ADAPTIVE MULTIPROCESSOR REAL-TIME SYSTEMS

Aaron D. Block

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2008

ABSTRACT

AARON D. BLOCK: Adaptive Multiprocessor Real-Time Systems
(Under the direction of James H. Anderson)

Over the past few years, as multicore technology has become cost-effective, multiprocessor systems have become increasingly prevalent. The growing availability of such systems has spurred the development of computationally-intensive applications for which single-processor designs are insufficient. Many such applications have timing constraints; such timing constraints are often not static, but may change in response to both external and internal stimuli. Examples of such applications include tracking systems and many multimedia applications. Motivated by these observations, this dissertation proposes several different adaptive scheduling algorithms that are capable of guaranteeing flexible timing constraints on multiprocessor platforms.

Under traditional task models (e.g., periodic, sporadic, etc.), the schedulability of a system is based on each task's *worst-case execution time* (WCET), which defines the maximum amount of time that each of its jobs can execute. The disadvantage of using WCETs is that systems may be deemed unschedulable even if they would function correctly most of the time when deployed. Adaptive real-time scheduling algorithms allow the timing constraints of applications to be adjusted based upon runtime conditions, instead of always using fixed timing constraints based upon WCETs. While there is a substantial body of prior work on scheduling applications with static timing constraints on multiprocessor systems, prior to this dissertation, no adaptive multiprocessor scheduling approach existed that is capable of ensuring bounded "error" (where error is measured by comparison to an ideal allocation).

In this dissertation, this limitation is addressed by proposing five different multiprocessor scheduling algorithms that allow a task's timing constraints to change at runtime. The five proposed adaptive algorithms are based on different non-adaptive multiprocessor scheduling algorithms that place different restrictions on task migrations and preemptions. The relative advantages of these algorithms are compared by simulating both the Whisper human tracking system and the Virtual Exposure Camera (VEC), both of which were developed at The University of North Carolina at Chapel Hill. In addition, a feedback-based adaptive framework is proposed that not only allows timing constraints to adapt at runtime, but also detects which adaptions are needed. An implementation of this adaptive framework on a real-time multiprocessor testbed is discussed and its performance is evaluated by using the core operations of both Whisper and VEC. From this dissertation, it can be concluded that feedback and optimization techniques can be used to determine at runtime which adaptions are needed. Moreover, the accuracy of an adaptive algorithm can be improved by allowing more frequent task migrations and preemptions; however, this accuracy comes at the expense of higher migration and preemption costs, which impacts average-case performance. Thus, there is a tradeoff between accuracy and average-case performance that depends on the frequency of task migrations/preemptions and their cost.

To my wife, parents, and brother,

without whom I would only be three sevenths of the man I am today.

ACKNOWLEDGMENTS

No document this large can be written without help from many people. I would first like to thank my advisor, James Anderson, who taught me how to research and write over six dyslexia-filled years. I cannot imagine a better advisor.

I would also like to than the rest of my committee: Tarek Abdelzaher, Sanjoy Baruah, Gary Bishop, Kevin Jeffay, and Stephen Quint. I am lucky to have such a qualified committee, and I deeply appreciate all of the help and feedback they have provided me over the years.

I am indebted to the many collegues with whom I have published over the years: Gary Bishop, Stephen Quint, Uma Devi, Björn Brandenburg, John Calandrino, Hennadiy Leontyev, Anand Srinivasan, Warren Davis, Russell Hamm, Sarah Knoop, and Peter Schwarz. I will always remember fondly the many crushed-red-pepper-pizza filled nights that I spent with you. Also, I owe many thanks to my other real-time colleagues: Shelby Funk, Phil Holman, and Nathan Fisher. I never wrote a paper with you, but I always wished I had.

So many of my friends (with whom I never published) helped to contribute to this dissertation and my life in general that it is hard to name them all, but here it goes (I apologize to any friends I may have omitted): Alex Higbee, Joel Heires, Craig Morris, Mike West, Austin Parker, the entire Haverford fencing team, Joanna Grayer, Julia Diepold, Jack and Shilpa McManus, Peter and Lori Adler, Galvin Chow, Chas Budnick, Harrison Breuer, Carl Knutson, Matthew Gjenvic, Jesse Milnes, Luv Kohli, Brian Eastwood, Chris VanderKnyff, Jeremy Wendt, Russell Gayle, Stephen Oliver, Sasa Junuzovic, Sandra Neely, Keith Lee, Jeff Terrell, and Avneesh Sud. Last, but not least, I want to thank my groomsmen, Adam Ruder, John Stevens "Terry" McMahon III, and Eric Bennett. You guys have been there for me in the best of times and the worst of times. I owe you more than I can imagine.

I would also like to thank my in-laws. Terri, I cannot conceptualize a more fun sister. Also, you are probably the best person I have ever known at keeping a surprise a secret. Brian and Brenda, you guys have been there for me nearly every Friday night for the past six years. Thank you so much for always being there for me to kvetch about my week.

Next I would like to thank my brother. Stefan, you are the Platonic form of a brother to which all other brothers aspire in an attempt to reach that ideal. I want to apologize for any references to younger brothers as the PEDF scheduling algorithm; I felt they were necessary for the literary quality of the document. (An alternative joke would be “Along side this processor there is another, there are places where you can adapt.”)

Penultimately, I would like to thank my parents, who (aside from teaching me transition words) taught me everything I know. Thank you so much for helping me grow from a nerdy little boy into a relatively less nerdy man. It is possible to mathematically prove that you are the best parents I could have.

Finally, I want to thank my wife. Nicki, you are the most wonderful wife I could have asked for. So, to quote Jerry Maguire, as I did in my wedding speech,“Show me the money.” Wait, that doesn’t sound right. What I want to say is that you enrich my life every day and always bring a smile to my face—I love you.

That’s it. Enjoy the dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AIS	Adaptable Intra-Sporadic Task Model
AGIS	Adaptable Generalized Intra-Sporadic Task Model
AGEDF	Adaptive GEDF
AAOE	Average Absolute Overall Error
AROE	Average Relative Overall Error
EDF	Earliest-Deadline-First
EPDF	Earliest-Pseudo-Deadline-First
EEVDF	Earliest-Eligible-Virtual-Deadline-First
FF	Fairness Factor
FCS	Feedback-Control Real-Time Scheduling
FMLP	Flexible Multiprocessor Locking Protocol
GEDF	Global EDF
IS	Intra-Sporadic
MAOE	Maximal Absolute Overall Error
MROE	Maximal Relative Overall Error
NP-GEDF	Non-Preemptable GEDF
NP-PEDF	Non-Preemptable PEDF
PEDF	Partitioned EDF
RBED	Rate-Based Earliest-Deadline
RTOS	Real-Time Operating System
RAD	Reasonable Allocation Decreasing
UA	Average Under Allocation

CHAPTER 1

INTRODUCTION

The goal of this dissertation is to extend research on multiprocessor real-time systems in order to enable such systems to *adapt* tasks' processor shares—a process called *reweighting*—in response to both external and internal stimuli. The particular focus of this work is on adaptive systems that are deployed in environments in which tasks may *frequently* require *significant* share changes. Such environments are commonplace in computationally-intensive multimedia applications. Prior to the research in this dissertation, no multiprocessor reweighting algorithms had been proposed that could change task shares with bounded overhead. In this dissertation, we extend prior work on uniprocessor and multiprocessor systems to construct reweighting algorithms with minimal overhead for several different types of multiprocessor systems. Furthermore, we examine how *feedback* and *optimization* techniques can be use to determine, at run time, *which* reweighting events are needed. Finally, we evaluate the proposed adaptive scheduling algorithms by using two multimedia applications developed at The University of North Carolina at Chapel Hill: the *Whisper* human tracking system (Vallidis, 2002) and the *Virtual Exposure Camera* (VEC) night vision system (Bennett and McMillan, 2005).

To motivate the need for adaptive real-time scheduling, we begin this chapter with a brief overview of Whisper and VEC. Next, we discuss the core real-time concepts that are relevant to this dissertation. We then review prior work on adaptive real-time systems and state the thesis of this dissertation. We conclude this chapter by summarizing this dissertation's contributions and providing an outline for the remainder of the dissertation.

1.1 Applications

Brief descriptions of Whisper and VEC are provided below; more detailed descriptions will be given later in Chapter 7, where our experimental results are given. Before discussing Whisper and VEC, it is important to point out that, since each is a multimedia application, both must ensure certain timing constraints to provide an acceptable user experience and thus are examples of *real-time application*. One of the questions we will answer in this dissertation is how the timing constraints of real-time applications can be changed at run time while still providing an acceptable quality-of-service (QoS).

1.1.1 Whisper

As mentioned above, Whisper performs full-body tracking in virtual environments (Vallidis, 2002). Whisper tracks users via an array of wall- and ceiling-mounted microphones that detect signals (i.e., white noise) emitted from speakers attached to each user's hands, feet, and head. Specifically, by calculating the time-shift in the signal for each microphone-speaker pair, Whisper is able to triangulate each speaker's position. The amount of time required to calculate the distance between a microphone-speaker pair is indirectly related to the signal-to-noise ratio. As the distance between a microphone-speaker pair increases, the signal-to-noise ratio decreases, which increases the amount of time required to calculate this distance. Also, other factors, like ambient noise, further degrade the signal-to-noise ratio causing total computation time to increase. Because the computational cost associated with calculating the distance between a microphone speaker-pair can change at run time, the tasks comprising Whisper must be scheduled using algorithms that either allow task parameters to adapt or allow task shares to be defined based on worst-case scenarios. Unfortunately, provisioning the system based on worst-case scenarios may not be a viable option because there exist scenarios for which no reasonable computational platform can correctly track a user (e.g., a room with a 100dB of ambient noise). While using adaptive techniques reduces the resources required to correctly track a user (relative to over-provisioning), the workload is still intensive enough to necessitate a multiprocessor system.

1.1.2 Virtual Exposure Camera

The second application considered in this dissertation is the VEC video-enhancement system (Bennett and McMillan, 2005).[1] VEC is capable of improving the quality of an under-exposed video feed so that objects that are indistinguishable from the background become clear and in full color. In VEC, darker objects require more computation to correct. Thus, as dark objects move in the video, the processor shares of the tasks assigned to process different areas of the video will change. Like all multimedia applications, to create an acceptable user experience, VEC must update the corrected image at a regular rate. VEC will eventually be deployed in a full-color night vision system, so tasks will need to change shares as fast as a user's head can turn. In the planned configuration, a multicore platform consisting of approximately ten processing cores will be used.

1.2 Real-Time Systems

The distinguishing characteristic of a real-time system is the need to satisfy timing constraints. The timing constraints of recurrent applications (e.g., Whisper and VEC) can be represented using the *sporadic task model*. In this model, each piece of sequential recurrent code is called a *task*. Each invocation of such a task is called a *job*. We denote the i^{th} task of a set of tasks T as T_i (where tasks are ordered by some arbitrary method), and denote the j^{th} job of the task T_i as T_i^j (where jobs are ordered by the sequence in which they are invoked). Associated with a sporadic task is a *worst-case execution time* (WCET), denoted $\mathsf{e}(T_i)$, and a *period*, denoted $\mathsf{p}(T_i)$. The WCET denotes the maximum amount of time any job of the task requires; the period denotes the minimum separation between consecutive job invocations *and* defines a relative "deadline" for each job. The time at which a job is invoked is called its *release time*, denoted $\mathsf{r}(T_i^j)$, and the (absolute) time by which a job must complete is called its *deadline*, denoted $\mathsf{d}(T_i^j)$. The *weight* of a task T_i, denoted $\mathsf{wt}(T_i)$, is the fraction of a process it requires to be correctly scheduled and is defined as $\mathsf{e}(T_i)/\mathsf{p}(T_i)$. For shorthand, we will use $T_i{:}(e, p)$ to denote a task T_i with a WCET of e and a period of p.

[1]In prior work (Block and Anderson, 2006; Block et al., 2008a; Block et al., 2008b), we referred to VEC as ASTA, which stands for *Adaptive Saptio-Temporal Accumulation Filter*; however, VEC is more technically

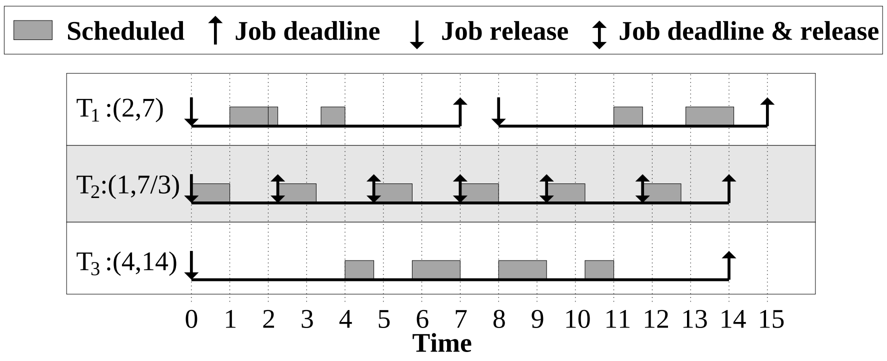


Figure 1.1: A one-processor example with three sporadic tasks.

The release and deadline of the job T_i^j of a sporadic task T_i can be specified as

$$\begin{aligned}\mathsf{r}(T_i^1) &= \theta(T_i^1)\\ \mathsf{r}(T_i^j) &= \mathsf{d}(T_i^{j-1}) + \theta(T_i^j),\ j > 1\\ \mathsf{d}(T_i^j) &= \mathsf{r}(T_i^j) + \mathsf{p}(T_i),\ j \geq 1\end{aligned}$$

where $\theta(T_i^j) \geq 0$ for $j \geq 1$. $\theta(T_i^j)$ denotes the sporadic separation between job releases. If $\theta(T_i^{j+1}) = 0$, then T_i^{j+1} is released at T_i^j's deadline.

Example (Figure 1.1). Consider the example in Figure 1.1, which depicts a one-processor system with three tasks: T_1:$(2, 7)$, which has a sporadic separation of one time unit between T_1^1 and T_1^2; T_2:$(1, 7/3)$; and T_3:$(4, 14)$. The grey boxes denote the time at which the associated job is scheduled. Down arrows denote a job release. Up arrows denote a job deadline. Up-and-down arrows denote that a job's deadline and its successor's release occur at the same time. Similar notation will be used in later figures. ☐

The *actual execution time of job* T_j^j, denoted $\mathsf{Ae}(T_i^j)$, is the amount of time for which T_i^j is scheduled; this value is upper-bounded by $\mathsf{e}(T_i)$. Depending on the scenario, this value may or may not be known before the job finishes execution. To facilitate further discussion, a few additional terms need to be defined.

Definition 1.1 (Window, Active, and Inactive). If T_i^j is a job in the task system T, then the *window* of T_i^j defined as the range $[\mathsf{r}(T_i^j), \mathsf{d}(T_i^j))$. Furthermore, the job T_i^j is *active*

correct.

at time t iff t is in T_i^j's window (i.e., $t \in [\mathsf{r}(T_i^j), \mathsf{d}(T_i^j)))$, and *inactive* otherwise. We use $\mathsf{ACT}(t)$ to denote the set of active jobs at time t.

For example, in Figure 1.1, T_1^1 is active over the range [1, 7) and is inactive at every other time.

Definition 1.2 (**Completed).** If $\mathcal{S}$ is a schedule of the task system T, then a job $T_i^j \in T$ is said to have *completed by time t in $\mathcal{S}$* iff T_i^j has executed for $\mathsf{Ae}(T_i)$ time units by t in $\mathcal{S}$. Similarly, a task T_i is said to be *complete at time t* iff at time t every job of T_i that was released by t has completed.

For example, in Figure 1.1, T_1^1 is complete at and after time 4. Also, at time 10/3, T_2 is complete since both T_2^1 and T_2^2 are complete by time 10/3.

Definition 1.3 (**Pending and Ready).** For an arbitrary scheduling algorithm $\mathcal{A}$, if $\mathcal{S}$ is a schedule of the task system T under $\mathcal{A}$, then a job T_i^j is said to be *pending at time t in $\mathcal{S}$* if $\mathsf{r}(T_i^j) \leq t$ and T_i^j is incomplete at t in $\mathcal{S}$. Note that a job can be both pending and inactive, if it misses its deadline. A pending job T_i^j is said to be *ready at time t in $\mathcal{S}$* if all prior jobs of task T_i have completed by t. A job T_i^j can be pending but not ready if T_i^{j-1} is incomplete at $\mathsf{r}(T_i^j)$. (Such a scenario may occur in some multiprocessor algorithms.)

For example, in Figure 1.1, T_1^1 is pending until time 4, and T_3^1 is pending until time 11.

Let $\mathcal{A}$ be an arbitrary scheduling algorithm, τ be an arbitrary task system, and $\mathcal{S}$ denote schedule of τ generated by $\mathcal{A}$. Then, we use $\mathsf{A}(\mathcal{S}, T_i^j, t_1, t_2)$ denote the total time allocated to T_i^j in $\mathcal{S}$ in $[t_1, t_2)$. Similarly, we use $\mathsf{A}(\mathcal{S}, T_i, t_1, t_2)$ and $\mathsf{A}(\mathcal{S}, \tau, t_1, t_2)$, respectively, to denote the total time allocated to all jobs of T_i in $\mathcal{S}$ and all tasks of τ in $\mathcal{S}$, over the interval $[t_1, t_2)$. We say that the value of $\mathsf{A}(\mathcal{S}, T_i^j, 0, t)$ is the amount that T_i^j has *executed by t*. For example in Figure 1.1, $\mathsf{A}(\mathcal{S}, T_1^1, 0, 2) = 1$ and $\mathsf{A}(\mathcal{S}, T_1^1, 1, 4) = 2$.

Depending on the consequences of missing a deadline, real-time systems can be classified as either "hard" or "soft." A system is a *hard real-time* (HRT) system if missing *any deadline* implies that the system fails. In contrast, in a *soft* real-time (SRT) system, deadlines may be missed. Examples of HRT systems include avionics and automotive applications. Examples of SRT systems include multimedia and virtual-reality applications. While SRT systems may miss an occasional deadline, it is still possible for such systems to "fail;" however, there is no single notion of a "correct" SRT system. Some possible notions of SRT correctness include: bounded *deadline tardiness* (i.e., all jobs complete within some bound of their deadline) (Devi and Anderson, 2008); a specified percentage of deadlines must be met (Lu et al., 2002); and m out of every k consecutive jobs of each task complete before their deadline (Hamadoui and Ramanathan, 1995). In this dissertation, we are primarily concerned with HRT systems and SRT systems with bounded deadline tardiness. For HRT systems, we say that a given task set is *schedulable* if it is possible to guarantee that no single task will miss its deadline; otherwise we say that it is *unschedulable*. Similarly, for SRT systems, we say that a given task set is schedulable if it is possible to guarantee that every task has bounded deadline tardiness, and is unschedulable otherwise. Moreover, for many types of system, we can determine if a given task set is schedulable using a *scheduability test*, i.e., a set of conditions that, when satisfied by the task set, imply that it is schedulable.

1.2.1 Uniprocessor Systems

The weight of a task can be used to define an *ideal schedule*, in which, at each instant, each task is allocated a fraction of a processor equal to its weight. While the ideal schedule represents the most equitable allocation of resources possible, it is infeasible to implement since it requires tasks to be preempted and swapped at arbitrarily small intervals. For a uniprocessor system, a more realistic scheduling algorithm is the *earliest-deadline-first* (EDF) algorithm, which schedules jobs based on their deadlines, with earlier deadlines having higher priority. On a uniprocessor system, EDF can guarantee that every job completes before its deadline if the total weight of all tasks is at most one, the total available utilization.

Example (Figure 1.2). Figure 1.2 depicts a one-processor example of an ideal and EDF schedule of a system with four tasks: T_1:$(1, 2)$; T_2:$(2, 8)$; T_3:$(1, 8)$; and T_4:$(1, 8)$ T_1. The numbers in each box denote the fraction of the processor consumed by the associated task. Insets (a) through (c) depict, respectively, an ideal schedule, an EDF schedule, and the actual and ideal allocations for T_1. □

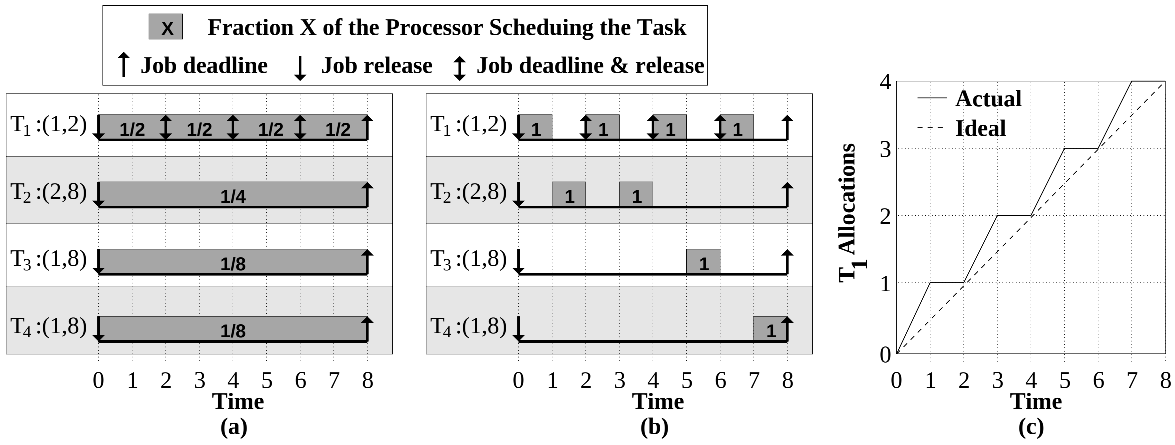


Figure 1.2: A one-processor example of an **(a)** ideal and **(b)** EDF schedule, and **(c)** T_1's allocation in both.

1.2.2 Multiprocessor Partitioned Scheduling

Most multiprocessor scheduling algorithms can be classified as either *partitioned* or *global*. Under partitioned algorithms, each task is permanently assigned to a specific processor and each processor independently schedules its assigned tasks using a uniprocessor scheduling algorithm. Alternatively, under global algorithms, a task may migrate among processors. The advantage of partitioned approaches over global approaches is that they have lower *migration/preemption costs*. This is because, under partitioned approaches, tasks maintain cache affinity for longer durations of time due to fewer task migrations than in global approaches. The disadvantage of partitioned approaches is that such systems have inferior scheduability conditions when compared to global approaches (as we shall see).

This section discusses the specifics of two partitioned scheduling algorithms: *preemptive* and *nonpreemptive partitioned* EDF (abbreviated as PEDF and NP-PEDF, respectively). Under PEDF and NP-PEDF each processor is scheduled independently using the EDF scheduling algorithm. The difference between them is that, under PEDF, a job can be preempted, and under NP-PEDF, a job cannot be preempted.

Example (Figure 1.3). Consider the example in Figure 1.3, which depicts a two-processor system with six tasks.: T_1:$(3, 12)$; T_2:$(1, 6)$; T_3:$(3, 6)$; T_4:$(1, 4)$; T_5:$(1, 4)$; and T_6:$(5, 12)$. Tasks T_1, T_3, and T_5 are assigned to Processor 1, and tasks T_2, T_4, and T_6 are assigned to Processor 2. Inset (a) depicts a PEDF schedule, and (b) depicts an NP-PEDF schedule. □

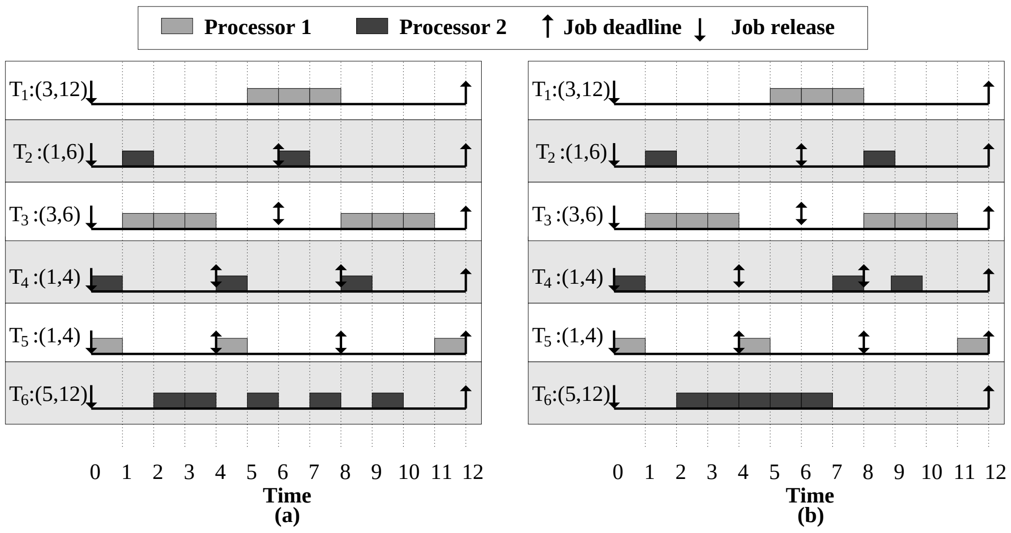


Figure 1.3: Two-processor **(a)** PEDF and **(b)** NP-PEDF schedules.

Notice that, since T_5 is assigned to Processor 1, Processor 2 is idle over the range [10, 11) even though T_5 has work to be completed. Also note that the difference between Figure 1.3(a) and Figure 1.3(b) is that in Figure 1.3(b) once a job begins executing it continues to do so until it completes. This behavior is illustrated by T_6^1, which has a contiguous execution in Figure 1.3(b) but not in Figure 1.3(a).

Before continuing, there is one subtlety that must be discussed. Throughout this dissertation, whenever we discuss partitioned scheduling algorithms, we will make a distinction between the *guaranteed weight* of a task and the *desired weight* of a task (given by $\mathsf{e}(T_i)/\mathsf{p}(T_i)$). This distinction is important because under partitioned algorithms, it is possible for a processor to be over-allocated, i.e., the processor is assigned tasks with a total weight exceeding one. When a processor is over-allocated, there are two options: reject one or more tasks; or reduce the shares of the tasks on that processor. For example, in a two-processor system with three tasks each of weight 2/3 (depicted in Figure 1.4), either one of the three tasks must be rejected or the shares of the tasks on the over-allocated processor must be reduced. Both of these options guarantee that the shares of tasks do not overutilize either of the processors even though the weights do. While both variants have their relative advantages, for adaptive systems, the latter option is likely a better option. (A thorough discussion of this is issue given in Section 4.2.) It is important to note that for the majority of the algorithms considered in

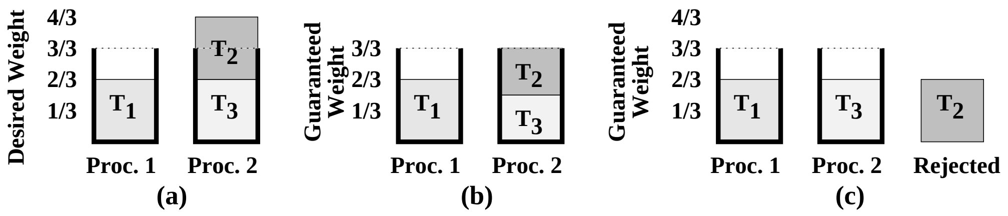


Figure 1.4: **(a)** A two-processor system with three tasks each with a desired weight of 2/3. **(b)** The guaranteed weights of tasks in (a) when no task is rejected. **(c)** The guaranteed weights of tasks in (a) when T_2 is rejected.

this dissertation, a task's guaranteed weight equals its desired weight. Thus, for brevity, we will only use the terms "desired weight" and "guaranteed weight" when discussing algorithms where the two may differ.

1.2.3 Restricted Global Multiprocessor Scheduling

In global scheduling algorithms, tasks are scheduled from a single priority queue and may migrate among processors. Global algorithms can be further classified as either *restricted* and *unrestricted*. A scheduling algorithm is considered to be restricted if the scheduling priority of each job (for any given schedule) does not change once it has been released. A scheduling algorithm is considered to be unrestricted if there exists a schedule in which some job changes its priority after it is released. In this section, we discuss two restricted global scheduling algorithms, *preemptive global*-EDF (GEDF) and *non-preemptive global*-EDF (NP-GEDF); unrestricted algorithms are considered in Section 1.2.4. Under both GEDF and NP-GEDF, tasks are scheduled from a single priority queue on an EDF basis. As with partitioned algorithms, the only difference between GEDF and NP-GEDF is that jobs can be preempted in GEDF and cannot be preempted in NP-GEDF.

Example (Figure 1.5). Consider the example in Figure 1.5, which pertains to a two-processor system with five tasks: T_1:$(2,7)$; T_2:$(1,7)$; T_3:$(1,7)$; T_4:$(3,7)$; and T_5:$(3,7)$. Inset (a) depicts a GEDF schedule and inset (b) depicts an NP-GEDF schedule. In (a), T_5^1 misses a deadline by one time unit at time 7, and T_5^2 misses a deadline by one time unit at time 14. In (b), T_5^2 misses a deadline by two time unit at time 14. □

Since tasks are not assigned to processors for prolonged periods of time in these algorithms,

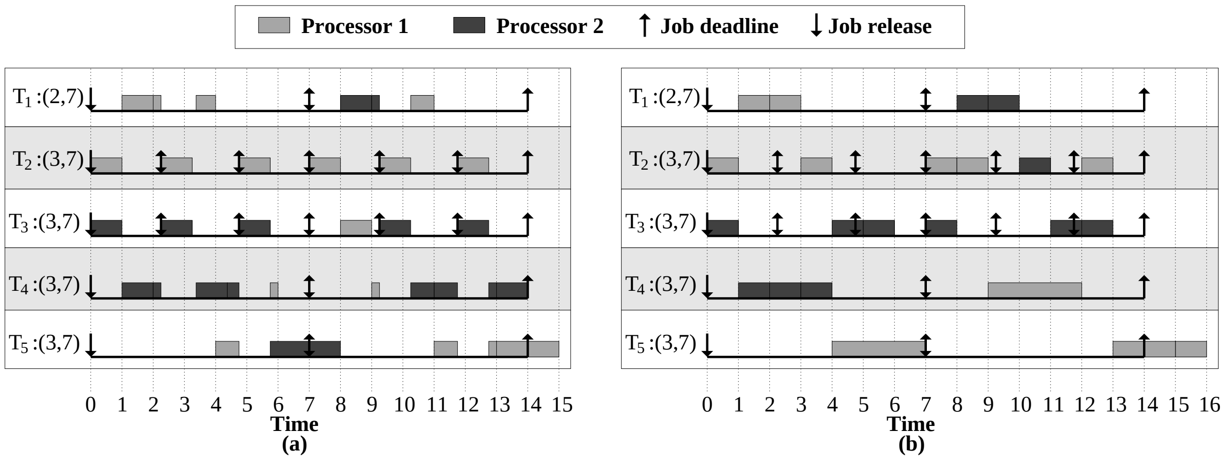


Figure 1.5: Two-processor **(a)** GEDF and **(b)** NP-GEDF schedules.

it is not possible for a processor to be over-allocated in the long term, provided that total utilization is at most m, the number of processors. However, short-term over-allocations that cause deadline misses are possible. Nonetheless, as shown by Devi and Anderson (Devi and Anderson, 2008), such misses are only by bounded amounts. The amount of time by which a job misses its deadline is called its *tardiness*. For example, in Figure 1.5(a), T_5^1 misses a deadline at time 7, and in both Figure 1.5(a) and Figure 1.5(b), T_5^2 misses a deadline at time 14. As shown in (Devi and Anderson, 2008), under GEDF, the maximal tardiness of any job of task a T_i is bounded by a formula given shortly; however, before presenting their equation, we briefly introduce a few needed terms. Let $\mathsf{e_{min}}$ denote the minimum execution time of any task in the system τ. Define $\mathsf{WT}(\tau)$ as

$$\mathsf{WT}(\tau) = \sum_{T_i \in \tau} \mathsf{wt}(T_i).$$

Additionally, let $\mathsf{max_e}(k)$ and $\mathsf{max_{wt}}(k)$ denote, respectively, the k^{th} largest execution time and weight of any task. Finally, let the value Γ denote

$$\Gamma = \begin{cases} \mathsf{WT}(\tau) - 1, & \mathsf{WT}(\tau) \text{ is integral} \\ \lfloor \mathsf{WT}(\tau) \rfloor, & \text{otherwise.} \end{cases}$$

Using these terms, the maximal tardiness (as given in (Devi and Anderson, 2008)) for any job of a task T_i is given as

$$\frac{\sum_{k=1}^{\Gamma} \mathsf{max_e}(k) - \mathsf{e_{min}}}{m - \sum_{k=1}^{\Gamma-1} \mathsf{max_{wt}}(k)} + \mathsf{e}(T_i), \tag{1.1}$$

provided $\mathsf{WT}(\tau) \leq m$, where m is the number of processors. Since tasks cannot be preempted in NP-GEDF, its tardiness bound is slightly larger than (1.1). (See (Devi and Anderson, 2008) for details.)

1.2.4 Unrestricted Global Multiprocessor Scheduling

The final multiprocessor scheduling algorithm we consider in this dissertation is the unrestricted global Pfair algorithm PD^2 (Srinivasan and Anderson, 2005). PD^2 schedules a task by breaking it into a sequence in which of *subtasks*, each of which represents one time unit of execution. Each such time unit is called a *quantum*. The j^{th} subtask of the task T_i is denoted as $T_i^{[j]}$, where subtasks are ordered by the sequence in which they are invoked. Associated with each subtask is a *pseudo-release* and *pseudo-deadline* (often called "release" and "deadline" for brevity), which are defined as

$$\begin{aligned}
\mathsf{r}(T_i^{[1]}) &= \theta(T_i^{[1]}) \\
\mathsf{r}(T_i^{[j]}) &= \mathsf{d}(T_i^{[j-1]}) + \left\lfloor \frac{j-1}{\mathsf{wt}(T_i)} \right\rfloor - \left\lceil \frac{j-1}{\mathsf{wt}(T_i)} \right\rceil + \theta(T_i^{j}),\ j > 1 \\
\mathsf{d}(T_i^{[j]}) &= \mathsf{r}(T_i^{[j]}) - \left\lfloor \frac{j-1}{\mathsf{wt}(T_i)} \right\rfloor + \left\lceil \frac{j}{\mathsf{wt}(T_i)} \right\rceil,\ j \geq 1
\end{aligned}$$

where $\theta(T_i^{[j]}) \geq 0$ for $j \geq 1$. $\theta(T_i^{[j]})$ denotes the "sporadic" separation between subtask releases (such separations are called "intra-sporadic"). PD^2 schedules subtasks on an earliest-pseudo-deadline-first basis with two tie-breaking rules, which are used in the event of a deadline tie. (A more thorough discussion of PD^2 can be found in Chapter 5.) Since tasks in PD^2 are scheduled one quantum at a time, a task's execution time is assumed to be a multiple of the quantum size (and must be rounded up if this is not the case) and the scheduling of a task depends only on its weight. The main advantage of PD^2 over all other aforementioned algorithms is that PD^2 is the only algorithm that can guarantee that every job is scheduled before its deadline and no processor is over-allocated provided total utilization is at most the number of processors. However, since tasks are scheduled on a per-quantum basis, it is

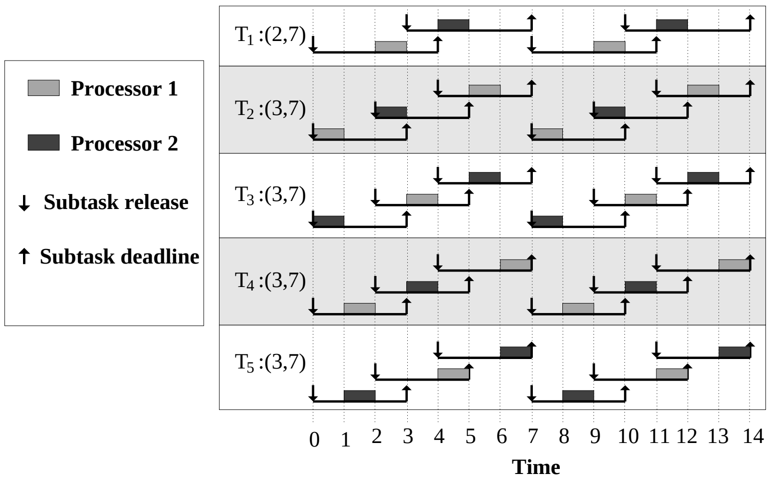


Figure 1.6: A two-processor system scheduled by PD^2.

possible that a task will be preempted and migrated every quantum, which can cause tasks to incur large migration/preemption costs. Another disadvantage of PD^2 is that task execution times must be rounded up, which can cause the system to be underutilized. For example, if a task has an execution time of 3.1 and a period of 4, then its weight would be $4/4 = 1$, since the execution time of 3.1 would be rounded up to one.

Example (Figure 1.6). Figure 1.6 shows a PD^2 schedule for the task system consider earlier in Figure 1.5. $\square$

Notice that, in this schedule, tasks are scheduled one quantum at a time. As a result, tasks may be preempted at the end of every quantum and may migrate nearly as frequently. Also note that a subtask (unlike a job) may be released before the deadline of its successor. Finally, notice that, in this schedule, every task is scheduled before its deadline and that tasks with the same weight (but different periods) receives allocations at the same rate, e.g., T_3 and T_4 receive approximately one allocation in any 7/3-quantum interval. $\square$

1.2.5 Impact of Migrations and Preemptions

Given these five algorithms, it is obvious that in the absence of migration and preemption costs, PD^2 should be the preferred algorithm since it is the only algorithm that can both guarantee that every job completes before its deadline and that the share of every task equals its weight. However, for many applications, migration and preemption costs may be substantial. Recently, our research group (Calandrino et al., 2006) constructed a multiprocessor testbed, called LITMUS^{RT}, to compare different real-time scheduling algorithms. We then used this testbed to implement all of the aforementioned algorithms (except NP-PEDF) on a four-processor system (with 2.7 GHz processors) in an effort to assess the impact of migration and preemption costs. In our work, we varied the weight of tasks being scheduled and the amount of cache used by each task. (Migrations and preemptions cause a loss of cache affinity, and as a result, if a task utilizes a larger fraction of the cache, then that task has both higher migration and preemption costs.)

Our experiments assessed the performance of each algorithm as measured by the number of processors that would be required to schedule a number of randomly generated task sets with a maximal utilization of four. These experiments showed the following:

- The HRT performance of PEDF and GEDF improve (relative to the other algorithms) as migration and preemption costs increase and/or the weights of tasks decrease. However, PEDF always has better HRT performance than GEDF.
- The performance of PD^2 improves (relative to the other algorithms) as migration and preemption costs decrease and/or the weights of the tasks increase.
- PD^2 has virtually the same performance in both HRT and SRT systems.
- PEDF has virtually the same performance in both HRT and SRT systems.
- Both GEDF and NP-GEDF always perform better than any other algorithm for SRT systems. Furthermore, NP-GEDF has slightly better performance than GEDF. (The HRT performance of NP-GEDF was not considered since there does not exist a scheduability test for it that would return meaningful results.)

The reason why the performance of PEDF is adversely impacted by increasing task weights is that, if tasks have higher weights, then it is harder to produce a valid partitioning. Similarly, for GEDF, as task weights increase, it becomes more likely that a job will be tardy. As a result, additional processing capacity is needed to prevent such a scenario. The performance of PD^2 is positively impacted by increasing task weights because PD^2 is more likely to schedule tasks with larger weights in consecutive quanta, thus reducing the number of migrations and preemptions. GEDF and NP-GEDF perform well for SRT systems because even though these algorithms can cause tasks to miss deadlines, they incur relatively little migration/preemption cost. Moreover, since tasks are not partitioned under GEDF and NP-GEDF, the performance (in terms of the required number of processors to guarantee bounded tardiness) does not substantially degrade as task weights increase. Given these results, it is easy to see that there there does not exist a single "best" multiprocessor scheduling algorithm, and the choice of which algorithm depends on the scenario in which it will be used. The theoretical and empirical results from this section are summarized in Tables. 1.1 and 1.2, respectively.

1.2.6 Research Needed

Under traditional task models (e.g., the sporadic model), the scheduability of a system is based on each task's WCET. The disadvantage of using WCETs is that a system may be deemed unschedulable even if it would function correctly most (or possibly all) of the time when deployed. Adaptive real-time scheduling algorithms allow per-task processor shares to be adjusted based upon run time conditions, instead of always using constant share allocations based upon WCETs. Prior to the research in the dissertation, for multiprocessor systems, one approach for reweighting a task had been proposed, as we will discuss in Section 2.1 (Srinivasan and Anderson, 2005); however, this approach only allows tasks to reweight at job boundaries. By delaying a task's reweighting request until its next job boundary, the system may "drift" from its "ideal" allocation by an arbitrarily large amount. As a result, for applications like Whisper and VEC, where timing constraints are continually changing, such delays may cause unacceptably poor performance. In this dissertation, we remedy this shortcoming by proposing a set of rules that allow tasks to reweight without causing unbounded drift, and an adaptive framework that determines at run time which adaptions are needed.

Scheme	**Desired = Guaranteed Weight**	**Guaranteed Deadlines**
PEDF	No	Yes
NP-PEDF	No	No
GEDF	Yes	No
NP-GEDF	Yes	No
PD^2	Yes	Yes

Scheme	**Migrations**	**Preemptions**
PEDF	Never	At Job Completions and Releases
NP-PEDF	Never	Never
GEDF	At Job Completions and Releases	At Job Completions and Releases
NP-GEDF	In Between Jobs	Never
PD^2	Every Quantum	Every Quantum

Table 1.1: Summary of algorithms and their properties. Note that deadlines can be guaranteed under PEDF only at the expense of allowing guaranteed weights to be less than desired weights.

Scheme (Hard)	**Light Tasks (Hard)**	**Heavy Tasks (Hard)**
PEDF	Best	Poor
GEDF	Good	Worst
NP-GEDF	N/A	N/A
PD^2	Worst	Best

Scheme (Hard)	**High Mig./Preemp. Costs (Hard)**	**Low Mig./Preemp. Costs (Hard)**
PEDF	Best	Poor
GEDF	Good	Worst
NP-GEDF	N/A	N/A
PD^2	Worst	Excellent

Scheme (All cases)	**Soft-Real Time**
PEDF	Same as hard
GEDF	Excellent
NP-GEDF	Best
PD^2	Same as hard

Table 1.2: Empirical performance of the algorithms under different conditions. *Heavy* tasks have a weight of at least 1/2, and *light* tasks have a weight less than 1/2. (Results are relative to other algorithms.)

1.3 Adaptivity

This section provides a review of prior work on adaptive uniprocessor real-time schemes in which tasks are reweighted based on external and internal stimuli.

1.3.1 Leave/join Reweighting

Under *leave/join reweighting* (Srinivasan and Anderson, 2005), a task's weight is changed at job boundaries by forcing it to leave with its old weight and rejoin with its new weight.

Example (Figure 1.7). Consider the example Figure 1.7, which depicts a one-processor system with three tasks: T_1:$(1, 2)$ that leaves at time 2; T_2, which has an initial weight of 1/4, an execution cost of 1, and "initiates" a weight increase at time 2 to a weight of 3/4, which is enacted by leave/join reweighting at T_2^1's deadline (i.e., time 4); and T_3:$(2, 8)$. **(a)** illustrates the EDF schedule and **(b)** illustrates the ideal and actual allocations to task T_2. □

Notice that, even though T_2 "initiates" its change at time 2 and capacity exists for T_2 to increase its weight, this weight change cannot be "enacted" until its deadline. This illustrates that the primary drawback to leave/join reweighting is that a task can *only* change its weight at job boundaries. As a result, over the time range [2, 4), T_2 behaves as though it is a task of weight 1/4, even though in the ideal system (which can instantly enact weight changes) T_2 would behave as a task of weight 3/4, is depicted in Figure 1.7(b). As a result, T_2's allocation in the actual schedule "drifts" from its allocation in the ideal schedule by one time unit. (Briefly, *drift* is the difference between a task's ideal and actual allocations caused by a single reweighting event.) Moreover, since leave/join reweighting cannot enact a reweighting event until a job boundary, it is possible that a task can incur an *arbitrarily large* amount of drift for one reweighting event. Section 1.5.3 provides a more detailed discussion concerning drift.

1.3.2 Rate-Based Earliest Deadline

Under *rate-based earliest-deadline* (RBED) scheduling (Brandt et al., 2003), which was proposed for uniprocessor systems, tasks are scheduled on an EDF basis and can change their

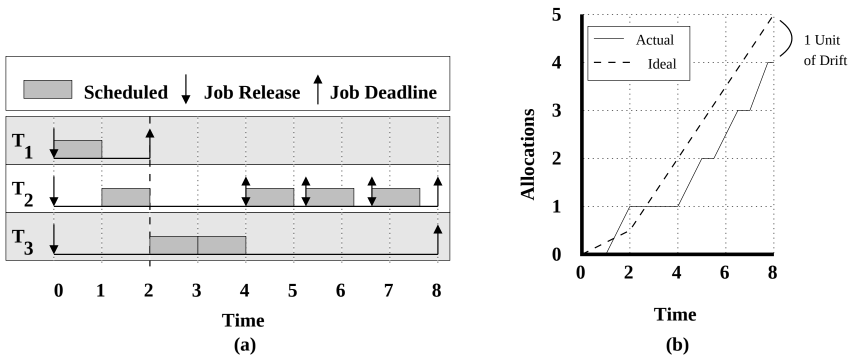


Figure 1.7: The **(a)** EDF schedule and the **(b)** ideal and actual allocations to T_2 in a one-processor example of leave/join reweighting.

weights and periods via two different rules based on whether the execution time or period is changed. While these rules are more responsive than leave/join reweighting, it is still possible for a task to incur an arbitrarily large amount of drift under RBED. In Chapter 2, we will review this work in detail.

1.3.3 Proportional Share Scheduling

Under *proportional share scheduling* (Stoica et al., 1996), the guaranteed weight of each task is determined as a function of its desired weight and the desired weight of all other tasks. Specifically, the guaranteed weight of the task T_i at time t is defined as

$$\mathsf{Gwt}(T_i, t) = \frac{\mathsf{wt}(T_i)}{\mathsf{WT}(\tau,\, t)}, \tag{1.2}$$

where $\mathsf{WT}(\tau,\, t)$ is the total desired weight of all active tasks in the system τ at time t. For example, consider a one-processor system that consists of four tasks: T_1 that has a desired weight of 1/2; T_2 that has a desired weight of 1/4; T_3 that has a desired weight of 1/4; and T_4 that has a desired weight of 1/8. If, at some time t_1 all four tasks are active, then the guaranteed weight for each task would be $1/(1/2 + 1/4 + 1/4 + 1/8) = 8/9$ of its desired weight. So, the desired weight of T_1 would be 4/9. Alternatively, if at some time, t_2, only T_1 and T_2 were active, then the guaranteed weight of each would be 4/3 of its desired weight for each task. So, the guaranteed weight of T_1 would be 2/3. It is worthwhile to note that proportional share algorithms cannot change the weight of only one task in a system without using leave/join techniques. In Section 2.3, we will review one of the more popular proportional share scheduling algorithms.

1.3.4 Adaptive Feedback-Based Frameworks

In *adaptive feedback-based scheduling algorithms*, the *execution time* of each job is unknown until it is complete. As a result, in these systems, each task's weight is defined as a function of its *estimated execution time*, which is calculated for a job by using the actual execution times of the task's prior jobs. Moreover, a user can fine-tune a feedback-based system to achieve desired behaviors.

Lu et al. (Lu et al., 2000) were the first to propose such a scheduling algorithm, which was directed at uniprocessor systems.[2] Under their algorithm, called $\mathsf{FC\text{-}EDF}^2$, each task has multiple versions (called *service levels*), each of which has a different level of *QoS* and a different *nominal processor share*, representing the fraction of a processor the task will require on average if it executes at that service level. A task can only execute at one service level at a time. In order to control the system, $\mathsf{FC\text{-}EDF}^2$ monitors the system's *utilization* and *miss-ratio*, i.e., the fraction of jobs with missed deadlines. In order to minimize the miss-ratio while maximizing utilization, $\mathsf{FC\text{-}EDF}^2$ adjusts the set of scheduled tasks and their service levels. More recently, Lu et al. extended this work to create a comprehensive feedback scheduling framework (Lu et al., 2002) that more explicitly incorporates the value to the system associated with each service level. This framework is the basis for the approach we propose in this paper. One drawback of $\mathsf{FC\text{-}EDF}^2$ is that, because only the utilization and the *system-wide* miss-ratio are monitored, the system cannot identify whether an individual task has an actual execution time that deviates substantially from its estimated execution time. Thus, the system can only respond to differences between the actual and estimated execution times of tasks by changing the entire system instead of only a few tasks.

Alternatively, Abeni et al. have proposed a uniprocessor feedback algorithm in which each task has its own feedback-controller rather than one controller for the entire system (Abeni

[2]Specifically, the first *correct* feedback algorithm was proposed in (Lu et al., 2000). The original system, FC-EDF, proposed in (Lu et al., 1999) could not satisfy its design specification because it was possible for the controller to become saturated, thus rendering it unable to correctly adjust the system.

et al., 2002). In order to attempt to maintain an accurate processor share for each task, their algorithm monitors, *for each task*, the difference between the estimated and actual execution times of each job. Once the system has calculated a new estimated execution time for a future job, it adjusts the task's weight. More recently, Cucinotta et al. extended this approach to provide stochastic guarantees concerning per-task processor shares (Cucinotta et al., 2004). One drawback of their approach is that it ignores the possibility that some tasks are more important than others.

In addition to general-purpose real-time scheduling algorithms, feedback-based scheduling has become increasingly important for managing *control tasks*, i.e., tasks that control external devices. In work by Martí et al. (Marti et al., 2004), an approach is proposed that is similar to that of Abeni et al., except that in (Marti et al., 2004), each period of each task has an associated "importance value" that denotes the task's value to the system in that period. By using importance values, Martí et al. determined the optimal period for each task via standard linear programming techniques. One limitation of this approach is that it cannot adjust the amount of time for which a task executes, like other approaches (e.g., like that of Lu et al. (Lu et al., 2002)).

1.4 Thesis Statement

There are two major limitations of prior work on adaptive uniprocessor systems. First, there does not exist a set of metrics for comparing different reweighting algorithms. Second, existing methods for changing the *weights* of tasks (i.e., leave/join reweighting and RBED) may give rise to an unacceptably long delay after initiating a weight change before it is enacted. For multiprocessor systems, the limitations are even more severe since the only method for changing the weight of a task is to use leave/join reweighting. Moreover, for multiprocessor systems, there is an inherent tradeoff, which must be explored, between the level of migration/preemption and the "accuracy" of the adaptive protocol. Also, there is no global feedback-based adaptive framework, which would be necessary for implementing applications like Whisper and VEC. The main thesis of this dissertation, which attempts to resolves these issues, is the following.

Multiprocessor real-time scheduling algorithms can be made more adaptive by allowing tasks to reweight between job releases. Feedback and optimization techniques can be used to determine at run time which reweighting events are needed. The accuracy of such an algorithm can be improved by allowing more frequent task migrations and preemptions; however, this accuracy comes at the expense of higher migration and preemption costs, which impacts average-case performance. Thus, there is a tradeoff between accuracy and average-case performance that will be dependent on the frequency of task migrations/preemptions and their cost.

1.5 Contributions

In this section, we briefly discuss the contributions of this dissertation.

1.5.1 Adaptable Task Model and Reweighting Algorithms

The first contribution we discuss in the dissertation is the adaptable task model (originally proposed in (Block et al., 2008b)). This model is an extension of the sporadic task model, where the weight of each task T, $\mathsf{wt}(T, t)$, is a function of time t, and a task's execution time can vary between job releases. A task T *changes weight* or *reweights* at time $t + 1$ if $\mathsf{wt}(T, t) \neq \mathsf{wt}(T, t + 1)$. If a task T changes weight at a time t_c between the release and the deadline of some job T_i^j, then the following two actions *may* occur:

(i) If T_i^j has not been scheduled by t_c, then T_i^j may be "halted" at t_c.

(ii) $\mathsf{r}(T_i^{j+1})$ *may* be redefined to be less than $\mathsf{d}(T_i^j)$.

In the sporadic model defined earlier, every job's deadline is at or before its successor's release. As we will discuss in Section 1.5.2, the reason why the above two actions may occur is that the value of $\mathsf{r}(T_i^{j+1})$ may change as a result of a reweighting event. The reweighting rules we present later in this section state the conditions under which the above actions may occur and the number of time units before $\mathsf{d}(T_i^j)$ that job T_i^{j+1} can be released.

As has already been discussed, when a task reweights, there can be a difference between when it initiates the change and when the change is enacted. The time at which the change is *initiated* is a user-defined time; the time at which the change is *enacted* is dictated by a set of conditions that differ slightly for each type of multiprocessor reweighting algorithm. Furthermore, the release and deadline of a job of an adaptable task is defined based on the weight of the task when it was released.

1.5.2 Reweighting Rules.

The second major contribution of this dissertation is the construction of reweighting rules for PEDF, NP-PEDF, GEDF, NP-GEDF, and PD^2. (The reweighting rules for PEDF, and by extension for NP-PEDF, were proposed by Block and Anderson in (Block and Anderson, 2006); the reweighting rules for GEDF and NP-GEDF were proposed by Block et al. in (Block et al., 2008b); and the reweighting rules for PD^2 were proposed by Block et al. in (Block et al., 2008a).) In all five reweighting algorithms, tasks change weight via one of two rules that are based on whether a task's active job is over- or under-allocated relative to an ideal schedule.

- If a task is *under-allocated*, then the change is enacted by immediately halting the active job and releasing a new job with the remaining execution time.
- If a task is *over-allocated*, then one of two actions occurs: **(i)** if the task increases its weight, then the change is enacted by immediately halting the active job and releasing the next job when the task's "ideal" allocation equals its "actual" allocation; **(ii)** if the task decreases its weight, then the active job immediately halts and when the task's "ideal" allocation equals its "actual" allocation, the change is enacted and the next job is released.

When a T_i^j is *halted* at time t, T_i^j is not scheduled after time t.

Example (Figure 1.8). Consider the examples depicted in Figure 1.8. Insets (a) and (b) pertain to a one-processor system with: T_1:$(1, 2)$, which leaves the system at time 2; T_2:$(1, 6)$; and T_3, with an execution cost of 3 and an initial weight of 1/6 that increases to 4/6 at time 2. Since the first jobs of T_2 and T_3 have the same deadline, we can arbitrarily choose which job has a higher scheduling priority. Inset (a) depicts the case where T_2 has higher priority, and as a result, T_3 is "under-allocated" relative to the ideal system when it reweights at time 2.

Thus, when T_3 reweights, its current job "halts" and its next job is immediately released. Inset (b) depicts the case where T_3 has higher priority, and as a result, T_3 is "over-allocated" relative to the ideal system, when it reweights at time 2. Thus, when T_3 reweights, its current job "halts" and its next job is not released until the difference between T_3's ideal and actual allocations is zero (at time 3). The dotted lines denote the interval of the first job of T_3 that has been changed by the reweighting event. Inset (c) depicts the ideal allocation of T_3 as a function of time. Notice that, at time 2, when T_3 increases its weight from 1/6 to 4/6, the ideal allocation rate increases from 1/6 to 4/6, and that at time 3, T_3's total ideal allocation equals 1. Inset (d) depicts a one-processor system with four tasks: T_1:$(1, 2)$, which joins the system at time 1.5; T_2:$(1, 6)$; T_3:$(1, 6)$; and T_4, which has an initial weight of 4/6 that decreases to 1/6 at time 1. Since T_4 is over-allocated at time 1 and decreases its weight, its weight change is enacted when its ideal allocation equals its actual allocation (at time 1.5). Inset (e) depicts ideal allocation for T_4. $\square$

One important property of the above reweighting rules is that reweighting events are *task-independent*. That is, if a task T_i changes its weight, then *only* the releases and deadlines of jobs of T_i change. As a result of this independence, a task can only increase its weight at time t if enough capacity for the reweighting event is available. For example, if a two-processor task system has three tasks each of weight 0.6, then none of these tasks can initiate a weight increase to 0.9 (since this would cause the system load to be 2.1), unless one of the other two tasks first decreases its weight or leaves the system. (If reweighting events were not task-independent, then it would be possible for a proportional-share scheduling algorithm to increase the weight of a task at the expense of the shares of other tasks in the system.)

1.5.3 Evaluating Algorithms

The next contribution of this dissertation is a comparison of the reweighting algorithm for different multiprocessor scheduling frameworks. In order to evaluate the reweighting algorithms, we use three different metrics: *overload*, *tardiness*, and *drift*. Overload is the maximal amount that a single processor is overutilized (assuming that the system is not overutilized). Tardiness is the maximal amount by which a task can miss a deadline. Drift is the maximal amount of

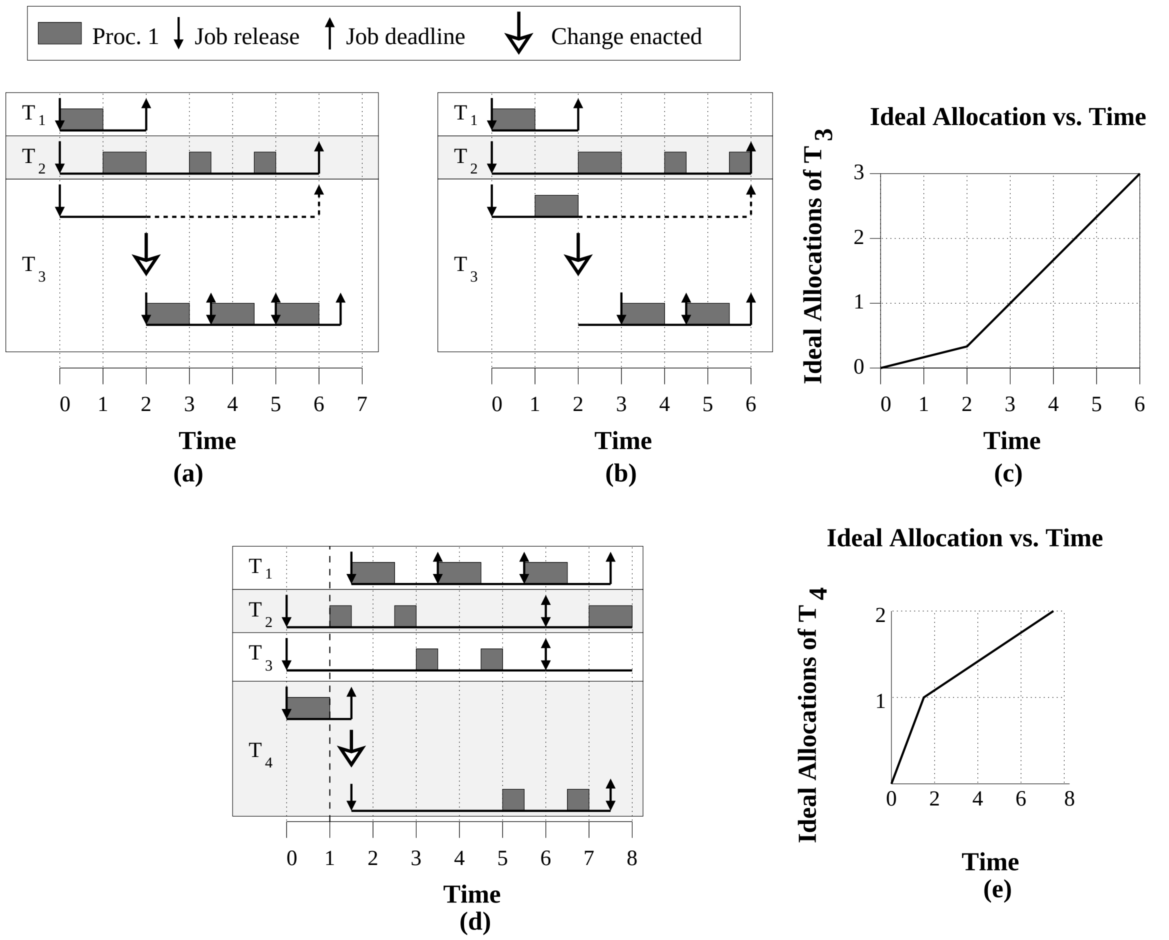


Figure 1.8: Several one-processor systems scheduled by EDF using our reweighting rules. **(a)** Increasing the weight of a task (T_3) when it is under-allocated. **(b)** Increasing the weight of a task (T_3) when it is over-allocated. **(c)** T_3's ideal allocation. **(d)** Decreasing the weight of a task (T_4). **(e)** T_4's ideal allocation.

Scheme	**Tardiness**	**Drift**	**Overload**
PD^2	0^3	2	0
PEDF	1	$\mathsf{e}_{\mathsf{max}}(T_i)$	W
NP-PEDF	$\mathsf{e}(T_i^j)$ $+\mathsf{e}_{\mathsf{max}}(T_i)+1$	$\mathsf{e}_{\mathsf{max}}(T_i)$	W
GEDF	$\kappa(m-1)+\mathsf{e}_{\mathsf{max}}(T_i)$	$\mathsf{e}_{\mathsf{max}}(T_i)$	0
NP-GEDF	$\kappa(m)+\mathsf{e}_{\mathsf{max}}(T_i)$	$\mathsf{e}_{\mathsf{max}}(T_i)$	0

Table 1.3: Summary of worst-case results for reweighting systems.

computation time a task "loses" between the initiation and enaction of a reweighting event. A comparison of the adaptable scheduling algorithms considered in this dissertation is given in Table 1.3. In this table, $\mathsf{e}_{\mathsf{max}}(T_i)$ denotes the *maximum execution time* of any job of task T_i; $\mathsf{wt}_{\mathsf{max}}(T_i)$ denotes the *maximal weight* of task T_i at any time; and W denotes the maximal weight of the $(m \cdot \lfloor 1/X \rfloor + 1)^{st}$ heaviest task (by maximal weight) in τ, where m is the number of processors, τ is the set of all tasks, and X is the maximal weight of the heaviest task. In addition,

$$\kappa(\ell) = \frac{\sum_{T_z \in \mathcal{E}(\ell)} \mathsf{e}_{\mathsf{max}}(T_z)}{m - \sum_{T_z \in \mathcal{X}(T)\ell-1} \mathcal{W}(T_z)} + \mathsf{e}_{\mathsf{max}}(T_i).$$

where $\mathcal{E}(\ell)$ is the set of ℓ tasks in τ with the highest *maximal* execution time and $\mathcal{X}(\ell-1)$ is the set of $\ell-1$ tasks in τ of largest *maximal* weight.

Overload. As mentioned above, *overload* is the maximal amount that a single processor is overutilized. For example, in Figure 1.4(b), the system is not overutilized (it has two processors and its utilization is two), yet one processor is overloaded by 1/3. Hence, overload is 1/3. Assuming that any *reasonable allocation decreasing* partitioning algorithm (i.e., tasks are sorted decreasing by weight before being assigned to processors) is used, no processor can be overutilized by more than the weight of the lightest task assigned to it (Lopez et al., 2004). More specifically, for partitioned algorithms, no processor is overutilized by more than W, where W is the weight of of the $\left(m \cdot \lfloor \frac{1}{X} \rfloor + 1\right)^{st}$ heaviest task and X is the weight of the heaviest task (Block and Anderson, 2006). (In Figure 1.4, T_1 is the $\left(m \cdot \lfloor \frac{1}{X} \rfloor + 1\right)^{st}$ heaviest task.) Thus, for partitioned algorithms, the overload error is W. As discussed in Section 1.4, under global scheduling algorithms, no single processor can be overutilized. Hence, for global scheduling algorithms, overload is zero.

Tardiness. Recall that one of the advantages for both PEDF and PD^2 is that both of these algorithms have zero tardiness (but, for PEDF, this requires allowing guaranteed weights to be less than desired weights). GEDF has a maximal tardiness that is at most the value given in (1.1). For the adaptive variants of these algorithms, the tardiness for both PEDF and PD^2 is still zero, whereas the tardiness bound for GEDF must be modified to include dynamic weights and execution times. Specifically, in Chapter 3, we establish that tardiness under adaptive GEDF is at most

$$\frac{\sum_{k=1}^{\Gamma(t)} \mathsf{max_e}(k)}{m - \sum_{k=1}^{\Gamma(t)-1} \mathsf{max_{wt}}(k)} + \mathsf{e}(T_i^j), \tag{1.3}$$

provided that, for all t, $\mathsf{WT}(\tau, t) \leq m$, where $\mathsf{WT}(\tau, t) = \sum_{T_i \in \mathcal{T}} \mathsf{wt}(T_i, t)$, and $\mathsf{max_e}(k)$ and $\mathsf{max_{wt}}(k)$ are, respectively, the k^{th} largest maximal execution time and weight of any task, and

$$\Gamma(t) = \begin{cases} \mathsf{WT}(\tau, t) - 1, & \mathsf{WT}(\tau, t) \text{ is integral} \\ \lfloor \mathsf{WT}(\tau, t) \rfloor, & \text{otherwise.} \end{cases}$$

Note that, because $\mathsf{NP\text{-}PEDF}$ and $\mathsf{NP\text{-}GEDF}$ are non-preemptive, their tardiness bounds are slightly larger than those for PEDF (which is zero) and GEDF, respectively.

Drift. For most non-adaptive real-time scheduling algorithms, the difference between a task's actual and ideal allocation lies within some bounded range centered at zero. For example, under a *uniprocessor* EDF schedule of a sporadic system, this difference lies within $(-\mathsf{e}(T_i), \mathsf{e}(T_i))$. When a weight change occurs, the same bounds are maintained except that they may be centered at a different value. For example, in Figure 1.8(b), the range is originally $(-1, 1)$ for task T_3, but after the reweighting event, it is $(-4/6, 8/6)$. This lost allocation is called *drift*. Given this loss (barring further reweighting events), T_i's drift will not change. In general, a task's drift per reweighting event will be non-negative if it increases its weight, or non-positive if it decreases its weight. For GEDF, $\mathsf{NP\text{-}GEDF}$, PEDF, and $\mathsf{NP\text{-}PEDF}$ the maximal absolute value of the drift for T_i^j is at most the maximal execution time of T_i (Block and Anderson, 2006; Block et al., 2008b). (For the remainder of this dissertation, we will use

[3] A tardiness of 0 is only guaranteed to hold for a given task T_i with an execution cost of a and a weight of b/c only if there exists some integer n such that $b \cdot n = a$; otherwise the tardiness of a task may be up to one quantum.

the term *absolute drift* to refer to the absolute value of the drift.) Under PD^2 the maximal absolute drift is 2 (Block et al., 2008a).

There is one subtle issue with the reweighting rules given in Section 1.5.2 that is important to mention. When designing the above rules, one of the guiding principals was that the reweighting rules be task-independent. As a result, when a task's weight decreases, if the active job is over-allocated and decreases its weight, then the capacity gained by decreasing the weight cannot be "released" until at or after the time at which the ideal allocation for the task equals its actual allocation. For example, in Figure 1.8(d), the capacity created by decreasing T_4's weight from 4/6 to 1/6 is not "released" until the time at which T_4's actual allocation equals its ideal allocation at time 1.5. This occurs even though T_4 decrease its weight at time 1. If such a delay did not exist, then a task could artificially increase its weight by continually decreasing and increasing its weight.

Example (Figure 1.9). Consider the example Figure 1.9, which depicts a one-processor system with six tasks: T_1:$(8, 10)$ and T_2, ..., T_6 each of which has an execution cost of one, an initial weight of 1/5, initiates a weight decrease to a weight of 0 immediately after being scheduled, and joins the system as soon capacity is available. Since the tasks in the set T_2, ..., T_6 continually decrease and increase their weights, over the range [0, 10), these tasks receive one-half of the available capacity, even though they should only receive one-fifth. As a result, T_1^1 misses its deadline by three time units. This example can be easily extended (by decreasing task weights) to construct scenarios with arbitrarily large tardiness. Since the capacity from a weight decrease cannot be released until ideal and actual allocations match, when a task decreases its weight there might exist an arbitrarily long length of time before the capacity is released. The problem with this delay is that, not only does it cause constant positive drift, but it may also cause a reweighting event *initiation* to be delayed for an arbitrarily long period of time. However, for *any* reweighting algorithm that both ensures bounded tardiness and uses task-independent reweighting, this delay is unavoidable. $\square$

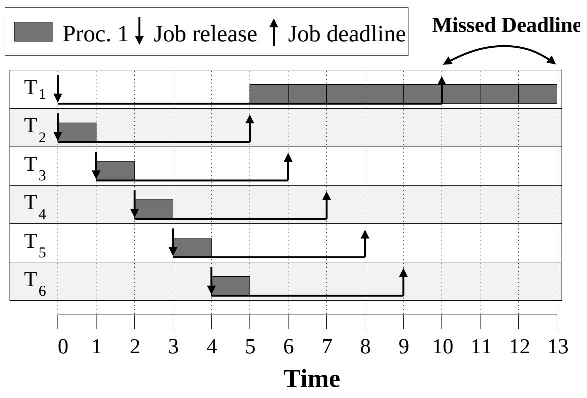


Figure 1.9: A one-processor system scheduled by EDF, which illustrates why task leaves must be delayed.

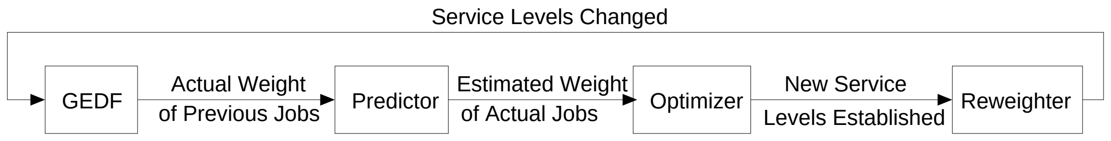

Service Levels Changed
GEDF
Actual Weight of Previous Jobs
Predictor
Estimated Weight of Actual Jobs
Optimizer
New Service Levels Established
Reweighter

Figure 1.10: The AGEDF system.

1.5.4 AGEDF

The multiprocessor reweighting algorithms we have developed require that the desired weights be provided. Desired weights can be determined using feedback-based techniques that use run time conditions. It is desirable that weight adjustments be enacted in a way that attempts to maximize overall QoS. Moreover, the allocation scheme should not “over-react” in adjusting weights when transient overloads occur. While there has been extensive work on adaptive feedback-based frameworks for uniprocessor systems (Abeni et al., 2002; Cucinotta et al., 2004; Lu et al., 2000; Lu et al., 1999), there has been relatively little work on multiprocessor feedback-based adaptive frameworks and the work that has been done has focused on non-preemptive systems where worst-case execution times, best-case execution times, and deadlines are static (Al-Omari et al., 2003; Sahoo et al., 2002).

In this dissertation, we remedy the shortcomings of prior adaptive multiprocessor real-time systems by presenting an *adaptive* GEDF *framework* called AGEDF (depicted in Figure 1.10), which consists of three components: a *predictor*, which uses feedback techniques to estimate the processor shares of future jobs; an *optimizer*, which uses the estimated processor shares of tasks to determine a new set of service levels; and several *reweighting rules*, which change the service levels to match those determined by the optimizer. To the best of our knowledge, this is the first such adaptive global framework for multiprocessor systems to be proposed. The reason why we constructed such a framework for GEDF and not for any of the other algorithms is because the feedback techniques used in this framework are best suited for SRT systems, and, as we discussed in Section 1.2.5, GEDF-based algorithms have superior SRT scheduability compared to PEDF- and PD^2-based algorithms.

1.5.5 AGEDF Implementation

As mentioned above, our research group developed a testbed called LITMUS$^{\mathrm{RT}}$ that allows different multiprocessor scheduling algorithms to be linked as plug-in components (Calandrino et al., 2006). As a final contribution, we implemented AGEDF as a LITMUS$^{\mathrm{RT}}$ plug-in. Our implementation of AGEDF consists of both a user-space library and kernel support added to LITMUS$^{\mathrm{RT}}$. In this section, we briefly discuss both parts.

Because LITMUS$^{\mathrm{RT}}$ was designed for sporadic tasks provisioned using WCETs, several modifications were needed to support adaptable sporadic tasks. These included: adjusting the internal structure of a task to allow each task to have multiple service levels; disabling the enforcement of WCETs to allow tasks to overrun their expected allocation; and modifying LITMUS$^{\mathrm{RT}}$ to allow task statistics such as actual execution times to be gathered.

After making these changes, we implemented AGEDF by changing the GEDF scheduling algorithm (which had already been implemented in LITMUS$^{\mathrm{RT}}$) in two ways. First, we introduced a system call to query the kernel in order for a task to determine its current service level. Second, we implemented the feedback, optimization, and reweighting components in kernel space.

After modifying LITMUS$^{\mathrm{RT}}$, we then evaluated its performance by using the core operations of both Whisper and VEC (correlation computations and bilateral filters, respectively). In this evaluation, AGEDF proved to be an extensible scheduling framework that can be easily configured to support different optimization criteria. Moreover, it exhibited good performance in scenarios in which the use of a non-adaptive GEDF algorithm would result in significant system over-utilization.

1.6 Organization

The organization of this dissertation is as follows. In Chapter 2, we review prior work on adaptive, real-time, and multimedia systems. In Chapters 3, 4, and 5, we define and prove the reweighting rules for GEDF, PEDF, and PD^2, respectively. In Chapter 6, we define the adaptable framework AGEDF. In Chapter 7, we discuss the implementation of AGEDF under LITMUS$^{\text{RT}}$ and present an experimental evaluation of AGEDF. Finally, we conclude in Chapter 8.

CHAPTER 2

PRIOR WORK

In this chapter, we review in detail prior work on adaptive real-time systems and feedback-control theory. As mentioned in Chapter 1, the adaptive framework we propose consists of two components: one that changes the parameters of running tasks (e.g., periods and weights) and one that determines such parameters. In Sections 2.1–2.2, we review three approaches for changing the periods and weights of running tasks. In Section 2.4, we review feedback-control theory, which is used for determining task execution times in both Lu et al.'s *Feedback-Control Real-Time Scheduling* (FCS) framework (Lu et al., 2002), which we cover in Section 2.5, and Abeni et al.'s *Adaptive Reservation-Based Scheduler* (Abeni et al., 2002), which we cover in Section 2.6.

2.1 Leave/Join Reweighting

As was mentioned in Section 1.3.1, under *leave/join reweighting* (Srinivasan and Anderson, 2005), a task's weight is changed at job boundaries by forcing it to leave with its old weight and rejoin with its new weight.

Example (Figure 2.1). Consider the example in Figure 2.1, which depicts a one-processor system with three tasks: T_1:$(1, 2)$, which leaves at time 2; T_2, which has an initial weight of 1/4, an execution cost of 1, and "initiates" a weight increase at time 2 to a weight of 3/4, which is enacted by leave/join reweighting at T_2^1's deadline (i.e., time 4); and T_3:$(2, 8)$. (This is the same system that was depicted in Figure 2.1, but has been repeated here to improve readability.) Inset (a) illustrates the EDF schedule and inset (b) illustrates the ideal and actual allocations to task T_2. Notice that, even though T_2 initiates its change at time 2 and capacity exists for T_2 to increase its weight, this weight change cannot be enacted until

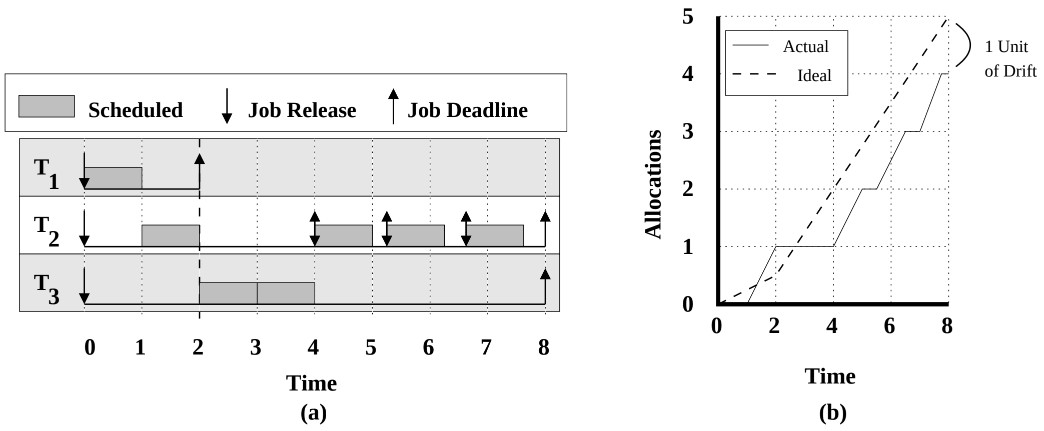


Figure 2.1: The **(a)** EDF schedule and **(b)** ideal and actual allocations to T_2 in a one-processor example of leave/join reweighting.

its deadline. This illustrates that the primary drawback of leave/join reweighting is that a task can *only* change its weight at job boundaries. As a result, T_2's allocation in the actual schedule drifts from its allocations in the ideal schedule by one quantum. $\square$

Since leave/join reweighting cannot enact a reweighting event until a job boundary, it is possible that a task can incur an *arbitrarily large* amount of drift for one reweighting event.

2.2 Rate-Based Earliest Deadline

Under *rate-based earliest-deadline* (RBED) scheduling (Brandt et al., 2003), tasks are scheduled on a uniprocessor on an EDF basis and can change their weights and periods via four different rules. Specifically, under RBED, if a task T_i changes its weight or period at time t, then the active job T_i^j of T_i at t is modified as follows, where x denotes the amount of time for which T_i^j has been scheduled before t:

1. T_i **increases its period to** P**.** T_i^j's deadline and period are immediately increased in accordance with the new period. Moreover, if T_i^j's deadline is increased to time D, then the execution time of T_i^j is increased by $\mathsf{wt}(T_i) \cdot (D - \mathsf{d}(T_i^j))$.

2. T_i **decreases its period to** P**.** If $x \geq \mathsf{wt}(T_i) \cdot (t - \mathsf{r}(T_i^j))$, then T_i^j's deadline is changed to $\mathsf{r}(T_i^j) + \mathsf{max}(x/\mathsf{wt}(T_i), P)$; otherwise, T_i^j's deadline is unchanged. Moreover, if T_i^j's deadline is decreased to time D, then T_i^j's execution time is reduced by $\mathsf{wt}(T_i) \cdot (\mathsf{d}(T_i^j) -$

D). Regardless of the value to which T_i^j's deadline was changed, T_i's period is changed to P. (Thus, $\mathsf{d}(T_i^{j+1}) - \mathsf{r}(T_i^{j+1}) = P$.)

3. T_i **increases its weight from** Ow **to** Nw**.** If $\mathsf{Nw} - \mathsf{Ow}$ is at most the amount of spare utilization (i.e., the total utilization of all other tasks at most $1 + \mathsf{Ow} - \mathsf{Nw}$), then T_i can increase its weight. If T_i increases its weight, then T_i^j's execution time increases by $(\mathsf{d}(T_i^j) - t) \cdot (\mathsf{Nw} - \mathsf{Ow})$.

4. T_i **decreases its weight from** Ow **to** Nw**.** T_i can decrease its weight to Nw only if $\mathsf{Nw} \geq \mathsf{Ow} - \frac{x}{t - \mathsf{r}(T_i^j)}$. Moreover, if T_i's weight is decreased at time t, then T_i^j's execution time is reduced by $(\mathsf{Ow} - \mathsf{Nw}) \cdot (\mathsf{d}(T_i^j) - t)$.

Intuitively, these rules seek to prevent a task from artificially increasing its allocations by repeatedly changing its weight and/or period. (Notice that in Rule 2, $\mathsf{r}(T_i^j) + x/\mathsf{wt}(T_i) \leq \mathsf{d}(T_i^j)$ since $x \leq \mathsf{e}(T_i^j)$ and $\mathsf{d}(T_i^j) = \mathsf{r}(T_i^j) + \mathsf{e}(T_i^j)/\mathsf{wt}(T_i)$.)

Example (Figure 2.2). Rules 1 through 4 are illustrated in Figure 2.2 via the following examples (each involving three tasks executing on one processor):

(a) T_1, T_2, and T_3 all have an initial weight of 1/3 and period of 3. At time 1, T_1 increases its period to 5 via Rule 1. As a result, T_1^1's execution time increases from 1 to 5/3.

(b) T_1, T_2, and T_3 all have an initial weight of 1/3 and period of 6. At time 1, T_1 decreases its period to 3 via Rule 2. As a result, T_1^1's execution time decreases from 2 to 1.

(c) T_1 has an initial weight of 1/6 and period of 3. T_2 and T_3 both have an initial weight of 1/3 and period of 3. At time 1, T_1 increases its weight to 1/3 via Rule 3. As a result, T_1^1's execution time increases from 1/2 to 5/6.

(d) T_1, T_2, and T_3 all have an initial weight of 1/3 and period of 6. At time 3, T_1 decreases its weight to 0 via Rule 4. As a result, T_1^1's execution time decreases from 2 to 0. $\square$

The primary drawback to RBED is that a task cannot decrease its period if it is under-allocated relative to the ideal schedule (i.e., $x < \mathsf{wt}(T_i) \cdot (t - \mathsf{r}(T_i^j))$). Thus, if an under-allocated task wants to decrease its period, it may be forced to wait until its deadline. Moreover, if

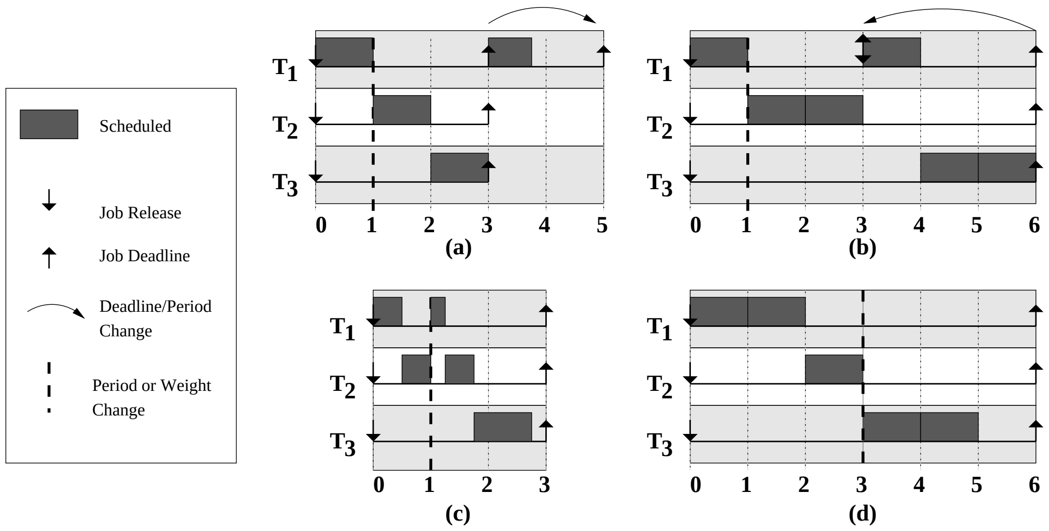


Figure 2.2: Several one-processor examples of RBED. Insets **(a)**–**(d)** illustrate Rules 1 through 4, respectively.

a task decreases its period in order to increase its weight (one of only two ways for a task to change its weight), then the task can incur an *arbitrarily large* amount of drift for one reweighting event.

2.3 Earliest Eligible Virtual Deadline First

As mentioned in Section 1.3.3, under *proportional share scheduling* (Stoica et al., 1996), the guaranteed weight of a task T_i at time t is defined as

$$\mathsf{Gwt}(T_i, t) = \frac{\mathsf{wt}(T_i)}{\mathsf{WT}(\tau,\, t)}, \tag{2.1}$$

where $\mathsf{wt}(T_i)$ is the desired weight of T_i and $\mathsf{WT}(\tau,\, t)$ is the total desired weight of all tasks in the system τ at time t.[1]

One of the preferred algorithms for proportional share scheduling on uniprocessors is the *earliest-eligible-virtual-deadline-first* (EEVDF) algorithm, proposed by Stoica et al. (Stoica et al., 1996). Under EEVDF, task releases and deadlines are defined based on an additional notion of time called *virtual time*. Virtual time, unlike "real time," scales with the processor

[1] In the literature on proportional share scheduling, weights can be arbitrary positive values; however in this dissertation, we will continue to view weights as desired processor shares.

load (i.e., as the processor load increases, virtual time slows down, and as the processor load decreases, virtual time speeds up). Intuitively, virtual time acts as a scaling factor for the system that allows task weights to be changed with minimal overhead. Specifically, the virtual time of the system τ at real time t is

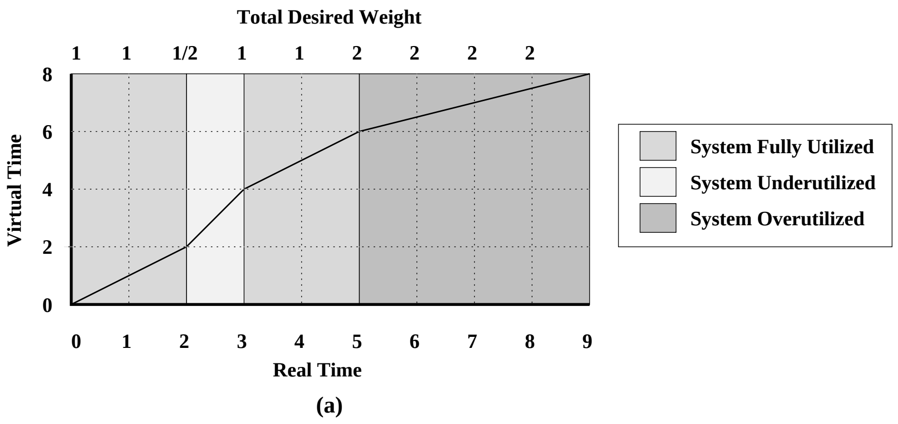
$$\mathsf{vt}(\tau, t) = \int_0^t \frac{1}{\mathsf{WT}(\tau,\, u)} du. \tag{2.2}$$

EEVDF utilizes the notion of virtual time by assigning each job both a *virtual deadline* and *virtual release*, which are defined as

$$\begin{aligned} \mathsf{vr}(T_i^j) &= (j-1) \cdot \frac{\mathsf{e}(T_i)}{\mathsf{wt}(T_i)} + \Theta(T_i^j) \\ \mathsf{vd}(T_i^j) &= \mathsf{vr}(T_i^j) + \frac{\mathsf{e}(T_i)}{\mathsf{wt}(T_i)}, \end{aligned}$$

respectively, where $j > 0$ and $\Theta(T_i^{j+1}) \geq \Theta(T_i^j) \geq 0$. $\Theta(T_i^j)$ is similar to the notion of a sporadic separation considered earlier except that it is measured in the virtual-time domain. Furthermore, under EEVDF, each task is scheduled on an EDF basis using virtual deadlines. Since virtual time scales with the system load and all deadlines are defined in terms of virtual time, the (real) time deadline of each job scales with the processor load so that no task misses its deadline.

Example (Figure 2.3). Consider the example in Figure 2.3, which depicts a one-processor system scheduled by EEVDF with six tasks, each of which has an execution time of one: T_1, which has a desired weight of 1/2, is initially in the system, and leaves at (real) time 2; T_2 and T_3, both of which have a desired weight of 1/4 and are initially in the system; T_4, which has a desired weight of 1/2 and joins the system at (real) time 3; and T_5 and T_6, both of which have a desired weight of 1/2 and join the system at (real) time 5. Inset (a) shows the mapping of virtual time to (real) time, and inset (b) shows the EEVDF schedule. In this example, the system is fully utilized over the (real) time ranges [0, 2) and [3, 5), and as a result, virtual time and (real) time progress at the same rate. Also, the system is under-utilized over the (real) time range [2, 3), and as a result, virtual time progresses faster than



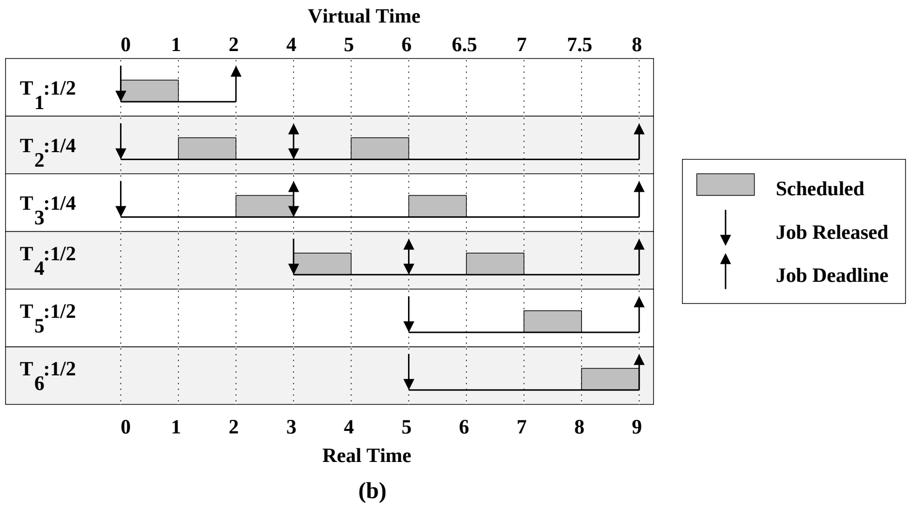


Figure 2.3: A one-processor system scheduled by EEVDF. **(a)** The mapping of virtual time to real time. The total desired weight of active tasks for each time range $[t,\ t+1)$ is labeled across the top axis. **(b)** The EEVDF schedule.

(real) time. Finally, the system is over-utilized over the (real) time range [5, 9), and as a result, virtual time progresses slower than real time. $\square$

One final note: while EEVDF can change the guaranteed weights of all tasks proportionally with minimal overhead, it cannot change the guaranteed (or desired) weights of tasks independently of each other without using leave/join reweighting. For example, in Figure 2.3(b), the only way to increase T_2's share to 1/2 is to decrease the processor load so that $\mathsf{wt}(T_2)$ accounts for half the processor load. Such a scenario occurs in the example over the (real) time range [2, 3).

2.4 Feedback-Control Theory

Feedback systems use the previous states of a system in order to predict and control the future behavior of the system. In this section, we describe the basics of feedback-control theory. The review presented in this section is taken from (Nise, 2004) and (Smith, 1997).

2.4.1 Basics of Feedback Theory

We begin by introducing some of the central definitions and terminology of feedback-control theory. Most feedback systems consist of the following components, as labeled in Figure 2.4: the *reference input value*, the *output value*, the *actuator*, the *error*, the *plant*, and the *controller*. The reference input value is the objective value for the system, while the output value is computed by the system. The actuator calculates the error by subtracting the output from the reference input value. The plant is the system we wish to control. The controller modifies the reference input value to change the behavior of the output. Depending on how frequently the system is *sampled*, i.e., the output of the system is fed back to the actuator, the system is classified as either an *analog* or a *discrete* system. Specifically, if the system is sampled continually, then it is an analog system; otherwise, it is a discrete system. Since we employ feedback techniques only at job completions (i.e., at a discrete set of times), we are only considered with discrete feedback-controlled systems, in this dissertation. In discrete feedback-controlled systems, the behavior of the plant and controller are specified as *difference equations*, i.e., for either a plant or a controller, if $x(k)$ is the output and $e(k)$ is the

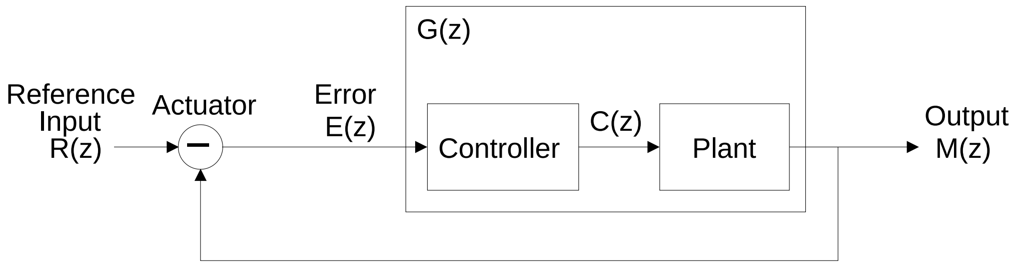


Figure 2.4: A simple feedback-control loop.

input, then $x(k)$ is of the form

$$x(k) = b_n e(k) + b_{n-1}e(k-1) + ... + b_o e(k-n) - a_{n-1}x(k-1) - ... - a_o x(k-n),$$

for some value of n, where a_i and b_i are constants and $x(k)$ and $e(k)$ are real-valued discrete functions of time k. Since $x(k)$ and $e(k)$ are functions of time, we say that they are in the *time domain*. Notice that difference equations are linear. One of the requirements for using feedback techniques is that there is a linear relationship between the input and the output.

The performance of a feedback system is measured in terms of *transient response*, *steady-state error*, and *stability*. The transient response of a system is the initial response of the system to a change in reference input value, as depicted in Figure 2.5. The steady-state error denotes the difference between the output and the reference input value of the system as time increases (also depicted in Figure 2.5). A system is considered to be *stable* if every bounded reference input value causes the system's steady-state error to be bounded. For feedback systems, it is crucially important that the system be stable.

While the behavior of a system is often defined in the time domain, when analyzing a feedback system, it is often helpful to transform the formulas that define the plant to the *frequency domain*, i.e., as a function of frequencies. In order to make this transformation, we use the *z-transform*, which is defined as

$$F(z) = \sum_{k=0}^{\infty} f(k)z^{-k}, \tag{2.3}$$

for the function $f(k)$ in the time domain. For the function $f(k)$, the convention is to write the

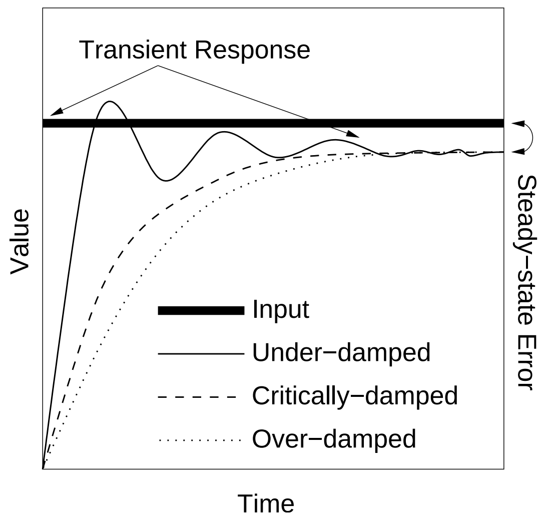


Figure 2.5: An example of an over-damped, under-damped, and critically-damped feedback system responding to a step input.

z-transformed function as $F(z)$, where z is a complex variable. A more detailed discussion of the z-transform can be found in (Nise, 2004).

By taking the z-transform of the plant (or controller), we get its *transfer function*, which relates the input of the plant (or controller) to its output. More specifically, if $i(k)$ is the input of a plant (or controller) and $p(k)$ is its output, then the transfer function for the plant (or controller) is given by $\frac{P(z)}{I(z)}$, where $I(z)$ is the z-transform of $i(k)$ and $P(z)$ is the z-transform of $p(k)$. For example, consider the system depicted in Figure 2.6 in which the difference equation for the controller is

$$c(k+1) = a_1 \cdot e(k) + a_2 \sum_{j=1}^{j=k-1} e(j),$$

and the plant is

$$m(k) = b \cdot c(k).$$

In this example, the transfer function for the controller is given as

$$\frac{C(z)}{E(z)} = \frac{a_1 \left(z - \frac{a_1 - a_2}{a_1}\right)}{z(z-1)},$$

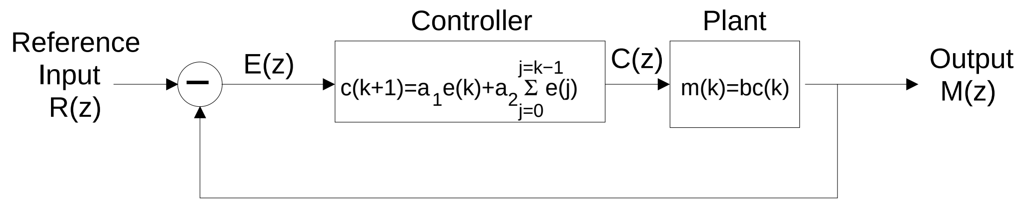


Figure 2.6: An example feedback-control loop, where the controller is defined as $c(k+1) = a_1 e(k) + a_2 \sum_{j=0}^{j=k-1} e(j)$ and the plant is defined as $m(k) = b \cdot c(k)$.

and the transfer function for the plant is given as

$$\frac{M(z)}{C(z)} = b.$$

By combining the transfer function of the plant and the controller, we get the *open-loop transfer function*, which is so called because it ignores the feedback loop. In Figure 2.6, the open-loop transfer function is given by

$$G(z) = \frac{M(z)}{E(z)} = \frac{M(z)}{C(z)} \frac{C(z)}{E(z)} = b \frac{a_1 \left(z - \frac{a_1 - a_2}{a_1}\right)}{z(z-1)}.$$

The *open-loop zeroes* of a system are the values of z such that the numerator of the open-loop transfer function equals zero. Similarly, the *open-loop poles* of a system are the values of z such that the denominator of the open-loop transfer function equals zero. In the system depicted in Figure 2.6, the open-loop zero is $\frac{a_1 - a_2}{a_1}$ and the open-loop poles are 0 and 1.

The *closed-loop transfer function*, which incorporates both the behavior of the controller and feedback loop, is given by

$$H(z) = \frac{G(z)}{1 + G(z)}, \tag{2.4}$$

where $G(z)$ is the open-loop transfer function. In Figure 2.6, the closed-loop transfer function is

$$H(z) = \frac{G(z)}{1 + G(z)} = \frac{b a_1 \left(z - \frac{a_1 - a_2}{a_1}\right)}{z^2 + (b a_1 - 1) z - b(a_1 - a_2)}.$$

The *closed-loop zeroes* of the system are the values of z such that the numerator of the closed-loop transfer function equals zero. Similarly, the *closed-loop poles* of the system are the values of z such that the denominator of the closed-loop transfer function equals zero. In our previous example, the closed-loop zero is $\frac{a_1 - a_2}{a_1}$ and the closed-loop poles are

$$\frac{(1 - ba_1) \pm \sqrt{(ba_1 - 1)^2 + 4b(a_1 - a_2)}}{2}.$$

2.4.2 Feedback Characteristics

In this section, we describe how to determine the stability, transient response, and steady-state error for feedback systems. For the remainder of this section, we use $\mathcal{R}(\mathcal{P})$, and $\theta(\mathcal{P})$ to denote, respectively, the radius and angle (in radians) of the pole $\mathcal{P}$ in polar-complex form, and we use $\mathcal{P}_m$ to denote the closed-loop pole that is farthest from the origin.

Stability. Recall from Section 2.4.1 that a system is stable if every bounded reference input value causes the system's steady-state error to be bounded. The test for stability is based on the location of the closed-loop poles in the complex plane: a system is *stable* if all closed-loop poles are within the unit circle in the complex plane, i.e., $\mathcal{P}_m < 1$. A system is *unstable* if any closed-loop pole is outside the unit circle, i.e., $\mathcal{P}_m > 1$. A system is *marginally stable*, in which case the output neither converges nor diverges, if at least one pole is on the unit circle and no pole is outside of the unit circle, i.e., $\mathcal{P}_m = 1$. Thus, in the system from Figure 2.6, if $a_1 = 2$ and $a_2 = 1$, then system is stable if $b \in (0, 2/3)$, the system is marginally stable if $b = 0$ or $b = 2/3$, and the system is unstable if $b > 2/3$ or $b < 0$.

Transient response. Two of the most important types of feedback systems are *first-* and *second-order* systems. A feedback system is considered to be a *first-order* system if it has one closed-loop pole and a system is considered to be a *second-order* system if has two closed-loop poles. These two types of systems are important because both have a set of simple formulas for determining their transient response. (Higher-order systems often have a transient response that is too complex to determine without approximation.)

The transient response of a system is usually evaluated by examining the behavior of the output when the system incurs a *step input*, i.e., the reference input value suddenly increases to a given value. Since feedback systems use previous results to predict future results, a step reference input value represents the worst-case scenario—a sudden change from one value to another. For a first-order system, the transient response is characterized by its *settling time* (i.e., the time it takes for the output to attain and stay within 2% of its steady-state value) and whether the output "overshoots" its final value. For example, such a scenario is depicted by the curve labeled under-damped in Figure 2.5. For a second-order system, the transient response is characterized by its settling time and whether it is *under-damped*, *over-damped*, or *critically-damped* (all three are depicted in Figure 2.5.) The settling time (where time is measured in terms of samples) of the system is given by the standard formula

$$\left\lceil \frac{-4}{ln\left(\mathcal{R}(\mathcal{P}_m)\right)} \right\rceil .$$

(This formula has a ceiling because time is discrete.)

For first-order systems, the output overshoots its final value if $\mathcal{P}_m < 0$. For first-order systems, it is typically undesirable for the output to overshoot its final value. (This is not the case for second-order systems since for second-order systems overshooting may be the only way to achieve the specified settling time. For most first-order systems, it is possible to achieve a desired settling time without overshooting the output.)

If a second-order system is over-damped, then the output will never overshoot the reference input value for a step input. If a second-order system is under-damped, then the output will overshoot the reference input value for a step input. For under-damped systems, the *percent overshoot* is an additional characteristic of transient response. If a second-order system is critically-damped, then the settling time is as small as possible without causing the output to overshoot the reference input value. Whether a system is under-, over-, or critically-damped depends on the location of the closed loop poles. If both poles are unique, real, and positive, then the system is over-damped. If both poles have the same radius, are real, and are positive, then the system is critically-damped. Otherwise, the system is under-damped. For under-damped systems, the percent overshoot is given by

$$e^{-(\zeta\pi/\sqrt{1-\zeta^2})} \cdot 100, \tag{2.5}$$

where ζ is a value called the *damping ratio* and is given by

$$\zeta = \frac{-ln\left(\mathcal{R}(\mathcal{P}_m)\right)}{\sqrt{\theta(\mathcal{P}_m)^2 + ln^2\left(\mathcal{R}(\mathcal{P}_m)\right)}}$$

For example, in the system from Figure 2.6, if $a_1 = 2$ and $a_2 = 1$, then the system is under-damped for any value of $b \in (0, 2/3)$. Alternatively, if $b = 1$, $a_1 \approx 0.102$, and $a_2 \approx 0.3035$, then the system is critically-damped and has a settling time of 6 time units. Finally, if $b = 1$, $a_1 = 1.4008$, and $a_2 = 1.60238$, then the system is under-damped, the settling time is 5 time units, and the percent overshoot is approximately 10.3%.

Steady-state error. Finally, we turn our attention to steady-state error. The steady-state error of a system is measured based on the system's response to a step and/or a *ramp input*. The ramp input simulates a reference input value that constantly increases by a rate of $\mathcal{T}$ per time unit. The steady-state error for a system is determined by using the *final value theorem*, which states that if $E(z)$ is the z-transform of a system's error, then the steady state error is given by

$$\lim_{z \to 1} \frac{z-1}{z} E(z). \tag{2.6}$$

Since $E(z)$ can be defined as

$$E(z) = R(z) - M(z), \tag{2.7}$$

where $R(z)$ is the reference input value and $M(z)$ is the output, and $M(z)$ can be defined as

$$M(z) = E(z)G(z),$$

where $G(z)$ is the open-loop transfer function, we get

$$E(z) = \frac{R(z)}{1 + G(z)}. \tag{2.8}$$

Since the z-transform of the step input is given by $R(z) = \frac{z}{z-1}$, from (2.6) and (2.8), we can derive the steady-state error of a system in response to a step input as

$$\lim_{z \to 1} \frac{1}{1 + G(z)}. \tag{2.9}$$

Since the z-transform of the ramp input is given by $R(z) = \frac{z\mathcal{T}}{(z-1)^2}$, from (2.6) and (2.8), we can derive the steady-state error of a system in response to a ramp input as

$$\frac{\mathcal{T}}{\lim_{z \to 1} (z-1)G(z)}. \tag{2.10}$$

The value to which the $(z-1)$ term is raised in the denominator of the open-loop transfer function, $G(z)$, is called the *system type*, and it is used to quickly determine if the steady-state error is zero, some non-zero constant, or ∞. Specifically, if $G(z)$ has a system type of zero (i.e., it has no $(z-1)$ terms in its denominator), then the system has a steady-state error of some constant value for the step input and ∞ for any ramp input. If $G(z)$ has a system type of one (i.e., it has one $(z-1)$ term in its denominator), then the system has a steady-state error of 0 for the step input and a constant for any ramp input. Finally, if $G(z)$ has a system type of two (i.e., it has two $(z-1)$ terms in its denominator), then the system has a steady-state error of 0 for any step or ramp input.

For example, in the system depicted in Figure 2.6, since there is one $(z-1)$ term in the denominator of $G(z)$, the system type is one. Thus, it has a steady-state error of zero for a step input and a constant steady-state error for any ramp input. Specifically, if $b = 0.5$, $a_1 = 2$ and $a_2 = 1$, then the steady-state error for the ramp response is $2\mathcal{T}$. Alternatively, if $b = 1$, $a_1 \approx 0.102$, and $a_2 \approx 0.3035$, then the steady-state error for the ramp input is approximately $3.295\mathcal{T}$. Finally, if $b = 1$, $a_1 = 1.4008$, and $a_2 = 1.60238$, then the steady-state error is approximately $0.624\mathcal{T}$.

2.4.3 Controllers

As mentioned above, the purpose of a controller is to improve the system response. There are three main types of of controllers: *proportional-integral* (PI) controllers; *proportional-derivative* (PD) controllers; and *proportional-integral-derivative* (PID) controllers. PI controllers improve the steady-state error of the system by increasing the system-type. Time-domain definitions of such controllers are of the form

$$c(k+1) = a \cdot e(k) + b \sum_{j=1}^{j=k-1} e(j),$$

and the z-transform of such a controller is

$$G(z) = \frac{C(z)}{E(z)} = \frac{a\left(z - \frac{a-b}{a}\right)}{z(z-1)}.$$

PD Controllers improve the transient response of the system by an additional closed-loop zero. Time-domain definitions of such controllers are of the form

$$c(k+1) = a \cdot e(k) + b(e(k) - e(k-1)),$$

and the z-transform of such a controller is

$$G(z) = \frac{C(z)}{E(z)} = \frac{(a+b)z - b}{z^2}.$$

PID Controllers improve the transient response and the steady-state error of the system by combining both PI and PD techniques. Such controllers are of the form

$$c(k+1) = a \cdot e(k) + b(e(k) - e(k-1)) + d \sum_{j=1}^{j=k-1} e(k),$$

and the z-transform of such a controller is

$$G(z) = \frac{C(z)}{E(z)} = \frac{(a+b)z^2 + (d-a-2b)z + b}{z^2(z-1)}.$$

The problem with PID controllers is that such controllers can easily increase the system beyond a second-order system. As a result, such systems are substantially more difficult to design for a specific transient response than PI or PD controllers.

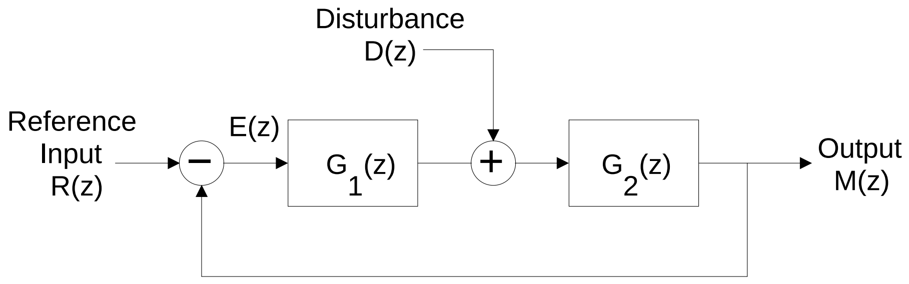


Figure 2.7: A feedback-control loop with a disturbance.

2.4.4 Disturbance

In addition to controlling a system based on its reference input value, a feedback system can be designed to handle *disturbances*, which represent an additional (typically unwanted) source of input to the system. For example, if we were constructing a feedback-controlled cruise control system, then the reference input value would be the car's desired speed and the disturbance would be the slope of the road. The typical model for a feedback system with a disturbance is shown in Figure 2.7. For such a system, the output of the system, $M(z)$, is given by

$$M(z) = E(z)G_1(z)G_2(z) + D(z)G_2(z), \tag{2.11}$$

where $E(z)$ is the z-transform of the error, $G_1(z)$ and $G_2(z)$ are the transfer functions for either a plant or a controller, and $D(z)$ is the z-transform of the disturbance.

When constructing a system that may have a disturbance, the primary design characteristic of interest is the steady-state error in response to a step input by the disturbance. Thus, just as in Section 2.4.2, to solve for the steady state error, we need to find a transfer function that relates $D(z)$ to $E(z)$. Recall from (2.7) that

$$E(z) = M(z) - R(z). \tag{2.12}$$

By substituting (2.12) into (2.11), we get

$$E(z) = \frac{1}{1 + G_1(z)G_2(z)}R(z) - \frac{G_2(z)}{1 + G_1(z)G_2(z)}D(z), \tag{2.13}$$

which defines the relationship between $E(z)$ and both the reference input value, $R(z)$, and the disturbance, $D(z)$. By applying the final value theorem to (2.13), we can obtain the steady-state error of this system as

$$\lim_{z \to 1} \frac{z-1}{z} \left(\frac{1}{1+G_1(z)G_2(z)} R(z) - \frac{G_2(z)}{1+G_1(z)G_2(z)} D(z) \right). \tag{2.14}$$

By isolating the disturbance term from (2.14) and substituting the transfer function for a step input into $D(z)$, i.e., $D(z) = \frac{z}{z-1}$ we obtain that the steady-state error for a step input in the disturbance is

$$\lim_{z \to 1} \left(- \frac{1}{\frac{1}{G_2(z)} + G_1(z)} \right). \tag{2.15}$$

2.4.5 Feedback Theory For a Predictor

It is worthwhile to note that while feedback-based techniques are primarily used to control the behavior of a plant for which the (reference) input is known, another viable use for such techniques is to *predict* future values of a changing and unknown input. The design of such a system is exactly the same as the typical feedback system, *except that the feedback loop does not directly impact the behavior of the system.* (Thus, the plant and the controller can be one-in-the-same.) In such a system, the transient response describes the initial accuracy of predictions after there has been a change in the input, and the steady-state error describes the difference between the predicted and actual values as system time increases.

By using feedback-based techniques in the construction of the predictor, instead of using a simpler approach, such as setting the current value to equal the previous value, the predictor both produces values that are less susceptible to ephemeral fluctuations in the workload and is capable of closely tracking trends in the value (i.e., such systems have a bounded steady state error for the ramp input).

2.5 The FCS Framework

In this section, we review the uniprocessor *Feedback-Control Real-Time Scheduling* (FCS) framework proposed in (Lu et al., 2002). (The FC-EDF and FC-EDF^2 algorithms proposed in (Lu et al., 1999) and (Lu et al., 2000) and mentioned in Chapter 1 are subsets of this framework.) As was mentioned in Chapter 1, in the FCS framework, unlike in most work on real-time feedback scheduling algorithms, it is assumed that the execution time of a job is unknown until it is complete. Given this limitation, the FCS framework has two objectives. First, maintain the total system utilization at some user-defined value, u_s. Second, if the system has a utilization greater than one, then maintain the *miss-ratio* (i.e., the fraction of jobs with a missed deadline) at some user-defined value, $\mathsf{m_s}$. In order to construct such a system, the FCS framework assumes a task model in which each task has multiple versions (called *service levels*), each of which has a period, an *estimated execution time*, which represents the amount of time the task will require on average if it executes at that service level, and a *QoS value*, which represents the value to system if the task finishes before its deadline while executing at that service level.[2] Throughout this section, we will use $\mathsf{p}(T_i, k)$, $\mathsf{e}(T_i, k)$, and $\mathsf{v}(T_i, k)$ to denote, respectively, the period, estimated execution time, and QoS value for the k^{th} service level of T_i, where service levels are ordered increasingly by *estimated weight*, which is defined as $\mathsf{e}(T_i, k)/\mathsf{p}(T_i, k)$. A task can only execute at one service level at a time. Without loss of generality, we assume that for any task T_i, service level 0 is defined such that $\mathsf{p}(T_i, 0) = 0$, $\mathsf{e}(T_i, 0) = 0$, and $\mathsf{v}(T_i, 0) = 0$.

2.5.1 The FCS Framework's Architecture

The major components of the FCS framework are depicted in Figure 2.8. At a high level, these components function as follows.

- *At each instant*, the pending job with the smallest deadline is scheduled.[3]

- *After a* sampling period *of length t time units*, where t is a user-defined value, several actions occur. First, the *monitor* calculates in the last t time units both the fraction of jobs that missed their deadlines (i.e., the miss-ratio) and the fraction of time that the processor was scheduled (i.e., the utilization). Next, the *controller* calculates the change

[2]The FCS framework can be used for aperiodic tasks; however, since the focus of this dissertation is on periodic/sporadic systems, we focus exclusively on this aspect of the FCS framework.

[3]The FCS framework can be extended to work with non-EDF based approaches; however, since those are beyond the focus of this dissertation, we do not discuss such extensions here.

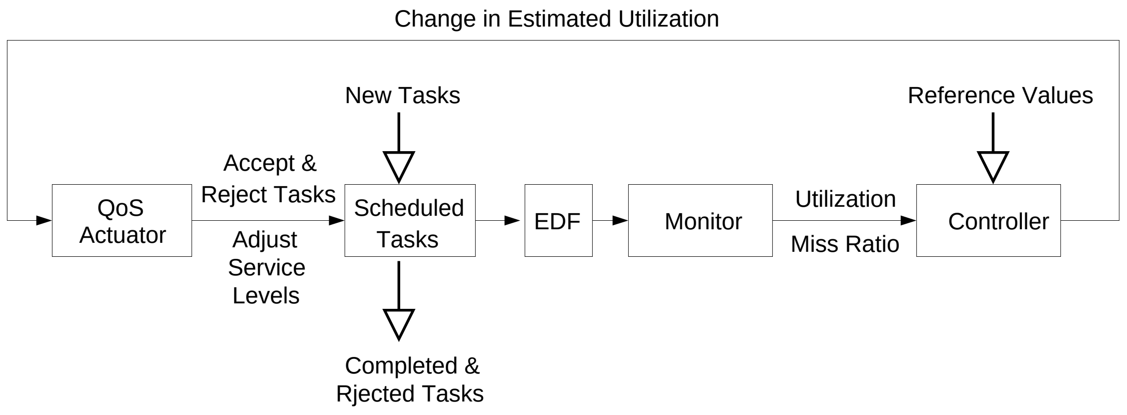


Figure 2.8: The design of the FCS.

in the total estimated utilization that is required to maintain the actual utilization at u_s or the miss-ratio at $\mathsf{m_s}$. Finally, the QoS actuator changes the set of running tasks and the service levels of running tasks to match the total estimated utilization as calculated by the controller.

It is important to note that in the FCS framework, the miss-ratio and the actual utilization are always calculated over the previous t time units.

2.5.2 Feedback in the Controller and QoS Actuator

The heart of the FCS framework is the controller and the QoS actuator, which use feedback techniques in order to determine the required change to the estimated utilization. In order to calculate this change, two different feedback loops are used: one loop determines the change in the estimated utilization that would maintain the actual utilization at u_s, and the other loop determines the change in the estimated utilization that would maintain the miss-ratio at $\mathsf{m_s}$. The actual change in the estimated utilization is calculated by dynamically switching between the values produced by these two loops. In this section, we describe these two feedback loops.

Before continuing, there is one subtle issues that must be discussed. As was mentioned in Section 2.4.1, the relationship between the input and output for a plant or a controller must be linear. Unfortunately, for the loop that monitors the miss-ratio, the relationship between the estimated utilization (again, the input) and the miss-ratio (the output) is non-linear. Moreover, for the loop that monitors the actual utilization, the relationship between the estimated utilization (the input to the plant/controller) and the actual utilization (the output to the plant/controller) cannot be determined exactly. As a result, Lu et al. use a linear estimation in both the utilization and miss-ratio feedback loops.

Utilization feedback loop. Before presenting difference equations for the utilization feedback loop, depicted in Figure 2.9(a), we introduce a few definitions. Let the value $\mathsf{a}(k)$ denote the fraction of time the processor was busy over the k^{th} sampling period, i.e., $\mathsf{a}(k)$ equals the fraction of time the processor was busy over the time range $[(k-1)\cdot q,\, k\cdot q)$. Let $\mathsf{Ew}(k)$ denote the total estimated utilization of all tasks in the k^{th} sampling period. Let $\mathsf{G_u}$ to denote the maximum value of $\mathsf{a}(k)/\mathsf{Ew}(k)$ for any value of k. As mentioned above, the relationship between the estimated utilization, $\mathsf{Ew}(k)$, and the actual utilization, $\mathsf{a}(k)$, cannot be determined exactly. As a result, the plant in the utilization feedback loop defines a relationship between the estimated utilization and the *worst-case estimated utilization*, which is denoted as $\mathsf{u}(k)$. Specifically, the difference equation for the plant is defined as

$$\mathsf{u}(k) = \mathsf{G_u}\mathsf{Ew}(k), \qquad \text{if } \mathsf{G_u}\mathsf{Ew}(k) \leq 1 \tag{2.16}$$

$$\mathsf{u}(k) = 1, \qquad \text{if } \mathsf{G_u}\mathsf{Ew}(k) > 1. \tag{2.17}$$

Notice that this system is linear so long as it is not *saturated*, i.e., as long as $\mathsf{G_u}\mathsf{Ew}(k) < 1$. Also note that in the absence of saturation, (2.16) can be rewritten as

$$\mathsf{u}(k) = \mathsf{u}(k-1) + \mathsf{G_u}\mathsf{c_u}(k-1), \tag{2.18}$$

where $\mathsf{c_u}(k) = \mathsf{Ew}(k+1) - \mathsf{Ew}(k)$. Thus, in the absence of saturation, the open-loop transfer function for the utilization of the system can be written as

$$\mathsf{P_u}(z) = \frac{\mathsf{U}(z)}{\mathsf{C_u}(z)} = \frac{\mathsf{G_u}}{(z-1)}. \tag{2.19}$$

The controller defines a relationship between the error of the system, which is defined as $\epsilon_u(k) = \mathsf{u}_s - \mathsf{u}(k)$, and the change in estimated utilization, $\mathsf{c_u}(k)$. Thus, the difference equation for the controller is given as

$$\mathsf{c_u}(k) = \mathsf{K_{pu}}\epsilon_u(k), \tag{2.20}$$

where $\mathsf{K_{pu}}$ is a tunable parameter The open-loop transfer function for (2.20) is given by

$$\mathsf{H}_u(z) = \mathsf{K_{pu}}. \tag{2.21}$$

Notice that this is a proportional controller (i.e., a P Controller). Lu et al. do not include an integral controller in their design because the formulation of the plant already has a system-type of one. Additionally, a derivative controller is not necessary since its primary purpose is to introudce an additional variable for controlling the transient response of the system, and for the types of inputs that Lu et al. are concerned with the system has a sufficient number of variables that control the system's behavior.

From (2.19) and (2.21), we can derive an open-loop transfer function for the plant and controller combined as

$$\mathsf{Q}_u(z) = \mathsf{H}_u(z)\mathsf{P_u}(z) = \frac{\mathsf{K_{pu}}\mathsf{G_u}}{(z-1)}, \tag{2.22}$$

and the closed-loop transfer function as

$$\mathsf{Y}_u(z) = \frac{\mathsf{Q}_u(z)}{1+\mathsf{Q}_u(z)} = \frac{\mathsf{K_{pu}}\mathsf{G_u}}{z-(1-\mathsf{K_{pu}}\mathsf{G_u})}. \tag{2.23}$$

Miss-ratio feedback loop. As mentioned above, the exact relationship between the estimated utilization, $\mathsf{Ew}(k)$, and the miss-ratio, $\mathsf{m}(k)$, in sampling period k is nonlinear. As a result, Lu et al. approximate the relationship between the two by using the derivative of this relationship around the vicinity of $\mathsf{m_s}$. (The notion of "around the vicinity of $\mathsf{m_s}$" is loosely defined by Lu et al., as we will discuss in Section 2.5.3.) Specifically, the plant is defined as

$$\mathsf{m}(k) = 0, \qquad \text{if } \mathsf{G_u}\mathsf{Ew}(k) \leq 1 \tag{2.24}$$

$$\mathsf{m}(k) = \mathsf{m}(k-1) + \mathsf{G_m}\mathsf{G_u}\mathsf{c_m}(k-1), \qquad \text{if } \mathsf{G_u}\mathsf{Ew}(k) > 1, \tag{2.25}$$

where $\mathsf{c_m}(k) = \mathsf{Ew}(k+1) - \mathsf{Ew}(k)$ and $\mathsf{G_m}$ is the maximum value of

$$\frac{d\mathsf{m}(k)}{d(\mathsf{G_u}\mathsf{Ew}(k))},$$

around the vicinity of $\mathsf{m_s}$. Lu et al. suggest deriving both $\frac{d\mathsf{m}(k)}{d(\mathsf{G_u}\mathsf{Ew}(k))}$ and $\mathsf{G_m}$ experimentally. Notice that, in the absence of *saturation*, i.e., as long as $\mathsf{m}(k) > 0$ holds, the plant is linear, and as a result, its open-loop transfer function is given by

$$\mathsf{P_m}(z) = \frac{\mathsf{M}(z)}{\mathsf{C_m}(z)} = \frac{\mathsf{G_m}\mathsf{G_u}}{(z-1)}. \tag{2.26}$$

Just as before, the controller defines a relationship between the error of the system, which is defined as $\epsilon_m(k) = \mathsf{m_s} - \mathsf{m}(k)$, and the change in estimated utilization, $\mathsf{c_m}(k)$. Thus, the difference equation for the controller is given as

$$\mathsf{c_m}(k) = \mathsf{K_{pm}}\epsilon_u(k), \tag{2.27}$$

where $\mathsf{K_{pm}}$ is a tunable parameter. The open-loop transfer function for (2.27) is given by

$$\mathsf{H}_m(z) = \mathsf{K_{pm}}. \tag{2.28}$$

Just as before, this is also a proportional controller (i.e., a P controller).

From (2.26) and (2.28), we can derive an open-loop transfer function for the plant and controller combined as

$$\mathsf{Q}_m(z) = \mathsf{H}_m(z)\mathsf{P_m}(z) = \frac{\mathsf{K_{pm}}\mathsf{G_m}\mathsf{G_u}}{(z-1)}, \tag{2.29}$$

and the closed-loop transfer function as

$$\mathsf{Y}_m(z) = \frac{\mathsf{Q}_m(z)}{1+\mathsf{Q}_m(z)} = \frac{\mathsf{K_{pm}}\mathsf{G_m}\mathsf{G_u}}{z-(1-\mathsf{K_{pm}}\mathsf{G_m}\mathsf{G_u})}. \tag{2.30}$$

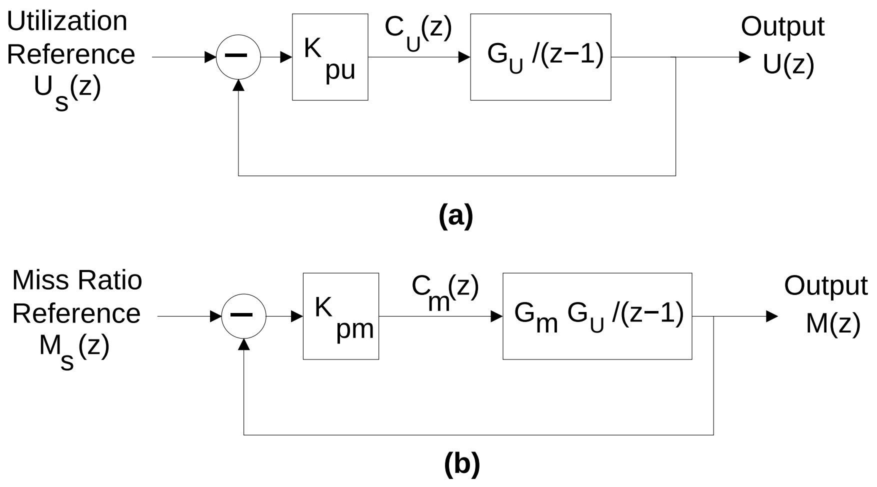


Figure 2.9: The feedback loops for the FCS framework. **(a)** The feedback loop for controlling the utilization. **(b)** The feedback loop for controlling the miss ratio.

Stability, steady-state error, and transient response. By using the analysis in Section 2.4.2, it is not difficult to show that the utilization feedback loop is stable iff

$$0 < \mathsf{K_{pu}} < \frac{2}{\mathsf{G_u}},$$

and the miss-ratio feedback loop is stable iff

$$0 < \mathsf{K_{pm}} < \frac{2}{\mathsf{G_m G_u}}.$$

Additionally, since it can be shown that both the systems described by (2.22) and (2.29) have a system type of one, the steady-state error for a step input in the reference input value is zero. (Lu et al. are not concerned with ramp inputs, although that could also be easily derived from the analysis presented in Section 2.4.2.)

Finally, since the systems described by (2.23) and (2.30) are first-order system, the utilization feedback loop does not overshoot its final value so long as

$$0 < \mathsf{K_{pu}} \leq \frac{1}{\mathsf{G_u}},$$

and the miss-ratio feedback loop does not overshoot its final value so long as

$$0 < \mathsf{K_{pm}} \leq \frac{1}{\mathsf{G_m G_u}}.$$

Additionally, the settling time (where time is measured by the number of sampling periods) for the utilization feedback loop is

$$\left\lceil \frac{-4}{ln\left(1 - \mathsf{K_{pu} G_u}\right)} \right\rceil,$$

and the settling time time for the miss-ratio feedback loop is

$$\left\lceil \frac{-4}{ln\left(1 - \mathsf{K_{pm} G_m G_u}\right)} \right\rceil.$$

Loop switching. As we previously discussed, one of the complications with controlling the miss ratio and the utilization is that it is possible for both of these variables to saturate. (Recall that the utilization saturates at 1 and the miss ratio saturates at 0%.) When one of these variable becomes saturated, it is no longer possible to use that variable to control the system in any meaningful way. To resolve this issue, Lu et al. switch between using the miss-ratio and utilization feedback loops. (Unfortunately, as we will discuss in Section 2.5.4, Lu et al. do not discuss how to handle the scenario when both the miss-ratio and utilization are saturated.) Specifically, the estimated utilization for sampling period $k+1$ is defined by

$$\mathsf{Ew}(k+1) = \mathsf{Ew}(k) + \mathsf{min}\left(\mathsf{c_u}(k), \mathsf{c_m}(k)\right)). \tag{2.31}$$

By combining the two controllers in such a fashion, it is possible to set a nominal desired utilization u_s at which deadlines should not be missed, and set an acceptable value of $\mathsf{m_s}$ that can be handled in overloaded scenarios.

Changing the system load. The QoS actuator changes the system load via a two-step process. First, the service levels are determined. Second, the service-level changes are enacted by leave/join reweighting. Since we have already discussed leave/join reweighting in Section 2.1, for the remainder of this section, we discuss the *highest-value-density first* (HVDF) algorithm that is used to determine the service levels of tasks. The *value-density* of the service level $k > 0$ of T_i is defined as

$$\frac{\mathsf{v}(T_i, k)\mathsf{p}(T_i, k)}{\mathsf{e}(T_i, k)}.$$

The HVDF algorithm determines the service levels for tasks as follows. First, the highest value-density is calculated for each task. Second, tasks are ordered by highest value-density service level from largest to smallest. Next, each task is assigned, in order, its highest value-density service level until the total estimated utilization of all tasks reaches a user-defined utilization threshold, which can be any value in the range $[0, 1]$.

Example. As an example, suppose that the utilization threshold is 0.51, and there are three tasks in the system each with three service levels and for any task T_i and service level $k > 0$, $\mathsf{p}(T_i, k) = 100$ (recall that for service level 0, $\mathsf{p}(T_i, 0) = 0$, $\mathsf{e}(T_i, 0) = 0$, and $\mathsf{v}(T_i, 0) = 0$). For T_1, $\mathsf{e}(T_1, 1) = 20$, $\mathsf{v}(T_1, 1) = 0.5$, $\mathsf{e}(T_1, 2) = 30$, and $\mathsf{v}(T_1, 2) = 0.6$. For T_2, $\mathsf{e}(T_2, 1) = 20$, $\mathsf{v}(T_2, 1) = 0.2$, $\mathsf{e}(T_2, 2) = 30$, and $\mathsf{v}(T_2, 2) = 0.5$. For T_3, $\mathsf{e}(T_3, 1) = 20$, $\mathsf{v}(T_3, 1) = 0.2$, $\mathsf{e}(T_3, 2) = 30$, and $\mathsf{v}(T_3, 2) = 0.6$. The service levels with the highest value-densities in this example are service level 1 of T_1 (a value density of 2.5), service level 2 of T_2 (a value density of $1.\bar{6}$), and service level 2 of T_3 (a value density of 2). Thus, according the HVDF, T_1 is first assigned service level 1. Next, T_3 is assigned service level 2. Finally, since no other service levels with a positive weight can be assigned without exceeding the desired estimated weight of 0.51, T_2 is assigned service level 0. □

2.5.3 Assumptions of the FCS Framework

Feedback systems are typically designed for specific scenarios. For this reason, most feedback algorithms are based on assumptions about the behavior of the system. In this section, we discuss the major assumptions made in the FCS framework.

Assumption 1. Each service level has an expected execution time that represents the "average-case" case. In Whisper such an assumption would require that there be an average signal-to-noise ratio for every microphone speaker pair. This is probably a valid assumption for a wide range of applications; however, it is not difficult to conceive of applications that are deployed in highly-variable environments for which this assumption does not hold.

Assumption 2. The value of $\mathsf{G_u}$ for each task can be determined experimentally. This is probably a valid assumption, although depending on the application $\mathsf{G_u}$ could be easily over-estimated.

Assumption 3. There exists some "vicinity" around the target miss-ratio for which the derivative of the relationship between the utilization and miss-ratio is a constant, i.e., the value $\mathsf{G_m}$ exists and is a constant. This assumption is probably more questionable than the previous two. Since depending on the types of tasks in the system, the relationship between the miss-ratio and utilization could vary widely.

Example (Figure 2.10). Consider the example in Figure 2.10, which depicts a one-processor system with 4 tasks: $T_1{:}(7,8)$; $T_2{:}(2,8)$; $T_3{:}(2,8)$; and $T_4{:}(2,8)$. In inset (a), T_1 has the highest priority. In inset (b), T_1 has the lowest priority. In Figure 2.10(a), since T_1^1 is scheduled first, T_2–T_4 all miss deadlines. In Figure 2.10(b), since T_1^1 is scheduled last, T_2–T_4 all make their deadlines. □

Notice that, even though these two schedules are both valid under EDF, the miss-ratio differs dramatically. It is worth mentioning that Lu et al. suggest that the value of $\mathsf{G_m}$ should be determined experimentally. Thus, while it is conceivable that for some applications it may be feasible to empirically determine the value of $\mathsf{G_m}$, such a value may not accurately reflect the typical behavior of the system.

2.5.4 Limitations of the FCS Framework

We conclude our discussion of the FCS Framework with a discussion of its limitations.

Limitation 1. The FCS framework adjusts tasks based on *only* the system-wide miss-ratio and utilization. As a result, if only a few tasks have actual utilizations that differ substantially

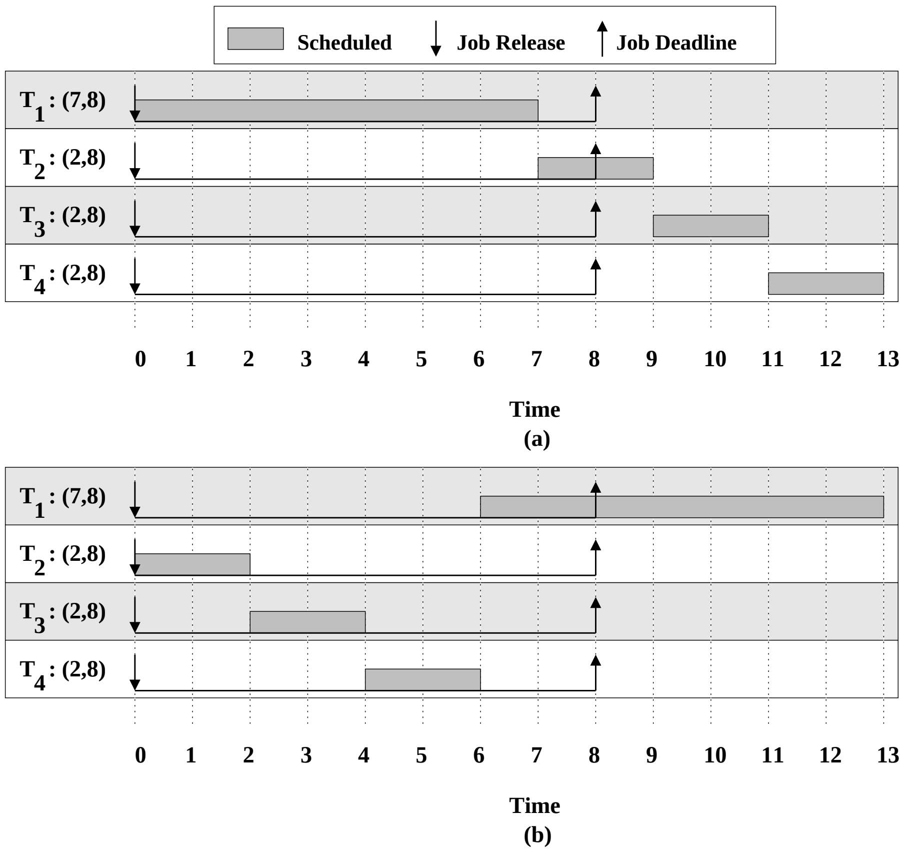


Figure 2.10: An example one of the FCS framework's assumptions. **(a)** T_1 has the highest scheduling priority. **(b)** T_1 has the lowest scheduling priority.

from their estimated utilizations, then the system is incapable of adjusting only those few tasks.

Limitation 2. Lu et al. state, without proof, that both the miss-ratio and utilization cannot be saturated at the same time in a given sampling period. While it is true that if the system is over-utilized, then jobs will eventually start missing their deadlines, they claim that in each sampling period the utilization and the miss-ratio are not both saturated. This stronger claim is not always true.

Example (Figure 2.11). Consider the example in Figure 2.11, which depicts a one-processor system with four tasks: T_1:$(3, 8)$; T_2:$(3, 8)$; T_3:$(3, 8)$; and T_4:$(3, 8)$. In this system, the miss-ratio and the utilization are monitored every three time units (as denoted by the dashed line). Thus, for the first two sampling periods, the miss-ratio is 0% and the utilization is one. As a

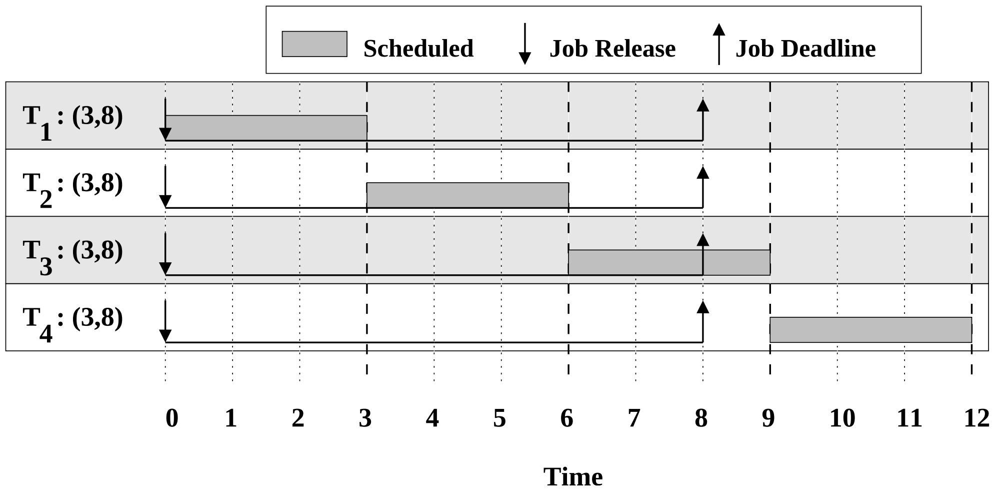


Figure 2.11: An example one of the FCS framework's limitations.

result, both variables are saturated. □

It is worthwhile to note that Lu et al. do not offer any guidelines for choosing the sampling period. However, even if we assume that the sampling periods are substantially larger than task periods (which would mitigate this limitation), it is not hard to construct example systems for which there exists at least one sampling period where both the miss-ratio and the utilization are saturated.

2.6 The Constant Bandwidth Server Feedback Scheduler

As mentioned in Section 2.5.4, one of the main limitations of the approach in (Lu et al., 2002) is that the system cannot identify whether an individual task has an actual execution time that deviates substantially from its estimated execution time. A uniprocessor feedback-controlled real-time scheduling algorithm that does not have this limitation was proposed in (Abeni et al., 2002). To accurately assign each task a weight, their algorithm monitors, *for each task*, the difference between the estimated and actual execution times of each job. Once the system has calculated a new estimated execution time for a future job, it adjusts the maximum fraction of the processor allocated to the task in order to reduce the number of deadline misses while ensuring that the task receives an accurate fraction of the processor.

2.6.1 Constant Bandwidth Server

Before describing Abeni et al.'s work in detail, we first review the *Constant Bandwidth Server* (CBS), first proposed by (Abeni and Buttazzo, 1998), which is the scheduling framework their system is built upon. Under CBS, each task is defined by a triple $(\mathsf{p}(T_i), \mathsf{band}(T_i), \mathsf{PR}(T_i))$, where $\mathsf{p}(T_i)$ denotes the period of T_i, $\mathsf{band}(T_i)$ is the *bandwidth* of T_i—which is the fraction of a processor allocated to T_i (this is similar to notion of a task weight)—and $\mathsf{PR}(T_i)$ is the *period of reservation*—in every $\mathsf{PR}(T_i)$ time units, T_i can be scheduled for up to $\mathsf{band}(T_i) \cdot \mathsf{PR}(T_i)$ time units. The value $\mathsf{band}(T_i) \cdot \mathsf{PR}(T_i)$ is called the *budget of* T_i and is denoted $\mathsf{budg}(T_i)$. At time $q \cdot \mathsf{PR}(T_i)$ (where $q \geq 1$ is an integer), T_i experiences a *budget renewal*. (Notice that a task's period defines when its jobs maybe released, but the period of reservation defines when the task's budget is renewed.) Tasks are scheduled on an *earliest-budget-renewal-time-first* basis. It is important to note that $\mathsf{PR}(T_i)$ can be defined as any value so long as $\mathsf{p}(T_i) = k \cdot \mathsf{PR}(T_i)$ for some integer value $k \geq 1$. One final note: if a job must execute for more than its alloted budget, then it will continue executing by using the next job's budget.

Example (Figure 2.12). Consider the example in Figure 2.12, which depicts a one-processor system scheduled by CBS with three tasks: T_1, which is defined by the triple $(4, 1/4, 4)$, T_2, which is defined by the triple $(8, 1/4, 4)$, and T_3, which is defined by the triple $(4, 1/2, 2)$. Also, T_3^1 requires one additional unit of execution beyond its budget. A budget renewal is denoted by a large down-arrow. Notice that T_3^1 uses one time unit of T_3^2's budget. As such, T_3^1 executes for three time units, even though T_3^1 is allocated only two time units over the range [0, 4). As a result, T_3^1's remaining execution time is scheduled using T_3^2's budget over the time range [4, 5). □

2.6.2 Feedback Framework

The objective of the CBS feedback scheduling algorithm is to maintain the smallest possible value of $\mathsf{band}(T_i)$ such that each job of T_i completes by its deadline. In order to satisfy this design object, each task has a feedback-control loop, as depicted in Figure 2.13, that monitors the "scheduling error" for each job. The *scheduling error* for the job T_i^j, denoted $\epsilon(T_i^j)$, is the

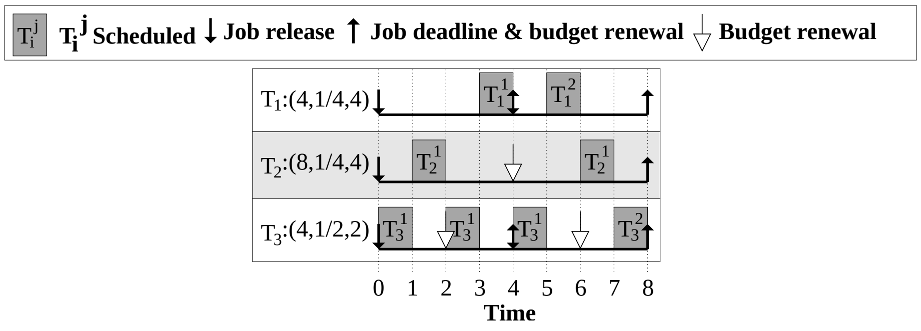


Figure 2.12: A one-processor example of the CBS with three tasks.

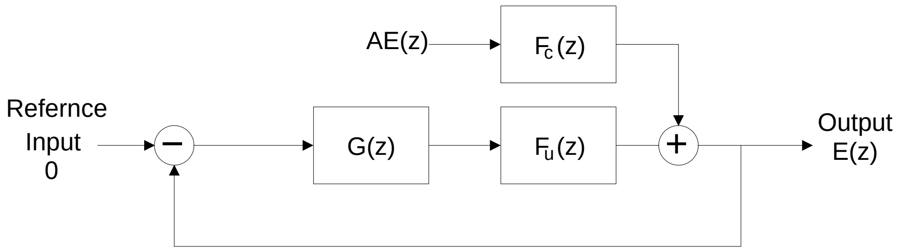


Figure 2.13: The adaptive reservation-based feedback design.

difference between its period, $\mathsf{p}(T_i)$, and the time that the job would finish if it were assigned to a processor with a speed of $\mathsf{band}(T_i)$. Specifically,

$$\epsilon(T_i^{j+1}) = \begin{cases} \epsilon(T_i^j) + \mathsf{Ae}(T_i^j) \cdot \mathsf{band}(T_i^j) - \mathsf{p}(T_i) & \epsilon(T_i^j) \geq 0 \\ \mathsf{Ae}(T_i^j) \cdot \mathsf{band}(T_i^j) - \mathsf{p}(T_i) & \epsilon(T_i^j) < 0 \end{cases}, \tag{2.32}$$

where $\mathsf{band}(T_i^j)$ is the bandwidth assigned to T_i when T_i^j is released. Recall that $\mathsf{Ae}(T_i^j)$ is the actual execution time of T_i^j. The reason why the term $\epsilon(T_i^j)$ is included in the calculation of $\epsilon(T_i^{j+1})$ when $\epsilon(T_i^j) \geq 0$ is because, in this case, T_i^j overran its budget. As a result, the first part of T_i^{j+1}'s budget is dedicated to finishing T_i^j. For example, in Figure 2.12, $\epsilon(T_3^1) = 1$, and as a result, one quantum of T_3^2's budget is spent completing T_3^1. Upon the completion of a job, T_i^j, the scheduling error is fed into a PI controller, which we will describe in Section 2.6.3. Notice that, in this system, unlike Lu et al.'s FCS framework, there is an exact linear relationship between the input (i.e., the actual execution time) and the output (i.e., the scheduling error). (This linear relationship could not be enacted under the FCS framework because of its use of the miss-ratio and system utilization.)

2.6.3 Stability

One complicating factor in this system is that the behavior of the plant, i.e., the scheduling error, changes dramatically depending on whether $\epsilon(T_i^j) < 0$. To resolve this issue, Abeni et al. assume that the equilibrium value of $\epsilon(T_i^j)$, denoted $\bar{\epsilon}(T_i)$, is far enough away from 0 that the system does not frequently switch between the two modes. By making such an assumption, the objective of the system is for $\epsilon(T_i^j) = \bar{\epsilon}(T_i)$ to hold. (In Section 2.6.4, we disucss the validity of this assumption.) After making such an assumption, Abeni et al. construct two different system equations based on whether $\bar{\epsilon}(T_i) < 0$. In the remainder of this section, we explain these equations.

Case 1: $\bar{\epsilon}(T_i) \geq 0$. We first consider the case where $\bar{\epsilon}(T_i) \geq 0$. In this case, assuming that the variation around the equilibrium quantities for scheduling error, execution time, and bandwidth are small, the formula for the plant can be written as

$$\Delta\epsilon(T_i^{j+1}) = \Delta\epsilon(T_i^j) + \frac{\mathsf{p}(T_i)}{\bar{\mathsf{u}}(T_i)}\Delta\mathsf{u}(T_i^j) + \bar{\mathsf{u}}(T_i)\Delta\mathsf{Ae}(T_i^j), \tag{2.33}$$

where $\Delta\epsilon(T_i^j) = \epsilon(T_i^j) - \bar{\epsilon}(T_i)$, $\mathsf{u}(T_i^j) = \frac{1}{\mathsf{band}(T_i^j)}$, $\bar{\mathsf{u}}(T_i)$ is the equilibrium value of $\mathsf{u}(T_i^j)$, $\Delta\mathsf{u}(T_i^j) = \mathsf{u}(T_i^j) - \bar{\mathsf{u}}(T_i)$, $\Delta\mathsf{Ae}(T_i^j) = \mathsf{Ae}(T_i^j) - \bar{\mathsf{Ae}}(T_i)$, and $\bar{\mathsf{Ae}}(T_i)$ is the equilibrium value of $\mathsf{Ae}(T_i^j)$. By taking the z-transform of (2.33), we get

$$\mathsf{E}(z) = \mathsf{F}_c(z)\mathsf{AE}(z) + \mathsf{F}_u(z)\mathsf{U}(z), \tag{2.34}$$

where $\mathsf{E}(z)$ is the z-transform of $\epsilon(T_i^j)$, $\mathsf{AE}(z)$ is the z-transform of $\Delta\mathsf{Ae}(T_i^j)$, $\mathsf{U}(z)$ is the z-transform of $\Delta\mathsf{u}(T_i^j)$, $\mathsf{F}_c(z) = \frac{\bar{\mathsf{u}}(T_i)}{z-1}$, and $\mathsf{F}_u(z) = \frac{\mathsf{p}(T_i)}{\bar{\mathsf{u}}(T_i)(z-1)}$.

The relationship between the the value $\Delta\epsilon(T_i^j)$ and $\Delta\mathsf{u}(T_i^j)$ is specified as a PI controller, which is given as

$$\Delta\mathsf{u}(T_i^j) = -a \cdot \Delta\epsilon(T_i^j) + b\sum_{q=1}^{j-1}(-\Delta\epsilon(T_i^q)), \tag{2.35}$$

where both a and b are constants determined by the system designer. The z-transform of

(2.35) is given as

$$\frac{\mathsf{U}(z)}{-\mathsf{E}(z)} = \mathsf{G}(z) = \frac{\alpha z + \beta}{z-1}, \tag{2.36}$$

where $\alpha = a$ and $\beta = b - a$. By substituting (2.36) into (2.34), we get the closed-loop transfer function, which is defined as

$$\mathsf{E}(z) = \frac{\mathsf{F}_c(z)}{1 + \mathsf{G}(z)\mathsf{F}_u(z)}\mathsf{AE}(z), \tag{2.37}$$

By expanding (2.37), we get

$$\mathsf{E}(z) = \frac{\bar{\mathsf{u}}(T_i)(z-1)}{z^2 + \left(\frac{\mathsf{p}(T_i)}{\bar{\mathsf{u}}(T_i)}\alpha - 2\right)z + \beta\frac{\mathsf{p}(T_i)}{\bar{\mathsf{u}}(T_i)} + 1}\mathsf{AE}(z). \tag{2.38}$$

Notice that the closed-loop poles of (2.38) are the values of $\mathcal{P}_1$ and $\mathcal{P}_2$ such that

$$z^2 + \left(\frac{\mathsf{p}(T_i)}{\bar{\mathsf{u}}(T_i)}\alpha - 2\right)z + \beta\frac{\mathsf{p}(T_i)}{\bar{\mathsf{u}}(T_i)} + 1 = z^2 - \left(\mathcal{P}_1 + \mathcal{P}_2\right)z + \mathcal{P}_1\mathcal{P}_2.$$

Thus, in terms of α and β, the closed-loop poles for (2.37) are the values of $\mathcal{P}_1$ and $\mathcal{P}_2$ such that

$$\alpha = \tfrac{\bar{\mathsf{u}}(T_i)(2-\mathcal{P}_1-\mathcal{P}_2)}{\mathsf{p}(T_i)}, \quad \beta = \tfrac{\bar{\mathsf{u}}(T_i)(\mathcal{P}_1\mathcal{P}_2)}{\mathsf{p}(T_i)}. \tag{2.39}$$

Since (2.37) is a second-order system, the system is stable if the distance of both poles from the origin is less than one. Thus, for a given α and β, the system is stable if the values of $\mathcal{P}_1$ and $\mathcal{P}_2$ that satisfy (2.39) also satisfy.

$$||\mathcal{P}_1|| < 1, \quad ||\mathcal{P}_2|| < 1.$$

Case 2: $\bar{\epsilon}(T_i) < 0$. From the above result, it is easy to show (by redoing the calculations) that if $\bar{\epsilon}(T_i) < 0$, then for a given value of α and β, the system is stable if the values of $\mathcal{P}_1$ and $\mathcal{P}_2$ that satisfy

$$\alpha = \tfrac{\bar{\mathsf{u}}(T_i)(1-\mathcal{P}_1-\mathcal{P}_2)}{\mathsf{p}(T_i)}, \quad \beta = \tfrac{\bar{\mathsf{u}}(T_i)(\mathcal{P}_1\mathcal{P}_2)}{\mathsf{p}(T_i)}$$

also satisfy

$$||\mathcal{P}_1|| < 1, \quad ||\mathcal{P}_2|| < 1.$$

2.6.4 Scheduling Error Assumption

The most important assumption made by Abeni et al. is that the the scheduling error has an average value and there is little variance around this value. For applications that are not deployed in highly variable environments, this is probably a reasonable assumption. For highly variable scenarios, this assumption could be problematic because if the scheduling error rapidly switches between positive and negative values, then their analysis is invalidated.

2.6.5 Limitations

The major limitations of this work stem from the fact that its only objective is to determine an accurate bandwidth value for each task. As a result, the system is not capable of minimizing deadline misses when the system is overloaded. Specifically, if the system is overloaded, i.e., $\sum_{T_i \in \tau} \mathsf{band}(T_i) > 1$, then the system will scale down the bandwidth of all tasks such that $\sum_{T_i \in \tau} \mathsf{band}(T_i) \leq 1$. As a result, in an overloaded scenario, it is possible that every task will miss its deadlines regardless of relative importance. Moreover, because the only manipulated variable is the bandwidth, it is not possible to mitigate an overloaded scenario by increasing either the period or reservation period of a task (which would have the impact of reducing the task's QoS to prevent deadline misses).

2.7 Conclusion

In this chapter, we reviewed several different techniques for changing the weight of a task (i.e., leave/join reweighting, RBED scheduling, and EEVDF scheduling). In addition, we discussed the basics of feedback control theory that will be used in this dissertation. Finally, we concluded this chapter with a discussion of two uniprocessor feedback-based approaches for adapting to external stimuli (i.e., the FCS framework and the feedback-controlled reservation based scheduling algorithm). In the remainder of this dissertation, we will extend the work reviewed in this chapter to function under a multiprocessor environment.

CHAPTER 3

GEDF and NP-GEDF[*]

In this chapter, we present the rules for reweighting tasks for both the GEDF and NP-GEDF scheduling algorithms. Before doing so so, we first define the "adaptable sporadic task model" as well as three theoretical scheduling algorithms that will be useful for describing these reweighting rules. (To improve readability, all of the terms in this chapter are summarized in Table 3.1.)

3.1 Adaptable Sporadic Task System

An *adaptable sporadic task system* is an extension of a sporadic task system, where the weight of each task T_i is a function of time t, denoted $\mathsf{wt}(T_i,\, t)$, and its execution time can vary with each job T_i^j, denoted $\mathsf{e}(T_i^j)$. (The behavior of an adaptable sporadic task is defined by an execution time and weight instead of an execution time and period pair—the two approaches are equivalent but the former results in less complex reweighting rules.) For simplicity, if every job of a task T_i has the same execution time, then we will denote this time by $\mathsf{e}(T_i^j)$, and if the weight of task T_i is constant, then we denote its weight as $\mathsf{wt}(T_i)$.

For adaptable sporadic tasks, the *absolute deadline* of a job T_i^j, denoted $\mathsf{d}(T_i^j)$, is defined as

$$\mathsf{d}(T_i^j) = \mathsf{r}(T_i^j) + \mathsf{e}(T_i^j)/\mathsf{wt}(T_i,\, \mathsf{r}(T_i^j)).$$

(Recall that $\mathsf{r}(T_i^j)$ is the release time of T_i^j.) In the absence of reweighting, consecutive job releases ($\mathsf{r}(T_i^j)$ and $\mathsf{r}(T_i^{j+1})$) of a task T_i must be separated by at least $\mathsf{e}(T_i^j)/\ \mathsf{wt}(T_i,\, \mathsf{r}(T_i^j))$

[*] Contents of this chapter previously appeared in preliminary form in the following paper:
Block, A., Anderson J., and Devi, U. (2008b). Task reweighting under global scheduling on multiprocessors. *Real-Time Systems, Special Issue on Selected Papers from the 18th Euromicro Conference on Real-Time Systems*, 39:123–167.

Notation	Definition
$\mathsf{wt}(T_i,\, t)$	Weight of T_i at time t.
$\mathsf{wt}(T_i)$	Weight of the T_i that does not change its weight.
$\mathsf{Swt}(T_i, t)$	Scheduling weight of T_i at time t.
$\mathsf{e}(T_i^j)$	Worst-case execution time of job T_i^j.
$\mathsf{e}(T_i)$	Worst-case execution time for all jobs of T_i.
$\mathsf{e}_{\mathsf{max}}(T_i)$	Maximal worst-case execution time of all jobs of T_i.
$\mathsf{Ae}(T_i^j)$	Actual execution time of job T_i^j.
$\mathsf{r}(T_i^j)$	Release time of T_i^j.
$\mathsf{d}(T_i^j)$	Deadline of T_i^j.
$\theta(T_i^j)$	IS separation between T_i^{j-1} and T_i^j.
SW	Non-clairvoyant scheduling-weight scheduling algorithm. While a task is active, this algorithm allocates the task its *scheduling weight* at each instant.
$\mathcal{S}W$	SW schedule of a task system τ.
CSW	Clairvoyant scheduling-weight scheduling algorithm. While a task is active *and* the allocation to its active job is less than its actual execution time, this algorithm allocates the task its *scheduling weight* at each instant.
$\mathcal{C}SW$	CSW schedule of task system τ.
IDEAL	Ideal scheduling algorithm. While a task is active, this algorithm allocates each task its *weight* at each instant.
$\mathcal{I}$	IDEAL schedule of task system τ.
$\mathcal{S}$	Actual (i.e., GEDF or NP-GEDF) schedule of task system τ.
$\mathsf{A}(\mathcal{B},\, T_i^j,\, t_1,\, t_2)$	Allocation to T_i^j in the schedule $\mathcal{B}$ over $[t_1,\, t_2)$.
$\mathsf{A}(\mathcal{B},\, T_i,\, t_1,\, t_2)$	Allocation to T_i in the schedule $\mathcal{B}$ over $[t_1,\, t_2)$.
$\mathsf{dev}(T_i^j,\, t)$	Deviance of T_i^j: $\mathsf{A}(\mathcal{S}W,\, T_i^j,\, 0,\, t) - \mathsf{A}(\mathcal{S},\, T_i^j,\, 0,\, t)$.
$\mathsf{drift}(T_i,\, t)$	Drift of T_i: $\mathsf{A}(\mathcal{I},\, T_i,\, 0,\, t) - \mathsf{A}(\mathcal{C}SW,\, T_i,\, 0,\, t)$.
Ow	Scheduling weight before a reweighting event.
Nw	New weight after a reweighting event.
$\mathsf{REM}(T_i^j, t)$	Remaining execution time of T_i^j at t. $\mathsf{e}(T_i^j) - \mathsf{A}(\mathcal{S},\, T_i^j,\, 0,\, t)$.
$\mathsf{nextE}(T_i^j, t)$	If $\mathsf{REM}(T_i^j, t) > 0$, then $\mathsf{nextE}(T_i^j, t) = \mathsf{REM}(T_i^j, t)$; else, $\mathsf{nextE}(T_i^j, t) = \mathsf{e}(T_i^{j+1})$.

Table 3.1: Summary of notation used in this chapter.

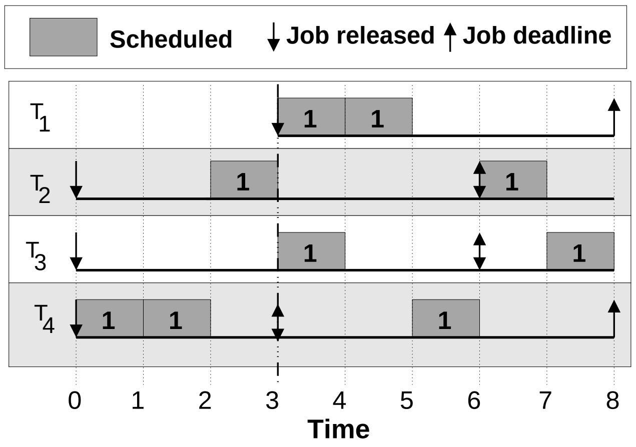


Figure 3.1: A one-processor example of an adaptable sporadic task system.

time units.

Example (Figure 3.1). Consider the example in Figure 3.1, which depicts a one-processor system with four tasks: T_1, which joins the system at time 3 with $\mathsf{wt}(T_1) = 2/5$ and $\mathsf{e}(T_1) = 2$; T_2 and T_3, both of which have a weight of 1/6 and an execution time of 1; and T_4, which has, $\mathsf{e}(T_4^1) = 2$, $\mathsf{e}(T_4^2) = 1$, and an initial weight of 2/3 that decreases to 1/5 at time 3. Notice that since $\mathsf{e}(T_4^1) = 2$ and $\mathsf{wt}(T_4, \mathsf{r}(T_4^1)) = 2/3$, both $\mathsf{d}(T_4^1) = 3$ and $\mathsf{r}(T_4^2) = 3$. Also, since $\mathsf{e}(T_4^2) = 1$ and $\mathsf{wt}(T_4, \mathsf{r}(T_4^2)) = 1/5$, $\mathsf{d}(T_4^2) = 8$. □

A task T_i *changes weight* or *reweights* at time t if $\mathsf{wt}(T_i, t - \epsilon) \neq \mathsf{wt}(T_i, t)$ where $\epsilon \to 0^+$. For example, in the system depicted in Figure 3.1, T_4 reweights at time 3.

We now explain some of the issues involved in processing such a reweighting event. If a task T_i changes weight at a time t_c between the release and the deadline of some job T_i^j, then the following two actions *may* occur:

- The execution time of T_i^j *may* be reduced to the amount of time for which T_i^j has executed prior to t_c, and the execution time of T_i^{j+1} *may* be redefined to be the amount of time "lost" by reducing the execution time of T_i^j.

- $\mathsf{r}(T_i^{j+1})$ *may* be redefined to be less than $\mathsf{r}(T_i^j)+ \mathsf{e}(T_i^j)/\mathsf{wt}(T_i, \mathsf{r}(T_i^j))$. In this case, since $\mathsf{d}(T_i^j) = \mathsf{r}(T_i^j) + \mathsf{e}(T_i^j)/\mathsf{wt}(T_i, \mathsf{r}(T_i^j))$, jobs T_i^j and T_i^{j+1} will "overlap." (For a standard sporadic task, every job's deadline is at or before its successor's release.)

The reweighting rules we present in Section 3.4 state under what conditions the above actions occur and by how much before $\mathsf{r}(T_i^j)+\mathsf{e}(T_i^j)/\mathsf{wt}(T_i,\ \mathsf{r}(T_i^j))$ the job T_i^{j+1} can be released. When the execution time of a job is reduced at a time t, then we say that it is "halted." Specifically, if a job T_i^j is *halted* at time t, then $\mathsf{Ae}(T_i^j)$ is set to $\mathsf{A}(\mathcal{S},\ T_i^j,\ 0,\ t)$.

Initiate and enact. As mentioned in Section 1.3.1, when a task reweights, there can be a difference between when it "initiates" the change and when the change is "enacted." The time at which the change is *initiated* is a user-defined time; the time at which the change is *enacted* is dictated by a set of conditions described in Section 3.4. We use the *scheduling weight of a task T_i at time t*, denoted $\mathsf{Swt}(T_i, t)$, to represent the "last enacted weight of T_i." Formally, $\mathsf{Swt}(T_i, t)$ equals $\mathsf{wt}(T_i,\ u)$, where u is the last time at or before t that a weight change was enacted for T_i (assuming an initial weight change occurred when T_i joined the system). It is important to note that *for adaptable sporadic tasks, we compute task deadlines and releases using scheduling weights.* Hence, we have the following formulas:

$$\begin{aligned}
\mathsf{r}(T_i^1) &= \theta(T_i^1) \\
\mathsf{d}(T_i^j) &= \mathsf{r}(T_i^j) + \mathsf{e}(T_i^j)/\mathsf{Swt}(T_i, \mathsf{r}(T_i^j)) \\
\mathsf{r}(T_i^{j+1}) &= \mathsf{d}(T_i^j) + \theta(T_i^{j+1}),
\end{aligned}$$

where $\theta(T_i^j) \geq 0$. The third equation only applies in the absence of reweighting events, which may cause release times to be redefined.

Because the reweighting rules may cause $\mathsf{r}(T_i^{j+1}) < \mathsf{d}(T_i^j)$, we must slightly modify the definition of "window," "active," and "inactive" presented in Section 1.2.

Definition 3.1 (Window, Active, and Inactive). If T_i^j is a job in the adaptable sporadic task system, T, then the *window* of T_i^j defined as the range $[\mathsf{r}(T_i^j),\ \mathsf{min}(\mathsf{d}(T_i^j), \mathsf{r}(T_i^{j+1})))$. Furthermore, job T_i^j is *active at time* t iff t is in T_i^j's window (i.e., $t \in [\mathsf{r}(T_i^j),\ \mathsf{min}(\mathsf{d}(T_i^j), \mathsf{r}(T_i^{j+1})))$), and *inactive* otherwise.

3.2 The SW Scheduling Algorithm and Deviance

The *scheduling weight* (SW) scheduling algorithm is a theoretical scheduling algorithm that is used to determine which reweighting rule to apply as well to prove tardiness bounds. Under the SW scheduling algorithm, at each instant t, each active job T_i^j in τ is allocated a share equal to its scheduling weight $\mathsf{Swt}(T_i, t)$. Hence, if a job T_i^j is active over the range $[t_1, t_2)$, then over this range, T_i^j is allocated $\int_{t_1}^{t_2} \mathsf{Swt}(T_i, u)du$ time. (If a job is inactive, then it receives no allocations in SW.) Throughout this dissertation we use $\mathcal{SW}$ to denote the SW schedule of a task system τ.

Example (Figure 3.2). Consider the example in Figure 3.2, which depict a one-processor system with four tasks: T_1, which has $\mathsf{e}(T_1) = 1$ and $\mathsf{wt}(T_1) = 1/3$; T_2, which has $\mathsf{e}(T_2) = 1$ and $\mathsf{wt}(T_2) = 1/6$; T_3, which has $\mathsf{e}(T_3) = 2$ and $\mathsf{wt}(T_3) = 1/4$ and leaves at time 8; and T_4, which has $\mathsf{e}(T_4) = 4$ and an initial weight of 1/4 and initiates and enacts a weight increase to 1/2 at time 8. (As we will discuss in Section 3.4, the reweighting rules stop T_4^1 from receiving more than one unit of allocation and cause T_4^2 to be released with the remaining three units of execution at time 8.) Inset (a) depicts the GEDF schedule. Inset (b) depicts the SW schedule. Notice that, T_4 initiates and enacts a weight increase from 1/4 to 1/2 at time 8. Inset (c) depicts the allocations to T_4 in the GEDF and SW. Hence, before time 8, in the SW schedule, T_4 receives 1/4 of the processor at each instant, and after time 8, T_4 receives 1/2 of the processor at each instant. □

Example (Figure 3.3). Consider the example in Figure 3.3, which depicts a one-processor system with four tasks (where the execution time of each job is one): T_1, which joins the system at time 2 with $\mathsf{wt}(T_1) = 1/2$; T_2 and T_3, both of which have a weight of 1/6; and T_4, which has an initial weight of 1/2 and initiates a weight decrease to 1/6 at time 1 that is enacted at time 2. (As we will discuss in Section 3.4, the reweighting rules do not allow T_2 to decrease its weight immediately.) Inset (a) depicts the GEDF schedule. Inset (b) depicts the SW schedule. Inset (c) depicts the allocations to T_4 by the GEDF and SW scheduling algorithms. Notice that T_4 initiates a weight decrease at time 1 from 1/2 to 1/6 that is enacted at time 2. As a result, in $\mathcal{SW}$ over the range [1, 2), T_4 receives 1/2 of the processor

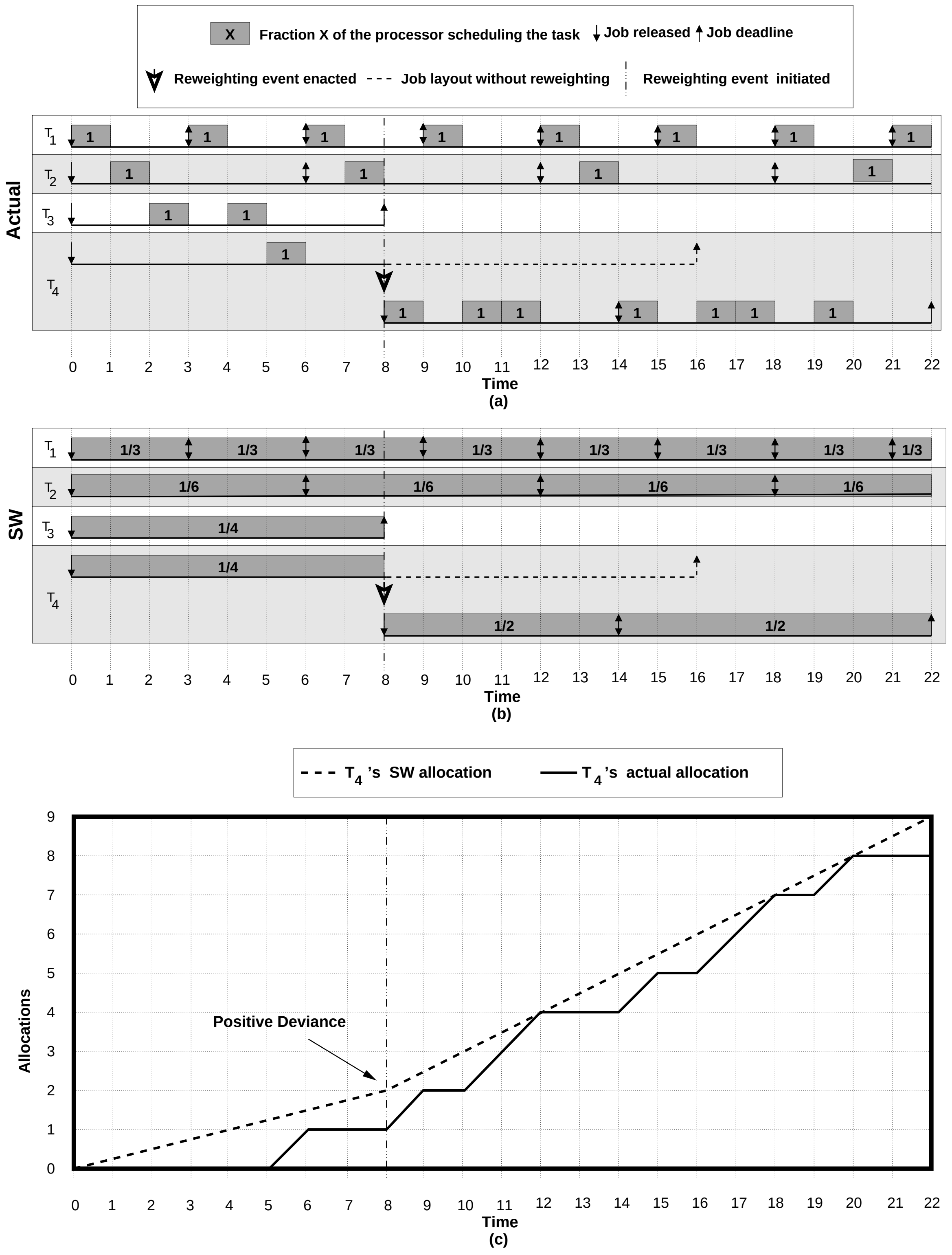


Figure 3.2: A one-processor example of a task that increases its weight. **(a)** The GEDF schedule. **(b)** The SW schedule. **(c)** The allocations to T_4 in the GEDF and schedules.

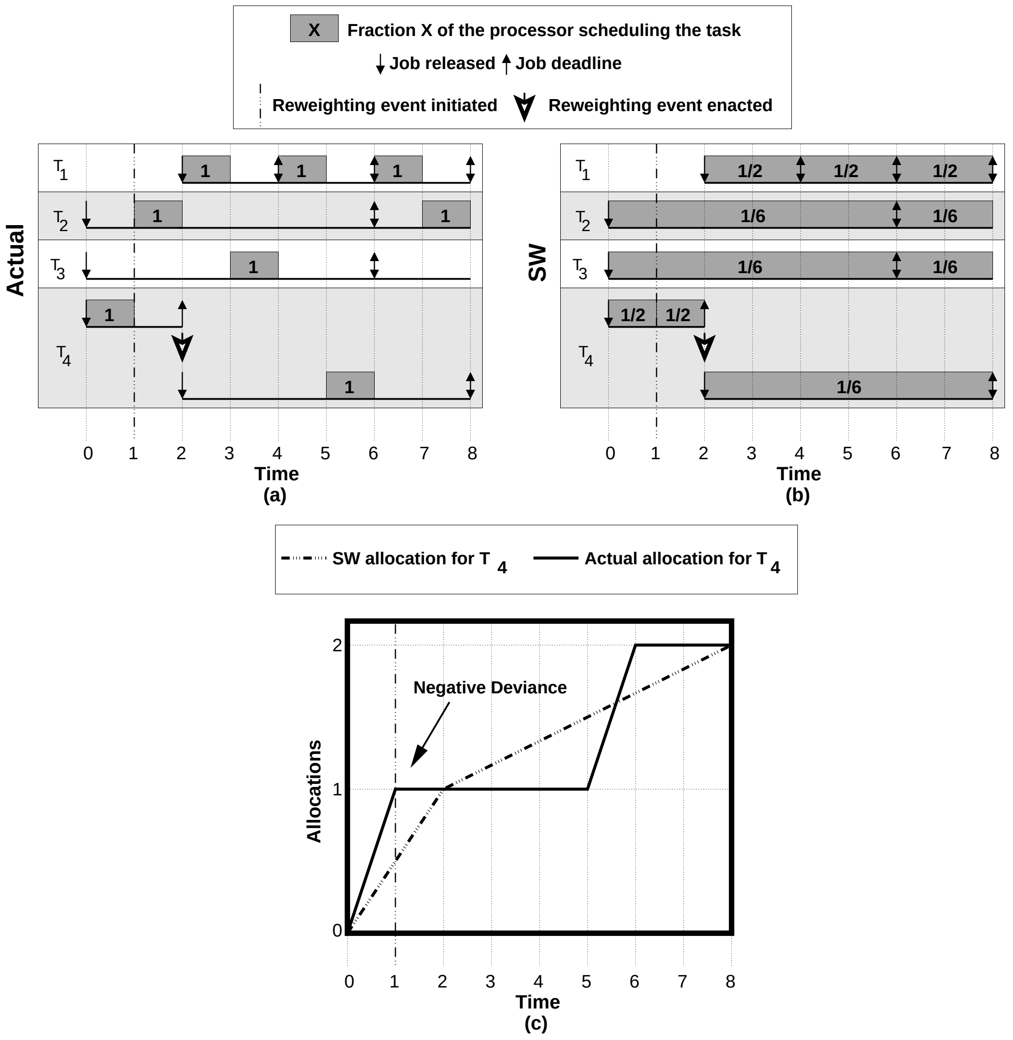


Figure 3.3: A one-processor example of a task that decreases its weight. **(a)** The GEDF schedule. **(b)** The SW schedule. **(c)** The allocations to T_4 in the GEDF and SW schedules.

at each instant, even though over $t \in [1, 2)$, $\mathsf{wt}(T_4, t) = 1/6$. $\square$

The *deviance of job T_i^j of task T_i at time t* is defined as $\mathsf{dev}(T_i^j, t) = \mathsf{A}(\mathcal{S}W, T_i^j, 0, t) - \mathsf{A}(\mathcal{S}, T_i^j, 0, t)$, where $\mathcal{S}$ is the actual schedule. The deviance of a job T_i^j represents the difference between T_i^j's actual and SW allocations up to time t. If the deviance is negative, then the job has received a greater allocation in the actual schedule, and if the deviance is positive, then the job has received a greater allocation in the SW schedule. For example, in the system depicted in Figure 3.2, at time 8, T_4^1's deviance is positive since it has been allocated more capacity in the SW schedule than in the actual schedule. On the other hand, in the system depicted in Figure 3.3, T_4^1 has negative deviance at time 1 because it has been allocated more capacity in the actual schedule than in the SW schedule. Whether the deviance of a task is positive or negative will determine which reweighting rule can be applied.

3.3 Modifications

In the adaptable sporadic task model, as presented in this chapter, the desired and guaranteed weight of each task is the same. Additionally, the fundamental unit for scheduling is a job. Both of these assumptions do not hold when scheduling tasks under PEDF or Pfair-based algorithms. Specifically, as we will discuss in Chapter 4, under our adaptive PEDF scheduling algorithm, a task's guaranteed and desired weight may differ. Also, as we will discuss in Chapter 5, under Pfair-based scheduling algorithms, the fundamental unit of scheduling is a subtask, not a job. As a result, the adaptable sporadic task model and theoretical scheduling algorithms presented here will be slightly modified in subsequent chapters to accommodate these differences.

3.4 Task Reweighting

Having defined the adaptable sporadic task model and its associated theoretical scheduling algorithms, we now define the reweighting rules for the GEDF and $\mathsf{NP\text{-}GEDF}$ scheduling algorithms and prove their tardiness and drift bounds.

We begin our discussion of reweighting under GEDF and $\mathsf{NP\text{-}GEDF}$ by defining the reweighting rules for GEDF. Then, in Section 3.4.2, we explain how these rules can be modified for $\mathsf{NP\text{-}GEDF}$. Before continuing, we introduce one assumption that we make for simplicity: we assume that the actual execution time for any job is equal to its specified execution time, *unless* a task reweights when it has an active job. Then *and only then* can the actual execution time of a job be less than its execution time.[1] In this scenario, the actual execution time of the job is determined by the rules we present shortly.

3.4.1 Reweighting Under GEDF

Let τ be a task system in which some task T_i initiates a weight change to weight Nw at time t_c. Let Ow be the last scheduling weight of T_i before the change is initiated at t_c. Let $\mathcal{S}$ be the m-processor GEDF schedule of τ. Let T_i^j be last-released job of T_i before t_c. If T_i^j does not exist or T_i^j is inactive at t_c before the reweighting event is initiated (i.e., $t_c \geq \mathsf{d}(T_i^j)$), then the weight change is immediately enacted, and future jobs of T_i are released with the new weight. In the following rules, we consider the remaining possibility, i.e., T_i^j exists and is active at t_c. (Notice that if $t_c = \mathsf{d}(T_i^k) = \mathsf{r}(T_i^{k+1})$, then T_i^k is the last-released job of T_i before t_c, and it is not active at t_c. Therefore, the change is immediately enacted and T_i^{k+1} is released with the new weight.)

Let $\mathsf{REM}(T_i^j, t_c) = \mathsf{e}(T_i^j) - \mathsf{A}(\mathcal{S}, T_i^j, 0, t_c)$. Note that $\mathsf{REM}(T_i^j, t_c)$ denotes the actual remaining computation in T_i's current job. Let $\mathsf{nextE}(T_i^j, t_c)$ equal $\mathsf{REM}(T_i^j, t_c)$, if $\mathsf{REM}(T_i^j, t_c) > 0$; otherwise, if $\mathsf{REM}(T_i^j, t_c) = 0$, then let $\mathsf{nextE}(T_i^j, t_c)$ equal the value of $\mathsf{e}(T_i^{j+1})$ had the weight-change event not occurred. Since $\mathsf{nextE}(T_i^j, t_c)$ is only used to determine the execution time of the next job released, it can be calculated at time $\mathsf{r}(T_i^{j+1})$. (Notice that, if T_i has no next job T_i^{j+1} to release at the time specified in the rules below, then $\mathsf{nextE}(T_i^j, t_c) = 0$. In this case, the rules are applied as stated, except that T_i^{j+1} is not released.)

As was mentioned earlier, the choice of which rule to apply depends on whether deviance is positive or negative. If positive, then we say that T_i is *positive-changeable at time t_c*

[1]Since reweighting events may modify the actual execution time of a job, removing this assumption would entail having notation to distinguish between redefined execution times as the result of reweighting and redefined execution times that occur due to a task executing for less than its specified execution time.

from weight Ow *to* Nw; otherwise T_i is *negative-changeable at time* t_c *from weight* Ow *to* Nw. Because T_i initiates its weight change at time t_c, $\mathsf{wt}(T_i,\, t_c) = \mathsf{Nw}$ holds; however, T_i's scheduling weight does not change until the weight change has been *enacted*, as specified in the rules below. Note that, if t_c occurs between the initiation and enaction of a previous reweighting event of T_i, then the previous event is *canceled*, i.e., treated as if it had not occurred. As discussed later, any "error" associated with canceling a reweighting event like this is accounted for when determining drift (formally defined in Section 3.5.3).

Rule P: If T_i is positive-changeable at time t_c from weight Ow to Nw, then one of two actions is taken: **(i)** if $\mathsf{d}(T_i^j) - t_c > \mathsf{REM}(T_i^j, t_c)/\mathsf{Nw}$, then immediately, T_i^j is halted, the weight change is enacted, a new job with an execution time of $\mathsf{nextE}(T_i^j, t_c)$ is released (if $\mathsf{nextE}(T_i^j, t_c) > 0$), and T_i^j becomes inactive; **(ii)** otherwise, at time $\mathsf{d}(T_i^j)$, the weight change is enacted, i.e., the scheduling weight of T_i does not change until the end of its current job.

Rule N: If T_i is negative-changeable at time t_c from weight Ow to Nw, then one of two actions is taken: **(i)** if $\mathsf{Nw} > \mathsf{Ow}$, then immediately, T_i^j is halted and its weight change is enacted, and at time t_r, a new job with an execution time of $\mathsf{nextE}(T_i^j, t_c)$ is released (if $\mathsf{nextE}(T_i^j, t_c) > 0$) and T_i^j becomes inactive, where t_r is the smallest time at or after t_c such that $\mathsf{dev}(T_i^j,\, t_r) = 0$ holds; **(ii)** otherwise, at time t_e, the weight change is enacted, a new job with an execution time of $\mathsf{nextE}(T_i^j, t_c)$ is released (if $\mathsf{nextE}(T_i^j, t_c) > 0$), and T_i^j becomes inactive, where $t_e = \mathsf{min}(t_r, \mathsf{d}(T_i^j))$, and t_r is smallest time at or after t_c such that $\mathsf{dev}(T_i^j,\, t_r) = 0$ holds.

Intuitively, Rule P changes a task's weight by halting its current job and issuing a new job with an execution time of $\mathsf{nextE}(T_i^j, t_c)$ with the new weight if doing so would improve its deadline.

Example (Figure 3.4). Consider the example in Figure 3.4, which depicts a one-processor system with four tasks (where the execution time of each job is one): T_1, which has $\mathsf{wt}(T_1) = 1/2$ and leaves at time at time 2; T_2 and T_3, both of which have a weight of $1/6$; and T_4, which has an initial weight of $1/6$ that increases to $4/6$ at time 2. In this system, T_4 initially has the

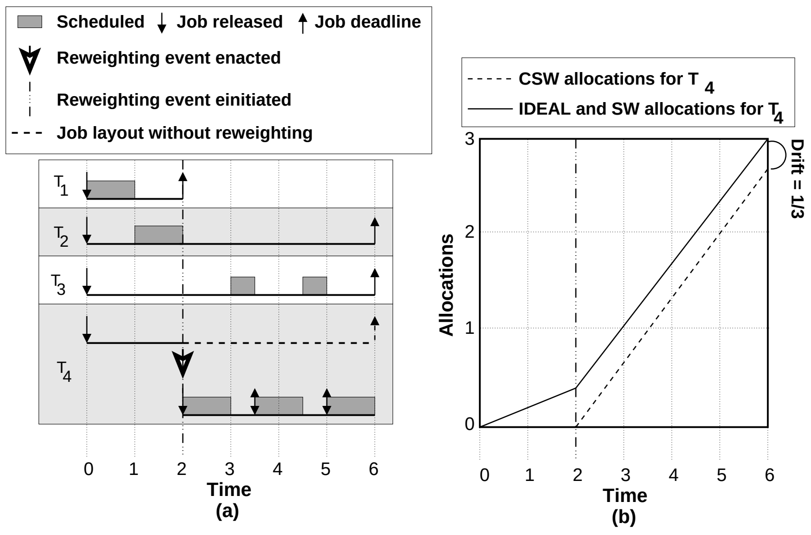


Figure 3.4: A one-processor example of reweighting via Case (i) of Rule P under GEDF. **(a)** The GEDF schedule. **(b)** T_4's allocations in the IDEAL, CSW, and SW schedules.

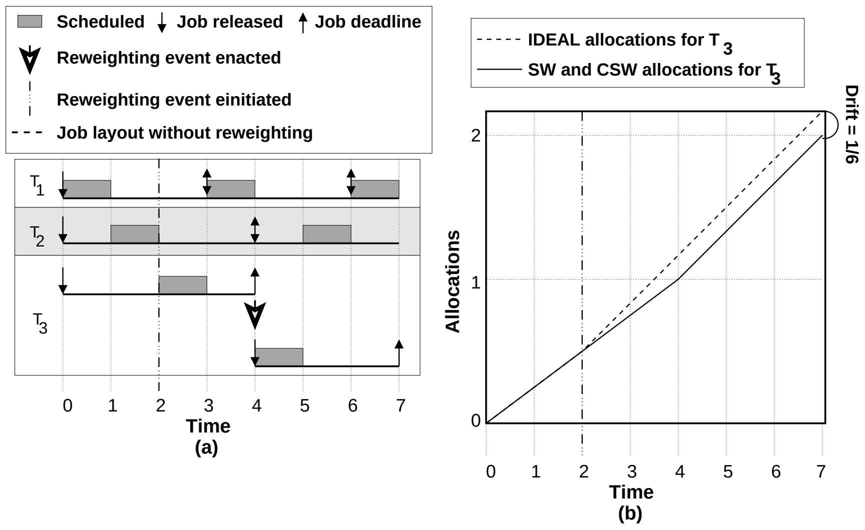


Figure 3.5: A one-processor example of reweighting via Case (ii) of Rule P under GEDF. **(a)** The GEDF schedule. **(b)** T_3's allocations in the IDEAL, CSW, and SW schedules.

lowest scheduling priority (there is a deadline tie). Inset (a) depicts the GEDF schedule. Inset (b) depicts T_4's allocations in the SW schedule and also in the other schedules, the IDEAL and CSW schedules, which are formally defined later in Section 3.5.2. Since T_4 is not scheduled by time 2 and because $\mathsf{d}(T_4^1) - t_c > \mathsf{REM}(T_4^1, t_c)/\mathsf{Nw}$, i.e., $\mathsf{d}(T_4^1) - 2 > 0/(4/6)$, it has positive deviance and changes its weight via Case (i) of Rule P. This, in turn, causes T_4^1 to be halted, T_4^2 to be released at time 2 with a deadline of 7/2, and T_4's drift to become 2/6. Note that halting T_4's current job and issuing a new job with an execution time of one improves T_4's scheduling priority, i.e., $\mathsf{d}(T_4^1) = 6 > \frac{7}{2} = \mathsf{d}(T_4^2)$. □

Example (Figure 3.5). Consider the example in Figure 3.5, which depicts a one-processor system with three tasks (where the execution time of each job is one): T_1, which has $\mathsf{wt}(T_1) = 1/3$; T_2, which has $\mathsf{wt}(T_2) = 1/4$; and T_3, which has an initial weight of 1/4 that increases to 1/3 at time 2. Inset (a) depicts the GEDF schedule. Inset (b) depicts T_3's allocations in the IDEAL, CSW, and SW schedules. (Again, the IDEAL and CSW schedules are formally defined later in Section 3.5.2.) Since T_3^1 has not been scheduled by time 2 its deviance is positive; furthermore, since $\mathsf{d}(T_3^1) - 2 < \mathsf{REM}(T_3^1, 2)/(1/3)$, T_1 enacts its weight change via Case (ii) of Rule P. Notice that if T_3^1 had been halted at time 2 and released a new job of weight 1/3, the deadline of this new job would equal time 5 (since $5 = 2 + 1/(1/3)$). Thus, if we were to enact the change via Case (i) of Rule P, then we would *increase* the deadline of the first scheduled job of T_3, even though the weight of the task increased (i.e., such a change would decrease the scheduling priority of T_3). Therefore, we enact the weight change via Case (ii) of Rule P, which delays enacting the weight change until the deadline of T_3^1. □

Rule N changes the weight of a task by one of two approaches: **(i)** if a task *increases* its weight, then Rule N causes the release time of its next job to be adjusted so that it is commensurate with the new weight; **(ii)** if a task *decreases* its weight, then Rule N causes the next job to be issued with a deadline that is commensurate with the new weight at the end of the current job.

Example (Figure 3.6). Consider the example in Figure 3.6, which depicts a one-processor system with four tasks (where the execution time of each job is one): T_1, which has $\mathsf{wt}(T_1) = 1/2$ and leaves at time at time 2; T_2 and T_3, both of which have a weight of 1/6; and T_4,

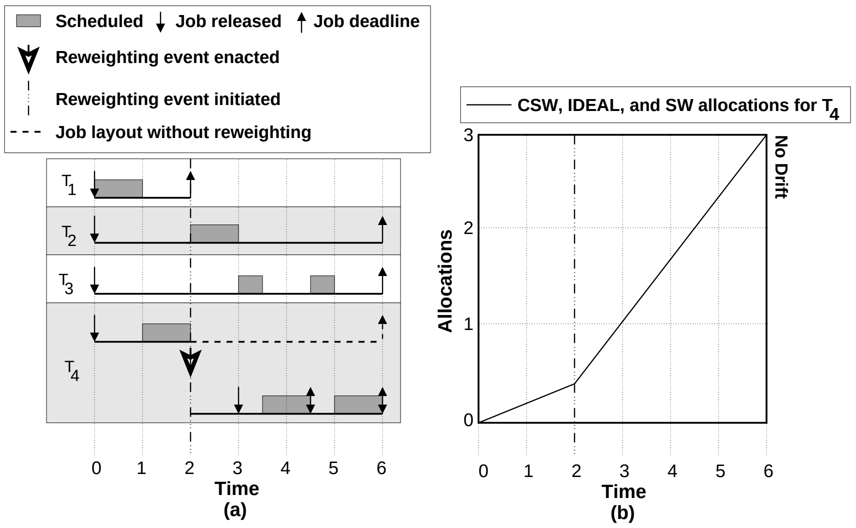


Figure 3.6: A one-processor example of reweighting via Case (i) of Rule N under GEDF. **(a)** The GEDF schedule. **(b)** T_4's allocations in the IDEAL, CSW, and SW schedules.

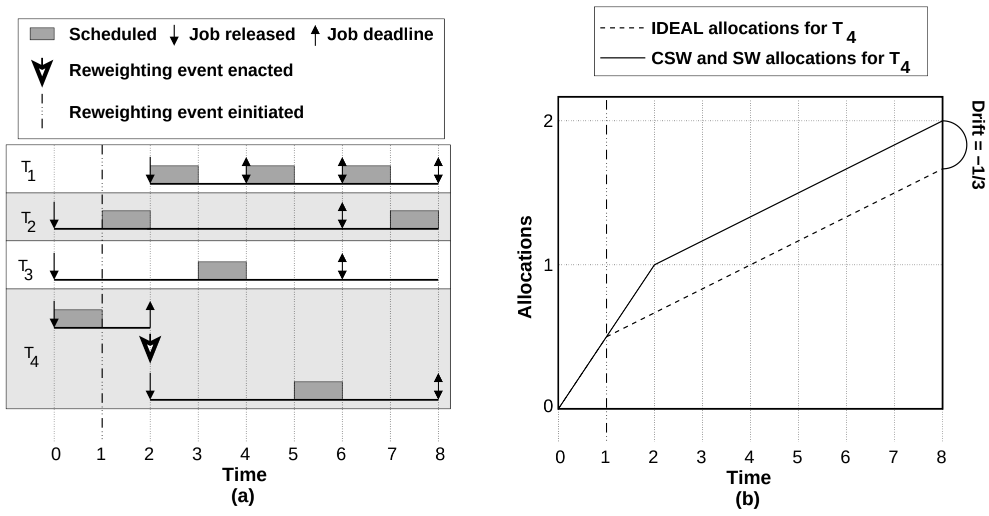


Figure 3.7: A one-processor example of reweighting via Case (ii) of Rule N under GEDF. **(a)** The GEDF schedule. **(b)** T_4's allocations in the IDEAL, CSW, and SW schedules.

which has an initial weight of 1/6 that increases to 4/6 at time 2. This is the same system as in Figure 3.4 except that T_4 has a higher priority than both T_2 and T_3. Inset (a) depicts the GEDF schedule. Inset (b) depicts T_4's allocations in the IDEAL, CSW, and SW schedules. (Again, the IDEAL and CSW schedules are formally defined later in Section 3.5.2.) Since T_4 has been scheduled by time 2, it has negative deviance and thus because it increases its weight, the change is enacted via Case (i) of Rule N. Thus, its next job is released time of 3, which is such that $\mathsf{dev}(T_4,\, 3) = \int_0^3 \mathsf{Swt}(T_i, u)du - \mathsf{A}(\mathcal{S},\, T_4,\, 0,\, 3) = 1 - 1 = 0$. By releasing the next job of T_4 at time 3, the drift incurred is zero. □

Example (Figure 3.7). Consider the example in Figure 3.7, which depicts a one-processor system with four tasks (where the execution time of each job is one): T_1, which joins the system at time 2 and has $\mathsf{wt}(T_1) = 1/2$; T_2 and T_3, both of which have a weight of 1/6; and T_4, which has an initial weight of 1/2 that initiates a weight decrease to 1/6 at time 1 that is enacted at time 2. Inset (a) depicts the GEDF schedule. Inset (b) depicts T_4's allocations in the IDEAL, CSW, and SW schedules. (Again, the IDEAL and CSW schedules are formally defined later in Section 3.5.2.) Since T_4 has negative deviance at time 1 and it decreases its weight, this weight change is enacted via Case (ii) of Rule N, causing T_4's next job to have a deadline of 8 and T_4 to have a drift of $-1/3$. □

Notice that if T_i initiates a weight change at time t_c while some job T_i^k of T_i (not necessarily its last-released job) has missed its deadline, then the Rules P and N specify that one of two actions is taken. If no job of T_i is active at t_c, then the weight change is enacted immediately. If there is a job T_i^j that is active at t_c, then since T_i^j has not been scheduled (because the earlier job T_i^k has missed its deadline), it follows that T_i is positive-changeable, and thus the weight change is enacted via Rule P (which may cause T_i^j *but not* T_i^k to halt). Notice that T_i^k is unaffected in both cases.

It is important to remember that when the Rules P and N halt a job, they do not abandon the computation that the job was performing. Rather, these rules split that computation across two jobs. Since these rules change the ordering of a task in the priority queues that determine scheduling, the time complexity for reweighting one task is $O(logN)$, where N is the number of tasks in the system (assuming priority queues are implement using binomial

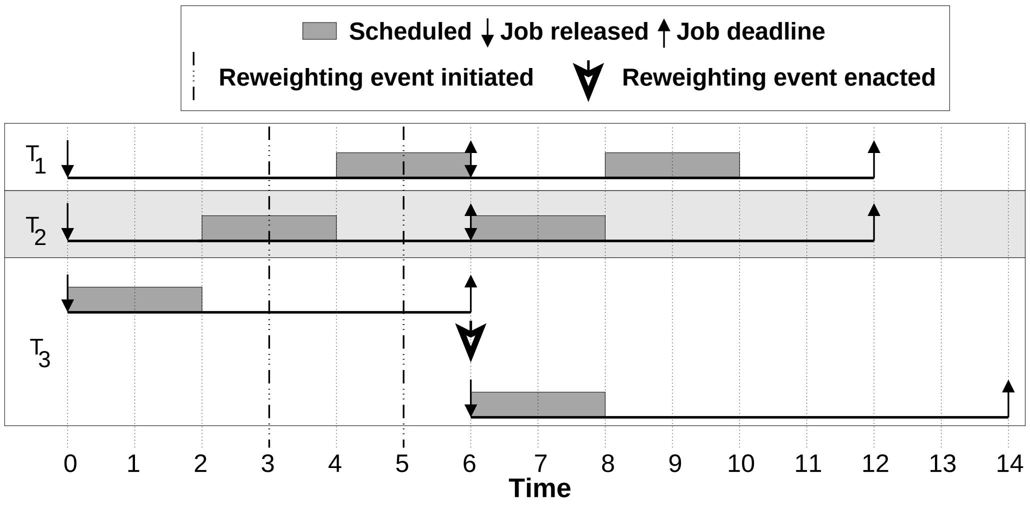


Figure 3.8: A one-processor example of canceling a reweighting event.

heaps).

Canceled reweighting events. We now introduce a property about the relationship between the initiation and enactment of a reweighting event in the case that some such events are canceled due to later reweighting events. Notice that, once a task T_i initiates a weight change at t_c, this weight change is eventually either canceled by another weight change or enacted. Further, Rules P and N enact any non-canceled reweighting event no later than the deadline of the last-released job T_i^j of T_i at t_c (if it exists and if $t_c \leq \mathsf{d}(T_i^j)$).

Example (Figure 3.8). Consider the example in Figure 3.8, which depicts a one-processor system with three tasks: T_1 and T_2, both of which have an execution time of 2 and a weight of 1/3; and T_3, which has $\mathsf{e}(T_3) = 2$ and an initial weight of 1/3 that changes to 1/10 at time 3 via Case (ii) of Rule N and then to 1/4 at time 5 via Case (ii) of Rule N. Notice that, because the change initiated at time 3 is via Case (ii) of Rule N, the change is not enacted until time 6. As a result, when a change is initiated at time 5, this new change cancels the previous change. Even though the change initiated at time 3 is canceled, the time of the next weight enactment is still at time 6. □

From Figure 3.8, we can see that, once a reweighting event has been initiated during an active job, some weight change will be enacted by the earlier of the deadline of that job

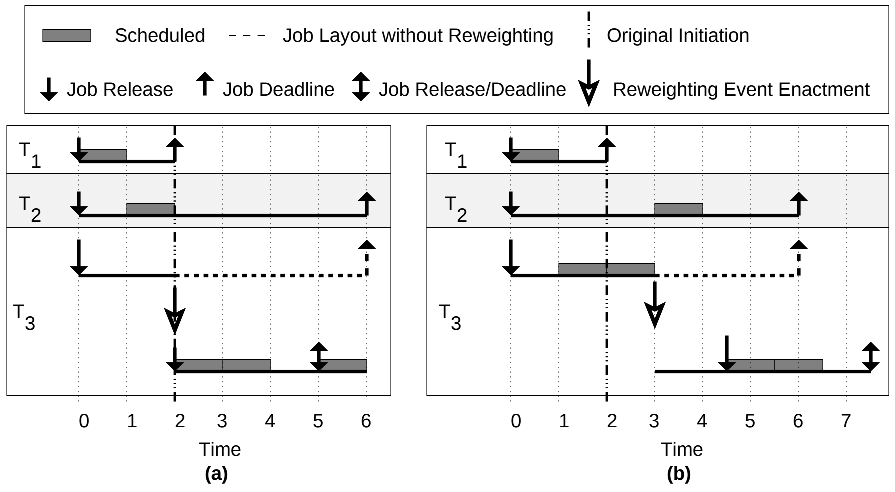


Figure 3.9: A one-processor example of NP-GEDF. **(a)** T_3 has a lower scheduling-priority than T_2. **(a)** T_2 has a lower scheduling-priority than T_3.

or when the job becomes inactive (which may be earlier, by Rules P and N). Property (X) formalizes this idea.

(X) If a task T_i initiates a weight change at time t_c and the job T_i^j is active at t_c, then some weight change is enacted according to Rule P or N by either $\mathsf{d}(T_i^j)$ or when T_i^j becomes inactive, whichever is first.

3.4.2 Modifications for NP-GEDF

In order to adapt Rules P and N to work for NP-GEDF, the only modification we need to make is when these rules are *initiated*. If a task with an active job reweights *before or after* that job has been scheduled, then Rules P and N are initiated as before. (Note that after the active job T_i^j has been released, if T_i^j *has not* been scheduled, then T_i is positive changeable, and if T_i^j *has* been scheduled, then T_i is negative changeable.) However, if a task changes its weight while the active job T_i^j is executing, then the initiation of the weight change is delayed *until* T_i^j *has completed* or T_i^j *is no longer active*, whichever is first. Note that, if a task T_i changes its weight from Ow to Nw at time t_c in NP-GEDF, then $\mathsf{wt}(T_i, t_c) = \mathsf{Nw}$ holds, regardless of whether the initiation of Rule P or N must be delayed.

Example (Figure 3.9). Consider the example in Figure 3.9, which depicts the NP-GEDF schedule of a one-processor system with three tasks: T_1, which has $\mathsf{e}(T_1) = 1$ and $\mathsf{wt}(T_1) = 1/2$

that leaves at time 2; T_2, which has $\mathsf{e}(T_2) = 1$ and $\mathsf{wt}(T_2) = 1/6$; and T_3, which has $\mathsf{e}(T_3) = 2$ and an initial weight of 1/3 that increases to 4/6 at time 2. In inset (a), T_3 has the lowest scheduling priority (there is a deadline tie). Since T_3 is not scheduled by time 2, it has positive deviance and changes its weight via Rule P, causing T_3^1 to be halted, T_3^2 to be released at time 2 with a deadline of 5. Inset (b) depicts the same scenario as in (a) except that T_3 has higher priority than T_2. Since T_3 is scheduled at time 2, and the system is schedule by NP-GEDF, the initiation of the reweighting event is delayed until T_3 stops executing at time 3. Since T_3^1 is complete by time 2, it has negative deviance and changes its weight via Rule N, causing its next job to have a release time of 9/2. □

3.5 Tardiness and Drift Bounds

In this section, we formally present and prove tardiness and drift bounds for the GEDF and NP-GEDF reweighting algorithms.

3.5.1 Tardiness Bounds

Instead of deriving tardiness bounds for GEDF or NP-GEDF when scheduling adaptable sporadic tasks from scratch (which would be quite tedious), we instead leverage the results reported by Devi and Anderson in (Devi and Anderson, 2008) concerning tardiness bounds that can be guaranteed under GEDF and NP-GEDF when scheduling *sporadic* tasks. In addition to deriving tardiness bounds under GEDF and NP-GEDF for sporadic task systems, Devi and Anderson also proposed an extension to the sporadic task model, referred to as the *extended sporadic task model*, and determined tardiness bounds that can be guaranteed to task systems that conform to the extended sporadic task model. We will show that any adaptable sporadic task system can be modeled as an extended sporadic task system; hence, tardiness bounds derived for extended sporadic task systems can be applied to adaptable sporadic task systems as well. We begin by describing the extended sporadic task model.

The extended sporadic task model. In the conventional sporadic task model, the number of tasks in a task system is fixed, and the sum of the weights of all its tasks is assumed to be at most m (the number of processors). On the other hand, in the extended model, the number of tasks associated with a task system is allowed to vary and the total weight of all tasks is allowed to exceed m. (The number of tasks could potentially be infinite.) Further, each task is assigned a static weight, and all jobs (except possibly the final job) of a task have equal execution times. However, to prevent overload, at any given time, only a subset of tasks whose total weight is at most m is allowed to be *effective*,[2] i.e., is allowed to release jobs. Additionally, the final job of a task can *stop*[3] at some time t_s before its deadline, provided the allocation that the job receives in the actual schedule is at most the allocation it receives up to t_s in the SW schedule,[4] i.e., the last job has non-negative deviance, and the job is not executing in a non-preemptive segment at t_s. When a job stops, its execution time is altered to equal the amount of time that the job actually executed for in the actual schedule up to time t_s. Thus, at any time t, each task T_i can be in one of the following states.

- *Effective*, if the first job of T_i is released at or before t, the deadline of its final job is after t, and its final job has not stopped at or before t. A task whose final job has its deadline at or before t is not considered effective at t even if the final job is pending at t.

- *Ineffective*, if the release time of the first job of T_i is after t.

- *Terminated*, if the deadline of T_i's final job is before t.

- *Stopped*, if the final job has stopped but its deadline has not elapsed.

As can be easily seen, a task that is either ineffective or terminated at time t cannot have active or effective jobs at t.

In order to provide tardiness bounds for extended sporadic task systems, Devi and Anderson proposed partitioning the set of all tasks associated with a task system into N task classes such that the following hold: **(i)** effective intervals are disjoint for every two tasks in

[2]In (Devi and Anderson, 2008), an effective task is referred to as an *active* task. We use this alternative term here to avoid conflicts in terminology.

[3]Here again, to avoid conflicting terminology, we differ from the term used in (Devi and Anderson, 2008). Stopping is referred to as *halting* in (Devi and Anderson, 2008).

[4]Since each extended sporadic task has only one weight, in an SW schedule the extended sporadic task T_i is allocated $\mathsf{wt}(T_i)$ at each instant it is effective.

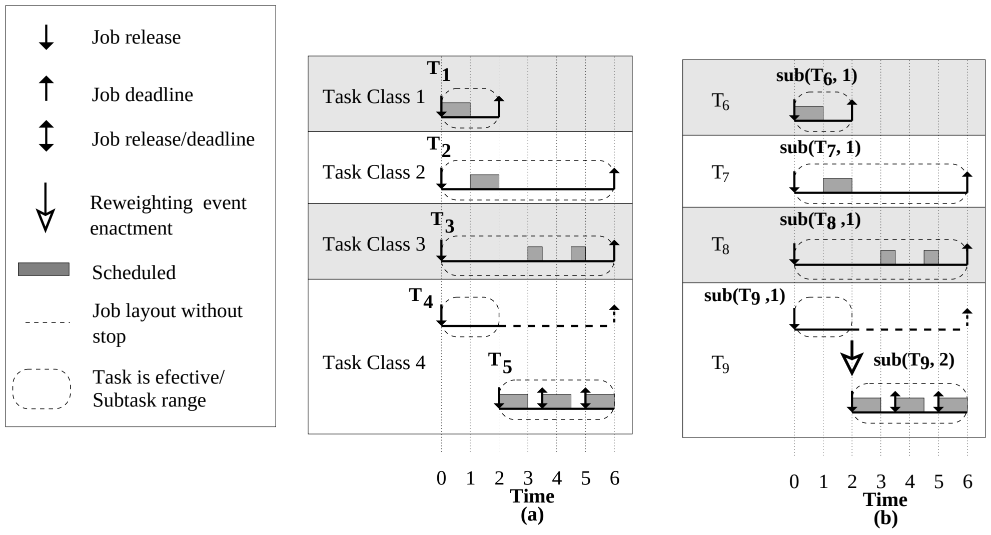


Figure 3.10: A one-processor example of task classes. **(a)** An extended sporadic task system. The effective range for each task is denoted by a dashed rectangle with rounded corners. **(b)** The same system as in Figure 3.4. Subtasks are denoted by a dashed rectangle with rounded corners.

each class and **(ii)** tasks within a class are governed by precedence constraints, i.e., the first job of a task cannot begin execution until all jobs of all tasks with earlier effective intervals in its class have completed execution. The second requirement implies that tasks that are not bound by precedence constraints should belong to different classes even if their effective intervals are disjoint.

Example (Figure 3.10). Consider the example in Figure 3.10(a), which depicts five tasks, each with an execution time of one: T_1, which has $\mathsf{wt}(T_1) = 1/2$ that leaves at time 2; T_2 and T_3, each of which have a weight of 1/6; T_4, which has $\mathsf{wt}(T_4) = 1/6$ and stops at time 2; and T_5, which has $\mathsf{wt}(T_5) = 4/6$, is in the same task class as T_4, and becomes effective as soon as T_4 stops. T_4^1 has a lower scheduling-priority than T_2^1 and T_3^1. Inset (b) depicts the same system as in Figure 3.4 (with the tasks renumbered for clarity). Notice that T_4 can stop at time 2 because its deviance is zero. Also note that, since T_4 and T_5 are in the same task class, only one of them can be effective at the same time. □

Let $T^{[\ell]}$ denote task class ℓ, and let $\mathsf{e}_{\mathsf{max}}(T^{[\ell]})$ and $\mathcal{W}(T^{[\ell]})$ denote the maximum execution time and weight, respectively, of any task in $T^{[\ell]}$. In (Devi and Anderson, 2008), it is shown that the tardiness for any task of any task class $T^{[i]}$ of an extended sporadic task system T

under global EDF is at most

$$\frac{\sum_{T^{[\ell]} \in \mathcal{E}(T,m-1)} \mathsf{e}_{\mathsf{max}}(T^{[\ell]})}{m - \sum_{T^{[\ell]} \in \mathcal{X}(T,m-2)} \mathcal{W}(T^{[\ell]})} + \mathsf{e}_{\mathsf{max}}(T^{[i]}), \tag{3.1}$$

and that under global NP-EDF is at most

$$\frac{\sum_{T^{[\ell]} \in \mathcal{E}(T,m)} \mathsf{e}_{\mathsf{max}}(T^{[\ell]})}{m - \sum_{T^{[\ell]} \in \mathcal{X}(T,m-1)} \mathcal{W}(T^{[\ell]})} + \mathsf{e}_{\mathsf{max}}(T^{[i]}), \tag{3.2}$$

where $\mathcal{E}(T,k)$ and $\mathcal{X}(T,k)$ are subsets of k task classes of T with the highest execution times and weights, respectively, for any of their tasks (i.e., with the highest values for $\mathsf{e}_{\mathsf{max}}(T^{[\ell]})$ and $\mathcal{W}(T^{[\ell]})$, respectively).

Extended and adaptable sporadic task systems. We now show how an adaptable sporadic task system can be modeled as an extended sporadic task system. We initially assume that no task changes its weight by Case (i) of Rule N, i.e., no negative-changeable task halts. Such weight changes are considered afterwards. We first show that each task of an adaptable sporadic task system can be modeled as a task class of an extended sporadic task system. For this, we decompose each adaptable sporadic task into disjoint "subtasks[5]," where a *subtask* $\mathsf{sub}(T_i, j)$ of a adaptable sporadic task T_i is a "maximal" set of jobs with the following properties: **(i)** the jobs in $\mathsf{sub}(T_i, j)$ are consecutive jobs of T_i; **(ii)** each job is released between the same pair of two consecutive weight-change enactments for T_i; **(iii)** each job has the same execution time; and **(iv)** no new job can be added to $\mathsf{sub}(T_i, j)$ without violating one or more of properties (i), (ii), and (iii), and in that sense, $\mathsf{sub}(T_i, j)$ is maximal. As an example, consider Figure 3.10(b), which depicts the same system as in Figure 3.4 with the subtasks marked.

Recall that in an extended sporadic task system, if a job T_i^j "stops" at time t_s, then $t_s < \mathsf{d}(T_i^j)$, T_i^j's deviance is non-negative, and when a job stops, its actual execution time is set to the value that the job had executed for in the actual schedule up to time t_s. Since we are assuming that only positive-changeable tasks may halt, which by definition have an

[5]This usage of the term "subtask" should not be confused with that used in work on Pfair scheduling.

active job that has non-negative deviance, it is easy to see that for positive-changeable tasks "stopping" in an extended sporadic task system has the same effect as "halting" an adaptable sporadic task system. For example, the impact on the system when T_4^1 stops in Figure 3.10(a) is the same as when T_6^1 halts in Figure 3.10(b).

By definition, all jobs of a subtask have equal execution times, and because all such jobs are released between two consecutive weight-change enactments, each subtask has a static scheduling weight. Also, as explained above, halting is the same as stopping for a positive-changeable task. Hence, if no task changes its weight via Case (i) of Rule N, then it follows that each subtask in the adaptable sporadic task model corresponds to a task, with a static weight and execution time, in the extended sporadic task model, and each task in a adaptable sporadic task model that consists of subtasks corresponds to a task class, composed of tasks with different weights or execution times or both, of the extended sporadic task model. Also note that intervals within which subtasks of an adaptable sporadic task are effective are disjoint. For example, consider insets (a) and (b) of Figure 3.10, which despite the notation change, have the identical schedules.

We now explain that an adaptable sporadic task can be modeled as an extended sporadic task even if tasks change their weight via Case (i) of Rule N. Before we continue, notice that the one difference between positive and negative-changeable tasks with respect to halting at time t_h is as follows: if T_i is positive-changeable at t_h and its job T_i^j is active at that time, then T_i^{j+1}, i.e., the next job of T_i, may be released at t_h, whereas if T_i is negative-changeable, then T_i^{j+1} may not be released until time $t_r > t_h$, where t_r is the earliest time at which the allocations to T_i^j are equal in the actual schedule and under SW, i.e., the next time T_i^j's deviance is zero. Thus, if T_i is negative-changeable and halts at t_h, then T_i^j may be thought of as stopping at time t_r, where t_r is as defined above. However, notice that by Case (i) of Rule N, if T_i^j halts at time t_h, then T_i enacts a weight increase at time t_h. Thus, the time t_r is calculated using dynamic weights, which are not explicitly included in Devi and Anderson's extended sporadic model.

Example (Figure 3.11). Consider the example in Figure 3.11(a), which depicts five tasks, each with an execution time of one: T_1, which has $\mathsf{wt}(T_1) = 1/2$ and leaves at time 2; T_2

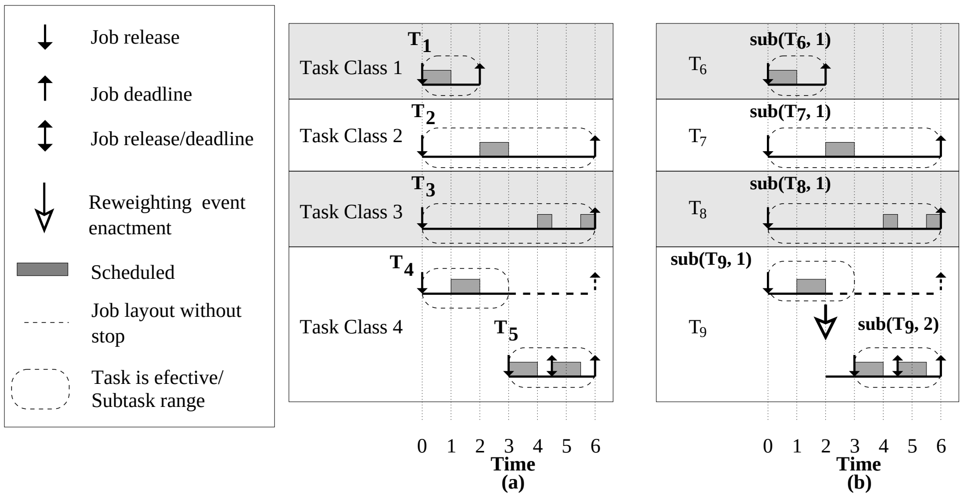


Figure 3.11: A one-processor example of task classes. **(a)** An extended sporadic task system. The effective range for each task is denoted by a dashed rectangle with rounded corners. **(b)** The same system as in Figure 3.6. Subtasks are denoted by a dashed rectangle with rounded corners.

and T_3, each of which has a weight of 1/6; T_4, which has an initial weight of 1/6, at time 2 increases its weight to 4/6, and at time 3 stops; and T_5, which has $\mathsf{wt}(T_5) = 4/6$, is in the same task class as T_4, and becomes effective as soon as T_4 stops. T_4^1 has a higher scheduling-priority than T_2^1 and T_3^1. Inset (b) depicts the same system as in Figure 3.6 (with the tasks renumbered for clarity). Since T_4 and T_5 are in the same task class, only one of them can be effective at the same time. Notice that T_4 can stop at time 3 because its actual allocation until then is no greater than its SW allocation; however, if we were using static weights than T_4^1 would not be able to stop until time 6. $\square$

Even though dynamic weights are not explicitly included in the extended sporadic model, the bounds in (3.1) and (3.2) still hold if the only time a task is allowed to change its weight is when its final job has finished executing and the weight change is an increase, which is exactly the scenario that arises when a task changes its weight via Case (i) of Rule N. The reason why (3.1) and (3.2) still hold in the presence of such weight changes is because the extended sporadic task model only requires that the total allocation in the SW schedule to a stopping job T_i^j at the time it stops be at least the allocation T_i^j received in the actual schedule; increasing the rate at which the T_i^j is allocated time in the SW schedule is not an issue as long as the total SW allocation to all tasks that are effective is at most m at each instant.

Informally, this holds by the following reasoning. Increasing a task T_i's allocation rate in the SW schedule would cause T_i's period to decrease. As a result, jobs of T_i would have a higher relative scheduling priority after the change. However, since the final job of T_i has already completed execution in the actual schedule when it halts, this increase in priority does not impact any job. In particular, it is not possible for another job to have a lower priority than the halting job T_i^j before the weight change and a higher priority after the weight change. Therefore, scheduling T_i^j (the halting job) in the past with a lower priority does not adversely impact how any other job was scheduled in the past. Since T_i^j has already completed execution before its deadline, T_i^j's tardiness is not impacted either.

Hence, if the task T_i enacts a weight change via Case (i) of Rule N at t_h and the system is not over-utilized after the change, no other job will be impacted. Thus, since a weight change is enacted at t_h regardless of whether T_i is positive- or negative-changeable, and T_i^{j+1} is released at or after t_h, in both the cases, T_i^j and T_i^{j+1} belong to different subtasks of T_i. Hence, the definition of a subtask is unaltered even in the presence of negative-changeable jobs by Case (i) of Rule N, and the correspondence described earlier between a adaptable sporadic task and an extended sporadic task holds.

Thus, the tardiness bounds specified in (3.1) and (3.2) and that can be guaranteed to extended sporadic task systems are also applicable to adaptable sporadic task systems if task class $T^{[\ell]}$ is replaced by adaptable sporadic task T_z, and $\mathsf{e}_{\mathsf{max}}(T_z)$ and $\mathcal{W}(T_z)$ are taken as the maximum execution time of any job of T_z and the maximum weight assigned to T_z at any time. (It should be noted that the tardiness bounds hold only if the sum of the weights of all tasks that are active at any instant is at most m.) Thus, we have the following theorem.

Theorem 3.1. *Let τ be an adaptable sporadic task system, where for any $t \geq 0$, $\sum_{T_i \in \mathcal{T}} \mathsf{Swt}(T_i, t) \leq m$. Then, for any task T_i, GEDF on m processors ensures a tardiness of at most*

$$\frac{\sum_{T_z \in \mathcal{E}(m-1)} \mathsf{e}_{\mathsf{max}}(T_z)}{m - \sum_{T_z \in \mathcal{X}(m-2)} \mathsf{wt}_{\mathsf{max}}(T_z)} + \mathsf{e}_{\mathsf{max}}(T_i),$$

and NP-GEDF *on m processors ensures a tardiness of at most*

$$\frac{\sum_{T_z \in \mathcal{E}(T,m)} \mathsf{e}_{\mathsf{max}}(T_z)}{m - \sum_{T_z \in \mathcal{X}(T,m-1)} \mathcal{W}(T_z)} + \mathsf{e}_{\mathsf{max}}(T_i).$$

3.5.2 Additional Theoretical Algorithms

"Drift bounds" (formally defined in Section 3.5.3) reflect a reweighting algorithm's accuracy at creating a job set that mimics an "ideal" task system, in which weight changes can always be initiated and enacted instantaneously. In order to define drift and prove drift bounds for the reweighting rules proposed in Section 3.4, we introduce two additional theoretical scheduling algorithms that are able to preempt and swap tasks at arbitrarily small intervals: the *clairvoyant scheduling-weight* (CSW) scheduling algorithm and the *ideal* (IDEAL) scheduling algorithm. The CSW scheduling algorithm allocates each task a fraction of the system equal to its *scheduling weight*, and *will not allocate* capacity to a task if its active job has received an allocation equal to its actual execution time. The IDEAL scheduling algorithm allocates each task a fraction of the system equal to its *weight* (i.e., not its scheduling weight) at each instant. Further, the IDEAL scheduling algorithm *continually allocates* capacity to a task as long as it has an active job.

We now discuss these two algorithms in more detail, by exploring their differences when scheduling the two example systems presented in Figures 3.12, 3.13, and 3.14.

Example (Figures 3.12 and 3.13). Consider the example in Figures 3.12 and 3.13, which depict a one-processor system with four tasks: T_1, which has $\mathsf{e}(T_1) = 1$ and $\mathsf{wt}(T_1) = 1/3$; T_2, which has $\mathsf{e}(T_2) = 1$ and $\mathsf{wt}(T_2) = 1/6$; T_3, which has $\mathsf{e}(T_3) = 2$ and $\mathsf{wt}(T_3) = 1/4$ and leaves at time 8; and T_4, which has $\mathsf{e}(T_4) = 4$ and an initial weight of 1/4 and initiates and enacts a weight increase to 1/2 at time 8 (the same system as in Figure 3.2). Figure 3.12(a) depicts the GEDF schedule. Figure 3.12(b) depicts the allocations to T_4 in the GEDF, IDEAL, SW, and CSW scheduling algorithms. Figure 3.13(a) depicts the SW schedule. Figure 3.13(b) depicts the CSW schedule. Figure 3.13(c) depicts the IDEAL schedule. Notice that T_4^1 receives no

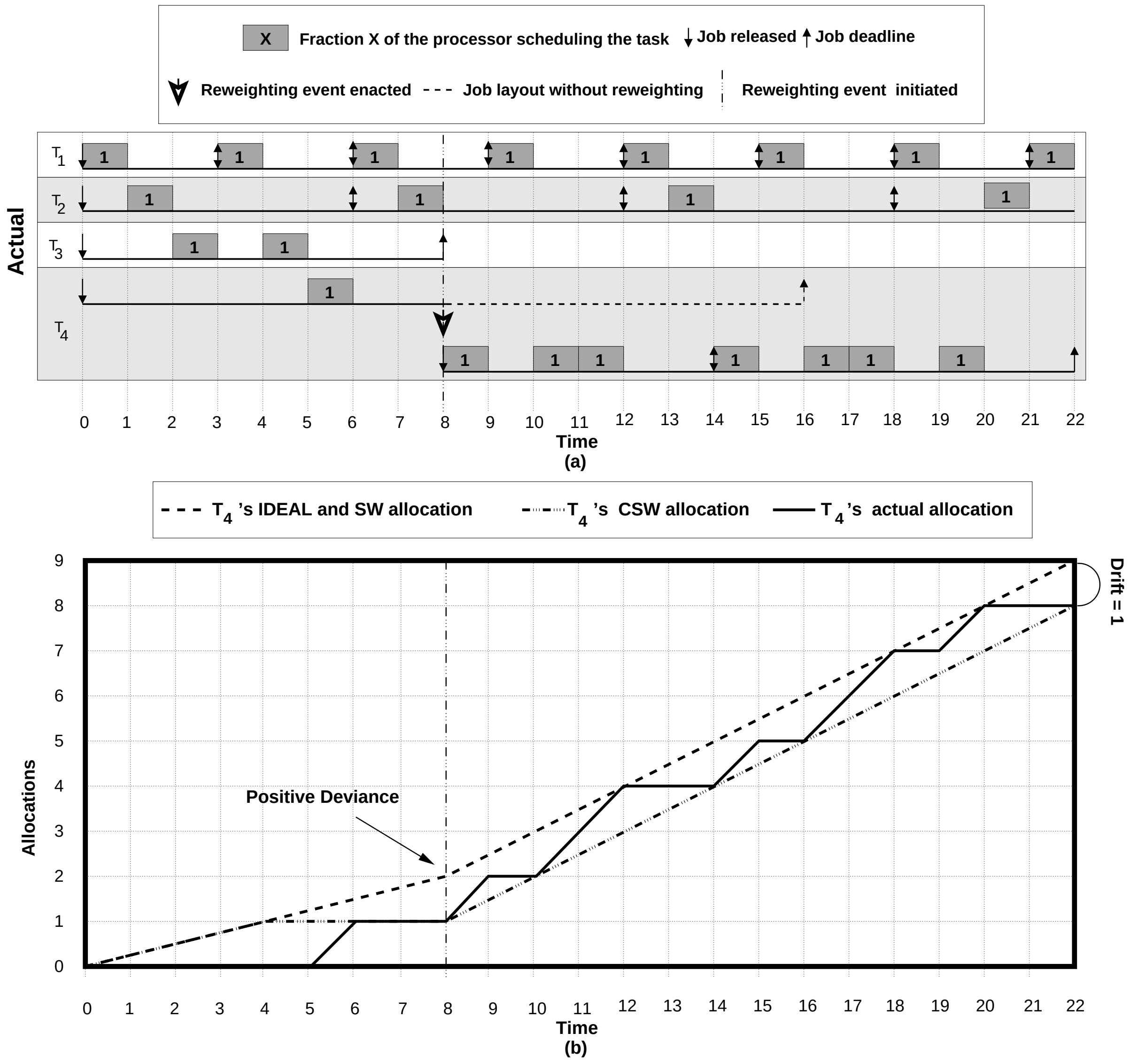

X Fraction X of the processor scheduling the task
Job released
Job deadline
Reweighting event enacted
Job layout without reweighting
Reweighting event initiated
Actual
T_1
T_2
T_3
T_4
Time
(a)
T_4's IDEAL and SW allocation
T_4's CSW allocation
T_4's actual allocation
Allocations
Positive Deviance
Drift = 1
Time
(b)

Figure 3.12: A one-processor example of a task that increases its weight. **(a)** The GEDF schedule. **(b)** The allocations to T_4 in the GEDF, SW, CSW, and IDEAL schedules.

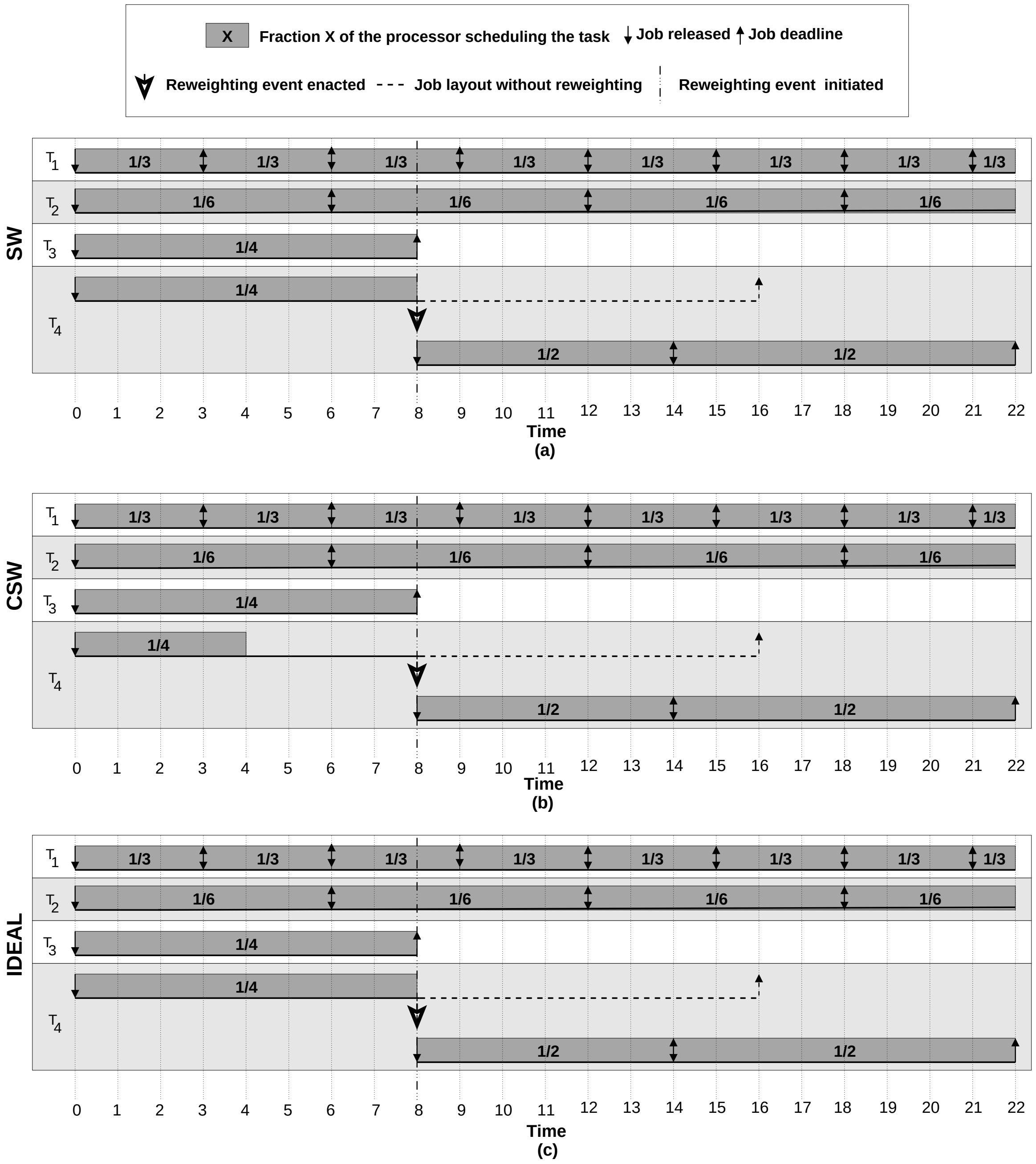


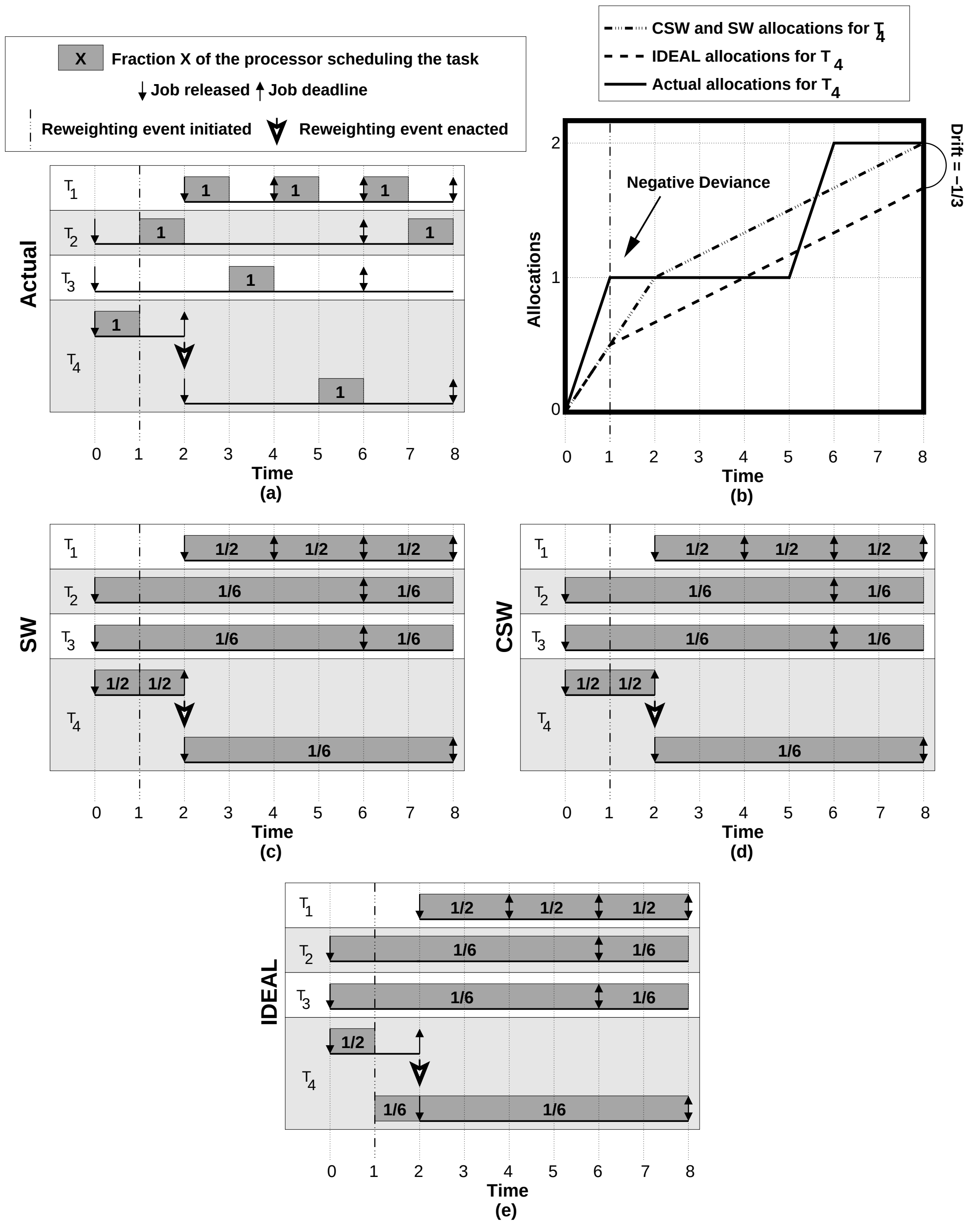
Figure 3.13: A continuation of Figure 3.12 that depicts **(a)** SW, **(b)** CSW, and **(c)** IDEAL schedules.

allocations in the CSW schedule once it has received one unit of execution (the amount T_4^1 is allocated in the GEDF schedule). □

Example (Figure 3.14). Consider the example in Figure 3.14, which depicts a one-processor system with four tasks (where the execution time of each job is one): T_1, which joins the system at time 2 with a $\mathsf{wt}(T_1) = 1/2$; T_2 and T_3, both of which have a weight of 1/6; and T_4, which has an initial weight of 1/2 and initiates a weight decrease to 1/6 at time 1 that is enacted at time 2 (the same system as in Figure 3.3). Inset (a) depicts the GEDF schedule. Inset (b) depicts the allocations to T_4 in the GEDF, IDEAL, SW, and CSW scheduling algorithms. Inset (c) depicts the SW schedule. Inset (d) depicts the CSW schedule. Inset (e) depicts the IDEAL schedule. □

The CSW scheduling algorithm. CSW is a theoretical scheduling algorithm that is used as a reference for calculating drift. Under the CSW scheduling algorithm, at each instant t, each job of each task T_i that is both active and incomplete (in the CSW schedule) is allocated a fraction of a processor equal to $\mathsf{Swt}(T_i, t)$. Furthermore, we consider CSW to be "clairvoyant" in the sense that CSW uses the actual execution time of T_i^j to determine if T_i^j has completed before it halts. More specifically, for any schedule $\mathcal{CSW}$ under CSW of any task system τ, we say that T_i^j has *completed by time t in $\mathcal{CSW}$* iff T_i^j has executed for $\mathsf{Ae}(T_i^j)$ by t. Thus, the difference between SW and CSW is that a job in SW will not stop receiving allocations as long as its active, whereas a job in CSW will stop receiving allocations as soon as it receives its actual execution time. Throughout this dissertation we use $\mathcal{CSW}$ to denote the CSW schedule of a task system τ.

Example (Figures 3.12 and 3.13). Consider the system depicted in Figures 3.12 and 3.13 (described above). In this system, the reweighting rules Rule P stops T_4^1 from scheduling its second unit of execution. As a result, $\mathsf{Ae}(T_4^1) = 1$. So, as illustrated in Figure 3.13(b), T_4^1 stops receiving allocation in the CSW schedule once it has been allocated one unit of execution (at time 4). T_4 resumes execution once its next job has been released at time 8. This differs from the SW schedule, where T_4 receives allocations over the range [4, 8). Notice that, in the example depicted in Figure 3.14, $\mathsf{Ae}(T_4^1) = \mathsf{e}(T_4^1)$, so T_4^1's allocations are identical in both



X Fraction X of the processor scheduling the task
Job released
Job deadline
Reweighting event initiated
Reweighting event enacted
CSW and SW allocations for T_4
IDEAL allocations for T_4
Actual allocations for T_4
Actual
SW
CSW
IDEAL
T_1
T_2
T_3
T_4
1
1/2
1/6
Negative Deviance
Drift = −1/3
Allocations
Time
(a)
(b)
(c)
(d)
(e)

Figure 3.14: A one-processor example of a task that decreases its weight. **(a)** The GEDF schedule. **(b)** The allocations to T_4 in the GEDF, IDEAL, SW, and CSW scheduling algorithms. **(c)** The SW schedule. **(c)** The CSW schedule. **(d)** The IDEAL schedule.

the $\mathcal{SW}$ and $\mathcal{CSW}$ schedules. Notice that, in both the CSW and actual schedule, the total time allocated to a job is the same. Thus, the CSW algorithm more accurately represents the behavior of a task than the SW algorithm. $\square$

The IDEAL scheduling algorithm. Under the *ideal* (IDEAL) scheduling algorithm, at each instant t, each task T_i in τ with an active job at t is allocated a fraction of the system equal to its weight (i.e., $\mathsf{wt}(T_i,\, t)$). Hence, if $\mathcal{I}$ is the IDEAL schedule of τ and T_i is active over the interval $[t_1,\, t_2)$, then over $[t_1,\, t_2)$, the task T_i is allocated $\mathsf{A}(\mathcal{I},\, T_i^j,\, t_1,\, t_2) = \int_{t_1}^{t_2} \mathsf{wt}(T_i,\, u)du$ time. As mentioned earlier, the IDEAL algorithm is similar to SW, with one major exception: each task receives an allocation equal to its *weight*, whereas under SW, each task receives an allocation equal to its *scheduling weight*. Throughout this dissertation we use $\mathcal{I}$ to denote the IDEAL schedule of a task system τ.

Example (Figures 3.12–3.14). Notice that in the system depicted in Figure 3.14, the reweighting event initiated at time 1 by T_4 is not enacted until time 2. As a result, over the range $[1,\, 2)$, in the IDEAL schedule, T_4 receives $\mathsf{wt}(T_4,\, t) = 1/6$ at each instant, whereas in the SW schedule, T_4 receives $\mathsf{Swt}(T_4, t) = 1/2$ at each instant. On the other hand, in the systems depicted in Figures 3.12 and 3.13, T_4's reweighting event is enacted as soon as it is initiated. As a result, the IDEAL and SW schedules are the same. $\square$

3.5.3 Drift

For most real-time scheduling algorithms, the difference between the IDEAL and actual allocations a task receives lies within some bounded range centered at zero. For example, under a *uniprocessor* EDF schedule, the difference between the ideal and actual allocations for a task lies within $(-\mathsf{e}_{\mathsf{max}}(T_i), \mathsf{e}_{\mathsf{max}}(T_i))$ (assuming the processor is not over-utilized). When a weight change occurs, the same bounds are maintained except that they may be centered at a different value. For example, in Figure 3.12, the range for T_4 is originally $(-4, 4)$, but after the reweighting event, it is $(-3, 5)$. This lost allocation is called *drift*. Given this loss (barring further reweighting events) T_i 's drift will not change. In general, a task's drift per reweighting event will be non-negative if it increases its weight, and a task's drift per reweighting event will be non-positive if it decreases its weight. The drift of a task T_i at time t is defined as

$$\mathsf{drift}(T_i,\, t) = \mathsf{A}(\mathcal{I},\, T_i,\, 0,\, t) - \mathsf{A}(\mathcal{CSW},\, T_i,\, 0,\, t).$$

(Notice that drift is defined in terms of CSW instead of SW. This is because, as we discussed earlier, the CSW scheduling algorithm is a more accurate representation of the actual schedule than the SW scheduling algorithm.) The *per-event absolute drift* is the absolute value of the amount of drift that is incurred as a result of a reweighting event. For example, if the per-event absolute drift is $\mathsf{e}_{\mathsf{max}}(T_i)$, then after n reweighting events, the maximal absolute drift is $n \cdot \mathsf{e}_{\mathsf{max}}(T_i)$.

Under GEDF, the drift per reweighting event is bounded as follows.

Theorem 3.2. *The per-event absolute drift under* GEDF *for each task T_i is at most $\mathsf{e}_{\mathsf{max}}(T_i)$.*

Proof. We first show that for any job T_i^j of any task T_i, $\mathsf{A}(\mathcal{CSW},\, T_i,\, \mathsf{r}(T_i^j),\, t_A) \leq \mathsf{e}_{\mathsf{max}}(T_i)$, where t_A is the time that T_i^j becomes inactive. Notice that, since a job is inactive by its deadline (i.e., $t_A \leq \mathsf{d}(T_i^j)$), and the deadline of T_i^j is defined as $\mathsf{r}(T_i^j) + \mathsf{e}(T_i^j)/\mathsf{Swt}(T_i, \mathsf{r}(T_i^j))$, it follows that if T_i does not enact a weight change over the range $[\mathsf{r}(T_i^j), t_A)$, i.e., T_i 's scheduling weight is static over the range $[\mathsf{r}(T_i^j), t_A)$, then $\mathsf{A}(\mathcal{CSW},\, T_i,\, \mathsf{r}(T_i^j),\, t_A) \leq \mathsf{e}(T_i^j) \leq \mathsf{e}_{\mathsf{max}}(T_i)$.

Thus, in order for, $\mathsf{A}(\mathcal{CSW},\, T_i,\, \mathsf{r}(T_i^j),\, t_A) > \mathsf{e}_{\mathsf{max}}(T_i)$ to hold, it must be that T_i enacted a weight change over the range $[\mathsf{r}(T_i^j), t_A)$. Thus, we assume that T_i enacts a change over the range $[\mathsf{r}(T_i^j), t_A)$, and that t_e is the last such time. Notice that if the change enacted at t_e is by Case (i) or (ii) of Rule P or Case (ii) of Rule N, then T_i^j becomes inactive at t_e, which contradicts our assumption that $t_e < t_A$. Thus, the change enacted at t_e must be by Case (i) of Rule N. By Case (i) of Rule N, T_i^j becomes inactive at the first time at or after t_e such that the deviance of T_i^j equals zero, i.e., t_A is the smallest time such that $\int_{\mathsf{r}(T_i^j)}^{t_A} \mathsf{Swt}(T_i, u)du = \mathsf{Ae}(T_i^j)$. Since $\mathsf{A}(\mathcal{CSW},\, T_i,\, \mathsf{r}(T_i^j),\, t_A) \leq \int_{\mathsf{r}(T_i^j)}^{t_A} \mathsf{Swt}(T_i, u)du$ and since $\mathsf{Ae}(T_i^j) \leq \mathsf{e}(T_i^j) \leq \mathsf{e}_{\mathsf{max}}(T_i^j)$, it follows that

$$\mathsf{A}(\mathcal{CSW},\, T_i,\, \mathsf{r}(T_i^j),\, t_A) \leq \mathsf{e}_{\mathsf{max}}(T_i). \tag{3.3}$$

Let t_c be a time such that some task T_i initiates a weight change. Let t_e denote the next time that the change initiated at t_c is either canceled or enacted, whichever is first. Thus, by the definition of canceled, no weight change is initiated over the range (t_c, t_e). We show that regardless of which rule T_i uses to change its weight, the drift incurred by this initiation is at most $\mathsf{e}_{\mathsf{max}}(T_i)$. Let T_i^j be the last-released job of T_i before t_c. Notice that if T_i^j is not active at t_c or T_i^j does not exist, then the change is immediately enacted and no job is halted. Since the only two potential sources of drift are delays in enacting a weight change and halting a job (which causes the actual execution time to be lower than the execution time), it follows that if T_i^j does not exist or T_i^j is inactive at t_c then no drift is incurred. Thus, for the rest of this proof, we assume that T_i^j is active at t_c, and we let t_A denote the time that T_i^j becomes inactive. By Property (X) and the fact that $t_A \leq \mathsf{d}(T_i^j)$ holds, we have

$$t_c \leq t_e \leq t_A \leq \mathsf{d}(T_i^j). \tag{3.4}$$

If T_i changes its weight at time t_c via Case (i) of Rule P, then since this weight change is immediately enacted (i.e., $t_c = t_e$), it is as though allocation equal to $\mathsf{A}(\mathcal{I},\, T_i,\, \mathsf{r}(T_i^j),\, t_c) - \mathsf{A}(\mathcal{CSW},\, T_i,\, \mathsf{r}(T_i^j),\, t_c)$ is "lost." For example in Figure 3.4, the task T_4 "loses" an allocation of 2/6. Notice that, per reweighting event, $\mathsf{A}(\mathcal{I},\, T_i,\, \mathsf{r}(T_i^j),\, t_c) - \mathsf{A}(\mathcal{CSW},\, T_i,\, \mathsf{r}(T_i^j),\, t_c) \leq \mathsf{e}_{\mathsf{max}}(T_i)$. Also note that since, by (3.3) and (3.4), $\mathsf{A}(\mathcal{CSW},\, T_i,\, \mathsf{r}(T_i^j),\, t_c) \leq \mathsf{e}_{\mathsf{max}}(T_i)$, it follows that $-\mathsf{e}_{\mathsf{max}}(T_i) \leq \mathsf{A}(\mathcal{I},\, T_i,\, \mathsf{r}(T_i^j),\, t_c) - \mathsf{A}(\mathcal{CSW},\, T_i,\, \mathsf{r}(T_i^j),\, t_c)$. Thus, since $-\mathsf{e}_{\mathsf{max}}(T_i) \leq \mathsf{A}(\mathcal{I},\, T_i,\, \mathsf{r}(T_i^j),\, t_c) - \mathsf{A}(\mathcal{CSW},\, T_i,\, \mathsf{r}(T_i^j),\, t_c) \leq \mathsf{e}_{\mathsf{max}}(T_i)$, it follows that the absolute drift is at most $\mathsf{e}_{\mathsf{max}}(T_i)$.

Suppose that T_i initiates a change to weight Nw via Case (ii) of Rule P at t_c. Since this change is enacted or canceled at time t_e, it is as though allocation equal to $\mathsf{A}(\mathcal{I},\, T_i,\, t_c,\, t_e) - \mathsf{A}(\mathcal{CSW},\, T_i,\, t_c,\, t_e)$ is "lost." Recall that a task only changes its weight via Case (ii) of Rule P if the deviance of T_i is positive and if $\mathsf{d}(T_i^j) - t_c \leq \mathsf{REM}(T_i^j, t_c)/\mathsf{Nw}$. Since by (3.4), $t_e \leq \mathsf{d}(T_i^j)$ and $\mathsf{REM}(T_i^j, t_c) \leq \mathsf{e}_{\mathsf{max}}(T_i)$, the previous inequality can be rewritten as

$$\mathsf{Nw} \cdot (t_e - t_c) \leq \mathsf{e}_{\mathsf{max}}(T_i). \tag{3.5}$$

Recall that no change is initiated over the range (t_c, t_e). Thus, by definition, $\mathsf{A}(\mathcal{I},\, T_i,\, t_c,\, t_e) =$

$\int_{t_c}^{t_e}(\mathsf{Nw})du = \mathsf{Nw} \cdot (t_e - t_c)$. Hence, by (3.5), $\mathsf{A}(\mathcal{I}, T_i, t_c, t_e) \leq \mathsf{e}_{\mathsf{max}}(T_i)$. Furthermore, by (3.3) and (3.4), $\mathsf{A}(\mathcal{CSW}, T_i, t_c, t_e) \leq \mathsf{e}_{\mathsf{max}}(T_i)$. Thus, $-\mathsf{e}_{\mathsf{max}}(T_i) \leq \mathsf{A}(\mathcal{I}, T_i, t_c, t_e) - \mathsf{A}(\mathcal{CSW}, T_i, t_c, t_e) \leq \mathsf{e}_{\mathsf{max}}(T_i)$. Since $\mathsf{A}(\mathcal{I}, T_i, t_c, t_e) - \mathsf{A}(\mathcal{CSW}, T_i, t_c, t_e)$ denotes the lost allocation for this reweighting event, it follows that the absolute drift is at most $\mathsf{e}_{\mathsf{max}}(T_i)$. For example, in Figure 3.5, over the range $[2, 4)$ for task T_3, an allocation of $\mathsf{A}(\mathcal{I}, T_3, 2, 4) - \mathsf{A}(\mathcal{CSW}, T_3, 2, 4) = 2/3 - 2/4 = 1/6$ is lost.

Suppose that T_i changes its weight to Nw at time t_c via Rule N. If T_i decreases its weight, then it is as though the allocation equal to $\mathsf{A}(\mathcal{I}, T_i, t_c, t_e) - \mathsf{A}(\mathcal{CSW}, T_i, t_c, t_e)$ is lost. Furthermore, since it was a weight decrease initiated at t_c, it follows that $\mathsf{A}(\mathcal{I}, T_i, t_c, t_e) < \mathsf{A}(\mathcal{CSW}, T_i, t_c, t_e)$. Thus, since by (3.3) and (3.4), $\mathsf{A}(\mathcal{CSW}, T_i, t_c, t_e) \leq \mathsf{e}_{\mathsf{max}}(T_i)$, it follows that $\mathsf{A}(\mathcal{I}, T_i, t_c, t_e) < \mathsf{A}(\mathcal{CSW}, T_i, t_c, t_e) \leq \mathsf{e}_{\mathsf{max}}(T_i)$. Thus, $-\mathsf{e}_{\mathsf{max}}(T_i) \leq \mathsf{A}(\mathcal{I}, T_i, t_c, t_e) - \mathsf{A}(\mathcal{CSW}, T_i, t_c, t_e) \leq \mathsf{e}_{\mathsf{max}}(T_i)$. Since $\mathsf{A}(\mathcal{I}, T_i, t_c, t_e) - \mathsf{A}(\mathcal{CSW}, T_i, t_c, t_e)$ denotes the lost allocation for this reweighting event, it follows that the absolute drift incurred is at most $\mathsf{e}_{\mathsf{max}}(T_i)$. For example, in Figure 3.7, the drift incurred by T_4 is $-1/3$, i.e., $\mathsf{drift}(T_4, t) = -1/3$, where $t \geq 2$. If T_i increases its weight (Case (i)), then it incurs zero drift, since it *immediately* enacts the weight change (i.e., the scheduling weight changes immediately). Hence, the absolute drift incurred by this reweighting event is less than $\mathsf{e}_{\mathsf{max}}(T_i)$. For example, in Figure 3.6, the drift incurred by T_4 is 0, i.e., $\mathsf{drift}(T_4, t) = 0$, where $t \geq 2$. □

Notice that the presence of jobs that miss their deadlines does not affect the drift bounds. The reason for this is that the reweighting rules are *only* based on the state of the active job at the time the reweighting event is initiated. Thus, if a job has not been scheduled by the time it reweights, then it does not matter whether a predecessor prevented the job from being scheduled or the job had the lowest scheduling priority, only that the job has not been scheduled, and therefore is positive-changeable.

Example (Figure 3.15). Consider the example in Figure 3.15, which depicts the partial schedule for a task T_i that has all of the following characteristics: an initial weight of 1/10 that increases to 1/2 at time t_c; a job T_i^{j-1} that has a deadline at $t_r = t_c - 5$, an execution time of 14, and misses its deadline by 11 time units; and all jobs released after T_i^{j-1} have an execution time of 1. Inset (a) depicts the GEDF schedule. Inset (b) depicts the CSW

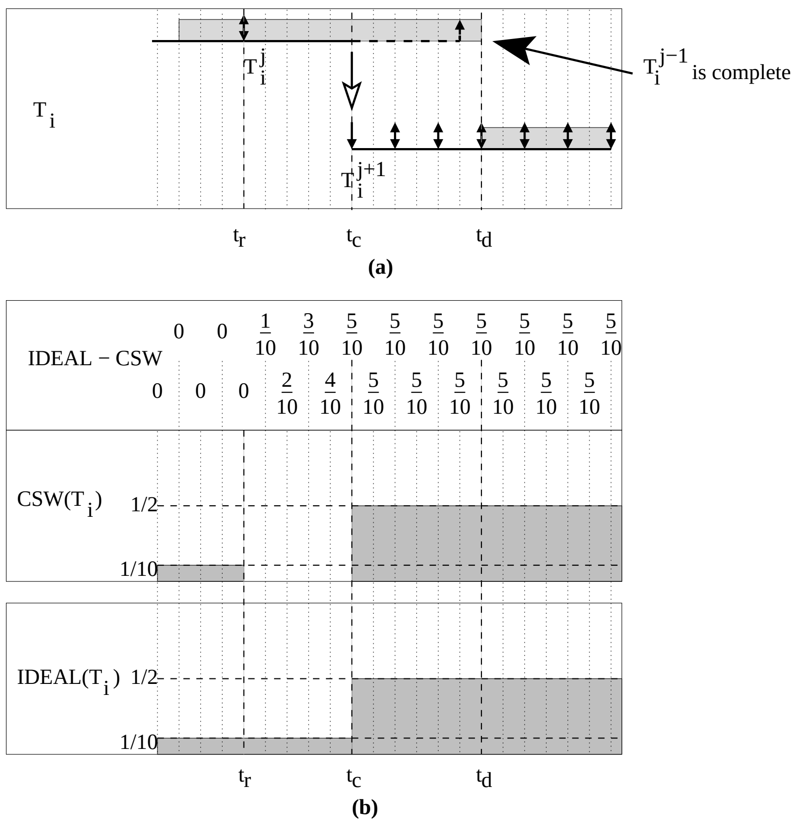


Figure 3.15: A partial schedule that illustrates drift when tasks miss deadlines. The partial **(a)** GEDF and **(b)** CSW and IDEAL schedules for the task T_i. The difference between T_i's allocation in CSW and IDEAL are labeled above inset (b).

and IDEAL schedules. Notice that, even though T_i^{j-1} misses its deadline at time t_r, when T_i initiates the change at time t_c, T_i^j is the active job, and since it has not been scheduled, T_i is positive-changeable at t_c. Therefore, by Rule P, T_i^j (but *not* T_i^{j-1}) is halted and T_i^{j+1} is immediately released with the new weight, which incurs a drift of 1/2. $\square$

Modifications for NP-GEDF. Note that delaying the initiation of a reweighting event due to non-preemptivity does not substantially increase the drift incurred per reweighting event, since the longest a reweighting event can be delayed is the execution time of the active job of the task being reweighted.

Suppose that the task T_i initiates a weight change at time t_c. If T_i^j is active at t_c, and if T_i's reweighting event is delayed until some time t (by a non-preemptive section), then at t either **(a)** T_i^j has a non-positive deviance (i.e., T_i^j completes before its deadline), or **(b)** t is the first time that T_i^j becomes inactive (i.e., $t = \mathsf{min}(\mathsf{r}(T_i^{j+1}), \mathsf{d}(T_i^j))$.)

If Case (a) occurs, then T_i is negative-changeable at t, and T_i^j is active at t. Hence, if T_i increases its weight, then the only drift T_i will incur for this reweighting event results from delaying the initiation of the event, i.e., at most $\mathsf{e_{max}}(T_i)$. If T_i decreases its weight, then delaying the reweighting event will not affect drift, since the enactment of the reweighting event would occur when T_i^j becomes inactive, regardless of whether the initiation of the reweighting event was delayed or not.

Example (Figure 3.16). Consider the example in Figure 3.16, which depicts a one-processor system scheduled by NP-GEDF with two tasks: T_1, which has $\mathsf{wt}(T_1) = 3/10$ and $\mathsf{e}(T_1) = 3$; and T_2, which has $\mathsf{e}(T_2) = 2$ and an initial weight of 1/5 and initiates a weight increase to 1/2 at time 4. Inset (a) depicts the NP-GEDF schedule. Inset (b) depicts the CSW schedule. Inset (c) depicts the IDEAL schedule. Inset (d) depicts T_2's allocations in the CSW and IDEAL schedules. Notice that T_2's weight change is delayed from time 4 to time 5 because T_2 is non-preemptively executing at time 4. As a result, T_2 is negative-changeable at time 5. Also note that, T_2^2 is released when T_2's actual allocation equals its allocation in the CSW schedule at time 7, i.e., when T_2^2's deviance equals zero. $\square$

If Case (b), mentioned earlier, occurs, then either no job of T_i is active at t or T_i^{j+1} is active at t. If no job of T_i is active at t, then the change is enacted immediately, and the drift that the task incurs from the reweighting event is a result of delaying the initiation of the event, i.e., $\mathsf{e_{max}}(T_i)$. If T_i^{j+1} is active at t, then since $t = \mathsf{min}(\mathsf{r}(T_i^{j+1}), \mathsf{d}(T_i^j))$, it must be the case that $\mathsf{r}(T_i^{j+1}) = t$. As a result, the weight change is enacted immediately and T_i^{j+1} is released with the new weight. Hence, the only drift that is incurred is as a result of delaying the initiation of the reweighting event, i.e., at most $\mathsf{e_{max}}(T_i)$.

Example (Figure 3.17). Consider the example in Figure 3.17, which depicts a partial NP-GEDF schedule for a task T_i, which has an initial weight of 1/10 that increases to 1/2 at time t_c while the last-released job of T_i before t_c, T_i^j, is both active and being scheduled. Note that T_i^j has an execution time of four, and all jobs released after T_i^j have an execution time of one. Moreover, T_i^j does not complete execution until after its deadline. Inset (a) depicts the NP-GEDF schedule. Inset (b) depicts the CSW schedule. Inset (c) depicts

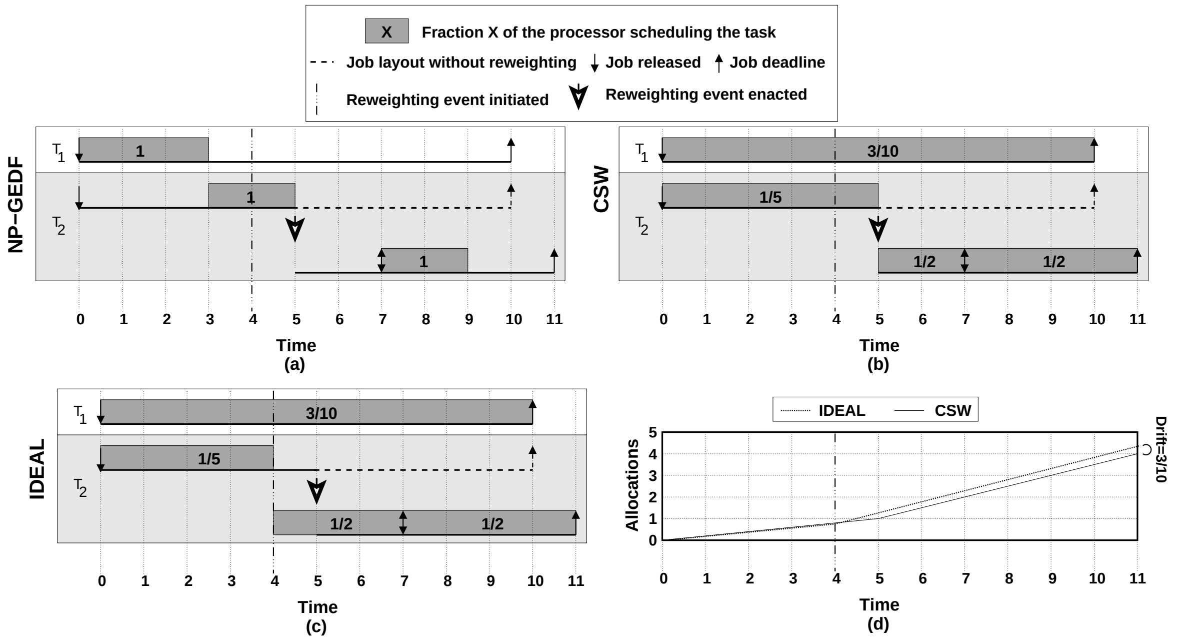


Figure 3.16: A one-processor example of drift in NP-GEDF, where T_2^1 completes before its deadline. **(a)** The NP-GEDF schedule. **(b)** The CSW schedule. **(c)** The IDEAL schedule. **(d)** T_2's allocations in the CSW and IDEAL schedules.

the IDEAL schedule. Inset (d) depicts T_i's allocation in the CSW and IDEAL schedules. Because T_i^j is not complete by its deadline, the initiation of the weight change is delayed until $t = \mathsf{d}(T_i^j) = \mathsf{r}(T_i^{j+1})$. Recall that, if a weight change is initiated when $\mathsf{d}(T_i^j) = \mathsf{r}(T_i^{j+1})$, then the weight change is immediately enacted and T_i^{j+1} is released with the new weight (even though T_i^j has not yet completed execution). Thus, the only source of drift is because the initiation of the reweighting event is delayed. □

From the reasoning presented in these examples, we can see that the following theorem holds.

Theorem 3.3. *The per-event absolute drift under* NP-GEDF *for each task T_i is at most* $\mathsf{e}_{\mathsf{max}}(T_i)$.

3.6 Conclusion

In this chapter, we presented the adaptable sporadic task model as well as the rules for reweighting a task under the GEDF and NP-GEDF scheduling algorithms. In addition, we

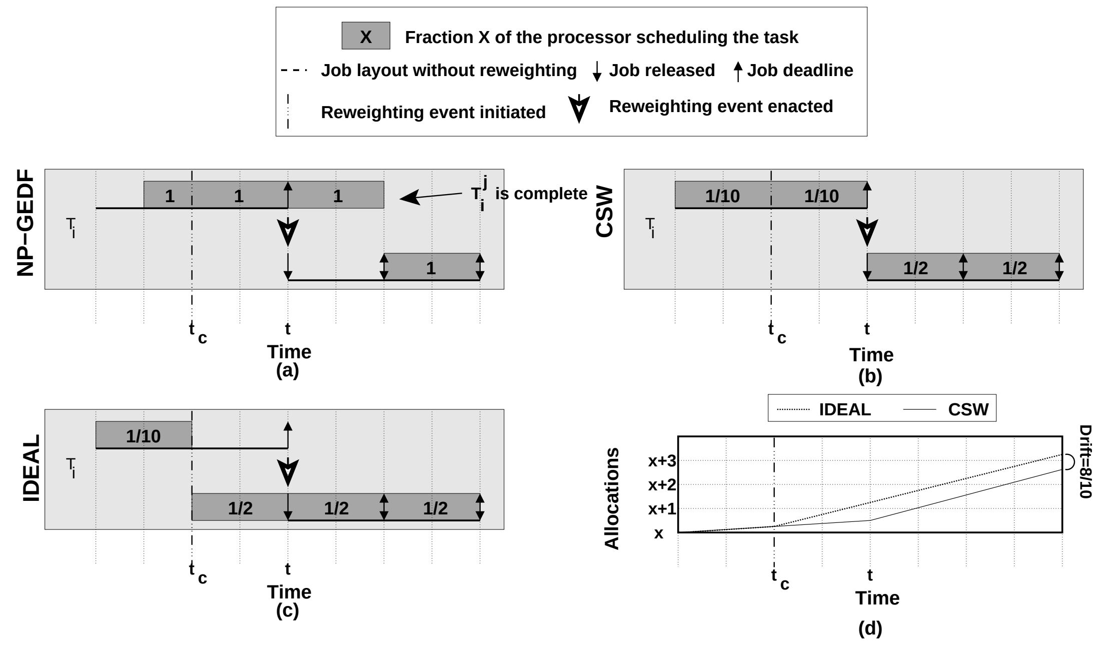


Figure 3.17: A partial schedule of a one-processor example of drift in NP-GEDF. T_i^j completes after its deadline. **(a)** The NP-GEDF schedule. **(b)** The CSW schedule. **(c)** The IDEAL schedule. **(d)** T_i's allocations in the CSW and IDEAL schedules.

proved tardiness bounds our reweighting rules by leveraging prior work by Devi and Anderson. In addition, we proved that the absolute value of the drift that can be incurred per reweighting event is at most the maximal execution time of a task.

CHAPTER 4

PEDF and NP-PEDF*

In this chapter, we examine the issue of reweighting in the context of partitioned algorithms. Because there cannot exist an optimal[1] partitioned scheduling algorithm, we focus our attention on different heuristic tradeoffs that can minimize different sources of error. Before discussing these tradeoffs in detail, we first define some necessary notation and consider a fundamental limitation of all partitioning algorithms.

4.1 Preliminaries

In this section, we introduce a few terms that will facilitate our discussion of partitioned systems. We denote the q^{th} processor in the system, where processors are ordered by some arbitrary method, as $P_{[q]}$. As a shorthand, we use $T_i \in P_{[q]}$ to denote that T_i is assigned to $P_{[q]}$. We denote the set of tasks that are assigned to $P_{[q]}$ at time t as $\mathsf{ASSN}(P_{[q]}, t)$. We denote the set of tasks that are assigned to $P_{[q]}$ and active at time t as $\mathsf{ACT}(P_{[q]}, t)$. (Recall that a task T_i is active at time if it has an active job at time t, and a job T_i^j is active at time t if $t \in [\mathsf{r}(T_i^j), \mathsf{min}(\mathsf{d}(T_i^j), \mathsf{r}(T_i^{j+1})))$.) We denote the *desired* and *guaranteed weight* of T_i at time t as $\mathsf{Dwt}(T_i, t)$ and $\mathsf{Gwt}(T_i, t)$, respectively. (As we discuss in Section 4.4.3, when a task's guaranteed and desired weight differ, the releases and deadlines of its jobs will be based on its guaranteed weight.) If a task's desired weight does not change with time, then we denote

* Contents of this chapter previously appeared in preliminary form in the following paper:
Block, A. and Anderson J. (2006). Accuracy versus migration overhead in multiprocessor reweighting algorithms. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 355–364.

[1] A reweighting algorithm is *optimal* if each task can always be granted a guaranteed weight equal to its desired weight, provided the sum of all desired weights is at most the number of available processors.

this value as $\mathsf{Dwt}(T_i)$.

We say that $T_i \in P_{[q]}$ is the *heaviest* task assigned to $P_{[q]}$ iff T_i has the largest desired weight of any task assigned to $P_{[q]}$. Similarly, we say that $T_i \in P_{[q]}$ is the *lightest* task assigned to $P_{[q]}$ iff T_i has the smallest desired weight of any task assigned to $P_{[q]}$. We say that a processor $P_{[q]}$ is *over-utilized by x* iff it has been assigned tasks with a total desired weight of $1 + x$. Similarly, we say that $P_{[q]}$ is *under-utilized by x* iff it has been assigned tasks with a total desired weight of $1 - x$. Additionally, we say that $P_{[q]}$ is *fully-utilized* iff it has been assigned tasks with a total desired weight of 1. If $P_{[q]}$ is over-utilized by x at time t, then we denote this value as $\omega(P_{[q]}, t)$; if $P_{[q]}$ is not over-utilized at t, then $\omega(P_{[q]}, t) = 0$. These terms (as well as other terms used throughout this chapter) are summarized in Table 4.1.

4.2 A Limitation of Partitioning Schemes

As was mentioned in Section 1.2.2, under any partitioning scheme, there exist task systems where only a subset of tasks can receive their desired allocation even though the total weight of all tasks is at most the number of processors. For example, consider a two-processor system with three identical periodic tasks with an execution cost of 2.0 and a period of 3.0. Because tasks are partitioned, one processor will be assigned two of these tasks, thus over-utilizing it. There are two approaches for handling this problem. First, we could cap the total utilization of all tasks in the system. Unfortunately, under any M-processor partitioning scheme, a cap of approximately $M/2$ is required in the worst case (Carpenter et al., 2004), which implies that as much as half the system's processing capacity could be lost. Such caps are due to connections to bin-packing.

An alternative approach is to assign a subset of tasks in the system guaranteed weights that are less than their desired weights. Although allocating a task a weight less than its desired weight is obviously undesirable, such an approach can guarantee that the system's overall capacity does not have to be restricted, which is a significant advantage in computationally-intensive systems like Whisper and VEC. Moreover, allowing the guaranteed weights of tasks to be somewhat malleable circumvents any bin-packing-like intractabilities that might otherwise arise—with frequent weight changes, such intractabilities would have to be dealt with

Notation	Definition
$P_{[q]}$	The q^{th} processor.
$T_i \in P_{[q]}$	T_i is assigned to $P_{[q]}$.
$\mathsf{ASSN}(P_{[q]}, t)$	Set of tasks assigned to $P_{[q]}$ at time t.
$\mathsf{ACT}(P_{[q]}, t)$	Set of tasks assigned to $P_{[q]}$ that are active at time t.
$\mathsf{e}(T_i^j)$	WECT of job T_i^j.
$\mathsf{e}_{\mathsf{max}}(T_i)$	Maximal worst-case execution time of all jobs of T_i.
$\mathsf{Ae}(T_i^j)$	Actual execution time of job T_i^j.
$\mathsf{r}(T_i^j)$	Release time of T_i^j.
$\mathsf{d}(T_i^j, t)$	Perceived deadline of T_i^j at time t.
$\mathsf{d}(T_i^j)$	Deadline of T_i^j.
$\theta(T_i^j)$	IS separation between T_i^{j-1} and T_i^j.
$\mathsf{Dwt}(T_i, t)$	T_i's desired weight at time t.
$\mathsf{Gwt}(T_i, t)$	T_i's guaranteed weight at time t.
$\mathsf{SDwt}(T_i, t)$	T_i's desired scheduling weight at time t.
$\mathsf{SGwt}(T_i, t)$	T_i's guaranteed scheduling weight at time t.
$\mathcal{T}\mathsf{D}(P_{[q]}, t)$	$P_{[q]}$'s desired weight scaling factor: $\mathsf{max}(1, \sum_{T_i \in \mathsf{ACT}(P_{[q]}, t)} \mathsf{Dwt}(T_i, t))$.
$\mathcal{T}\mathsf{S}(P_{[q]}, t)$	$P_{[q]}$'s desired scheduling weight scaling factor: $\mathsf{max}(1, \sum_{T_i \in \mathsf{ACT}(P_{[q]}, t)} \mathsf{SDwt}(T_i, t))$.
$\omega(P_{[q]}, t)$	$\mathsf{max}(0, 1 - \sum_{T_i \in \mathsf{ACT}(P_{[q]}, t)} \mathsf{Dwt}(T_i, t))$.
$\mathsf{Irem}(T_i^j, t)$	$\mathsf{e}(T_i^j) - \int_{\mathsf{r}(T_i^j)}^{t} \mathsf{SGwt}(T_i, u) du$.
SW	Scheduling-weight scheduling algorithm.
$\mathcal{SW}$	SW schedule of a task system τ.
CSW	Clairvoyant scheduling-weight scheduling algorithm.
$\mathcal{CSW}$	CSW schedule of a task system τ.
IDEAL	Ideal scheduling algorithm.
$\mathcal{I}$	IDEAL schedule of a task system τ.
PT	Partial ideal scheduling algorithm.
$\mathcal{PT}$	PT schedule of a task system τ.
$\mathcal{S}$	Actual schedule (i.e., GEDF or $\mathsf{NP\text{-}GEDF}$) of task system τ.
$\mathsf{A}(\mathcal{B}, T_i^j, t_1, t_2)$	Allocation to T_i^j in the schedule $\mathcal{B}$ over $[t_1, t_2)$.
$\mathsf{A}(\mathcal{B}, T_i, t_1, t_2)$	Allocation to T_i in the schedule $\mathcal{B}$ over $[t_1, t_2)$.
$\mathsf{dev}(T_i^j, t)$	Deviance of T_i^j: $\mathsf{A}(\mathcal{SW}, T_i^j, 0, t) - \mathsf{A}(\mathcal{S}, T_i^j, 0, t)$.
$\mathsf{drift}(T_i, t)$	Drift of T_i: $\mathsf{A}(\mathcal{I}, T_i, 0, t) - \mathsf{A}(\mathcal{CSW}, T_i, 0, t)$.
$\mathsf{Pdrift}(T_i, t)$	Partial drift of T_i: $\mathsf{A}(\mathcal{PT}, T_i, 0, t) - \mathsf{A}(\mathcal{CSW}, T_i, 0, t)$.
Ow	Desired scheduling weight before a reweighting event.
Nw	New desired weight after a reweighting event.
$\mathsf{REM}(T_i^j, t)$	Remaining execution time of T_i^j at t: $\mathsf{e}(T_i^j) - \mathsf{A}(\mathcal{S}, T_i^j, 0, t)$.
$\mathsf{nextE}(T_i^j, t)$	If $\mathsf{REM}(T_i^j, t) > 0$, then $\mathsf{nextE}(T_i^j, t) = \mathsf{REM}(T_i^j, t)$; else, $\mathsf{nextE}(T_i^j, t) = \mathsf{e}(T_i^{j+1})$.
$\mathsf{H}(T_i^j, t)$	Jobs with a scheduling priority higher than or equal to T_i^j's that are assigned to the same processor as T_i^j and are both active and pending at time t.
$\mathsf{lag}(T_i^j, t)$	Lag of T_i^j at t: $\mathsf{A}(\mathcal{CSW}, T_i^j, \mathsf{r}(T_i^j), t) - \mathsf{A}(\mathcal{S}, T_i^j, \mathsf{r}(T_i^j), t)$.
$\mathsf{LAG}(G, t)$	Lag of the job group G at time t: $\sum_{T_i^j \in G} \mathsf{lag}(T_i^j, t)$.

Table 4.1: Summary of notation used in this chapter.

frequently at *run-time*. Note that we are still able to offer some service guarantees with this approach, as discussed in Sections 4.7 and 4.9. (In particular, if the maximum amount by which a processor is over-utilized is relatively small, then the resulting guaranteed weight may be acceptable.) For these reasons, we use this approach in the schemes we propose. To the best of our knowledge, we are the first to suggest using such an approach to schedule dynamically-changing multiprocessor workloads. The fundamental limitation of partitioned schemes noted at the beginning of this chapter is formalized by the following theorem.

Theorem 4.1. *For any partitioned scheduling algorithm and any integers M and k such that $M \geq 2$ and $k \geq M+1$, there exists an M-processor system τ with k tasks that have a total desired weight at most M where at least one processor is assigned a set of tasks that have a total desired weight greater than one.*

Proof. In order to prove Theorem 4.1, we construct a system that satisfies the theorem for any value of M and k such that $M \geq 2$ and $k \geq M+1$. Let the first M tasks of τ have a desired weight $X = 1 - \epsilon$, where $0 < \epsilon < 0.5$. Let the $(M+1)^{st}$ task of τ have a desired weight of $W = \mathsf{min}(M \cdot \epsilon - \delta, 1 - \epsilon)$, where $0 < \delta < \epsilon$, and let the total desired weight of the remaining $k - (M+1)$ tasks of τ be δ. (For example, if $\epsilon = 1/3$, $k = 3$, and $M = 2$, then the system consists of three tasks of weight 2/3.) By definition, the desired weight of the first $(M+1)^{st}$ tasks have a total desired weight of at most $M \cdot (1-\epsilon) + M \cdot \epsilon - \delta = M - \delta$. Since the total desired weight of the remaining $k - (M+1)$ tasks is δ, the total desired weight of all the tasks is at most M.

Thus, it remains to be shown that one processor is assigned tasks with a total desired weight greater than one. Notice that no matter how the first $M+1$ tasks are partitioned, at least one processor will been assigned two of these tasks, i.e., at least one processor will be assigned either two of the first M tasks or one of the first M tasks and the $(M+1)^{st}$ task. If two of the first M tasks are assigned to the same processor, then the total desired weight on that processor is at least $2 \cdot (1 - \epsilon)$. In this case, since $\epsilon < 0.5$, the processor will have been assigned tasks with total desired weight greater than one. If one processor is assigned one of the first M tasks and the $(M+1)^{st}$ task, then the total desired weight of the tasks assigned to this processor would be $1 - \epsilon + W$. Thus, it remains to be shown that $W > \epsilon$. Thus, we consider two cases depending on whether $W = 1 - \epsilon$ or $W = M \cdot \epsilon - \delta$. If $W = 1 - \epsilon$, then since $\epsilon < 0.5$, $W > \epsilon$. If $W = M \cdot \epsilon - \delta$, then since $M \geq 2$ and $\delta < \epsilon$, it follows that $W > \epsilon$. This completes the proof. $\square$

4.3 Partitioning and Repartitioning

The problem of assigning tasks to processors is equivalent to the NP-hard bin-packing problem. Given that reweighting events may be frequent, an optimal assignment of tasks to processors is not realistic to maintain. In our approach, we partition N tasks onto M processors in $O(M + N \log N)$ time by first sorting them by desired weight from heaviest to lightest, and by then placing each on the processor that is the "best fit" (this partitioning method is called *descending best-fit*). We chose this method because it falls within a class of bin-packing heuristics called *reasonable allocation decreasing* (RAD), which has been shown by Lopez *et al.* to produce better packings than other types of heuristics (Lopez et al., 2004). Most importantly, the "descending best-fit" strategy can guarantee that no processor is over-utilized by more than W, where W is the desired weight of the $\left(M \cdot \left\lfloor \frac{1}{X} \right\rfloor + 1\right)^{st}$ heaviest task in the system and X is the desired weight of the heaviest task in the system. Also, under this strategy, no processor is over-utilized by more than the desired weight of the lightest task assigned to it.

As tasks are reweighted, the likelihood of a processor becoming *substantially* over-utilized increases dramatically, creating significant overall error (however assessed) on these processors. The extent of overall error can be controlled by repartitioning the system. In order to give the user control over migration overhead, we introduce *α-partitioning*: if a reweighting event causes the total desired weight of all tasks assigned to any one processor to be at least $1 + \alpha$, the system is *reset*, where α is a user-defined value. A reset causes the set of tasks to be repartitioned (using the descending best-fit method described earlier) and each active task to issue a new job with the remaining execution time of its pending job ("pending" is formally defined in Section 1.2). In Section 4.6, we formally define the rule for resetting a system.

Metric Name	**Metric Formula**
MAOE	$\max_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \{\mathsf{Dwt}(T_i,t) - \mathsf{Gwt}(T_i,t)\}$
AAOE	$\frac{1}{n} \cdot \sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} (\mathsf{Dwt}(T_i,t) - \mathsf{Gwt}(T_i,t))$
MROE	$\max_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \left\{\frac{\mathsf{Dwt}(T_i,t) - \mathsf{Gwt}(T_i,t)}{\mathsf{Dwt}(T_i,t)}\right\}$
AROE	$\frac{1}{n} \cdot \sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \left(\frac{\mathsf{Dwt}(T_i,t) - \mathsf{Gwt}(T_i,t)}{\mathsf{Dwt}(T,t)}\right)$

Table 4.2: The MAOE, AAOE, MROE, and AROE metrics.

4.4 Allowing Guaranteed and Desired Weights to Differ

Given that we allow a task's guaranteed and desired weight to differ, two questions arise. First, how do we determine each task's guaranteed weight? Second, how do we modify the adaptable sporadic task model (presented in Section 3.1) to accommodate a task with a different guaranteed and desired weight? In this section, we discuss several possible answers to these questions.

4.4.1 Determining Guaranteed Weights

When a processor $P_{[q]}$ is over-utilized, there is a degradation in the system's performance because at least one task assigned to $P_{[q]}$ will have a guaranteed weight that is less than its desired weight. Unfortunately, it is not immediately clear how this degradation should be measured. For example, should we measure the absolute or relative difference between each task's desired and guaranteed weight? In this section, we propose four different metrics for measuring the system's degradation, summarized in Tables 4.2 and 4.3, and explain how to determine the guaranteed weight of tasks in order to minimizing each of these metrics.[2] We illustrate these metrics via the following example.

Example (Figure 4.1). Consider the example in Figure 4.1, which depicts one processor that is assigned four tasks: T_1, which has a desired weight of 0.36; T_2 and T_3, both of which have a have a desired weight of 0.30; and T_4, which has a desired weight of 0.24. Inset (a) depicts the desired weight for each task. Insets (b), (c), and (d) depict, respectively, the

[2]In Table 4.3, the formula for minimizing the MAOE only holds if $\frac{\omega(P_{[q]},t)}{|\mathsf{ASSN}(P_{[q]},t)|} \leq \mathsf{Dwt}(T_z,t)$ for every $T_z \in \mathsf{ASSN}(P_{[q]},t)$. Additionally, the formula for minimizing the AROE only holds if $\omega(P_{[q]},t) \leq \mathsf{Dwt}(T_H,t)$.

Metric Name	**Guaranteed Weight Assignment**
MAOE	$\mathsf{Gwt}(T_i,t) = \mathsf{Dwt}(T_i,t) - \frac{\omega(P_{[q]},t)}{\lvert\mathsf{ASSN}(P_{[q]},t)\rvert}$
AAOE	$1 = \sum_{T_z \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{Gwt}(T_z,t)$
MROE	$\mathsf{Gwt}(T_i,t) = \frac{\mathsf{Dwt}(T_i,t)}{\sum_{T_z \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{Dwt}(T_z,t)}$
AROE	$\mathsf{Gwt}(T_i,t) = \begin{cases} \mathsf{Dwt}(T_i,t) - \omega(P_{[q]},t) & \text{if } T_i = T_H \\ \mathsf{Dwt}(T_i,t) & \text{otherwise} \end{cases}$

Table 4.3: The guaranteed weight assignments for $T_i \in \mathsf{ASSN}(P_{[q]},t)$ that minimize the MAOE, AAOE, MROE, and AROE metrics. T_H is the heaviest task in $\mathsf{ASSN}(P_{[q]},t)$.

guaranteed weights for the four tasks when such weights are chosen to minimize the MAOE, MROE, and AAOE metrics (discussed shortly). □

Absolute Error. The first two metrics we consider are based on the absolute difference between the guaranteed and desired weight of a task. Specifically, the *maximal absolute overall error* (MAOE) on an over-utilized processor, $P_{[q]}$, is given by

$$\max_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \left\{\mathsf{Dwt}(T_i,t) - \mathsf{Gwt}(T_i,t)\right\}.$$

To minimize this metric, the difference between the guaranteed and the desired weight for each task should be the same. Specifically, to minimize the MAOE metric, the guaranteed weight for $T_i \in P_{[q]}$ at time t is specified as

$$\mathsf{Gwt}(T_i,t) = \mathsf{Dwt}(T_i,t) - \frac{\omega(P_{[q]},t)}{\lvert\mathsf{ASSN}(P_{[q]},t)\rvert}. \qquad (4.1)$$

Notice that, in Figure 4.1, since the processor is over-utilized by 0.20 and there are four tasks assigned to it, MAOE is minimized by setting each task's guaranteed weight to be $\frac{0.2}{4}$ less than its desired weight.

It is worthwhile to note that (4.1) only produces non-negative guaranteed weights if the desired weight for each task is at least $\frac{\omega(P_{[q]},t)}{\lvert\mathsf{ASSN}(P_{[q]},t)\rvert}$. This condition is satisfied when $P_{[q]}$ is over-utilized by less than smallest desired weight of any task assigned to $P_{[q]}$. Since, as we discussed in Section 4.3, such a property can be guaranteed by any RAD partitioning

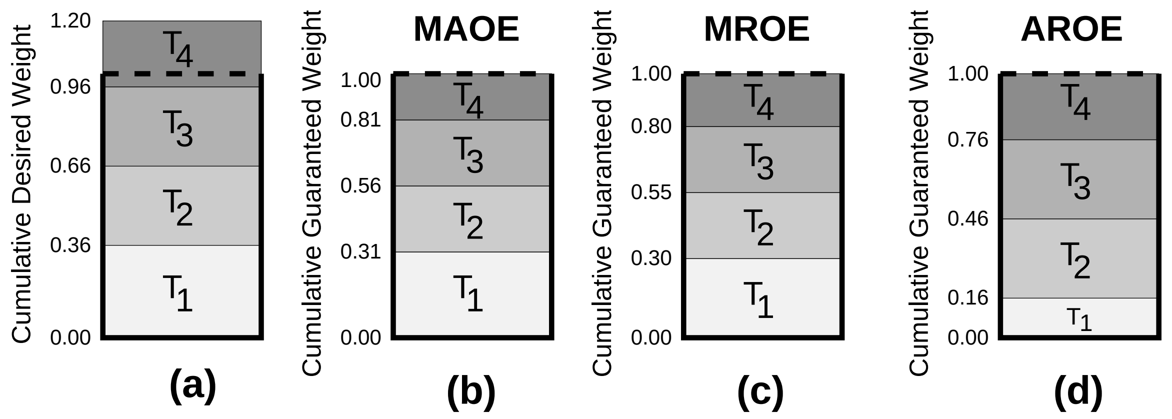


Figure 4.1: **(a)** The desired weights for four tasks assigned to one processor. Guaranteed weights for the four tasks when the guaranteed weights are chosen to minimize the **(b)** MAOE, **(c)** MROE and **(d)** AAOE metrics.

algorithm, it is possible to repartition the system in order to guarantee that (4.1) returns valid results. If repartitioning cannot be performed (e.g., for application-oriented reasons), then it is possible to use an iterative approach for determining the guaranteed weight of tasks.

The *average absolute overall error* (AAOE) on an over-utilized processor, $P_{[q]}$, is given by

$$\frac{1}{n} \cdot \sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \left(\mathsf{Dwt}(T_i,t) - \mathsf{Gwt}(T_i,t)\right).$$

It is easy to show that this metric is minimized whenever the guaranteed weight of all tasks assigned to a processor sum to 1. Since any reasonable method for determining the guaranteed weights of tasks will minimize the AAOE metric, this metric is of little value.

Relative error. The next two metrics we consider are based on the relative difference between the guaranteed and desired weights of a task. The *maximal relative overall error* (MROE) on an over-utilized processor, $P_{[q]}$, is given by

$$\mathsf{max}_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \left\{ \frac{\mathsf{Dwt}(T_i,t) - \mathsf{Gwt}(T_i,t)}{\mathsf{Dwt}(T_i,t)} \right\}.$$

This metric is minimized when all task shares are scaled by the same value. Specifically, we define the guaranteed weight of T_i assigned to the processor $P_{[q]}$ at time t as

$$\mathsf{Gwt}(T_i, t) = \frac{\mathsf{Dwt}(T_i, t)}{\mathcal{T}\mathsf{D}(P_{[q]}, t)}, \tag{4.2}$$

where

$$\mathcal{T}\mathsf{D}(P_{[q]}, t) = \mathsf{max}\left(1, \sum_{T_i \in \mathsf{ASSN}(P_{[q]}, t)} \mathsf{Dwt}(T_i, t)\right) \tag{4.3}$$

For example, consider the system in Figure 4.1(c). The depicted set of tasks over-utilizes the processor by 0.2, so each task's guaranteed weight should be $\frac{1}{1.2}$ times its desired weight. This scaling is the same as the *proportional-share* scaling used in EEVDF (Stoica et al., 1996).

The *average relative overall error* (AROE) on an over-utilized processor $P_{[q]}$ is given by

$$\frac{1}{n} \cdot \sum_{T_i \in P_{[q]}} \frac{\mathsf{Dwt}(T_i, t) - \mathsf{Gwt}(T_i, t)}{\mathsf{Dwt}(T, t)}.$$

This metric is minimized when the guaranteed weight for the heaviest task on an over-utilized processor $P_{[q]}$ is less than its desired weight by $\omega(P_{[q]}, t)$, and the guaranteed weight of every other task equals its desired weight. Specifically, the guaranteed weight of $T_i \in \mathsf{ASSN}(P_{[q]}, t)$ is defined as

$$\mathsf{Gwt}(T_i, t) = \begin{cases} \mathsf{Dwt}(T_i, t) - \omega(P_{[q]}, t), & \text{if } T_i \text{ is the heaviest task in } \mathsf{ASSN}(P_{[q]}, t) \\ \mathsf{Dwt}(T_i, t), & \text{otherwise.} \end{cases} \tag{4.4}$$

For example, consider the system depicted in Figure 4.1(d). In this system, since T_1 has the largest desired weight, its guaranteed weight is its desired weight minus the amount the processor is over-utilized, i.e., $0.36 - 0.20 = 0.16$.

Notice that (4.4) is valid only if $\omega(P_{[q]}, t) \leq \mathsf{Dwt}(T_H, t)$, where T_H is the heaviest task in $\mathsf{ASSN}(P_{[q]}, t)$. As with the MAOE metric, this condition is satisfied by repartitioning the system by using any RAD partitioning algorithm. If repartitioning cannot be performed (e.g., for application-oriented reasons), then it is possible to reduce the AROE by itierativly choosing the smallest possible guaranteed weight for tasks from the heaviest to lightest task in $\mathsf{ASSN}(P_{[q]}, t)$ until the total guaranteed weight equals one.

To simplify our discussion, we focus on minimizing MROE. It is worthwhile to note that under our adaptable framework, any method for determining the guaranteed weights of tasks on an over-utilized processor can be used so long as the sum of these weights is at most one. In Section 4.10, we explain how our system can be extended to accommodate any such method, including methods based the AROE and MAOE metrics.

4.4.2 The Adaptable Sporadic Task Model, Revisted

In this section, we incorporate the notions of guaranteed and desired weights into the adaptable sporadic task model that was presented in Section 3.1. It is important to note that, unless otherwise specified, all references to the adaptable sporadic task model in this chapter are to the task model presented in this section. We begin our discussion by formally defining when a task changes its weight. A task T_i *changes its desired weight* at time t if $\mathsf{Dwt}(T_i, t - \epsilon) \neq \mathsf{Dwt}(T_i, t)$ where $\epsilon \to 0^+$. Similarly, a task, T_i, *changes its guaranteed weight* at time t if $\mathsf{Gwt}(T_i, t - \epsilon) \neq \mathsf{Gwt}(T_i, t)$ where $\epsilon \to 0^+$.

Notice that, since we are attempting to minimize MROE, by (4.2), the guaranteed weight for every task on an over-utilized processor is a function of its desired weight and the desired weight of every other task assigned to the same processor. As a result, when a task, T_i, changes its desired weight at time t, the guaranteed weight for T_i and for every other task assigned to same processor may change.

Additionally, different actions may occur depending on whether the desired or guaranteed weight of a task changes. Specifically, if a task, T_i, changes its desired weight at time t_c when a job T_i^j is active, then the following two actions may occur.

- The execution time of T_i^j *may* be reduced to the amount of time for which T_i^j has executed prior to t_c, and the execution time of T_i^{j+1} *may* be redefined to be the amount of time "lost" by reducing the execution time of T_i^j.

- $\mathsf{r}(T_i^{j+1})$ *may* be redefined to be less than $\mathsf{d}(T_i^j)$, which would cause jobs T_i^j and T_i^{j+1} to "overlap."

If the guaranteed weight *but not the desired weight* of a task changes, then the following two actions may occur

- The deadline of T_i^j may be changed.
- The release time of T_i^{j+1} (if it exists) may be changed.

Scheduling weights. Just as with the adaptable sporadic task model presented in Section 3.1, there can be a difference between when a desired weight change is initiated and when it is enacted. We use the *desired scheduling weight of a task* T_i *at time* t, denoted $\mathsf{SDwt}(T_i, t)$, to represent the "last enacted desired weight of T_i." Formally, $\mathsf{SDwt}(T_i, t)$ equals $\mathsf{Dwt}(T_i, u)$, where u is the last time at or before t that a weight change was enacted for T_i. It is important to note that we use the desired scheduling weight of a task to compute the *guaranteed scheduling weight* of a task, denoted $\mathsf{SGwt}(T_i, t)$. In turn, the guaranteed scheduling weight is used to compute the deadlines and releases of tasks.

Formally, the guaranteed scheduling weight of T_i that is assigned to the processor $P_{[q]}$ at time t is

$$\mathsf{SGwt}(T_i, t) = \frac{\mathsf{SDwt}(T_i, t)}{\mathcal{TS}(P_{[q]}, t)}, \tag{4.5}$$

where

$$\mathcal{TS}(P_{[q]}, t) = \max\left(1, \sum_{T_i \in \mathsf{ASSN}(P_{[q]}, t)} \mathsf{SDwt}(T_i, t)\right) \tag{4.6}$$

Notice that, by (4.5), the sum of the guaranteed scheduling weights of all active tasks assigned to a processor is at most one, which is formalized by the following property

(W) For any processor $P_{[q]}$ and any time t, $\sum_{T_i^j \in \mathsf{ASSN}(P_{[q]}, t)} \mathsf{SGwt}(T_i, t) \leq 1$.

Because the rules for changing the guaranteed weight of a task are simpler than changing its desired weight, it is possible to integrate them directly into the definition of a job's release and deadline. To do so, we introduce the notion of a *perceived deadline* of T_i^j at time t, denoted $\mathsf{d}(T_i^j, t)$, which represents what the deadline of T_i^j would be if its guaranteed weight did not change. As a shorthand, we use $\mathsf{d}(T_i^j)$ to denote the time u such that $u = \mathsf{d}(T_i^j, u)$. ($\mathsf{d}(T_i^j)$ represents the actual deadline of the job, but it cannot be determined until it is reached (see the example below). Formally, we define $\mathsf{d}(T_i^j, t)$ as,

$$\mathsf{d}(T_i^j, t) = \begin{cases} t + \frac{\mathsf{Irem}(T_i^j, t)}{\mathsf{SGwt}(T_i, t)}, & \text{if } \mathsf{SGwt}(T_i, t) > 0 \\ \infty, & \text{otherwise.} \end{cases} \tag{4.7}$$

where

$$\mathsf{Irem}(T_i^j, t) = \begin{cases} \mathsf{e}(T_i^j), & \text{if } t < \mathsf{r}(T_i^j) \\ \mathsf{e}(T_i^j) - \int_{\mathsf{r}(T_i^j)}^{t} \mathsf{SGwt}(T_i, u)du, & \text{otherwise.} \end{cases} \tag{4.8}$$

Additionally, we define $\mathsf{r}(T_i^j)$ as,

$$\mathsf{r}(T_i^j) = \theta(T_i^j), \qquad j = 1 \tag{4.9}$$

$$\mathsf{r}(T_i^j) = \mathsf{d}(T_i^{j-1}) + \theta(T_i^j), \qquad j > 1, \tag{4.10}$$

where $\theta(T_i^j) \geq 0$. Notice that, if for some job T_i^j and $t > \mathsf{r}(T_i^j)$, $\mathsf{SGwt}(T_i, t) = 0$, then it is possible that $\mathsf{d}(T_i^j)$ cannot be reached since $\mathsf{d}(T_i^j, t) = \infty$. The scenario where $\mathsf{SGwt}(T_i, t) = 0$ is a special case that is used to represent T_i leaving the system. Thus, once $\mathsf{SGwt}(T_i, t) = 0$ holds, T_i cannot release any more jobs and is no longer allocated any capacity in any schedule. As a result, if $\mathsf{d}(T_i^j, t) = \infty$, then we set $\mathsf{d}(T_i^j)$ as ∞. Notice that all other terms of T_i are still well-defined since the only other term that is defined using $\mathsf{d}(T_i^j)$ is $\mathsf{r}(T_i^{j+1})$; however, if $\mathsf{d}(T_i^j) = \infty$, then T_i^{j+1} does not exist.

From the definition of $\mathsf{d}(T_i^j, t)$ in (4.7) it is not hard to see that the following property holds.

(D) For any two times u_1 and u_2 such that $\mathsf{r}(T_i^j) \leq u_1 \leq u_2 \leq \mathsf{d}(T_i^j)$, $\int_{u_1}^{u_2} \mathsf{SGwt}(T_i, t)dt \leq \mathsf{e}(T_i^j)$.

Notice that, since for $t \in [\mathsf{r}(T_i^j), \mathsf{d}(T_i^j))$, $\int_{\mathsf{r}(T_i^j)}^{t} \mathsf{SGwt}(T_i, u) \leq \mathsf{e}(T_i^j)$, by (4.8), it follows that for any time t

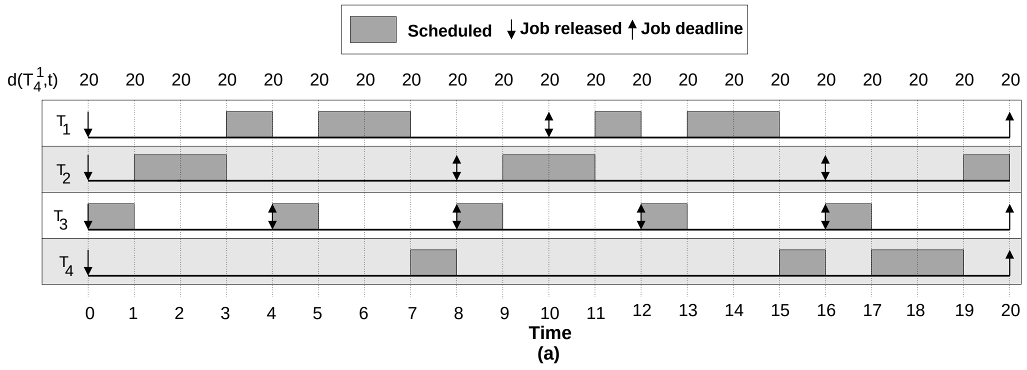
$$\mathsf{Irem}(T_i^j, t) \leq \mathsf{e}(T_i^j) \leq \mathsf{e}_{\mathsf{max}}(T_i) \tag{4.11}$$

It is important to note that the PEDF and NP-PEDF scheduling algorithms discussed in this chapter *prioritize jobs based on their perceived deadlines*; however, even though jobs are prioritized based on their perceived deadlines, the relative scheduling priority between any two jobs is time invariant. Specifically, it two jobs, T_i^j and T_a^b, are assigned to the same processor and at some time $t_1 \geq \mathsf{max}(\mathsf{r}(T_i^j), \mathsf{r}(T_a^b))$, $\mathsf{d}(T_i^j, t_1) < \mathsf{d}(T_a^b, t_1)$ holds, then at any time $t \geq \mathsf{max}(\mathsf{r}(T_i^j), \mathsf{r}(T_a^b))$, $\mathsf{d}(T_i^j, t) < \mathsf{d}(T_a^b, t)$ holds. Intuitively, the reason for this behavior is that the deadlines for both jobs always scale by the same factor, i.e., $\frac{1}{\mathcal{TS}(P_{[q]}, t)}$. Also, when the desired weight of a task changes, the relative scheduling priority between any two released jobs of any two tasks is unchanged. (When task's desired weight is changed, it is possible that the current job may halt and release a new job with a higher or lower scheduling priority. Such an action would change the relative scheduling priority between *two tasks* but not between the jobs themselves because the original job was halted.)

Example (Figure 4.2). Consider the example in Figure 4.2, which depicts a processor that is assigned four tasks: T_1, which has $\mathsf{e}(T_1) = 3$ and $\mathsf{Dwt}(T_1) = 2/5$; T_2, which has $\mathsf{e}(T_2) = 2$ and $\mathsf{Dwt}(T_2) = 1/3$; T_3, which has $\mathsf{e}(T_3) = 1$ and $\mathsf{Dwt}(T_3) = 1/3$; and T_4, which has $\mathsf{e}(T_4) = 4$ and $\mathsf{Dwt}(T_4) = 4/15$. The total desired weight is $\frac{4}{3}$. T_4^1's perceived deadline is shown above each inset. Inset (a) depicts the scenario where T_1 never leaves. Inset (b) depicts the scenario where T_1 leaves at time 10 causing the processor to be under-utilized. Notice that, in inset (b), T_4^1's perceived deadline changes when T_1 leaves (at time 10) from 20 to $18.\bar{3}$ because T_4^1's guaranteed scheduling weight changes. This differs from inset (a), in which T_4^1's guaranteed scheduling weight does not change, and as a result, T_4^1's perceived deadline remains constant. One final note: while, in inset (a), $\mathsf{d}(T_4^1) = 20$, and in inset (b), $\mathsf{d}(T_4^1) = 18.\bar{3}$, neither of these values are known until these corresponding points in time are reached. $\square$

Because the reweighting rules may cause $\mathsf{r}(T_i^{j+1}) < \mathsf{d}(T_i^j)$, we must slightly modify the definition of "window," "active," and "inactive" presented in Section 1.2.

Definition 4.1 (Window, Active, and Inactive). If T_i^j is a job in the adaptable sporadic task system, T, then the *window* of T_i^j is defined as the range $[\mathsf{r}(T_i^j), \mathsf{min}(\mathsf{d}(T_i^j), \mathsf{r}(T_i^{j+1})))$. Furthermore, the job T_i^j is *active at time t* iff t is in T_i^j's window (i.e., $t \in [\mathsf{r}(T_i^j), \mathsf{min}(\mathsf{d}(T_i^j), \mathsf{r}(T_i^{j+1})))$), and is *inactive* otherwise.



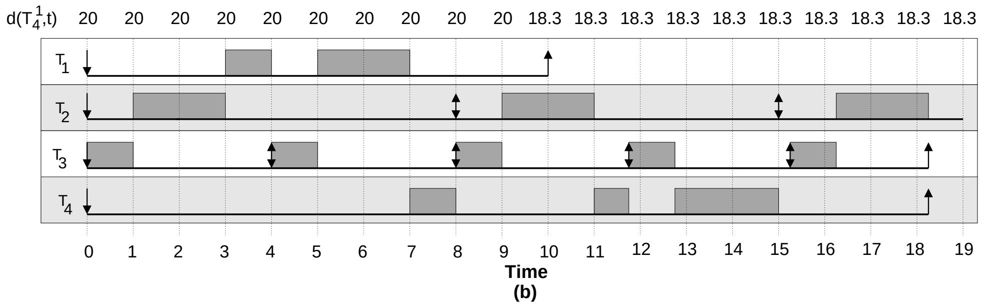


Figure 4.2: A one-processor system with four tasks. **(a)** T_1 never leaves. **(b)** T_1 leaves at time 10. The perceived deadline for T_4^1 is shown above each figure.

4.4.3 Modifying the SW Algorithm

Having extended the adaptable sporadic task model presented in Section 3.1, we now extend the SW theoretical algorithm, presented in Section 3.2, to incorporate desired and guaranteed weights. Under the SW scheduling algorithm, at each instant t, each active job T_i^j in τ is allocated a fraction of the system equal to its guaranteed scheduling weight $\mathsf{SGwt}(T_i, t)$. Hence, if a job T_i^j is active over the range $[t_1, t_2)$, then over this range, T_i^j is allocated $\int_{t_1}^{t_2} \mathsf{SGwt}(T_i, u)du$ time. Throughout this chapter, we use $\mathcal{S}W$ to denote the SW schedule of a task system τ.

Deviance. The *deviance of the job T_i^j of the task T_i at time t* is defined as

$$\mathsf{dev}(T_i^j,\, t) = \mathsf{A}(\mathcal{S}W,\, T_i^j,\, 0,\, t) - \mathsf{A}(\mathcal{S},\, T_i^j,\, 0,\, t), \tag{4.12}$$

where $\mathcal{S}$ is the actual schedule.

Example (Figure 4.3). Consider the example in Figure 4.3, which depicts one processor that is assigned four tasks: T_1, which has $\mathsf{e}(T_1) = 1$ and $\mathsf{Dwt}(T_1) = 1/3$; T_2, which has $\mathsf{e}(T_2) = 1$ and $\mathsf{Dwt}(T_2) = 1/6$; T_3, which has $\mathsf{e}(T_3) = 2$, $\mathsf{Dwt}(T_3) = 1/4$, and leaves at time 8; and T_4, which has $\mathsf{e}(T_4) = 4$ and an initial desired weight of 1/4 that increases to 1/2 at time 8. (The reweighting rules presented in Section 4.5 stop T_4^1 from receiving more than one unit of allocation and cause T_4^2 to be released with the remaining three units of execution at time 8.) Inset (a) depicts the PEDF schedule. Inset (b) depicts the SW schedule. Inset (c) depicts the allocations to T_4 in the PEDF and SW schedules. Notice that T_4^1 has positive deviance at time 8. □

Example (Figure 4.4). Consider the example in Figure 4.4, which depicts one processor that is assigned four tasks (where the execution time of each job is one): T_1, which has $\mathsf{Dwt}(T_1) = 1/2$; T_2 and T_3, both of which have a desired weight of 1/6; and T_4, which has an initial desired weight of 1/2 that initiates a desired weight decrease to 1/6 at time 1 that is enacted at time $2.\bar{6}$. (The reweighting rules presented in Section 4.5 do not allow T_2 to decrease its desired weight immediately.) Notice that T_4^1 has negative deviance at time 1. □

4.5 Changing Desired Weights

We now introduce two rules for changing the desired weight of a task. (These rules are similar to reweighting rules for changing the weight of a task under GEDF that were presented in Section 3.4.) These rules work by modifying future release times and deadlines. (The rules below are applied on a single processor; reweighting events that trigger a repartitioning are discussed in Section 4.6.) We first describe how to change the desired weight of a task under PEDF in Section 4.5.1 and then explain how to change the desired weight of a task under NP-PEDF in Section 4.5.2.

4.5.1 Changing Desired Weights in PEDF

Let τ be a task system in which some task T_i initiates a desired weight change to Nw at time t_c. Let Ow be the last desired scheduling weight of T_i before the change is initiated at t_c. Let

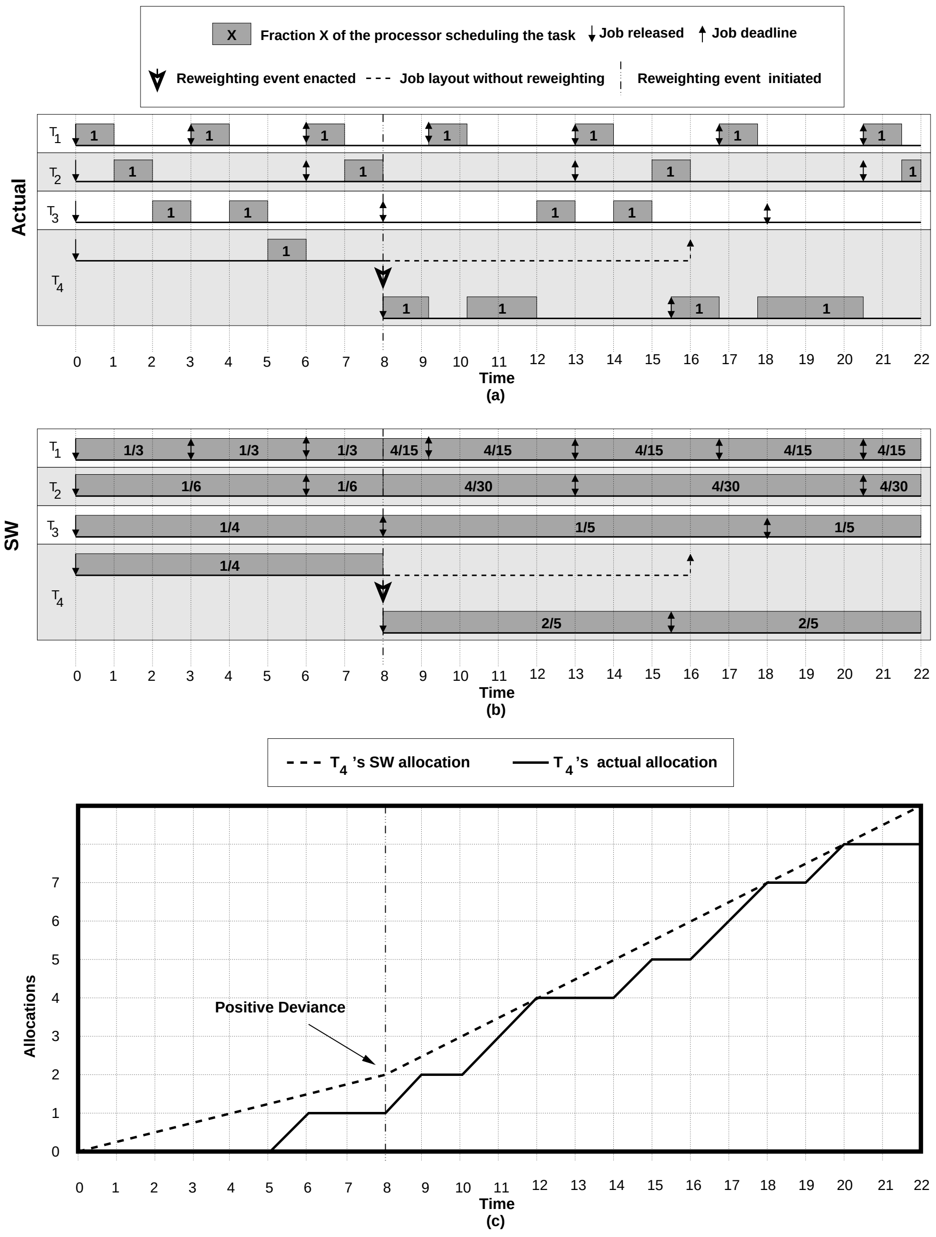


Figure 4.3: A one-processor system with four tasks. **(a)** The PEDF schedule. **(b)** The SW schedule. **(c)** The allocations to T_4 in the PEDF and SW schedules.

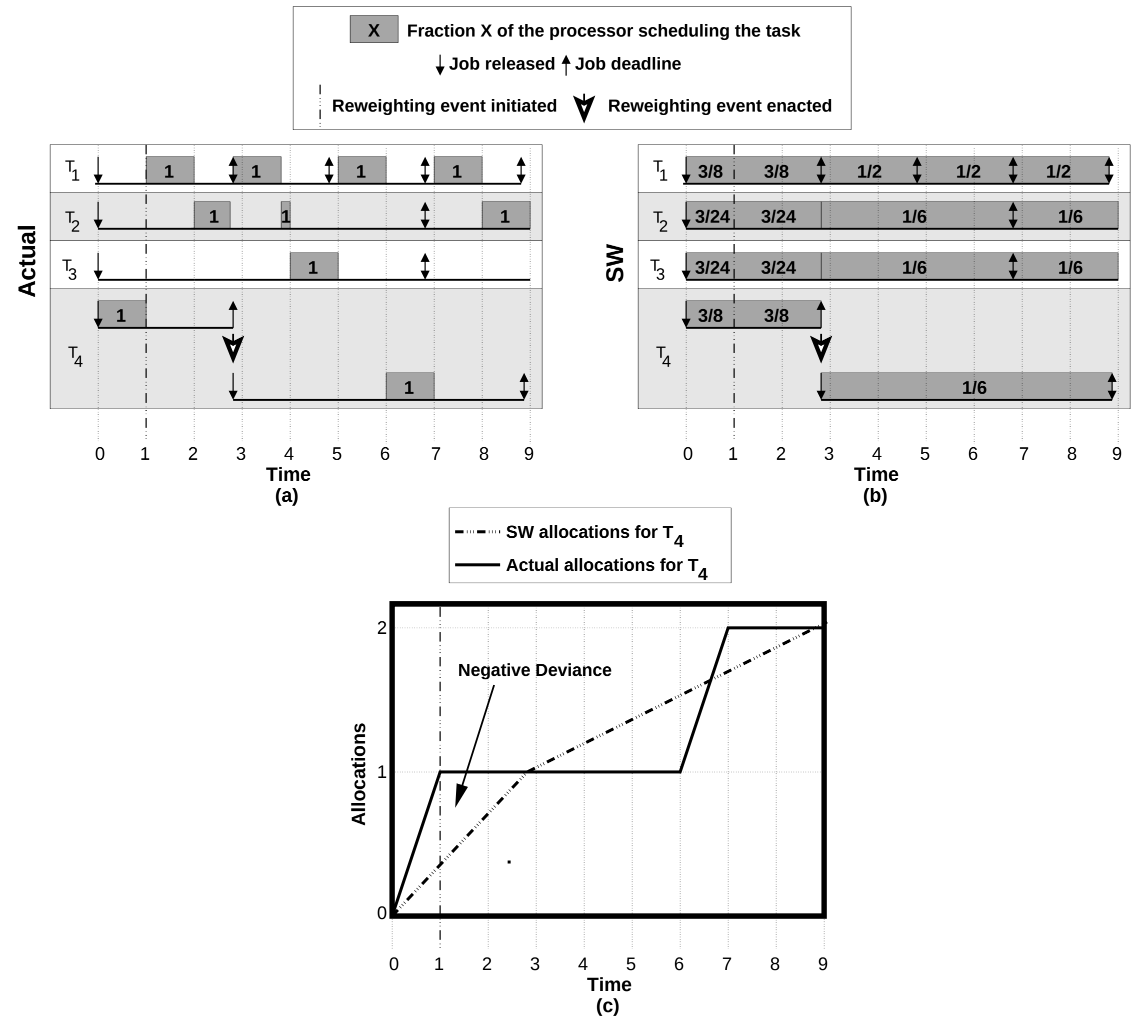


Figure 4.4: A one-processor system with four tasks. **(a)** The PEDF schedule. **(b)** The SW schedule. **(c)** The allocations to T_4 in the PEDF and SW scheduling algorithms.

$\mathcal{S}$ be the M-processor PEDF schedule of τ. Let T_i^j be last-released job of T_i before t_c. If T_i^j does not exist or T_i^j is inactive at t_c before the reweighting event is initiated (i.e., $t_c \geq \mathsf{d}(T_i^j)$), then the desired weight change is immediately enacted, and future jobs of T_i are released with the new desired weight. In the following rules, we consider the remaining possibility, i.e., T_i^j exists and is active at t_c. (Notice that, if $t_c = \mathsf{d}(T_i^k) = \mathsf{r}(T_i^{k+1})$, then T_i^k is the last-released job of T_i before t_c, and it is not active at t_c. Therefore, the change is immediately enacted and T_i^{k+1} is released with the new desired weight.)

Let $\mathsf{REM}(T_i^j, t) = \mathsf{e}(T_i^j) - \mathsf{A}(\mathcal{S},\, T_i^j,\, 0,\, t)$. Note that $\mathsf{REM}(T_i^j, t)$ denotes the actual remaining computation in T_i's current job. Since for any time t, $\mathsf{A}(\mathcal{S},\, T_i^j,\, 0,\, t) \geq 0$,

$$\mathsf{REM}(T_i^j, t) \leq \mathsf{e}(T_i^j) \leq \mathsf{e}_{\mathsf{max}}(T_i). \tag{4.13}$$

Let $\mathsf{nextE}(T_i^j, t_c)$ equal $\mathsf{REM}(T_i^j, t_c)$, if $\mathsf{REM}(T_i^j, t_c) > 0$; otherwise, if $\mathsf{REM}(T_i^j, t_c) = 0$, then let $\mathsf{nextE}(T_i^j, t_c)$ equal the value of $\mathsf{e}(T_i^{j+1})$ had the desired weight change event not occurred. Since $\mathsf{nextE}(T_i^j, t_c)$ is only used to determine the execution time of the next job released, it can be calculated at time $\mathsf{r}(T_i^{j+1})$. (Notice that, if T_i has no next job T_i^{j+1} to release at the time specified in the rules below, then $\mathsf{nextE}(T_i^j, t_c) = 0$. In this case, the rules are applied as stated, except that T_i^{j+1} is not released.)

As was the case for reweighting under GEDF, the choice of which rule to apply depends on whether deviance is positive or negative. If positive, then we say that T_i is *positive-changeable at time t_c from a desired weight of* Ow *to* Nw; otherwise T_i is *negative-changeable at time t_c from a desired weight of* Ow *to* Nw. Because T_i initiates its desired weight change at time t_c, $\mathsf{Dwt}(T_i, t_c) = \mathsf{Nw}$ holds; however, T_i's desired scheduling weight does not change until the desired weight change has been *enacted*, as specified in the rules below. Note that, if t_c occurs between the initiation and enaction of a previous reweighting event of T_i, then the previous event is *canceled*, i.e., treated as if it had not occurred. As discussed later, any "error" associated with canceling a reweighting event like this is accounted for when determining drift (formally defined in Section 4.9).

Rule P: If T_i is positive-changeable at time t_c from a desired weight of Ow to Nw, then one of two actions is taken: **(i)** if $\frac{\mathsf{Irem}(T_i^j,t_c)}{\mathsf{Ow}} > \frac{\mathsf{REM}(T_i^j,t_c)}{\mathsf{Nw}}$, then immediately, T_i^j is halted, the desired weight change is enacted, a new job T_i^{j+1} with an execution time of $\mathsf{nextE}(T_i^j, t_c)$ is released (if $\mathsf{nextE}(T_i^j, t_c) > 0$), and T_i^j becomes inactive; **(ii)** otherwise, at time $\mathsf{d}(T_i^j)$, the desired weight change is enacted, i.e., the desired scheduling weight of T_i does not change until the end of its current job.

Rule N: If T_i is negative-changeable at time t_c from a desired weight of Ow to Nw, then one of two actions is taken: **(i)** if $\mathsf{Nw} > \mathsf{Ow}$, then immediately, T_i^j is halted and its desired weight change is enacted, and at time t_r, a new job T_i^{j+1} with an execution time of $\mathsf{nextE}(T_i^j, t_c)$ is released (if $\mathsf{nextE}(T_i^j, t_c) > 0$) and T_i^j becomes inactive, where t_r is the smallest time at or after t_c such that $\mathsf{dev}(T_i^j,\, t_r) = 0$ holds; **(ii)** otherwise, at time t_e, the desired weight change is enacted, a new job with an execution time of $\mathsf{nextE}(T_i^j, t_c)$ is released (if $\mathsf{nextE}(T_i^j, t_c) > 0$), and T_i^j becomes inactive, where $t_e = \mathsf{min}(t_r, \mathsf{d}(T_i^j))$, and t_r is smallest time at or after t_c such that $\mathsf{dev}(T_i^j,\, t_r) = 0$ holds.

Intuitively, Rule P changes a task's desired weight by halting its current job and issuing a new job with an execution time of $\mathsf{nextE}(T_i^j, t_c)$ with the new desired weight if doing so would improve its scheduling priority. Notice that, at time t, job T_i^j has a higher scheduling priority than job T_ℓ^w if

$$\frac{\mathsf{Irem}(T_i^j, t_c)}{\mathsf{SGwt}(T_i, t_c)} < \frac{\mathsf{Irem}(T_\ell^w, t_c)}{\mathsf{SGwt}(T_\ell, t_c)}.$$

Hence, it is not difficult to show that if $\frac{\mathsf{Irem}(T_i^j,t_c)}{\mathsf{Ow}} > \frac{\mathsf{REM}(T_i^j,t_c)}{\mathsf{Nw}}$ holds, then halting T_i^j and issuing a new job with an execution time of $\mathsf{nextE}(T_i^j, t_c)$ would improve T_i's scheduling priority.

Example (Figure 4.5). Consider the example of Case (i) of Rule P illustrated in Figure 4.5, which depicts one processor that is assigned four tasks (where the execution cost of each job is one): T_1, which has $\mathsf{Dwt}(T_1) = 1/2$ and leaves at time at time 2; T_2 and T_3, both of which have a desired weight of 1/6; and T_4, which has an initial desired weight of 1/6 that increases to 4/6 at time 2. In this system, T_4 initially has the lowest scheduling priority (there is a deadline tie). Inset (a) depicts the PEDF schedule. Inset (b) depicts T_4's allocations in the SW schedule as well three other schedules, namely, the CSW, IDEAL, and PT schedules, which are defined later in Sections 4.7.1 and 4.8. Since T_4 is not scheduled by time 2, it has positive deviance and changes its weight via Rule P, causing T_4^1 to be halted, T_4^2 to be released at time 2 with a deadline of 7/2, and T_4's drift to become 2/6. Note that halting T_4's current job and issuing a new job with an execution time of one improves T_4's scheduling priority, i.e., $\frac{\mathsf{Irem}(T_4^1,2)}{\mathsf{Ow}} = \frac{4/6}{1/6} = 4 > 6/4 = \frac{1}{4/6} = \frac{\mathsf{REM}(T_4^1,2)}{\mathsf{Nw}}$. Notice also that the third job of T_4 is issued 6/4 time units after time 2. This spacing is in keeping with a new job of desired weight 4/6 issued at time 2. $\square$

Example (Figure 4.6). Consider the example of Case (ii) of Rule P illustrated in Figure 4.6, which depicts one processor that is assigned three tasks (where the execution cost of each job is one): T_1, which has $\mathsf{Dwt}(T_1) = 1/3$; T_2, which has $\mathsf{Dwt}(T_2) = 1/4$; and T_3, which has an initial desired weight of 1/4 that initiates an increase to 1/3 at time 2. Inset (a) depicts the PEDF schedule. Inset (b) depicts T_3's allocations in the SW schedule. (Again, the CSW, IDEAL, and PT scheduling algorithms are defined later in Sections 4.7.1 and 4.8.) Since T_1^1 has not been scheduled by time 2, its deviance is positive; furthermore, since $\frac{\mathsf{Irem}(T_3^1,2)}{\mathsf{Ow}} = \frac{.50}{.25} = 2 < 3 = \frac{1}{1/3} = \frac{\mathsf{REM}(T_3^1,2)}{\mathsf{Nw}}$, T_1 enacts its weight change via Case (ii) of Rule P. Notice that, if T_3^1 had been halted at time 2 and released a new job of desired weight 1/3, its scheduling priority would be decreased. Thus, if we were to enact the change via Case (i) of Rule P, then we would in effect be *increasing* the deadline of the first scheduled job of T_3, even though the desired weight of the task increased. $\square$

Rule N changes the desired weight of a task by one of two approaches: **(i)** if a task *increases* its desired weight, then Rule N causes the release time of its next job to be adjusted so that it is commensurate with the new desired weight; **(ii)** if a task *decreases* its desired weight, then Rule N causes the next job to be issued with a deadline that is commensurate with the new desired weight at the end of the current job.

Example (Figure 4.7). Consider the example of Case (i) of Rule N illustrated in Figure 4.7, which depicts the same system as in Figure 4.5, except that T_4 has the highest priority. Inset (a) depicts the PEDF schedule. Inset (b) depicts T_3's allocations in the SW schedule. (Again, the CSW, IDEAL, and PT scheduling algorithms are defined later in Sections 4.7.1

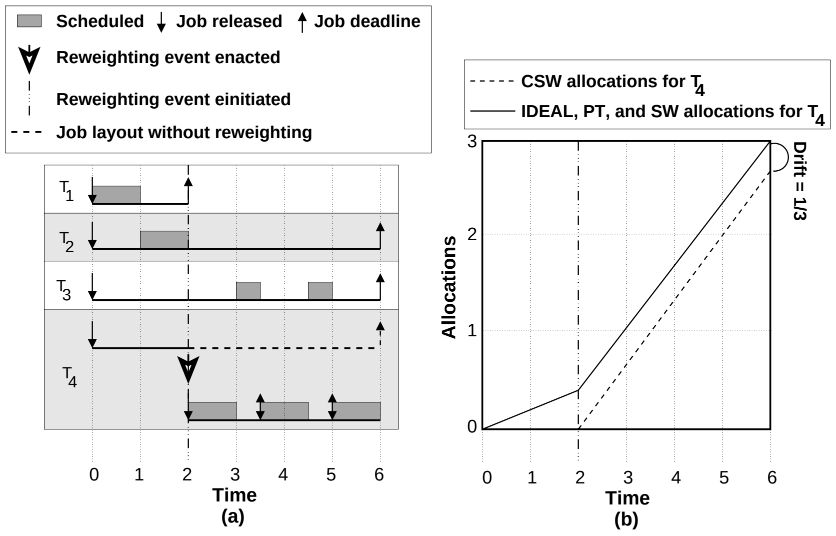


Figure 4.5: An illustration of reweighting via Case (i) of Rule P under PEDF. **(a)** The PEDF schedule for a one-processor systems with four tasks. (T_4 has the lowest scheduling priority.) **(b)** T_4's allocations in the IDEAL, CSW, PT, and SW schedules.

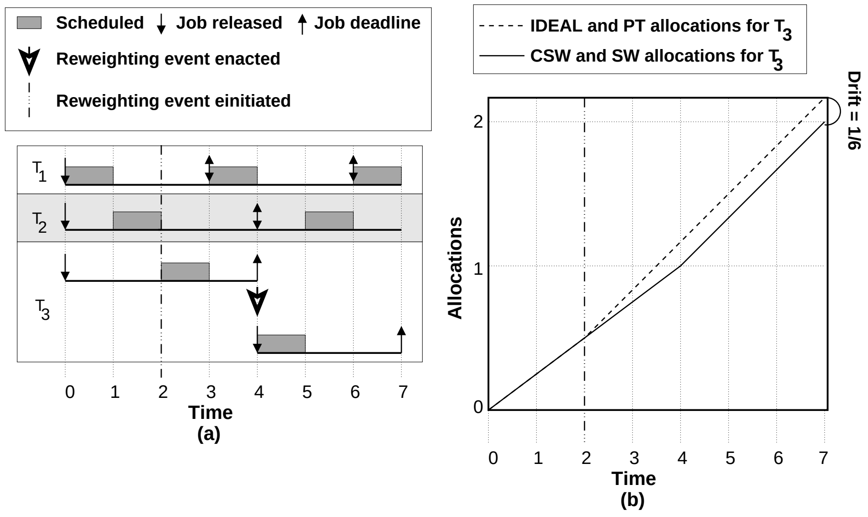


Figure 4.6: An illustration of reweighting via Case (ii) of Rule P under PEDF. **(a)** The PEDF schedule for a one-processor systems with four tasks. **(b)** T_3's allocations in the IDEAL, CSW, PT, and SW schedules.

and 4.8.) Notice that the second job of T_4 is released at time 3, which is the time such that $\mathsf{dev}(T_4, 3) = \int_0^3 \mathsf{SGwt}(T_i, u)du - \mathsf{A}(\mathcal{S}, T_4, 0, 3) = 1 - 1 = 0$. □

Example (Figure 4.8). Consider the example of Case (ii) of Rule N illustrated in Figure 4.8, which depicts a one-processor systems with four tasks (where the execution cost of each job is one): T_1, which has $\mathsf{Dwt}(T_1) = 1/2$ and joins the system at time 2; T_2 and T_3, both of which have a desired weight of 1/6; and T_4, which has an initial desired weight of 1/2 that initiates a weight decrease to 1/6 at time 1 that is enacted at time 2. Inset (a) depicts the PEDF schedule. Inset (b) depicts T_4's allocations in the SW schedule. (Again, the CSW, IDEAL, and PT scheduling algorithms are defined later in Sections 4.7.1 and 4.8.) Since T_4 has negative deviance at time 1, and it decreases its desired weight, this change is enacted via Case (ii) of Rule N, causing T_4's next job to have a deadline of 8 and T_4 to have a drift of $-1/3$. □

It is important to remember that when the Rules P and N halt a job, they do not abandon the computation that the job was performing. Rather, these rules split that computation across two jobs. Since these rules change the ordering of a task in the priority queues that determine scheduling, the time complexity for reweighting one task is $O(logN)$, where N is the number of tasks in the system (assuming that binomial heaps are used to implement the priority queues).

Canceled reweighting events. We now introduce a property about the relationship between the initiation and enactment of a desired weight change in the case that some such changes are canceled due to later desired weight changes. Notice that, once a task T_i initiates a desired weight change at t_c, this desired weight change is eventually either canceled by another desired weight change or enacted. Further, Rules P and N enact any non-canceled desired weight change no later than the deadline of the last-released job T_i^j of T_i at t_c (if it exists and if $t_c \leq \mathsf{d}(T_i^j)$).

Example (Figure 4.9). Consider the example in Figure 4.9, which depicts one processor that is assigned three tasks: T_1 and T_2, both of which have an execution time of 2 and a weight of 1/3; and T_3, which has $\mathsf{e}(T_3) = 2$ and an initial weight of 1/3 that changes to 1/10

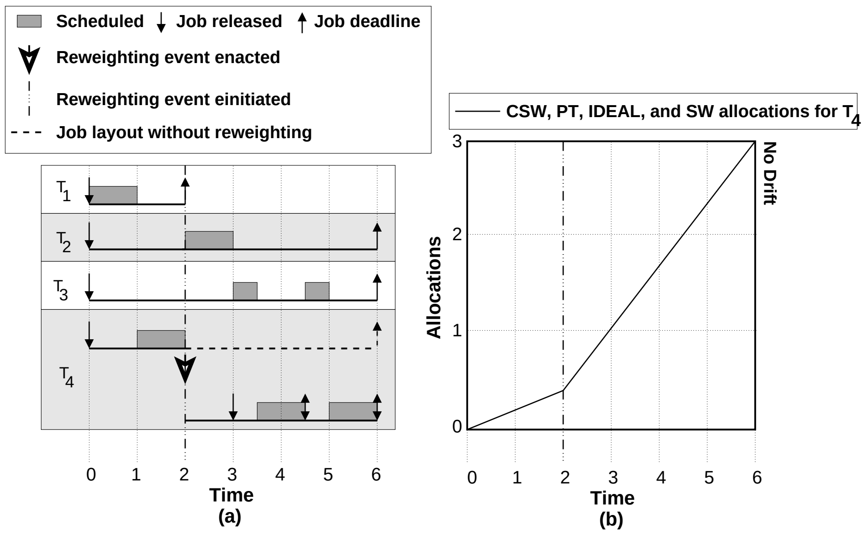


Figure 4.7: An illustration of reweighting via Case (i) of Rule N under PEDF. **(a)** The PEDF schedule for a one-processor systems with four tasks. (T_4 has the highest scheduling priority.) **(b)** T_4's allocations in the IDEAL, CSW, PT, and SW schedules.

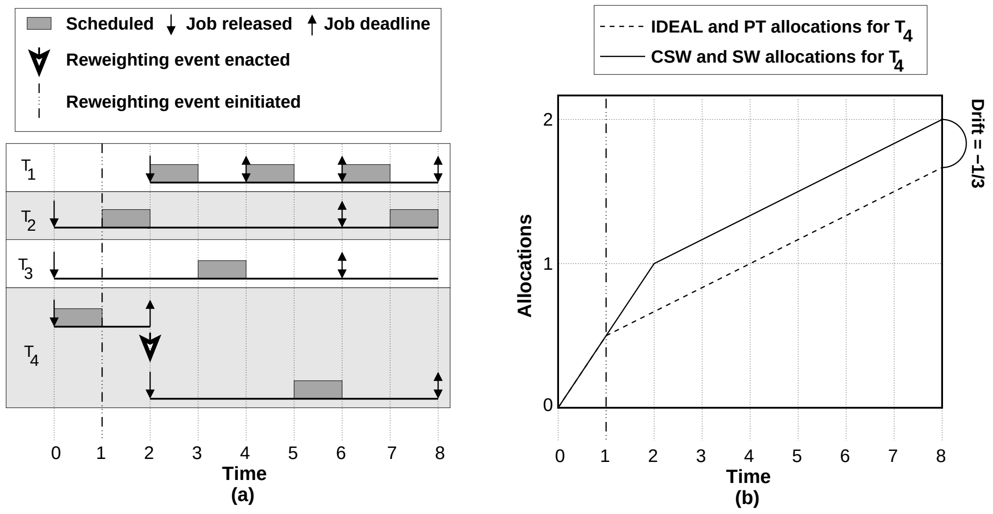


Figure 4.8: An illustration of reweighting via Case (ii) of Rule N under PEDF. **(a)** The PEDF schedule for a one-processor systems with four tasks. **(b)** T_4's allocations in the IDEAL, CSW, PT, and SW schedules.

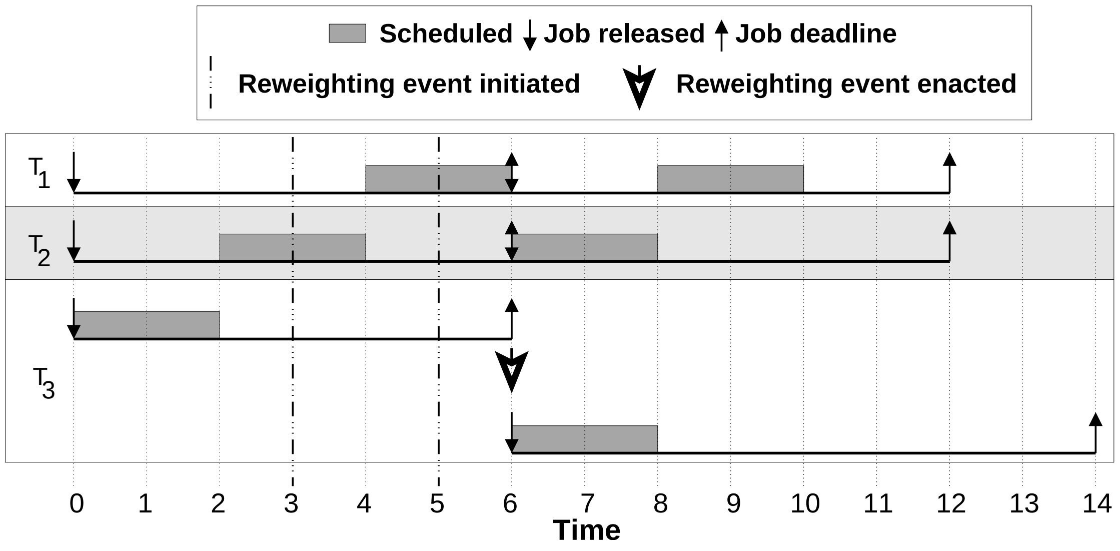


Figure 4.9: A one-processor example of canceling a reweighting event.

at time 3 via Case (ii) of Rule N and then to 1/4 at time 5 via Case (ii) of Rule N. Notice that, because the change initiated at time 3 is via Case (ii) of Rule N, the change is not enacted until time 6. As a result, when a change is initiated at time 5, this new change cancels the previous change. Even though the change initiated at time 3 is canceled, the time of the next weight enactment is still at time 6. □

From this example, we can see that, once a desired weight change has been initiated during an active job, some desired weight change will be enacted by the earlier of the deadline of that job or when the job becomes inactive (which may be earlier, by Rules P and N). Property (X) formalizes this idea.

(X) If a task T_i initiates a desired weight change at time t_c and the job T_i^j is active at t_c, then some desired weight change is enacted according to Rule P or N by either $\mathsf{d}(T_i^j)$ or when T_i^j becomes inactive, whichever is first.

4.5.2 Modifications for NP-PEDF

In order to adapt Rules P and N to work for NP-PEDF, the only modification we need to make is when these rules are *initiated*. If a task with an active job changes its desired weight *before or after* that job has been scheduled, then Rules P and N are initiated as before. (Note that, if the active job *has not* been scheduled, then its deviance is positive, and if the active

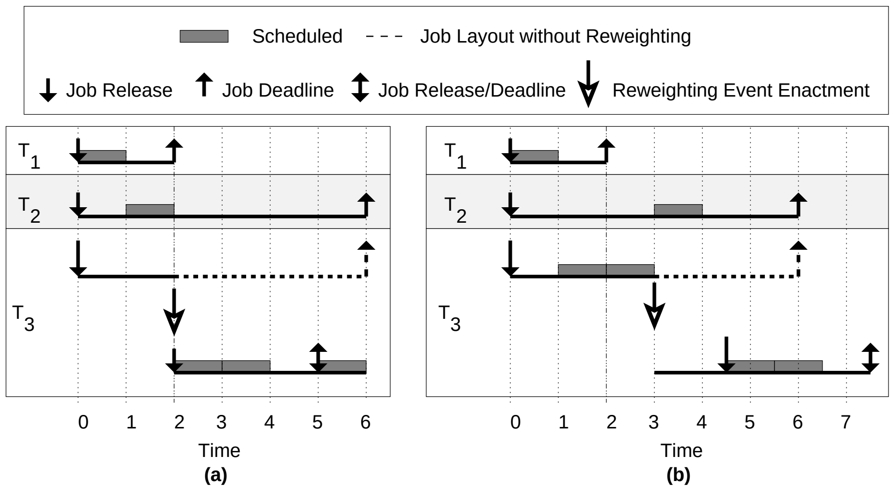


Figure 4.10: A one-processor example of reweighting under NP-PEDF. **(a)** A reweighting event is initiated before T_3^1 is scheduled. (T_3^1 has a lower scheduling priority than T_2^1.) **(b)** A reweighting event is initiated while T_3^1 is being scheduled. (T_3^1 has a lower scheduling priority than T_2^1.)

job *has* been scheduled, then its deviance is negative.) However, if a task changes its desired weight while the active job T_i^j is executing, then the initiation of the desired weight change is delayed *until* T_i^j *has completed* or T_i^j *is no longer active*, whichever is first. Note that, if a task T_i changes its desired weight from Ow to Nw at time t_c in NP-PEDF, then $\mathsf{Dwt}(T_i, t_c) = \mathsf{Nw}$ holds, regardless of whether the initiation of Rule P or N must be delayed.

Example (Figure 4.10). Consider the example in Figure 4.10, which depicts one processor that is assigned three tasks: T_1, which has $\mathsf{e}(T_1) = 1$, $\mathsf{Dwt}(T_1) = 1/2$, and leaves at time 2; T_2, which has $\mathsf{e}(T_2) = 1$ and $\mathsf{Dwt}(T_2) = 1/6$; and T_3, which has $\mathsf{e}(T_3) = 2$ and an initial desired weight of 1/3 and initiates an increases to 4/6 at time 2. Inset (a) depicts the scenario where T_3^1 has the lowest scheduling priority (there is a deadline tie). Since T_3 is not scheduled by time 2, it has positive deviance and changes its weight via Rule P, causing T_3^1 to be halted, and T_3^2 to be released at time 2 with a deadline of 5. Inset (b) depicts the same scenario as in (a) except that T_3 has higher priority than T_2. Since T_3 is scheduled at time 2, and the system is schedule by NP-PEDF, the initiation of the reweighting event is delayed until T_3 stops executing at time 3. Since T_3^1 is complete by time 2, it has negative deviance and changes its weight via Rule N, causing its next job to have a release time of 9/2. □

4.6 Resetting Rules

In this section, we formally define the rules for resetting a system. Let τ be a task system in which the system is reset at time t_c. Let $\mathcal{S}$ be an M-processor PEDF schedule of τ. Let T_i^j be the last-released job (if any) of some task T_i before t_c. Let $P_{[q]}$ and $P_{[\ell]}$ be, respectively, the processor that T_i is assigned to before and after the system is reset at time t_c. Then, there are two possibilities. First, T_i^j exists and is active at t_c, (e.g., $\mathsf{r}(T_i^j) \le t_c < \mathsf{d}(T_i^j)$). Second, T_i^j either is inactive at t_c or does not exist (because T_i^1 is not released until after t_c). Thus, we have the following rule, with two cases, for resetting a task

Rule R: **(i)** If T_i^j exists and is active at t_c, then T_i^j is halted, and T_i^{j+1} is released immediately on $P_{[\ell]}$, with an execution time of $\mathsf{nextE}(T_i^j, t_c)$. **(ii)** If T_i^j either does not exist or is inactive at t_c, then the next job of T_i is released at $t_c + \theta(T_i^{j+1})$ on $P_{[\ell]}$ (or at $t_c + \theta(T_i^1)$, if T_i^j does not exist).

Notice that, by this rule, whenever a system is reset, all active jobs are halted, and new jobs are released for the newly repartitioned system. Thus, if some task T_a is assigned to $P_{[z]}$ when T_a^b is released, then T_a^b will only be scheduled on $P_{[z]}$.

Example (Figure 4.11). Consider the example in Figure 4.11, which depicts a two-processor system that is assigned four tasks: T_1, which has $\mathsf{e}(T_1) = 1$ and an initial desired weight of 1/2 that decreases to 1/4 at time 2; T_2, which has $\mathsf{e}(T_2) = 2$ and $\mathsf{Dwt}(T_2) = 1/2$; T_3, which has $\mathsf{e}(T_3) = 3$ and an initial desired weight of 1/2 that increases to 3/4 at time 2; and T_4, which has $\mathsf{e}(T_4) = 4$ and $\mathsf{Dwt}(T_4) = 1/2$. Initially, T_1 and T_2 are assigned to the first processor and T_3 and T_4 are assigned to the second. Inset (a) depicts the PEDF schedule (the other insets are considered later). At time 2, the system is repartitioned, i.e., is reset, and T_1 and T_3 are assigned to one processor and T_2 and T_4 are assigned to the other processor. $\square$

Complications with NP-PEDF. Under NP-PEDF, repartitioning is slightly more complex since a job cannot be preempted if it is currently being scheduled. Thus, the entire system cannot be reset at the same time. Thus, there are two viable options for reseting an NP-PEDF-scheduled system. First, whenever the system is reset, ignore non-preemptive behavior, and

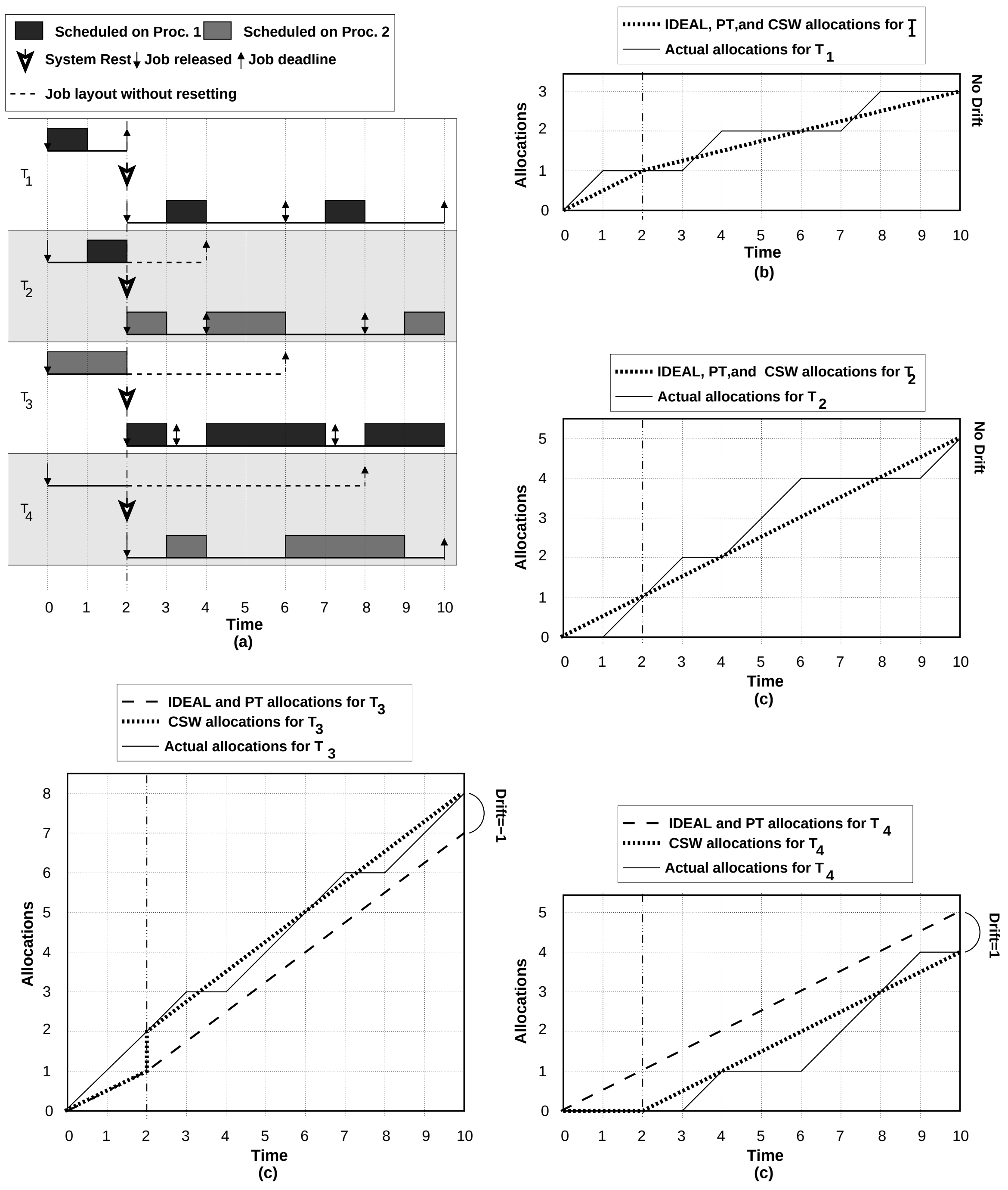


Figure 4.11: **(a)** An illustration of resetting in PEDF. Insets **(b)**–**(e)** depict, respectively, the allocations for T_1, ..., T_4.

Property	Definition
(SW-1) [Page 127]	If the system is not reset over the range $(t_1, t_2]$, then the function $\mathsf{A}(\mathcal{CSW}, T_i, 0, t)$ is continuous for any $t \in [t_1, t_2]$.
(SW-2) [Page 127]	For any job T_i^j and any $t \geq \mathsf{r}(T_i^j)$, $\mathsf{A}(\mathcal{CSW}, T_i^j, \mathsf{r}(T_i^j), t) \leq \mathsf{Ae}(T_i^j) \leq \mathsf{e}(T_i^j)$.
(SW-3) [Page 127]	Any job T_i^j is complete in the CSW schedule by time $\mathsf{d}(T_i^j)$.
(V) [Page 128]	For the jobs T_i^j and T_i^{j+1}, if $\mathsf{d}(T_i^j) > \mathsf{r}(T_i^{j+1})$, then T_i^j is complete in both the CSW and actual schedules by time $\mathsf{r}(T_i^{j+1})$.
(L-1) [Page 129]	For any time $t \leq \mathsf{r}(T_i^j)$, $\mathsf{lag}(T_i^j, t) = 0$.
(L-2) [Page 129]	If $\mathsf{lag}(T_i^j, t) > 0$ for some time $t \geq \mathsf{d}(T_i^j)$, then the value of $\mathsf{lag}(T_i^j, t)$ denotes the amount of time remaining to be scheduled for T_i^j after time t.
(L-3) [Page 129]	If the system is not reset over the time range $(t_1, t_2]$, then the function $\mathsf{lag}(T_i^j, t)$ is continuous over $[t_1, t_2]$.
(H-1) [Page 130]	If a job T_a^b has a scheduling priority at least T_i^j's and is assigned to the same processor as T_i^j, then $T_a^b \in \mathsf{H}(T_i^j, t)$ for every value of $t \in [\mathsf{r}(T_a^b), t_e)$, where t_e is the first time at which T_a^b is both inactive and not pending.
(H-2) [Page 130]	If the system is not reset over the range $(t_1, t_2]$, then for any job, T_i^j the function $\mathsf{LAG}(\mathsf{H}(T_i^j, t), t)$ is continuous over the time range $[t_1, t_2]$.

Table 4.4: Summary of properties used in Section 4.7.

immediately halt all active tasks. Second, apply Rule R to all tasks that are not currently running when the system reset at t_c, and migrate the tasks that are scheduled at t_c when they complete. Notice that the first option can only be used in systems where non-preemptive behavior is desirable but not required. The second option is more complex to implement than the first option, but is the only viable option when non-preemptive behavior is required.

4.7 Scheduling Correctness

In this section, we prove that no job misses a deadline under our adaptive PEDF scheduling algorithm and that jobs have bounded tardiness under our adaptive NP-PEDF scheduling algorithm. The properties used throughout this section are summarized in Table 4.4.

4.7.1 The CSW Algorithm

Scheduling correctness is established by considering the *clairvoyant scheduling-weight* (CSW) scheduling algorithm. Under it, at each instant t, each job of each task T_i that is both active and incomplete (in the CSW schedule) is allocated a fraction of a processor equal to

$\mathsf{SGwt}(T_i, t)$. We consider CSW to be "clairvoyant" in the sense that CSW does not allocate a job T_i^j more than $\mathsf{Ae}(T_i^j)$ time. More specifically, for any schedule $\mathcal{CSW}$ under CSW of any task system τ, we say that T_i^j has *completed by time t in $\mathcal{CSW}$* iff T_i^j has executed for $\mathsf{Ae}(T_i^j)$ by t. For example, in Figure 4.5, T_4^1 receives no allocations in the CSW scheduling algorithm since $\mathsf{Ae}(T_4^1) = 0$. In addition, if $\mathcal{S}$ is the actual schedule for a task system τ, $\mathcal{CSW}$ is the CSW schedule of τ, the system is reset (re-partitioned) at some time t_r, and some task T_i has received X more time units of allocation in $\mathcal{S}$ than in $\mathcal{CSW}$, then for any $t \geq t_r$, the value X is added to $\mathsf{A}(\mathcal{CSW}, T_i, 0, t)$. For example, in Figure 4.11, when the system is reset at time 2, T_3 is allocated more capacity in the actual schedule than in the CSW schedule. As a result, T_3's allocations in the CSW schedule jumps at time 2 from 1 to 2.

From Figure 4.11, we can see that the following property holds:

(SW-1) If the system is not reset over the range $(t_1, t_2]$, then the function $\mathsf{A}(\mathcal{CSW}, T_i, 0, t)$ is continuous for any $t \in [t_1, t_2]$.

By the definition of the CSW scheduling algorithm, we have the following property:

(SW-2) For any job T_i^j and any two times t_a and t_b, where $\mathsf{r}(T_i^j) \leq t_a \leq t_b$, $\mathsf{A}(\mathcal{CSW}, T_i^j, t_a, t_b) \leq \mathsf{Ae}(T_i^j) \leq \mathsf{e}(T_i^j)$.

In addition, by the definition of $\mathsf{d}(T_i^j)$, it is not difficult to see that the following property holds:

(SW-3) Any job T_i^j is complete in the CSW schedule by time $\mathsf{d}(T_i^j)$.

Throughout this chapter, we use $\mathcal{CSW}$ to denote the CSW schedule of a task system τ.

Notice that, since the CSW schedule is identical to the SW schedule, except for jobs that halt, which receive a smaller allocation in the CSW schedule, it follows that for any job T_i^j and times t_1 and t_2, where $t_1 \leq t_2$, the following property holds:

$$\mathsf{A}(\mathcal{CSW}, T_i^j, t_1, t_2) \leq \mathsf{A}(\mathcal{SW}, T_i^j, t_1, t_2). \tag{4.14}$$

Moreover, since for the same times t_1 and t_2, $\mathsf{A}(\mathcal{SW}, T_i^j, t_1, t_2) = \int_{t_1}^{t_2} \mathsf{SGwt}(T_i, u)du$, the reweighting rules allow at most one job of a task to be active at any given point in time t, and by the Property (W), $\sum_{T_i^j \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{SGwt}(T_i, t) \leq 1$, holds for any job, at any time t, it follows that is not difficult to prove the following lemma.

Lemma 4.1. *For any time interval* $[t_1, t_2]$ *on any processor* $P_{[q]}$,

$$\sum_{T_i^j \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{A}(\mathcal{C}SW,\, T_i^j,\, t_1,\, t_2) \leq \sum_{T_i^j \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{A}(\mathcal{S}W,\, T_i^j,\, t_1,\, t_2) \leq t_2 - t_1.$$

4.7.2 Overlap

One consequence of the reweighting rules is that for some job, T_i^j, it is possible that $\mathsf{r}(T_i^{j+1}) < \mathsf{d}(T_i^j)$. Such a scenario can arise through one of four possibilities: at some time t, T_i changes its desired weight via Case (i) of Rule P; T_i changes its desired weight via Case (i) of Rule N; T_i changes its desired weight via Case (ii) of Rule N; or T_i is reset via Case (i) of Rule R. In all four of these scenarios, T_i^j is halted (and thus completes in the actual schedule) at time t. Moreover, in all three cases, T_i^j is complete in the CSW schedule by time $\mathsf{r}(T_i^{j+1}) \geq t$. (In Case (i) of Rule P, T_i^j has positive deviance, which guarantees that in the CSW schedule T_i^j has received an allocation of $\mathsf{Ae}(T_i^j)$ by time t. In Case (i) of Rule N, T_i^{j+1} is released when its deviance is zero, which guarantees that in the CSW schedule T_i^j has received an allocation of $\mathsf{Ae}(T_i^j)$ by time $\mathsf{r}(T_i^{j+1})$. In Case (i) of Rule R, if T_i^j has a smaller allocation in the CSW schedule than the actual schedule, then this difference is added to T_i^j's allocation at time t, which again guarantees that in the CSW schedule T_i^j has received an allocation of $\mathsf{Ae}(T_i^j)$ by time $\mathsf{r}(T_i^{j+1})$.) Thus, we have the following property:

(V) For the jobs T_i^j and T_i^{j+1}, if $\mathsf{d}(T_i^j) > \mathsf{r}(T_i^{j+1})$, then T_i^j is complete in both the CSW and actual schedules by time $\mathsf{r}(T_i^{j+1})$.

4.7.3 Lag

Before continuing, we introduce the notion of lag, which represents the difference between the actual allocations to a task and its CSW allocations. Formally, the *lag of a job at time* t is given by

$$\mathsf{lag}(T_i^j, t) = \mathsf{A}(\mathcal{CSW}, T_i^j, \mathsf{r}(T_i^j), t) - \mathsf{A}(\mathcal{S}, T_i^j, \mathsf{r}(T_i^j), t), \tag{4.15}$$

where $\mathcal{CSW}$ is the CSW schedule for the system and $\mathcal{S}$ is the actual schedule for the system. Notice that lag is similar to deviance, except that while deviance relates the actual schedule to the SW schedule, lag relates the actual schedule to the CSW schedule. For example, in Figure 4.5, $\mathsf{lag}(T_4^1, 2) = 1/3$ and $\mathsf{lag}(T_4^1, 3) = -1/6$.

We now present a few simple properties about lag will be used in the following proofs.

(L-1) For any time $t \leq \mathsf{r}(T_i^j)$, $\mathsf{lag}(T_i^j, t) = 0$.

(L-2) If $\mathsf{lag}(T_i^j, t) > 0$ for some time $t \geq \mathsf{d}(T_i^j)$, then the value of $\mathsf{lag}(T_i^j, t)$ denotes the amount of time remaining to be scheduled for T_i^j after time t.

(L-3) If the system is not reset over the time range $(t_1, t_2]$, then the function $\mathsf{lag}(T_i^j, t)$ is continuous over $[t_1, t_2]$.

Property (L-1) is trivially true, since for any $t \leq \mathsf{r}(T_i^j)$, $\mathsf{A}(\mathcal{CSW}, T_i^j, \mathsf{r}(T_i^j), t) = \mathsf{A}(\mathcal{S}, T_i^j, \mathsf{r}(T_i^j), t) = 0$. Property (L-2) holds since by time $t \geq \mathsf{d}(T_i^j)$, $\mathsf{A}(\mathcal{CSW}, T_i^j, \mathsf{r}(T_i^j), t) = \mathsf{Ae}(T_i^j)$ and $\mathsf{Ae}(T_i^j) - \mathsf{A}(\mathcal{S}, T_i^j, \mathsf{r}(T_i^j), t)$ denotes the amount of time remaining to be scheduled for T_i^j at and after t. Property (L-3) holds because as long as the system is not reset over the range $(t_1, t_2]$, both $\mathsf{A}(\mathcal{S}, T_i^j, \mathsf{r}(T_i^j), t)$ and $\mathsf{A}(\mathcal{CSW}, T_i^j, \mathsf{r}(T_i^j), t)$ are continuous for any $t \in [t_1, t_2]$. (Notice that the function $\mathsf{A}(\mathcal{S}, T_i^j, \mathsf{r}(T_i^j), t)$ is continuous by definition and the function $\mathsf{A}(\mathcal{CSW}, T_i^j, \mathsf{r}(T_i^j), t)$ is continuous by Property (SW-1).)

For brevity, we use $\mathsf{LAG}(G, t)$ to denote

$$\mathsf{LAG}(G, t) = \sum_{T_i^j \in G} \mathsf{lag}(T_i^j, t),$$

where G is some set of jobs.

We now prove a simple property about lag in PEDF schedules.

Lemma 4.2. *If no deadline of a job of task T_i has been missed by time t and the job T_i^j is not pending at time t, then $\mathsf{lag}(T_i^j, t) \leq 0$.*

Proof. Let T_i^j be as defined in the lemma, i.e., T_i^j is not pending at time t. Since all previous jobs of T_i have completed before their deadlines, by the definition of pending, either $t < \mathsf{r}(T_i^j)$ or $t \geq \mathsf{r}(T_i^j) \wedge \mathsf{A}(\mathcal{S}, T_i^j, \mathsf{r}(T_i^j), t) = \mathsf{Ae}(T_i^j)$ holds. By Property (L-1), if $t < \mathsf{r}(T_i^j)$, then $\mathsf{lag}(T_i^j, t) = 0$. If $t \geq \mathsf{r}(T_i^j) \wedge \mathsf{A}(\mathcal{S}, T_i^j, \mathsf{r}(T_i^j), t) = \mathsf{Ae}(T_i^j)$ holds, then by Property (SW-2), $\mathsf{A}(\mathcal{S}, T_i^j, \mathsf{r}(T_i^j), t) = \mathsf{Ae}(T_i^j) \geq \mathsf{A}(\mathcal{C}SW, T_i^j, \mathsf{r}(T_i^j), t)$ holds. This, in turn, implies that $\mathsf{lag}(T_i^j, t) \leq 0$. □

4.7.4 Higher-Priority Jobs

In the following proofs, we use the term $\mathsf{H}(T_i^j, t)$ to denote the set of jobs that are assigned to the same processor as T_i^j, have a scheduling priority higher than or equal to T_i^j, and are either pending or active at time t. For example, in Figure 4.3, $\mathsf{H}(T_3^1, 0) = \{T_1^1, T_2^1, T_3^1\}$, $\mathsf{H}(T_3^1, 3) = \{T_1^2, T_2^1, T_3^1\}$ and $\mathsf{H}(T_3^1, 7) = \{T_3^1\}$. Since, as we discussed in Section 4.4.2, the relative scheduling priority between any two jobs is time invariant, we have the following property:

(H-1) If a job T_a^b has a scheduling priority at least T_i^j's and is assigned to the same processor as T_i^j, then $T_a^b \in \mathsf{H}(T_i^j, t)$ for every value of $t \in [\mathsf{r}(T_a^b), t_e)$, where t_e is the first time at which T_a^b is both inactive and not pending.

Notice that, if a job T_a^b becomes an element of $\mathsf{H}(T_i^j, t)$ at some time t, then by Property (H-1) it must be that T_a^b was released at time t, and thus, by Property (L-1), $\mathsf{lag}(T_a^b, t) = 0$ holds. Moreover, by Property (H-1), the only way a job T_a^b can leave the set $\mathsf{H}(T_i^j, t)$ is if by time t, T_a^b is both inactive and not pending. If T_a^b is inactive at t, then $\mathsf{r}(T_a^{b+1}) \leq t$ or $\mathsf{d}(T_a^b) \leq t$ (or both). If $\mathsf{r}(T_a^{b+1}) \leq t$ holds, then by Property (V), T_a^b is complete in both the actual and CSW schedules, in which case $\mathsf{lag}(T_a^b, t) = 0$. If $\mathsf{d}(T_a^b) \leq t$, then since T_a^b is not pending at time t, i.e., $\mathsf{Ae}(T_a^b) = \mathsf{A}(\mathcal{S}, T_a^b, \mathsf{r}(T_a^b), t)$, by Property (L-2), $\mathsf{lag}(T_a^b, t) = 0$. Thus, given that the lag of a job is zero when it either joins or leaves $\mathsf{H}(T_i^j, t)$ and that (by Property (L-3)) $\mathsf{lag}(T_a^b, t)$ is continuous so long as the system is not reset, we have the following property:

(H-2) If the system is not reset over the range $(t_1, t_2]$, then for any job T_i^j, the function $\mathsf{LAG}(\mathsf{H}(T_i^j, t), t)$ is continuous over the time range $[t_1, t_2]$.

4.7.5 PEDF Correctness

We now show that in a PEDF scheduled system, no job misses a deadline.

Theorem 4.2. *Let τ be an adaptable sporadic task system. Then, for any job T_i^j that is in τ, T_i^j completes by $\mathsf{d}(T_i^j)$ under* PEDF.

Proof. Suppose, to derive a contradiction, that there exists some job T_i^j in τ such that T_i^j misses its deadline at time t_d. Without loss of generality, let t_d be the first time that a job deadline is missed in τ. Notice that, by Property (V), if $\mathsf{d}(T_i^{j-1}) > \mathsf{r}(T_i^j)$, then T_i^{j-1} must be complete by $\mathsf{r}(T_i^j)$. Moreover, if $\mathsf{d}(T_i^{j-1}) \leq \mathsf{r}(T_i^j) < \mathsf{d}(T_i^j) = t_d$, then by the definition of t_d, T_i^{j-1} is complete by $\mathsf{r}(T_i^j)$. Thus, in either case, T_i^{j-1} is complete by $\mathsf{r}(T_i^j)$.

By Property (L-2), the value of $\mathsf{lag}(T_a^b, t)$ at any time $t \geq \mathsf{d}(T_a^b)$ represents T_a^b's remaining computation time at time t. Thus, if T_i^j misses a deadline at time t_d, then $\mathsf{lag}(T_i^j, t_d) > 0$ must hold. Moreover, since every job in $\mathsf{LAG}(\mathsf{H}(T_i^j, t_d), t_d)$ has a deadline at or before t_d, by Property (L-2), $\mathsf{LAG}(\mathsf{H}(T_i^j, t_d), t_d)$ represents the amount of computation remaining for all jobs in $\mathsf{LAG}(\mathsf{H}(T_i^j, t_d), t_d)$, including T_i^j. Thus, if T_i^j misses a deadline at t_d, then $\mathsf{LAG}(\mathsf{H}(T_i^j, t_d), t_d) > 0$ must hold. The objective of this proof is to show that $\mathsf{LAG}(\mathsf{H}(T_i^j, t_d), t_d) \leq 0$, which contradicts our assumption and completes the proof.

Before continuing, we introduce a few terms. Let $P_{[q]}$ denote the processor that T_i^j is assigned to at time t_d. Let $\mathcal{S}$ denote the PEDF schedule of τ. Let t_1 be the first time before t_d such that over the range $(t_1, t_d]$ the system is not reset, and over the range $[t_1, t_d]$, $P_{[q]}$ is continually scheduling jobs from $\mathsf{H}(T_i^j, t)$. Notice that t_1 must exist because T_i^j is pending at time t_d, and since (as we established above) T_i^{j-1} is complete by $\mathsf{r}(T_i^j)$, either T_i^j or some higher-priority job is scheduled immediately before t_d. In addition, the system is not reset at t_d because if it were, then T_i^j would be halted at t_d (and thus complete). Figure 4.12 illustrates both t_1 and t_d.

There are two remaining components of this proof. First, we show that $\mathsf{LAG}(\mathsf{H}(T_i^j, t_1), t_1) \leq 0$ holds. Second, we show that for any $t \in [t_1, t_d]$, $\mathsf{LAG}(\mathsf{H}(T_i^j, t), t) \leq 0$ holds, which implies that $\mathsf{lag}(T_i^j, t_d) \leq 0$ holds.

Claim 4.1. $\mathsf{LAG}(\mathsf{H}(T_i^j, t_1), t_1) \leq 0$.

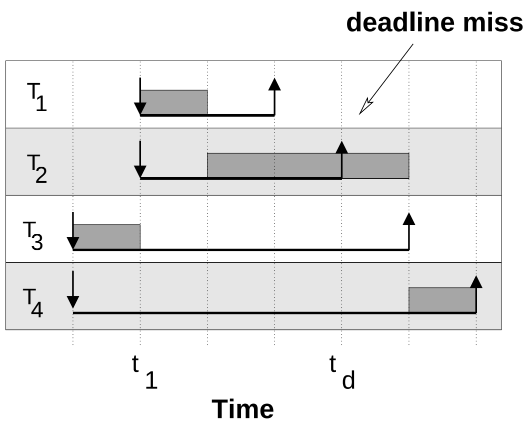


Figure 4.12: An illustration of time t_1 and t_d.

Proof. By Rule R, if the system is reset at time t_1 or $t_1 = 0$, then every job that is active at t_1 must also be released at t_1. Thus, by Property (L-1) the lag of every active job at t_1 equals 0. This, in turn, implies that $\mathsf{LAG}(\mathsf{H}(T_i^j, t_1), t_1) = 0$. Thus, for the remainder of the proof of Claim 4.1, we assume that $t_1 > 0$ and the system is not reset at time t_1.

In this case, by the definition of t_1, no job with a priority higher than or equal to T_i^j's is scheduled t_1. Thus, there must exist some value $\epsilon_1 > 0$ such that for any time $t \in [t_1 - \epsilon_1,\, t_1)$, either $P_{[q]}$ is idle or some job T_a^b with a scheduling priority lower than any job in $\mathsf{H}(T_i^j, t)$ is scheduled on $P_{[q]}$. For example, in Figure 4.12, T_3 is scheduled over $[t_1 - 1,\, t_1)$. Regardless of whether $P_{[q]}$ is idle or scheduling a lower-priority job over the range $[t_1 - \epsilon_1,\, t_1)$, no job in $\mathsf{H}(T_i^j, t)$ is pending over $t \in [t_1 - \epsilon_1,\, t_1)$. For example, in Figure 4.12, neither T_1 or T_2 is pending over the range $[t_1 - 1,\, t_1)$. As a result, at time t_1 every job in $\mathsf{H}(T_i^j, t_1)$ is either released, i.e., becomes a pending job, or is active but not pending. Thus, by Property (L-1), Lemma 4.2, and because no deadline is missed by t_1 (since $t_1 < t_d$), $\mathsf{LAG}(\mathsf{H}(T_i^j, t_1), t_1) \leq 0$. □

Claim 4.2. *For any* $t \in [t_1, t_d]$, $\mathsf{LAG}(\mathsf{H}(T_i^j, t), t) \leq 0$.

Proof. To derive a contradiction, we assume that there exists some $t \in [t_1, t_d]$ such that $\mathsf{LAG}(\mathsf{H}(T_i^j, t), t) > 0$. Since the system is not reset over the time range $(t_1, t_d]$, by Property (H-2), it follows that $\mathsf{LAG}(\mathsf{H}(T_i^j, t), t)$ is continuous over the range $[t_1, t_d]$. As a result, by Claim 4.1 ($\mathsf{LAG}(\mathsf{H}(T_i^j, t_1), t_1) \leq 0$) and our assumption that $\mathsf{LAG}(\mathsf{H}(T_i^j, t), t) > 0$ holds, it follows, by the definition of continuity, that there exists a time $t_z \in [t_1,\, t_d)$ and a value

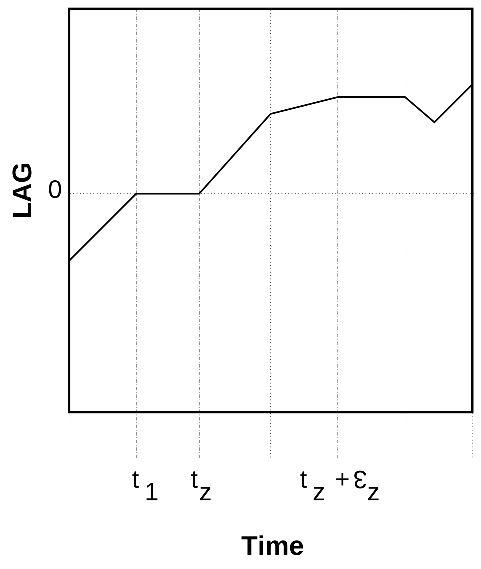


Figure 4.13: An illustration of time t_z and ϵ_z.

$\epsilon_z \in (0, t_d - t_z)$ such that the following conditions hold

(i) $\mathsf{LAG}(\mathsf{H}(T_i^j, t_z), t_z) = 0$,

(ii) $\mathsf{LAG}(\mathsf{H}(T_i^j, t), t)$ is strictly monotonically increasing over the range $t \in [t_z,\, t_z + \epsilon_z)$,

Such a scenario is illustrated in Figure 4.13.

We now show that t_z and ϵ_z cannot exist, which completes the proof of Claim 4.2. Since, by the definition of t_1, over the range $[t_1, t_d]$, $P_{[q]}$ is continually scheduling jobs from $\mathsf{H}(T_i^j, t)$, it follows that some job from $\mathsf{H}(T_i^j, t)$ is scheduled for some range (t_3, t_4) that is contained within $(t_z, t_z + \epsilon_z)$. Thus, by Lemma 4.1,

$$\mathsf{A}(\mathcal{C}SW,\, \mathsf{ASSN}(P_{[q]}, t),\, t_3,\, t_4) \leq t_4 - t_3 = \mathsf{A}(\mathcal{S},\, \mathsf{H}(T_i^j, t),\, t_4,\, t_3).$$

Moreover, since $\mathsf{A}(\mathcal{C}SW,\, \mathsf{H}(T_i^j, t),\, t_3,\, t_4) \leq \mathsf{A}(\mathcal{C}SW,\, \mathsf{ASSN}(P_{[q]}, t),\, t_3,\, t_4)$ it follows that

$$\mathsf{A}(\mathcal{C}SW,\, \mathsf{H}(T_i^j, t),\, t_3,\, t_4) \leq \mathsf{A}(\mathcal{C}SW,\, \mathsf{ASSN}(P_{[q]}, t),\, t_3,\, t_4) \leq \mathsf{A}(\mathcal{S},\, \mathsf{H}(T_i^j, t),\, t_4,\, t_3).$$

Thus,

$$\mathsf{LAG}(\mathsf{H}(T_i^j, t), t_4) - \mathsf{LAG}(\mathsf{H}(T_i^j, t), t_3) \leq 0.$$

This, in turn, implies that over the range (t_3, t_4), $\mathsf{LAG}(\mathsf{H}(T_i^j, t), t)$ is not strictly monotonically increasing, which contradicts Property (ii) of t_z and ϵ_z. Thus, t_z and ϵ_z cannot exist, which completes the proof of the claim. □

Since Claim 4.2 implies that $\mathsf{LAG}(\mathsf{H}(T_i^j, t_d), t_d) \leq 0$ holds, it follows that at t_d no computation time remains for any job in $\mathsf{H}(T_i^j, t_d)$, which includes T_i^j. Thus, T_i^j is complete by its deadline. □

4.7.6 NP-PEDF Correctness

In this section, we show that the tardiness of a task T_i under NP-PEDF is at most $\mathcal{X}$, where $\mathcal{X}$ is the largest execution time of any task. Since the proof for tardiness bounds under NP-PEDF is similar to the scheduling correctness proof for PEDF, rather than repeat the entire proof, we state where they differ.

The primary difference between the proofs for NP-PEDF and PEDF is that in the proof of Theorem 4.2, if the system is not reset at t_1 and $t_1 > 0$, then we can guarantee that no job in $\mathsf{H}(T_i^j, t)$ is pending immediately before t_1. However, in proving the tardiness bounds for NP-PEDF this is not the case. For example, consider the scenario, depicted in Figure 4.14, where a job not in $\mathsf{H}(T_i^j, t)$ becomes pending immediately after a lower-priority job begins being scheduled. In this scenario, the job in $\mathsf{H}(T_i^j, t)$ must wait until the lower-priority job completes before it begins being scheduled. Since this delay can be up to $\mathcal{X}$ time units long, and since it is possible that the sum of the guaranteed weights of all tasks in $\mathsf{H}(T_i^j, t)$ may be close to one, it is possible that $\mathsf{LAG}(\mathsf{H}(T_i^j, t_1), t_1)$ may be close to $\mathcal{X}$. As a result, $\mathsf{LAG}(\mathsf{H}(T_i^j, t_d), t_d)$ may be close to $\mathcal{X}$, which implies that the amount of work remaining for all tasks in $\mathsf{LAG}(\mathsf{H}(T_i^j, t_d), t_d)$ is at most $\mathcal{X}$, which means that T_i^j may miss its deadline by at most $\mathcal{X}$.

One final note: notice that if a task cannot be migrated immediately when the system is reset (because it is non-preemptable and is being scheduled), then this does not impact the correctness proof. The reason why is because delaying a task's migration does not cause the guaranteed weight of all tasks assigned to a processor to be larger than one.

From this discussion, we have the following theorem.

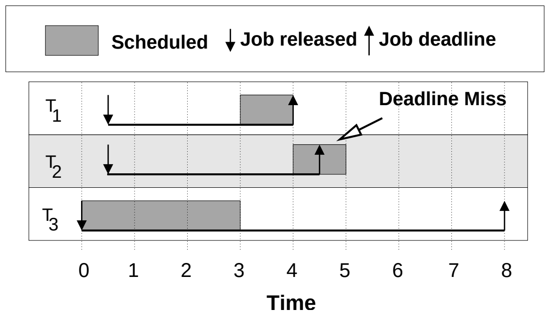


Figure 4.14: A one-processor example of an NP-PEDF system where two tasks are released after a lower-priority job begins executing.

Theorem 4.3. *Let τ be an adaptable sporadic task system. Then, for any job T_i^j of a task in τ, T_i^j has tardiness at most $\mathcal{X}$ under* NP-PEDF, *where $\mathcal{X}$ is the largest execution time of any job in τ.*

4.8 The IDEAL and PT Algorithms

In section, Section 4.9, we turn our attention to proving "drift" bounds. For this purpose, we introduce two new theoretical scheduling algorithms, namely the IDEAL and PT algorithms.

Under the IDEAL scheduling algorithm, at each instant t, each task T_i in τ with an active job at t is allocated a fraction of the system equal to its guaranteed weight, $\mathsf{Gwt}(T_i, t)$. Hence, if $\mathcal{I}$ is the IDEAL schedule of τ and T_i is active over the interval $[t_1, t_2)$, then over $[t_1, t_2)$, the task T_i, assigned to $P_{[q]}$, is allocated

$$\mathsf{A}(\mathcal{I}, T_i, t_1, t_2) = \int_{t_1}^{t_2} \mathsf{Gwt}(T_i, u)du = \int_{t_1}^{t_2} \frac{\mathsf{Dwt}(T_i, u)}{\mathcal{T}\mathsf{D}(P_{[q]}, u)}du \tag{4.16}$$

time. Throughout this chapter, we use $\mathcal{I}$ to denote an IDEAL schedule of the task system τ.

In the *partial ideal* (PT) scheduling algorithm, each task T_i with an active job at each instant is allocated a fraction of the system equal to

$$\frac{\mathsf{Dwt}(T_i, t)}{\mathcal{T}\mathsf{S}(P_{[q]}, t)}.$$

Hence, if $\mathcal{PT}$ is a PT schedule of τ and the task T_i is active over the interval $[t_1, t_2)$, then over $[t_1, t_2)$, T_i is allocated

$$\mathsf{A}(\mathcal{PT}, T_i, t_1, t_2) = \int_{t_1}^{t_2} \frac{\mathsf{Dwt}(T_i, u)}{\mathcal{T}\mathsf{S}(P_{[q]}, u)} du \qquad (4.17)$$

time. The PT scheduling algorithm will be used to help calculate the drift incurred by changing the desired weight of a task. Throughout this chapter, we use $\mathcal{PT}$ to denote a PT schedule of the task system τ.

Example (Figures 4.15 and 4.16). Consider the example in Figures 4.15 and 4.16, which pertain to a one-processor system that is assigned four tasks: T_1, which has $\mathsf{e}(T_1) = 1$ and $\mathsf{Dwt}(T_1) = 1/3$; T_2, which has $\mathsf{e}(T_2) = 1$ and $\mathsf{Dwt}(T_2) = 1/6$; T_3, which has $\mathsf{e}(T_3) = 2$, $\mathsf{Dwt}(T_3) = 1/4$, and leaves at time 8; and T_4, which has $\mathsf{e}(T_4) = 4$ and an initial desired weight of 1/4 that increases to 1/2 at time 8 via Case (i) of Rule P. (This is the same system as in Figure 4.3.) Figure 4.15(a) depicts the PEDF schedule. Figure 4.15(b) depicts the allocations to T_4 in the PEDF, IDEAL, SW, CSW, and PT scheduling algorithms. Figure 4.16(a) depicts the SW, IDEAL, and PT schedules. Figure 4.16(b) depicts the CSW schedule. Notice that T_4^1 receives no allocations in CSW once it has received one unit of execution (the amount that T_4^1 is allocated in the PEDF schedule). $\square$

Example (Figure 4.17). Consider the example in Figure 4.17, which depicts one processor that is assigned four tasks (where the execution time of each job is one): T_1, which has $\mathsf{Dwt}(T_1) = 1/2$; T_2 and T_3, both of which have a desired weight of 1/6; and T_4, which has an initial desired weight of 1/2 that initiates a desired weight decrease to 1/6 at time 1 that is enacted at time $2.\bar{6}$ via Case (ii) of Rule N. (This is the same system as in Figure 4.4.)

Notice that, since the IDEAL scheduling algorithm allocates capacity to each task based on its guaranteed weight (rather than based on its guaranteed scheduling weight), when one task initiates a decrease in its desired weight on an over-utilized processor, the allocation to all other tasks in the IDEAL schedule immediately increases. For example, in Figure 4.17(e), when T_4 initiates a desired weight decrease at time 1, the rate of allocation to all other tasks immediately increases even though the desired weight change is not enacted until time $2.\bar{6}$. (In the IDEAL schedule, before time 1, T_1 is allocated 3/8 of the processor at each instant,

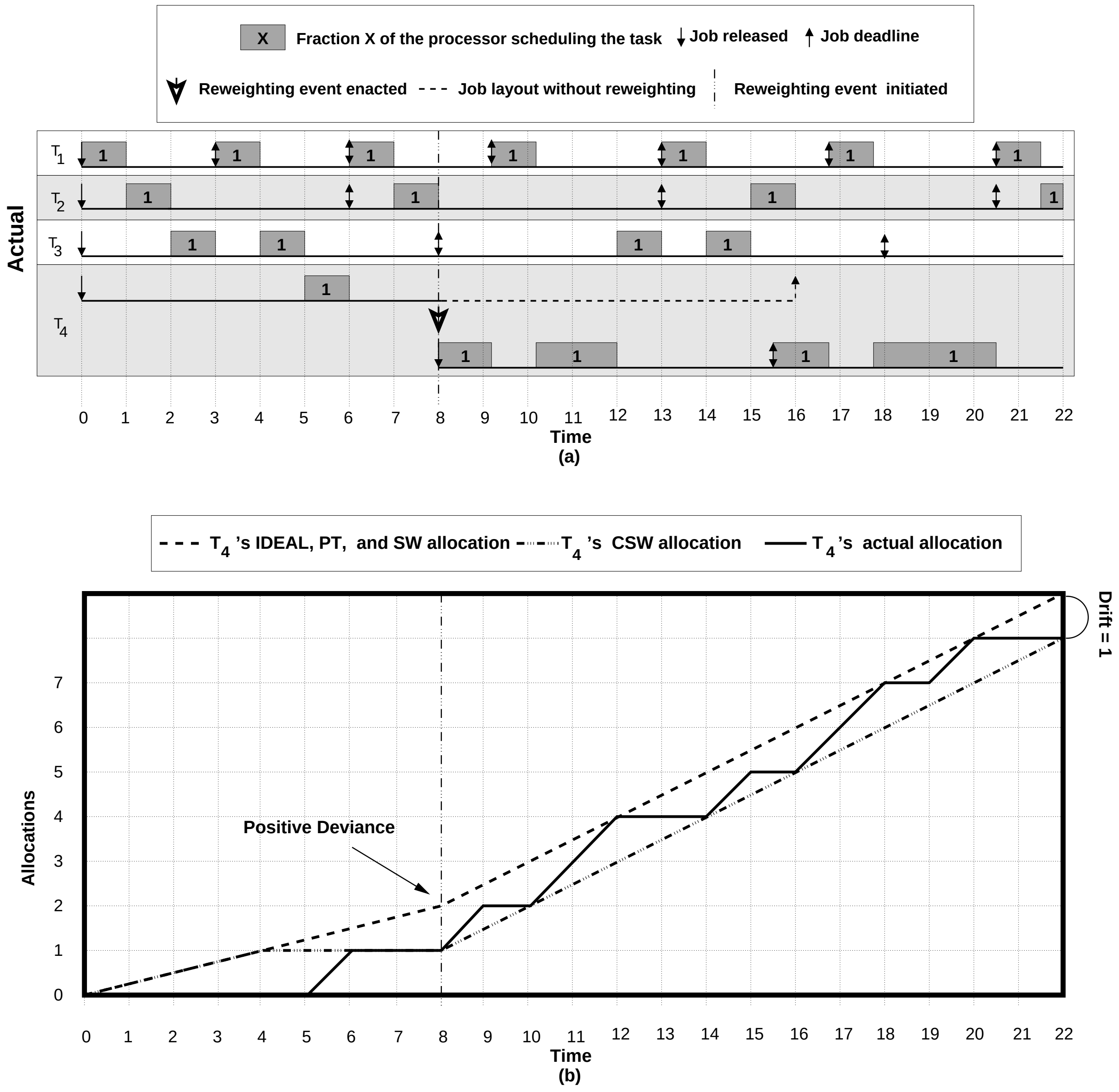


Figure 4.15: A one-processor system with four tasks. **(a)** The PEDF schedule. **(b)** The allocations to T_4 in the PEDF, IDEAL, SW, CSW, and PT scheduling algorithms. (Figure 4.16 depicts the IDEAL, SW, CSW, and PT schedules.)

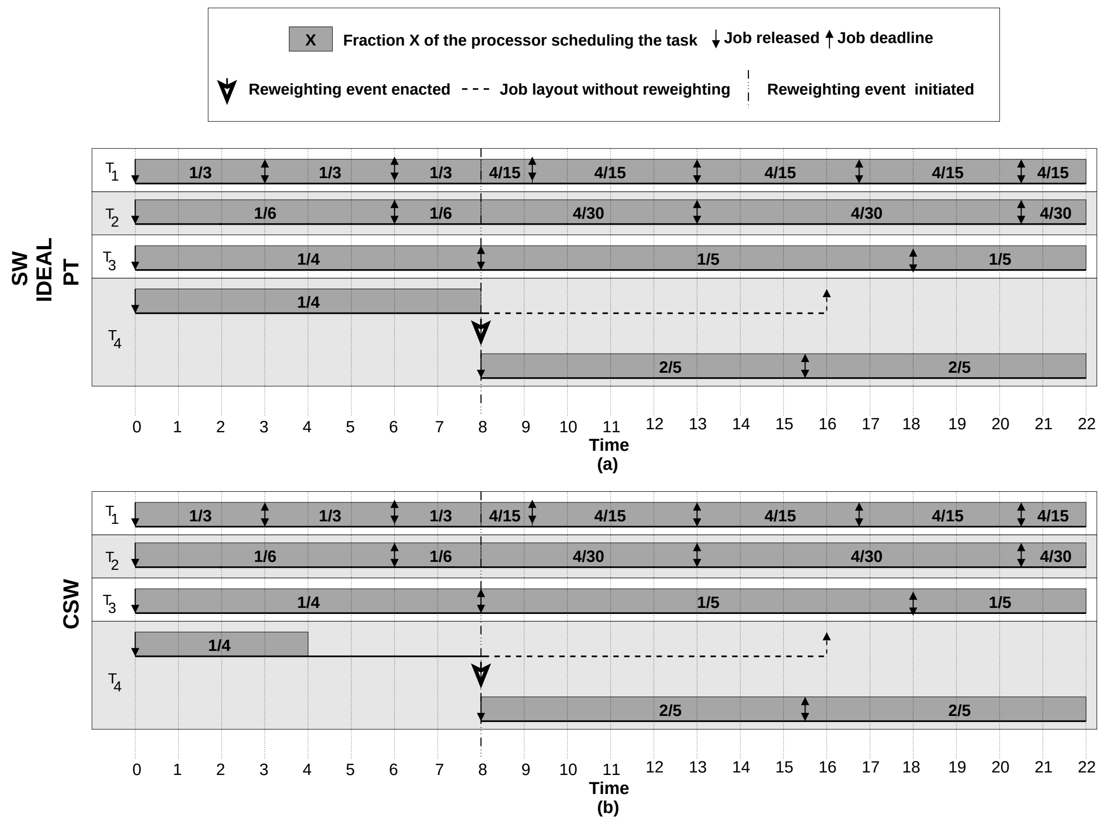


Figure 4.16: The IDEAL, CSW, SW, PT schedules for the same system as in Figure 4.15. **(a)** The SW, IDEAL, and PT schedules. **(b)** The CSW schedule.

and after time 1 it is allocated 1/2 of the processor at each instant.)

It is important to note that since the scaling factor for each task in the PT scheduling algorithm is based on the the total guaranteed scheduling weight (rather than the total guaranteed weight), when one task initiates a decrease in its desired weight on an over-utilized processor, the allocation to all other tasks in the PT schedule remains the same until the desired weight change is enacted. For example, in Figure 4.17(f), when T_4 initiates a desired weight decrease at time 1, the rate of allocation to all other tasks remains the same until the change is enacted at time $\frac{8}{3}$. This example illustrates the difference between the IDEAL and PT scheduling algorithms. □

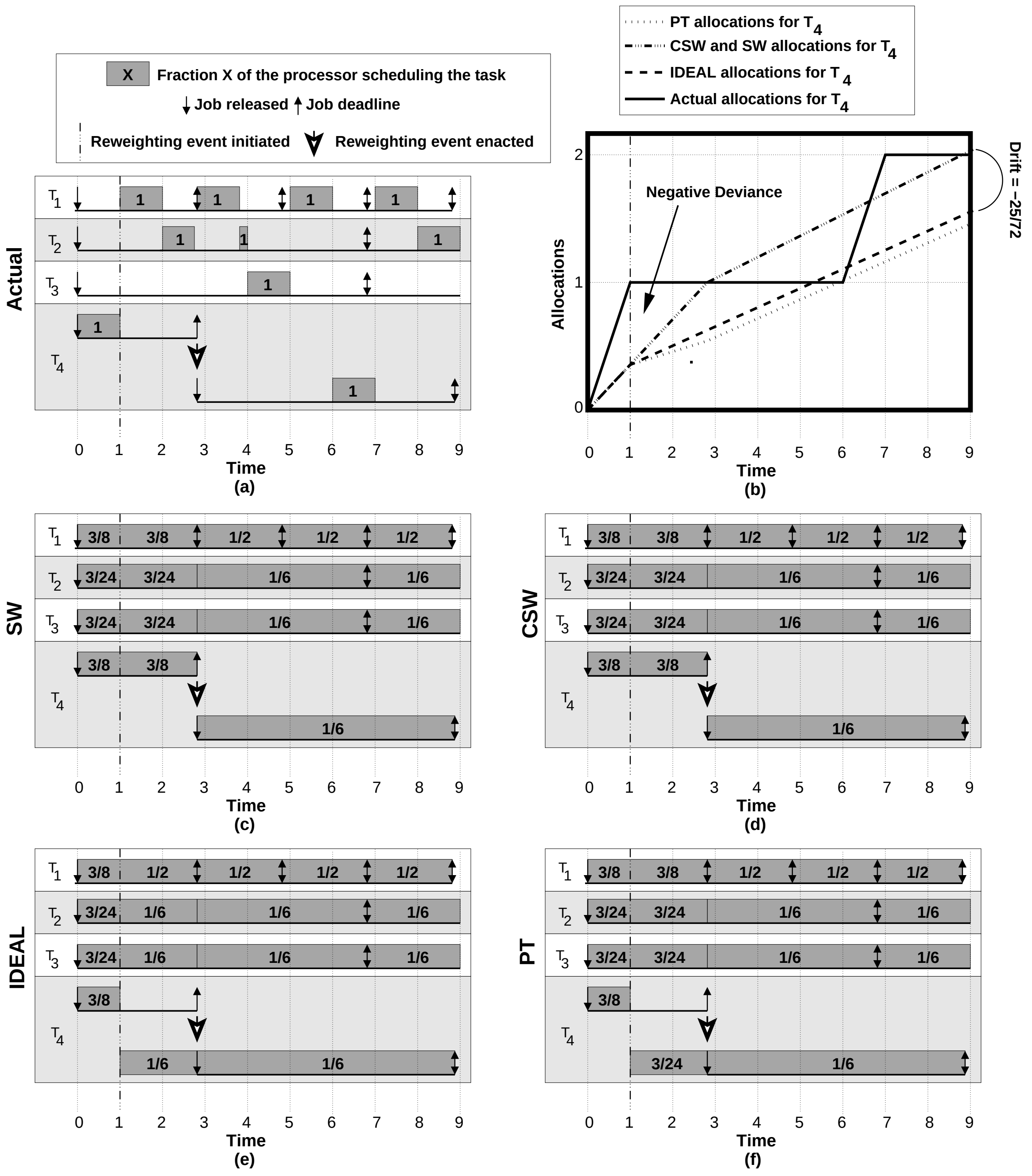


Figure 4.17: A one-processor system with four tasks. **(a)** The PEDF schedule. **(b)** The allocations to T_4 in the PEDF, IDEAL, SW, CSW, and PT scheduling algorithms. **(c)** The SW schedule. **(d)** The CSW schedule. **(e)** The IDEAL schedule. **(f)** The PT schedule.

Property	**Definition**
(D) [Page 110]	For any two times u_1 and u_2 such that $\mathsf{r}(T_i^j) \leq u_1 \leq u_2 \leq \mathsf{d}(T_i^j)$, $\int_{u_1}^{u_2} \mathsf{SGwt}(T_i, t)dt \leq \mathsf{e}(T_i^j)$.
(SW-2) [Page 127]	For any job T_i^j and any $t \geq \mathsf{r}(T_i^j)$, $\mathsf{A}(\mathcal{CSW}, T_i^j, \mathsf{r}(T_i^j), t) \leq \mathsf{Ae}(T_i^j) \leq \mathsf{e}(T_i^j)$.
(SW-2) [Page 127]	Any job T_i^j is complete in the CSW schedule by time $\mathsf{d}(T_i^j)$.
(X) [Page 122]	If a task T_i initiates a desired weight change at time t_c and the job T_i^j is active at t_c, then some desired weight change is enacted according to Rule P or N by either $\mathsf{d}(T_i^j)$ or when T_i^j becomes inactive, whichever is first.
(V) [Page 128]	For the jobs T_i^j and T_i^{j+1}, if $\mathsf{d}(T_i^j) > \mathsf{r}(T_i^{j+1})$, then T_i^j is complete in both the CSW and actual schedules by time $\mathsf{r}(T_i^{j+1})$.

Table 4.5: Summary of properties used in Section 4.9.

4.9 Drift

We now turn our attention to the issue of measuring drift under PEDF. The properties used throughout this section are summarized in Table 4.5. We begin this section, by formally defining drift. The *drift of a task T_i at time t* is defined as

$$\mathsf{drift}(T_i, t) = \mathsf{A}(\mathcal{I}, T_i, 0, t) - \mathsf{A}(\mathcal{CSW}, T_i, 0, t). \tag{4.18}$$

In this section, we show that at time t for T_i, if t satisfies one of conditions (T-1), ..., (T-3) (defined below), then the value of $\mathsf{drift}(T_i, t)$ is bounded by $[-\mathcal{X} \cdot \mathcal{Q},\ \mathcal{X} \cdot \mathcal{Q}]$, where $\mathcal{X}$ is the maximal execution time of any task in the system and $\mathcal{Q}$ denotes the number of system resets plus the number of desired weight changes for any task assigned to the same processor as T_i that are initiated before t. In the following conditions, T_i^j denotes the last-released job (if any) of T_i before t.

(T-1) The first job of T_i (if it exists) is released at or after t.

(T-2) $\mathsf{d}(T_i^j) \leq t$.

(T-3) $\mathsf{r}(T_i^j) < t_e \leq t$, where t_e is the time that the last-initiated change (at or before t) by T_i was enacted.

The reason why we must constrain the times at which the drift is measured is because it is possible for T_i to incur drift for a reweighting event of T_i that has yet to be initiated. For example, in Figure 4.5, T_4 incurs drift over the range [0, 2) even though it does not initiate a desired weight change until time 2. This complication arises because the reweighting rules may halt the last-released job of a task. Since the CSW scheduling algorithm is clairvoyant, it accounts for this drift before the reweighting event occurs. Thus, to accurately assess the drift T_i incurs per reweighting event, we can only measure the drift at the times described above.

We begin our discussion by first calculating the drift that is incurred in a system with no system resets. We then factor in resets in Section 4.9.4.

Lemma 4.3. *For any job T_i^j of any task T_i in a adaptable sporadic task system scheduled by* PEDF*,*

$$\mathsf{A}(\mathcal{CSW},\, T_i,\, \mathsf{r}(T_i^j),\, t_I) \leq \mathsf{e}_{\mathsf{max}}(T_i),$$

where t_I is the time that T_i^j becomes inactive.

Proof. By Property (SW-2), $\mathsf{A}(\mathcal{CSW},\, T_i^j,\, \mathsf{r}(T_i^j),\, t) \leq \mathsf{e}_{\mathsf{max}}(T_i)$ holds for any $t \geq \mathsf{r}(T_i^j)$, including t_I. Thus, if it can be shown that no other job of T_i receives allocations in $\mathcal{CSW}$ over the range $[\mathsf{r}(T_i^j),\, t_I)$, then the proof is complete. Since T_i^j becomes inactive at $\mathsf{min}(\mathsf{d}(T_i^j), \mathsf{r}(T_i^{j+1}))$, it follows that no job of T_i that is released after $\mathsf{r}(T_i^j)$ recives any allocations in $\mathcal{CSW}$ over the range $[\mathsf{r}(T_i^j),\, t_I)$.

Thus, it remains to be shown that no job T_i^a that is released before $\mathsf{r}(T_i^j)$ receives any allocations in $\mathcal{CSW}$ over the range $[\mathsf{r}(T_i^j),\, t_I)$. By Property (V), if $\mathsf{d}(T_i^a) > \mathsf{r}(T_i^j)$, then T_i^a is complete by $\mathsf{r}(T_i^j)$. By Property (SW-3), if $\mathsf{d}(T_i^a) \leq \mathsf{r}(T_i^j)$, then T_i^a is complete by $\mathsf{r}(T_i^j)$. Thus, in either case, T_i^a, receives no allocations in CSW over the range $[\mathsf{r}(T_i^j),\, t_I)$. This completes the proof. ☐

Two types of drift. One complication with calculating the drift associated with a non-resetting reweighting event is that on an over-utilized processor a task may incur drift even when its desired weight does not change. For example, in Figure 4.17, T_1, T_2, and T_3, incur drift over the range [1, 8/3), during which time T_4 has initiated a reweighting event that has not yet been enacted. This behavior is the result of defining each task's guaranteed weight as a function of its desired weight and the desired weight of every task in the system. Thus, when T_4 initiates its desired weight decrease at time 1, in the IDEAL schedule, in Figure 4.17(e), the guaranteed weight of every other task immediately increases. However, in the CSW schedule, in Figure 4.17(d), the guaranteed weight of T_1, T_2, and T_3 cannot change until T_4's decrease is enacted at time 8/3.

As a result, a task T_i can incur two types of drift: **(i)** T_i can incur drift because it changes its desired weight and **(ii)** T_i can incur drift because another task on the same processor initiated a desired weight change that is not immediately enacted. (Recall that, for now, we are ignoring the drift incurred by resetting the system.) In order to determine the total amount of drift caused by a reweighting event, we consider these two types of drift separately. In order to differentiate between type-(i) and -(ii) drift, we use the PT scheduling algorithm (which was defined in Section 4.8).

Recall that under the PT scheduling algorithm at each instant each active task T_i (assigned to $P_{[q]}$) receives an allocation equal to

$$\frac{\mathsf{Dwt}(T_i, t)}{\mathcal{T}\mathsf{S}(P_{[q]}, t)}.$$

Since the scaling factor in the PT scheduling algorithm, i.e., $\frac{1}{\mathcal{T}\mathsf{S}(P_{[q]},t)}$, is defined in terms of the total desired *scheduling* weight instead of the total desired weight (i.e., in terms of $\mathcal{T}\mathsf{S}(P_{[q]}, t)$ instead of $\mathcal{T}\mathsf{D}(P_{[q]}, t)$), the only time there is a difference between T_i's allocations in the PT and CSW schedules, is when T_i's desired weight is changed. We describe this difference as the *partial drift of a task T_i at time t*, which is formally defined as

$$\mathsf{Pdrift}(T_i, t) = \mathsf{A}(\mathcal{P}T,\, T_i,\, 0,\, t) - \mathsf{A}(\mathcal{C}SW,\, T_i,\, 0,\, t). \tag{4.19}$$

For example, in Figures 4.15 and 4.16, only T_4 incurs partial drift because it is the only task that changes its desired weight. Also, in Figure 4.17, T_4 is the only task to incur partial drift because it is the only task that changes its desired weight. Thus, the partial drift incurred by T_i changing its desired weight via Rules P and N is equal to drift of type (i).

In addition, since the only difference between a task's allocations in the IDEAL and PT schedules is the scaling factor (i.e., in the IDEAL schedule T_i receives $\frac{\mathsf{Dwt}(T_i,t)}{\mathcal{T}\mathsf{D}(P_{[q]},t)}$ allocations at each instant, while in the PT, schedule T_i receives $\frac{\mathsf{Dwt}(T_i,t)}{\mathcal{T}\mathsf{S}(P_{[q]},t)}$ allocations at each instant), the amount of type (ii) drift a task incurs is equal to the difference in its allocations between the PT and IDEAL schedules.

4.9.1 Partial Drift

In this section, we establish bounds on the partial drift incurred by a reweighting event.

Lemma 4.4. *Under* PEDF*, let T_i be a task that initiates a desired weight change at t_c. Assume that T_i has released a job before time t_c and let T_i^j be its last-released job before t_c. Let t_e denote the first time at or after t_c at which T_i enacts a desired weight change. Let t_I denote the time that T_i^j becomes inactive. If t_c is the first change initiated by T_i in the range $(\mathsf{r}(T_i^j), t_I]$, then for any $t_b \in [t_e, t_I]$ and any time $t_a \in [\mathsf{r}(T_i^j), t_b]$, $\mathsf{Pdrift}(T_i, t_b) - \mathsf{Pdrift}(T_i, t_a)$ is bounded by $[-\mathsf{e}_{\mathsf{max}}(T_i) \cdot (\mathcal{G} + \mathcal{P}_1),\ \mathsf{e}_{\mathsf{max}}(T_i) \cdot (\mathcal{P}_1 + \mathcal{P}_2)]$, where $\mathcal{G}$, $\mathcal{P}_1$, and $\mathcal{P}_2$ denote, respectively, the number of weight decreases, the number of weight increases via Case (i) of Rule P, and the number of weight increases via Case (ii) of Rule P by T_i that were initiated at or before t_b and enacted or canceled after t_a.*

Proof. Let t_c, t_e, t_b, t_a, t_I, and T_i^j be as defined in the statement of the lemma. If $t_I < t_e$, then the lemma is vacuously true, so assume that $t_e \leq t_I$. Notice that, by the definition of inactive (Definition 4.1), $t_I = \mathsf{min}(\mathsf{d}(T_i^j), \mathsf{r}(T_i^{j+1}))$. Thus, by the statement of the lemma, we have

$$t_c \leq t_e \leq t_b \leq t_I \leq \mathsf{d}(T_i^j). \tag{4.20}$$

Suppose for the moment that T_i initiates a desired weight change at time t'_c such that $t_e < t'_c \leq t_I$. Notice that t'_c's existence implies that the change enacted at t_e must have been by Case (i) of Rule N, since for all other rules, $t_e = t_I$ (specifically, all other scenarios release T_i^{j+1} at t_e). Thus, by Case (i) of Rule N, T_i^j is halted before t'_c. Thus, we having the following property:

(HA) No change initiated in the range $(t_e, t_I]$ can halt T_i^j.

Having established (4.20) and Property (HA), we now show that the value of $\mathsf{Pdrift}(T_i, t_b) - \mathsf{Pdrift}(T_i, t_a)$ is bounded. Notice that, by (4.19),

$$\mathsf{Pdrift}(T_i, t_b) - \mathsf{Pdrift}(T_i, t_a) = \mathsf{A}(\mathcal{PT},\, T_i,\, t_a,\, t_b) - \mathsf{A}(\mathcal{CSW},\, T_i,\, t_a,\, t_b). \tag{4.21}$$

By adding and subtracting $\int_{t_a}^{t_b} \mathsf{SGwt}(T_i, u)du$, the right-hand side of the above formula can be rewritten as

$$\mathsf{A}(\mathcal{PT},\, T_i,\, t_a,\, t_b) - \int_{t_a}^{t_b} \mathsf{SGwt}(T_i, u)du + \int_{t_a}^{t_b} \mathsf{SGwt}(T_i, u)du - \mathsf{A}(\mathcal{CSW},\, T_i,\, t_a,\, t_b).$$

We now bound the terms $\mathsf{A}(\mathcal{PT},\, T_i,\, t_a,\, t_b) - \int_{t_a}^{t_b} \mathsf{SGwt}(T_i, u)du$ and $\int_{t_a}^{t_b} \mathsf{SGwt}(T_i, u)du - \mathsf{A}(\mathcal{CSW},\, T_i,\, t_a,\, t_b)$.

Bounding $\mathsf{A}(\mathcal{PT},\, T_i,\, t_a,\, t_b) - \int_{t_a}^{t_b} \mathsf{SGwt}(T_i, u)du$. Notice that, since t_c is the first change initiated by T_i over the range $(\mathsf{r}(T_i^j), t_I]$, it follows that for any $t \in [\mathsf{r}(T_i^j),\, t_c)$, $\mathsf{Dwt}(T_i, t) = \mathsf{SDwt}(T_i, t)$. Thus, since, by (4.5), $\mathsf{SGwt}(T_i, t) = \frac{\mathsf{SDwt}(T_i,t)}{\mathcal{T}\mathsf{S}(P_{[q]},t)}$ and T_i receives $\frac{\mathsf{Dwt}(T_i,t)}{\mathcal{T}\mathsf{S}(P_{[q]},t)}$ allocations at each instant in the PT schedule, it follows that $\mathsf{A}(\mathcal{PT},\, T_i,\, \mathsf{r}(T_i^j),\, t_c) - \int_{\mathsf{r}(T_i^j)}^{t_c} \mathsf{SGwt}(T_i, u)du = 0$, and, by extension, if $t_a \in [\mathsf{r}(T_i^j), t_b]$, then $\mathsf{A}(\mathcal{PT},\, T_i,\, t_a,\, t_c) - \int_{t_a}^{t_c} \mathsf{SGwt}(T_i, u)du = 0$. Thus, for the remainder of this case we bound the value of $\mathsf{A}(\mathcal{PT},\, T_i,\, \mathsf{max}(t_c, t_a),\, t_b) - \int_{\mathsf{max}(t_c,t_a)}^{t_b} \mathsf{SGwt}(T_i, u)du$.

Notice that, if $t_c = t_b$, then $t_b = t_e = \mathsf{max}(t_c, t_a)$ (since, by the definition of t_b and t_a given in the statement of the lemma, both $t_a \leq t_b$ and $t_e \leq t_b$ hold, and by (4.20), $t_c \leq t_e$) and $\mathsf{A}(\mathcal{PT},\, T_i,\, \mathsf{max}(t_c, t_a),\, t_b) - \int_{\mathsf{max}(t_c,t_a)}^{t_b} \mathsf{SGwt}(T_i, u)du = 0$. Thus, for the remainder of this case, we assume that $t_c < t_b$.

Let z denote the number of times T_i initiates or enacts a desired weight change over the range $[\mathsf{max}(t_c, t_a),\, t_b)$. We begin by decomposing the interval $[\mathsf{max}(t_c, t_a),\, t_b)$ into several subregions. Let $t_1 = \mathsf{max}(t_c, t_a)$ and let $t_{z+1} = t_b$. For $k \in \{2, 3, ..., z\}$, let t_k be the first time after t_{k-1} such that T_i either initiates or enacts a desired weight change. Notice that, by statement of the lemma, $\mathsf{r}(T_i^j) < \mathsf{max}(t_c, t_a)$, and by the definition of our decomposition,

$\mathsf{max}(t_c, t_a) \leq t_k$ for any $k \in \{1, ..., z+1\}$. Thus,

$$\text{for any } k \in \{1, ..., z+1\}, \mathsf{r}(T_i^j) < \mathsf{max}(t_c, t_a) \leq t_k. \tag{4.22}$$

Notice that, if we can bound the value of $\mathsf{A}(\mathcal{P}T, T_i, t_k, t_{k+1}) - \int_{t_k}^{t_{k+1}} \mathsf{SGwt}(T_i, u)du$ for every value of $k \in \{2, 3, ..., z\}$, then we can compute a bound for $\mathsf{A}(\mathcal{P}T, T_i, t_a, t_b) - \int_{t_a}^{t_b} \mathsf{SGwt}(T_i, u)du$.

Example (Figure 4.18). Consider the example in Figure 4.18, which depicts one processor that is assigned three tasks: T_1 and T_2, both of which have an execution time of 3 and a desired weight of 1/4; and T_3, which has $\mathsf{e}(T_3) = 2$ and an initial desired weight of 1/6 that changes to 1/10 at time 3 via Case (ii) of Rule N, then to 1/2 at time 5 via Case (i) of Rule N, and then again to 1/3 at time 7 via Case (ii) of Rule N. Notice that, because the change initiated at time 3 is via Case (ii) of Rule N, the change is not enacted until T_3^1's deadline. As a result, when a change is initiated at time 6, this new change cancels the previous change. Moreover, since the change enacted at time 6 is a weight increase, it is immediately enacted via Case (i) of Rule N. Finally, before the deviance of T_3^1 becomes zero, T_3 initiates a weight decrease at time 7, which is enacted once the T_3^1's deviance becomes zero at time 8. Notice that, if $t_a = \mathsf{r}(T_3^1)$ and $t_b = 8$, then there are three weight initiations and two enactments over the range $[\mathsf{r}(T_3^1), t_b)$. Thus, $t_1 = 3$, $t_2 = 6$, $t_3 = 7$, and $t_4 = t_b = 8$. $\square$

To continue the proof, let k denote some value in $\{1, 2, ..., z\}$. If T_i enacts a change at t_k, then since T_i does not initiate or enact a desired weight change until t_{k+1}, $\mathsf{SGwt}(T_i, u) = \mathsf{Gwt}(T_i, u)$. Thus, $\mathsf{A}(\mathcal{P}T, T_i, t_k, t_{k+1}) = \int_{t_k}^{t_{k+1}} \mathsf{SGwt}(T_i, u)du$ holds. Thus, we henceforth assume that T_i initiates a change at t_k. Since $t_k < t_{k+1}$, the change initiated at t_k must have been by either Case (ii) of Rule P or Case (ii) of Rule N. We now consider these two possibilities.

Sub-Case 1: Case (ii) of Rule P. By definition, T_i does not enact any desired weight changes over the range (t_k, t_{k+1}). Thus, we have the following property

(SC) The value $\mathsf{SDwt}(T_i, t)$ is constant within $[t_k, t_{k+1})$.

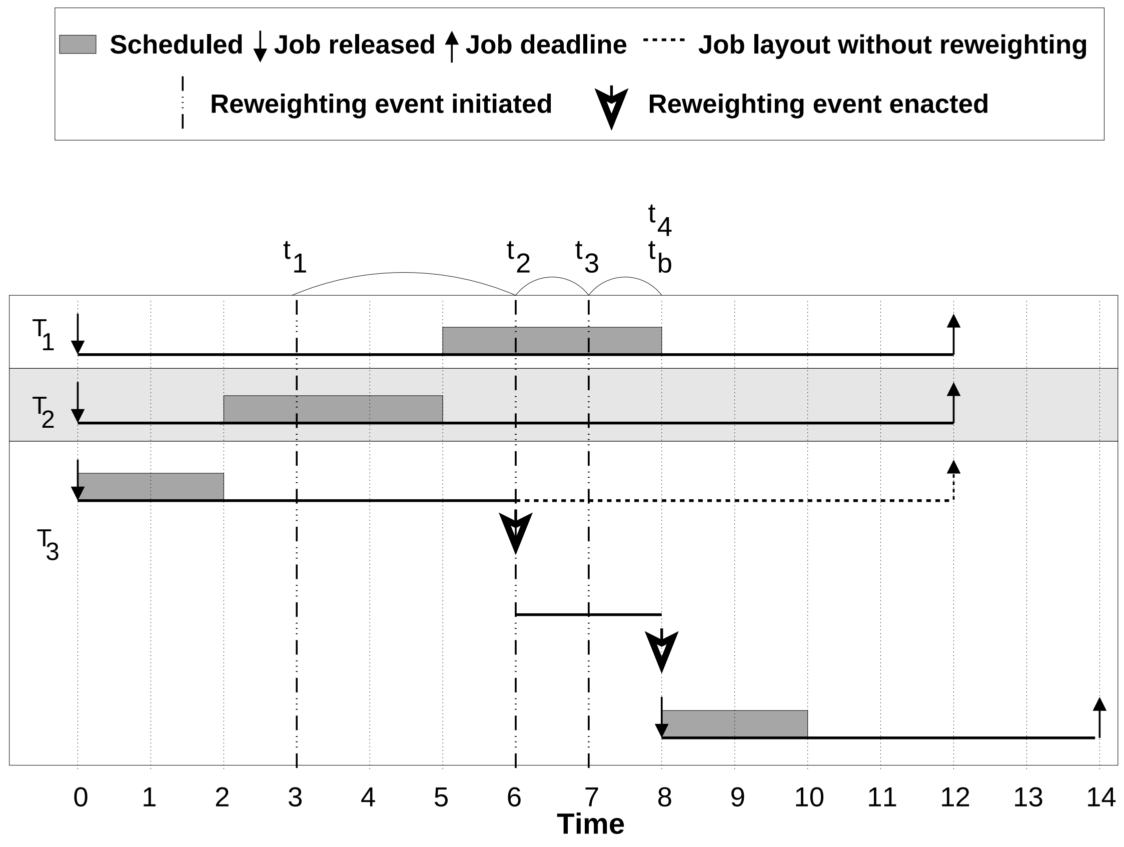


Figure 4.18: A one-processor example of the task decomposition in Lemma 4.4.

Thus, for brevity, we denote $\mathsf{SDwt}(T_i, t_k)$ as Ow.

Suppose that T_i initiates a change in its desired weight from Ow to Nw via Case (ii) of Rule P at t_k. Thus, for $t \in [t_k,\, t_{k+1})$, $\mathsf{Dwt}(T_i, t) = \mathsf{Nw}$. Since, by Property (SC), for $t \in [t_k,\, t_{k+1})$, $\mathsf{SDwt}(T_i, t) = \mathsf{Ow}$, $\mathsf{A}(\mathcal{PT},\, T_i,\, t_k,\, t_{k+1}) = \int_{t_k}^{t_{k+1}} \frac{\mathsf{Dwt}(T_i,t)}{\mathcal{TS}(P_{[q]},t)}dt$, and, by (4.5), $\mathsf{SGwt}(T_i, t) = \frac{\mathsf{SDwt}(T_i,t)}{\mathcal{TS}(P_{[q]},t)}$, it follows that

$$\mathsf{A}(\mathcal{PT},\, T_i,\, t_k,\, t_{k+1}) - \int_{t_k}^{t_{k+1}} \mathsf{SGwt}(T_i, t)dt = \int_{t_k}^{t_{k+1}} \frac{\mathsf{Nw} - \mathsf{Ow}}{\mathcal{TS}(P_{[q]}, t)}dt. \tag{4.23}$$

Thus, for the remainder of this subcase, we bound the value of $\int_{t_k}^{t_{k+1}} \frac{\mathsf{Nw}-\mathsf{Ow}}{\mathcal{TS}(P_{[q]},t)}dt$.

Recall that a task only changes its weight via Case (ii) of Rule P if the deviance of T_i is positive and the following equation holds:

$$\frac{\mathsf{Irem}(T_i^j, t_k)}{\mathsf{Ow}} \leq \frac{\mathsf{REM}(T_i^j, t_k)}{\mathsf{Nw}}. \tag{4.24}$$

By our decomposition and (4.20), $t_{k+1} \leq t_e \leq \mathsf{d}(T_i^j)$. Thus, by Property (D), $\int_{\mathsf{r}(T_i^j)}^{t_{k+1}} \mathsf{SGwt}(T_i, u)du \leq \mathsf{e}(T_i)$. Notice that, by (4.8), $\mathsf{Irem}(T_i^j, t_k) = \mathsf{e}(T_i^j) - \int_{\mathsf{r}(T_i^j)}^{t_k} \mathsf{SGwt}(T_i, u)du$. By substituting $\int_{\mathsf{r}(T_i^j)}^{t_{k+1}} \mathsf{SGwt}(T_i, u)du \leq \mathsf{e}(T_i)$ into $\mathsf{Irem}(T_i^j, t_k) = \mathsf{e}(T_i^j) - \int_{\mathsf{r}(T_i^j)}^{t_k} \mathsf{SGwt}(T_i, u)du$, it follows that $\int_{t_k}^{t_{k+1}} \mathsf{SGwt}(T_i, u)du \leq \mathsf{Irem}(T_i^j, t_k)$.

Since, by (4.5), $\mathsf{SGwt}(T_i, t) = \frac{\mathsf{SDwt}(T_i,t)}{\mathcal{TS}(P_{[q]},t)}$, and for any $t \in [t_k, t_{k+1})$, $\mathsf{SDwt}(T_i, t) = \mathsf{SDwt}(T_i, t_k) = \mathsf{Ow}$, by (4.11), we have

$$\int_{t_k}^{t_{k+1}} \frac{\mathsf{Ow}}{\mathcal{TS}(P_{[q]}, t)} dt = \int_{t_k}^{t_{k+1}} \mathsf{SGwt}(T_i, u)du \leq \mathsf{Irem}(T_i^j, t_k) \leq \mathsf{e}_{\mathsf{max}}(T_i). \tag{4.25}$$

We now consider two scenarios depending on whether $\mathsf{Ow} > \mathsf{Nw}$ or $\mathsf{Ow} < \mathsf{Nw}$. We first consider the scenario where $\mathsf{Ow} > \mathsf{Nw}$. In this case, since, by (4.6), $\mathcal{TS}(P_{[q]}, t) \geq 1$, it is trivial to show that

$$0 \geq \int_{t_k}^{t_{k+1}} \frac{\mathsf{Nw} - \mathsf{Ow}}{\mathcal{TS}(P_{[q]}, t)} dt$$

holds. By (4.25) and because $\mathsf{Nw} - \mathsf{Ow} \geq -\mathsf{Ow}$ and $\mathcal{TS}(P_{[q]}, t) \geq 1$, it follows that

$$0 \geq \int_{t_k}^{t_{k+1}} \frac{\mathsf{Nw} - \mathsf{Ow}}{\mathcal{TS}(P_{[q]}, t)} dt \geq \int_{t_k}^{t_{k+1}} \frac{-\mathsf{Ow}}{\mathcal{TS}(P_{[q]}, t)} dt \geq -\mathsf{e}_{\mathsf{max}}(T_i).$$

Thus, by (4.23), if T_i initiates a weight decrease via Case (ii) of Rule P at time t_k, then

$$0 \geq \mathsf{A}(\mathcal{P}T,\, T_i,\, t_k,\, t_{k+1}) - \int_{t_k}^{t_{k+1}} \mathsf{SGwt}(T_i, u)du \geq -\mathsf{e}_{\mathsf{max}}(T_i). \tag{4.26}$$

We now consider the scenario where $\mathsf{Ow} < \mathsf{Nw}$. In this case, since by (4.6), $\mathcal{TS}(P_{[q]}, t) \geq 1$, it is trivial to show that

$$0 \leq \int_{t_k}^{t_{k+1}} \frac{\mathsf{Nw} - \mathsf{Ow}}{\mathcal{TS}(P_{[q]}, t)} dt$$

holds. By (4.24) and (4.25), it follows that

$$\int_{t_k}^{t_{k+1}} \frac{1}{\mathcal{TS}(P_{[q]}, t)} dt \leq \frac{\mathsf{Irem}(T_i^j, t_k)}{\mathsf{Ow}} \leq \frac{\mathsf{REM}(T_i^j, t_k)}{\mathsf{Nw}}.$$

Thus, by (4.13),

$$\int_{t_k}^{t_{k+1}} \frac{\mathsf{Nw}}{\mathcal{TS}(P_{[q]}, t)} dt \le \mathsf{REM}(T_i^j, t_c) \le \mathsf{e}_{\mathsf{max}}(T_i).$$

Since, $\mathsf{Nw} > \mathsf{Ow} \ge 0$,

$$0 \le \int_{t_k}^{t_{k+1}} \frac{\mathsf{Nw} - \mathsf{Ow}}{\mathcal{TS}(P_{[q]}, t)} dt \le \mathsf{REM}(T_i^j, t_c) \le \mathsf{e}_{\mathsf{max}}(T_i).$$

Thus, by (4.23), if T_i initiates a weight increase via Case (ii) of Rule P at time t_k, then

$$0 \le \mathsf{A}(\mathcal{PT},\, T_i,\, t_k,\, t_{k+1}) - \int_{t_k}^{t_{k+1}} \mathsf{SGwt}(T_i, t) dt \le \mathsf{e}_{\mathsf{max}}(T_i). \tag{4.27}$$

Case (ii) of Rule N. Suppose that T_i changes its desired weight to Nw at time t_k via Rule N. If T_i initiates a desired weight decrease at time t_k (i.e., the change initiated is via Case (ii) of Rule N), it follows that for any $t \in [t_k,\, t_{k+1})$, $\mathsf{Dwt}(T_i, t) \le \mathsf{SDwt}(T_i, t)$. Thus, since $\mathsf{A}(\mathcal{PT},\, T_i,\, t_k,\, t_{k+1}) = \int_{t_k}^{t_{k+1}} \frac{\mathsf{SDwt}(T_i,u)}{\mathcal{TS}(P_{[q]},t)} du$ (by (4.17)), and $\mathsf{SGwt}(T_i, t) = \frac{\mathsf{SDwt}(T_i,t)}{\mathcal{TS}(P_{[q]},t)}$ (by (4.5)), it follows that $0 \le \mathsf{A}(\mathcal{PT},\, T_i,\, t_k,\, t_{k+1}) \le \int_{t_k}^{t_{k+1}} \mathsf{SGwt}(T_i, u) du$ holds.

We now show that $\int_{t_k}^{t_{k+1}} \mathsf{SGwt}(T_i, u) du \le \mathsf{e}_{\mathsf{max}}(T_i)$. By the definition of our decomposition and (4.20), $t_{k+1} \le t_b \le \mathsf{d}(T_i^j)$. Thus, by Property (D), $\int_{\mathsf{r}(T_i^j)}^{t_{k+1}} \mathsf{SGwt}(T_i, u) du \le \mathsf{e}(T_i)$. Since, by (4.22), $\mathsf{r}(T_i^j) < t_k$, and $\mathsf{SGwt}(T_i, t) \ge 0$ for all t, it follows that $\int_{t_k}^{t_{k+1}} \mathsf{SGwt}(T_i, u) du \le \int_{\mathsf{r}(T_i^j)}^{t_{k+1}} \mathsf{SGwt}(T_i, u) du \le \mathsf{e}(T_i) \le \mathsf{e}_{\mathsf{max}}(T_i)$. Thus, if T_i initiates a weight decrease via Case (ii) of Rule N at time t_k, then

$$0 \ge \mathsf{A}(\mathcal{PT},\, T_i,\, t_k,\, t_{k+1}) - \int_{t_k}^{t_{k+1}} \mathsf{SGwt}(T_i, u) du \ge -\mathsf{e}_{\mathsf{max}}(T_i). \tag{4.28}$$

By combining (4.26), (4.27), and (4.28), it follows that

$$-\mathsf{e}_{\mathsf{max}}(T_i) \cdot \mathcal{G} \le \mathsf{A}(\mathcal{PT},\, T_i,\, t_a,\, t_b) - \int_{t_a}^{t_b} \mathsf{SGwt}(T_i, u) du \le \mathsf{e}_{\mathsf{max}}(T_i) \cdot \mathcal{P}_2 \tag{4.29}$$

where $\mathcal{G}$ denotes the number of weight decreases by T_i that were initiated at or before t_b and enacted or canceled after t_a, over the range and $\mathcal{P}_2$ denotes the number of desired weight increases via Case (ii) of Rule P that were initiated at or before t_b and enacted or canceled after t_a.

Bounding $\int_{t_a}^{t_b} \mathsf{SGwt}(T_i, u)du - \mathsf{A}(\mathcal{CSW}, T_i, t_a, t_b)$. Notice that $\int_{t_a}^{t_b} \mathsf{SGwt}(T_i, u)du = \mathsf{A}(\mathcal{CSW}, T_i, t_a, t_b)$ unless T_i^j is halted by a desired weight change. By Property (HA), no change initiated over the range $(t_e, t_I]$ can halt T_i^j. Thus, if $\int_{t_a}^{t_b} \mathsf{SGwt}(T_i, u)du \neq \mathsf{A}(\mathcal{CSW}, T_i, t_a, t_b)$, then T_i^j enacted a desired weight change at t_e that halts T_i^j. Moreover, by the reweighting rules, if the change enacted at t_e halts T_i^j, then that change must also have been initiated at t_e and must have been by either Case (i) of Rule P or Case (i) of Rule N.

Suppose that the change initiated at t_e is via Case (i) of Rule N. Notice that, by Property (SW-2), $\mathsf{A}(\mathcal{CSW}, T_i, t_a, t_e) \leq \mathsf{Ae}(T_i^j)$. Moreover, at the smallest value of t_r such that $\mathsf{A}(\mathcal{SW}, T_i, \mathsf{r}(T_i^j), t_r) = \mathsf{Ae}(T_i^j)$ holds, T_i^j becomes inactive, i.e., $t_I = t_r$. Recall, from the definition of $\mathcal{CSW}$, that $\mathsf{A}(\mathcal{CSW}, T_i, \mathsf{r}(T_i^j), t) = \mathsf{A}(\mathcal{SW}, T_i, \mathsf{r}(T_i^j), t)$ up to the first t such that $\mathsf{A}(\mathcal{CSW}, T_i, \mathsf{r}(T_i^j), t) = \mathsf{Ae}(T_i^j)$. Thus, since, by (4.20), $t_e \leq t_b \leq t_I$, T_i^j receives an allocation of $\mathsf{SGwt}(T_i, t)$ at each instant $t \in [\mathsf{r}(T_i^u), t_b)$ in $\mathcal{CSW}$. Therefore, since $t_a \in [\mathsf{r}(T_i^j), t_b]$ (by the statement of the lemma), we have $\int_{t_a}^{t_b} \mathsf{SGwt}(T_i, u)du = \mathsf{A}(\mathcal{CSW}, T_i, t_a, t_b)$. Thus, if $\int_{t_a}^{t_b} \mathsf{SGwt}(T_i, u)du \neq \mathsf{A}(\mathcal{CSW}, T_i, t_a, t_b)$, then the change must have been via Case (i) of Rule P.

If the change initiated at t_e is via Case (i) of Rule P, then T_i^j has positive deviance at t_e and $t_e = t_I$. Thus, by (4.20), $t_e = t_b$. Since $t_e = t_b = t_I \leq \mathsf{d}(T_i^j)$ (by (4.20)), by Property (D), $0 \leq \int_{t_a}^{t_b} \mathsf{SGwt}(T_i, u)du \leq \mathsf{e}_{\mathsf{max}}(T_i)$. Also, by Lemma 4.3, since $t_b = t_I$, $0 \leq \mathsf{A}(\mathcal{CSW}, T_i, t_a, t_b) \leq \mathsf{e}_{\mathsf{max}}(T_i)$. Thus,

$$-\mathsf{e}_{\mathsf{max}}(T_i) \leq \int_{t_a}^{t_b} \mathsf{SGwt}(T_i, u)du - \mathsf{A}(\mathcal{CSW}, T_i, t_a, t_b) \leq \mathsf{e}_{\mathsf{max}}(T_i). \tag{4.30}$$

From (4.29) and (4.30), we have $-\mathsf{e}_{\mathsf{max}}(T_i) \cdot (\mathcal{G} + \mathcal{P}_1) \leq \mathsf{A}(\mathcal{PT}, T_i, t_a, t_b) - \mathsf{A}(\mathcal{CSW}, T_i, t_a, t_b) \leq \mathsf{e}_{\mathsf{max}}(T_i) \cdot (\mathcal{P}_1 + \mathcal{P}_2)$, where $\mathcal{G}$, $\mathcal{P}_1$, and $\mathcal{P}_2$ are as defined in the statement of the lemma. Thus, by (4.19)

$$-\mathsf{e}_{\mathsf{max}}(T_i) \cdot (\mathcal{G} + \mathcal{P}_1) \leq \mathsf{Pdrift}(T_i, t_b) - \mathsf{Pdrift}(T_i, t_a) \leq \mathsf{e}_{\mathsf{max}}(T_i) \cdot (\mathcal{P}_1 + \mathcal{P}_2),$$

which completes the proof of Lemma 4.4. □

Notice that, if a task T_i is inactive over the range $[t_a,\ t_b)$, then $\mathsf{Pdrift}(T_i, t_b) - \mathsf{Pdrift}(T_i, t_a) = 0$. Also notice that if T_i never changes its desired weight while the job T_i^j is active, then for any two times t_a and t_b such that $\mathsf{r}(T_i^j) \leq t_a \leq t_b \leq t_I$, where t_I is the time that T_i^j becomes inactive, $\mathsf{Pdrift}(T_i, t_b) - \mathsf{Pdrift}(T_i, t_a) = 0$. Thus, a task T_i can only incur partial drift over a range in which a job of T_i is active and T_i initiates a desired weight change. Such a scenario is addressed in Lemma 4.4. Thus, by iteratively applying Lemma 4.4, it is possible to show that the partial drift incurred over any range $[t_a, t_b]$ is bounded by $[-\mathsf{e}_{\mathsf{max}}(T_i) \cdot (\mathcal{G} + \mathcal{P}_1),\ \mathsf{e}_{\mathsf{max}}(T_i) \cdot (\mathcal{P}_1 + \mathcal{P}_2)]$, where $\mathcal{G}$, $\mathcal{P}_1$, and $\mathcal{P}_2$ denote, respectively, the number of weight decreases by T_i, the number of weight increases by T_i via Case (i) of Rule P, and the number of weight increases by T_i via Case (ii) of Rule P that were initiated at or before t_b and enacted or canceled after t_a. Assuming that t_b and T_i satisfy one of conditions (T-1), ..., (T-3).

This bound is summarized in the following corollary

Corollary 4.1. *Let T_i denote a task that is assigned to $P_{[q]}$ over some range $[t_a,\ t_b)$. If T_i and t_b satisfy one of Conditions (T-1), ..., (T-3). Then, $\mathsf{Pdrift}(T_i, t_b) - \mathsf{Pdrift}(T_i, t_a)$ is bounded by $[-\mathsf{e}_{\mathsf{max}}(T_i) \cdot (\mathcal{G} + \mathcal{P}_1),\ \mathsf{e}_{\mathsf{max}}(T_i) \cdot (\mathcal{P}_1 + \mathcal{P}_2)]$, where $\mathcal{G}$, $\mathcal{P}_1$, and $\mathcal{P}_2$ denote, respectively, the number of weight decreases, the number of weight increases via Case (i) of Rule P, and the number of weight increase via Case (ii) of Rule P by T_i that were initiated at or before t_b and enacted or canceled after t_a.*

Example (Figure 4.19). Consider the example in Figure 4.9, which depicts one processor that is assigned three tasks: T_1 and T_2, both of which have an execution time of 2 and a weight of 1/3; and T_3, which has $\mathsf{e}(T_3) = 1$ and an initial weight of 1/10 that changes to 1/7 at time 1 via Case (i) of Rule P and then to 1/3 at time 3 via Case (i) of Rule P. Inset (a) depicts the PEDF schedule. Inset **(b)** depicts T_3's partial drift. By Lemma 4.4, the maximal partial drift incurred by T_3 is one (its execution time) over both ranges [0, 1) and [1, 3) (even though the actual partial drift incurred is 0.1 over the range [0, 1) and 2/7 over the range [1, 3)). Notice that, Lemma 4.4 cannot be used to determine the maximal partial drift over the range [1, 2.5) because T_3^2 has not enacted a weight change. Moreover, notice that the last-initiated change at or before time 2.5 was enacted at time 1, i.e., $t_e = 1$, and $\mathsf{r}(T_3^2) = 1$. Hence, $t_e = \mathsf{r}(T_3^2)$. Thus, Condition (T-3) does not hold at time 2.5 (even though $1 = t_e \leq 2.5$ holds). On the other hand, the last-initiated change at or before time 3 was enacted at time 3, i.e., $t_e = 3$. Thus, in this case, $\mathsf{r}(T_3^2) < t_e$. Hence, Condition (T-3) holds at time 3 because $\mathsf{r}(T_3^2) < t_e \leq 3$. Corollary 4.1 is used to sum the partial drift over multiple jobs, i.e., by Corollary 4.1, the maximal partial drift over the range [0, 3) is two. □

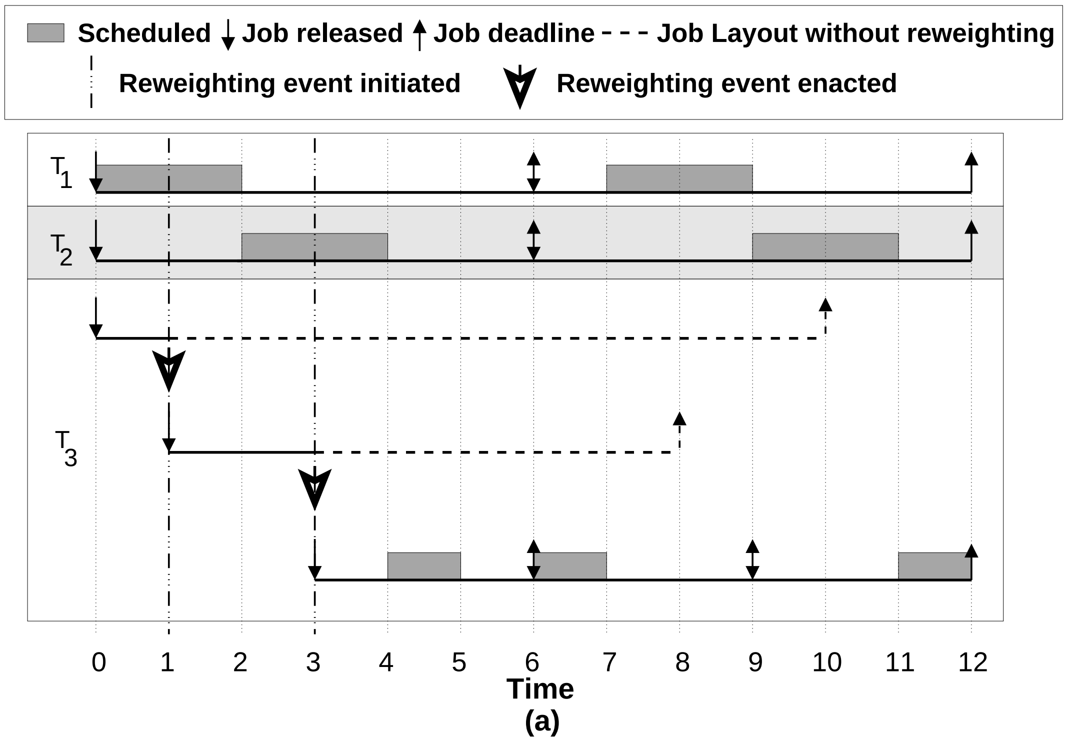
4.9.2 Relationship Between PT and IDEAL

Now that we have established the partial drift incurred by a change in the desired weight of a task, we can determine the difference between a task's allocation in the IDEAL and PT schedules. In this section, we show that this difference up to time t, which can be easily determined from (4.16) and (4.17) as $\int_0^t \mathsf{Dwt}(T_i, u) \cdot (\frac{1}{\mathcal{TD}(P_{[q]}, u)} - \frac{1}{\mathcal{TS}(P_{[q]}, u)})du$, is bounded by the range $[-\mathcal{C} \cdot \mathcal{X}, \mathcal{D} \cdot \mathcal{X}]$, where $\mathcal{C}$ denotes the number of weight increases via Case (ii) of Rule P, $\mathcal{D}$ denotes the number of weight decreases via Case (ii) of Rule P or Case (ii) of Rule N, and $\mathcal{X}$ denotes the maximal execution time of any task assigned to the same processor as T_i. Before we can establish this bound, we must establish some preliminarily lemmas.

Lemma 4.5. *If a task T_i initiates a desired weight decrease at time t_c from* Ow *to* Nw *while T_i^j is active, T_i is assigned to $P_{[q]}$, and the decrease is enacted or canceled at time t_e, then* $0 \leq \int_{t_c}^{t_e} \frac{\mathsf{SDwt}(T_i,t) - \mathsf{Dwt}(T_i,t)}{\mathcal{TS}(P_{[q]},t)\mathcal{TD}(P_{[q]},t)} dt \leq \mathsf{e}_{\mathsf{max}}(T_i)$.

Proof. Notice that, if $t_c = t_e$, then lemma is trivially true. Thus, for the remainder of this proof we assume that $t_c < t_e$. Since $t_c < t_e$, the change initiated at t_c is not immediately enacted. Moreover, by the definition of t_c and t_e, T_i does not initiate a desired weight change over the range (t_c, t_e). Thus, both the desired weight and the desired scheduling weight of T_i are constant over the range $[t_c, t_e)$. Specifically, for any $t \in [t_c, t_e)$, both

$$\mathsf{SDwt}(T_i, t) = \mathsf{Ow} \tag{4.31}$$


Scheduled
Job released
Job deadline
Job Layout without reweighting
Reweighting event initiated
Reweighting event enacted
T1
T2
T3
0 1 2 3 4 5 6 7 8 9 10 11 12
Time
(a)

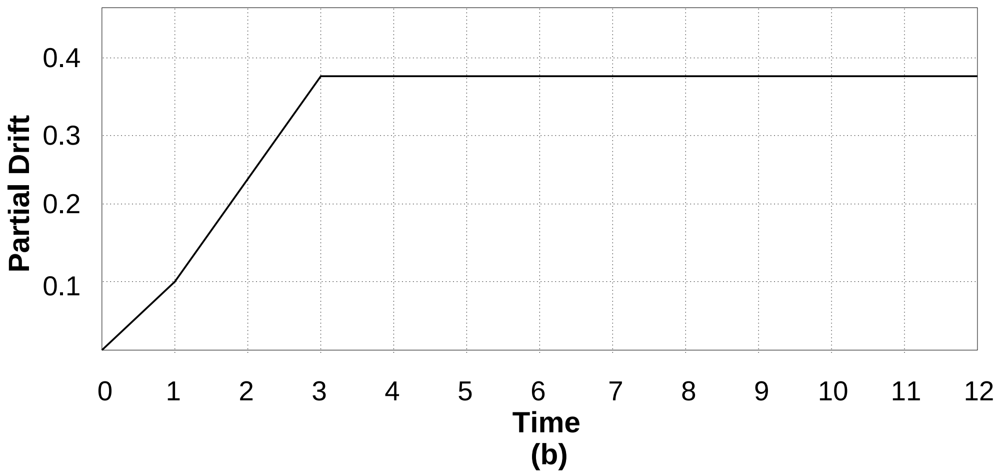


Figure 4.19: A one-processor example of multiple reweighting events. **(a)** The PEDF schedule. **(b)** T_3's partial drift.

and $\mathsf{Dwt}(T_i, t) = \mathsf{Nw}$ hold. Thus, if

$$0 \leq \int_{t_c}^{t_e} \frac{\mathsf{Ow} - \mathsf{Nw}}{\mathcal{TS}(P_{[q]}, t)\mathcal{TD}(P_{[q]}, t)} dt \leq \mathsf{e}_{\mathsf{max}}(T_i), \tag{4.32}$$

holds, then the proof is complete.

Since $\mathsf{Ow} > \mathsf{Nw}$, $t_c < t_e$, $\mathcal{TS}(P_{[q]}, t) \geq 1$ (by (4.6)), and $\mathcal{TD}(P_{[q]}, t) \geq 1$ (by (4.3)), it follows that

$$0 \leq \int_{t_c}^{t_e} \frac{\mathsf{Ow} - \mathsf{Nw}}{\mathcal{TS}(P_{[q]}, t)\mathcal{TD}(P_{[q]}, t)} dt$$

holds. In, the remainder of this proof, we establish the upper bound of (4.32).

Because $t_e > t_c$, the change initiated at time t_c was either by Case (ii) of Rule P or Case (ii) of Rule N. In either case, by Property (X), the weight change is either enacted or canceled by $\mathsf{d}(T_i^j)$, i.e., $t_e \leq \mathsf{d}(T_i^j)$. Since $\mathsf{r}(T_i^j) \leq t_c < t_e \leq \mathsf{d}(T_i^j)$, by Property (D),

$$\int_{t_c}^{t_e} \mathsf{SGwt}(T_i, t) dt \leq \mathsf{e}(T_i^j).$$

Moreover, since, by (4.31), for any $t \in [t_c, t_e)$, $\mathsf{SDwt}(T_i, t) = \mathsf{Ow}$, and by (4.5), $\mathsf{SGwt}(T_i, t) = \frac{\mathsf{SDwt}(T_i, t)}{\mathcal{TS}(P_{[q]}, t)}$, we have

$$\int_{t_c}^{t_e} \frac{\mathsf{Ow}}{\mathcal{TS}(P_{[q]}, t)} dt \leq \mathsf{e}(T_i^j). \tag{4.33}$$

Since, by the statement of the lemma, $\mathsf{Ow} > \mathsf{Nw} \geq 0$, and by (4.3), $\mathcal{TD}(P_{[q]}, t) \geq 1$, we have $\mathsf{Ow} - \mathsf{Nw} \leq \mathsf{Ow}$ and $\mathcal{TS}(P_{[q]}, t)\mathcal{TD}(P_{[q]}, t) \geq \mathcal{TS}(P_{[q]}, t)$. Hence, by (4.33),

$$\int_{t_c}^{t_e} \frac{\mathsf{Ow} - \mathsf{Nw}}{\mathcal{TS}(P_{[q]}, t)\mathcal{TD}(P_{[q]}, t)} dt \leq \int_{t_c}^{t_e} \frac{\mathsf{Ow}}{\mathcal{TS}(P_{[q]}, t)} dt \leq \mathsf{e}(T_i^j) \leq \mathsf{e}_{\mathsf{max}}(T_i)$$

Thus, it follows that the upper bound of (4.32) holds, which completes the proof. □

Notice that, if a task initiates a desired weight increase at t_c via Case (i) of Rule P or Case (i) of Rule N, then it is enacted at time t_c. Similarly, if the last-released job of T_i is not active or T_i has not release such a job, then the change is enacted at t_c. From these observations, we have the following lemma.

Lemma 4.6. *Let $T_i \in P_{[q]}$ be a task that initiates a desired weight increase from* Ow *to* Nw

at time t_c, let T_i^j be the last-released job (if any) of T_i, and let t_e denote the time the change is enacted or canceled. If T_i^j is not active at t_c or no such job exists, or the change initiated at t_c is via either Case (i) of Rule P or Case (i) of Rule N, then $\int_{t_c}^{t_e} \frac{\mathsf{SDwt}(T_i,t)-\mathsf{Dwt}(T_i,t)}{\mathcal{TS}(P_{[q]},t)\mathcal{TD}(P_{[q]},t)}dt = 0.$

Lemma 4.7. *If a task T_i initiates a desired weight increase at time t_c via Case (ii) of Rule P from Ow to Nw while T_i^j is active, T_i is assigned to $P_{[q]}$, and the increase is enacted or canceled at time t_e, then* $0 \leq \int_{t_c}^{t_e} \frac{\mathsf{Dwt}(T_i,t)-\mathsf{SDwt}(T_i,t)}{\mathcal{TS}(P_{[q]},t)\mathcal{TD}(P_{[q]},t)}dt \leq \mathsf{e}_{\mathsf{max}}(T_i)$.

Proof. Notice that, if $t_c = t_e$, then lemma is trivially true. Thus, for the remainder of the proof, we assume that $t_c < t_e$. Since $t_c < t_e$, the change initiated at t_c is not immediately enacted. Moreover, by the definition of t_c and t_e, T_i does not initiate a desired weight change over the range (t_c, t_e). Thus, both the desired weight and the desired scheduling weight of T_i are constant over the range $[t_c, t_e)$. Specifically, for any $t \in [t_c, t_e)$, both $\mathsf{SDwt}(T_i,t) = \mathsf{Ow}$ and $\mathsf{Dwt}(T_i,t) = \mathsf{Nw}$ hold. Thus, if

$$0 \leq \int_{t_c}^{t_e} \frac{\mathsf{Nw}-\mathsf{Ow}}{\mathcal{TS}(P_{[q]},t)\mathcal{TD}(P_{[q]},t)}dt \leq \mathsf{e}_{\mathsf{max}}(T_i). \tag{4.34}$$

holds, then the proof is complete.

Since $\mathsf{Nw} > \mathsf{Ow}$, $t_c < t_e$, $\mathcal{TS}(P_{[q]},t) \geq 1$ (by (4.6)), and $\mathcal{TD}(P_{[q]},t) \geq 1$ (by (4.3)), it follows that

$$0 \leq \int_{t_c}^{t_e} \frac{\mathsf{Nw}-\mathsf{Ow}}{\mathcal{TS}(P_{[q]},t)\mathcal{TD}(P_{[q]},t)}dt$$

holds. In the remainder of this proof, we establish the upper bound of (4.34). By Property (X), the weight change initiated at t_c is either enacted or canceled by $\mathsf{d}(T_i^j)$, i.e., $t_e \leq \mathsf{d}(T_i^j)$. Thus, by Property (D), $\int_{\mathsf{r}(T_i^j)}^{t_e} \mathsf{SGwt}(T_i,u)du \leq \mathsf{e}(T_i)$. By (4.8), $\mathsf{Irem}(T_i^j,t_c) = \mathsf{e}(T_i^j) - \int_{\mathsf{r}(T_i^j)}^{t_c} \mathsf{SGwt}(T_i,u)du$. By substituting $\int_{\mathsf{r}(T_i^j)}^{t_e} \mathsf{SGwt}(T_i,u)du \leq \mathsf{e}(T_i)$ into $\mathsf{Irem}(T_i^j,t_c) = \mathsf{e}(T_i^j) - \int_{\mathsf{r}(T_i^j)}^{t_c} \mathsf{SGwt}(T_i,u)du$, it follows that $\int_{t_c}^{t_e} \mathsf{SGwt}(T_i,u)du \leq \mathsf{Irem}(T_i^j,t_c)$.

Since, by (4.5), $\mathsf{SGwt}(T_i,t) = \frac{\mathsf{SDwt}(T_i,t)}{\mathcal{TS}(P_{[q]},t)}$, for any $t \in [t_c, t_e)$, $\mathsf{SDwt}(T_i,t) = \mathsf{SDwt}(T_i,t_c) = \mathsf{Ow}$, and by (4.11), $\mathsf{Irem}(T_i^j,t_v) \leq \mathsf{e}_{\mathsf{max}}(T_i)$, we have

$$\int_{t_c}^{t_e} \frac{\mathsf{Ow}}{\mathcal{TS}(P_{[q]},t)}dt = \int_{t_c}^{t_e} \mathsf{SGwt}(T_i,u)du \leq \mathsf{Irem}(T_i^j,t_v) \leq \mathsf{e}_{\mathsf{max}}(T_i). \tag{4.35}$$

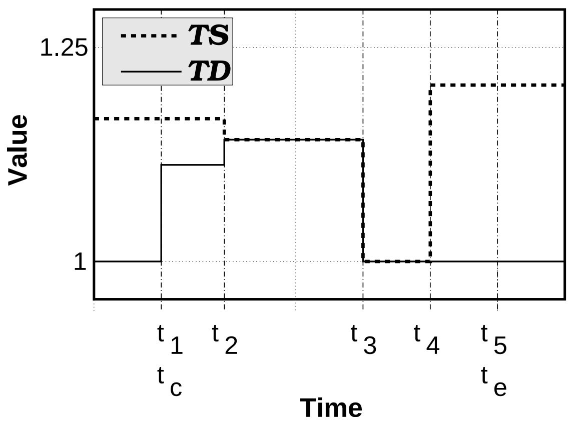


Figure 4.20: Decomposition of a task for Lemma 4.7.

We now decompose the interval $[t_c,\, t_e)$ into the set of regions $\{[t_1,\, t_2), [t_2,\, t_3), ..., [t_z,\, t_{z+1})\}$. Let $t_1 = t_c$ and $t_{z+1} = t_e$. For, $k \in \{1, 2, ..., z-1\}$, let t_{k+1} equal the first time after t_k such that the value of either $\mathcal{T}\mathsf{S}(P_{[q]}, t)$ or $\mathcal{T}\mathsf{D}(P_{[q]}, t)$ changes. Note that the value of either $\mathcal{T}\mathsf{S}(P_{[q]}, t)$ or $\mathcal{T}\mathsf{D}(P_{[q]}, t)$ can only change as a result of a task enacting or initiating a desired weight change. Thus, there exist a discrete number of times over the region $[t_c,\, t_e)$ that the value of either $\mathcal{T}\mathsf{S}(P_{[q]}, t)$ or $\mathcal{T}\mathsf{D}(P_{[q]}, t)$ can change, which makes this decomposition possible. An example decomposition is given in Figure 4.20.

Since, by the definition of our decomposition, the value of $\mathcal{T}\mathsf{S}(P_{[q]}, t)$ does not change within $[t_k,\, t_{k+1})$, where $k = \{1, 2, ..., z\}$, (4.35) implies

$$\mathsf{Ow} \cdot \left(\sum_{k=1,...,z} \frac{t_{k+1} - t_k}{\mathcal{T}\mathsf{S}(P_{[q]}, t_k)} \right) \leq \mathsf{Irem}(T_i^j, t_c). \tag{4.36}$$

Additionally, since the value of $\mathcal{T}\mathsf{D}(P_{[q]}, t)$ also does not change within $[t_k,\, t_{k+1})$, where $k = \{1, 2, ..., z\}$, we have

$$\int_{t_c}^{t_e} \frac{\mathsf{Nw} - \mathsf{Ow}}{\mathcal{T}\mathsf{S}(P_{[q]}, t)\mathcal{T}\mathsf{D}(P_{[q]}, t)} dt = (\mathsf{Nw} - \mathsf{Ow}) \sum_{k=1,...,z} \frac{t_{k+1} - t_k}{\mathcal{T}\mathsf{D}(P_{[q]}, t_k)\mathcal{T}\mathsf{S}(P_{[q]}, t_k)}. \tag{4.37}$$

Notice that, since $\mathsf{Nw} > \mathsf{Ow}$ and $\mathcal{T}\mathsf{D}(P_{[q]}, t) \geq 1$ holds (by (4.3)), it follows that

$$(\mathsf{Nw} - \mathsf{Ow}) \sum_{k=1,\ldots,z} \frac{t_{k+1} - t_k}{\mathcal{T}\mathsf{D}(P_{[q]}, t_k)\mathcal{T}\mathsf{S}(P_{[q]}, t_k)} \leq (\mathsf{Nw} - \mathsf{Ow}) \sum_{k=1,\ldots,z} \frac{t_{k+1} - t_k}{\mathcal{T}\mathsf{S}(P_{[q]}, t_k)}$$

Thus, by (4.36) and the fact that $\mathsf{Ow} > 0$ holds, it follows that

$$(\mathsf{Nw} - \mathsf{Ow}) \sum_{k=1,\ldots,z} \frac{t_{k+1} - t_k}{\mathcal{T}\mathsf{D}(P_{[q]}, t_k)\mathcal{T}\mathsf{S}(P_{[q]}, t_k)} \leq (\mathsf{Nw} - \mathsf{Ow}) \frac{\mathsf{Irem}(T_i^j, t_c)}{\mathsf{Ow}}.$$

Thus, by (4.37),

$$\int_{t_c}^{t_e} \frac{\mathsf{Nw} - \mathsf{Ow}}{\mathcal{T}\mathsf{S}(P_{[q]}, t)\mathcal{T}\mathsf{D}(P_{[q]}, t)} dt \leq (\mathsf{Nw} - \mathsf{Ow}) \frac{\mathsf{Irem}(T_i^j, t_c)}{\mathsf{Ow}}.$$

Thus, if we can show that

$$(\mathsf{Nw} - \mathsf{Ow}) \frac{\mathsf{Irem}(T_i^j, t_c)}{\mathsf{Ow}} \leq \mathsf{e}_{\mathsf{max}}(T_i)$$

holds, then the proof is complete.

Recall that the change initiated at t_c was via Case (ii) of Rule P. Thus,

$$\frac{\mathsf{Irem}(T_i^j, t_c)}{\mathsf{Ow}} \leq \frac{\mathsf{REM}(T_i^j, t_c)}{\mathsf{Nw}}.$$

Therefore, since $\mathsf{Nw} - \mathsf{Ow} > 0$,

$$(\mathsf{Nw} - \mathsf{Ow}) \frac{\mathsf{Irem}(T_i^j, t_c)}{\mathsf{Ow}} \leq (\mathsf{Nw} - \mathsf{Ow}) \frac{\mathsf{REM}(T_i^j, t_c)}{\mathsf{Nw}}.$$

By rearranging terms, we get

$$(\mathsf{Nw} - \mathsf{Ow}) \frac{\mathsf{Irem}(T_i^j, t_c)}{\mathsf{Ow}} \leq \left(1 - \frac{\mathsf{Ow}}{\mathsf{Nw}}\right) \mathsf{REM}(T_i^j, t_c).$$

Since $\mathsf{Ow} < \mathsf{Nw}$ and $\mathsf{REM}(T_i^j, t_c) \leq \mathsf{e}(T_i^j) \leq \mathsf{e}_{\mathsf{max}}(T_i)$ (by (4.13)), it follows that

$$(\mathsf{Nw} - \mathsf{Ow}) \frac{\mathsf{Irem}(T_i^j, t_c)}{\mathsf{Ow}} \leq \left(1 - \frac{\mathsf{Ow}}{\mathsf{Nw}}\right) \mathsf{REM}(T_i^j, t_c) \leq \mathsf{REM}(T_i^j, t_c) \leq \mathsf{e}(T_i^j) \leq \mathsf{e}_{\mathsf{max}}(T_i).$$

This completes the proof. □

Before continuing, we introduce an additional function that will facilitate our discussion. Let $\gamma(T_i, t)$ be defined as

$$\gamma(T_i, t) = \begin{cases} 0, & \text{if } T_i \in \mathsf{ACT}(P_{[q]}, t) \\ \frac{\mathsf{SDwt}(T_i,t) - \mathsf{Dwt}(T_i,t)}{\mathcal{TS}(P_{[q]},t)\mathcal{TD}(P_{[q]},t)}, & \text{otherwise,} \end{cases} \tag{4.38}$$

where $P_{[q]}$ is the processor that T_i is assigned to at time t.

Lemma 4.8. *Let t_a and t_b be any two times such that $t_a < t_b$, and the system is not reset over the interval (t_a, t_b). Let $\mathcal{D}$ denote the number of desired weight decreases for T_i that are both initiated at or before t_b and enacted or canceled after t_a, and let $\mathcal{C}$ denote the number of desired weight increases via Case (ii) of Rule P for T_i that are both initiated at or before t_b and enacted or canceled after t_a. Then,*

$$\begin{aligned} -\mathcal{C} \cdot \mathsf{e_{max}}(T_i) &\leq \int_{t_a}^{t_b} \mathsf{min}\,(0, \gamma(T_i, t))\, dt, \\ \mathcal{D} \cdot \mathsf{e_{max}}(T_i) &\geq \int_{t_a}^{t_b} \mathsf{max}\,(0, \gamma(T_i, t))\, dt. \end{aligned}$$

Proof. Notice that, if $t_a = t_b$, then the lemma is trivially true. Thus, for the remainder of this proof we assume that $t_a < t_b$. If T_i initiates a change before t_a that is not enacted or canceled until after t_a, then let t'_a denote the time that change was initiated; otherwise, let $t'_a = t_a$. If T_i initiates a change before t_b that is not enacted or canceled until after t_b, then let t'_b denote the time that change is enacted or canceled; otherwise, let $t'_b = t_b$. Since $t'_a \leq t_a$ and $t_b \leq t'_b$, it suffices to bound $\int_{t'_a}^{t'_b} \mathsf{min}(0, \gamma(T_i, t))dt$ and $\int_{t'_a}^{t'_b} \mathsf{max}(0, \gamma(T_i, t))dt$.

We now decompose the internal $[t'_a, t'_b)$ into the set of regions $\{[t_1, t_2), [t_2, t_3), ..., [t_z, t_{z+1})\}$. Let $t_1 = t'_a$, let $t_{z+1} = t'_b$, and for $k \in \{1, 2, ..., z-1\}$, let t_{k+1} the next time after t_k that T_i initiated or enacted a desired weight change. Notice that, by the definitions of $\mathcal{C}$ and $\mathcal{D}$ given in the statement of the lemma, there are $\mathcal{C}$ values of $k \in \{1, 2, ..., z\}$ such that at t_k, T_i initiated a desired weight increase via Case (ii) of Rule P, and $\mathcal{D}$ values of $k \in \{1, 2, ..., z\}$ such that at t_k, T_i initiated a desired weight decrease. Thus, we can complete the proof by showing that, for any value of $k \in \{1, 2, ...z\}$,

- if T_i initiates a desired weight increase via Case (ii) of Rule P at time t_k, then $-\mathsf{e_{max}}(T_i) \le \int_{t_k}^{t_{k+1}} \mathsf{min}(0, \gamma(T_i, t))dt$;
- if T_i initiates a desired weight decrease at time t_k, then $\mathsf{e_{max}}(T_i) \ge \int_{t_k}^{t_{k+1}} \mathsf{max}(0, \gamma(T_i, t))dt$;
- if T_i initiates any other type of desired weight change at t_k or does not initiate a desired weight change at t_k, then $0 = \int_{t_k}^{t_{k+1}} \mathsf{max}(0, \gamma(T_i, t))dt = \int_{t_k}^{t_{k+1}} \mathsf{min}(0, \gamma(T_i, t))dt$.

Before we discuss these three case, notice that, for all values of $k \in \{1, 2, ..., z\}$, for any $t \in [t_k, t_{k+1})$, $\mathsf{SDwt}(T_i, t) = \mathsf{SDwt}(T_i, t_k)$ and $\mathsf{Dwt}(T_i, t) = \mathsf{Dwt}(T_i, t_k)$ both hold since T_i does not initiate or enact a change within (t_k, t_{k+1}), Thus, for any $k \in \{1, 2, ..., z\}$, if $\mathsf{Dwt}(T_i, t_k) = \mathsf{SDwt}(T_i, t_k)$ holds, then $\int_{t_k}^{t_{k+1}} \mathsf{min}(0, \gamma(T_i, t))dt = \int_{t_k}^{t_{k+1}} \mathsf{max}(0, \gamma(T_i, t))dt = 0$ holds as well.

T_i initiates a desired weight increase via Case (ii) of Rule P at time t_k. If the change initiated at t_k is enacted immediately, then $\mathsf{Dwt}(T_i, t_k) = \mathsf{SDwt}(T_i, t_k)$, which as we already discussed implies $\int_{t_k}^{t_{k+1}} \mathsf{min}(0, \gamma(T_i, t))dt = \int_{t_k}^{t_{k+1}} \mathsf{max}(0, \gamma(T_i, t))dt = 0$. Thus, we assume that the change initiated at t_k is not immediately enacted. Thus, by the definition of our decomposition, t_{k+1} represents the time that T_i enacts or cancels the change initiated at t_k. Thus, by Lemma 4.7, $0 \le \int_{t_k}^{t_{k+1}} \frac{\mathsf{Dwt}(T_i,t) - \mathsf{SDwt}(T_i,t)}{\mathcal{TS}(P_{[q]},t)\mathcal{TD}(P_{[q]},t)} dt \le \mathsf{e_{max}}(T_i)$, which implies

$$-\mathsf{e_{max}}(T_i) \le \int_{t_k}^{t_{k+1}} \mathsf{min}(0, \gamma(T_i, t))dt. \tag{4.39}$$

T_i initiates a desired weight decrease at time t_k. If the change initiated at t_k is enacted immediately, then $\mathsf{Dwt}(T_i, t_k) = \mathsf{SDwt}(T_i, t_k)$, which as we already discussed implies $\int_{t_k}^{t_{k+1}} \mathsf{min}(0, \gamma(T_i, t))dt = \int_{t_k}^{t_{k+1}} \mathsf{max}(0, \gamma(T_i, t))dt = 0$. Thus, we assume that the change initiated at t_k is not immediately enacted. In this case, by the definition of our decomposition, t_{k+1} represents the time that T_i enacts or cancels the change initiated at t_k. Thus, by Lemma 4.5, $0 \le \int_{t_k}^{t_{k+1}} \frac{\mathsf{SDwt}(T_i,t) - \mathsf{Dwt}(T_i,t)}{\mathcal{TS}(P_{[q]},t)\mathcal{TD}(P_{[q]},t)} dt \le \mathsf{e_{max}}(T_i)$, which implies

$$\mathsf{e_{max}}(T_i) \ge \int_{t_k}^{t_{k+1}} \mathsf{max}(0, \gamma(T_i, t))dt. \tag{4.40}$$

T_i initiates alternative change at t_k or does not initiate a change at t_k. If T_i initiates an alternative type of desired weight change or does not initaite a desired weight change at t_k, then $\mathsf{Dwt}(T_i, t_k) = \mathsf{SDwt}(T_i, t_k)$, which as we already discussed implies

$$\int_{t_k}^{t_{k+1}} \mathsf{min}(0, \gamma(T_i, t))dt = \int_{t_k}^{t_{k+1}} \mathsf{max}(0, \gamma(T_i, t))dt = 0. \tag{4.41}$$

Since, there are at most $\mathcal{C}$ values of $k \in \{1, 2, ..., z\}$ such that (4.39) holds, $\mathcal{D}$ values of $k \in \{1, 2, ..., z\}$ such that (4.40) holds, and for every other value of $k \in \{1, 2, ..., z\}$, (4.41) holds, we have

$$-\mathcal{C} \cdot \mathsf{e}_{\mathsf{max}}(T_i) \quad \leq \quad \int_{t_a}^{t_b} \mathsf{min}\left(0, \gamma(T_i, t)\right) dt \leq \int_{t'_a}^{t'_b} \mathsf{min}\left(0, \gamma(T_i, t)\right) dt$$

and

$$\mathcal{D} \cdot \mathsf{e}_{\mathsf{max}}(T_i) \quad \geq \quad \int_{t_a}^{t_b} \mathsf{max}\left(0, \gamma(T_i, t)\right) dt \geq \int_{t'_a}^{t'_b} \mathsf{max}\left(0, \gamma(T_i, t)\right) dt.$$

This completes the proof. □

Having proven the prerequisite lemmas, we can now show that the difference between $\frac{1}{\mathcal{T}\mathsf{D}(P_{[q]},t)}$ and $\frac{1}{\mathcal{T}\mathsf{S}(P_{[q]},t)}$ is a function of the number of reweighting events. Bounding this difference will enable us to bound the difference between T_i's allocations in the IDEAL and PT schedules, which is given by $\int_0^t \mathsf{Dwt}(T_i, u) \cdot (\frac{1}{\mathcal{T}\mathsf{D}(P_{[q]},u)} - \frac{1}{\mathcal{T}\mathsf{S}(P_{[q]},u)})du$.

Lemma 4.9. *Let t_a and t_b be any two times such that $t_a < t_b$, and the system is not reset over the interval (t_a, t_b). If $\mathcal{D}$ denotes the number of desired weight decreases by tasks assigned to $P_{[q]}$ that are both initiated at or before t_b and enacted or canceled after t_a and $\mathcal{C}$ denotes the number of desired weight increases via Case (ii) of Rule P by tasks assigned to $P_{[q]}$ that are both initiated at or before t_b and enacted or canceled after t_a, then*

$$-\mathcal{C} \cdot \mathcal{X} \quad \leq \quad \int_{t_a}^{t_b} \mathsf{min}\left(0, \frac{1}{\mathcal{T}\mathsf{D}(P_{[q]}, t)} - \frac{1}{\mathcal{T}\mathsf{S}(P_{[q]}, t)}\right) dt,$$

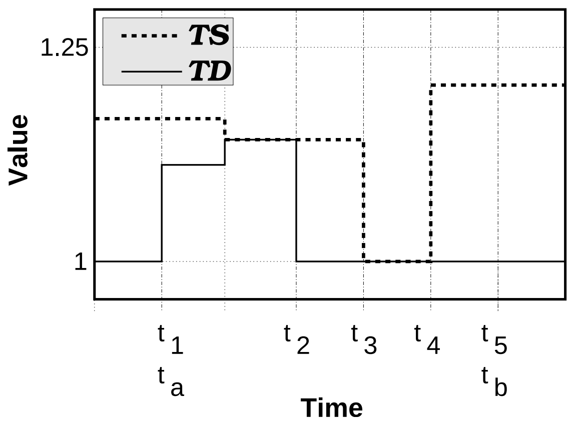


Figure 4.21: Decomposition of a task for Lemma 4.9.

$$\mathcal{D} \cdot \mathcal{X} \quad \geq \quad \int_{t_a}^{t_b} \mathsf{max}\left(0, \frac{1}{\mathcal{T}\mathsf{D}(P_{[q]}, t)} - \frac{1}{\mathcal{T}\mathsf{S}(P_{[q]}, t)}\right) dt,$$

where $\mathcal{X}$ is the largest execution time of any task assigned to $P_{[q]}$ over the range $[t_a, t_b)$.

Proof. We begin by decomposing the interval $[t_a, t_b)$ into several subregions. Let $t_1 = t_a$ and let $t_{z+1} = t_b$. For $k \in \{1, 2, ..., z-1\}$, let t_{k+1} be the first time after t_k such that at least one of the following conditions holds:

1. If $\mathcal{T}\mathsf{S}(P_{[q]}, t_k) = 1$, then $\mathcal{T}\mathsf{S}(P_{[q]}, t_{k+1}) > 1$.

2. If $\mathcal{T}\mathsf{D}(P_{[q]}, t_k) = 1$, then $\mathcal{T}\mathsf{D}(P_{[q]}, t_{k+1}) > 1$.

3. If $\mathcal{T}\mathsf{S}(P_{[q]}, t_k) > 1$, then $\mathcal{T}\mathsf{S}(P_{[q]}, t_{k+1}) = 1$.

4. If $\mathcal{T}\mathsf{D}(P_{[q]}, t_k) > 1$, then $\mathcal{T}\mathsf{D}(P_{[q]}, t_{k+1}) = 1$.

In other words, t_{k+1} denotes the time at which the total desired weight or total desired scheduling weight changes from over-utilizing $P_{[q]}$ to either under- or fully-utilizing $P_{[q]}$, or vice versa. An example of this decomposition is given in Figure 4.21. Notice that it is possible that neither the total desired scheduling weight nor the total desired weight will change from over-utilizing to under/fully-utilizing a processor or vice versa. In this case, $t_1 = t_a$ and $t_2 = t_b$.

Before continuing, notice that $\frac{1}{\mathcal{T}\mathsf{D}(P_{[q]},t)} - \frac{1}{\mathcal{T}\mathsf{S}(P_{[q]},t)}$ can be rearranged to equal $\frac{\mathcal{T}\mathsf{S}(P_{[q]},t) - \mathcal{T}\mathsf{D}(P_{[q]},t)}{\mathcal{T}\mathsf{D}(P_{[q]},t)\mathcal{T}\mathsf{S}(P_{[q]},t)}$. Thus, since our decomposition completely covers the range $[t_a, t_b)$, we have

$$\begin{aligned}\int_{t_a}^{t_b} \mathsf{min}\left(0, \tfrac{1}{\mathcal{T}\mathsf{D}(P_{[q]},t)} - \tfrac{1}{\mathcal{T}\mathsf{S}(P_{[q]},t)}\right) dt &= \int_{t_a}^{t_b} \mathsf{min}\left(0, \tfrac{\mathcal{T}\mathsf{S}(P_{[q]},t)-\mathcal{T}\mathsf{D}(P_{[q]},t)}{\mathcal{T}\mathsf{D}(P_{[q]},t)\mathcal{T}\mathsf{S}(P_{[q]},t)}\right) dt \\ &= \sum \int_{t_k}^{t_{k+1}} \mathsf{min}\left(0, \tfrac{\mathcal{T}\mathsf{S}(P_{[q]},t)-\mathcal{T}\mathsf{D}(P_{[q]},t)}{\mathcal{T}\mathsf{D}(P_{[q]},t)\mathcal{T}\mathsf{S}(P_{[q]},t)}\right) dt,\end{aligned} \tag{4.42}$$

$$\begin{aligned}\int_{t_a}^{t_b} \mathsf{max}\left(0, \tfrac{1}{\mathcal{T}\mathsf{D}(P_{[q]},t)} - \tfrac{1}{\mathcal{T}\mathsf{S}(P_{[q]},t)}\right) dt &= \int_{t_a}^{t_b} \mathsf{max}\left(0, \tfrac{\mathcal{T}\mathsf{S}(P_{[q]},t)-\mathcal{T}\mathsf{D}(P_{[q]},t)}{\mathcal{T}\mathsf{D}(P_{[q]},t)\mathcal{T}\mathsf{S}(P_{[q]},t)}\right) dt \\ &= \sum \int_{t_k}^{t_{k+1}} \mathsf{max}\left(0, \tfrac{\mathcal{T}\mathsf{S}(P_{[q]},t)-\mathcal{T}\mathsf{D}(P_{[q]},t)}{\mathcal{T}\mathsf{D}(P_{[q]},t)\mathcal{T}\mathsf{S}(P_{[q]},t)}\right) dt.\end{aligned} \tag{4.43}$$

Thus, it suffices to bound the values of

$$\int_{t_k}^{t_{k+1}} \mathsf{min}\left(0, \frac{\mathcal{T}\mathsf{S}(P_{[q]},t)-\mathcal{T}\mathsf{D}(P_{[q]},t)}{\mathcal{T}\mathsf{D}(P_{[q]},t)\mathcal{T}\mathsf{S}(P_{[q]},t)}\right) dt$$

and

$$\int_{t_k}^{t_{k+1}} \mathsf{max}\left(0, \frac{\mathcal{T}\mathsf{S}(P_{[q]},t)-\mathcal{T}\mathsf{D}(P_{[q]},t)}{\mathcal{T}\mathsf{D}(P_{[q]},t)\mathcal{T}\mathsf{S}(P_{[q]},t)}\right) dt$$

for each interval $[t_k,\, t_{k+1})$. We now consider the different possible values for $\mathcal{T}\mathsf{S}(P_{[q]},t)$ and $\mathcal{T}\mathsf{S}(P_{[q]},t)$ for any $t \in [t_k,\, t_{k+1})$.

Interval type 1: $\mathcal{T}\mathsf{S}(P_{[q]},t) = 1$ and $\mathcal{T}\mathsf{D}(P_{[q]},t) = 1$. If $\mathcal{T}\mathsf{S}(P_{[q]},t) = 1$ and $\mathcal{T}\mathsf{D}(P_{[q]},t) = 1$ both hold, then

$$\frac{\mathcal{T}\mathsf{S}(P_{[q]},t)-\mathcal{T}\mathsf{D}(P_{[q]},t)}{\mathcal{T}\mathsf{D}(P_{[q]},t)\mathcal{T}\mathsf{S}(P_{[q]},t)} = 0. \tag{4.44}$$

Interval type 2: $\mathcal{T}\mathsf{S}(P_{[q]},t) > 1$ and $\mathcal{T}\mathsf{D}(P_{[q]},t) > 1$. By (4.3) and (4.6), if $\mathcal{T}\mathsf{S}(P_{[q]},t) > 1$ and $\mathcal{T}\mathsf{D}(P_{[q]},t) > 1$ both hold, then $\mathcal{T}\mathsf{S}(P_{[q]},t) = \sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{SDwt}(T_i,t)$ and $\mathcal{T}\mathsf{D}(P_{[q]},t) = \sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{Dwt}(T_i,t)$ hold. Thus,

$$\frac{\mathcal{T}\mathsf{S}(P_{[q]},t)-\mathcal{T}\mathsf{D}(P_{[q]},t)}{\mathcal{T}\mathsf{D}(P_{[q]},t)\mathcal{T}\mathsf{S}(P_{[q]},t)}$$

can be rewritten as

$$\frac{\sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{SDwt}(T_i,t) - \sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{Dwt}(T_i,t)}{\mathcal{T}\mathsf{D}(P_{[q]},t)\mathcal{T}\mathsf{S}(P_{[q]},t)},$$

which equals

$$\sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \frac{\mathsf{SDwt}(T_i,t) - \mathsf{Dwt}(T_i,t)}{\mathcal{TD}(P_{[q]},t)\mathcal{TS}(P_{[q]},t)} = \sum_{T_i \in P_{[q]}} \gamma(T_i,t).$$

Thus,

$$\frac{\mathcal{TS}(P_{[q]},t) - \mathcal{TD}(P_{[q]},t)}{\mathcal{TD}(P_{[q]},t)\mathcal{TS}(P_{[q]},t)} = \sum_{T_i \in P_{[q]}} \gamma(T_i,t). \tag{4.45}$$

Interval type 3: $\mathcal{TS}(P_{[q]},t) > 1$ and $\mathcal{TD}(P_{[q]},t) = 1$. By (4.6), if $\mathcal{TS}(P_{[q]},t) > 1$ holds, then $\mathcal{TS}(P_{[q]},t) = \sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{SDwt}(T_i,t)$ holds. Thus,

$$\frac{\mathcal{TS}(P_{[q]},t) - \mathcal{TD}(P_{[q]},t)}{\mathcal{TD}(P_{[q]},t)\mathcal{TS}(P_{[q]},t)} = \frac{\sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{SDwt}(T_i,t) - 1}{\mathcal{TD}(P_{[q]},t)\mathcal{TS}(P_{[q]},t)}.$$

By (4.3), $\mathcal{TD}(P_{[q]},t) = 1$, implies $\sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{Dwt}(T_i,t) \leq 1$. Thus, since $1 < \mathcal{TS}(P_{[q]},t) = \sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{SDwt}(T_i,t)$, we have

$$0 \leq \frac{\sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{SDwt}(T_i,t) - 1}{\mathcal{TD}(P_{[q]},t)\mathcal{TS}(P_{[q]},t)} \leq \sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \frac{\mathsf{SDwt}(T_i,t) - \mathsf{Dwt}(T_i,t)}{\mathcal{TD}(P_{[q]},t)\mathcal{TS}(P_{[q]},t)} = \sum_{T_i \in P_{[q]}} \gamma(T_i,t).$$

Thus,

$$0 \leq \frac{\mathcal{TS}(P_{[q]},t) - \mathcal{TD}(P_{[q]},t)}{\mathcal{TD}(P_{[q]},t)\mathcal{TS}(P_{[q]},t)} = \sum_{T_i \in P_{[q]}} \gamma(T_i,t) \tag{4.46}$$

Interval type 4: $\mathcal{TS}(P_{[q]},t) = 1$ and $\mathcal{TD}(P_{[q]},t) > 1$. By (4.3), if $\mathcal{TD}(P_{[q]},t) > 1$ holds, then $\mathcal{TD}(P_{[q]},t) = \sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{Dwt}(T_i,t)$ holds. Thus,

$$\frac{\mathcal{TS}(P_{[q]},t) - \mathcal{TD}(P_{[q]},t)}{\mathcal{TD}(P_{[q]},t)\mathcal{TS}(P_{[q]},t)} = \frac{1 - \sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{Dwt}(T_i,t)}{\mathcal{TD}(P_{[q]},t)\mathcal{TS}(P_{[q]},t)}.$$

By (4.6), $\mathcal{TS}(P_{[q]},t) = 1$ implies $\sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{SDwt}(T_i,t) \leq 1$. Thus, since $1 < \mathcal{TS}(P_{[q]},t) = \sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{SDwt}(T_i,t)$, we have

$$0 \geq \frac{1 - \sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \mathsf{Dwt}(T_i,t)}{\mathcal{TD}(P_{[q]},t)\mathcal{TS}(P_{[q]},t)} \geq \sum_{T_i \in \mathsf{ASSN}(P_{[q]},t)} \frac{\mathsf{SDwt}(T_i,t) - \mathsf{Dwt}(T_i,t)}{\mathcal{TD}(P_{[q]},t)\mathcal{TS}(P_{[q]},t)} = \sum_{T_i \in P_{[q]}} \gamma(T_i,t).$$

Thus,

$$0 \geq \frac{\mathcal{T}\mathsf{S}(P_{[q]}, t) - \mathcal{T}\mathsf{D}(P_{[q]}, t)}{\mathcal{T}\mathsf{D}(P_{[q]}, t)\mathcal{T}\mathsf{S}(P_{[q]}, t)} = \sum_{T_i \in P_{[q]}} \gamma(T_i, t). \tag{4.47}$$

Putting it together. From (4.44)–(4.47), it follows that for any interval $[t_k,\ t_{k+1})$,

$$\int_{t_k}^{t_{k+1}} \mathsf{min}\left(0, \frac{\mathcal{T}\mathsf{S}(P_{[q]}, t) - \mathcal{T}\mathsf{D}(P_{[q]}, t)}{\mathcal{T}\mathsf{D}(P_{[q]}, t)\mathcal{T}\mathsf{S}(P_{[q]}, t)}\right) dt \quad \geq \quad \sum_{T_i \in P_{[q]}} \int_{t_k}^{t_{k+1}} \mathsf{min}\left(0, \gamma(T_i, t)\right) dt,$$

and

$$\int_{t_k}^{t_{k+1}} \mathsf{max}\left(0, \frac{\mathcal{T}\mathsf{S}(P_{[q]}, t) - \mathcal{T}\mathsf{D}(P_{[q]}, t)}{\mathcal{T}\mathsf{D}(P_{[q]}, t)\mathcal{T}\mathsf{S}(P_{[q]}, t)}\right) dt \quad \leq \quad \sum_{T_i \in P_{[q]}} \int_{t_k}^{t_{k+1}} \mathsf{max}\left(0, \gamma(T_i, t)\right) dt.$$

Thus, from (4.42),

$$\begin{aligned}\int_{t_a}^{t_b} \mathsf{min}\left(0, \frac{\mathcal{T}\mathsf{S}(P_{[q]}, t) - \mathcal{T}\mathsf{D}(P_{[q]}, t)}{\mathcal{T}\mathsf{D}(P_{[q]}, t)\mathcal{T}\mathsf{S}(P_{[q]}, t)}\right) dt \quad &\geq \quad \sum_{k=\{1,\ldots,z\}} \sum_{T_i \in P_{[q]}} \int_{t_k}^{t_{k+1}} \mathsf{min}\left(0, \gamma(T_i, t)\right) dt, \\ &\geq \quad \sum_{T_i \in P_{[q]}} \int_{t_a}^{t_b} \mathsf{min}\left(0, \gamma(T_i, t)\right) dt, \end{aligned} \tag{4.48}$$

and from (4.43),

$$\begin{aligned}\int_{t_a}^{t_b} \mathsf{max}\left(0, \frac{\mathcal{T}\mathsf{S}(P_{[q]}, t) - \mathcal{T}\mathsf{D}(P_{[q]}, t)}{\mathcal{T}\mathsf{D}(P_{[q]}, t)\mathcal{T}\mathsf{S}(P_{[q]}, t)}\right) dt \quad &\leq \quad \sum_{k=\{1,\ldots,z\}} \sum_{T_i \in P_{[q]}} \int_{t_k}^{t_{k+1}} \mathsf{max}\left(0, \gamma(T_i, t)\right) dt, \\ &\leq \quad \sum_{T_i \in P_{[q]}} \int_{t_a}^{t_b} \mathsf{max}\left(0, \gamma(T_i, t)\right) dt. \end{aligned} \tag{4.49}$$

By Lemma 4.8, we have

$$-\sum_{T_i \in P_{[q]}} \mathcal{C}_i \cdot \mathsf{e}_{\mathsf{max}}(T_i) \leq \sum_{T_i \in P_{[q]}} \int_{t_a}^{t_b} \mathsf{min}\left(0, \gamma(T_i, t)\right) dt,$$

and

$$\sum_{T_i \in P_{[q]}} \mathcal{D}_i \cdot \mathsf{e}_{\mathsf{max}}(T_i) \geq \sum_{T_i \in P_{[q]}} \int_{t_a}^{t_b} \mathsf{max}\left(0, \gamma(T_i, t)\right) dt,$$

where $\mathcal{D}_i$ denotes the number of desired weight decreases for T_i that are both initiated at or before t_b and enacted or canceled after t_a and $\mathcal{C}_i$ denotes the number of desired weight increases via Case (ii) of Rule P for T_i that are both initiated at or before t_b and enacted or canceled after t_a. Notice that $\sum \mathcal{C}_i = \mathcal{C}$, $\sum \mathcal{D}_i = \mathcal{D}$, and for any T_i, $\mathsf{e}_{\mathsf{max}}(T_i) \leq \mathcal{X}$. Thus, from (4.48) and (4.49), we have

$$-\mathcal{C} \cdot \mathcal{X} \leq \int_{t_a}^{t_b} \mathsf{min}\left(0, \frac{1}{\mathcal{T}\mathsf{D}(P_{[q]}, t)} - \frac{1}{\mathcal{T}\mathsf{S}(P_{[q]}, t)}\right) dt,$$

and

$$\mathcal{D} \cdot \mathcal{X} \geq \int_{t_a}^{t_b} \mathsf{max}\left(0, \frac{1}{\mathcal{T}\mathsf{D}(P_{[q]}, t)} - \frac{1}{\mathcal{T}\mathsf{S}(P_{[q]}, t)}\right) dt,$$

which completes the proof. □

4.9.3 Calculating Drift

Having established Corollary 4.1 and Lemma 4.9 we can now calculate the total drift incurred. Before continuing, we introduce some terminology to facilitate our discussion. Assume that T_i is assigned to the processor $P_{[q]}$ over the range $[t_a, t_b)$.

- Let $\mathcal{P}_1$ denote the number of weight increases via Case (i) of Rule P by T_i that were initiated at or before t_b and enacted or canceled after t_a.

- Let $\mathcal{P}_2$ denote the number of weight increases via Case (ii) of Rule P by T_i that were initiated at or before t_b and enacted or canceled after t_a.

- Let $\mathcal{G}$ denote the number of weight decreases by T_i that were initiated at or before t_b and enacted or canceled after t_a.

- Let $\mathcal{C}$ denote the number of weight increases via Case (ii) or Rule P by any task assigned to $P_{[q]}$ that were initiated at or before t_b and enacted or canceled after t_a.

- Let $\mathcal{D}$ denote the number of weight decreases by any task assigned to $P_{[q]}$ that were initiated at or before t_b and enacted or canceled after t_a.

- Let $\mathcal{X}$ denote the maximal execution time of any task assigned to $P_{[q]}$ over the range $[t_a, t_b)$.

By Corollary 4.1, the partial drift incurred by the task T_i over the range $[t_a,\, t_b)$ is bounded by

$$-\left(\mathcal{G}+\mathcal{P}_1\right)\mathsf{e}_{\mathsf{max}}(T_i) \le \mathsf{Pdrift}(T_i,t_b) - \mathsf{Pdrift}(T_i,t_a) \le \left(\mathcal{P}_1+\mathcal{P}_2\right)\mathsf{e}_{\mathsf{max}}(T_i), \tag{4.50}$$

assuming that t_b and T_i satisfy one of Conditions (T-1), ..., (T-3).

In addition, since T_i's allocation over the range $[t_a,\, t_b)$ is given by $\int_{t_a}^{t_b} \frac{\mathsf{Dwt}(T_i,t)}{\mathcal{T}\mathsf{D}(P_{[q]},t)}dt$ in the IDEAL schedule (by (4.16)), and by $\int_{t_a}^{t_b} \frac{\mathsf{Dwt}(T_i,t)}{\mathcal{T}\mathsf{S}(P_{[q]},t)}dt$ in the PT schedule (by (4.17)), the difference in T_i's allocation in the IDEAL and PT schedules over the range $[t_a,\, t_b)$ is given by

$$\int_{t_a}^{t_b} \mathsf{Dwt}(T_i,t)\left(\frac{1}{\mathcal{T}\mathsf{D}(P_{[q]},t)} - \frac{1}{\mathcal{T}\mathsf{S}(P_{[q]},t)}\right)dt.$$

Since $0 \le \mathsf{Dwt}(T_i,t) \le 1$, and $\mathsf{min}(0, \frac{1}{\mathcal{T}\mathsf{D}(P_{[q]},t)} - \frac{1}{\mathcal{T}\mathsf{S}(P_{[q]},t)}) \le \frac{1}{\mathcal{T}\mathsf{D}(P_{[q]},t)} - \frac{1}{\mathcal{T}\mathsf{S}(P_{[q]},t)} \le \mathsf{max}(0, \frac{1}{\mathcal{T}\mathsf{D}(P_{[q]},t)} - \frac{1}{\mathcal{T}\mathsf{S}(P_{[q]},t)})$, it follows that $\int_{t_a}^{t_b} \mathsf{min}\left(0, \frac{1}{\mathcal{T}\mathsf{D}(P_{[q]},t)} - \frac{1}{\mathcal{T}\mathsf{S}(P_{[q]},t)}\right)dt \le \int_{t_a}^{t_b} \mathsf{Dwt}(T_i,t)\left(\frac{1}{\mathcal{T}\mathsf{D}(P_{[q]},t)} - \frac{1}{\mathcal{T}\mathsf{S}(P_{[q]},t)}\right)dt \le \int_{t_a}^{t_b} \mathsf{max}\left(0, \frac{1}{\mathcal{T}\mathsf{D}(P_{[q]},t)} - \frac{1}{\mathcal{T}\mathsf{S}(P_{[q]},t)}\right)dt$. Thus, by Lemma 4.9,

$$-\mathcal{C}\cdot\mathcal{X} \le \int_{t_a}^{t_b} \mathsf{Dwt}(T_i,t)\left(\frac{1}{\mathcal{T}\mathsf{D}(P_{[q]},t)} - \frac{1}{\mathcal{T}\mathsf{S}(P_{[q]},t)}\right)dt \le \mathcal{D}\cdot\mathcal{X}. \tag{4.51}$$

The total drift incurred by T_i within $[t_a,\, t_b)$ is given by $\mathsf{Pdrift}(T_i,t_b) - \mathsf{Pdrift}(T_i,t_a) + \int_{t_a}^{t_b} \mathsf{Dwt}(T_i,t)\left(\frac{1}{\mathcal{T}\mathsf{D}(P_{[q]},t)} - \frac{1}{\mathcal{T}\mathsf{S}(P_{[q]},t)}\right)dt$. By (4.50) and (4.51), this incurred drift is within the range

$$\left[-\left(\mathcal{G}+\mathcal{P}_1\right)\cdot\mathsf{e}_{\mathsf{max}}(T_i) - \mathcal{C}\cdot\mathcal{X},\;\; \left(\mathcal{P}_1+\mathcal{P}_2\right)\cdot\mathsf{e}_{\mathsf{max}}(T_i) + \mathcal{D}\cdot\mathcal{X}\right].$$

Thus, we have the following lemma.

Lemma 4.10. *Let t_a and t_b be any two times such that $t_a < t_b$, and the system is not reset over the interval (t_a, t_b). Let $\mathcal{Q}$ denote the number of desired weight changes for any task on $P_{[q]}$ that are both initiated at or before t_b and enacted or canceled after t_a, and let T_i be assigned to $P_{[q]}$ over the range $[t_a,\, t_b)$. If T_i and t_b satisfy one of Conditions (T-1), ...,(T-3), then the absolute drift incurred by T_i within $[t_a,\, t_b)$ is at most $\mathcal{Q}\cdot\mathcal{X}$, where $\mathcal{X}$ is the largest execution time of any task assigned to $P_{[q]}$ over the range $[t_a,\, t_b)$.*

4.9.4 Incorporating Resets

Having established Lemma 4.10, we now determine the additional drift incurred by resetting the system.

Lemma 4.11. *If the system is reset at time t_c, then the drift incurred by any task T_i is within the range* $[-\mathsf{e}_{\mathsf{max}}(T_i), \mathsf{e}_{\mathsf{max}}(T_i)]$.

Proof. Let T_i^j be the last-released job (if any) of T_i before t_c. Notice that, if T_i^j is not active at t_c or T_i^j does not exist, then via Case (ii) of Rule R, T_i is simply assigned to a (possibly different) processor and T_i's next job is released at either $t_c + \theta(T_i^{j+1})$ or $t_c + \theta(T_i^1)$, depending on whether T_i^j exists. Since the only two potential sources of drift are delays in enacting a desired weight change and halting a job, it follows that no partial drift is incurred in this case. Thus, for the rest of this proof, we assume that T_i^j is active at t_c. Thus,

$$t_c \in [\mathsf{r}(T_i^j),\, \mathsf{min}(\mathsf{r}(T_i^{j+1}), \mathsf{d}(T_i^j))). \tag{4.52}$$

Since T_i^j is active at t_c, it is reset via Case (i) of Rule R. In this case, T_i releases a job immediately with execution time $\mathsf{nextE}(T_i^j, t_c)$. Moreover, its current job T_i^j is halted, so it is as though allocation equal to $\mathsf{A}(\mathcal{P}T,\, T_i,\, \mathsf{r}(T_i^j),\, t_c) - \mathsf{A}(\mathcal{C}SW,\, T_i,\, \mathsf{r}(T_i^j),\, t_c)$ is "lost." Notice that, since, T_i^j becomes inactive at t_c (because it is halted), by Lemma 4.3

$$\mathsf{A}(\mathcal{C}SW,\, T_i,\, \mathsf{r}(T_i^j),\, t_c) \le \mathsf{e}_{\mathsf{max}}(T_i). \tag{4.53}$$

Also, notice that, if T_i does not initiate any desired weight increase over the range $[\mathsf{r}(T_i^j),\, t_c)$ that is not immediately enacted, then for all $t \in [\mathsf{r}(T_i^j),\, t_c)$, $\mathsf{A}(\mathcal{P}T,\, T_i,\, \mathsf{r}(T_i^j),\, t) \le \int_{\mathsf{r}(T_i^j)}^{t} \mathsf{SGwt}(T_i, t)dt$. Thus, in this case, by Property (D) and (4.52),

$$\mathsf{A}(\mathcal{P}T,\, T_i,\, \mathsf{r}(T_i^j),\, t) \le \mathsf{e}(T_i^j). \tag{4.54}$$

Thus, if T_i does not initiate any desired weight increase over the range $[\mathsf{r}(T_i^j),\, t_c)$ that is not immediately enacted, then by (4.53) and (4.54),

$$-\mathsf{e}_{\mathsf{max}}(T_i) \leq \mathsf{A}(\mathcal{P}T,\, T_i,\, \mathsf{r}(T_i^j),\, t_c) - \mathsf{A}(\mathcal{C}SW,\, T_i,\, \mathsf{r}(T_i^j),\, t_c) \leq \mathsf{e}_{\mathsf{max}}(T_i). \tag{4.55}$$

In addition, if T_i initiates a desired weight increase over the range $[\mathsf{r}(T_i^j),\, t_c)$ that is not immediately enacted, then this change must have been via Case (ii) of Rule P. Notice that, if such a change occurs, then it is possible that the upper bound in (4.55) may not hold, i.e., it is possible that $\mathsf{A}(\mathcal{P}T,\, T_i,\, \mathsf{r}(T_i^j),\, t_c) - \mathsf{A}(\mathcal{C}SW,\, T_i,\, \mathsf{r}(T_i^j),\, t_c) = \mathsf{e}_{\mathsf{max}}(T_i) + X$, where $X > 0$. However, if the upper bound in (4.55) is violated by X, then it must have been the case that T_i incurred at least X units of drift from reweighting events initiated over the range $[\mathsf{r}(T_i^j),\, t_c)$. Moreover, this drift would have been accounted for by Lemma 4.4. Thus, if $\mathsf{A}(\mathcal{P}T,\, T_i,\, \mathsf{r}(T_i^j),\, t_c) - \mathsf{A}(\mathcal{C}SW,\, T_i,\, \mathsf{r}(T_i^j),\, t_c) = \mathsf{e}_{\mathsf{max}}(T_i) + X$ holds, then the maximal amount of additional drift incurred by resetting T_i is $\mathsf{A}(\mathcal{P}T,\, T_i,\, \mathsf{r}(T_i^j),\, t_c) - \mathsf{A}(\mathcal{C}SW,\, T_i,\, \mathsf{r}(T_i^j),\, t_c) - X = \mathsf{e}_{\mathsf{max}}(T_i)$.

Thus, it follows that T_i's drift *due to the reset* is bounded within $[-\mathsf{e}_{\mathsf{max}}(T_i), \mathsf{e}_{\mathsf{max}}(T_i)]$. $\square$

4.9.5 Total Drift Incurred

From Lemmas 4.10 and 4.11, we have the following.

Theorem 4.4. *For any task T_i and for any interval of time $[t_a,\, t_b)$, let $\mathcal{Q}$ denote the number of system resets plus the number of desired weight changes for any task assigned to the same processor as T_i that are both initiated at or before t_b and enacted or canceled after t_a. If T_i and t_b satisfy one of Conditions (T-1), ...,(T-3), then the absolute drift incurred by T_i is at most $\mathcal{Q} \cdot \mathcal{X}$, where $\mathcal{X}$ is the largest execution time of any task in the system.*

Moreover, if a task is never assigned to an over-utilized processor, then the partial drift of a task always equals its drift. As a result, we can tighten Theorem 4.4.

Theorem 4.5. *For any task T_i that is never assigned to an over-utilized processor and any interval of time $[t_b,\, t_a)$, let $\mathcal{Q}$ denote the number of system resets plus the number of desired weight changes by T_i that are both initiated at or before t_b and enacted or canceled after t_a.*

If T_i and t_b satisfy one of Conditions (T-1), ...,(T-3), then the absolute drift incurred by T_i is at most $\mathcal{Q} \cdot \mathsf{e}_{\mathsf{max}}(T_i)$.

4.9.6 Modifications for NP-PEDF

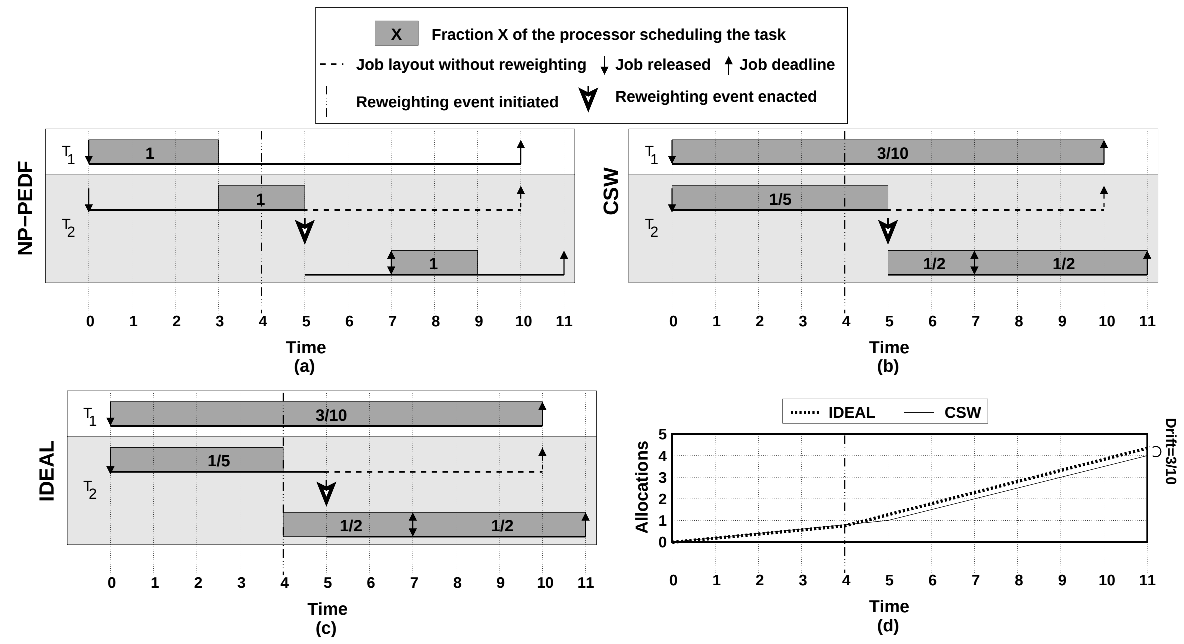
Note that delaying the initiation of a reweighting event due to non-preemptivity does not substantially increase the drift incurred per reweighting event, since the longest a reweighting event can be delayed is the execution time of the active job of the task being reweighted.

Suppose that task T_i initiates a weight change at time t_c. If T_i^j is active at t_c, and if T_i's reweighting event is delayed until some time t (by a non-preemptive section), then at t either **(a)** T_i^j has a non-positive deviance (i.e., T_i^j completes before its deadline), or **(b)** t is the first time that T_i^j becomes inactive (i.e., $t = \mathsf{min}(\mathsf{r}(T_i^{j+1}), \mathsf{d}(T_i^j))$.)

If Case (a) occurs, then T_i is negative-changeable at t, and T_i^j is active at t. Hence, if T_i increases its weight, then the only drift T_i will incur for this reweighting event results from delaying the initiation of the event, i.e., at most $\mathsf{e}_{\mathsf{max}}(T_i)$. If T_i decreases its weight, then delaying the reweighting event will not affect partial drift, since the enactment of the reweighting event would occur when T_i^j becomes inactive, regardless of whether the initiation of the reweighting event was delayed or not.

Example (Figure 4.22). Consider the example in Figure 4.22, which depicts a one-processor system scheduled by NP-PEDF with two tasks: T_1, which has $\mathsf{wt}(T_1) = 3/10$ and $\mathsf{e}(T_1) = 3$; and T_2, which has $\mathsf{e}(T_2) = 2$ and an initial weight of $1/5$ and initiates a weight increase to $1/2$ at time 4. Inset (a) depicts the NP-PEDF schedule. Inset (b) depicts the CSW schedule. Inset (c) depicts the IDEAL schedule. Inset (d) depicts T_2's allocations in the CSW and IDEAL schedules. Notice that T_2's weight change is delayed from time 4 to time 5 because T_2 is non-preemptively executing at time 4. As a result, T_2 is negative-changeable at time 5. Also note that, T_2^2 is released when T_2's actual allocation equals its allocation in the CSW schedule at time 7, i.e., when T_2^2's deviance equals zero. $\square$

If Case (b), mentioned earlier, occurs, then either no job of T_i is active at t or T_i^{j+1} is active at t. If no job of T_i is active at t, then the change is enacted immediately, and the partial drift that the task incurs from the reweighting event is a result of delaying the initiation of



X Fraction X of the processor scheduling the task
Job layout without reweighting
Job released
Job deadline
Reweighting event initiated
Reweighting event enacted
NP−PEDF
CSW
IDEAL
T1
T2
1
3/10
1/5
1/2
Time
(a)
(b)
(c)
(d)
IDEAL
CSW
Allocations
Drift=3/10

Figure 4.22: A one-processor example of drift in NP-PEDF, where T_2^1 completes before its deadline. **(a)** The NP-PEDF schedule. **(b)** The CSW schedule. **(c)** The IDEAL schedule. **(d)** T_2's allocations in the CSW and IDEAL schedules.

the event, i.e., $\mathsf{e}_{\mathsf{max}}(T_i)$. If T_i^{j+1} is active at t, then since $t = \min(\mathsf{r}(T_i^{j+1}), \mathsf{d}(T_i^j))$, it must be the case that $\mathsf{r}(T_i^{j+1}) = t$. As a result, the weight change is enacted immediately and T_i^{j+1} is released with the new weight. Hence, the only partial drift that is incurred is as a result of delaying the initiation of the reweighting event, i.e., at most $\mathsf{e}_{\mathsf{max}}(T_i)$.

Example (Figure 4.23). Consider the example in Figure 4.23, which depicts a partial NP-PEDF schedule for a task T_i, which has an initial weight of 1/10 that increases to 1/2 at time t_c while the last-released job of T_i before t_c, T_i^j, is both active and being scheduled. Note that T_i^j has an execution time of four, and all jobs released after T_i^j have an execution time of one. Moreover, T_i^j does not complete execution until after its deadline. Inset (a) depicts the NP-PEDF schedule. Inset (b) depicts the CSW schedule. Inset (c) depicts the IDEAL schedule. Inset (d) depicts T_i's allocation in the CSW and IDEAL schedules. Because T_i^j is not complete by its deadline, the initiation of the weight change is delayed until $t = \mathsf{d}(T_i^j) = \mathsf{r}(T_i^{j+1})$. Recall that, if a weight change is initiated when $\mathsf{d}(T_i^j) = \mathsf{r}(T_i^{j+1})$, then the weight change is immediately enacted and T_i^{j+1} is released with the new weight (even though T_i^j has not yet completed execution). Thus, the only source of partial drift is because the initiation of the

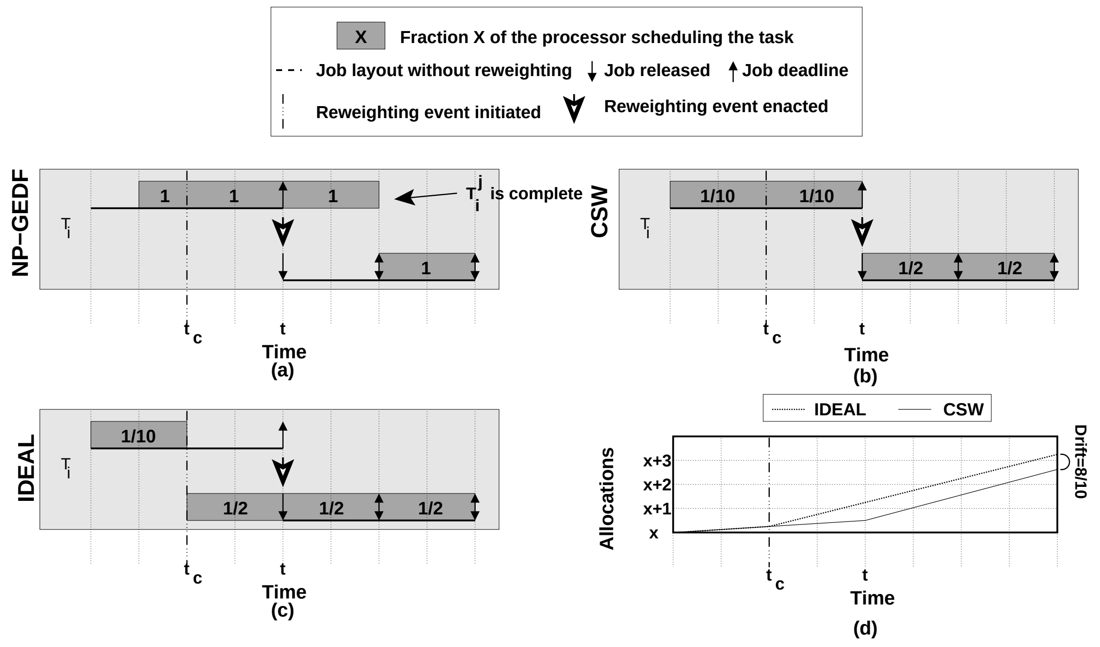


Figure 4.23: A partial schedule of a one-processor example of drift in NP-PEDF. T_i^j completes after its deadline. **(a)** The NP-PEDF schedule. **(b)** The CSW schedule. **(c)** The IDEAL schedule. **(d)** T_i's allocations in the CSW and IDEAL schedules.

reweighting event is delayed. □

In addition, similar reasoning holds for describing the impact of non-preemptability on the difference between a task's allocation in the IDEAL and PT schedules. From this reasoning, the following theorem holds.

Theorem 4.6. *In a* NP-PEDF *scheduled system, for any task T_i, and for any interval of time $[t_a, t_b)$, let $\mathcal{Q}$ denote the number of system resets plus the number of desired weight changes for any task assigned to the same processor as T_i that are both initiated at or before t_b and enacted or canceled after t_a. If T_i and t_b satisfy one of Conditions (T-1), ...,(T-3), then the absolute drift incurred by T_i is at most $\mathcal{Q} \cdot \mathcal{X}$, where $\mathcal{X}$ is the largest execution time of any task assigned to $P_{[q]}$ over this range.*

It is important to note that Theorem 4.6 still holds even if a task T_i is not immediately migrated from $P_{[q]}$ to $P_{[k]}$ at a system reset because its last-released job T_i^j was executing when the system was reset. The reason why is because T_i's allocations in the IDEAL schedule are based on its assigned processor. Thus, T_i does not incur any additional drift by remaining on its old processor. Similarly, no other task on $P_{[q]}$ or $P_{[k]}$ incurs additional drift by T_i remaining assigned to $P_{[q]}$ while T_i^j completes its execution. It is possible that if T_i remains on $P_{[q]}$ while T_i^j completes execution, then tasks on $P_{[q]}$ will receive a guaranteed weight that is less than their desired weight; however, this difference is already captured by the MROE metric and it is not considered to contribute to drift.

Again, if a task is never assigned to an over-utilized processor, then the partial drift of a task always equals its drift. As a result, we can tighten Theorem 4.6.

Theorem 4.7. *In a* NP-PEDF *scheduled system, for any task T_i that is never assigned to an over-utilized processor and for any interval of time $[t_a, t_b)$, let $\mathcal{Q}$ denote the number of system resets plus the number of desired weight changes by T_i that are both initiated at or before t_b and enacted or canceled after t_a. If T_i and t_b satisfy one of Conditions (T-1), ...,(T-3), then the absolute drift incurred by T_i is at most $\mathcal{Q} \cdot \mathsf{e}_{\mathsf{max}}(T_i)$.*

4.10 Adjusting PEDF for Use with any Metric

In order to allow PEDF to determine the guaranteed weights of tasks via any non-MROE metric, we must make some small changes to our adapative PEDF algorithm. Before continuing, notice that the following property holds for the MROE metric.

QS (queue stability): At any reweighting event on a processor $P_{[q]}$, the guaranteed weight of each non-reweighting task assigned to $P_{[q]}$ changes by the same multiple: $\frac{old}{new}$, where *old* (*new*) is the total desired weight of all tasks assigned to $P_{[q]}$ immediately before (after) the reweighting event.

Recall that Rules P and N function by changing the future releases and deadlines of a reweighted task. Any currently-queued job of such a task must be reinserted into the scheduler's priority queue. Since QS guarantees that the guaranteed weight of all non-reweighted tasks change by the same multiple, the jobs of these tasks already appear in the queue in the correct order, so they do not have to be reinserted into the priority queue via a rule like Rule P or N. (Moreover, if the concept of *virtual time* is introduced, then the deadlines of such jobs do not have to be recomputed (Stoica et al., 1996).) In fact, the primary purpose of Rules

P and N is to remove jobs from the scheduler's priority queues as needed and reinsert them into their proper places.

Example. Suppose that four tasks T_1, T_2, T_3, and T_4 with desired weights 0.5, 0.2, 0.2, and 0.2, respectively, are assigned to a processor, and at some time, T_4 changes its desired weight to 0.3. Under the MROE metric, T_4's weight change causes T_1's guaranteed weight to change from $0.5/1.1$ to $0.5/1.2$, and the guaranteed weight of both T_2 and T_3 to change from $0.2/1.1$ to $0.2/1.2$. Thus, the guaranteed weights of T_1, T_2, and T_3 all change by the same factor, $1.1/1.2$ (the old total desired weight divided by the new total desired weight). $\square$

Under metrics that are not equivalent to MROE, *QS does not hold.* Consider the same example as above except that the AROE metric is used. Then, T_1's guaranteed weight changes from $0.5 - (1.1 - 1) = 0.4$ to $0.5 - (1.2 - 1) = 0.3$, while the guaranteed weights of both T_2 and T_3 remain at 0.2. Thus, T_1 and T_2 (as well as T_1 and T_3) change guaranteed weights by a different multiple. As a result, T_1 (or both T_2 and T_3) must change its (their) guaranteed weight by a rule (i.e., Rule P or N) that may remove and reinsert currently-queued jobs into the scheduler's priority queue.

Since, from a reweighting perspective, the primary difference between the MROE metrics and non-MROE metrics is the Property QS, and as we mentioned above, the primary objective of Rules P and N is to remove a job from the scheduler's priority queues and reinsert it into its proper place, is possible to adapt Rules P and N to work for non-MROE metrics by using them whenever a task changes its *guaranteed weight* (recall that under the MROE metric the Rules P and N are only used when a task changes its *desired weight*). Thus, in the above example under the AROE metric, when T_4 changes its desired weight both T_1 and T_4 must uses the modified Rules P and N to possibly remove jobs the scheduler's priority queues and reinsert them into their proper places. Under the MROE metric, only T_4 had to use Rules P and N.

4.11 Time Complexity

As noted earlier, the time complexity for PEDF to partition N tasks onto M processors is $O(M + N \log N)$. If we were to implement PEDF using binomial heaps, then the time complexity to make a scheduling decision on a processor $P_{[q]}$ is $O(\log n)$, where n is the number of tasks assigned to $P_{[q]}$. Recall that when a task changes its desired weight using either Rule P or N, one of its jobs may be removed from its processor's priority queue and reinserted. Thus, $O(\log n)$ time is required to change a task's desired weight via Rule P or N using the MROE metric. Under non-MROE metrics, $O(n \log n)$ time is required, due to the potential need to re-enqueue jobs of non-reweighted tasks. Hence, the MROE metric has a clear advantage over the non-MROE metrics.

4.12 Conclusion

In this chapter, we presented several different methods for calculating the error on an over-utilized processor, presented a variant of the adaptable sporadic task model that allows the guaranteed and desired weights of a task to differ, and presented rules for reweighting a task under the PEDF and NP-PEDF scheduling algorithms. In addition, we proved scheduling correctness and established tardiness and drift bounds for our reweighting rules.

We conclude this chapter with two important remarks. First, it is worth reiterating that even though PEDF does not miss a deadline by our reweighting rules, if a task T_i is assigned to an over-utilized processor, then there could possibly be an *arbitrarily large* difference between T_i's allocations in the actual schedule and a schedule in which it receives a fraction of the system equal to its *desired weight* at each instant. This differs from GEDF-scheduled systems, in which the difference between T_i's allocations in the actual system and a system in which it receives its desired weight at each instant is bounded (if we defined job deadlines based on desired weights, then PEDF could have unbounded tardiness). Second, we do *not* claim that our adaptive variant of PEDF is the final word regarding partitioned reweighting schemes. However, we have tried hard to devise reasonable approaches for dealing with the fundamental limitation discussed earlier in Section 4.2 to which such schemes are subject. Thus, we believe that our adaptive variant of PEDF is a good candidate partitioning approach.

CHAPTER 5

PD^{2*}

In this chapter, we examine the issue of reweighting in Pfair-scheduled systems. Before continuing, we first review the basics of Pfair scheduling.

5.1 Preliminaries

As was mentioned in Section 1.2.4, in Pfair-scheduled systems, processor time is allocated in discrete time units, called *quanta*. The time interval $[t, t+1)$, where t is a nonnegative integer, is called *slot* t. (Hence, time t refers to the beginning of slot t.) In this chapter, all time values are assumed to indicate an integral number of quanta, unless specified otherwise.

Recall, from Section 1.2, that the function $\mathsf{A}(\mathcal{S}, T_i, t_1, t_2)$ denotes the allocations to the task T_i in the schedule $\mathcal{S}$ over the range $[t_1, t_2)$. Similarly, we use $\mathsf{A}(\mathcal{S}, T_i^{[j]}, t_1, t_2)$ and $\mathsf{A}(\mathcal{S}, \tau, t_1, t_2)$ to denote, respectively, the total allocations to the "subtask" $T_i^{[j]}$ (as defined below) and to all tasks in the set τ over the range $[t_1, t_2)$. As a shorthand, we denote $\mathsf{A}(\mathcal{S}, T_i, t, t+1)$ as $\mathsf{A}(\mathcal{S}, T_i, t)$. Let $\mathcal{S}$ be the Pfair schedule of the system τ; if $\mathsf{A}(\mathcal{S}, T_i^{[j]}, t) = 1$, then we say that $T_i^{[j]}$ *is scheduled in slot* t. For reference, all terms used in this chapter are listed in Table 5.1.

5.1.1 Periodic Pfair Scheduling

In defining notions relevant to Pfair scheduling, we limit attention (for now) to periodic tasks, all of which begin execution at time 0, where each task's relative deadline equals its period. A periodic task T_i with an integer *period* $\mathsf{p}(T_i)$ and an integer *execution time* $\mathsf{e}(T_i)$ has a *weight*

[*] Contents of this chapter previously appeared in preliminary form in the following paper:
Block, A., Anderson J., and Bishop, G. (2008a). Fine-Grained task reweighting on multiprocessors. *Journal of Embedded Computing*, (to appear).

Notation	**Definition**
$T_i^{[j]}$	The j^{th} subtask of T_i.
$\mathsf{wt}(T_i, t)$	Weight of T_i at time t.
$\mathsf{wt}(T_i)$	Weight of a task T_i that does not change its weight.
$\mathsf{Swt}(T_i, t)$	Scheduling weight of T_i at time t.
$\mathsf{r}(T_i^{[j]})$	Release time of T_i^j.
$\mathsf{d}(T_i^{[j]})$	Deadline of T_i^j.
$\mathsf{w}(T_i^{[j]})$	Window of $T_i^{[j]}$, i.e., $[\mathsf{r}(T_i^{[j]}), \mathsf{d}(T_i^{[j]}))$.
$\mathsf{b}(T_i^{[j]})$	b-bit of T_i^j.
$\mathsf{C}(\mathcal{B}, T_i^{[j]})$	Time $T_i^{[j]}$ completes in the schedule $\mathcal{B}$.
$\mathsf{D}(T_i^{[j]})$	Group deadline of T_i^j.
$\mathsf{En}(T_i, t)$	Last time at or before t that T_i was reset.
$\mathsf{Id}(T_i^{[j]})$	The index of the first subtask of T_i after $\mathsf{En}(T_i, t)$.
$\omega(T_i^{[j]})$	Subtask associated with $\mathsf{D}(T_i^{[j]})$.
$\theta(T_i^{[j]})$	IS separation between $T_i^{[j-1]}$ and $T_i^{[j]}$.
SW	Non-clairvoyant scheduling-weight scheduling algorithm. While a task is active, this algorithm allocates the task its *scheduling weight* at each instant.
$\mathcal{SW}$	SW schedule of a task system τ.
CSW	Clairvoyant scheduling-weight scheduling algorithm. Only allocates capacity to non-halted subtasks.
$\mathcal{CSW}$	CSW schedule of a task system τ.
IDEAL	Ideal scheduling algorithm. While a task is active, this algorithm allocates a task its *weight* at each instant.
$\mathcal{I}$	IDEAL schedule of a task system τ.
$\mathcal{S}$	Actual schedule (i.e., PD^2 schedule) of task system τ.
$\mathsf{A}(\mathcal{B}, T_i^{[j]}, t_1, t_2)$	Allocation to $T_i^{[j]}$ in the schedule $\mathcal{B}$ over $[t_1, t_2)$.
$\mathsf{A}(\mathcal{B}, T_i, t_1, t_2)$	Allocation to T_i in the schedule $\mathcal{B}$ over $[t_1, t_2)$.
$\mathsf{drift}(T_i, t)$	Drift of T_i: $\mathsf{A}(\mathcal{I}, T_i, 0, t) - \mathsf{A}(\mathcal{CSW}, T_i, 0, t)$.
Ow	Scheduling weight before a reweighting event.
Nw	New weight after a reweighting event.
$\mathsf{lag}(\mathcal{S}, \mathcal{I}, T_i, t)$	Lag of T_i at time t: $\mathsf{A}(\mathcal{I}, T_i, 0, t) - \mathsf{A}(\mathcal{S}, T_i, 0, t)$.
$\mathsf{lag}(T_i, t)$	Lag of T_i at time t when the actual and ideal schedules are implicit.
$\mathsf{LAG}(\mathcal{S}, \mathcal{I}, \tau, t)$	Total lag of all tasks in τ at time t.
$\mathsf{LAG}(\tau, t)$	Total lag of all tasks in τ at time t when the actual and ideal schedules are implicit.
$\mathsf{X}^{(j)}$	j^{th} subtask in a chain of displacements.
$\langle \mathsf{X}^{(j)}, t_j, \mathsf{X}^{(j+1)}, t_{j+1} \rangle$	Displacement tuple.

Table 5.1: Brief description of the notation used in this chapter.

(or *utilization*) $\mathsf{wt}(T_i) = \mathsf{e}(T_i)/\mathsf{p}(T_i)$, where $0 < \mathsf{wt}(T_i) < 1$. We say that a task is *light* if its weight is in the range $(0, 1/2)$, and *heavy* if its weight is in the range $[1/2, 1)$. (For simplicity, we ignore the possibility of a task having a weight of 1. Such tasks can be included, but at the expense of more complicated notation in the reweighting rules.)

The *ideal schedule* for a periodic task system allocates $\mathsf{wt}(T_i)$ processing time to each task in each time slot. More specifically, in the ideal schedule, $\mathcal{E}$, of the task system τ, $\mathsf{A}(\mathcal{S},\, T_i,\, t) = \mathsf{wt}(T_i)$ holds for any $T_i \in \tau$ and any time $t \geq 0$.

In order to compare the difference in the task T_i's allocations in the actual schedule $\mathcal{S}$ and the ideal schedule $\mathcal{E}$ up to time t, we use the function $\mathsf{lag}(\mathcal{S}, \mathcal{E}, T_i, t) = \mathsf{A}(\mathcal{E},\, T_i,\, 0,\, t) - \mathsf{A}(\mathcal{S},\, T_i,\, 0,\, t)$. Additionally, we use the function $\mathsf{LAG}(\mathcal{S}, \mathcal{E}, \tau, t) = \sum_{T_i \in \tau} \mathsf{lag}(\mathcal{S}, \mathcal{E}, T_i, t)$ to compare the differences in allocations for *all tasks* in the task set τ in schedules $\mathcal{S}$ and $\mathcal{E}$. We assume $\mathsf{lag}(\mathcal{S}, \mathcal{E}, T_i, 0) = 0$. Thus, $\mathsf{LAG}(\mathcal{S}, \mathcal{E}, \tau, t)$ can be rewritten as

$$\mathsf{LAG}(\mathcal{S}, \mathcal{E}, \tau, t) = \mathsf{LAG}(\mathcal{S}, \mathcal{E}, \tau, t-1) + \mathsf{A}(\mathcal{E},\, \tau,\, t-1) - \mathsf{A}(\mathcal{S},\, \tau,\, t-1). \qquad (5.1)$$

For brevity, we denote $\mathsf{lag}(\mathcal{S}, \mathcal{E}, T_i, t)$ as $\mathsf{lag}(T_i, t)$ and $\mathsf{LAG}(\mathcal{S}, \mathcal{E}, \tau, t)$ as $\mathsf{LAG}(\tau, t)$, when $\mathcal{S}$ and $\mathcal{E}$ are well-defined and obvious. (Later, we apply (5.1) in contexts where the ideal schedule is defined to reflect changes caused by reweighting events.)

The schedule $\mathcal{S}$ is *Pfair* iff $(\forall T_i \in \tau, t :: -1 < \mathsf{lag}(T_i, t) < 1)$. Informally, each task's allocation error must always be less than one quantum. These error bounds are ensured by treating each quantum of a task's execution, henceforth called a *subtask*, as a schedulable entity. Scheduling decisions are made only at quantum boundaries. The j^{th} subtask of task T_i, denoted $T_i^{[j]}$, where $j \geq 1$, has an associated *pseudo-release*

$$\mathsf{r}(T_i^{[j]}) = \left\lfloor \frac{j-1}{\mathsf{wt}(T_i)} \right\rfloor$$

and *pseudo-deadline*

$$\mathsf{d}(T_i^{[j]}) = \left\lceil \frac{j}{\mathsf{wt}(T_i)} \right\rceil.$$

(For brevity, we often drop the prefix "pseudo-.") It can be shown that if each subtask $T_i^{[j]}$

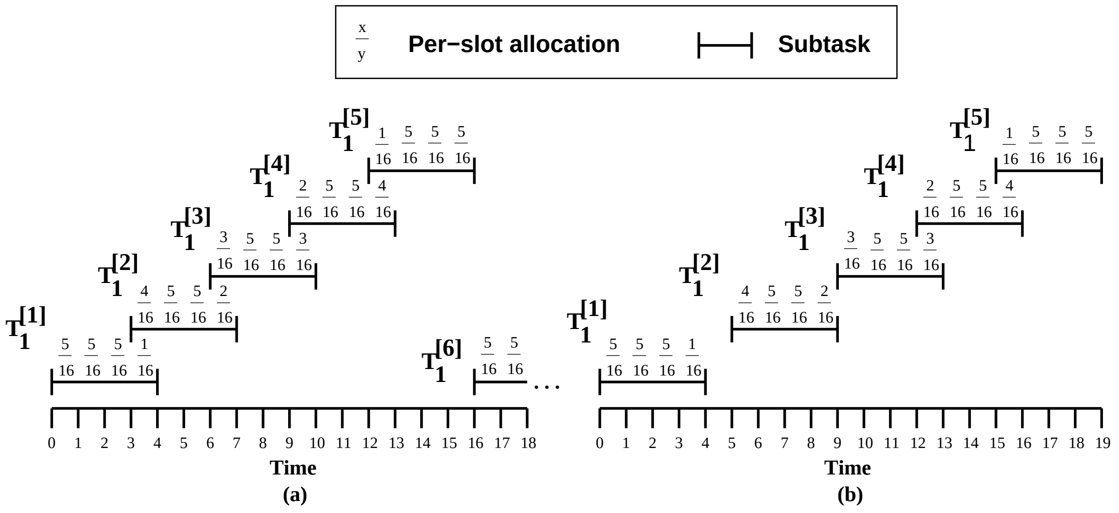


Figure 5.1: $\mathsf{A}(\mathcal{I}, T_i^{[j]}, t)$ for a (a) periodic and (b) IS task T_1 of weight 5/16.

is scheduled in the interval $\mathsf{w}(T_i^{[j]}) = [\mathsf{r}(T_i^{[j]}), \mathsf{d}(T_i^{[j]}))$, termed its *window*, then $(\forall T_i \in \tau, t :: -1 < \mathsf{lag}(T_i, t) < 1)$ is maintained (Baruah et al., 1996).

Example (Figure 5.1). Consider the example in Figure 5.1, which depicts the releases and deadlines for a task T_1, which has $\mathsf{wt}(T_1) = 5/16$. (This figure also depicts per-slot ideal allocations for each subtask, which are considered below.) In this example, $\mathsf{r}(T_1^{[2]}) = 3$, $\mathsf{d}(T_1^{[2]}) = 7$, and $\mathsf{w}(T_1^{[2]}) = [3, 7)$. Thus, $T_1^{[2]}$ must be scheduled in slots 3–6. (Tasks execute sequentially, so if $T_1^{[1]}$ is scheduled in slot 3, then $T_1^{[2]}$ must be scheduled in slots 4–6.) $\square$

5.1.2 The Intra-Sporadic Task Model

The *intra-sporadic* (IS) *task model* (Srinivasan and Anderson, 2006) generalizes the well-known sporadic task model (Mok, 1983) by allowing subtasks to be released late. This extra flexibility is useful in many applications where processing steps may be delayed. Fig. 5.1(b) illustrates the Pfair windows of an IS task of weight 5/16 in which the release of $T_1^{[2]}$ is delayed by two quanta and the release of $T_1^{[3]}$ is delayed by an additional quantum. Each subtask $T_i^{[j]}$ of an IS task has an *offset*, $\theta(T_i^{[j]})$, that gives the amount by which its release has been delayed. For example, in Figure 5.1(b), $\theta(T_1^{[1]}) = 0$, $\theta(T_1^{[2]}) = 2$, and for $j \geq 3$, $\theta(T_1^{[j]}) = 3$.

The release and deadline of a subtask $T_i^{[j]}$ of an IS task T_i are defined as

$$\begin{aligned}
\mathsf{r}(T_i^{[1]}) &= \theta(T_i^{[1]}) && (5.2)\\
\mathsf{r}(T_i^{[j+1]}) &= (\theta(T_i^{[j+1]}) - \theta(T_i^{[j+1]})) + \mathsf{d}(T_i^{[j]}) - \left\lceil \frac{j}{\mathsf{wt}(T_i)} \right\rceil + \left\lfloor \frac{j}{\mathsf{wt}(T_i)} \right\rfloor && (5.3)\\
\mathsf{d}(T_i^{[j]}) &= = \mathsf{r}(T_i^{[j]}) + \left\lceil \frac{j}{\mathsf{wt}(T_i)} \right\rceil - \left\lfloor \frac{j-1}{\mathsf{wt}(T_i)} \right\rfloor && (5.4)
\end{aligned}$$

where the offsets satisfy the property $k \geq j \Rightarrow \theta(T_i^{[k]}) \geq \theta(T_i^{[j]})$. A subtask $T_i^{[j]}$ is *active* at time t iff $\mathsf{r}(T_i^{[j]}) \leq t < \mathsf{d}(T_i^{[j]})$, and a task T_i is *active* at t iff it has an active subtask at t. For example, in Figure 5.1(b), T_i is active in every slot except slot 4. If $\theta(T_i^{[j+1]}) > \theta(T_i^{[j]})$, then we say that there is an IS *separation* between $T_i^{[j]}$ and $T_i^{[j+1]}$. For example, in Figure 5.1(b), there is an IS separation between $T_1^{[1]}$ and $T_1^{[2]}$, as well as between $T_1^{[2]}$ and $T_1^{[3]}$. (Note that an extension of the IS model exists in which a subtask $T_i^{[j]}$ can become eligible before $\mathsf{r}(T_i^{[j]})$ (Srinivasan and Anderson, 2006). All the results of this chapter can be easily extended to such a model, but for clarity, we do not consider this extension to the IS model.)

5.1.3 The PD^2 Algorithm

The PD^2 Pfair scheduling algorithm (Srinivasan and Anderson, 2006) is optimal for scheduling IS tasks on an arbitrary number of processors. It prioritizes subtasks on an earliest-pseudo-deadline-first (EPDF) basis, and uses two tie-breaking rules: the *b-bit* and the *group deadline*. The b-bit of the subtask $T_i^{[j]}$ is defined as

$$\mathsf{b}(T_i^{[j]}) = \left\lceil \frac{j}{\mathsf{wt}(T_i)} \right\rceil - \left\lfloor \frac{j}{\mathsf{wt}(T_i)} \right\rfloor . \tag{5.5}$$

The group deadline of the subtask $T_i^{[j]}$ of a task where $\mathsf{wt}(T_i) \geq 1/2$ is defined as

$$\mathsf{D}(T_i^{[j]}) = \begin{cases} 0, & \mathsf{wt}(T_i) < 1/2 \\ \theta(T_i^{[j]}) + \left\lceil \frac{\left\lceil \left\lceil \frac{j}{\mathsf{wt}(T_i)} \right\rceil \cdot (1-\mathsf{wt}(T_i)) \right\rceil}{1-\mathsf{wt}(T_i)} \right\rceil, & \mathsf{wt}(T_i) \geq 1/2 \end{cases} . \tag{5.6}$$

$\mathsf{b}(T_i^{[j]})$ is 1 in a periodic task system (or an IS system where $\theta(T_i^{[j]}) = 0$ for every subtask), if $T_i^{[j]}$'s window overlaps $T_i^{[j+1]}$'s, and is 0 otherwise. For example, in both insets in Figure 5.1, $\mathsf{b}(T_i^{[j]}) = 1$ for $1 \leq i \leq 4$ and $\mathsf{b}(T_i^{[5]}) = 0$. If two subtasks have equal deadlines, then a subtask with a b-bit of 1 is favored over one with a b-bit of 0. Notice that, in the absence of IS separations, $\mathsf{r}(T_i^{[j+1]}) = \mathsf{d}(T_i^{[j]}) - \mathsf{b}(T_i^{[j]})$. For example, in Figure 5.1(a), $\mathsf{r}(T_i^{[2]}) = \mathsf{d}(T_i^{[1]}) - \mathsf{b}(T_i^{[1]}) = 4 - 1 = 3$, and $\mathsf{r}(T_i^{[6]}) = \mathsf{d}(T_i^{[5]}) - \mathsf{b}(T_i^{[5]}) = 16 - 0 = 16$. Also, if $\mathsf{b}(T_i^{[j]}) = 1$, $\theta(T_i^{[j+1]}) \geq \theta(T_i^{[j]}) + 1$, and $T_i^{[j+1]}$ exists, then $\mathsf{r}(T_i^{[j+1]}) = \mathsf{d}(T_i^{[j]})$. For example, in Figure 5.1(b), $\mathsf{r}(T_1^{[3]}) = \mathsf{d}(T_1^{[2]})$.

In a periodic task system, the group-deadline of the subtask $T_i^{[j]}$ of a heavy task T_i represents the slot after which T_i will not have two overlapping subtask windows, because some subtask has either a b-bit of 0 or a window length of three. (Note that, by (5.6), if T_i is light, then all of its subtasks have a group deadline of zero.) If two subtasks have equal deadlines and b-bits, then subtask with a larger group deadline is favored over a subtask with a smaller group deadline. Further ties are broken arbitrarily. By breaking ties in this manner, the PD^2 scheduling algorithm reduces the impact current scheduling decisions have on future ones. Notice that the subtask associated with the group deadline of $T_i^{[j]}$, $T_i^{[k]}$, satisfies one of the following two conditions (assuming T_i is heavy):

(ω-i) $\mathsf{r}(T_i^{[k]}) > \mathsf{r}(T_i^{[j]})$ and $\mathsf{d}(T_i^{[k]}) - \mathsf{r}(T_i^{[k]}) = 3$.

(ω-ii) $\mathsf{r}(T_i^{[k]}) \geq \mathsf{r}(T_i^{[j]})$, $\mathsf{d}(T_i^{[k]}) - \mathsf{r}(T_i^{[k]}) = 2$, and $\mathsf{b}(T_i^{[k]}) = 0$.

Example (Figure 5.2). Consider the example in Figure 5.1, which depicts the releases and deadlines for the task T_1, which has $\mathsf{wt}(T_1) = 7/9$. (Again, the per-slot allocations are considered later.) Notice that, for $j \in \{1, 2, 3\}$, $\mathsf{D}(T_1^{[j]}) = 5$, and for $j \in \{4, 5, 6, 7\}$, $\mathsf{D}(T_1^{[j]}) = 9$. $T_1^{[4]}$ is the subtask associated with the group deadline of $T_1^{[1]}$, ..., $T_1^{[3]}$ and satisfies condition (ω-i). $T_1^{[7]}$ is the subtask associated with the group deadline of $T_1^{[4]}$, ..., $T_1^{[7]}$ and satisfies condition (ω-ii).

We use $\omega(T_i^{[j]})$ to denote the subtask that is associated with the group deadline of $T_i^{[j]}$.

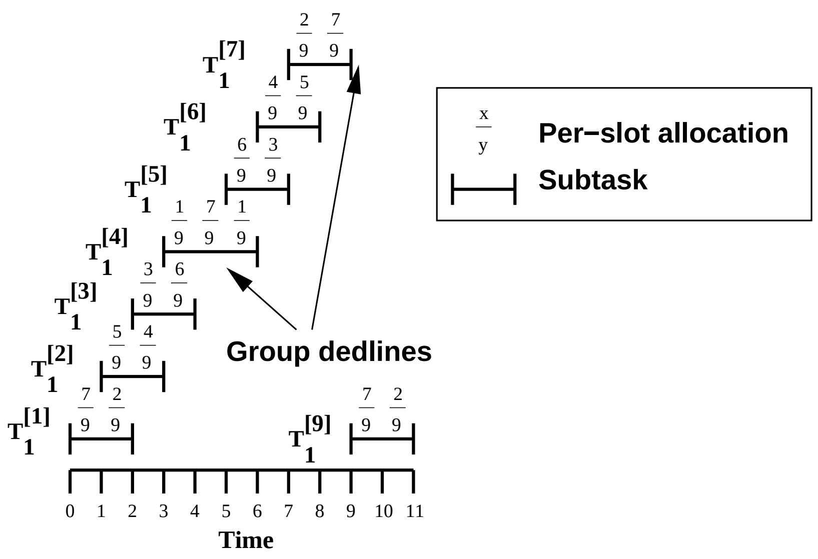


Figure 5.2: $\mathsf{A}(\mathcal{I},\, T_i^{[j]},\, t)$ for a periodic task with weight 7/9.

Specifically,

$$\omega(T_i^{[j]}) = \begin{cases} \text{first subtask that satisfies } (\omega\text{-i}) \text{ or } (\omega\text{-ii}) \text{ for } T_i^j, & \text{if } \mathsf{D}(T_i^{[j]}) > 0 \\ \text{undefined}, & \text{if } \mathsf{D}(T_i^{[j]}) = 0 \end{cases} \tag{5.7}$$

For example, in Figure 5.2, for $j \in \{1,2,3\}$, $\omega(T_1^{[j]}) = T_1^{[4]}$, and for $j \in \{4,5,6,7\}$, $\omega(T_1^{[j]}) = T_1^{[7]}$. Notice that, if a subtask has a b-bit of 0, then its group deadline is its own deadline (e.g., in Figure 5.2, $\mathsf{D}(T_1^{[7]}) = \mathsf{d}(T_1^{[7]}) = 9$); however, if a subtask has a window length of three, then its group deadline is after its deadline (e.g., $\mathsf{D}(T_1^{[4]}) = 9 \geq 6 = \mathsf{d}(T_1^{[4]})$).

It is easy to see that for IS task systems, $\omega(T_i^{[j]})$ is defined if T_i is heavy and undefined if T_i is light. However, this will not necessarily hold for the adaptable IS task model, which is defined in Section 5.2.

From the definition of a group deadline it is not difficult to show that the following properties holds.

(GD-1) For any $T_i^{[j]}$ such that $\mathsf{D}(T_i^{[j]}) > 0$, if for all $T_i^{[q]} \in \{T_i^{[j]}, ..., \omega(T_i^{[j]})\}$, $\theta(T_i^{[q]}) = \theta(T_i^{[j]})$ (i.e., there are no IS separations between subtasks until the group deadline), then:

(i) $\mathsf{r}(\omega(T_i^{[j]})) = \mathsf{D}(T_i^{[j]}) - 2$;

(ii) either $\mathsf{d}(\omega(T_i^{[j]})) = \mathsf{D}(T_i^{[j]})$ and $\mathsf{b}(\omega(T_i^{[j]})) = 0$ (i.e., $\omega(T_i^{[j]})$ has a window length of

```
$\mathsf{A}(\mathcal{I}_{\mathsf{IS}}, T_i^{[j]}, t)$
1:  if $(t < \mathsf{r}(T_i^{[j]})) \vee (t \geq \mathsf{d}(T_i^{[j]}))$ then
2:      $\mathsf{A}(\mathcal{I}_{\mathsf{IS}}, T_i^{[j]}, t) := 0$
3:  else if $t = \mathsf{r}(T_i^{[j]})$ then
4:      if $j = 1 \vee \mathsf{b}(T_i^{[j-1]}) = 0$ then
5:          $\mathsf{A}(\mathcal{I}_{\mathsf{IS}}, T_i^{[j]}, t) := \mathsf{wt}(T_i)$
6:      else
7:          $\mathsf{A}(\mathcal{I}_{\mathsf{IS}}, T_i^{[j]}, t) :=$
                $\mathsf{wt}(T_i) - \mathsf{A}(\mathcal{I}_{\mathsf{IS}}, T_i^{[j-1]}, \mathsf{d}(T_i^{[j-1]}) - 1)$
8:      fi
9:  else
10:     $\mathsf{A}(\mathcal{I}_{\mathsf{IS}}, T_i^{[j]}, t) :=$
            $\min(\mathsf{wt}(T_i), 1 - \mathsf{A}(\mathcal{I}_{\mathsf{IS}}, T_i^{[j]}, 0, t))$
11: fi
```

Figure 5.3: Pseudo-code defining $\mathsf{A}(\mathcal{I}_{\mathsf{IS}}, T_i^{[j]}, t)$.

two and a b-bit of zero) or $\mathsf{d}(\omega(T_i^{[j]})) = \mathsf{D}(T_i^{[j]}) + 1$ and $\mathsf{b}(\omega(T_i^{[j]})) = 1$ (i.e., $\omega(T_i^{[j]})$ has a window length of three).

(GD-2) If $\mathsf{D}(T_i^{[j]}) > 0$, then $\mathsf{D}(T_i^{[j]}) \geq \mathsf{d}(T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$.

5.1.4 IS Ideal Schedule

Ideal allocations within the IS task model are defined so that the cumulative allocation for one subtask is one and the total per-slot allocation is at most the weight of the corresponding task (Srinivasan and Anderson, 2006). The total allocation to a task in a given time slot equals the total allocation to all of its subtasks in that slot. Thus, for any ideal schedule $\mathcal{E}$,

$$\mathsf{A}(\mathcal{E}, T_i, t) = \sum_{T_i^{[j]} \in T_i} \mathsf{A}(\mathcal{E}, T_i^{[j]}, t).$$

For example, in Figure 5.1(a), $\mathsf{A}(\mathcal{E}, T_i, 6) = \mathsf{A}(\mathcal{E}, T_1, 6) + \mathsf{A}(\mathcal{E}, T_2, 6) + \mathsf{A}(\mathcal{E}, T_3, 6) + ... = 0 + 2/16 + 3/16 + 0 + ... = 5/16$. Thus, per-task and per-task-set allocations in the ideal schedule $\mathcal{E}$ over an arbitrary interval can be defined by simply defining $\mathsf{A}(\mathcal{E}, T_i^{[j]}, t)$ for an arbitrary subtask $T_i^{[j]}$ and time slot t.

For an arbitrary IS task system τ, we let $\mathcal{I}_{\mathsf{IS}}$ denote the ideal schedule of τ. $\mathsf{A}(\mathcal{I}_{\mathsf{IS}}, T_i^{[j]}, u)$ can be defined using an arithmetic expression, but we have opted instead for a more intuitive pseudo-code-based definition in Figure 5.3. The ideal IS schedule allocates each subtask $T_i^{[j]}$ some amount of processing time in each slot of its window. For slots other than $\mathsf{r}(T_i^{[j]})$ and $\mathsf{d}(T_i^{[j]})-1$, this allocation is $\mathsf{wt}(T_i)$. $T_i^{[j]}$'s allocation in slots $\mathsf{r}(T_i^{[j]})$ and $\mathsf{d}(T_i^{[j]})-1$ are adjusted so that

(i) $T_i^{[j]}$'s entire allocation (across all slots in its window) is one.

(ii) $T_i^{[j]}$'s allocation in slot $\mathsf{r}(T_i^{[j]})$ plus $T_i^{[j-1]}$'s allocation in slot $\mathsf{d}(T_i^{[j-1]}) - 1$ equals $\mathsf{wt}(T_i)$ (assuming $T_i^{[j]}$ and $T_i^{[j-1]}$ exist).

(iii) $T_i^{[j]}$'s allocation in slot $\mathsf{d}(T_i^{[j]}) - 1$ plus $T_i^{[j+1]}$'s allocation in slot $\mathsf{r}(T_i^{[j+1]})$ equals $\mathsf{wt}(T_i)$ (assuming $T_i^{[j]}$ and $T_i^{[j+1]}$ exist).

Examples of such allocations are given in Figure 5.1.

5.1.5 Dynamic Task Systems

The *dynamic* IS *task model* is an extension of the IS model in which tasks can leave and join by conditions defined in (Srinivasan and Anderson, 2005), which are stated below.

J: (*join condition*) A task T_i can join at time t iff the sum of the weights of all tasks after joining is at most M.

L: (*leave condition*) Let $T_i^{[j]}$ denote the last-scheduled subtask of T_i. If T_i is light, then T_i can leave at time t iff $t \geq \mathsf{d}(T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$. If T_i is heavy, then T_i can leave at time t iff $t \geq \mathsf{D}(T_i^{[j]})$.

For example, in Figure 5.1(b), if T_1 were to leave after $T_1^{[1]}$, then T_1 could not leave until time 5 because $5 = \mathsf{d}(T_1^{[1]}) + \mathsf{b}(T_1^{[1]}) = 4 + 1$. Moreover, if T_1 were to leave after $T_1^{[5]}$, then T_i could not leave until time 19 because $19 = \mathsf{d}(T_1^{[5]}) + \mathsf{b}(T_1^{[5]}) = 19 + 0$.

Theorem 5.1 (From (Srinivasan and Anderson, 2005)). PD^2 *correctly schedules any dynamic* IS *task system satisfying J and L.*

By Theorem 5.1, a task may be reweighted by leaving with its old weight and rejoining with its new weight.

5.2 Adaptable Task Model

In this section, we introduce the *adaptable* IS (AIS) task model. The AIS task model is an extension of IS task model, where the weight of each task T_i, $\mathsf{wt}(T_i,\, t)$, is a function of time t. (The AIS task model is similar to the adaptable sporadic task model presented in Chapter 3, except that the AIS task model is designed for Pfair-scheduled systems.) For brevity, we use $\mathsf{wt}(T_i)$ to denote the value of a task T_i that never changes its weight.

Before continuing, it is important to mention that while for periodic and IS tasks the weight of a task is bounded by the range $(0, 1)$, for AIS tasks, it is possible for a task's weight to equal 0. When a task's weight equals 0, we assume, without loss of generality, that it has left the system and will not return.

A task T_i *changes weight* or *reweights* at time $t+1$ if $\mathsf{wt}(T_i,\, t) \neq \mathsf{wt}(T_i,\, t+1)$. If a task T_i changes weight at a time t_c between the release and the deadline of some subtask $T_i^{[j]}$, then the following two actions *may* occur:

(i) If $T_i^{[j]}$ has not been scheduled by t_c, then $T_i^{[j]}$ may be "halted" at t_c.

(ii) $\mathsf{r}(T_i^{[j+1]})$ *may* be redefined to be less than $\mathsf{d}(T_i^{[j]}) - \mathsf{b}(T_i^{[j]})$.

In addition, if a heavy task, T_i, changes its weight at time t_c and $T_i^{[j]}$ is the last-released subtask of T_i, then the following three additional actions *may* occur:

(iii) The window length of every subtask released in the range $\{\mathsf{r}(T_i^{[j]}) + 1, ..., \mathsf{D}(T_i^{[j]}) - 2\}$ may have a window length of two, a b-bit of 1, and a group deadline of $\mathsf{D}(T_i^{[j]})$ regardless of the new weight of the task.

(iv) No subtask of T_i will be released at time $\mathsf{D}(T_i^{[j]}) - 1$.

(v) If T_i decreases its weight, then no task, other than T_i, can use the "freed" capacity until $\mathsf{D}(T_i^{[j]})$.

As we discuss shortly, the reason why the above actions may occur is because the releases, deadlines, b-bits, and group deadlines of subtasks may change as a result of a reweighting event. The reweighting rules we present at the end of this section state the conditions under

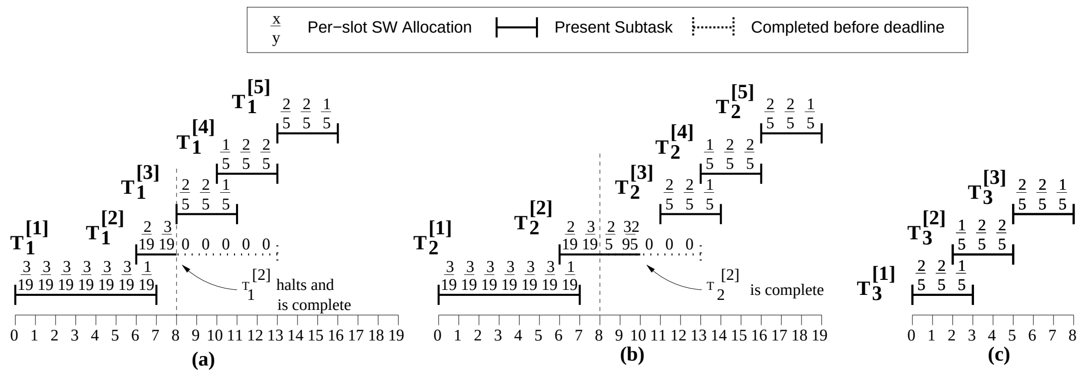


Figure 5.4: The per-slot SW allocations for three different AIS tasks. **(a)** T_1 changes its weight from 3/19 to 2/5 and $T_1^{[2]}$ halts. **(b)** T_2 changes its weight from 3/19 to 2/5 and $T_1^{[2]}$ does not halt. **(c)** T_3 does not change its weight from 2/5.

which the above actions may occur and the number of slots before $\mathsf{d}(T_i^{[j]}) - \mathsf{b}(T_i^{[j]})$ that subtask $T_i^{[j+1]}$ can be released.

Halting. If $T_i^{[j]}$ is *halted* before it is scheduled, then it is never scheduled. (Note that a subtask can only be halted if it has not yet been scheduled in the PD^2 schedule.) Since a subtask is only halted as a result of a reweighting event, if we do not have *a priori* knowledge of such events, then we cannot determine whether a released subtask will be halted in the future. It is important to note that we consider a subtask $T_i^{[j]}$ to be active at time t only if $\mathsf{r}(T_i^{[j]}) \leq t < \mathsf{d}(T_i^{[j]})$ and $T_i^{[j]}$ has not been halted by t.

Example (Figure 5.4). Consider the example in Figure 5.4, which depicts three different tasks. Inset (a) depicts a task T_1, which has an initial weight of 3/19 and at time 8 both halts its current subtask and changes its weight to 2/5. Inset (b) depicts a task T_2, which has an initial weight of 3/19 and at time 8 changes its weight to 2/5 but *but does not halt* the current subtask. Inset (c) depicts a periodic task T_3, which has $\mathsf{wt}(T_3) = 2/5$. The dotted window lines indicate that the window would have existed if the subtask task did not reweight. Notice that, in inset (a), we have no knowledge when $T_1^{[2]}$ is released at time 6 that it will be halted at time 8. (We will consider the per-slot SW allocations depicted in the figure later.) □

Definition 5.1 (Initiated, Enacted, Freed Capacity, and Reset). When a task reweights, there can be a difference between when it "initiates" the change, when the change is "enacted," and when any newly-available capacity is "freed." The time at which the change is *initiated* is a user-defined time; the time at which the change is *enacted* and the *capacity is freed* (in the case of a weight decrease) are both dictated by a set of conditions discussed shortly. We use the *scheduling weight of a task T_i at time t*, denoted $\mathsf{Swt}(T_i, t)$, to represent the "last enacted weight of T_i." Formally, $\mathsf{Swt}(T_i, t)$ equals $\mathsf{wt}(T_i,\ u)$, where u is the last time at or before t that a weight change was enacted for T_i. (For the purposes of this definition, we assume an initial weight change occurred for T_i when it initially joined the system.) It is important to note that, *we henceforth compute subtask deadlines and releases using scheduling weights.* If T_i decreases its scheduling weight from Ow to Nw, then until the capacity has been freed, no other task can use the capacity $\mathsf{Nw} - \mathsf{Ow}$ gained by decreasing T_i's scheduling weight. A weight change is finalized by "resetting" the corresponding task. When a task T_i *is reset at time t*, its future releases and deadlines are changed so that it is as though the T_i joined at time t. The times at which a task can be reset are described in the following reweighting rules. We use $\mathsf{En}(T_i, t)$ to denote the last time at or before time t that T_i was reset, and $\mathsf{Id}(T_i^{[j]})$ to denote the smallest index k such that $\mathsf{En}(T_i, \mathsf{r}(T_i^{[j]})) \leq \mathsf{r}(T_i^{[k]})$ holds.

Example (Figure 5.4). In Figure 5.4(b), $\mathsf{En}(T_2, t) = 0$, for $0 \leq t < 8$; for $j \in \{1, 2\}$, $\mathsf{Id}(T_2^{[j]}) = 1$; for $t \geq 8$, $\mathsf{En}(T_2, t) = 8$; and for $j \in \{3, 4, 5\}$, $\mathsf{Id}(T_2^{[j]}) = 3$. Note that, if $\mathsf{Id}(T_i^{[j]}) = j$, then $T_i^{[j]}$ is the first subtask of T_i released at or after a weight change for T_i has been enacted. $\square$

Example (Figure 5.5). Consider the one-processor example in Figure 5.5, which consists of three tasks: T_1, which has $\mathsf{wt}(T_1) = 1/2$; T_2, which has an initial weight of 1/5 that increases to 3/10 at time 5; and T_3, which has an initial weight of 3/10 that initiates a weight decrease to 1/5 at time 2 that is enacted at time 5. The capacity gained from T_3's weight decrease is not freed until time 5. Thus, T_2 cannot increase its weight to 3/10 until this time. $\square$

Definition 5.2 (Complete). If $\mathcal{S}$ is a schedule for the task system τ, then a subtask $T_i^{[j]}$ of $T_i \in \tau$ is said to have *completed by time t in $\mathcal{S}$* iff $t \geq \mathsf{r}(T_i^{[j]})$ and one of the following holds: **(i)** $T_i^{[j]}$ has been allocated one quantum by t in $\mathcal{S}$; or **(ii)** $T_i^{[j]}$ is halted by time t. We use the function $\mathsf{C}(\mathcal{S}, T_i^{[j]})$ to denote the earliest (integral) time at which $T_i^{[j]}$ is complete in $\mathcal{S}$.

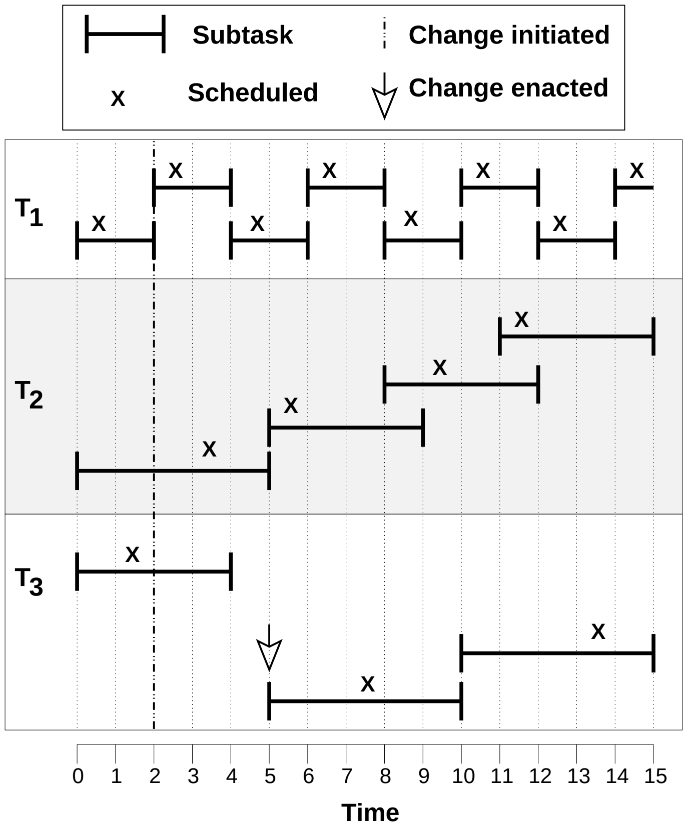


Figure 5.5: A one-processor system with three tasks: T_1, which has $\mathsf{wt}(T_1) = 1/2$; T_2, which has an initial weight of 1/5 that increases to 3/10 at time 5; and T_3, which has an initial weight of 3/10 that initiates a weight decrease to 1/5 at time 2 that is enacted at time 5.

Example (Figure 5.6). Consider the example in Figure 5.6, which depicts a one-processor PD^2 schedule for two tasks: T_1, which has $\mathsf{wt}(T_1) = 2/5$, and T_2, which has an initial weight of 2/5 that increases to 1/2 at time 3 by halting $T_2^{[2]}$. In this example, $T_1^{[1]}$ is complete in the PD^2 schedule by time 1 because it is scheduled in slot 0, whereas $T_2^{[1]}$ does is not complete in the PD^2 schedule until time 2 because it is not scheduled until slot 1. Notice that, since $T_2^{[2]}$ is halted at time 3, it is complete at time 3 even though it is never scheduled. □

For an adaptable task, the *deadline*, *b-bit*, *release*, and group deadline of a subtask $T_i^{[j]}$, respectively, are defined by (5.8)–(5.12) below, where $z = \mathsf{ld}(T_i^{[j]}) - 1$, $\theta(T_i^{[j+1]}) \geq \theta(T_i^{[j]}) \geq 0$.

$$\mathsf{d}(T_i^{[j]}) \quad = \quad \mathsf{r}(T_i^{[j]}) + \left\lceil \frac{j-z}{\mathsf{Swt}(T_i, \mathsf{r}(T_i^{[j]}))} \right\rceil - \left\lfloor \frac{j-z-1}{\mathsf{Swt}(T_i, \mathsf{r}(T_i^{[j]}))} \right\rfloor \tag{5.8}$$

$$\mathsf{b}(T_i^{[j]}) \quad = \quad \left\lceil \frac{j-z}{\mathsf{Swt}(T_i, \mathsf{r}(T_i^{[j]}))} \right\rceil - \left\lfloor \frac{j-z}{\mathsf{Swt}(T_i, \mathsf{r}(T_i^{[j]}))} \right\rfloor \tag{5.9}$$

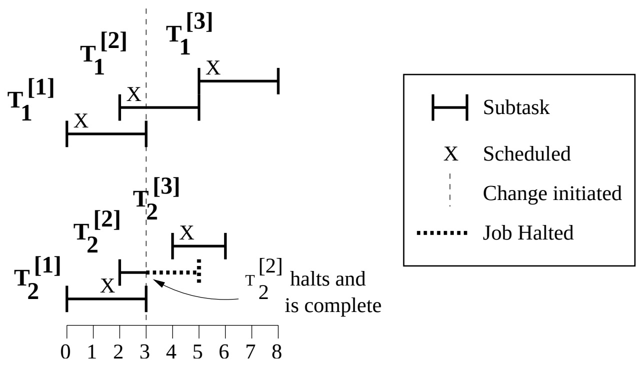


Figure 5.6: A one-processor PD^2 schedule for two tasks. The X's denote where each subtask is scheduled.

$$\mathsf{r}(T_i^{[1]}) = \theta(T_i^{[1]}) \tag{5.10}$$

$$\mathsf{r}(T_i^{[j+1]}) = \mathsf{d}(T_i^{[j]}) - \mathsf{b}(T_i^{[j]}) + \left(\theta(T_i^{[j+1]}) - \theta(T_i^{[j]})\right) \tag{5.11}$$

$$\mathsf{D}(T_i^{[j]}) = \begin{cases} 0, & \text{if } \mathsf{Swt}(T_i, \mathsf{r}(T_i^{[j]})) < 1/2 \\ \left\lceil \dfrac{\left\lceil \left\lceil \frac{j-z}{\mathsf{Swt}(T_i,\mathsf{r}(T_i^{[j]}))} \right\rceil \cdot \left(1 - \mathsf{Swt}(T_i,\mathsf{r}(T_i^{[j]}))\right) \right\rceil}{1-\mathsf{Swt}(T_i,\mathsf{r}(T_i^{[j]}))} \right\rceil & , \quad \text{if } \mathsf{Swt}(T_i, \mathsf{r}(T_i^{[j]})) \geq 1/2 \\ \quad + \left(\theta(T_i^{[j]}) - \theta(T_i^{[z+1]}) + \mathsf{r}(T_i^{[z+1]})\right) & \end{cases} \tag{5.12}$$

It is important to note that, since reweighting events may change a subtask's release time, b-bit, deadline, and group deadline, values obtained from all of these formulas are subject to be changed by the reweighting rules presented in Section 5.4.

The above equations differ (5.2)–(5.6) in two ways. First, (5.8), (5.9), and (5.12) define the deadline, b-bit, and group-deadline of a subtask based on the *scheduling weight* of the task at the time the subtask is *released*. Second, after a task enacts a weight change, its release, deadline, b-bit, and group deadline are defined as though a new task with the new weight joined the system. (Recall that a subtask $T_i^{[j]}$ is the first-released subtask after the task is reset iff $\mathsf{ld}(T_i^{[j]}) = j$.) For example, in Figure 5.4(a), after T_1 changes its weight to $2/5$, the subtasks $T_1^{[3]}$–$T_1^{[5]}$ have similar releases, deadlines, and b-bits as the first three subtasks of the task T_3 with weight $2/5$ in inset (c).

```
A($\mathcal{S}W$, $T_i^{[j]}$, $t$)
1:  if $t < \mathsf{r}(T_i^{[j]}) \vee t \geq \mathsf{C}(\mathcal{S}W, T_i^{[j]})$ then
2:      $\mathsf{A}(\mathcal{S}W,\ T_i^{[j]},\ t) := 0$
3:  else if $t = \mathsf{r}(T_i^{[j]})$ then
4:      if $j = \mathsf{ld}(T_i^{[j]}) \vee \mathsf{b}(T_i^{[j-1]}) = 0$ then
5:          $\mathsf{A}(\mathcal{S}W,\ T_i^{[j]},\ t) := \mathsf{Swt}(T_i, t)$
6:      else
7:          $\mathsf{A}(\mathcal{S}W,\ T_i^{[j]},\ t) := \mathsf{Swt}(T_i, t) -$
                $\mathsf{A}(\mathcal{S}W,\ T_i^{[j-1]},\ \mathsf{C}(\mathcal{S}W, T_i^{[j-1]}) - 1)$
8:      fi
9:  else
10:     $\mathsf{A}(\mathcal{S}W,\ T_i^{[j]},\ t) :=$
            $\mathsf{min}(\mathsf{Swt}(T_i, t), 1 - \mathsf{A}(\mathcal{S}W,\ T_i^{[j]},\ 0,\ t))$
11: fi
```

Figure 5.7: Pseudo-code defining the $\mathsf{A}(\mathcal{S}W,\ T_i^{[j]},\ t)$.

5.3 SW Scheduling Algorithm

Just as with the adaptable sporadic task model in Chapters 3–4, in order to prove correctness and drift properties, we introduce the *scheduling-weight* (SW) scheduling algorithm for the AIS task model. Just as with the adaptable sporadic task model, we use $\mathcal{S}W$, to denote the SW schedule for a given system.

As with $\mathcal{I}_{\mathsf{IS}}$, $\mathsf{A}(\mathcal{S}W,\ T_i^{[j]},\ t)$ can be defined mathematically, but we opt instead for a pseudo-code-based definition, shown in Figure 5.7. There are three differences between the definitions of $\mathsf{A}(\mathcal{I}_{\mathsf{IS}},\ T_i^{[j]},\ t)$ (in Figure 5.3) and $\mathsf{A}(\mathcal{S}W,\ T_i^{[j]},\ t)$: in lines 5, 7, and 10 (in Figure 5.7), $\mathsf{Swt}(T_i, t)$ is used instead of $\mathsf{wt}(T_i)$; and in lines 1 and 7 (in Figure 5.7), $\mathsf{C}(\mathcal{S}W, T_i^{[j]})$ is used instead of $\mathsf{d}(T_i^{[j]})$. These two changes account for T_i's time-varying weight. The final change is that, in line 4, $j = \mathsf{ld}(T_i^{[j]})$ is used instead of $j = 1$. This change causes the per-slot allocations of $T_i^{[z]}$, where $z = \mathsf{ld}(T_i^{[j]})$, to equal that of a task that joins the system at $\mathsf{r}(T_i^{[z]})$.

Example (Figure 5.4). In Figure 5.4(a), since $\mathsf{ld}(T_1^{[3]}) = 3$, by lines 4 and 5, $\mathsf{A}(\mathcal{S}W,\ T_1^{[3]},\ \mathsf{r}(T_1^{[3]})) = \mathsf{Swt}(T_i, \mathsf{r}(T_1^{[3]})) = 2/5$, which is the same per-slot allocation that $T_3^{[1]}$ in Figure 5.4(c) receives at time $\mathsf{r}(T_3^{[1]})$. □

Before continuing, there are two important issues to note. First, in the absence of reweighting events, $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) = \mathsf{d}(T_i^{[j]})$. Second, when a task is halted via the reweighting rules given below, it is halted in both the PD^2 and $\mathcal{S}W$ schedules. Since $\mathcal{S}W$ is not clairvoyant, it will allocate "normally" to a subtask until that subtask halts, after which the subtask's per-slot allocations are zero, as with $T_1^{[2]}$ in Figure 5.4(a). Also note that, in Figure 5.4(b), $T_2^{[2]}$ is complete at time 10, since $\mathsf{A}(\mathcal{S}W, T_2^{[2]}, 0, 10) = 1$. Several other examples of $\mathcal{S}W$ allocations are also given in Figure 5.4.

5.4 Reweighting Rules

In this section, we introduce three reweighting rules that improve upon leave/join reweighting by changing future subtask releases. It is important to note that, in the following rules, for a given subtask $T_i^{[j]}$, the value $\mathsf{d}(T_i^{[j]})$ is used to determine the scheduling priority of $T_i^{[j]}$ in the PD^2 algorithm and *does not change* once $T_i^{[j]}$ has been released. Furthermore, $\mathsf{C}(\mathcal{S}W, T_i^{[j]})$ is used for some of these rules to determine the *release time* of $T_i^{[j]}$'s successor, $T_i^{[j+1]}$. As mentioned earlier, the completion time of a subtask cannot be accurately predicted without *a priori* knowledge of weight changes; however, in the reweighting rules below, the completion time of a subtask in $\mathcal{S}W$ is only used *after* the subtask has completed, and therefore it is well-defined.

Assumptions and definitions. Let τ be a task system in which some task T_i initiates a weight change from weight Ow to weight Nw at time t_c. If there does not exist a subtask $T_i^{[j]}$ of T_i such that $\mathsf{r}(T_i^{[j]}) \leq t_c$, then the weight change is enacted immediately; otherwise, let $T_i^{[j]}$ denote the last-released subtask of T_i. *For simplicity, we assume that the first subtask after a weight change by the corresponding task is released as early as possible. In addition, we assume that for a heavy task all subtasks released before the group deadline are released as early as possible.* These assumptions can be removed at the cost of more complex notation.

The choice of which rule to apply depends on the weight of T_i, whether $t_c \leq \mathsf{d}(T_i^{[j]})$, and whether $T_i^{[j]}$ has been scheduled by t_c. We say that T_i is *heavy-changeable at time t_c from Ow to Nw* if $t_c < \mathsf{D}(T_i^{[j]})$ (recall that light tasks have a group deadline of 0). If T_i is not heavy-changeable at time t_c and $\mathsf{d}(T_i^{[j]}) \leq t_c$, then the weight change is enacted at time $\mathsf{max}(t_c, \mathsf{d}(T_i^{[j]}) + \mathsf{b}(T_i^{[j]}))$. If $\mathsf{D}(T_i^{[j]}) \leq t_c < \mathsf{d}(T_i^{[j]})$ and $T_i^{[j]}$ has been scheduled by time t_c, then we say that T_i is *negative-changeable at time* t_c *from weight* Ow *to* Nw; otherwise, if $T_i^{[j]}$ has not been scheduled by t_c, then we say T_i is *positive-changeable at time* t_c *from* Ow *to* Nw.[1]

Because T_i initiates a weight change at time t_c, $\mathsf{wt}(T_i, t_c) = \mathsf{Nw}$ holds; however, T_i's scheduling weight does not change until the weight change has been *enacted*, as specified in the rules below. Note that, if t_c occurs between the initiation and enaction of a previous reweighting event of T_i, then the previous event is *canceled*, i.e., treated as if it had not occurred. As discussed later, any "error" associated with skipping a reweighting event like this is accounted for when determining drift.

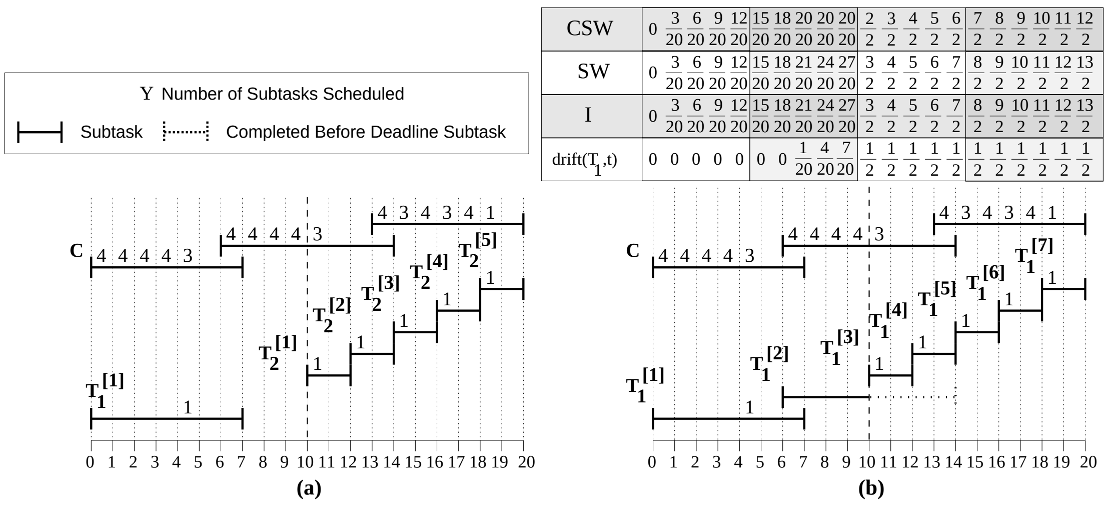
5.4.1 Positive- and Negative-Changeable

We first describe the rules for reweighting positive- and negative-changeable tasks, then, in Section 5.4.2, describe the rule for reweighting heavy-changeable tasks. Notice that, for both of the rules below, the capacity gained by decreasing a task's weight is freed when the change is enacted.

Rule P: If T_i is positive-changeable at time t_c from weight Ow to Nw, T_i has released a subtask prior to t_c and $T_i^{[j]}$ is the last such subtask, $t_c < \mathsf{d}(T_i^{[j]})$, and $j > 1$, then at time t_c, subtask $T_i^{[j]}$ is halted and at time $\mathsf{max}(t_c, \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]}))$, T_i's weight change is enacted, T_i is reset, and a new subtask $T_i^{[j+1]}$ is released. If $j = 1$, then at time t_c, $T_i^{[j]}$ is halted, T_i's weight change is enacted, T_i is reset, and a new subtask $T_i^{[j+1]}$ is released.

Rule N: If T_i is negative-changeable at time t_c from weight Ow to Nw, T_i has released a subtask prior to t_c and $T_i^{[j]}$ is the last such subtask, and $t_c < \mathsf{d}(T_i^{[j]})$, then one of two actions is taken: **(i)** if $\mathsf{Nw} > \mathsf{Ow}$, then the weight change is *immediately* enacted, and at time $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$, T_i is reset and a new subtask $T_i^{[j+1]}$ is released; **(ii)** otherwise, at time $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$, T_i is reset, the change is enacted, and a new subtask $T_i^{[j+1]}$ is released.

[1]Originally, in (Block et al., 2008a), a negative-changeable task was called *ideal-changeable* and a task that was positive-changeable was called *omission-changeable*. The names have been changed here to be consistent with the reweighting rules for GEDF and PEDF.



CSW	0	$\frac{3}{20}$	$\frac{6}{20}$	$\frac{9}{20}$	$\frac{12}{20}$	$\frac{15}{20}$	$\frac{18}{20}$	$\frac{20}{20}$	$\frac{20}{20}$	$\frac{20}{20}$	$\frac{2}{2}$	$\frac{3}{2}$	$\frac{4}{2}$	$\frac{5}{2}$	$\frac{6}{2}$	$\frac{7}{2}$	$\frac{8}{2}$	$\frac{9}{2}$	$\frac{10}{2}$	$\frac{11}{2}$	$\frac{12}{2}$
SW	0	$\frac{3}{20}$	$\frac{6}{20}$	$\frac{9}{20}$	$\frac{12}{20}$	$\frac{15}{20}$	$\frac{18}{20}$	$\frac{21}{20}$	$\frac{24}{20}$	$\frac{27}{20}$	$\frac{3}{2}$	$\frac{4}{2}$	$\frac{5}{2}$	$\frac{6}{2}$	$\frac{7}{2}$	$\frac{8}{2}$	$\frac{9}{2}$	$\frac{10}{2}$	$\frac{11}{2}$	$\frac{12}{2}$	$\frac{13}{2}$
I	0	$\frac{3}{20}$	$\frac{6}{20}$	$\frac{9}{20}$	$\frac{12}{20}$	$\frac{15}{20}$	$\frac{18}{20}$	$\frac{21}{20}$	$\frac{24}{20}$	$\frac{27}{20}$	$\frac{3}{2}$	$\frac{4}{2}$	$\frac{5}{2}$	$\frac{6}{2}$	$\frac{7}{2}$	$\frac{8}{2}$	$\frac{9}{2}$	$\frac{10}{2}$	$\frac{11}{2}$	$\frac{12}{2}$	$\frac{13}{2}$
drift(T_1,t)	0	0	0	0	0	0	0	$\frac{1}{20}$	$\frac{4}{20}$	$\frac{7}{20}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

Figure 5.8: A four-processor illustration of Rule P under PD^2. C is a set of 19 tasks with a weight of 3/20 each. **(a)** T_1 leaves at time 8 and T_2 joins at time 10. **(b)** T_1 reweights to 1/2 via rule P at slot 10.

Both rules are extensions of Rules L and J given earlier in Section 5.1. However, the rules above exploit the specific circumstances that occur when a task changes its weight to "short circuit" Rules L and J, so that reweighting is accomplished faster. By Rule L, T_i can leave at time $\mathsf{d}(T_i^{[k]}) + \mathsf{b}(T_i^{[k]})$, where $T_i^{[k]}$ is its last-scheduled subtask. We can easily extend Rule L to show that T_i can leave at time $\mathsf{C}(\mathcal{S}W, T_i^{[k]}) + \mathsf{b}(T_i^{[k]})$. If task T_i (as defined above) is positive-changeable, then its subtask $T_i^{[j]}$ has not been scheduled by time t_c. Such a task can be viewed as having "left" the system at time $\max(t_c, \mathsf{C}(\mathcal{S}W, T_i^{[j-1]}) + \mathsf{b}(T_i^{[j-1]}))$, in which case, it can rejoin the system immediately.

Example (Figure 5.8). Consider the four-processor example in Figure 5.8, which illustrates Rule P. Inset (a) depicts a set C of 19 tasks each with a weight of 3/20, a task T_1 with a weight of 3/20 that leaves at time 8, and T_2 that joins at time 10 with a weight of 1/2. Inset (b) depicts the same set C plus a task T_1 that has an initial weight of 3/20 that increases its weight to 1/2 at time 10 via Rule P. Notice that T_1 and T_2 in inset (a) are scheduled in the same time slots as T_1 in inset (b). The top of inset (b) depict T_1's per-slot allocations for the SW, CSW, and IDEAL scheduling algorithms as well as T_1's drift. The terms CSW, IDEAL, and drift are defined later in Section 5.6. $\square$

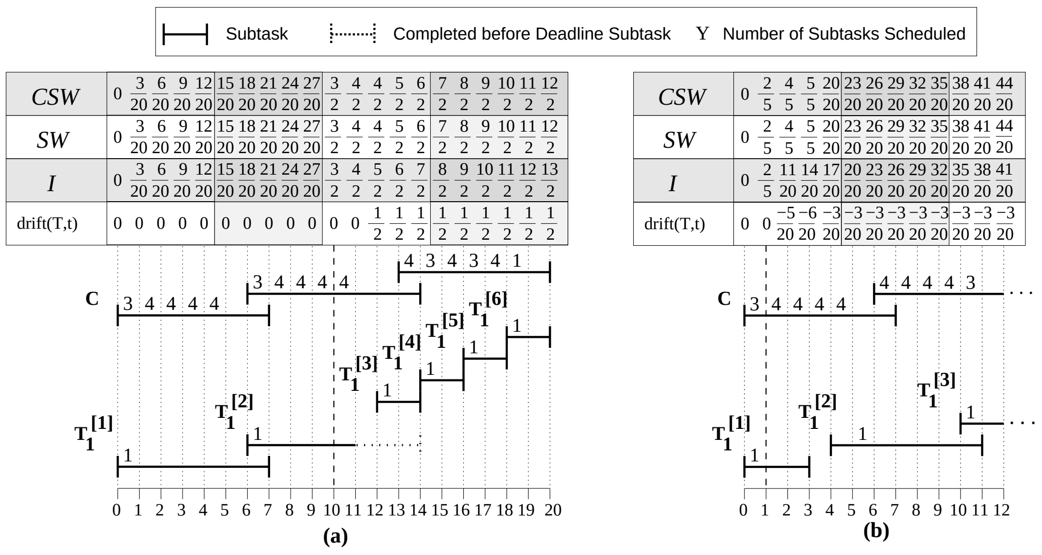


Figure 5.9: A four-processor illustration of Rule N under PD^2. C is a set of 19 tasks with a weight of 3/20 each. **(a)** T_1 increases its weight from 3/20 to 1/2 at time 10 via rule N. **(b)** T_1 decreases its weight from 2/5 to 3/20 via rule N at time 1.

If T_i is negative-changeable, then by Rule L, it may "leave and rejoin" with a new weight at time $\mathsf{d}(T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$, i.e., its weight change can be enacted at time $\mathsf{d}(T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$. However, if $\mathsf{C}(\mathcal{S}W, T_j) < \mathsf{d}(T_i^{[j]})$, then T_i may "leave and rejoin" with a new weight at time $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$.

Example (Figure 5.9). Consider the four-processor example in Figure 5.9, which illustrates Rule N. Inset (a) depicts of a set C of 19 tasks each with a weight of 3/20 and a task T_1 that increases its weight from 3/20 to 1/2 via Rule N. Inset (b) depicts of the same set C and a task T_1 that decreases its weight from 2/5 to 3/20. The top of each inset depict T_1's per-slot allocations for the SW, CSW, and IDEAL scheduling algorithms as well as T_1's drift. Again, the terms CSW, IDEAL, and drift are defined later in Section 5.6. □

Notice that the difference in Rule N between Cases (i) and (ii) is that, when a task increases its weight, the weight change is immediately enacted, whereas when a task decreases its weight, its weight change is not enacted until time $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$. Thus, T_i's scheduling weight is redefined at different times.

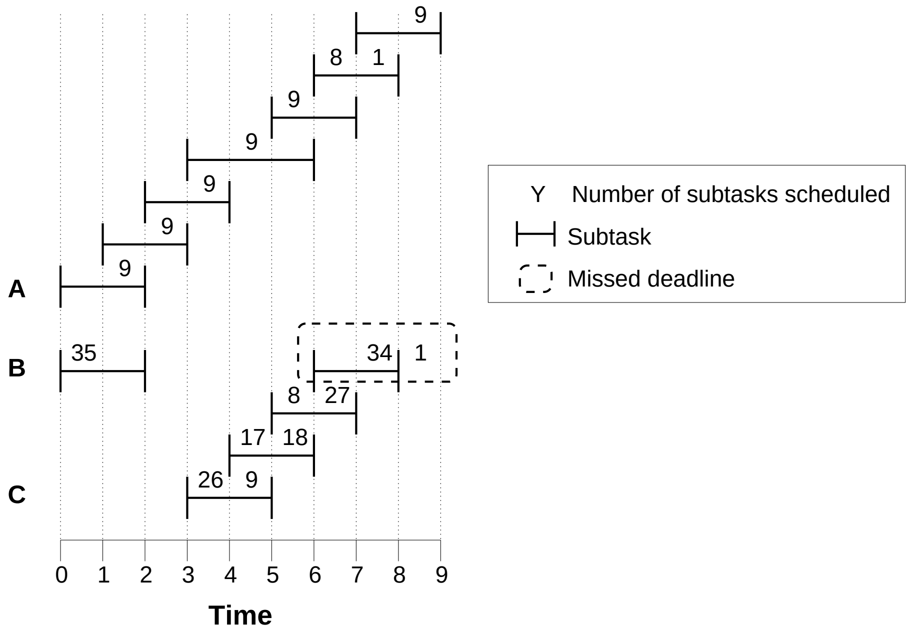


Figure 5.10: A 35-processor system consisting of a set A with nine tasks of weight 7/9; B with 35 tasks with an initial weight of 4/5 that decreases to 0 at time 2; and C with 35 tasks of weight 4/5 that joins at time 3. All tie-breaks go against tasks in set A.

5.4.2 Heavy-Changeable

One of the major complications with changing the weight of a heavy task is that when such a task decreases its weight, capacity cannot be freed until the group deadline of the last-released subtask of that task. If this capacity is freed sooner, then a subtask may miss its deadline.

Example (Figure 5.10). Consider the example in Figure 5.10, which depicts a 35-processor system that is assigned 79 tasks: set A, which consists of nine tasks each with a of weight 7/9; set B, which consists of 35 tasks each with an initial weight of 4/5 that decreases to 0 at time 2, with the corresponding capacity being freed at time 3; and set C, which consists of 35 tasks each with a weight 4/5 that all join the system at time 3. All tie-breaks (not resolved by by PD^2) go against tasks in set A. Since the first subtasks of tasks in both A and B have a group deadline at time 5 and all ties are broken against tasks in A, the tasks in B are scheduled in the slot 0. Also, since the capacity gained by decreasing the weight of tasks in set B to 0 has been freed before the group deadline of the tasks in B, the tasks in set C can join the system at time 3. As a result, a task in set C misses its deadline at time 3. If the capacity gained from decreasing the weight of tasks in set B had not been freed until time 5 (i.e., the group deadline of tasks in set B), then all deadline misses would have been avoided. $\square$

In light of this complication, we propose the following rule for reweighting heavy-changeable tasks.

Rule H: If T_i is heavy-changeable at time t_c from weight Ow to Nw and $T_i^{[j]}$ both exists and is the last-released subtask of T_i, then the following actions occur.

1. If $\mathsf{Ow} > \mathsf{Nw}$, then the capacity of $\mathsf{Ow} - \mathsf{Nw}$ is not freed until time $\mathsf{D}(T_i^{[j]})$.
2. If $T_i^{[j]}$ has been scheduled by time t_c, then at time $\mathsf{max}(t_c, \mathsf{d}(T_i^{[j]}) + \mathsf{b}(T_i^{[j]}))$, the weight change is enacted, T_i is reset, and $T_i^{[j]}$ is complete in the SW schedule (i.e., it stops receiving allocations); otherwise, at time t_c, $T_i^{[j]}$ is halted, and at time $\mathsf{max}(t_c, \mathsf{d}(T_i^{[j-1]}) + \mathsf{b}(T_i^{[j-1]}))$, T_i is reset, and the weight change is enacted (if $T_i^{[j-1]}$ does not exist, at time t_c T_i is reset and the change is enacted at t_c).
3. Any subtask $T_i^{[q]}$ released between t_c and time $\mathsf{D}(T_i^{[j]}) - 2$ has a window length of two, a b-bit of 1, a group deadline of $\mathsf{D}(T_i^{[j]})$, and a release time of
$$\mathsf{r}(T_i^{[q]}) = t_e + \left\lfloor \frac{q-1-j}{\mathsf{Nw}} \right\rfloor,$$
where t_e is the time the change was enacted.
4. If $T_i^{[\ell]}$ is the last subtask of T_i released before $\mathsf{D}(T_i^{[j]}) - 1$, then its successor is released and T_i is reset at time
$$\mathsf{r}(T_i^{[\ell+1]}) = \mathsf{max}\left(\mathsf{D}(T_i^{[j]}), t_e + \left\lfloor \frac{\ell - j}{\mathsf{Nw}} \right\rfloor\right),$$
where t_e is the time the change was enacted.

Rule H changes the release pattern of subtasks of T_i from the time the weight change is enacted until time $\mathsf{D}(T_i^{[j]}) - 1$ so that the allocation T_i receives over that range of time is commensurate with Nw. It is important to note that the term $\mathsf{r}(T_i^{[\ell+1]}) = \mathsf{max}\left(\mathsf{D}(T_i^{[j]}), t_e + \left\lfloor \frac{\ell-j}{\mathsf{Nw}} \right\rfloor + \theta(T_i^{[\ell+1]}) - \theta(T_i^{[\ell]})\right)$ in Part 4 is used to guarantee that no subtask of T_i is released at time $\mathsf{D}(T_i^{[j]}) - 1$. If such a subtask did exist, then it could cause a

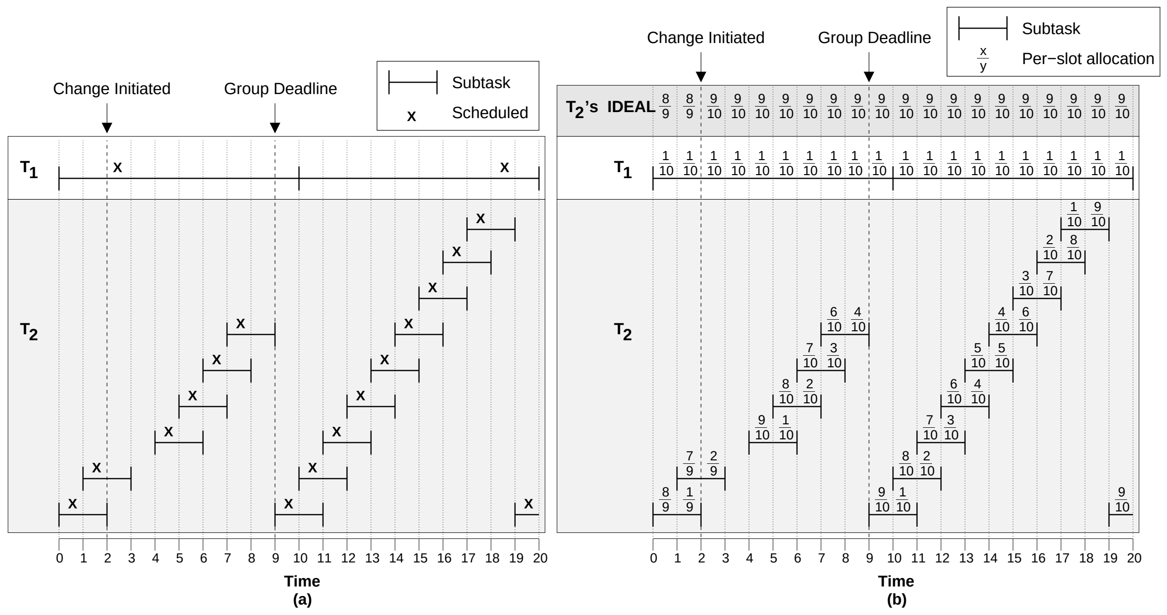


Figure 5.11: A one-processor system consisting of a task T_1, which has $\mathsf{wt}(T_1) = 1/10$, and T_2, which has an initial weight of 8/9 and initiates a weight increase to 9/10 at time 2. **(a)** The actual schedule. **(b)** The allocations in $\mathcal{SW}$ (and $\mathcal{CSW}$) and T_2's allocations in $\mathcal{I}$.

deadline to be missed. (Notice that Rule H can cause a T_i to be reset twice. Once when the change is enacted and a second time when T_i releases its first subtask at or after $\mathsf{D}(T_i^{[j]})$.)

Example (Figure 5.11). Consider the example in Figure 5.11(a), which depicts a one-processor system that is assigned two tasks: T_1, which has $\mathsf{wt}(T_1) = 1/10$, and T_2, which has an initial weight of 8/9 and initiates a weight increase to 9/10 at time 2. Since $T_2^{[2]}$ is the last-released subtask of T_2 before 2 and it has been scheduled by time 2, the change is enacted at time $\mathsf{d}(T_2^{[2]}) + \mathsf{b}(T_2^{[2]}) = 4$. Moreover, over the time range [4, 8), the subtasks of T_2 are released in a pattern that is commensurate with a task of weight 9/10, and $T_2^{[6]}$'s successor is released at $\mathsf{D}(T_2^{[2]}) = 9$. □

Example (Figure 5.12). Consider the example in Figure 5.12(a), which depicts a one-processor system that is assigned two tasks: T_1, which has an initial weight of 1/10 and initiates and enacts a weight increase to 2/3 at time 9, and T_2, which has an initial weight of 8/9 and initiates a weight decrease at time 2 to 1/3. Since $T_2^{[2]}$ has been scheduled by time 2, T_2's change is enacted at time 4. Notice that, over the time range [4, 8) the subtasks are released in a pattern that is commensurate with a task of weight 1/3, i.e., one subtask

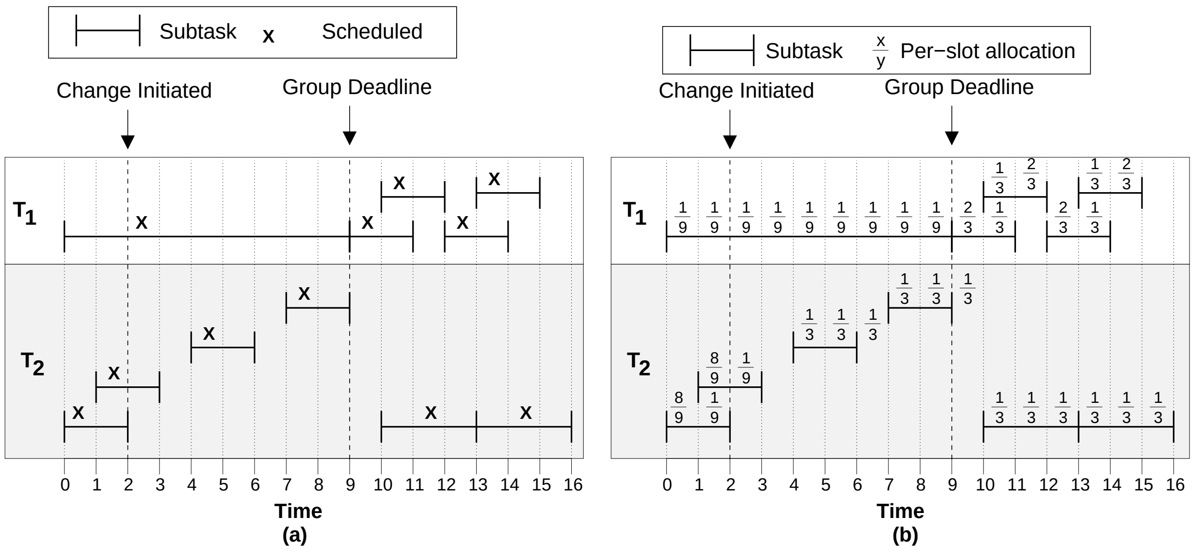


Figure 5.12: A one-processor system consisting of a task T_1, which has an initial weight of 1/10 and initiates a weight and enacts a weight increase to 2/3 at time 9, and T_2, which has an initial weight of 8/9 and initiates a weight decrease at time 2 to 1/3. **(a)** The actual schedule. **(b)** The SW (and CSW) schedule.

is released every 3 quanta, even though every subtask has a deadline two quanta after it is released. After time $\mathsf{D}(T_2^{[2]})$, i.e., time 9, T_2 behaves exactly like a "normal" task of weight 1/3. □

It is important to note that while the capacity freed from decreasing a heavy-changeable task T_i is not available to the rest of the system until the group deadline of its last-released subtask, it remains available for T_i.

Example (Figure 5.13). Consider the example in Figure 5.13, which depicts a one-processor system that is assigned two tasks: T_1, which has an initial weight of 1/14 and both initiates and enacts a weight increase to 1/4 at time 14, and T_2, which has an initial weight of 13/14, initiates a weight decrease to 1/3 at time 2, and initiates a weight increase to 3/4 at time 9. Notice that, while T_1 cannot increase its weight (by using the capacity gained from T_2 decreasing its weight at time 2) until time $\mathsf{D}(T_2^{[2]}) = 14$, T_2 can use this capacity to increase its weight from 1/3 to 3/4 at time 9. □

Because the weight of every task is in the range $(0, 1)$, heavy tasks (and those with a positive group deadline) have a window length of at most three, and because all tasks with a

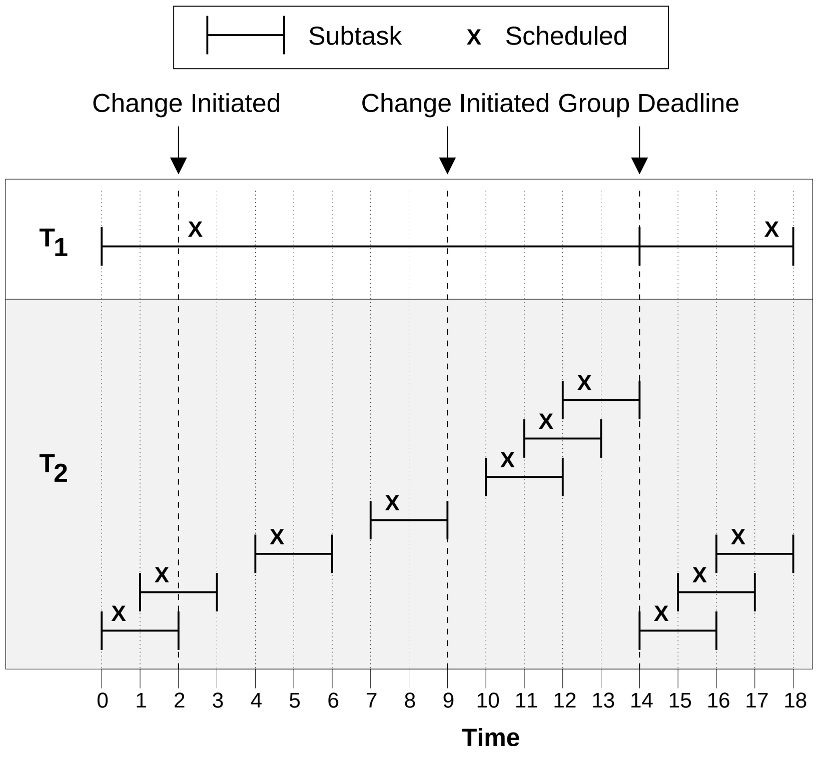


Figure 5.13: A one-processor system consisting of a task T_1, which has an initial weight of 1/14 and both initiates and enacts a weight increase to 1/4 at time 14, and T_2, which has an initial weight of 13/14, initiates a weight decrease to 1/3 at time 2, and initiates a weight increase to 3/4 at time 9.

group deadline of 0 are light tasks, it is not difficult to show that the following property holds

(WL) For any subtask $T_i^{[j]}$, if $\mathsf{D}(T_i^{[j]}) = 0$, then $\mathsf{d}(T_i^{[j]}) - \mathsf{r}(T_i^{[j]}) \geq 3$; otherwise, $2 \leq \mathsf{d}(T_i^{[j]}) - \mathsf{r}(T_i^{[j]}) \leq 3$.

Light and heavy tasks. Recall from Section 5.1.3 that for IS light tasks, if $\mathsf{D}(T_i^{[j]}) = 0$, then T_i is light, and if $\mathsf{D}(T_i^{[j]}) > 0$, then T_i is heavy. We can see that for AIS tasks, because of Rule H, the terms "light" and "heavy" are more difficult to distinguish. Specifically, it is possible for a task T_i to have a scheduling weight less than 1/2 and still release subtasks with a group deadline greater than zero. For this reason, for the remainder of this chapter, we do not distinguish between light and heavy tasks, but rather between tasks that release subtasks with a group deadline of zero and those that release subtasks with a group deadline greater than zero.

Before concluding this section, we make one final observation. When a heavy-changeable task decreases its weight, it is possible that a subtask will receive allocations in the SW schedule beyond its deadline. For example, in Figure 5.12(b), $T_2^{[3]}$ and $T_2^{[4]}$ both receive allocations in $\mathcal{S}W$ after their deadlines. This behavior may occur because Rule H "artificially" decreases the window length of all subtasks until the group deadline of the last-released subtask to two; however, this does not negatively impact the scheduling correctness or drift bounds of PD^2.

Throughout this chapter we use PD-PNH (respectively, PD-LJ) to refer to reweighting via Rules P, N, and H (respectively, the Rules L and J) under PD^2. Unless otherwise specified, throughout this chapter, we use $\mathcal{S}$ to denote the PD-PNH schedule of a system. Since these rules change the ordering of a task in the priority queues that determine scheduling, the time complexity for reweighting one task is $O(logN)$, where N is the number of tasks in the system (assuming that binomial heaps are used to implement needed priority queues).

5.5 Scheduling Correctness

In this section, we prove that, in PD-PNH-scheduled systems, deadlines are not missed. The properties used throughout this section are summarized in Table 5.2. We begin with a property pertaining to cancelled weight-change events.

(C) If T_i initiates two consecutive weight-change events at t_c and t'_c, where $t_c < t'_c < t_e$, and t_e denotes the time at which the change initiated at t_c would have been enacted in the absence of other reweighting events, then $t'_e \leq t_e$, where t'_e denotes the time at which the change initiated at t_c would have been enacted in the absence of other reweighting events.

Proof of (C). Assume that t_c, t_e, t'_c, and t'_e are as defined in Property (C). Notice that, if $\mathsf{d}(T_i^{[j]}) \leq t'_c$, where $T_i^{[j]}$ is the last-released subtask of T_i at t_c (as defined in Rules P, N, and H), then the change initiated at t'_c is enacted by $t'_c + 1$. Since $t'_c < t_e$, this implies that $t'_e \leq t_e$ holds. In the rest of the proof, we assume $t'_c < \mathsf{d}(T_i^{[j]})$ and consider the different types of reweighting events initiated at t_c.

Property	**Definition**
(C) [Page 199]	If T_i initiates two consecutive weight-change events at t_c and t'_c, where $t_c < t'_c < t_e$, and t_e denotes the time at which the change initiated at t_c would have been enacted in the absence of other reweighting events, then $t'_e \le t_e$, where t'_e denotes the time at which the change initiated at t_c would have been enacted in the absence of other reweighting events.
(X1) [Page 203]	If a task T_i initiates a weight change at time t_c and T_i releases a subtask $T_i^{[j]}$ at or after t_c, then the change initiated at t_c is enacted no later than time $\mathsf{r}(T_i^{[j]})$.
(X2) [Page 203]	If a weight change is enacted over the range $(\mathsf{r}(T_i^{[\ell]}), \mathsf{d}(T_i^{[\ell]})]$, then that change must have been initiated over the range $[\mathsf{r}(T_i^{[\ell]}), \mathsf{d}(T_i^{[\ell]})]$.
(W) [Page 203]	For any time t, $\sum_{T_i \in \tau} \mathsf{Swt}(T_i, t) \le M$, where M is the number of processors.
(V) [Page 203]	For any two subtasks, $T_i^{[j]}$ and $T_i^{[j+1]}$, if $\mathsf{r}(T_i^{[j+1]}) < \mathsf{d}(T_i^{[j]}) - \mathsf{b}(T_i^{[j]})$, then $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) \le \mathsf{r}(T_i^{[j+1]})$ and $\mathsf{C}(\mathcal{S}, T_i^{[j]}) \le \mathsf{r}(T_i^{[j+1]})$.
(RW) [Page 210]	Suppose T_i initiates a weight change at time $t_c \ge \mathsf{r}(T_i^{[1]})$ and $T_i^{[j]}$ is the last-released subtask of T_i at t_c. If $\mathsf{r}(T_i^{[j]}) \le t_c < \mathsf{d}(T_i^{[j]})$, then T_i is either positive-, negative-, or heavy-changeable at t_c.
(GV) [Page 211]	For the subtasks $T_i^{[j]}$ and $T_i^{[k]}$, where $j < k$, if $\mathsf{r}(T_i^{[k]}) < \mathsf{d}(T_i^{[j]}) - \mathsf{b}(T_i^{[j]})$, then $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) \le \mathsf{r}(T_i^{[k]})$ and $\mathsf{C}(\mathcal{S}, T_i^{[j]}) \le \mathsf{r}(T_i^{[k]})$.
(GD-1) [Page 182]	For any $T_i^{[j]}$ such that $\mathsf{D}(T_i^{[j]}) > 0$, if for all $T_i^{[q]} \in \{T_i^{[j]}, ..., \omega(T_i^{[j]})\}$, $\theta(T_i^{[q]}) = \theta(T_i^{[j]})$ (i.e., there are no IS separations between subtasks until the group deadline), then:**(i)** $\mathsf{r}(\omega(T_i^{[j]})) = \mathsf{D}(T_i^{[j]}) - 2$; and **(ii)** either $\mathsf{d}(\omega(T_i^{[j]})) = \mathsf{D}(T_i^{[j]})$ and $\mathsf{b}(\omega(T_i^{[j]})) = 0$ or $\mathsf{d}(\omega(T_i^{[j]})) = \mathsf{D}(T_i^{[j]}) + 1$ and $\mathsf{b}(\omega(T_i^{[j]})) = 1$.
(GD-2) [Page 182]	If $\mathsf{D}(T_i^{[j]}) > 0$, then $\mathsf{D}(T_i^{[j]}) \ge \mathsf{d}(T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$.
(GD-3) [Page 211]	For the subtasks $T_i^{[j]}$ and $T_i^{[k]}$, where $j < k$, if $\mathsf{r}(T_i^{[k]}) < \mathsf{d}(T_i^{[j]}) - \mathsf{b}(T_i^{[j]})$, then $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) \le \mathsf{r}(T_i^{[k]})$ and $\mathsf{C}(\mathcal{S}, T_i^{[j]}) \le \mathsf{r}(T_i^{[k]})$.
(AF1) [Page 211]	For all $t \ge 0$, $\mathsf{A}(\mathcal{S}W,\, T_i,\, t) \le \mathsf{Swt}(T_i, t)$.
(AF2) [Page 211]	For any present subtask $T_i^{[j]}$ of the task $T_i \in \tau$ and its successor $T_i^{[k]}$, if $\mathsf{b}(T_i^{[j]}) = 1$ and $\mathsf{r}(T_i^{[k]}) \ge \mathsf{C}(\mathcal{S}W, T_i^{[j]})$, then $\mathsf{A}(\mathcal{S}W,\, T_i,\, \mathsf{C}(\mathcal{S}W, T_i^{[j]}) - 1,\, \mathsf{C}(\mathcal{S}W, T_i^{[j]}) + 1) \le \mathsf{Swt}(T_i, \mathsf{C}(\mathcal{S}W, T_i^{[j]}))$.
(AF3) [Page 211]	For any subtask $T_i^{[j]}$, $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) \le \mathsf{d}(T_i^{[j]})$.
(AF4) [Page 212]	For any subtask $T_i^{[j]}$ and any time t, if $t < \mathsf{r}(T_i^{[j]})$ or $t \ge \mathsf{C}(\mathcal{S}W, T_i^{[j]})$, then $\mathsf{A}(\mathcal{S}W,\, T_i^{[j]},\, t) = 0$.
(AF5) [Page 212]	For any present subtask $T_i^{[j]}$, let $T_i^{[k]}$ be its next present successor $T_i^{[k]}$ (if it exists). If $T_i^{[k]}$ does not exist, and $\mathsf{D}(T_i^{[j]}) > 0$, then for any t such that $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) - 1 \le t \le \mathsf{D}(T_i^{[j]})$, $\mathsf{A}(\mathcal{S}W,\, T_i,\, \mathsf{C}(\mathcal{S}W, T_i^{[j]}) - 1,\, t+1) \le \mathsf{Swt}(T_i, t)$ holds. Otherwise, if $T_i^{[k]}$ exists, $\mathsf{D}(T_i^{[j]}) > 0$, and $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) \le \mathsf{r}(T_i^{[k]})$, then for any t such that $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) - 1 \le t \le \mathsf{min}(\mathsf{r}(T_i^{[k]}), \mathsf{D}(T_i^{[j]}) - 1)$, $\mathsf{A}(\mathcal{S}W,\, T_i,\, \mathsf{C}(\mathcal{S}W, T_i^{[j]}) - 1,\, t+1) \le \mathsf{Swt}(T_i, t)$ holds.
(T1) [Page 213]	τ misses a deadline under PD-PNH at t_d.
(T2) [Page 214]	No task system satisfying (T1) has fewer present subtasks in $[0, t_d)$ than τ.

Table 5.2: Summary of properties used in Section 5.5.

Positive-changeable. We begin by considering the case wherein T_i is positive-changeable at t_c. In this case, the change initiated at t_c is enacted at time $t_e = \mathsf{max}(t_c, \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]}))$. If $t_e = t_c$, then since $t_c < t'_c < t_e$, t'_c cannot exist. Thus, we assume $t_c < t_e$, which implies $t_e = \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]}) > t_c$. Since T_i is positive-changeable at t_c, $T_i^{[j]}$ is halted at t_c and no successor subtask can be released until the change initiated at t_c (or a future change) has been enacted. Hence, since $t'_c < t_e$, $T_i^{[j+1]}$ is not released until at or after t'_c. Thus, since $t_c < t'_c$, $T_i^{[j]}$ is the last-released subtask of T_i at t'_c. Because, by Rule P, $T_i^{[j]}$ was halted at $t_c < t'_c$ and (as we assumed at the beginning of the proof) $t'_c < \mathsf{d}(T_i^{[j]})$, T_i is therefore positive-changeable at t'_c. Thus, by Rule P, the change initiated at t'_c is enacted at time $t'_e = \mathsf{max}(t'_c, \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]}))$. Again, since $t'_c < t_e$ and $t_e = \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]})$, it follows that $t'_e = \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]}) = t_e$.

Decreasing-weight negative changeable. We next consider the case wherein T_i is decreasing-weight negative-changeable at t_c. By Rule N, such a change initiated at t_c will be enacted at time $t_e = \mathsf{C}(\mathcal{S}W, T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$. Since T_i is negative-changeable at t_c, no subtask can be released until the change that was initiated at t_c (or a future change) has been enacted. Hence, since $t'_c < t_e$, $T_i^{[j+1]}$ is not released until at or after t'_c. Thus, since $t_c < t'_c$, $T_i^{[j]}$ is the last-released subtask of T_i at t'_c. Because $T_i^{[j]}$ is scheduled before $t_c < t'_c$, and (as we assumed at the beginning of the proof) $t'_c < \mathsf{d}(T_i^{[j]})$, T_i is negative-changeable at t'_c. If the change at t'_c is a decreasing-weight event, then by Rule N, it is enacted at time $t'_e = \mathsf{C}(\mathcal{S}W, T_i^{[j]}) + \mathsf{b}(T_i^{[j]}) = t_e$.

Increasing-weight negative-changeable. Notice that, if T_i at t_c is increasing-weight negative-changeable, then the change initiated at t_c is enacted immediately, i.e., $t_c = t_e$. Thus, since $t_c < t'_c < t_e$, if T_i at t_c is increasing-weight negative-changeable, then t'_c cannot exist.

Heavy-changeable. Notice that, if T_i is heavy-changeable at t_c, then $\mathsf{D}(T_i^{[j]}) > t_c$ holds, and, by Part 2 of Rule H, if $T_i^{[j]}$ has been scheduled by t_c, then $t_e = \mathsf{max}(t_c, \mathsf{d}(T_i^{[j]}) + \mathsf{b}(T_i^{[j]}))$, and if $T_i^{[j]}$ has *not* been scheduled by t_c, then $t_e = \mathsf{max}(t_c, \mathsf{d}(T_i^{[j-1]}) + \mathsf{b}(T_i^{[j-1]}))$ (or $t_e = t_c$ if $T_i^{[j-1]}$ does not exist). If $t_c = t_e$, then since $t_c < t'_c < t_e$, t'_c cannot exist. Thus, assume that $t_c < t_e$.

We first consider the case where $T_i^{[j]}$ has been scheduled before t_c. In this case, since we have assumed that $t_c < t_e$, it follows that $t_e = \mathsf{d}(T_i^{[j]}) + \mathsf{b}(T_i^{[j]}) > t_c$. Thus, in this case,

$$t'_c \in (t_c, \mathsf{d}(T_i^{[j]}) + \mathsf{b}(T_i^{[j]})), \tag{5.13}$$

and (by Part 3 of Rule H), $\mathsf{r}(T_i^{[j+1]}) \geq t_e$. Thus, it follows that at $t'_c > t_c$, $T_i^{[j]}$ is the last-released subtask of T_i.

By Property (GD-2), $\mathsf{D}(T_i^{[j]}) \geq \mathsf{d}(T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$. Thus, since (as we assumed at the beginning of the proof) $t'_c < \mathsf{d}(T_i^{[j]})$, it follows that $t'_c < \mathsf{d}(T_i^{[j]}) \leq \mathsf{D}(T_i^{[j]})$. Thus, since $T_i^{[j]}$ is the last-released subtask of T_i at t'_c and $t'_c < \mathsf{D}(T_i^{[j]})$, T_i is heavy-changeable at t'_c. Hence, by Step 2 of Rule H, $t'_e = \mathsf{max}(\mathsf{d}(T_i^{[j]}) + \mathsf{b}(T_i^{[j]}), t'_c)$. Since, by (5.13), $t'_c < \mathsf{d}(T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$, it follows that $t'_e = \mathsf{d}(T_i^{[j]}) + \mathsf{b}(T_i^{[j]}) = t_e$.

We now consider the case where $T_i^{[j]}$ has not been scheduled before t_c. Since we have assumed that $t_c < t_e$, if $T_i^{[j]}$ has not been scheduled by t_c, then $t_c < t_e = \mathsf{d}(T_i^{[j-1]}) + \mathsf{b}(T_i^{[j-1]})$. Thus, in this scenario,

$$t'_c \in (t_c, \mathsf{d}(T_i^{[j-1]}) + \mathsf{b}(T_i^{[j-1]})), \tag{5.14}$$

and (by Step 3 of Rule H), $\mathsf{r}(T_i^{[j+1]}) \geq t_e = \mathsf{d}(T_i^{[j-1]}) + \mathsf{b}(T_i^{[j-1]})$. Thus, it follows that at $t'_c > t_c$, $T_i^{[j]}$ is the last-released subtask of T_i.

By Property (GD-2), $\mathsf{D}(T_i^{[j]}) \geq \mathsf{d}(T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$. Thus, since (as we assumed at the beginning of the proof) $t'_c < \mathsf{d}(T_i^{[j]})$, it follows that $t'_c < \mathsf{D}(T_i^{[j]})$. Thus, since $T_i^{[j]}$ is the last-released subtask of T_i at t'_c and $t'_c < \mathsf{D}(T_i^{[j]})$, T_i is heavy-changeable at t'_c. Moreover, since $T_i^{[j]}$ had not been scheduled at t_c, it follows by Part 2 of Rule H, that $T_i^{[j]}$ was halted at t_c. Thus, $T_i^{[j]}$ is not scheduled by $t'_c > t_c$. Thus, by Part 2 of Rule H, $t'_e = \mathsf{max}(\mathsf{d}(T_i^{[j-1]}) + \mathsf{b}(T_i^{[j-1]}), t'_c)$. Since, by (5.14), $t'_c < \mathsf{d}(T_i^{[j-1]}) + \mathsf{b}(T_i^{[j-1]})$, it follows that $t'_e = \mathsf{d}(T_i^{[j-1]}) + \mathsf{b}(T_i^{[j-1]}) = t_e$. $\square$

Initiation and enactment properties. We now state two properties about the relationship between the initiation and enactment of reweighting events.

(X1) If a task T_i initiates a weight change at time t_c and T_i releases a subtask $T_i^{[j]}$ at or after t_c, then the change initiated at t_c is enacted no later than time $\mathsf{r}(T_i^{[j]})$.

(X2) If a weight change is enacted over the range $(\mathsf{r}(T_i^{[\ell]}), \mathsf{d}(T_i^{[\ell]})]$, then that change must have been initiated over the range $[\mathsf{r}(T_i^{[\ell]}), \mathsf{d}(T_i^{[\ell]})]$.

Given that Property (C) guarantees that no sequence of reweighting events can delay the next weight-change enactment after a weight change has been initiated, and that the Rules N, P, and H guarantee a subtask is released within one quantum of a weight change enactment, the subtask $T_i^{[j]}$ (as defined in Property (X1)) will eventually be released, if it exists. Thus, from the definitions of Rules P, N, and H, Property (X1) should be fairly intuitive. Property (X2) is implied by Property (X1).

When Srinivasan and Anderson (Srinivasan and Anderson, 2005) proved the scheduling correctness of PD-LJ for an IS task system, they assumed that the weight of all tasks is at most M and utilized the property that in an IS task system the windows for any subtask $T_i^{[j]}$ and its successor $T_i^{[j+1]}$ do not "overlap" by more than $\mathsf{b}(T_i^{[j]})$ quanta, i.e., $\mathsf{d}(T_i^{[j]}) - \mathsf{b}(T_i^{[j]}) \leq \mathsf{r}(T_i^{[j+1]})$. However, this property can be weakened without affecting most of their proof, so that their proof can be applied to an AIS task system. Specifically, their proof can be used to establish the scheduling correctness of PD-PNH for any AIS task system τ, if the following conditions hold, which parallel the assumption that the weight of all tasks is at most M and the property that in IS system $\mathsf{d}(T_i^{[j]}) - \mathsf{b}(T_i^{[j]}) \leq \mathsf{r}(T_i^{[j+1]})$. (In these properties, we denote the PD-PNH schedule of τ as $\mathcal{S}$.)

(W) For any time t, $\sum_{T_i \in \tau} \mathsf{Swt}(T_i, t) \leq M$, where M is the number of processors.

(V) For any two subtasks, $T_i^{[j]}$ and $T_i^{[j+1]}$, if $\mathsf{r}(T_i^{[j+1]}) < \mathsf{d}(T_i^{[j]}) - \mathsf{b}(T_i^{[j]})$, then $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) \leq \mathsf{r}(T_i^{[j+1]})$ and $\mathsf{C}(\mathcal{S}, T_i^{[j]}) \leq \mathsf{r}(T_i^{[j+1]})$.

Since Property (W) can be satisfied by policing weight-change requests, we focus our attention on showing that $\mathcal{S}$ and $\mathcal{S}W$ satisfy Property (V).

Example (Figures 5.8 and 5.9). In Figure 5.8(b), $\mathsf{d}(T_1^{[2]}) + \mathsf{b}(T_1^{[2]}) = 15$ and $\mathsf{r}(T_1^{[3]}) = 10$. Notice that, by Rule P, $T_1^{[2]}$ is halted (and hence complete) in both the SW and PD^2 schedules when the change is enacted at time $10 = \mathsf{r}(T_1^{[3]})$. In Figure 5.9(a), $\mathsf{d}(T_1^{[2]}) + \mathsf{b}(T_1^{[2]}) = 15$ and $\mathsf{r}(T_1^{[3]}) = 12$. Notice that, $T_1^{[2]}$ has been scheduled (and is therefore complete) before time $10 < \mathsf{r}(T_1^{[3]})$. Further, $T_1^{[3]}$ is released once $T_1^{[2]}$ has received one unit of allocation in the SW schedule, and as a result $T_1^{[2]}$ is complete in the SW schedule by $\mathsf{r}(T_1^{[3]})$. $\square$

Proof of (V). Referring to Property (V), let $T_i^{[j]}$ be some subtask such that $\mathsf{d}(T_i^{[j]}) - \mathsf{b}(T_i^{[j]}) > \mathsf{r}(T_i^{[j+1]})$. By the definition of a subtask release, if $\mathsf{d}(T_i^{[j]}) - \mathsf{b}(T_i^{[j]}) > \mathsf{r}(T_i^{[j+1]})$, then T_i enacted a weight change $t_e \in (\mathsf{r}(T_i^{[j]}), \mathsf{r}(T_i^{[j+1]})]$; otherwise, we would have $\mathsf{d}(T_i^{[j]}) - \mathsf{b}(T_i^{[j]}) \leq \mathsf{r}(T_i^{[j+1]})$, by (5.11). Since a weight change is enacted in the range $(\mathsf{r}(T_i^{[j]}), \mathsf{r}(T_i^{[j+1]})]$, by Property (X2) a change must have been initiated in the range $[\mathsf{r}(T_i^{[j]}), \mathsf{r}(T_i^{[j+1]})]$. Without loss of generality, let t_c be the earliest time in this range that T_i initiates a weight change.

At time t_c, T_i is positive-, negative-, or heavy-changeable. If at t_c, T_i is positive-changeable, then by Rule P, $T_i^{[j]}$ is complete by $t_c \leq t_e \leq \mathsf{r}(T_i^{[j+1]})$ in both $\mathcal{S}$ and $\mathcal{SW}$. If T_i is negative-changeable at t_c, then $T_i^{[j]}$ has been scheduled in $\mathcal{S}$ before t_c, and hence, $T_i^{[j]}$ is complete by $t_c \leq \mathsf{r}(T_i^{j+1})$ in $\mathcal{S}$. Furthermore, in this case $T_i^{[j+1]}$ is not released until time $\mathsf{C}(\mathcal{SW}, T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$. If at t_c, T_i is heavy-changeable and $T_i^{[j]}$ has not been scheduled by t_c, then by Rule H, $T_i^{[j]}$ is complete by $t_c \leq t_e \leq \mathsf{r}(T_i^{[j+1]})$ in both $\mathcal{S}$ and $\mathcal{SW}$. If at t_c, T_i is heavy-changeable and $T_i^{[j]}$ has been scheduled by t_c, then by Rule H, $T_i^{[j]}$ is complete by $t_e \leq \mathsf{r}(T_i^{[j+1]})$ in both $\mathcal{S}$ and $\mathcal{SW}$. $\square$

We now prove that PD-PNH correctly schedules any AIS task system that satisfies Property (W). Note that, this proof is only a slight modification of the correctness proof for PD-LJ originally presented by Srinivasan and Anderson in (Srinivasan and Anderson, 2005).

Before proving that PD-PNH correctly schedules any AIS task system, we introduce some basic concepts and properties that are useful in the proof. We begin by introducing the "adaptive generalized intra-sporadic" task model. After this, we introduce the notion of a "displacement." Lastly, we introduce some properties and definitions pertaining to PD-PNH.

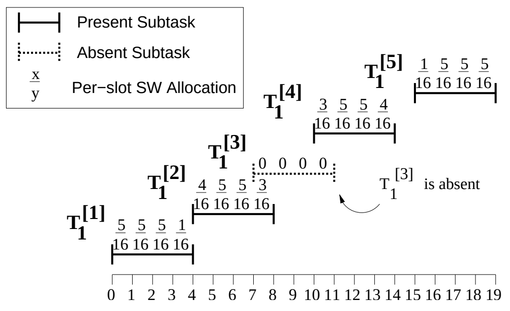


Figure 5.14: An illustration of the AGIS task model. The dashed window lines indicate that a subtask is absent.

5.5.1 The AGIS Task Model

To prove the scheduling correctness of PD-PNH in an AIS system, we consider an extension of the AIS task model called the *adaptive generalized intra-sporadic* (AGIS) *task model.* The AGIS model generalizes the AIS model by allowing some subtasks to be "absent." An *absent* subtask is never scheduled; however, such subtasks are considered to be part of a given task system, and as such, they have both releases and deadlines. If a subtask is not absent then we say that it is *present.* In an AGIS task system, $T_i^{[j]}$ is $T_i^{[k]}$'s *predecessor* (and $T_i^{[k]}$ is $T_i^{[j]}$'s *successor*) iff $T_i^{[j]}$ and $T_i^{[k]}$ are both present and there are no present subtasks that have an index between j and k. The per-slot allocations to a subtask $T_i^{[j]}$ in the AGIS variant of an $\mathcal{S}W$ schedule are the same as in the AIS variant, except that if a subtask is absent, then its per-slot allocation is zero in all slots.

Example (Figure 5.14). Consider the example depicted in Figure 5.14, which consists of one task, T_1, which has $\mathsf{wt}(T_1) = 5/16$, and has one absent subtask, $T_1^{[3]}$. $T_1^{[2]}$ is $T_1^{[4]}$'s predecessor, and $T_1^{[4]}$ is $T_1^{[2]}$'s successor. The per-slot allocations to all subtasks except $T_1^{[3]}$ are the same as in an AIS system, and $T_1^{[3]}$'s per-slot allocation is zero for each slot. □

Example (Figure 5.15). As a second example, consider Figure 5.15, which consists of one task, T_1, which has $\mathsf{wt}(T_1) = 7/9$. In inset (a), no tasks are absent. In inset (b), $T_1^{[2]}$, $T_1^{[3]}$, and $T_1^{[5]}$ are absent. □

Throughout this section, we use $\mathsf{LAG}(\tau, t)$ to denote $\mathsf{LAG}(\mathcal{S}, \mathcal{S}W, \tau, t)$ and $\mathsf{lag}(T_i, t)$ to denote $\mathsf{lag}(\mathcal{S}, \mathcal{S}W, T_i, t)$, where $\mathcal{S}$ is the PD-PNH schedule of a task system τ.

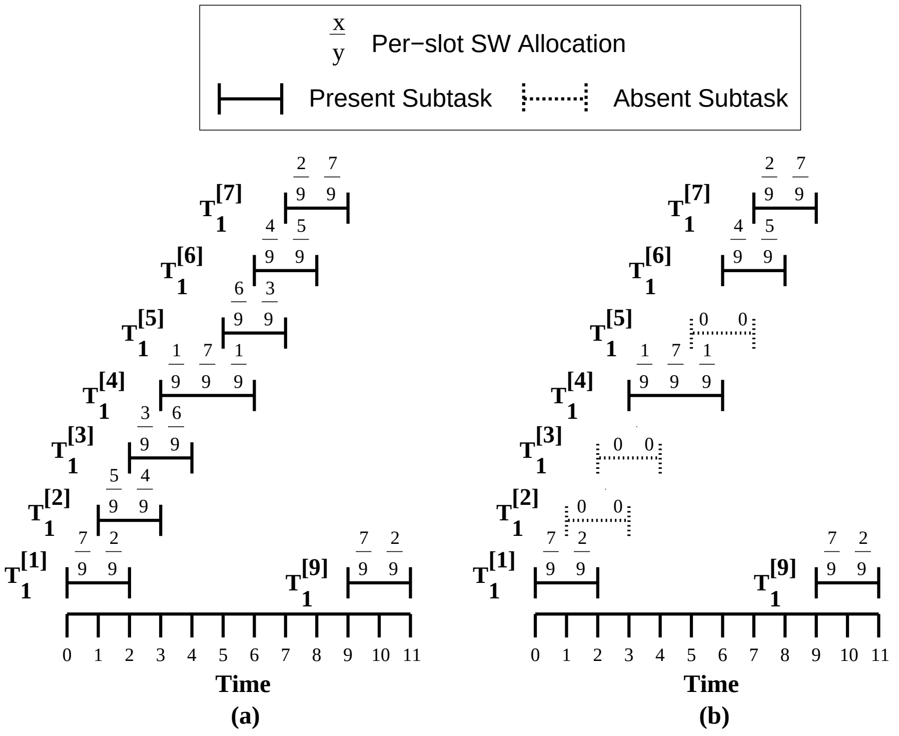


Figure 5.15: $\mathsf{A}(\mathcal{S}W,\ T_i^{[j]},\ t)$ for a periodic task with weight 7/9. **(a)** No subtasks are absent. **(b)** $T_1^{[2]}$, $T_1^{[3]}$, and $T_1^{[5]}$ are absent.

Completed. Since, under the AGIS task model, absent tasks never receive any allocations, by the definition of completed, presented earlier, such a subtask would never complete unless it was halted. Therefore, we amend the definition of completed so that an absent subtask $T_i^{[j]}$ is considered to be complete in all schedules as soon as it is released, i.e., $\mathsf{C}(\mathcal{S}, T_i^{[j]}) = \mathsf{C}(\mathcal{S}W, T_i^{[j]}) = \mathsf{r}(T_i^{[j]})$. For example, in Figure 5.14, $\mathsf{C}(\mathcal{S}W, T_1^{[3]}) = \mathsf{r}(T_1^{[3]}) = 7$.

Absent subtasks and reweighting. Notice that, if a task T_q initiates a weight change at time t_c, and the last-released subtask of T_q, $T_q^{[k]}$, is absent, then that subtask has not yet been scheduled. Therefore, if $t_c < \mathsf{d}(T_q^{[k]})$ and T_q is not heavy-changeable at t_c, then T_q is positive-changeable at t_c. Similarly, if T_q is heavy-changeable at t_c, then $T_q^{[k]}$ is "halted" and its successor (which may also be absent) is released at the appropriate time. In such cases, an absent subtask is considered to be "halted," even though it was never eligible to be scheduled.

Example (Figure 5.16). Consider the example in Figure 5.16, which depicts the impact of the reweighting rules on an AGIS task T_1, which changes its weight from 3/19 to 2/5. In

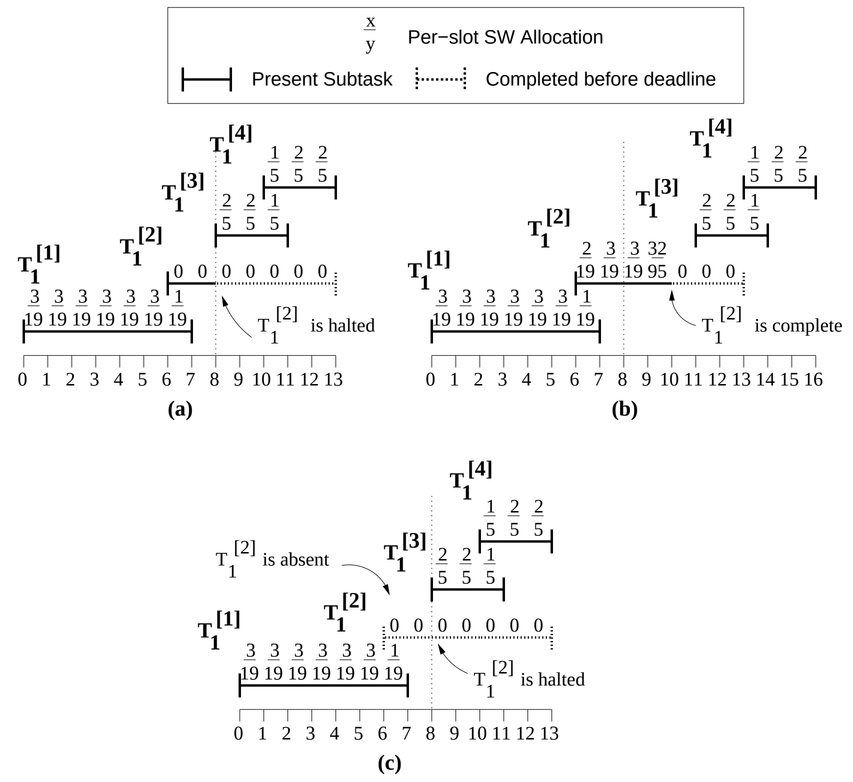


Figure 5.16: The impact of reweighting on an AGIS task. **(a)** T_1 changes its weight via Rule P. **(b)** T_1 changes its weight via Rule N. **(c)** $T_1^{[2]}$ is absent when T_1 is changed causing it to change via Rule P.

inset (a), T_2 changes its weight via Rule P causing $T_1^{[2]}$ to be halted at time 8. In inset (b), T_1 changes its weight via Rule N causing $T_1^{[2]}$ to be complete at time 10. In inset (c), $T_1^{[2]}$ is absent, and so T_1 is positive-changeable at time 8. Notice that, in inset (a), the subtask $T_1^{[2]}$ is halted at time 8, so $\mathsf{C}(\mathcal{S}W, T_1^{[2]}) = \mathsf{r}(T_1^{[2]}) = 8$. In contrast, in inset (b), the subtask $T_i^{[2]}$, is never halted, and therefore $\mathsf{C}(\mathcal{S}W, T_1^{[2]}) = 10$. Also note that, in inset (c), when T_1 changes its weight via Rule P at time 8, $T_1^{[2]}$ is considered to be halted, even though it is absent. □

5.5.2 Displacements

We now introduce the notions of an "instance" of a task system and a task "displacement." An *instance* of a task system is obtained by specifying a unique assignment of release times for each subtask and weight changes for each task.

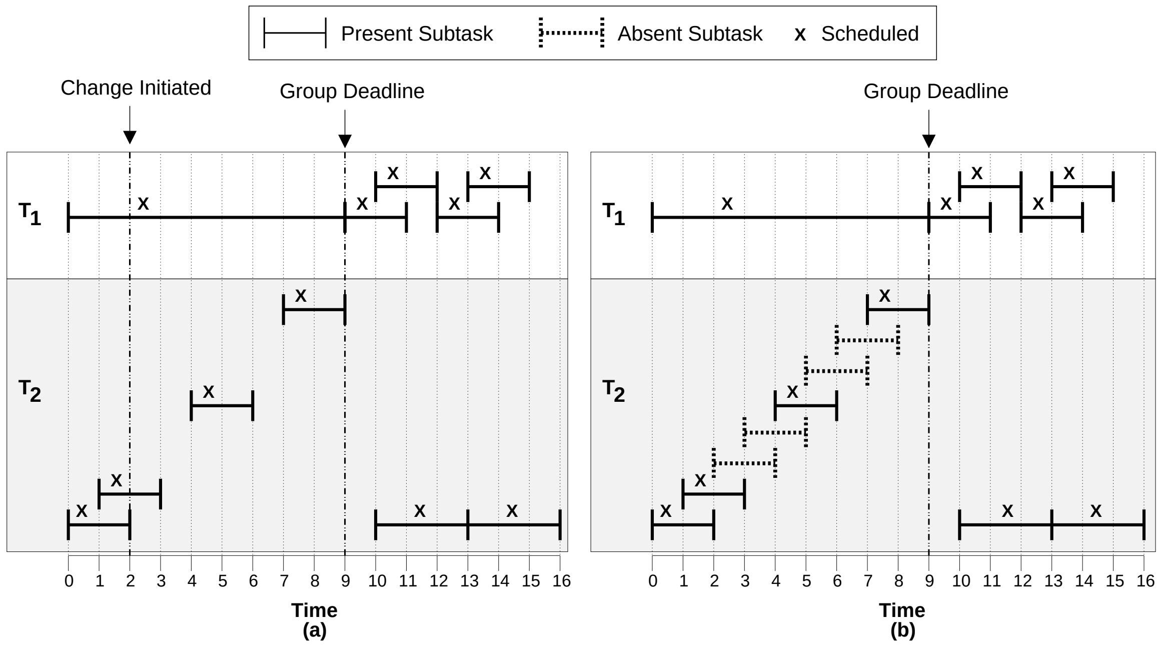


Figure 5.17: A one-processor system consisting of a task T_1 with an initial weight of 1/9 that increases its weight at time 9 to 2/3, and a task T_2 that has an initial weight of 8/9 that decreases to 1/3. **(a)** T_2's weight change is initiated at time 2. **(b)** Subtasks $T_2^{[3]}$, $T_2^{[4]}$, $T_2^{[6]}$, $T_2^{[7]}$ are absent and T_2's weight change is initiated at time 9.

Equivalent instances. Before continuing, it is worth pointing out that since halted subtasks are never scheduled, and Rules P, N, and H behave the same whether the last-released subtask is absent or not, PD-PNH produces the same schedule regardless of whether a halted subtask is absent or present. Thus, we assume that in every task instance presented in the remainder of this section, if a subtask is halted, then it is absent.

Similarly, notice that, Rule H can be emulated by delaying the initiation of a weight change until the group deadline of the last-released subtask and selectively choosing some subtasks to be absent.

Example (Figure 5.17). Consider the example in Figure 5.17, which depicts a one-processor system that contains two tasks: T_1, which has an initial weight of 1/9 that increases to 2/3 at time 9; and T_2, which has an initial weight of 8/9 that decreases to 1/3. In inset (a), T_2 initiates its weight change at time 2. In inset (b), subtasks $T_2^{[3]}$, $T_2^{[4]}$, $T_2^{[6]}$, $T_2^{[7]}$ are absent, and T_2 initiates its weight change at time $\mathsf{D}(T_2^{[2]}) = 9$. Notice that, these two systems have *exactly* the same schedule. □

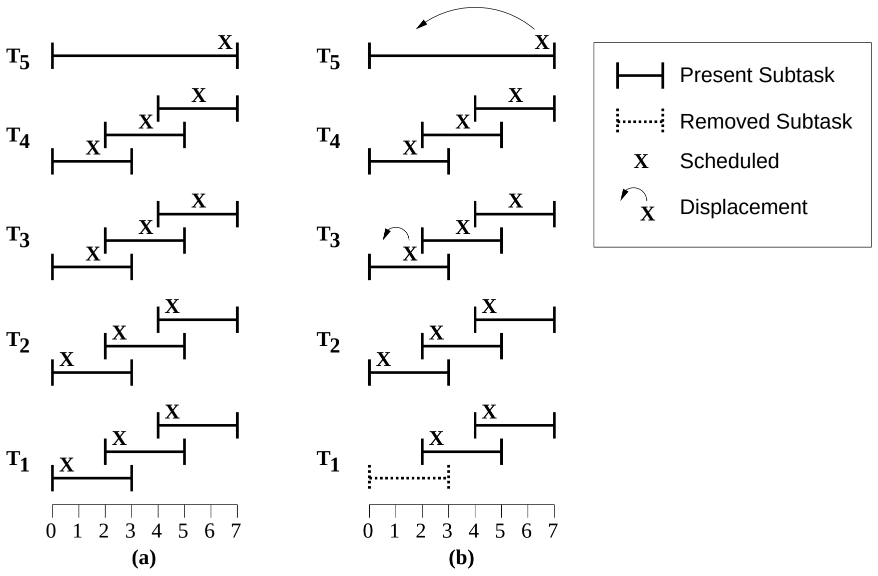


Figure 5.18: An illustration of displacements. **(a)** The original system. **(b)** The system with $T_1^{[1]}$ removed.

Given the equivalence demonstrated above, in every AGIS task instance considered in the remainder of this section, we assume that a heavy task only initiates a weight change at or after the group deadline of the last-released subtask. (This assumption removes the possibility that a subtask of a heavy-changeable task could receive allocations in the SW schedule after its deadline.)

Definition 5.3 (**Removal and Displacements (Srinivasan, 2003)).** By definition, the *removal* of a subtask (i.e., changing a subtask from present to absent) from one instance of an AGIS task system results in another instance. (Note that, only present subtasks can be removed.) Let $\mathsf{X}^{(j)}$ denote a subtask of any task in an AGIS task system τ. Let $\mathcal{S}$ denote the $\mathsf{PD\text{-}PNH}$ schedule of τ. Assume that removing $\mathsf{X}^{(1)}$ scheduled at slot t_1 in $\mathcal{S}$ causes $\mathsf{X}^{(2)}$ to shift from slot t_2 to t_1, where $t_1 \neq t_2$, which in turn may cause other shifts. We call this shift a *displacement* and represent it by the four-tuple $\langle \mathsf{X}^{(1)}, t_1, \mathsf{X}^{(2)}, t_2 \rangle$. A displacement $\langle \mathsf{X}^{(1)}, t_1, \mathsf{X}^{(2)}, t_2 \rangle$ is *valid* iff $\mathsf{r}(\mathsf{X}^{(2)}) \leq t_1$. Because there can be a cascade of shifts, we may have a *chain* of displacements.

Example (Figure 5.18). Consider the two-processor example in Figure 5.18, which depicts the schedule for a system with four task T_1, ..., T_4, each of which has a weight of 3/7, and T_5, which has $\mathsf{wt}(T_5) = 1/7$. Inset (a) depicts the system in which all tasks are present. Inset (b) depicts the system where $T_1^{[1]}$ has been removed causing a chain of displacements. □

Removing a subtask may also lead to slots in which some processors are idle. In a schedule $\mathcal{S}$, if k processors are idle in slot t, then we say that there are k *holes* in $\mathcal{S}$ in slot t. Note that, holes may exist because of late subtask releases or absent subtasks, even if total utilization is M. We now present three lemmas that describe the relationship among subtasks in a chain of displacements. These three lemmas were originally proven by Srinivasan and Anderson for the "generalized IS task model," i.e., an IS task system, where subtasks can be absent (Srinivasan and Anderson, 2006). Since the logic of their proof holds for AGIS task systems, we state these lemmas without proof.

Lemma 5.1 (From (Srinivasan, 2003)). *Let $\mathsf{X}^{(1)}$ be a subtask that is removed from τ, where all halted subtasks are absent, and let the resulting chain of displacements in $\mathcal{S}$ be $C = \Delta_1, \Delta_2, ..., \Delta_k$, where $\Delta_j = \langle \mathsf{X}^{(j)}, t_j, \mathsf{X}^{(j+1)}, t_{j+1} \rangle$. Then $t_{j+1} > t_j$, for all $j \in \{1, ..., k\}$*

Lemma 5.2 (From (Srinivasan, 2003)). *Let $\Delta = \langle \mathsf{X}^{(1)}, t_1, \mathsf{X}^{(2)}, t_2 \rangle$ be a valid displacement in $\mathcal{S}$, in which all halted subtasks are absent. If $t_1 < t_2$ and there is a hole in slot t_1 in that schedule, then $\mathsf{X}^{(2)}$ is $\mathsf{X}^{(1)}$'s successor in τ.*

Lemma 5.3 (From (Srinivasan, 2003)). *Let $\Delta = \langle \mathsf{X}^{(1)}, t_1, \mathsf{X}^{(2)}, t_2 \rangle$ be a valid displacement in $\mathcal{S}$, in which all halted subtasks are absent. If $t_1 < t_2$ and there is a hole in slot t' such that $t_1 \le t' < t_2$ in that schedule, then $t' = t_1$ and $\mathsf{X}^{(2)}$ is the successor of $\mathsf{X}^{(1)}$ in τ.*

5.5.3 Reweighting Properties

In this section, we introduce some properties that are necessary to prove scheduling correctness.

Basic properties. The following simple property follows directly from the definition of PD-PNH.

(RW) Suppose T_i initiates a weight change at time $t_c \ge \mathsf{r}(T_i^{[1]})$ and $T_i^{[j]}$ is the last-released subtask of T_i at t_c. If $\mathsf{r}(T_i^{[j]}) \le t_c < \mathsf{d}(T_i^{[j]})$, then T_i is either positive-, negative-, or heavy-changeable at t_c.

In Section 5.5, we presented Property (V), which relates the release times, completion times, and deadlines of two subtasks $T_i^{[j]}$ and $T_i^{[j+1]}$. In the following proof, it is useful to extend this property to relate the release times, completion times, and deadlines of two subtasks $T_i^{[j]}$ and $T_i^{[k]}$, where $j < k$.

(GV) For the subtasks $T_i^{[j]}$ and $T_i^{[k]}$, where $j < k$, if $\mathsf{r}(T_i^{[k]}) < \mathsf{d}(T_i^{[j]}) - \mathsf{b}(T_i^{[j]})$, then $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) \leq \mathsf{r}(T_i^{[k]})$ and $\mathsf{C}(\mathcal{S}, T_i^{[j]}) \leq \mathsf{r}(T_i^{[k]})$.

The proof of Property (GV) follows directly from Property (V). Property (GV) holds even if $T_i^{[j]}$ or $T_i^{[j]}$ is absent.

Group deadlines of light tasks. Notice that, if a present subtask $T_i^{[j]}$ has a group-deadline of 0, then it must be the case that $\mathsf{Swt}(T_i, \mathsf{r}(T_i^{[j]})) < 1/2$. Thus, unless T_i enacts a weight change by time $\mathsf{r}(T_i^{[k]})$, where $T_i^{[k]}$ is $T_i^{[j]}$'s successor, then $\mathsf{D}(T_i^{[k]}) = 0$ must hold. Moreover, by Rules P and N, if $T_i^{[j]}$ is scheduled in slot $\mathsf{d}(T_i^{[j]}) - 1$, $\mathsf{b}(T_i^{[j]}) = 1$, and $\mathsf{r}(T_i^{[k]}) \leq \mathsf{d}(T_i^{[j]}) - 1$, then it is not possible for T_i to enact a weight change before $T_i^{[k]}$ is released. Thus, we have the following property

(GD-3) For any subtask $T_i^{[j]}$, and its successor $T_i^{[k]}$, if $\mathsf{D}(T_i^{[j]}) = 0$, $\mathsf{b}(T_i^{[j]}) = 1$, $T_i^{[j]}$ is scheduled in slot $\mathsf{d}(T_i^{[j]}) - 1$, and $\mathsf{r}(T_i^{[k]}) \leq \mathsf{d}(T_i^{[j]}) - 1$, then $\mathsf{D}(T_i^{[k]}) = 0$.

Per-slot allocation properties. We now introduce five properties about the per-slot allocations of a task and the completion time of a subtask in an AGIS system τ that are useful in the correctness proof. (Recall that we assume that in τ all halted subtasks are absent and no heavy task initiates a weight change before the group deadline of its last-released subtask.)

(AF1) For all $t \geq 0$, $\mathsf{A}(\mathcal{S}W,\, T_i,\, t) \leq \mathsf{Swt}(T_i, t)$.

(AF2) For any present subtask $T_i^{[j]}$ of the task $T_i \in \tau$ and its successor $T_i^{[k]}$, if $\mathsf{b}(T_i^{[j]}) = 1$ and $\mathsf{r}(T_i^{[k]}) \geq \mathsf{C}(\mathcal{S}W, T_i^{[j]})$, then $\mathsf{A}(\mathcal{S}W,\, T_i,\, \mathsf{C}(\mathcal{S}W, T_i^{[j]}) - 1,\, \mathsf{C}(\mathcal{S}W, T_i^{[j]}) + 1) \leq \mathsf{Swt}(T_i, \mathsf{C}(\mathcal{S}W, T_i^{[j]}))$.

(AF3) For any subtask $T_i^{[j]}$, $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) \leq \mathsf{d}(T_i^{[j]})$.

(AF4) For any subtask $T_i^{[j]}$ and any time t, if $t < \mathsf{r}(T_i^{[j]})$ or $t \geq \mathsf{C}(\mathcal{S}W, T_i^{[j]})$, then $\mathsf{A}(\mathcal{S}W,\ T_i^{[j]},\ t) = 0$.

(AF5) For any present subtask $T_i^{[j]}$, let $T_i^{[k]}$ be its next present successor $T_i^{[k]}$ (if it exists). If $T_i^{[k]}$ does not exist, and $\mathsf{D}(T_i^{[j]}) > 0$, then for any t such that $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) - 1 \leq t \leq \mathsf{D}(T_i^{[j]})$, $\mathsf{A}(\mathcal{S}W,\ T_i,\ \mathsf{C}(\mathcal{S}W, T_i^{[j]}) - 1,\ t+1) \leq \mathsf{Swt}(T_i, t)$ holds. Otherwise, if $T_i^{[k]}$ exists, $\mathsf{D}(T_i^{[j]}) > 0$, and $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) \leq \mathsf{r}(T_i^{[k]})$, then for any t such that $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) - 1 \leq t \leq \mathsf{min}(\mathsf{r}(T_i^{[k]}), \mathsf{D}(T_i^{[j]}) - 1)$, $\mathsf{A}(\mathcal{S}W,\ T_i,\ \mathsf{C}(\mathcal{S}W, T_i^{[j]}) - 1,\ t+1) \leq \mathsf{Swt}(T_i, t)$ holds.

Given the examples in Figure 5.14 and Figure 5.16, both Properties (AF1) and (AF4) should be fairly intuitive. As for Property (AF2), notice that, in Figure 5.14, $\mathsf{A}(\mathcal{S}W,\ T_1,\ \mathsf{C}(\mathcal{S}W, T_1^{[1]}) - 1,\ \mathsf{C}(\mathcal{S}W, T_1^{[1]}) + 1) = 1/16 + 4/16 \leq \mathsf{Swt}(T_1, 4) = 5/16$ and $\mathsf{A}(\mathcal{S}W,\ T_1,\ \mathsf{C}(\mathcal{S}W, T_1^{[4]}) - 1,\ \mathsf{C}(\mathcal{S}W, T_1^{[4]}) + 1) = 4/16 + 0 \leq \mathsf{Swt}(T_1, 14) = 5/16$. Also notice that, in Figure 5.16(b), which depicts a task T_i that changes its weight from 3/19 to 2/5 via Rule N at time 8, $\mathsf{A}(\mathcal{S}W,\ T_i,\ \mathsf{C}(\mathcal{S}W, T_i^{[2]}) - 1,\ \mathsf{C}(\mathcal{S}W, T_i^{[2]}) + 1) = 32/95 + 0 \leq \mathsf{Swt}(T_i, 10) = 2/5$. Also note that, in Figure 5.16(c), which depicts a task T_1 that changes its weight from 3/19 to 2/5 via Rule P at time 8, $\mathsf{A}(\mathcal{S}W,\ T_i,\ \mathsf{C}(\mathcal{S}W, T_i^{[1]}) - 1,\ \mathsf{C}(\mathcal{S}W, T_i^{[1]}) + 1) = 1/19 + 0 \leq \mathsf{Swt}(T_i, 7) = 3/19$.

As for Property (AF3), recall that (for systems in which no heavy task initiates a weight change before the group deadline of its last-released subtask) in the absence of reweighting events, $\mathsf{d}(T_i^{[j]}) = \mathsf{C}(\mathcal{S}W, T_i^{[j]})$. To increase the completion time of $T_i^{[j]}$ (and hence $\mathsf{C}(\mathcal{S}W, T_i^{[j]})$) in $\mathcal{S}W$, T_i would have to enact a weight change in the range $(\mathsf{r}(T_i^{[j]}), \mathsf{d}(T_i^{[j]}))$ that decreases the weight of T_i, without halting $T_i^{[j]}$. However, by Property (X2), a change enacted in the range $(\mathsf{r}(T_i^{[j]}), \mathsf{d}(T_i^{[j]}))$ must have been initiated in the range $[\mathsf{r}(T_i^{[j]}), \mathsf{d}(T_i^{[j]}))$. Thus, by Property (RW) when such a change is initiated, T_i is either positive, negative-, or heavy-changeable. Since only Rule N can enact a weight change before $\mathsf{d}(T_i^{[j]})$ without halting the last-released subtask (i.e., $T_i^{[j]}$), when such a change is initiated, T_i must be negative-changeable. However, by Rule N, no weight decrease that is initiated in the range $[\mathsf{r}(T_i^{[j]}), \mathsf{d}(T_i^{[j]}))$ can be enacted before time $\mathsf{C}(\mathcal{S}W, T_i^{[j]})$. Thus, the completion time for a subtask in $\mathcal{S}W$ is upper bounded by the deadline of the subtask.

As for Property (AF5), notice that, in Figure 5.15(a), as successive subtasks of a task T_i are released, the allocations to each of T_i's subtasks in the SW schedule in the first slot of each subtask's window decreases up until the group deadline. For example, $\mathsf{A}(\mathcal{S}W,\, T_1^{[1]},\, \mathsf{r}(T_1^{[1]})) = 7/9$, $\mathsf{A}(\mathcal{S}W,\, T_1^{[2]},\, \mathsf{r}(T_1^{[2]})) = 5/9$, $\mathsf{A}(\mathcal{S}W,\, T_1^{[3]},\, \mathsf{r}(T_1^{[3]})) = 3/9$, and $\mathsf{A}(\mathcal{S}W,\, T_1^{[4]},\, \mathsf{r}(T_1^{[4]})) = 1/9$. As a result, if T_i has no present subtasks between $T_i^{[j]}$ and $T_i^{[k]}$ and $\mathsf{r}(T_i^{[k]}) \leq \omega(T_i^{[j]})$, then the allocation over the range $[\mathsf{C}(\mathcal{S}W, T_i^{[j]}) - 1,\, \mathsf{r}(T_i^{[k]}))$ is at most the scheduling weight of T_i. For example, in Figure 5.15(b), $\mathsf{A}(\mathcal{S}W,\, T_1,\, \mathsf{C}(\mathcal{S}W, T_1^{[1]}) - 1,\, \mathsf{r}(T_1^{[4]})) = 2/9 + 1/9 = 3/9 \leq \mathsf{Swt}(T_1, 4) = 7/9$ and $\mathsf{A}(\mathcal{S}W,\, T_1,\, \mathsf{C}(\mathcal{S}W, T_1^{[4]}) - 1,\, \mathsf{r}(T_1^{[6]}) + 1) = 1/9 + 4/9 \leq 5/9 = \mathsf{Swt}(T_1, 14)$.

It is worthwhile to note that the proofs of Properties (AF1), (AF3), and (AF5) follow directly from corresponding IS properties that were proven in (Srinivasan, 2003).

As a consequence of Properties (AF1) and (W), LAG can only increase over a slot if there is a hole in that slot. Hence, the lemma below follows.

Lemma 5.4. *If* $\mathsf{LAG}(\tau, t) < \mathsf{LAG}(\tau, t+1)$, *then there is a hole in slot* t.

5.5.4 Correctness Proof

Having defined the AGIS task model, displacements, and some basic properties, we can now prove the following theorem.

Theorem 5.2. *Under* $\mathsf{PD\text{-}PNH}$, *no subtask is scheduled after its deadline, provided that Property* (W) *holds.*

To prove of Theorem 5.2, suppose that it does not hold. Then, there exists a time t_d and a task system τ as given in the definitions below.

Definition 5.4 (t_d)**.** t_d is the earliest time at which any AGIS task system instance misses a deadline under $\mathsf{PD\text{-}PNH}$.

Definition 5.5 (τ **and** $\mathcal{S}$)**.** τ is an instance of an AGIS task system with the following properties.

(T1) τ misses a deadline under $\mathsf{PD\text{-}PNH}$ at t_d.

(T2) No task system satisfying (T1) has fewer present subtasks in $[0, t_d)$ than τ.

In the remainder of this proof, we let $\mathcal{S}$ denote PD-PNH schedule of τ.

By (T1), (T2), and the definition of t_d, exactly one subtask in τ misses its deadline at t_d: if several subtasks miss their deadlines, then all but one can be removed and the remaining subtask will still miss its deadline, contradicting (T2). We now prove several properties about $\mathcal{S}$.

Lemma 5.5. *The following properties hold for τ and $\mathcal{S}$, where $T_i^{[j]}$ is any subtask in $\mathcal{S}$.*

(a) *For any present subtask $T_i^{[j]}$, $\mathsf{d}(T_i^{[j]}) \leq t_d$.*

(b) *There are no holes in slot $t_d - 1$.*

(c) $\mathsf{LAG}(\tau, t_d) = 1$.

(d) $\mathsf{LAG}(\tau, t_d - 1) \geq 1$.

(e) *No present subtask halts.*

Proof of (a). Suppose that τ contains a subtask $T_i^{[j]}$ with a deadline greater than t_d. $T_i^{[j]}$ can be removed without affecting the scheduling of higher-priority subtasks with earlier deadlines. Thus, if $T_i^{[j]}$ is removed, then a deadline still missed at t_d. This contradicts (T2). □

Proof of (b). If there were a hole in slot $t_d - 1$, then the subtask that misses its deadline at t_d would have been scheduled there, which is a contradiction. (Note that, by the minimality of t_d, its predecessor meets its deadline at or before $t_d - 1$ and hence is not scheduled in slot $t_d - 1$.) □

Proof of (c). By (5.1), we have

$$\mathsf{LAG}(\tau, t_d) = \mathsf{A}(\mathcal{S}W,\, \tau,\, 0,\, t_d) - \mathsf{A}(\mathcal{S},\, \tau,\, 0,\, t_d).$$

In the above equation, the term $\mathsf{A}(\mathcal{S}W,\, \tau,\, 0,\, t_d)$ equals the total number of present subtasks in τ. The second term corresponds to the number of subtasks scheduled by PD-PNH in $[0, t_d)$.

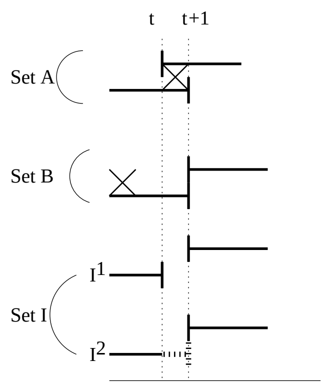


Figure 5.19: The task sets A, B, and I.

Since exactly one subtask misses its deadline, the difference between these two terms is one, i.e., $\mathsf{LAG}(\tau, t_d) = 1$. $\square$

Proof of (d). By (b), there are no holes in slot $t_d - 1$. Hence, by Lemma 5.4, $\mathsf{LAG}(\tau, t_d - 1) \geq \mathsf{LAG}(\tau, t_d)$. Therefore, by (c), $\mathsf{LAG}(\tau, t_d - 1) \geq 1$. $\square$

Proof of (e). If $T_i^{[j]}$ is a halted subtask, then it is never scheduled. Hence, it can be removed and a deadline will still be missed at t_d, contradicting (T2). $\square$

Definition 5.6 (A_t**,** B_t**, and** I_t**).** A_t, B_t, and I_t are all defined with respect to schedule $\mathcal{S}$ and some time t. A_t denotes the set of tasks that have a subtask scheduled at t. B_t denotes the set of tasks that are not scheduled at t, and receive some allocation in $\mathcal{S}W$ at slot t, i.e., $\mathsf{A}(\mathcal{S}W, T_i, t) > 0$ for $T_i \in \mathsf{B}_t$. I_t denotes the set of all tasks that are in the system at t but are not in A_t or B_t. A_t, B_t, and I_t are illustrated in Figure 5.19. Notice that, there are two types of tasks in set I_t: tasks that have a deadline at or before t (denoted I_t^1) and tasks that have a deadline after t but complete in $\mathcal{S}W$ at or before t (denoted I_t^2).

Displacement-based proofs. We now prove several lemmas about tasks in A_t and B_t when there is a hole at time t. The proofs of the following lemmas all use the same basic technique. First, we assume, to derive a contradiction, that the lemma does not hold for some subtask, $T_u^{[j]}$. Then, we show that $T_u^{[j]}$ can be removed from τ without causing a chain of displacements that extends beyond time t. Thus, the system without $T_u^{[j]}$ misses a deadline at time t_d. This, in turn, implies that τ violates property (T2), which is a contradiction.

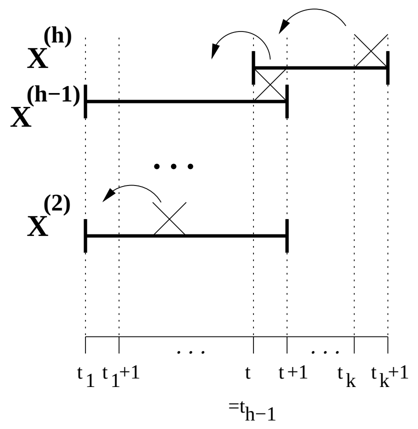


Figure 5.20: An illustration of the chain of displacements that occurs by removing $\mathsf{X}^{(1)} = T_u^{[j]}$ in Lemma 5.6.

Lemma 5.6. *Let $t < t_d - 1$ be a time at which there is a hole in $\mathcal{S}$. Let T_u be any task in B_t or A_t. Let $T_u^{[j]}$ be the subtask with the largest index such that $\mathsf{r}(T_u^{[j]}) \leq t < \mathsf{d}(T_u^{[j]})$ and $T_u^{[j]}$ is scheduled at or before t. Then, $\mathsf{d}(T_u^{[j]}) = t + 1 \wedge \mathsf{b}(T_u^{[j]}) = 1$.*

Proof. Let t and T_u be as defined in the statement of the lemma. If $T_u \in \mathsf{A}_t$, then since $t < t_d - 1$ and $T_u^{[j]}$ is scheduled at or before t, $\mathsf{d}(T_u^{[j]}) \geq t + 1$. If $T_u \in \mathsf{B}_t$, then since $\mathsf{A}(\mathcal{SW}, T_u, t) > 0$ it follows that T_u is not complete in $\mathcal{SW}$ before $t + 1$. Thus, by Property (AF3), $\mathsf{d}(T_u^{[j]}) \geq \mathsf{C}(\mathcal{SW}, T_i^{[j]}) \geq t + 1$. Thus, if we can show that a contradiction follows from

$$\mathsf{d}(T_u^{[j]}) > t + 1 \text{ or } \mathsf{d}(T_u^{[j]}) = t + 1 \wedge \mathsf{b}(T_u^{[j]}) = 0, \tag{5.15}$$

then the proof is complete.

We now show that if (5.15) holds, then $T_u^{[j]}$ can be removed and a deadline will still be missed at t_d, contradicting Property (T2). (Before continuing, notice that, since $T_u^{[j]}$ is scheduled, it is present, and as a result, $T_u^{[j]}$ can be removed.) Let the chain of displacements caused by removing $T_u^{[j]}$ be $\Delta_1, \Delta_2, ..., \Delta_k$, where $\Delta_i = \langle \mathsf{X}^{(i)}, t_i, \mathsf{X}^{(i+1)}, t_{i+1} \rangle$ and $\mathsf{X}^{(1)} = T_u^{[j]}$. By Lemma 5.1, $t_{i+1} > t_i$, for $1 \leq i \leq k$. (This chain of displacements is illustrated in Figure 5.20.)

Note that, at slot t_i, the priority of $\mathsf{X}^{(i)}$ is at least that of $\mathsf{X}^{(i+1)}$, because $\mathsf{X}^{(i)}$ was chosen over $\mathsf{X}^{(i+1)}$ in $\mathcal{S}$. Thus, because $\mathsf{X}^{(1)} = T_u^{[j]}$, by (5.15), for each subtask $1 \leq i \leq k + 1$, either $\mathsf{d}(\mathsf{X}^{(i)}) > t + 1$ or $\mathsf{d}(\mathsf{X}^{(i)}) = t + 1 \wedge \mathsf{b}(\mathsf{X}^{(i)}) = 0$. We now show that the displacements do not extend beyond slot t. Assume to the contrary that $t_{k+1} > t$. Consider $h \in \{2, ..., k+1\}$ such that $t_h > t$ and $t_{h-1} \leq t$. Such an h exists because $t_1 \leq t < t_{k+1}$. Because there is a hole in slot t and $t_{h-1} \leq t < t_h$, by Lemma 5.3, $t_{h-1} = t$ and $\mathsf{X}^{(h)}$ is $\mathsf{X}^{(h-1)}$'s successor. Since a subtask cannot be scheduled before it is released, $\mathsf{r}(\mathsf{X}^{(h)}) < t+1$. Since $h-1 \leq k$, either $\mathsf{d}(\mathsf{X}^{(h-1)}) > t+1$ or $\mathsf{d}(\mathsf{X}^{(h-1)}) = t+1 \wedge \mathsf{b}(\mathsf{X}^{(h-1)}) = 0$ holds. Therefore, since $\mathsf{r}(\mathsf{X}^{(h)}) < t+1$, $\mathsf{d}(\mathsf{X}^{(h-1)}) - \mathsf{b}(\mathsf{X}^{(h-1)}) > \mathsf{r}(\mathsf{X}^{(h)})$ holds. Thus, by Property (GV), $\mathsf{X}^{(h-1)}$ is scheduled before $\mathsf{r}(\mathsf{X}^{(h)}) \leq t$ in $\mathcal{S}$, contradicting our assumption that $\mathsf{X}^{(h-1)}$ is scheduled in slot t.

Thus, the displacements do not extend beyond slot t. Hence, no subtask scheduled after t is "left-shifted." Thus, a deadline is still missed at time t_c, contradicting Property (T2). Hence, $\mathsf{d}(T_u^{[j]}) = t+1 \wedge \mathsf{b}(T_u^{[j]}) = 1$. $\square$

Lemma 5.7. *Let $t < t_d - 1$ be a time at which there is a hole in $\mathcal{S}$. Let T_u be any task in B_t. Let $T_u^{[j]}$ be the subtask with the largest index such that $\mathsf{r}(T_u^{[j]}) \leq t < \mathsf{d}(T_u^{[j]})$ and $T_u^{[j]}$ is scheduled before t. Then, there exists a subtask $T_a^{[b]}$ scheduled in slot t such that $\mathsf{D}(T_a^{[b]}) \leq \mathsf{D}(T_u^{[j]})$.*

Proof. Let t and T_u be as defined in the statement of the lemma. Suppose that all subtasks scheduled in slot t have a group deadline greater than $\mathsf{D}(T_u^{[j]})$. We now show that $T_u^{[j]}$ can be removed and a deadline will still be missed at t_d, contradicting Property (T2). (Before continuing, notice that, since $T_u^{[j]}$ is scheduled, it is present, and as a result, $T_u^{[j]}$ can be removed.) Let the chain of displacements caused by removing $T_u^{[j]}$ be $\Delta_1, \Delta_2, ..., \Delta_k$, where $\Delta_i = \langle \mathsf{X}^{(i)}, t_i, \mathsf{X}^{(i+1)}, t_{i+1} \rangle$, $\mathsf{X}^{(1)} = T_u^{[j]}$. By Lemma 5.1, $t_{i+1} > t_i$, for $1 \leq i \leq k$.

Note that, at slot t_i, the priority of $\mathsf{X}^{(i)}$ is at least that of $\mathsf{X}^{(i+1)}$, because $\mathsf{X}^{(i)}$ was chosen over $\mathsf{X}^{(i+1)}$ in $\mathcal{S}$. Thus, because $\mathsf{X}^{(1)} = T_u^{[j]}$ and, by Lemma 5.6, $\mathsf{d}(T_u^{[j]}) = t+1 \wedge \mathsf{b}(T_u^{[j]}) = 1$, we have the following property.

(Q) For any value of i such that $1 \leq i \leq k+1$, if $\mathsf{d}(\mathsf{X}^{(i)}) = t+1$ and $\mathsf{b}(\mathsf{X}^{(i)}) = 1$, then $\mathsf{D}(\mathsf{X}^{(i)}) \leq \mathsf{D}(T_u^{[j]})$.

We now show that the displacements do not extend beyond slot t. Assume to the contrary that $t_{k+1} > t$. Consider $h \in \{2, ..., k+1\}$ such that $t_h > t$ and $t_{h-1} \leq t$. Such an h exists because $t_1 \leq t < t_{k+1}$. Because there is a hole in slot t and $t_{h-1} \leq t < t_h$, by Lemma 5.3, $t_{h-1} = t$. Thus, $\mathsf{X}^{(h-1)}$ is scheduled in slot t and, by extension, $\mathsf{X}^{(h-1)}$ has the largest index of any subtask of its task that is scheduled at or before t. Thus, by Lemma 5.6, $\mathsf{d}(\mathsf{X}^{(h-1)}) = t+1 \wedge \mathsf{b}(\mathsf{X}^{(h-1)}) = 1$. Thus, by Property (Q), $\mathsf{D}(\mathsf{X}^{(h-1)}) \leq \mathsf{D}(T_u^{[j]})$, which contradicts our assumption all subtasks scheduled in slot t have a group deadline greater than $\mathsf{D}(T_u^{[j]})$.

Thus, the displacements do not extend beyond slot t. Hence, no subtask scheduled after t is "left-shifted." Thus, a deadline is still missed at time t_d, contradicting Property (T2). Hence, there exists a subtask $T_a^{[b]}$ scheduled in slot t such that $\mathsf{D}(T_a^{[b]}) \leq \mathsf{D}(T_u^{[j]})$. $\square$

Lemma 5.8. *Let $t < t_d - 1$ be a slot in which there is a hole in $\mathcal{S}$. Let T_u be any task in A_t. Let $T_u^{[j]}$ be the subtask of T_u that is scheduled in slot t. Then, $T_u^{[j]}$'s successor is released at time t.*

Proof. Let t and T_u be as defined in the statement of the lemma. By Lemma 5.6, $\mathsf{d}(T_u^{[j]}) = t+1$ and $\mathsf{b}(T_u^{[j]}) = 1$. Since $T_u^{[j]}$ is scheduled at time t, $T_u^{[j]}$ is not complete by time t in $\mathcal{S}$. Thus, by Property (GV), $T_u^{[j]}$'s successor cannot be released before $\mathsf{d}(T_u^{[j]}) - \mathsf{b}(T_u^{[j]}) = t$.

We now show that $T_u^{[j]}$'s successor must be released by time t, which completes the proof. Suppose, to derive a contradiction, that $T_u^{[j]}$'s successor is released after time t. We now show that $T_u^{[j]}$ can be removed and a deadline will still be missed at t_d, contradicting Property (T2). (Before continuing, notice that, since $T_u^{[j]}$ is scheduled, it is present, and as a result, $T_u^{[j]}$ can be removed.) Let the chain of displacements caused by removing $T_u^{[j]}$ be $\Delta_1, \Delta_2, ..., \Delta_k$, where $\Delta_i = \langle \mathsf{X}^{(i)}, t_i, \mathsf{X}^{(i+1)}, t_{i+1} \rangle$, $\mathsf{X}^{(1)} = T_u^{[j]}$. By Lemma 5.1, $t_{i+1} > t_i$, for $1 \leq i \leq k$.

We now show that the displacements do not extend beyond slot t. Assume to the contrary that $t_{k+1} > t$. Consider $h \in \{2, ..., k+1\}$ such that $t_h > t$ and $t_{h-1} \leq t$. Such an h exists because $t_1 \leq t < t_{k+1}$. Because there is a hole in slot t and $t_{h-1} \leq t < t_h$, by Lemma 5.3, $t_{h-1} = t$, $\mathsf{X}^{(h)}$ is $\mathsf{X}^{(h-1)}$'s successor. Moreover, by Lemma 5.1, $\mathsf{X}^{(h-1)} = T_u^{[j]}$. Notice that the displacement $\langle \mathsf{X}^{(h-1)}, t_{h-1}, \mathsf{X}^{(h)}, t_h \rangle$ is valid iff $\mathsf{r}(\mathsf{X}^{(h)}) \leq t$, which implies that $T_u^{[j]}$'s successor is released by time t. However, this contradicts our assumption that $T_u^{[j]}$'s successor is released after time t.

Thus, the displacements do not extend beyond slot t. Hence, no subtask scheduled after t is "left-shifted." Thus, a deadline is still missed at time t_d, contradicting Property (T2). Hence, $T_u^{[j]}$'s successor is released at time t. $\square$

Consecutive holes. Notice that, if there are two consecutive slots with holes, t_a and t_b, then by Lemma 5.8, for any subtask $T_u^{[j]}$ scheduled in slot t_a, $T_u^{[j]}$'s successor, $T_u^{[q]}$, is released at time t_a. As a result, since $\mathsf{r}(T_u^{[q]}) = t_a < t_b$, there is a hole in slot t_b, and $T_u^{[j]}$ is scheduled in slot t_a, it follows that $T_u^{[q]}$ is scheduled in slot t_b. Thus, we have the following lemma.

Lemma 5.9. *Let t_a and t_b be two consecutive slots such that $t_a = t_b - 1 < t_d - 1$ and both t_a and t_b contain holes in the schedule $\mathcal{S}$. Let T_u be any task in A_{t_a}. Let $T_u^{[j]}$ be the subtask of T_u that is scheduled in slot t_a. Then, $T_u^{[j]}$'s successor is scheduled in slot t_b.*

Lemma 5.10. *Let t_a and t_b be two consecutive slots such that $t_a = t_b - 1 < t_d - 1$ and both t_a and t_b contain holes in the schedule $\mathcal{S}$. Let T_u be any task in A_{t_b}. Let $T_u^{[j]}$ be the subtask of T_u that is scheduled in slot t_b. Then, $T_u^{[j]}$'s predecessor is scheduled in slot t_a.*

Proof. Let t_a, t_b, and T_u be as defined in the statement of the lemma. By Lemma 5.6, $\mathsf{d}(T_u^{[j]}) = t_b + 1$ and $\mathsf{b}(T_u^{[j]}) = 1$. Thus, by Property (WL), $\mathsf{r}(T_u^{[j]}) \leq \mathsf{d}(T_u^{[j]}) - 2 = t_b - 1 = t_a$. Since there is a hole at t_a and $\mathsf{r}(T_u^{[j]}) \leq t_a$, $T_u^{[j]}$ would be scheduled in slot t_a, unless its predecessor was scheduled there. Since $T_u^{[j]}$ is scheduled in slot t_b, its predecessor must be scheduled in slot t_a. □

From Lemmas 5.9 and 5.10, the subsequent corollary follows.

Corollary 5.1. *Let t_a and t_b be two slots such $t_a \leq t_b \leq t_d - 1$ and there is a hole in every slot in the range $\{t_a, ..., t_b\}$. If a task T_x is not scheduled in slot t_a, then it is not scheduled in any slot over the range $\{t_a, ..., t_b\}$.*

Notice that, by Lemma 5.9, any task T_u that is scheduled in the first hole in a sequence of slots with holes must be scheduled in every slot in the sequence. Moreover, by Lemma 5.8, a subtask of T_u must be released at every slot in this sequence. Thus, there can be no IS separations between subtasks until the sequence of holes is over. In addition, every subtask of T_u that is released over the range $\{t_a, t_b - 2\}$ will be scheduled in the second slot of its window. Thus, we have the following corollaries (depicted in Figure 5.21).

Corollary 5.2. *Let t_a and t_b be two slots such that $t_a \leq t_b \leq t_d - 1$ and there is a hole in every slot in the range $\{t_a, ..., t_b\}$. If $T_u^{[j]}$ is scheduled at time t_a, then for $q \in \{j, ..., j + 1 + t_b - t_a\}$, $\theta(T_u^{[q]}) = \theta(T_u^{[j]})$,* i.e., *there are no IS separations between any two subtasks of T_u.*

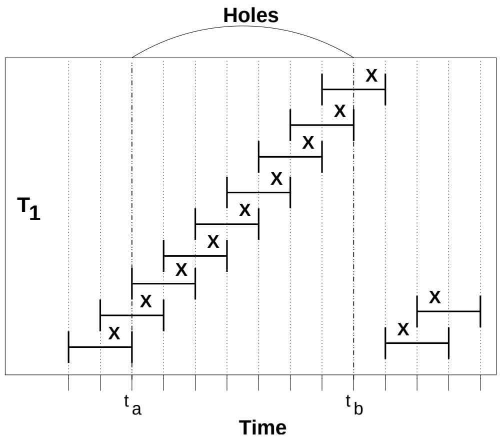


Figure 5.21: An illustration of Corollaries 5.2 and 5.3.

Corollary 5.3. *Let t_a and t_b be two slots such $t_a < t_b \leq t_d - 1$ and there is a hole in every slot in the range $\{t_a, ..., t_b\}$. If $T_u^{[j]}$ is scheduled at time t_a, then for any subtask $T_u^{[q]}$ such that $t_a \leq \mathsf{r}(T_u^{[q]}) \leq t_b - 1$, $T_u^{[q]}$ is scheduled in slot $\mathsf{r}(T_u^{[q]}) + 1$.*

Time $t_{\mathcal{H}}$. Since, by part (d) of Lemma 5.5, $\mathsf{LAG}(\tau, t_d - 1) \geq 1$, and by the definition of LAG, $\mathsf{LAG}(\tau, 0) = 0$, there exists a time $t_{\mathcal{H}}$ such that

$$0 \leq t_{\mathcal{H}} < t_d - 1 \wedge \mathsf{LAG}(\tau, t_{\mathcal{H}}) < 1 \wedge \mathsf{LAG}(\tau, t_{\mathcal{H}} + 1) \geq 1. \tag{5.16}$$

Without loss of generality, let $t_{\mathcal{H}}$, be the latest such time, i.e., for all u such that $t_{\mathcal{H}} < u \leq t_d - 1$, $\mathsf{LAG}(\tau, u) \geq 1$. We now show that such a $t_{\mathcal{H}}$ cannot exist, thus contradicting our starting assumption that t_d and τ exist. For brevity, we use A to denote $\mathsf{A}_{t_{\mathcal{H}}}$, B to denote $\mathsf{B}_{t_{\mathcal{H}}}$, and I to denote $\mathsf{I}_{t_{\mathcal{H}}}$.

Lemma 5.11. *B is non-empty*

Proof. Let the number of holes in slot $t_{\mathcal{H}}$ be h. Then, $\mathsf{A}(\mathcal{S}, \tau, t_{\mathcal{H}}) = M - h$. By (5.1), $\mathsf{LAG}(\tau, t_{\mathcal{H}} + 1) = \mathsf{LAG}(\tau, t_{\mathcal{H}}) + \mathsf{A}(\mathcal{SW}, \tau, t_{\mathcal{H}}) - \mathsf{A}(\mathcal{S}, \tau, t_{\mathcal{H}})$ (recall that for this section we have assumed that $\mathsf{LAG}(\tau, t) = \mathsf{LAG}(\mathcal{S}, \mathcal{SW}, \tau, t)$). Thus, because $\mathsf{LAG}(\tau, t_{\mathcal{H}} + 1) > \mathsf{LAG}(\tau, t_{\mathcal{H}})$ (by (5.16)), we have $\mathsf{A}(\mathcal{SW}, \tau, t_{\mathcal{H}}) > M - h$. Since, for every $T_v \notin A \cup B$, $\mathsf{A}(\mathcal{SW}, T_v, t_{\mathcal{H}}) = 0$, it follows that $\mathsf{A}(\mathcal{SW}, A \cup B, t_{\mathcal{H}}) > M - h$. Therefore, by Property (AF1), $\sum_{T_i \in A} (\mathsf{Swt}(T_i, t_{\mathcal{H}})) + \sum_{T_i \in B} (\mathsf{A}(\mathcal{SW}, T_i, t_{\mathcal{H}})) > M - h$. Because the number of tasks scheduled in slot $t_{\mathcal{H}}$ is $M-h$, $|A| = M-h$. Because $\mathsf{Swt}(T_i, t) \leq 1$, for any task T_i at any time t, $\sum_{T_i \in A} \mathsf{Swt}(T_i, t_{\mathcal{H}}) \leq M - h$. Thus, $\sum_{T_i \in B} \mathsf{A}(\mathcal{S}W,\, T_i,\, t_{\mathcal{H}}) > 0$. Hence, B is not empty. $\square$

(TK) Let T_u be any task in B and let $T_u^{[j]}$ be the subtask of T_u with the largest index such that $\mathsf{r}(T_u^{[j]}) \leq t_{\mathcal{H}} < \mathsf{d}(T_u^{[j]})$ and $T_u^{[j]}$ is scheduled before $t_{\mathcal{H}}$. Then, no present subtask of T_u with an index greater than j (including $T_u^{[j]}$'s successor) is released before $\mathsf{d}(T_u^{[j]})$.

Notice that Lemma 5.6 implies $\mathsf{d}(T_u^{[j]}) = t_{\mathcal{H}} + 1$. Thus, Property (TK) easily follows from the fact that there is a hole in slot $t_{\mathcal{H}}$ and no subtask of any task in B is scheduled in slot $t_{\mathcal{H}}$.

Lemma 5.12. *Let T_u be any task in B. Let $T_u^{[j]}$ be the subtask of T_u with the largest index such that $\mathsf{r}(T_u^{[j]}) \leq t_{\mathcal{H}} < \mathsf{d}(T_u^{[j]})$ and $T_u^{[j]}$ is scheduled before $t_{\mathcal{H}}$. Then, $\mathsf{C}(\mathcal{S}W, T_u^{[j]}) = t_{\mathcal{H}} + 1$.*

Proof. Let T_u and $T_u^{[j]}$ be defined as in the statement of the lemma. Since $T_u^{[j]}$ is the subtask of T_u with the largest index such that $\mathsf{r}(T_u^{[j]}) \leq t_{\mathcal{H}} < \mathsf{d}(T_u^{[j]})$ and $T_u^{[j]}$ is scheduled before $t_{\mathcal{H}}$ (by the definition of B, no subtask of $T_u \in B$ is scheduled in slot $t_{\mathcal{H}}$), by Lemma 5.6, $\mathsf{d}(T_u^{[j]}) = t_{\mathcal{H}} + 1$. Thus, by Property (AF3), $\mathsf{C}(\mathcal{S}W, T_u^{[j]}) \leq t_{\mathcal{H}} + 1$.

We now show that for $\ell \neq j$, $\mathsf{A}(\mathcal{S}W,\, T_u^{[\ell]},\, t_{\mathcal{H}}) = 0$. First, we consider $\ell > j$. From the definition of $T_u^{[j]}$, at least one of the following three conditions must hold: **(i)** $\mathsf{r}(T_u^{[\ell]}) > t_{\mathcal{H}}$; **(ii)** $\mathsf{d}(T_u^{[\ell]}) \leq t_{\mathcal{H}}$; or **(iii)** $T_u^{[\ell]}$ is not scheduled by time $t_{\mathcal{H}}$. If Conditions (i) or (ii) hold, then by Properties (AF3) and (AF4), $\mathsf{A}(\mathcal{S}W,\, T_u^{[\ell]},\, t_{\mathcal{H}}) = 0$. If Condition (iii) holds, then by Property (TK), either $\mathsf{r}(T_u^{[\ell]}) > \mathsf{d}(T_u^{[j]})$, in which case Condition (i) holds, or $T_u^{[\ell]}$ is not present, in which case $\mathsf{A}(\mathcal{S}W,\, T_u^{[\ell]},\, t_{\mathcal{H}}) = 0$. Thus, for $\ell > j$, $\mathsf{A}(\mathcal{S}W,\, T_u^{[\ell]},\, t_{\mathcal{H}}) = 0$. Now, consider $\ell < j$. By Lemma 5.6, $\mathsf{d}(T_u^{[j]}) = t_{\mathcal{H}} + 1 \wedge \mathsf{b}(T_u^{[j]}) = 1$. By Property (WL), every subtask has a window length of at least two. Hence, $\mathsf{r}(T_u^{[j]}) \leq t_{\mathcal{H}} - 1$. By Property (GV), if $\mathsf{d}(T_u^{[\ell]}) - \mathsf{b}(T_u^{[\ell]}) > \mathsf{r}(T_u^{[j]})$ holds, then $\mathsf{C}(\mathcal{S}W, T_u^{[\ell]}) \leq \mathsf{r}(T_u^{[j]})$ holds. Thus, either $\mathsf{C}(\mathcal{S}W, T_u^{[\ell]}) \leq \mathsf{r}(T_u^{[j]})$ or $\mathsf{d}(T_u^{[\ell]}) - \mathsf{b}(T_u^{[\ell]}) \leq \mathsf{r}(T_u^{[j]})$ holds. Since $\mathsf{r}(T_u^{[j]}) \leq t_{\mathcal{H}} - 1$, this implies that either $\mathsf{C}(\mathcal{S}W, T_u^{[\ell]}) \leq \mathsf{r}(T_u^{[j]}) \leq t_{\mathcal{H}} - 1$ or $\mathsf{d}(T_u^{[\ell]}) \leq \mathsf{r}(T_u^{[j]}) + \mathsf{b}(T_u^{[\ell]}) \leq t_{\mathcal{H}} - 1 + \mathsf{b}(T_u^{[\ell]}) \leq t_{\mathcal{H}}$ holds. Since, by Property (AF3), $\mathsf{C}(\mathcal{S}W, T_u^{[\ell]}) \leq \mathsf{d}(T_u^{[\ell]})$, $\mathsf{C}(\mathcal{S}W, T_u^{[\ell]}) \leq t_{\mathcal{H}}$ holds in either case. Thus, by Property (AF4), $\mathsf{A}(\mathcal{S}W,\, T_u^{[\ell]},\, t_{\mathcal{H}}) = 0$.

Thus, the allocation to each subtask of T_u, except $T_u^{[j]}$, in $\mathcal{S}W$ in slot $t_{\mathcal{H}}$ is zero. By the definition of B, $\mathsf{A}(\mathcal{S}W,\, T_u,\, t_{\mathcal{H}}) > 0$. Thus, since $\mathsf{A}(\mathcal{S}W,\, T_u,\, t_{\mathcal{H}}) = \sum_{T_u^{[i]} \in T_u} \mathsf{A}(\mathcal{S}W,\, T_u^{[i]},\, t_{\mathcal{H}})$,

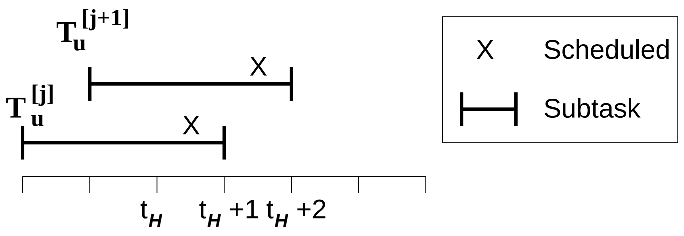


Figure 5.22: An illustration of the proof of Lemma 5.13, where there is a hole in time slots $t_{\mathcal{H}}$ and $t_{\mathcal{H}}+1$.

there must exist at least one present subtask with a positive allocation in $\mathcal{S}W$ in the slot $t_{\mathcal{H}}$. Thus, $\mathsf{A}(\mathcal{S}W,\, T_u^{[j]},\, t_{\mathcal{H}}) > 0$. By Property (AF4), this implies that $\mathsf{C}(\mathcal{S}W, T_u^{[j]}) \geq t_{\mathcal{H}}+1$. Since we have already established that $\mathsf{C}(\mathcal{S}W, T_u^{[j]}) \leq t_{\mathcal{H}}+1$, we have $\mathsf{C}(\mathcal{S}W, T_u^{[j]}) = t_{\mathcal{H}}+1$. □

Lemma 5.13. *Let T_u be a task in A and let $T_u^{[j]}$ be the subtask of T_u that is scheduled in slot $t_{\mathcal{H}}$. If $\mathsf{D}(T_u^{[j]}) = 0$, then there is no hole in slot $t_{\mathcal{H}}+1$.*

Proof. Let T_u and $T_u^{[j]}$ be as defined in the statement of the lemma. Since tasks $t_{\mathcal{H}} < t_d$, $T_u^{[j]}$ is scheduled before its deadline, i.e., $t_{\mathcal{H}} < \mathsf{d}(T_u^{[j]})$. Moreover, since tasks are scheduled sequentially, it follows that $T_u^{[j]}$ has the largest index of any subtask of T_u scheduled at or before $t_{\mathcal{H}}$. (T_u is illustrated in Figure 5.22.)

Thus, by Lemma 5.6, $\mathsf{d}(T_u^{[j]}) = t_{\mathcal{H}}+1 \wedge \mathsf{b}(T_u^{[j]}) = 1$. In addition, since there is a hole in $t_{\mathcal{H}}$, by Lemma 5.8, $T_u^{[j]}$'s successor, $T_u^{[j+1]}$, is released at time $t_{\mathcal{H}}$, i.e., $\mathsf{r}(T_u^{[j+1]}) = t_{\mathcal{H}}$.

We now assume that there is a hole in slot $t_{\mathcal{H}}+1$ to derive a contradiction. By this assumption and Lemma 5.9, it follows that $T_u^{[j+1]}$ is scheduled in slot $t_{\mathcal{H}}+1$. Since $T_u^{[j+1]}$ is scheduled at $t_{\mathcal{H}}+1 < t_d$, $T_u^{[j+1]}$ does not miss its deadline. Hence, $\mathsf{d}(T_u^{[j+1]}) \geq t_{\mathcal{H}}+2$. Since subtasks are scheduled in index order, $T_u^{[j+1]}$ has the largest index of any subtask of T_u scheduled at or before $t_{\mathcal{H}}+1$. Thus, by Lemma 5.6, $\mathsf{d}(T_u^{[j+1]}) = t_{\mathcal{H}}+2 \wedge \mathsf{b}(T_u^{[j+1]}) = 1$. Thus, because (as was already established) $\mathsf{r}(T_u^{[j+1]}) = t_{\mathcal{H}}$, it follows that $\mathsf{d}(T_u^{[j+1]}) - \mathsf{r}(T_u^{[j+1]}) = 2$.

Since $T_u^{[j]}$ is scheduled in slot $t_{\mathcal{H}}$, $\mathsf{b}(T_u^{[j]}) = 1$, $\mathsf{D}(T_u^{[j]}) = 0$, and $\mathsf{r}(T_u^{[j+1]}) = t_{\mathcal{H}}$, by Property (GD-3), $\mathsf{D}(T_u^{[j+1]}) = 0$. Thus, by Property (WL), $\mathsf{d}(T_u^{[j+1]}) - \mathsf{r}(T_u^{[j+1]}) \geq 3$; however, this contradicts our earlier assertion that $\mathsf{d}(T_u^{[j+1]}) - \mathsf{r}(T_u^{[j+1]}) = 2$. Thus, there cannot exist a hole in slot $t_{\mathcal{H}}+1$, which completes the proof. □

Corollary 5.4. *Let T_b be a task in B and let $T_b^{[c]}$ be the subtask of T_b with the largest index such that $\mathsf{r}(T_b^{[c]}) \le t_{\mathcal{H}} < \mathsf{d}(T_b^{[c]})$. If $\mathsf{D}(T_b^{[c]}) = 0$, then there is no hole in slot $t_{\mathcal{H}} + 1$.*

Proof. Let T_b and $T_b^{[c]}$ be as defined in the statement of the lemma. Since by the statement of a lemma, $\mathsf{D}(T_b^{[c]}) = 0$, it follows, by Lemma 5.7, that there is a subtask, $T_u^{[j]}$, of a task T_u in A such that $\mathsf{D}(T_u^{[j]}) = 0$ and $T_u^{[j]}$ is scheduled at time $t_{\mathcal{H}}$. Thus, by Lemma 5.13, there is no hole in slot $t_{\mathcal{H}} + 1$. □

Lemma 5.14. *Let T_u be a task in B and let $T_u^{[j]}$ be the subtask of T_u with the largest index such that $\mathsf{r}(T_u^{[j]}) \le t_{\mathcal{H}} < \mathsf{d}(T_u^{[j]})$. If $\mathsf{D}(T_u^{[j]}) > 0$, then there exists a slot with no holes in $[\mathsf{d}(T_u^{[j]}), \mathsf{min}(\mathsf{D}(T_u^{[j]}), t_d))$.*

Proof. By Lemma 5.6, $\mathsf{d}(T_u^{[j]}) = t_{\mathcal{H}} + 1 \wedge \mathsf{b}(T_u^{[j]}) = 1$. By (5.16), $t_{\mathcal{H}} < t_d - 1$. Thus, $\mathsf{d}(T_u^{[j]}) < t_d - 1$. Thus, by Lemma 5.5(b), if $\mathsf{D}(T_u^{[j]}) \ge t_d - 1$, then the proof is complete. Therefore, for the remainder of the proof, we assume that

$$\mathsf{D}(T_u^{[j]}) < t_d - 1. \tag{5.17}$$

Since $\mathsf{D}(T_u^{[j]}) > 0$ and $\mathsf{b}(T_u^{[j]}) = 1$, by Property (GD-2), we have that $\mathsf{D}(T_u^{[j]}) > \mathsf{d}(T_u^{[j]})$. Thus,

$$\mathsf{D}(T_u^{[j]}) \ge \mathsf{d}(T_u^{[j]}) + 1 = t_{\mathcal{H}} + 2. \tag{5.18}$$

By Lemma 5.7, we have the following property:

(E-1) There exists a subtask $T_x^{[z]}$ scheduled in slot $t_{\mathcal{H}}$ such that $\mathsf{D}(T_x^{[z]}) \le \mathsf{D}(T_u^{[j]})$.

(T_x is illustrated in Figure 5.23.) By Lemma 5.13, if $\mathsf{D}(T_x^{[z]}) = 0$, then there is no hole in slot $t_{\mathcal{H}}+1$. Since, we have assumed that $\mathsf{D}(T_u^{[j]}) < t_d - 1$ and, by (5.18), $\mathsf{D}(T_u^{[j]}) \ge t_{\mathcal{H}} + 1 = \mathsf{d}(T_u^{[j]})$, it follows that if $\mathsf{D}(T_x^{[z]}) = 0$, then there exists a slot with no holes in $[\mathsf{d}(T_u^{[j]}), \mathsf{min}(\mathsf{D}(T_u^{[j]}), t_d))$. Thus, for the remainder of the proof we assume that

$$\mathsf{D}(T_x^{[z]}) > 0. \tag{5.19}$$

Since $T_x^{[z]}$ is scheduled in slot $t_{\mathcal{H}}$, and $t_{\mathcal{H}} < t_d$, it follows $T_x^{[z]}$ is scheduled before its deadline. Thus, since a subtask must be released before it is scheduled, $\mathsf{r}(T_x^{[z]}) \leq t_{\mathcal{H}} < \mathsf{d}(T_x^{[z]})$. Moreover, since subtasks are scheduled sequentially, $T_x^{[z]}$ has the largest index of any subtask of T_x that is scheduled at or before $t_{\mathcal{H}}$ such that $\mathsf{r}(T_x^{[z]}) \leq t_{\mathcal{H}} < \mathsf{d}(T_x^{[z]})$. Thus, by Lemma 5.6, $\mathsf{d}(T_x^{[z]}) = t_{\mathcal{H}} + 1 \wedge \mathsf{b}(T_x^{[z]}) = 1$. Thus, by (5.19) and Properties (E-1) and (GD-2), it follows that

$$t_{\mathcal{H}} + 2 \leq \mathsf{D}(T_x^{[z]}) \leq \mathsf{D}(T_u^{[j]}). \tag{5.20}$$

To derive a contradiction, we assume that every slot in the set $\{t_{\mathcal{H}}, ..., \mathsf{D}(T_u^{[j]}) - 1\}$ contains a hole. Since by (5.20), $\mathsf{D}(T_x^{[z]}) \leq \mathsf{D}(T_u^{[j]})$, if there is a hole in every slot in the set $\{t_{\mathcal{H}}, ..., \mathsf{D}(T_u^{[j]}) - 1\}$, then we have the following property.

(E-2) Every slot in the set $\{t_{\mathcal{H}}, ..., \mathsf{D}(T_x^{[z]}) - 1\}$ contains a hole.

(Notice that the set $\{t_{\mathcal{H}}, ..., \mathsf{D}(T_x^{[z]}) - 1\}$ is non-empty since by (5.20), $t_{\mathcal{H}} + 2 \leq \mathsf{D}(T_x^{[z]})$.) By Lemma 5.9 and Properties (E-1) and (E-2), it follows that T_x is scheduled in every slot in the set $\{t_{\mathcal{H}}, ..., \mathsf{D}(T_x^{[z]}) - 1\}$. Thus, by Corollary 5.2, we have the following property.

(E-3) For all $q \in \{z, ..., z + (\mathsf{D}(T_x^{[z]}) - t_{\mathcal{H}})\}$, $\theta(T_x^{[z]}) = \theta(T_x^{[q]})$.

Let k equal the index of the subtask $\omega(T_x^{[z]})$. Notice that $\mathsf{D}(T_x^{[z]}) - t_{\mathcal{H}} + z = k$ since the number of T_x's subtasks that are released after $\mathsf{r}(T_x^{[z]})$ but before $\mathsf{D}(T_x^{[z]})$ equals $\mathsf{D}(T_x^{[z]}) - t_{\mathcal{H}}$. Thus, by Property (E-3), it follows that for all $T_x^{[q]} \in \{T_x^{[z]}, ..., \omega(T_x^{[z]})\}$, $\theta(T_x^{[z]}) = \theta(T_x^{[q]})$. Thus, by Property (GD-1), the subtask $\omega(T_x^{[z]})$, has the following two properties

(Y-1) $\mathsf{r}(\omega(T_x^{[z]})) = \mathsf{D}(T_x^{[z]}) - 2$.

(Y-2) Either $\mathsf{d}(\omega(T_x^{[z]})) = \mathsf{D}(T_x^{[z]})$ and $\mathsf{b}(\omega(T_x^{[z]})) = 0$ or $\mathsf{d}(\omega(T_x^{[z]})) = \mathsf{D}(T_x^{[z]}) + 1$ and $\mathsf{b}(\omega(T_x^{[z]})) = 1$.

We now show that $\omega(T_x^{[z]})$ cannot satisfy both Properties (Y-1) and (Y-2), which contradicts Property (E-2) and completes the proof.

Since $t_{\mathcal{H}} < \mathsf{D}(T_x^{[z]}) - 1$ (by (5.20)), and $\mathsf{D}(T_x^{[z]}) - 1 < t_d - 1$ (by 5.17), by Properties (E-1) and (E-2), we have the following.

(E-4) $t_{\mathcal{H}} < \mathsf{D}(T_x^{[z]}) - 1 \leq t_d - 1$, where there is a hole in every slot in the range $\{t_{\mathcal{H}}, ..., \mathsf{D}(T_x^{[z]}) - 1\}$, and $T_x^{[z]}$ is scheduled in slot $t_{\mathcal{H}}$.

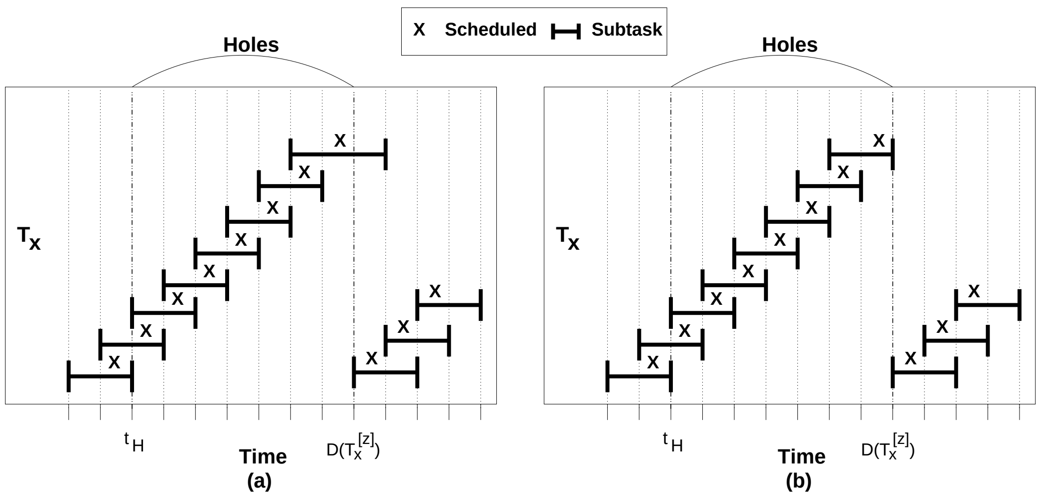


Figure 5.23: An illustration of the two possibilities for T_x in Lemma 5.14.

Recall that, by Property (Y-1), $\mathsf{r}(\omega(T_x^{[z]})) = \mathsf{D}(T_x^{[z]}) - 2$. Thus, by Corollary 5.3 and Property (E-4), it follows that $\omega(T_x^{[z]})$ is scheduled in slot $\mathsf{r}(\omega(T_x^{[z]})) + 1 = \mathsf{D}(T_x^{[z]}) - 1$.

Since $\omega(T_x^{[z]})$ is scheduled at time $\mathsf{D}(T_x^{[z]}) - 1$, by Property (E-2), there is a hole at the time it is scheduled. Thus, by Lemma 5.6, it follows that $\mathsf{b}(\omega(T_x^{[z]})) = 1$ and $\mathsf{d}(\omega(T_x^{[z]})) = \mathsf{D}(T_x^{[z]})$; however, this contradicts Property (Y-2). ☐

The following lemma contradicts our choice of $t_{\mathcal{H}}$ as the last slot such that $\mathsf{LAG}(\tau, t_{\mathcal{H}}) < 1$.

Lemma 5.15. *Let T_b be a task in B and let $T_b^{[c]}$ be the subtask of T_b with the largest index such that $\mathsf{r}(T_b^{[c]}) \le t_{\mathcal{H}} < \mathsf{d}(T_b^{[c]})$. If $\mathsf{D}(T_b^{[c]}) = 0$, then $\mathsf{LAG}(\tau, t_{\mathcal{H}} + 2) < 1$.*

Proof. Let the number of holes in slot $t_{\mathcal{H}}$ be h. Assume that $T_b^{[c]}$ exists and is defined as in the statement of the lemma. We now derive some properties about the per-slot allocations to tasks in the $\mathcal{S}W$ schedule in slots $t_{\mathcal{H}}$ and $t_{\mathcal{H}} + 1$.

By the definition of I, if task T_y is in I, then $\mathsf{A}(\mathcal{S}W, T_y, t_{\mathcal{H}}) = 0$. Since $\tau = A \cup B \cup I$, $\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W, T_y, t_{\mathcal{H}}) = \sum_{T_y \in A \cup B} \mathsf{A}(\mathcal{S}W, T_y, t_{\mathcal{H}})$. Since $\mathsf{Swt}(T_y, t) \le 1$, for any task T_y and any time t, we have $\sum_{T_y \in A} \mathsf{Swt}(T_y, t_{\mathcal{H}}) \le |A|$. Thus, by Property (AF1), $\sum_{T_y \in A} \mathsf{A}(\mathcal{S}W, T_y, t_{\mathcal{H}}) \le |A|$. Because there are h holes in slot $t_{\mathcal{H}}$, $M - h$ tasks are scheduled at $t_{\mathcal{H}}$, i.e., $|A| = M - h$. Thus, $\sum_{T_y \in A} \mathsf{A}(\mathcal{S}W, T_y, t_{\mathcal{H}}) \le M - h$, and hence

$$\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W, T_y, t_{\mathcal{H}}) \le M - h + \sum_{T_y \in B} \mathsf{A}(\mathcal{S}W, T_y, t_{\mathcal{H}}). \tag{5.21}$$

Let C denote the set of tasks that receive a positive allocation in $\mathcal{S}W$ in slot $t_{\mathcal{H}}+1$ *and* are not in B. Then, the set of tasks that receive a positive allocation in $\mathcal{S}W$ is a subset of $C \cup B$. Thus, by Property (W) in Section 5.5,

$$\sum_{T_y \in C \cup B} \mathsf{Swt}(T_y, t_{\mathcal{H}}+1) \leq M. \tag{5.22}$$

Also, $\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}}+1) = \sum_{T_y \in C \cup B} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}}+1)$. By Property (AF1), this implies that $\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}}+1) \leq \sum_{T_y \in C} \mathsf{Swt}(T_y, t_{\mathcal{H}}+1) + \sum_{T_y \in B} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}}+1)$. Thus, by (5.21),

$$\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{H}}+2) \leq M - h + \sum_{T_y \in C} \mathsf{Swt}(T_y, t_{\mathcal{H}}+1) + \sum_{T_y \in B} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{H}}+2) \tag{5.23}$$

Consider $T_u \in B$. Let $T_u^{[j]}$ be the subtask of T_u with the largest index such that $\mathsf{r}(T_u^{[j]}) \leq t_{\mathcal{H}} < \mathsf{d}(T_u^{[j]})$ that is scheduled before $t_{\mathcal{H}}$. Let D denote the set of such subtasks for all tasks in B. Then, by Lemmas 5.6 and 5.12,

$$\text{for all } T_u^{[j]} \in D, \mathsf{C}(\mathcal{S}W, T_u^{[j]}) = \mathsf{d}(T_u^{[j]}) = t_{\mathcal{H}}+1 \wedge \mathsf{b}(T_u^{[j]}) = 1. \tag{5.24}$$

By (TK), $T_u^{[j]}$'s successor $T_u^{[k]}$ (if it exists) is not released until at or after $t_{\mathcal{H}}+1 \geq \mathsf{C}(\mathcal{S}W, T_u^{[j]})$. Since $\mathsf{r}(T_u^{[k]}) \geq \mathsf{C}(\mathcal{S}W, T_u^{[j]})$ and $\mathsf{b}(T_u^{[j]}) = 1$, by (AF2), $\mathsf{A}(\mathcal{S}W,\, T_u,\, t_{\mathcal{H}},\, t_{\mathcal{H}}+2) \leq \mathsf{Swt}(T_u, t_{\mathcal{H}}+1)$. Thus, $\sum_{T_y \in B} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{H}}+2) \leq \sum_{T_y \in B} \mathsf{Swt}(T_y, t_{\mathcal{H}}+1)$.

By (5.23), this implies that

$$\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{H}}+2) \leq M - h + \sum_{T_y \in C \cup B} \mathsf{Swt}(T_y, t_{\mathcal{H}}+1).$$

Thus, from (5.22) it follows that

$$\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{H}}+2) \leq M - h + M. \tag{5.25}$$

Notice that, by Corollary 5.4, the existence of $T_b^{[c]}$ (from the statement of the lemma) implies that there is no hole in slot $t_{\mathcal{H}} + 1$. Thus, since there are h holes in slot $t_{\mathcal{H}}$, we have $\mathsf{A}(\mathcal{S},\, \tau,\, t_{\mathcal{H}},\, t_{\mathcal{H}} + 2) = M - h + M$. Hence, by (5.25), $\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{H}} + 2) \leq \sum_{T_y \in \tau} \mathsf{A}(\mathcal{S},\, \tau,\, t_{\mathcal{H}},\, t_{\mathcal{H}} + 2)$. Using this relation in (5.1) we obtain, $\mathsf{LAG}(\tau, t_{\mathcal{H}}{+}2) = \mathsf{LAG}(\tau, t_{\mathcal{H}}) + \sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{H}} + 2) - \sum_{T_y \in \tau} \mathsf{A}(\mathcal{S},\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{H}} + 2)$ (recall that for this section, we have assumed that $\mathsf{LAG}(\tau, t) = \mathsf{LAG}(\mathcal{S}, \mathcal{S}W, \tau, t)$). Since, $\mathsf{LAG}(\tau, t_{\mathcal{H}}) < 1$, we have $\mathsf{LAG}(\tau, t_{\mathcal{H}} + 2) < 1$. □

Lemma 5.16. *If $t_{\mathcal{N}}$ is the first time after $t_{\mathcal{H}}$ such that there are no holes in the schedule $\mathcal{S}$, then $\mathsf{LAG}(\tau, t_{\mathcal{N}} + 1) < 1$.*

Proof. By Corollary 5.4, if there exists a task T_b in B with a subtask $T_b^{[c]}$, where $\mathsf{D}(T_b^{[c]}) = 0$, and $T_b^{[c]}$ has the largest index of any task in T_b such that $\mathsf{r}(T_b^{[c]}) \leq t_{\mathcal{H}} < \mathsf{d}(T_b^{[c]})$ holds, then there is no hole in slot $t_{\mathcal{H}} + 1$ (i.e., $t_{\mathcal{N}} = t_{\mathcal{H}} + 1$), and by Lemma 5.15, $\mathsf{LAG}(\tau, t_{\mathcal{N}} + 1) < 1$. Thus, for the remainder of this proof we assume that no such task is in B.

Let the number of holes in slot $t_{\mathcal{H}}$ be h. By Corollary 5.1, only tasks in A are scheduled in slots in the range $\{t_{\mathcal{H}}, ..., t_{\mathcal{N}} - 1\}$, and by Lemma 5.9, every task in A is scheduled in every slot in this range. Thus, we have the following property

(H) There are h holes in every slot in the range $\{t_{\mathcal{H}}, ..., t_{\mathcal{N}} - 1\}$.

We now derive some properties about the per-slot allocations to tasks in the $\mathcal{S}W$ schedule in the slots $\{t_{\mathcal{H}}, ..., t_{\mathcal{N}}\}$.

Allocations in $\mathcal{S}W$ in slot $t_{\mathcal{H}}$. By the definition of set I, if a task T_y is in I, then $\mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}}) = 0$. Since $\tau = A \cup B \cup I$, $\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}}) = \sum_{T_y \in A \cup B} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}})$. Since $\mathsf{Swt}(T_y, t) \leq 1$ for any task T_y, we have $\sum_{T_y \in A} \mathsf{Swt}(T_y, t_{\mathcal{H}}) \leq |A|$. Thus, by Property (AF1), $\sum_{T_y \in A} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}}) \leq |A|$. Because there are h holes in slot $t_{\mathcal{H}}$, $M - h$ tasks are scheduled at $t_{\mathcal{H}}$, i.e., $|A| = M - h$. Thus, $\sum_{T_y \in A} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}}) \leq M - h$, and hence

$$\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}}) \leq M - h + \sum_{T_y \in B} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}}). \tag{5.26}$$

Allocations in $\mathcal{S}W$ between slots $t_{\mathcal{H}}$ and $t_{\mathcal{N}}$. By Corollary 5.1, only tasks in A release subtasks in the slots $\{t_{\mathcal{H}} + 1, ..., t_{\mathcal{N}} - 1\}$. (If a subtask of a task not in A was released within this range, then it would be scheduled within this range since there are holes in every slot.) Moreover, by Lemma 5.6, any task that is released at or before $t_{\mathcal{H}}$ and is not part of A must have a deadline no later than $t_{\mathcal{H}} + 1$. Thus, by Properties (AF3) and (AF4), the allocations to all tasks in $\mathcal{S}W$, not in A, in the slots $\{t_{\mathcal{H}} + 1, ..., t_{\mathcal{N}} - 1\}$ is zero. Moreover, reasoning as above, $\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t) \leq M - h$ holds for any slot t in this range. Therefore,

$$\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}} + 1,\, t_{\mathcal{N}}) \leq (t_{\mathcal{N}} - t_{\mathcal{H}} - 2) \cdot (M - h). \tag{5.27}$$

Allocations in $\mathcal{S}W$ in slot $t_{\mathcal{N}}$. Let C denote the set of tasks that receive a positive allocation in $\mathcal{S}W$ in slot $t_{\mathcal{N}}$ *and* are not in B. Then, the set of tasks that receive a positive allocation in $\mathcal{S}W$ is a subset of $C \cup B$. By Property (W) in Section 5.5,

$$\sum_{T_y \in C \cup B} \mathsf{Swt}(T_y, t_{\mathcal{N}}) \leq M. \tag{5.28}$$

Also, $\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{N}}) = \sum_{T_y \in C \cup B} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{N}})$. By Property (AF1), this implies that

$$\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{N}}) \leq \sum_{T_y \in C} \mathsf{Swt}(T_y, t_{\mathcal{N}}) + \sum_{T_y \in B} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{N}}). \tag{5.29}$$

By (5.26), (5.27), and (5.29), the total SW allocation in the slots $\{t_{\mathcal{H}}, ..., t_{\mathcal{N}}\}$ is

$$\begin{aligned} \textstyle\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{N}}) \leq \; & \textstyle\sum_{T_y \in C} \mathsf{Swt}(T_y, t_{\mathcal{N}}) + (t_{\mathcal{N}} - t_{\mathcal{H}} - 1) \cdot (M - h) \\ & + \textstyle\sum_{T_y \in B} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{N}} + 1). \end{aligned} \tag{5.30}$$

Constructing an upper bound for $\sum_{T_y \in B} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{N}} + 1)$. Consider $T_u \in B$. Let $T_u^{[j]}$ be the subtask of T_u with the largest index such that $\mathsf{r}(T_u^{[j]}) \leq t_{\mathcal{H}} < \mathsf{d}(T_u^{[j]})$ that is scheduled before $t_{\mathcal{H}}$. Let D denote the set of such subtasks for all tasks in B. Notice that, by the assumption we made at the beginning of this proof, for any subtask $T_u^{[j]}$ in D,

$$\mathsf{D}(T_u^{[j]}) > 0 \tag{5.31}$$

and $\omega(T_u^{[j]})$ is defined. By Lemmas 5.6 and 5.12,

$$\mathsf{C}(\mathcal{S}W, T_u^{[j]}) = \mathsf{d}(T_u^{[j]}) = t_{\mathcal{H}} + 1 \wedge \mathsf{b}(T_u^{[j]}) = 1. \tag{5.32}$$

In addition, since, by the statement of the lemma $t_{\mathcal{H}} < t_{\mathcal{N}}$, it follows from (5.32) that

$$\mathsf{C}(\mathcal{S}W, T_u^{[j]}) = t_{\mathcal{H}} + 1 \leq t_{\mathcal{N}}. \tag{5.33}$$

Moreover, by Lemma 5.14, $t_{\mathcal{N}}$ is before the earliest group deadline of any subtask in D. Thus,

$$t_{\mathcal{N}} < \mathsf{D}(T_u^{[j]}). \tag{5.34}$$

Let $T_u^{[f]}$ be $T_u^{[j]}$'s next present successor. We now show that, if $T_u^{[f]}$ does not exist, then by Property (AF5), $\mathsf{A}(\mathcal{S}W,\, T_u,\, t_{\mathcal{H}},\, t_{\mathcal{N}} + 1) \leq \mathsf{Swt}(T_u, t_{\mathcal{N}})$. In order to apply (AF5) the following conditions must hold: **(i)** $\mathsf{D}(T_u^{[j]}) > 0$; **(ii)** $\mathsf{C}(\mathcal{S}W, T_u^{[j]}) - 1 \leq t_{\mathcal{N}}$; and **(iii)** $t_{\mathcal{N}} \leq \mathsf{D}(T_u^{[j]})$. Condition (i) holds by (5.31). Condition (ii) holds by (5.33). Condition (iii) holds by (5.34). Thus, Property (AF5) applies, and $\mathsf{A}(\mathcal{S}W,\, T_u,\, t_{\mathcal{H}},\, t_{\mathcal{N}} + 1) \leq \mathsf{Swt}(T_u, t_{\mathcal{N}})$.

We now consider the possibility where $T_u^{[f]}$ exists. By Corollary 5.1, since no subtask of any task in B is scheduled in slot $t_{\mathcal{H}}$, no subtask of any task in B is scheduled over the range $\{t_{\mathcal{H}}, ..., t_{\mathcal{N}} - 1\}$. Thus, since there are holes in every slot in this range, it follows that no task in B releases a present subtask at a slot within the range $\{t_{\mathcal{H}}, ..., t_{\mathcal{N}} - 1\}$. Thus, if $T_u^{[f]}$ exists, then $t_{\mathcal{N}} \leq \mathsf{r}(T_u^{[f]})$. Hence, by (5.34),

$$t_{\mathcal{N}} \leq \mathsf{min}(\mathsf{r}(T_u^{[f]}), \mathsf{D}(T_u^{[j]})). \tag{5.35}$$

We now apply Property (AF5) to show that $\mathsf{A}(\mathcal{S}W,\, T_u,\, t_{\mathcal{H}},\, t_{\mathcal{N}} + 1) \leq \mathsf{Swt}(T_u, t_{\mathcal{N}})$. In order for Property (AF5) to apply, the following conditions must hold: **(i)** $\mathsf{D}(T_u^{[j]}) > 0$; **(ii)** $\mathsf{C}(\mathcal{S}W, T_u^{[j]}) \leq \mathsf{r}(T_u^{[f]})$; **(iii)** $\mathsf{C}(\mathcal{S}W, T_u^{[j]}) - 1 \leq t_{\mathcal{N}}$; and **(iv)** $t_{\mathcal{N}} \leq \mathsf{min}(\mathsf{r}(T_u^{[f]}), \mathsf{D}(T_u^{[j]}) - 1)$. Condition (i) holds by (5.31). By (5.35) and (5.33), $\mathsf{C}(\mathcal{S}W, T_u^{[j]}) = t_{\mathcal{H}} + 1 \leq t_{\mathcal{N}} \leq \mathsf{min}(\mathsf{r}(T_u^{[f]}), \mathsf{D}(T_u^{[j]}))$. This implies that $\mathsf{C}(\mathcal{S}W, T_u^{[j]}) - 1 \leq \mathsf{r}(T_u^{[f]})$ holds, which satisfies Condition (ii). Similarly, (5.33) implies that Condition (iii) holds, and (5.35) implies that Condition

(iv) holds. Thus, by Property (AF5), it follows that $\mathsf{A}(\mathcal{S}W,\, T_u,\, t_{\mathcal{H}},\, t_{\mathcal{N}}+1) \le \mathsf{Swt}(T_u, t_{\mathcal{N}})$.

Thus, regardless of whether $T_u^{[f]}$ exists or not, $\sum_{T_y \in B} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{N}}+1) \le \sum_{T_y \in B} \mathsf{Swt}(T_y, t_{\mathcal{N}})$.

By (5.30), this implies that

$$\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{N}}+1) \le (t_{\mathcal{N}} - t_{\mathcal{H}} - 1) \cdot (M-h) + \sum_{T_y \in C \cup B} \mathsf{Swt}(T_y, t_{\mathcal{N}}).$$

Thus, from (5.28) it follows that

$$\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{N}}+1) \le (t_{\mathcal{N}} - t_{\mathcal{H}} - 1) \cdot (M-h) + M. \tag{5.36}$$

Completing the proof. Since, by the statement of the lemma, there is no hole in slot $t_{\mathcal{N}}$ and by Property (H), there are h holes in every slot in the range $\{t_{\mathcal{H}}, ..., t_{\mathcal{N}} - 1\}$, we have

$$\mathsf{A}(\mathcal{S},\, \tau,\, t_{\mathcal{H}},\, t_{\mathcal{N}}+1) = (t_{\mathcal{N}} - t_{\mathcal{H}} - 1) \cdot (M-h) + M.$$

Hence, by (5.36), $\sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{N}}+1) \le \sum_{T_y \in \tau} \mathsf{A}(\mathcal{S},\, \tau,\, t_{\mathcal{H}},\, t_{\mathcal{N}}+1)$. By (5.1), $\mathsf{LAG}(\tau, t_{\mathcal{N}}+1) = \mathsf{LAG}(\tau, t_{\mathcal{H}}) + \sum_{T_y \in \tau} \mathsf{A}(\mathcal{S}W,\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{N}}+1) - \sum_{T_y \in \tau} \mathsf{A}(\mathcal{S},\, T_y,\, t_{\mathcal{H}},\, t_{\mathcal{N}}+1)$ (recall that for this section, we have assumed that $\mathsf{LAG}(\tau, t) = \mathsf{LAG}(\mathcal{S}, \mathcal{S}W, \tau, t)$). Since, $\mathsf{LAG}(\tau, t_{\mathcal{H}}) < 1$, we obtain $\mathsf{LAG}(\tau, t_{\mathcal{N}}+1) < 1$. □

It follows, by Lemma 5.16, there exists a time, t after $t_{\mathcal{H}}$ such that $\mathsf{LAG}(\tau, t) < 1$. This contradicts (5.16), which implies that t_d does not exist. Thus, Theorem 5.2 holds.

5.6 Drift

We now turn our attention to the issue of measuring drift under PD-PNH. In order to measure the drift of a task system τ, we introduce two additional theoretical scheduling algorithms: the *clairvoyant scheduling-weight* (CSW) scheduling algorithm; and the *ideal* (IDEAL) scheduling algorithm. The CSW scheduling algorithm is the same as the SW scheduling algorithm except that the CSW scheduling algorithm is “clairvoyant” so that it does not allocate capacity to tasks that will halt. Under the IDEAL scheduling algorithm, at each instant t, each task T_i in τ is allocated a share equal to its weight $\mathsf{wt}(T_i,\, t)$. Hence, over the interval $[t_1,\, t_2)$, the task T_i is allocated $\mathsf{A}(\mathcal{I},\, T_i,\, t_1,\, t_2) = \int_{t_1}^{t_2} \mathsf{wt}(T_i,\, u)du$ time. We use $\mathcal{CSW}$ and $\mathcal{I}$ to denote, respectively, CSW and IDEAL schedules for a given system.

Using the definition of $\mathcal{SW}$, we can simply define $\mathcal{CSW}$ as follows:

$$\mathsf{A}(\mathcal{CSW},\, T_i^{[j]},\, t) = \begin{cases} \mathsf{A}(\mathcal{SW},\, T_i^{[j]},\, t), & \text{if } T_i^{[j]} \text{ never halts} \\ 0, & \text{otherwise.} \end{cases}$$

For example, in Figure 5.4(a) $\mathsf{A}(\mathcal{SW},\, T_i^{[j]},\, t) = \mathsf{A}(\mathcal{CSW},\, T_i^{[j]},\, t)$ except for $T_1^{[2]}$, where $\mathsf{A}(\mathcal{CSW},\, T_1^{[2]},\, t) = 0$, for all t. We use $\mathcal{CSW}$ and $\mathcal{I}$ to denote, respectively, CSW and IDEAL schedules for a given system.

For the remainder of this section, we assume that every subtask in T_i is released as early as possible. This assumption can be removed at the cost of more complex notation. If we did not make this assumption, then the allocation function for $\mathcal{I}$ would equal zero between active subtasks.

Comparing IDEAL to SW and CSW. $\mathcal{I}$ is similar to $\mathcal{SW}$ and $\mathcal{CSW}$, with three major exceptions: **(i)** tasks in $\mathcal{I}$ continually receive allocations, whereas tasks in $\mathcal{SW}$ and $\mathcal{CSW}$ receive allocations only at quantum boundaries; **(ii)** under $\mathcal{I}$, each task receives an allocation equal to its *weight*, whereas under $\mathcal{SW}$ and $\mathcal{CSW}$, each task receives allocations according to its *scheduling weight*; and **(iii)** the total allocation each task receives in $\mathcal{SW}$ and $\mathcal{CSW}$ is calculated based on the releases and completion times of its active subtasks, whereas allocations in $\mathcal{I}$ are independent of subtask releases and completion times. Hence, even if all active subtasks of a given task are halted, $\mathcal{I}$ still allocates capacity to that task.

Example (Figure 5.24). Consider the example in Figure 5.24, which depicts the allocations in the schedules $\mathcal{CSW}$ and $\mathcal{I}$ (insets (a) and (b), respectively) to a task T_1 that has an initial weight of 3/19 that increases to 2/5 (via Rule N) at time 8. Notice that, over the range $[8, 10)$ in $\mathcal{I}$, T_1 receives an allocation equal to its weight at every instant (for a total allocation of 4/5 over $[8, 10)$). Compare this to $\mathcal{CSW}$, in which T_1 receives only an allocation of 44/95 over

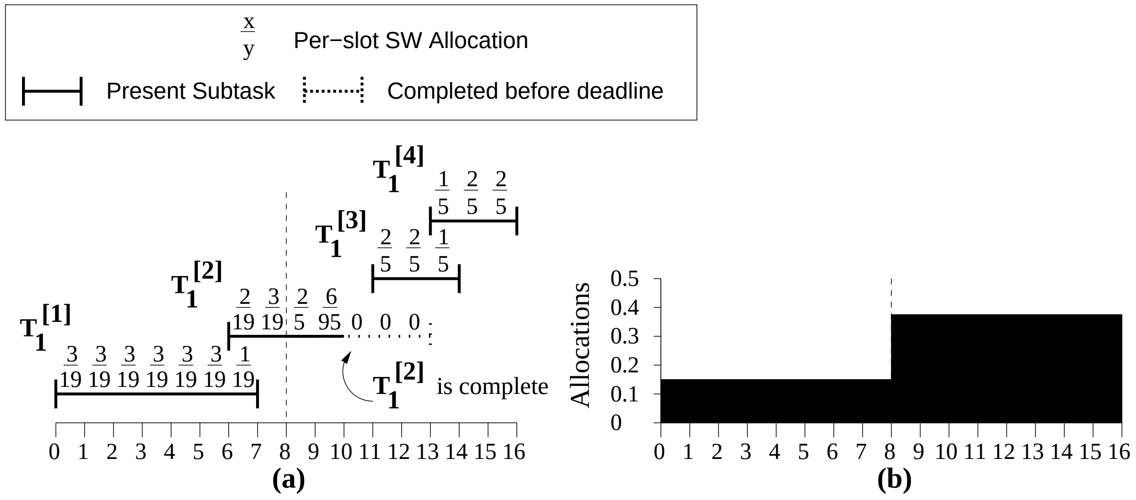


Figure 5.24: Allocations for a task T_1 with an initial weight of 3/19 that changes to 2/5. **(a)** The value of $\mathsf{A}(\mathcal{CSW}, T_1^{[j]}, u)$ for each slot and subtask. **(b)** The allocations to T_1 in $\mathcal{I}$ at each instant.

the same range. □

As was the case for the adaptable sporadic task model and the modified adaptable sporadic task model, the *drift* of a task T_i is calculated as the difference between T_i's allocations in the $\mathcal{CSW}$ and the $\mathcal{I}$ schedules. Formally, under PD-PNH, the *drift of a task T_i is defined* as

$$\mathsf{drift}(T_i, t) = \mathsf{A}(\mathcal{I}, T_i, 0, t) - \mathsf{A}(\mathcal{CSW}, T_i, 0, t). \tag{5.37}$$

Example (Figure 5.8). For example, in Figure 5.8(b), the drift of task T_1 at time $t = 9$ is $\mathsf{A}(\mathcal{I}, T_1, 0, 9) - \mathsf{A}(\mathcal{CSW}, T_1, 0, 9) = 27/20 - 20/20 = 7/20$, whereas at time $t = 10$, the drift of T_i is $\mathsf{A}(\mathcal{I}, T_1, 0, 10) - \mathsf{A}(\mathcal{CSW}, T_1, 0, 10) = 3/2 - 1 = 1/2$. Notice that, since $T_1^{[2]}$ is halted at time 10, $\mathsf{A}(\mathcal{CSW}, T_1^{[2]}, 0, 10) = 0$. □

We say that a reweighting algorithm is *fine-grained* iff there exists some constant value c such that the drift per weight change is less than c. We say that a reweighting algorithm is *coarse-grained* otherwise.

5.6.1 PD-LJ is Not Fine-Grained

We now prove that PD-LJ is not fine-grained. (The definition of drift for PD-LJ is the same as the definition of drift for PD-PNH, except that $\mathcal{CSW}$ is determined by using PD-LJ.)

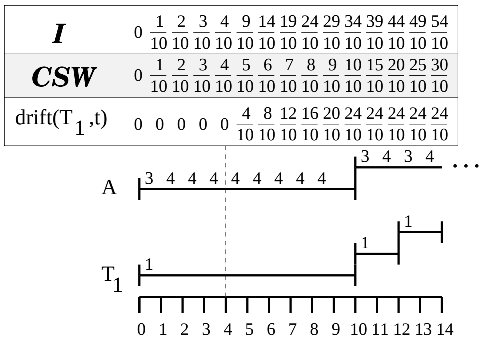


Figure 5.25: A four-processor example illustrating why PD-LJ is coarse-grained. A is a set of 35 tasks each with a weight 1/10. T_1 has a weight of 1/10 that increases to 1/2 at time 4.

Example (Figure 5.25). Consider the four-processor example in Figure 5.25, which depicts the PD-LJ schedule for a system that consists of a set A of 35 tasks with weight 1/10 and a task T_1 with weight 1/10 that increases to 1/2 at time 4. By Rule L, T_1 cannot "leave" until time 10. Hence, the change is not enacted until time 10. Thus, over the range $[4, 10)$, T_i receives a 1/10 per-slot allocation in $\mathcal{CSW}$ and 1/2 in $\mathcal{I}$. Hence, T_i's drift reaches a value of 24/10 at time 10. This example can be generalized by decreasing the weights of the tasks in set A and the initial weight of T_1 to $\frac{1}{10c}$ and increasing the number of tasks in A to $35c$, where c is a positive integer, in which case $\mathsf{drift}(T_1, \mathsf{d}(T_1^{[1]})) = 5c - 3 + \frac{2}{5c}$. □

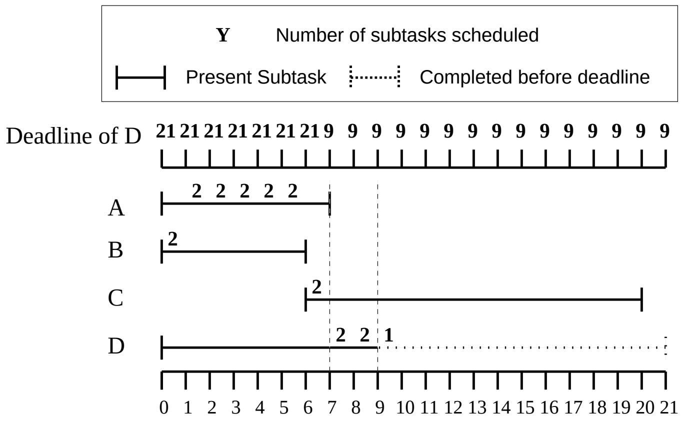
From the generalization of Figure 5.25, the theorem below follows.

Theorem 5.3. PD-LJ *is not fine-grained.*

5.6.2 All EPDF Scheduling Algorithms Incur Drift

Next, we show that any EPDF scheduling algorithm incurs some drift.

Example (Figure 5.26). Consider the example in Figure 5.26, which depicts a two-processor system that consists of a set A of 10 tasks with weight 1/7 that leave at time 7, a set B of two tasks with weight 1/6 that leave at time 6, a set C of two tasks with weight 1/14 that join at time 6, and a set D of five tasks with a weight of 1/21 that increases to 1/3 at time 7. The projected deadlines of tasks in D based on their IDEAL allocation are labeled above the schedule, $\mathcal{I}$. With subtask deadlines defined by $\mathcal{I}$, the deadline for each task in set D changes at time 7 from 21 to 9. The tasks in D have an original deadline of 21



Y Number of subtasks scheduled
Present Subtask
Completed before deadline
Deadline of D 21 21 21 21 21 21 21 9 9 9 9 9 9 9 9 9 9 9 9 9 9
A 2 2 2 2 2
B 2
C 2
D 2 2 1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

Figure 5.26: A two-processor system illustrating that all EPDF algorithms incur drift.

because that is the projected time at which their $\mathcal{I}$ allocations will equal one if their weights do not change. These tasks change their deadlines to 9 at time 7 because the new weight, 1/3, changes the projected time by which their allocations in $\mathcal{I}$ will equal one to time 9. Note that any EPDF algorithm will not schedule the tasks in D until time 7. As a result, a deadline is missed. Notice also that any EPDF algorithm would need to use projections for determining subtask deadlines if we assume no prior knowledge of weight changes. To prevent a deadline miss, the lag-bound range must be shifted, thus incurring drift. ☐

From Figure 5.26, the theorem below follows.

Theorem 5.4. *All* EPDF *algorithms can incur non-zero drift per reweighting event.*

5.6.3 PD-PNH is fine-grained

Finally, we show that PD-PNH is fine-grained. We first show that the rules for reweighting non-heavy-changeable tasks are fine-grained, and then we show that the rules for reweighting heavy-changeable tasks are fine grained.

Non-heavy-changeable tasks. By the definition of drift, in order to prove that PD-PNH is fine-grained, for non-heavy-changeable tasks, we merely need to consider wether a subtask has been halted and the window placement of a task after it is reweighted. Suppose that a non-heavy-changeable task T_i initiates a weight change at t_c. Let t_e be the next time at which T_i enacts a change at or after t_c, assume that T_i releases a subtask at or before t_c, and let

$T_i^{[j]}$ be the last-released such subtask. (If T_i does not release a subtask at or before t_c, then T_i's drift does not change when reweighted at t_c.)

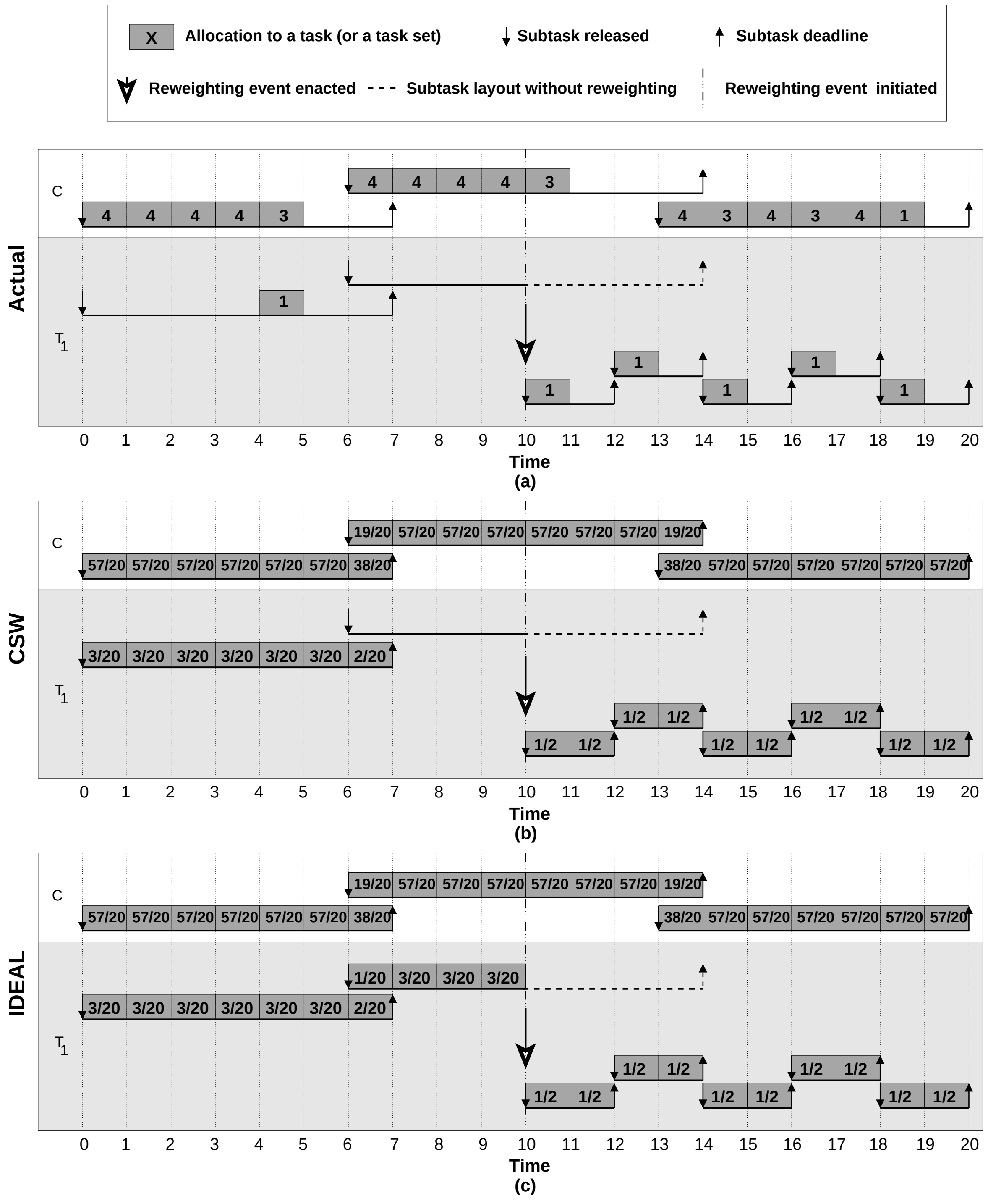
We now show that if the change initiated at t_c is enacted at t_e, then absolute drift increases by at most two. We begin by observing that there are three sources of drift: drift incurred because $T_i^{[j]}$ is halted; drift incurred because $t_c < t_e$; and drift that is incurred because $T_i^{[j+1]}$ is not released at time t_e. (Throughout this discussion we assume that a reweighting event is not canceled, since canceling a reweighting event would decrease the amount of drift incurred.)

Example (Figure 5.27). Consider the four-processor example in Figure 5.27, which consists of a set C of 19 tasks, each with a weight of 3/20, and a task T_1 that increases its weight from 3/20 to 1/2 via Rule P (tie-breaks not resolved by PD^2 go against task T_1). Inset (a) depicts the PD-PNH schedule. Inset (b) depicts the CSW schedule. Inset (c) depicts the IDEAL schedule. Notice that T_1 incurs 1/2 of a quantum of drift over the range [6, 10) because $T_1^{[2]}$ receives no allocation in the CSW schedule. □

Example (Figure 5.28). Consider the four-processor example in Figure 5.28, which consists of a set C of 19 tasks each with a weight of 3/20 and a task T_1 that increases its weight from 3/20 to 1/2 via Rule N (tie-breaks not resolved by PD^2 go against tasks in C). Inset (a) depicts the PD-PNH schedule. Inset (b) depicts the CSW schedule. Inset (c) depicts the IDEAL schedule. Notice that T_1 incurs 1/2 of a quantum of drift over the range [11, 12) because $T_1^{[3]}$'s release is delayed. □

Example (Figure 5.29). Consider the four-processor example in Figure 5.29, which consists of a set C of 19 tasks each with a weight of 3/20 and a task T_1 that decreases its weight from 2/5 to 3/20 via Rule N at time 1. Inset (a) depicts the PD-PNH schedule. Inset (b) depicts the CSW schedule. Inset (c) depicts the IDEAL schedule. Notice that T_1 incurs $-3/20$ of a quantum of drift over the range [1, 4) because the enactment of the weight change occurs after the change is initiated. □

Positive-changeable. We now consider the amount of drift that is incurred based on whether T_i is positive- or negative-changeable at t_c. If T_i is positive-changeable



X Allocation to a task (or a task set)
Subtask released
Subtask deadline
Reweighting event enacted
Subtask layout without reweighting
Reweighting event initiated
Actual
CSW
IDEAL
C
T_1
Time
(a)
(b)
(c)

Figure 5.27: A four-processor illustration of Rule P under PD^2. C is a set of 19 tasks with weight of 3/20. T_1 increases its weight from 3/20 to 1/2 at time 10 via rule P. (Tie-breaks not resolved by PD^2 go against T_1.) **(a)** The PD-PNH schedule. **(b)** The CSW schedule. **(c)** The IDEAL schedule.

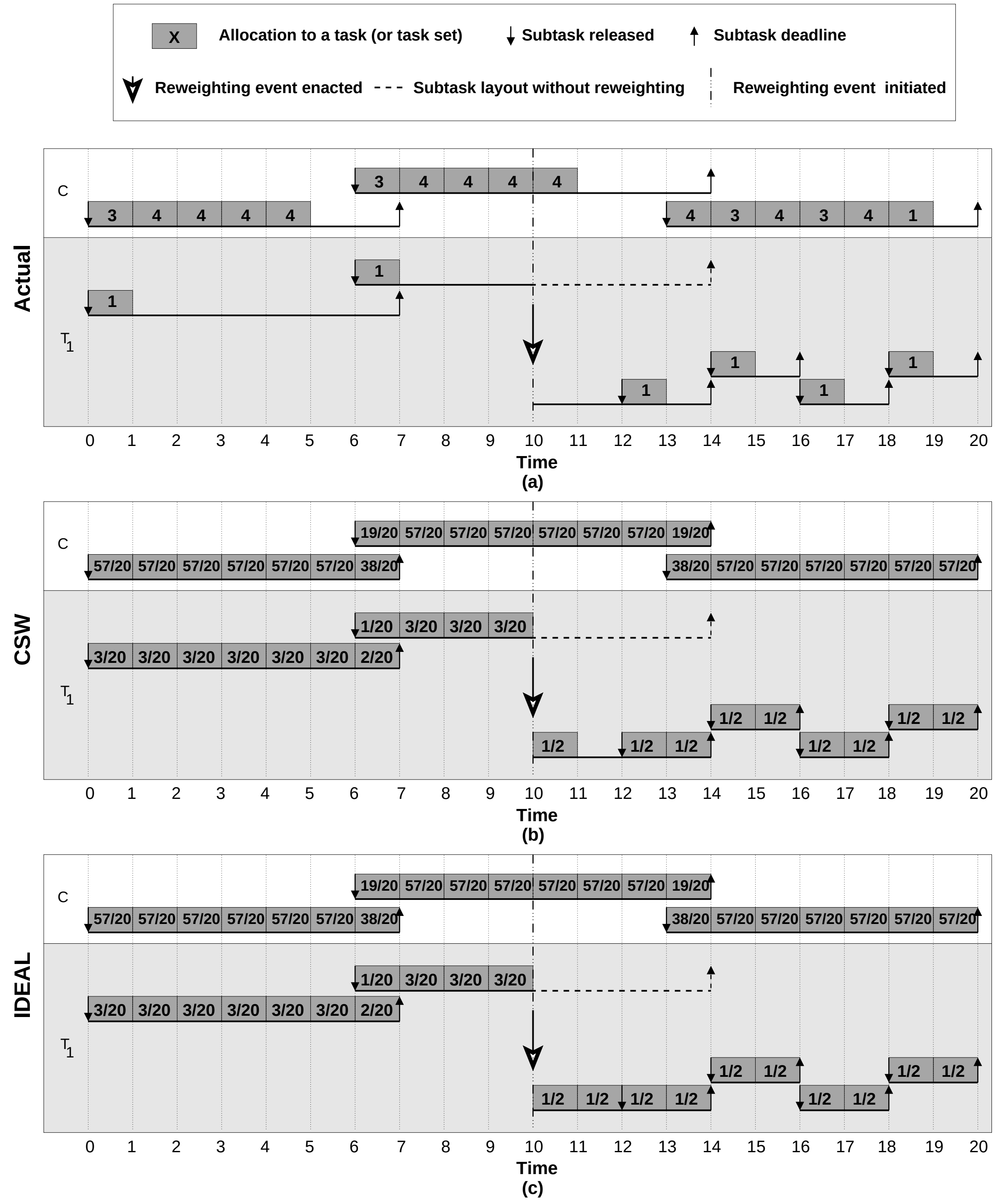


Figure 5.28: A four-processor illustration of Rule N under PD^2. C is a set of 19 tasks with weight of 3/20. (Tie-breaks not resolved by PD^2 go against tasks in C.) T_1 increases its weight from 3/20 to 1/2 at time 10 via rule N. **(a)** The PD-PNH schedule. **(b)** The CSW schedule. **(c)** The IDEAL schedule.

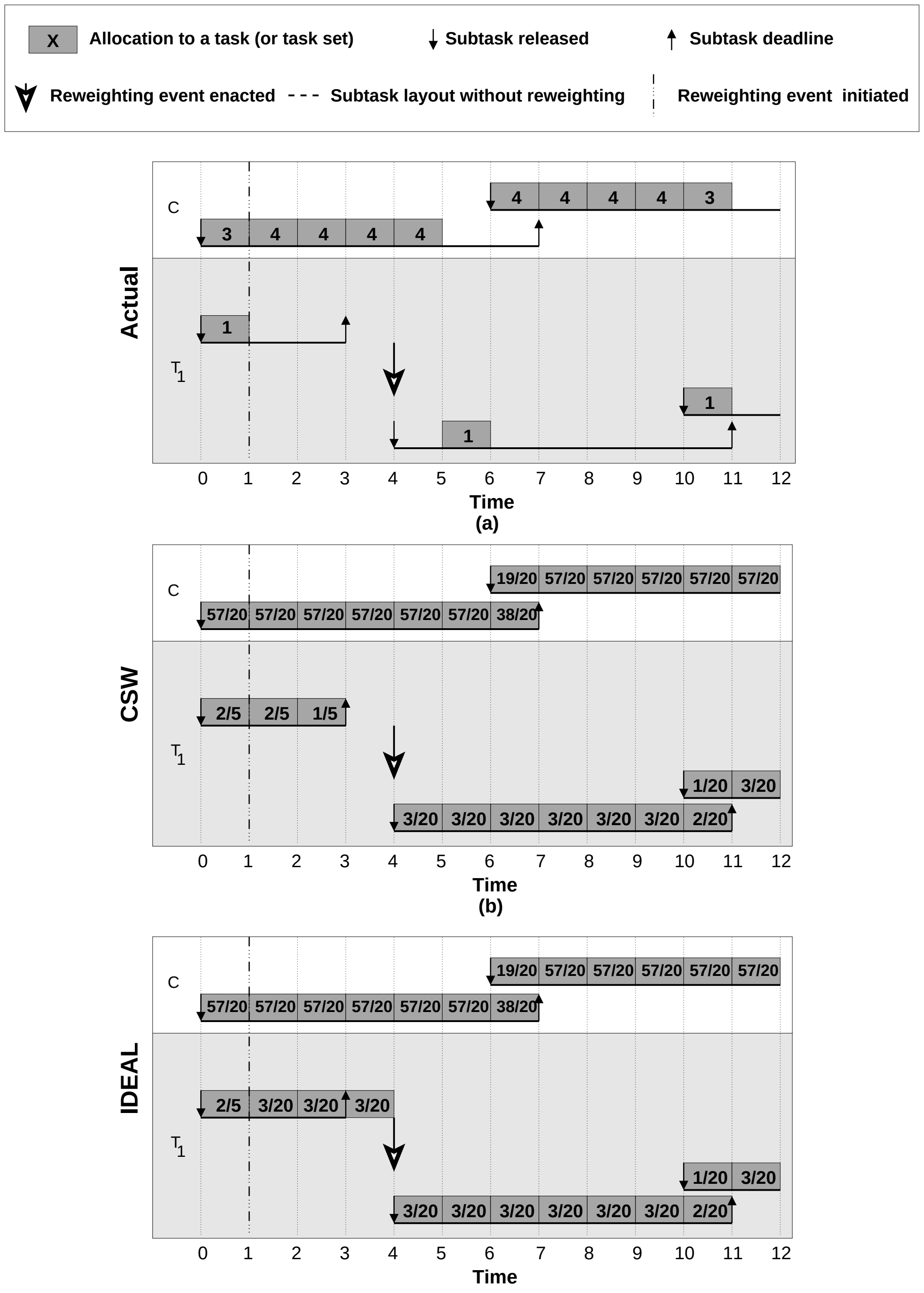


Figure 5.29: A four-processor illustration of Rule N under PD^2. C is a set of 19 tasks with weight of 3/20. T_1 decreases its weight from 2/5 to 3/20 at time 1 via rule N. **(a)** The PD-PNH schedule. **(b)** The CSW schedule. **(c)** The IDEAL schedule.

at t_c, then it changes its weight by Rule P. Thus, $T_i^{[j]}$ is halted and at time $\mathsf{max}(t_c, \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]}))$, the change is enacted and $T_i^{[j+1]}$ is released. Thus, if T_i is positive-changeable, then it may incur drift because $T_u^{[j]}$ is halted and because the change is not immediately enacted.

Since it is trivial to show that T_i incurs at most one quantum of drift because $T_u^{[j]}$ is halted, we focus on showing that T_i incurs at most one quantum of absolute drift because the change is not enacted immediately. Notice that, if $\mathsf{max}(t_c, \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]})) \leq t_c$, then the change is immediately enacted and the only drift T_i incurs is a result of $T_i^{[j]}$ halting. Thus, we assume that $t_c < \mathsf{max}(t_c, \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]}))$.

Since we have assumed that $T_i^{[j]}$ is the last-released subtask of T_i at t_c, it follows that $\mathsf{r}(T_i^{[j]}) \leq t_c$. Thus, since we are assuming that $t_c < \mathsf{max}(t_c, \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]}))$, we have

$$\mathsf{r}(T_i^{[j]}) \leq t_c < \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]}) \leq \mathsf{d}(T_i^{[j-1]}) + \mathsf{b}(T_i^{[j-1]}).$$

We now show that the range $[t_c, \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]}))$ is at most two. Notice that, in order for the range $[t_c, \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]}))$ to be greater than two, $\mathsf{r}(T_i^{[j]}) < \mathsf{d}(T_i^{[j-1]}) - \mathsf{b}(T_i^{[j-1]})$ must hold. However, by Property (V), if $\mathsf{r}(T_i^{[j]}) < \mathsf{d}(T_i^{[j-1]}) - \mathsf{b}(T_i^{[j-1]})$, then $\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}) \leq \mathsf{r}(T_i^{[j]})$. This, in turn, implies that the range $[t_c, \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]}))$ is at most one. Thus, the range $[t_c, \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]}))$ is at most two. This implies that

$$\mathsf{max}(t_c, \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]})) - t_c = t_e - t_c \leq 2. \tag{5.38}$$

Since the maximal weight for any non-heavy changeable task is less than $1/2$, and $t_e - t_c \leq 2$, it follows that $\mathsf{A}(\mathcal{CSW},\, T_i,\, t_c,\, t_e) < 1$ and $\mathsf{A}(\mathcal{I},\, T_i,\, t_c,\, t_e) < 1$. Thus, since the absolute drift incurred over the range $[t_c,\, t_e)$ is given as $|\mathsf{A}(\mathcal{I},\, T_i,\, t_c,\, t_e) - \mathsf{A}(\mathcal{CSW},\, T_i,\, t_c,\, t_e)|$, it follows that T_i incurs up to one additional quantum of absolute drift waiting for the change initiated at t_c to be enacted. Thus, combined with the additional quantum of drift incurred by halting $T_i^{[j]}$, if T_i is positive-changeable at t_c, then it incurs at most two quanta of absolute drift.

Negative-changeable. If T_i is negative-changeable at t_c, then it changes its weight by Rule N. Therefore, $T_i^{[j]}$ does not halt and, as a result, $T_i^{[j]}$ receives the same allocation in $\mathcal{S}W$ and $\mathcal{C}SW$. Thus,

$$\mathsf{C}(\mathcal{S}W, T_i^{[j]}) = \mathsf{C}(\mathcal{C}SW, T_i^{[j]}). \tag{5.39}$$

If T_i increases its weight at t_c, then the change is immediately enacted and $T_i^{[j+1]}$ is released at time $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$. Thus, by (5.39), $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) + \mathsf{b}(T_i^{[j]}) = \mathsf{C}(\mathcal{C}SW, T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$ Since the weight change is immediately enacted, T_i receives the same allocation in $\mathcal{C}SW$ and $\mathcal{I}$ in every time slot in the range $[t_c, \mathsf{C}(\mathcal{C}SW, T_i^{[j]}) - 1)$. (Specifically, if T_i increases its weight to Nw at time t_c, then in every slot in the range $[t_c, \mathsf{C}(\mathcal{C}SW, T_i^{[j]}) - 1)$, T_i is allocated Nw in both $\mathcal{C}SW$ and $\mathcal{I}$.) Thus, since Rule N does not halt $T_i^{[j]}$, the only source of drift that can be incurred is over the range $[\mathsf{C}(\mathcal{C}SW, T_i^{[j]}) - 1, \mathsf{r}(T_i^{[j+1]}))$. Since $\mathsf{r}(T_i^{[j+1]}) = \mathsf{C}(\mathcal{C}SW, T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$, the length of this interval is at most two. Since the weight of a non-heavy-changeable task is less than 1/2, the increase in drift is at most $2 \cdot 1/2$.

For example, in Figure 5.9(a), $\mathsf{A}(\mathcal{C}SW, T_1, 10, 12) = \mathsf{A}(\mathcal{C}SW, T_1, 0, 12) - \mathsf{A}(\mathcal{C}SW, T_1, 0, 10) = 4/2 - 3/2 = 1/2 < 1 = 5/2 - 3/2 = \mathsf{A}(\mathcal{I}, T_1, 0, 12) - \mathsf{A}(\mathcal{I}, T_1, 0, 10) = \mathsf{A}(\mathcal{I}, T_1, 10, 12)$.

If T_i decreases its weight at t_c via Rule N, then $T_i^{[j+1]}$ is released at time $\mathsf{C}(\mathcal{S}W, T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$. Thus, by (5.39), $T_i^{[j+1]}$ is released at time $\mathsf{C}(\mathcal{C}SW, T_i^{[j]}) + \mathsf{b}(T_i^{[j]})$. Since T_i decreases its weight, over the range $[t_c, \mathsf{C}(\mathcal{C}SW, T_i^{[j]}))$, T_i is allocated at most one quantum more in $\mathcal{C}SW$ than in $\mathcal{I}$. Furthermore, over the range $[\mathsf{C}(\mathcal{C}SW, T_i^{[j]}), \mathsf{C}(\mathcal{C}SW, T_i^{[j]}) + \mathsf{b}(T_i^{[j]}))$, T_i is allocated less than 1/2 quanta more in $\mathcal{I}$ than in $\mathcal{C}SW$, since the length of this range is at most one and the weight of a non-heavy changeable task is less than 1/2. Thus, the maximal possible decrease in drift is one and the maximal possible increase in drift is 1/2. For example, in Figure 5.9(b), the drift incurred by changing the weight of T_1 from 2/5 to 3/20 is $-3/20$, i.e., $\mathsf{A}(\mathcal{C}SW, T_1, 1, 4) = \mathsf{A}(\mathcal{C}SW, T_1, 0, 4) - \mathsf{A}(\mathcal{C}SW, T_1, 0, 1) = 5/5 - 2/5 = 3/5 > 9/20 = 17/20 - 2/5 = \mathsf{A}(\mathcal{I}, T_1, 0, 4) - \mathsf{A}(\mathcal{I}, T_1, 0, 1) = \mathsf{A}(\mathcal{I}, T_1, 1, 4)$.

Heavy-changeable tasks. We now show that the drift of a heavy-changeable task is at most five. By the definition of drift, in order to prove that PD-PNH is fine-grained, for heavy-changeable tasks, we only need to consider the window placement of a task T_i, from the reweighting initiation, at time t_c, until $\mathsf{r}(T_i^{[k]})$, where $T_i^{[k]}$ is the first subtask of T_i released at or after $\mathsf{D}(T_i^{[j]})$ and $T_i^{[j]}$ is the last-released subtask of T_i at or before time t_c. (If T_i does not release a subtask at or before t_c or $\mathsf{D}(T_i^{[j]}) \leq t_c$, then T_i is not heavy-changeable.) We bound the drift by showing that the maximal absolute difference between $\mathcal{CSW}$ and $\mathcal{I}$ in allocating to T_i over the interval $[t_c, \mathsf{r}(T_i^{[j+1]}))$ is at most four and that one additional quantum of drift is incurred in the slot $\mathsf{D}(T_i^{[j]}) - 1$. (Notice that, even if the reweighting event initiated at t_c were canceled, then the absolute drift per reweighting event would still be at most five.)

If $\mathsf{d}(T_i^{[j]}) \leq t_c$, then, by Part 2 of Rule H, the weight change is enacted within one quantum. Moreover, by Part 3 of Rule H, $T_i^{[j+1]}$ is released when the change is enacted. Thus, the range $[t_c, \mathsf{r}(T_i^{[j+1]}))$ is at most one quantum long. Since the weight of a heavy-changeable task is less than one, the maximal increase in the absolute value of drift in such a case is less than one over the time range $[t_c, \mathsf{r}(T_i^{[j+1]}))$. If $\mathsf{d}(T_i^{[j]}) > t_c$, then, again by Part 2 of Rule H, the weight change is enacted within four quanta since the maximal window length of any heavy-changeable task is three (and the enactment may be delayed by an additional quantum if the b-bit is one). Moreover, by Part 3 of Rule H, $T_i^{[j+1]}$ is released when the change is enacted. Thus, the range $[t_c, \mathsf{r}(T_i^{[j+1]}))$ is at most four quantum long. As a result, since the weight of a heavy-changeable subtask is less than one, the maximal increase in the absolute drift in such a case is less than four over the time range $[t_c, \mathsf{r}(T_i^{[j+1]}))$. For example, in Figure 5.11(b), T_2 receives an allocation 18/10 in the schedule $\mathcal{I}$ over the range [2, 4), whereas it receives only 2/9 over this same range in the schedule $\mathcal{CSW}$. Thus, T_2 incurs $18/10 - 2/9 \approx 1.58$ units of drift over this region.

Notice that, over the range $[\mathsf{r}(T_i^{[j+1]}), \mathsf{r}(T_i^{[k]}))$, with one possible exception in the slot $\mathsf{D}(T_i^{[j]}) - 1$, the allocation to T_i in $\mathcal{I}$ and $\mathcal{CSW}$ are the same (assuming no additional reweighting events are initiated), despite the fact that the window length of each subtask of T_i released over this time range is two (where $T_i^{[k]}$ is, as above, the first subtask of T_i released at or after $\mathsf{D}(T_i^{[j]})$). The reason for this behavior is two-fold. First, the deadline of a subtask is not used in the definition of $\mathcal{CSW}$, which is based on the pseudo-code in Figure 5.7. Hence, T_i's allocation in the CSW schedule is not affected by the fact that its subtasks may have had their window lengths "artificially" shrunk to two by Rule H. For example, in Figure 5.11, T_2 receives an allocation of 36/10 in both the CSW and IDEAL schedules over the range [4, 8). Second, by Rule H, T_i releases subtasks with the same frequency as a "normal" task. For example, in Figure 5.12, over the range [4, 10), T_2 releases one subtask every three quantum, which is exactly the same frequency at which "normal" tasks with a weight of 1/3 release subtasks.

As we mentioned in the previous paragraph, it is possible that in the slot $\mathsf{D}(T_i^{[j]}) - 1$, T_i's allocation may differ in $\mathcal{I}$ and $\mathcal{CSW}$. The reason for this behavior is that, by Part 4 of Rule H, no subtask of T_i is released at time $\mathsf{D}(T_i^{[j]}) - 1$. Thus, if the definition of release given in Part 3 of Rule H specifies that a job should be released in slot $\mathsf{D}(T_i^{[j]}) - 1$, then drift is incurred since that subtask will not be released. Since the maximal weight of a heavy-changeable task is less than 1, in the slot $\mathsf{D}(T_i^{[j]}) - 1$, less than one additional unit of drift may be incurred.

Also, notice that after $\mathsf{r}(T_i^{[k]})$, T_i behaves as a normal task with its new weight. Thus, the maximal absolute drift that can be incurred by a heavy-changeable task changing its weight is five. For example, in Figure 5.11(b), T_2 receives an allocation of 9/10 in the schedule $\mathcal{I}$ over the range [8, 9), whereas it receives only 4/10 over this same range in the schedule $\mathcal{CSW}$. Thus, T_2 incurs $9/10 - 4/10 = 1/2$ units of drift over this interval. Thus, the total drift incurred as a result of this change is $(\mathsf{A}(\mathcal{I}, T_2, 2, 4) - \mathsf{A}(\mathcal{CSW}, T_2, 2, 4)) + (\mathsf{A}(\mathcal{I}, T_2, 4, 8) - \mathsf{A}(\mathcal{CSW}, T_2, 4, 8)) + (\mathsf{A}(\mathcal{I}, T_2, 8, 9) - \mathsf{A}(\mathcal{CSW}, T_2, 8, 9)) = (18/10 - 2/9) + 0 + (9/10 - 4/10) \approx 2.08$.

One final note: if T_i initiates an additional reweighting event over the range $[\mathsf{r}(T_i^{[j+1]}), \mathsf{D}(T_i^{[j]}) - 1)$, then T_i may experience three additional quanta of drift (as opposed to four quanta). The reason why it experiences an additional three quanta of drift is because the window length of a subtask in the range $[\mathsf{r}(T_i^{[j+1]}), \mathsf{D}(T_i^{[j]}) - 1)$ is at most 2. Thus, this enactment may be delayed for up to three quantum. Since such a change would be an additional reweighting event, this does not impact the per-reweighting-event measurement of drift.

Theorem 5.5. PD-PNH *is fine-grained; moreover, the absolute value of the per-event drift under* PD-PNH *is at most two for non-heavy-changeable tasks and at most five for heavy-*

changeable tasks.

5.7 Lost Utilization

As we mentioned earlier, when a task decreases its weight, there is a delay between when the weight change is initiated and when the capacity gained by this decrease is freed. When a non-heavy-changeable initiates a weight decrease, the capacity is freed when the change is enacted. For example, in Figure 5.9(b), only at or after time 4 could another task could use the capacity of $2/5 - 3/20$ gained by T_2 decreasing its weight. For heavy-changeable tasks, the capacity is freed at the group deadline of the last-released subtask. For example, in the system depicted in Figure 5.12, T_1 cannot use the additional capacity gained by T_2's weight decrease until time 9, which is the group deadline of $T_1^{[2]}$.

This delay is tantamount to "idling" a faction of the system, which could cause some utilization to be lost with respect to a system in which weight changes could be enacted instantly. The *lost utilization* caused by T_i initiating a weight decrease at time t_c is defined as

$$\int_{t_c}^{t_f} \mathsf{LC}(T_i, u)du, \tag{5.40}$$

where $\mathsf{LC}(T_i, t)$ is the capacity that has not been freed by T_i at time t, and t_f is the earliest time t such that T_i frees its capacity, $\mathsf{wt}(T_i,\, t) \geq \mathsf{Ow}$, or T_i initiates another weight decrease.

Example (Figures 5.13 and 5.29). Notice that in Figure 5.29, over the range [1, 4), the lost utilization is $2/5 - 3/20$ at each instant. Thus, the total lost utilization as the result of this weight change is. $\int_1^4 (2/5 - 3/20)dt = 3 \cdot 5/20 = 15/20$. Also notice that, in Figure 5.13, T_2 incurs $13/14 - 1/3$ of lost utilization at each instant over the range [2, 9), and $13/14 - 3/4$ over the range [9, 14). Thus, the total lost utilization incurred by T_2 decreasing its weight at time 2 is $\int_2^{14} (13/14 - \mathsf{wt}(T_2,\, t))dt = \int_2^9 (13/14 - 1/3)dt + \int_9^{14} (13/14 - 3/4)dt \approx 5.059$. Notice that the weight increase at time 9 does not stop T_2 from accruing lost utilization because the new weight after the change (i.e., 3/4) is less than the original scheduling weight (i.e., 13/14). $\square$

Non-heavy-changeable tasks. Notice that, if a task is not heavy-changeable when it initiates a weight change, then the time the capacity is freed is the time the weight change is enacted. Thus, by (5.40), if T_i is non-heavy-changeable when it initiates a weight decrease to Nw at time t_c, then the lost utilization caused by this weight change equals

$$\int_{t_c}^{t_e} \mathsf{LC}(T_i, u)du, \tag{5.41}$$

where t_e is the time that the weight change is enacted or canceled. Notice that, if the change initiated at t_c is canceled at t_e, then this implies that either $\mathsf{wt}(T_i,\, t_e) \geq \mathsf{Ow}$ holds or T_i initiated another weight decrease at t_e. Thus, for non-heavy-changeable tasks (5.41) is equivalent to (5.40).

Let $T_i^{[j]}$ denote the last-released subtask (if any) of T_i at t_c. If $T_i^{[j]}$ does not exist, then the change is enacted immediately, and the lost utilization is zero. If $T_i^{[j]}$ exists but $\mathsf{d}(T_i^{[j]}) \leq t_c$, then the change is enacted within one quantum. Moreover, since the weight of a non-heavy-changeable task is less than 1/2, $\mathsf{LC}(T_i, t) < 1/2$ for any time t. Thus, by (5.41), the maximal amount of lost utilization is 1.

Positive-changeable. If T_i is positive-changeable at t_c, then the change is initiated via Rule P. Thus, the change is enacted at $\mathsf{max}(t_c, \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]}))$. As we proved in (5.38), $\mathsf{max}(t_c, \mathsf{min}(\mathsf{C}(\mathcal{S}W, T_i^{[j-1]}), \mathsf{d}(T_i^{[j-1]})) + \mathsf{b}(T_i^{[j-1]})) - t_c \leq 2$. Thus, since the weight of a non-heavy-changeable task is less than 1/2, by (5.41), the lost utilization T_i is negative-changeable at t_c is less than one.

Negative-changeable. If T_i is negative-changeable at t_c, then by Rule N, the weight change is enacted or canceled by $\mathsf{C}(\mathcal{S}W, T_i^{[j]})+\mathsf{b}(T_i^{[j]})$, i.e., $t_e \leq \mathsf{C}(\mathcal{S}W, T_i^{[j]})+\mathsf{b}(T_i^{[j]})$. By the definition of completed, $\int_{\mathsf{r}(T_i^{[j]})}^{\mathsf{C}(\mathcal{S}W, T_i^{[j]})} \mathsf{Swt}(T_i, u)du = 1$. Since $T_i^{[j]}$ is the last-released subtask of T_i at t_c, $\mathsf{r}(T_i^{[j]}) \leq t_c$. Thus, since $t_e - \mathsf{b}(T_i^{[j]}) \leq \mathsf{C}(\mathcal{S}W, T_i^{[j]})$, $\mathsf{Swt}(T_i, t) < 1/2$ for every $t \in [t_c,\, t_e)$, and by assumption $t_c \leq t_e$,

$$\int_{t_c}^{t_e} \mathsf{Swt}(T_i, u)du = \int_{t_c}^{t_e - \mathsf{b}(T_i^{[j]})} \mathsf{Swt}(T_i, u)du + \int_{t_e - \mathsf{b}(T_i^{[j]})}^{t_e} \mathsf{Swt}(T_i, u)du < 1 + 1/2. \tag{5.42}$$

Since T_i decreases its weight at t_c, it follows that for every $t \in [t_c, t_e)$, $\mathsf{LC}(T_i, t) \leq \mathsf{Swt}(T_i, t)$. Thus, by (5.42), $\int_{t_c}^{t_e} \mathsf{LC}(T_i, u)du < 3/2$. Thus, by (5.41), if T_i is negative-changeable at t_c, then the lost utilization is less than 3/2. Notice that, for non-heavy-changeable tasks, the lost utilization is closely related to the drift incurred.

Heavy-changeable tasks. For a heavy-changeable task, the amount of lost utilization could be substantially larger than the drift incurred. Let $T_i^{[j]}$ denote the last-released subtask of a heavy-changeable task T_i at t_c. (Notice that, $T_i^{[j]}$ must exist, because by the definition of heavy-changeable, a T_i must have released a subtask at or before t_c that had a group deadline after t_c.) Notice that, by Part 1 of Rule H, if T_i decreases its weight from Ow to Nw at t_c, then the capacity is freed at time $\mathsf{D}(T_i^{[j]})$. Thus, (5.40), i.e., the lost utilization incurred by this change, is upper-bounded by

$$\int_{t_c}^{\mathsf{D}(T_i^{[j]})} (\mathsf{Ow} - \mathsf{Nw})dt. \tag{5.43}$$

Before continuing, notice that, a heavy task must have at least one group deadline every period. As a result, $\mathsf{D}(T_i^{[j]}) - t_c \leq \mathsf{p}(T_i)$. Moreover, recall that the weight of T_i is given by $\mathsf{wt}(T_i) = \mathsf{e}(T_i)/\mathsf{p}(T_i)$. Thus, (5.43) can be upper-bounded by

$$(\mathsf{Ow} - \mathsf{Nw}) \cdot (\mathsf{D}(T_i^{[j]}) - t_c) \leq \mathsf{Ow} \cdot \mathsf{p}(T_i) \leq \mathsf{e}_{\mathsf{max}}(T_i).$$

The above bound still holds even if T_i initiates additional changes over the range $[t_c, \mathsf{D}(T_i^{[j]}))$.

For example, in the system depicted in Figure 5.12, the amount of lost utilization caused by T_2 decreasing its weight from 8/9 is $(\mathsf{Ow} - \mathsf{Nw}) \cdot (\mathsf{D}(T_2^{[2]}) - t_c) = (8/9 - 1/3) \cdot (9 - 2) = 35/9 \approx 3.88$. Since under Pfair scheduling, all execution times and periods are an integral number of quanta, the only way T_2 could have a weight of 8/9 is if its execution time is at least 8. Thus, the amount of lost utilization caused by this decrease is at least ≈ 4.12 less than the execution time for T_2.

Theorem 5.6. *The lost utilization per reweighting event initiation under under* PD-PNH *is at most* 3/2 *for non-heavy-changeable tasks and at most* $\mathsf{e}_{\mathsf{max}}(T_i)$ *for heavy-changeable tasks.*

5.8 Conclusion

In this chapter, we presented the AIS task model as well as the rules for reweighting a task under the PD^2 scheduling algorithm. In addition, by using techniques borrowed from (Srinivasan and Anderson, 2005), we proved that no subtask misses its deadline using our reweighting rules. Also, we proved that the absolute drift that can be incurred per reweighting event is at most five. Finally, we proved that the amount of lost utilization caused by a weight decrease is at most $3/2$ for non-heavy changeable tasks and at most $\mathsf{e}_{\mathsf{max}}(T_i)$ for heavy-changeable tasks.

CHAPTER 6

AGEDF*

In this section, we present the *adaptable* GEDF (AGEDF) scheduling algorithm, which extends the GEDF algorithm by using feedback techniques in order to determine when a task should enact a weight change.

6.1 Adaptable Service Level Tasks

Before discussing the AGEDF scheduling algorithm, we first introduce the *adaptable service-level task model*, which is based on a task model presented in (Lu et al., 2002) and extends the notion of a *sporadic* task system in two major ways. First, *worst-case* execution times are not assumed. Second, each task T_i has a set of *service levels*, denoted $\mathsf{SL}(T_i)$, each of which represents a different level of QoS for T_i, and a *weight translation function*, denoted $\mathsf{g}(T_i, e, k, q)$, which is used to compute the weight of T_i at different service levels, as explained below.

The k^{th} service level of $\mathsf{SL}(T_i)$ is defined by an *importance value*, $\mathsf{v}(T_i, k)$, a *period*, $\mathsf{p}(T_i, k)$, and a *code segment*. Without loss of generality, we assume that the service levels in $\mathsf{SL}(T_i)$ are indexed from 1 to $|\mathsf{SL}(T_i)|$ by increasing importance value, where $|\mathsf{SL}(T_i)|$ is the number of elements in $\mathsf{SL}(T_i)$. The importance value represents some user-defined notion of "goodness," where 0.0 represents a service level that has no value and 1.0 represents the maximal possible value associated with any service level of any task in the system.

At any point in time t, one service level in $\mathsf{SL}(T_i)$ is said to be the *functional service level*

* Contents of this chapter previously appeared in preliminary form in the following paper:
Block, A., Brandenburg, B., Anderson J., and Quint, S. (2008). An Adaptive Framework for Multiprocessor Real-Time Systems. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 23–33.

of T_i. The index of the functional service level of T_i at time t is denoted $\mathsf{f}(T_i, t)$. For now, we assume that the functional service level of a task T_i does not change within $(\mathsf{r}(T_i^j), \mathsf{d}(T_i^j))$ for any job T_i^j of T_i. In Section 6.2.3, we discuss how to change the functional service level of a task at any time. If k is the functional service level at $\mathsf{r}(T_i^j)$, then T_i^j is said to be *functioning at service level* k. If T_i^j is functioning at service level k, then both $\mathsf{r}(T_i^{j+1}) \geq \mathsf{r}(T_i^j) + \mathsf{p}(T_i, k)$ and $\mathsf{d}(T_i^j) = \mathsf{r}(T_i^j) + \mathsf{p}(T_i, k)$ hold. We consider a task T_i to be *active* at time t if there exists a job T_i^j (called T_i*'s active job*) such that $t \in [\mathsf{r}(T_i^j), \mathsf{d}(T_i^j))$. We use $\mathsf{ACT}(t)$ to denote the set of active jobs at time t.

The code segment associated with the k^{th} service level is the code segment that a job T_i^j will execute if T_i^j is functioning at service level k. Depending on the specific application, there are numerous different methods for defining such code segments. For some applications, each service level may execute the same code segment, and the only difference between service levels is the period. For other applications, the difference between service levels may be something as simple as the number of iterations in a loop, while for others, each service level may use entirely different code. As we discuss in Section 6.2, how the code segment is implemented will impact the efficacy of AGEDF at adapting tasks.

Just as for sporadic tasks, the value of $\mathsf{Ae}(T_i^j)$ denotes the amount of time for which T_i^j is actually scheduled. The *actual weight of a job* T_i^j, denoted $\mathsf{Aw}(T_i^j)$, represents the actual fraction of a processor that T_i^j requires and is defined by $\mathsf{Aw}(T_i^j) = \mathsf{Ae}(T_i^j)/\mathsf{p}(T_i^j, k)$, where T_i^j is functioning at service level k. Just as with sporadic tasks, we assume that the value of $\mathsf{Ae}(T_i^j)$ (and by extension $\mathsf{Aw}(T_i^j)$) is not known until T_i^j finishes execution.

As we discuss in Section 6.2.1, since the actual weight of a job is not known until it completes, AGEDF uses an *estimated weight* for incomplete jobs, denoted $\mathsf{Ew}(T_i^j)$. When AGEDF calculates the estimated weight for a job T_i^j, it does so for a specific service level (typically, the same service level at which T_i^{j-1} was functioning). The weight translation function is used to map the estimated weight as calculated by AGEDF for a specific job T_i^j functioning at a specific service level to what the estimated weight of T_i^j would be if it functioned at a *different* service level. Specifically, if e is the estimated weight of T_i^j assuming that T_i^j is functioning at service level k, then the weight translation function, $\mathsf{g}(T_i, e, k, q)$,

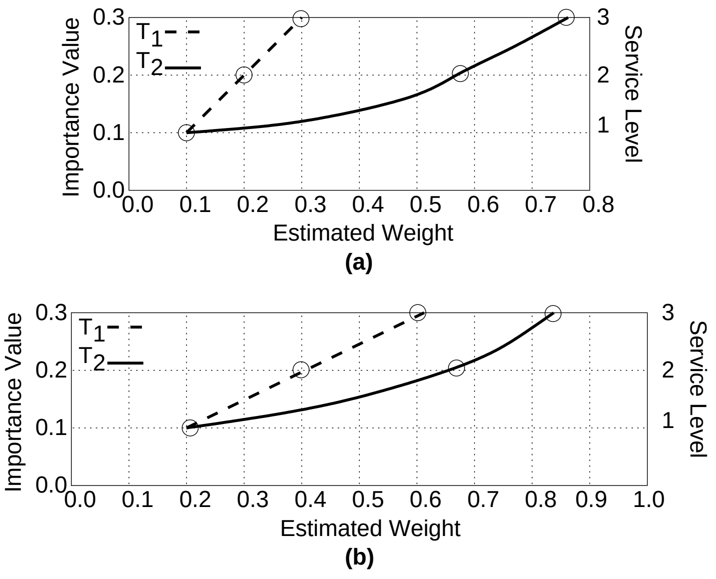


Figure 6.1: Estimated weight vs. importance value/service level for two tasks. **(a)** $e = 0.1$. **(b)** $e = 0.2$.

returns the estimated weight of T_i^j if it *were to be functioning at* q^{th} *service level instead of the* k^{th} *service level.*

Example (Figure 6.1). Consider the example in Figure 6.1, which depicts the estimated weight vs. importance value/service level for two tasks: T_1 and T_2, each of which have three service levels with importance values of 0.1, 0.2, and 0.3. For T_1, $\mathsf{g}(T_1,\, e,\, 1,\, 2) = 2e$ and $\mathsf{g}(T_1,\, e,\, 1,\, 3) = 3e$, and for T_2, $\mathsf{g}(T_2,\, e,\, 1,\, 2) = e^{1/4}$ and $\mathsf{g}(T_2,\, e,\, 1,\, 3) = e^{1/8}$, where e is the estimated weight while functioning at service level one. Inset (a) depicts the scenario where $e = 0.1$ for both tasks. Inset (b) depicts the scenario where $e = 0.2$ for both tasks. Notice that, in inset (a), if $e = 0.1$, $k = 1$, and $q = 3$, then $\mathsf{g}(T_1,\, e,\, k,\, q) = 0.3$, which is the estimated weight of a job of T_1 if it had been calculated for the third service level instead of the first. Also, for T_2, $\mathsf{g}(T_2,\, e,\, k,\, q) \approx 0.75$, which is the estimated weight of a job of T_2 if it had been calculated for the third service level instead of the first. $\square$

As we discuss in Section 6.2.2, the weight translation function is used to determine the effect on the system caused by changing the functional service level of a task. We make only two assumption about the behavior of $\mathsf{g}(T_i,\, e,\, k,\, q)$: if $q < k$, then $\mathsf{g}(T_i,\, e,\, k,\, q) \leq e$; and if $\mathsf{g}(T_i,\, e_1,\, k,\, q) = e_2$, then $\mathsf{g}(T_i,\, e_2,\, q,\, k) = e_1$. It is important to note that the function $\mathsf{g}(T_i, e, k, q)$ can return approximate values; however, the accuracy of $\mathsf{g}(T_i, e, k, q)$ will impact the performance of AGEDF's optimizer component, which determines the functional service level of each task. Like service levels and code segments, the weight translation function is defined by the application developer and can be determined empirically.

The primary difference between the task model presented in (Lu et al., 2002) and our task model is that in (Lu et al., 2002) each service level of a task T_i has a *static* notion of "estimated weight" that represents the nominal fraction of a processor required by T_i. Statically assigning an estimated weight to a task implies that the task has a typical behavior and that if it requires a smaller or larger fraction of a processor, then such a scenario is an anomaly. While this may be true for many applications, for systems like Whisper and VEC, predetermining the nominal weight of a task can be difficult if not impossible. Thus, as we will discuss in Section 6.2, rather than statically determining estimated weights, AGEDF will dynamically calculate the estimated weight for each job.

6.2 The AGEDF Scheduling Algorithm

We now present the AGEDF scheduling algorithm and its three components: the *predictor* (Section 6.2.1), which uses feedback-based techniques to estimate the actual weights of future jobs; the *optimizer* (Section 6.2.2), which given estimated job weights, attempts to determine an optimal set of functional service levels; and several *reweighting rules* (Section 6.2.3), which are used to change the functional service level of a task to match that chosen by the optimizer. In the following, we assume that AGEDF is used on an M-processor system.

The major components of AGEDF are depicted in Figure 6.2. At a high level, these components function as follows.

- *At each instant*, the M pending jobs with the smallest deadlines are scheduled.
- *At T_i^j's completion*, the predictor is used to estimate the weight for the next job release of T_i. If maintaining a constant weight is important, then the reweighting rules may change T_i^{j+1}'s functional service level.
- *After some user-specified threshold*, the optimization component is run to determine

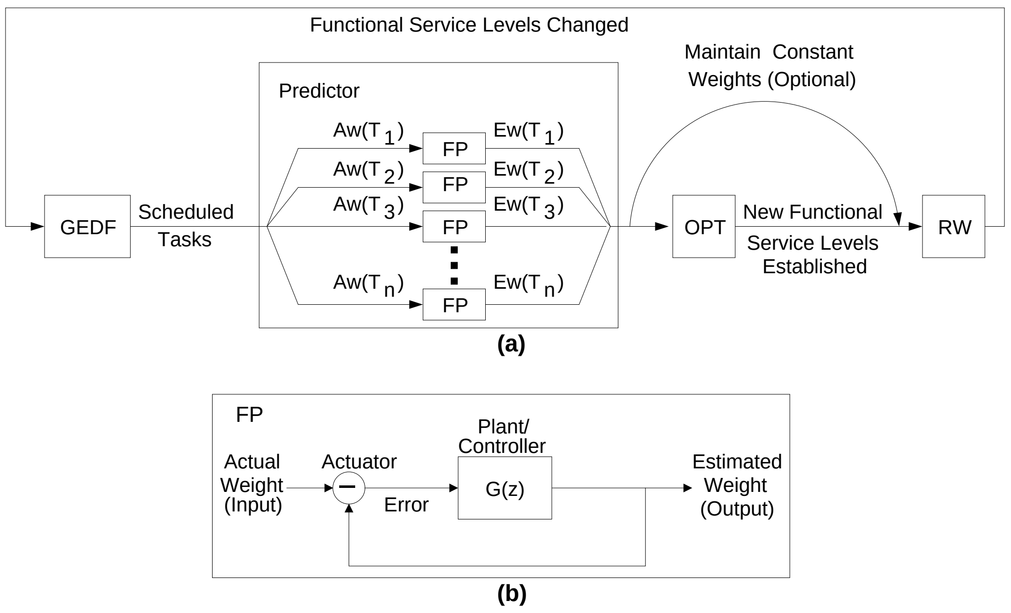


Figure 6.2: **(a)** The AGEDF scheduling algorithm. **(b)** The model of AGEDF's feedback component.

new service levels for each task. Then, the following two steps are performed. First, if some tasks require an estimated weight *decrease*, then the reweighting rules are used to change the service levels of those tasks. This creates spare capacity in the system. Second, as the spare capacity created by weight decreases becomes available, if some tasks require an estimated weight *increase*, then the reweighting rules are used to change the service levels of those tasks.

It is worthwhile to note that the optimization component (and hence large-scale changes to task functional service levels) is only executed after some user-specified threshold. We offer some guidelines for choosing this threshold in Section 6.2.4.

6.2.1 The Feedback Predictor

Before continuing, we briefly review the basics of *feedback systems* (a thorough review of feedback systems can be found in Section 2.4). Most feedback systems consist of the following components, which are labeled in the model of our system in Figure 6.2(b): the *input value*, the *output value*, the *actuator*, the *error*, the *plant*, and the *controller*. The input value is the reference value for the system, while the output value is value computed by the system.

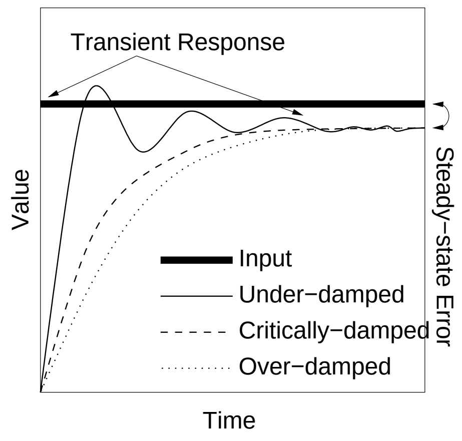


Figure 6.3: An example of an over-damped, under-damped, and critically-damped feedback system responding to a step input.

The actuator calculates the error by subtracting the output from the input. The plant is the system we wish to control. The controller modifies the input to change the behavior of the output.

The performance of a feedback system is measured in terms of *transient response*, *steady-state error*, and *stability*. The transient response of a system is the initial output of the system to a change in input, as depicted in Figure 6.3. The steady-state error denotes the difference between the output and the input of the system as time increases (also depicted in Figure 6.3). A system is considered to be *stable* if every bounded input causes the system's steady-state error to be bounded.

It is worthwhile to note that while feedback-based techniques are primarily used to control the behavior of a plant for which the (reference) input is known, another viable use for such techniques is to *predict* future values of a changing and unknown input. The design of such a system is exactly the same as the typical feedback system, *except that the feedback loop does not directly impact the behavior of the system.* In such a system, the transient response describes the initial accuracy of predictions after there has been a change in the input, and the steady-state error describes the difference between the predicted and actual values as system time increases.

The feedback predictor. Since the predictor in AGEDF uses feedback-based techniques to predict the weight of future jobs instead of using a simpler approach, such as setting

$\mathsf{Ew}(T_i^j) = \mathsf{Aw}(T_i^{j-1})$, the predictor both produces values of $\mathsf{Ew}(T_i^j)$ that are less susceptible to ephemeral fluctuations in the workload and is capable of closely tracking trends in the actual weight (e.g., when the actual weight of the task changes at a constant rate). Using a feedback loop to predict the weight of future jobs is similar to the approach in (Abeni et al., 2002), described earlier in Section 2.4.

As depicted in Figure 6.2(a), in the predictor, each task has its own feedback loop. Also, as depicted in Figure 6.2(b), for each feedback loop, the input is the actual weight; the output is the estimated weight; the error is the actual weight minus the estimated weight; and the controller is a *proportional-integral* (PI) controller that uses information about the current error and the sum of all previous errors in order to calculate a new estimated weight. Specifically, the controller is defined as

$$\mathsf{Ew}(T_i^{j+1}) = a \cdot \epsilon(T_i^j) + b \sum_{k=1}^{k=j-1} \epsilon(T_i^k), \tag{6.1}$$

where $\mathsf{Ew}(T_i^1) = 0$, $\epsilon(T_i^j) = \mathsf{Aw}(T_i^j) - \mathsf{Ew}(T_i^j)$, and both a and b are user-defined values that we discuss shortly. Taking the Z-transform of (6.1) and rearranging, we get

$$\mathsf{G}(z) = \frac{a(z-c)}{z(z-1)}, \tag{6.2}$$

where $c = (a-b)/a$. As discussed in Section 2.4, in control theory parlance, (6.2) is called the *open-loop transfer function* because it represents the behavior of the controller *ignoring the feedback loop*. The *closed-loop transfer function*, which incorporates both the behavior of the controller and feedback loop is given by

$$\mathsf{H}(z) = \frac{\mathsf{G}(z)}{1+\mathsf{G}(z)} = \frac{a(z-c)}{z^2 + (a-1)z - ac}. \tag{6.3}$$

From the above equation, the predictor has a *closed-loop zero* (the value of z for which $H(z) = 0$) at $z = c$, and *closed-loop poles* (the values of z for which $H(z)$ is undefined) at

$$\frac{(1-a) \pm \sqrt{(a-1)^2 + 4ac}}{2}. \tag{6.4}$$

Because the predictor has two closed-loop poles, it is a *second-order system.* This is the reason why we chose to use a PI controller instead of a *proportional-integral-derivative* (PID) controller. Since PID controllers are *third-order systems*, i.e., such systems have three poles, the transient response analysis is substantially more complex. In fact, the typical means for determining the transient response of a third-order system is to approximate it as a second-order system.

Feedback characteristics. We use standard techniques, which are discussed in detail in Section 2.4, for analyzing the feedback characteristics of our system. We begin by rewriting (6.4) as

$$\mathcal{P}_1 = \frac{(1-a)+\sqrt{(a-1)^2+4ac}}{2} \tag{6.5}$$

$$\mathcal{P}_2 = \frac{(1-a)-\sqrt{(a-1)^2+4ac}}{2}. \tag{6.6}$$

We let $\mathcal{P}_m$ denote the pole from (6.5) and (6.6) that is the farthest from the origin. Also, we use $\mathcal{R}(\mathcal{P})$ and $\theta(\mathcal{P})$ to denote, respectively, the radius and angle (in radians) of the pole $\mathcal{P}$ in polar-complex form.

Stability. By using both the open- and closed-loop transfer functions and the closed-loop poles, we can discuss how setting the values of a and c impact stability, transient response, and steady-state error. First, we address the stability of the system. The system is *stable* if both closed-loop poles are within the unit circle in the complex plane, i.e., $\mathcal{R}(\mathcal{P}_m) < 1$. The system is *unstable* if either closed-loop pole is outside the unit circle, i.e., $\mathcal{R}(\mathcal{P}_m) \geq 1$. The system is in a state called *marginally stable*, in which case the output neither converges nor diverges, if one pole is on the unit circle and the other is within it, i.e., $\mathcal{R}(\mathcal{P}_m) = 1$.

Transient response. The transient response is usually evaluated by the behavior of the output when the system incurs a *step input*, i.e., the input suddenly increases to a given value. Since feedback systems use previous results to predict future results, a step input represents the worst-case scenario—a sudden change from one value to a substantially different value.

The transient response of a second-order system is characterized by both the *settling time* (i.e., the time it takes for the output to attain and stay within 2% of its steady-state value), and whether the system is *over-damped*, *under-damped*, or *critically-damped* (depicted in Fig. 6.3). The settling time (where time is measured in terms of job releases) of the system is given by the standard formula

$$\left\lceil \frac{-4}{ln\left(\mathcal{R}(\mathcal{P}_m)\right)} \right\rceil . \tag{6.7}$$

A lower settling time will improve the system's capacity to respond to sudden changes in the execution time of a task; however, it will also make the system more susceptible to ephemeral fluctuations in the execution time of tasks. As a result, a low settling time may be undesirable for the purposes of predicting future values.

If a system is over-damped, then the output will never "overshoot" the input for a step input. If a system is under-damped, then the output will overshoot the input for a step input. For under-damped systems, the *percent overshoot* is an additional characteristic of transient response. If the system is critically-damped, then the settling time is as small as possible without causing the output to overshoot the input. Whether a system is under-, over-, or critically-damped depends on the location of the closed loop poles in (6.5) and (6.6). If both poles are unique and both poles are real then the system is over-damped. If both poles have the same radius and are real, then the system is critically-damped. Otherwise, the system is under-damped.

For under-damped systems, the percent overshoot is given by

$$e^{-(\zeta\pi/\sqrt{1-\zeta^2})} \cdot 100, \tag{6.8}$$

where ζ is a value called the *damping ratio* and is given by

$$\zeta = \frac{-ln\left(\mathcal{R}(\mathcal{P}_m)\right)}{\sqrt{\theta(\mathcal{P}_m)^2 + ln^2\left(\mathcal{R}(\mathcal{P}_m)\right)}} \tag{6.9}$$

Steady-state error. The steady-state error of a system is measured based on the system's response to a step and/or a *ramp* input. The ramp input simulates an input that constantly increases by a rate of $\mathcal{T}$ per job release. The steady-state error is based on the *system type*, which is given by the power to which $(z-1)$ is raised in the denominator of (6.2). Since our system has a system type of one, the steady-state error for the step input is zero, and the steady-state error of the ramp input is given by

$$\frac{\mathcal{T}}{\lim_{z \to 1}(z-1)\mathsf{G}(z)} = \frac{\mathcal{T}}{a(1-c)}. \tag{6.10}$$

The fact that a PI controller has zero steady-state error for a step response is the reason why we chose a PI controller instead of a *proportional-derivative* (PD), which would have a superior transient response but would have a non-zero value for a step input (i.e., if the actual weight is constant, a PD controller would still have error).

Putting it together. Now that we have established formulas for stability, transient response, and steady-state error, it is possible to choose values for a and c (and thus implicitly set the value of b) that satisfy our design objectives. Suppose, for example, that we wish to construct a critically-damped system with a settling time of five job releases. From the definitions of critically-damped and settling time, it is not difficult to calculate that if $a = 0.10206228$ and $c = -1.975$, then these two design objectives are achieved. Specifically, in this case, by (6.5) and (6.6), the closed-loop poles are

$$\begin{aligned} \mathcal{P}_1 &= \frac{(1-a) + \sqrt{(a-1)^2 + 4ac}}{2} \approx 0.449 \\ \mathcal{P}_2 &= \frac{(1-a) - \sqrt{(a-1)^2 + 4ac}}{2} \approx 0.449. \end{aligned}$$

Thus, $\mathcal{P}_1 \approx \mathcal{P}_2$, which implies that the system is critically-damped (or at least close to it). In addition, by (6.7), the settling time (in terms of number of jobs) is

$$\left\lceil \frac{-4}{ln\left(\mathcal{R}(\mathcal{P}_m)\right)} \right\rceil = \lceil 4.997 \rceil = 5.$$

Moreover, by (6.10), we can calculate that the steady-state error of this system for a ramp input that increases at a constant rate of $\mathcal{T}$ per job release is

$$\frac{\mathcal{T}}{\lim_{z\to 1}(z-1)\mathsf{G}(z)} = \frac{\mathcal{T}}{a(1-c)} \approx 3.29 \cdot \mathcal{T}.$$

However, if we wish to construct an under-damped system with a settling time of five job releases, and a percent overshoot of approximately 10%, then it is not difficult to show that $a = 1.4008$ and $c = -0.1439$ satisfy these design objectives. Specifically, by (6.5) and (6.6), the closed-loop poles are

$$\begin{aligned} \mathcal{P}_1 &= \frac{(1-a)+\sqrt{(a-1)^2+4ac}}{2} \approx -0.2004 + 0.4018i \\ \mathcal{P}_2 &= \frac{(1-a)-\sqrt{(a-1)^2+4ac}}{2} \approx -0.2004 - 0.4018i. \end{aligned}$$

Because $\mathcal{P}_1$ and $\mathcal{P}_2$ are complex values, it follows that the system is under-damped. In addition, because both $\mathcal{P}_1$ and $\mathcal{P}_2$ are equidistance from the origin, either $\mathcal{P}_m \approx -0.2004 + 0.4018i$ or $\mathcal{P}_m \approx -0.2004 - 0.4018i$ holds (the results are the same either way). Because $\mathcal{R}(\mathcal{P}_m)$ and $\theta(\mathcal{P}_m)$ denote, respectively, the radius and angle (in radians) of the pole $\mathcal{P}_m$ in polar-complex form, if we set $\mathcal{P}_m \approx -0.2004 + 0.4018i$, then $\mathcal{R}(\mathcal{P}_m) \approx 0.449$ and $\theta(\mathcal{P}_m) \approx -1.108$.

Thus, by (6.7), the settling time (in terms of number of jobs) is given by

$$\left\lceil \frac{-4}{ln\left(\mathcal{R}(\mathcal{P}_m)\right)} \right\rceil = \lceil 4.995 \rceil = 5.$$

Furthermore, by (6.9), we can calculate the damping ratio as

$$\zeta = \frac{-ln\left(\mathcal{R}(\mathcal{P}_m)\right)}{\sqrt{\theta(\mathcal{P}_m)^2 + ln^2\left(\mathcal{R}(\mathcal{P}_m)\right)}} \approx 0.5857.$$

Thus, by (6.8), we can calculate the percent overshoot as

$$e^{-(\zeta\pi/\sqrt{1-\zeta^2})} \cdot 100 \approx 10.33\%.$$

Moreover, by (6.10), we can calculate that the steady-state error of this system for a ramp input that increases at a constant rate of $\mathcal{T}$ per job release as

$$\frac{\mathcal{T}}{\lim_{z \to 1}(z-1)\mathsf{G}(z)} = \frac{\mathcal{T}}{a(1-c)} \approx 0.624 \cdot \mathcal{T}.$$

6.2.2 Optimization

As mentioned above, the optimization component of AGEDF uses the estimated weights of tasks in order to choose service levels for each task. There are a variety of different methods for implementing this component depending on what metric the user wants to optimize and the behavior of $\mathsf{g}(T_i,\, e,\, k,\, q)$.

For example, suppose the objective is to optimize the total importance value in the system. In this case, if the relationship between the importance value and weight is linear (like T_1 in Figure 6.1), then an approximate solution for this objective could be achieved by assigning the highest service level possible to those tasks with the highest *value density*, as given by,

$$\frac{\mathsf{v}(T_i,\, |\mathsf{SL}(T_i)|) - \mathsf{v}(T_i,\, 1)}{\mathsf{g}(T_i,\, \mathsf{Ew}(T_i^j),\, k,\, |\mathsf{SL}(T_i)|) - \mathsf{g}(T_i,\, \mathsf{Ew}(T_i^j),\, k,\, 1)}, \tag{6.11}$$

while ensuring at least every task receives its minimum service level and the system is not over-utilized. In this approach, the value $\mathsf{v}(T_i,\, |\mathsf{SL}(T_i)|) - \mathsf{v}(T_i,\, 1)$ denotes by how much T_i's importance value improves by changing from the lowest service level to the highest service level. Additionally, the value $\mathsf{g}(T_i,\, \mathsf{Ew}(T_i^j),\, k,\, |\mathsf{SL}(T_i)|) - \mathsf{g}(T_i,\, \mathsf{Ew}(T_i^j),\, k,\, 1)$ represents by how much T_i's weight would have to be changed to improve its service level from the lowest service level to the highest service level. Notice that, for two tasks T_1 and T_2, if T_1 has a larger value for (6.11) than T_2, and both tasks had their estimated weight increase by the same amount, then T_1's importance value would improve more than T_2's. This approach is similar to the *highest-value-density-first* approach used in (Lu et al., 2002).

Example (Figure 6.4). Consider the example in Figure 6.4, which depicts the estimated weight vs. importance value/service level for two tasks: T_1 and T_2, each of which has three service levels with importance values of 0.1, 0.2, and 0.3. Inset (a) depicts the scenario

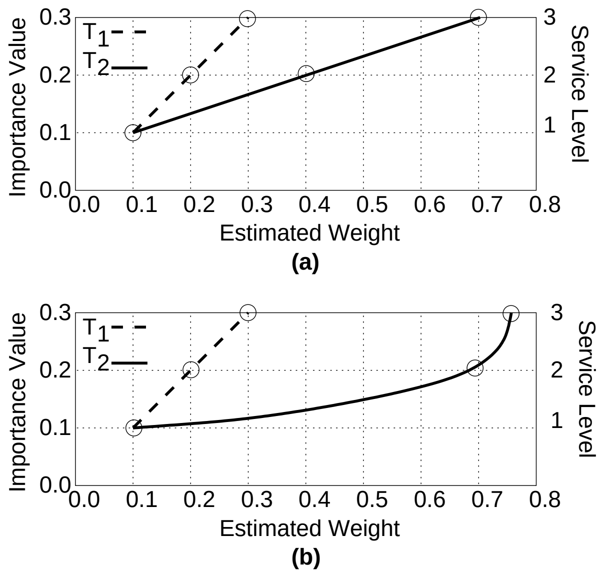


Figure 6.4: Estimated weight vs. importance value/service level for two tasks. **(a)** This relationship is linear for both tasks. **(b)** This relationship is linear for T_1 and non-linear for T_2.

where both tasks have a linear relationship between the estimated weight and the importance value/service level. Inset (b) depicts the scenario where the estimated weight and the importance value/service level relationship is linear for T_1 and non-linear for T_2. Notice that, in inset (a), the value densities for T_1 and T_2 are, respectively, $\frac{0.2}{0.2} = 1$ and $\frac{0.2}{0.6} = \frac{1}{3}$. Thus, improving T_1's service level requires less weight. Hence, by the highest-value-density-first rule, the service level of T_1 is improved before T_2. □

On the other hand, if the relationship between the importance value and weight is non-linear (like T_2 in Figure 6.1), then an approximate solution for this objective could be achieved by using nonlinear programming techniques such as *steepest descent* or *Newton's method* (Bertsekas, 1999). If exact solutions are required, then techniques like *branch-and-bound* can be used offline, and the optimization component could then switch between several predetermined system states. (In Section 7.5, we discuss how we implemented the optimizer for both Whisper and the VEC.)

Example (Figure 6.4). Notice that, in Figure 6.4(b), the relationship between the the estimated weight and the importance value/service level for T_2 is non-linear. Moreover, the value density for T_1 and T_2 are, respectively, $\frac{0.2}{0.2} = 1$ and $\frac{0.2}{0.65} \approx 0.308$. Thus, if the highest-value-density-first approach were used to determine the service levels of these two tasks, then T_1 would be improved before T_2. However, improving T_2's service level from Level 2 to Level 3 requires less weight than improving T_1's service level by one level. Thus, the highest-value-density-first approach would not produce an optimal distribution of weights for these two tasks. □

It is important to note that the use of weight translation functions is the reason why the optimization component is extensible because it allows any optimizing function to assess the impact of changing the functional service level. In prior work on adaptive real-time systems, the two primary methods for optimizing service levels have been to assume either that each service level has a "nominal" utilization (Lu et al., 2002) or the relationship between the service level and importance value is linear (Marti et al., 2004). As we discussed in Section 6.1, the problem with the first approach is that assessing a meaningful "nominal" utilization may be difficult if not impossible for many applications. The problem with the second approach is that there exist applications for which linearity cannot be assumed. For example, consider any video application in which each service level corresponds to a different resolution. Typically, in such a system, as the service level (and by extension the resolution) increases, the amount of benefit to user perception per pixel added decreases. It is easy to see that in such a scenario, the relationship between importance value and estimated weight is nonlinear.

6.2.3 Reweighting

Whenever a task is reweighted (i.e., changes its functional service level) either by the optimization component or by the main AGEDF algorithm, its code segment and/or period may change. If no job of a task, T_i, is active when T_i changes its functional service level from the ℓ_0^{th} to ℓ_1^{th} service level at time t, then the change is simple—the next released job of T_i has the period and code segment associated with the ℓ_1^{th} service level. If a job of T_i is active at t, then the situation is more complicated. For the remainder of this section, let T_i^j denote the active job of T_i at t. Recall from Chapter 3 that, when a task with an active job reweights, there can be a difference between when it "initiates" the change and when the change is "enacted." The time at which the change is *initiated* is defined externally to the reweighting component (by either the optimization component or the main AGEDF algorithm); the time at which the change is *enacted*, i.e., the functional service level is changed, is dictated by reweighting rules.

Changing the period. In order to change the period of a job, we use the GEDF reweighting Rules P and N, presented in Section 3.4.1. For brevity, we do not review these rules here.

Changing the code segment. Whether the code segment of the task T_i that released T_i^j can change depends on the implementation of T_i. For example, if the ℓ_0^{th} and ℓ_1^{th} service levels have substantially different code segments, then T_i^j cannot change its code segment. On the other hand, suppose that the difference between the code segments for the ℓ_0^{th} and ℓ_1^{th} service levels is simply the number of iterations in a loop. Then, as long as T_i^j is not complete and this change would not cause either $\mathsf{Ew}(T_i^j) > 1$ or $\sum_{T_a^b \in \mathsf{ACT}(t)} \mathsf{Ew}(T_a^b) > M$, T_i^j can change its code segment immediately. Moreover, if the code segment is changed, then $\mathsf{Ew}(T_i^j)$ is changed to

$$\frac{\max(\mathsf{Nw} \cdot \mathsf{p}(T_i,\, \ell_1), \mathsf{A}(\mathcal{S},\, T_i^j,\, \mathsf{r}(T_i^j),\, t))}{\mathsf{p}(T_i,\, \ell_0)}, \tag{6.12}$$

where $\mathcal{S}$ is the AGEDF schedule, and $\mathsf{Nw} = \mathsf{g}(T_i,\, \mathsf{Ew}(T_i^j),\, \ell_0,\, \ell_1)$. Notice that the estimated amount of time for which T_i^j will execute as a consequence of changing its code segment is the larger of the amount of time it has already been schedule by time t, i.e., $\mathsf{A}(\mathcal{S},\, T_i^j,\, \mathsf{r}(T_i^j),\, t)$, and the amount of time that T_i^j would have been scheduled if the ℓ_1^{th} service level was the functional service level at $\mathsf{r}(T_i^j)$, $\mathsf{Nw} \cdot \mathsf{p}(T_i,\, \ell_1)$. Thus, the estimated weight of T_i^j is the estimated amount of time that T_i^j will be scheduled divided by $\mathsf{p}(T_i,\, \ell_0)$. (In Section 7.5, we discuss how we implemented the code segment for both Whisper and the VEC.)

Example (Figure 6.5). Consider the example in Figure 6.5, which depicts a three-processor system scheduled by AGEDF with three tasks, all of which have a period of 7, an estimated weight of 3/7, and in the absence of a weight change, would be scheduled for 3 time units. At time 1, all three tasks experience a service level change that changes the code segment for each job. Moreover, we assume that $\mathsf{g}(T_1,\, \mathsf{Ew}(T_1^1),\, \ell_{1,0},\, \ell_{1,1}) = 4/7$, $\mathsf{g}(T_2,\, \mathsf{Ew}(T_2^1),\, \ell_{2,0},\, \ell_{2,1}) = 2/7$, and $\mathsf{g}(T_3,\, \mathsf{Ew}(T_3^1),\, \ell_{3,0},\, \ell_{3,1}) = 1/14$, where $\ell_{i,0}$ is the initial service level of T_i and $\ell_{i,1}$ is the service level that T_i changes to at time 1. Thus, as a result of the change, T_1^1 executes for 4

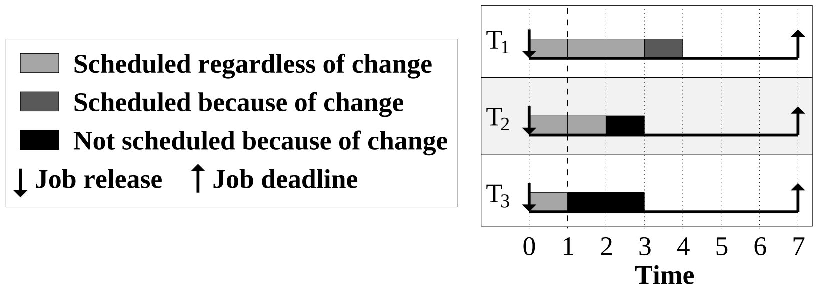


Figure 6.5: An illustration of changing the code segment.

time units, T_2^1 executes for 2 time units, and T_3^1 executes for 1 time unit. Notice that $\mathsf{Ew}(T_3^1)$ is changed to 1/7 even though $\mathsf{g}(T_3, 3/7, \ell_{3,0}, \ell_{3,1}) = 1/14$. The reason for this is that $1 = \mathsf{A}(\mathcal{S}, T_3^1, 0, 1) > \mathsf{Nw} \cdot \mathsf{p}(T_i, \ell_1) = \frac{1}{14} \cdot 7 = 1/2$. Thus, by (6.12), $\mathsf{Ew}(T_3^1) = \frac{\mathsf{max}(1,1/2)}{7} = 1/7$. $\square$

6.2.4 User-Defined Threshold

Choosing a specific user-defined threshold for invoking the optimizer will depend largely on the targeted application. Some possible thresholds could include a duration of time, a substantial change in the estimated weight for one task, or a substantial change in the total estimated weight for all tasks. While running the optimizer more frequently will increase the accuracy of the system, it will also increase the amount of time the scheduler is active with the system not producing "useful" work. Additionally, as we discussed in Section 6.2.3, the reweighting rules cannot always be enacted immediately. Thus, if the optimizer is called before all changes have been enacted, then it may produce an inaccurate result. Notice that, if the weight translation function is accurate, then after all reweighting events have been enacted, the system will remain in an "optimal" state, unless the actual weight of a task changes. Thus, if the separation between optimizer invocations is sufficiently large for all tasks to enact their functional service level changes (i.e., at least the largest period of a job in the system), then it is possible to guarantee that no task will unnecessarily "thrash" between service levels. (In Section 7.5, we discuss how we chose the user-defined threshold for both Whisper and the VEC.)

6.3 Conclusion

In this chapter, we presented the adaptable service-level task model, which is based on a task model presented in (Lu et al., 2002). In addition, we presented the AGEDF scheduling framework, in which tasks are scheduled by GEDF augmented with three components to facilitate adaption: a feedback predictor, an optimizer, and a task reweighter. It is worth noting that these components are modular. As a result, a developer could modify each of these components to improve the performance for a specific application.

CHAPTER 7

IMPLEMENTATION and EXPERIMENTS

In this chapter, we present two sets of experiments. First, we present simulations in which Whisper and VEC are scheduled by our adaptive variants of GEDF, NP-GEDF, PEDF, NP-PEDF, and PD^2. Second, we present experiments conducted using the real-time Linux testbed $\text{LITMUS}^{\text{RT}}$ to evaluate the performance of AGEDF when running the core operations of Whisper and VEC.

Unfortunately at this time, it is not feasible to produce experiments involving a complete implementation of either Whisper or VEC, for two reasons. First, both the existing Whisper and VEC designs are single-threaded (and non-adaptive) and consist of several thousands of lines of code. Converting each implementation to a multi-threaded implementation is a nontrivial task. Indeed, because of this, it is *essential* that we first understand the scheduling and resource-allocation trade-offs involved. The development of our various adaptive algorithms can be seen as an attempt to articulate these tradeoffs. Second, support for *task synchronization* is required, and while there has been work on real-time task synchronization on multiprocessors (Block et al., 2007; Brandenburg et al., 2008; Brandenburg and Anderson, 2008; Chen and Tripathi, 1994; Devi et al., 2006; Gai et al., 2003; Holman and Anderson, 2006; Lopez et al., 2004; Rajkumar, 1991; Sha et al., 1990), applying such work in the context of adaptive scheduling algorithms is non-trivial. For these reasons, we have chosen to conduct our evaluation using both simulations and an implementation of the core operations for Whisper and VEC, i.e., correlation computations for Whisper and bilateral filters for VEC.

We begin this chapter with brief descriptions of Whisper (Section 7.1), VEC (Section 7.2), and $\text{LITMUS}^{\text{RT}}$ (Section 7.3). Then, in Section 7.4, we present our simulations of Whisper

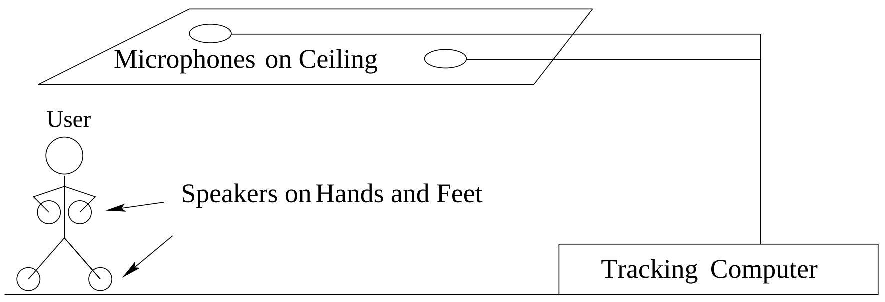


Figure 7.1: The Whisper system.

and VEC when scheduled via the adaptive variants of GEDF, NP-GEDF, PEDF, NP-PEDF, and PD^2. In Section 7.5, we present our evaluation of the AGEDF algorithm when implemented under LITMUS$^{\mathrm{RT}}$ and scheduling the core operations of both Whisper and VEC. (We emphasize that this set of experiments involved running *real* code on a *real* OS kernel and are not merely simulations.) Finally, we conclude in Section 7.6.

7.1 Whisper

As depicted in Figure 7.1, Whisper tracks users via speakers that each emit a unique sound wave and are attached to each user's hands, feet, and head. Microphones located on the wall and ceiling receive these signals and a tracking computer calculates (via a speed-of-sound computation) each speaker's position by measuring signal delays. Whisper is able to compute the signal delay between the transmitted and received versions of the sound by performing a *correlation* calculation on the most recent set of samples. Because correlations are computationally intensive, Whisper uses a *Kalman filter* to decrease the number of correlations required to track a user.

We begin this section by reviewing the concepts of correlation (Section 7.1.1) and the Kalman filter (Section 7.1.2). Next, we discuss the impact of occluding objects (Section 7.1.3). We conclude this section with a discussion of Whisper's real-time characteristics (Section 7.1.4).

```
Cor($x$: array [0...$m-1$] of doubles, $t$: array [0...$n-1$] of doubles):
        array [0...$m-n-1$] of doubles
1:  $i, j$: integer;
2:  $y$: array [0...$m-n-1$] of doubles
3:  for $i := 0$ to $m-n-1$ do
4:      $y[i] := 0$;
5:      for $j := 0$ to $n-1$ do
6:          $y[i] := y[i] + x[i+j] \cdot t[j]$
7:      od
8:  od;
9:  return $y$
```

Figure 7.2: Pseudo-code defining correlation. x is the received signal, t is the target signal, and y is cross-correlation signal.

7.1.1 Correlation

Correlation is a signal-processing technique for locating a known waveform in a signal. In this section, we briefly review the central concepts behind correlation computations. A more detailed discussion can be found in (Smith, 1997).

As an input, correlation takes two discrete signals, t and x, where t is the known waveform, called the *target signal*, that contains n samples, and x is the *received signal* that contains m samples all of which have some level of white noise. As an output, correlation produces a discrete signal, y, called the *cross-correlation signal*, of $m - n$ samples, where $y[i] = \sum_{0 \leq j < n} (t[j] \cdot x[i+j])$. (Pseudo-code for the correlation computation is given in Figure 7.2.) The cross-correlation signal has the property that the value of i for which $y[i]$ is the maximal value in y denotes the index at which the signal t likely appears in x.

Example (Figure 7.3). Consider the example in Figure 7.3, which illustrates the received signal with noise x, known waveform t, and output of the correlation y. Notice that the value of $y[2]$ is substantially larger than any other value in y. Thus, it is easy to see that the waveform t begins at $x[2]$. $\square$

Whisper uses correlation computations to determine the number of samples that have elapsed from the time a signal is emitted by a speaker to the time it is received by a microphone. Whisper is capable of making such a calculation because the microphones and speakers are synchronized. So, in Figure 7.3, if x is the signal received by a microphone from

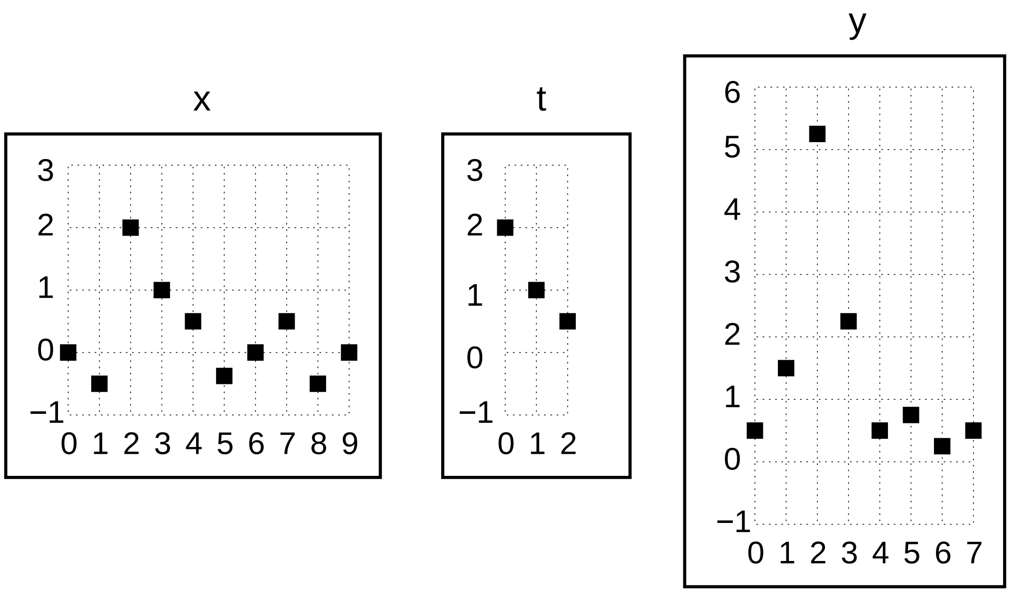


Figure 7.3: An illustration of correlation.

a speaker starting at time 0, and t is the signal sent by that speaker at time 0, then two samples would have elapsed between the time when the signal was sent to the time it was received.

Signal-to-noise ratio. It is important to note that the ability of a correlation computation to determine the location of the target signal in the received signal is directly related to the signal-to-noise ratio. This behavior occurs because, as the signal-to-noise ratio decreases, the relative difference between the maximal value in the cross-correlation value and the other values in the cross-correlation signal decreases. It is possible to compensate for a decreasing signal-to-noise ratio by increasing the number of samples in the target signal; however, this increases the computation time of a task.

Speed of sound. Notice that, once Whisper has used a correlation computation to determine the location of the target signal in the received signal, then (given the number of samples emitted/received per second and the speed of sound) it is a simple matter to compute the distance between a speaker/microphone pair. Specifically, if a speaker/microphone pair emits/receives k samples per second, and from a correlation computation it is determined that ℓ samples have elapsed from the time the signal was sent from the speaker to the time it was received by the microphone, then the total time it took for the sound to reach the microphone from the speaker is ℓ/k. Thus, the distance between the speaker/microphone pair is

$$c \cdot \frac{\ell}{k},$$

where c is the speed of sound.

Example. Consider a scenario where a speaker/microphone pair emits/receives 1,000 samples per second. If a correlation computation determines that 50 samples have elapsed between the time a signal was sent from the speaker to the time it was received by the microphone, then the sound from the speaker took 0.05 seconds to reach the microphone. Thus, given that the speed of sound is approximately $343m/s$, the distance between the microphone speaker pair is $0.05s \cdot 343m/s \approx 17.15m$. $\square$

7.1.2 Kalman Filter

One of the drawbacks to using correlation to locate a known waveform is that it is computationally expensive. Specifically, the cost of using a correlation to locate a target signal t of length m in a received signal x of length n is $O(m \cdot n)$. In order to reduce this cost, Whisper attempts to estimate the location of t in x by using a Kalman filter. The correlation computation merely verifies the estimated location of the target rather than searching through the entire received signal.

The Kalman filter is a recursive mathematical process that combines multiple measurements and the error associated with those measurements to produce an estimated value that is more accurate than any previous measurement. Because understanding the Kalman filter requires knowledge of digital signal processing that is will beyond the scope of this dissertation, we present an intuitive example here and refer the reader to (Welch and Bishop, 1995) for a more detailed discussion.

Example (Figure 7.4). Consider the following example (originally presented in (Maybeck, 1979)), depicted in Figure 7.4, in which two sailors attempt to find their location in one-dimension by using the stars. By using his sextant, the first sailor estimates the ship's

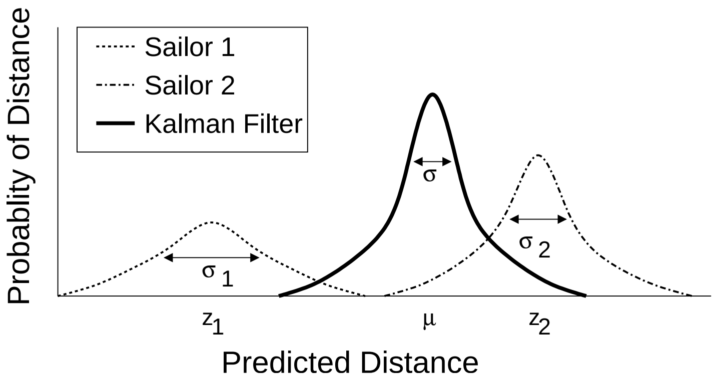


Figure 7.4: An illustration of the Kalman Filter.

position as z_1; however, due to both human and equipment error, the standard deviation for this measurement is σ_1. A second sailor measures their position as z_2, with a standard deviation of σ_2, which is less than σ_1. The Kalman filter combines both measurements to produce a measurement, μ, with a smaller deviance, σ, as given by the formulas

$$\mu = \frac{\sigma_2^2 \cdot z_1}{\sigma_1^2 + \sigma_2^2} + \frac{\sigma_1^2 \cdot z_2}{\sigma_1^2 + \sigma_2^2}$$

$$\frac{1}{\sigma^2} = \frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}.$$

Notice that, while the value μ is between the two values z_1 and z_2, it is closer to z_2 since σ_2 is less than σ_1. Also notice that the standard deviation produced by the Kaman filter, i.e., σ, is less than both σ_1 and σ_2. □

The Kalman filter and Whisper. In Whisper, the Kalman filter is used both before and after the correlation computation. Before Whisper runs the correlation computation, the Kalman filter uses the previous position and velocity of a tracked object to produce an estimate of its position. The second time the Kalman filter is used, the position of an object, produced by the correlation computation, is fed back into the Kalman filter to refine its internal state. This loop is depicted in Figure 7.5. It is important to note that the computation time associated with the Kalman filter is dwarfed in comparison to the computation time required for the correlations.

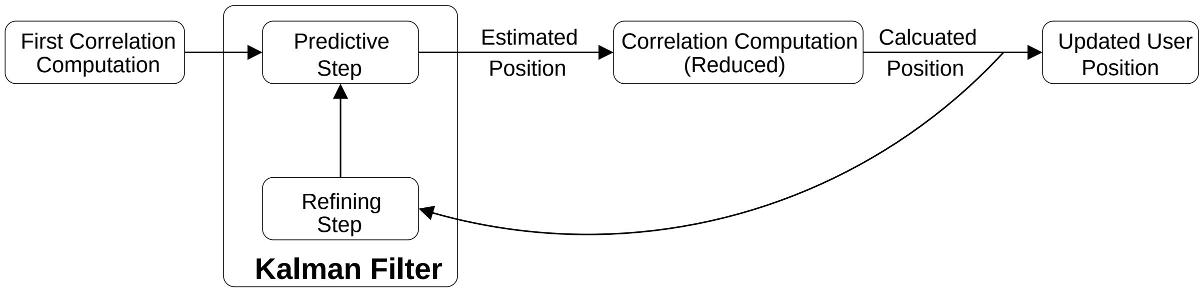


Figure 7.5: The core loop in Whisper.

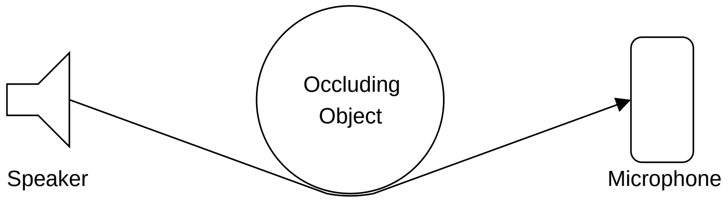


Figure 7.6: An illustration of an occluding object.

7.1.3 Occlusion

One of the advantages of using sound to track a user is that it can bend around objects. As a result, if a user's torso, arms, head, or any other body part obstruct the path between a speaker/microphone pair, then it is still possible to estimate the distance between the pair. This being said, occlusions impact the performance of Whisper in two ways. First, an occluding object acts as a low-pass filter since high frequencies are attenuated as they pass around objects. As a result, the correlation computation will not be as precise. Second, as depicted in Figure 7.6, occluding objects increase the perceived distance between the speaker/microphone pair. The problem with increasing the perceived distance is that it is extremely difficult to correct because the system cannot distinguish between interference by an occluding object or an increase in distance. As a result, this introduces an additional source of error in Whisper's measurements. Whisper handles the additional source of error by estimating the maximal amount of occlusion that an object is likely to cause in order to calculate an estimated upper bound on the error associated with any measurement. Then, this information is used in the Kalman filter, which can partially mitigate the error associated with occlusion.

7.1.4 Real-time Characteristics

As noted above, when the signal-to-noise ratio decreases, correlation become less effective at finding the target signal. To compensate for a decreasing signal-to-noise ratio, the accuracy of Whisper can be improved at the expense of additional computation by either increasing the number of samples in the target signal or increasing the number of position updates per second. In addition, notice that the signal-to-noise ratio is inversely related to the distance between a speaker/microphone pair (since as the distance between a speaker/microphone pair grows, the signal becomes weaker). As a result, when the distance between a speaker/microphone pair is large, the signal-to-noise ratio will be small, which implies that the task associated with this pair will need more computation time to compensate. Thus, as users move around in a virtual environment, the processor shares of tasks assigned to different speaker/microphone pairs must change to compensate for dynamic signal-to-noise ratios.

In addition, since Whisper continuously performs calculations on incoming data, at any point in time, it does not have a significant amount of "useful" data stored in cache. As a result, migration/preemption costs in Whisper are fairly small (at least, on a tightly-coupled system, as assumed here, where the main cost of a preemption or migration is a loss of cache affinity). In addition, fairness and real-time guarantees are important due to the inherent "tight coupling" among tasks required to accurately perform triangulation calculations.

7.2 VEC

In this section, we provide a brief introduction to VEC. Just as with Whisper, a detailed discussion of VEC would involve aspects of multimedia systems that are well beyond the scope of this dissertation. Thus, we refer readers to (Bennett, 2007) for a detailed description of VEC. We begin our discussion of VEC by first reviewing some basics of videography and providing an overview of the system.

Video as a collection of frames. All video is a collection of still images called *frames*. Associated with each pixel in a video frame are *luminance* and *chrominance* values. The luminance value denotes the brightness of the pixel (the higher the value, the brighter the pixel) and the chrominance values denote the color. The luminance of a frame can be increased by lengthening the time the camera's shutter is open, called the *exposure time*. Frames with faster exposure times capture moving objects with more detail, while frames with slower exposure times are brighter. If a frame is *underexposed* (i.e., the exposure time is too fast), then the image can be too dark to discern any object.

VEC corrects underexposed video while maintaining the detail captured by faster exposure times by combining the information of multiple frames. To intuitively understand how VEC achieves this behavior, consider the following example. If a camera, **A**, has an exposure time of $1/30^{th}$ of a second, and a second camera, **B**, has an exposure time of $1/15^{th}$ of a second, then for every two frames shot by camera **A**, the shutter is open for the same time as one frame shot by **B**. VEC is capable of exploiting this observation in order to allow camera **A** to shoot frames with the detail of $1/30^{th}$ of a second exposure time but the brightness of $1/15^{th}$ of a second exposure time.

7.2.1 Bilateral Filter

The primary complication with adjusting the luminance values of pixels to correctly expose a frame is the presence of noise in an image. As a result, a pixel may have different luminance values across multiple frames even if the image being recorded is static, and adjacent pixels may have different luminance values even if all represent the same object.

Example (Figure 7.7). Consider the example in Figure 7.7, which illustrates a 5×5 pixel region of a video frame. The black pixels all represent the same dark object and would all have a luminance value of 20, if not for noise. Thus, in the absence of noise the luminance of $p13$ is 20. Inset (a) depicts the luminance value of each pixel. Insets (b), (c), and (d) are discussed later. Notice that, because of noise, pixels that represent the same object may have subtle variations in luminance intensity. □

One approach for removing the luminance noise from a pixel, henceforth referred to as the *origin pixel*, is to change its luminance to be a weighted[1] average of every pixel within

[1]The usage of the term "weight" here should not be confused with that elsewhere in this dissertation, where this term is used to indicate a processor share.

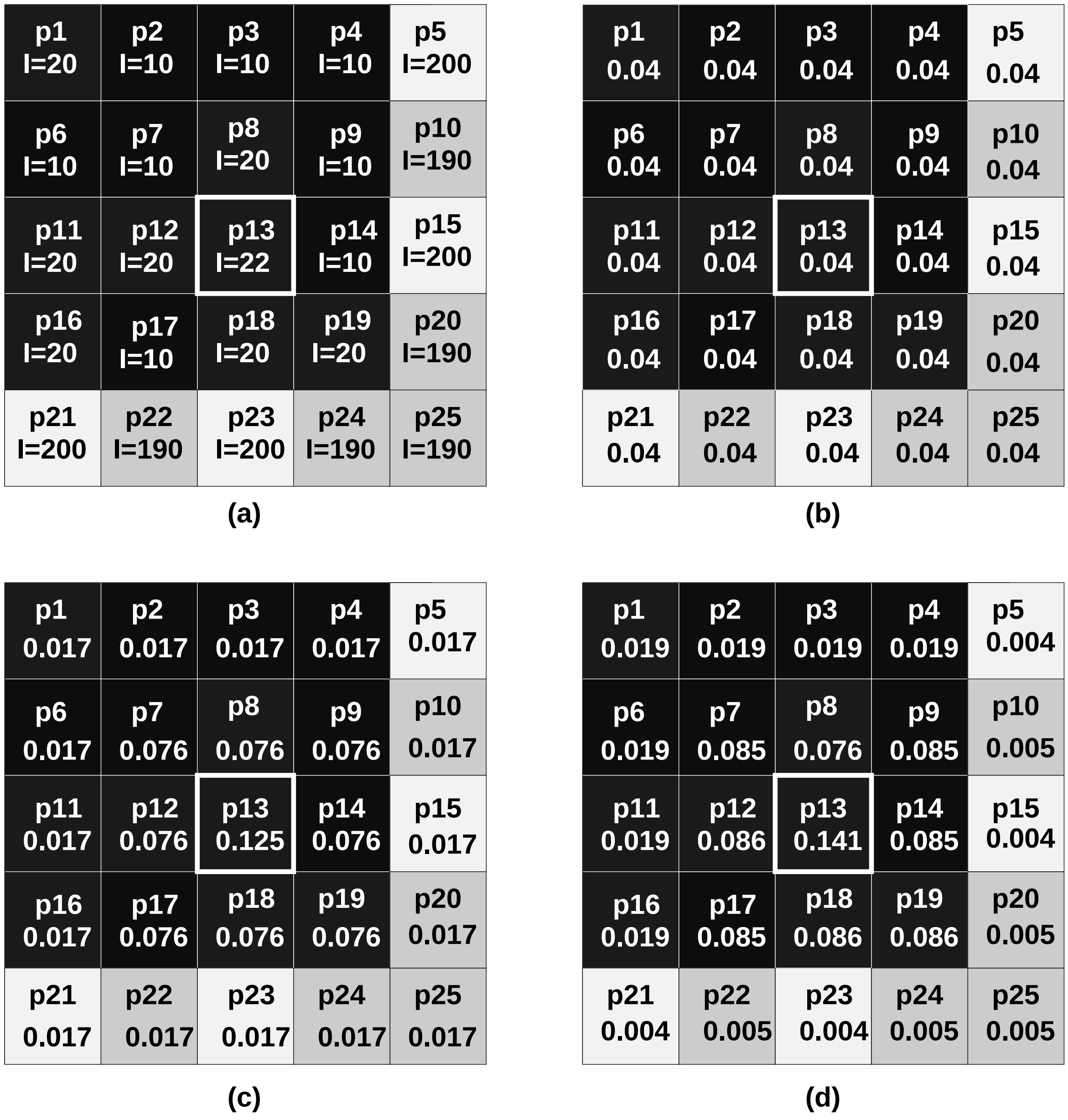


Figure 7.7: Example of different methods for removing noise from a pixel.

a nearby proximity. (This distance can be calculated either *spatially*, i.e., the space between two pixels on the same frame, or *temporally*, i.e., the number of frames between two pixels on different frames.) While such an approach will remove the noise from the origin pixel, it is not immediately obvious what is the best method for determining how much each surrounding pixel should contribute to the origin pixel's final luminance value. The simplest approach is to value all pixels within a given distance equally. The problem with this approach is that it does not account for the fact that the closer a pixel is to the origin pixel the more likely both pixels represents the same object.

Example (Figure 7.7(b)). Such a solution is illustrated in Figure 7.7(b). In this inset (as well as insets (c) and (d)), the values depict the contribution associated with each pixel when computing a weighted average for pixel $p13$, e.g., in this inset, the weighted average for $p13$ is $0.04 \cdot I_{p1} + 0.04 \cdot I_{p2} + 0.04 \cdot I_{p3} + ...0.04 \cdot I_{p25}$, where I_p is the luminance intensity of the pixel p. The weighted average of $p13$'s luminance computed in this way is 79.68, which is nearly four-times its actual noise-free luminance of 20. The reason why this weighted average calculation is so inaccurate is because it values the luminance of all pixels equally, regardless of how close they are to the origin pixel, e.g., pixel $p18$ has the same weight as pixel $p23$, even though pixel $p18$ is closer to pixel $p13$. □

An alternative approach is to value pixels that are closer (again, either temporally or spatially) over those that are far away. One method of determining a pixel's contribution to the weighted average is to use a Gaussian distribution. Specifically, if x is the temporal or spatial distance between a pixel p from the origin pixel s, and σ is used to determine the rate of fall off (i.e., the higher the value of σ the steeper the Gaussian distribution), then p's contribution to the weighted average of s is

$$g(x, \sigma) = \frac{e^{\frac{-x^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}}. \tag{7.1}$$

The problem with this approach is that nearby pixels may not represent the same object.

Example (Figure 7.7(c)). Such a solution is illustrated in Figure 7.7(c). In this case, in this inset, the weighted average of $p13$'s luminance is 43.42, which is nearly two-times its actual noise-free luminance of 20. This measurement is more accurate than that given by Figure 7.7(b) because the closer a pixel is to $p13$ the more it contributes to the weighted average; however, this approach still produces an inaccurate result because nearby pixels may not represent the same object. To observe this behavior, notice that, the luminance of $p11$ should have a greater impact on the final value of $p13$ than $p15$ (since $p11$ and $p13$ represent the same object in the video, while $p15$ represents a different object); however, since $p15$ and $p11$ are the same distance from $p13$, a purely distance-based approach values both pixels equally. $\square$

A third approach, called the *bilateral filter*, involves taking a weighted average that not only gives preference to closer pixels but also pixels that have a similar luminance. By adding similarity in luminance as an additional constraint, such a technique would not be as prone to interference by abutting objects.

Example (Figure 7.7(d)). Such a solution is illustrated in Figure 7.7(d). Notice that, in this inset, the weighted average of $p13$'s luminance is 23.23, which nearly equals its actual noise-free luminance of 20. $\square$

Formally, the bilateral filter of the pixel s, with a spatial/temporal fall off of σ_h and a luminance similarity fall off of σ_i, is given by the formula

$$B(s, \sigma_h, \sigma_i) = \frac{\sum_{p \in N_s} g\left(||p - s||, \sigma_h\right) \cdot g\left(D(p, s), \sigma_i\right) \cdot I_p}{\sum_{p \in N_s} g\left(||p - s||, \sigma_h\right) \cdot g\left(D(p, s), \sigma_i\right)}, \tag{7.2}$$

where I_p denotes the luminance intensity of the pixel p, $||p - s||$ denotes either the spatial or temporal distance between pixels p and s, $D(p, s)$ is the difference in luminance intensity between pixels p and s, and N_s is called the *kernel* and denotes the space of pixels that could contribute to s.

7.2.2 A Few Observations

Before continuing, we make a few observations about the bilateral filter and image processing. First, notice that if an object is not moving, then applying the temporal bilateral filter over the same pixel in multiple frames will likely produce better results than the spatial filter, since each pixel on each frame represents the same object.

Second, because there is a base level of noise in every pixel, the signal-to-noise ratio will be smaller for objects with lower luminance than those with higher luminance (i.e., dark pixels are nosier than light pixels). In addition, humans are more capable of perceiving minor differences between pixels with low luminance than those with high luminance (Pappas and Safranek, 2000). As a result (of both the signal-to-noise and human perception issues), pixels with a low luminance require more noise correction than those with high luminance.

Third, while the bilateral filter (and by extension VEC) can trade accuracy for computational intensity by increasing the number of pixels used to correct a single pixel, the exact number of pixels required to correct a single pixel s is a function of both s's luminance and the similarity of luminance of pixels that surround s. For example, in Figure 7.7(a), Pixel $p13$ will require fewer pixels than $p19$ since all pixels that are adjacent to $p13$ have similar luminance values. On the other hand, the pixel $p19$ has only three adjacent pixels with similar luminance.

7.2.3 VEC's Algorithm

VEC consists of two phases (illustrated in Figure 7.8). First, VEC uses both spatial and temporal bilateral filters to correct the noise of each pixel. Second, VEC applies a technique called *tone mapping* to each noise-reduced frame to change the luminance levels of each pixel into a range that is more palatable for human perception. Since tone mapping requires relatively little processing time in comparison to removing the noise from a frame, and since understanding the process by which VEC applies a tone mapping requires knowledge of image processing techniques, which are beyond the scope of this dissertation, we focus on VEC's noise-removal technique and refer the reader to (Bennett, 2007) for a discussion of tone mapping.

In order to remove the noise from a frame, VEC first calculates the luminance level of each pixel. Next, VEC calculates the *gain factor* for each pixel. The gain factor for a pixel represents the amount of noise correction that pixel requires. The higher the gain factor,

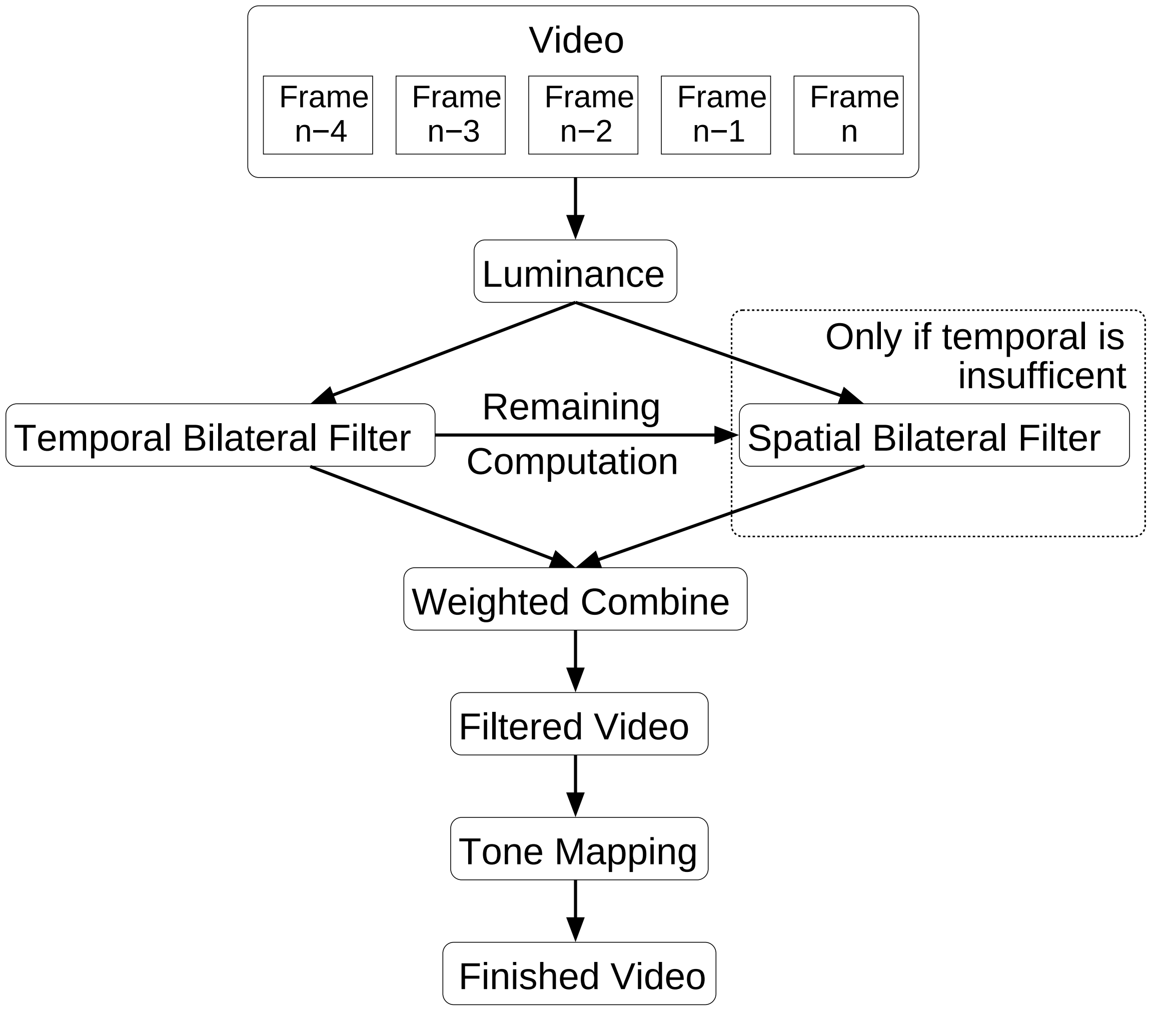


Figure 7.8: The flow diagram of the VEC.

the more noise correction that pixel requires. As we observed earlier, the lower a pixel's luminance, the more error correction it will require. As a result, pixels with a low luminance will have a high gain factor and pixels with a high luminance will have a low gain factor. We denote the gain factor for the pixel s as λ_s. λ_s is linearly proportional to the ratio between the output tone-mapped luminance and the input luminance at s.[2]

As we observed in the previous section, some pixels contribute more than others when correcting noise. To formalize this notion, we say that a pixel p when used to correct a pixel s has a *vote* of $g\left(||p-s||, \sigma_h\right) \cdot g\left(D(p,s), \sigma_i\right)$ (where $g\left(x, \sigma\right)$ is as defined in (7.1)). Moreover, the total amount of pixel votes that are required to correct the pixel s equals $\lambda_s \cdot g\left(0, \sigma_h\right) \cdot g\left(0, \sigma_i\right)$. (Notice that, by (7.1), the larger the value of x, the smaller the value of $g\left(x, \sigma\right)$. Thus, since the vote of a task is defined as $g\left(||p-s||, \sigma_h\right) \cdot g\left(D(p,s), \sigma_i\right)$, the maximal vote of a pixel is $g\left(0, \sigma_h\right) \cdot g\left(0, \sigma_i\right)$.)

After the gain factor has been computed for each pixel, VEC then runs a temporal bilateral filter over a predetermined range of frames. If the total votes of all pixels used by the temporal bilateral filter, γ_s, is less than $\lambda_s \cdot g\left(0, \sigma_h\right) \cdot g\left(0, \sigma_i\right)$, then VEC runs a spatial bilateral filter over a circular area around the pixel s with a radius approximately $3\sigma_i$. If VEC runs both spatial and temporal bilateral filters on the pixel s, then the luminance value of s is a weighted average of the values produced by the spatial and temporal bilateral filters, where the value of each filter is weighted based on the number of votes its pixels contributed. (Notice that, if the total number of votes is still less than $\lambda_s \cdot g\left(0, \sigma_h\right) \cdot g\left(0, \sigma_i\right)$ after the spatial and temporal filters have been run, then the pixel will still have some noise.)

7.2.4 Real-time Characteristics

The most straightforward method for using real-time tasks in VEC is to assign each task a region of each frame to correct, as depicted in Figure 7.9. As a result, since darker objects require more computation than lighter objects to correct, as dark objects move in the video, the processor shares of the tasks assigned to process different areas of the video will

[2]Since understanding λ_s's exact formula in detail requires knowledge of image processing techniques that are beyond the scope of this dissertation, we refer the reader to (Bennett, 2007) for a complete discussion of this topic.

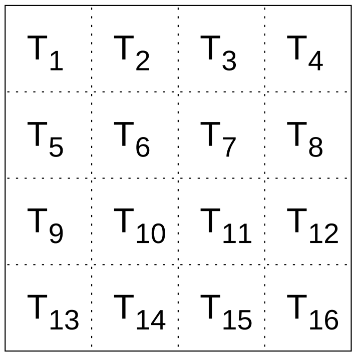


Figure 7.9: The VEC system divided into real-time tasks.

change. Hence, tasks will need to adjust their weights as quickly as an object can move across the screen. Since VEC continuously performs calculations based on previous frames, it performs best when a substantial amount of "useful" data is stored in cache. As a result, migration/preemption costs in VEC are fairly high. In addition, while strong real-time and fairness guarantees would be desirable in VEC, they are not as important here as in Whisper, because tasks can function more independently in VEC.

7.3 LITMUS$^{\text{RT}}$

In this section, we discuss the LITMUS$^{\text{RT}}$ testbed. Since LITMUS$^{\text{RT}}$ is a joint effort by our entire research group with work that has spanned multiple publications (Block et al., 2008c; Brandenburg et al., 2008; Brandenburg et al., 2007; Brandenburg and Anderson, 2008; Calandrino et al., 2006), an in depth description of it would be outside of the scope of this dissertation. Instead, we provide a brief overview of LITMUS$^{\text{RT}}$ and refer the reader to the aforementioned papers for a more detailed discussion.

LITMUS$^{\text{RT}}$ is an extension of Linux that supports a variety of real-time multiprocessor scheduling policies. In its current state, it is most useful as a testbed within which different scheduling policies can be implemented and empirically evaluated. LITMUS$^{\text{RT}}$ is designed in such a way that adding support for additional scheduling policies is straightforward.

LITMUS$^{\text{RT}}$ was implemented by modifying the Linux 2.6.20 kernel[3] configured to run on a symmetric multiprocessor (SMP) architecture. Our particular development platform is an SMP consisting of four 32-bit Intel(R) Xeon(TM) processors running at 2.70 GHz, with 8K instruction and data caches, and a unified 512K L2 cache per processor, and 2 GB of main memory.

Why provide real-time support in Linux? We chose to create our testbed by modifying Linux instead of an existing *real-time operating system* (RTOS) for two reasons. First, Linux is free, open-source software that is easy to obtain and modify, and is widely accepted by both developers and end users. Second, the potential client base for LITMUS$^{\text{RT}}$ as it evolves will include many real-time graphics and multimedia applications developed within our own department. The developers of those applications actually prefer Linux as a development platform.

We acknowledge that producing system designs in any Linux-based system in which real-time correctness is *guaranteed* with certainty is not feasible. Therefore, we expect systems to be provisioned in LITMUS$^{\text{RT}}$ using experimentally-determined worst-case (average-case) values for execution costs and system overheads in the hard (soft) real-time case, instead of using analytically-determined, verified values. Thus, in LITMUS$^{\text{RT}}$, the term "hard real-time" should really be interpreted to mean that deadlines are *almost never* missed, and "soft real-time" to mean that deadline tardiness *almost always* remains within some bound, even if individual tasks misbehave. These are stronger guarantees than provided by most real-time Linux variants in commercial use today.

7.3.1 The Design of LITMUS$^{\text{RT}}$

LITMUS$^{\text{RT}}$ has been implemented via changes to the Linux kernel and the creation of user-space libraries. Since LITMUS$^{\text{RT}}$ is concerned with real-time scheduling, most kernel changes affect the scheduler and timer interrupt code. The kernel modifications can be split into roughly three components. The *core infrastructure* consists of modifications to the Linux

[3]This is true of the LITMUS$^{\text{RT}}$ release that was used in performing the experiments in this chapter. Recently, LITMUS$^{\text{RT}}$ was re-based to Linux 2.6.24 and a number of improvements were made.

scheduler, as well as support structures and services such as tracing and sorted run queues that can be used by scheduler plugins. The *scheduler plugins* encapsulate the available real-time scheduling algorithms by providing functions that implement the methods of the scheduler plugin interface. Finally, a collection of *system calls* provides a user-space API for real-time tasks to interact with the kernel. In the following subsections, we describe each component in turn.

Note that, in the discussion that follows, the term *real-time task* means tasks that are scheduled by LITMUS$^{\mathrm{RT}}$. Normal Linux tasks that run with a static priority from the "POSIX real-time range" are not considered to be real-time tasks in LITMUS$^{\mathrm{RT}}$. Since they do not follow the sporadic task model, they are considered to be just best-effort tasks with a high static priority.

7.3.2 Core Infrastructure

To facilitate the releasing and queuing of real-time tasks, LITMUS$^{\mathrm{RT}}$ provides the abstraction of a *real-time domain*, which consists of a ready queue and a release queue (as well as one lock per queue). When a real-time domain is instantiated, it is parametrized with an *order function* that is used to sort tasks in the ready queue (the release queue is ordered by ascending release time). Wrapper functions are provided in the real-time domain for operations such as queuing, dequeuing, and inspecting designated queue elements. This removes the need for list-handling in most scheduler plugins, thereby reducing development effort (and also removing a common source of bugs).

Scheduling quanta are defined to be the intervals between local timer interrupts. To realize aligned quanta, LITMUS$^{\mathrm{RT}}$ synchronizes timer interrupts during boot across all processors. This is done by having each processor disable its local timer within the local timer interrupt handler, enter a barrier, and restart its timer immediately afterward. When all processors reach the barrier, they will be simultaneously released, resulting in all processors restarting their timers at approximately the same time. Using this method, we have been able to achieve aligned quanta with an error of at most 10 μs on our test platform—in some cases, error is as low as 1-2 μs.

The core LITMUS$^{\text{RT}}$ infrastructure also includes an implementation of the MCS queue lock (Mellor-Crummey and Scott, 1991). Ideally, deterministic locking primitives should be used throughout the kernel. Unfortunately, the Linux kernel uses non-FIFO spin locks, and it is not currently feasible to replace all kernel spin locks with queue locks. Thus, we must be aware of their potential impact on the real-time guarantees that can be made.

At the heart of LITMUS$^{\text{RT}}$, the core infrastructure is also responsible for interfacing with the rest of Linux. It initializes a real-time scheduler plugin (based on a kernel command-line parameter) during system boot. To pass control to the plugin, it hooks into the Linux `scheduler_tick()` and `schedule()` functions. Overriding the Linux scheduler works as follows. Real-time tasks are assigned the highest static Linux scheduling priority upon creation. However, they are not kept in the standard Linux run queues. Instead each plugin is responsible for managing its own run queue. (Similarly, time-slice management is also delegated to plugins for real-time tasks.) When `schedule()` is invoked, control is passed to the current scheduler plugin. If it selects a real-time task to be scheduled on the local processor, then the task is inserted into the run queue and the Linux scheduler is bypassed. When a real-time task is preempted, it is removed again from the run queue, thereby taking it out of the reach of the Linux scheduler.

LITMUS$^{\text{RT}}$ has two modes of operation, real-time and non-real-time. When started, the system is initially in non-real-time mode. Real-time tasks are not scheduled as long as the system is in non-real-time mode. This feature allows complete task systems to be set up before they are scheduled, thereby allowing for the synchronous release of the first jobs of all tasks.

7.3.3 Scheduler Plugins

As mentioned before, real-time scheduling policies are implemented as *scheduler plugins.* Such plugins are realized similarly to other pluggable components in Linux such as file systems. To create a scheduler plugin, functions that realize several methods[4] of the plugin interface must be implemented and registered to the LITMUS$^{\text{RT}}$core. These methods include adding

[4]Sometimes also called "operations" or "callbacks."

a task to the set, "tearing-down" a task, and scheduling a task.[5]

7.3.4 System Call API

LITMUS^{RT} introduces a number of new system calls to Linux. While some of these system calls can be used directly, most of them are intended to be used by *liblitmus*, a user-space library that provides higher-level abstractions. The introduced system calls are organized by purpose into five groups: managing real-time tasks, querying state information, controlling job releases, system setup, and synchronization.[6] Real-time management APIs handle setting up a task and adding it to the task set. State information APIs are used to query a task about their real-time characteristics, e.g., WCET and period. Job control APIs are invoked when a job completes. System setup APIs are used to configure scheduler-specific settings. Finally, synchronization APIs are used to synchronize data across tasks. It is important to note that, currently, synchronization APIs are only implemented for non-adaptive tasks.

7.4 Comparison

In this section, we discuss the first set of experiments we conducted, in which we evaluated the performance of the adaptive algorithms proposed in Chapters 3, 4, and 5 when scheduling Whisper- and VEC-like tasks on a simulated four-processor system. While these experiments are just simulations, most of the parameters used here were obtained by implementing and timing the scheduling algorithms discussed in this dissertation and some of the signal-processing and video-enhancement code in Whisper and VEC, respectively, on a real multiprocessor testbed. Thus, the behaviors in these simulations should fairly accurately reflect what one would see in a real Whisper or VEC implementation.

For both Whisper and VEC, the simulated platform was assumed to be a shared-memory multiprocessor, with four 2.7-GHz processors and a 1-ms quantum, like our test platform. All simulations were run 100 times. Both systems were simulated for 10 secs. (Note that decreasing and increasing the simulation time gives similar results.) We implemented and

[5]A complete list of all 13 methods can be found in (Brandenburg et al., 2007).
[6]A complete discussion of the system call APIs can be found in (Brandenburg et al., 2007).

timed each scheduling scheme considered in our simulations on our test platform and found that all scheduling and reweighting computations could be completed within 5μs. We considered this value to be negligible in comparison to a 1-ms quantum and thus did not consider scheduling overheads in our simulations. For both Whisper and VEC, we conducted two types of experiments: **(i)** all preemption and migration costs were the same and corresponded to a loss of cache affinity; and **(ii)** the preemption cost was set to a fixed value and the migration cost was varied. If a task was preempted and then migrated, we assumed that it incurred the maximum of the two costs. Based on measurements taken on our testbed system, we estimated Whisper's migration/preemption cost as 2μs–10μs, and VEC's as 50μs–60μs. While we believe that these costs may be typical for a wide range of systems, in our experiments we varied the preemption/migration cost over a slightly larger range. (It is worth noting that, in related work by our research group (Calandrino et al., 2006), the average-case preemption (migration) cost for a 4KB working set size, i.e., applications that randomly access 4KB of data, was found to be 15.70μs (15.80μs), and the worst-case preemption (migration) cost for a 4KB working set size as 42.00μs (44.00μs). This research was conducted using the same test platform as considered here.) For all experiments, the maximum execution time was 7ms for PEDF and NP-PEDF and 5ms for GEDF and NP-GEDF. These values were determined by profiling each system beforehand to determine the "best" compromise of accuracy and performance.

While the ultimate metric for determining the efficacy of both systems would be user perception, such a metric would require a full implementation of both systems (which as we discussed at the beginning of this chapter is not currently feasible). Therefore, we compared each of the tested schemes by comparing allocations against each algorithm's respective notion of an IDEAL schedule. In particular, we measured both the "average under-allocation" and "fairness factor" for each task set at the end of each simulation (i.e., 10 secs.). The *average under-allocation* (UA) is the average amount each task is behind its IDEAL allocation (this value is defined to be nonnegative, i.e., for a task that is not behind its IDEAL, this value is zero). The *fairness factor* (FF) of a task set is the largest deviance from the allocations in IDEAL between any two tasks (e.g., if a system has three tasks, one that deviates from its

IDEAL allocation by -10, another by 20, and the third by 50, then the FF is $50-(-10)=60$). The FF is a good indication of how fairly a scheme allocates processing capacity. A lower FF means the system is more fair. For applications like Whisper, where the output generated by multiple tasks is periodically combined, a low FF is important, since if any one task is "behind," then performance of the entire system is impacted; however, for applications like VEC, where tasks are more independent, a high FF does not affect the system performance nearly as much. These metrics should provide us with a reasonable impression of how well the tested schemes will perform when Whisper and VEC are fully re-implemented.

7.4.1 Whisper Experiments

In our Whisper experiments, we simulated three speakers (one per object) revolving around pole in a 1m $\times$ 1m room with a microphone in each corner, as shown in Figure 7.10—the results of these simulations appear in Figure 7.11 and Figure 7.12. The pole creates potential occlusions. One task is required for each speaker/microphone pair, for a total of 12 tasks. In each simulation, the speakers were evenly distributed around the pole at an equal distance from the pole, and rotated around the pole at the same speed. The starting position for each speaker was set randomly. As mentioned in Section 7.1.1, as the distance between a speaker and microphone changes, so does the amount of computation necessary to correctly track the speaker. This distance is (obviously) impacted by a speaker's movement, but is also lengthened when an occlusion is caused by the pole. The range of weights of each task was determined (as a function of a tracked object's position) by implementing and timing the basic computation of the correlation algorithm (an accumulate-and-multiply operation) on our testbed system.

In the Whisper simulations, we made several simplifying assumptions. First, all objects are moving in only two dimensions. Second, there is no ambient noise in the room. Third, no speaker can interfere with any other speaker. Fourth, all objects move at a constant rate. Fifth, the weight of each task changes only once for every 5cm of distance between its associated speaker and microphone. Sixth, all speakers and microphones are omnidirectional. Finally, all tasks have a minimum weight based on measurements from our testbed system

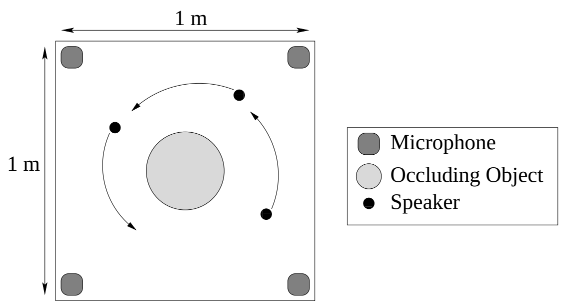
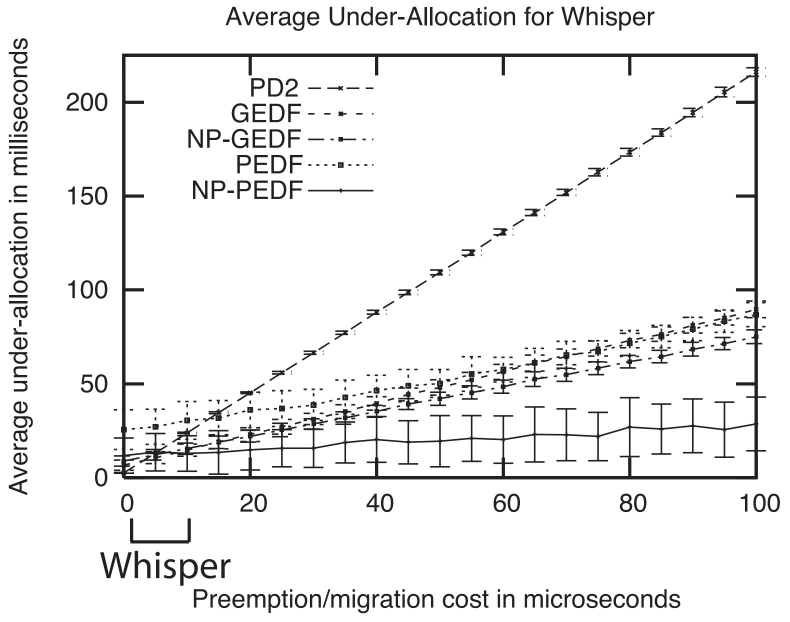


Figure 7.10: The simulated Whisper system.

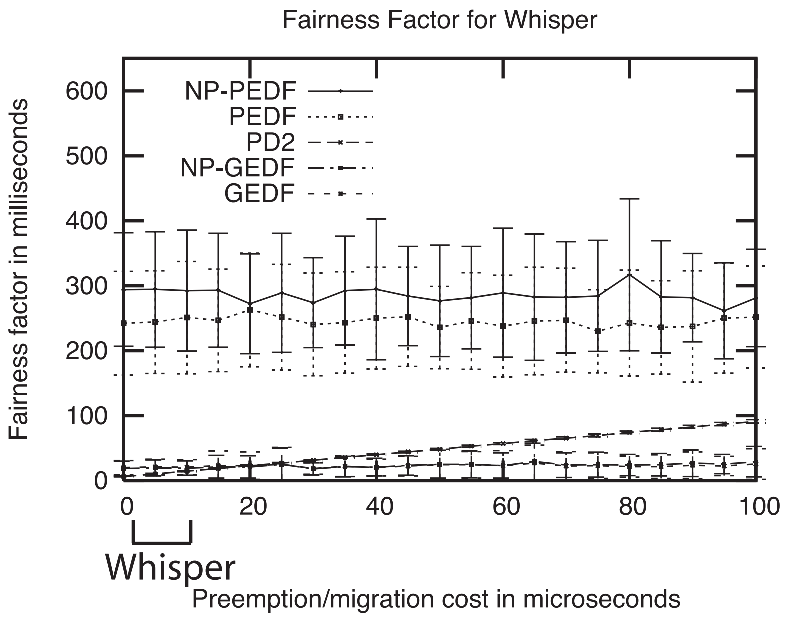
and a maximum weight of 1.0. A task's current weight at any time lies between these two extremes and depends on the corresponding speaker's current position. Even with theses assumptions, frequent share adaptations are required.

We conducted Whisper experiments in which the tracked objects were sampled at a rate of 1,000 Hz, the distance of each object from the room's center was set at 50cm, the speed of each object was set at 5 m/sec. (this is within the speed of human motion), and the maximum execution cost, migration, and preemption cost were varied.

The first set of graphs in Figure 7.11 and Figure 7.12 show the result of the Whisper simulations conducted to compare PD^2, PEDF, NP-PEDF, GEDF, and NP-GEDF. Figure 7.11 depicts the average UA and FF, respectively, for each scheme, where the preemption cost is varied from 0 to 100μs and the migration cost equals the preemption cost. Figure 7.12 depicts the average UA and FF, respectively, for each scheme, where the preemption cost is set at 10μs (the maximum expected preemption cost for Whisper) and the migration cost is varied from 0 to 100μs. There are five things worth noting here. First, when the preemption/migration cost is varied over the range 2 to 10μs (the expected range for Whisper, as noted on each graph), the UA is about the same for all schemes (Figure 7.11(a)); however, PD^2 has the best FF (Figure 7.11(b)). Second, while GEDF and NP-GEDF do not have the best UA for the expected preemption/migration costs for Whisper, for higher preemption/migration costs, i.e., preemption/migration costs larger than 10μs, GEDF and NP-GEDF both have a substantially better UA than PD^2 and better FF than either PEDF or NP-PEDF. Third, as the migration cost (but not preemption cost) of a task increases, the UA of PEDF and NP-PEDF increases slowly (Figure 7.12(a)). However the performance of the other three schemes decays quickly.

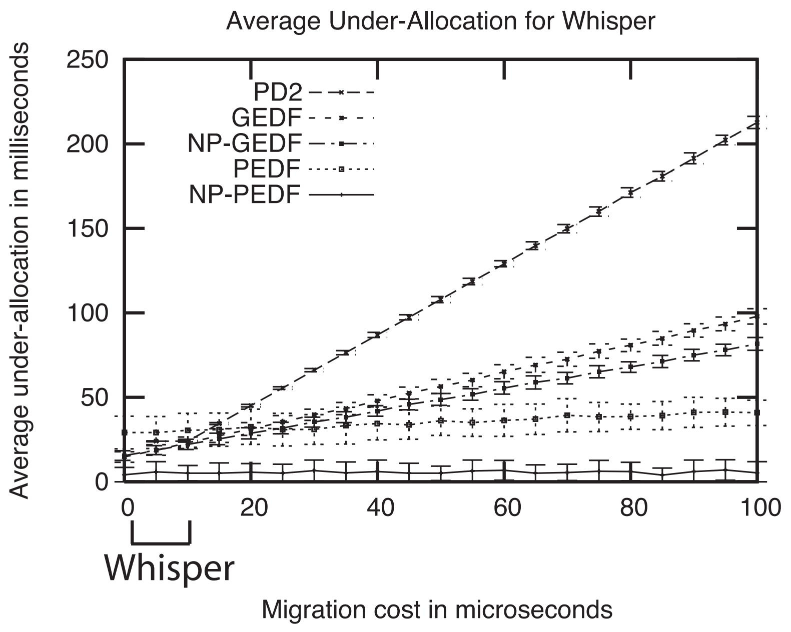


(a)

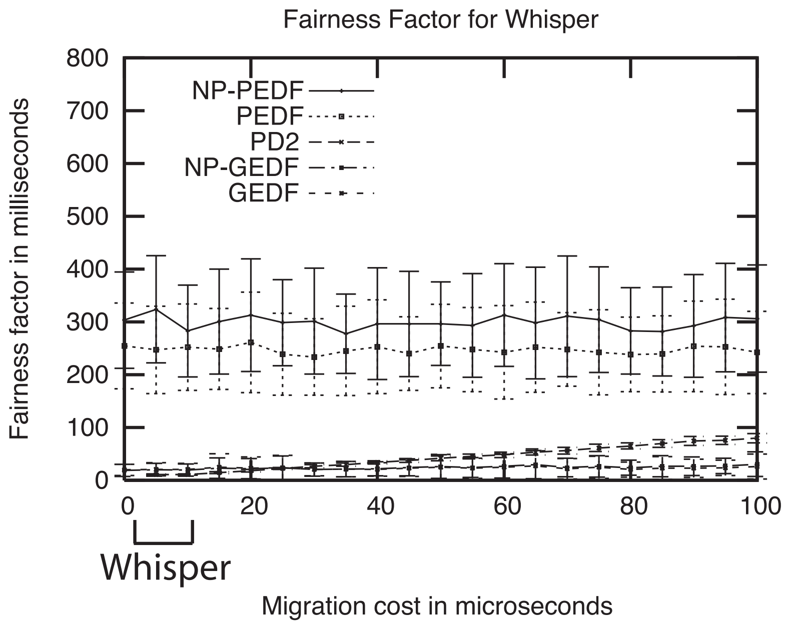


(b)

Figure 7.11: **(a)** The average UA and **(b)** FF for Whisper as a function of preemption/migration cost, as scheduled by each tested algorithm. The key in each graph is in the order that the schemes appear in that graph at 100μs. Standard deviations are shown. Note that, in (b), GEDF and NP-GEDF are indistinguishable from each other.



(a)



(b)

Figure 7.12: **(a)** The average UA and **(b)** FF for Whisper as a function of migration cost (preemption cost is fixed at 10μs), as scheduled by each tested algorithm. The key in each graph is in the order that the schemes appear in that graph at 100μs. Standard deviations are shown. Note that, in (b), GEDF and NP-GEDF are indistinguishable from each other.

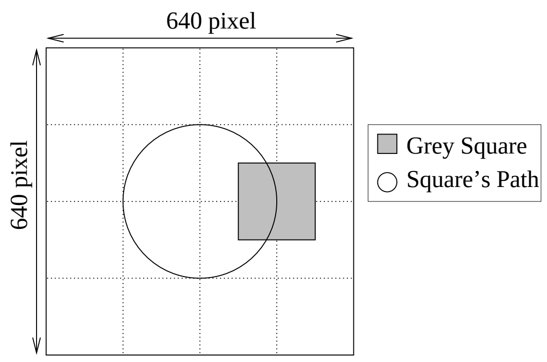


Figure 7.13: The simulated VEC system.

Fourth, standard deviations of the FF for GEDF, NP-GEDF, and PD^2 are smaller than for PEDF and NP-PEDF, since GEDF, NP-GEDF, and PD^2 have better accuracy. Fifth, as seen in Figure 7.12, PD^2's and GEDF's UA and FF do not appreciably increase until the migration cost exceeds 10μs. This is because, until the migration cost is 10μs, PD^2 and GEDF incur the maximum of the migration or preemption cost, which is 10μs.

7.4.2 VEC Experiments

In our VEC experiments, we simulated a 640×640-pixel video feed where a grey square that is 160×160 pixels moves around in a circle with a radius of 160 pixels on a white background. This is illustrated in Figure 7.13. The grey square makes one complete rotation every ten seconds. The position of the grey square on the circle is random. Each frame is divided into sixteen 160×160-pixel regions; each of these regions is corrected by a different task. A task's weight is determined by whether the grey square covers its region. By analyzing VEC's code, we determined that the grey square takes three times more processing time to correct than the white background. Hence, if the grey square completely covers a task's region, then its weight is three times larger than that of a task with an all-white region. The video is shot at a rate of 25 frames per second, and as a result, each frame has an exposure time of 40ms.

Figures 7.14 and 7.15 show the results of the VEC simulations conducted to compare the five tested scheduling algorithms. Figure 7.14 depicts the average UA and FF, for each scheme, where the preemption cost is varied from 0 to 100μs and the migration cost equals the preemption cost. Figure 7.15 depicts the average UA and FF for each scheme, where the preemption cost is set at $60\mu s$ (the maximum expected preemption cost for VEC) and the migration cost is varied from 0 to $100\mu s$. There are two things worth noting here. First, when the preemption/migration cost is varied over the range 50 to $60\mu s$ (the expected range for VEC, as noted on each graph), NP-PEDF and PEDF have the smallest UA (Figure 7.14(a)); however, GEDF and NP-GEDF both have a UA that is competitive with both PEDF and NP-PEDF (Figure 7.14(a)) and have a *substantially* smaller FF (Figure 7.14(b)). Second, as seen in Figure 7.15, PD^2's and GEDF's UA and FF do not appreciably increase until the migration cost equals $60\mu s$. This occurs for the same reason that PD^2 and GEDF did not noticeably increase until $10\mu s$ in Figure 7.12.

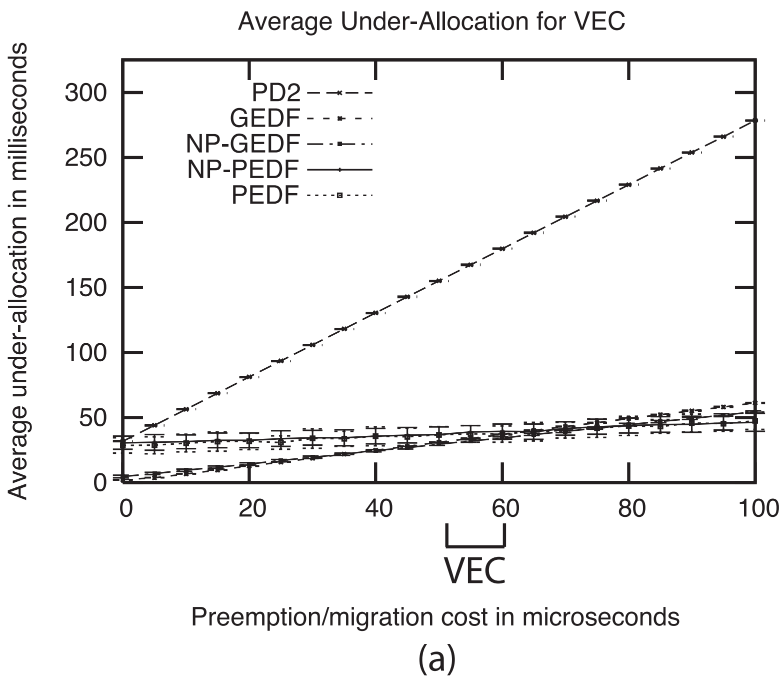
7.5 AGEDF Implementation and Evaluation

In this section, we first describe our implementation of AGEDF in the LITMUS^{RT}framework. Next, we describe the experiments that we used to evaluate our implementation.

7.5.1 Implementation

Because LITMUS^{RT} was designed for sporadic tasks provisioned using WCETs, several modifications were needed to support adaptable sporadic tasks. These included: adjusting the internal structure of the task control block to allow each task to have multiple service levels; disabling the enforcement of WCETs to allow tasks to overrun their expected allocation; and modifying LITMUS^{RT} to allow task statistics such as actual execution times to be gathered.

After making these changes, we implemented the AGEDF framework by changing the GEDF scheduling algorithm (which had already been implemented in LITMUS^{RT}) in two ways. First, we introduced a system call to query the kernel in order for a task to determine its current service level. Second, we implemented the feedback, optimization, and reweighting components in kernel space. Since in Linux floating point operations cannot be used in kernel space, these components were implemented using fixed-point calculations instead.



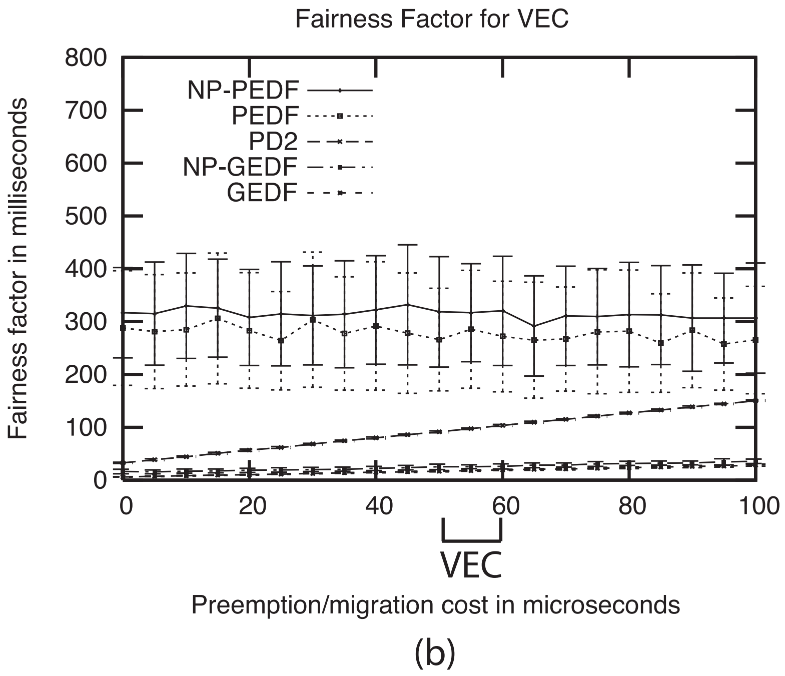
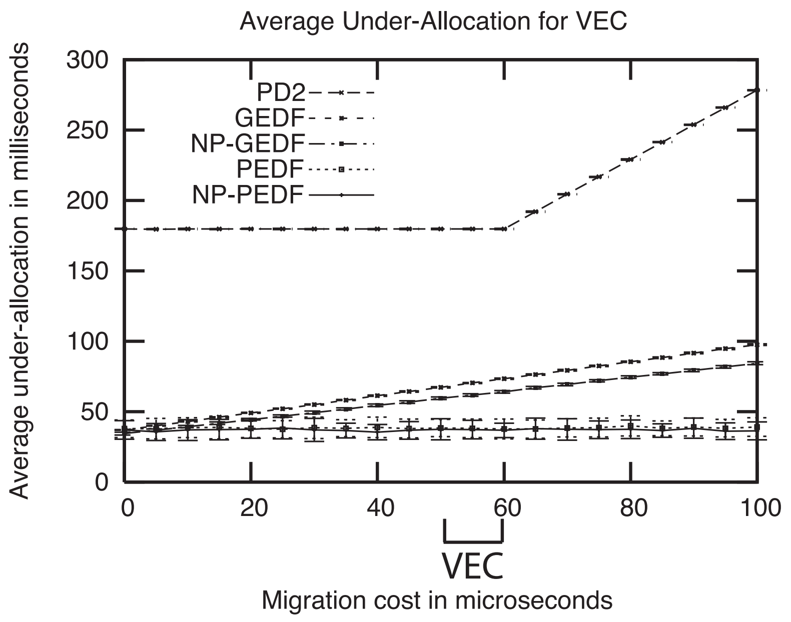
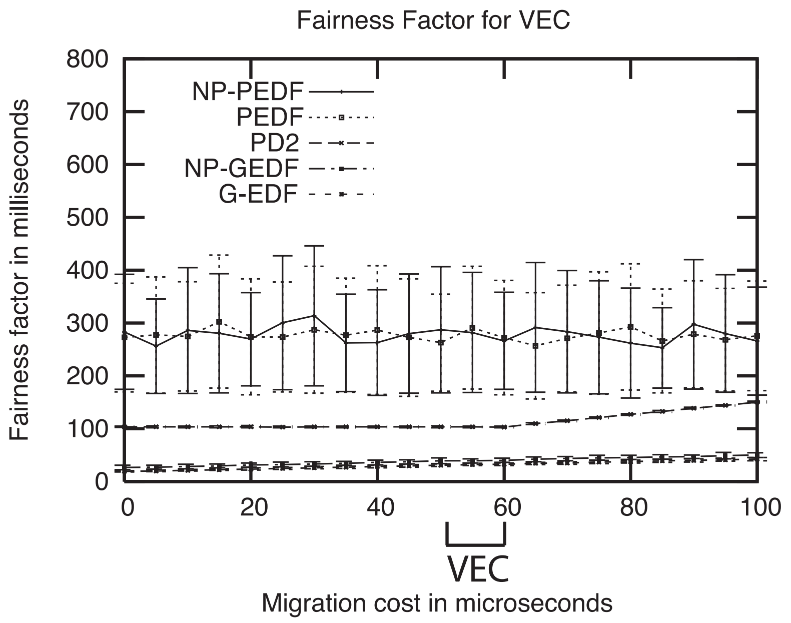


Figure 7.14: **(a)** The average UA and **(b)** FF for VEC as a function of preemption/migration cost, as scheduled by each tested algorithm. The key in each graph is in the order that the schemes appear in that graph at 100μs. Standard deviations are shown. Note that, in (b), GEDF and NP-GEDF are indistinguishable from each other.



(a)



(b)

Figure 7.15: **(a)** The average UA and **(b)** FF for VEC as a function of migration cost (preemption cost is fixed at $60\mu s$), as scheduled by each tested algorithm. The key in each graph is in the order that the schemes appear in that graph at $100\mu s$. Standard deviations are shown. Note that, in (b), GEDF and NP-GEDF are indistinguishable from each other.

7.5.2 Evaluation

The development platform used in our experiments is an SMP consisting of four 32-bit Intel(R) Xeon(TM) processors running at 2.7 GHz, with 8K L1 instruction and data caches, and a unified 512K L2 cache per processor, and 2 GB of main memory (this is the same system that was used for the experiments in Section 7.4). For each task, each job was implemented as a loop in which the core operations of Whisper and VEC are performed iteratively. The exact manner in which jobs behave is discussed below. In implementing the optimizing component, we assumed that there is a linear relationship between importance value and estimated weight, and attempted to maximize the total importance value for all tasks via the highest-value-density-first rule (described in Section 6.2.2). The optimizer was configured to run at least once every second, and also whenever the estimated weight of a task changed by at least 50%, or upon a job completion, the total estimated system weight exceeded four. However, it was constrained to run at most once every 200ms. Note that, in full Whisper and VEC implementations, these choices could possibly be improved upon by carefully considering human-factors issues of relevance to virtual-reality or night-vision systems.

In all experiments, we defined the PI controller using $a = 0.102$ and $c = -1.975$ (see Section 6.2.1 for a discussion of the feedback characteristics associated with these values). We chose these values because we believe that they represent a good tradeoff between transient response and steady-state error (for a ramp input).

Whisper experiments. In our Whisper experiments, we simulated three speakers (one per object) revolving at a speed of 2m/s (this is within the speed of human motion) in a 10m $\times$ 10m room with a microphone in each corner, as shown in Figure 7.16. Tracked objects were sampled at a rate of 2,000 kHz, and the distance of each object from the room's center was set at 5m. While this test scenario may seem simple (since the path of each object is simple and pre-determined), it is actually a challenging test case for Whisper. This is because objects moving at a relatively high speed of 2m/s require significant computational resources to track. Moreover, while it is possible to simulate objects that start, stop, and change directions, such scenarios actually require *less* computational resources and adapt

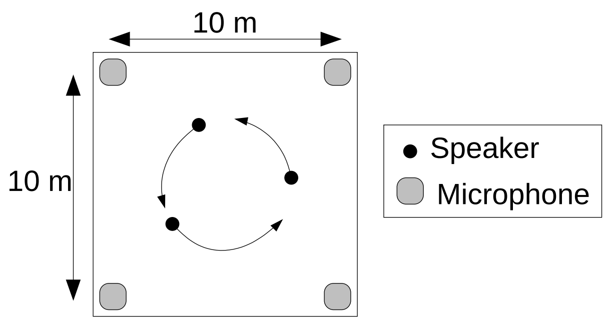


Figure 7.16: The simulated Whisper system.

task service levels *less* frequently, because user motion is typically slower when motion is not continuous.

In the above scenario, one task is required per speaker/microphone pair, for a total of 12 tasks. Each task was configured to have three service levels, with periods/importance values of 66ms/0.25, 33ms/0.5, 22ms/0.75, respectively, and $\mathtt{g}(T_i,\, e,\, 1,\, 2) \approx 5e$ and $\mathtt{g}(T_i,\, e,\, 1,\, 3) \approx 9e$ ($\mathtt{g}(T_i,\, e,\, \ell_1,\, \ell_2)$ is defined in Section 6.1). The importance values were selected somewhat arbitrarily after some trial-and-error experimentation; in an actual deployment, user studies would be required to assess the impact of different settings. Since the weight/importance value relationship is linear, we used the approach in Section 6.2.2 to optimize the system. The other parameters were selected based upon the existing Whisper implementation. As we discussed in Section 7.1, in Whisper, the QoS provided is directly related to the number of correlation computations (CCs) performed per second. When the signal-to-noise ratio decreases, the number of CCs must be increased to maintain the same QoS. Similarly, the QoS provided can be increased by increasing the number of CCs per second. A change in the functional service level of a Whisper task changes the number of CCs per second. We estimated that the existing Whisper implementation, if implemented on our test platform, would perform approximately 27,600,000 CCs per second in the average case. The task periods and $\mathtt{g}(T_i,\, e,\, \ell_1,\, \ell_2)$ values given above were defined so that the average number of CCs per second for the second service level matches this rate. Note that, because the code segments of the three service levels differ only in the number of CCs performed, the code segment of an active job can be changed.

The first experiment we discuss was conducted to see if adaptivity is even needed in implementing Whisper. In this experiment, we ran each of the twelve speaker/microphone pair tasks individually as a normal Linux task for 20 seconds at all three service levels and measured their actual weight. The results for one of the twelve tasks is shown in Figure 7.17 (the other eleven tasks have a similar behavior). After 5.5 and 12.3 seconds, the system experiences 480ms of noise that doubles the number of correlation computations required per job. The average weight of the task at the first, second, and third service level is, respectively, 0.05, 0.25, and 0.45. Notice that, for a system with 12 tasks, operating each task at its highest service level and allocating it a processor share based on its worst-case weight (which is 1.0) gives a total actual weight of 12, which substantially over-utilizes the system. Even with an average-case provisioning, the system is still over-utilized, as the total actual weight is 5.4 in this case. On the other hand, configuring each task to run at its second service level using its average-case weight gives a total actual weight of 3.0, which does not over-utilize the system; however, using a constant average-case allocation would likely cause the system to be over-utilized when noise is encountered. Thus, from this experiment, we can infer that, in order for Whisper to schedule tasks at any service level higher than the lowest one, adaptive scheduling is needed (as is a multiprocessor).

In the second experiment, we ran all 12 Whisper tasks on LITMUS^{RT}, scheduled by AGEDF, for 20 seconds with 480ms bursts of ambient noise after 5.5 and 12.3 seconds that double the number of CCs required to maintain the same QoS. Figure 7.18 shows the total actual weight and the total importance value of the system as a function of time. Figures 7.19–7.24 depict the actual and estimated weights and error (defined as the difference between the actual and estimated weight) for all twelve tasks as a function of time. They also show the functional service level for each task as a function of time. There are several interesting things to notice about these graphs. First, for the tasks depicted in Figure 7.19–7.24, error is typically within the range $[-0.05, 0.05]$. Second, whenever the functional service level changes or the system encounters noise, error briefly spikes but quickly falls back within the range $[-0.05, 0.05]$. Third, when the task in Figure 7.19(a) encounters noise at time 5.5, its service level is changed and there is a substantial drop in its weight; however, when it encounters noise at time 12.3, its service level is not decreased because its actual weight is so low. Note that its low weight at this time is coincidental: its weight varies between times 8 and 20

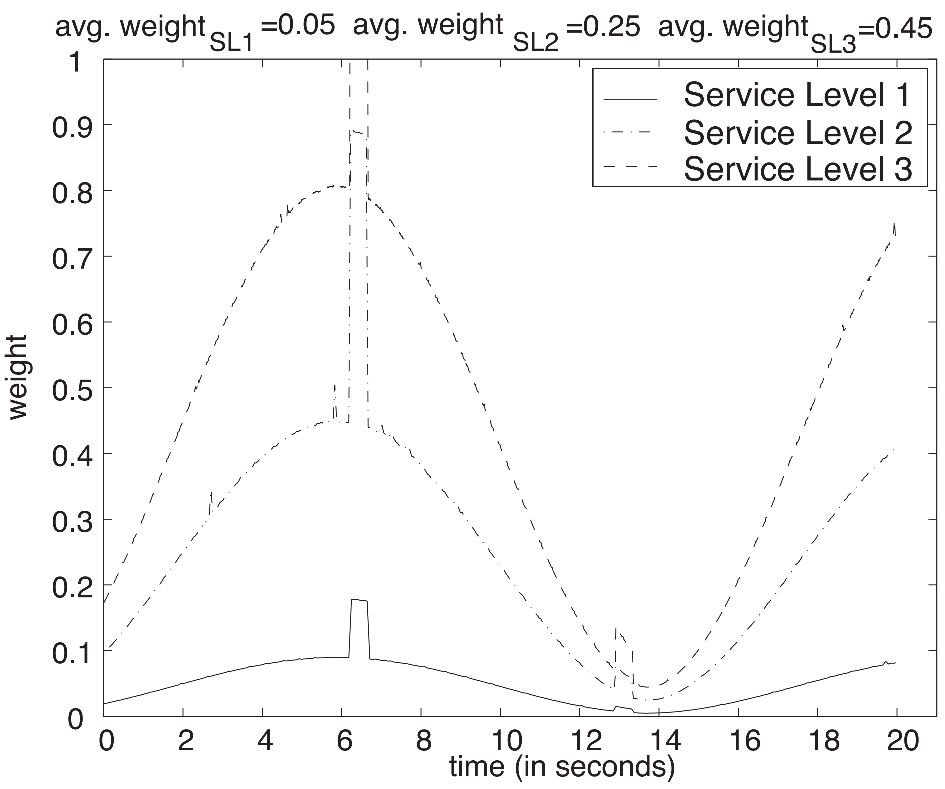


Figure 7.17: The actual weight of a Whisper task at three different service levels over a 20-second run with two bursts of noise at approximately times 6 and 13.

as depicted because of the movement of the corresponding object, and that object happens to be closest to the microphone for which this task is defined at approximately time 14. Fourth, when a job of the task in Figure 7.19(b) completes after the noise at time 12.3, the total estimated weight is greater than four, so the optimizer is invoked causing this task to decrease its service level. This is why the actual weight of this task is briefly greater than one. Fifth, the total utilization of the system is typically close to four, and the system is briefly over-utilized when noise is encountered. Because the total actual weight is always close to four, *this system would not be schedulable using a partitioning approach*. Sixth, the total importance value of the system is typically in the range $[7.5, 8]$ and drops below 6.0 only when noise is encountered. In contrast, if tasks were statically assigned their second service level and scheduled by GEDF, then the total importance value would never *exceed* 6.0.

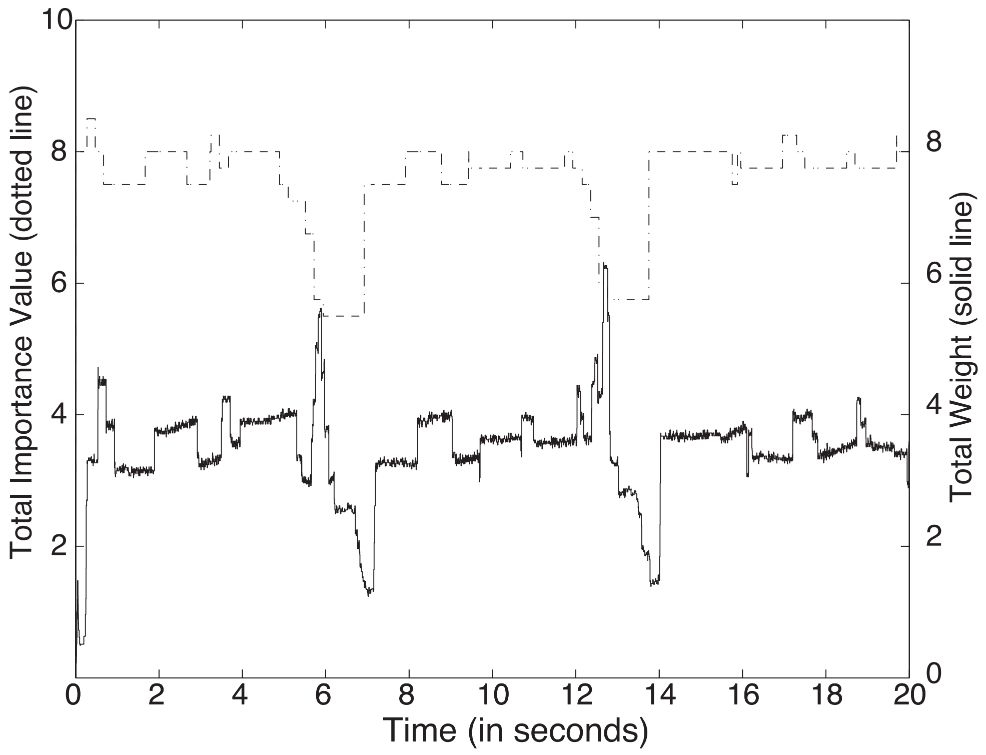
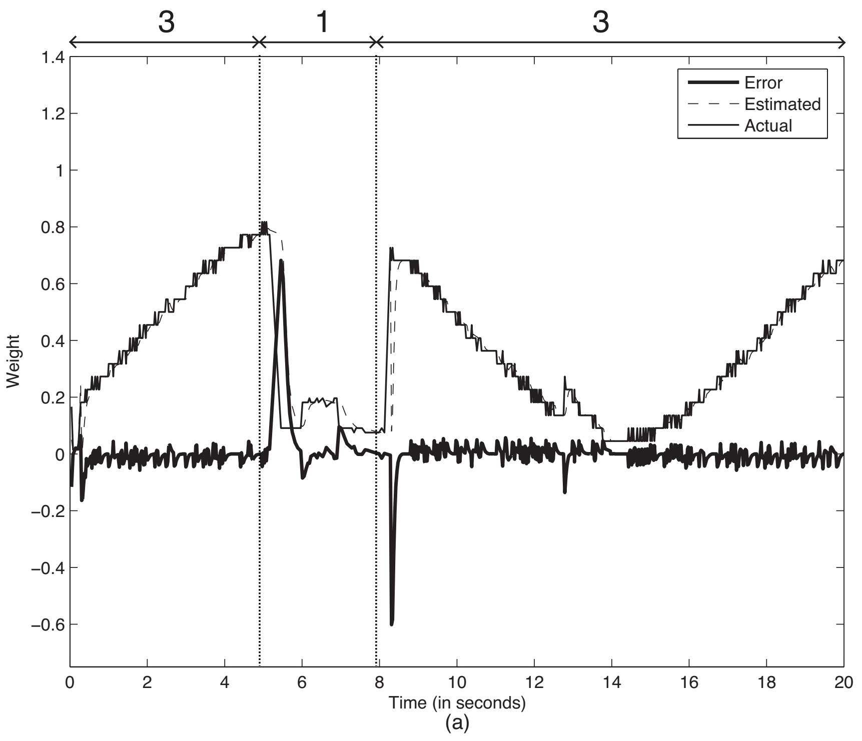


Figure 7.18: Total actual weight and importance value as a function of time, when executing 12 Whisper tasks for 20 seconds.

VEC experiments. In the VEC experiments, we considered a 320×320-pixel video feed where a grey square that is 80×80 pixels moves around in a circle with a radius of 80 pixels on a white background, as illustrated in Figure 7.25. Each frame is divided into sixteen 80×80-pixel regions (as depicted in Figure 7.9); each of these regions is corrected by a different task. The amount of execution time a task requires is determined by whether the grey square covers its region. We assumed that the grey square is sufficiently dark that it takes three times more processing time to correct than the white background. Hence, if the grey square completely covers a task's region, then its weight is three times larger than that of a task with an all-white region. Moreover, occasionally the video briefly becomes dark, doubling the execution time for each job. For each experiment, we considered two different sets of three service levels—one in which periods change and one in which code segments change (by varying the number of iterations described earlier in Section 6.2.3). The periods/importance levels were set at 66ms/0.25, 33ms/0.5, and 22ms/0.75, respectively. Furthermore, we do not allow the code segment of an *active* job to change. Also, $\mathtt{g}(T_i, q, 1, 2) = 2q$ and $\mathtt{g}(T_i, q, 1, 3) = 3q$



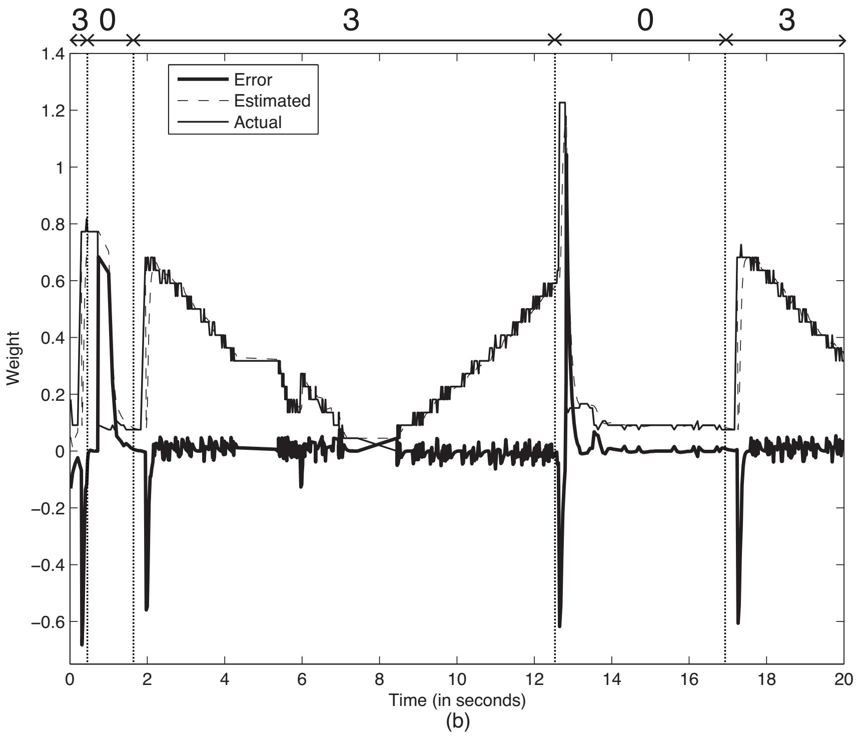
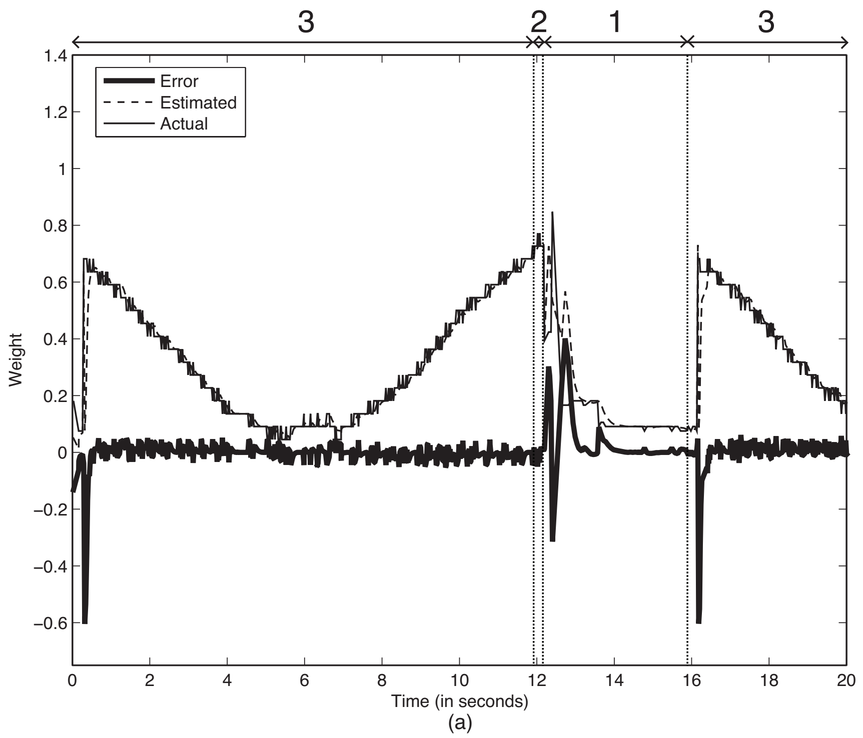


Figure 7.19: Results from executing 12 Whisper tasks for 20 seconds. Actual and estimated weights and error for **(a)** T_1 and **(b)** T_2 as a function of time. The service level for each task is depicted across the top of each inset. In both insets, the error line is centered around zero, and the actual and estimated lines are often indistinguishable.



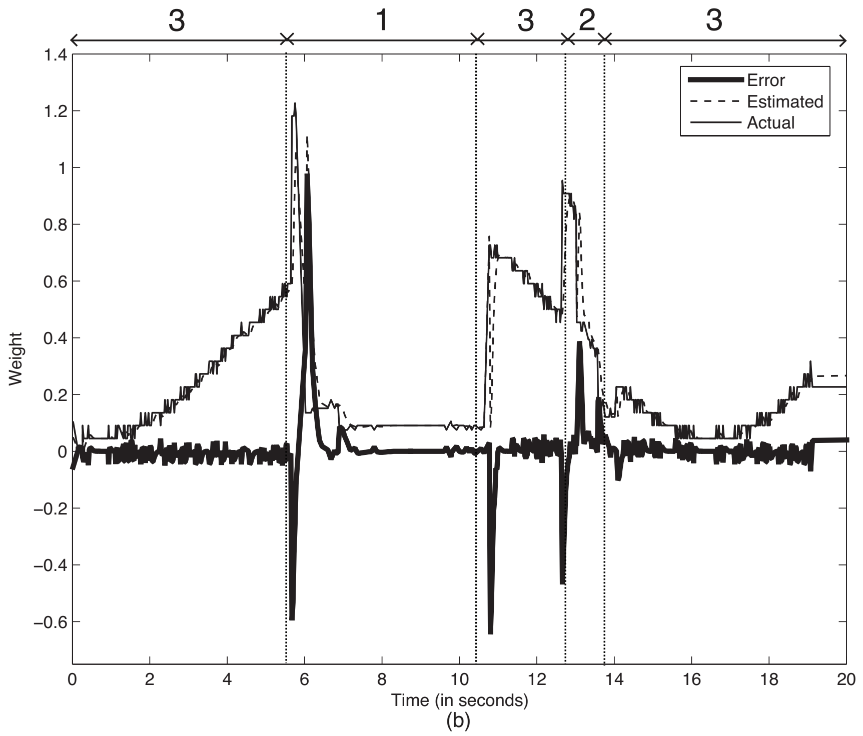
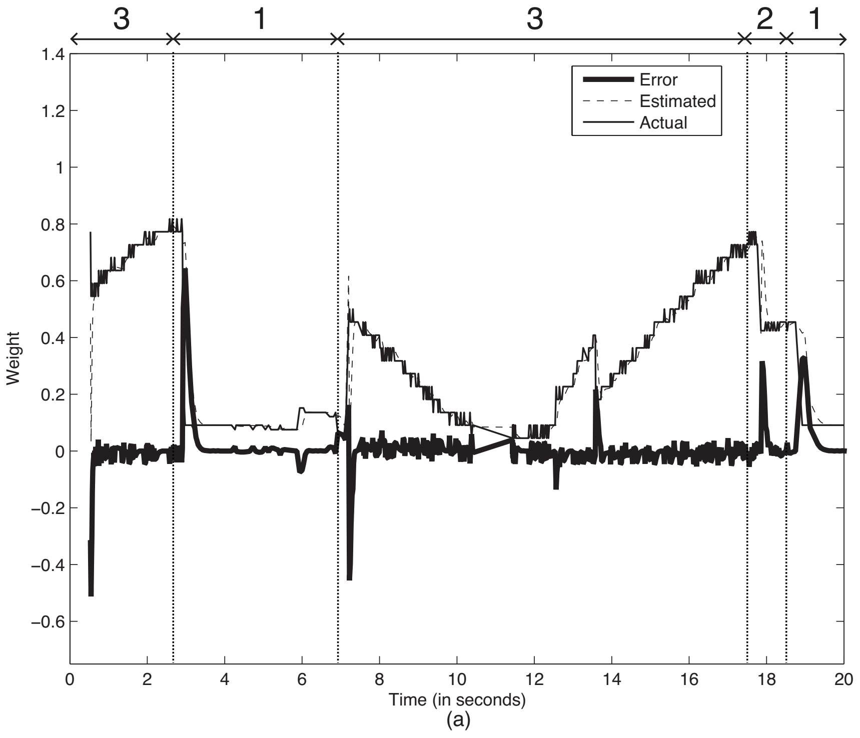


Figure 7.20: Results from executing 12 Whisper tasks for 20 seconds. Actual and estimated weights and error for **(a)** T_3 and **(b)** T_4 as a function of time. The service level for each task is depicted across the top of each inset. In both insets, the error line is centered around zero, and the actual and estimated lines are often indistinguishable.



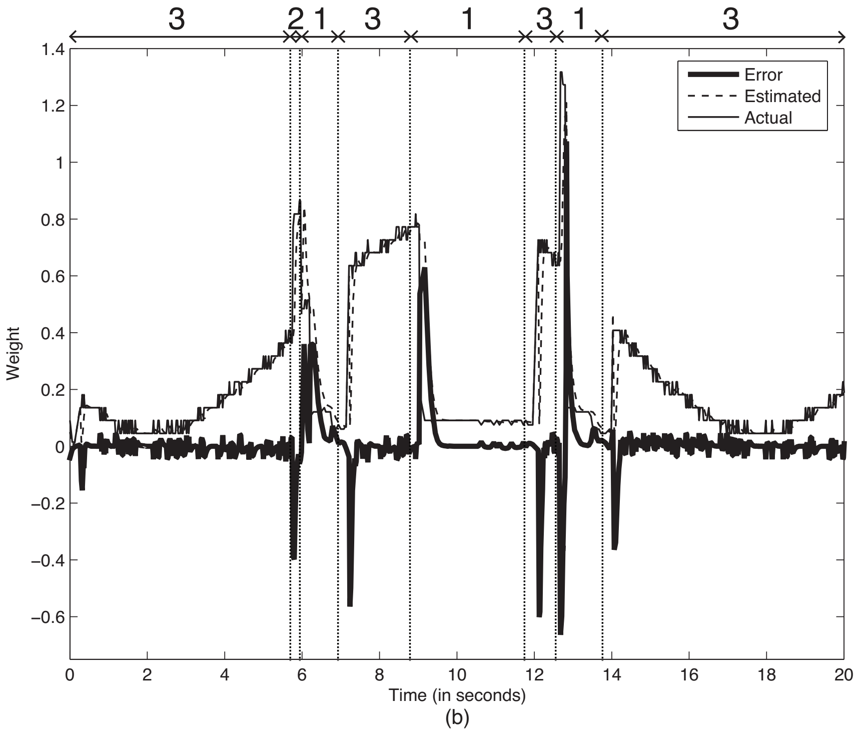
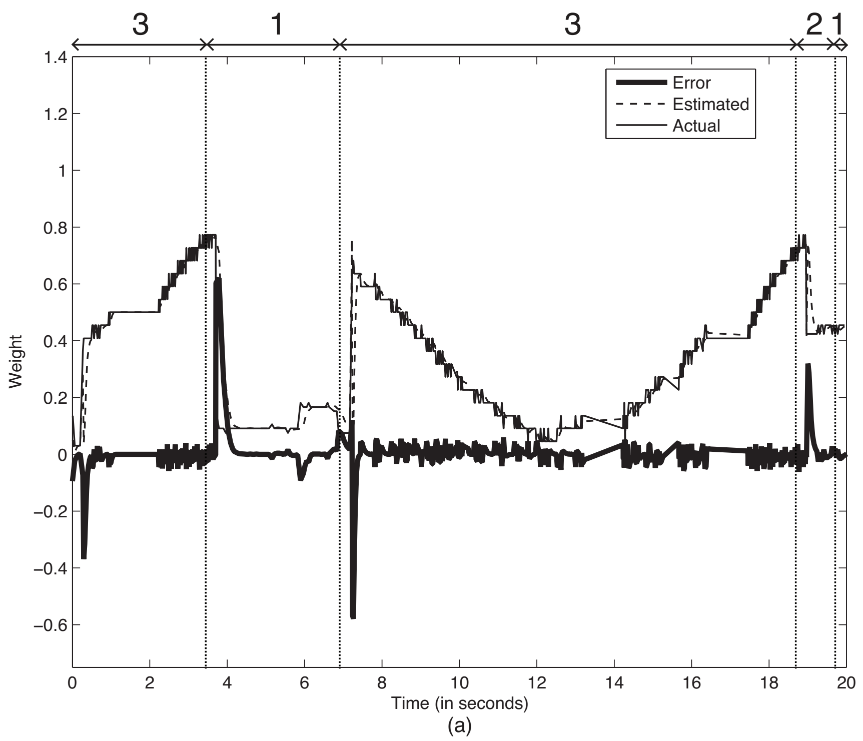


Figure 7.21: Results from executing 12 Whisper tasks for 20 seconds. Actual and estimated weights and error for **(a)** T_5 and **(b)** T_6 as a function of time. The service level for each task is depicted across the top of each inset. In both insets, the error line is centered around zero, and the actual and estimated lines are often indistinguishable.



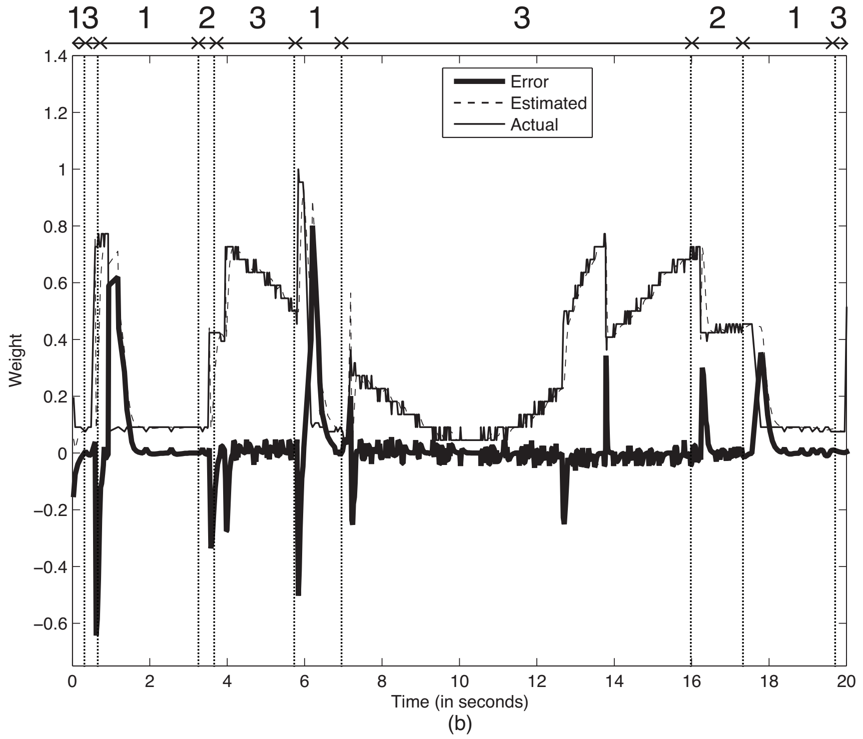
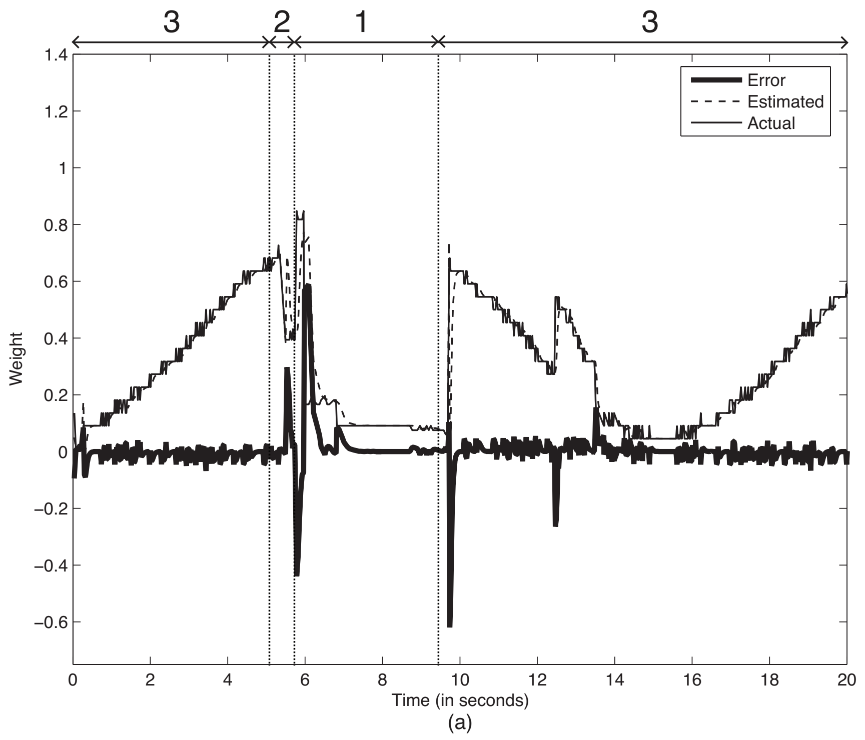


Figure 7.22: Results from executing 12 Whisper tasks for 20 seconds. Actual and estimated weights and error for **(a)** T_7 and **(b)** T_8 as a function of time. The service level for each task is depicted across the top of each inset. In both insets, the error line is centered around zero, and the actual and estimated lines are often indistinguishable.



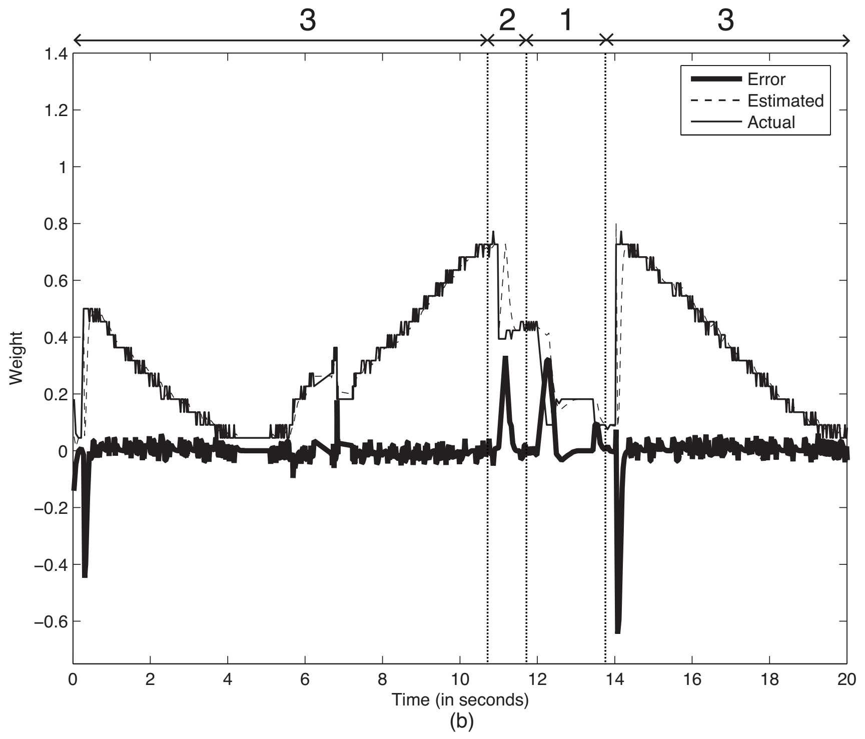
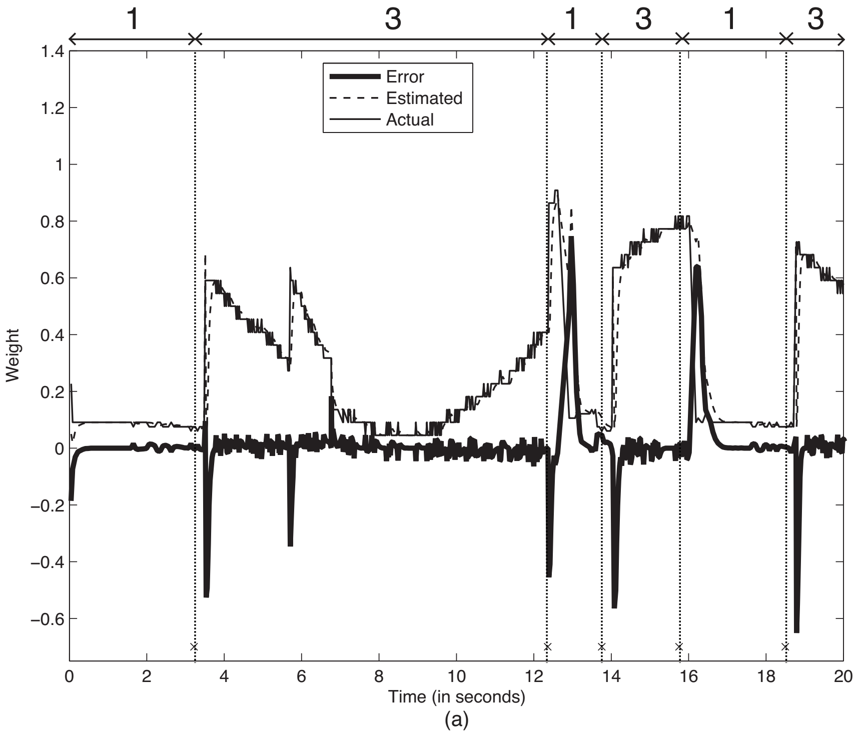


Figure 7.23: Results from executing 12 Whisper tasks for 20 seconds. Actual and estimated weights and error for **(a)** T_9 and **(b)** T_{10} as a function of time. The service level for each task is depicted across the top of each inset. In both insets, the error line is centered around zero, and the actual and estimated lines are often indistinguishable.



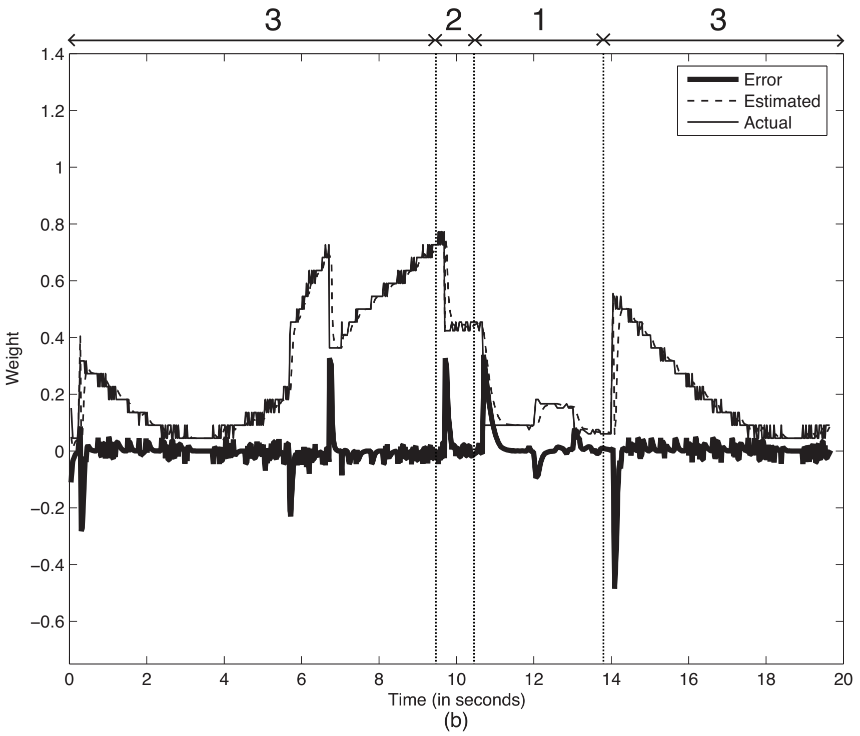


Figure 7.24: Results from executing 12 Whisper tasks for 20 seconds. Actual and estimated weights and error for **(a)** T_{11} and **(b)** T_{12} as a function of time. The service level for each task is depicted across the top of each inset. In both insets, the error line is centered around zero, and the actual and estimated lines are often indistinguishable.

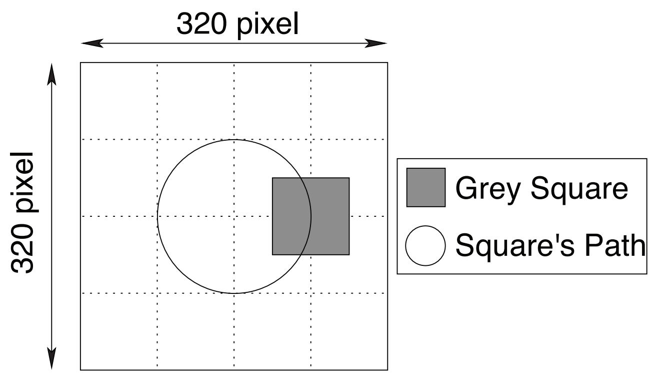
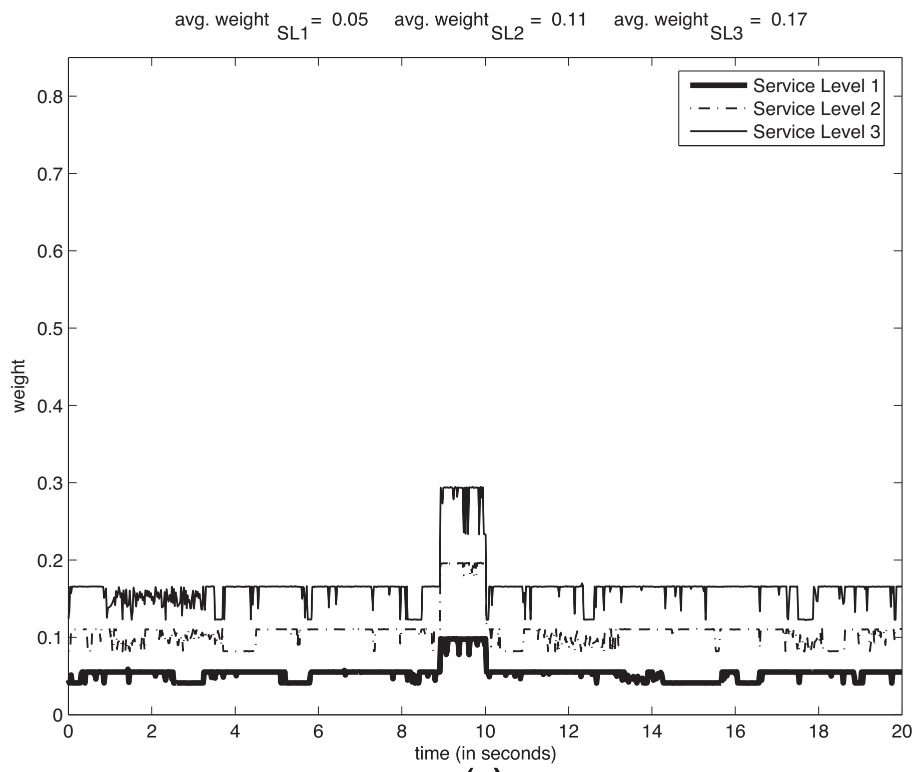


Figure 7.25: The simulated VEC system.

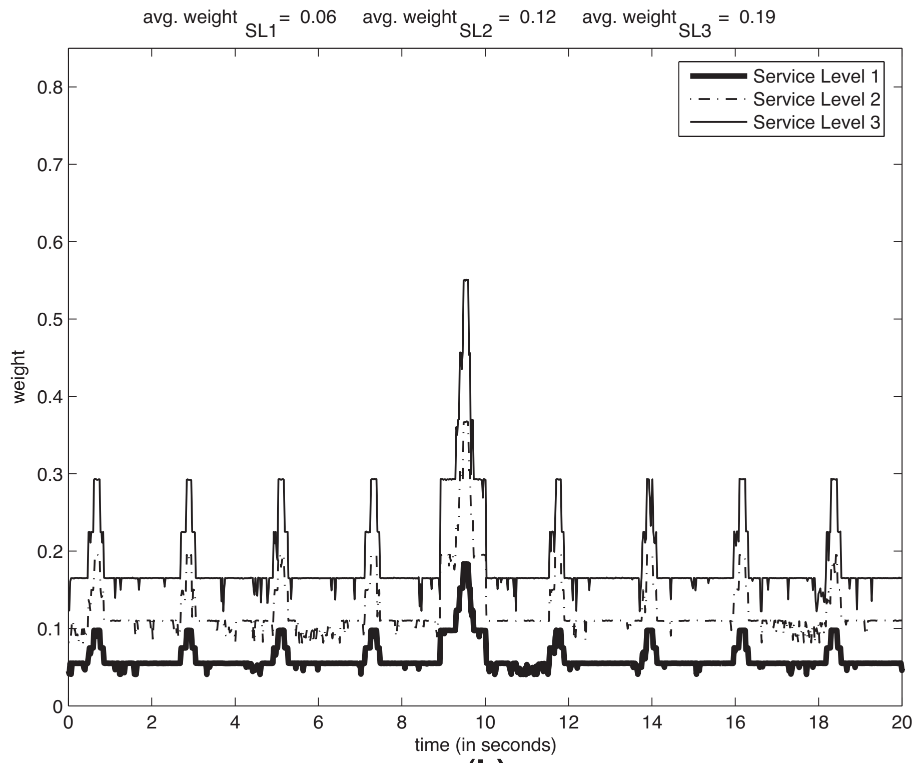
(g(T_i, q, ℓ_1, ℓ_2) is defined in Section 6.1).

Video is shot at a rate of 30 frames per second, and as a result, each frame has an exposure time of approximately 33ms. In the experiment that we report on here, we ran VEC for 20 seconds, the dark square makes one revolution approximately every 2.2 seconds, and after 8.9 seconds, the system encountered 1,100ms of darkness that doubled the number of computations required by each task.

In the first set of VEC experiments, we determined if adaptivity is needed in implementing VEC. In this experiment, we ran each of the sixteen VEC tasks individually as normal Linux tasks for 20 seconds at all three service levels and measured their actual weight. The results for T_1, T_2, and T_6 are given in Figures 7.26(a), 7.26(b), and 7.27, respectively. (As seen in Figure 7.9, the corner tasks, T_4, T_{13}, and T_{16}, have the same behavior as T_1; the side tasks, T_3, T_5, T_8, T_9, T_{12}, T_{14}, and T_{15}, have the same behavior as T_2; and the center tasks, T_7, T_{10}, and T_{11}, have the same behavior as T_6.) Notice that, if tasks are statically assigned their maximum weight and either their second or third service level, then the total weight of all tasks would exceed four. On the other hand, if tasks are statically assigned their average weight, then the total weight of all tasks at their highest service level would not exceed four (the corner, side, and center tasks in this case are assigned weights that are approximately, 0.17, 0.19, and 0.22, respectively). However, in this case, each task would be incapable of correcting its associated region when either the grey square was present or the system incurred darkness. Thus, from this experiment, we can infer that, in order for VEC to fully utilize the system, adaptive scheduling is needed (as is a multiprocessor).



(a)



(b)

Figure 7.26: The actual weight of a VEC task at three different service levels. **(a)** T_1. **(b)** T_2.

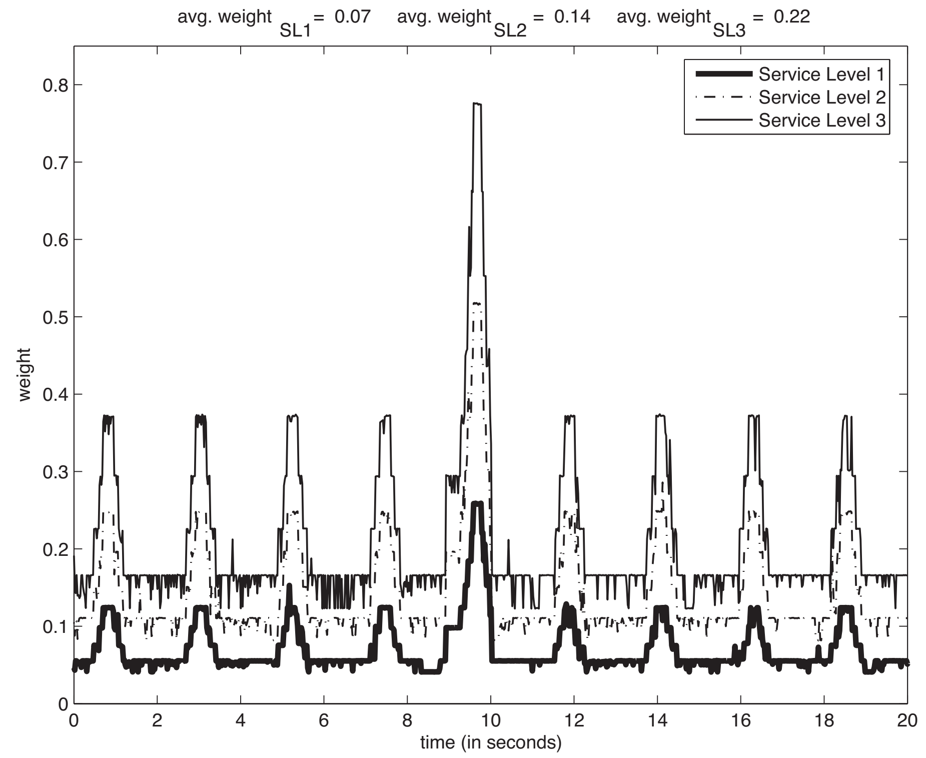


Figure 7.27: The actual weight of a VEC task, T_6, at three different service levels.

In the second set of experiments, we ran all sixteen tasks as real-time tasks in LITMUS$^{\text{RT}}$ at the same time. Figure 7.28 shows the total actual weight and total importance value of the system as a function of time. Figures 7.29–7.36 depicts the actual and estimated weights and error for all of the tasks as a function of time. (As before, the mapping of tasks to regions of the screen is as given in Figure 7.9. Thus, T_1 corrects the upper-left corner of the screen, and T_{16} corrects the lower-right corner of the screen.) There are a few interesting things to notice about these graphs. First, the error of the task in Figures 7.29–7.36 is typically within the range $[-0.08, 0.08]$, except when either the system starts or a major change to the system occurs, i.e., darkness. Second, the total importance value of the system has less jitter than for Whisper. This is because in our experimental set-up, fewer tasks are changing at any given point in time. Third, the weights of tasks correcting the corners of the screen (i.e., T_1, T_4, T_{13}, and T_{16}) do not change that much (except when darkness is incurred). The reason why this behavior occurs is because the dark square barley enters the four corners. On the other hand, the weights of the center four tasks (i.e., T_6, T_7, T_{10}, and T_{11}) have substantial

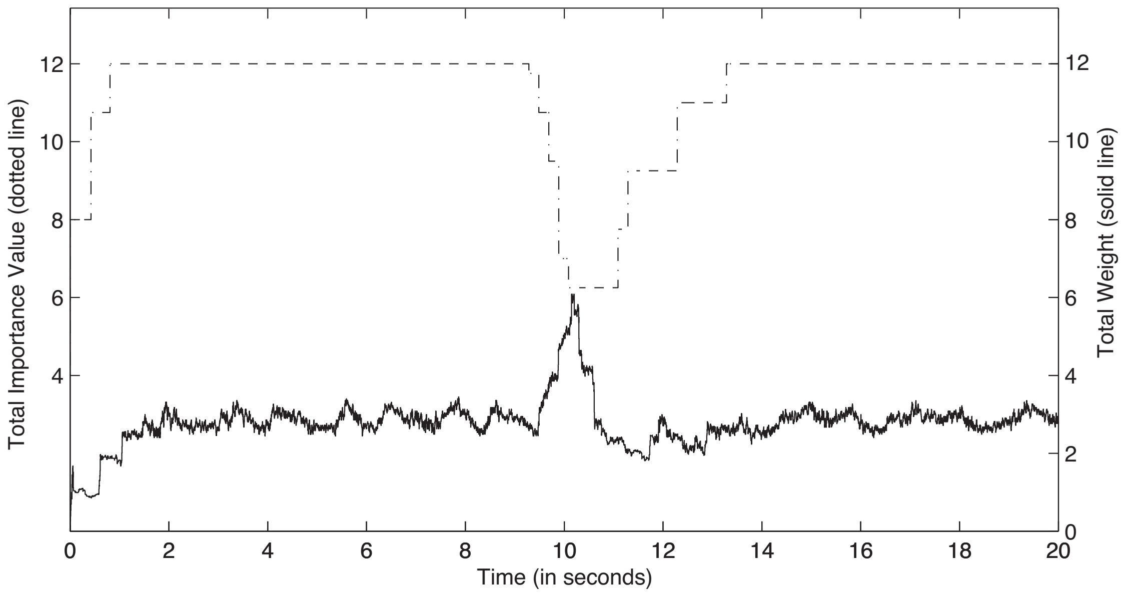
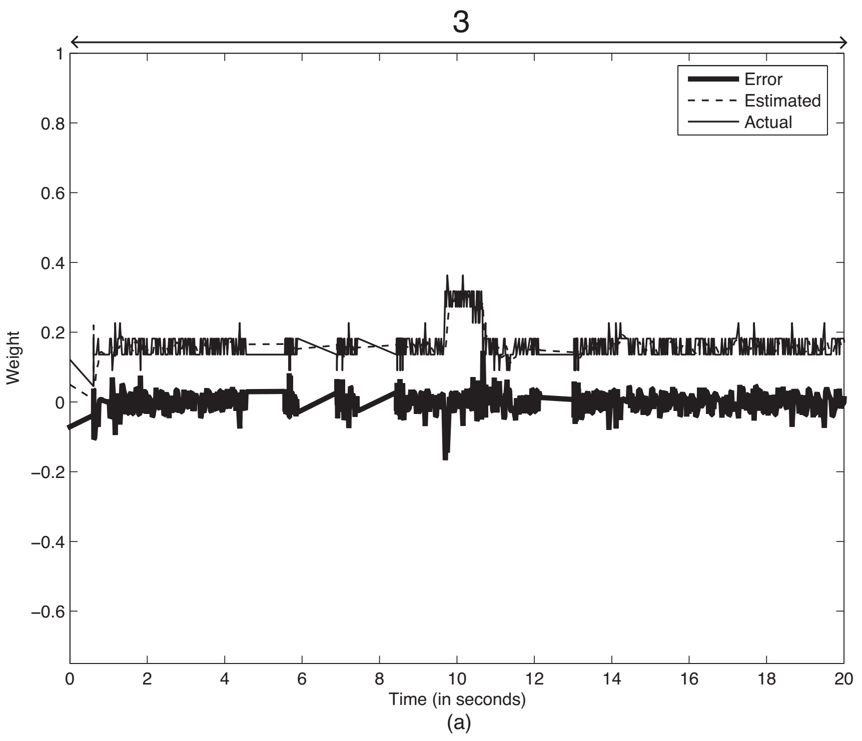


Figure 7.28: Results from executing 16 VEC tasks for 20 seconds. Total actual weight and importance value as a function of time.

variance. This behavior occurs because the dark square frequently covers a large fraction of the center four regions.

One final note. Notice that, in the above evaluations of AGEDF, we used real code from both Whisper and VEC in simulations of deterministic scenarios. Since we are primarily interested in the behavior of AGEDF rather than the behavior of either Whisper or VEC, we have chosen to simulate simple environments. These simulations (elements moving around in a circle with occasional bursts of noise or darkness) provide a sufficient evaluation of AGEDF since they allow us to measure its behavior when scheduling systems that require a substantial amount of adaption (in both systems, tasks continually change their execution time and there are occasional periods of stress). As a result, if we were to simulate more complex scenarios (objects moving in random or different deterministic patterns), then the results would be similar. In addition, by evaluating simple scenarios, we can more easily discern the efficacy of AGEDF when scheduling either Whisper or VEC.



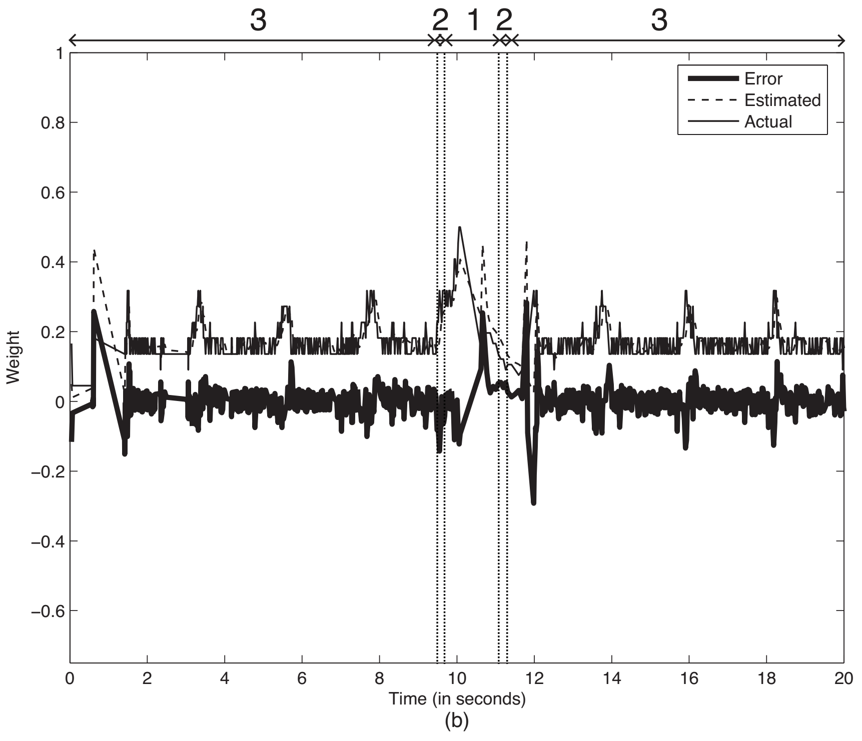
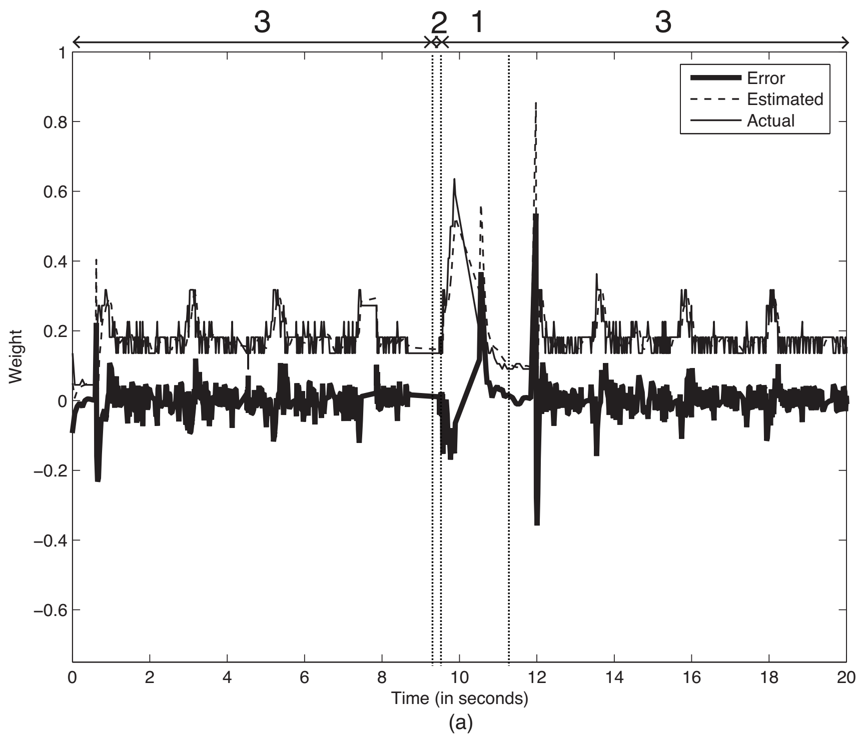


Figure 7.29: Results from executing 16 VEC tasks for 20 seconds. Actual and estimated weights and error for **(a)** T_1 and **(b)** T_2 as a function of time. The service level for each task is depicted across the top of each inset. In both insets, the error line is centered around zero, and the actual and estimated lines are often indistinguishable.



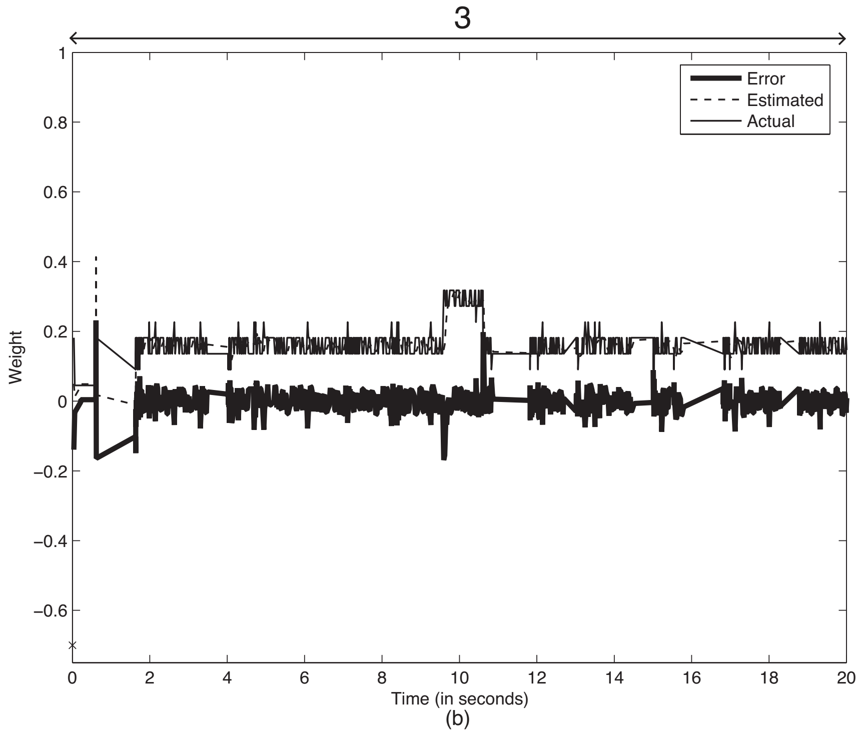
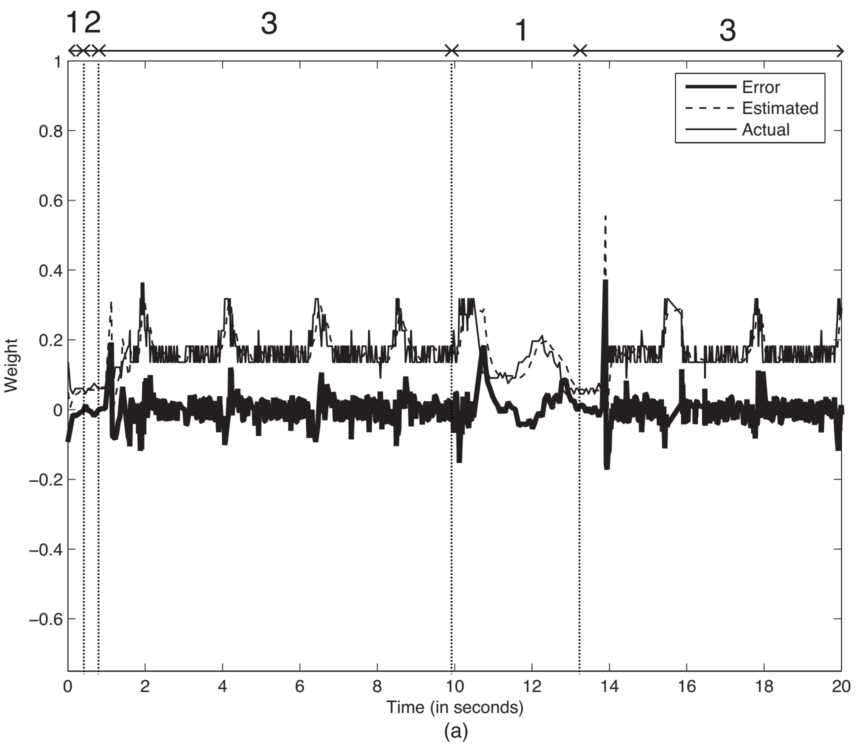


Figure 7.30: Results from executing 16 VEC tasks for 20 seconds. Actual and estimated weights and error for **(a)** T_3 and **(b)** T_4 as a function of time. The service level for each task is depicted across the top of each inset. In both insets, the error line is centered around zero, and the actual and estimated lines are often indistinguishable.



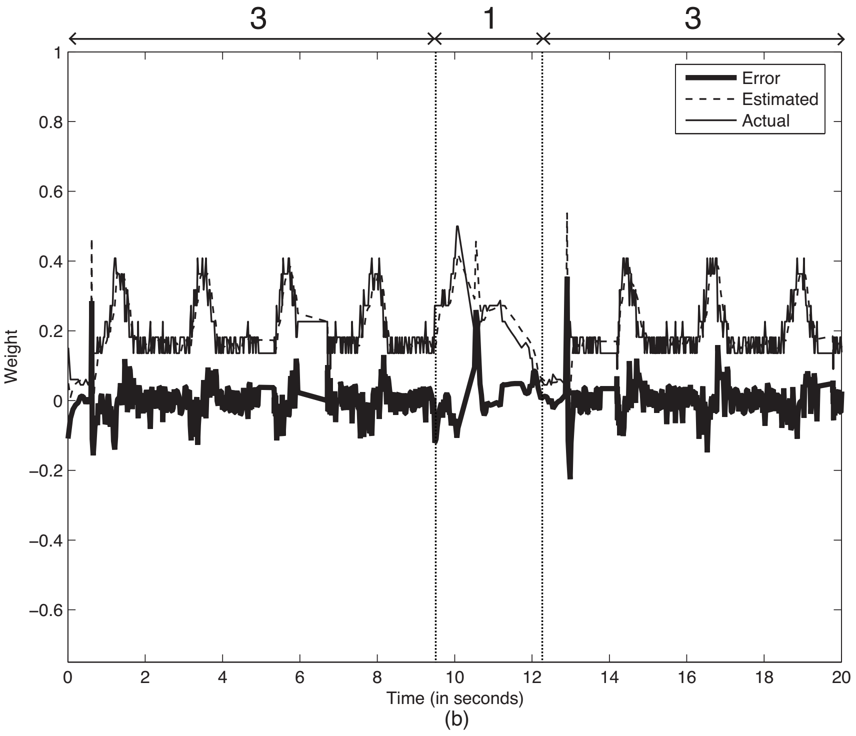
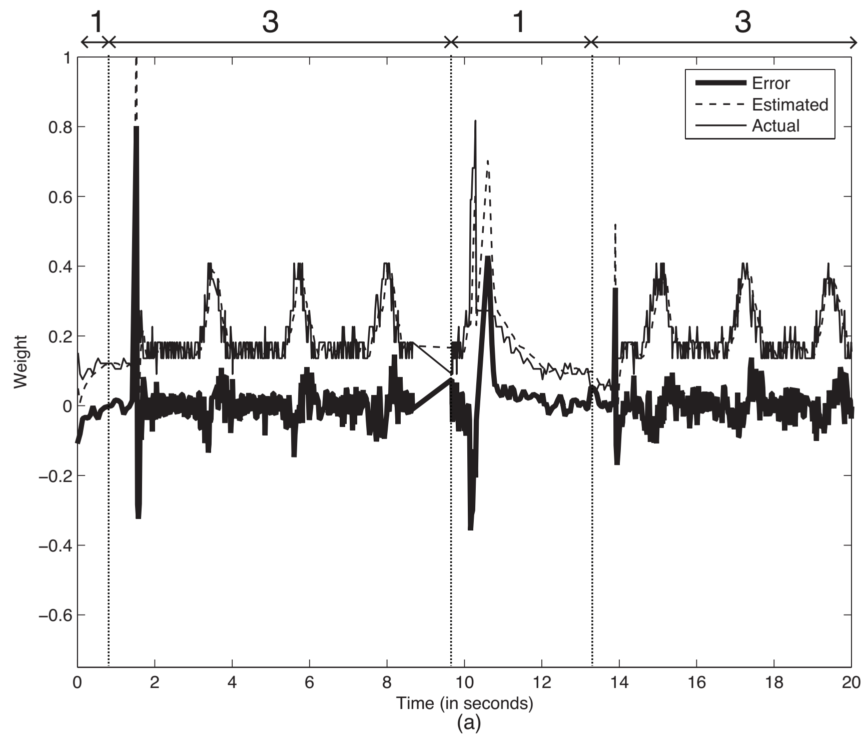


Figure 7.31: Results from executing 16 VEC tasks for 20 seconds. Actual and estimated weights and error for **(a)** T_5 and **(b)** T_6 as a function of time. The service level for each task is depicted across the top of each inset. In both insets, the error line is centered around zero, and the actual and estimated lines are often indistinguishable.



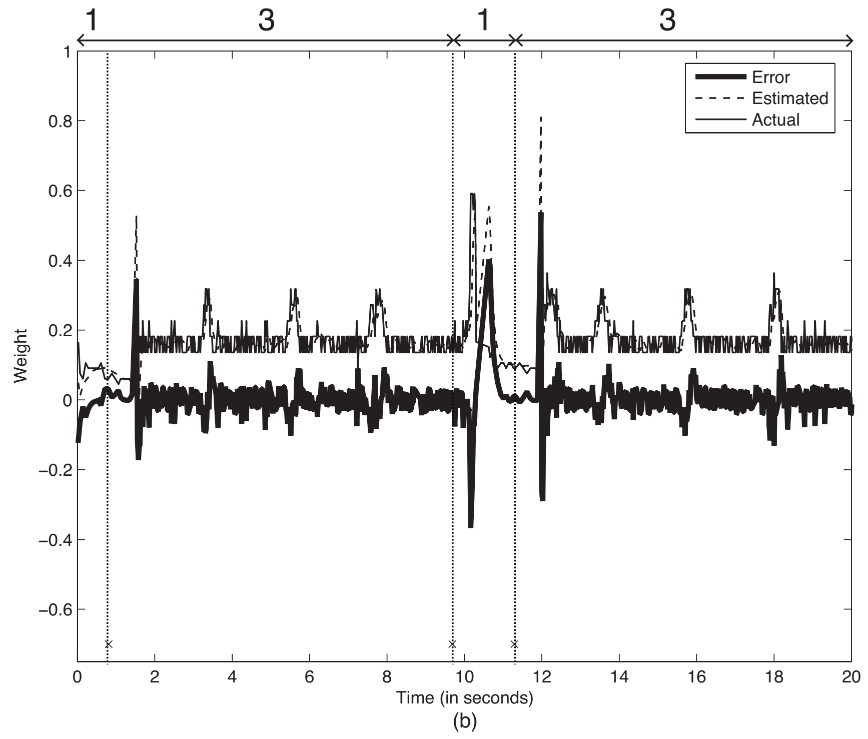
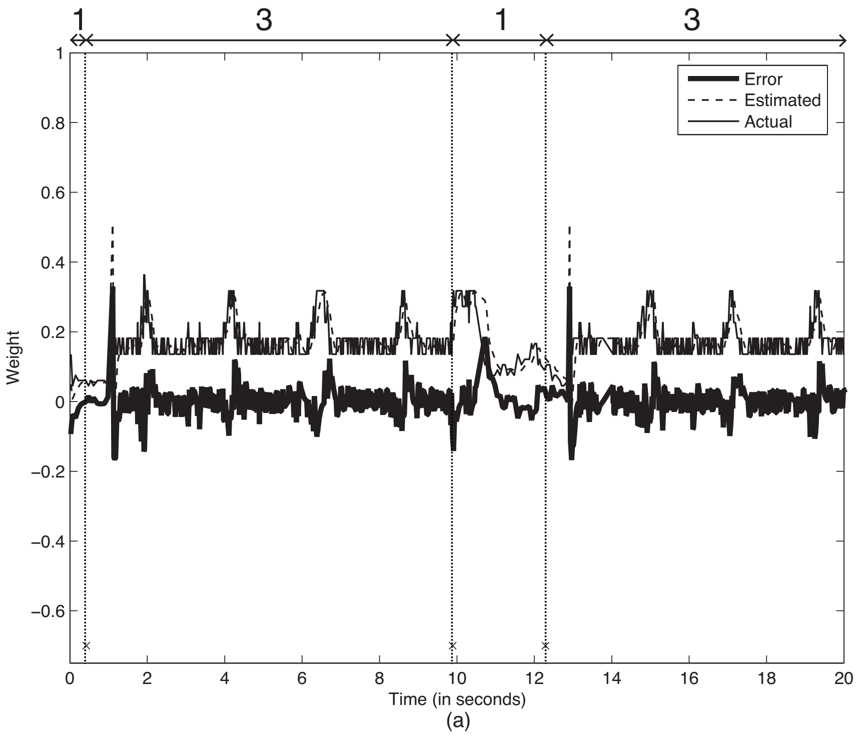


Figure 7.32: Results from executing 16 VEC tasks for 20 seconds. Actual and estimated weights and error for **(a)** T_7 and **(b)** T_8 as a function of time. The service level for each task is depicted across the top of each inset. In both insets, the error line is centered around zero, and the actual and estimated lines are often indistinguishable.



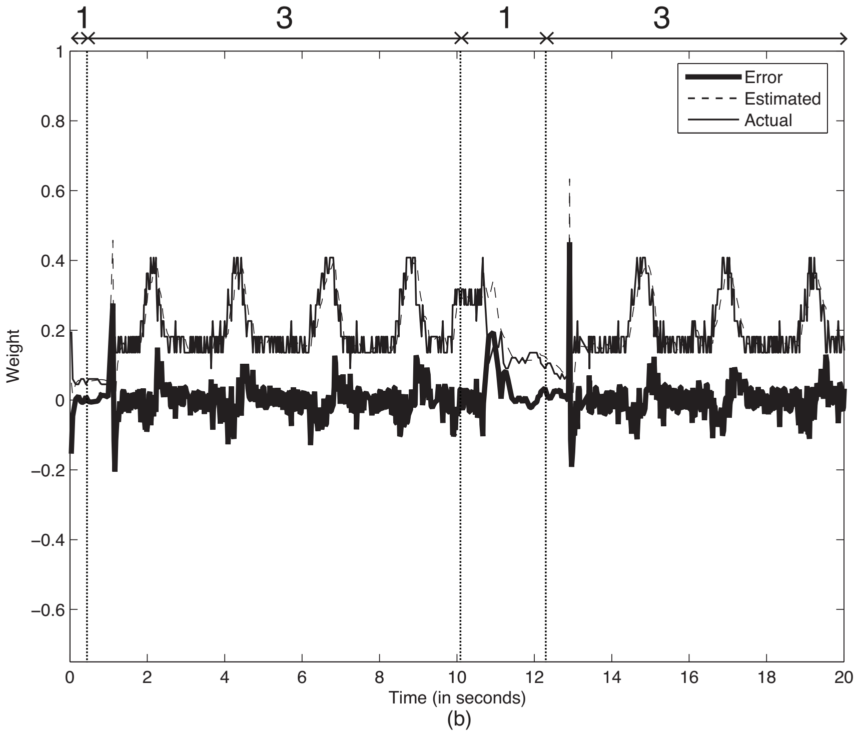
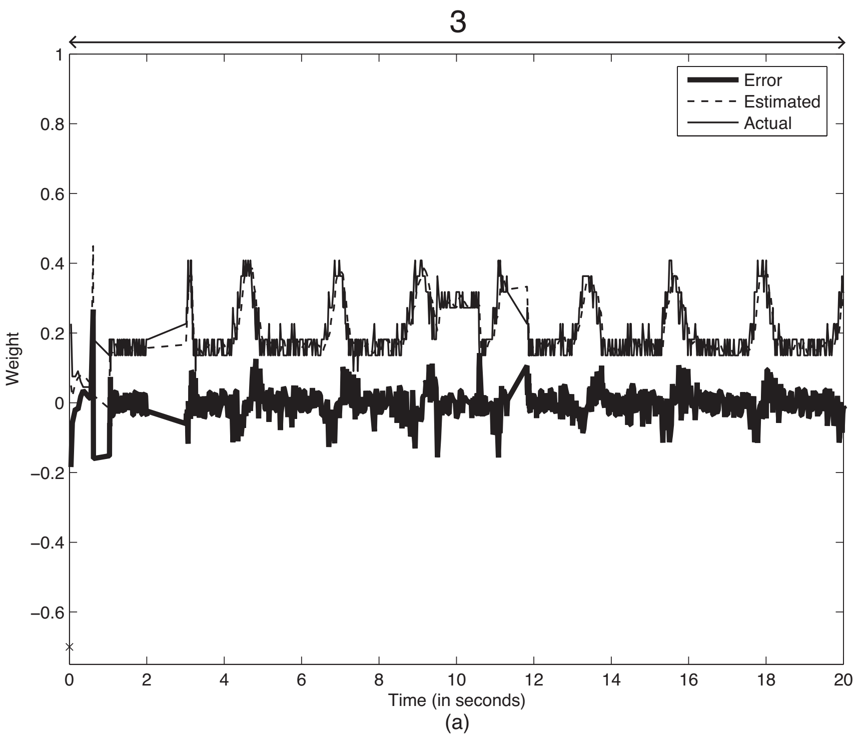


Figure 7.33: Results from executing 16 VEC tasks for 20 seconds. Actual and estimated weights and error for **(a)** T_9 and **(b)** T_{10} as a function of time. The service level for each task is depicted across the top of each inset. In both insets, the error line is centered around zero, and the actual and estimated lines are often indistinguishable.



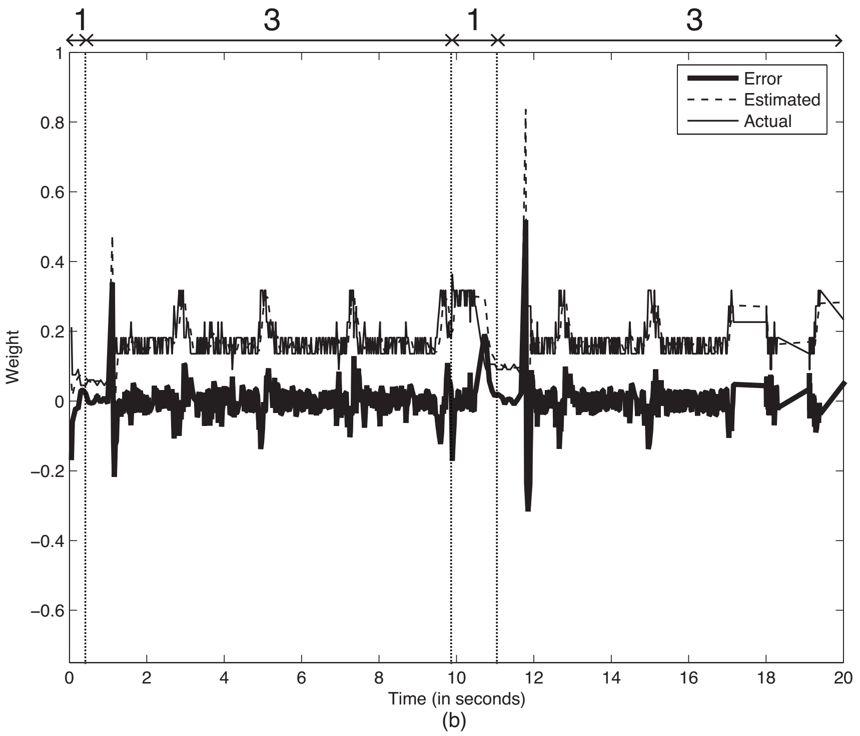
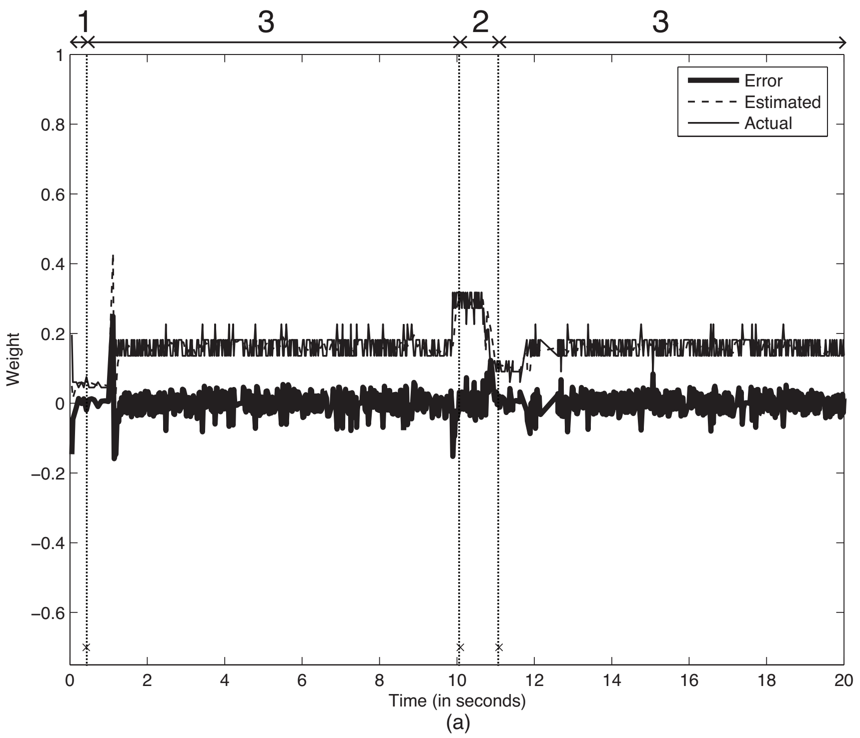


Figure 7.34: Results from executing 16 VEC tasks for 20 seconds. Actual and estimated weights and error for **(a)** T_{11} and **(b)** T_{12} as a function of time. The service level for each task is depicted across the top of each inset. In both insets, the error line is centered around zero, and the actual and estimated lines are often indistinguishable.



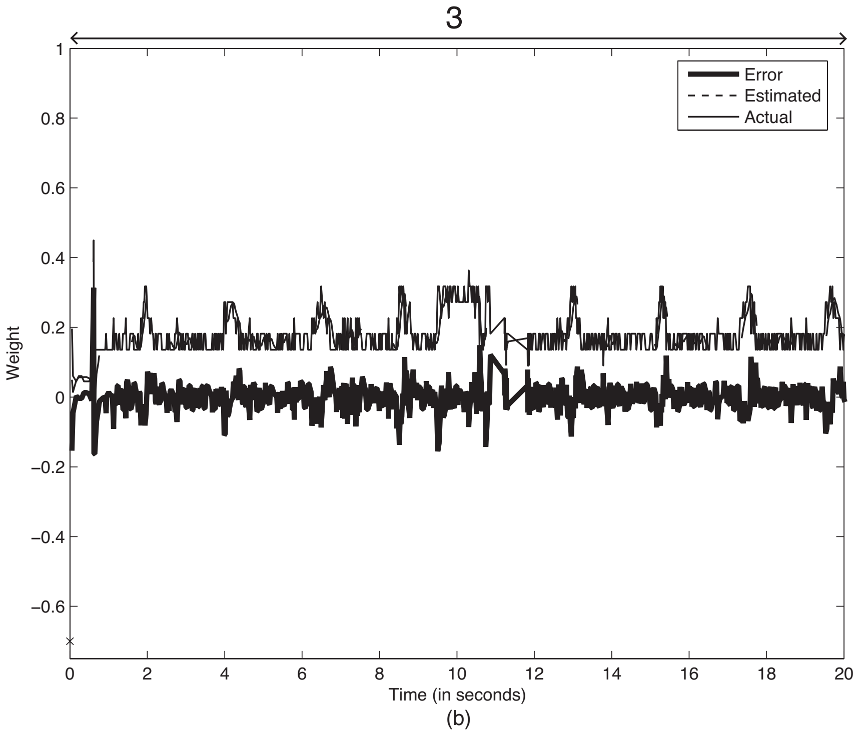
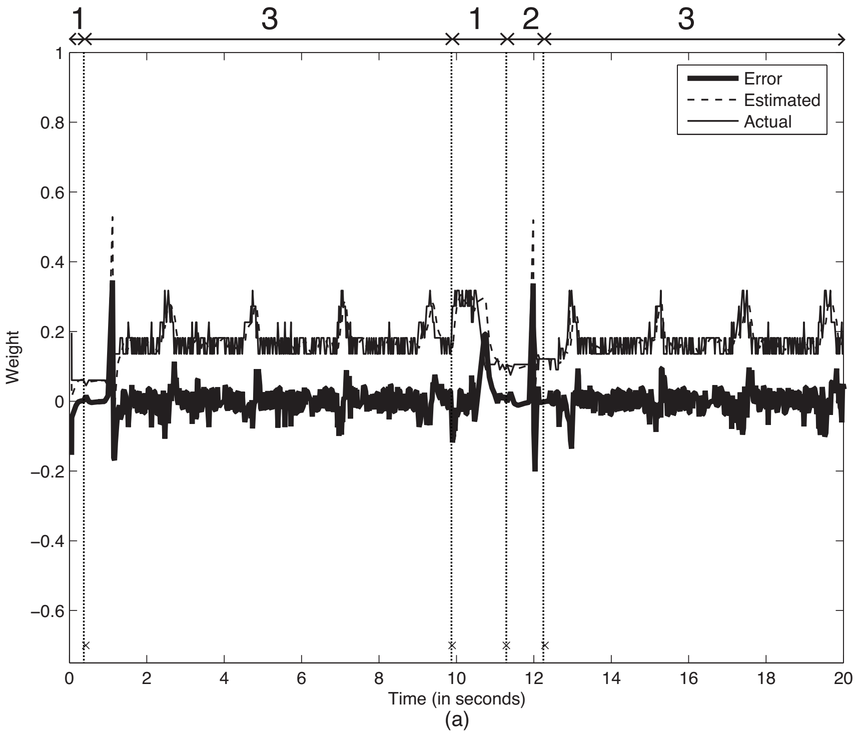


Figure 7.35: Results from executing 16 VEC tasks for 20 seconds. Actual and estimated weights and error for **(a)** T_{13} and **(b)** T_{14} as a function of time. The service level for each task is depicted across the top of each inset. In both insets, the error line is centered around zero, and the actual and estimated lines are often indistinguishable.



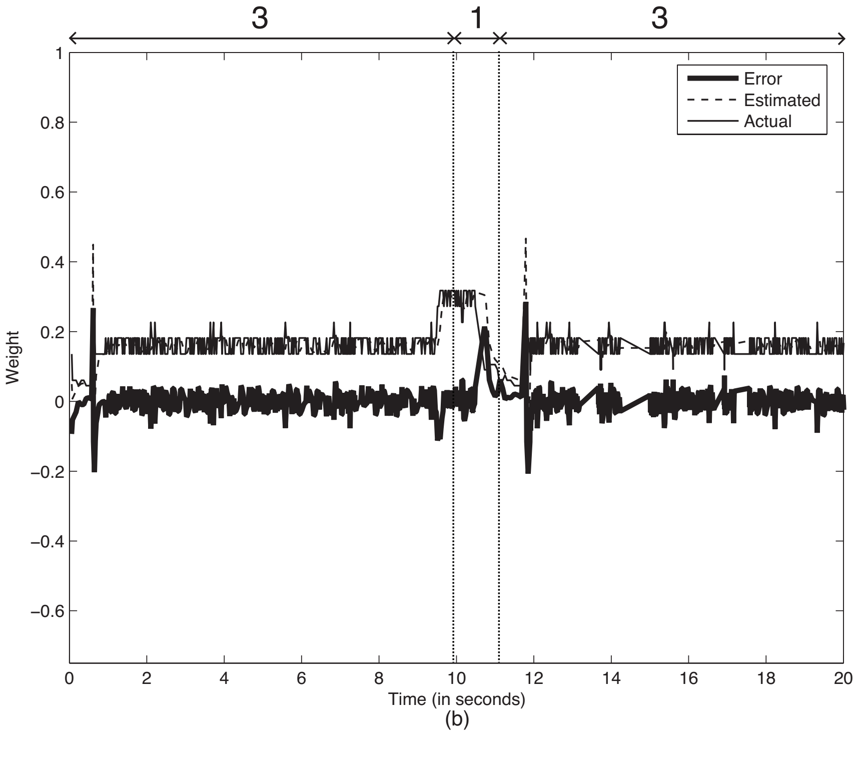


Figure 7.36: Results from executing 16 VEC tasks for 20 seconds. Actual and estimated weights and error for **(a)** T_{15} and **(b)** T_{16} as a function of time. The service level for each task is depicted across the top of each inset. In both insets, the error line is centered around zero, and the actual and estimated lines are often indistinguishable.

7.6 Conclusion

In this section, we presented two sets of experiments. First, we presented a simulation-based comparison of our adaptive variants of GEDF, NP-GEDF, PEDF, NP-PEDF, and PD^2. Second, we presented an implementation and evaluation of our AGEDF framework using LITMUS$^{\text{RT}}$. The results of our simulation-based comparison suggest the following: first, when it is critical that every task make its deadline and migration/preemption costs are low (i.e., systems like Whisper), PD^2 is the best choice. Second, when preemption/migration costs are high (i.e., either Whisper or VEC as implemented on a system where the processors are not as tightly integrated), average case performance is of the utmost importance, and fairness and timeliness are less important, then either PEDF or NP-PEDF may be the best choice. Third, when migration/preemption costs are high and a good mix of average-case performance and fairness factors is beneficial (i.e., systems like VEC), then either GEDF or NP-GEDF may the best choice. In addition, our evaluation of the AGEDF framework shows that it is capable of enacting needed adaptions in a way that enhances overall QoS for both Whisper and VEC.

CHAPTER 8

CONCLUSION AND FUTURE WORK

In research on real-time systems, *multiprocessor* platforms are of growing importance, due to both hardware trends such as the emergence of multicore technologies and the prevalence of computationally-intensive applications for which single-processor designs are insufficient. While research on real-time systems has traditionally focused on applications with static timing constraints, for many applications, such constraints can change at run time. For these applications, *adaptive* real-time scheduling techniques are needed. This dissertation has focused on developing *multiprocessor adaptive real-time* scheduling techniques and studying the usefulness of such techniques when developing computationally-intensive multimedia applications such as human-tracking and night-vision systems.

8.1 Summary of Results

In this dissertation, we examined the thesis that *multiprocessor real-time scheduling algorithms can be made more adaptive by allowing tasks to reweight between job releases. Feedback and optimization techniques can be used to determine at run time which reweighting events are needed. The accuracy of such an algorithm can be improved by allowing more frequent task migrations and preemptions; however, this accuracy comes at the expense of higher migration and preemption costs, which impacts average-case performance. Thus, there is a tradeoff between accuracy and average-case performance that will be dependent on the frequency of task migrations/preemptions and their cost.*

The main difficulty in constructing an adaptable system is that it is often necessary to delay *enacting* weight changes that have been *initiated*. (If weight changes are always immediately enacted, then it is possible for a task to artificially boost its weight, possibly causing over-utilization, by aggressively requesting weight changes that, if delayed appropriately, would not cause over-utilization.) As a result, *fundamentally*, whenever a task's weight changes, there may be some "loss" to the system relative to an "ideal" system in which each weight change can be enacted as soon it is initiated. The allocation difference between the ideal and actual systems that arises for a single task as a result of one weight change is called *drift*. Before the research presented in this dissertation, the only available method for changing the weight of a task on a multiprocessor was for the task to "leave" with its old weight and "rejoin" with its new weight. The problem with this method is that drift can be *arbitrarily large*. (This is because a task is permitted to leave only after its next deadline. Since deadlines can be arbitrarily large, drift can be as well.) As a result, any adaptive algorithm that changes weights by such a method would be unresponsive to the frequent and substantial changes that occur in applications such as Whisper and VEC. Moreover, before the research presented in this dissertation, the only methods for multiprocessor systems for detecting when task weights should change, and the extent of change required, made assumptions about the behavior of tasks that are too conservative for these applications. In this section, we review the multiprocessor framework that we have presented and implemented for scheduling tasks that require adaptation, and our evaluation of this framework by using the core operations of Whisper and VEC (correlation computations for Whisper and bilateral filters for VEC). While this framework was designed for Whisper and VEC, it is general enough to be used for any real-time application that has a workload that is both intensive and time-varying.

Adaptive algorithm for restricted global systems. As stated in Chapter 3 under restricted global scheduling algorithms (i.e., GEDF and NP-GEDF), jobs may miss their deadlines by a bounded amount. In Chapter 3, we presented rules that allow a task to change its weight, even if that task has expired deadlines.

Adaptive algorithm for partitioned systems. In Chapter 4, we presented for partitioned scheduling algorithms (i.e., PEDF and NP-PEDF) rules for changing the weight of a task that are similar to the rules for restricted global scheduling. However, because in partitioned systems each task is assigned to a specific processor, it is possible that when task weights change, a single processor may become *overloaded*, i.e., the sum of the weights of all tasks assigned to it is greater than one. To resolve this issue, we presented two techniques in Chapter 4. In the first, the guaranteed weight each task receives is proportional to its "desired" weight relative to the "desired" weight of all tasks assigned to the same processor. By assigning task weights in this manner (i.e., allocating to a task a guaranteed weight that is less then its desired weight), it is possible to guarantee that jobs do not miss deadlines even when a processor is overloaded. (While this technique will guarantee that deadlines are not missed, this behavior only occurs because tasks that are assigned to an overloaded processor receive a guaranteed weight less than their desired weights. If tasks were to receive their desired weights, then, in such a case, deadline tardiness would grow *unboundedly*.) In the second technique, whenever a processor becomes overloaded by some user-defined value, the entire system is repartitioned. By initiating such repartitioning events, it can be guaranteed that no processor is overloaded for "too long."

Adaptive algorithm for unrestricted global systems. One of the most important unrestricted global scheduling algorithms is the Pfair algorithm PD^2. PD^2 is the most efficient known algorithm (in terms of achievable utilization) for scheduling a set of hard real-time sporadic tasks that fully utilize a multiprocessor system. In Chapter 5, we presented a modification of PD^2 that allows a task's weight to change at run-time with only a small, constant amount of drift.

Empirical comparison. In Section 7.4, we presented an experimental comparison of our adaptive variants of GEDF, NP-GEDF, PEDF, NP-PEDF, and PD^2. The results of our experimental comparison suggest the following: first, when it is critical for every task to meet its deadlines and migration/preemption costs are low (i.e., systems like Whisper), PD^2 is the best choice. Second, when preemption/migration costs are high (i.e., either Whisper or VEC as implemented on a system where the processors are not tightly integrated), average case performance is of the utmost importance, and both fairness and timeliness are less important, then either PEDF or NP-PEDF may be the best choice. Third, when migration/preemption costs are high and a good mix of average-case performance and fairness is beneficial (i.e., systems like VEC), then either GEDF or NP-GEDF may the best choice.

Multiprocessor feedback-controlled adaptive algorithm. One method for determining task weights is to assume that each task has multiple *service levels*, each of which represents a different level of QoS and a different processor weight. When a task has multiple service levels, there may exist times when it must be forced to change its service level because of system constraints. For example, in VEC, if one region is particularly dark, then tasks associated with other lighter regions may be forced to reduce their weights in order to provide more resources to the task assigned to the dark region. Forced changes in service levels present two challenges: how does the system detect when the service level of the system should change, and how are the service levels of tasks determined? In Chapter 6 we presented the *adaptable* GEDF (AGEDF) framework, which attempts to solve this problem by using feedback control and optimization techniques. In feedback-controlled systems, prior states of the system are used to predict the future state of the system. By employing feedback-control techniques, it is possible to determine needed service-level changes. In addition to constructing this algorithm, in Section 7.5, we presented an implementation of the AGEDF framework on LITMUS^{RT}, a real-time multiprocessor testbed developed by our research group, and evaluated its performance when performing the core operations of both Whisper and VEC. This evaluation showed that, by using a feedback-controlled mechanism, the QoS of both Whisper and VEC can be improved.

8.2 Other Related Work

In this section, we briefly discuss other contributions by the author to the field of real-time systems that are outside of the scope of this dissertation.

Quick-release fair scheduling. One drawback to PD^2 is that it is not *work conserving*, i.e., a processor can become idle even if there exists pending work. This can cause task response times to be unnecessarily long. In prior work on introducing work conserving behavior to PD^2, techniques were used that can cause a task that consumes otherwise-idle processing capacity to be "unfairly" penalized later. In joint work with Anderson and Srinivasan, a technique called *quick-release fair scheduling* was developed that ensures that such tasks are treated fairly (Anderson et al., 2003).

LITMUS$^{\mathbf{RT}}$. In order to better understand how PD^2, GEDF, NP-GEDF, PEDF, and NP-PEDF would behave in practice, our research group constructed the aforementioned LITMUS$^{\mathrm{RT}}$ real-time multiprocessor testbed. Using this system, we evaluated the performance of PD^2, GEDF, NP-GEDF, PEDF, and NP-PEDF. We found that, for hard real-time systems, if tasks have relatively small weights, PEDF performs better in terms of scheduability; however, if task weights are relatively large, then PD^2 has superior performance. On the other hand, if bounded deadline tardiness is acceptable, then GEDF and NP-GEDF have superior performance regardless of any task's weight (Brandenburg et al., 2007; Brandenburg et al., 2008; Calandrino et al., 2006).

Multiprocessor synchronization. In work on multiprocessor real-time systems, there is a dearth of research on task synchronization techniques for use in scheduling algorithms where task priorities can change at run time (e.g., PD^2, GEDF, NP-GEDF, PEDF, and NP-PEDF). To remedy this situation, members of our research group developed the *flexible multiprocessor locking protocol* (FMLP), which is capable of synchronizing tasks via either semaphores or non-preemptable queue locks in globally-scheduled systems. We also implemented this protocol in LITMUS$^{\mathrm{RT}}$ and compared both the semaphore and queue-lock versions of the FMLP with each other, and, when implementing shared data objects, with lock-free and wait-free algorithms. We found that for simple data structures, lock-free and wait-free approaches are superior to locking approaches; however, for more complex data structures, non-preemptable queue locks provide superior performance. Interestingly, we found that the use of semaphores *always results in worse scheduability than non-preemptable queue locks* (Block et al., 2007; Brandenburg et al., 2008).

8.3 Future Work

One limitation of this dissertation is that we were not able to fully re-implement Whisper and VEC. In order to do this, four objectives must be achieved. First, an *adaptive* synchronization protocol must be devised. Second, AGEDF should be extended to more complex machine models. Third, tools must be developed that allow software engineers to easily specify desired adaptive behaviors. Fourth, user studies are needed in order to optimize the performance of both Whisper and VEC. Below, we briefly outline a plan for achieving these three objectives.

An adaptive synchronization protocol. Such a protocol could be devised by modifying the aforementioned FMLP. While an adaptive variant of this protocol will probably be similar to the non-adaptive version, determining the protocol's impact on scheduability will likely be complex since the non-adaptive protocol's scheduability analysis is based on the assumption that task weights do not change. Another complication with the construction of an adaptive FMLP is that critical sections impose additional constraints on the times at which weight changes can be enacted. As a result, introducing synchronization into an adaptive system will likely increase the maximal possible drift because of synchronization-related delays in enacting weight changes.

More complex machine models. In addition to constructing an adaptive synchronization protocol, it would be interesting to extend AGEDF to incorporate more complex machine models in which various factor (e.g., caching, page faults, TLB misses, etc.) that affect execution costs are directly considered. One complication with such machine models is that it becomes more difficult to predict the future weight of a task based solely on its prior weights. For this reason, in more complex machine models, it may be worthwhile for applications to provide additional information about the expected weight of a task to assist in the calculation of future task weights.

Developer tools. One of the advantages of AGEDF is that there are several parameters that can be adjusted for each task in order to control how an application responds to workload changes. In order for a user to take advantage of this flexibility, it would be desirable to have a set of tools that assist a developer in choosing these parameters under different types of system load. In addition, constructing multithreaded code where each thread (i.e., task) has multiple service levels is a non-trivial issue. Such a toolset could be configured to include three utilities. The first would allow a developer to express a desired system response to different types of workload changes, and from this information, compute appropriate values for the AGEDF parameters. The second would consist of a simulator that allows a developer to quickly assess the behavior of AGEDF for a given set of parameters. The final utility would assist the developer in constructing multithreaded code where each thread may have multiple versions and periods.

User studies. Since both Whisper and VEC are multimedia applications, the ultimate metric of success is user perception. Thus, while many of the parameters of AGEDF can be chosen by simulations conducted by the developer, fine-tuning the parameters for Whisper and VEC will involve conducting user studies. Upon completing these three objectives, we will have built a system for which applications can be easily constructed to fully utilize a multiprocessor system while at the same time providing real-time guarantees.

BIBLIOGRAPHY

Abeni, L. and Buttazzo, G. (1998). Integrating multimedia applications in hard real-time systems. In *Proceedings of the19th IEEE Real-Time Systems Symposium*, pages 4–13. IEEE Computer Society.

Abeni, L., Palopoli, L., Lipari, G., and Walpole, J. (2002). Analysis of a reservation-based feedback scheduler. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 71–80. IEEE Computer Society Press.

Al-Omari, R., Manimaran, G., Salapaka, M. V., and Somani., A. K. (2003). Novel algorithms for open-loop and closed-loop scheduling of real-time tasks in multiprocessor systems based on execution time estimation. In *Proceedings of the 2003 International Parallel and Distributed Processing Symposium*, pages 7–14. IEEE Computer Society Press.

Anderson, J., Block, A., and Srinivasan, A. (2003). Quick-release fair scheduling. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 130–141. IEEE Computer Society Press.

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

Bennett, E. (2007). *Computational Video Enhancement*. PhD thesis, University of North Carolina at Chapel Hill.

Bennett, E. and McMillan, L. (2005). Video enhancement using per-pixel virtual exposures. *ACM Transactions on Graphics (SIGGRAPH)*, 24(3):845–852.

Bertsekas, D. (1999). *Nonlinear Programming*. Athena Scientific, second edition.

Block, A. and Anderson, J. (2006). Accuracy versus migration overhead in multiprocessor reweighting algorithms. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 355–364. IEEE Computer Society Press.

Block, A., Anderson, J., and Bishop, G. (2008a). Fine-grained task reweighting on multiprocessors. *Journal of Embedded Computing, Special Issue on Multiprocessor Real-Time Scheduling* (to appear).

Block, A., Anderson, J., and Devi, U. (2008b). Task reweighting under global scheduling on multiprocessors. *Real-Time Systems, Special Issue on Selected Papers from the 18th Euromicro Conference on Real-Time Systems*, 39:123–167.

Block, A., Brandenburg, B., Anderson, J., and Quint, S. (2008c). Feedback-controlled adaptive multiprocessor real-time systems. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 23–33. Kluwer Academic Publishers.

Block, A., Leontyev, H., Brandenburg, B., and Anderson, J. (2007). A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–57. IEEE Computer Society Press.

Brandenburg, B. and Anderson, J. (2008). A comparison of the M-PCP, D-PCP, FMLP, on LITMUS$^{\mathrm{RT}}$(in submission).

Brandenburg, B., Block, A., Calandrino, J., Devi, U., Leontyev, H., and Anderson, J. (2007). LITMUS$^{\mathrm{RT}}$: A status report. In *Proceedings of the 9th Real-Time Linux Workshop*, pages 107–123. The Real-Time Linux Foundation.

Brandenburg, B., Calandrino, J., Block, A., Leontyev, H., and Anderson, J. (2008). Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353. IEEE Computer Society Press.

Brandt, S. A., Banachowski, S., Lin, C., and Bisson, T. (2003). Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 396–407. IEEE Computer Society Press.

Calandrino, J., Leontyev, H., Block, A., Devi, U., and Anderson, J. (2006). LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 111–126. IEEE Computer Society Press.

Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, J., and Baruah, S. (2004). *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms, pages 30.1–30.19. Chapman and Hall/CRC.

Chen, C. and Tripathi, S. (1994). Multiprocessor priority ceiling based protocols. CS-TR-3252, University of Maryland.

Cucinotta, T., Palopoli, L., Marzario, L., Lipari, G., and Abeni, L. (2004). Adaptive reservations in a linux environment. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 238–245. IEEE Computer Society Press.

Devi, U. and Anderson, J. (2008). Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189.

Devi, U., Leontyev, H., and Anderson, J. (2006). Efficient synchronization under global EDF scheduling on multiprocessors. In *18th Euromicro Conference on Real-Time Systems*, pages 75–84. Kluwer Academic Publishers.

Gai, P., Natale, M. D., Lipari, G., Ferrari, A., Gabellini, C., and Marceca, P. (2003). A comparison of MPCP and MSRP when sharing resources in the Janus multiple processor on a chip platform. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 189–198. IEEE Computer Society Press.

Hamadoui, M. and Ramanathan, P. (1995). A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451.

Holman, P. and Anderson, J. (2006). Locking under Pfair scheduling. *ACM Transactions on Computer Systems*, 24(2):140–170.

Lopez, J. M., Diaz, J. L., and Garcia, D. F. (2004). Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68.

Lu, C., Stankovic, J., Abdelzaher, T., Gang, T., Son, S., and Marley, M. (2000). Performance specifications and metrics for adaptive real-time systems. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 13–23. IEEE Computer Society Press.

Lu, C., Stankovic, J., Son, S., and Tao, G. (2002). Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23(1-2):85–126.

Lu, C., Stankovic, J., Tao, G., and Son, S. (1999). Design and evaluation of a feedback control EDF scheduling algorithm. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 56–67. IEEE Computer Society Press.

Marti, P., Lin, C., Brandt, S., Velasco, M., and Fuertes, J. (2004). Optimal state feedback based resource allocation for resource-constrained control tasks. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 161–172. IEEE Computer Society Press.

Maybeck, P. (1979). *Stochastic models, estimation, and control*, volume 141 of *Mathematics in Science and Engineering*. Academic Press, Inc.

Mellor-Crummey, J. and Scott, M. (1991). Algorithms for synchronization on shared-meory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65.

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

Nise, N. (2004). *Control Systems Engineering*. Wiley and Sons, fourth edition.

Pappas, T. and Safranek, R. (2000). *Handbook of Image and Video Processing*, chapter Perceptual Criteria for Image Quality Evaluation, pages 669–684. Academic Press, Inc.

Rajkumar, R. (1991). *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers.

Sahoo, D., Swaminathan, S., Al-Omari, R., Salapaka, M., Manimaran, G., and Somani, A. (2002). Feedback control for real-time scheduling. In *Proceedings of the 21st American Control Conference*, Volume 2, pages 1254–1259. IEEE Computer Society Press.

Sha, L., Rajkumar, R., and Lehoczky, J. (1990). Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185.

Smith, S. (1997). *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing. Available at www.dspguide.com.

Srinivasan, A. (2003). *Efficient and Flexible Fair Scheduling of Real-time Tasks on Multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill.

Srinivasan, A. and Anderson, J. (2005). Fair scheduling of dynamic task systems on multiprocessors. *Journal of Software Systems*, 77(1):67–80.

Srinivasan, A. and Anderson, J. (2006). Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Science*, 72(6):1094–1117.

Stoica, I., Abdel-Wahab, H., Jeffay, K., Baruah, S., Gehrke, J. E., and Plaxton, C. G. (1996). A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 288–299. IEEE Computer Society Press.

Vallidis, N. (2002). *WHISPER: A Spread Spectrum Approach to Occlusion in Acoustic Tracking*. PhD thesis, The University of North Carolina at Chapel Hill, North Carolina.

Welch, G. and Bishop, G. (1995). An introduction to the Kalman filter. Technical Report TR95-041, Department of Computer Sceience. The Univerity of North Carolina at Chapel Hill.