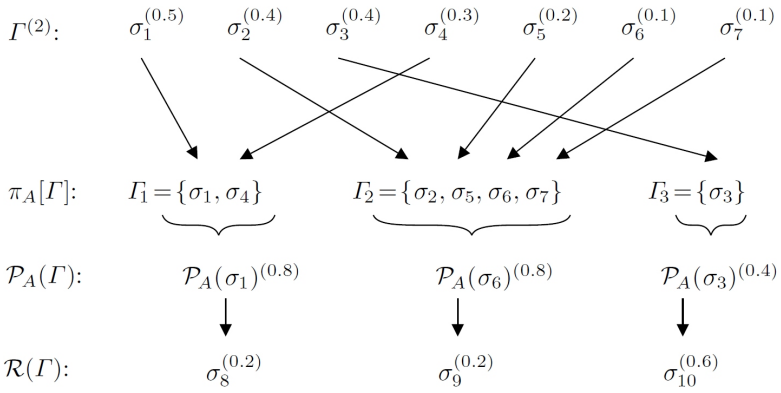


Figure 3.7: **A Single Reduction Level**
Packing, PACK, and DUAL operations applied to $\Gamma = \{\sigma_1, \sigma_2, \ldots, \sigma_7\}$, resulting in a reduction to a unit set of three servers $\{\sigma_8, \sigma_9, \sigma_{10}\}$ with $\sigma_8 = \mathcal{D} \circ \mathcal{P}_A(\sigma_1)$, $\sigma_9 = \mathcal{D} \circ \mathcal{P}_A(\sigma_6)$, $\sigma_{10} = \mathcal{D} \circ \mathcal{P}_A(\sigma_3)$. The notation $X^{(\mu)}$ means that $\rho(X) = \mu$.

Definition 3.7 (Packing)**.** Let Γ be a set of servers. A partition $\{\Gamma_1, \Gamma_2, \ldots, \Gamma_k\}$ of Γ is a *packing* of Γ if $\rho(\Gamma_i) \leqslant 1$ for all i and $\rho(\Gamma_i) + \rho(\Gamma_j) > 1$ for all $i \neq j$. An algorithm A is a *packing algorithm* if it partitions any set of servers into a packing. In such a case, we denote the packing of Γ produced by A as $\pi_A[\Gamma]$. □

An example of packing a set Γ of 7 servers into three sets Γ_1, Γ_2 and Γ_3, is illustrated by rows 1 and 2 of Figure 3.7. We will use traditional bin-packing algorithms as our packing algorithms. These are described in detail in Appendix B.

Theorem 3.4. The first-fit, worst-fit and best-fit bin-packing algorithms are packing algorithms.

Proof. At any step of these three algorithms, a new bin can only be created if the current task to be allocated does not fit in any of the existing partially filled bins. Now suppose that $\rho(\Gamma_i) + \rho(\Gamma_j) \leqslant 1$ for some two bins, where Γ_j was created after Γ_i. Then the first item τ placed in Γ_j must have $\rho(\tau) \leqslant \rho(\Gamma_j) \leqslant 1 - \rho(\Gamma_i)$. That is, τ fits in bin Γ_i, contradicting the need to create Γ_j for it. Therefore $\rho(\Gamma_i) + \rho(\Gamma_j) > 1$ must hold for any pair of bins. □

Hereafter, we assume that A is a packing algorithm. We now wish to assign a

dedicated server to schedule each part Γ_i of the partition $\pi_A[\Gamma]$.

Definition 3.8 (PACK operation)**.** Let Γ be a set of servers, A a packing algorithm, and $\pi_A[\Gamma]$ the resultant packing. For each $\Gamma_i \in \pi_A[\Gamma]$, we assign it a dedicated server $\mathrm{ser}(\Gamma_i)$. The PACK *operation* $\mathcal{P}_A$ is the mapping from Γ onto the set $\{\mathrm{ser}(\Gamma_i) \mid \Gamma_i \in \pi_A[\Gamma]\}$ of these servers such that, if $\Gamma_i \in \pi_A[\Gamma]$ and $\sigma \in \Gamma_i$, then $\mathcal{P}_A(\sigma) = \mathrm{ser}(\Gamma_i)$. That is, $\mathcal{P}_A$ associates a client $\sigma \in \Gamma$ with the server $\mathrm{ser}(\Gamma_i)$ responsible for scheduling σ and the other clients which π_A packs into Γ_i. □

By our notational conventions, $\mathcal{P}_A(\Gamma) = \{\mathrm{ser}(\Gamma_i) \mid \Gamma_i \in \pi_A[\Gamma]\}$ is the set of all aggregated servers responsible for scheduling Γ. For example, Figure 3.7 shows that $\mathcal{P}_A(\sigma_6) = \mathrm{ser}(\Gamma_2)$ is the aggregated server responsible for scheduling all the clients in Γ_2, and that $\mathcal{P}_A(\Gamma) = \{\mathcal{P}_A(\sigma_1), \mathcal{P}_A(\sigma_6), \mathcal{P}_A(\sigma_3)\}$.

Definition 3.9 (Packed Server Set)**.** A set of servers Γ is *packed* if it is a singleton, or if $|\Gamma| \geqslant 2$ and for any two distinct servers σ and σ' in Γ, $\rho(\sigma) + \rho(\sigma') > 1$ and $\mathrm{cli}(\sigma) \cap \mathrm{cli}(\sigma') = \{\}$. □

Consequently, the packing of a packed server set Γ is the collection of singleton sets $\{\{\sigma\}\}_{\sigma \in \Gamma}$. As we are only concerned with packings that result from the application of some packing algorithm A, we will henceforth drop the implicitly understood $_A$ from $\pi[\Gamma]$ and $\mathcal{P}$.

3.4.3 REDUCE Operation

We now compose the DUAL and PACK operations into the REDUCE operation. As will be shown, a sequence of reductions transforms a multiprocessor scheduling problem into a collection of uniprocessor scheduling problems. This off-line transformation is the cornerstone of the RUN algorithm.

Lemma 3.5. Let Γ be a packed set of servers, and let $\mathcal{D}(\Gamma)$ be the dual set of Γ. Suppose we apply a PACK operation $\mathcal{P}$ to $\mathcal{D}(\Gamma)$. Then

$$|\,\mathcal{P}\circ\mathcal{D}(\Gamma)\,| \;\leqslant\; \left\lceil \frac{|\Gamma|+1}{2} \right\rceil \;.$$

Proof. Let $n = |\Gamma|$. Since Γ is packed, there is at most one server σ in Γ such that $\rho(\sigma) \leqslant 1/2$. This implies that at least $n-1$ servers in $\mathcal{D}(\Gamma)$ have rates less than $1/2$. When these $n-1$ dual servers are packed, they will be, at a minimum, paired off. Thus, the packing will partition $\mathcal{D}(\Gamma)$ into at most $\lceil (n-1)/2 \rceil + 1$ subsets. Hence, $|\mathcal{P}\circ\mathcal{D}(\Gamma)| \leqslant \lceil (n+1)/2 \rceil$. □

Thus, packing the dual of a packed set reduces the number of servers by at least (almost) half. Since we will use this pair of operations repeatedly, we define the REDUCE operation to be their composition.

Definition 3.10 (REDUCE Operation)**.** Given a set of servers Γ and a packing algorithm A, the REDUCE *operation* on a server σ in Γ, denoted $\mathcal{R}(\sigma)$, is the composition of the DUAL operation $\mathcal{D}$ with the PACK operation $\mathcal{P}$ associated with A, *i.e.,* $\mathcal{R}(\sigma) = \mathcal{D}\circ\mathcal{P}(\sigma)$. □

Figure 3.7 illustrates the steps of the REDUCE operation $\mathcal{R}$. As we intend to apply REDUCE repeatedly until we are left with only unit servers, we now define a *reduction sequence*.

Definition 3.11 (Reduction Level/Sequence)**.** Let $i \geqslant 1$ be an integer, Γ a set of servers, and σ a server in Γ. The operator $\mathcal{R}^i$ is recursively defined by $\mathcal{R}^0(\sigma) = \sigma$ and $\mathcal{R}^i(\sigma) = \mathcal{R}\circ\mathcal{R}^{i-1}(\sigma)$. $\{\mathcal{R}^i\}_i$ is a *reduction sequence*, and the server system $\mathcal{R}^i(\Gamma)$ is said to be at *reduction level* i. □

Theorem 3.8 will show that a reduction sequence on a server set Γ with $\rho(\Gamma) = m$ always arrives at a collection of *terminal unit servers*. Table 3.1 shows 10 tasks (or

Table 3.1: Sample Reduction and Proper Subsets

	Server Rate									
$\mathcal{R}^0(\Gamma)$	0.6	0.6	0.6	0.6	0.6	0.8	0.6	0.6	0.5	0.5
$\mathcal{P}(\mathcal{R}^0(\Gamma))$	0.6	0.6	0.6	0.6	0.6	0.8	0.6	0.6	**1**→	
$\mathcal{R}^1(\Gamma)$	0.4	0.4	0.4	0.4	0.4	0.2	0.4	0.4	0	
$\mathcal{P}(\mathcal{R}^1(\Gamma))$	0.8		0.8		0.4	**1**→				
$\mathcal{R}^2(\Gamma)$	0.2		0.2		0.6	0				
$\mathcal{P}(\mathcal{R}^2(\Gamma))$	**1**									

servers) transformed into a unit server via two REDUCE operations and a final PACK. Notice that two unit servers appear before the final reduction level (indicated in the table by **1**→). A subset of tasks in Γ which eventually gets reduced to a terminal unit server is referred to as a *proper subset*. A proper subset always has an integral summed rate, and may be scheduled on a subset of processors independently from the rest of Γ. A proper subset, all the intermediate primal and dual servers down to and including the unit server, and the processor(s) assigned to them, are collectively known as a *proper subsystem*. The three proper subsystems in Table 3.1 are separated by blank columns.

When intermediate unit servers are encountered prior to the final reduction level, there are two ways of dealing with them: we may isolate them or ignore them. Under the first approach, when an intermediate unit server is found, it is treated as a terminal unit server. The proper subsystem that generated it is isolated from the rest of the system, assigned its own processors, and scheduled independently. For example, in Table 3.1, the proper subset $\{0.8, 0.6, 0.6\}$ would be assigned two processors, which would be scheduled independently from the other four processors and seven tasks. This approach is more efficient in practice, because these independent subsystems do not impose events on, or migrate into, the rest of the system. The simulations detailed in Section 3.6.6 use this approach.

We may also deal with intermediate unit servers by ignoring them. The dual of a unit server is a null server. If we ignore the fact that we have found an intermediate unit server, then its dual null server simply gets packed into some other server in the

next level. In Table 3.1, observe the unit server that results from the tasks $\{0.5, 0.5\}$. The unit server's dual has rate 0, and may be packed along with the dual servers $\{0.2, 0.4, 0.4\}$ into another unit server. Under this approach, we do not consider these intermediate unit servers to be *terminal* unit servers, nor do we consider them to be the root of a proper subsystem. Under this view, Table 3.1 contains only one terminal unit server and one proper subsystem. It is still possible for a system to have more than one terminal unit server, but only if they all appear at the same final reduction level. For the remainder of this section and the next, we will adopt this approach for dealing with intermediate unit servers, because it simplifies our exposition and proofs.

We now provide two intermediate results which will be used to establish Theorem 3.8.

Lemma 3.6. Let Γ be a set of servers, and let $\mathcal{P}(\Gamma)$ be the set of servers assigned to the packing $\pi[\Gamma]$ of some PACK operation on Γ. Then $\rho(\Gamma) \leqslant |\mathcal{P}(\Gamma)|$. Further, if not all servers in $\mathcal{P}(\Gamma)$ are unit servers, then $\rho(\Gamma) < |\mathcal{P}(\Gamma)|$

Proof. Since $\rho(\sigma) \leqslant 1$ for all servers $\sigma \in \mathcal{P}(\Gamma)$,

$$\rho(\Gamma) \;=\; \sum_{\Gamma_i \in \pi[\Gamma]} \rho(\Gamma_i) \;=\; \sum_{\sigma \in \mathcal{P}(\Gamma)} \rho(\sigma) \;\leqslant\; \sum_{\sigma \in \mathcal{P}(\Gamma)} 1 \;=\; |\,\mathcal{P}(\Gamma)\,|\,.$$

If not all servers in $\mathcal{P}(\Gamma)$ are unit servers, then $\rho(\sigma) < 1$ for some $\sigma \in \mathcal{P}(\Gamma)$, and the inequality above is strict. □

Lemma 3.7. Let Γ be a packed set of servers, not all of which are unit servers. If $\rho(\Gamma)$ is a positive integer, then $|\Gamma| \geqslant 3$.

Proof. If $\Gamma = \{\sigma_1\}$ and σ_1 is not a unit server, then $\rho(\Gamma) < 1$, not a positive integer. If $\Gamma = \{\sigma_1, \sigma_2\}$ is a packed set, then $\rho(\Gamma) = \rho(\sigma_1) + \rho(\sigma_2) > 1$; but $\rho(\Gamma)$ is not 2 unless σ_1 and σ_2 are both unit servers. Thus $|\Gamma|$ is neither 1 nor 2. □

Theorem 3.8 (Reduction Convergence)**.** Let Γ be a set of servers where $\rho(\Gamma)$ is a positive integer. Then for some $p \geqslant 0$, $\mathcal{P}(\mathcal{R}^p(\Gamma))$ is a set of unit servers.

Proof. Let $\Gamma^k = \mathcal{R}^k(\Gamma)$ and $\Gamma^k_{\mathcal{P}} = \mathcal{P}(\Gamma^k)$, and suppose that $\rho(\Gamma^k_{\mathcal{P}})$ is a positive integer. If $\Gamma^k_{\mathcal{P}}$ is a set of unit servers, then $p = k$ and we're done.

Otherwise, according to Lemma 3.7, $|\Gamma^k_{\mathcal{P}}| \geqslant 3$. Observe that

$$\begin{aligned}\Gamma^{k+1}_{\mathcal{P}} &= \mathcal{P}(\Gamma^{k+1}) \\ &= \mathcal{P} \circ \mathcal{R}(\Gamma^k) \\ &= \mathcal{P} \circ \mathcal{D} \circ \mathcal{P}(\Gamma^k) \\ &= (\mathcal{P} \circ \mathcal{D})(\Gamma^k_{\mathcal{P}})\end{aligned}$$

Since $\Gamma^k_{\mathcal{P}}$ is a packed set of servers, Lemma 3.5 tells us that

$$|\Gamma^{k+1}_{\mathcal{P}}| \leqslant \left\lceil \frac{|\Gamma^k_{\mathcal{P}}| + 1}{2} \right\rceil$$

Since $\lceil (x+1)/2 \rceil < x$ whenever $x \geqslant 3$, and we know $|\Gamma^k_{\mathcal{P}}| \geqslant 3$, it follows that

$$|\Gamma^{k+1}_{\mathcal{P}}| < |\Gamma^k_{\mathcal{P}}| \ .$$

Note that packing a set does not change its rate, so $\rho(\Gamma^k) = \rho(\Gamma^k_{\mathcal{P}})$. We've assumed that $\rho(\Gamma^k_{\mathcal{P}})$ is a positive integer, and that $\Gamma^k_{\mathcal{P}}$ are not all unit servers, so Lemma 3.6 tells us that $\rho(\Gamma^k_{\mathcal{P}}) = \rho(\Gamma^k) < |\Gamma^k_{\mathcal{P}}|$. By setting $m = \rho(\Gamma^k_{\mathcal{P}})$ and $n = |\Gamma^k_{\mathcal{P}}|$, so that $m < n$, we may apply Theorem 3.3 to the dual of $\Gamma^k_{\mathcal{P}}$ to deduce that $\rho(\mathcal{D}(\Gamma^k_{\mathcal{P}})) = \rho(\Gamma^{k+1}) = \rho(\Gamma^{k+1}_{\mathcal{P}})$ is also a positive integer.

We now see that $\Gamma^{k+1}_{\mathcal{P}}$ also has positive integer rate, but contains fewer servers than $\Gamma^k_{\mathcal{P}}$. Hence, starting with the packed set $\Gamma^0_{\mathcal{P}} = \mathcal{P}(\Gamma)$, each iteration of $\mathcal{P} \circ \mathcal{D}$ either produces a set of unit servers or a smaller set with positive integer rate. This iteration

Table 3.2: Reduction Example with Different Outcomes.

	First Packing				
$\mathcal{R}^0(\Gamma)$	.4	.4	.2	.2	.8
$\mathcal{P}(\mathcal{R}^0(\Gamma))$	.8		.4		.8
$\mathcal{R}^1(\Gamma)$	.2		.6		.2
$\mathcal{P}(\mathcal{R}^1(\Gamma))$	1				

Second Packing				
.4	.4	.2	.8	.2
1			1	

can only occur a finite number of times, and once $|\Gamma^k_{\mathcal{P}}| < 3$, Lemma 3.7 tells us that $\Gamma^k_{\mathcal{P}}$ must be a set of unit servers; we have found our $p = k$, and are done. □

In other words, a reduction sequence on any set of servers eventually produces a set of unit servers. We will show how to schedule the proper subsystem of each unit server in the next section. First, note that the behavior of the $\mathcal{R}$ operator is dependent on the packing algorithm associated with its PACK operation $\mathcal{P}_A$. For example, Table 3.2 shows two packings of the same set of servers. One produces one unit server after one reduction level and the other produces two unit servers with no reductions. While some packings may be "better" than others (*i.e.,* lead to a more efficient schedule), Theorem 3.8 implicitly proves that all PACK operations "work"; they all lead to a correct reduction to *some* set of unit servers.

3.5 RUN On-Line Scheduling

Now that we have transformed a multiprocessor system into one or more uniprocessor systems, we show how the schedule of the multiprocessor system can be deduced from the (virtual) schedules of the derived uniprocessor systems. First we schedule the clients of the terminal unit servers using EDF. We then iteratively filter this schedule backwards through the reduction hierarchy, using Dual Scheduling Equivalence at DUAL levels, and EDF at PACK levels. This comprises the on-line scheduling portion of our optimal RUN algorithm.

Theorem 3.8 says that a reduction sequence produces a collection of one or

more terminal unit servers. As shown in Table 3.1, the original task set may be partitioned into the proper subsystems associated with these unit servers, which may then be scheduled independently. So without loss of generality, we assume in this section that $\mathcal{T}$ is a proper subset, *i.e.,* that it is handled by a single terminal unit server at the final reduction level.

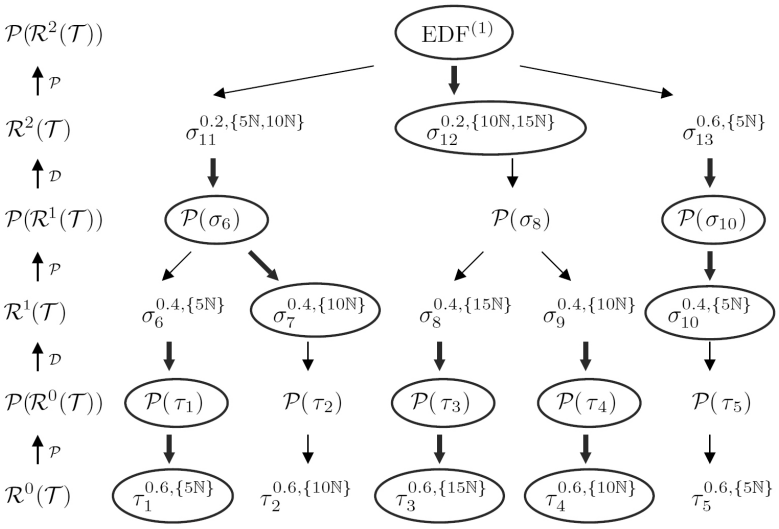
The scheduling process is illustrated by inverting the reduction tables from the previous section and creating a *server tree* whose nodes are the servers generated by iterations of the PACK and DUAL operations. The terminal unit server becomes the root of the server tree, which represents the top-level virtual uniprocessor system. The root's children are the unit server's clients, which are scheduled by EDF.

As an illustrative example, let us consider the first proper subsystem in Table 3.1. To these 5 tasks with rates of 3/5, we will assign periods of 5, 10, 15, 10, and 5. Then our initial task set becomes

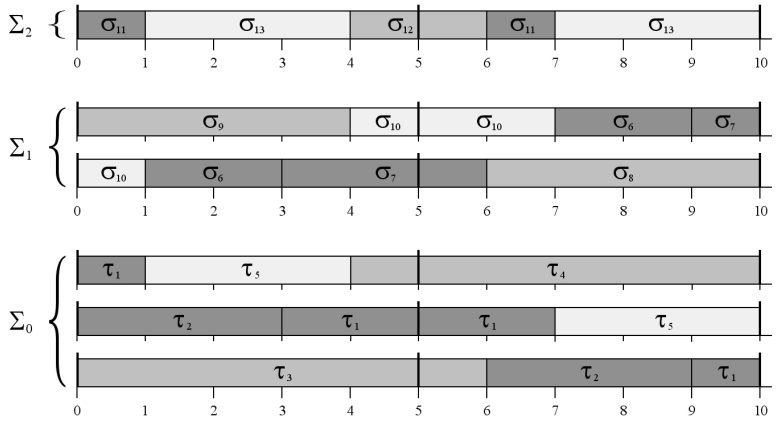
$$\mathcal{T} = \{\tau_1 : (3/5, 5\mathbb{N}),\ \tau_2 : (3/5, 10\mathbb{N}),\ \tau_3 : (3/5, 15\mathbb{N}),\ \tau_4 : (3/5, 10\mathbb{N}),\ \tau_5 : (3/5, 5\mathbb{N})\}.$$

Figure 3.8(a) shows the inverted server tree, with these 5 tasks as the leaves, and the terminal unit server as the root. We demonstrate the scheduling process that occurs at time $t = 4$ by circling the servers chosen for execution. First, the terminal unit server schedules its children using EDF. At $t = 4$, only the child node σ_{12} has any work remaining, so it is chosen to execute (circled). After this, we propagate the circles down the tree using Rules 1 (schedule clients with EDF) and 2 (DSE). For convenience, we restate these here in terms of the server tree:

Rule 1 (EDF Server)**.** If a packed server is executing (circled), execute the child node with the earliest deadline among those children with work remaining; if a packed server is not executing (not circled), execute none of its children.



(a) RUN decision tree at time $t = 4$



(b) Run schedule

Figure 3.8: **RUN Server Tree and Schedules at all Reduction Levels**
$\mathcal{T} = \{\tau_1{:}(3/5, 5\mathbb{N}), \tau_2{:}(3/5, 10\mathbb{N}), \tau_3{:}(3/5, 15\mathbb{N}), \tau_4{:}(3/5, 10\mathbb{N}), \tau_5{:}(3/5, 5\mathbb{N})\}$. Σ_0 is the schedule of $\mathcal{T}$ on 3 physical processors. Σ_1 is the schedule of $\mathcal{R}(\mathcal{T}) = \{\sigma_6, \sigma_7, \sigma_8, \sigma_9, \sigma_{10}\}$ on 2 virtual processors, and Σ_2 is the schedule of $\mathcal{R}^2(\mathcal{T}) = \{\sigma_{11}, \sigma_{12}, \sigma_{13}\}$ on 1 virtual processor.

Rule 2 (Dual Server)**.** Execute (circle) the child (packed server) of a dual server if and only if the dual server is *not* executing (not circled).

Let us continue to follow the branch beneath σ_{12}. σ_{12} is the dual of $\mathcal{P}(\sigma_8)$, which will be idle since σ_{12} is executing. Since $\mathcal{P}(\sigma_8)$ is not executing (not circled), neither are either of its clients. However, these clients, σ_8 and σ_9, are the duals of $\mathcal{P}(\tau_3)$ and $\mathcal{P}(\tau_4)$, respectively, so these two servers will be executing (circled). Finally, $\mathcal{P}(\tau_3)$

and $\mathcal{P}(\tau_4)$ are each servers for a single client, so each of these clients, namely τ_3 and τ_4, will also execute. The full schedule for all levels of the system is shown through time $t = 10$ in Figure 3.8(b), and indeed, we see that τ_3 and τ_4 are executing at time $t = 4$. By similarly propagating the circling of nodes down the tree (Figure 3.8(a)), we see that τ_1 also executes at $t = 4$, while τ_2 and τ_5 are idle.

In practice, we only need to invoke the above scheduler when some subsystem's EDF scheduler generates a scheduling event (*i.e.,* Work Complete or Job Release). In fact, when some EDF server's client has a Work Complete event, we only need to propagate the changes down from that server. For example, if we examine the EDF server $\mathcal{P}(\sigma_6)$ and its clients at time $t = 3$, we see that σ_6 generates a Work Complete event, and σ_7 takes over its execution. While this has the effect of context switching from τ_2 to τ_1 in our schedule of $\mathcal{T}$, the event has no effect on other branches of the tree. On the other hand, because a server inherits its arrival times from its clients, any arrival time of a task in $\mathcal{T}$ will become an arrival time for a client of the terminal unit server, and will cause a scheduler invocation at the top level. Therefore any Job Release event will cause a rescheduling of the entire tree.

Each child server scheduled by a packed server must keep track of its own workloads and deadlines. These workloads and deadlines are based on the clients of the packed server below it. That is, each server node which is not a task of $\mathcal{T}$ simulates being a task so that its parent node can schedule it along with its siblings in its virtual system. The process of setting deadlines and allocating workloads for virtual server jobs is detailed in Section 3.3.1.

The process described so far, from reducing a task set to unit servers, to the scheduling of those tasks with EDF servers and duality, is collectively referred to as the RUN algorithm, and is summarized in Algorithm 1. We now finish proving it correct.

Algorithm 1: Outline of the RUN algorithm

I. OFF-LINE

A. Generate a reduction sequence for $\mathcal{T}$

B. Invert the sequence to form a server tree

C. For each proper subsystem $\mathcal{T}'$ of $\mathcal{T}$

Define the client/server at each virtual level

II. ON-LINE

Upon a scheduling event:

A. If the event is a job arrival at level 0

1. Update deadline sets of servers on path up to root
2. Create jobs for each of these servers accordingly

B. Apply Rules 3 & 4 to schedule jobs from root to leaves, determining the m jobs to schedule at level 0

C. Assign the m chosen jobs to processors, according to some task-to-processor assignment scheme

Theorem 3.9 (RUN correctness)**.** If Γ is a proper set under the reduction sequence $\{\mathcal{R}^i\}_{i \leqslant p}$, then the RUN algorithm produces a valid schedule Σ for Γ.

Proof. Again, let $\Gamma^k = \mathcal{R}^k(\Gamma)$ and $\Gamma^k_{\mathcal{P}} = \mathcal{P}(\Gamma^k)$ with $k < p$. Also, let Σ^k and $\Sigma^k_{\mathcal{P}}$ be the schedules generated by RUN for Γ^k and $\Gamma^k_{\mathcal{P}}$, respectively.

By Definition 3.8 of the PACK operation $\mathcal{P}$, $\Gamma^k_{\mathcal{P}}$ is the set of servers in charge of scheduling the packing of Γ^k. Hence, $\rho(\Gamma^k) = \rho(\Gamma^k_{\mathcal{P}})$. Finally, let $\mu^k = \rho(\Gamma^k) = \rho(\Gamma^k_{\mathcal{P}})$, which, as seen in the proof of Theorem 3.8, is always an integer.

We will work inductively to show that schedule validity propagates down the reduction tree, *i.e.,* that the validity of Σ^{k+1} implies the validity of Σ^k.

Suppose that Σ^{k+1} is a valid schedule for $\Gamma^{k+1} = \mathcal{D}(\Gamma^k_{\mathcal{P}})$ on μ^{k+1} processors, where $k + 1 \leqslant p$. Since $k < p$, $\Gamma^k_{\mathcal{P}}$ is not the terminal level set, and so must contain more than one server, as does its equal-sized dual Γ^{k+1}. Further, since Γ^{k+1} is the dual of a packed set, none of these servers can be unit servers, and so $|\Gamma^{k+1}| > \mu^{k+1}$. The conditions of Theorem 3.3 are satisfied (where $n = |\Gamma^{k+1}|$, $m = \mu^{k+1}$, and $n > m$), so it follows from our assumption of the validity of Σ^{k+1} that $\Sigma^k_{\mathcal{P}} = (\Sigma^{k+1})^*$ is a valid schedule for $\Gamma^k_{\mathcal{P}}$ on μ^k processors.

Also, since $\Gamma^k_{\mathcal{P}}$ is a collection of aggregated servers for Γ^k, it follows from Theorem 3.2 that Σ^k is a valid schedule for Γ^k (*i.e.,* scheduling the servers in $\Gamma^k_{\mathcal{P}}$ correctly ensures that all of their client tasks in Γ^k are also scheduled correctly). Thus the validity of Σ^{k+1} implies the validity of Σ^k, as desired.

Since uniprocessor EDF generates a valid schedule Σ^p for the clients of the terminal unit server at the final reduction level p, it follows inductively that $\Sigma = \Sigma^0$ is valid for Γ on $\rho(\Gamma)$ processors. □

3.6 Assessment

3.6.1 RUN Implementation

The PACK operation described in Section 3.4.2 is carried out by a standard bin-packing algorithm, where "bins" are servers of capacity one, and items are tasks or other servers whose "sizes" are their rates. For our implementation of RUN's PACK step, we use best-fit decreasing bin-packing, as it consistently outperforms other bin-packing heuristics (albeit by only a small margin; see Section 3.6.5 and Appendix B for details). This runs in $O(n \log n)$ time, where n is the number of items being packed.

Whenever an intermediate unit server is encountered during the reduction process, we make it a terminal unit server, and isolate its proper subsystem from the rest of the system, as discussed in Section 3.4.3.

At each scheduler invocation, once the set of m running tasks is determined (as in Figure 3.8(a)), we use a simple greedy task-to-processor assignment scheme. In three passes through these m tasks, we:

1. leave already executing tasks on their current processors
2. assign idle tasks to their last-used processor, when its available, to avoid unnecessary migrations
3. assign remaining tasks to free processors arbitrarily.

Best-fit decreasing bin-packing and EDF are not the only choices for partitioning and uniprocessor scheduling. RUN may be modified so that it reduces to a variety of partitioned scheduling algorithms. Best-fit bin packing can be replaced with any other bin-packing (partitioning) scheme that (i) uses additional "bins" when a proper partitioning onto m processors is not found, and (ii) creates a packed server set. Similarly, any optimal uniprocessor scheduling algorithm can be substituted for EDF. In this way, the RUN scheme can be used as an extension of different partitioned scheduling algorithms, but one that could, in theory, handle cases when a proper partition on m processors can't be found.

3.6.2 Reduction Complexity

We now observe that the time complexity of a reduction procedure is polynomial and is dominated by the PACK operation. However, as there is no optimality requirement on the (off-line) reduction procedure, any polynomial-time heuristic suffices. There are, for example, linear and log-linear time packing algorithms available [12, 24].

Lemma 3.10. If Γ is a packed set of at least 2 servers, then $\rho(\Gamma) > |\Gamma|/2$.

Proof. Let $n = |\Gamma|$, and let $\mu_i = \rho(\sigma_i)$ for $\sigma_i \in \Gamma$. Since Γ is packed, there exists at most one server in Γ, say σ_n, such that $\mu_n \leqslant 1/2$; all others have $\mu_i > 1/2$. Thus, $\sum_{i=1}^{n-2} \mu_i > (n-2)/2$. As $\mu_{n-1} + \mu_n > 1$, it follows that $\rho(\Gamma) = \sum_{i=1}^{n} \mu_i > n/2$. □

Theorem 3.11 (Reduction Complexity)**.** RUN's off-line generation of a reduction sequence for n tasks on m processors requires $O(\log m)$ reduction steps and $O(f(n))$ time, where $f(n)$ is the time needed to pack n tasks.

Proof. Let $\{\mathcal{R}^i\}_{i \leqslant p}$ be a reduction sequence on $\mathcal{T}$, where p is the terminal level described in Theorem 3.8. Lemma 3.5 shows that a REDUCE, at worst, reduces the number of servers by about half, so $p = O(\log n)$.

Since constructing the dual of a system primarily requires computing n dual rates, a single REDUCE requires $O(f(n) + n)$ time. The time needed to perform the entire reduction sequence is described by $T(n) \leqslant T(n/2) + O(f(n) + n)$, which gives $T(n) = O(f(n))$.

Since $\mathcal{T}$ is a full utilization task set, $\rho(\mathcal{T}) = m$. If we let $n' = |\mathcal{P}(\mathcal{T})|$, Lemma 3.10 tells us that $m = \rho(\mathcal{T}) = \rho(\mathcal{P}(\mathcal{T})) > n'/2$. But as $\mathcal{P}(\mathcal{T})$ is just the one initial packing, it follows that p also is $O(\log n')$, and hence $O(\log m)$. □

3.6.3 On-line Complexity

Since the reduction tree is computed off-line, the on-line complexity of RUN can be determined by examining scheduling Rules 1 and 2, and the task-to-processor assignment scheme.

Theorem 3.12 (On-line Complexity)**.** Each scheduler invocation of RUN takes $O(n)$ time, for a total of $O(jn \log m)$ scheduling overhead during any time interval when n tasks releasing a total of j jobs are scheduled on m processors.

Proof. First, let's count the nodes in the server tree. In practice, σ and $\mathcal{D}(\sigma)$ may be implemented as a single object / node. There are n leaves, and as many as n servers in $\mathcal{P}(\mathcal{T})$. Above that, each level has at most (approximately) half as many nodes as the preceding level. This gives us an approximate node bound of $n+n+n/2+n/4+\cdots \approx 3n$.

Next, consider the scheduling process described by Rules 1 and 2. The comparison of clients performed by EDF in Rule 1 does no worse than inspecting each client once. If we assign this cost to the client rather than the server, each node in the tree is inspected at most once per scheduling invocation. Rule 2 is constant time for each node which "dualed". Thus the selection of m tasks to execute is constant time per node, of which there are at most $3n$. The previously described task-to-processor assignment

requires 3 passes through a set of m tasks, and so may be done in $O(m) \leqslant O(n)$ time. Therefore, each scheduler invocation is accomplished in $O(n)$ time.

Since we only invoke the scheduler at WORK COMPLETE or JOB RELEASE events, any given job (real or virtual) can cause at most two scheduler invocations. The virtual jobs of servers are only released at the arrival times of their leaf descendants, so a single real job can cause no more than $O(\log m)$ virtual jobs to be released, since there are at most $O(\log m)$ reduction levels (Theorem 3.11). Thus j real jobs result in no more than $jO(\log m)$ virtual jobs, so a time interval where j jobs arrive will see a total scheduling overhead of $O(jn \log m)$. □

3.6.4 Preemption Bounds

We now prove an upper bound on the average number of preemptions per job through a series of lemmas. To do so, we count the preemptions that a job *causes*, rather than the preemptions that a job *suffers*. Thus, while an arbitrarily long job may be preempted arbitrarily many times, the *average* number of preemptions per job is bounded. When a context switch occurs where A begins running and B becomes idle, we say that A *replaces* B; if the current job of B still has work remaining, we say that A *preempts* B. Because all scheduling decisions are made by EDF, we need only consider the preemptions caused by two types of scheduling events: WORK COMPLETE events (W.C.E.), and JOB RELEASE events (J.R.E.) (which occur concurrently with job deadlines).

Lemma 3.13. Each job from a task or server has exactly one J.R.E. and one W.C.E.. Further, the servers at any one reduction level cannot release more jobs than the original task set over any time interval.

Proof. The first claim is obvious and is merely noted for convenience.

Next, since servers inherit deadlines from their clients and jobs are released at

deadlines, a server cannot have more deadlines, and hence not release more jobs, than its clients. A server's dual has the same number of jobs as the server itself. Moving inductively up the server tree, it follows that a set of servers at one level cannot have more deadlines, or more job arrivals, than the leaf level tasks. □

Lemma 3.14. Scheduling a set $\mathcal{T}$ of $n = m + 1$ tasks on m processors with RUN produces an average of no more than one preemption per job.

Proof. When $n = m+1$, there is only one reduction level and no packing; $\mathcal{T}$ is scheduled by applying EDF to its uniprocessor dual system. We claim that dual J.R.E.s cannot cause preemptions in the primal system.

Since all scheduling is done by applying EDF to $\mathcal{T}^*$ on a single virtual processor, a J.R.E. can only cause a context switch when the arriving job J_h^*, say from task τ^*, has an earlier deadline than, and thus preempts, the previously running job. Let J_h be the corresponding job of τ^*'s primal task τ, and recall that $J_h^*.a = J_h.a = J_{h-1}^*.d = J_{h-1}.d$.

Now let us consider such a context switch, where J_h^* *starts* executing in the dual at its arrival time $J_h^*.a$. In the primal system, τ's previous job J_{h-1} *stops* executing at time $J_{h-1}.d = J_h^*.a$. When a job stops executing at its deadline in a valid schedule, it must be the case that it completes its work exactly at its deadline, and stopping a completed job does not count as a preemption. Thus dual J.R.E.s do not cause preemptions in the primal system. By Lemma 3.13, there can be at most one W.C.E. in the dual, and hence one preemption in the primal, for each job released by a task in $\mathcal{T}$, as desired. □

Lemma 3.15. A context switch at any level of the server tree causes exactly one context switch between two original leaf tasks in $\mathcal{T}$.

Proof. We proceed by induction on the level where the context switch occurs, showing

that a context switch at any level of the server tree causes exactly one context switch in the next level below (less reduced than) it.

Consider some tree level where the switch occurs: suppose we have a pair of client nodes (not necessarily of the same server parent) C_+ and C_-, where C_+ replaces C_-. All other jobs' "running" statuses at this level are unchanged. Let σ_+ and σ_- be their dual children in the server tree (*i.e.,* $C_+ = \sigma_+^*$ and $C_- = \sigma_-^*$), so by Rule 2, σ_- replaces σ_+ (see Figure 3.9 for node relationships).

Now, when σ_+ was running, it was executing exactly one of its client children, call it $C_{+,1}$; when σ_+ gets switched off, so does $C_{+,1}$. Similarly, when σ_- was off, none of its clients were running; when it gets switched on, exactly one of its clients, say $C_{-,1}$, begins executing. Just as the context switch at the higher (more reduced) level only effects the two servers C_+ and C_-, so too are these two clients $C_{+,1}$ and $C_{-,1}$ the only clients at this lower level affected by this operation; thus, $C_{-,1}$ must be replacing $C_{+,1}$. So here we see that a context switch at one client level of the tree causes only a single context switch at the next lower client level of the tree (in terms of Figure 3.9, (i) causes (ii)). This one context switch propagates down to the leaves, so inductively, a context switch anywhere in the tree causes exactly one context switch in $\mathcal{T}$. □

Lemma 3.16. If RUN requires p reduction levels for a task set $\mathcal{T}$, then any J.R.E. by a task $\tau \in \mathcal{T}$ can cause at most $\lceil (p+1)/2 \rceil$ preemptions in $\mathcal{T}$.

Proof. Suppose task τ releases job J at time $J.a$. This causes a job release at each ancestor server node above τ in the server tree (*i.e.,* on the path from leaf τ to the root). We will use Figure 3.9 for reference, and note that this is only meant to represent the portion of the server tree relevant to our discussion.

Let σ be the highest (furthest reduction level) ancestor EDF server of τ on which this J.R.E. causes a context switch among its clients (σ may be the root of the server tree). In such a case, some client of σ (call it C_+) has a job arrive with an

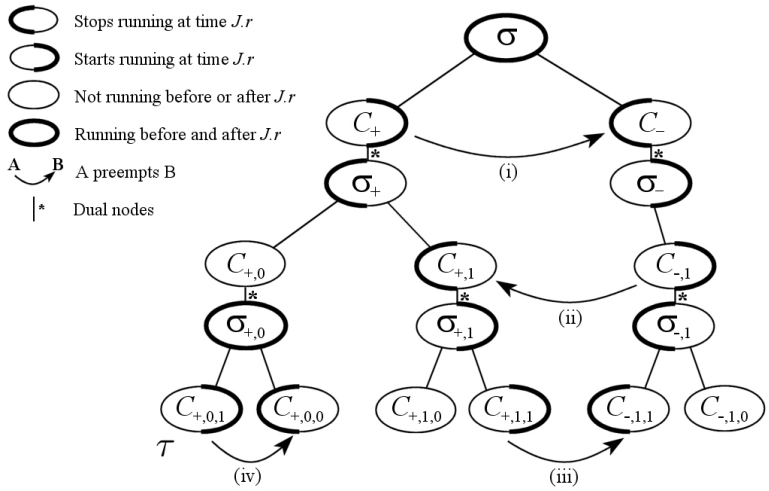


Figure 3.9: **Two Preemptions from One Job Release**
In this 3-level (portion of a) server tree, a job release by τ corresponds to a job release and context switch at the top level (i), which propagates down to the right of the tree (ii, iii). That same job release by τ can cause it to preempt (iv) another client $C_{+,0,0}$ of its parent server $\sigma_{+,0}$.

earlier deadline than the currently executing client (call it C_-), so C_+ preempts C_-. As described in the proof of Lemma 3.15, C_-'s dual σ_- replaces C_+'s dual σ_+, and this context switch propagates down to a context switch between two tasks in $\mathcal{T}$ (see preemption (iii) in Figure 3.9).

However, as no client of σ_+ remains running at time $J.a$, the arrival of a job for τ's ancestor $C_{+,0}$ at this level cannot cause a J.R.E. preemption at this time (it may cause a different client of σ_+ to execute when σ_+ begins running again, but this context switch will be charged to the event that causes σ_+ to resume execution). Thus, when an inherited J.R.E. time causes a context switch at one level, it cannot cause a *different* (second) context switch at the next level down. However, it may cause a second context switch two levels down (see preemption (iv)). Figure 3.9 shows two context switches, (iii) and (iv), in $\mathcal{T}$ that result from a single J.R.E. of τ. One is caused by a job release by τ's ancestor child of the root, which propagates down to another part of the tree (iii). τ's parent server is not affected by this, stays running, and allows τ to preempt its sibling client when its new job arrives (iv).

While σ is shown as the root and τ as a leaf in Figure 3.9, this argument would still apply if there were additional nodes above and below those shown, and τ were a descendant of node $C_{+,0,1}$. If there were additional levels, then τ's J.R.E. could cause an additional preemption in $\mathcal{T}$ for each two such levels. Thus, if there are p reduction levels (*i.e.,* $p+1$ levels of the server tree), a J.R.E. by some original task τ can cause at most $\lceil (p+1)/2 \rceil$ preemptions in $\mathcal{T}$. □

Theorem 3.17. Suppose RUN performs p reductions on task set $\mathcal{T}$ in reducing it to a single EDF system. Then RUN will suffer an average of no more than $\lceil (3p+1)/2 \rceil = O(\log m)$ preemptions per job (and no more than 1 when $n = m+1$) when scheduling $\mathcal{T}$.

Proof. The $n = m+1$ bound comes from Lemma 3.14. Otherwise, we use Lemma 3.13 to count preemptions based on jobs from $\mathcal{T}$ and the two EDF event types. By Lemma 3.16, a J.R.E. by $\tau \in \mathcal{T}$ can cause at most $\lceil (p+1)/2 \rceil$ preemptions in $\mathcal{T}$. The context switch that happens at a W.C.E. in $\mathcal{T}$ is, by definition, not a preemption. However, a job of $\tau \in \mathcal{T}$ corresponds to one job released by each of τ's p ancestors, and each of these p jobs may have a W.C.E. which causes (at most, by Lemma 3.15) one preemption in $\mathcal{T}$. Thus we have at most $p + \lceil (p+1)/2 \rceil = \lceil (3p+1)/2 \rceil$ preemptions that can be attributed to each job from $\mathcal{T}$, giving our desired result since $p = O(\log m)$ (see proof of Theorem 3.12). □

In our simulations, we almost never observed a task set that required more than two reductions. For $p = 2$, Theorem 3.17 gives a bound of 4 preemptions per job. While we never saw more than 3 preemptions per job in our randomly generated task sets, it is possible to do worse. The following 6-task set on 3 processors averages 3.99 preemptions per job, suggesting that our proven bound is tight: $\mathcal{T} = \{(.57, 4000), (.58, 4001), (.59, 4002), (.61, 4003), (.63, 4004), (.02, 3)\}$.

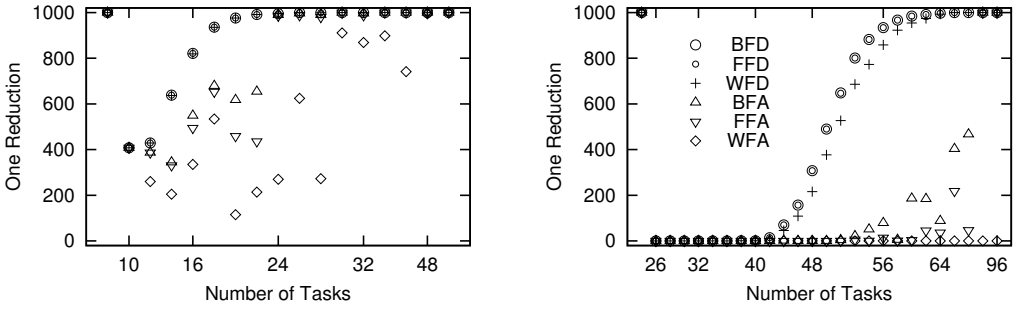
Also, there do exist task sets that require more than 2 reductions. A set of only 11 jobs with rates of 7/11 is sufficient, with a primal reduction sequence of rates:

$$\left\{(11)\ \frac{7}{11}\right\} \rightarrow \left\{(5)\ \frac{8}{11}, \frac{4}{11}\right\} \rightarrow \left\{\frac{10}{11}, \frac{9}{11}, \frac{3}{11}\right\} \rightarrow \{1\}$$

Such constructions require narrowly constrained rates, and randomly generated task sets requiring 3 or more reductions are rare. By generating tens of thousands of task sets, we eventually found a 3-reduction task set on 18 processors, and a 4-reduction set on 24 processors, but these were not part of the thousand-set samples that we ran simulations on. Even when we generated task sets with 100 processors and hundreds of tasks, 3- and 4-reduction sets occurred in fewer than 1 in 600 of the observed sets.

3.6.5 Heuristics

We chose best-fit decreasing as our bin-packing subroutine based on simulations involving multiple bin-packing heuristics (methodology of simulation will be discussed in the next section). By far the strongest indicator of performance, measured in terms of preemptions per job, is the number of reduction levels required by a given task set. And as Table 3.2 shows, the number of reductions is a function of the particular packing used. In all of our various simulations on randomly generated task sets, we only encountered task sets that required 0, 1, or 2 reduction levels, and 0 reduction levels were only found at lower total utilizations. We have tested the best-fit, first-fit and worst-fit bin-packing heuristics with sizes (rates) arranged in both arbitrary and decreasing order (see Appendix B for more details on bin packing algorithms). Figure 3.10 shows the number of full-utilization task sets out of 1000 simulations which require only one reduction level, as a function of the number of tasks. As fewer reduction levels are preferred, we see that packing tasks in decreasing order always outperforms arbitrary order. Although the three decreasing heuristics we tried performed similarly, best-fit



(a) 8 processors (b) 24 processors

Figure 3.10: **Reduction Levels from Various Packing Algorithms**
Number of task sets out of 1000 requiring 1 reduction for RUN simulations on 8 and 24 processors at full utilization. Any task set not requiring 1 reduction required only 2.

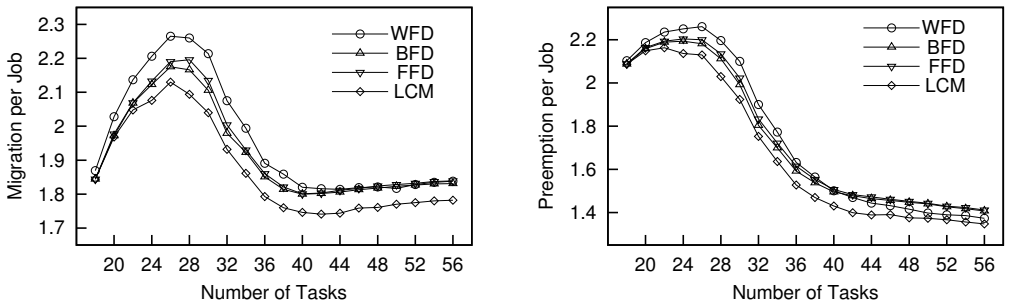


Figure 3.11: **Scheduling Performance of the LCM Packing Algorithm**
Migrations- and preemptions-per-job by RUN using 4 different bin-packing heuristics. Simulations were run on $m = 16$ processors for $n = 18, \ldots, 56$ tasks at 100% utilization.

was consistently the best, and hence was our choice for RUN's implementation.

Since we observed that all decreasing bin-packing heuristics perform similarly, we concluded that, so long as tasks are packed in decreasing rate order, it doesn't much matter which bin a task is placed in, so long as it fits. With this in mind, we tested a different bin-packing criteria, where bins were selected based on compatibility of task periods. As with other decreasing heuristics, tasks are placed in bins one at a time, in decreasing rate order, with a new bin created whenever a task doesn't fit in any existing bin. When a task would fit in multiple bins, we select the bin where the addition of the task caused the smallest increase to the *Least Common Multiple* (LCM) of the periods of tasks already in that bin. For this purpose, the "period" of an aggregated server was

taken to be the LCM of the "periods" of its clients. The benefit of this approach is that, when tasks of compatible (*i.e.,* large common divisor) periods are grouped together in servers, those servers have fewer job releases, and consequently cause fewer scheduler invocations and preemptions. As a simple example, if we have two tasks with period 5, and these are placed on different servers (bins), then each of those servers must be switched on and off every 5 units of time as its jobs are released and completed; if both tasks are on the same server, only one server is burdened with job releases on this period.

Figure 3.11 shows average migrations- and preemptions-per-job for a varying number of tasks simulated on 16 processors. As in Figure 3.10, best-fit slightly outperformed our new "LCM-fit" heuristic in terms of task sets requiring only one reduction level. However, when task set schedules were simulated, the LCM-fit packer suffered 4-5% fewer preemptions and migrations per job than any of the other bin-packers. This is attributable to the reduction in server jobs that comes from grouping tasks of compatible periods onto the same server. In spite of this slightly improved performance, we chose to use the best-fit heuristic for our primary simulations. The LCM-fit bin packer is complex, and its particular benefit is heavily dependent on our choice of randomly generating integral periods in the range [5,100]. We include these results merely to demonstrate that there is potential benefit in grouping tasks according to periods, and that this benefit could be significant in environments where some tasks have strongly compatible periods.

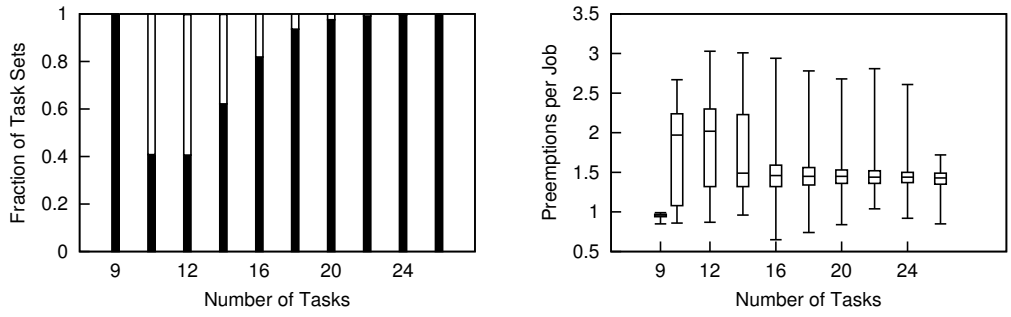
Recall from Section 3.4 that duality is only defined for task sets with 100% utilization. When there is not full utilization, we may fill in the task set slack with one or more dummy tasks. Any collection of dummy tasks with individual rates no more than one and which bring the summed rate to m will suffice. But if we create these dummy tasks intelligently, we may greatly improve scheduling performance. To

this end, we introduce the *slack packing* heuristic to distribute a task system's slack $S(\mathcal{T}) = m - \rho(\mathcal{T})$ among the aggregated servers at the end of the initial PACK step. A bin packing will rarely result in all bins being completely full; however, we may create full bins by adding dummy tasks which precisely fill up their remaining room. Such a filled bin is a unit server, and may be assigned its own dedicated processor. This server, and its client tasks and processor, are scheduled with uniprocessor EDF, and do not interact with the rest of the system. As more servers are filled with dummy tasks and isolated, the system becomes smaller and more efficient to schedule.

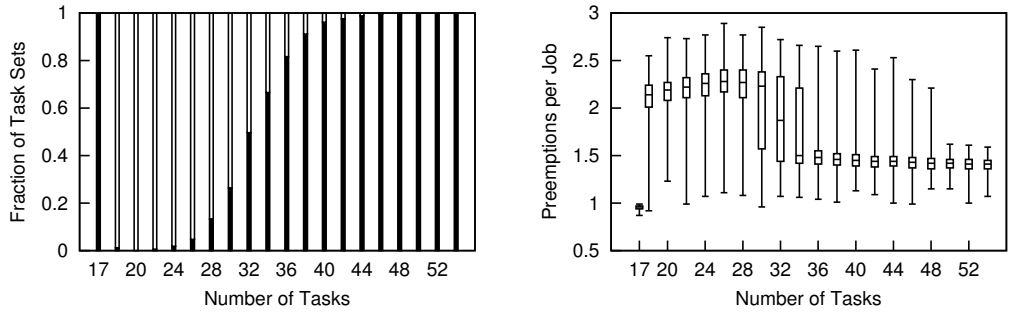
For example, suppose that the task set from Figure 3.8 runs on four processors instead of three. The initial PACK can only place one 0.6 rate task per server. From the 1 unit of slack provided by our fourth processor, we create a dummy task τ_1^d with $\rho(\tau_1^d) = 0.4$ (and arbitrarily large deadline), pack it with τ_1 to get a unit server and give it its own processor. Similarly, τ_2 also gets a dedicated processor. The remaining 0.2 units of slack are put into a third dummy task, which is scheduled along with τ_3, τ_4, and τ_5 on the remaining two processors in the usual fashion. But now τ_1 and τ_2 need never preempt or migrate, so the schedule is more efficient. With 5 processors, this approach yields a fully partitioned system, where each task has its own processor. With low enough utilization, the first PACK usually results in m or fewer servers. In these cases, slack packing gracefully reduces RUN to Partitioned EDF.

3.6.6 Simulation

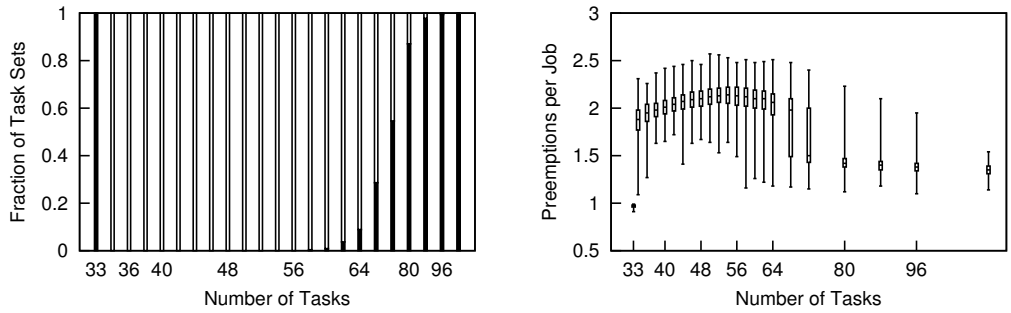
We have evaluated RUN via extensive simulation using task sets generated for various levels of n tasks, m processors, and total utilization $\rho(\mathcal{T})/m$. Task rates were generated in the range of $[0.01, 0.99]$ following a suggested procedure by Emberson *et al.* [18] for the use of Stafford's `randfixedsum()` function [46]. Task periods were drawn independently from a uniform integer distribution in the range $[5, 100]$ and simulations were run for 1000 time units. Values reported for migrations and preemptions



(a) 8 processors



(b) 16 processors



(c) 32 processors

Figure 3.12: **Reductions and Preemptions vs Number of Tasks**
Fraction of task sets requiring 1 (filled box) and 2 (empty box) reduction levels; Distributions of the average number of preemptions per job, their quartiles, and their minimum and maximum values. All RUN simulations on 8, 16 and 32 processor systems at full utilization.

are *per job* averages, that is, total counts were divided by the number of jobs released during the simulation, averaged over all task sets. For each data point shown, 1000 task sets were generated.

For direct evaluation, we generated one thousand random n-task sets for each

value $n = 17, 18, 20, 22, \ldots, 52$ (we actually took n up to 64, but results were nearly constant for $n \geqslant 52$). Each task set fully utilizes a system with 8, 16 or 32 processors. We measured the number of reduction levels and the number of preemption points. Job completion is not considered a preemption point.

The left plots of Figure 3.12 shows the number of reduction levels; none of the task sets generated require more than two reductions. For $m = n + 1$ (9, 17 and 33) tasks, only one level is necessary, as seen in Figure 3.1, and implied by Theorem 3.3. For 8, 16 and 32 processors, we observe that one or two levels are needed for $n \in [10, 22]$, $n \in [18, 44]$ and $n \in [34, 95]$, respectively. None of our observed task sets require a second reduction for $n > 22$, $n > 44$ and $n > 96$. With low average task rates, the first PACK gives servers with rates close to 1; the very small dual rates then sum to 1, yielding the terminal level.

The box-plots in the right column of Figure 3.12 show the distribution of preemption points as a function of the number of tasks. We see a strong correlation between the number of preemptions and number of reduction levels; where there is mostly only one reduction level, preemptions per job is largely independent of the size of the task set. Indeed, for $n \geqslant 16$, $n \geqslant 36$ and $n \geqslant 80$, the median preemption count stays nearly constant just below 1.5.

This correlation between preemptions and reduction levels is shown explicitly in Figure 3.13. In the two plots, we vary number of processors and number of tasks, while keeping the number of tasks in a range where both 1- and 2-reduction task sets are common. We observer that average preemptions per job is nearly constant as processors and tasks vary, and is almost entirely a function of whether a task set requires 1 or 2 reduction levels. Single reduction task sets average about 1.46 preemptions per job, while two reduction task sets average about 2.15 preemptions per job. As can be seen in Figure 3.12, the variance in average preemptions as the number of jobs changes (right-hand plots) tracks very closely with changes in the proportion of one- and two-

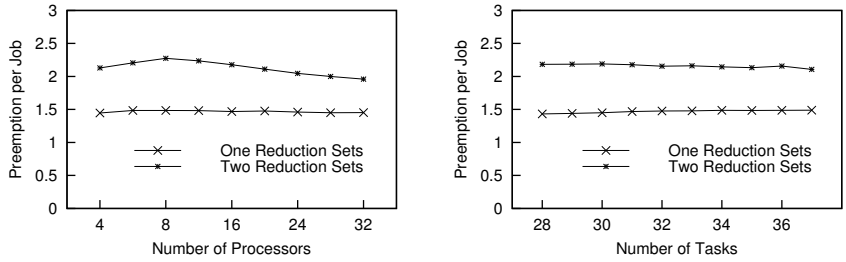


Figure 3.13: **Preemptions for One- and Two-Reduction Task Sets**
Preemptions-per-job by RUN on task sets requiring 1 and 2 reduction levels. In the first plot, $m = 4, \ldots, 32$ and $n = 2m$; in the second plot, $m = 16$ and $n = 28, \ldots, 37$; both show task sets with 100% utilization.

reduction task sets (left-hand plots). Although the range bars in the right-hand figures show considerable variance between task sets, no task set generated for these simulations ever incurred more than 3 average preemptions per job.

Next, we ran comparison simulations against other optimal algorithms. In Figure 3.14, we count migrations and preemptions made by RUN, LLREF [10], EKG [3] and DP-Wrap (with these last two employing the simple *mirroring* heuristic) while increasing processor count from 2 to 32. Most of LLREF's results are not shown to preserve the scale of the rest of the data. Whereas the performance of LLREF, EKG and DP-Wrap get substantially worse as m increases, the overhead for RUN quickly levels off, showing that RUN scales quite well with system size. This is to be expected, as there is an observed upper bound of 3, and a theoretical upper bound of 4, preemptions per job for one- and two-reduction level task sets, and sets requiring more reduction levels are exceedingly rare.

Finally, we simulated EKG, RUN, and Partitioned EDF at lower task set utilizations (LLREF and DP-Wrap were excluded, as they consistently perform worse than EKG). Because 100% utilization is unlikely in practice, and because EKG is optimized for utilizations in the 50-75% range, we felt these results to be of particular interest. For RUN, we employed the slack-packing heuristic. Because this often reduces RUN to Partitioned EDF for lower utilization task sets, we include Partitioned EDF for com-

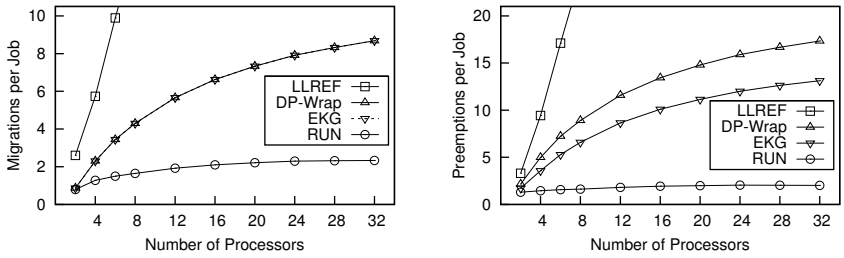


Figure 3.14: **Migrations/Preemptions vs Processors for Various Schedulers**
Migrations- and preemptions-per-job by LLREF, DP-Wrap, EKG, and RUN as number of processors m varies from 2 to 32, with full utilization and $n = 2m$ tasks. Note: DP-Wrap and EKG have the same migration curves.

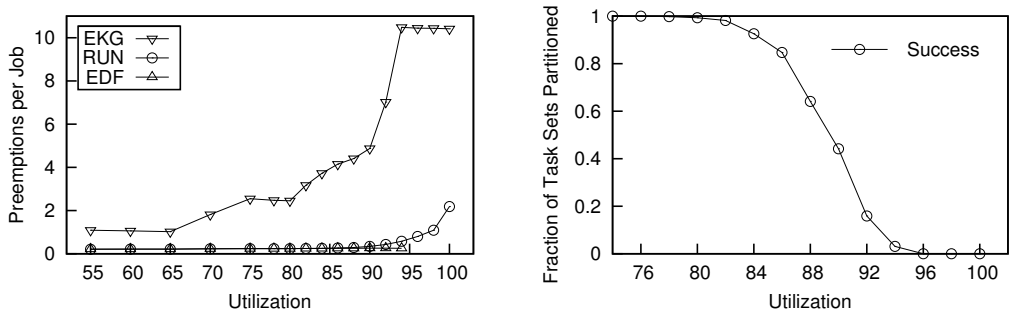


Figure 3.15: **Preemptions and Partitioning vs Utilization**
Preemptions per job for EKG, RUN, and Partitioned EDF as utilization varies from 55 to 100%, with 24 tasks on 16 processors; Partitioning success rate for best-fit bin packing under the same conditions.

parison in Figure 3.15's preemptions per job plot. Values for Partitioned EDF are only averaged over task sets where a successful partition occurs, and so stop at 94% utilization. The second plot shows the fraction of task sets that achieve successful partition onto m processors, and consequently, where RUN reduces to Partitioned EDF.

With its few migrations and preemptions at full utilization, its efficient scaling with increased task and processor counts, and its frequent reduction to Partitioned EDF on lower utilization task sets, RUN represents a substantial performance improvement in the field of optimal schedulers.

3.7 Related Work

We have already surveyed numerous optimal and other multiprocessor scheduling algorithms in Section 2.4, so we will note only briefly the relationship of these and other works with RUN. Subsequent to PFAIR [4], all previous optimal algorithms (*e.g.,* BF [47], LLREF [10], EKG [3], and DP-WRAP) used some DP-FAIR scheme to achieve optimality. These all relied on the enforcement of proportional fairness at all system deadlines. RUN greatly outperforms all these by relying instead on the less restrictive scheme of *limited proportional fairness*, which gives a server and its clients their collective proportional allotment of processor time between each of the server's deadlines.

Other recent works have used the *semi-partitioning* approach to limit migrations [2, 3, 17, 28, 36]. Under this scheme, some tasks are allocated off-line to fixed processors, much like in the partitioned approach, while other tasks are designated to migrate. These approaches present a trade-off between implementation overhead and achievable utilization; optimality may sometimes be achieved at the cost of higher migration overhead. RUN employs a different semi-partitioning approach, where tasks are partitioned among servers instead of processors. These servers will schedule their client tasks much as a dedicated processor would, but the server and its clients are not fixed on any physical processor. Both versions of semi-partitioning apply, in various ways, the simpler problem of uniprocessor scheduling to multiprocessor scheduling.

Other related work may be found on the topic servers. The concept of task servers has been extensively used to provide a mechanism to schedule soft real-time tasks [34], for which timing attributes like period or execution time are not known *a priori*. There are server mechanisms for uniprocessor systems which share some similarities with those presented herein [13, 44]. Other server mechanisms have been designed for multiprocessor systems [2,3,39]. Unlike these approaches, RUN treats each server as if it were a uniprocessor system, allowing us to hide the complexities of multiprocessor

scheduling within the server tree structure.

3.8 Conclusion

We have presented the optimal RUN multiprocessor real-time scheduling algorithm. RUN transforms the multiprocessor scheduling problem into an equivalent set of uniprocessor problems. Theory and simulation show that only a few preemption points per job are generated on average, allowing RUN to significantly outperform prior optimal algorithms. RUN reduces to the more efficient partitioned approach of Partitioned EDF whenever best-fit bin packing finds a proper partition, and scales well as the number of tasks and processors increases.

These results have both practical and theoretical implications. The overhead of RUN is low enough to justify implementation on actual multiprocessor architectures. At present, our approach only works for fixed-rate task sets with implicit deadlines. Theoretical challenges include extending the model to more general problem domains such as sporadic tasks with constrained deadlines. The use of uniprocessor scheduling to solve the multiprocessor problem raises interesting questions in the analysis of fault tolerance, energy consumption and adaptability. We believe that this novel approach to optimal scheduling introduces a fertile field of research to explore and further build upon.

Chapter 4

Conclusion

4.1 Future Work

While the work herein represents a considerable step forward on the problem of multiprocessor scheduling, it also leaves a number of questions unanswered, and suggests future avenues of research. One problem generalization not addressed herein is that of *uniform multiprocessors*, where processors run at different speeds, but treat all tasks uniformly. While optimal schedulers have been found which extend the LLREF algorithm [9,22], we believe that the general DP-Fair theory may be extended to cover this broader problem model. The difficulty in this is highlighted by this observation: since different processors will consume a task at different rates, the concept of *zero laxity* becomes ill-defined; a task with zero laxity on one processor may have considerable laxity on another, and Rule 1 of DP-Fair then becomes unclear. Nonetheless, some work has been done in this direction, and we believe that the simplicity and flexibility of DP-Fair makes an eventual solution promising.

Also of interest is extending RUN to more generalized problem domains, like tasks with sporadic arrivals. Because RUN's duality relies on full utilization at all times, and an inactive task implies an underutilized system, it is not immediately obvious how to extend duality and RUN when arrivals don't coincide with deadlines. We also believe there is potential for other non-deadline-partitioning algorithms like RUN. Work is being done on one such algorithm called QPS (**Q**uasi-**P**artitioned **S**cheduler) [37]. While QPS has a slightly higher overhead than RUN, it does not rely on duality, and appears to be adaptable to sporadic arrivals. More generally, RUN and QPS represent a new family of algorithms that are as yet not fully understood. We would like to develop a general theory for these algorithms in the same way that DP-Fair explained and unified previous deadline partitioning approaches.

Finally, perhaps the greatest challenge is to take these theoretical algorithms and implement them on a real CPU scheduler. In the past, partitioned schedulers

have been preferred because they are simple, and because task migration represents a significant overhead. However, these partitioned approaches have the potential for a substantial amount of unutilized processor capacity. With RUN's significant reduction in the number of context switches and migrations, it may be approaching the point where it would be a more efficient practical scheduler on some real systems. The implementation and successful comparison of RUN or some similar optimal algorithm on a real system would be the ultimate fruition of this research.

4.2 Contributions

Prior to the work in this dissertation, there was no general theory for the optimal multiprocessor scheduling of real-time tasks. The few optimal algorithms that existed were mostly unrelated, with disparate approaches and complex exposition and proofs. The first, PFAIR [4], maintained strict proportionate fairness, with work complete curves nearly matching fluid rate curves at all times, and thus incurred an excessive overhead of context switches and migrations. Subsequent algorithms reduced overhead by relying on *deadline partitioning*. The first of these was BF [47], which made substantial improvements over PFAIR, but still used the same discrete time model, and suffered the same resultant complexity. LLREF [10] employed the novel TL-Plane visualization and used a continuous time model, but incurred a considerable overhead from its reliance on frequent least laxity sorts and the resultant migrations and context switches. EKG [3] was the most efficient of these previous optimal algorithms, but was still complex in its presentation and proof of correctness.

In this dissertation we have presented a theoretical model to unify and clarify the previous deadline partitioning algorithms. Our DP-FAIR theory provides three simple, nearly obvious rules for multiprocessor scheduling. And while these rules are clearly *necessary* for successful scheduling, we provide a simple proof that, when com-

bined with the deadline partitioning approach, they are also *sufficient* to guarantee that a scheduling algorithm is optimal. From these three simple rules, we derive DP-Wrap, the simplest optimal scheduling algorithm to date. Using DP-Wrap as a framework, we can easily explain the behavior of previous optimal algorithms, see their common characteristics, and immediately infer their correctness. We thus have a simple framework for understanding all prior efforts in the field of optimal multiprocessor scheduling. The simplicity of this framework has also allowed us to extend it and the DP-Wrap algorithm to the more general domain of sporadic tasks and arbitrary deadlines. For this work we were awarded Best Paper at the 2010 Euromicro Conference on Real-Time Systems (ECRTS) [30].

Unfortunately, while deadline partitioning makes multiprocessor scheduling very simple, its requirement that all tasks match their fluid rate curves at all deadlines imposes a substantial overconstraint on the system, and results in potentially unnecessary context switches and migrations. However, prior to the work in this dissertation, no other approach was know. With the introduction of our RUN algorithm, we move beyond this limitation. By employing the novel mechanism of *scheduling duality*, we are able to ensure correct schedules while imposing much less restrictive overconstraints on our algorithm. The improvements are considerable: in simulated comparisons with previous algorithms, we observe, on average, 80% fewer context switches and migrations. And, more generally, while other optimal algorithms see their performance degrade as system size increases, RUN has provably low, constant upper bounds on overhead for nearly all task sets. Finally, for task sets where a partitioned approach will suffice, RUN reduces naturally to Partitioned EDF, making this one algorithm the ideal solution in all cases. RUN represents a new paradigm in optimal multiprocessor scheduling, and a significant improvement in performance. For this work we received the Best Paper award at the 2011 Real-Time Systems Symposium (RTSS) [43].

Appendix A

Notatation

m	number of processors
n	number of tasks
τ, J, σ	task, job, server
$\mathcal{T}$	set of tasks $\{\tau_1, \ldots, \tau_n\}$
$\tau = (p, c, \delta)$	task with period p, workload c, deadline δ
$\tau = (p, c)$	task with period p, workload c, implicit deadline
p_i	period (minimum interarrival time) of τ_i
c_i	workload of each job of τ_i
δ_i	time between arrival and deadline of τ_i
$a_{i,h}$	arrival time of h^{th} job of τ_i
ρ_i	$c_i / \min\{p_i, \delta_i\}$ (rate of τ_i)
$\rho(\mathcal{T})$	$\sum_i \rho_i$ (total rate of $\mathcal{T}$)
$S(\mathcal{T})$	$m - \rho(\mathcal{T})$ (total slack of $\mathcal{T}$)
$\mathbb{S}_j$	j^{th} time slice, time interval $= [t_{j-1}, t_j)$
t_j	j^{th} system deadline (end time of $\mathbb{S}_j$)
L_j	$t_j - t_{j-1}$ (length of $\mathbb{S}_j$)
$e_{i,t}$	local execution remaining for τ_i at t
$r_{i,t}$	$e_{i,t}/(t_j - t)$ (local remaining rate of τ_i at t)
E_t	total local execution remaining at t
R_t	total local rate remaining at t
$\kappa_{i,t}$	local capacity remaining for τ_i at t
$\alpha_{i,j}(t)$	time τ_i has been active in $\mathbb{S}_j$ as of t
$f_{i,j}(t)$	time τ_i has freed slack in $\mathbb{S}_j$ as of t
$w_{i,j}(t)$	work executed by τ_i in $\mathbb{S}_j$ as of t
$F_j(t)$	$\sum_i \rho_i f_{i,j}(t)$ (total slack freed in $\mathbb{S}_j$ as of t)
$I_j(t)$	total idle time in $\mathbb{S}_j$ as of t
$\tau{:}(\rho, A)$	fixed-rate task with rate ρ, and arrival times A
$A(\tau)$	set of arrival times of task τ
$J.a$, $J.c$, $J.d$	arrival time, execution time, deadline of J
$e_{J,t}$	work remaining for job J at time t
$e_{\sigma,t}$	budget of server σ at time t
Γ	set of servers
Σ	schedule of a set of tasks or servers
$\mathrm{cli}(\sigma)$	set of client servers (tasks) of σ
$\mathrm{ser}(\Gamma)$	server of the set of servers (tasks) Γ
τ^*; $\mathcal{D}(\tau)$	dual task of τ; DUAL operator
$\pi_A[\Gamma]$	partition of Γ by packing algorithm A
$\mathcal{P}(\sigma)$	server of σ given by PACK operator
$\mathcal{R} = \mathcal{D} \circ \mathcal{P}$	REDUCE operator

Table A.1: **Summary of Notation**
The first group of symbols defines a task set, the second group is used throughout Chapter 2, the third is used in Section 2.3, and the fourth in Chapter 3.

Appendix B

Bin Packing

The packing algorithms described in Section 3.4.2 are actually just traditional bin-packing algorithms. The bin-packing problem is described as follows:

Definition A.1 (Bin Packing)**.** Given a set of objects $S = \{\sigma_1, \ldots, \sigma_n\}$, each having some weight between 0 and 1, we wish to partition S into as few subsets as possible, subject to the restriction that no subset contains elements whose weights sum to more than one. That is, we wish to "pack" the elements of S into "bins", each with a maximum capacity of one, and use as few bins as possible. □

The decision version of this problem, namely, "Can we pack S into at most k bins?", is NP-Complete. We refer to *any* partition of S into subsets of weights at most one as a "packing". A *bin-packing algorithm* is simply an algorithm which partitions a weighted set into a proper packing.

The standard bin-packing algorithms we will consider all follow the same general outline: for each σ in S (taken one at a time), find some existing bin into which σ fits (that is, where adding σ to the bin will not cause that bin's total weight to exceed one), and place σ in it; if σ does not fit into any existing bin, create a new one, and place σ in that. The algorithms vary in (i) how to choose a bin for σ if it fits in multiple existing bins, and (ii) how to order the elements of S. The *first-fit* algorithm places σ

into the first examined bin into which it fits (bins are typically examined in the order in which they were created). The *best-fit* algorithm places σ into the bin with the least remaining room into which it still fits. The *worst-fit* algorithm places σ into the bin with the most remaining room, if it fits. S is ordered randomly unless otherwise specified. The suffix *decreasing* is appended to any of these (*e.g.,* "worst-fit decreasing") if S is ordered by decreasing weight.

As bin-packing is NP-Complete, none of these simple algorithms are optimal, that is, none are expected to find a packing with the fewest possible bins. However, they are fast, and do well enough for our purposes. In Section 3.4.2, we use bin-packing algorithms to assign client tasks (or servers) to aggregated servers, where the "weight" of a client task is its rate, and no "bin" (server) can contain clients of total rate exceeding one.

Appendix C

Minimizing Migrations is NP-Complete

We now show that finding a feasible schedule with the fewest possible migrations is NP-Complete via a reduction from Bin Packing. Thus, although we have searched for heuristics to reduce the number of migrations and preemptions in a feasible scheduling, it is unlikely that we could find an algorithm that produced a schedule with the *fewest* migrations or preemptions.

Bin Packing: An instance $(s_1, \ldots, s_N, M)$ of the bin packing decision problem (where $0 < s_i \leqslant 1$ and $M \in \mathbb{N}$) asks, "Can a set of N items with sizes $s_1, \ldots, s_N$ be fit into M bins, where each bin can hold any subset of items whose total size is no more than 1?"

Multiprocessor Real-Time Periodic Task Scheduling: An instance $(p_1, \ldots, p_n, c_1, \ldots, c_n, m, t, k)$ of the MRTPTS decision problem (where $0 < c_i \leqslant p_i, \quad t > 0$, $m, k \in \mathbb{N}$, and $\sum c_i/p_i \leqslant m$) asks, "Given n periodic tasks $\tau_i = (p_i, c_i)$ on m processors, the constraints guarantee that a feasible scheduling exists for these tasks, given that migrations are allowed. Does there exist a feasible scheduling in which at most k migrations occur by time t?"

Bin packing is known to be NP-Complete. We prove that MRTPTS is NP-Hard via a reduction from Bin Packing.

Consider the following transform $T : BP \rightarrow MRTPTS$ defined by

$$T(s_1, \ldots, s_N, M) = (p_1, \ldots, p_n, c_1, \ldots, c_n, m, t, k) \ ,$$

where

$$\begin{aligned} n &= N \\ p_1 = \cdots = p_n &= 1 \\ c_i &= s_i \text{ , for } i = 1, \ldots, n \\ m &= M \\ t &= 1 \\ k &= 0 \end{aligned}$$

Since we are allowing no migrations ($k = 0$), a feasible scheduling would be a partitioned scheduling, with jobs finishing by their deadlines at time 1 if and only if the jobs on each processor have workloads which sum to no more than one. Thus a feasible scheduling corresponds precisely with a successful bin packing, and our reduction transformation preserves yes/no instances. MRTPTS is therefore NP-Hard.

A feasible schedule with at most k migrations for a "yes" instance of MRTPTS would be a verifiable polynomial certificate, so MRTPTS is in NP, and thus is NP-Complete.

References

Bibliography

[1] Björn Andersson and Konstantinos Bletsas. Sporadic multiprocessor scheduling with few preemptions. *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 243–252, 2008.

[2] Björn Andersson, Konstantinos Bletsas, and Sanjoy K. Baruah. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 385–394, 2008.

[3] Björn Andersson and Eduardo Tovar. Multiprocessor scheduling with few preemptions. In *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 322–334, 2006.

[4] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[5] S. K. Baruah and Joel Goossens. Scheduling real-time tasks: Algorithms and complexity. In Joseph Y-T Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman Hall/CRC Press, 2004.

[6] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 182–190, 1990.

[7] Sanjoy K. Baruah and John Carpenter. Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. *Journal of Embedded Computing*, 1(2):169–178, 2004.

[8] 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.

[9] Shih-Ying Chen and Chih-Wen Hsueh. Optimal dynamic-priority real-time scheduling algorithms for uniform multiprocessors. *IEEE Real-Time Systems Symposium (RTSS)*, pages 147–156, 2008.

[10] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 101–110, 2006.

[11] 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.

[12] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. *Approximation algorithms for NP-hard problems*, chapter Approximation algorithms for bin packing: a survey, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[13] Z. Deng, J. W.-S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 191–199, 1997.

[14] Michael Dertouzos. Control robotics : the procedural control of physical processors. *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[15] Michael Dertouzos and Aloysius K. Mok. Multiprocessor scheduling in a hard real-time environment. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, 1989.

[16] S.K. Dhall and C.L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[17] Arvind Easwaran, Insik Shin, and Insup Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, 43(1):25–59, 2009.

[18] Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–11, 2010.

[19] Nathan Fisher, Joël Goossens, and Sanjoy Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Systems*, 45(1-2):26–71, June 2010.

[20] Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. New abstraction for optimal real-time scheduling on multiprocessors. *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 357–364, 2008.

[21] Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 13–22, 2008.

[22] S. Funk and V. Nadadur. Lre-tl: An optimal multiprocessor algorithm for sporadic task sets. *Conference on Real-Time and Network Systems (RTNS)*, pages 159–168, 2009.

[23] Shelby Funk, Greg Levin, Caitlin Sadowski, Ian Pye, and Scott Brandt. Dp-fair:

a unifying theory for optimal hard real-time multiprocessor scheduling. *Real-Time Systems*, 47(5):389–429, Sep 2011.

[24] Dorit S. Hochbaum, editor. *Approximation algorithms for NP-hard problems.* PWS Publishing Co., Boston, MA, USA, 1997.

[25] Kwang Soo Hong and Joseph Y.-T. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41:1326–1331, 1992.

[26] Shinpei Kato and Nobuyuki Yamasaki. Real-time scheduling with task splitting on multiprocessors. *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 441–450, 2007.

[27] Shinpei Kato and Nobuyuki Yamasaki. Portioned edf-based scheduling on multiprocessors. *ACM International Conference on Embedded Software (EMSOFT)*, pages 139–148, 2008.

[28] Shinpei Kato, Nobuyuki Yamasaki, and Yutaka Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 249–258, 2009.

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

[30] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. Dp-fair: a simple model for understanding optimal multiprocessor scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–13, 2010.

[31] Caixue Lin and Scott A. Brandt. Improving soft real-time performance through better slack management. *IEEE Real-Time Systems Symposium (RTSS)*, pages 3–14, 2005.

[32] Caixue Lin, Tim Kaldewey, Anna Povzner, and Scott A. Brandt. Diverse soft real-time processing in an integrated system. *IEEE Real-Time Systems Symposium (RTSS)*, pages 369–378, 2006.

[33] 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.

[34] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

[35] Jose Maria López, M. Garcia, José Luis Diaz, and Daniel F. Garcia. Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems. *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 25–34, 2000.

[36] E. Massa and G. Lima. A bandwidth reservation strategy for multiprocessor real-time scheduling. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 175–183, 2010.

[37] E. Massa, G. Lima, P. Regnier, G. Levin, and S. Brandt. Quasi-partition scheduling: Achieving optimality in sporadic real-time multiprocessor systems. *To appear*, 2013.

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

[39] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 294–303, 1999.

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

[41] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Reducing preemptions and migrations in real-time multiprocessor scheduling algorithms by releasing the fairness. In *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 15–24, Aug 2011.

[42] S.H. Oh and S.M. Yang. A modified least-laxity-first scheduling algorithm for real-time tasks. *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 31–36, 1998.

[43] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, 29 2011-dec. 2 2011.

[44] M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, 1996.

[45] Anand Srinivasan, Philip Holman, James H. Anderson, and Sanjoy K. Baruah. The case for fair multiprocessor scheduling. *International Symposium on Parallel and Distributed Processing (IPDPS)*, April 2003.

[46] Roger Stafford. Random vectors with fixed sum. http://www.mathworks.com/matlabcentral/fileexchange/9700, Jan. 2006.

[47] D. Zhu, D. Mossé, and R. Melhem. Multiple-resource periodic scheduling problem: how much fairness is necessary? In *IEEE Real-Time Systems Symposium (RTSS)*, pages 142–151, 2003.

[48] Dakai Zhu, Xuan Qi, Daniel Mossé, and Rami Melhem. An optimal boundary fair scheduling algorithm for multiprocessor real-time systems. *Journal of Parallel and Distributed Computing*, 71(10):1411–1425, 2011.