REAL-TIME SYSTEM DESIGN UNDER PHYSICAL AND RESOURCE CONSTRAINTS

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Thidapat Chantem

Xiaobo Hu, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
April 2011

REAL-TIME SYSTEM DESIGN UNDER
PHYSICAL AND RESOURCE CONSTRAINTS

Abstract

by

Thidapat Chantem

This dissertation presents several design techniques for resource-constrained, dependable embedded real-time systems, which can be found everywhere from cellular phones to automotive electronic systems to medical devices. Specifically, three important challenges in embedded real-time system design are identified and addressed: temporal overloads, energy concerns, and temperature problem.

Due to cost constraints, as well as our inability to foresee worst-case operating scenarios, many systems are designed based on average-case scenarios and must provide graceful performance degradation during occasional system overloads. In this dissertation, we focus on two types of overloads: processor overloads and network overloads. Processor overload occurs because of user's inputs or external conditions. We present robust algorithms that allow the system to quickly handle different operating scenarios without requiring advanced knowledge of the situations based on optimization theory. As a result, fewer resources are needed to deliver expected performance.

As for network overloads, which can often occur especially now that most electronic devices have wireless capability, we discuss an energy-aware integrated framework that allows for the processor and network card to work together to

transmit the most important information first. This framework is robust against rapid changes in network conditions, guarantees the transmission of the most important set of information, and allows the system to save energy.

Finally, as device sizes continue to shrink, physical constraints such as system temperature have become a main concern. High temperature can severely reduce both performance and system lifetime. We first present an optimal voltage selection policy to maximize computation without exceeding the temperature threshold. We then present a method to minimize the peak temperature of a hard real-time system running on multi-core architectures that help to avoid deadline misses at run time.

To my parents, who have always insisted that
education is their only lasting gift to me.

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENTS

This dissertation is a result of many years of mentoring and much patience on the part of my advisor, Dr. Sharon Hu. From the day you took me on as a brand new graduate student to today, you have always made time for me and have taught me the value of hard work and persistence. Thank you for making me into the researcher I am today.

To Dr. Robert Dick, I have many reasons to thank you but I am especially grateful for your philosophy in not only producing high quality products but also that the process matters. Because of your high standards, I have continued (and will continue) to improve.

Dr. Michael Lemmon, thank you for your wisdom, especially during my early years at Notre Dame. You were instrumental in my growth as a researcher and I have fond memories of our discussions. Your sense of humor is simply refreshing.

To Dr. Christian Poellabauer, it never ceases to amaze me how you always give incredibly insightful suggestions and ask important (and often overlooked) questions in the most casual way possible. Thank you for keeping me grounded and for reminding me of the true motivations behind my research.

I would like to thank Dr. Liqiang Zhang not only for lending me his expertise in networking, but also for his helpful suggestions during our discussions on my career path and employment search.

The work in this dissertation is supported in part by NSF under grant numbers

CNS-0347941, CNS-0404341, CNS-0410771, CCF-0444405, IIS-0536994, CCF-0702705, CCF-0702761, CNS-0720691, CNS-0834180, CNS-0834230, CCF-0964763, and CNS07-20457, and in part by SRC under grant numbers 2007-TJ-1589 and 2007-HJ-1593. In addition, I gratefully acknowledge the support of the U.S. Department of Education through a GAANN Fellowship (award P200A090044).

I would like to thank my early mentors, Dr. Manimaran Govindarasu and Dr. Mani Mina, for their guidance during my tenure at Iowa State and thereafter. You always have time for me and I feel so privileged. A heartfelt thank goes to all my teachers and professors without whom I would not be here today.

Big thanks are due to my colleagues whom I have worked with or received help from (in no particular order): Dr. Bren Mochocki, Dr. Xiaofeng Wang, Jun Yi, Yun Xiang, Shengyan Hong, David Bild, Xi Chen, Lide Zhang, and Dinesh Rajan.

I am indebted to the support staff in the department. Thank you Curt, M.D., Joyce, Ginny, and Diane.

I have been so fortunate to have such a strong support system, which I find in my family and friends. Thank you Moms and Dads, Grandmas, and Josh. I love you all. To my sister Kate, who takes such good care of me I feel like I am the little sister sometimes, thank you and I love you. I also owe a big thank to all my friends around the world.

Finally, I would like to thank my husband, Ryan, especially for having driven at least 62,000 miles over the years to see me, for letting me make up food menus and cooking the meals without complaints, and, quite simply, for being just the way he is.

CHAPTER 1

INTRODUCTION

This chapter introduces readers to some basic concepts in real-time systems as well as important challenges in designing such systems due to physical and resource constraints (temporal overloads, energy concerns, and temperature problem). The chapter concludes with an outline of the main contributions in this dissertation.

1.1 Overview of Real-Time Systems

In a general-purpose computing system, the primary performance requirement is correct functionality. For instance, if a user wishes to obtain a list of the first thousand prime numbers, a system meets a functionality requirement if it correctly outputs such a list to the user.

In a real-time system, which can be thought of as a special case of general-purpose computing systems, timeliness, along with correct functionality, is also a requisite. Consider, for example, a train that is attempting to make an emergency stop within 5 seconds. The fact that the train does stop demonstrates that it performs its task correctly. However, the completion of this task is not useful, to say the least, if the train does not stop on time (i.e., within 5 seconds). A train that does is said to meet the timeliness requirement.

There are three classes of real-time systems: hard, firm, and soft. In a hard real-time system (e.g., the braking system of our example train), missing a dead-

line could result in catastrophic consequences. For such systems, offline analysis and hardware redundancy are often used to guarantee performance and timing requirements.

As for firm real-time systems, a few deadline misses are not detrimental to the overall performance of the system, but tasks that complete their executions after their deadlines are of no or even negative value. For instance, an out-of-date sample may trigger a control command that worsens the state of the system.

Finally, in a soft real-time system, deadlines may be missed or tasks may be completed late, though as many deadlines should be met as possible to maximize the performance of the entire system. For example, a user who watched a video clip online may rate his or her experience as mostly positive if he or she noticed only a few frame freezes during the entire session. The worst consequence that could result from an extremely unhappy user might be loss of business.

Regardless of the class of the real-time system under consideration, a real-time task is usually associated with worst-case, average-case, and best-case execution times. In this dissertation, we focus on tasks that need to be executed in a periodic manner. An example application might be a control task that needs to sample the water level in a pipe every 10 ms. For periodic real-time systems, task deadlines may be implicit (for the same control task just mentioned, the deadlines would be 10, 20, 30, ... ms), or explicit (e.g., a control task that must execute every 10 ms with a deadline of 5 ms, which means that the deadlines are 5, 15, 25, ... ms.

While task worst-case execution times are usually well-defined given that the physical characteristics of the system are known, acceptable task periods and deadlines may not be a fixed value. Referring back to our water level sampling example, the overall performance of the system may in fact be acceptable if a

sample is taken every 7 ms to 15 ms.

1.2 Design Challenges in Real-Time Systems

This dissertation tackles three important challenges in designing real-time systems: temporal overloads, power/energy concerns, and temperature problem. Each of these challenges is briefly discussed below.

1.2.1 Temporal Overload Conditions

For the purpose of this dissertation, we focus on two different types of temporal overloads: processing overloads and network overloads.

1.2.1.1 Processing Overloads

A real-time system is said to experience a processing overload if the computation demand exceeds the available computational resources. Consider, for example, the city sewer control system, which is responsible for drainage and waste management. Said system consists of several sensors and actuators, which are spread throughout the city and are controlled by a central computer. The sensors report, among other things, the level of water in their respective pipes, while the actuators may be used to control the flow rate of water in those pipes. Suppose now that part of the city has been experiencing a heavy rain. Since the water level in some of the pipes may be increasing at an alarming rate, we may need to sample the state of the pipes more frequently and send control commands to the relevant actuators more often to avoid flooding. Because of these new requirements, the city sewer control system is temporally under an overload situation, as more attention needs to be paid to the part of the city receiving heavy rainfall. For more

information regarding work in control systems on combined sewer overflow (CSO) events, the reader is referred to Ruggaber et al. [90] and Wan and Lemmon [103].

While temporal processing overload conditions can be alleviated to some extent using hardware overprovisioning, this solution is costly and wastes precious resources, especially since overloads tend to rarely occur and the average-case system utilization may be quite low. Additionally, in real systems, many factors are at work and it may be difficult, if not impossible, to predict the worst-case scenarios. For these reasons, many systems are designed based on the average operating scenarios and efficient run-time mechanisms to manage occasional overloads must be in place to rapidly solve the problem. An example overload management mechanism to the city sewer system may be to temporarily reduce the sampling (actuating) rate of sensors (actuators) that are not located in the part of the city receiving heavy rainfall in order to allocate more resources to managing the critical area in the city.

1.2.1.2 Network Overloads

A network overload is similar to a processing overload, except that network communications, and not the processing power, are the bottleneck. As most electronic devices now have wireless capability, network overloads occur more frequently, especially in sensor network applications where each node may need to transmit a large amount of data. During a network overload, only a limited amount of information can be transmitted and it is crucial in terms of performance maximization to judiciously select which information to send. For example, in a surveillance system used for intrusion detection, a video frame showing a potential intruder should take precedence over other information.

1.2.2 Power/Energy Concerns

As device size continues to shrink, a relatively larger number of transistors can be packed into a small area, causing an exponential increase in chip power density. Figure 1.1 shows the projected chip power density as a function of years. High power density results in high temperature (we will discuss temperature problem in more detail in the next section) and has several implications.

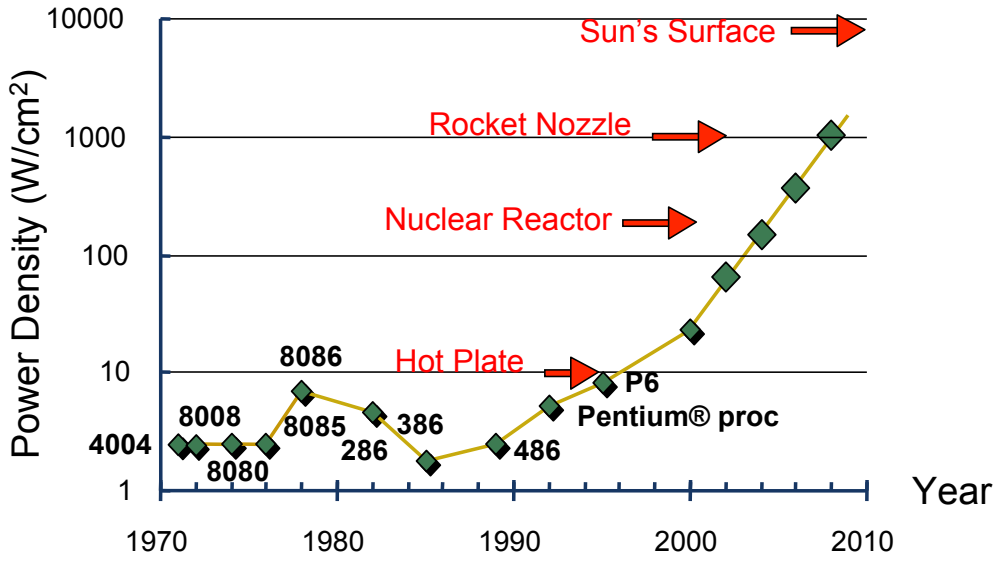

Figure 1.1. Power density trends (adapted from Rabaey [86])

.

1. In data centers, which can be considered soft real-time systems since response time is usually a main performance metric, about half of the power is used for cooling [37]. In 2005, about 1.2% of all U.S. electricity use went

towards total data center power and cost about $4.5 billion [37]. These numbers doubled from 2000 to 2005 and the trend is simply unsustainable [37].

2. As wireless nodes become common in a variety of applications, their operating lifetimes are now an important performance metric. Although battery technology has advanced over the years, the improvement remains modest and system designers are expected to use energy-aware methods to increase battery life without sacrificing performance.

3. Lastly, static or leakage power increases exponentially as supply voltage and threshold voltage are scaled down. According to Borkar, leakage power now makes up about 50% of total power consumption [14].

The high cooling cost and exponential increase in leakage power, as well as wireless and mobility requirements lead to an ever increasing need in designing power/energy-aware systems. Despite the large amount of existing literature on this topic, there are still many challenges to be addressed, especially as multicore systems with networking capabilities become more popular in a wide variety of real-time applications.

1.2.3 Temperature Problem

To achieve higher performance and better power efficiently, computer architects have proposed multiprocessor system-on-chips (MPSoCs) as an alternative to and improvement on uniprocessor architectures. However, as a result of shrinking device size, increasing transistor counts, and aggressive frequency scaling, MPSoCs often have high power density and temperature, which causes significant reliability concerns. In fact, a 10–15 °C difference in operating temperature can result

in a 2× difference in the lifespan of a device [102]. Even when high temperature does not immediately lead to permanent failures, it can cause increased interconnect latency and reduction of charge carrier mobility in transistors, which degrade performance. More importantly, high temperature leads to higher temperature-dependent leakage power, which in turns increases power consumption and hence temperature. Clearly, this cycle of dependency must to be broken.

A solution to the temperature problem in MPSoCs would be to design packages and cooling solutions that can handle worst-case thermal profiles. However, this solution is extremely expensive, as the cooling cost increases super-linearly in power consumption [43]. We have also discussed the high cost of cooling data centers in the previous section.

To reduce temperature, most modern processors use hardware throttling to control chip temperature at run time. Throttling is automatically activated when the chip temperature exceeds some prespecified threshold temperature. In addition to saving energy, techniques such as dynamic voltage and frequency scaling (DVFS) can also help to reduce temperature. The most difficult challenge that arises in the design of temperature-constrained real-time system is that processor throttling can cause real-time deadlines to be missed and therefore specific design techniques targeted at real-time systems must be used.

1.3 Main Contributions and Organization

This dissertation extends state-of-the-art techniques on managing physical and resource constrains in real-time systems in several directions.

(1). We present an efficient online technique to solve the temporal processing overload problem for a more general set of real-time tasks (Chapter 2). Prior

work assumes task deadlines are fixed and independent of task periods. However, in many control applications, the relationship between task periods and task deadlines is tightly coupled. We show in this chapter that by taking this period-deadline relationship into account, at least 74% of the task sets previously thought to be infeasible are in fact schedulable. An immediate consequence of this work is that fewer resources are needed to handle temporal processing overloads.

(2). We propose an energy-aware adaptive framework to address network overloads (Chapter 3). Existing work either do not consider job scheduling when performing packet scheduling or vice versa. As a result, packets that are meant to be transmitted may in fact miss their deadlines, degrading performance. In this chapter, we show that using an integrated, importance-based job and packet scheduling framework, system performance doubles while processing energy consumption is significantly reduced.

(3). We provide an optimal DVFS speed selection policy to maximize the work completed for temperature-constrained soft real-time systems (Chapter 4). The proposed policy is practical in that it does not assume that (1) the system can continuously adjust speed, and (2) a specific set of speed levels is available. As a consequence, our policy can be easily deployed and is shown to improve performance by about 47.7% on average and up to about 68%.

(4). We present both optimal and heuristic methods to minimize the peak temperature of a multicore system running hard real-time applications (Chapter 5). The optimal method reveals that an energy-minimal system is not necessarily a cool system. Specifically, minimizing energy can lead to a system that is about 9 °C hotter on average and up to about 25 °C when compared to

systems in which the maximum temperature is minimized. This is because energy minimization ignores both temporal and spatial thermal variations. To solve large problem instances, we propose an efficient heuristic framework that allows system designers to trade off accuracy against running time and insert idle times into a schedule to further cool the system down.

The dissertation concludes in Chapter 6 with some recommendations on future research directions.

CHAPTER 2

PERIOD AND DEADLINE SELECTION FOR SCHEDULABILITY IN REAL-TIME SYSTEMS

To alleviate occasional overload conditions in real-time control systems, some control tasks are executed less often to reduce computational demands on the processor. Since control tasks are usually activated periodically, their periods can be used as a tuning knob to determine how often these tasks should execute without compromising control system stability. Existing frameworks assume that only task periods are adjustable and that task deadlines remain unchanged at all times. This chapter formally introduces a more general real-time task model where task deadlines are functions of task periods. This tight coupling between task deadlines and task periods has been discussed in a recent work in control systems and presents a novel real-time scheduling challenge.

Because of the aforementioned task period and deadline relationship, an overload management mechanism must now not only determine suitable task periods, but also consider the impact of the newly changed task deadlines on system schedulability. To solve this problem, which we shall refer to as the period and deadline selection problem, this chapter identifies a feasible period-deadline combination and proposes a heuristic, which iteratively adjusts task periods and deadlines in such a way that the task set becomes schedulable. Experimental results show that

the heuristic finds a solution to the period and deadline selection problem over 73% of the time, using less than three search iterations.

2.1 Introduction

In this section, we provide an overview of the problem, review related work, state our contributions, and present the organization of this chapter.

2.1.1 Problem Overview

Task scheduling has long been an important research topic in real-time systems. As stated in Chapter 1, missing a deadline in a hard real-time system may lead to catastrophic consequences, such as failure to stop an automatically controlled train on time [67].

Despite having been traditionally treated as hard real-time systems, many control systems are quite robust in the presence of certain timing perturbations. Generally speaking, depending on the system state, the sampling rate of a control system can vary within some interval without causing significant performance degradation. This observation is very useful when temporal overload situations occur. A real-time system is said to experience an overload when it cannot finish executing one or more tasks on time due to resource constraints. Although robust, if too many deadlines have been missed or if such misses occur in a highly unpredictable manner, a control system may no longer stabilize, even if all system resources are now dedicated to it.

In this chapter, we focus on alleviating temporal overload conditions in real-time control systems by increasing some (or all) task periods in such a way that all task deadlines can be met. As stated previously, we exploit the knowledge

that control tasks are quite robust, which allows their periods to vary within some intervals. As an added challenge, whenever a task period changes, its deadline also varies according to some prespecified function. In other words, task deadlines, which are less than or equal to task periods, are also a function of the periods. This interdependency between task periods and task deadlines is not merely an academic exercise; the work in [108] shows that such a relationship is indeed real (we will explain this novel task model later on in this chapter). In summary, we are interested in solving the period and deadline selection problem for real-time systems under overload conditions.

2.1.2 Related Work

There are two main approaches to dealing with overloads in real-time systems: (i) dropping some instances of tasks (i.e., jobs) in a controlled manner, and (ii) increasing task periods, equivalently decreasing the sampling rates, in such a way that no deadlines are missed and the performance of the system remains acceptable.

Many algorithms have been proposed to control job dropping patterns. Some examples are the (m, k) scheduling algorithms [44], the Dynamic Window-Constrained Scheduling (DWCS) algorithm [109], the skip-over algorithms [54], and the algorithms for weakly hard real-time systems [11]. In other work such as the imprecise computation model [29] and reward-based model [6], the aim is to maximize system workload, which is assumed to be proportional to the quality of service (QoS).

Since it is sometimes more suitable to execute jobs less often instead of dropping them or allocating fewer cycles [4], we focus on such an approach in this chapter. Many previous research papers can be found on the management of overloads

in real-time systems based on task period adjustments (e.g., [58]). The works in [94], [93] and [12] solve the period selection problem for the earliest-deadline first (EDF), rate-monotone (RM), and fixed-priority scheduling algorithms, respectively. Cervin et al. propose an online period adjustment mechanism with varying task computation times [55]. In [23], Caccamo et al. consider scenarios where the worst-case task execution times can be large but the normal task execution times tend to be very small. To efficiently use system resources while avoiding overruns, the idea of task rate adaptations is combined with the use of a constant bandwidth server to guarantee hard real-time deadlines. Buttazzo et al. propose an optimal period selection algorithm in [20] based on the elastic task model. Many extensions to the elastic task model can be found in [17, 18, 21, 22].

In terms of schedulability tests for task sets with deadlines less than periods, Baruah et al. proposed an exact test with pseudo-polynomial running time [10]. For efficiency, we will use the sufficient test provided in [24, 25]. However, there exist other sufficient conditions for schedulability when task deadlines are less than task periods. For instance, Devi proposed a set of sufficient schedulability tests in [34]. The main difference between this set of tests and the one in [24] is that the former requires N checks while the latter requires only one check. Some extensions to Devi's work include, but are not limited to, an approximate schedulability test [1], an adaptation to fixed-priority systems [38], and novel feasibility tests that are shown to outperform Devi's schedulability conditions [71].

Most previous works on overload management assume that only task periods can change. In [95], task deadlines vary with time, but the tasks do not have periods (i.e., tasks are non-periodic). There has also been work on determining the lower bound on task deadlines using sensitivity analysis in a periodic task

model [7]. However, to the best of our knowledge, there has been no work that allows task periods and deadlines to change simultaneously.

2.1.3 Contributions

Our first main contribution is the introduction of a more general and realistic task model where both task deadlines and task periods can vary within some intervals. The deadline of a task in a real-time system really denotes the maximum allowable delay that task can tolerate. As shown by the authors in [108], different sampling rates for a control system lead to different acceptable maximum delays (deadlines). Specifically, a higher sampling rate means that the corresponding control task executes more often, which, in turns, allows the system to be more tolerant to a relatively larger delay. Conversely, a larger sampling period could make the system more susceptible to delays and thus a relatively smaller deadline may be required. In other words, the deadline of a task is a function of its period.

The relationship between task periods and task deadlines poses an interesting scheduling problem, as one can no longer assume that increasing task periods will always improve schedulability. Although it is possible to set task deadlines to be the smallest deadlines (specified by the applications) and only vary task periods, doing so may significantly worsen schedulability. As our second main contribution, we study some interesting relationships between task periods and task deadlines that will help to solve the period and deadline selection problem. We then propose an efficient heuristic that can be used to find a set of feasible task periods and deadlines and alleviate an overload situation in a timely manner. Our heuristic can be applied to any real-time task set where task deadlines are less than or equal to task periods and where task deadlines are piecewise first-order

differentiable functions of their respective periods. Experimental results indicate that our heuristic finds a solution to the period and deadline selection problem over 73% of the time.

2.1.4 Organization

We introduce the system model and formally define the problem in Section 2.2. Section 2.3 provides a motivating example to highlight the importance and usefulness of our work. We present our formal analysis and heuristic in Sections 2.4. Section 2.5 summarizes some experimental results and the chapter concludes with Section 2.6.

2.2 System Model and Problem Definition

In this section, we describe the system model and review some relevant schedulability tests. We also give a formal definition of the period and deadline selection problem.

2.2.1 Task Model

Our system consists of a set of N periodic, synchronous tasks specified by the following 5-tuple: $(C_i, T_i, T_{i_{min}}, T_{i_{max}}, D_i)$, $i = 1, \ldots, N$, where C_i is the worst-case execution time of task τ_i, and T_i is τ_i's period (to be decided), which must lie somewhere between $T_{i_{min}}$ and $T_{i_{max}}$. The parameter $T_{i_{min}}$ denotes the most desirable period of τ_i, as specified by the application, whereas $T_{i_{max}}$ represents the maximum period beyond which the system performance is no longer acceptable. The parameter D_i is the deadline of τ_i, and is dependent on the actual task period T_i. That is, the deadline of a task is a function of its period. Specifically,

$D_i \leq T_i$, $T_i \in [T_{i_{min}}, T_{i_{max}}]$ and D_i is some function that is piecewise first-order differentiable.

The utilization of each task τ_i is defined as $U_i = C_i/T_i$ and denotes system resources dedicated to τ_i. Since the period of τ_i, $i = 1, \ldots, N$, can vary between $T_{i_{min}}$ and $T_{i_{max}}$, the minimum utilization of τ_i, $U_{i_{min}} = C_i/T_{i_{max}}$, and its maximum (desired) utilization, $U_{i_{max}} = C_i/T_{i_{min}}$, are also defined, for $i = 1, \ldots, N$.

2.2.2 Schedulability Tests

Throughout this chapter, we will assume that the Earliest Deadline First (EDF) scheduling algorithm [66] is used. When one or more tasks need to decrease their period and/or deadline in response to either internal (e.g., change in sampling rate of one or more tasks in the system) or external (e.g., network traffic) factors, a schedulability test must be performed to assess whether the task set is still schedulable. A schedulability test may also provide some guidance on how to adjust task parameters in such a way that a feasible task set can be obtained. Based on the assumption that the EDF scheduling algorithm is used, there exist some useful schedulability conditions that are briefly reviewed here.

A necessary condition for schedulability of any given task set is stated in the following lemma.

Lemma 2.1 [24] *Consider a task set Γ, let C_i and D_i be the execution time and the deadline of task τ_i, $i = 1, \ldots, N$, respectively. In addition, let all tasks start at time 0 and let the tasks in Γ be ordered in a non-decreasing order of deadlines. Regardless of the choices of periods, any task set that is schedulable must satisfy the following property*

$$\sum_{i=1}^{j} C_i \leq D_j, j = 1, \ldots, N. \tag{2.1}$$

Since task deadlines can be less than or equal to periods, there exist an exact, albeit complex, schedulability test for EDF as specified by Baruah et al [10]. Said test is restated in the following theorem.

Theorem 2.1 [10] *Consider a periodic task set with C_i, D_i, and T_i as the execution time, deadline, and period of task τ_i, $i = 1, \ldots, N$, respectively. Let $D_i \leq T_i$, $i = 1, \ldots, N$, the task set is schedulable if and only if the following constraint is satisfied $\forall L \in \{kT_i + D_i \leq \min(B_p, H)\}$ and $k \in \mathcal{N}$ (the set of natural numbers including 0), where B_p and H denote the busy period and hyperperiod as defined in [16], respectively,*

$$L \geq \sum_{i=1}^{N} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i. \tag{2.2}$$

Verifying that (2.2) is satisfied for all L is the main source of complexity in the above schedulability test. To reduce the complexity of the test in Theorem 2.1, the authors in [24] proposed the following sufficient condition for schedulability.

Theorem 2.2 [24] *Given a set Γ of N tasks that satisfy Lemma 2.1. Let C_i, D_i, and T_i be the execution time, deadline, and period of task τ_i, $i = 1, \ldots, N$, respectively. In addition, let the tasks in Γ be sorted in a non-decreasing order of deadlines. The task set Γ is schedulable if*

$$L^* \geq \sum_{i=1}^{N} \left(\frac{L^* - D_i}{T_i} + 1 \right) C_i \tag{2.3}$$

where

$$L^* = \begin{cases} D_2 & : \quad D_1 + T_1 \leq D_2 \\ \min_{i=1}^{N} (T_i + D_i) & : \quad \text{otherwise.} \end{cases}$$

For completeness, we include another existing sufficient condition for EDF schedulability.

Theorem 2.3 **[67]** *Consider a set* Γ *of* N *tasks where* C_i *and* D_i *are the execution time and deadline of task* τ_i, $i = 1, \ldots, N$, *respectively. The task set* Γ *is schedulable by the EDF policy if*

$$\sum_{i=1}^{N} \frac{C_i}{D_i} \leq 1. \tag{2.4}$$

We will use some of these schedulability conditions in Section 2.4.

2.2.3 Problem Definition

Given an initially infeasible set Γ of N real-time tasks where the period T_i of task τ_i must lie somewhere between $[T_{i_{min}}, T_{i_{max}}]$, and the deadline D_i of τ_i is some piecewise first-order differentiable function of its period, determine a period-deadline combination (T_i, D_i), $i = 1, \ldots, N$, such that the task set Γ becomes schedulable. In other words, we wish to find (T_i, D_i), $i = 1, \ldots, N$, such that

$$\sum_{i=1}^{N} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i \leq L \tag{2.5}$$

$$T_i \geq T_{i_{min}} \quad \text{for } i = 1, 2, \cdots, N \tag{2.6}$$

$$T_i \leq T_{i_{max}} \quad \text{for } i = 1, 2, \cdots, N, \tag{2.7}$$

where L is defined as in Theorem 2.1, C_i is the worst-case execution time of τ_i, and both $T_{i_{min}}$ and $T_{i_{max}}$ are specified by the applications under consideration.

The constraint in (2.5) ensures the schedulability of the task set. The constraints in (2.6) and (2.7) bound the period of τ_i, $i = 1, \ldots, N$, to ensure performance.

2.3 Motivations

In control systems, an advantage in using the traditional periodic task model where task deadlines are fixed is that these systems can be treated as discrete-time systems for which there exists a variety of mature controller synthesis methods. However, when the periodic task model is used, task periods and deadlines are often chosen conservatively to guarantee stability. This leads to wasted resources and system over-provisioning. For these reasons, there has been a recent movement in the control system community to investigate alternative approaches to the periodic task model.

The work in [108] is such an example. Each task determines its next release time based on the current system state as sampled by the current job. This type of control systems is known as state-based self-triggering systems. Self-triggering can be viewed as a closed-loop form of releasing tasks for execution, whereas the traditional periodic task model is considered open-loop. Since each control task is aware of its system state, it can adjust its period and deadline in such a way that only the required system resource is requested. More precisely, with a small period, a task is executed relatively often and the system is thus more tolerant to delays, permitting the task deadline to be relatively larger (e.g., perhaps almost as large as the task period itself). On the other hand, when the task period is large, the system is more susceptible to disturbances, requiring that the task deadline be relatively smaller (compared to the task period) to reduce jitters.

To understand how the deadline as a function of the period affects schedulability, let us consider a simple task set, which consists of two identical tasks whose attributes are shown in Table 2.1. The deadline of each task can be computed as shown in the last column of Table 2.1 (all units are in milliseconds). Figure 2.1

plots the task deadlines as a function of task periods where the vertical dotted lines limit the acceptable period range for the example tasks. Initially, the task set is not schedulable with $T_1 = T_2 = 0.5\,\text{ms}$, since the initial deadlines $D_1 = D_2 = 0.303\,\text{ms}$ and the aggregate execution time required is $0.36\,\text{ms}$. If we simply set $T_1 = T_2 = 3.5\,\text{ms}$, which is the maximum allowable periods, then the corresponding deadlines will be $D_1 = D_2 = 0.106\,\text{ms}$. The task set is, again, not schedulable and one may wrongly conclude that the task set cannot be made feasible. However, there exists many feasible period-deadline combinations. For example, when $T_1 = T_2 = 1\,\text{ms}$ and $D_1 = D_2 = 0.368\,\text{ms}$, the task set is schedulable.

TABLE 2.1

TASK SET FOR MOTIVATING EXAMPLE ILLUSTRATING THE IMPORTANCE OF THE PERIOD AND DEADLINE SELECTION PROBLEM

Task	C_i	$T_{i_{min}}$	$T_{i_{max}}$	D_i
τ_1	0.18	0.5	3.5	$T_1 e^{-T_1}$, $T_1 \in [0.5, 3.5]$
τ_2	0.18	0.5	3.5	$T_2 e^{-T_2}$, $T_2 \in [0.5, 3.5]$

In the traditional periodic task model, since task deadlines are considered fixed, system designers must use the smallest possible deadlines to ensure that, given a specific range of task periods, the system will always meet the minimum

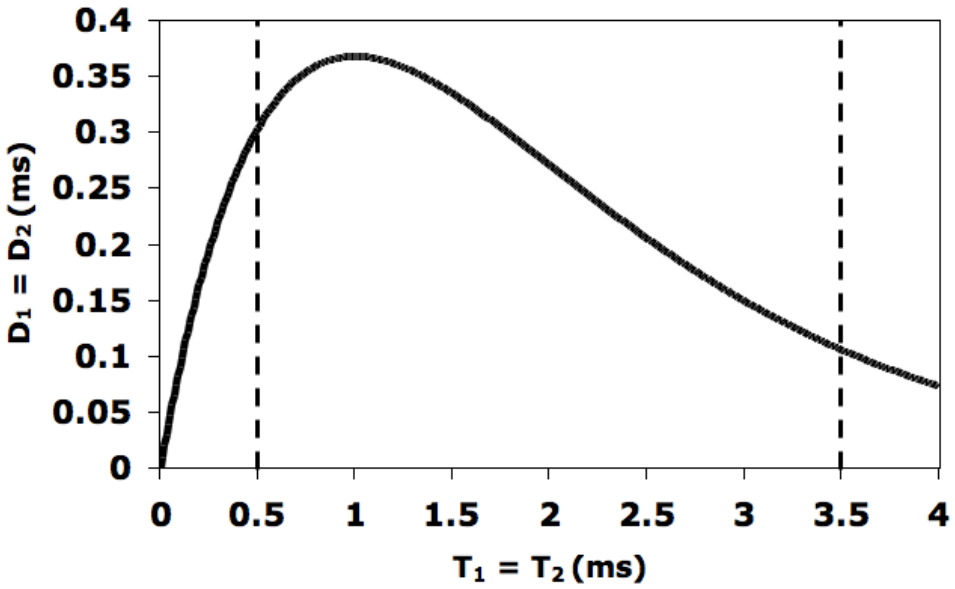

Figure 2.1. Task deadlines in milliseconds as a function of task periods, also in milliseconds. The area inside the vertical dotted lines denotes the acceptable period ranges for the task set in Table 2.1.

performance requirements. For the above example, the smallest deadline for both tasks is 0.106 ms, which means that the task set can never be made schedulable using existing techniques. It is not difficult to see in this example that the task deadlines can be set to 0.36 ms for the task set to be feasible, regardless of the resultant periods. In general, however, both task periods and task deadlines must be considered simultaneously, since different tasks may have different timing requirements.

2.4 Period and Deadline Selection Heuristic

As shown in the previous section, since a task deadline is a function of its period, adjusting the period affects both the corresponding deadline and the schedulability of the entire task set. Due to the condition in (2.5), the problem defined in Section 2.2.3 is nonlinear, non-convex, and non-continuous because of the variable L. Solving the above problem directly using a nonlinear solver is inefficient and the solver cannot guarantee that a solution will be found, even if one exists. (Nonlinear solvers usually employ interior-point methods [39] or branch-and-bound techniques [13, 30] to solve the problem.) For these reasons, we propose using an efficient heuristic to find a solution. In a nutshell, the heuristic starts by performing some simple schedulability tests to determine a feasible period-deadline combination. Such tests also serve to eliminate some infeasible period and deadline values should they fail to identify a feasible task set. The heuristic then uses this knowledge to conduct an efficient search process.

2.4.1 Identifying Infeasible Regions Using Simple Tests

We now describe our idea of using the simple tests in more detail. We first determine the minimum and maximum deadlines, $D_{i_{min}}$ and $D_{i_{max}}$, respectively, for each task τ_i, $i = 1, \ldots, N$. The maximum deadline of τ_i, $D_{i_{max}}$ can directly be solved by finding the maximum of D_i. (Recall that the maximum of a function can be obtained by taking its derivative and subsequently finding the root(s) of said derivative.) The corresponding period value is denoted T_i^{Dmax}, $i = 1, \ldots, N$.

To determine the lower bound on the deadline of a task τ_i, $i = 1, \ldots, N$, we would ideally use Lemma 2.1. However, Lemma 2.1 requires that tasks be sorted in a non-decreasing order of deadlines. Since a task deadline is a variable

to be determined, we cannot directly use Lemma 2.1 to compute the minimum deadline. Instead, let $\widetilde{D}_i$ be the smallest deadline of task τ_i, i.e., $\widetilde{D}_i \leq D(T_i)$, $T_i \in [T_{i_{min}}, T_{i_{max}}]$, $i = 1, \ldots, N$. We say that task τ_i dominates task τ_j (denoted by $\tau_i \succeq \tau_j$) if $\widetilde{D}_i > D_{j_{max}}$. Otherwise, we say that τ_i and τ_j are non-comparable. Using the above dominance definition, a partial order can be built for a given set of tasks. It is easy to see that Lemma 2.1 holds true for tasks with deadlines as variables if we sort the tasks using the partial order established above. For example, consider a simple task set consisting of task τ_j and τ_k. If $\tau_j \succeq \tau_k$ then $D_{k_{min}} = C_k$ and $D_{j_{min}} = C_k + C_j$. In general, for a task τ_i, $D_{i_{min}} = \max\{\mathbf{DS}\} + C_i$, where **DS** is the set of deadlines of tasks that are dominated by τ_i. Since $D_{i_{min}}$ set in this way is a lower bound on the minimum task deadline for task τ_i, $i = 1, \ldots, N$, we can eliminate some infeasible period-deadline combinations (shown by the right-slanted pattern in Figure 2.2). The task period that corresponds to when the task deadline is $D_{i_{min}}$ is referred to as T_i^{Dmin}, $i = 1, \ldots, N$. Clearly, if $D_{i_{min}}$, $i = 1, \ldots, N$, satisfy the condition in Lemma 2.1, but if $D_{i_{max}} < D_{i_{min}}$ for some $i = 1, \ldots, N$, then the task set cannot be made schedulable.

Once we have found the minimum and maximum deadlines for each task in the task set, we can apply a series of efficient schedulability tests to avoid searching for a solution, if possible. We start with the sufficient condition from Theorem 2.3 using $D_{i_{max}}$, $i = 1, \ldots, N$, as the task deadlines. The following lemma helps to explain why only $D_{i_{max}}$, $i = 1, \ldots, N$, need to be considered when applying Theorem 2.3 on the current task set.

Lemma 2.2 *Consider a set Γ of N tasks. Let C_i and D_i be the execution time and deadline of task τ_i, $i = 1, \ldots, N$, respectively. If the schedulability condition from Theorem 2.3 is not satisfied for $D_{i_{max}}$, $i = 1, \ldots, N$, then it is not satisfied*

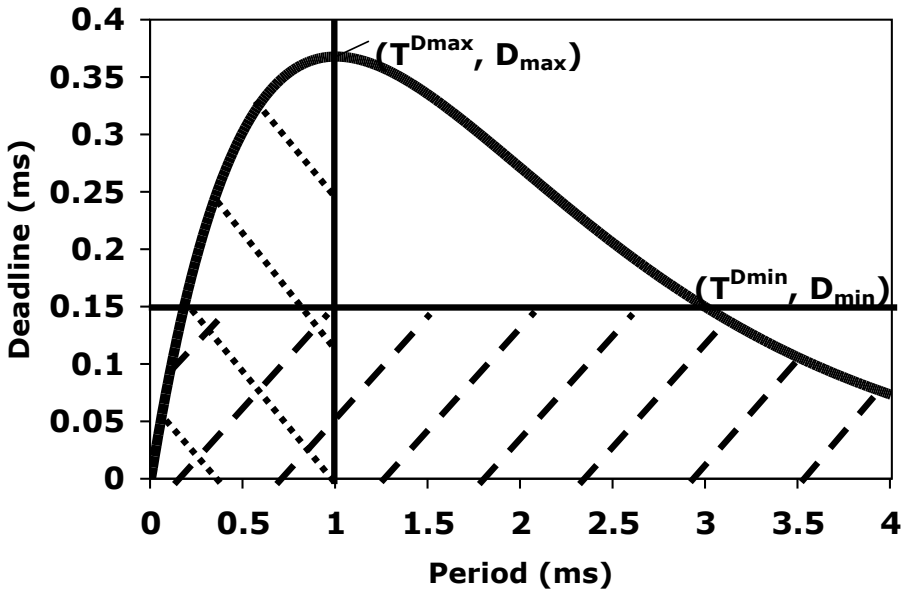

Figure 2.2. Task deadline in milliseconds as a function of task periods, also in milliseconds. The left-slanted and right-slanted areas denote infeasible period and deadline combinations. The unpatterned region represents possibly feasible period and deadline combinations and is where the search process is conducted.

for any $D_i < D_{i_{max}}$, $i = 1, \ldots, N$.

Proof: If the task set Γ fails the schedulability test in Theorem 2.3, then

$$\sum_{i=1}^{N} \frac{C_i}{D_{i_{max}}} > 1. \tag{2.8}$$

Using any $D_i < D_{i_{max}}$, $i = 1, \ldots, N$, would yield

$$\sum_{i=1}^{N} \frac{C_i}{D_i} > \sum_{i=1}^{N} \frac{C_i}{D_{i_{max}}} > 1. \tag{2.9}$$

Therefore, the lemma holds.

□

Note that if the condition from Theorem 2.3 is satisfied for $D_{i_{max}}$, $i = 1, \ldots, N$, then we have identified a feasible solution. Otherwise, we apply the schedulability test from Theorem 2.2 for a special point $(T_i^{Dmax}, D_{i_{max}})$, $i = 1, \ldots, N$ (see Figure 2.2). (To use Theorem 2.2, we order the tasks in a non-decreasing order of deadlines using $D_i = D_{i_{max}}$ and $T_i = T_i^{Dmax}$, $i = 1, \ldots, N$, whenever L needs to be determined). We choose to test the point $(T_i^{Dmax}, D_{i_{max}})$, $i = 1, \ldots, N$, because if the task set is not schedulable at this point according to Theorem 2.2, then it is not schedulable for any $(\overline{T}_i, \overline{D}_i)$, $\overline{T}_i \leq T_i^{Dmax}$ and $\overline{D}_i \leq D_{i_{max}}$, $i = 1, \ldots, N$. The following theorem proves this claim and explains why the left-slanted region in Figure 2.2 can be eliminated from further consideration if the point $(T_i^{Dmax}, D_{i_{max}})$, $i = 1, \ldots, N$, is found to be unschedulable according to Theorem 2.2.

Lemma 2.3 *Consider a set Γ of N tasks. Let C_i and D_i be the execution time and deadline of task τ_i, $i = 1, \ldots, N$, respectively. Let T_i be the period obtained when $D_i = D_{i_{max}}$, $i = 1, \ldots, N$. If the condition in Theorem 2.2 is not satisfied for (T_i, D_i), $i = 1, \ldots, N$, then it is not satisfied for any $(\overline{T}_i, \overline{D}_i)$, $\overline{D}_i \leq D_i$, $\overline{T}_i \leq T_i$, $i = 1, \ldots, N$.*

Proof: Since the task set is not schedulable at (T_i, D_i), $i = 1, \ldots, N$, we have

$$L < \sum_{i=1}^{N} \left(\frac{L - D_i}{T_i} + 1 \right) C_i. \tag{2.10}$$

In addition, since $\overline{D}_i \leq D_i$ and $\overline{T}_i \leq T_i$, $i = 1, \ldots, N$,

$$\begin{aligned} L &< \sum_{i=1}^{N} \left(\frac{L - D_i}{T_i} + 1 \right) C_i \\ &< \sum_{i=1}^{N} \left(\frac{L - \overline{D}_i}{\overline{T}_i} + 1 \right) C_i, \end{aligned} \tag{2.11}$$

since $L - \overline{D}_i > L - D_i$ and $\overline{T}_i < T_i$, $i = 1, \ldots, N$. Finally, as $L > \overline{L}$, where $\overline{L} = \overline{D}_2$ if $\overline{D}_1 + \overline{T}_1 \leq \overline{D}_2$ and $\overline{L} = \min_{i=1}^{N}(\overline{T}_i + \overline{D}_i)$ otherwise,

$$\overline{L} < \sum_{i=1}^{N} \left(\frac{L - D_i}{T_i} + 1 \right) C_i. \tag{2.12}$$

□

Observe that we use the schedulability conditions in Theorems 2.2 and 2.3 in conjunction to one another. This is because a task set that is feasible according to one of the aforementioned schedulability conditions is not necessarily feasible according to the other (and vice versa).

The left-slanted region in Figure 2.2 is a result of Lemma 2.3 and can be eliminated from further consideration. (Note that since we test the point $(T_i^{Dmax}, D_{i_{max}})$ using the schedulability test from Theorem 2.2, which presents only a sufficient condition, such a point may in fact be feasible. However, to exactly determine schedulability, the condition in Theorem 2.1 needs to be satisfied and is often too time consuming to be used during an overload situation.) In any case, the area with no pattern indicates the remaining search region. Note that for a specific period T_i, $i = 1, \ldots, N$, any deadline $0 < \overline{D}_i < D_i$ is also acceptable from the system performance point of view. However, since using $\overline{D}_i$ will worsen schedulability, we only consider D_i. All the simple tests described thus far appear as part

of our heuristic and are shown in Algorithm 1. Lines 1–3 show the first simple test discussed in Lemma 2.2. The second simple test from Lemma 2.3 is shown in Lines 4–7. Finally, Lines 8–11 show that we perform an additional schedulability test for $(T_i^{Dmin}, D_{i_{min}})$, $i = 1, \ldots, N$, since this point has already been computed and such an additional test does not incur a significant amount of additional overhead.

Algorithm 1 SimpleTests(Γ)

1: $result \leftarrow \sum_{i=1}^{N} \frac{C_i}{D_{i_{max}}}$
2: **if** $result \leq 1$ **then**
3: **return** $\left[D_{i_{max}}, T_i^{Dmax}\right]$, for $i = 1, \ldots, N$
4: compute L as in (2.14) using $D_{i_{max}}$ and T_i^{Dmax}, $1 \leq i \leq N$
5: $result \leftarrow \sum_{i=1}^{N} \left(\frac{L - D_{i_{max}}}{T_i^{Dmax}} + 1\right) \cdot C_i$
6: **if** $result \leq L$ **then**
7: **return** $\left[D_{i_{max}}, T_i^{Dmax}\right]$, for $i = 1, \ldots, N$
8: compute L as in (2.14) using $D_{i_{min}}$ and T_i^{Dmin}, $1 \leq i \leq N$
9: $result \leftarrow \sum_{i=1}^{N} \left(\frac{L - D_{i_{min}}}{T_i^{Dmin}} + 1\right) \cdot C_i$
10: **if** $result \leq L$ **then**
11: **return** $\left[D_{i_{min}}, T_i^{Dmin}\right]$, for $i = 1, \ldots, N$
12: **return** $\emptyset$

2.4.2 Efficiently Conducting the Search Process

If all the aforementioned simple tests fail, we will have to search along the unpatterned region of Figure 2.2 to find a feasible period-deadline combination, (T_i, D_i), $i = 1, \ldots, N$. Since the main source of complexity of the problem defined in Section 2.2.3 is that (2.5) must be satisfied for all possible values of L, the

search process will instead use the schedulability test from Theorem 2.2. In other words, the problem in Section 2.2.3 is modified to

$$\sum_{i=1}^{N} \left(\frac{L - D_i}{T_i} \right) C_i \leq L - \sum_{i=1}^{N} C_i \tag{2.13}$$

$$L = \begin{cases} D_2 & : \ D_1 + T_1 \leq D_2 \\ \min\left(T_i + D_i\right) & : \ \text{otherwise} \end{cases} \tag{2.14}$$

$$T_i \leq \min\left\{T_{i_{max}}, T_i^{Dmin}\right\}, \ i = 1, 2, \cdots, N \tag{2.15}$$

$$T_i \geq \max\left\{T_{i_{min}}, T_i^{Dmax}\right\}, \ i = 1, 2, \cdots, N, \tag{2.16}$$

where T_i^{Dmin} and T_i^{Dmax}, $i = 1, \ldots, N$ are as defined previously.

Given an initially infeasible task set, one can compute the corresponding value of L as in (2.14). Let us first assume that the value of L is fixed once it has been computed. To satisfy the condition in (2.13), observe that since the right-hand side of (2.13) can be treated as a constant, one way to solve the above problem is to adjust task periods and deadlines such that the left-hand side becomes as small as possible. We can express this idea mathematically as the following constrained optimization problem.

$$\min: \quad \sum_{i=1}^{N} \left(L - \widehat{D}_i \right) \cdot U_i \tag{2.17}$$

$$\text{s.t.}: \quad U_i \leq U_{i_{max}} \quad \text{for } i = 1, 2, \cdots, N \tag{2.18}$$

$$U_i \geq U_{i_{min}} \quad \text{for } i = 1, 2, \cdots, N, \tag{2.19}$$

where $\widehat{D}_i$ is the task deadline function that depends on U_i, i.e., $\widehat{D}_i \equiv \widehat{D}_i(U_i) = D_i(C_i/U_i)$. (For notational simplicity, D_i always refers to $D_i(T_i)$ and $\widehat{D}_i$ to $D_i(U_i)$.) $U_i = C_i/T_i$, $U_{i_{max}} = \frac{C_i}{\max\left\{T_{i_{min}}, T_i^{Dmax}\right\}}$, and $U_{i_{min}} = \frac{C_i}{\min\left\{T_{i_{max}}, T_i^{Dmin}\right\}}$.

Solving the above constrained optimization problem is attractive because if a solution to the problem in (2.13)–(2.16) exists for a fixed value of L, then we will find it by solving the above constrained optimization problem. This claim is formally stated in the following lemma.

Lemma 2.4 *Consider an initially infeasible task set Γ where C_i, U_i, and $\widehat{D}_i$ denote the execution time, utilization, and deadline (as a function of the utilization) of task τ_i, $i = 1, \ldots, N$, respectively. For a fixed value of L, if there exists a solution to the problem in (2.13)–(2.16), it will be found by solving the problem defined in (2.17)–(2.19).*

Proof: The lemma can be trivially proved by observing that (2.13) can be rewritten as

$$\sum_{i=1}^{N} \left(L - \widehat{D}_i \right) \cdot U_i \leq L - \sum_{i=1}^{N} C_i. \tag{2.20}$$

The constrained optimization problem in (2.13)–(2.16) minimizes the left-hand side of the above equation. Thus, if we can adjust task periods and deadlines such that (2.20) is true, then the solution to the optimization problem in (2.17)–(2.19) will also be a solution to the problem in (2.13)–(2.16).

□

The following theorem presents a globally optimal solution to the problem in (2.17)–(2.19) and hence a solution to the problem in (2.13)–(2.16), for a fixed value of L.

Theorem 2.4 *Consider the constrained optimization problem as specified in (2.17)–(2.19). Let U_i be the utilization of task τ_i, $i = 1, \ldots, N$. Let $\widehat{D}_i$ be the deadline of*

τ_i where $\widehat{D}_i$ is a function of the U_i, i.e., $\widehat{D}_i \equiv \widehat{D}_i(U_i) = D_i(C_i/U_i)$, and let

$$G_i(U_i) = (L - \widehat{D}_i) \cdot U_i. \tag{2.21}$$

For a fixed value of L, the solution, U_i, is optimal if and only if

$$U_i = \text{argmin}_{U_i \in \left\{ \overline{\mathbf{U}}_i \cup U_i^{min} \cup U_i^{max} \right\}} \{G_i(U_i)\}, \tag{2.22}$$

where $\overline{\mathbf{U}}_i$ is a set of $\overline{U}_i$ such that $L - \widehat{D}_i - \widehat{D}_i' \cdot \overline{U}_i = 0$.

Proof: We prove that if U_i, $i = 1, \ldots, N$, is an optimal solution to the constrained optimization problem in (2.17)–(2.19), then (2.22) must be true by utilizing the Kuhn-Tucker (KKT) necessary conditions for optimality for constrained optimization problem, which can be written in terms of the Lagrangian function for the problem as

$$J_a(\mathbf{U}, \mu) = \sum_{i=1}^{N} (L - \widehat{D}_i) \cdot U_i + \sum_{i=1}^{N} \mu_i (U_i^{min} - U_i) + \sum_{i=1}^{N} \lambda_i (U_i - U_i^{max}) \tag{2.23}$$

where μ_i's and λ_i's are Lagrange multipliers, $\mu_i \geq 0$ and $\lambda_i \geq 0$, $i = 1, \ldots, N$. The necessary conditions for the existence of a relative minimum at U_i are, for $i = 1, \ldots, N$,

$$0 = L - \widehat{D}_i' \cdot U_i - \widehat{D}_i - \mu_i + \lambda_i \tag{2.24}$$

$$0 = \mu_i (U_i^{min} - U_i) \tag{2.25}$$

$$0 = \lambda_i (U_i - U_i^{max}) \tag{2.26}$$

From (2.24)

$$L - \widehat{D}_i' \cdot U_i - \widehat{D}_i = \mu_i - \lambda_i \tag{2.27}$$

If $L - \widehat{D}_i' \cdot U_i - \widehat{D}_i < 0$ for $U_i \in [U_i^{min}, U_i^{max}]$, then μ_i must be 0 and $\lambda_i > 0$. Hence, $U_i = U_i^{max}$. If $L - \widehat{D}_i' \cdot U_i - \widehat{D}_i > 0$ for $U_i \in [U_i^{min}, U_i^{max}]$ then $\lambda_i = 0$ and $\mu_i > 0$. Therefore, $U_i = U_i^{min}$. Otherwise, $L - \widehat{D}_i' \cdot U_i - \widehat{D}_i = 0$ at least once when $U_i \in [U_i^{min}, U_i^{max}]$. In such a case, we can find the value(s) of U_i by finding all the extreme points in the interval $[U_i^{min}, U_i^{max}]$, which is equivalent to solving the equation $L - \widehat{D}_i - \widehat{D}_i' \cdot U_i = 0$ for U_i. Note that since the KKT conditions are necessary for optimality, we have completed the proof for this part.

Now, we prove that if U_i, $i = 1, \ldots, N$, is determined as in (2.22), then it is an optimal solution to the constrained optimization problem in (2.17)–(2.19). We start by observing that, given a piecewise differentiable function $G_i(U_i)$, the global minimum of $G_i(U_i)$ in the interval $[U_i^{min}, U_i^{max}]$ must either be at one of the extreme points inside $[U_i^{min}, U_i^{max}]$ or at the boundaries, i.e., at U_i^{min} or U_i^{max}. This is indeed captured by the expression in (2.22).

Finally, since the objective function in (2.17) can be rewritten as $\min_{i=1}^{N} G_i(U_i)$, minimizing each individual $G_i(U_i)$, $i = 1, \ldots, N$, is equivalent to minimizing (2.17).

□

Remark: If the left-hand side of (2.17) is a convex function, then the KKT necessary conditions for optimality also become sufficient conditions. In such a case, a global optimal solution to the optimization problem in (2.17)–(2.19) for a non-fixed L can be found using Theorem 2.4.

We use the result from the above theorem directly in the main part of our heuristic (Line 22 in Algorithm 2). Although the heuristic can optimally solve the

problem in (2.17)–(2.19) for a fixed value of L, it needs to iteratively search for a feasible task set. This is because the value of L may either increase or decrease as D_i and T_i, $i = 1, \ldots, N$, change. Consider two consecutive iterations h and $h+1$. If the task set with periods $T_i^{(h)}$ and deadlines $D_i^{(h)}$, $i = 1, \ldots, N$, satisfies the constraints in (2.13)–(2.16) given some fixed value of $L^{(h)}$ and $L^{(h+1)} \geq L^{(h)}$, then the task set is guaranteed to be schedulable (as shown in the following lemma) and the search process ends.

Algorithm 2 FindFeasiblePeriodsDeadlines(Γ, *maxIter*)

1: **for** each $\tau_i \in \Gamma$ **do**
2: $D_{i_{max}} \leftarrow \max_{T_i \in [T_{i_{min}}, T_{i_{min}}]} D_i$
3: $T_i^{Dmax} \leftarrow$ period when deadline is $D_{i_{max}}$
4: $D_{i_{min}} \leftarrow C_i$
5: $T_i^{Dmin} \leftarrow$ period when deadline is $D_{i_{min}}$
6: **result** $\leftarrow$ `SimpleTests`(Γ)
7: **if result** $\neq \emptyset$ **then**
8: **return result**
9: $D_i \leftarrow D_{i_{max}}$, $i = 1, \ldots, N$
10: $T_i \leftarrow T_i^{Dmax}$, $i = 1, \ldots, N$
11: *done* $\leftarrow$ *false*
12: *iterNum* $\leftarrow 0$
13: **while not** *done* **do**
14: *iterNum* $\leftarrow$ *iterNum* $+ 1$
15: compute L as in (2.14) using D_i and T_i, $1 \leq i \leq N$
16: *result* $\leftarrow \sum_{i=1}^{N} \left(\frac{L - D_i}{T_i} + 1 \right) \cdot C_i$
17: **if** *result* $\leq L$ **then**
18: **return** $[D_i, T_i]$, $i = 1, \ldots, N$
19: **if** *iterNum* $>$ *maxIter* **then**
20: *done* $\leftarrow$ *true*
21: **for** each $\tau_i \in \Gamma$ **do**
22: compute U_i as in Theorem 2.4
23: $T_i \leftarrow \frac{C_i}{U_i}$
24: determine D_i accordingly
25: **return** $\emptyset$

Lemma 2.5 *Consider a set Γ of N tasks, and let C_i, T_i, and D_i be the execution time, period, and deadline of task τ_i, $i = 1, \ldots, N$, respectively. If the task set satisfies the condition in Theorem 2.2 for some L, then it also satisfies the condition in Theorem 2.2 for any $\overline{L} \geq L$.*

Proof: We have

$$L \geq \sum_{i=1}^{N} \left(\frac{L - D_i}{T_i} + 1 \right) \cdot C_i \tag{2.28}$$

$$L - \sum_{i=1}^{N} L \cdot \frac{C_i}{T_i} \geq \sum_{i=1}^{N} C_i - \frac{D_i \cdot C_i}{T_i} \tag{2.29}$$

$$L \left(1 - \sum_{i=1}^{N} \frac{C_i}{T_i} \right) \geq \sum_{i=1}^{N} C_i - \frac{D_i \cdot C_i}{T_i} \tag{2.30}$$

which clearly holds for any $\overline{L} \geq L$.

□

Now, if $L^{(h+1)} < L^{(h)}$, the schedulability condition in Theorem 2.2 must explicitly be checked (Lines 16–17 in Algorithm 2). In this way, the heuristic will either return a feasible task set or continue searching until the number of maximum iterations, *maxIter*, has been reached (Lines 19–20). The value *maxIter* is a user-defined constant and, from our experiments in the next section, can be set to some small number such as 100.

The time complexity of our heuristic is dominated by the while loop on Line 13 of Algorithm 2. Inside the while loop, the most time consuming operations appear inside the for-loop on Line 21. Let $|\overline{\mathbf{U}}_i|$ be the maximum size of $\overline{\mathbf{U}}_i$ as defined in Theorem 2.4 over all iterations and let $O(G')$ be the worst-case time complexity required to find all solutions to the equation $L - \widehat{D}_i - \widehat{D}'_i \cdot U_i = 0$,

also from Theorem 2.4. The running time of our heuristic is then $O(maxIter \cdot N \cdot (|\overline{\mathbf{U}}_i| + O(G')))$, where N is the number of tasks in the task set.

Remark: In our approach, we assume that when a task set is infeasible, each task is equally responsible for reducing its processor demand (if possible) to alleviate the overload situation. In practice, however, some tasks may be more important than the others. As a result, a weight may be associated with each task to denote its importance. In such a case, our approach can be extended by factoring in the weight of each task when deciding the amount of processor demand reduction that each task should be responsible for. Specifically, the problem formulation in Section 2.2.3 can be modified to the following constrained optimization problem

$$\min : \quad \sum_{i=1}^{N} w_i (T_{i_{min}} - T_i)^2 \tag{2.31}$$

$$\text{s.t. :} \quad \sum_{i=1}^{N} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i \leq L \tag{2.32}$$

$$T_i \geq T_{i_{min}} \quad \text{for } i = 1, 2, \cdots, N \tag{2.33}$$

$$T_i \leq T_{i_{max}} \quad \text{for } i = 1, 2, \cdots, N, \tag{2.34}$$

where w_i is the weight of the task τ_i, $i = 1, \ldots, N$, and all other parameters retain their meaning as previously defined. Again, the modified problem can be too time consuming (and perhaps too difficult) to solve using a nonlinear solver and thus the use of a heuristic similar to the one presented earlier is recommended.

2.5 Experimental Results

In this section, we explain the experimental setup and discuss the performance of our proposed heuristic.

2.5.1 Experimental Setup

Since directly solving the period and deadline selection problem in Section 2.2.3 using a commercial non-linear solver can be very time consuming and it cannot be guaranteed that a solution will be found, even if one exists, we proposed an efficient heuristic in Section 2.4. In this section, we evaluate the performance of our approach.

Due to the lack of realistic benchmarks suitable for the intended experiment, we randomly generated 80 task sets consisting of 5 tasks each. In order to scrutinize the search aspect of the heuristic, each task set is chosen such that it is initially infeasible with the guarantee that all three simple tests from Algorithm 1 will fail. In addition, given a task set, there exists at least one feasible period-deadline combination, (T_i^*, D_i^*), for each task τ_i, $i = 1, \ldots, N$, using the schedulability test from Theorem 2.2 (and hence satisfies the necessary and sufficient condition from Theorem 2.1).

In our experiment, we use the following deadline function, whose curve is representative of the relationship between task periods and task deadlines of the type of control systems under consideration. (It is worth noting, however, that any deadline function can be used, as long as it is piecewise first-order differentiable.)

$$D_i = \frac{k1_i}{T_i - k2_i}, \tag{2.35}$$

for $i = 1, \ldots, N$, where $k1_i$ and $k2_i$, $i = 1, \ldots, N$, are some constants that depend on the specific task under consideration.

The following steps were taken to generate a task set. First of all, the following parameters were specified: utilization, maximum hyperperiod, minimum period, maximum period, precision, and maximum number of tries. Based on these pa-

rameters, task periods are generated in such a way that the hyperperiod is no larger than the maximum hyperperiod. (This could take a number of tries.) In our experiment, we set the maximum hyperperiod, minimum period, and maximum period to 500,000, 10,000, and 40,000 time units, respectively. The precision was specified to be 100, whereas the maximum number of tries was set to 10,000. The precision denotes the minimum increment in any task period. For example, if the precision is set to 100, a task period could be 5200 time units, but not 5010 time units. Finally, for the task sets used in our experiment, the range for the utilization was between 0.5 and 0.7.

Afterwards, the point $(T_i^{Dmax}, D_{i_{max}})$, $i = 1, \ldots, N$, which denotes the maximum deadline value for task τ_i, is randomly generated. In addition, we ensure that the point $(T_i^{Dmax}, D_{i_{max}})$, $i = 1, \ldots, N$, is not schedulable according to Theorems 2.2 and 2.3. (Recall that the purpose of the experiment is to test the search aspect of the heuristic and therefore we have to ensure that the simple tests fail.) Note that the deadline function in (2.35) is defined only for $T_i \geq T_i^{Dmax}$, $i = 1, \ldots, N$, since according to Lemma 2.3, any task set that is not schedulable for $(T_i^{Dmax}, D_{i_{max}})$, $i = 1, \ldots, N$, will not be schedulable for any (T_i, D_i), $T_i \leq T_i^{Dmax}$, $D_i \leq D_{i_{max}}$, $i = 1, \ldots, N$. In other words, any period $T_i < T_i^{Dmax}$, $i = 1, \ldots, N$, can be ignored by the search process.

Each task is randomly assigned an execution time such that the total utilization equals that specified by the user. No task will have a utilization that is greater than half of the specified total utilization. Then, each task is randomly assigned a deadline D_i that ensures that $\sum_{i=1}^{N} \frac{C_i}{D_i} > 1$. As a final step, the random task set generator tests the schedulability of the task set using the sufficient condition from Theorem 2.2. If the task set is unschedulable, task deadlines are randomly

increased while ensuring that $\sum_{i=1}^{N} \frac{C_i}{D_i}$ is still greater than 1. This final step is repeated until either a feasible task set has been found or the maximum number of tries has been reached.

After the generation of the aforementioned random points, each task set will be associated with two points: $(T_i^{Dmax}, D_{i_{max}})$ and (T_i^*, D_i^*), $i = 1, \ldots, N$, where the former point is not schedulable according to Theorem 2.2, but the latter point is. Using these two points, the constants $k1_i$ and $k2_i$, $i = 1, \ldots, N$, can be found. Finally, the point $(T_i^{Dmin}, D_{i_{min}})$, $i = 1, \ldots, N$ can be determined as described in Section 2.4.

2.5.2 Heuristic Performance

We implemented the heuristic proposed in the last section in C++. The user-defined parameters *maxIter* was set to 100, which means that at most 100 search iterations were conducted for each task set (benchmark). The proposed heuristic found a feasible period-deadline combination for 59 out of the 80 task sets. For these benchmarks, if we were to use existing techniques where task deadlines are fixed (which do not directly apply to the system model under consideration), then no solution will be found for any of these task sets because such techniques assume that if the task set is not schedulable for $(T_i^{Dmin}, D_{i_{min}})$, $i = 1, \ldots, N$, then it cannot be made feasible. (In other words, the schedulability test from Theorem 2.2 is performed for $(T_i^{Dmin}, D_{i_{min}})$, $i = 1, \ldots, N$. This test is referred to as the "fixed deadline technique" in Table 2.2.) Clearly, due to the dependency between task periods and task deadlines, the fixed deadline technique is shown to be too pessimistic. Table 2.2 summarizes the results which show that our heuristic has an overall success rate of over 73% while the fixed deadline technique has a

success rate of 0%. Further, since the left-hand side of (2.17) is a convex function (due to the deadline function used), the solutions found by the heuristic are also optimal solutions to the optimization problem in (2.17)–(2.19).

TABLE 2.2

PERFORMANCE COMPARISON: PERIOD AND DEADLINE SELECTION HEURISTIC VS. FIXED DEADLINE TECHNIQUE

Method	Number of solutions found	% solutions found
Fixed deadline technique	0/80	0%
Our heuristic	59/80	73.8%

Table 2.3 shows the number of iterations needed by the proposed heuristic to find a solution for each task set. As can be seen from the table, the task sets that the heuristic could find a feasible solution to took fewer than 3 search iterations. On the other hand, 100 search iterations were not enough to find a feasible period-deadline combination for 13 task sets.

2.6 Summary

In this chapter, we proposed a more general and realistic real-time task model where each task deadline is a function of the corresponding period. This task

TABLE 2.3

PERIOD AND DEADLINE SELECTION HEURISTIC: NUMBER OF ITERATIONS REQUIRED

Number of task sets	Number of iterations needed
37 (solution found)	< 3
13 (solution not found)	> 100

model facilitates the feasibility analysis of the real-time control systems where task deadlines reflect the maximum allowable delays as tolerated by any given system and vary according to the sampling periods. Since existing techniques cannot adequately be used to determine schedulability for this novel task model, we also proposed a heuristic to identify a schedulable task period-deadline combination. Our heuristic minimizes the search region and iteratively finds a feasible period-deadline combination. Experimental results show that our method is much less pessimistic than existing techniques that consider task deadlines to be fixed parameters; our heuristic found a solution to the problem over 73% of the time using fewer than 3 search iterations.

As future work, we intend on (i) obtaining more experimental results, particularly using benchmarks derived from real applications, and (ii) implementing the proposed heuristic on a real-time operating system such as the S.Ha.R.K. kernel [41].

CHAPTER 3

A HOLISTIC SCHEDULING FRAMEWORK FOR ENERGY-CONSTRAINED WIRELESS REAL-TIME APPLICATIONS

We consider wireless nodes that execute computationally-intensive applications and must transmit packets over the network in a timely manner. Existing methods do not consider the importance (i.e., urgency) of a packet as perceived by end users in conjunction with energy consumption, real-time task deadlines, and packet deadlines, inadvertently causing packet priority inversion during transmissions and possibly starvation of some streams. We present a holistic scheduling framework that explicitly considers packet importance to select packets to transmit and guarantee their deadline requirements using both packet and energy-aware job assignment and scheduling. Our framework is applicable to wireless nodes equipped with either a single processor or a multicore system. Based on extensive simulations, we show that our proposed method allows for timely transmissions of the most important packets, which helps to control packet urgency, while saving processor(s) energy.

3.1 Introduction

In this section, we provide an overview of the problem to be solved, review key existing work, summarize our main contributions, and outline the organization of this chapter.

3.1.1 Problem Overview

Wireless networks are now common in a variety of applications such as habitat monitoring [69] and surveillance [45]. While many wireless sensor nodes typically require minimum hardware to perform lightweight tasks (e.g., periodically waking up to sense and transmit data), powerful processing nodes can also be found in certain applications for executing computationally intensive tasks and transmitting packets over the network. Some typical example applications are surveillance and mobile gaming systems. In a surveillance system, a wireless node periodically captures a video, processes frames and transmits them to clients. For the gaming system, the processor is kept busy with a large number of tasks (e.g., rendering graphics), while a large amount of data is sent to the user's opponents.

Typically, the performance focus of the systems described above is the quality of service (QoS) perceived by end users. For example, in a surveillance system, if an intruder enters a premise, it is crucial that not only the equipment captures the intruder on tape, but also that the information is sent to appropriate personnel in a timely manner so necessary actions can be taken. Similarly, gamers in a shooting game would want to see what their opponents are attempting to do from their screen within a certain time window.

For the types of systems described above, there are some important factors to be considered. Due to the time-sensitive nature of these systems, they are usually implemented using real-time tasks, which generate packets to be transmitted over the network. These packets, in turn, often contain time-sensitive materials. The systems have periodic access to send and receive packets, although the time and duration of access may depend on current network conditions (e.g., less often if there are many nodes in the network). Finally, energy consumption of these

wireless systems must be minimized since only occasional battery recharge may be possible.

The usual implementation of the systems under consideration is as follows. The processor executes the periodic real-time tasks using some real-time scheduling algorithm. Offline analysis based on task worst-case execution times can be used to guarantee that all deadlines will be met. When a packet is generated by a real-time task, it is sent by the processor to the network queue. A network scheduler then takes over and decides which packets to send and when. The main problem with this setup is that the decision to execute tasks and transmit packets are made independently and changing performance requirements are not communicated. Specifically, since a node's access to the network may sometimes be limited (due to event storms in sensor networks, high levels of interferences, and varying degrees of network density, for example), fewer packets can be transmitted during that time. While the network scheduler can select the most important packets to send first, the task scheduler, unaware of the situation, may not execute tasks that generate these important packets until later (e.g., when the node no longer has network access). To fully exploit periodic network access, a holistic scheduling approach, which considers packet importance levels and deadlines, as well as real-time task timing constraints, is needed.

3.1.2 Related Work

There is a large body of research on energy minimization for real-time applications, both for uniprocessors (e.g., [82, 113]) and multiprocessors (e.g., [3, 5, 27]). The majority of the work solely focuses on optimizing real-time task performance without any consideration for packet deadlines. At the same time, network-aware

work usually focuses on trading network energy with packet latency using packet scheduling and ignores task deadlines, e.g., [62, 101]. The problem of energy-aware scheduling of multiple components has been studied in the past, e.g., [46]. The most relevant papers propose some type of network-aware energy-minimization algorithms for real-time task scheduling [74, 83], which are still task-centric in that packet deadlines are not explicitly considered.

To the best of our knowledge, the work by Yi et al. is the only energy-aware solution that explicitly considers packet deadlines for systems executing real-time tasks [114]. However, this approach treats each packet as equally important, possibly resulting in scenarios where some tasks consistently get their packets transmitted while the packets of other tasks starve. In addition, it is unclear whether or how the work in [114] can be extended to multiprocessor architectures.

One way to ensure that both task and packet deadlines are met is to perform offline analysis to determine how much and when a node requires network access for all packets to be transmitted by their deadlines. However, network conditions unavoidably change and it cannot be guaranteed that each and every node will receive network access as requested. Also, while it is possible to model the relationship between tasks and packets using a directed acyclic graph (DAG), this solution is not viable as solving scheduling problems involving DAGs are usually performed offline due to the high time complexity.

3.1.3 Contributions

We address the gap in existing research by proposing a holistic scheduling framework that considers packet importance and current network conditions to make assignment and scheduling decisions. The proposed framework provides a

method to select and schedule the more important packets for transmission to control packet urgency. The packet schedule is then used to derive job deadlines. An LpEDF [113] based scheduling algorithm is used for nodes consisting of a single processor to save energy. For multicore nodes, a collection of energy-aware job assignment algorithms is studied. The solutions obtained by each of the components in the framework are evaluated against the optimal solutions and are shown to be a viable alternative to exhaustive search approaches. Comparison with an existing work reveals that the proposed framework improves on a specific QoS metric by about 30% on average using about 37% less energy. In addition, our approach reduces window constraint violations [109] by about 57.7%.

3.1.4 Organization

This chapter is organized as follows. We start by describing the system model and formally define the problem in Section 3.2. Section 3.2.3 serves to motivate the need for this work. Section 3.3 presents the holistic scheduling framework while the technical details are given in Sections 3.4 and 3.5. Section 3.6 discusses extensions to the framework. Simulation results are given in Section 3.7 and Section 3.8 concludes the chapter.

3.2 Preliminaries

We now describe our system model and provide some motivations for our work.

3.2.1 Task and Packet Model

We consider a set of n independent periodic real-time tasks. Each task τ_i is described by its worst-case execution time C_i, period T_i, implicit deadline $D_i = T_i$.

All tasks are synchronous. The j-th instance (job) of task τ_i is denoted by $J_{i,j}$. The absolute release time and deadline of $J_{i,j}$ are $r_{i,j}$ and $d_{i,j}$, respectively. Jobs are executed using EDF [66]. For the rest of the chapter, we simply use J_i to denote any individual job of τ_i when it is irrelevant to distinguish between, say, $J_{i,j}$ and $J_{i,k}$.

Without loss of generality, we assume that every job generates a packet at the end of its execution. Packets have firm real-time deadlines, i.e., they must be transmitted by their deadlines or they will be dropped. Packets from different instances of the same task are equal in size while packets from different tasks may vary in size. A packet $S_{i,j}$ is generated by $J_{i,j}$. We simply use S_i to denote a packet generated by any job of τ_i when it is irrelevant to distinguish between, say, $S_{i,j}$ and $S_{i,k}$. A packet S_i is described by its deadline offset X_i, worst-case transmission time Z_i, and importance W_i. Packet importance is a dynamically changing value that is used to capture the the urgency of a given packet. The actual importance of a packet depends on its content as well as past transmission history of earlier packets from the same stream. The job J_i, which generates S_i, also inherits that importance level. That is, J_i is more important than J_k if $W_i > W_k$. It is important to note that the importance of a task may be different from one instance to another. Finally, the absolute deadline of packet S_i is $Y_i = d_i + X_i$ where d_i is the absolute deadline of the job that generates S_i.

3.2.2 Hardware and Power Model

For the uniprocessor case, the processor runs at k discrete frequency levels. Note that the maximum frequency level $f_{\max} = f_k$. Each frequency level f_j is described by tuple (P_j, V_j), where P_j and V_j denote the power and voltage when

running the processor at frequency level f_j, respectively. For the multicore case, cores are homogeneous and can independently change frequency levels. For now, we assume P_j, $j = 1, \ldots, k$, only consists of dynamic power, which is a function of the frequency level, i.e., $P_j = f_j^3 \cdot P_{nom}$, where P_{nom} is the nominal power consumption. In other words, we ignore leakage power for now. This assumption will be relaxed in Section 3.6.2. The processor energy consumption is defined to be a product of power and time. Transition overheads associated with switching from one frequency level to another have been included in the task worst-case execution times.

There is a network card for each node. We assume packet transmissions cannot be preempted (so that transmission overheads are reduced). As in [114], each node uses TDMA-like periodic time slots to send and receive packets (Figure 3.1). Such a network communication model allows for contention-free communication among nodes. No network communication for a particular node takes place outside of its designated TDMA-like time slots. We assume that incoming packets are buffered at the sender and arrive at the beginning of each transmission window (denoted by the RX boxes in Figure 3.1). The time slots, described by a period T_{ts} and transmission window of length C_{ts}, may change over time to reflect different network usage levels given by the MAC layer protocol. For instance, event storms in sensor networks, high levels of interferences, and varying degrees of network density and traffic due to mobility may cause a node to be temporarily granted less network access (e.g., larger T_{ts} or smaller C_{ts}).

Since a node may be granted less network access due to situations described above, some of its packets may need to be dropped. Note that since packets have firm real-time deadlines, jobs that generate them must meet their deadlines or

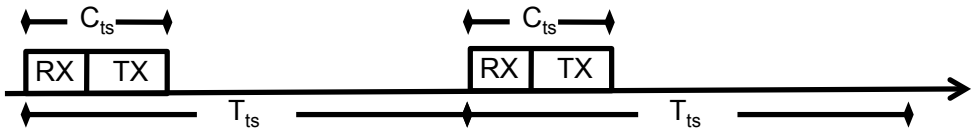

Figure 3.1. Network communication model.

there is no point in executing them. While it is possible to have scenarios where a job may miss its deadline but its packet is transmitted on time, we do not consider them in this work since we make an implicit assumption that job deadlines are required to ensure data freshness.

The system must minimize its energy consumption to stay alive for as long as possible. In this work, we will focus on the energy used to execute tasks since the energy consumption at the network card is already inherently considered via the periodic time slots (i.e., the network card is turned off or put to sleep outside of C_{ts}) and will not be further discussed.

3.2.3 Motivation

We use some simple examples to motivate the need for a holistic approach to job scheduling and packet transmission and highlight some key challenges. Assume that we have a set of three tasks (Table 3.1) running on a processor with the maximum normalized frequency level $f_{\max} = 1$. Task (and packet) importance levels are as shown in the last column of Table 3.1, with τ_3 being the most important and τ_1 being the least important. For the sake of clarity, we assume task and packet importance levels are fixed for this example. Recall that the absolute deadline of a packet $S_{i,j}$ generated by $J_{i,j}$ is $d_{i,j} + X_i$, where $d_{i,j}$ and X_i are the

TABLE 3.1

EXAMPLE TASK SET

Task	C	T	D	X	Z	W
τ_1	1	2	2	3	1	1
τ_2	1	3	3	3	1	2
τ_3	1	6	6	4	1	3

absolute deadline of J_i and the packet deadline offset of $S_{i,j}$, respectively.

To ease explanation, we assume in this example that jobs always require their worst-case execution times. Since the total utilization of the system is 1, the processor executes at the maximum frequency level for the entire duration. The job schedule is shown in Figure 3.2.

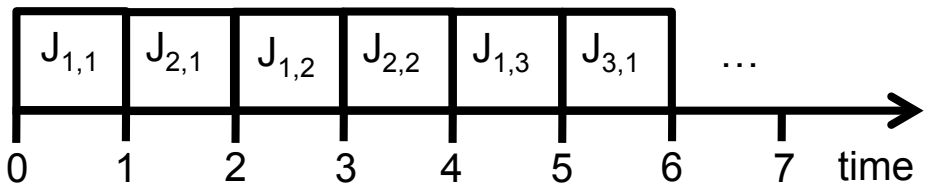

Figure 3.2. EDF job schedule.

Let us assume that the system has network access from time 3 to 6 (and will not get access again until time 12). Using EDF [66], the network schedule is as shown in Figure 3.3. Observe that $S_{3,1}$, despite being the most important packet, is not transmitted.

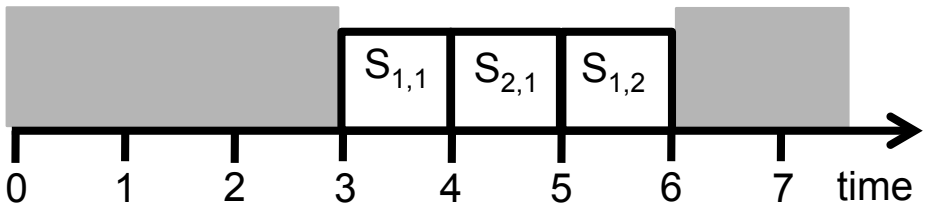

Figure 3.3. EDF packet schedule.

One solution is to modify the network scheduler to consider packet importance. If the network scheduler were to select the most important packet to schedule first, the resultant packet transmissions are as shown in Figure 3.4. Since the job scheduler is unaware of the needs at the network side, it does not give $J_{3,1}$ a higher priority over the less important jobs. In addition, in both scenarios described so far, many jobs were executed in hope that their packets would be transmitted. This unnecessarily wastes energy.

There exist value-based scheduling algorithms such as [19] in literature. Using a job scheduler that selects the most important job to schedule first, the resultant job and network schedules are as shown in Figures 3.5 and 3.6, respectively.

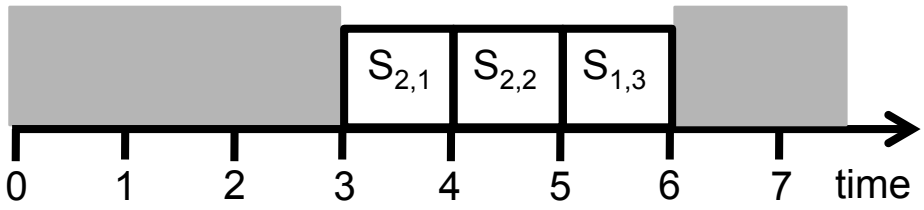

Figure 3.4. Importance-based packet schedule.

Here, given the network restriction, the value-based job scheduler performs well. However, if the system is granted more access time, say from time 1 to 7, the value-based job scheduler no longer leads to the best packet schedule (Figure 3.7). In this case, it is possible to transmit all packets in the interval under consideration. Note that scheduling algorithms that exploit skip models (e.g., [44] and [109]) suffer from similar shortcomings as value-based scheduling algorithms.

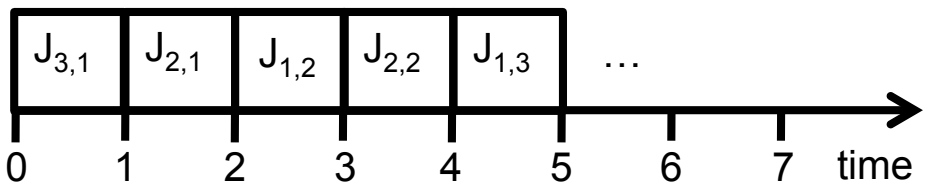

Figure 3.5. Value-based job schedule.

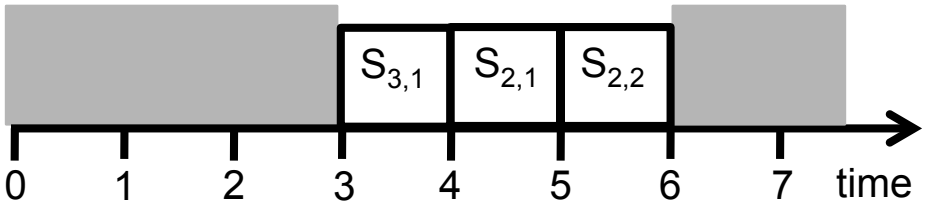

Figure 3.6. Resulting packet schedule.

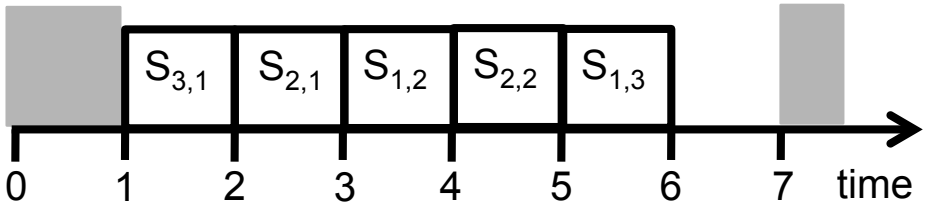

Figure 3.7. Resulting packet schedule with longer network access time.

To summarize, without specifically considering packet transmission schedule and network conditions while performing job scheduling, it is not possible to control actual packet transmissions. The situation is even more complicated when packet priority dynamically changes due to past transmission history. A scheduling approach that considers both job scheduling and packet transmission is needed to provide a holistic view of the system and allows for more processor energy to be saved.

3.3 Holistic Scheduling Framework

The interdependencies between job scheduling and packet scheduling, as well as their combined effect on meeting packet deadlines, make the goals stated earlier very difficult to solve exactly during run-time. We therefore resort to an adaptive approach. In contrast to traditional job scheduling techniques, we propose selecting and pre-scheduling more important packets to determine the latest packet release times (which is equivalent to determining the deadlines of corresponding jobs) and use these release times to make energy-aware job assignment and scheduling decisions.

The flow of our framework is shown in Figure 3.8 and is especially designed for systems in volatile networks. In the proposed framework, there are four main steps: (i) dynamically assigning packet importance based on past transmission history, (ii) pre-scheduling packets based on current network conditions, (iii) assigning and scheduling jobs, and (iv) actual job execution and packet transmission.

The primary goal in assigning packet importance is to associate packets with priorities. Assigning importance (i.e., priority) to a packet (or a job) is an old problem that has received significant research attention, e.g., [50, 104, 109]. In general, packets are assigned priorities dynamically based on several considerations such as the usefulness of its content as well as past transmission history. Specifically, whenever a packet misses its deadline, the importance (i.e., urgency) of subsequent packets in the same stream is increased to denote the urgency in servicing these packets and to avoid starvation [109]. As an example, consider the following scenario. Assume there are two surveillance tasks, τ_A and τ_B, in the system. Task τ_A monitors the door while task τ_B monitors the safe. In general, packets sent by τ_A may be considered more important than the ones generated

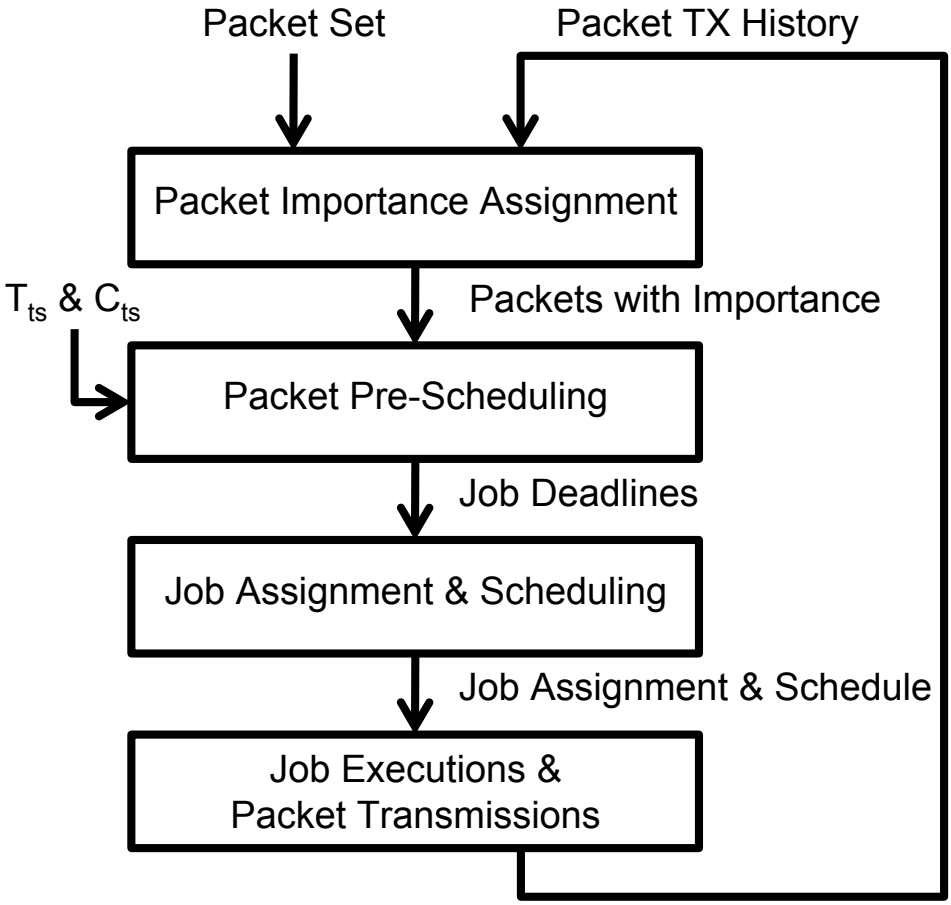

Figure 3.8. Proposed framework.

by τ_B, since the door is the first line of defense against intruders. However, once it is known that the door has been breached, the packets generated by τ_B likely becomes more important. Also, the importance of a packet may depend on past transmission history. For instance, even if packets generated by τ_A are generally considered more important than the ones generated by τ_B, it would be a poor use of the network if all of τ_A's packets and none of τ_B's packets are successfully transmitted in, say, the last 30 minutes.

The work in [109] provides a way to dynamically and adaptively determine how

important a packet is based on both its contents and past transmission history. Therefore, for the rest of this work, we will assume the method proposed in [109] is used to determine packet importance. Specifically, the current importance level of a task τ_i is defined to be

$$W_i = \frac{y_i - x_i}{y_i}, \tag{3.1}$$

where x_i is the maximum number of packets that can be dropped within a fixed window of y_i packets to ensure that the window constraint is not violated [109]. Both x_i and y_i are adjusted over time [109]. It is crucial to note, however, that our framework can be used in conjunction with any scheme that assigns packet importance.

Since we use existing work to assign packet importance, we focus on packet pre-scheduling and job assignment and scheduling. In the packet pre-scheduling step, given a set of available packets, the most important packets (possibly all) are selected for transmission and are scheduled in a way that maximizes the schedulability of the corresponding job set. As a result of this step, job deadlines may be modified. In the job assignment and scheduling steps, the jobs that generated selected packets are assigned and scheduled in an energy-aware manner.

3.4 Pre-Scheduling Packets

In this step, the objective is to select and schedule the most important packets for transmission. One main challenge is that the job schedule is undetermined at this point and since jobs generate packets, it is difficult to guarantee that the resultant packet schedule is indeed achievable. To address this challenge, there are two goals in pre-scheduling packets. First, we wish to select the most important packets to transmit. Second, we would like to schedule the selected packets in

such a way that packet release times are maximized in order to increase job laxity, which is defined to be the maximum time a task can be delayed once it has been released to complete within its deadline [16]. The latter objective helps to increase schedulability and provide more opportunities to reduce processor energy consumption.

Let t be the start time of next network access time interval C_{ts}. In addition, let $t + t_{RX}$ be the time instant when the node may start transmitting packets, where t_{RX} denotes the network access time reserved for receiving packets as shown in Figure 3.1. We set the current *time interval* $I = [t + t_{RX}, t + C_{ts}]$. During I, we consider a set of packets Ψ, where $\forall S_i \in \Psi$, the corresponding job release time occurs before the end of I and the absolute deadline of S_i occurs before the start of the next I. Also, for $S_i \in \Psi$, $\forall S_i$, if the original absolute deadline of S_i is greater than $t + C_{ts}$, then it will be set to $t + C_{ts}$. Note that this selection of the time interval I it is reasonable since the value of T_{ts} is usually in the order of several milliseconds, which tends to be much larger than typical real-time task periods (which are in the order of microseconds). That said, if the value of T_{ts} is comparable to task periods, I can be set to encompass several transmission windows of length C_{ts} each.

To determine $\Psi' \subseteq \Psi$ such that Ψ' contains the most important packets and is schedulable, we observe that to minimize the maximum value of dropped packets, the least important packets should be dropped first. However, finding a feasible packet set that minimize the value of the most important packet dropped is not our only goal. Recall that we also wish to increase the laxity of corresponding jobs. Let Ω' be the job set containing all jobs that generate the selected packets in Ψ'. Given a packet schedule, the transmission start time TXS_i of a packet S_i,

$\forall S_i \in \Psi'$ is known. This transmission start time in fact coincides with the latest time that job J_i must finish executing. In other words, for packet S_i, $\forall S_i \in \Psi'$, to be transmitted on time, the new absolute deadline d_i' of J_i is

$$d_i' = \min\{d_i, TXS_i\}. \tag{3.2}$$

We also define the transmission finish time $TXF_i = TXS_i + Z_i$.

As mentioned earlier, since a packet schedule, and hence packet transmission start times, is yet to be derived, it is not possible to determine if the job set Ω' is schedulable. However, we observe that from the job scheduling perspective, a job set is more likely to be feasible if its jobs have more laxity (i.e., more time to finish from the time they are released). For this reason, we attempt to schedule the packets in the select set Ψ' in such a way that TXS_i, and hence d_i', of selected packets is maximized in order to increase the laxity of the jobs in Ω'. As an added benefit, more energy savings can potentially be achieved when schedulable jobs have large laxities.

The goals in the packet pre-scheduling step can be formally stated as follows. Given the packet set Ψ, select Ψ' and schedule the packets $S_i \in \Psi'$, $\forall S_i$ such that for the corresponding job set Ω',

$$\alpha = \min_{J_i \in \Omega'}\{d_i' - r_i - C_i\} \tag{3.3}$$

is maximized and S_i meets its transmission deadline, $\forall S_i \in \Psi'$. Note that if $d_i' < r_i + C_i$, for some $J_i \in \Omega'$, then J_i cannot meet its deadline.

Given some packet set, one way to find a packet schedule that optimizes (3.3) is to formulate the problem as two mathematical programming instances, as shown

in Algorithm 3.

Algorithm 3 Optimal_Packet_Schedule(Ψ, Ω)

1: $[\Psi', \Omega'] \leftarrow$ MILP_1(Ψ, Ω) // Determine optimal Ψ'
2: MILP_2(Ψ', Ω') // Schedule packets in Ψ' to maximize minimum laxity

The first mixed-integer linear programming (MILP_1) problem can be expressed as follows. The variables we would like to solve for is the modified absolute job deadlines d'_j, $\forall J_j \in \Omega$. We first define some variables.

1.

$$\chi_i = \begin{cases} 1 & \text{if } S_i \text{ is not dropped} \\ 0 & \text{otherwise.} \end{cases} \tag{3.4}$$

2.

$$\eta_{i,j} = \begin{cases} 1 & \text{if } d'_j \geq d'_i \\ 0 & \text{otherwise.} \end{cases} \tag{3.5}$$

3.

$$\sigma_{i,j} = \begin{cases} 1 & \text{if } TXS_i \leq TXS_j \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

The objective function of the optimization problem is as follows.

$$\text{minimize} : \mathit{maxDropped} \tag{3.7}$$

where

$$\mathit{maxDropped} \geq W_i \cdot (1 - \chi_i), \forall S_i \in \Psi. \tag{3.8}$$

The value of the actual deadline of each job must satisfy the following two constraints.

$$d'_j \leq d_j, \forall J_j \in \Omega, \tag{3.9}$$

$$r_j + C_j \leq d'_j, \forall J_j \in \Omega. \tag{3.10}$$

Each packet must meet its deadline and cannot start until its release time at the earliest. That is,

$$\forall S_j \in \Psi$$

$$TXF_j = TXS_j + Z_j \cdot \chi_j, \tag{3.11}$$

$$Y_j \geq TXF_j, \tag{3.12}$$

$$d'_j \leq TXS_j + (1 - \chi_j) \cdot \Lambda, \tag{3.13}$$

where Λ is some large constant.

The following constraints are used to ensure that packet transmissions do not

overlap.

$$\forall S_i, S_j \in \Psi, S_i \neq S_j$$

$$TXF_i \leq TXS_j + (1 - \sigma_{i,j}) \cdot \Lambda + (1 - \chi_j) \cdot \Lambda + (1 - \chi_i) \cdot \Lambda, \quad (3.14)$$

$$TXF_j \leq TXS_i + \sigma_{i,j} \cdot \Lambda + (1 - \chi_j) \cdot \Lambda + (1 - \chi_i) \cdot \Lambda. \quad (3.15)$$

The following constraints are needed to ensure that the variables $\eta_{i,j}$, and $\sigma_{i,j}$, $\forall S_i, S_j \in \Psi$, are as defined.

$$\forall S_i, S_j \in \Psi$$

$$1 \leq \eta_{i,j} + \eta_{j,i}, \quad (3.16)$$

$$d'_i \leq d'_j + (1 - \eta_{i,j}) \cdot \Lambda, \quad (3.17)$$

$$d'_j \leq d'_i + \eta_{i,j} \cdot \Lambda - \epsilon, \quad (3.18)$$

$$\epsilon \leq d'_j - d'_i + \eta_{j,i} \cdot \Lambda, \quad (3.19)$$

$$1 \leq \sigma_{i,j} + \sigma_{j,i}, \quad (3.20)$$

$$TXS_i \leq TXS_j + (1 - \sigma_{i,j}) \cdot \Lambda, \quad (3.21)$$

$$TXS_j \leq TXS_i + \sigma_{i,j} \cdot \Lambda, \quad (3.22)$$

where ϵ is some small constant.

The second MILP formulation (MILP_2) is very similar, except that the objective function now becomes

$$\text{maximize} : minLax \quad (3.23)$$

where

$$minLax \leq d'_j - r_j - C_j, \forall J_j \in \Omega', \tag{3.24}$$

and the variables χ_i, $\forall S_i \in \Psi$, are removed.

While the MILP solver is guaranteed to return an optimal solution, it is not suitable for online use when the problem size is large since it is too computationally intensive. However, Algorithm 3 can be used to assess the performance of our simple heuristic, which we now introduce. To maximize (3.3), packets should be scheduled as late as possible. We propose scheduling packets with later deadlines first to maximize laxity. If two or more packets share the same deadline, ties are broken in favor of the packet S_i with the largest $r_i + C_i$, again, to maximize laxity. We keep track of the end of the current schedule using the variable t_{end}. That is, a packet cannot be scheduled after t_{end}. However, if a packet deadline occurs before t_{end}, then its transmission finish time will be set to its deadline. Algorithm 4 shows the steps required to select Ψ' and derive a packet schedule that is guaranteed to be feasible and that attempts to maximize (3.3).

When two or more packets share the same deadline, the packet S_i, which has a larger r_i+C_i, will start later than the packet S_j with a smaller r_j+C_j to maximize laxity. Note that Algorithm 4 ensures that the constraints in (3.9) and (3.10) are satisfied to ensure that the resultant packet transmission start times, and hence the deadlines of the corresponding jobs, are valid. Specifically, the constraint in (3.9) is ensured by Lines 7–8 of Algorithm 4. On the other hand, the constraint in (3.10) is ensured by Line 16. That is, if there exists a packet S_i which is set to be transmitted before the job that generates it can possibly finish executing, the least important packet is dropped and the schedule is reconstructed.

One problem that may arise with the above method of finding a packet sched-

Algorithm 4 Packet_Schedule(Ψ)

1: sort Ψ in non-increasing order of deadlines, ties broken in favor of the packet S_i with the largest $r_i + C_i$
2: *droppedSet* $\leftarrow \emptyset$
3: *done* $\leftarrow$ *false*
4: **while** *done* = *false* **do**
5: $t_{end} \leftarrow Y_0$ // Y_0 is the absolute deadline of the first packet in Ψ
6: **for** each $S_i \in \Psi$ **do**
7: **if** $t_{end} \geq Y_i$ **then**
8: $TXS_i \leftarrow Y_i - Z_i$ // schedule S_i by Y_i
9: $TXF_i \leftarrow Y_i$
10: **else** // Other packets have been scheduled later on so schedule S_i now
11: $TXS_i \leftarrow t_{end} - Z_i$
12: $TXF_i \leftarrow t_{end}$
13: $t_{end} \leftarrow TXS_i$
14: *flag* $\leftarrow$ *false*
15: **for** each $S_i \in \Psi$ **do**
16: **if** $r_i + C_i > TXS_i$ **then** // $r_i + C_i$ is the earliest time S_i can be generated
17: *flag* $\leftarrow$ *true*
18: **if** *flag* = *true* **then**
19: $\Psi \leftarrow \Psi - S_q$ // S_q is the packet of lowest importance
20: *droppedSet* $\leftarrow$ *droppedSet* $\cup S_q$
21: **else**
22: *done* $\leftarrow$ *true*
23: $\Upsilon \leftarrow$ packet schedule represented by a linked list sorted in non-increasing order of transmission start times // The node for packet S_i is tagged with TXS_i and TXF_i
24: sort *droppedSet* in a non-increasing order of importance
25: **for** $S_i \in$ *droppedSet* **do**
26: **for** each node $N_j \in \Upsilon$ **do**
27: $N_k \leftarrow N_j.nextNode$
28: **if** $N_k \neq \emptyset$ **then**
29: $s \leftarrow N_k.TXF$
30: **else**
31: $s \leftarrow t + t_{RX}$
32: $f \leftarrow N_j.TXS$
33: **if** $f \leq Y_i$ **and** $f - s \geq Z_i$ **and** $s \geq r_i + C_i$ **then**
34: $TXS_i \leftarrow s$
35: $TXF_i \leftarrow f$
36: $\Psi \leftarrow \Psi \cup S_i$
37: **break**

ule is that packets with lower importance may be dropped, even when it is not necessary. Specifically, consider two packets S_i and S_j where $W_i < W_j$. If S_j is dropped, then S_i is also dropped. While this dropping technique does not affect the maximum value of the most important packet dropped (see (3.8)), it is still desirable to execute S_i in the current time interval (if possible) since doing so may prevent an increase in W_i in the next time interval. For this reason, after a packet schedule has been obtained, we test to see if some of the dropped packets can be restored, as shown in Lines 23–37 of Algorithm 4. The worst-case running time of Algorithm 4 is in $\mathcal{O}(|\Psi|^2)$.

It is important to note that our approach requires a small modification to the network packet scheduler, which will use the packet schedule generated by Algorithm 4 instead of its default scheduling algorithm.

3.5 Energy-Aware Job Assignment and Scheduling

In the next step of our framework, energy-aware job assignment and scheduling algorithms are proposed in order to meet as many job deadlines as possible to minimize the maximum value of the most important packet that is dropped (equation (3.8)) while saving energy. We start by considering uniprocessor architectures before moving on to multicore systems.

3.5.1 Uniprocessors

In a single processor system, there is no need to consider how to assign jobs, only how to schedule them (i.e., there is only one processor that can execute these jobs). Recall that from the last section, we are provided with a job set Ω', which generate the packets that we would like to transmit. Our goal in this section is

to schedule as many jobs in Ω' as possible by their deadlines while minimizing energy.

While our tasks are originally periodic (Section 3.2), job deadlines may be modified (i.e., shortened) in the previous step of the framework. In other words, for the job set under consideration, each job has a release time, worst-case execution time, and deadline. To schedule these jobs, we propose using LpEDF [113], which has been proved to be optimal in terms of minimizing energy consumption of ideal processors. (For discrete-speed processors, two speed levels can be used to approximate the desired speed level, if the latter is not available [59]). LpEDF determines a frequency schedule for a given job set. Observe, however, that the job set Ω' is not necessarily schedulable, i.e., the resultant frequency schedule after applying LpEDF may contain frequency levels that are higher than $f_{\max}$. In such a case, some jobs will need to be dropped.

Since jobs inherit the importance of the packets they generate, determining which jobs (and therefore packets) to drop in this step can be accomplished in the same manner as selecting the most important packets in Algorithm 4. Therefore, we omit the repeated explanation but provide our job scheduling algorithm in Algorithm 5.

Algorithm 5 Uniprocessor_Job_Sched(Ω')

1: $\Pi \leftarrow$ job schedule obtained from Ω' using LpEDF [113] // each entry in Π is tuple $(e_i,\ J_i)$ where e_i and J_i denote the frequency level to be used for executing job J_i
2: **while** $\exists e_j \in \Pi : e_j > f_{\max}$ **do**
3: $\quad \Omega' \leftarrow \Omega' - J_q$ // J_q is the job whose packet is of lowest importance
4: $\quad \Pi \leftarrow$ job schedule obtained from Ω' using LpEDF [113]
5: **return** Π

We now discuss the performance of Algorithm 5 in the following theorem. The proof is trivial and is thus omitted.

Theorem 3.1 *For a given job set Ω' running on an ideal processor, Algorithm 5 minimizes the value of the most important packet that is dropped.*

The worst-case time complexity of Algorithm 5 is $\mathcal{O}(|\Omega'|^3 \log^2|\Omega'|)$ since LpEDF requires $\mathcal{O}(|\Omega'|^2 \log^2|\Omega'|)$ to run in the worst-case [113] and there can be at most $|\Omega'|$ iterations of the while loop. The time complexity of Algorithm 5 can be reduced to $\mathcal{O}(|\Omega'|^2 \log^3|\Omega'|)$ if binary search is used in the while loop instead of linear search. As with Algorithm 4, to prevent jobs with lower importance from being dropped even when it is unnecessary, a test can be performed to see if some of the dropped jobs can be restored. This addition is very similar to Lines 23–37 of Algorithm 4 and increases the worst-case running time of Algorithm 5 to $\mathcal{O}(|\Omega'|^3 \log^3|\Omega'|)$.

3.5.2 Multicore Systems

In a multicore system, we first need to assign $J_i \in \Omega'$, $\forall J_i$, to cores before scheduling can take place. The problem of task assignment (sometimes called partitioning) is NP-hard in general, except for frame-based tasks where all tasks share a common deadline [27]. Most existing work assume periodic tasks and fixed deadlines (e.g., [75]). Recall that the jobs under consideration have release times, worst-case execution times, and deadlines. In addition, job deadlines may be smaller than their implicit deadlines and the deadlines are not necessarily constant from one task instance to another. There exist research results on job-level assignment, e.g., [96], but they do not consider energy.

We focus on the energy-aware job assignment phase since once it is complete, the scheduling technique discussed in the last section can be straightforwardly applied on each processor. The state-of-the-art approach to assigning jobs to cores is to perform load balancing, as in [27]. To reduce energy, jobs should be assigned to cores in such a way so as to balance the energy consumption among cores. In the absence of significant leakage power, it is well-known that less energy is consumed if the processor runs as slow as possible (i.e., using the lowest possible frequency levels). We propose to study a collection of heuristics (three of which are our own designs) whose performance will be assessed in Section 3.7.2. In Section 3.6.2, we discuss how our approach can be extended to processors in which leakage power is significant.

- **Largest-Job First (LJF) [27]**: Sort jobs in a non-decreasing order of worst-case computation time and send a given job to the core with the least aggregated computation time.

- **Largest-Density First (LDF) [27]**: Same as LJF except that job densities are used instead of job worst-case computation times. The density of a job J_i is defined as $\delta = \frac{C_i}{D_i - r_i}$.

- **Most-Important First v.1 (MIF-1)**: Sort jobs in a non-decreasing order of importance and send a given job to the core with the least intensity during the job's active interval, which starts at the job's release time and ends at the job deadline. The intensity of the active interval of length a of a job J_i is $\frac{\sum_{J_k \in \Pi} C_k \cdot ov_k}{a}$, where Π is a set of jobs currently scheduled on the core under consideration and ov_k is the the ratio of the overlap between J_k and J_i over the active interval of J_i.

- **Most-Important First v.2 (MIF-2)**: Same as MIF-1 except that a given job is sent to the core with the least aggregated densities (similar to LDF).
- **Most-Important First v.3 (MIF-3)**: Same as MIF-1 except that a given job is sent to the core with the least aggregated importance.

Except for MIF-1, which runs in $\mathcal{O}(|\Omega'|^2 \cdot M)$, where M is the number of cores in the system, all the algorithms presented above have a worst-case time complexity of $\mathcal{O}(|\Omega'| \cdot \log |\Omega'| + |\Omega'| \cdot M)$.

Once all the jobs in Ω' have been assigned, Algorithm 5 can be used on each core to determine job schedules (and possibly drop some jobs).

3.6 Notes on Framework

We now discuss some generalizations and limitations of the proposed framework.

3.6.1 Extensions to Task and Packet Models

In Section 3.2, we made some simplifying assumptions with regards to the system model to make the proposed framework and technical details easier to digest. Specifically, we assumed that all jobs generate packets and packets are generated at the end of job execution. We first discuss the inclusion of non-packet generating tasks in our framework. If there are non-packet generating soft real-time tasks in the system, they can be assigned and scheduled when the system is idle and during the times where the node has no network access. On the other hand, the inclusion of hard real-time non-packet generating tasks in the system is part of our ongoing investigation.

The assumption that each packet is generated at the end of job execution is in fact not required in our framework. All analyses from the previous sections hold with regards to packet feasibility and job schedules; if a job generates a packet before it finishes executing, that packet arrives to the network queue earlier and has no adverse effect on the packet schedule (although it may increase the network buffer size). That said, more energy can be saved if it is known exactly when a job will generate a packet since job deadlines set in Section 3.4 can be extended accordingly.

Similarly, though all the algorithms implicitly assume that jobs (packets) require their worst-case execution times (transmission times), all the analyses in this chapter remain valid. Obviously, more energy can be saved if some type of slack reclamation is used but this topic is beyond the scope of this work.

3.6.2 Leakage Considerations

Due to shrinking device sizes and aggressive voltage scaling, subthreshold leakage current increases exponentially as the supply voltage is reduced, causing energy to be wasted when the system runs at very low frequency levels [49]. For such systems, running the processor below the critical frequency $f_{critical}$ is suboptimal [49] and can cause the system to consume more energy than necessary. While it is possible to adjust an existing frequency schedule by substituting any frequency level $f_j < f_{critical}$ with $f_{critical}$, a more energy-efficient schedule may exist.

There is a wealth of research on leakage-aware energy minimization for real-time systems, e.g., [49, 79]. In particular, the work by Niu and Quan [79] can be used on each processor after Algorithm 5 to obtain a leakage-aware schedule.

3.6.3 Applicable Network Types

Our framework can be used in both single-hop and multi-hop networks. In a multi-hop network, some types of deadline decomposition methods (e.g., [51]) can be applied to determine local packet deadlines at each hop and used in our framework.

3.7 Evaluation

This section presents simulation results to demonstrate the effectiveness of our framework. We discuss the performance of the packet pre-scheduling and job assignment and scheduling steps, and compare our framework with an existing work that solves a similar problem.

3.7.1 Pre-Scheduling Packets

Recall that the primary goals of Algorithm 4 are to select the most important packets to schedule and derive a schedule that maximizes (3.3). We compared the performance of our heuristic (Algorithm 4) with that of the MILP (Algorithm 3), which was solved using CPLEX with AMPL. The time limit was set to 1 minute for the CPLEX solver, which means that if an optimal solution has not been found within that time, the best feasible solution found so far will be returned.

For our benchmarks, 100 tasks sets consisting of between 10-45 tasks each were randomly generated for 10 different utilizations ($U = 0.5, 0.75, \ldots, 2.75$) with a total of 1000 task sets overall. The utilization is defined to be $\sum_{i=1}^{n'} \frac{C_i}{T_i}$ where n' is the total number of tasks in the system. Task periods ranged from 50 to 300 time units, with the worst-case execution time being set between 30-60% of the period. The absolute deadline of a packet is set to be equal to the absolute deadline of

the corresponding job. Packet sizes vary randomly between 1-3.5% of C_{ts}, which is set to 300 time units. Packet importance is randomly generated and is between $[0, 1]$. Finally, $|I| = 300$.

The first set of results are summarized in Figure 3.9, which shows the value of the most important packet dropped (i.e., the value of equation (3.8)) on the y-axis as a function of system utilization. When the utilization is low, there are fewer tasks in the system and hence fewer packets. On the other hand, more packets are generated when the utilization is high because there are more tasks in the system. Algorithm 3 was able to keep every packet while our heuristic approach dropped some of them. That said, Algorithm 4 dropped very few packets and only the least important ones. On average, the normalized value of the most important packet dropped is 0.0073 (0.73% deviation on average in other words).

In terms of maximizing the minimum job laxity (Figure 3.10), Algorithm 4 performs very well when the total task utilization is low (and fewer packets are generated). As the total task utilization increases, Algorithm 4 deviates more from the results obtained using Algorithm 3. Specifically, the average minimum laxity from Algorithm 4 is about 17.68% smaller than that of Algorithm 3 and up to 47.59%. It must be emphasized, however, that Algorithm 4 performs very well in terms of minimizing the value of the most important packet dropped. In addition, while the MILP solver can be used for very small problem instances, it is too computationally intensive for solving medium to large problem instances online (and it cannot guarantee that any feasible solution will be found within some time limit). Algorithm 4, which is very efficient, can be used to bridge the gap.

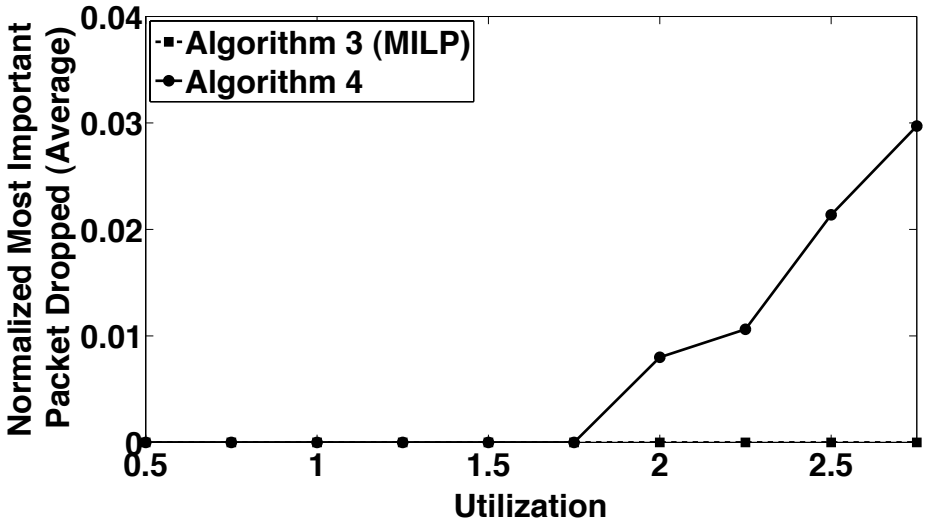

Figure 3.9. Value of most important packet dropped from different packet scheduling algorithms.

3.7.2 Job Assignment & Scheduling

In this section, we assess the performance of the proposed energy-aware job assignment and scheduling algorithms presented in Section 3.5.2. We compared our solutions with the optimal solutions obtained by a brute-force algorithm. In the brute-force approach, all possible job assignments are explored and the job assignment that is both feasible and results in the least amount of energy consumed is identified.

In the simulations, we assume a multicore system consisting of two identical cores. Each core is modeled after the Intel Core 2 Duo [47], with a maximum power consumption of 65 W and seven normalized frequency levels: 0.462, 0.615, 0.692, 0.769, 0.846, 0.923, and 1. System utilization ranges from $1.0, 1.1, \ldots, 1.9$. A total of 100 task sets were generated for each utilization (2000 task sets in total).

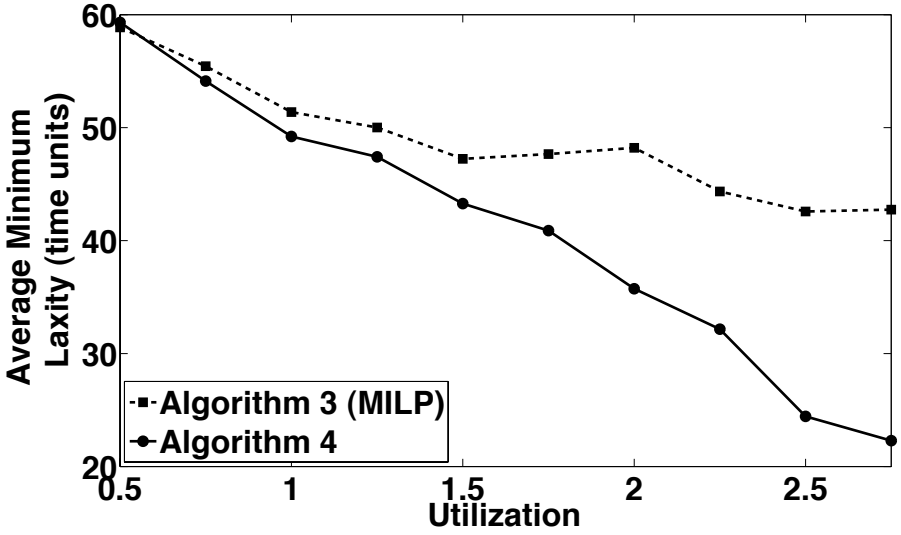

Figure 3.10. Minimum job laxity of different packet scheduling algorithms.

The same method and parameters as in the last section were used to generate the task sets. Since the total number of job assignment combinations is $|M|^{|\Omega|}$ where $|M|$ is the number of cores and $|\Omega|$ is the number of jobs, we limit the number of jobs to 20 to stay within the range of a long integer (this is also the reason why we limit the number of cores to only two). Specifically, we discarded any task sets containing more than 20 jobs during I and regenerate them until 100 task sets were found for each utilization.

Figure 3.11 shows the percentage of schedulable task sets as a function of system utilization. With only two cores and at most 20 jobs, LJF yields the best results, with 100% of the task sets found as feasible, followed by MIF-2 (97%). MIF-1 shows the worst performance, with only 83% of the task sets determined as feasible. In terms of energy consumption, LJF again yields the best results,

followed closely by MIF-2 and LDF (Figure 3.12).

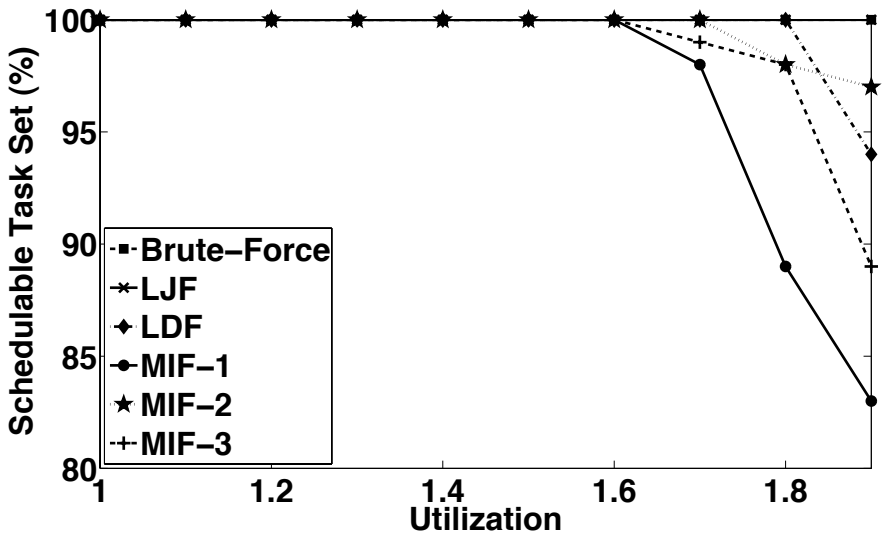

Figure 3.11. Number of dropped jobs by different job assignment algorithms for the two-core case.

To assess the performance of the different job assignment algorithms when the problem size is larger, we performed an additional set of simulations. The system is assumed to have four identical cores, which are again modeled after the Intel Core 2 Duo [47]. The system utilization ranges from $2.5, 2.75, \ldots, 5$. A total of 100 task sets were generated for each of the utilization (1100 task sets in total).

Figure 3.13 shows the maximum importance level of dropped jobs as a function of system utilization for the five job assignment algorithms discussed in Section 3.5.2. Overall, MIF-2 and MIF-3 yield the best results, though they can

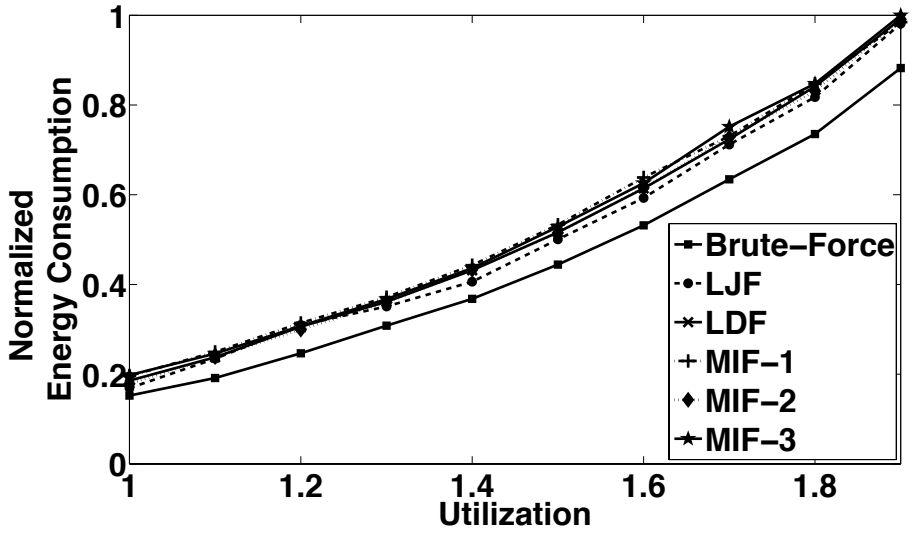

Figure 3.12. Energy consumption of different job assignment algorithms for the two-core case.

be outperformed by LJF and LDF in some cases. MIF-2 and MIF-3 improve on the value of most important job dropped by about 1.08-2.67% on average when compared to LJF and LDF. In terms of energy consumption, MIF-2 and MIF-3 once again outperform LJF and LDF on average (though not by much, as shown in Figure 3.14).

Based on the simulation data, MIF-2 provides the best performance for a given energy consumption level, although the performance of all five algorithms are close, which may suggest the limits of this type of job assignment algorithms. In any case, since the brute-force approach takes several minutes to find the optimal solution for even a small benchmark, it cannot be used online. Based on the results given above, MIF-2 performs well enough (and is very efficient) to be a viable alternative.

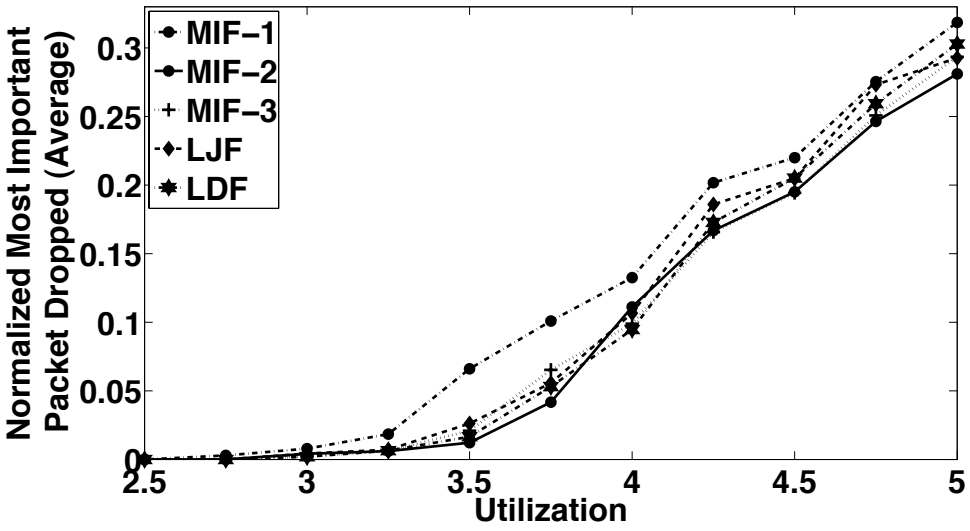

Figure 3.13. Value of most important job dropped from different job assignment algorithms for the four-core case.

3.7.3 Entire Framework

We compare the effectiveness of our proposed framework with the most closely related work, which is by Yi et al. [114]. In Yi's algorithm, both packet and job deadlines are considered but all packets are assumed to be equally important. Simulations were performed assuming a single processor core is used since the work in [114] only focuses on uniprocessor scenarios. The task sets and other parameters are generated in the same way as in Sections 3.7.1 and 3.7.2 except for the following parameters. The average system utilization is set to 0.5 and the size of a single packet varies randomly between 0.5-30% of C_{ts}. The larger the packet size, the more loaded the network is and the harder it is to transmit all packets by their deadlines.

The graph in Figure 3.15 shows the maximum importance level of dropped

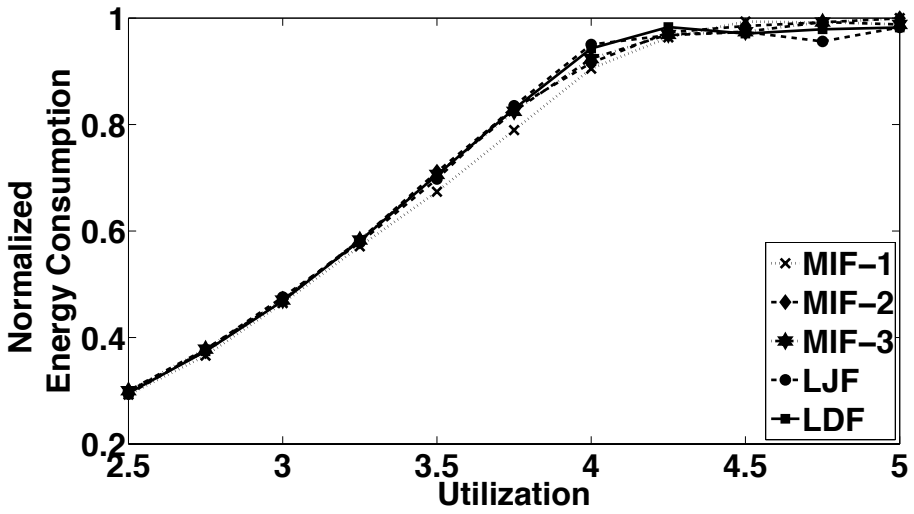

Figure 3.14. Energy consumption of different job assignment algorithms for the four-core case.

packets as a function of packet size. As expected, our proposed holistic framework always outperforms Yi's algorithm [114], which does not consider packet nor job importance. On average, our framework reduces the average maximum importance of dropped packets by 30% (up to 96.4%). As the network becomes more and more overloaded, our framework is forced to drop more and more packets (but always the least important ones first). This is the reason why the curve representing the holistic framework tends to increase as the packet size increases. As for Yi's algorithm, it is hard to predict which packets will be dropped since all packets are considered to be equally important. For instance, Yi's algorithm often results in important packets being dropped (Figure 3.15). However, at times, it may by chance transmit more important packets instead of less important ones (i.e., the dip in Figure 3.15).

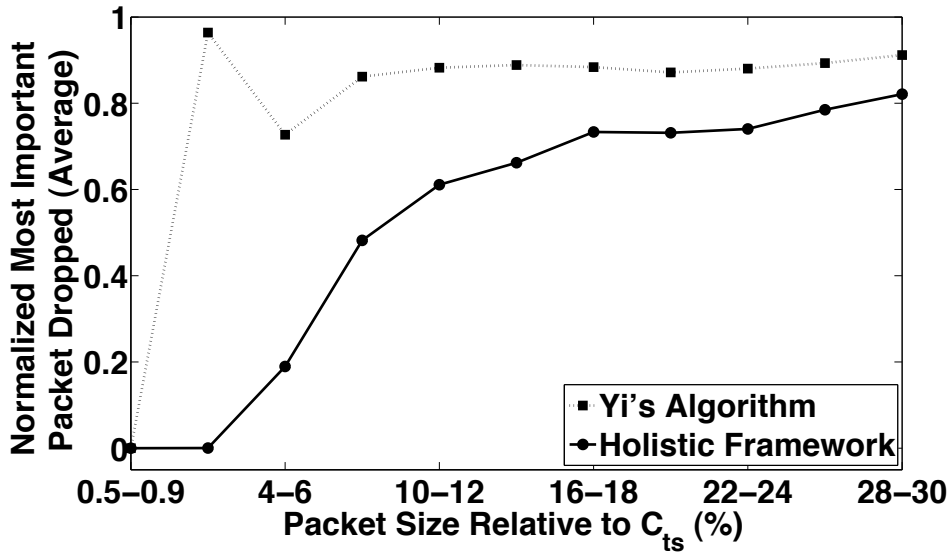

Figure 3.15. Comparison to the work in [114] in terms of minimizing the maximum importance of dropped packets.

As expected, our proposed framework also results in less energy consumed (Figure 3.16). When the network is underloaded (shown towards the left side of Figure 3.16), our method pre-schedules jobs and packets ahead of time to maximize job laxity and hence reduces energy consumption. In contrast, Yi's algorithm greedily tries to send out packets as soon as possible, even when it is unnecessary. This results in large energy consumption. As the network becomes more and more loaded, our approach saves energy by dropping jobs whose packets are never transmitted. On the other hand, Yi's algorithm always executes jobs, even if the corresponding packets are dropped. The dip in the curve representing Yi's algorithm can be explained as follows. As the network becomes more loaded, the next available transmission time becomes further away in the future, allowing jobs to be executed at lower speed. As the packet size increases, however, more energy

will be consumed because jobs need to finish earlier to send larger packets out. This is the reason why the curve slowly increases again after the dip. In summary, our approach saves energy by 37.3% on average when compared to Yi's algorithm (up to 78.5% and at least 10%).

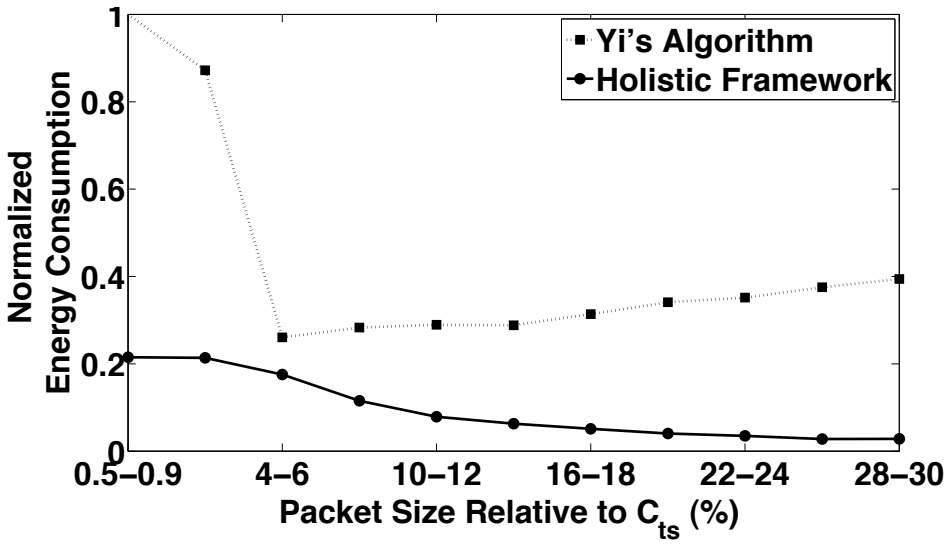

Figure 3.16. Comparison to the work in [114] in terms of energy consumption.

A performance metric that is often used when all packets are considered to be equally important is the deadline meet ratio, which is the ratio between the number of packets transmitted by the deadline and the total number of packets during some time interval. Since our approach always gives a higher priority to the most important packets first, it is expected that Yi's algorithm will outperform

our framework in terms of packet deadline meet ratio. This is indeed the case, as shown in Figure 3.17. However, the performance difference is very small. On average, Yi's algorithm sends out 1.7% more packets on average and up to 3.1% when compared to our framework. This is because even when more important packets are dropped, we still attempt to send out as many packets as possible.

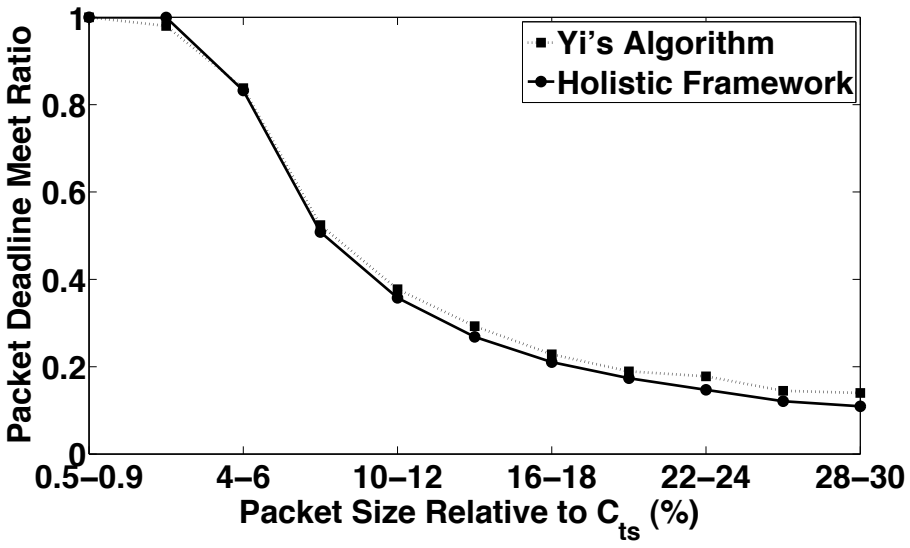

Figure 3.17. Comparison to the work in [114] in terms of packet deadline meet ratio.

Finally, to show that our approach is able to control packet urgency over time, we performed a set of simulations where task importance is dynamically adjusted in each window using the rules in [109]. A total of 100 task sets consisting of 10 tasks each were simulated for 15,000 time units. The task set utilization is 0.5 and

the average packet sizes vary randomly between 0.5-2% of the C_{ts}. For the DWCS window constraint [109], each task starts with a y value of 20. The starting x value varies randomly between 1 and 19. The DWCS window constraint is set to $\frac{x}{y}$, which indicates that at most x packets can be dropped within a window of y packets. A violation occurs when more than x packets are transmitted in a window of y packets for any given stream. The values of x and y changes over time depending on past transmission history. In general, the value of y for a stream is decreased if its previous packet is serviced before its deadline. For more information, readers are referred to Figures 2 and 3 in [109]. Additional parameters used in this set of simulations are shown in Table 3.2. The last column of Table 3.2 denotes the increase in packet sizes to simulate changing transmission rate.

Table 3.3 shows the results. Since we are using the window constraint in [109] as the performance metric, the goal is to minimize the number of window constraint violations while saving energy. As shown in Table 3.3, our approach significantly improves upon Yi's algorithm both in terms of the number of window constraint violations and energy by intelligently dropping some packets. As a result, our method has a higher deadline miss ratio. Based on this set of data, it can be concluded that selecting the most important packets for transmission is an effective way to dynamically adapt to changing packet priorities. Also, since the holistic framework only executes jobs that are necessary, it offers significant energy savings compared to Yi's algorithm.

TABLE 3.2

ADDITIONAL PARAMETERS FOR SIMULATIONS OVER SEVERAL TIME INTERVALS

Time Interval	C_{ts}	T_{ts}	Packet size increase
$[0 : 5,000]$	300	500	0%
$[5,000 : 10,000]$	100	500	0%
$[10,000 : 15,000]$	300	500	10%

TABLE 3.3

RESULTS FOR SIMULATIONS OVER SEVERAL TIME INTERVALS

	Yi's Algorithm [114]			Holistic Framework		
	avg	max	min	avg	max	min
Number of violations	350.2	581.0	196.0	148.3	402.0	1.0
Normalized energy (%)	84.66	100	72.95	30.33	40.82	24.54
Deadline miss ratio	0.467	0.560	0.376	0.710	0.877	0.592

3.8 Summary

We presented a holistic scheduling framework that considers packet and job deadlines, as well as packet importance, to ensure timely transmissions of the most important packets while saving processor(s) energy consumption. The proposed framework contains three main steps, two of which include novel algorithms that can be used to partition and schedule general real-time jobs.

While our main future goal is to implement the proposed framework on a real system, there are some extensions to be explored. Such extensions include support for preemptive packet transmissions, processor transition overheads, and heterogeneous multicore systems.

CHAPTER 4

ONLINE WORK MAXIMIZATION UNDER A PEAK TEMPERATURE CONSTRAINT

Increasing power densities and the high cost of low thermal resistance packages and cooling solutions make it impractical to design processors for worst-case temperature scenarios. As a result, packages and cooling solutions are designed for less than worst-case power densities and dynamic voltage and frequency scaling (DVFS) is used to prevent dangerous on-chip temperatures at run time. Unfortunately, DVFS can cause unpredicted drops in performance (e.g., long response times). We propose and optimally solve the problem of thermally-constrained online work maximization for general-purpose computing systems on uniprocessors with discrete speed levels and extend our policy to consider non-negligible transition overhead. Simulation results show that our approach completes 47.7% on average and up to 68.0% more cycles than a naïve policy.

4.1 Introduction

We start by providing an overview of the problem. We then present some related work, state our contributions, and present the chapter outline.

4.1.1 Problem Overview

In response to the increasing computing demands made by applications, system designers have been delivering processors with higher performance at the expense of increasing power densities and temperatures. High chip temperature has significant impact on reliability, performance, cost, and power consumption; microprocessor failure rate depends exponentially upon operating temperature [99]. Temperature also affects speed; reduction of charge carrier mobility in transistors and increased interconnect latency from high temperature degrade performance. To guarantee that the processors do not reach unsafe temperatures, packages and cooling solutions can be designed to handle worst-case temperature profiles. However, this solution is prohibitively expensive, since the cost of cooling solutions increases super-linearly in power consumption [43].

Another, less expensive, solution to the problem is to use processor throttling at run time: when the chip temperature exceeds some pre-specified threshold, the processor power consumption and performance are temporarily reduced by hardware or the operating system. Unfortunately, throttling can cause significant, and difficult-to-predict, increase in response times. In general-purpose computing, throttling may lead to significant performance loss. In real-time systems, throttling may cause more deadline misses.

In this chapter, the goal is to minimize task response times by maximizing the work completed over time in an online manner where no prior knowledge of the workload is required. Dynamic Voltage and Frequency Scaling (DVFS) is used to keep the chip temperature under some threshold. Maximizing the work completed can also be useful in soft real-time systems where the objective is to meet as many deadlines as possible, since it can reduce the number of deadline misses. This

claim is substantiated in Section 4.6.

In real processors such as the Intel Core Duo, the processor will run at the highest speed (as long as there is still work to be completed) until the chip temperature reaches the threshold temperature. Once this occurs, the system will either reduce its duty cycle or switch to a lower voltage and frequency pair to cool the chip down [47]. However, it is unclear how much and how long the processor should temporarily reduce its performance level in order to cool down while maximizing the total work completed over time.

4.1.2 Related Work

While the problem of maximizing work to be completed using DVFS has been well explored for many different systems, only a small number of papers consider temperature constraints, e.g., [26, 48, 85, 87, 111]. Existing online solutions either assume that the processor has the ability to transition from one speed level to another in a continuous manner (e.g, [9], [89], and [78]) or that the processor can use some arbitrary speed values [89, 106, 107], which is not a realistic assumption since speed levels can vary from one chip to another even for the same type of processors due to process variation. On a related note, for processors that can continuously adjust speed, the Pontryagin Maximum Principle [84] can be used to find an optimal solution.

The work in [106] is the most closely related work. In that work, the processor runs at the highest speed until the threshold temperature is reached. Once this happens, the *equilibrium* speed, which depends on the processor power consumption, thermal resistance, and threshold temperature $T_{\max}$, will be used to keep the chip temperature at the threshold temperature. This simple scheme allows

the authors to perform many important analyses such as bounding the worst-case response time of a task and deriving a schedulability test [106]. However, the equilibrium speed is determined based on the assumption that the processor can adjust its speed in a continuous manner. The problem with the equilibrium speed is that it may fall between two discrete speed levels. Even if manufacturers select the equilibrium speed as one of the discrete levels supported, we cannot realistically expect that due to different packaging, ambient temperature, and workload. Most importantly, due to process variation, chip characteristics vary from one chip to another, even if these chips come from the same batch.

In general, the assumptions on processor speed levels made by existing work are unrealistic for several reasons. First, a processor cannot have infinite number of speed levels due to design and validation difficulties. Even if a processor did support many speed levels, there is overhead associated with each speed transition and hence the number of transitions is limited. Second, it is impractical to assume that processors can support any discrete set of speed levels once the chip has been fabricated.

To the best of our knowledge, there is no existing work that identifies and explains the characteristics of an optimal clock throttling policy for maximizing the work completed for processors with discrete speed levels and non-negligible transition overhead. The Intel chips have two thermal management policies [47]. Once the chip temperature reaches the threshold temperature, the first mechanism (known as *Thermal Monitor 1*), which is also the default mechanism, reduces the duty cycle by some factory-configured percentage until the chip temperature drops below the maximum temperature and the hysteresis timer has expired. The second mechanism (*Thermal Monitor 2*), which is user-configurable, is very similar to

the first one, except that the processor will throttle to cool the processor down. While these mechanisms seem reasonable, there exists no work that studied their effectiveness. In addition, it is unclear how the user may select the appropriate speed levels and the associated time durations to maximize the amount of work to be completed. Finally, most existing industry thermal management solutions have operated under the assumption that thermal emergencies are rare events, for which reactive techniques are sufficient. Now and in the future, processors will often operate near their thermal emergency temperatures, requiring proactive thermal management to maintain performance.

4.1.3 Contributions

In this chapter, we tackle the problem of determining speed schedules that maximizes the work completed under a given maximum temperature constraint. We propose an optimal clock throttling policy based on several fundamental observations for processors with discrete speed levels and non-negligible transition overhead. Our policy is applicable to any uniprocessor architecture and requires only two speed levels to maximize the work completed. The two speed levels are alternated in a periodic manner (see Section 4.3 for more details), with some high speed being applied until the chip temperature reaches the threshold temperature. Extensive simulation results are used to verify our claims and demonstrate that our optimal clock throttling policy completes 47.64% on average and up to 67.97% more cycles than a naïve clock throttling policy.

4.1.4 Organization

This chapter is outlined as follows. We present our system model and problem definition in the next section. Section 4.3 provides our formal approach to solving the work maximizing problem under peak temperature constraint when transition overhead is negligible. The impact of transition overhead is studied in Section 4.4. The performance of our optimal clock throttling policy is presented in Section 4.6. Section 4.7 concludes this chapter.

4.2 System Model and Problem Definition

We describe the system model, state our assumptions and formally define the problem.

4.2.1 Task and Processor Models

We consider a DVFS-enabled processor with a temperature threshold $T_{\max}$. When the processor temperature reaches this threshold, the processor starts throttling, i.e., switching from some high speed level to some lower speed level to reduce power consumption and performance.

We adopt the lumped RC thermal model similar to that used by Zhang and Chatha [115], which is based on Fourier heat transfer model [40]. The die temperature above the ambient temperature after t time units is

$$T = \hat{T} + (T_0 - \hat{T}) \cdot e^{-\frac{t}{\tau}} \tag{4.1}$$

where $\hat{T}$ is the die's steady-state temperature and $\hat{T} = P_{dyn} \cdot R$, with P_{dyn} being the dynamic power consumption of the die and R its resistance. In addition, τ is

the chip time constant, and T_0 is the initial die temperature. The expression in (4.1) was obtained by solving the following differential equation for T:

$$RC\frac{dT}{dt} + T - RP = 0. \tag{4.2}$$

Although we will use (4.1) for the rest of the chapter, all of our derivations hold for any exponential temperature equation of the same form. For instance, we can replace (4.1) with the temperature equation obtained by Rao and Vrudhula where the die and package are modeled separately [87]. We can also extend (4.1) to account for leakage power by noting that a piecewise-linear function can be used to estimate leakage power in the operating temperature ranges with roughly 5% error [68]. That is, the modified (4.1) can be obtained by solving the following:

$$RC\frac{dT}{dt} + T - R(P_{dyn} + P_{leak}) = 0, \tag{4.3}$$

where $P_{leak} = \alpha T + \beta$ for some constants α and β [68].

For each speed level k of the processor, we define an associated tuple (V_k, S_k, P_k), where V_k, S_k, and P_k are the required voltage, speed, and power consumption of the processor when it executes at speed level k, respectively. Without loss of generality, we assume that each speed level S_k has been normalized to fall within the interval $[0, 1]$. For speed level k, (4.1) can be written as

$$T = \hat{T}(S_k) + (T_0 - \hat{T}(S_k)) \cdot e^{-\frac{t}{\tau}}, \tag{4.4}$$

where $\hat{T}(S_k)$ is the steady-state temperature when the processor executes at speed level k and $\hat{T}(S_k) = S_k^3 P_k R$.

4.2.2 Problem Definition

Given a processor that is kept busy with work to be completed, determine a speed schedule such that the peak temperature constraint is met and total work completed is maximized.

4.3 Work Maximizing Speed Selection Strategy for Processors with Negligible Transition Overhead

We describe a policy for maximizing the work completed over a schedule length based on some crucial observations. Namely, we determine (i) the speed levels needed, (ii) the length of time the processor should spend in each speed level, and (iii) the sequence of speed levels in which the processor should execute.

For now, we assume that the processors under consideration have negligible transition overhead. This simplifying assumption allows us to identify some important characteristics of an optimal clock throttling policy for maximizing the work completed. As will be shown in Section 4.4, the policy described in this section can be adjusted to handle processors with non-negligible transition overhead.

Our objective is to develop an optimal DVFS control policy for use once the chip reaches its threshold temperature, and not a pre-throttling policy. In many systems, the time from startup to reaching the temperature constraint is a negligible percentage of total time. In addition, it is important to note that our DVFS control policy does not perform task scheduling, though it can be used in conjunction with any existing task scheduling algorithm.

Consider a high speed level S_H where $\hat{T}(S_H) \geq T_{\max}$. During throttling, if the processor execute tasks using S_H for long enough, the chip peak temperature will eventually reach $T_{\max}$. Our first question is whether such a high speed should be

used until the chip temperature reaches $T_{\max}$ or be used for a shorter amount of time. The following lemma answers this question. In addition, the "sufficiently large time interval" requirement is there to ensure that the time interval under consideration is long enough for the chip temperature to reach $T_{\max}$ at least once.

Lemma 4.1 *Given a sufficiently large time interval $[t_a, t_b]$, let S_{L1}, S_{L2}, and S_H be some speed levels satisfying $\hat{T}(S_{L1}) < T_{\max}$, $\hat{T}(S_{L2}) < T_{\max}$, and $\hat{T}(S_H) \geq T_{\max}$ and let the transition overhead be negligible. Consider the speed schedules that apply S_{L1}, S_H, and then S_{L2} in a consecutive manner during $[t_a, t_b]$ given some initial temperature $T_a \geq \min\{\hat{T}(S_{L1}), \hat{T}(S_{L2})\}$ and some end temperature T_b. A schedule that completes the maximum amount of work must allow the chip temperature to reach $T_{\max}$ at the end of the application of S_H.*

Proof: Let t_1 and t_3 be the time durations during which S_{L1} and S_{L2} are applied, respectively. In addition, let T_1 and T_2 be the temperatures at the end of the applications of S_{L1} and S_H, respectively. Figure 4.1 provides a graphical depiction of the corresponding speed schedule. The work completed during $[t_a, t_b]$ can be expressed as

$$W = S_{L1} \cdot t_1 + S_H \cdot (t_b - t_a - t_1 - t_3) + S_{L2} \cdot t_3 \tag{4.5}$$

where

$$t_1 = \tau \ln \left(\frac{T_2 - \hat{T}(S_H) - (T_a - \hat{T}(S_{L1})) \cdot e^{-\frac{(t_b - t_a)}{\tau}} \cdot \left(\frac{T_2 - \hat{T}(S_{L2})}{T_b - \hat{T}(S_{L2})} \right)}{(\hat{T}(S_{L1}) - \hat{T}(S_H)) \cdot e^{-\frac{(t_b - t_a)}{\tau}} \cdot \left(\frac{T_2 - \hat{T}(S_{L2})}{T_b - \hat{T}(S_{L2})} \right)} \right) \tag{4.6}$$

$$t_3 = \tau \ln \left(\frac{T_2 - \hat{T}(S_{L2})}{T_b - \hat{T}(S_{L2})} \right). \tag{4.7}$$

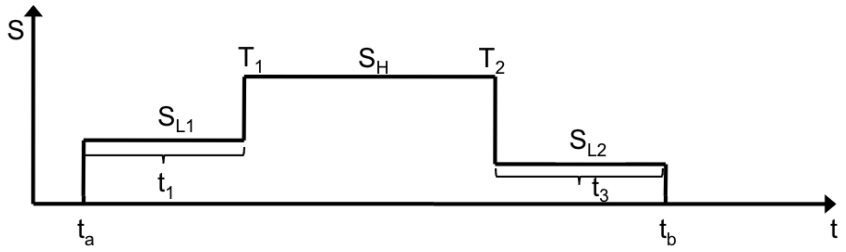

Figure 4.1. Graphical depiction of the speed schedule in the statement of Lemma 4.1.

To determine how the work completed changes as a function of T_2, we take the partial derivative of W with respect to T_2, which yields

$$\frac{\partial W}{\partial T_2} = \frac{\tau}{T_2 - \hat{T}(S_{L2})} \cdot \left[S_{L2} - S_H + \frac{(S_{L1} - S_H) \cdot (\hat{T}(S_H) - \hat{T}(S_{L2}))}{T_2 - \hat{T}(S_H) - (T_a - \hat{T}(S_{L1})) \cdot e^{-\frac{(t_b - t_a)}{\tau}} \cdot \left(\frac{T_2 - \hat{T}(S_{L2})}{T_b - \hat{T}(S_{L2})} \right)} \right]. \tag{4.8}$$

By writing the temperature equations for T_1 and T_2 and using substitutions, it can be shown that

$$\begin{aligned} &(\hat{T}(S_H) - \hat{T}(S_{L1}))e^{-\frac{t_b - t_a - t_1 - t_3}{\tau}} \\ &= T_2 - \hat{T}(S_H) - (T_a - \hat{T}(S_{L1})) \cdot e^{-\frac{(t_b - t_a)}{\tau}} \cdot \left(\frac{T_2 - \hat{T}(S_{L2})}{T_b - \hat{T}(S_{L2})} \right). \end{aligned} \tag{4.9}$$

Hence,

$$\frac{\partial W}{\partial T_2} = \frac{\tau}{T_2 - \hat{T}(S_{L2})} \cdot \left[S_{L2} - S_H + \frac{(S_H - S_{L1}) \cdot (\hat{T}(S_H) - \hat{T}(S_{L2}))}{(\hat{T}(S_H) - \hat{T}(S_{L1}))e^{-\frac{t_b - t_a - t_1 - t_3}{\tau}}} \right]. \tag{4.10}$$

Since $T_2 \geq \hat{T}(S_{L2})$, $\frac{\partial W}{\partial T_2} \geq 0$ if the expression inside the parentheses of (4.10) is greater than zero. We know that

$$\hat{T}(S_H) - \hat{T}(S_{L2}) \geq \hat{T}'(S_H)(S_H - S_{L2}), \tag{4.11}$$

which leads to the following

$$\begin{aligned} &(S_{L2} - S_H) \cdot (\hat{T}(S_H) - \hat{T}(S_{L1}))e^{-\frac{t_b - t_a - t_1 - t_3}{\tau}} + (S_H - S_{L1}) \cdot (\hat{T}(S_H) - \hat{T}(S_{L2})) \\ &\geq (S_{L2} - S_H) \cdot (\hat{T}(S_H) - \hat{T}(S_{L1}))e^{-\frac{t_b - t_a - t_1 - t_3}{\tau}} + (S_H - S_{L1})(S_H - S_{L2}) \cdot \hat{T}'(S_H) \\ &\geq 0. \end{aligned} \tag{4.12}$$

We now show that the expression in (4.12) is greater than or equal to zero as follows.

$$\begin{aligned} 0 &\leq \hat{T}'(S_H)(S_H - S_{L1}) - (\hat{T}(S_H) - \hat{T}(S_{L1}))e^{-\frac{t_b - t_a - t_1 - t_3}{\tau}} \\ \hat{T}'(S_H)(S_{L1} - S_H) &\leq (\hat{T}(S_{L1} - \hat{T}(S_H))e^{-\frac{t_b - t_a - t_1 - t_3}{\tau}} \\ \hat{T}(S_{L1}) - \hat{T}(S_H) &\leq (\hat{T}(S_{L1}) - \hat{T}(S_H))e^{-\frac{t_b - t_a - t_1 - t_3}{\tau}} \\ 1 &\geq e^{-\frac{t_b - t_a - t_1 - t_3}{\tau}} \end{aligned} \tag{4.13}$$

This means that $\frac{\partial W}{\partial T_2} \geq 0$ and the work completed is maximized when T_2 is maximized. This, in turns, implies that T_2 must equal $T_{\max}$ for the work completed to

be maximized.

□

In Lemma 4.1, we referred to the high (S_H) and low (S_{L1} and S_{L2}) speeds with the requirements that $\hat{T}(S_{L1}), \hat{T}(S_{L2}) < T_{\max}$ and $\hat{T}(S_H) \geq T_{\max}$. Since modern processors often have several speed levels, we need to determine which of these speed levels actually allow the maximum amount of work to be completed. The following lemmas provide the basis for our speed selection policy when the processor starts to throttle. That is, once the chip temperature reaches $T_{\max}$.

Lemma 4.2 *Given a sufficiently large time interval $[t_a, t_b]$, let S_{L1}, S_{L2}, and S_H be speed levels satisfying $\hat{T}(S_{L1}) < T_{\max}$, $\hat{T}(S_{L2}) < T_{\max}$, and $\hat{T}(S_H) \geq T_{\max}$ and let the transition overhead be negligible. Given an initial temperature $T_a \geq \min\{\hat{T}(S_{L1}), \hat{T}(S_{L2})\}$ and an end temperature T_b, consider the speed schedules that apply S_{L1}, S_H, and then S_{L2} in a consecutive manner during $[t_a, t_b]$. If the duration of the application of S_{L1} or S_{L2} is not zero, a schedule that completes the maximum amount of work must satisfy $S_{L1} = S_{L2} = \max\{s|\hat{T} < T_{\max}\}$ and $S_H = \min\{s|\hat{T} \geq T_{\max}\}$.*

Proof: Let t_1 and t_2 denote the durations of the application of S_{L1} and S_{L2}, respectively. Let t_3 denotes the duration of the application of S_H and $t_3 = t_b - t_a - t_1 - t_2$. In addition, let $T(t_1)$ and $T(t_3)$ be the temperatures at the end of the application of S_{L1} and S_H, respectively. We can write three associated

temperature equations as follows.

$$T(t_1) = \hat{T}(S_{L1}) + (T_a - \hat{T}(S_{L1}))e^{-\frac{t_1}{\tau}} \tag{4.14}$$

$$T(t_3) = \hat{T}(S_H) + (T(t_1) - \hat{T}(S_H))e^{-\frac{t_b - t_a - t_1 - t_2}{\tau}} \tag{4.15}$$

$$T_b = \hat{T}(S_{L2}) + (T(t_3) - \hat{T}(S_{L2}))e^{-\frac{t_2}{\tau}}. \tag{4.16}$$

In addition, from Lemma 4.1, we know that $T(t_3) = T_{\max}$. Letting

$$A = e^{-\frac{t_b - t_a}{\tau}} \cdot \left(\frac{T_{\max} - \hat{T}S_{L2}}{T_b - \hat{T}S_{L2}} \right) \tag{4.17}$$

and combining (4.14) with (4.15) yield

$$t_1 = \tau \ln \left(\frac{\hat{T}(S_H) - T_{\max} + (T_a - \hat{T}(S_{L1}))A}{(\hat{T}(S_H) - \hat{T}(S_{L1}))A} \right). \tag{4.18}$$

By definition, the total work completed during $[t_a, t_b]$ is

$$W = (S_{L1} - S_H) \cdot t_1 + (S_{L2} - S_H) \cdot t_2 + S_H \cdot (t_b - t_a). \tag{4.19}$$

To determine the appropriate value of S_{L1} that maximizes W, we take the partial derivative of W with respect to S_{L1}, observing that neither A nor t_2 depends on S_{L1}. We obtain the following result.

$$\begin{aligned} \frac{\partial W}{\partial S_{L1}} = \tau \ln & \left(\frac{\hat{T}(S_H) - T_{\max} + (T_a - \hat{T}(S_{L1}))A}{(\hat{T}(S_H) - \hat{T}(S_{L1}))A} \right) \\ & + \frac{\tau(S_{L1} - S_H)\hat{T}'(S_{L1})}{\hat{T}(S_H) - \hat{T}(S_{L1})} \cdot \left(\frac{\hat{T}(S_H) - T_{\max} + A(T_a - \hat{T}(S_H))}{\hat{T}(S_H) - T_{\max} + A(T_a - \hat{T}(S_{L1}))} \right). \end{aligned} \tag{4.20}$$

From (4.20), we observe that $\frac{\partial W}{\partial S_{L1}} = 0$ if $T_{\max} = \hat{T}(S_H) - A(\hat{T}(S_H - T_a))$. However,

we know that $T_{\max} < \hat{T}(S_H) - A(\hat{T}(S_H - T_a))$ due to the very definition of S_H. This means that $T_a > A^{-1}(T_{\max} - \hat{T}(S_H) + \hat{T}(S_H)A)$. Now, we wish to determine how $\frac{\partial W}{\partial S_{L1}}$ changes as T_a increases. To do so, we take the second partial derivative of W, this time with respect to T_a. We have

$$\frac{\partial^2 W}{\partial T_a \partial S_{L1}} = \tau \frac{A}{\hat{T}(S_H) - T_{\max} + A(T_a - \hat{T}(S_{L1}))} \cdot \left(1 + \frac{A(S_{L1} - S_H)\hat{T}'(S_{L1})}{\hat{T}(S_H) - T_{\max} + A(T_a - \hat{T}(S_{L1}))}\right). \tag{4.21}$$

Since

$$\hat{T}(S_H) - T_{\max} + A(T_a - \hat{T}(S_{L1})) > 0 \tag{4.22}$$

and

$$1 + \frac{A(S_{L1} - S_H)\hat{T}'(S_{L1})}{\hat{T}(S_H) - T_{\max} + A(T_a - \hat{T}(S_{L1}))} \geq 0, \tag{4.23}$$

we have

$$\frac{\partial^2 W}{\partial T_a \partial S_{L1}} \geq 0. \tag{4.24}$$

As a consequence, as T_a increases, $\frac{\partial W}{\partial S_{L1}}$ also increases. In other words, $\frac{\partial W}{\partial S_{L1}} \geq 0$ for all valid values of T_a and $S_{L1} = \max\{s|\hat{T} < T_{\max}\}$.

We can use the same technique to prove that $S_H = \min\{s|\hat{T} \geq T_{\max}\}$ and $S_{L2} = \max\{s|\hat{T} < T_{\max}\}$.

□

The following theorem generalizes the results from the above lemma to an arbitrary number of speed transitions.

Theorem 4.1 *Given a sufficiently large time interval* $[t_a, t_b]$*, let* S_{L1}*,* S_{L2}*, and* S_H *be speed levels satisfying* $\hat{T}(S_{L1}) < T_{\max}$*,* $\hat{T}(S_{L2}) < T_{\max}$*, and* $\hat{T}(S_H) \geq T_{\max}$

and let the transition overhead be negligible. Given an initial temperature $T_a \geq \min\{\hat{T}(S_{L1}), \hat{T}(S_{L2})\}$ *and an end temperature* T_b*, consider the speed schedules that apply* S_{L1}*,* S_H*, and then* S_{L2} *in a consecutive manner for a fixed number of times during* $[t_a, t_b]$*. If the duration of the application of* S_{L1} *or* S_{L2} *is not zero, a schedule that completes the maximum amount of work must satisfy* $S_{L1} = S_{L2} = \max\{s|\hat{T} < T_{\max}\}$ *and* $S_H = \min\{s|\hat{T} \geq T_{\max}\}$*.*

Proof: We prove the theorem by induction on the number of speed transitions within $[t_a, t_b]$, which is denoted by i.

The basis: For $i = 1$, there are two time intervals, and some low speed S_L is used either in the first or the second time interval. In either case, it is trivial to show that selecting $S_L = \max\{s|\hat{T} < T_{\max}\}$ will maximize the work completed for the corresponding scenario. Additionally, in the other interval, the high speed S_H that maximizes that work completed must satisfy $S_H = \min\{s|\hat{T} \geq T_{\max}\}$.

The induction step: Assume that the theorem holds for $i \leq n$. At the $(i = n+1)$-th transition, there are four possibilities: (i) $S_H \rightarrow S_{L1}$, (ii) $S_{L1} \rightarrow S_H$, (iii) $S_H \rightarrow S_{L2}$, (ii) $S_{L2} \rightarrow S_H$. For (i), let t_1 be an arbitrary time point in the S_{L1} interval immediately before the newly added S_{L1} interval. According to the induction assumption, regardless of the actual value of $T(t_1)$, the maximum work completed in $[t_a, t_1]$ is achieved by setting $S_{L1} = \max\{s|\hat{T} < T_{\max}\}$. For interval $[t_1, t_b]$, by Lemma 4.2, the maximum work completed is also completed in $[t_1, t_b]$ when $S_{L1} = \min\{s|\hat{T} < T_{\max}\}$. A similar reasoning can be used for (ii), (iii), and (iv). Thus, the total work completed is maximized when $S_{L1} = S_{L2} = \max\{s|\hat{T} < T_{\max}\}$ and $S_H = \min\{s|\hat{T} \geq T_{\max}\}$.

□

As a direct consequence of Theorem 4.1, a DVFS control policy that maximizes

the work completed only needs to use two speed levels: $S_H = \min\{s|\hat{T} \geq T_{\max}\}$ and $S_L = \max\{s|\hat{T} < T_{\max}\}$. Incorporating these results, the following theorem generalizes the observation from Lemma 4.1 to an arbitrary number of high speed intervals.

Theorem 4.2 *Given a sufficiently large time interval $[t_a, t_b]$, let S_L and S_H be two speed levels satisfying $\hat{T}(S_L) < T_{\max}$ and $\hat{T}(S_H) \geq T_{\max}$ and let the transition overhead be negligible. Given an initial temperature $T_a \geq \min\{\hat{T}(S_{L1}), \hat{T}(S_{L2})\}$ and an end temperature T_b, let S_L and S_H be alternately applied during the time interval $[t_a, t_b]$ and let the total number of speed transitions be fixed. A schedule that completes the maximum amount of work must allow the chip temperature to reach $T_{\max}$ at the end of every application of S_H.*

Proof: We prove the theorem by induction on the number of speed transitions within $[t_a, t_b]$, which is denoted by i.

The basis: For $i = 1$, there are two time intervals, and S_H can occur either in the first or the second time interval. In either case, it is trivial to show that reaching $T_{\max}$ at the end of S_H leads to the maximum work completed for the corresponding scenario.

The induction step: Assume that the theorem holds for $i \leq n$. At the $(i = n + 1)$-th transition, either (i) $S_H \rightarrow S_L$ or (ii) $S_L \rightarrow S_H$. For (i), let t_1 be an arbitrary time point in the S_L interval immediately before the newly added S_L interval. According to the induction assumption, regardless of the actual value of $T(t_1)$, the maximum work completed in $[t_a, t_1]$ is achieved by each S_H interval reaching $T_{\max}$ at the end of the interval. For interval $[t_1, t_b]$, by Lemma 4.1, the maximum work completed is also completed in $[t_1, t_b]$ when $T_{\max}$ is reached at the end of the single S_H interval. Thus, the total work completed is maximized when

$T_{\max}$ occurs at the end of every S_H interval. A similar but simpler reasoning can be used for (ii).

□

Theorems 4.1 and 4.2 provide a theoretical foundation for any work-maximizing, DVFS control policy. That is, the theorems specify the speed levels needed and indicate that it is advantageous in terms of maximizing the work completed to alternate between the high and low speed levels while allowing the chip to reach the maximum temperature at the end of every high speed application interval. We now need to determine how long the low speed level should be applied and whether each low speed level interval should have the same duration. To answer these questions, we begin by defining a periodic speed schedule then showing that such a schedule is part of the optimal DVFS control policy.

Definition 1 *A periodic speed schedule is a speed schedule that alternately applies S_L and S_H (where $\hat{T}(S_L) < T_{\max}$ and $\hat{T}(S_H) \geq T_{\max}$) in a time interval such that the durations of all applications of S_H are the same and the durations of all applications of S_L are the same.*

Lemma 4.3 *Given a time interval $[t_a, t_b]$ with the initial chip temperature of $T_{\max}$, let S_L and S_H be two speed levels satisfying $\hat{T}(S_L) < T_{\max}$ and $\hat{T}(S_H) \geq T_{\max}$ and let the transition overhead be negligible. Let S_L and S_H be alternately applied during the time interval $[t_a, t_b]$ with S_H being applied until the chip temperature reaches $T_{\max}$ and let the speed schedules follow the pattern S_L, S_H, S_L, and S_H. A schedule that completes the maximum amount of work must be a uniform speed schedule.*

Proof: The total work completed using the uniform schedule can be expressed as

$$W = S_L(t_b - t_a) + (S_H - S_L) \cdot 2p, \tag{4.25}$$

where p is the duration of each of the two applications of S_H. On the other hand, the total work completed using the non-uniform schedule can be written as

$$W' = S_L(t_b - t_a) + (S_H - S_L) \cdot (r + q), \tag{4.26}$$

where r and q denote the duration of the first and second applications of S_H, respectively. Observe that to compare W with W', we only need to compare $2p$ with $r + q$.

We can derive an expression for p as

$$p = \tau \, ln \left(\frac{\hat{T}(S_H) - \hat{T}(S_L)}{\hat{T}(S_H) - T_{\max} + (T_{\max} - \hat{T}(S_L))e^{-\frac{t}{2\tau}}} \right). \tag{4.27}$$

where $t = t_b - t_a$.

On the other hand, r and q can be expressed as

$$r = \tau \, ln \left(\frac{\hat{T}(S_H) - \hat{T}(S_L)}{\hat{T}(S_H) - T_{\max} + (T_{\max} - \hat{T}(S_L))e^{-\frac{t_R}{\tau}}} \right) \tag{4.28}$$

$$q = \tau \, ln \left(\frac{\hat{T}(S_H) - \hat{T}(S_L)}{\hat{T}(S_H) - T_{\max} + (T_{\max} - \hat{T}(S_L))e^{-\frac{t_Q}{\tau}}} \right), \tag{4.29}$$

where $t_R + t_Q = t_b - t_a$. We wish to show that the uniform speed profile yields more work completed. In other words, that $2p - (r + q) \geq 0$, which will be the

case if

$$(\hat{T}(S_H) - T_{\max} + (T_{\max} - \hat{T}(S_L))e^{-\frac{t_R}{\tau}})(\hat{T}(S_H) - T_{\max} + (T_{\max} - \hat{T}(S_L))e^{-\frac{t_Q}{\tau}}) \geq \left(\hat{T}(S_H) - T_{\max} + (T_{\max} - \hat{T}(S_L))e^{-\frac{t}{2\tau}}\right)^2, \quad (4.30)$$

since $ln(1) = 0$ and ln is an increasing function. Simplifying the above expression, we need to show that

$$e^{-\frac{t_R}{\tau}} + e^{-\frac{t_Q}{\tau}} - 2e^{-\frac{t}{2\tau}} \geq 0. \quad (4.31)$$

Letting $t_R = \frac{t}{2} + \Delta$, $t_Q = \frac{t}{2} - \Delta$, and ignoring τ, we have

$$e^{-\left(\frac{t}{2}+\Delta\right)} + e^{-\left(\frac{t}{2}-\Delta\right)} - 2e^{-\frac{t}{2\tau}} \geq 0$$

$$e^{\Delta} + e^{-\Delta} \geq 2, \quad (4.32)$$

which is true when $\Delta = 0$. In addition, since

$$\frac{d(e^{\Delta} + e^{-\Delta})}{d\Delta} = e^{\Delta} - e^{-\Delta}, \quad (4.33)$$

which is greater than or equal to zero, the lemma holds.

□

We now generalize the results from the above lemma to an arbitrary number of speed transitions.

Theorem 4.3 *Given a sufficiently large time interval $[t_a, t_b]$ with the initial chip temperature of $T_{\max}$, let S_L and S_H be two speed levels satisfying $\hat{T}(S_L) < T_{\max}$ and $\hat{T}(S_H) \geq T_{\max}$ and let the transition overhead be negligible. Let S_L and S_H be alternately applied during the time interval $[t_a, t_b]$ with S_H being applied until the*

chip temperature reaches $T_{\max}$ and let the total number of speed transitions be n. A schedule that completes the maximum amount of work must be a periodic speed schedule.

Proof: The theorem can be proved using the same technique as in Lemma 4.3. That is, ignoring τ, equation (4.31) can be generalized to

$$\begin{aligned} 0 \leq C_1 \left(\sum_{t_i \in \Gamma} e^{-t_i} - n \cdot e^{-\frac{t(n-(n-1))}{n}} \right) + \\ C_2 \left(\sum_{t_i, t_j \in \Gamma, t_i \neq t_j} e^{-(t_i+t_j)} - n \cdot e^{-\frac{t(n-(n-2))}{n}} \right) + \\ C_3 \left(\sum_{t_i, t_j, t_k \in \Gamma, t_i \neq t_j \neq t_k} e^{-(t_i+t_j+t_k)} - n \cdot e^{-\frac{t(n-(n-3))}{n}} \right) + \\ \ldots \end{aligned} \tag{4.34}$$

where Γ contains the durations of all n applications of S_H for the non-uniform speed schedule and where $C_1, C_2, \ldots$ are constants. The above inequality can be shown to hold using substitutions and derivatives as before.

□

Though Lemma 4.3 describes a desired property of a work-maximizing speed schedule, it does not specify the length of the S_L intervals. The time duration in which the processor applies S_L determines the number of speed transitions in a given time interval. For processors with negligible transition overhead, more transitions would lead to more work completed (i.e., the duration of every application of S_L should be minimized), as shown by the following theorem.

Theorem 4.4 *Given a sufficiently large time interval $[t_a, t_b]$, let S_L and S_H be two speed levels satisfying $\hat{T}(S_L) < T_{\max}$ and $\hat{T}(S_H) \geq T_{\max}$ and let the transition overhead be negligible. Let S_L and S_H be alternately applied. Consider the periodic speed schedules that use the same initial and final speed levels. A schedule with m speed transitions completes more work than a schedule with n speed transitions if $m > n$.*

Proof: The speed schedule with m speed transitions completes the following amount of work

$$W = S_L(t_b - t_a) + (S_H - S_L) \cdot m \cdot p, \tag{4.35}$$

where p is the duration of each of the m applications of S_H. Similarly, the speed schedule with n speed transitions completes the following amount of work

$$W' = S_L(t_b - t_a) + (S_H - S_L) \cdot n \cdot q, \tag{4.36}$$

where q is the duration of each of the n applications of S_H.

As previously, p and q can be expressed as

$$p = \tau \, ln \left(\frac{\hat{T}(S_H) - \hat{T}(S_L)}{\hat{T}(S_H) - T_{\max} + (T_{\max} - \hat{T}(S_L))e^{-\frac{t}{m\tau}}} \right) \tag{4.37}$$

$$q = \tau \, ln \left(\frac{\hat{T}(S_H) - \hat{T}(S_L)}{\hat{T}(S_H) - T_{\max} + (T_{\max} - \hat{T}(S_L))e^{-\frac{t}{n\tau}}} \right), \tag{4.38}$$

where $t = t_b - t_a$ and

$$m \cdot p = \tau \, ln \left(\frac{\hat{T}(S_H) - \hat{T}(S_L)}{\hat{T}(S_H) - T_{\max} + (T_{\max} - \hat{T}(S_L))e^{-\frac{t}{m\tau}}} \right)^m \tag{4.39}$$

$$n \cdot q = \tau \, ln \left(\frac{\hat{T}(S_H) - \hat{T}(S_L)}{\hat{T}(S_H) - T_{\max} + (T_{\max} - \hat{T}(S_L))e^{-\frac{t}{n\tau}}} \right)^n. \tag{4.40}$$

Although $q \geq p$, $m \cdot p \geq n \cdot p$ due to the exponential nature of equations (4.39) and (4.40) (and since $m > n$). Hence, $W \geq W'$.

□

We now summarize our **optimal DVFS control policy**. To maximize the work completed, the processor should periodically alternate between the low speed $S_L = \max\{s|\hat{T} < T_{\max}\}$ and the high speed $S_H = \min\{s|\hat{T} \geq T_{\max}\}$. In addition, the processor should run at the high speed S_H until the chip temperature reaches $T_{\max}$. With negligible transition overhead, the processor should minimize the time it spends running at the low speed (i.e., the throttling time) by switching to the high speed as soon as possible.

4.4 Work Maximizing Speed Selection Strategy for Processors with Non-Negligible Transition Overhead

Each speed transition imposes some overhead, reducing the amount of time spent on computation. Figure 4.2 illustrates the typical trajectories for voltage and speed levels during two transitions. When transitioning from a lower speed level to a higher speed level, the voltage is gradually increased until it reaches the required value (we have simplified the voltage curve to a straight line when in reality it is a staircase curve). Once this happens, the processor switches to the higher speed. During this transition, there is a small time interval α during which the processor clock is halted and no work is completed. The process of transitioning from a higher speed to a lower speed is similar, except that the processor switches to the new speed immediately and gradually decreases the voltage. Once again, the processor clock is halted for a short duration β. Typical

values for α and β are on the order of tens of microseconds. The voltage changing times, a and b, are on the order of hundreds of microseconds.

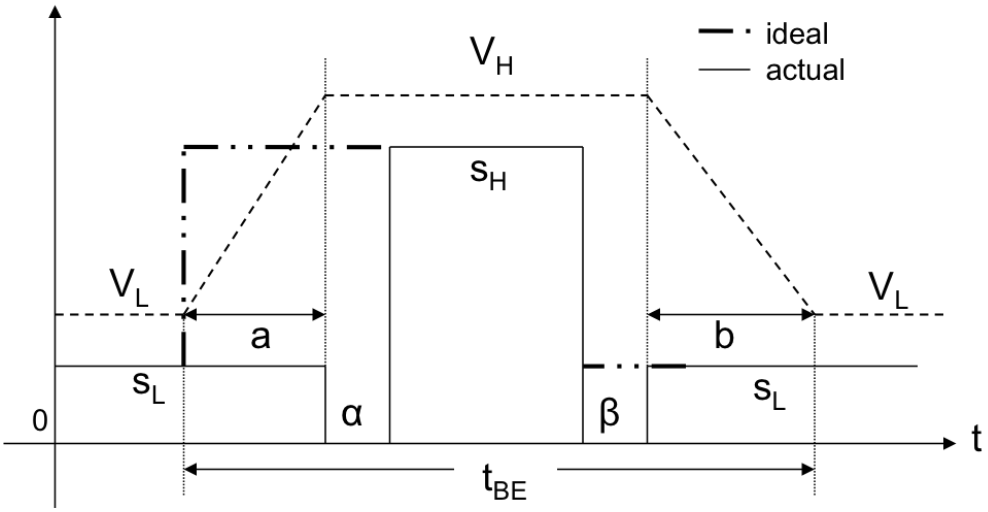

Figure 4.2. Waveforms of speed and voltage levels for two transitions, where S and V denote speed and voltage, respectively. During transitions, the voltage is increased or decreased in a small step size and thus there is a delay associated with each transition.

Compared to the scenario where there is no transition overhead (denoted as "ideal" in Figure 4.2), there is no work loss during b. During α and β, the number of cycles lost is $S_H \cdot \alpha$ and $S_L \cdot \beta$, respectively. Finally, during a, the number of cycles lost is $(S_H - S_L) \cdot a$. To find the optimal value of the time the processor spends at the low speed level t_l, we find the maximum value of the net work completed function that accounts for transition overhead.

Figure 4.3 confirms this intuition. It shows the net work completed in cycles as a function of throttling times for different values of a. As we can see from the curves, when the throttling times are too small, the net work completed is actually negative (that is, no useful work is being done). Clearly, to maximize the work completed when transition overhead is non-negligible, we must find the optimal value of throttling time t_l, which we now present. Our method involves finding the maximum point of the function that represents the net work completed when transition overhead are accounted for.

Given a schedule length L, the net work completed W^* is

$$W^* = [(t_l - \beta + a) \cdot S_L + (t_h - \alpha - a) \cdot S_H] \cdot \frac{L}{t_l + t_h}, \tag{4.41}$$

where t_h is the duration of the high speed level application.

The following theorem identifies the optimal value for t_l.

Theorem 4.5 *Given a schedule length L, let S_L and S_H be two speed levels satisfying $\hat{T}(S_L) < T_{\max}$ and $\hat{T}(S_H) \geq T_{\max}$ Let t_l and t_h be the time durations the processor spends at the low and high speed levels, respectively, and let $\lambda = \beta \cdot S_L + \alpha \cdot S_H + (S_H - S_L) \cdot a$, where α, β, and a are constants associated with transition overhead as defined previously. Further, assume that the processor uses the DVFS control policy presented in Section 4.3. A speed schedule that maximizes the net work completed over L must have t_l^* that satisfies*

$$(S_H - S_L) \cdot (t_h - t_l^* \cdot t_h') - \lambda \cdot (1 + t_h') = 0, \tag{4.42}$$

where t_h is expressed as a function of t_l^* and $t_h' = \frac{\partial t_h}{\partial t_l^*}$.

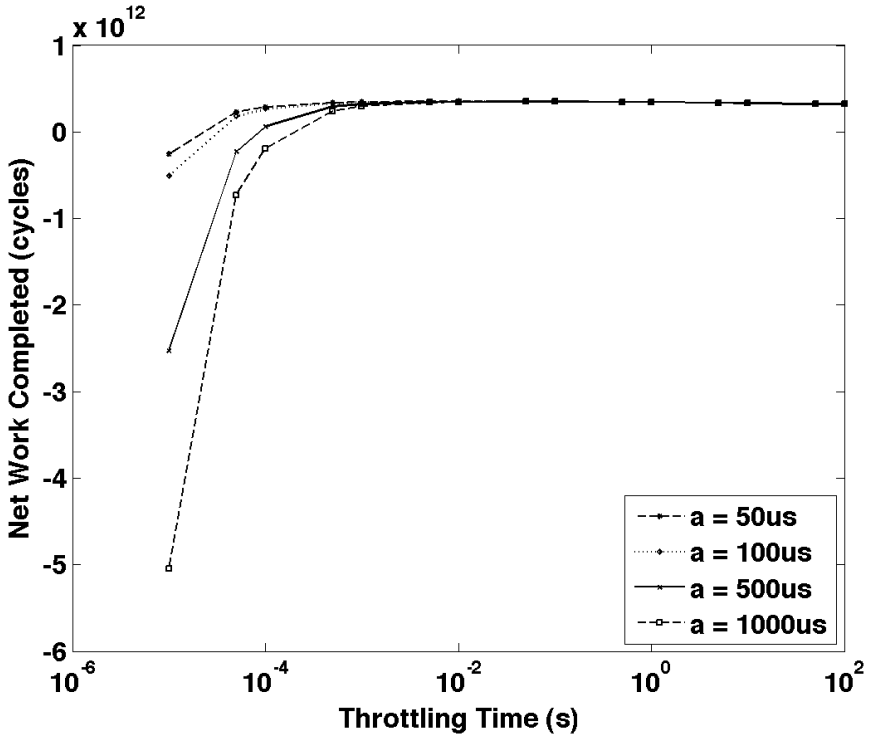

Figure 4.3. Net number of cycles completed as a function of throttling time, where a is the voltage changing time from the low speed level to the high speed level. With non-negligible transition overhead, infinitesimally small throttling time does not lead to the maximum amount of work completed.

Proof: Since $\lambda = \beta \cdot S_L + \alpha \cdot S_H + (S_H - S_L) \cdot a$, (4.41) becomes

$$W^* = \left[S_H - \left(\frac{(S_H - S_L) \cdot t_l^*}{t_l^* + t_h} + \frac{\lambda}{t_l^* + t_h} \right) \right] \cdot L. \tag{4.43}$$

The symbol λ denotes the number of cycles lost due to transition overhead provided that the processor transitions from high speed level to low speed level once and from low speed level to high speed level once. To find the maximum value of

W^*, we compute $\partial W^*/\partial t_l^*$, noting that t_h can be expressed as

$$t_h = \tau \, ln \left(\frac{\hat{T}(S_H) - \hat{T}(S_L) - (T_0 - \hat{T}(S_L))e^{-\frac{t_l^*}{\tau}}}{\hat{T}(S_H) - T_{\max}} \right). \tag{4.44}$$

The partial derivative of W^* with respect to t_l^* is

$$\frac{\partial W^*}{\partial t_l^*} = -\frac{L}{(t_l^* + t_h)^2} \left[(S_H - S_L)(t_h - t_l^* \cdot t_h') - (1 + t_h')\lambda \right]. \tag{4.45}$$

The maximum of W^* occurs when $(S_H - S_L) \cdot (t_h - t_l^* \cdot t_h') - O \cdot (1 + t_h') = 0$. Solving this equation for t_l^* yields an optimal throttling time.

□

As a result, we can use our previously proposed policy to maximize the work completed for processors with non-negligible transition overhead if the throttling time is found by solving (4.42) using a nonlinear equation solver.

4.5 Workloads with Different Power Consumptions

So far, we have assumed that processor power consumption is fixed over time for a given speed level. In reality, the required power consumption may depend on the details of operation and hardware, resulting in steady-state temperatures differing among workloads even for the same processor speed level. Since our DVFS control policy relies on the steady-state temperature of different speed levels to determine S_H and S_L, some modifications are needed. Consider a system that must execute different workloads with different power consumption over time. For the duration of a workload, which may consist of a number of applications, we can determine S_H and S_L as follows: $S_H = \min\{S | S^3 P_i R \geq T_{\max}\}$ and

$S_L = \max\{S|S^3 P_i R < T_{\max}\}$, where P_i is the maximum power consumption when executing workload WL_i. As long as the power consumption of the workload in a given time interval is known at run-time, e.g., by using performance counter based power models [53], our proposed DVFS control policy will maximize the work completed. The more complex case where tasks in a workload require different power consumptions is left as future work.

4.6 Simulation Results

In this section, we provide simulation results to demonstrate the effectiveness of our optimal clock throttling policy.

4.6.1 Simulation Setup

Using a Java simulator, we modeled our processor based on the Alpha 21264 processor, which consumes 120 W of power when running at the highest frequency of 4 GHz with the maximum temperature of 110 °C [87]. The silicon die and copper package have the dimensions of 16 mm × 16 mm × 0.5 mm and 24 mm × 24 mm × 2 mm, respectively. The threshold temperature is set to 90 °C. To compute the time constant for (4.1), we obtained temperature data via simulations in ISAC [112], which is a static and dynamic thermal analysis software package with improved time-domain solver for higher performance, using the default settings for all thermal-related parameters (e.g., heat capacity).

Since we did not have the data on the available voltage and frequency pairs of the Alpha processor, we assumed that it can switch to the same speed levels as the Intel Core Duo [47]. That is, we used the speed levels of the Intel Core Duo but calculated the corresponding power consumption and frequency for the

Alpha processor. For our system, the available speed levels are: 0.462, 0.615, 0.692, 0.769, 0.846, 0.923, and 1.

As our policy does not perform scheduling, it is insensitive to the type of applications that may be running. We used a periodic soft real-time system as an example application. Each simulation consisted of 100 randomly generated task sets of 20 tasks each for 30 different utilizations ($U = 0.05, 0.1, \cdots, 1.5$), for a total of 3,000 task sets. The utilization signifies how loaded the processor is; a utilization of 1 or greater means that the system is overloaded. Task periods ranged from 1 s to 10 s, with task execution times falling somewhere between 20% and 80% of the periods. For underloaded systems, a background job was also added. Each task set was simulated for a duration of 1,000 s.

For each run, we recorded the following data: number of cycles completed, number of deadline missed, average delays for jobs that missed their deadlines, and associated transition overhead, if applicable. Note that while we use a soft real-time system as an example here, metrics such as the number of cycles completed are relevant for general-purpose computing systems as well. In addition, while maximizing the work completed is not the same as maximizing the number of deadlines met, our results show that completing more work often leads to meeting more job deadlines.

4.6.2 Performance Comparison of Different Speed Selection Policies with Negligible Transition Overhead

We now discuss the simulation results for the different speed selection policies: (i) the naïve approach where the highest and lowest speed levels are both used (corresponding to the speeds of 1 and 0.462, respectively), (ii) the one speed

approach where the highest of the low speed levels is used the entire time (corresponding to the speed of 0.846), and (iii) the best approach where the lowest of the high and highest of the low speed levels are selected (corresponding to the speeds of 0.923 and 0.846, respectively). Note here that the naïve approach could represent the policy used by *Thermal Monitor 2* described in Section 4.2. We did not specifically compare with *Thermal Monitor 1* since it is non-configurable. The initial chip temperature is set to $T_{\max}$ since we are interested in the performance of the system once the processor starts to throttle. The throttling time used for this set of simulations is 10 s, which is similar to the default throttling time for the 2.13 GHz Pentium M-770 CPU [70].

The results are shown in Figures 4.4–4.6, which compare the average number of cycles completed, average number of deadline misses, and average delays for jobs that missed their deadlines, respectively. The average number of cycles completed is compared with the number of cycles completed when using the equilibrium speed described in Section 4.2. We can see that the proposed speed selection policy consistently outperforms the naïve policy in terms of all performance metrics. A comparison between the naïve and best speed selection policies reveals that our approach completes 47.65% on average and up to 67.99% more cycles than the naïve approach. Our policy also improves the number of cycles completed by the one speed policy by 1.60% on average and up to 3.29%. When compared to the number of cycles completed using the equilibrium speed, our approach deviates on average by only 2.76% and up to 2.93%. In addition, our policy reduces deadline misses by 59.38% on average and up to 100% compared to the naïve policy. Our policy also reduces deadline misses compared to the one speed approach by 3.65% on average and up to 45.74%. Note that the best policy would

yield more substantial performance improvements over the one-speed policy for systems with fewer speed levels and/or when the lowest of the high speed differs significantly from the equilibrium speed.

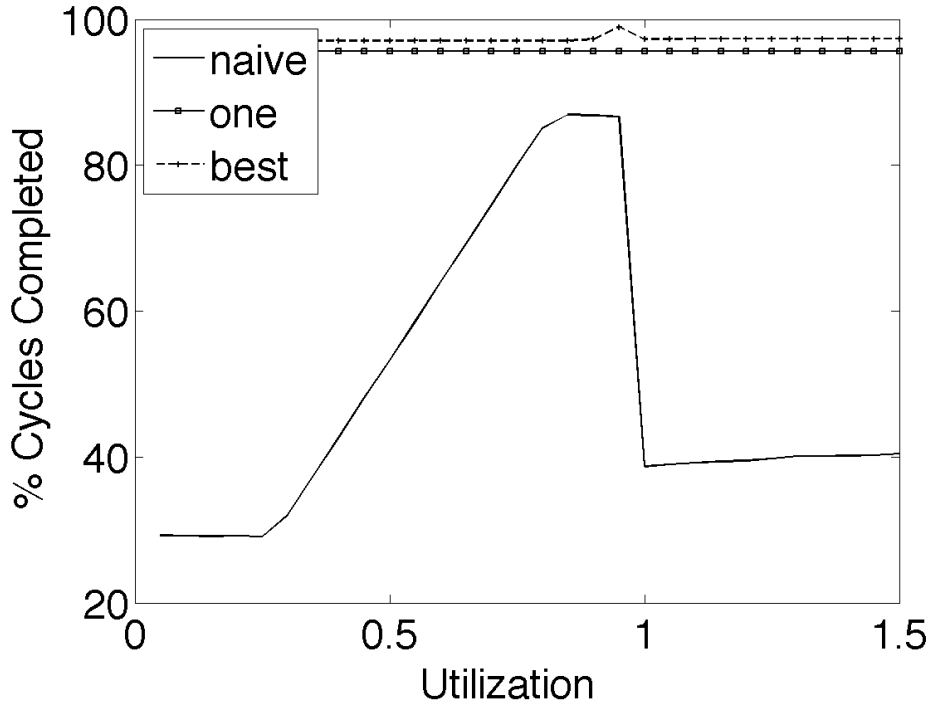

Figure 4.4. Average number of cycles completed (%) for different speed selection policies with throttling time of 10 s.

Figures 4.7–4.9 show the effect of different throttling times (i.e., the times the processor spends executing at the low speed level) on system performance when using our policy. We used the throttling times of 0.1 s, 0.5 s, 1 s, 5 s, and 10 s.

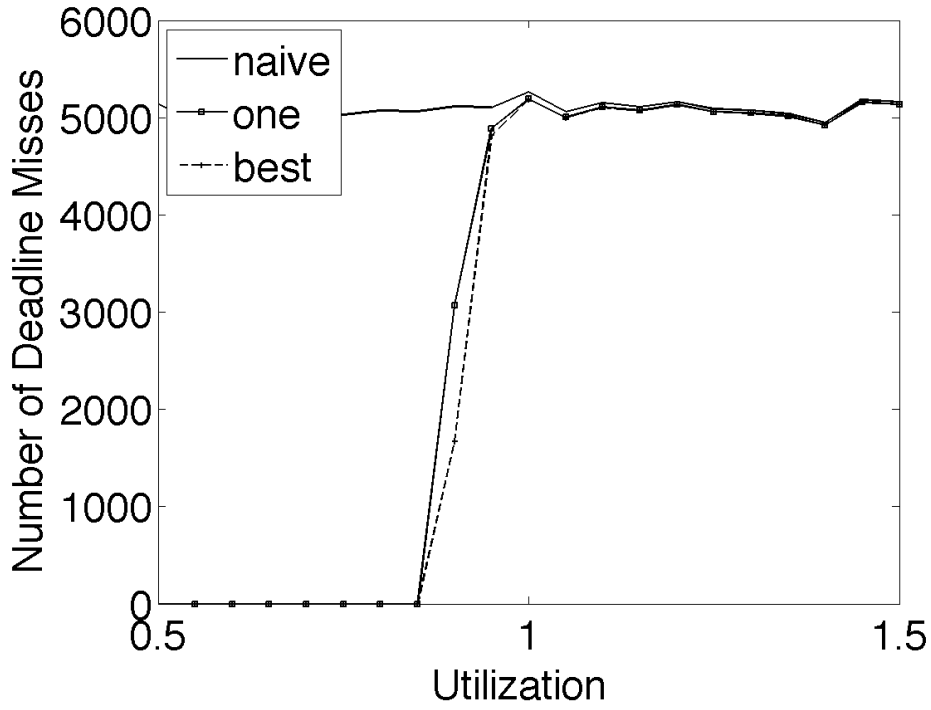

Figure 4.5. Average number of deadline misses for different speed selection policies with throttling time of 10 s.

Recall that the 2.13 GHz Pentium M-770 CPU uses a throttling time of about 10 s [70]. With a throttling time of 0.1 s, our DVFS control policy completes between 0.20% and 2.51% on average and up to between 0.22% and 2.68% more cycles than the throttling times of 0.5 s, 1 s, 5 s, and 10 s, respectively. When compared to the policy that uses the equilibrium speed, our policy completes 0.25% fewer cycles (using the throttling time of 0.1 s). Last but not least, the smallest throttling time (0.1 s) reduces deadline misses by between 8.14% and 9.42% on average and up to 100%.

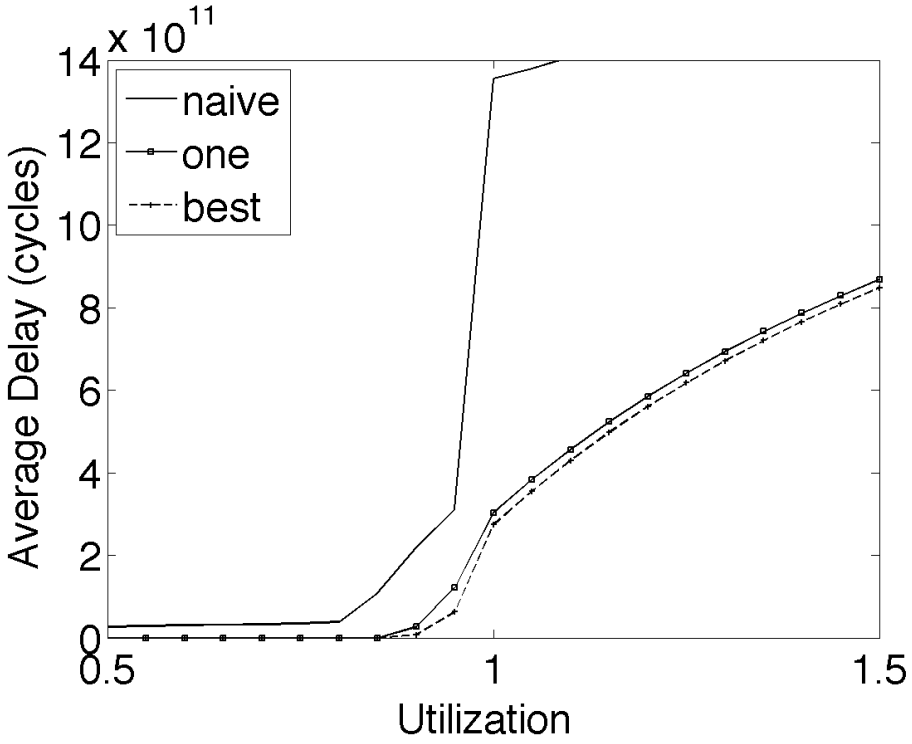

Figure 4.6. Average deadline miss delay (cycles) for different speed selection policies with throttling time of 10 s.

4.6.3 Performance Comparison of Different Speed Selection Policies with Non-Negligible Transition Overhead

As described in Section 4.4, the constants associated with transition overhead that we need to consider are α, β, and a, which we set to 10 us, 5 us, and 100 us, respectively. Since these constants are much smaller than the die time constants, it is reasonable to assume that the die temperature does not change during speed transitions.

Figure 4.10 plots the number of cycles completed with transition overhead considered as a function of the throttling times using our clock throttling policy.

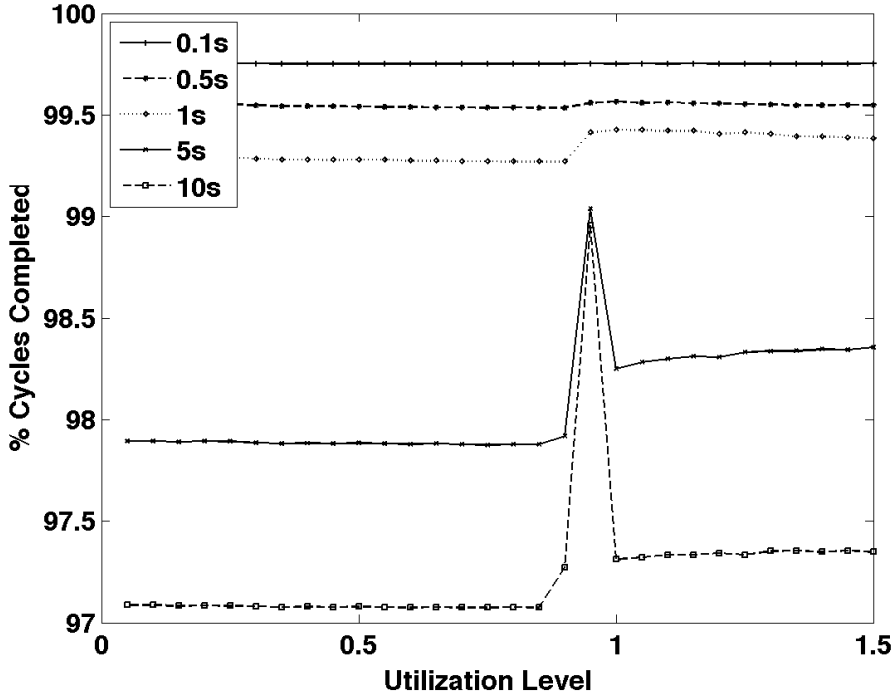

Figure 4.7. Average number of cycles completed (%) for different throttling times for the best speed selection policy.

By solving the equation in (4.42), we know that the optimal throttling time is 43.23 ms and this is confirmed by the results of this simulation. As expected, when transition overhead are non-negligible, switching from the low to high speed levels more often than optimal decreases the rate of computation due to the increasing proportion of time spend idle during transitions. On the other hand, switching very infrequently can reduce the computation rate because a lower percentage of time can be spent at the higher speed level.

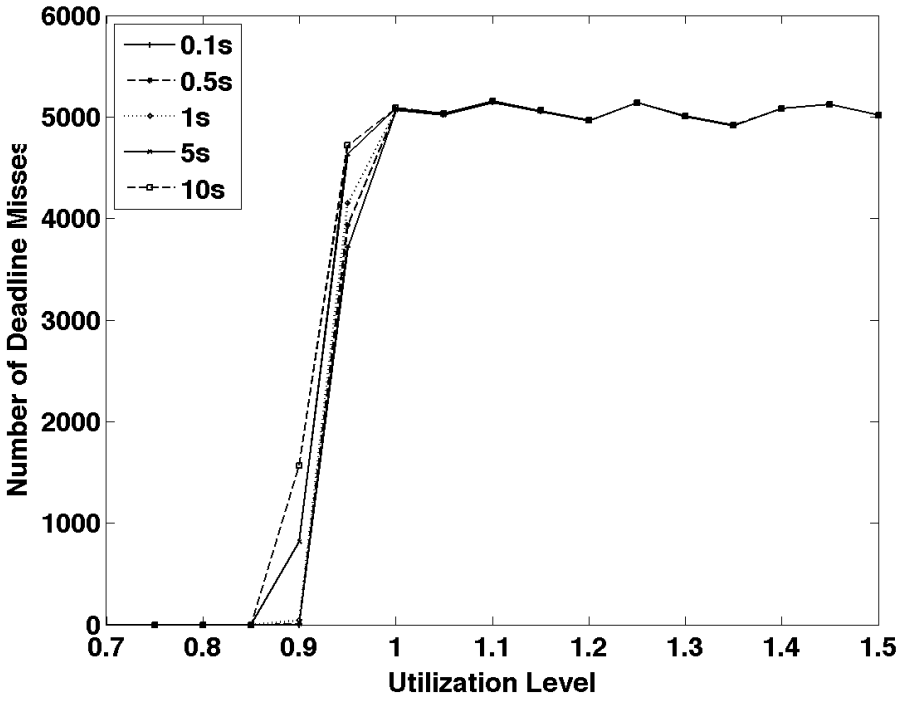

Figure 4.8. Average number of deadline misses for different throttling times for the best speed selection policy.

4.7 Summary

We proposed an optimal online DVFS control policy to maximize instruction cycles subject to a peak temperature constraint. Our solution is applicable to any processor with discrete speed levels and non-negligible transition overhead. Our policy completed 47.7% on average and up to 68.0% more cycles when compared to the naïve policy.

We plan on extending our proposed policy to consider the time interval before the system reaches the threshold temperature for the first time. To handle situations where external factors such as ambient temperature dynamically changes

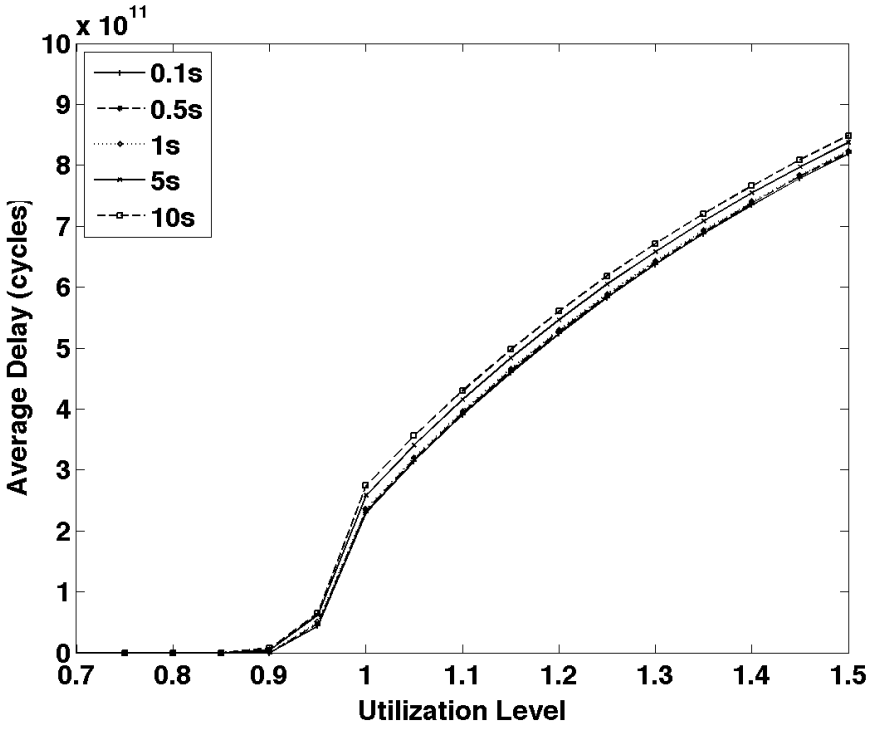

Figure 4.9. Average deadline miss delay (cycles) for different throttling times for the best speed selection policy.

and when tasks have different power consumption, modifications to the proposed policy are needed. Finally, we would like to extend our policy to consider multi-processor architectures.

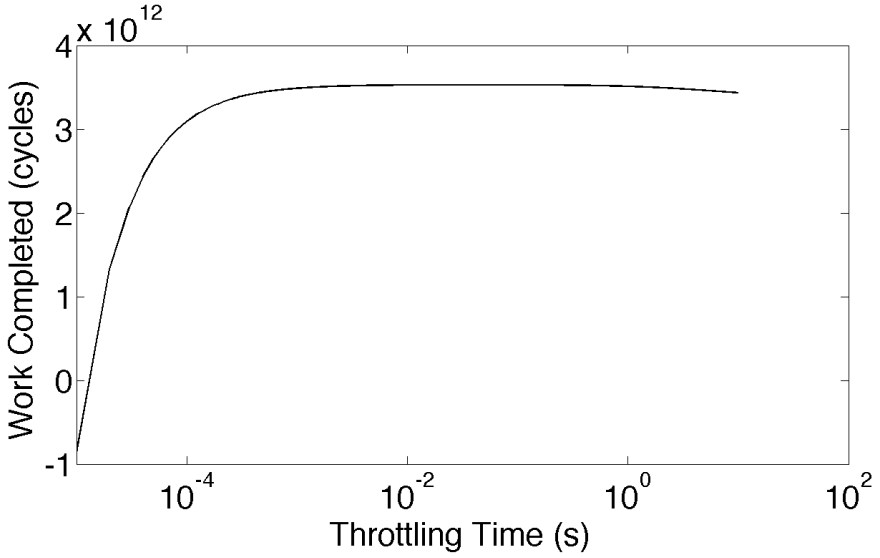

Figure 4.10. Average number of cycles completed as a function of throttling time when transition overhead is considered. A very small throttling time can result in a negative net number of cycles completed.

CHAPTER 5

TEMPERATURE-AWARE SCHEDULING AND ASSIGNMENT FOR HARD REAL-TIME APPLICATIONS ON MPSOCS

Increasing integrated circuit (IC) power densities and temperatures may hamper multiprocessor system-on-chip (MPSoC) use in hard real-time systems. This chapter formalizes the temperature-aware real-time MPSoC assignment and scheduling problem and presents an optimal phased steady-state mixed integer linear programming based solution that considers the impact of scheduling and assignment decisions on MPSoC thermal profiles to directly minimize the chip peak temperature. We also introduce a flexible heuristic framework for task assignment and scheduling that permits system designers to trade off accuracy against running time when solving large problem instances. Finally, for task sets with sufficient slack, we show that inserting idle times between task executions can further reduce the peak temperature of the MPSoC quite significantly.

5.1 Introduction

We begin by providing an in-depth overview of the problem under consideration, review existing work, present our main contributions, and provide the organization of this chapter.

5.1.1 Problem Overview

Multiprocessor systems-on-chips (MPSoCs) are now widely used in application-specific systems and high-performance computing. They offer performance, power consumption, and implementation complexity advantages over highly superscalar uniprocessor architectures. Their use, and scale, will increase dramatically in the coming years. According to Milchman [73], 16-core processors will be common within the next four years. Intel plans to deliver processors that have dozens or hundreds of cores during the next decade [15]. The use of heterogeneous MPSoCs can sometimes dramatically improve performance and power consumption relative to homogenous MPSoCs [57]. However, it can also increase complexity. It is likely that some future MPSoCs will be homogeneous and some will be heterogeneous. With the current use of MPSoCs in soft real-time applications such as gaming [110], it is expected that many hard real-time applications will soon be implemented using MPSoCs. In fact, FreeScale is now offering the QorIQ Embedded Multicore Processor [80] that is intended for application domains that require real-time computing, e.g., aerospace applications.

MPSoC temperature is a strong function of power density. Increasing transistor counts and aggressive frequency scaling result in a significant increase in chip power density and temperature. Increasing chip temperature has significant impact on other design metrics including reliability, performance, and cost, as microprocessor failure rate depends exponentially upon operating temperature [99]. A 10–15 °C difference in operating temperature can result in a 2× difference in the lifespan of a device [102]. Temperature also affects speed; reduction of charge carrier mobility in transistors and increased interconnect latency resulting from high temperature degrade performance, requiring reduced clock frequencies or,

worse yet, resulting in run-time failures.

Increasing power densities make package and cooling design for the worst-case scenario prohibitively expensive, since the cost of cooling solutions increases super-linearly with power consumption [43]. It is therefore necessary to design packaging and cooling solutions based on less than worst-case thermal profiles and compensate by preventing, hopefully rare, dangerous thermal scenarios at run-time. Most popular approaches react to critical temperatures by reducing frequency and voltage (i.e., performing hardware throttling), or by temporarily preventing instruction issue to reduce the power consumption, and hence temperature, of the processor [63].

Since the execution times of real-time tasks, and hence total system utilization, tend to vary significantly due to factors such as conditional branches and system inputs [116], real-time applications can exhibit great temperature variation at run-time. When the system utilization is low, the MPSoC may not have a high temperature problem, thanks to the amount of slack available in the system. On the other hand, a system with high utilization can push an MPSoC to its thermal limit [28, 106, 111]. In the worst case, the host MPSoC may lack run-time thermal management, leading to overheating and signal timing violations or permanent failure. More subtly, even those real-time systems containing MPSoCs that support run-time thermal management may fail when a temperature bound is reached, but for a different reason.

Most run-time thermal management techniques use thermal sensors to detect when the maximum safe temperature is approached and react by decreasing processor power consumption, e.g., by decreasing frequency or stalling instruction issue. These techniques share a common weakness: they decrease performance. If

a real-time task running on an MPSoC with run-time thermal management ever triggers throttling when there is little timing slack, the real-time task will miss its deadline. Missing hard real-time deadlines is unacceptable; an example would be failure to stop an automatically controlled train on time [67]. To guarantee hard real-time performance, designers should consider thermal effects by explicitly optimizing peak temperature while meeting all functionality and real-time deadlines. This motivates our work on the temperature-aware real-time MPSoC assignment and scheduling problem.

Existing power-aware techniques, such as energy minimization, peak power minimization, as well as global dynamic voltage and frequency scaling (DVFS), cannot solve the temperature problem in MPSoCs because they do not consider spatial thermal variation; heat generated by an active core also affects other neighboring thermal elements, be they other cores or portions of the heatsink. The net heat flow from one thermal element to another depends on the conductance parameters and the current temperatures of these thermal elements. Ignoring spatial thermal variation can lead to unnecessarily high peak temperatures, especially for high power density chips.

5.1.2 Related Work

Researchers have only recently started work on temperature-aware high-level synthesis [76] and design space exploration [64]. The objective is usually to optimize system performance subject to a peak temperature constraint. For uniprocessor architectures, Wang and Bettati presented a reactive two-speed policy to control peak temperature [106]. To guarantee real-time deadlines, a proactive thermal management policy was later proposed [28]. Rao et al. presented an

optimal processor speed control policy to maximize the work completed under a temperature constraint [89]. The thermal model was later improved by the same authors [87]. Mutapcic et al. focused on energy minimization under thermal and task constraints [78]. Quan et al. presented a necessary and sufficient condition for schedulability as well as a novel scheduling algorithm for real-time applications running on processors with a temperature constraint [85]. A temperature-constrained optimization technique is valuable in maximizing application performance, but it does not necessarily increase system reliability nor guarantee hard real-time deadlines. The lower the operating temperature, the better the system reliability.

Unfortunately, there is little research that targets peak temperature control directly. Bansal et al. were among the first to study the problem of peak temperature minimization using continuous dynamic speed scaling for uniprocessors running independent tasks [8]. Jayaseelan and Mitra presented a task sequencing technique to minimize the peak temperature for periodic real-time tasks running on a single processor [48]. Neither work considers MPSoCs nor task dependencies.

The problem of assigning and scheduling real-time tasks in systems containing multiprocessors and MPSoCs has received significant research attention. Some papers focus on meeting hard real-time constraints [42] while others aim to optimize energy consumption in the presence of timing constraints [91]. Since most real-time scheduling problem variants are $\mathcal{NP}$-hard, many heuristics have been proposed to solve large problem instances with different optimality criterion [97]. Once again, the focus is usually placed on meeting the thermal constraint instead of minimizing peak temperature. For example, Rao et al. presented a method to maximize throughput by determining speeds of different cores subject to a peak

temperature constraint [88]. Mulas et al. proposed a task migration algorithm that balances the loads on different cores to reduce hotspots [77]. Coskun et al. used online learning [31] and integer linear program (ILP) [32] to reduce the frequency of peak temperature constraint violations. Jung et al. used dynamic thermal management (DTM) to minimize energy while meeting a peak temperature constraint [52]. An approximation algorithm for minimizing the peak temperature of ideal processors was proposed by Chen et al. for real-time tasks with no precedence constraints [26]. In addition, a temperature-aware task assignment and voltage selection algorithm was proposed by Sun et al. for three-dimensional stacked-wafer MPSoCs [100]. However, this solution cannot be used to solve our problem since only homogeneous cores are considered and lateral thermal variation is ignored (the peak temperature of 3-D MPSoCs is strongly influenced by vertical inter-core heat flow).

Xie and Hung were the first to propose a collection of heuristics for temperature-aware processor allocation, task assignment, and scheduling [111]. However, their heuristics consider spatial or temporal thermal variation, but not both types of variations. In Section 5.5, we show that their technique can deviate significantly from optimality.

Finally, Paci et al. claim that temperature-aware design is unnecessary in low-power embedded systems [81]. While their conclusions hold for very low-power embedded processors because on-die temperature variation is small, our results show substantial ($> 30\,^{\circ}\mathrm{C}$ improvement) benefits from temperature-aware design for MPSoCs containing several embedded processor cores (see Section 5.7.1 for more details), using the thermal model in Section 5.2.2.

5.1.3 Contributions

This chapter makes the following main contributions. We present a mixed-integer linear programming (MILP) formulation for assigning and scheduling tasks with hard real-time constraints on an MPSoC to minimize the chip peak temperature. Our formulation considers spatial and temporal thermal variations. It relies on a *phased steady-state* thermal analysis directly integrated within the MILP formulation. This analysis produces a separate steady-state thermal profile for each power profile occurring during the schedule. Extensions for temperature-dependent leakage power modeling, DVFS, finer-grained thermal modeling, and inter-task communication modeling are given.

To solve problem instances that are large or for which the effects of heat capacitance are significant, we propose a heuristic task assignment and scheduling framework in which the actual method for computing the thermal profile can be selected as appropriate. Specifically, phased steady-state thermal analysis is used when task durations are long relative to the time constants of the cores. *Transient thermal analysis*, in which temperatures are time-dependent, is used otherwise.

To exploit slack in the system where the effects of heat capacity are significant, we use the concept of delay (i.e., idle time) insertion in our transient analysis based heuristic to further reduce the chip peak temperature while guaranteeing hard real-time deadlines.

5.1.4 Organization

The chapter is organized as follows. In Section 5.2, we introduce our system model, state our assumptions, and formally define the problem. We motivate the need for a temperature-aware assignment and scheduling algorithm in Section 5.3.

We describe our formal approach in Section 5.4 and present our flexible heuristic framework in Section 5.5. We introduce and incorporate the concept of delay insertion into our heuristic framework in Section 5.6. The benefits and efficiency of our approach are experimentally determined in Section 5.7. Section 5.8 concludes the chapter.

5.2 System Model and Problem Definition

The system model and the temperature-aware real-time MPSoC assignment and scheduling problem are now described.

5.2.1 Task Model

In our model, J represents the set of hard real-time tasks to be executed. For each task $j \in J$, the worst-case execution time when running on core m is denoted by $E(j, m)$, the deadline by $D(j)$, and the release time by $R(j)$. Note that $R(j) = 0$ and $D(j) = \infty$ if no release time and deadline constraints are associated with task j. A directed acyclic graph (DAG) is used to capture data dependencies among tasks. In a DAG, nodes represent tasks and directed edges indicate data dependencies between pairs of tasks. Let Γ_{j_1,j_2} denotes the dependency between tasks j_1 and j_2 where

$$\Gamma_{j_1,j_2} = \begin{cases} 1 & \text{if task } j_1 \text{ immediately precedes task } j_2 \\ 0 & \text{otherwise.} \end{cases} \tag{5.1}$$

A task j may execute only after all its predecessor tasks have completed and j has been released, i.e., the current time is greater than or equal to $R(j)$. For now, we assume that there is no cost for communication among dependent tasks. This

assumption will be relaxed in Section 5.4.5. For periodic systems, we guarantee schedule validity by scheduling out to the hyperperiod of all tasks [60]. The *hyperperiod* is the least common multiple of the periods of all tasks in the problem specification.

5.2.2 Thermal Model

We model an MPSoC with a set of cores, M. For each core $m \in M$, its width, height, and location are specified. Based on the floorplan, the set of neighbors of core m, N_m, thermal conductance to a neighbor n, $G_n(m, n)$, and thermal conductance to the heatsink element above it, $G_h(m)$, can be calculated. For each task and core combination, $P(j, m)$ indicates the power consumption of core m when executing task j. We discuss an extension to this power model to account for leakage power in Section 5.4.2.

Thermal analysis estimates heat transfer through heterogeneous materials among heat producers (e.g., transistors) and heat consumers (e.g., heatsinks attached to an MPSoC). In the task assignment and scheduling phase, we will adopt a coarse-grained discrete heat flow model analogous to widely used compact models [98] to balance thermal analysis efficiency and accuracy. However, the algorithm framework proposed in Section 5.5 can be used with any thermal analysis technique.

In our thermal model, which is based on the classical Fourier heat flow model, each core corresponds to a discrete thermal element; Section 5.4.4 discusses how our approach can be modified to support finer-grained thermal element modeling. The heatsink on top of the cores is modeled using multiple thermal elements and its partitioning corresponds to the layout of the cores. Since the heatsink is usually larger than the processor itself, we model heatsink overhang using ad-

ditional thermal elements; the heatsink overhangs the chip by 25% of its length and width. The interface layer is included within the heatsink instead of being modeled explicitly. The interface material is usually very thin so lateral heat flow within it can be neglected. Lateral heat flow between cores and heatsink elements is modeled.

To perform thermal analysis, we take advantage of the well-known duality between electrical and thermal circuits. The temperature of each thermal element can be expressed as a function of its power consumption, the ambient temperature, and the temperatures of neighboring thermal elements. Figure 5.1 depicts the circuit representation of this model. Here, T_A denotes the ambient temperature, $G_A(h)$ is the conductance from the heatsink element h to the ambient, and $G_{nh}(h, g)$ is the conductance between heatsink elements h and g. The current source P_m denotes the power consumption of core m. The terms $G_h(m)$ and $G_n(m, n)$ were as defined previously.

The temperature of core m at time t, $T(t, m)$, can be determined using the node thermal analysis of the circuit in Figure 5.1:

$$\begin{aligned} 0 = & \sum_{n \in N_m} (T(t,m) - T(t,n)) \cdot G_n(m,n) + C(m) \cdot \frac{dT(t,m)}{dt} \\ & + (T(t,m) - T(t,h)) \cdot G_h(m) - \sum_{j \in J} \alpha(t,j,m) \cdot P(j,m), \end{aligned} \tag{5.2}$$

$$\begin{aligned} 0 = & \sum_{g \in N_h} (T(t,h) - T(t,g)) \cdot G_{nh}(h,g) + C(h) \cdot \frac{dT(t,h)}{dt} \\ & + (T(t,h) - T(t,m)) \cdot G_h(m) + (T(t,h) - T_A) \cdot G_A(h), \end{aligned} \tag{5.3}$$

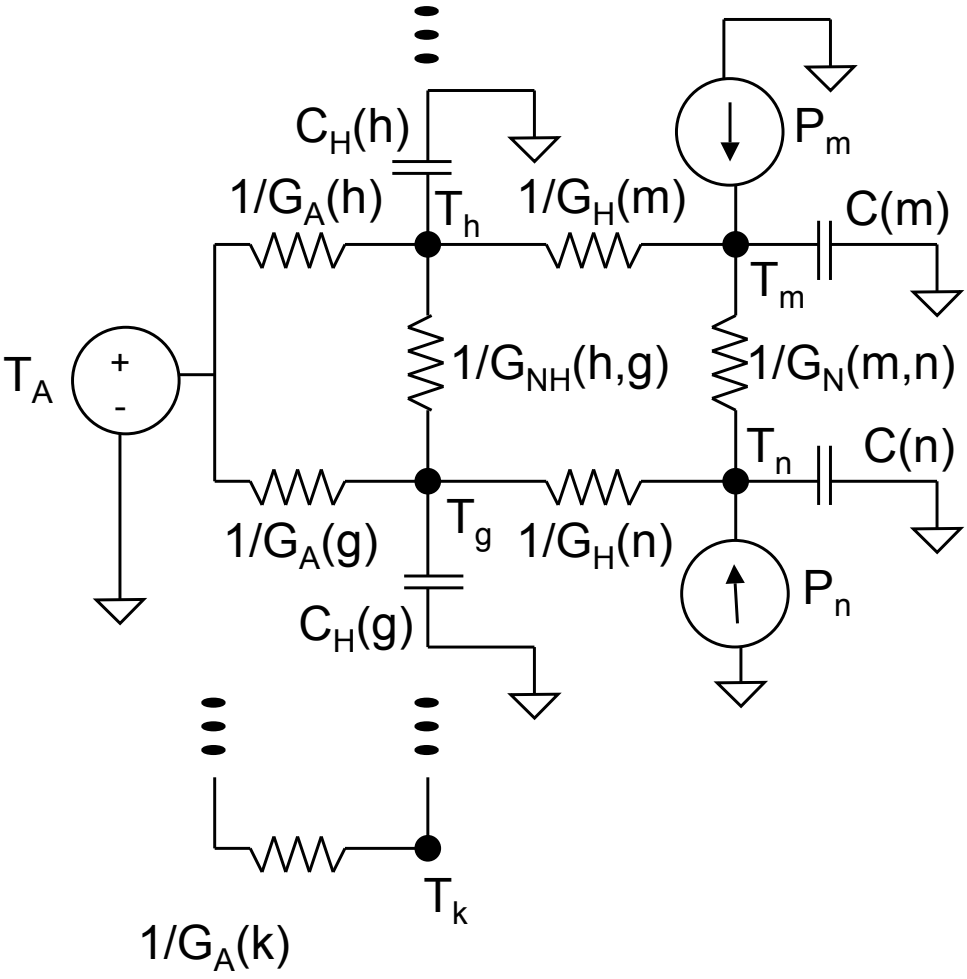

Figure 5.1. Equivalent circuit diagram of the thermal model. In this example, core m is connected to its neighboring core n and is under heatsink element h. The power consumption and capacitance of core m are represented by the current source P_m and the capacitor $C(m)$, respectively. The heatsink element h is connected to other neighboring heatsink elements, a capacitor, as well as to the ambient, which is denoted by the voltage source T_A.

where $T(t,h)$ is the temperature of the heatsink element h directly above core m at time t, $C(m)$ is the thermal capacitance of core m, $C(h)$ is the thermal capacitance of heatsink element h, and $\alpha(t,j,m) = 1$ if task j is active on core m at time t. In the above two equations, if the heatsink element h is a heatsink overhang element, then the term $(T_h - T_m) \cdot G_H(m)$ is set to 0.

The thermal conductance of core m to the heatsink element h directly above it, $G_h(m)$, can be computed as described by Serway [92]:

$$G_h(m) = \frac{Area_m}{R_{chip} \cdot Area_{chip}}, \tag{5.4}$$

where $Area_m$ denotes the area of core m, $Area_{chip}$ represents the area of the chip, and $R_{chip} = \frac{th_{si}}{K_{si} \cdot Area_{chip}}$, th_{si} is the thickness of silicon, and K_{si} denotes its thermal conductivity. In our experiments, we set th_{si} and K_{si} to be 0.6 mm and 148 W/mK, respectively.

The conductance of a heatsink element h to the ambient can be calculated in a similar manner. That is, we substitute $Area_m$ and $Area_{chip}$ in (5.4) by the area of the heatsink element under consideration and the area of the entire heatsink, respectively. In addition, we replace R_{chip} with R_{HS} in (5.4), where $R_{HS} = \frac{T_{active} - T_{ambient}}{P_{chip}} - R_{chip}$, P_{chip} being the total power consumption of the chip and T_{active} the average active layer temperature when all cores are busy and $T_{ambient}$ being the ambient temperature. We set T_{active} and $T_{ambient}$ to 90 °C and 45 °C, respectively.

We compute the conductance between core m and its neighbor core n as follows:

$$G_n(m,n) = \frac{w_{mn} \cdot th_{si} \cdot K_{si}}{L_{mn}}, \tag{5.5}$$

where w_{mn} is the length of intersection between cores m and n and L_{mn} is the distance between the midpoint of m and that of n. The lateral conductance between two heatsink elements can be computed in a similar fashion. We assume that the heatsink is made of copper, with a thickness of 1 mm and thermal conductivity of 400 W/mK.

5.2.3 Problem Definition

Given the floorplan of a chip containing a set of cores, M, and a set of hard real-time tasks, J, as described above, determine a static assignment of tasks to cores and a static, non-preemptive schedule of tasks on the cores such that all precedence constraints and real-time deadlines are met and the chip peak temperature, T_{max}, is minimized.

5.3 Motivations

Since average power (i.e., energy) for a fixed duration and peak power are related to chip temperature, it is natural to question whether optimizing peak temperature can produce significantly different results than optimizing peak power or average power. Let us consider a task set containing two identical tasks, j_1 and j_2, each with a deadline of 5 ms. For this example, the MPSoC is arranged as shown in Figure 5.2 (core sizes are not necessarily drawn to scale in the diagram). Task execution times (E) and associated power consumptions (P) are shown near the respective cores. To minimize energy, tasks j_1 and j_2 are both assigned to core $m2$. The resultant chip peak temperature is 65.30 °C. If our objective were to minimize peak power, then task j_1 would be assigned to core $m2$ and task j_2 to core $m1$, also resulting in a peak temperature of 65.30 °C. However, if task j_1

were executed on core $m4$ and task j_2 on core $m1$, the peak temperature would be reduced to 65.16 °C, which is about 0.14 °C cooler. This difference is the first point in the plot in Figure 5.3.

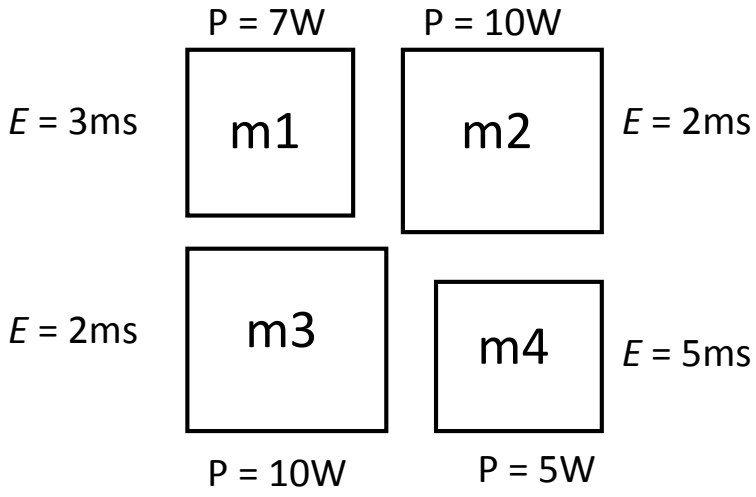

Figure 5.2. Floorplan for the motivating example in Section 5.3, where both task execution times (E) and power consumption (P) are shown for each core.

While the improvement in this case is small, the power density of the chip in the above example is only 0.19 W/mm^2. The power density can be as high as 0.79 W/mm^2 for 90 nm processors, 2.02 W/mm^2 for 65 nm processors, and 7.24 W/mm^2 for 45 nm processors [65]. To obtain similar chip power densities, we repeated the previous experiment but increased each core power consumption by

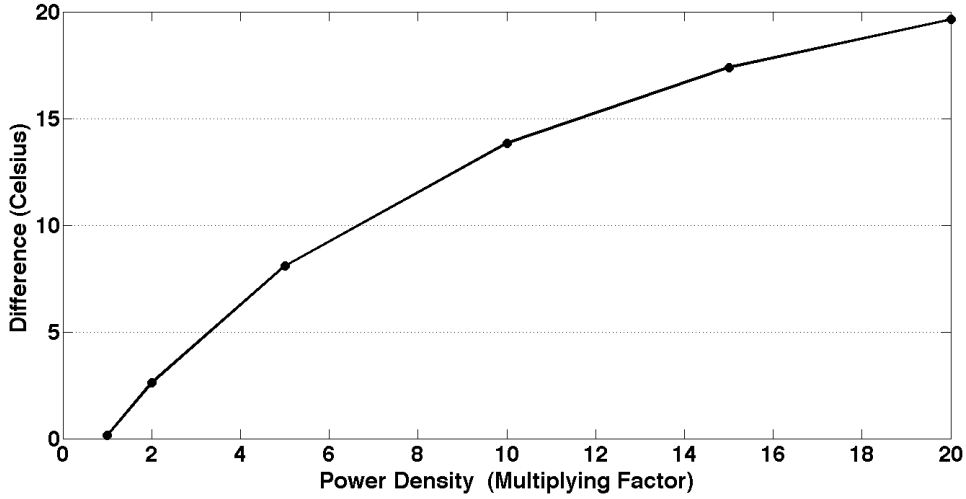

Figure 5.3. Differences in peak temperatures between energy minimization and peak temperature approaches. The x-axis shows the values (i.e., multiplying factors) that were multiplied to the original chip power density. In this example, peak power minimization yields the same peak temperature as energy minimization.

factors of 2, 5, 10, 15, and 20. The resulting chip power densities are $0.39\,\mathrm{W/mm^2}$, $0.97\,\mathrm{W/mm^2}$, $1.95\,\mathrm{W/mm^2}$, $2.92\,\mathrm{W/mm^2}$, and $3.89\,\mathrm{W/mm^2}$. In each case, the heatsink conductance to the ambient is adjusted to model the improved cooling solutions necessary to maintain an average active layer temperature of $90\,^{\circ}\mathrm{C}$.

Although task assignments and schedules are the same as before, chip peak temperatures increase when the higher power density cores are used. Figure 5.3 shows the reductions in chip peak temperatures when the peak temperature is optimized instead of peak power or energy for the example in Figure 5.2 and the chip power densities mentioned above. The x-axis shows the chip power density. The y-axis shows the difference between the peak temperature obtained from energy or peak power minimization and that from peak temperature minimization. As can

be seen from the plot, the advantages of minimizing the chip peak temperature increase with increasing chip power density, resulting in up to 20 °C reduction in peak temperature for this example.

Energy and peak power minimization suffer from the same weakness: neither considers spatial thermal effects. In fact, energy minimization ignores both temporal and spatial thermal variation while peak power minimization considers temporal thermal variation but ignore spatial thermal variation. The peak temperature of an MPSoC is increased by crowding the same amount of energy consumption into less time and space. Hence, to minimize the chip peak temperature, tasks should be assigned and scheduled in careful consideration of thermal interaction with neighboring cores. In addition, our example indicates that although there are many cases in which minimizing peak power produces different (and potentially better) results than minimizing energy, the same results are produced for some problem instances.

5.4 MILP-based Approach

In this section, we present our approach to solving the problem defined in Section 5.2.3. We also describe how our model can be extended to account for leakage power, dynamic voltage and frequency scaling (DVFS), and inter-task communication, or adjusted to use a finer-grained thermal model. Limitations of the MILP-based approach are described at the end of the section.

5.4.1 MILP Formulation

We now present our MILP formulation for the problem defined in Section 5.2.3. We define the following variables.

$$\delta(j,m) = \begin{cases} 1 & \text{if task } j \text{ is assigned to core } m \\ 0 & \text{otherwise.} \end{cases} \tag{5.6}$$

$$\eta(j_1,j_2) = \begin{cases} 1 & \text{if task } j_1 \text{ starts before task } j_2 \\ 0 & \text{otherwise.} \end{cases} \tag{5.7}$$

$$\beta(j_1,j_2,m) = \begin{cases} 1 & \text{if task } j_2 \text{ executes on core } m, \text{ precedes}^1, \\ & \text{and overlaps with task } j_1 \\ 0 & \text{otherwise.} \end{cases} \tag{5.8}$$

For the sake of consistency, we let $\beta(j,j,m) \equiv \delta(j,m)$. The $\beta(j_1,j_2,m)$ variables capture the overlapping execution of different tasks and play a key role in computing the peak temperature. They are also useful in computing the peak power, as will be shown later. We use $ts(j)$ and $tf(j)$ to denote the start and finish time of task j, respectively, yielding

$$tf(j) \equiv ts(j) + \sum_{m \in M} \delta(j,m) \cdot E(j,m) \tag{5.9}$$

$$\equiv ts(j) + et(j). \tag{5.10}$$

MILP formulations have long been proposed for modeling the task assignment and scheduling problem in a heterogeneous multiprocessor environment [61]. How-

[1]Precedence is not necessary but is sufficient and simplifies the test. Also, j_1 does not execute on core m; its execution overlaps in time with the execution of j_2.

ever, energy minimization has often been the main objective. Such solutions ignore both temporal and spatial thermal variation. Even peak power minimization only considers temporal thermal variation. To take both types of thermal variation into account, we directly minimize the chip peak temperature, $T_{\max}$, which is the highest temperature at any position on the chip during a schedule of duration SL, i.e.,

$$T_{\max} = \max_{m \in M, t \in [0, SL]} \tau_m(t), \tag{5.11}$$

Using (5.2) and (5.3) to compute the temperature at each node at any given time corresponds to dynamic or transient thermal analysis. Unfortunately, transient thermal analysis is computationally expensive. This makes its use in the MILP formulation impractical; the MILP solver would only be able to handle very small problem instances, thereby making it difficult to validate a heuristic. For this reason, we set the capacitance values in (5.2) and (5.3) to zero to obtain the steady-state temperature at each node when predicting temperatures in our MILP formulation. In Section 5.4.6, we indicate the situations in which the MILP-based approach with steady-state analysis is appropriate and inappropriate. In addition, Section 5.5 presents a solution to the more general problem of dynamic temperature optimization.

From the thermal model in Section 5.2.2, it might appear necessary to compute the steady-state temperature, $\tau_m(t)$, of a core m at every time instant t to determine $T_{\max}$. Even if we discretize the time duration SL, this approach may still be too costly; task execution times can vary dramatically, resulting in some tasks executing for hundreds of thousands or millions of time units. To overcome this difficulty, we make the following observations: (1) core power consumptions only change at the beginning or end of a task execution, and (2) the steady-state

temperature of a core only experiences a rapid change when the power consumption of at least one core on the chip changes. Hence, we can significantly reduce the amount of computation needed to obtain $T_{\max}$. Specifically, we only evaluate the temperature of each core m immediately after every task i starts or finishes executing on any core in the MPSoC and denote this temperature with $T(i,m)$. Consequently, the objective function of the MILP can be expressed as

$$\min T_{\max}, \text{ where } T_{\max} \geq T(i,m), \forall m \in M, \forall i \in J. \tag{5.12}$$

$T(i,m)$ satisfies the constraints given in (5.2) and (5.3), which are rewritten in (5.13) and (5.14), respectively.

$$\begin{aligned} T(i,m) \equiv T_{HS}(i,h) + \frac{1}{G_H(m)} \left[\sum_{j \in J} \beta(i,j,m) \cdot P(j,m) \right] \\ + \frac{1}{G_H(m)} \sum_{n \in N_m} G_N(m,n) \cdot [T(i,n) - T(i,m)] \end{aligned} \tag{5.13}$$

$$\begin{aligned} 0 = (T_{HS}(i,h) - T(i,m)) \cdot G_H(m) + (T_{HS}(i,h) - T_A) \\ \cdot G_A(h) + \sum_{g \in N_h} (T_{HS}(i,h) - T_{HS}(i,g)) \cdot G_{NH}(h,g). \end{aligned} \tag{5.14}$$

Note that (5.13) is only linear if we can treat $P(j,m)$ as a constant given task j and core m. For now, we assume that this is the case. We will discuss the more general case where $P(j,m)$ is not a constant in Section 5.4.2. Also, it should be clear that (5.14) can be rewritten and expressed as a function of T_{HS}.

The following constraints are used to guarantee schedulability.

1. Every task j is assigned to exactly one core m:

$$\forall j \in J \qquad \sum_{m \in M} \delta(j, m) = 1. \tag{5.15}$$

2. Every task j meets its deadline:

$$\forall j \in J \qquad ts(j) + et(j) \leq d(j). \tag{5.16}$$

3. Precedence constraints are honored:

$$\forall i, j \in J \qquad ts(j) \geq tf(i) \cdot \Gamma_{i,j}. \tag{5.17}$$

4. All tasks execute for their durations without overlap on any given core:

$$\forall j_1, j_2 \in J$$

$$1 \leq \eta(j_1, j_2) + \eta(j_2, j_1), \tag{5.18}$$

$$ts(j_1) \leq ts(j_2) + (1 - \eta(j_1, j_2)) \cdot \Lambda, \tag{5.19}$$

$$ts(j_2) \leq ts(j_1) + \eta(j_1, j_2) \cdot \Lambda, \tag{5.20}$$

$$\forall j_1, j_2 \in J, j_1 \neq j_2, \forall m \in M$$

$$tf(j_1) \leq (2 - \delta(j_1, m) - \delta(j_2, m)) \cdot \Lambda ts(j_2) + \Lambda \cdot (1 - \eta(j_1, j_2)), \tag{5.21}$$

$$tf(j_2) \leq (2 - \delta(j_1, m) - \delta(j_2, m)) \cdot \Lambda + ts(j_1) + \Lambda \cdot \eta(j_1, j_2), \tag{5.22}$$

where Λ is a constant greater than or equal to the largest deadline in the task set. Constraint (5.19) states that task j_1 must start before task j_2 if $\eta(j_1, j_2) = 1$. Constraint (5.21) guarantees that task j_1 finishes before task j_2 starts if tasks j_1

and j_2 are executed on the same processor and task j_1 precedes task j_2. Similar conditions hold for (5.20) and (5.22).

Consider a situation where tasks i and j execute on cores m and n, respectively. Further, task i precedes task j and their executions overlap. At the start of task i, we only need to consider the power consumption of core m. However, at the start of task j, we must take into account the power consumptions of both cores to correctly compute the chip peak temperature. For this reason, we must ensure that $\beta(j_1, j_2, m) = 1$ only when $\delta(j_2, m) = 1$ and $ts(j_2) \leq ts(j_1) \leq tf(j_2) - \epsilon$, where ϵ is a small constant used to prevent imprecise floating point computations from making it appear as if contiguous tasks overlap in time. Therefore,

$$\begin{aligned}
&\forall m \in M, \forall j_1, j_2 \in J, j_1 \neq j_2 \\
&\qquad tf(j_2) \geq ts(j_1) + (\beta(j_1, j_2, m) - 1) \cdot \Lambda, && (5.23) \\
&\qquad ts(j_2) \leq ts(j_1) + (1 - \beta(j_1, j_2, m)) \cdot \Lambda, && (5.24) \\
&\qquad 1 \geq \beta(j_1, j_2, m) + \delta(j_1, m), && (5.25) \\
&tf(j_2) - \epsilon - \ (1 - \eta(j_2, j_1)) \cdot \Lambda - (1 - \delta(j_2, m)) \cdot \Lambda \\
&\qquad \leq ts(j_1) + \beta(j_1, j_2, m) \cdot \Lambda. && (5.26)
\end{aligned}$$

The above MILP formulation finds an assignment and schedule that minimize the chip peak temperature. To minimize peak power P_{max}, we simply substitute the object function as follows:

$$P_{max} \geq \forall_{i \in J} \sum_{m \in M} \sum_{j \in J} \beta(i, j, m) \cdot P(j, m). \tag{5.27}$$

On the other hand, if total energy is to be minimized, the following objective

function can be used

$$E_{total} \geq \sum_{j \in J} \sum_{m \in M} P(j,m) \cdot E(j,m) \cdot \delta(j,m), \tag{5.28}$$

where E_{total} denotes the total energy.

5.4.2 Modeling Power Consumption

In (5.13), the parameter $P(j,m)$ captures the power consumption of core m while executing task j, and

$$P(j,m) = P_{dyn}(j,m) + P_{leak}(j,m), \tag{5.29}$$

where P_{dyn} and P_{leak} are the dynamic power and the leakage power when running task j on core m.

Assuming average switching activity is used to evaluate $P_{dyn}(j,m)$ allows us to treat $P_{dyn}(j,m)$ as a constant. The leakage power, $P_{leak}(j,m)$, however, is a superlinear function of temperature. Simply treating $P_{leak}(j,m)$ as a constant may lead to an underestimation of the chip peak temperature. Though integrated circuit (IC) leakage power is a superlinear function of temperature, a 4-segment piecewise linear function can be used to approximate leakage in the operating temperature ranges of integrated circuits with only 0.69% error [68]. Therefore, we model the power consumption required to execute task j on core m at temperature T_m as a piecewise linear function as follows (more segments can be added as needed)

$$P(j,m) = K_1(j,m) \cdot T_m + K_2(j,m), \tag{5.30}$$

where $K_1(j,m)$ and $K_2(j,m)$ are constants that depend on core m and task j.

Consequently, (5.13) can be rewritten as

$$T(i,m) \equiv T_h(i,h) + \frac{1}{G_h(m)} \sum_{n \in N_m} G_n(m,n) \cdot [T(i,n) - T(i,m)] + \frac{1}{G_h(m)} \cdot \left[\sum_{j \in J} \beta(i,j,m) \cdot (K_1(j,m) \cdot T(i,m) + K_2(j,m)) \right]. \tag{5.31}$$

To eliminate the nonlinear term $\beta(i,j,m) \cdot T(i,m)$, we replace $\beta(i,j,m) \cdot T(i,m)$ with a new variable $\lambda(i,j,m)$. In other words,

$$\lambda(i,j,m) = \begin{cases} K_1(j,m) \cdot T(i,m) & \text{if } \beta(i,j,m) = 1 \\ 0 & \text{otherwise.} \end{cases} \tag{5.32}$$

We then add the following constraints to our MILP formulation.

$$\forall m \in M, \forall i,j \in J$$

$$\lambda(i,j,m) \geq 0, \tag{5.33}$$

$$\lambda(i,j,m) \leq \beta(i,j,m) \cdot \Lambda, \tag{5.34}$$

$$\lambda(i,j,m) \geq (K_1(j,m) \cdot T(i,m)) - (1 - \beta(i,j,m)) \cdot \Lambda, \tag{5.35}$$

$$\lambda(i,j,m) \leq (K_1(j,m) \cdot T(i,m)) - (\beta(i,j,m) - 1) \cdot \Lambda. \tag{5.36}$$

Solving the MILP instance given in (5.13), (5.14) (with $\beta(i,j,m) \cdot T(i,m)$ being replaced by $\lambda(i,j,m)$), (5.14)–(5.26), and (5.33)–(5.36) leads to an exact solution to the problem defined in Section 5.2.3.

Another power consumption related issue is that we model neither inter-core interconnect nor cache power consumption. This omission is due to the following two observations. According to Ku et al. [56], the power density of the cache,

and hence temperature, is relatively low due to its size. A similar conclusion can be drawn about interconnect temperatures; a simulation with default parameters using the Orion interconnect network simulator [105] revealed that typical core power density [65] is about 6.5× that of a high-performance router and 10.6× that of global interconnect wire.

5.4.3 Incorporating Dynamic Voltage Scaling

Many modern processors support dynamic voltage and frequency scaling (DVFS). Although using DVFS to minimize energy will generally also reduce peak temperature, energy minimization alone is not equivalent to peak temperature minimization. Energy minimization does not consider temporal or spatial thermal variation. It is, however, possible and beneficial to consider DVFS in conjunction with our peak temperature optimization technique. Our MILP formulation from Section 5.4.1 can be modified as follows.

For each core m, the set of discrete voltage levels, K_m, must be specified. We redefine $E(j,k,m)$ to be the execution time of task j on core m at voltage level k and $P(j,k,m)$ to be the power consumption required to execute task j on core m at voltage level k. The binary variables $\delta(j,k,m)$ are also redefined to be 1 if task j is assigned to core m at voltage level k. Consequently, from (5.13),

$$\sum_{j \in J} \beta(i,j,m) \cdot P(j,m) = \sum_{k \in K} \sum_{j \in J} \nu(i,j,k,m) \cdot P(j,k,m),$$

where

$$\nu(j_1, j_2, k, m) = \begin{cases} 1 & \text{if } \delta(j_2, k, m) = 1,\ \beta(j_1, j_2, m) = 1 \\ 0 & \text{otherwise.} \end{cases}$$

The constraints in (5.13)–(5.26) can be readily modified for use in the new formulation.

5.4.4 Using Finer-Grained Thermal Model

The thermal model described in Section 5.2.2 can further be refined by using multiple thermal elements for each core, where each thermal element may have different power consumption and/or correspond to a particular functional unit of the core. Specifically, we redefine the variables $\delta(j, m, x)$ and $\beta(j_1, j_2, m, x)$ as follows.

$$\delta(j, m, x) = \begin{cases} 1 & \text{if task } j \text{ executes on functional unit } x \text{ of core } m \\ 0 & \text{otherwise.} \end{cases}$$

$$\beta(j_1, j_2, m, x) = \begin{cases} 1 & \text{if task } j_2 \text{ executes on functional unit } x \text{ of core } m, \\ & \text{precedes, and overlaps with task } j_1 \\ 0 & \text{otherwise.} \end{cases}$$

The constraints in (5.13)–(5.26) can be readily modified for use in the new formulation.

Core power consumption may vary depending on the individual instructions

being executed. However, the relative change in temperature at the functional-unit level is relatively slow due to the large thermal time constants of the functional units (i.e., the constant influencing the rate of temperature change in response to power consumption change). For this reason, the finer-grained thermal model presented here remains accurate; it is not necessary to model instruction-by-instruction variation in power consumption [2].

5.4.5 Modeling Inter-Task Communication

In some situations, communication cost for a task to send data to its successors is significant. Given that the time to send data from task i to task j using shared memory is expressed by parameter $C(i,j)$, our MILP formulation from Section 5.4.1 can be modified to capture inter-task communication by simply substituting (5.17) with the following expression:

$$\forall i,j \in J \qquad ts(j) \geq (tf(i) + C(i,j)) \cdot \Gamma_{i,j}. \tag{5.37}$$

5.4.6 Limitations of MILP-based Approach

While the solution provided by the MILP formulation in Section 5.4.1 is optimal, there are two main limitations to the MILP-based approach: (1) the MILP formulation cannot be used to efficiently solve large problem instances, as the problem defined in Section 5.2.3 is $\mathcal{NP}$-hard, and (2) due to the use of steady-state thermal analysis, the MILP formulation may overestimate the chip peak temperature when task execution times are short relative to the thermal time constant of the cores. That is, steady-state analysis can be used to accurately predict the temperature when task execution times are long compared to the core time constants, and transient analysis should otherwise be used to permit more

accurate temperature prediction, thereby allowing more aggressive scheduling of short tasks that do not cause temperatures to converge to steady-state values during execution.

5.5 Scheduling Heuristic Framework

Our framework uses a binary search based approach to minimize peak temperature under functionality and timing constraints. It takes as inputs upper and lower temperature bounds, as well as the maximum number of iterations, *maxIter*. It then uses the average of the upper and lower bounds on the peak temperatures as the target peak temperature to find an assignment and schedule. If an assignment and schedule is found while staying below the target temperature, the current target temperature will be used as the upper temperature bound for the next iteration of the binary search. Otherwise, it will be used as the lower temperature bound.

We introduce the key part of our framework: `ThermalSched`, a list scheduling [33] algorithm summarized in Algorithm 6. For a given task j, the earliest start time (*EST(j)*) and latest start time (*LST(j)*) are computed. The mobility of task j can then be calculated as the difference between *LST(j)* and *EST(j)*. A potential challenge in computing *EST(j)* and *LST(j)* is that the execution time of task j is unknown prior to the selection of a core. Our solution is to use the smallest execution time of task j as given by the fastest core when computing *EST(j)* and *LST(j)* to maximize the mobility of task j, for all $j \in J$.

The steps for task assignment and scheduling follow. Ready tasks are ordered in a non-decreasing order of mobility. A *ready task* is a task whose predecessors have finished executing. Given a ready task j, `ThermalSched` selects the fastest

Algorithm 6 ThermalSched($G(V, E)$, *targetTemp*, *dMaxIter*)

compute *EST(j)* and *LST(j)*, for all tasks
compute *avgE* // average execution time over all tasks and cores
mobility(j) $\leftarrow$ *LST(j)* - *EST(j)*, for all tasks
currentTime $\leftarrow 0$, *busy(m)* $\leftarrow 0$, for all cores
while there are unscheduled tasks **do**
  $RT \leftarrow$ ready tasks in non-decreasing order of mobility
  for each $j \in RT$ **do**
    invalidCount $\leftarrow 0$, *fastestCore* $\leftarrow -1$, *bestExeTime* $\leftarrow \infty$
    for each $m \in M$ **do**
      $\delta(j, m) \leftarrow 0$
      endTime $\leftarrow$ *E(j,m)* + *currentTime*
      if not *busy(m)* **and** *endTime* $\leq D(j)$ **then**
        compute projected thermal profile for [*currentTime*, *nextIdleTime*] // *nextIdleTime* is the next earliest time when all cores become idle
        peakTemp $\leftarrow \max_{m \in M} T(j, m)$
        if *peakTemp* $\leq$ *targetTemp* **then**
          if *E(j,m)* $<$ *bestExeTime* **then**
            fastestCore $\leftarrow m$, *bestExeTime* $\leftarrow$ *E(j,m)*, *currentDelay* $\leftarrow 0$
        else if *currentTime* > 0 **then**
          [*fastestCore*, *bestExeTime*, *currentDelay*] $\leftarrow$ DelayInsertion($G(V, E)$, j, m, *currentTime*, *targetTemp*, *dMaxIter*, *avgE*)
      else if not *busy(m)* **then**
        invalidCount $\leftarrow$ *invalidCount* + 1
    if *invalidCount* $= |M|$ **then**
      return INFEASIBLE
    else if *fastestCore* $\neq -1$ **then**
      $\delta(j,$ *fastestCore*$) \leftarrow 1$ // assign j to *fastestCore*
      ts(j) $\leftarrow$ *currentTime* + *currentDelay*
      tf(j) $\leftarrow$ *ts(j)* + *E(j, fastestCore)*
      busy(fastestCore) $\leftarrow 1$
    if *currentDelay* > 0 **then**
      break // allow no tasks to start executing between *currentTime* and *currentTime* + *currentDelay*
  update *EST(j)*, for all unscheduled tasks
  update *mobility(j)*, for all unscheduled tasks
  nextSchedPoint $\leftarrow \min\{tf(j) : tf(j) >$ *currentTime* + *currentDelay*$\}$
  for each $m \in M$ **do**
    if m becomes idle at *nextSchedPoint* **then**
      busy(m) $\leftarrow 0$
  currentTime $\leftarrow$ *nextSchedPoint*
return FEASIBLE

available core that allows the task to meet its deadline while keeping the peak temperature below the target temperature. The fastest available core is chosen to maximize the mobility of the successors of task j, thereby improving schedulability. If no core is fast enough to execute task j by its deadline, `ThermalSched` terminates. (We ignore Lines 18–19 and 29–30 in Algorithm 6, and the variables *currentDelay* and *dMaxIter* for now. Their use will be explained in the next section.) Our search-based scheduling approach permits the use of an efficient list scheduler without global knowledge of temperature variation.

Observe that Algorithm 6 does not provide any details on computing the thermal profile (Line 17). Since predicting highly accurate thermal profiles increases time complexity, we propose two techniques based on the observations made in Section 5.4.6. These techniques achieve different trade-offs between accuracy and time complexity.

5.5.1 Steady-State Analysis Based Heuristic

As explained in Section 5.4.6, if task execution times are long compared to the thermal time constants of the cores, steady-state analysis can usually rapidly and accurately predict the resulting chip temperature.

The steady-state thermal profile can be computed by expressing (5.2) and (5.3) for all the thermal elements as a system of linear equations of the form $A \cdot \mathbf{T} + B = 0$, and of size $|E| \times |E|$, where $|E|$ is the total number of thermal elements. Since the thermal conductance matrix A is fixed once a floorplan is given, the inverse of the matrix can be pre-computed once and the temperature matrix can be updated using a constant number of multiplications in each iteration. In our work, we use Matrix TCL Lite [72], which is a `C++` class library, to perform matrix

operations. `ThermalSched` has a time complexity in $\mathcal{O}(|J|^2 \cdot |M|^3)$. The time complexity of the **steady-state thermal analysis based heuristic (SSAB)** is in $\mathcal{O}(|J|^2 \cdot |M|^3 \cdot maxIter)$.

5.5.2 Transient Analysis Based Heuristic

If task execution times are short, it is desirable to use transient analysis to compute the projected thermal profile, as explained in Section 5.4.6. Essentially, any existing thermal analysis technique can be used in our task assignment and scheduling heuristic framework. To validate our **transient analysis based heuristic (TAB)**, we will use `HotSpot` [98] in our experiments. `HotSpot` is an architecture-level thermal modeling tool that is based on an equivalent circuit of thermal resistances and capacitances that correspond to microarchitecture blocks and essential aspects of the thermal package [98]. The transient analysis based heuristic has a time complexity of $\mathcal{O}(|J|^2 \cdot |M| \cdot maxIter \cdot H)$, where H denotes the running time of `HotSpot` and depends on a number of input parameters such as task execution times and number of cores in the MPSoCs.

5.6 Delay Insertion

Algorithm 6 always tries to schedule as many tasks as possible at every scheduling point to maximize the mobility of later tasks. One possible consequence of this greedy approach to task assignment and scheduling is that the chip peak temperature may be so close to the target temperature that no future ready task can execute without violating the target temperature bound, thus requiring a higher target temperature to find a feasible task assignment and schedule. To address this weakness in our heuristic framework, we introduce the concept of

delay insertion. That is, when the chip peak temperature is at or near the target peak temperature, we delay the execution of the next ready task by introducing an idle interval before the task starts to allow the chip to cool down. This improves the probability of later tasks being scheduled without exceeding the target temperature bound.

Since steady-state thermal analysis depends on fast temperature rises and falls, an idle time (or a delay inserted) between task executions has no effect on the resulting peak temperature. On the other hand, transient thermal analysis would capture the cooling effects of delay insertions. Hence, the concept of delay insertions applies to the TAB heuristic only.

To demonstrate the potential benefits of delay insertions, we use the following illustrative example. Consider a system with 5 identical real-time tasks running on the MPSoC shown in Figure 5.1 with the associated execution time and 10× the power consumption for each core. Without inserting any idle times, our TAB heuristic finds a feasible assignment and schedule with a peak temperature of 51.60 °C. An algorithm capable of inserting appropriate delays would reduce the peak temperature to 49.88 °C.

In the above example, delay insertion only reduces the chip peak temperature by 1.72 °C because there are only 5 tasks in the system with relatively short execution times. In other words, executing these tasks on the example MPSoC does not significantly raise the chip peak temperature. If our tasks require 10× the original execution times, i.e., a mean of 30 ms, then delay insertions would reduce the chip peak temperature by 3.87 °C, from 66.22 °C to 62.35 °C.

We now explain the use of delay insertions in our heuristic framework. Whenever an attempt to schedule a task on a core fails because the target temperature

bound is exceeded, `DelayInsertion` is called (Lines 18–19 of Algorithm 6). If an idle time has successfully been inserted into the schedule, our heuristic will immediately move on to the next scheduling point by setting *currentTime* to $\min\{tf(j) : tf(j) > currentTime + currentDelay\}$ and continue the assignment and scheduling process (Lines 29–30 and Line 33 of Algorithm 6). No pending tasks are allowed to run during $[currentTime, currentTime + currentDelay)$. This not only simplifies the algorithm, but is also reasonable since when an idle time has been inserted, the current chip peak temperature is at or near the target temperature; executing another task within the interval $[currentTime, currentTime + delay)$ would likely cause the target temperature to be exceeded.

Our delay insertion algorithm is shown in Algorithm 7. To find the appropriate idle time to insert delays without sacrificing the schedulability of future tasks, we use a binary search based approach. In each iteration, Algorithm 7 attempts to schedule the current task onto the core currently under consideration such that the resulting peak temperature does not exceed the target peak temperature. If this is possible, Algorithm 7 will keep this scheduling and assignment only if the current configuration has minimized the task finish time thus far. Hence, if an assignment and schedule exists for the current task that does not require delay insertions, that assignment and schedule will likely be selected (this design choice maximizes the mobility of future tasks). Algorithm 7 halts when the maximum number of iterations *dMaxIer* has been reached or an appropriate idle time has been found. The appropriate idle time is found when the current assignment and schedule do not exceed the target temperature and the chip peak temperature has converged (Line 10 of Algorithm 7).

In our implementation, the search begins by setting the upper bound on the

Algorithm 7 DelayInsertion($G(V, E)$, j, m, *currentTime*, *targetTemp*, *dMaxIter*, *avgE*)

1: $upperDelay \leftarrow avgE$
2: $lowerDelay \leftarrow 0$
3: $delay \leftarrow (upperDelay + lowerDelay)\ /\ 2$
4: $iter \leftarrow 0$
5: $oldPeakT \leftarrow 0$
6: $newPeakT \leftarrow 0$
7: **while** $iter < dMaxIter$ **do**
8: compute projected thermal profile for $[currentTime, currentTime + delay]$ and $[currentTime + delay, nextIdleTime]$ // *nextIdleTime* is the next earliest time when all cores become idle
9: $newPeakT \leftarrow \max_{m \in M} T(j, m)$
10: **if** $targetTemp > newPeak$ **and** $|oldPeakT - newPeakT| < \epsilon$ **then**
11: **if** $E(j,m) + delay < bestExeTime$ **and** $currentTime + delay + E(j, m) \leq D(j)$ **then**
12: $fastestCore \leftarrow m$
13: $bestExeTime \leftarrow E(j,m)$
14: **return** $[fastestCore, bestExeTime, delay]$
15: **else**
16: **return** FAILURE
17: **else if** $targetTemp > newPeakT$ **then**
18: $upperDelay \leftarrow delay$
19: **else**
20: $lowerDelay \leftarrow delay$
21: $oldPeakT \leftarrow newPeakT$
22: $delay \leftarrow (upperDelay + lowerDelay)\ /\ 2$
23: $iter \leftarrow iter + 1$

delay to the average execution time of all task and core combinations and the lower bound delay to 0. This design choice has the following justifications: (1) if the upper bound is too large, `DelayInsertion` can be slow and (2) `DelayInsertion` is most likely invoked when the chip temperature is near the target temperature. Inserting an idle time similar to the average task execution time would allow, on average, the chip to cool down enough to permit most tasks to eventually be scheduled on the current core without sacrificing the efficiency of the heuristic.

5.7 Experimental Results

This section quantifies the benefits of our proposed approach and assess the quality of our heuristic framework.

5.7.1 Experimental Setup

In our experiments, we used the Embedded System Synthesis Benchmarks Suite (E3S) [36]. E3S contains 17 processing elements (PEs). In our experiments, we used the following 11 cores: AMD K6-2 450, AMD K6-2E 400 Mhz/ACR, AMD K6-2E+ 500 Mhz/ACR, AMD K6-IIIE+ 550 Mhz/ACR, IBM PowerPC 405GP 266 Mhz, IBM PowerPC 750CX 500 MHz, IDT32334 100 MHz, IDT79RC32V334-150, IDT79RC64575 250 MHz, Motorola MPC555 40 MHz, and TI TMS320C6203 300 MHz. (Note that we did not use all 17 cores because for each floorplan, we attempted to use cores with similar sizes.) The E3S task sets follow the organization of the EEMBC benchmarks [36]. There are five benchmarks in total: Auto (24 tasks), Consumer (12 tasks), Networking (13 tasks), Office (5 tasks), and Telecom (30 tasks). Each benchmark represents an application, as its name indicates. Each sink task, which does not have any successors, has a hard real-time deadline.

For the E3S benchmarks, we experimented with a number of floorplans with 2×2, 2×3, and 3×3 core arrangements. We use the coarse-grained thermal model presented in Section 5.2.2 instead of the fine-grained thermal model in Section 5.4.4 due to the lack of realistic benchmarks for which power profile variations within cores are known. Each benchmark has different floorplans, as specific tasks are required to run on specific cores among the 11 cores mentioned above. The specific configuration of each floorplan and the corresponding core names are provided in Tables 5.1 and 5.2, respectively. The chips consist of heterogeneous cores. Since cores with different power consumptions tend to have different areas, the vertical and lateral thermal conductance between neighboring cores, and between cores and heatsink elements will vary and can be computed as described in Section 5.2.2.

We also used TGFF [35], which is a pseudo-random task graph generator, in our experiment to generate 10 additional benchmarks. For each benchmark, there are up to 5 task graphs and the total number of tasks ranges from 4 to 29 tasks (this is similar to the number of tasks in the E3S benchmarks). Each task has at most 3 predecessors and 2 successors. A 2×2 core arrangement was used, with an average core width and height of 5 mm and an average power consumption of 10 W.

5.7.2 MILP Formulation Performance

In this set of experiments, we used `CPLEX` with `AMPL` to solve instances of the MILP formulation in Section 5.4 for optimal peak temperature, energy, and peak power. Each E3S benchmark was run for two 2×2, one 2×3, and one 3×3 floorplan.

TABLE 5.1

FLOORPLAN CONFIGURATIONS

Benchmark	First Row	Second Row	Third Row
Auto-2×2-1	14, 7	1, 7	
Auto-2×2-2	1, 7	1, 7	
Auto-2×3	1, 7, 7	1, 7, 7	
Auto-3×3	14, 14, 14	1, 1, 1	7, 7, 7
Consumer-2×2-1	7, 7	11, 11	
Consumer-2×2-2	7, 7	9, 7	
Consumer-2×3	7, 7, 7	11, 11, 11	
Consumer-3×3	7, 7, 7	9, 9, 9	7, 7, 7
Networking-2×2-1	2, 3	4, 5	
Networking-2×2-2	5, 5	4, 4	
Networking-2×3	5, 4, 5	5, 4, 5	
Networking-3×3	5, 5, 5	4, 4, 4	5, 5, 5
Office-2×2-1	4, 3	4, 12	
Office-2×2-2	4, 3	4, 3	
Office-2×3	4, 8, 4	3, 8, 3	
Office-3×3	3, 8, 3	8, 12, 8	3, 8, 3
Telecom-2×2-1	14, 7	1, 7	
Telecom-2×2-2	1, 7	1, 7	
Telecom-2×3	1, 7, 7	1, 7, 7	
Telecom-3×3	14, 14, 14	1, 1, 1	7, 7, 7

TABLE 5.2

CORE NAMES

Index	Core
1	AMD ElanSC520-133 MHz
2	AMD K6-2 450
3	AMD K6-2E 400Mhz/ACR
4	AMD K6-2E+ 500Mhz/ACR
5	AMD K6-IIIE+ 550Mhz/ACR
6	Analog Devices 21065L - 60 MHz
7	IBM PowerPC 405GP - 266 Mhz
8	IBM PowerPC 750CX - 500 MHz
9	IDT32334-100 MHz
10	IDT79RC32364-100
11	IDT79RC32V334-150
12	IDT79RC64575-250MHz
13	Imsys Cjip 40 Mhz
14	Motorola MPC555 - 40MHz
15	NEC VR5432 - 167 MHz
16	ST20C2 50 Mhz
17	TI TMS320C6203-300MHz

We first examine the temperature differences between optimizing peak temperature and optimizing energy or peak power. The solutions from the MILP solver are shown in Figure 5.4. The x-axis shows the different benchmarks and floorplans. The y-axis shows the resulting peak temperatures. Some results are unavailable due to the MILP solver running out of memory before finding a solution. Our approach reduces peak temperatures by 9.19 °C on average, and up to 24.66 °C, when compared to the method that minimizes energy. Most of the improvement results from considering the effects of temporal thermal variations.

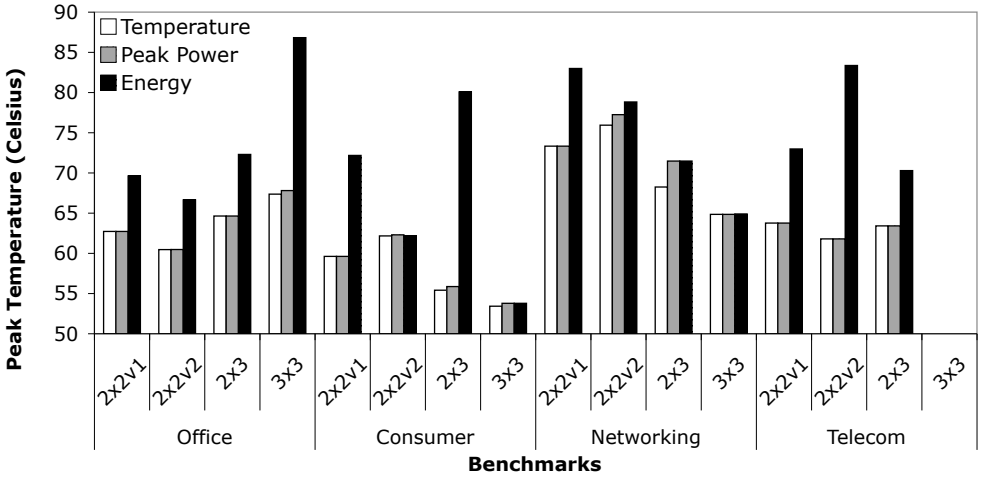

Figure 5.4. Peak temperature minimization vs. energy and peak power minimization. The results were obtained by solving the MILPs directly. Clearly, energy minimization is not effective in reducing the chip peak temperature.

The results in Figure 5.4 do not show significant differences in peak temperatures between our approach and the approach that minimizes peak power. This is because the low-power embedded cores used in our benchmarks have low power densities. For example, the floorplans for the Consumer benchmarks resulted in a chip power density ranging from $0.27\,\mathrm{W/mm^2}$ to $0.36\,\mathrm{W/mm^2}$ with an average chip power density of $0.32\,\mathrm{W/mm^2}$. As a result, little spatial temperature variation was observed. However, spatial temperature variation will increase when higher power density chips are used, as explained in Section 5.3.

In the first run, we used the original chip power density. The chip power density was then increased by an order of magnitude (once again to obtain similar power densities to those described by Link and Vijaykrishnan [65]) in the second run. The resulting chip power densities ranged from $0.75\,\mathrm{W/mm^2}$ to $3.13\,\mathrm{W/mm^2}$ with an average power density of $2.28\,\mathrm{W/mm^2}$. As shown in Figure 5.5, for these cores our method reduces peak temperature by $9.58\,^{\circ}\mathrm{C}$ on average, and up to $23.25\,^{\circ}\mathrm{C}$, when compared to peak power minimization. These results demonstrate the advantage of considering spatial thermal variations.

There exist situations where optimal peak temperature cannot be obtained by either energy or peak power minimization. This situation was observed in the Networking benchmark with a 2×3 core arrangement (depicted in Figure 5.6, diagram not drawn to scale). Due to the characteristics of this benchmark, at least two cores must be active simultaneously at some time. In the case of energy and peak power minimization, the optimal solution consists of executing on cores *m2* and *m5* in parallel. This yields an optimal peak power of 28 W and an optimal energy of 35.46 J. The peak temperature obtained in this case is $61.45\,^{\circ}\mathrm{C}$. Using our approach, cores $m3$ and $m4$ will be executing simultaneously. This solution

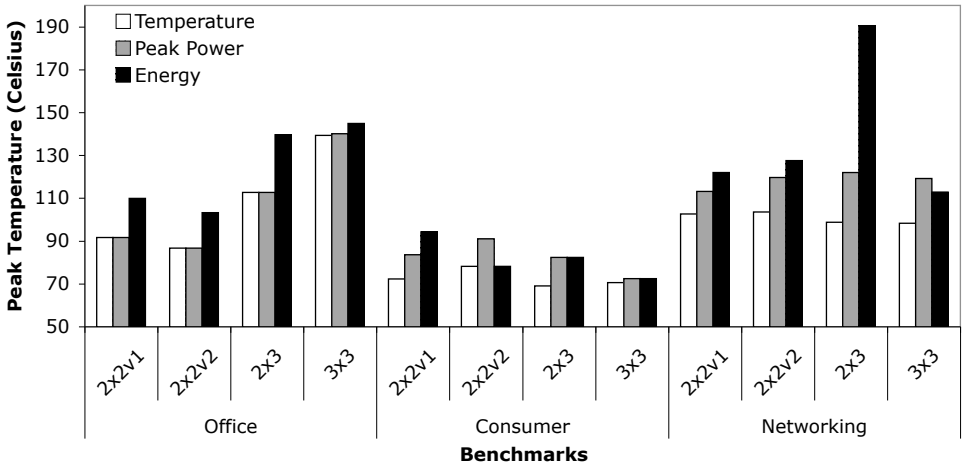

Figure 5.5. Peak temperature minimization vs. energy and peak power minimization for higher power density chips. The results were obtained by solving the MILPs directly. Here, as the chip power density increases, spatial thermal variation becomes significant and peak power minimization is not effective in reducing the chip peak temperature.

gives an optimal peak temperature of 58.15 °C, which is about 3 °C lower than either energy or peak power minimization. However, this solution yields a peak power of 32 W and a total energy of 36.20 J, which means that neither energy nor peak power minimization can achieve this optimal peak temperature.

Even in the cases where peak power (or energy) minimization can yield optimal peak temperature, it is still relevant to minimize peak temperature directly. Firstly, there is no guarantee that an optimal peak temperature will be obtained by minimizing peak power, as the latter considers temporal thermal variation but ignores spatial thermal variation. Secondly, there may exist a range of possible peak temperatures as a result of a single optimal peak power. If such a range is large, the actual peak temperature of a chip can vary significantly.

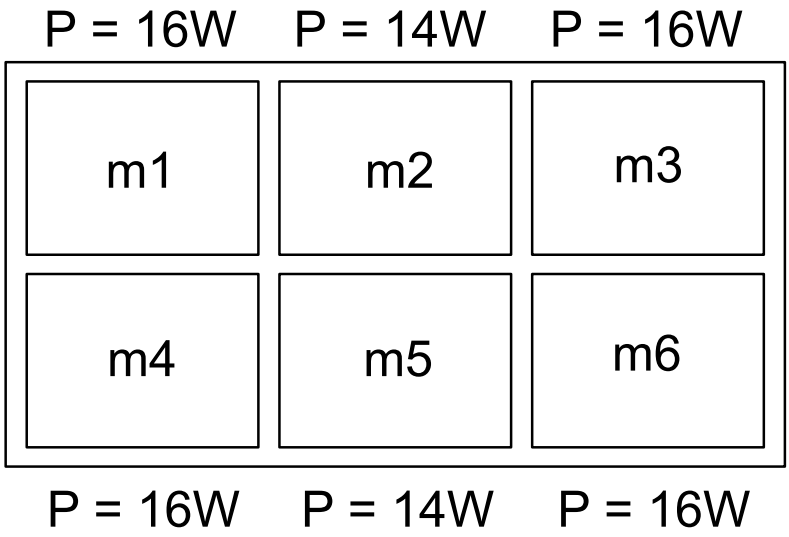

Figure 5.6. An example 2×3 floorplan with associated power consumption for each core.

To illustrate this scenario, we performed an additional experiment using a slightly modified Consumer benchmark. In this version, task deadlines were modified in such a way that at least two tasks must execute in parallel. We used a 2×3 floorplan with homogeneous processors. The experiment was run twice. In the first run, we used the original chip power density. The chip power density was then increased by a factor of 10 (once again to obtain similar power densities as in [65]) in the second run. For this benchmark, there exist four distinct parallel core assignments yielding the same optimal peak power but different peak temperatures. The left bars in Figure 5.7 show the peak temperatures for each of these four assignments in the first run. The right bars show the range of possible peak temperatures of the chip with a higher power density. The results show that the difference in peak temperature for the same peak power can be over 5 °C for the higher power density chip. Clearly, peak power minimization is not sufficient, especially when it is predicted that the power density of future chips will continue to increase.

Finally, when using the TGFF benchmarks described earlier, the MILP solver could handle eight out of ten problem instances. Minimizing peak temperature directly instead of minimizing energy reduced peak temperatures by 9.41 °C on average and up to 24.19 °C. In addition, when compared to peak power minimization, peak temperature minimization reduced peak temperatures by 1.27 °C on average and up to 6.71 °C.

The above results allow for a general conclusion to be drawn. Average power minimization suffers due to temporal and spatial thermal variation. Peak power minimization ignores spatial thermal variation. Peak temperature minimization takes both types of thermal variation into account. Variation in floorplan and

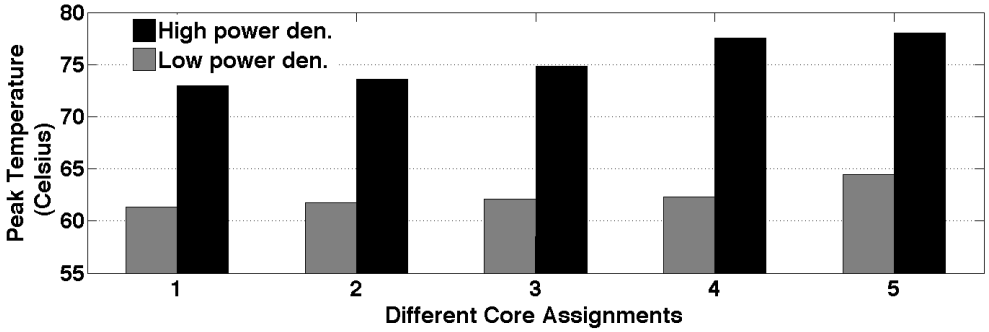

Figure 5.7. Bar plot illustrating the resultant peak temperatures for the different core assignments that can be obtained given the same peak power. The maximum difference in peak temperatures is higher than 5 °C for the Networking benchmark, which contains only 12 tasks.

other task execution times results in variation in task mobility, resulting in variation in the peak temperature improvements achieved by our approach.

As previously mentioned, the MILP-based approach is only suitable for small problem instances. We next examine the performance of our heuristic framework, which is more scalable.

5.7.3 Performance of Steady-State Analysis Based Heuristic

We assess the performance of our SSAB algorithm (Section 5.5) by comparing its solutions to the ones from the MILP solver (Section 5.4.1) as well as the results from Xie's and Hung's *Heuristic 1* [111], which we refer to as the X&H heuristic. The X&H heuristic calls HotSpot to compute the temperatures. Figure 5.8 compares the results from the SSAB and X&H heuristics to the optimal solution from the MILP formulation. We used HotSpot to compare the peak temperatures for a fair comparison. Results for benchmarks that were not successfully solved by the

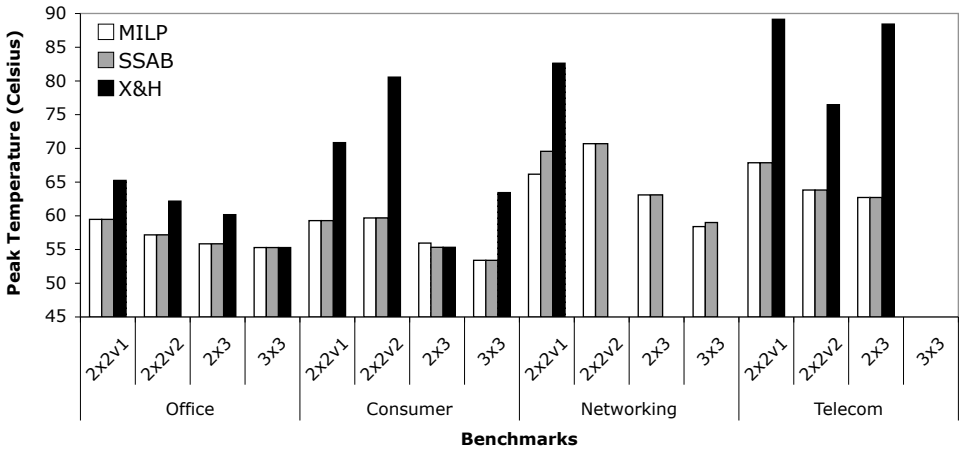

Figure 5.8. Performance comparison between the steady-state analysis based heuristics (based on HotSpot), the MILP, and Xie and Hung's heuristic [111].

X&H algorithm are omitted.

The X&H heuristic deviates from the optimal solution by 10.94 °C on average and 38.40 °C in the worst case. On the other hand, the SSAB heuristic finds an optimal solution in many cases while giving results that deviate by at most 3.40 °C from optimality (and 0.22 °C on average) requiring at most 50 binary search iterations for each benchmark. Both heuristics require similar running times, but the SSAB heuristic never performs worse than the X&H heuristic.

When the X&H heuristic was tested on the 10 TGFF benchmarks mentioned previously, we found that it could only solve two of the benchmarks, with a maximum deviation from optimality of 7.74 °C. On the other hand, the SSAB heuristic could solve four, one of which was a problem instance so large that it proved intractable for the MILP solver. For the three other benchmarks, the SSAB heuristic found the same solutions as the MILP solver.

To demonstrate that the SSAB heuristic can efficiently solve larger problem instances, we considered a benchmark that consists of 30 tasks and a 4×4 core arrangement of homogeneous processors. First, we attempted to use the MILP solver. As expected, no solution was returned, as the 3.58 GB RAM workstation on which `CPLEX` was executing ran out of memory. The SSAB heuristic found a solution using at most 50 binary search iterations (although the quality of the solution is unknown).

We used a discretized Fourier heat flow model in the SSAB heuristic. It differs from that in HotSpot. However, the experiments in this section show that the peak temperatures from the two models differed by less than 5 °C on average. This indirectly served as a validation of our thermal model.

5.7.4 Performance of Transient Analysis Based Heuristic

We now assess the performance of the TAB heuristic described in Section 5.5 using the benchmarks from Section 5.7.1. The TAB heuristic calls HotSpot to determine transient temperatures. Since the original task execution times for the E3S benchmarks tend to be short, dynamic thermal effects can be significant. We compare the peak temperatures obtained by the MILP solver and the TAB heuristic, as shown in Figure 5.9. When compared to the results from the MILP solver, the TAB heuristic reduces the peak temperature by up to 0.67 °C and 0.06 °C on average. This is because transient analysis can more accurately predict temperatures when performing assignment and scheduling.

The TAB heuristic also improves the task finish times. Let the speedup be the ratio of the finish time of the last task in the MILP schedule to that in the TAB schedule. The maximum, minimum, and average speedups are 78.13×, 1.21×,

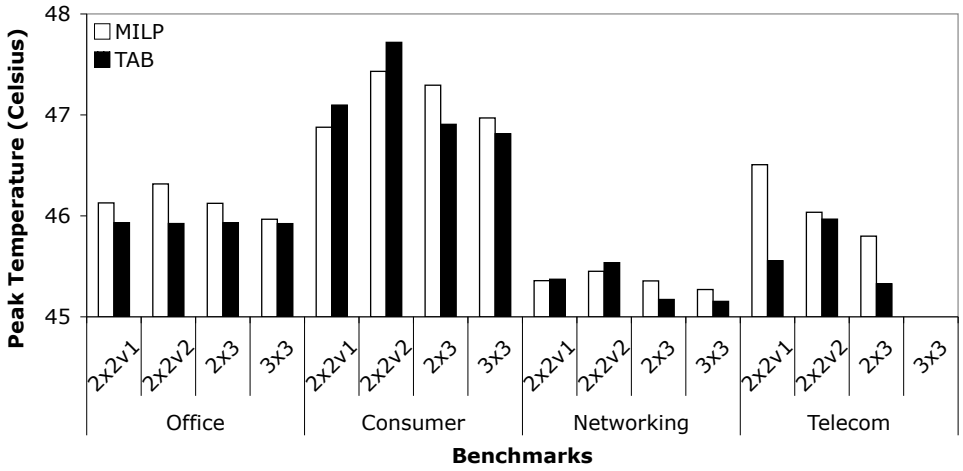

Figure 5.9. Performance comparison between the transient analysis based heuristics (based on HotSpot) and the MILP. Here, the MILP sometimes results in a higher peak temperature because it uses the steady-state thermal analysis, which ignores dynamic thermal effects.

and 9.02×, respectively. Such a significant speedup results from the TAB heuristic being much less pessimistic in estimating temperatures and hence scheduling more tasks in parallel. However, the SSAB heuristic is more efficient than the TAB heuristic. Specifically, the SSAB heuristic is about 175× faster than the TAB heuristic on average for benchmarks with short task execution times; this difference further increases for benchmarks with longer task execution times.

5.7.5 Performance of Transient Analysis Based Heuristic with Delay Insertions

To determine the impact of delay insertions (Section 5.6) on reducing the chip peak temperature, we once again used the E3S and TGFF benchmarks. We compared the solutions from the original TAB heuristic to those from the improved TAB heuristic (iTAB). As before, since the power densities of the E3S cores are

quite low compared to those of today's processors [65], we increased each cores power consumption by a factor of 10. The results for the E3S benchmarks are shown in Table 5.3. The first column shows the benchmark names and associated floorplans. The second main column presents the peak temperatures from the TAB heuristic and the iTAB heuristic, as well as the differences in peak temperatures for the original task execution times. Finally, the last main column presents the data for the case where task execution times are multiplied by a factor of 10.

With the original task execution times, the effect of delay insertions is minimal for some benchmarks (e.g., networking). In fact, for two of the telecom benchmarks, iTAB actually performs worse than TAB. This is because iTAB always tries to insert delays, even when it may not be optimal to do so. For instance, if the chip is currently too hot to execute the next ready task, iTAB tries to insert idle time before scheduling that task. However, it may sometimes be better to select a different task that can meet the peak temperature constraint in Algorithm 6 without inserting delays.

When the execution times are increased by a factor of 10, we see the benefits of using the iTAB heuristic. This is because the average chip peak temperature is much higher than in the original cases and inserting idle times between task executions cools the chip down. iTAB produces solutions that reduce peak temperatures by 3.15 °C on average and up to 11.92 °C.

iTAB did not significantly improve on the TGFF benchmark solutions found by TAB; both solved four out of the ten problem instances. The largest improvement in peak temperature was 1.71 °C. This is because most tasks in the TGFF benchmarks did not have enough slack for delay insertion to be effective.

Based on these results, we conclude that iTAB reduces the peak temperature

of systems with time slack. It must be noted, however, that iTAB also requires longer running times. On average, iTAB takes 19.2× longer to run than the original TAB algorithm. There is a trade-off between solution quality and time complexity of these algorithms.

5.8 Summary

We presented an assignment and scheduling technique that uses a mixed-integer linear program solver to optimize IC peak temperature under precedence and hard real-time constraints based on phased steady-state thermal analysis. Experimental results show a peak temperature reduction of 10.09 °C on average and up to 30.75 °C for embedded processors when compared to energy minimization. When compared to peak power minimization, our approach reduced peak temperature of 8.98 °C on average and up to 23.25 °C for high power density chips.

To efficiently solve this $\mathcal{NP}$-hard problem, we also proposed a task assignment and scheduling heuristic framework in which the actual method for temperature prediction depends on task durations. Phased steady-state analysis is appropriate when task execution times are long compared to the thermal time constants of the cores and transient analysis should be used otherwise. Our phased steady-state analysis based heuristic finds solutions with a maximum deviation from optimality of 3.40 °C. When compared to previous work, the heuristic achieves a temperature reduction of 10.94 °C on average. The transient analysis based heuristic models and exploits the transient thermal effects of short tasks to further improve upon the existing solution by 0.67 °C in the best case. Finally, we showed that incorporating the concept of delay insertion into the proposed heuristic framework results in an additional peak temperature reduction of up to 11.92 °C.

TABLE 5.3

EFFECTIVENESS OF DELAY INSERTIONS IN REDUCING CHIP PEAK TEMPERATURES

Benchmark		Temperature (°C)					
		Original Exe.Times (s)			Exe. Times (s) × 10		
		TAB	iTAB	Diff.	TAB	iTAB	Diff.
Consumer	2×2-1	65.78	60.56	5.22	84.30	80.90	3.40
	2×2-2	86.24	81.47	4.77	86.24	81.47	4.77
	2×3	63.32	60.06	3.26	79.24	76.81	2.43
	3×3	61.97	60.28	1.69	78.60	76.85	1.75
Networking	2×2-1	47.49	47.45	0.04	57.81	55.43	2.38
	2×2-2	47.29	46.85	0.45	57.83	53.48	4.35
	2×3	46.80	46.80	0.00	53.96	53.87	0.09
	3×3	46.80	46.79	0.01	53.45	53.36	0.09
Office	2×2-1	54.22	54.22	0.00	75.96	75.96	0.00
	2×2-2	54.14	54.14	0.00	75.37	75.37	0.00
	2×3	54.21	54.21	0.00	67.91	67.89	0.02
	3×3	54.13	54.13	0.00	67.43	57.94	9.49
Telecom	2×2-1	50.28	51.70	-1.42	72.06	65.53	6.53
	2×2-2	49.48	52.79	-3.31	71.26	59.34	11.92
	2×3	46.38	47.40	-1.02	51.37	51.30	0.08

Since real-time systems can experience great temperature variations at run-time due to the differences in actual task execution times, we intend to solve the peak temperature minimization problem online to further reduce temperature and increase system reliability.

CHAPTER 6

CONCLUSIONS

6.1 Summary

Designing real-time systems under physical and resource constraints is an important problem that has received significant research attention in the past several years. In this dissertation, we discussed the design of real-time systems under energy, processing, and network constraints (i.e., resource constraints), as well as temperature constraints (i.e., physical constraints). To address processing overloads in real-time control applications, we proposed an efficient online heuristic that is compatible with a more general real-time task model in Chapter 2. In terms of network overload management for wireless soft real-time applications, we presented an energy-aware holistic scheduling framework in Chapter 3. Chapter 4 discussed an optimal DVFS speed selection policy to maximize the performance of temperature-constrained systems. Finally, a temperature-aware assignment and scheduling algorithm for hard real-time applications running on a multicore system is given in Chapter 5.

6.2 Future Research Directions

Chapters 2.6, 3.8, 5.8, and 4.7 outlined some specific future research directions that logically follow from the work in respective chapters. We now present a

high-level overview of two boarder research problems.

6.2.1 Designing for High-Performance and Dependability on Multicore and Many-Core Systems

Embedded systems are traditionally designed in an ad-hoc basis. To make their design feasible when a large number of processor cores is available, there are some fundamental questions that need to be answered. For instance, communication delays and cost may cause centralized task assignment and scheduling to become obsolete. A localized approach that is adaptive in nature is needed.

In addition, with a large number of processor cores, the management of resources such as memory, network cards, and disks is crucial in time-sensitive systems. A low-overhead arbitration mechanism must be in place.

Finally, as voltage scaling continues to take place to save energy, the system is more prone to soft errors during task execution. Providing fault-tolerance is essential but is often in conflict with other design goals such as energy savings. The problem is even more challenging in large-scale systems and must be addressed.

6.2.2 Designing Highly-Adaptive, Highly-Reconfigurable Cyber-Physical Systems (CPS)

We are now entering the era where systems that have previously been classified as belonging to science fiction are quickly becoming a reality, e.g., remote surgery or in-home medical care. With a large number of devices forming a large network, there are two main challenges that need to be addressed: (i) locally and dynamically reconfigure the nodes to keep the CPS alive in face of node failures, and (ii) distribute dynamic workload to save energy.

BIBLIOGRAPHY

1. K. Albers and F. Slomka. Efficient feasibility analysis for real-time systems with EDF scheduling. In *Proceedings of the Design, Automation & Test in Europe Conference*, pages 492–497, Mar. 2005.

2. N. Allec, Z. Hassan, L. Shang, R. P. Dick, and R. Yang. ThermalScope: multi-scale thermal analysis for nanometer-scale integrated circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 603–610, Nov. 2008.

3. J. Anderson and S. Baruah. Energy-efficient synthesis of periodic task systems upon identical multiprocessor platforms. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 428–435, Mar. 2004.

4. P. Antsaklis and J. Baillieul. Special issue on networked control systems. *ACM Transactions on Automatic Control*, 49(9):1421–1423, Sept. 2004.

5. H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, page 113.2, Apr. 2003.

6. H. Aydin, R. Melhem, D. Mosse, and P. Alvarez. Optimal reward-based scheduling for periodic real-time tasks. In *Proceedings of the Real-Time Systems Symposium*, pages 79–89, Dec. 1999.

7. P. Balbastre, I. Ripoll, and A. Crespo. Minimum deadline calculation for periodic real-time tasks in dynamic priority systems. *IEEE Transactions on Computers*, 57(1):96–109, Jan. 2008.

8. N. Bansal, T. Kimbrel, and K. Pruhs. Dynamic speed scaling to manage energy and temperature. In *Proceedings of the Symposium on Foundations of Computer Science*, pages 520–529, Oct. 2004.

9. N. Bansal, T. Kimbrel, and K. Pruhs. Speed scaling to manage energy and temperature. *Journal of the ACM*, 54(1):1–39, Mar. 2007.

10. S. Baruah, L. Rosier, and R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, Nov. 1990.

11. G. Bernat, A. Burns, and A. Llamosí. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, Apr. 2001.

12. E. Bini and M. D. Natale. Optimal task rate selection in fixed priority systems. In *Proceedings of the Real-Time Systems Symposium*, pages 399–409, Dec. 2005.

13. B. Borchers and J. Mitchell. An improved branch and bound algorithm for mixed integer nonlinear programs. *Computers and Operations Research*, 21: 359–367, Apr. 1994.

14. S. Borkar. Exponential challenges, exponential rewards – the future of moore's law. Website, 2004. `http://nanohub.org/resources/177`.

15. S. Y. Borkar, P. Dubey, K. C. Kahn, D. J. Kuck, H. Mulder, S. S. Pawlowski, and J. R. Rattner. Platform 2015: Intel processor and platform evolution for the next decade. Technical report, Intel Corporation, Mar. 2005.

16. G. Buttazzo. *Hard Real-Time Computing Systems*. Springer, second edition, 2004.

17. G. Buttazzo and L. Abeni. Adaptive workload management through elastic scheduling. *Real-Time Systems*, 23(1–2):7–24, July 2002.

18. G. Buttazzo and L. Abeni. Smooth rate adaptation through impedance control. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 3–10, June 2002.

19. G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. In *Proceedings of the Real-Time Systems Symposium*, pages 90–99, Dec. 1995.

20. G. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *Proceedings of the Real-Time Systems Symposium*, pages 286–295, Dec. 1998.

21. G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3): 289–302, Mar. 2002.

22. M. Caccamo, G. Buttazzo, and L. Sha. Elastic feedback control. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 121–128, June 2000.

23. M. Caccamo, G. Buttazzo, and L. Sha. Handling execution overruns in hard real-time control systems. *IEEE Transactions on Computers*, 51(7):835–849, July 2002.

24. T. Chantem, X. Hu, and M. Lemmon. Generalized elastic scheduling. In *Proceedings of the Real-Time Systems Symposium*, pages 236–245, Dec. 2006.

25. T. Chantem, X. Hu, and M. Lemmon. Generalized elastic scheduling for real-time tasks. *IEEE Transactions on Computers*, 58(4):480–495, Apr. 2009.

26. J.-J. Chen, C.-M. Hung, and T.-W. Kuo. On the minimization of the instantaneous temperature for periodic real-time tasks. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, pages 236–248, Apr. 2007.

27. J.-J. Chen, C.-Y. Yang, H.-I. Lu, and T.-W. Kuo. Approximation algorithms for multiprocessor energy-efficient scheduling of periodic real-time tasks with uncertain task execution time. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, pages 13–23, Apr. 2008.

28. J.-J. Chen, S. Wang, and L. Thiele. Proactive speed scheduling for real-time tasks under thermal constraints. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, pages 141–150, Apr. 2009.

29. J.-Y. Chung, J. W. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, 39(9):1156–1175, Sept. 1990.

30. J. Clausen. Branch and bound algorithms - principles and examples, 2003.

31. A. K. Coskun, T. S. Rosing, and K. Gross. Temperature management in multiprocessor SoCs using online learning. In *Proceedings of the Design Automation Conference*, pages 890–893, June 2008.

32. A. K. Coskun, T. S. Rosing, K. Whisnant, and K. Gross. Temperature-aware MPSoC scheduling for reducing hot spots and gradients. In *Proceedings of the Asia & South Pacific Design Automation Conference*, pages 49–54, Jan. 2008.

33. G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Book Company, NY, 1994.

34. U. Devi. An improved schedulability test for uniprocessor periodic task systems. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 23–30, July 2003.

35. R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: task graphs for free. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 97–101, Mar. 1998.

36. Embedded Microprocessor Benchmark Consortium. Website, 2011. `http://www.eembc.org`.

37. EPA Report to Congress on Server and Data Center Energy Efficiency. Website, 2007. `http://www.energystar.gov/ia/partners/prod_development/downloads/EPA_Report_Exec_Summary_Final.pdf`.

38. N. Fisher and S. Baruah. A polynomial-time approximation scheme for feasibility analysis in static-priority systems with bounded relative deadlines. In *Proceedings of the International Conference on Real-Time Systems*, pages 233–249, Apr. 2005.

39. A. Forsgren, P. Gill, and M. Wright. Interior methods for nonlinear optimization. *SIAM Review*, 44(4):525–597, Dec. 2002.

40. J. Fourier. *The Analytical Theory of Heat*. Cambridge University Press, 2009.

41. P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 199–208, June 2001.

42. P. Gai, L. Abeni, and G. Buttazzo. Multiprocessor DSP scheduling in system-on-a-chip architectures. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 231–238, June 2002.

43. S. H. Gunther, F. Binns, D. M. Carmean, and J. C. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, 5(1):1–9, Feb. 2001.

44. M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Transactions on Computers*, 44 (12):1443–1451, Dec. 1995.

45. T. He, S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J. Stankovic, T. Abdelzaher, J. Hui, and B. Krogh. VigilNet: An integrated sensor network system for energy-efficient surveillance. *ACM Transactions on Sensor Networks*, 2(1):1–38, 2006.

46. C.-M. Hung, J.-J. Chen, and T.-W. Kuo. Energy-efficient real-time task scheduling for a DVS system with a non-DVS processing element. In *Proceedings of the Real-Time Systems Symposium*, pages 303–312, Dec. 2006.

47. Intel Core Duo Processor and Intel Core Solo Processor on 65 nm Process Datasheet. Website, 2007. `http://www.intel.com/design/mobile/datashts/309221.htm`.

48. R. Jayaseelan and T. Mitra. Temperature aware task sequencing and voltage scaling. In *Proceedings of the International Conference on Computer-Aided Design*, pages 618–623, Nov. 2008.

49. R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the Design Automation Conference*, pages 275–280, June 2004.

50. E. Jensen, C. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *Proceedings of the Real-Time Systems Symposium*, pages 112–122, Dec. 1985.

51. J. Jonsson and K. G. Shin. Robust adaptive metrics for deadline assignment in distributed hard real-time systems. *Real-Time Systems*, 23(3):239–271, Nov. 2002.

52. H. Jung, P. Rong, and M. Pedram. Stochastic modeling of a thermally-managed multi-core system. In *Proceedings of the Design Automation Conference*, pages 728–733, June 2008.

53. T. Karkhanis and J. E. Smith. Automated design of application specific superscalar processors: An analytical approach. In *Proceedings of the International Symposium on Computer Architecture*, pages 402–411, 2007.

54. G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Proceedings of the Real-Time Systems Symposium*, pages 110–117, Dec. 1995.

55. X. Koutsoukos, R. Tekumalla, B. Natarajan, and C. Lu. Hybrid supervisory utilization control of real-time systems. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, pages 12–21, Mar. 2005.

56. J. C. Ku, S. Ozdemir, G. Memik, and Y. Ismail. Thermal management of on-chip caches through power density minimization. *IEEE Transactions on Very Large Scale Integration Systems*, 15(5):592–604, May 2007.

57. R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 23–32, Sept. 2006.

58. T.-W. Kuo and A. Mok. Load adjustment in adaptive real-time systems. In *Proceedings of the Real-Time Systems Symposium*, pages 160–171, Dec. 1991.

59. W.-C. Kwon and T. Kim. Optimal voltage allocation techniques for dynamically variable voltage processors. *ACM Transactions on Embedded Computing Systems*, 4(1):211–230, Feb. 2005.

60. E. L. Lawler and C. U. Martel. Scheduling periodically occurring tasks on multiple processors. *Information Processing Letters*, 7:9–12, Feb. 1981.

61. L.-F. Leung, C.-Y. Tsui, and W.-H. Ki. Simultaneous task allocation, scheduling and voltage assignment for multiple-processors-core systems using mixed integer nonlinear programming. In *Proceedings of the International Symposium on Circuits and Systems*, pages 309–321, May 2003.

62. H. Li, P. Shenoy, and K. Ramamritham. Scheduling messages with deadlines in multi-hop real-time sensor networks. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, pages 415–425, Mar. 2005.

63. Y. Li, D. Brooks, Z. Hu, and K. Skadron. Performance, energy, and thermal considerations for SMT and CMP architectures. In *Proceedings of the International Symposium on Computer Architecture*, pages 71–82, Feb. 2005.

64. P. Lim and T. Kim. Thermal-aware high-level synthesis based on network flow method. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 124–129, Oct. 2006.

65. G. M. Link and N. Vijaykrishnan. Thermal trends in emerging technologies. In *Proceedings of the International Symposium on Quality of Electronic Design*, pages 625–632, Mar. 2006.

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

67. J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, NJ, 2000.

68. Y. Liu, R. P. Dick, L. Shang, and H. Yang. Accurate temperature-dependent integrated circuit leakage power estimation is easy. In *Proceedings of the Design, Automation & Test in Europe Conference*, pages 204–209, Mar. 2007.

69. A. Mainwaring, J. Polastre, R. S. D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the International Workshop on Wireless Sensor Networks and Applications*, pages 88–97, Sept. 2002.

70. A. Mallik, J. Cosgrove, R. P. Dick, G. Memik, and P. Dinda. PICSEL: Measuring user-perceived performance to control dynamic frequency scaling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2008.

71. A. Masrur, S. Drossler, and G. Farber. Improvements in polynomial-time feasibility testing for EDF. In *Proceedings of the Design, Automation & Test in Europe Conference*, pages 1033–1038, Mar. 2008.

72. Matrix TCL Lite. Website, 2004. `http://www.techsoftpl.com/matrix/`.

73. E. Milchman. Intel dual-core FAQ. *Wired News*, July 2006.

74. B. Mochocki, D. Rajan, X. Hu, C. Poellabauer, K. Otten, and T. Chantem. Network-aware dynamic voltage and frequency scaling. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, pages 215–224, Apr. 2007.

75. A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, pages 75–84, May 2001.

76. R. Mukherjee, S. Öğrenci Memik, and G. Memik. Temperature-aware resource allocation and binding in high-level synthesis. In *Proceedings of the Design Automation Conference*, pages 196–201, June 2005.

77. F. Mulas, M. Pittau, M. Buttu, S. Carta, A. Acquaviva, L. Benini, and D. Atienza. Thermal balancing policy for streaming computing on multiprocessor architectures. In *Proceedings of the Design, Automation & Test in Europe Conference*, pages 734–739, Mar. 2008.

78. A. Mutapcic, S. Boyd, S. Murali, D. Atienza, G. D. Micheli, and R. Gupta. Processor speed control with thermal constraints. *IEEE Transactions on Circuits and Systems I*, 56(9):1994–2008, Sept. 2009.

79. L. Niu and G. Quan. Reducing both dynamic and leakage energy consumption for hard real-time systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 140–148, Sept. 2004.

80. P4040: QorIQ Embedded Multicore Processor. Website, 2009. `http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4040&fsrch=1`.

81. G. Paci, P. Marchal, F. Poletti, and L. Benini. Exploring "temperature-aware design" in low-power MPSoCs. In *Proceedings of the Design, Automation & Test in Europe Conference*, pages 838–843, Mar. 2006.

82. P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. *ACM Operating Systems Review*, 35(5):89–102, Oct. 2001.

83. C. Poellabauer and K. Schwan. Energy-aware traffic shaping for wireless real-time applications. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, pages 48–55, May 2004.

84. L. Pontryagin. *Mathematical Theory of Optimal Processes*. CRC Press, 1987.

85. G. Quan, Y. Zhang, W. Wiles, and P. Pei. Guaranteed scheduling for repetitive hard real-time tasks under the maximum temperature constraints. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 267–272, Oct. 2008.

86. J. Rabaey. *Low Power Design Essentials*. Springer, NY, 2009.

87. R. Rao and S. Vrudhula. Performance optimal processor throttling under thermal constraints. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 257–266, Oct. 2007.

88. R. Rao and S. Vrudhula. Efficient online computation of core speeds to maximize the throughput of thermally constrained multi-core processors. In *Proceedings of the International Conference on Computer-Aided Design*, pages 537–542, Nov. 2008.

89. R. Rao, S. Vrudhula, C. Chakrabarti, and N. Chang. An optimal analytical solution for processor speed control with thermal constraints. In *Proceedings of the International Symposium on Low Power Electronics & Design*, pages 292–297, Oct. 2006.

90. T. P. Ruggaber, J. W. Talley, and L. A. Montestruque. Using embedded sensor networks to monitor, control and reduce CSO events: A pilot study. *Environmental Engineering Science*, 24(2):172–182, Mar. 2007.

91. E. Seo, Y. Koo, and J. Lee. Dynamic repartitioning of real-time schedule on a multicore processor for energy efficiency. In *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, pages 69–78, Aug. 2006.

92. R. A. Serway. *Physics for Scientists & Engineers with Modern Physics*. Saunders College Publishing, 1990.

93. D. Seto, J. Lehoczky, L. Sha, and K. Shin. On task schedulability in real-time control systems. In *Proceedings of the Real-Time Systems Symposium*, pages 13–21, Dec. 1996.

94. D. Seto, J. Lehoczky, and L. Sha. Task period selection and schedulability in real-time systems. In *Proceedings of the Real-Time Systems Symposium*, pages 188–199, Dec. 1998.

95. C.-S. Shih and J. W. Liu. State-dependent deadline scheduling. In *Proceedings of the Real-Time Systems Symposium*, pages 3–14, Dec. 2002.

96. K. Shin and C. Hou. Design and evaluation of effective load sharing in distributed real-time systems. *IEEE Transactions on Parallel & Distributed Systems*, 5(7):704–719, July 1994.

97. G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel & Distributed Systems*, 4(2):175–187, Feb. 1993.

98. K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the International Symposium on Computer Architecture*, pages 2–13, June 2003.

99. J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *Proceedings of the International Symposium on Computer Architecture*, pages 520–531, June 2005.

100. C. Sun, L. Shang, and R. P. Dick. Three-dimensional multi-processor system-on-chip thermal optimization. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 117–122, Oct. 2007.

101. E. Uysal-Biyikoglu, B. Prabhakar, and A. E. Gamal. Energy-efficient packet transmission over a wireless link. *IEEE Transactions on Networking*, 10(4): 487–499, Aug. 2002.

102. R. Viswanath, V. Wakharkar, A. Watwe, and V. Lebonheur. Thermal performance challenges from silicon to systems. *Intel Technology Journal*, 4(3): 1–16, Aug. 2000.

103. P. Wan and M. Lemmon. Distributed flow control using embedded sensor-actuator networks for the reduction of combined sewer overflow (CSO) events. In *Proceedings of the Conference on Decision and Control*, pages 1529–1534, Dec. 2007.

104. F. Wang, K. Ramamrithnam, and J. Stankovic. Determining redundancy levels for fault-tolerant real-time systems. *IEEE Transactions on Computers*, 44(2):292–301, Feb. 1995.

105. H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik. Orion: a power-performance simulator for interconnection networks. In *Proceedings of the International Symposium on Microarchitecture*, pages 294–305, Nov. 2002.

106. S. Wang and R. Bettati. Delay analysis in temperature-constrained hard real-time systems with general task arrivals. In *Proceedings of the Real-Time Systems Symposium*, pages 323–332, Dec. 2006.

107. S. Wang and R. Bettati. Reactive speed control in temperature-constrained real-time systems. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 161–170, July 2006.

108. X. Wang and M. D. Lemmon. Self-triggered feedback control systems with finite-gain l2 stability. *IEEE Transactions on Automatic Control*, 45(3):452–467, Mar. 2009.

109. R. West and C. Poellabauer. Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In *Proceedings of the Real-Time Systems Symposium*, pages 239–248, Dec. 2000.

110. Xbox 360 Xenon. Website, 2006. `http://domino.research.ibm.com/comm/research_projects.nsf/pages/multicore.Xbox360.html`.

111. Y. Xie and W.-L. Hung. Temperature-aware task allocation and scheduling for embedded multiprocessor systems-on-chip (MPSoC) design. *Journal of VLSI Signal Processing*, 45(3):177–189, Dec. 2006.

112. Y. Yang, Z. Gu, C. Zhu, R. P. Dick, and L. Shang. ISAC: Integrated space and time adaptive chip-package thermal analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(1):86–99, Jan. 2007.

113. F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the Symposium on Foundations of Computer Science*, pages 374–382, Oct. 1995.

114. J. Yi, C. Poellabauer, X. Hu, J. Simmer, and L. Zhang. Energy-conscious co-scheduling of tasks and packets in wireless real-time environments. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, pages 265–274, Apr. 2009.

115. S. Zhang and K. S. Chatha. Approximation algorithm for the temperature-aware scheduling problem. In *Proceedings of the International Conference on Computer-Aided Design*, pages 281–288, Nov. 2007.

116. T. Zhou, X. Hu, and E.-M. Sha. Probabilistic performance estimation for real-time embedded systems. In *Proceedings of the International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 83–88, Mar. 1999.