ENERGY EFFICIENT SCHEDULING FOR REAL-TIME SYSTEMS

A Dissertation

by

NIKHIL GUPTA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2011

Major Subject: Computer Science

Energy Efficient Scheduling for Real-Time Systems

ENERGY EFFICIENT SCHEDULING FOR REAL-TIME SYSTEMS

A Dissertation

by

NIKHIL GUPTA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Rabi N. Mahapatra
Committee Members,	Jiang Hu
	Eun Jung Kim
	Duncan M. Walker
Head of Department,	Duncan M. Walker

December 2011

Major Subject: Computer Science

ABSTRACT

Energy Efficient Scheduling for Real-Time Systems. (December 2011)

Nikhil Gupta, B.Tech, Indian Institute of Technology, Guwahati

Chair of Advisory Committee: Dr. Rabi N. Mahapatra

The goal of this dissertation is to extend the state of the art in real-time scheduling algorithms to achieve energy efficiency. Currently, Pfair scheduling is one of the few scheduling frameworks which can optimally schedule a periodic real-time taskset on a multiprocessor platform. Despite the theoretical optimality, there exist large concerns about efficiency and applicability of Pfair scheduling in practical situations. This dissertation studies and proposes solutions to such efficiency and applicability concerns. This dissertation also explores temperature aware energy management in the domain of real-time scheduling. The thesis of this dissertation is:

> *the implementation efficiency of Pfair scheduling algorithms can be improved. Further, temperature awareness of a real-time system can be improved while considering variation of task execution times to reduce energy consumption.*

This thesis is established through research in a number of directions. First, we explore the applicability of Dynamic Voltage and Frequency Scaling (DVFS) feature of the underlying platform, within Pfair scheduled systems. We propose techniques to reduce energy consumption in Pfair scheduling by integrating DVFS into the optimal Pfair scheduling algorithm. The integration was achieved by modifying the original Pfair scheduling algorithm to dynamically vary the weight of a task. Our experimental evaluation with synthetic and real benchmarks shows up to 66% sav-

ings in energy consumption compared to the basic Pfair scheduling algorithm. Next, we explore the problem of quantum size selection in Pfair scheduled systems so that runtime overheads are minimized. We study the system overhead as a function of quantum size and present quotient search (QS) – a quantum size selection heuristic to reduce the overall scheduling overhead of Pfair scheduling. Our results show that there is a notable difference in the runtime overhead (3% on the average), between QS and other quantum size selection strategies. We also propose a hardware design for a central Pfair scheduler core in a multiprocessor system to minimize the overheads and energy consumption of Pfair scheduling. Three different implementation schemes for the Pfair scheduling algorithm were considered: replicated software scheduler running on each processor, single software scheduler running on a dedicated processor and the proposed hardware scheduler. Experimental evaluation shows that the hardware scheduler outperforms the other two implementation schemes by orders of magnitude in terms of scheduling delay and energy consumption. Finally, we propose a temperature aware energy management scheme for tasks with varying execution times. The proposed scheme, TA-DVS, reduces temperature constraint violations by 18.9% on the average, compared to existing schemes without adversely affecting energy consumption.

DEDICATION

To my parents.

ACKNOWLEDGMENTS

First, I want to thank my advisor, Dr. Rabi Mahapatra for his kindness, wisdom and advice throughout graduate school. He has continuously inspired and challenged me to do well. This thesis would not have come to fruition without his guidance. I would also like to thank my thesis committee members, Dr. Duncan Walker, Dr. Eun Jung Kim and Dr. Jiang Hu for agreeing to be on the committee and for providing valuable inputs and criticisms on my research. Special thanks to Dr. Riccardo Bettati for his valuable guidance and comments on my work. The staff at Texas A&M's Department of Computer Science have been very nice and helpful during the course of graduate school.

My work would not have been possible without the support of my family. My parents have always been an unending source of inspiration. I would also like to thank my sister, Neha Bansal, for taking care of my mother during my absence. My uncle Dr. Deepak Gupta has continuously inspired and supported me throughout my stay in the U.S.

Finally, I would like to thank my friends Suman, Gaurav, Suneil, Sandip and Vinita for making the five years spent at College Station an unforgettable journey. During time away from home, they have been a constant source of support, motivation and happiness.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

FIGURE Page

1. INTRODUCTION

Although technology scaling has yielded tremendous performance benefits, it has also led to concerns related to power density and energy consumption. Increasing power densities has necessitated efficient energy management for not only mobile, battery operated devices, but also high performance computers connected directly to the power grid [29] [37]. As computation demands increase the energy consumption of computer systems is expected to increase requiring energy management at both hardware and software level.

At the same time technology scaling has led to reliability concerns by pushing current CMOS materials to their physical reliability limits [45]. The reliability concerns are two-fold. Firstly, the use of ever decreasing threshold voltages has led to increasing transient fault rates [1] [7]. Secondly, temperature related electronic wear-out phenomena have reduced the expected lifetime of processing elements [50] [52]. Electronic wear-out is caused by several physical degradation phenomena, including electro-migration, hot carrier injection and negative bias temperature instability, which are intensified by lower feature sizes, higher power densities and higher operating temperatures [13]. The problem of electronic wear out is even more pronounced in the latest multicore chips with multiple processing cores [23, 28]

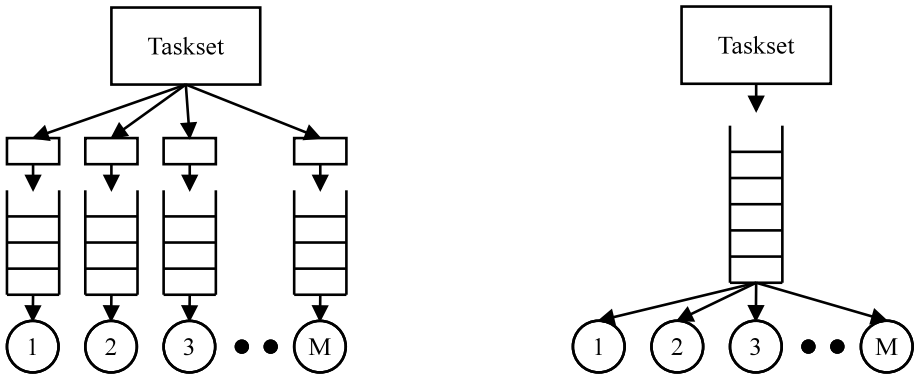
Increasing power densities has also resulted in the processor industry relying on increasing the number of processing elements on the chip to increase processor performance. However, processor performance does not scale linearly with the number of processing cores due to the effects of task migration, synchronization, load balancing etc. This trend necessitates efficient operating systems for managing resources on a multiprocessor system.

This dissertation follows the style of *IEEE Transactions on Parallel and Distributed Systems*.

1.1 Real-Time Scheduling

Real-Time systems form an important class of computing. The distinguishing feature of real-time systems in comparison to non-real-time systems is that besides being logically correct, real-time systems must also be temporally correct. Thus, in real-time computing, programs need to produce logically correct results within specific time frames. The timing constraints in a real-time system can be conveniently thought of a timing deadlines within which computation must finish for it to be useful. Process control systems, weather information systems and air traffic control are some examples of real-time systems. Consider a weather satellite system for example. To send satellite imagery to a base station the satellite system must perform two high level activities or tasks: capture images of regions of interest on the earth; and send the images to the base station. Both these tasks need to be performed repeatedly with a minimum frequency so that a base station has updated and fresh information about the monitored region. The frequency requirements translate to deadline constraints that the high level tasks must follow.

1.2 Multiprocessor Real-Time Scheduling



(a) Partitioned scheduling (b) Global Scheduling

Fig. 1.1. Multiprocessor Scheduling Schemes

Increasing the number of cores on a chip is currently the most popular method of increasing processor performance. This trend makes the study of efficient multiprocessor scheduling algorithms extremely important. Although the real-time scheduling theory is well established for uniprocessor systems, multiprocessor real-time scheduling has received comparatively lesser attention. For example, well known optimal uniprocessor scheduling algorithms e.g. EDF and LLF have been thoroughly studied. However, only a few multiprocessor scheduling algorithms can guarantee optimality. Multiprocessor real-time scheduling algorithms can broadly be divided into two categories: Partitioned scheduling and Global scheduling, Figure 1.1. In partitioned scheduling, Figure 1.1a, the real-time taskset is partitioned into M subsets, one for each processor in the multiprocessor platform. Tasks can only execute on the processor assigned. Subsequently, uniprocessor scheduling algorithms like EDF etc. can be applied on each of the subsets. In contrast, global scheduling, Figure 1.1b, uses a single queue of tasks. Each processor dequeues a tasks from the same global queue when it needs a new task to execute. There are both advantages and disadvantages of using partitioned or global scheduling. Table 1.1 lists some of the advantages and disadvantages of each scheduling scheme. This dissertation proposes techniques to mitigate the disadvantages of Global scheduling mentioned in bold in the table.

Pfair (Proportionate fairness) is an optimal scheduling algorithm for multiprocessor real-time systems [11]. The Pfair scheduling algorithm optimally solves the problem of scheduling periodic tasks on a multiprocessor system in polynomial time. This problem was previously viewed as NP-hard by most researchers [4]. Pfair scheduling can correctly schedule a periodic taskset with utilization M upon a multiprocessor system with M processors. At the same time, Dynamic voltage and frequency scaling (DVFS) is a widely used technique for reducing energy consumption. Although well studied for uniprocessor systems, DVFS techniques and slack management schemes for multiprocessor systems are still immature. This dissertation studies DVFS techniques and slack management for the Pfair scheduling algorithm. The overheads in-

Table 1.1
Comparison of Partitioned and Global Scheduling

Partitioned Scheduling	**Global Scheduling**
Advantages	Advantages
• The multiprocessor scheduling problem is broken down into simpler uniprocessor scheduling problems. • Well studied uniprocessor scheduling algorithms can be applied.	• Better suited for dynamic task systems because of the absence of taskset partitioning. • Optimal global scheduling algorithms exist.
Disadvantages	Disadvantages
• The Bin-packing problem involved in partitioning is NP-hard. • Partitioned scheduling algorithms are sub-optimal.	• **Although theoretically optimal, practical implementation can be inefficient.** • **Scheduling overhead can be large.** • **Can result in a large number of context switches and task migrations.**

volved in the practical implementation of the Pfair scheduling algorithm are largely dependent on the quantum size used. The quantum size used in Pfair scheduling must be well balanced according the the set of tasks running on the system. This dissertation studies the problem of choosing a good quantum size for Pfair scheduling to reduce these runtime overheads.

Despite the theoretical optimality of the Pfair scheduling algorithm, it can be inefficient when implemented in serialized software. In Pfair scheduling, the computation involved in determining the runtime schedule of the tasks grows linearly with

the number of tasks in the system. This creates uncertainty about the overall utilization of the system and thereby motivates the hardware implementation of Pfair scheduling. This dissertation studies the design of a Hardware Pfair Scheduler to make Pfair scheduling fast, energy efficient and predictable.

In recent times, the power density of microprocessors has doubled every three years. This increase in power density has led to higher temperatures that directly affect reliability and cooling costs. Current estimates predict that cooling costs will rise at $1-$3 per watt of heat dissipated [53]. This dissertation will also study temperature aware energy management schemes for real-time scheduling.

1.3 Research Focus

This dissertation focuses on the efficient use of Dynamic Voltage and Frequency Scaling (DVFS) features in modern processors for energy and temperature management. DVFS is a recent development in the microprocessor industry whereby the processor can dynamically change its operating frequency to save energy at runtime. Although easily employed in general purpose systems, DVFS needs special considerations when employed in real-time systems. DVFS saves energy by reducing operating frequency thereby quadratically reducing energy consumption while only linearly increasing processing time. However, in real-time systems, any increase in processing time must be carefully examined so that no task deadlines are missed.

1.4 Contributions

The contributions of this dissertation are described below:

- **Power Aware Pfair Scheduling:** In this work we present a scheme to integrate DVFS into the optimal Pfair scheduling algorithm. The integration of DVFS was achieved by modifying the original Pfair scheduling algorithm to dynamically vary the weight of a task. Experimental evaluation with synthetic

and real benchmarks shows up to 66% energy savings compared to the basic Pfair scheduling algorithm.

- **Choosing a Good Quantum Size for Pfair Scheduling:** In this work we provide a method to choose a good quantum size for Pfair scheduling. We study the system overhead as a function of quantum size and present *quotient search* (QS) – a quantum size selection heuristic to reduce the overall scheduling overhead of Pfair scheduling. Our results show that there is a notable difference in the runtime overhead (upto 10%), between QS and existing quantum size approaches.

- **Hardware Pfair Scheduler:** This work presents the design and implementation of a low- power hardware scheduler for multiprocessor system-on-chips. The Pfair scheduling algorithm is considered with three different implementation schemes: replicated software scheduler running on each processor, single software scheduler running on a dedicated processor and the proposed hardware scheduler. Experimental evaluation with benchmarks shows that the hardware scheduler outperforms the other two schemes in terms of energy consumption by an order of magnitude of 10^5 and scheduling delay by an order of magnitude of 10^3.

- **Temperature Aware Dynamic Power Management:** In this work, we present a *best effort* Temperature Aware Dynamic Voltage and frequency Scaling (TA-DVS) scheme for real-time task scheduling using run time slack management. We experimentally conclude that when the system utilization is within a certain limit, energy management alone can satisfy system temperature constraint. Our proposed scheme, TA-DVS, reduces temperature constraint violations for tasksets with unconstrained utilizations by 18.9% on the average, compared to existing approaches.

1.5 Organization

The rest of this dissertation is organized as follows. Section 2 discusses the assumption and system model used in this dissertation. Section 3 describes the basics of Pfair scheduling. In Section 4, we present our work on Power Aware Pfair Scheduling to support DVFS on top of the Pfair scheduling algorithm. Section 5 presents the analysis of overheads involved in Pfair scheduling and presents techniques to choose a good quantum size for Pfair scheduling. In Section 6 a hardware design for Pfair scheduler is described. Section 7 explores the similarities between temperature awareness and energy management and presents a technique for temperature aware DVFS when running tasks with varying execution times. Finally , Section 8 concludes this dissertation with directions for future research.

2. SYSTEM OVERVIEW AND ASSUMPTIONS

2.1 Task Model

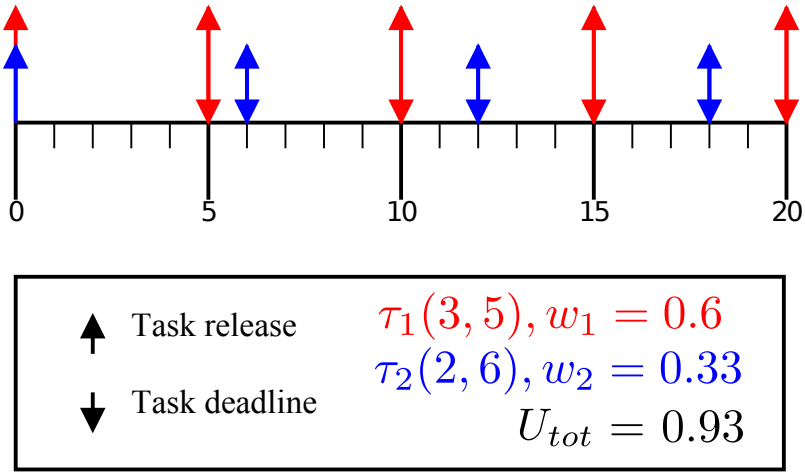


Fig. 2.1. Task Model

We assume a periodic task model where each task can be preempted and all tasks are independent. Formally Γ denotes a task set where each task $\tau_i \in \Gamma$ is denoted by $\{e_i, p_i\}$, $i = 1, 2, \ldots, N$. Here $e_i, p_i \in \mathbb{N}$ denote the worst case execution time (WCET) at highest processor frequency and period of the task respectively. The weight of a task is defined as $w_i = e_i/p_i$ and $0 < w_i < 1, \forall i$. The worst case utilization of a taskset is given by $U_{tot} = \sum_{i=i}^{n} e_i/p_i$. Each invocation of the task is called a job. The j^{th} job of task τ_i is denoted by τ_{ij} and has its release time and deadline at $(j-1) \cdot p$ and $j \cdot p$ respectively. The actual execution time of job τ_{ij} is represented by a_{ij}. For e.g. Figure 2.1 shows the release times and deadlines of two tasks $\tau_1 = (2, 5)$ and $\tau_2 = (2, 6)$. The total utilization of this taskset is 0.93.

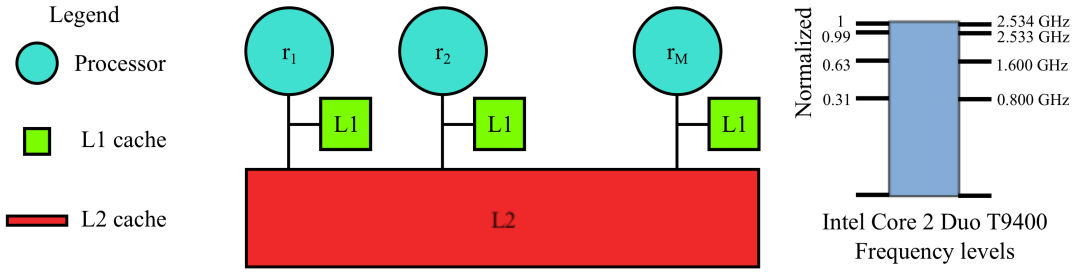


Fig. 2.2. Platform Model Representation

2.2 Platform Model

In this research we model the multiprocessor platform as a shared memory symmetric multiprocessor with M processors, as in Figure 2.2. We also assume that each processor supports a set of L discrete speed levels, denoted by $S = \{s_1, \ldots, s_L\}$. Changing the processor speed is accompanied by a time overhead given by:

$$O(s_i, s_j) = C + K \cdot \mid s_i - s_j \mid \tag{2.1}$$

where s_i and s_j are the old and new frequencies respectively [65]. C and K are technology dependent constants. In our system, at any scheduling instant, we decide to use DVFS only if the energy savings due to DVFS are greater than the overhead energy corresponding to the frequency change.

2.3 Energy Model

Here, we describe the energy model used in this dissertation. The total power consumption in a processor can be modeled as:

$$P = P_s + \hbar(P_{ind} + P_d) \tag{2.2}$$

where P_s is the static power consumption [65]. P_{ind} and P_d are the frequency independent and dependent components of dynamic power consumption respectively. $\hbar = 0$ if the system is in sleep state and $\hbar = 1$ otherwise. In this work, we concentrate on the dynamic power consumption which can be modeled as:

$$P = P_{ind} + C_{ef} f^m \tag{2.3}$$

where f is the frequency of operation [65]. m and C_{ef} are system dependent constants which, for the purpose of this analysis have been assumed to be equal to 3 and 1 respectively. Therefore the energy consumption of a job, τ_{ij} can be modeled as:

$$E_{ij} = e_{ij}(P_{ind} + f^3) \tag{2.4}$$

where e_{ij} is the execution time of τ_{ij}. Equation 2.4 shows that DVFS can quadratically reduce energy consumption while only linearly increasing task execution time.

3. OVERVIEW OF PFAIR SCHEDULING

With the proliferation of multiprocessor systems in the market, multiprocessor real-time scheduling algorithms have received renewed attention in research. Pfair (proportionately fair) scheduling is one such scheduling framework which has received considerable attention. Since hard real-time systems require strong guarantees on deadline satisfaction, traditional optimal uniprocessor scheduling algorithms like EDF and LLF are sub-optimal when applied to multiprocessor systems. Pfair is one of the few scheduling frameworks that guarantees optimal use of a multiprocessor platform (i.e. it correctly schedules any taskset with total utilization M upon M processors) when scheduling periodic tasks. A number of Pfair scheduling algorithms exist today, e.g. PF [11], PD [12], PD2 [3], and considerable work has been done on practical implementation of Pfair scheduling [17], [55].

The concept of Pfairness (Proportionate fairness) was proposed by Baruah et al. to solve the multiprocessor periodic scheduling problem [11]. The solution to the multiprocessor periodic scheduling problem was significant because the problem was previously viewed by most researchers as NP-hard. In the Pfair scheduling algorithm, time is discretized into slots, Figure 3.1. The unit time interval $[t, t+1)$ is called the slot t. During each slot, only a single task may execute on a processor. However, a task may execute on different processors during different slots. Hence, task migration is allowed while task parallelism is not. In Pfair scheduling, tasks make progress at a rate approximately equal to the weight of the task which is defined as the ratio of the task's WCET and period. The deviation of a task's actual progress rate from the ideal rate of progress is measured in terms of the parameter *lag* which is computed at the beginning of each scheduling slot. In an ideal schedule, the lag of each task would remain equal to zero at any given instant of time. Figure 3.2 shows the comparison of an ideal fluid schedule to a Pfair schedule for three tasks with weight 1/4, 1/4 and 1/2 respectively [36]. The lag trend is shown on the right. Pfair scheduling ensures

that the rate of progress of a task does not deviate too much from the constant ideal rate of progress. The deviation is modeled by *lag* which is formally defined as:

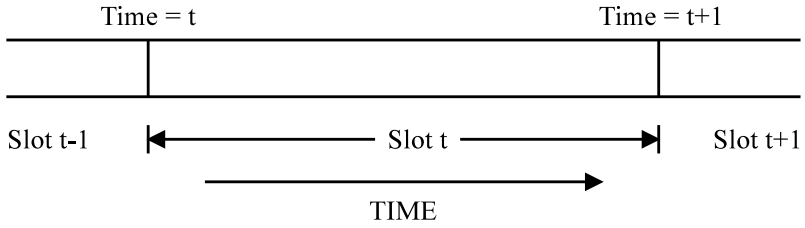


Fig. 3.1. Discretization of Time into Slots

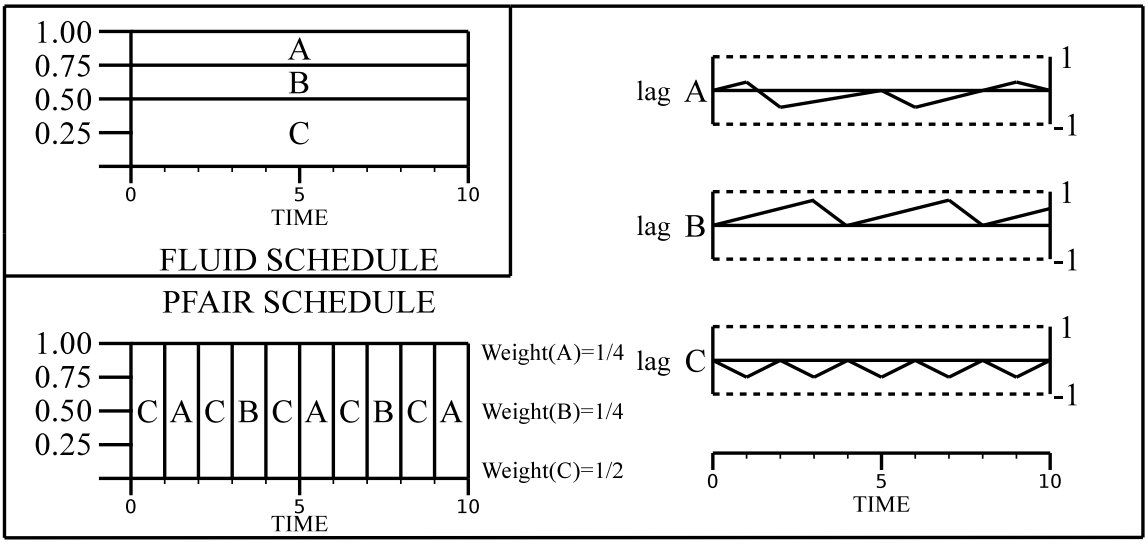


Fig. 3.2. Comparison of Fluid and Pfair Schedules

$$lag(\tau_i, t) = w_i \times t - allocated(\tau_i, t) \tag{3.1}$$

where $allocated(\tau_i, t)$ is the amount of processor time allocated to τ_i in $[0, t)$. A schedule is said to be *Pfair* iff:

$$\forall \tau_i, t : t \in \mathbb{N} : -1 < lag(\tau_i, t) < 1 \tag{3.2}$$

It is important to note that in a Pfair schedule for a periodic taskset none of the task deadlines are missed. At the end of each period, $w_i \times t$ is an integer. Since $-1 < lag(\tau_i, t) < 1$, $lag(\tau_i, t) = 0$. Hence at each period boundary, $allocated(\tau_i, t) = e_i$ and the corresponding deadline is not missed. Pfairness is a strictly stronger requirement that periodicity. While a periodic schedule only requires that $lag = 0$ at period boundaries, a *Pfair* schedule additionally requires that $-1 < lag < 1$ at all times.

The first scheduling algorithm which guaranteed Pfair correctness for a periodic taskset was the PF algorithm [11]. In the PF algorithms, a parameter called the characteristic symbol, $\alpha_t(\tau_i)$ is computed at the beginning of each slot. The characteristic symbol for a task τ_i at time t is defined as:

$$\alpha_t(\tau_i) = \text{sign}(w_i \cdot (t+1) - \lfloor w_i \cdot t \rfloor - 1) \in \{+, 0, -\} \tag{3.3}$$

Based on the values of lag and $\alpha_t(\tau_i)$, the input taskset is partitioned into three disjoint sets namely: *urgent*, *contending* and *tnegru*.

$$urgent = \{\tau_i : \alpha_t(\tau_i) \neq - \wedge lag(\tau_i, t) > 0\} \tag{3.4}$$

$$tnegru = \{\tau_i : \alpha_t(\tau_i) \neq + \wedge lag(\tau_i, t) < 0\} \tag{3.5}$$

$$cont = \{\tau_i : \tau_i \notin tnegru \wedge \tau_i \notin urgent\} \tag{3.6}$$

Intuitively, the characteristic symbol and lag parameters allow the PF algorithm to make decisions on which tasks definitely need an allocation during the slot (represented by the subset *urgent*), or else their lag will become $>= 1$ at the end of slot. Similarly, the tasks in subset *tnegru* should not be allocated in the current slot or else their lag will become $<= -1$ at the end of the slot. The remaining tasks are assigned to the contending set.

Once the sets are populated, the contending set is sorted according to a total order defined over the characteristic substrings of the tasks. The characteristic substring of task τ_i at time t is defined as

$$\alpha(\tau_i, t) = \alpha_t(\tau_i)\alpha_{t+1}(\tau_i)\alpha_{t+2}(\tau_i)\ldots\alpha_{t+k}(\tau_i) \tag{3.7}$$

where $\alpha_{t+k}(\tau_i) = 0$ and $\alpha_{t+k'}(\tau_i) \neq 0$ for $k' >= 0, k' < k$. The lexicographic $(+ > 0 > -)$ ordering of $\alpha(\tau_i, t)$ defines total order on contending set. The tasks are selected for execution in the current slot as follows: All tasks in *urgent* set are selected. Greatest (based on the total order) tasks from the *contending* set are selected until all processors are occupied or the contending set is empty. No task from the *tnegru* set is selected.

During each slot, the Pfair algorithm performs the following operations:

1. Calculate lag and alpha of each task for the current time slot.
2. Partition the taskset into urgent, tnegru and contending subsets.
3. Compute the characteristic substring for each contending task and define the total order on the contending set.
4. Schedule each task from the urgent set on a different processor. For the remaining processors, select the greatest (based on the total ordering) tasks from the contending set and schedule them.

Pfair scheduling algorithms ensure near-ideal rate of progress by breaking a task into quantum length subtasks. Each subtask must execute within its window of eligibility. A subtask's release time and deadline are at the beginning and end of its window respectively. Pfair scheduling algorithms prioritize subtasks by their deadlines. The basic difference between PF [11], PD [12] and PD2 [3] is in the way in which deadline ties are broken. The PF algorithm looks at future subtask deadlines to break ties. This is intuitively expensive and hence the running time of the PF

algorithm is not linear in the size of the input. The PD, and PD2 algorithms use a constant time tie breaking procedure resulting in a runtime of $O(\min(M \lg N, N))$. While the PD scheduling algorithm uses four tie-break parameters, the PD^2 scheduling algorithm uses only two tie break parameters [3].

Optimal uniprocessor scheduling algorithms like EDF and LL are sub-optimal when applied to multiprocessor systems. The sub-optimality arises mainly due to the NP-hardness of the bin-packing procedure when the taskset is partitioned. On the other hand, Pfair scheduling guarantees optimal use of the multiprocessor platform, i.e. any taskset with total utilization M will be correctly scheduled by a Pfair scheduling algorithm. The optimality of Pfair scheduling algorithms is primarily due to the fact that each task is broken into quantum length subtasks which can be scheduled independently of each other (but not in parallel). Although the slot based nature of Pfair scheduling leads to theoretical optimality, it is a concern for practical implementation. A job may be preempted/migrated multiple times during its execution. Also, the scheduler overhead in Pfair scheduling can be large due to the per-slot scheduling scheme.

Pfair scheduling offers some unique benefits over partitioning approaches [55]. Firstly, Pfair scheduling naturally support dynamic task systems. In Pfair scheduling tasks can join and leave the system at period boundaries as long as the utilization of the system remains below M. On the other hand, in partitioned systems, the task partitioning algorithm must be run each time a new task joins to systems to determine if the new taskset is feasible. Secondly, the fairness guarantees of Pfair scheduling temporally isolate one application from another, since tasks are guaranteed to receive their share of the processor even when an application overruns its worst case execution time. Further, Pfair scheduling can offer peaceful degradation within the system during failures. If some of the processors within a multiprocessor system experience faults and need to be turned off, then tasks can be reweighted to adapt to the reduced capability of the system.

4. POWER AWARE PFAIR SCHEDULING IN MULTIPROCESSOR REAL TIME SYSTEMS

Modern embedded systems are increasingly being used in mission critical applications such as avionics, complex process control and space applications. The nature of these applications demands real-time performance guarantees. To meet these performance demands, multiprocessor real-time systems are widely being adopted. In the near future, multiprocessor systems are also expected to be widely used in battery powered devices. The increased performance of multiprocessor systems comes at the cost of higher energy consumption. At the same time, higher heat dissipation caused by increased energy consumption decreases system reliability. Thus it is extremely important to have efficient power management in multiprocessor real-time systems.

Although power management on uniprocessor systems has been widely studied, it has received relatively lesser focus in the multiprocessor domain. Earliest Deadline First (EDF) is an optimal scheduling algorithm in uniprocessor systems but it is suboptimal when applied in multiprocessor domain. Baruah et al. [11] proposed the Pfair scheduling algorithm which is optimal for multiprocessors. However, Pfair scheduling is not power aware and hence cannot benefit from the Dynamic Voltage and Frequency Scaling (DVFS) features of today's Multiprocessor System on Chips (MPSoC). Dynamic frequency scaling can cubically reduce power consumption while only linearly increasing task latencies. This makes frequency scaling an attractive choice for runtime power management. Recent works by Zhu et al. [65] and Chen et al. [20] have used dynamic voltage and frequency scaling to reduce energy consumption in multiprocessor real-time systems. However, none of these schemes work in conjunction with an optimal multiprocessor real-time scheduling algorithm.

In this research, we introduce Power Aware Pfair (PAPF), a dynamic power management (DPM) scheme for multiprocessor real-time system using the Pfair scheduling algorithm, Figure 4.1. Our work introduces the notion of DVFS in the Pfair

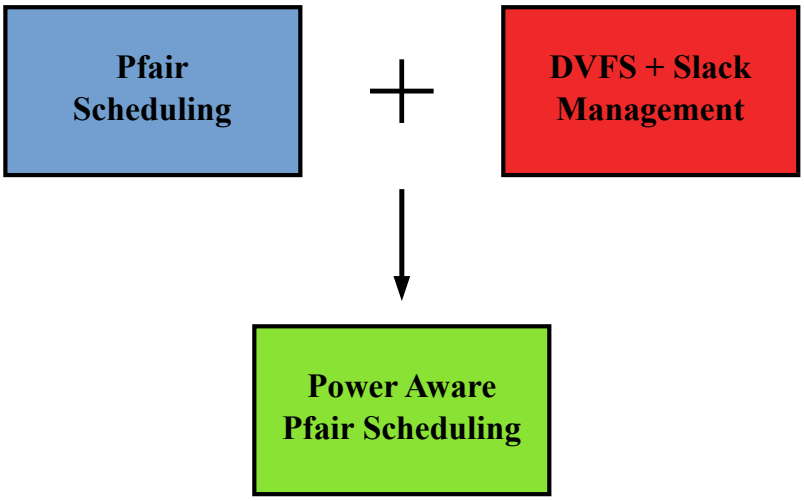


Fig. 4.1. PAPF Overview

scheduling algorithm to make it power aware. The notion of DVFS was introduced in Pfair by dynamically increasing the weight of a task whose execution time is being increased due to DVFS. The increased weight of task ensures that the task remains punctual even with the increase in its execution time. We performed detailed simulations to evaluate the effectiveness of our scheme. The overheads of DVFS and Pfair scheduling algorithm were considered and analyzed in the experiments. Our scheme results in up to 66% saving in energy consumption compared to the basic Pfair scheduling algorithm.

The rest of the section is organized as follows: Section 4.1 discusses related work. Section 4.2 discusses the preliminaries of power aware scheduling. Our scheme of integrating DVFS with Pfair algorithm is presented in Section 4.3. Section 4.4 discusses the correctness of the PAPF scheduling algorithm. Section 4.5 presents task assignment heuristics to reduce overheads involved in PAPF. Results and discussions are presented in Section 4.6. Section 4.7 concludes the section.

4.1 Related Work

Dynamic power management through frequency scaling has been well studied in literature. In uniprocessor systems, the optimal EDF algorithm has been widely adopted for DPM. However, there is no solution to the problem of DPM using optimal multiprocessor scheduling algorithm for periodic real-time tasksets. Previous work on DPM in multiprocessor systems is either scheduling algorithm oblivious or has focused on sub-optimal scheduling algorithms. Selected works closely related to our work on integration of DPM in multiprocessor scheduling algorithm are discussed here.

Zhu et al. in [65] studied the problem of DPM in Multiprocessor systems. In their scheme, dynamic voltage scaling was used based on shared slack reclamation to achieve dynamic power management. Their work considers non-preemptive scheduling where a task runs to completion once it starts executing. At the same time, they use the Longest Task First scheduling algorithm which is sub-optimal in terms of schedulable utilization for multiprocessor scheduling. They assume frame based task sets where all tasks share a common deadline. In contrast, our solution considers the more flexible periodic task set model and uses the optimal Pfair scheduling algorithm.

In [44], Li studied the problem of energy minimization in a multiprocessor real-time system as a combinatorial optimization problem. Unlike the more generic periodic task model that we assume, this work assumes the frame based task model. At the same time, this work considers an ideal task execution model where task execution time is fixed. We consider the more generic scenario where the execution time of a time may vary across time.

Anderson et al. developed a cache aware Pfair based scheduling scheme [5] for multicore platforms. This scheme avoids co-scheduling tasks which may thrash the L2 cache and increase the L2 cache to memory traffic. Their work is similar to ours in that it helps in reducing energy consumption by reducing L2 cache to memory

traffic. However, their work does not directly consider energy reduction through DVFS.

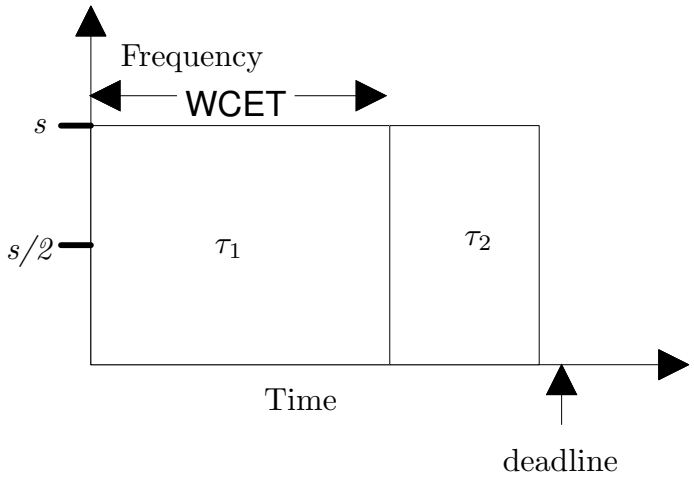
Mishra et al. proposed a scheduling algorithm oblivious, static and dynamic power management technique in [46]. Their scheme works for preemptive scheduling and does not allow task migration. Unlike our work, the availability of continuous frequency levels for speed change is assumed and the overheads involved with DVFS are not considered.

The lack of power aware optimal multiprocessor scheduling algorithms restricts us from providing comparison results with other approaches. Instead, we provide detailed simulation results with varying system and task characteristics for real and synthetic benchmarks compared to the basic Pfair scheduling algorithm.

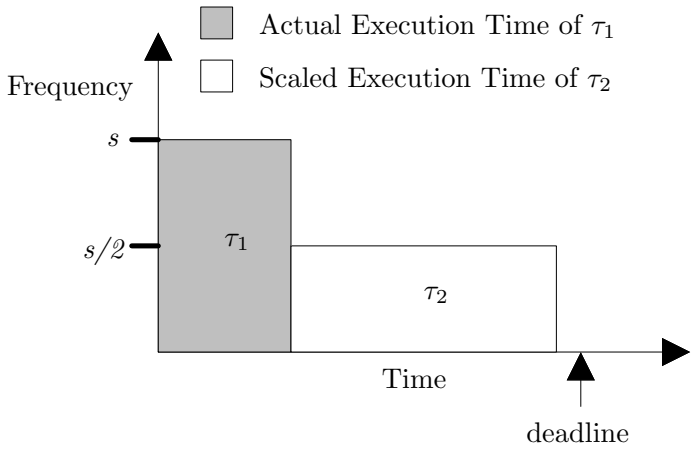
4.2 Power Aware Scheduling

In this section, we describe power aware scheduling in general and how DVFS is integrated with a real-time scheduling algorithm. Dynamic Voltage and Frequency Scaling is a well known technique for dynamic power management in clocked circuits. Often at run time, idle CPU periods occur when the CPU is not executing any task. These idle periods are termed as slack and can be used to reduce the frequency of the processor by using DVFS. In our proposed technique, we associate frequencies with tasks and the process of using slack to run a task with a lower frequency is termed as scaling down the task. Slack is created in the system due to two primary reasons which can be categorized as follows: 1. *Static Slack:* Static slack arises whenever the utilization of the taskset being executed is lesser than the schedulable utilization of the scheduling algorithm. Roughly speaking, this means that the load on the system is lesser than the maximum load that the system can handle. The extra room is termed as static slack. 2. *Dynamic Slack:* Dynamic slack arises because in real systems the worst case execution time (WCET) of a task must be considered for real-time scheduling, while actual execution time of the task can be as small as

only 50% of the WCET [51]. The idle period created whenever a job finishes earlier than its WCET, gives rise to dynamic slack. An example of using DVFS to scale down a task is shown in Figure 4.2. Here the white box in 4.2b(b) shows the reduced frequency execution of τ_2 utilizing the idle period generated by early completion of task τ_1, shown by the shaded box.



(a) Schedule Based on WCET



(b) Schedule Based on Early Finish of τ_1

Fig. 4.2. Slack Generation Due to Early Completion

Our proposed dynamic power management scheme utilizes both static and dynamic slack available in the system for power management. Our scheme could utilize the available slack most efficiently if it were possible to run the processor at con-

tinuously varying frequencies. However practical constraints restrict the variation of processor frequency to a few predefined discrete steps, thus limiting the manner in which we utilize system slack.

4.3 Power Aware Pfair Scheduling

In this section, we describe our technique of integrating DVFS in the Pfair scheduling algorithm. We call the Power Aware version of the Pfair algorithm, PAPF. The implementation of slots are timer dependent, Figure 4.3. Hence changes in processor frequency affect the number of cycles per slot, Figure 4.4. The main challenge in integrating DVFS with the Pfair algorithm is that a task's parameters must change when we want to scale down the task to reduce energy consumption. When a task is scaled down, its execution time increases according to the new frequency and the Pfair scheduling algorithm needs to be made aware of this increase. In PAPF, we temporarily increase the weight of the task being scaled down, to achieve this awareness, Figure 4.5. Increasing the weight of a task increases its rate of progress which allows the task to remain punctual even though it is running at a lower frequency. PAPF ensures that the increase in weight exactly accounts for the increase in execution time due to DVFS.

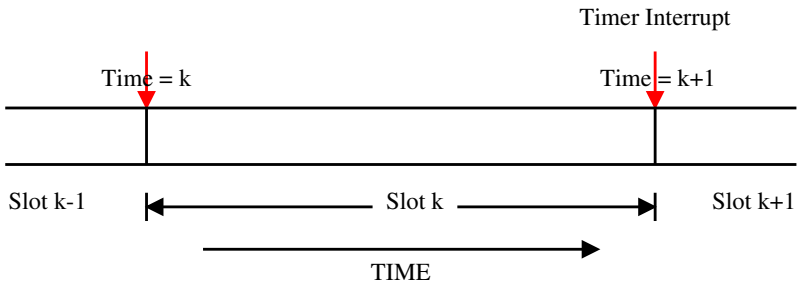


Fig. 4.3. Timer Based Implementation of Slots

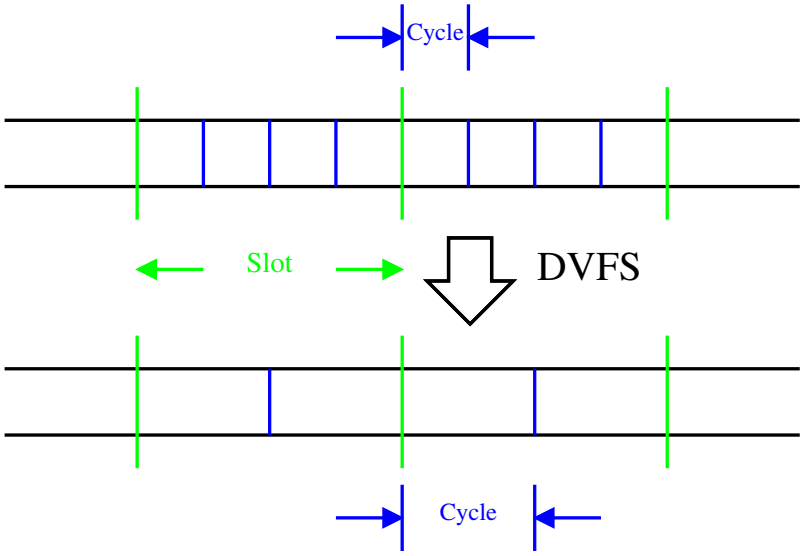


Fig. 4.4. Cycles Per Slot Under DVFS

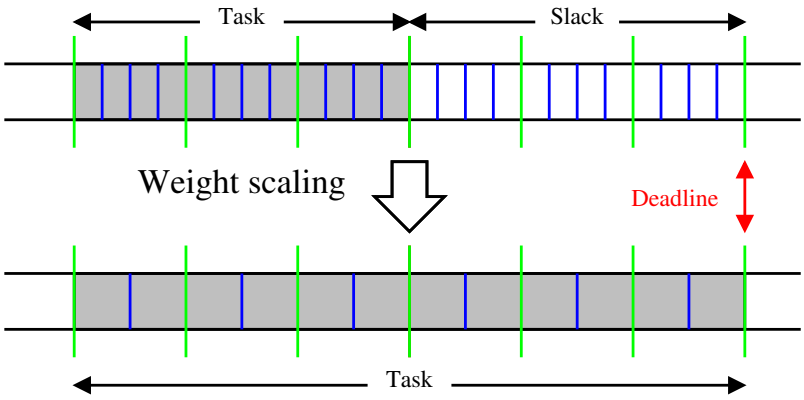


Fig. 4.5. Weight Scaling to Account for DVFS

In the original Pfair algorithm, the weight of a task remains constant over time. However, due to DVFS, the weight of a task changes over time and necessitates changes in the computation of lag and the characteristic string of the task. To maintain correctness of the Pfair scheduling algorithm and to accommodate the notion of variable weights, in PAPF, we calculate the lag and characteristic string

of a task incrementally. Before formally describing the algorithm, we provide the updated definitions of lag and characteristic string.

- The weight of task τ_i in slot t is defined as w_{i_t}.
- The lag of a task at time t is defined incrementally as follows:

$$lag(\tau_i, t) = \begin{cases} lag(\tau_i, t-1) + w_{i_{t-1}} - S(\tau_i, t-1) & \text{if } t > 0 \\ 0 & \text{if } t = 0 \end{cases} \tag{4.1}$$

- The ideal allocation to task τ_i till time t is given by:

$$ideal_{i_t} = \begin{cases} ideal_{i_{t-1}} + w_{i_{t-1}} & \text{if } t > 0 \\ 0 & \text{if } t = 0 \end{cases} \tag{4.2}$$

- $\alpha(\tau_i)$ the characteristic string of task τ_i is a string over $\{-, 0, +\}$ where the t^{th} symbol is given by:

$$\alpha_t(\tau_i) = sign(ideal_{i_{t+1}} - \lfloor ideal_{i_t} \rfloor - 1) \tag{4.3}$$

Slack management is an important part of the PAPF algorithm. Slack elements are generated when tasks finish execution earlier than their worst cast execution time. In our scheme, each slack element l are associated with a weight w_l and deadline d_l. When a slack element is generated, its weight (deadline) is set to the weight (deadline) of the task generating the slack. The set of slack elements is maintained in a priority queue, DS, based on the deadline of the slack elements.

Algorithm 1 illustrates the PAPF algorithm. Here $e_i(f)$ represents the scaled execution time of task τ_i at frequency s. The procedure SCHEDULE is a modified version of PF presented in [11]. Although the complexity of PF has been reduced in algorithms based on PF such as PD, PD^2 [4], we chose PF because of its ease of presentation and implementation. It should be straightforward to extend our scheme

Algorithm 1
PAPF Scheduling

1: **procedure** SCHEDULE(Γ)
2: **for all** $\tau_i \in \Gamma$ **do**
3: $S(\tau_i, t) \leftarrow 0$
4: Calculate $lag(\tau_i, t), \alpha_t(\tau_i)$ and the total order on *cont*
5: **end for**
6: Populate $urgent, cont, tnegru$
7: **for all** $\tau_i \in urgent$ **do**
8: $S(\tau_i, t) \leftarrow 1$
9: **end for**
10: **for all** $\tau_j \in cont$ based on the total order **do**
11: **if** $\sum_{i=1}^{N} S(\tau_i, t) < M$ **then**
12: $S(\tau_j, t) \leftarrow 1$
13: **end if**
14: **end for**
15: **end procedure**
16: **procedure** JOB RELEASE(τ_i)
17: $w_{i_t} \leftarrow e_i/p_i$
18: $slack_{max} \leftarrow \sum_{l \in DS, d_l > d_i} w_l$
19: $w_{max} \leftarrow min(w_{i_t} + slack_{max}, 1)$
20: $s_{min} \leftarrow min\{s \in S \mid e_i(s)/p_i < w_{max}\}$
21: $w_{new} \leftarrow e_i(s_{min})/p_i$
22: Claim total slack $(w_{new} - w_{i_t})$ with $d_l > d_i$
23: $w_{i_t} \leftarrow w_{new}$
24: **end procedure**
25: **procedure** JOB END(τ_i)
26: **if** $t \neq d_i$ **then**
27: Create slack. l, with $w_l = w_{i_t}$ and $d_l = d_i$
28: $DS \leftarrow DS \cup l$
29: **end if**
30: **end procedure**

to the more efficient PD and PD^2 algorithms. The procedure SCHEDULE is called at the beginning of each slot when it updates the task parameters. Then, it selects M out of the N tasks in the system to schedule in the current slot according to the base Pfair scheduling algorithm.

The procedure JOB RELEASE is called every time a new job of a task is released, i.e., at every period boundary. This procedure is responsible for utilizing available slack for scaling down a job to a lower frequency while updating the task's weight to maintain correctness. The maximum usable slack, calculated in line 18, is given by the sum of weights of all slacks with deadline greater than or equal to deadline of the current task. While scaling down a task the procedure ensures that the total utilization of the system remains below M, line 18, and that the task's weight remains below 1, line 19. Then the procedure selects the minimum frequency that the system can accommodate based on the increased weight of the task, line 20. Here $e_i(s)$ represents the execution time of task τ_i when it is runs at speed s. Due to the discreteness of available processor frequencies, the total weight of the slack to be claimed from the system is updated based on the minimum frequency achieved, line 21. In line 22, slack with a total weight of $w_{new} - w_{i_t}$ is claimed from DS. To claim slack, elements are removed from the DS queue in increasing order of deadlines starting from the deadline of the current job till the total weight of removed elements becomes greater than or equal to the weight being claimed.

The procedure JOB FINISH is called whenever a job finishes execution and the lag of the corresponding task is not zero. When a job finishes execution earlier than its deadline, we create a slack element with weight (resp. deadline) equal to the weight (resp. deadline) of the task.

4.4 Correctness of PAPF

In this section we argue that the PAPF algorithm is correct in the sense that no task misses its deadline as a result of task reweighting and DVFS. The original Pfair scheduling algorithm was developed for a fixed set of tasks with static weights throughout their lifetime. Conditions for allowing tasks to join and leave the system and for tasks to be reweighted dynamically were developed in [56]. Task reweighting can be modeled as a combination of a *leave* and a *join* where the task with the

current weight leaves the system and a task with the new weight joins the system. It has been proved in [56] that the following conditions for *join* and *leave* maintain the correctness of the Pfair scheduling algorithm:

1. *join*: Tasks may join the system as long as the total utilization of the system remains below M.

2. *leave*: A task may leave the system when its lag is 0.

In our system, we reweight a task by reclaiming the slack produced by other tasks. The weight increment of the task is equal to the total weight of slack consumed. During this process, we ensure that the total utilization of the system remains below M and also that the task's utilization remains below 1. At the same time, we allow a task to leave the system only when its lag is zero. By following the above mentioned rules, we ensure that we maintain the correctness of the Pfair scheduling algorithm during task reweighting and DVFS.

4.5 Optimizing Power Aware Pfair Scheduling

The PAPF algorithm tries to minimize the energy consumption of the system by scaling down execution frequency of tasks whenever possible. However, PAPF does not determine the assignment of scheduled tasks on processors. Hence the basic PAPF algorithm might end up choosing a task assignment that increases overheads in the system. In this section, we identify the overheads in the PAPF algorithm and present intelligent task assignment strategies to minimize these overheads.

4.5.1 Overheads in PAPF

There are two kinds of overheads present in the PAPF scheduler. a) Task migration and b) Frequency switch. Task migrations are caused by the per slot scheduling decision in Pfair scheduler and the latter is the result of dynamic power management

on top of it. A naive task assignment scheme may result in an unacceptably high level of task migrations and processor frequency switches. Task migrations can adversely affect the cache performance. Although concepts like Megatasking [5] have been developed which can help in mitigating these effects, the PAPF algorithm in itself must be made aware of the task migrations. Similarly, the number of processor frequency switches must be minimized because of the time and energy overhead associated.

4.5.2 Mitigating Overheads in PAPF

The PAPF algorithm is suitably modified with an intelligent task assignment technique which is aware of the above mentioned overheads. In each scheduling slot, the algorithm must assign tasks to processors so that these overheads are minimized.

Each task prefers to be scheduled on a processor based on how often it has been assigned to that processor in the past. More recent assignments are preferable. To minimize the task migration overheads, we define the metric task processor affinity, $TPA(\tau, r, t)$ that denotes the affinity of a task τ to be scheduled on a processor r at time t. The task processor affinity is incrementally defined based on its value in the last slot and is given by:

$$TPA(\tau, r, t) = TPA(\tau, r, t-1) * \alpha + S_r(\tau, t) \tag{4.4}$$

Here $0 \leq \alpha \leq 1$ is a temporal coefficient which decreases the processor affinity of a task for an assignment in the past.

The cost of the frequency switch is proportional to the difference between the two frequencies. To minimize the number of frequency switches, this cost must be considered in the task assignment algorithm. We define another metric, processor frequency affinity, $PFA(r, s, t)$ to represent the affinity of processor r to run at

frequency s in slot t. The processor frequency affinity is defined in terms of the overhead for changing the processor frequency as:

$$PFA(r, s, t) = 1 - \beta O(s, s_r(t-1)) \tag{4.5}$$

where $s_r(t-1)$ is the frequency of processor r in slot $t-1$ and β is a normalization factor such that $0 < \beta O(s, s_r(t-1)) < 1$.

To minimize the mentioned overheads, we formulate the allocation of tasks to processors as a bipartite assignment problem with the edge weights defined as a function of the task processor and processor frequency affinities. The objective is to maximize the total weight of the assignment. Figure 4.6 illustrates this formulation. In Figure 4.6, there is an edge corresponding to each task-processor pair. The edge

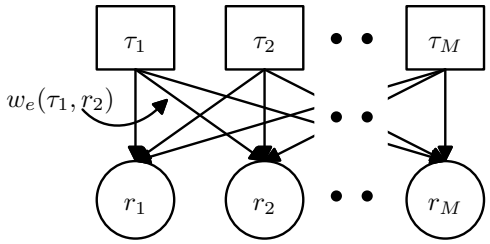


Fig. 4.6. Task Allocation as a Bipartite Assignment Problem

weight for assigning task τ to processor r is defined as:

$$w_e(\tau, r) = w_{TPA} * TPA + w_{PFA} * PFA \tag{4.6}$$

Here $0 \leq w_{TPA}, w_{PFA} \leq 1$, and $w_{TPA} + w_{PFA} = 1$. w_{TPA} and w_{PFA} are weights representing the relative importance of minimizing task migrations and minimizing processor frequency switches respectively. By varying these weights the system designer can trade-off the severity of tasks migrations and frequency switches. The results section presents the effects of varying these weights.

Algorithm 2
Edge-Greedy assignment

1: **procedure** ASSIGN(Γ_t, R, E)
2: $E_s \leftarrow \phi$
3: **for all** $\tau_i \in \Gamma_t$ **do**
4: Add edge $e \in E$ with maximum weight to E_s
5: Remove from E edges incident on vertices of e
6: **end for**
7: **end procedure**

Given this problem definition, we use Kuhn's Hungarian method for the assignment problem [42] as the optimal solution and also propose two greedy heuristics. Both the heuristics are based on multiple passes, each of which assigns a single task to a processor. They take as input the set of tasks to be scheduled in the current slot (Γ_t), the set of processors (R), and the set of edges (E). The output is produced in E_s, the set of selected edges.

Algorithm 3
Vertex-Greedy assignment

1: **procedure** ASSIGN(Γ_t, R, E)
2: $E_s \leftarrow \phi$
3: **for all** $\tau_i \in \Gamma_t$ **do**
4: $e \leftarrow$ Edge with maximum weight incident on τ_i
5: $E_s \leftarrow E_s \cup \{e\}$
6: Remove from E edges incident on vertices of e
7: **end for**
8: **end procedure**

The first heuristic implemented in Algorithm 2, called Edge-Greedy iterates over all the available edges of the graph and selects the edge with the maximum weight (line 4). By selecting an edge the corresponding task is assigned to the corresponding processor, and during the subsequent passes edges incident upon these task and processor nodes are not considered (line 5). Algorithm 3 implements the Vertex-Greedy heuristic which iterates over the set of tasks and chooses the most heavy edge for each task (line 5). Kuhn's Hungarian algorithm takes $O(M^4)$ operations while the Edge-Greedy and Vertex-Greedy algorithms take $O(M^3)$ and $O(M^2)$ operations respectively. In section 4.6 we present results on the relative effectiveness of each approach.

4.6 Results and Discussions

4.6.1 Experimental Setup

Simulator

A Java based simulator was developed to evaluate the effectiveness of PAPF and optimization schemes. The simulator supports simulation of the basic and PAPF Pfair scheduling algorithms with and without optimizations. The simulator incorporates the energy model and the task model described in Section 2. The simulator can be configured to run in five simulation modes (Table 4.1).

Table 4.1
Simulation Modes in PAPF

Name	Description
PF	The basic Pfair scheduling algorithm
PAPF	Power aware Pfair, without optimizations
VG	PAPF with vertex greedy assignment
EG	PAPF with edge greedy assignment
OPT	PAPF with optimal assignment

Benchmarks

We evaluate the proposed approach using both synthetic and real benchmarks. We generated 10 synthetic tasksets with utilizations varying from 0.4 to 4.0. Each taskset consists of 20 tasks whose utilizations are normally distributed with a mean value of $\mu = (task\ set\ utilization)/20$ and a standard deviation of $\sigma = \mu/2$. The periods of tasks are uniformly distributed in the range [100,10000] ms, which is similar to the range used in [26] [20]. The worst case execution time of each task is calculated from its utilization and period. During simulation the actual execution time of a job is calculated as a fraction of the worst case execution time. This fraction is uniformly distributed between [0.5, 1]. We consider a default slot size of 1ms. The DVFS overhead parameters C and K were set to 0.5 and 0.0 by default respectively.

Table 4.2
Multimedia Benchmark for Evaluation of PAPF

Application	Description	p_i	e_i
mpegplay	MPEG video decoder	30	11
madplay	MP3 audio decoder	30	1
tmn	H263 video encoder	400	165
tmndec	H263 video decoder	30	12
toast	GSM speech decoder	25	1
adpcm	ADPCM speech decoder	80	7

For benchmarking with real tasks, we use a multimedia taskset which consists of six applications: **mpegplay**, **madplay**, **tmn**, **tmn**, **dec**, **toast** and **adpcm** as shown in Table 4.2. This taskset has also been used in [61] for benchmarking and has been studied in [64] for their worst case execution times.

For energy calculations, we use the frequency and power values for Intel XScale [63] [21] processor (Table 4.3).

Table 4.3
Intel XScale Frequency and Power Levels

Frequency(MHz)	150	400	600	800	1000
Power(mW)	80	170	400	900	1600

Simulation Parameters

Six sets of experiments were conducted in order to analyze the effectiveness of our scheme. In experiment 1, the benefits of PAPF over basic Pfair algorithm are evaluated by varying the taskset and number of processors. In experiment 2, we vary the slot size and study its effect on the energy consumption. In experiment 3, we vary the DVFS overhead parameters, C and K and study the effect on energy consumption. Experiment 4 evaluates the effectiveness of different optimization techniques compared to basic PAPF. The effect of changing weight of the task migration cost and that of frequency switching cost is studied in experiment 5. Finally, experiment 6 shows the effect of varying system load on the task migration and frequency switch rates. The system load is defined as the ratio of the total utilization of the taskset to the number of processors. All the experiments were run for 1 million slots. Table 4.4 lists the other relevant details of each experiment.

Table 4.4
Simulation Parameters for Evaluation of PAPF

Experiment	Optimization heuristic	weights (w_{TPA}, w_{FPA})	Slot size (ms)	C,K
1	None	N/A	1	1,1
2	None	N/A	varying	1,1
3	None	N/A	1	varying
4	Varying	0.5,0.5	1	1,1
5	EG	Varying	1	1,1
6	VG	0.5,0.5	1	1,1

4.6.2 Results

We evaluate the effectiveness of PAPF in terms of the energy consumption of the PAPF algorithm normalized to that of the basic Pfair algorithm. For optimization heuristics, we compare the rate of occurrence of overhead events. In all the experiments, none of the tasks missed deadlines showing that the PAPF scheduling algorithm can reduce energy consumption without compromising correctness. Results of the mentioned experiments are discussed in the following sections.

Evaluation of PAPF

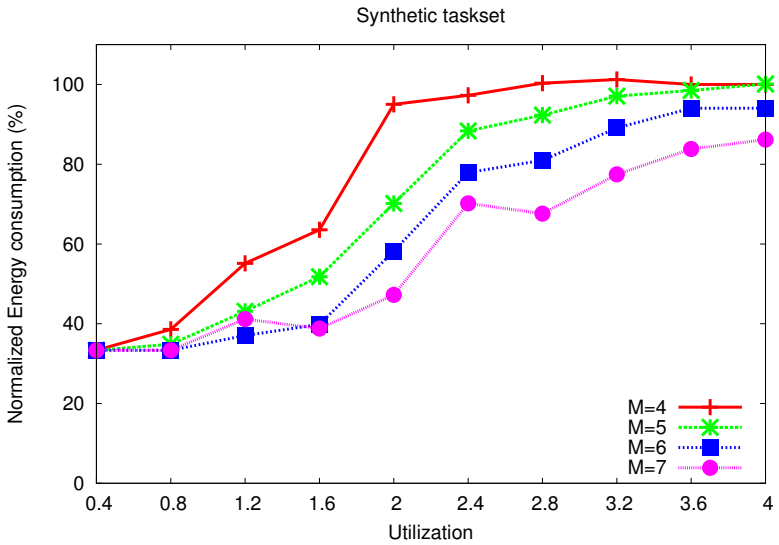


Fig. 4.7. Normalized Energy Consumption of PAPF with Utilization using Synthetic Taskset

Figure 4.7 shows the energy consumption of PAPF normalized to that of the basic Pfair algorithm, with varying taskset utilizations in the synthetic benchmark. The result shows that the energy reduction directly depends on the task set utilization and the number of processors. Lower utilization tasksets create more static slack which PAPF can use to scale down execution frequency of tasks resulting in lower

energy consumption. Increasing the number of processors also reduces the energy consumption. The result also indicates the effect of the DVFS overheads on the energy consumption of the PAPF algorithm. The benefits of PAPF over basic Pfair quickly reach a saturation point after a certain utilization. Beyond this utilization, the amount of available slack is not enough and the overheads of DVFS are greater than the energy savings due to DVFS.

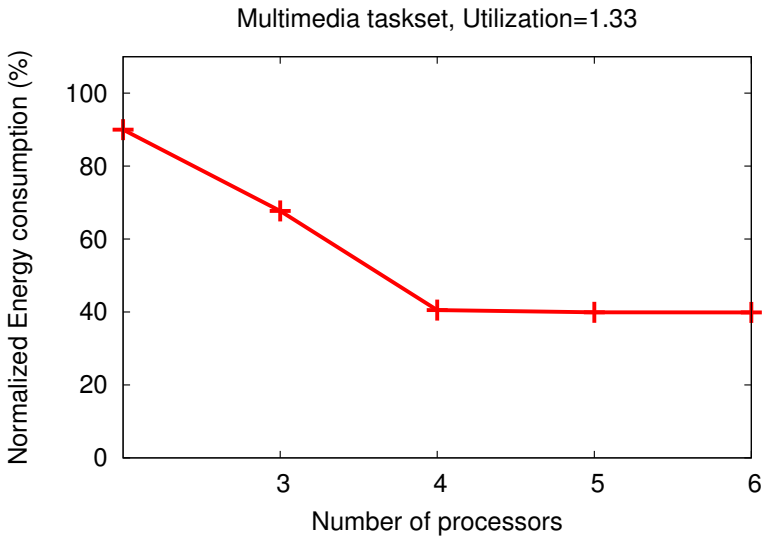


Fig. 4.8. Normalized Energy Consumption of PAPF with Utilization using Multimedia Taskset

Figure 4.8 shows the normalized energy consumption using the multimedia benchmark running on 3 processors. Since the multimedia taskset has a fixed utilization, the energy improvements were studied by only varying M. It can be seen that increasing the number of processors can increase the energy improvements, but there is no improvement beyond 4 processors, because the all tasks are already running at minimum frequency.

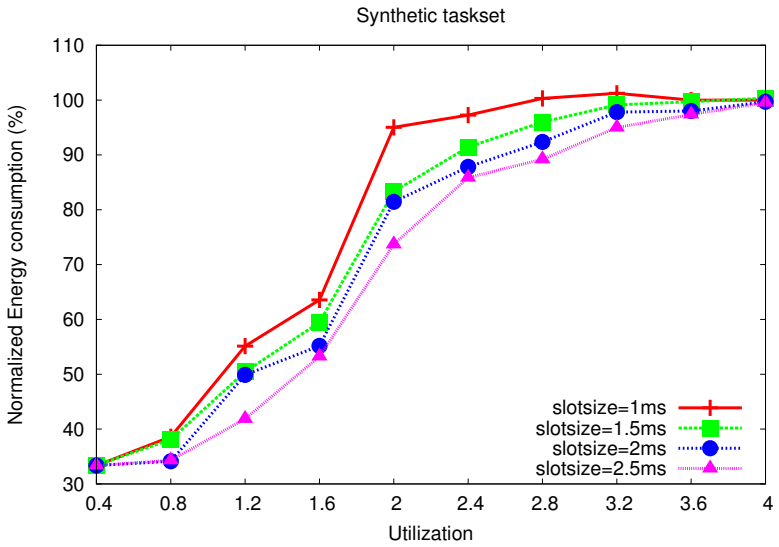


Fig. 4.9. Normalized Energy Consumption of PAPF with Slot Size

Energy Consumption with Slot Size

In the Pfair scheduling algorithm scheduling decisions are taken at the beginning of each slot. Hence the total scheduler overhead and the DVFS overhead depends largely on the slot size used for the Pfair algorithm. To further study the effect of slot size, Figure 4.9 shows the normalized energy consumption of different tasks with varying slot size and $M = 4$. It can be seen that PAPF performs better when larger slot sizes are used. With slot sizes below 1 ms, our algorithm quickly reaches a saturation point. This result shows a limitation of our scheme in that our scheme improves energy consumption only when sufficiently large slot sizes are used.

Energy Consumption with Overhead Parameters

Figure 4.10 shows the effect of varying the DVFS overhead parameters C and K on the normalized energy consumption of PAPF. In this experiment, a taskset with weight 2.4 was used and $M = 4$. This result shows that the overall energy

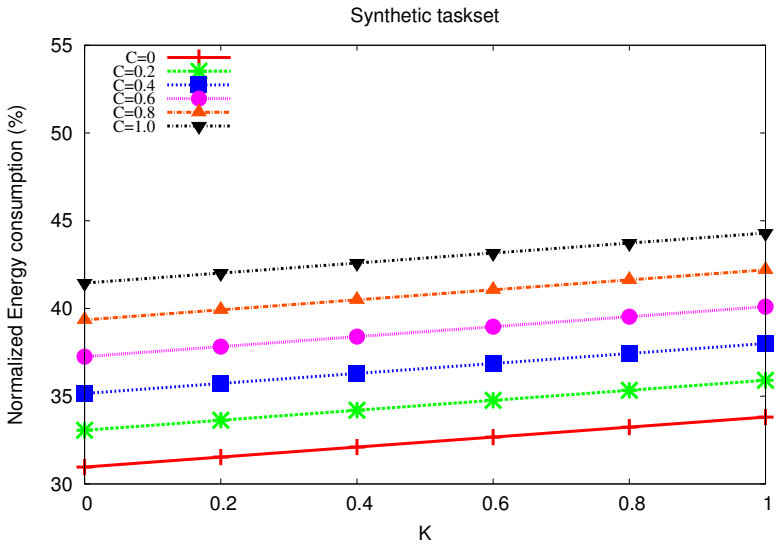


Fig. 4.10. Normalized Energy Consumption of PAPF with Overhead Parameters

consumption of a schedule is linear in both C and K. The result also shows that PAPF is able to reduce energy consumption even in the presence of large DVFS overhead costs. This is due to the fact that, our scheme employs DVFS only when the overall energy benefits of DVFS are positive.

Evaluation of Optimization Schemes

Figure 4.11 shows a comparison of VG and EG heuristics with the basic PAPF and OPT in terms of the task migration and frequency switch rates. Synthetic tasksets with utilizations 0.8, 2.4, 3.2 and 4.0 were used. Both VG and EG are able to achieve significant reductions in overheads. Figures 4.11a,4.11a and 4.11a also show the drawback with the greedy nature of the heuristics. EG results in higher overheads than VG even though edge greedy has higher run time complexity. Although OPT achieves maximum reduction in overheads, its high computational complexity could be prohibitive. The proposed heuristics provide significant improvements (up to 32% reduction in task migrations and up to 16% in frequency switches for utilization 0.8)

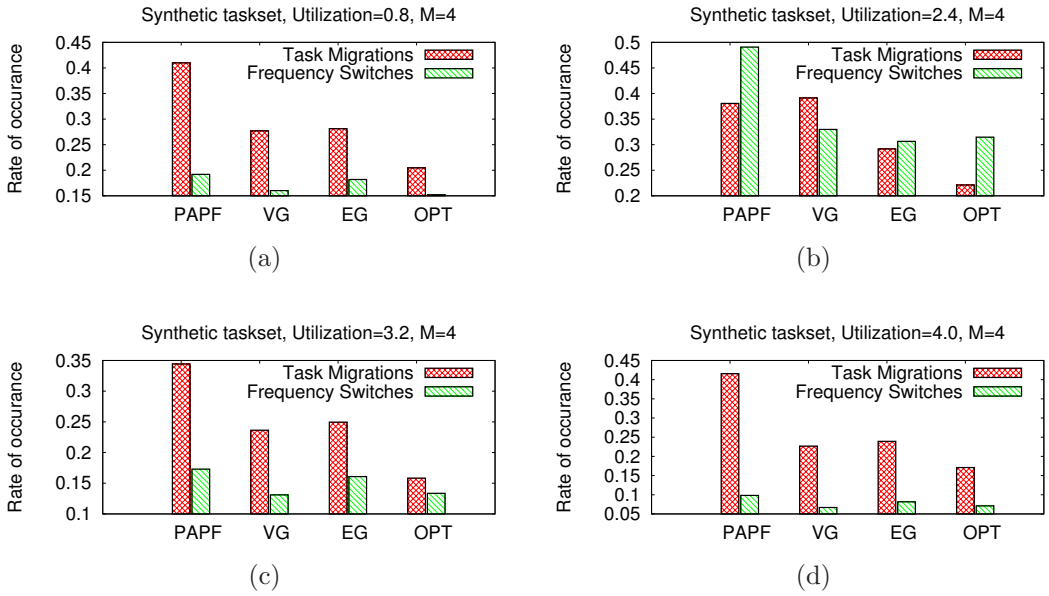


Fig. 4.11. Comparison of Optimization Schemes using Synthetic Tasksets

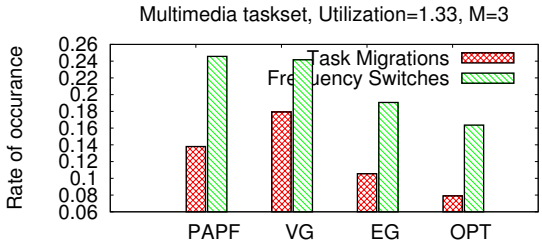


Fig. 4.12. Comparison of Optimization Schemes using the Multimedia Taskset

over PAPF at much lower costs. Figure 4.12 shows the comparison of optimization schemes using the multimedia taskset.

Effect of Varying Affinity Weights

The weights assigned to the processor frequency affinity and the task processor affinity represent the relative importance of minimizing frequency switches versus task migrations. Figures 4.13 and 4.14 show that varying w_{TPA} and w_{PFA} allows

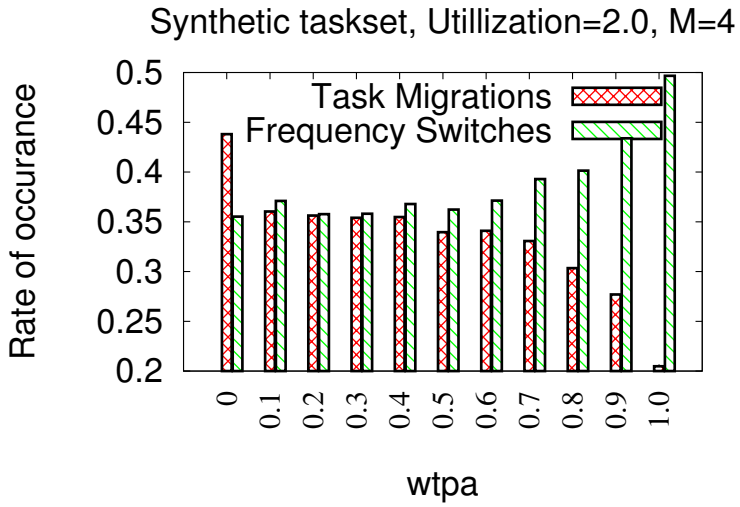


Fig. 4.13. Effect of Varying w_{TPA} in a Synthetic Taskset with Utilization 2.0

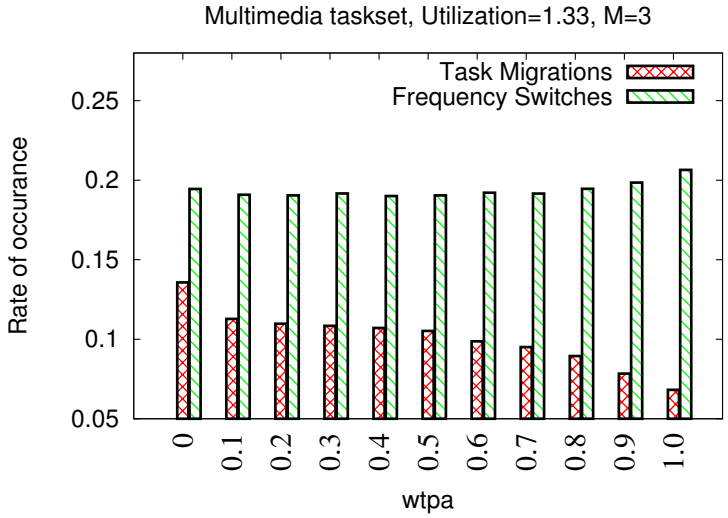


Fig. 4.14. Effect of Varying w_{TPA} in the Multimedia Taskset

fine grained control over the severity of the overheads considered. By varying these weights the system designer can trade off the relative severity of each overhead. For the multimedia taskset the frequency switch overhead does not vary much with w_{PFA}. This is because of the high variance in the task utilizations and the small periods

of tasks in the multimedia taskset. A high variance in task utilizations means that the system is effectively running a high number different frequencies. Also, smaller periods will result in higher number of frequency switches per slot.

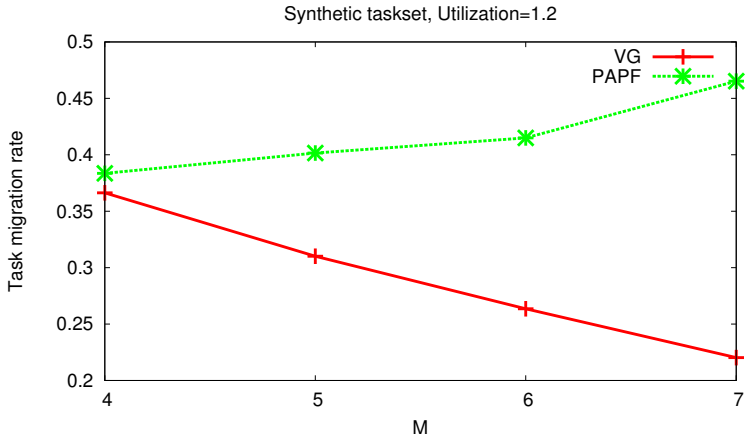


Fig. 4.15. Task Migration Rate with Varying System Load

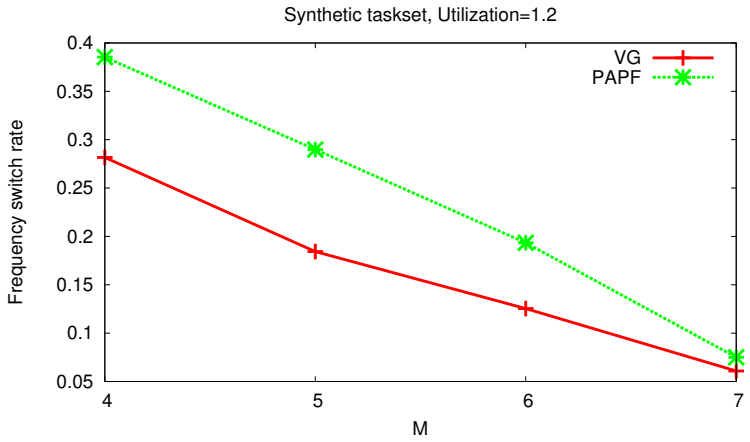


Fig. 4.16. Frequency Switch Rate with Varying System Load

Effect of Varying Number of Processors

In experiment 6, we vary the number of available processors and study the effects on task migration and frequency switch rates for a taskset with utilization 1.2. Figure 4.15 shows that as the system load decreases, the VG heuristic is able to gradually decrease the task migration rate, whereas the basic PAPF algorithm could increase the task migration rate. This is because the VG heuristic can use the extra slack to maintain task-processor mappings, whereas there are is a higher probability that the basic PAPF algorithm could use a task-processor assignment that increases the task migration rate. Figure 4.16 shows that as the system load decreases, PAPF is able to achieve frequency switch rates comparable to that of VG. As the load decreases, PAPF is able to run all tasks at minimum frequency which minimizes the frequency switch rate. It is evident that, for low system loads task migration rate optimization becomes more important compared to frequency switch rate optimization.

4.7 Conclusions and Future Work

Power management in real-time embedded systems is becoming increasingly important. A novel power aware Pfair scheduling algorithm for efficient dynamic power management in multiprocessor real-time system was introduced. The scheduling scheme was able to achieve up to 66% energy savings over a basic Pfair scheduling approach. The algorithm is further optimized using task processor assignment techniques to achieve up to 32% reduction in the overheads associated. As the optimal assignment may be computationally expensive, two heuristics for the assignment problem were proposed and evaluated. Accounting for the energy savings due to reduced overheads requires detailed cache modeling and is considered as a future work.

5. CHOOSING A GOOD SLOT SIZE FOR PFAIR SCHEDULING

In the Pfair scheduling framework, task parameters (period and execution time) are specified in terms of fixed length intervals of time, called slots. The slot length is an implementation dependent parameter that does not effect the algorithmic design, but as we show in this section, can have significant effects on runtime efficiency. For efficient implementation of Pfair based scheduling algorithms in a real multiprocessor system, the slot length must be prudently chosen such that the overall scheduling overhead involved in Pfair scheduling is minimized. Improvements in scheduling overhead are significant because scheduling overheads are continuous and ever lasting.

Optimality guarantees of Pfair scheduling algorithms are derived based on the assumption that overheads due to per-slot scheduling activity are negligible. However, in practical implementations these overheads must be properly accounted. Further, in real tasksets, task execution time and periods cannot be expected to be multiples of slot length, leading to additional overheads when task parameters are adjusted to meet Pfair scheduling requirements. A task's execution time in slots is given by dividing its execution time by the slot duration. Any remainder must be accounted for by using an extra slot. We consider the unused portion of this extra slot as an overhead. Similarly, the period of a task in slots is given by dividing its period by the slot duration, and the remainder must be discarded. We argue that the choice of quantum size must be well balanced according to the taskset being scheduled. Large quantum sizes will lead to increased overhead due to the extra remainder slots. At the same time, small quantum sizes will increase the overhead resulting from per-slot scheduling activity.

The quantum size selection procedure takes as input the current taskset being scheduled by the scheduling algorithm. In dynamic task systems, which model general purpose systems more accurately, the scheduled taskset might change with time making a previously selected quantum size stale and inefficient. Hence we envision

the quantum size selection procedure to be an online and continuous process. For e.g. the quantum size reconfiguration process may be run conservatively at hyper-period boundaries, or aggressively each time a task-set change[1] occurs. The actual choice will depend on the implementation and the runtime of the quantum size selection procedure. As with anything that is runtime and continuous, the quantum size selection procedure must be efficient enough to not introduce additional large overheads into the system. Although it has been previously suggested that quantum sizes are constrained by the resolution of hardware clocks [55], [16], we nevertheless believe that a quantum size selection technique is needed to choose from within the set of available quantum sizes and to provide the motivation for removing or finding solutions to the hardware limitations.

The primary technical contributions of this section are as follows:

1. We present a model to measure the overall overhead of Pfair scheduling as a function of quantum size.
2. Based on the overhead model, we present an efficient quantum size selection heuristic, *quotient search* (QS) to minimize the overall overhead of Pfair scheduling without introducing much additional overhead into the system.
3. Through simulation based results, we show that QS performs considerably better than other quantum size selection strategies.

The rest of the section is organized as follows: Section 5.1 discusses related work. Augmentations to the task model presented in Section 2 are presented in section 5.2. Modeling of overheads involved in Pfair scheduling is presents in section 5.3. Our quantum size selection scheme, QS is presented in Section 5.4. Comparisons with other quantum size selection schemes, results and discussions are presented in Section 5.5. Section 5.6 concludes the section.

[1]In Pfair scheduled dynamic task systems, taskset changes only occur at task period boundaries.

5.1 Related Work

There has been considerable related work on implementation of Pfair scheduling in real systems. Most noteworthy is the work related to the LITMUS^{RT} project [17] which is an extension to the Linux kernel introducing support for real-time workloads on multiprocessor platforms, based on the recent advances in algorithmic research on multiprocessor real-time scheduling. There have been a number of studies on realizing and analyzing Pfair scheduling in Linux, based on the LITMUS^{RT} platform [35] [15] [14] [36] [27].

In [55], Srinivasan et al. compare the PD^2 scheduling algorithm against a first-fit partitioned Earliest Deadline First approach. They model the schedulability loss due to the per-slot scheduling nature of Pfair algorithms. Our work builds up on this work and extends the overhead model for Pfair scheduling, by considering quantization overhead besides the per-slot overhead.

Gupta et al. considered the problem of finding an optimal quantum size for Round Robin scheduling that minimizes the average response time [33]. Their solution is analytical in nature which they later verify through numerical analysis. In contrast, we concentrate on minimizing the overall overhead of Pfair scheduling. Since overhead minimization for Pfair scheduling is a non-smooth optimization problem [22], we concentrate on simple optimization heuristics instead of an analytical solution.

Recently, Funk et al. [30] presented a unifying theory, DP-FAIR for deadline partitioning and presented the DP-WRAP algorithm that relaxes the over-strict nature of per-slot scheduling and lag constraints in Pfair scheduling. Their scheme is motivated towards reducing context-switch and task migration overheads by making scheduling decisions only when necessary instead of every slot, thereby enabling efficient implementation of optimal multiprocessor scheduling algorithms in real systems. In contrast, we try to reduce the overheads in Pfair scheduling by choosing a suitable quantum size.

5.2 Task Model Augmentations

In this work we consider the absolute parameters of a task expressed in terms of absolute units of time. $\hat{e}_i$ and $\hat{p}_i$ are the absolute execution time and period of the task respectively. The tuple, (e_i, p_i), where $e_i, p_i \in \mathbb{N}$ denote the execution time and period of the task respectively, in terms of slots. The weight of a task is defined as $w_i = e_i/p_i$ and $0 < w_i < 1, \forall i$.

5.3 Overheads in Pfair Scheduling

As mentioned earlier, there are two categories of overheads that come into play while executing real tasksets with Pfair scheduling algorithms:

1. Overhead due to **per-slot scheduling activity**, and
2. Quantization overhead due to **remainder slots**.

Figure 5.1 shows the slot-overhead and quantization overheads graphically.

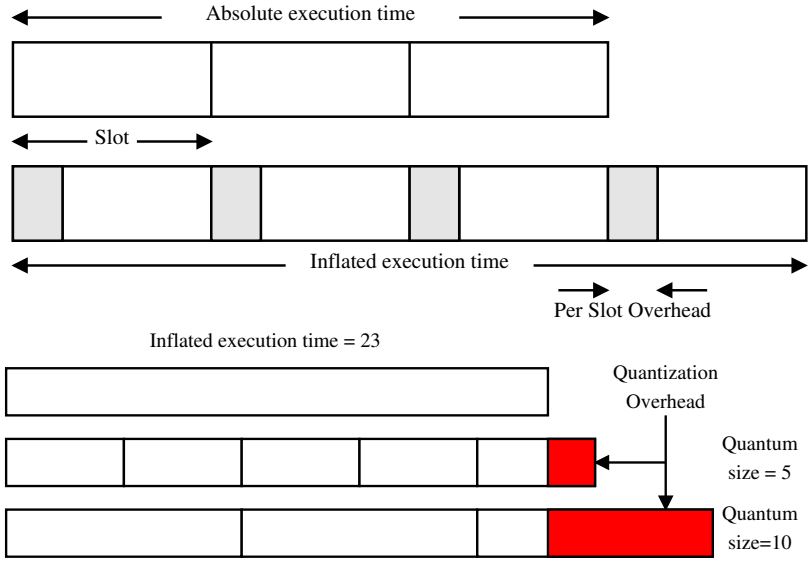


Fig. 5.1. Per Slot and Quantization Overheads

As in Figure 5.2, the per-slot scheduling overhead consists of :

1. **Context switching overhead:** The time taken to save the context of a preempted task and then load the context of a new task.

2. **Cache-related delays:** Depending on whether a task resumes executing on the same processor and/or which tasks execute on the processor in the mean time, a task may suffer cache misses leading to increase of execution time.

3. **Run time of scheduling algorithm:** The time spent by the scheduler in determining which task to run next is itself an overhead. We use the runtime of the PD^2 algorithm, the fastest Pfair algorithm known, to account for scheduling algorithm runtime overhead.

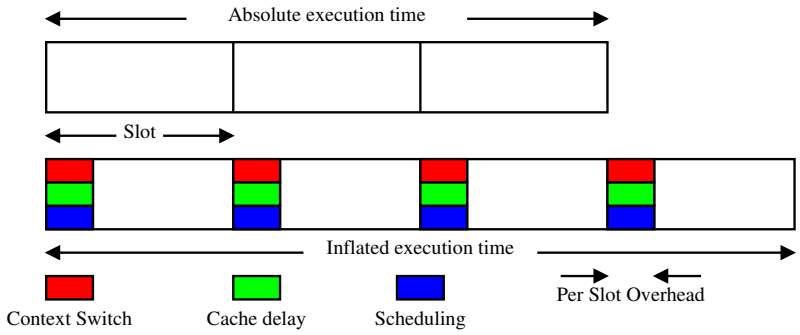


Fig. 5.2. Per Slot Overhead Components

Since the schedulability guarantees of Pfair scheduling algorithms are derived under the assumption that the costs of these overheads are zero, the execution time of tasks must be inflated before using the schedulability tests. We partly adopt the overhead model by Srinivasan et al. in [55] to account for the overhead resulting from per-slot scheduling activity. Let S_{PD^2}, C, D denote the runtime of PD^2 algorithm,

context switch overhead and average cache related preemption delay respectively. Then the inflated execution time e'_i, given a quantum size Q is given by:

$$e'_i = \hat{e}_i + \left\lceil \frac{e'_i}{Q} \right\rceil \times S_{PD^2} + C + n \times (C + D) \tag{5.1}$$

$$n = \mathbf{min}\left(\frac{e'_i}{Q} - 1, \frac{P}{Q} - \frac{e'_i}{Q}\right) \tag{5.2}$$

As in [55], we see that the term e'_i appears on both sides of equation 5.1. We arrive at a value for e'_i by initially setting $e'_i = \hat{e}_i$ and then repeatedly applying the formula until its value converges. The task execution time inflation overhead for Pfair scheduling is given by:

$$I(L) = \sum_{i=1}^{N} (e'_i - \hat{e}_i) \times L/\hat{p}_i \tag{5.3}$$

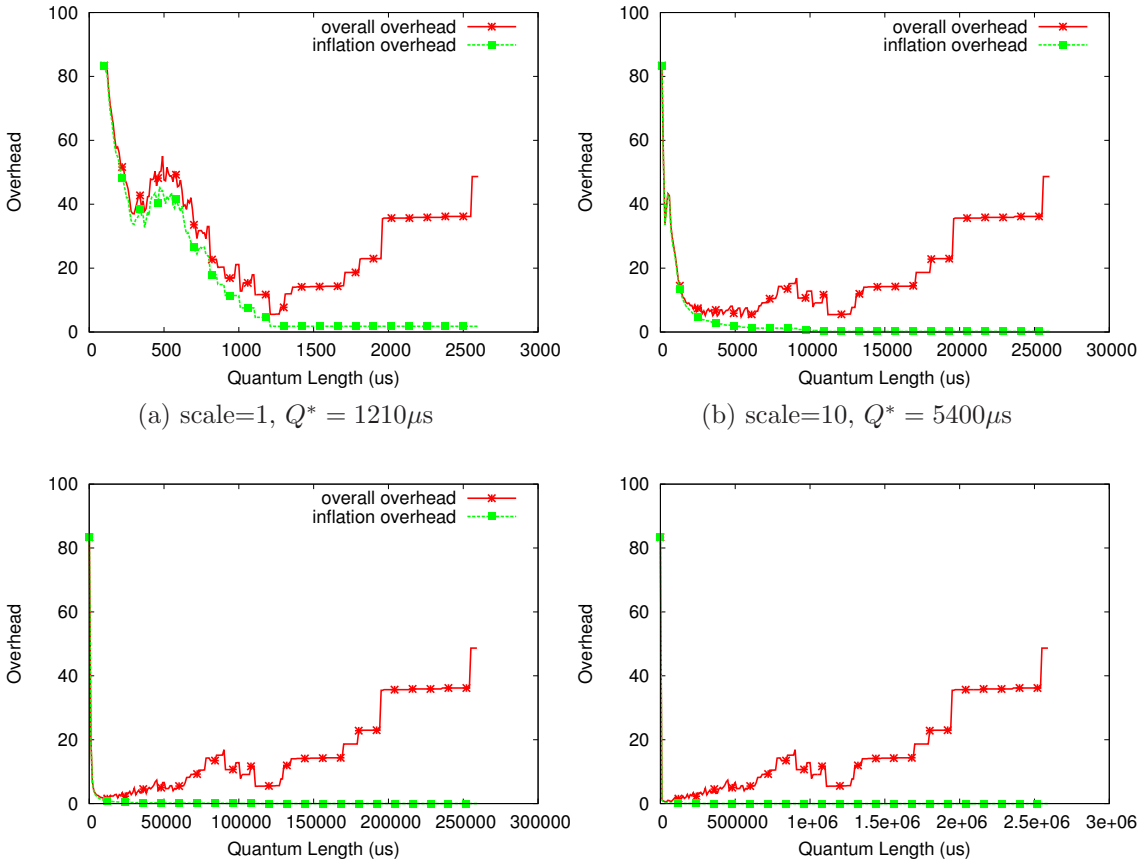
where L is a given, large interval of time, suitable for studying and comparing overheads.

Given a quantum size Q, the inflated execution time and period of a task, e'_i, $\hat{p}_i$ are transformed into their Pfair counterparts, $e_i = \lceil e'_i/Q \rceil$ and $p_i = \lfloor \hat{p}_i/Q \rfloor$, suitable for use in the Pfair scheduling algorithm. As Figure 5.1 shows, the per-slot overheads can become significantly large when large quantum sizes are used.

The overall overhead $H(L)$ of a Pfair scheduling algorithm is now given by:

$$H(L) = \sum_{i=1}^{N} \left(\frac{e_i}{p_i} - \frac{\hat{e}_i}{\hat{p}_i}\right) \times L \tag{5.4}$$

$$= \sum_{i=1}^{N} \left(\frac{\lceil e'_i/Q \rceil}{\lfloor \hat{p}_i/Q \rfloor} - \frac{\hat{e}_i}{\hat{p}_i}\right) \times L \tag{5.5}$$



(a) scale=1, $Q^* = 1210\mu$s

(b) scale=10, $Q^* = 5400\mu$s

(c) scale=100, $Q^* = 12100\mu$s

(d) scale=1000, $Q^* = 12100\mu$s

Fig. 5.3. Variation of Overheads with Quantum Size

5.4 Choosing a Good Quantum Size for Pfair Scheduling

In this section, we analyze the effects of quantum size on scheduling overhead and present the QS scheme to choose a good quantum size. The choice of quantum size in a system must be well balanced according to taskset being executed. Large quantum sizes will lead to wastage through the remainder slots in task execution times. At the same time small quantum sizes will lead to a high degree of scheduling activity and will hence increase scheduling overhead.

As a motivational example, Figure 5.3 shows the variation of overall and inflation overheads with quantum size for four tasksets with four tasks each. Each taskset is

obtained by multiplying the base task parameters (period and execution time) by a scaling factor (i.e. scale=1 represents the base taskset). The base taskset is given by : {(1100, 5400); (900, 3900); (1000, 5100); (800, 23600)} ($(\hat{e}_i, \hat{p}_i)$ in μs.) The quantum sizes considered for each taskset are varied from 100μs to 2600 * scale μs. We set $L = 1e^6$ in this experiment. The overheads were calculated by using the formulae in section 5.3 and then normalized by dividing with L and the number of tasks (4 in this case). Hence the Y-axis represents the per-task per-unit time overhead for a given taskset.

Figure 5.3 shows that up to a certain quantum size, the overall overhead is roughly equal to the inflation overhead. Beyond that quantum size, the remainder slot overhead becomes significant and the overall overhead increases. In each of the three plots, there exists an optimal quantum size, Q^*, which minimizes the overall overhead. Using quantum sizes smaller or larger than this value leads to increase in the overall overhead. In the above example, the optimal quantum sizes are 1210μs, 5400μs, 12100μs and 40100μs for scale 1, 10, 100 and 1000 respectively. Figure 5.3 also shows that it is difficult to arrive at an analytical formula for the optimal quantum size that can minimize the overall overheads of Pfair scheduling algorithms. Further, it is much more important to choose a quantum size carefully for tasksets with small tasks (scale=1) than for tasksets with large tasks(scale = 100). In the case of large tasks (scale = 10, 100 and 1000) as compared to small tasks (scale=1), there are many more quantum sizes that lead to roughly the same overhead as the optimal quantum size.

Finding a good quantum size for Pfair scheduling is challenging because of the discontinuity in the overhead trends. The problem of finding the best quantum size that will lead to minimal overheads falls under the domain of non-smooth optimization. We view the quantum size optimization procedure as a run-time process which might run every time the taskset changes due to task entry and exit. Hence instead

of running computationally intensive optimization algorithms, we choose to use and compare simple heuristics which might be better suited for runtime usage.

For a given task τ_i, we define its base quantum size e_i^* as the inflated execution time obtained by choosing its execution time as the quantum size. The base quantum size is given by:

$$e_i^* = \hat{e}_i + \left\lceil \frac{e_i^*}{\hat{e}_i} \right\rceil \times S_{PD^2} + C + n^* \times (C + D) \tag{5.6}$$

$$n^* = \mathbf{min} \left(\frac{e_i^*}{\hat{e}_i} - 1, \frac{P}{\hat{e}_i} - \frac{e_i^*}{\hat{e}_i} \right) \tag{5.7}$$

Algorithm 4
Quotient Search

```
1: procedure QS(Γ)                          ▷ Find a quantum size for taskset Γ
2:     for all τ_i ∈ Γ do
3:         for div ← div_low, div_high do
4:             Q ← e_i^*/div
5:             new ← evaluateQuantum(Q, Γ)
6:             if new ≤ current then
7:                 current ← new
8:                 Q^* ← Q
9:             end if
10:        end for
11:    end for
12:    return Q^*
13: end procedure
```

A naive approach to finding a suitable quantum size would be to scan through the range of possible quantum sizes with small increments and choose the quantum size that results in minimal overhead. In Figure 5.3, it can be seen that the overhead trend shows a number of local minima. This happens due to the *floor* and *ceiling* functions used in the overhead function $H(L)$. Through extensive experiments, we have observed that these local minima occur whenever the quantum size is a factor of the base quantum size for one or more tasks in the taskset. Based on our observation,

we present a simple heuristic, *quotient search (QS)*, to find a good quantum size for Pfair scheduling which compares the overheads only at factors of task base quantum sizes rather than searching throughout the possible quantum size range. Obviously this observation drastically reduces the computation required to find a good quantum size.

The operation of algorithm 4 is simple. The algorithm iterates over each of the tasks in the taskset (line 2) and for each task computes the overhead for each quotient of the base quantum size (lines 4,5). The algorithm uses two additional parameters, div_{low} and div_{high} to limit the search space of the QS (line 3). In our experiments, we set the values of div_{low} and div_{high} as in equations 5.8 and 5.9.

$$div_{low} = \left\lceil \frac{e_i^*}{\mathbf{min}_{\tau_i \in \Gamma}(\hat{p}_i)} \right\rceil \tag{5.8}$$

$$div_{high} = \left\lfloor \frac{e_i^*}{(S_{PD^2} + C + D)} \right\rfloor \tag{5.9}$$

These values of div_{low} and div_{high} are based on the following rationale: (1) It is impossible to use a quantum size smaller than the per-slot overhead and, (2) Using a using a quantum size greater than the minimum of the periods will mean that $\left\lfloor \frac{\hat{p}_i}{Q} \right\rfloor = 0$ for at least one task, implying $H(L) = \infty$. We believe that there are other possible values for div_{low} and div_{high} too, which would affect the runtime of QS but not the output of the algorithm. The algorithm selects the quantum size corresponding to the minimum of the evaluated overheads (lines 8,12).

5.5 Results and Discussions

5.5.1 Experimental Setup

We conducted a series of simulation experiments to evaluate QS. We also implemented four simple and intuitive quantum selection heuristics to compare the performance of QS:

1. Random Execution Time (**RET**): Select the base quantum size for a randomly chosen task from the taskset.

2. Median Execution Time (**MET**): Select the median of the base quantum sizes of the tasks in the taskset.

3. Average Execution Time (**AET**): Select the average of the base quantum sizes for all tasks in the taskset.

4. Exhaustive Search (**ES**): Search the possible range of quantum sizes exhaustively with small increments of 1μs.

Besides these heuristics, we also evaluate QS against the default fixed quantum size of 1000μs which is currently being used in LITMUS^{RT} [17].

We have observed that the overall overhead, $H(L)$, in Pfair scheduling is highly dependent on the execution time of tasks. Hence, we experiment with tasksets having task average execution time in the range of [1000, 30000]μs with increments of 1000μs. The number of tasks in the taskset was varied in the set {10, 20, 50, 100, 250}. For each combination of average execution time and taskset task count, we generate 50 random tasksets where the execution times of the tasks are distributed normally and the task utilizations are varied uniformly between [1/30, 1/3]. Task periods were calculated from their execution times and utilizations. For each generated taskset we compute and record the runtime of the PD^2 algorithm, S_{PD^2} by running a binary heap based **C** implementation of the PD^2 algorithm. We assume that $C = 5\mu$s and $D = 500\mu$s based on the results in [55], [2].

5.5.2 Results

In the first experiment, we study the variation of overhead with average execution time and number of tasks. Overheads are calculated according to equation 5.4 and then normalized by dividing by the length of the comparison period, L, and by

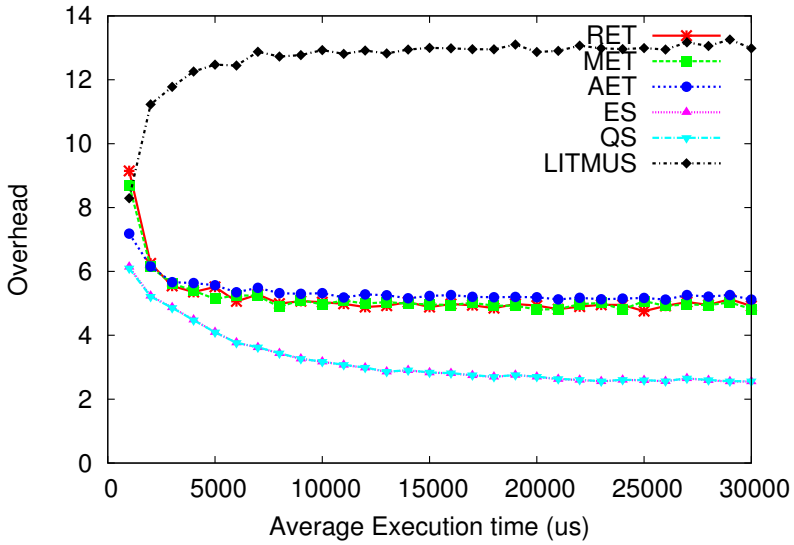


Fig. 5.4. Variation of Per Task Overhead Percentage with Average Execution Time, 250 Tasks

the number of tasks in the taskset. Hence the overhead numbers that we report are per task per unit time, percentage measures. Figure 5.4 shows the variation of overhead with average execution time. As expected, the ES heuristic performs the best resulting in least overheads. At the same time, the proposed QS heuristic performs equally well. The RET, MET and AET strategies perform roughly the same as each other. They result in about 2-3% more overhead than the ES and QS schemes. Figure 5.4 also shows that the fixed quantum size of 1000μs leads to considerable higher overheads (upto 10%) than QS and ES. These differences are significant because the overhead measure is per task per unit time. For higher execution times, the ES and QS schemes are able to find and use higher quantum sizes resulting in even lower overheads. On the other hand, in the LITMUS^{RT} approach the overheads increase with higher execution times due to the increase in per slot overhead.

Figure 5.5 shows the variation of overheads with number of tasks. Similar to Figure 5.4, the ES and QS schemes lead to about 2-3% lower overheads than the

RET, MET and AET schemes and up to 10% lower overheads than the LITMUS^{RT}. As the number of tasks increases, the overheads increase because of the increase in the values of S_{PD^2}. In summary, a taskset agnostic quantum size may lead to significantly high overheads than a taskset aware quantum size selection strategy.

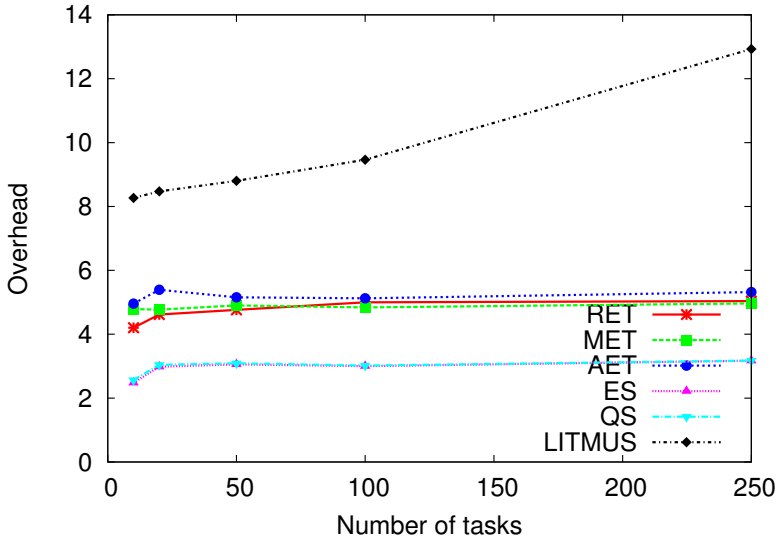


Fig. 5.5. Variation of Per Task Overhead Percentage with Number of Tasks, Average Execution Time 10000μs

In the next experiment, we observe the normalized runtime of the quantum size selection procedure against average execution time. As mentioned in section 5, we envision that in dynamic task systems, a quantum size selection algorithm will be used at runtime to optimize the system operation based on the current set of tasks running in the system. Although the quantum size selection process might seem to be a system initialization procedure for static task systems, it should be a continuous process for dynamic task systems and hence the runtime of the heuristic is an important selection criterion. In this experiment we compare the per task, per unit time, percentage measure of algorithm runtime. Figure 5.6 shows that ES is an expensive scheme for choosing a quantum size and is hence unsuitable for dynamic task systems. The RET, MET and AET strategies, being the simplest, expectedly lead

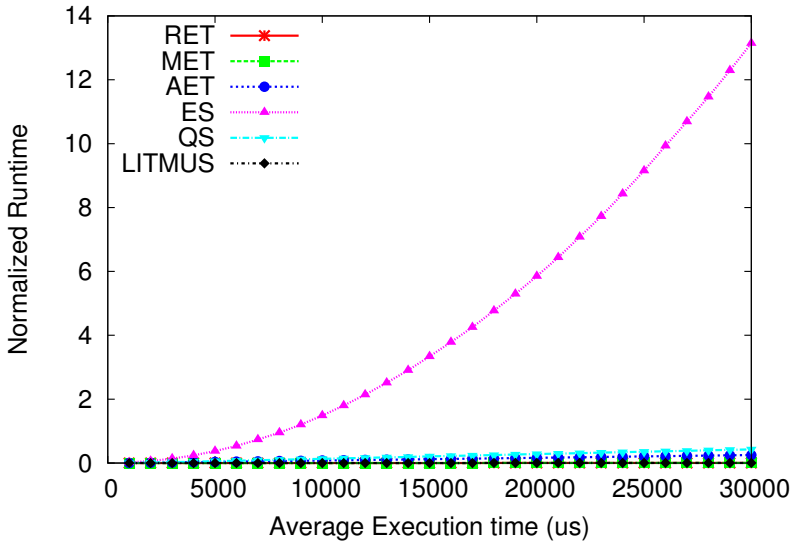


Fig. 5.6. Variation of Runtime with Average Execution Time, 250 Tasks

to lowest runtimes. The QS technique leads to runtimes only slightly higher than the execution time based strategies making it an suitable choice for taskset aware, run-time quantum size selection.

In the last experiment, we study the percentage of tasksets for which the compared selection scheme leads to a correct quantum size. For deciding correctness, we assume that if the resultant overhead measure of a scheme is within 95% of the overhead in ES, then the selection process is correct. Figure 5.7 shows the variation of percentage correctness with average execution time. We see that the RET, MET and AET schemes are correct, only up to a maximum of 37% of the time whereas the QS scheme is correct almost all of the time. The fixed quantum size in LITMUS^{RT} leads to worse results compared to any of the taskset aware schemes. Figure 5.8 shows the variation of percentage correctness with number of tasks. Again, the QS scheme leads to very good performance while the RET, MET and AET schemes remain below 20%. At the same time, as the number of tasks increases, the execution time based schemes lead to worse performances while no such effect is observed in

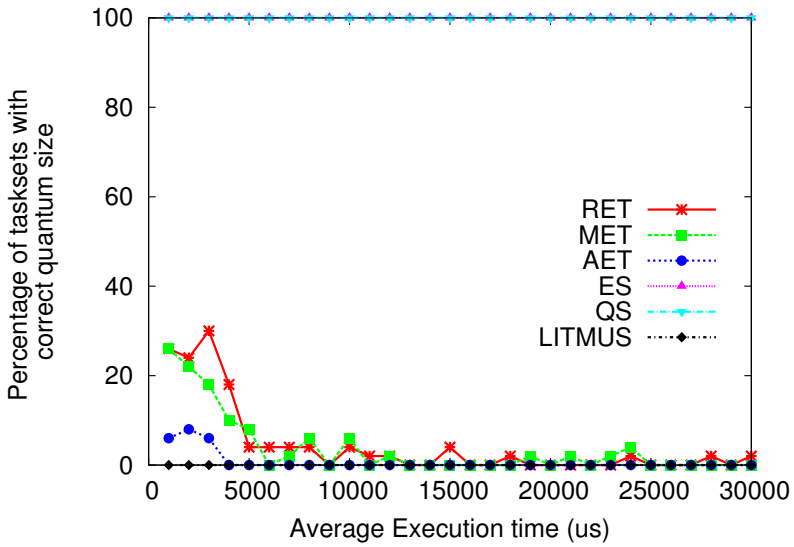


Fig. 5.7. Variation of Percentage Correct Quantum Size with Average Execution Time, 50 Tasks

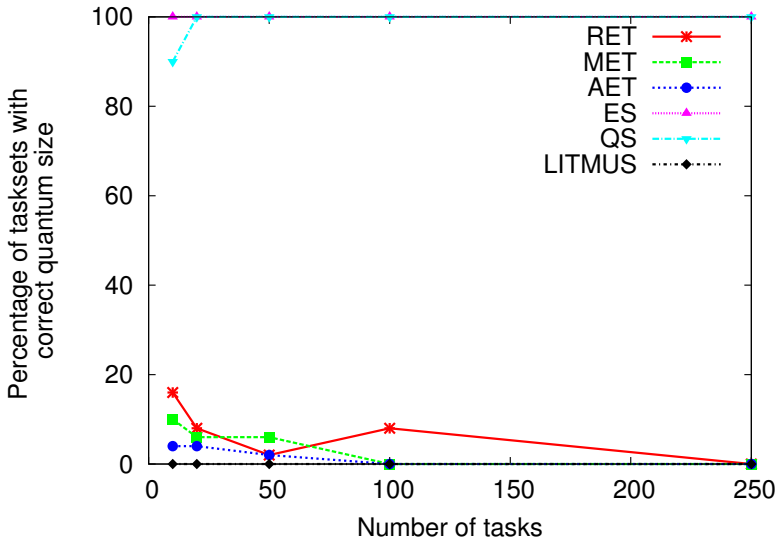


Fig. 5.8. Variation of Percentage Correct Quantum Size with Number of Tasks, Average Execution Time 20000μs

the case of QS. The decrease in percentages is because the execution time based schemes choose a quantum size based on the base quantum size for a single task in

the taskset. Hence when the number of tasks increases, the number of tasks whose base quantum size is not a multiple of the quantum size, increases.

Based on the above results we argue that *Quotient* search is a useful technique to solve the non-smooth optimization problem of finding the best quantum size. The scheme selects a good quantum size without highly impacting system load making it a suitable choice for quantum size selection, especially in dynamic task systems.

5.6 Conclusions and Future Work

Pfair scheduling is one of the few optimal multiprocessor scheduling algorithms. However, due to inherent slot based scheduling, Pfair algorithms are prone to experiencing considerable overheads when implemented in real systems. In this section, the system overhead was analyzed as a function of quantum size. It was shown that prudent quantum size selection is important to minimize system overheads. Based on the analysis of system overhead, a simple quantum size selection heuristic *Quotient Search* (QS) was proposed which was shown to reduce system overhead considerably compared to other quantum size selection heuristics. As future extensions of this work, we will integrate QS into LITMUS^{RT} and measure the benefits gained in real systems. We will also evaluate the results of online quantum size reconfiguration in dynamic task systems.

6. HARDWARE IMPLEMENTATION OF PFAIR SCHEDULER[1]

Today, most desktops run on multiprocessor systems. The popularity of multiprocessor systems is expected to continue, and in the future, most mobile, even embedded systems will feature multiprocessors. Increasing the number of processors on a chip is the most viable method of increasing processor performance. But, along with the performance gains, multiprocessor systems also present new challenges for system designers. One of these challenges is the efficient scheduling of real-time tasks upon multiprocessor systems.

Traditionally, scheduling in multiprocessor system has been implemented in an inefficient fashion. Each processor in the system runs a copy of the scheduling algorithm in software to decide on the next task to run. Running multiple scheduler copies is inefficient use of resources in multiprocessor systems like MPSoC. The inefficiency is further aggravated when either the number of tasks or the number of processors in the system is high. This results in increased overhead in terms of scheduling time and context switching which in turn, translates to higher energy consumption. It is possible to reduce these overheads by replacing the replicated scheduling operations by a central scheduler unit. The central scheduler can communicate the scheduling decision to the processors upon completion of schedule calculation. Although the centralized approach is more efficient than the replicated approach, such a scheduler should be fast enough to support a large number of tasks for multiple processors without suffering from unpredictability of scheduling delays. This motivates the use of a hardware scheduler that will meet the above goals.

Despite the optimal nature of Pfair algorithm, it can be inefficient and computationally expensive when implemented in serialized software. Pfair algorithm involves computation that grows linearly with the number of tasks to schedule. This adds to uncertainty in the effective utilization of the system if the scheduler and the tasks

[1]The work in this section has been derived from the paper [32]

share resources. A parallel implementation can get rid of the unpredictability in scheduling delay and can increase effective utilization. Since scheduling is performed in a dedicated hardware module, time overhead is minimized. The parallelizable nature of Pfair can lead to a significant speed up of the scheduling process when implemented in hardware.

In this section, we propose a low power hardware Pfair scheduler for MPSoC. The speed-area-energy trade-offs involved in the design of a hardware Pfair scheduler were analyzed. We compare its performance in terms of scheduling delay and energy consumption with two other implementation schemes:

1. The replicated Pfair scheduling algorithm running in software on all the processors in the multiprocessor system; and

2. The Pfair scheduling algorithm implemented in software on a dedicated processor.

We also report the area and energy consumption of hardware scheduler through suitable synthesis work.

The main technical contributions of this section are as follows:

1. Introduced the use of a hardware Pfair scheduler in MPSoC to reduce energy consumption.

2. Designed, implemented and evaluated the Low-power Hardware Pfair scheduling scheme suitable for multiprocessor environment.

3. Evaluated the performance of the Low-power Hardware Pfair scheduler using real-time benchmarks in terms of scheduling delay and energy consumption.

The rest of this section is organized as follows: Section 6.1 discusses related work. In section 6.2 we augment our energy model presented in Section 2 suitably for this work. Section 6.4 describes our hardware implementation of the Pfair scheduling

algorithm. Results and discussions are presented in section 6.5. Finally, section 6.6 concludes the section.

6.1 Related Work

While the concept of implementing run time schedulers in software is not new, to the best of our knowledge, this is the first implementation of a hardware scheduler for the Pfair scheduling algorithm.

There have been similar works in the literature which implement a part or whole of the runtime scheduler in hardware to improve predictability and the ability to meet real-time constraints [49] [48] [43] [47]. Mooney et al. developed a tool for run time scheduler synthesis from a system specification [47]. Recently, Kumar et al. proposed and approach to accelerate dynamic task scheduling on multiprocessor systems [43]. Their design accelerates task queues in hardware to overcome the deficiencies of software queues and achieve better load balancing.

Hildebrandt and Timmermann developed a scheduling co-processor for uniprocessor real-time systems [34]. The coprocessor was aimed at speeding up the scheduling task of a RTOS by parallelizing the task prioritization in hardware. They present results on the synthesis of the co-processor module and did not consider benchmarks for evaluation of scheduling performance. Our design achieves similar synthesis results but in a multiprocessor environment.

Danne et al. proposed a hardware scheduler design for programmable devices [25]. They implemented a scheduler that performs the MSDL scheduling for real-time tasks. They reported a linear scheduling time with the increased number of tasks and processors. The motivation was similar to our design; i.e. reducing overall scheduling overhead. Since the scheduling algorithm was different and the hardware design varies significantly from ours, we do not compare our results with theirs.

Anderson et al. discussed the implementation of Pfair scheduling in hardware in the context of network processor design [54]. However, they did not describe a

detail implementation due to several applicability issues of Pfair in network processor design context.

6.2 Energy Model Augmentations

For studying this problem we augment our energy model by assuming that the MPSoC consists of StrongARM processors (SA1100) and other hardware IP blocks. The scheduler runs on SA1100 processors when it runs in software. Thus we use the Intel SA1100 processor energy model to compute the energy consumption of the scheduling algorithm for when it runs in software.

6.3 Scheduler Implementation Schemes

6.3.1 Replicated Software Scheduler

This is the most commonly used multiprocessor scheduling scheme. In this technique, the scheduling algorithm runs on every processor at the end of an execution slot and selects the corresponding tasks for execution in the next slot. The process involves a compulsory context switch and the runtime of the scheduling algorithm and possibly a task migration when the running task resumes on a different processor. The scenario, illustrated in Figure 6.1(C), leads to a high degree of scheduling overhead.

6.3.2 Software Scheduler on a Dedicated Processor

Another method of implementing a multiprocessor scheduling algorithm is to run the scheduler on an independent on-chip processor which communicates with the other processors and notifies them with the scheduling details prior to the next execution slot, illustrated in Figure 6.1(B). In this case, the scheduling time is not spent in the processors themselves and hence the scheduling overhead is reduced.

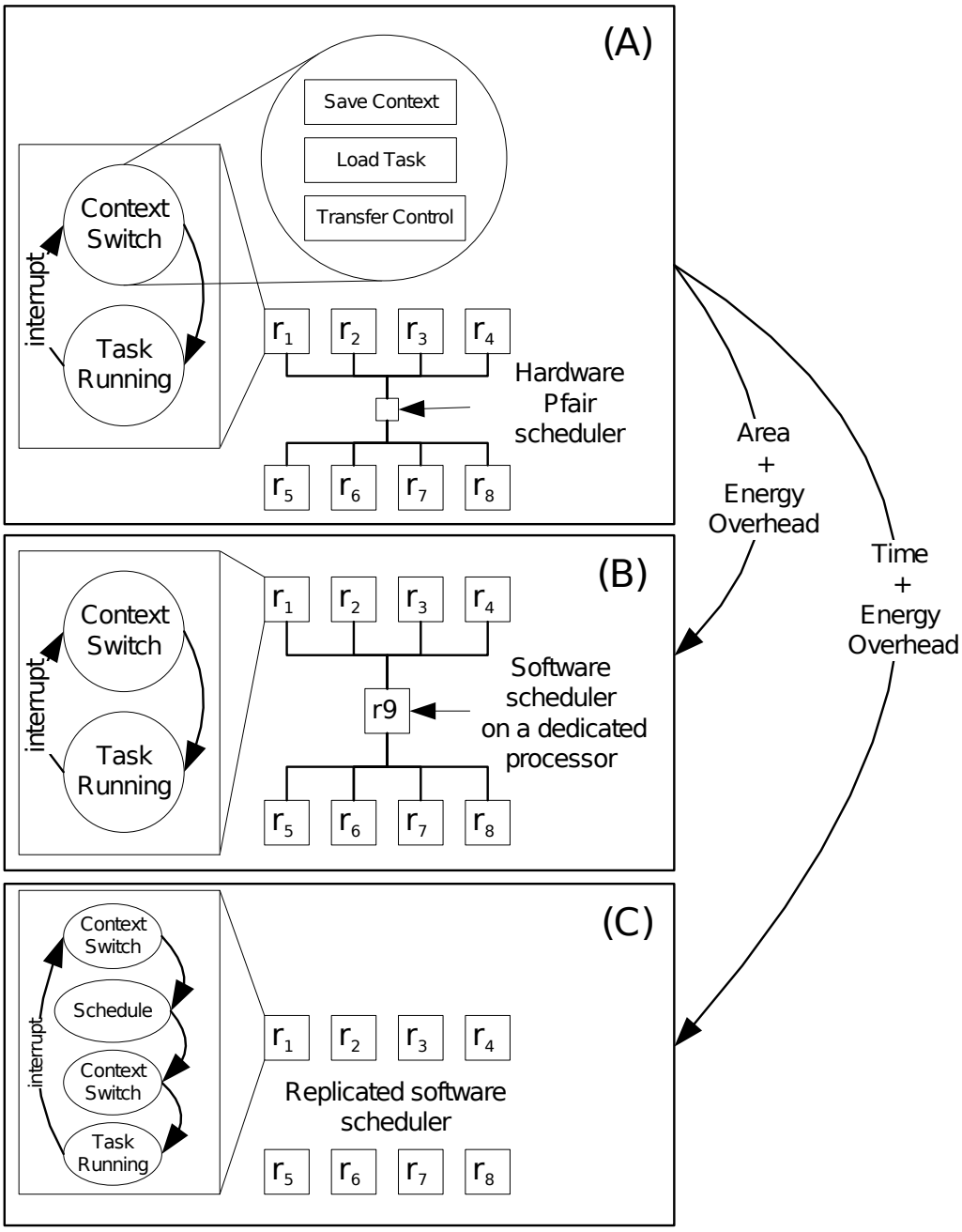


Fig. 6.1. Overview of the Three Different Implementation Options

This scheduling method is simple to implement since the already designed scheduler software can run on a separate processor and does not require many design changes.

However, this technique requires an extra processor and hence is costly in terms of area and power consumption.

6.3.3 Pfair Scheduler Core

The third option is an on chip dedicated hardware core running the scheduling algorithm, Figure 6.1(A). The hardware core works in a similar way as the dedicated software scheduler, but it has a faster response time and requires much lesser energy during operation. The nature of the Pfair scheduling algorithm offers many parallelization options for fast scheduling. Like dedicated software scheduling, this technique is free from the scheduling delay overhead. The area requirement of a dedicated scheduler core is also expected to be much less compared to a general purpose processor core.

6.4 Hardware Pfair Scheduler

The original Pfair algorithm does not yield itself to a straight forward hardware implementation. Pfair works by calculating the proportionate progress of a task from its period and execution time. The steps involve maintaining the data structures for each task which include information about the task and current slot number. Primary computations are calculation of lag, characteristic symbol and characteristic string. The original scheme of computation involves floating point multiplication for updating task lag and characteristic string. We have modified the definitions to make the computation incremental so that the only additions and comparisons are used. This was done by multiplying all the task parameters with the corresponding task's period, since all fractional values are result of division by period of the task. This eliminates the floating point computation required by the original Pfair scheme and simplifies the mathematics to a great deal. Also, as required by computation of

$\alpha, \lfloor w \cdot t \rfloor$ can be maintained in its integral equivalent after multiplication by period The modified definitions are as follows:

$$\text{Period} = p^2 \tag{6.1}$$

$$\text{Execution Time} = e \cdot p \tag{6.2}$$

$$\text{Weight} = e \tag{6.3}$$

$$S(t) = \begin{cases} e & \text{if the task is scheduled in slot } t \\ 0 & \text{otherwise} \end{cases} \tag{6.4}$$

$$\text{lag}(t) = \text{lag}(t-1) + e - S_{t-1} \tag{6.5}$$

$$\text{Ideal}(t) = \begin{cases} \text{Ideal}(t-1) + e & \text{if } t > 0 \\ 0 & \text{if } t = 0 \end{cases} \tag{6.6}$$

$$\text{FIdeal}(t) = \begin{cases} 0, & \text{if } t = 0 \\ \text{FIdeal}(t-1) + p, & \text{if Ideal}(t-1) + e \geq \text{FIdeal}(t-1) + p \\ \text{FIdeal}(t-1) & \text{otherwise} \end{cases} \tag{6.7}$$

$$\alpha(t) = \text{sign}(\text{Ideal}(t+1) - \text{FIdeal}(t) - p) \tag{6.8}$$

In the above definitions, $S(t)$ denotes whether or not the task has been scheduler in slot t. Ideal(t) is equivalent to $w \cdot t$ and FIdeal(t) is equivalent to $\lfloor w \cdot t \rfloor$. The definition of the sets urgent, tnegru and contending remain the same. The original Pfair algorithm can be found in [11] and has been kept unchanged. The significant change that had to be done was the evaluation total order of characteristic string in parallel. In the following subsections, we discuss the scheduler design in detail.

6.4.1 System Architecture

The steps of the Pfair scheduling algorithm are clearly reflected in the hardware design. The scheduler consists of the following main blocks:

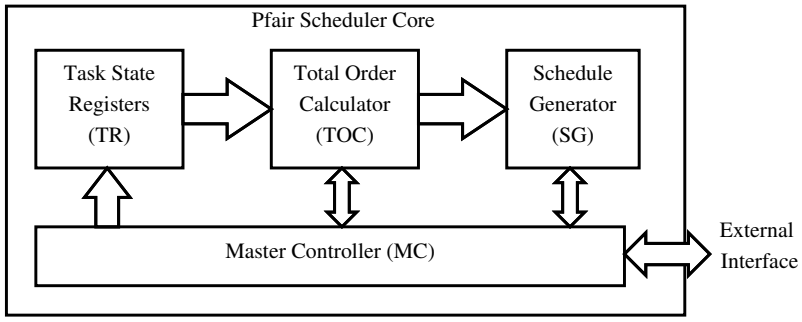


Fig. 6.2. Pfair Scheduler Block Schematic

1. Task State Registers (TR) along with logic to update the attributes lag and ideal
2. Total Order Calculator (TOC)
3. Schedule Generator (SG)
4. Master Controller (MC)

The overall organization of the hardware Pfair scheduler is illustrated in Figure 6.2. The details of each component are discussed in the following sections.

6.4.2 Task State Registers

This is the main data structure block in the design. It comprises of a persistent register for each task in the system. The fields in the register are shown in Figure 6.3. As our modified definitions are incremental, we only need to maintain the current values of task parameters. The fields are updated at the beginning of each slot using the incremental formulae 6.4-6.8 described earlier.

Along with the registers, this block also contains the logic that performs the computation required to update these fields at the beginning of each slot. The update logic is based on the incremental formulae in equations 6.4–6.8. This block can compute the urgent, tnegru and the contending sets at the beginning of each

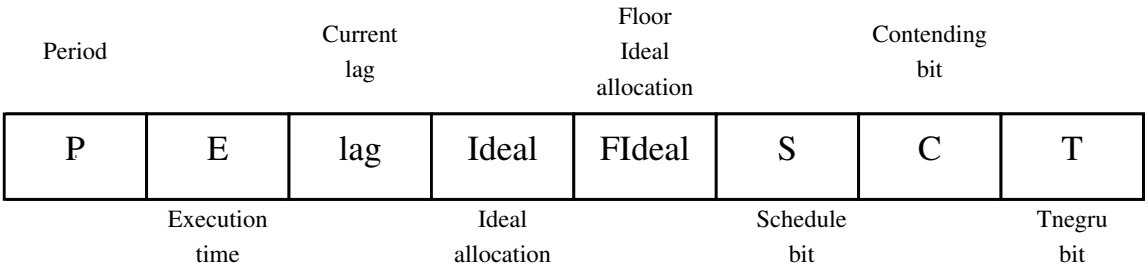


Fig. 6.3. Fields in a Task Register

slot. The next module does the total ordering of the contending set by looking at the contending bits in the task registers.

6.4.3 Total Order Calculator

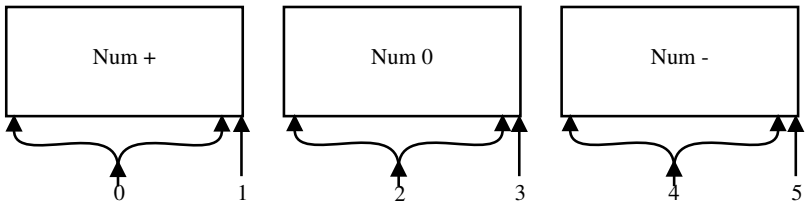


Fig. 6.4. Total Order Calculator

This module calculates the total order needed to select tasks from the contending set. The total ordering is defined over the characteristic string of each task in the contending set. This module, incrementally calculates the characteristic string of the relevant tasks as the calculation of total order progresses. We optimize this process by computing the characteristic string depending on the number of tasks to be selected from the contending set. At each stage of the incremental process, we disable tasks from being considered in subsequent iterations by looking at a mask which is calculated based on the number of '+' symbols, '0' symbols and '-' symbols

as in Figure 6.4. In the figure the arrows show the relative position of the number of tasks to be selected and the corresponding mask values. Mask values of 1, 3 and 5 represent completion of the total order calculation process. For mask values 0 (resp. 2) tasks with characteristic symbol 0, - (resp. -), do not need to be further evaluated. This technique is easily implemented in hardware and greatly simplifies the evaluation. The mask value is the output of this module and is used by the schedule generator.

6.4.4 Schedule Generator

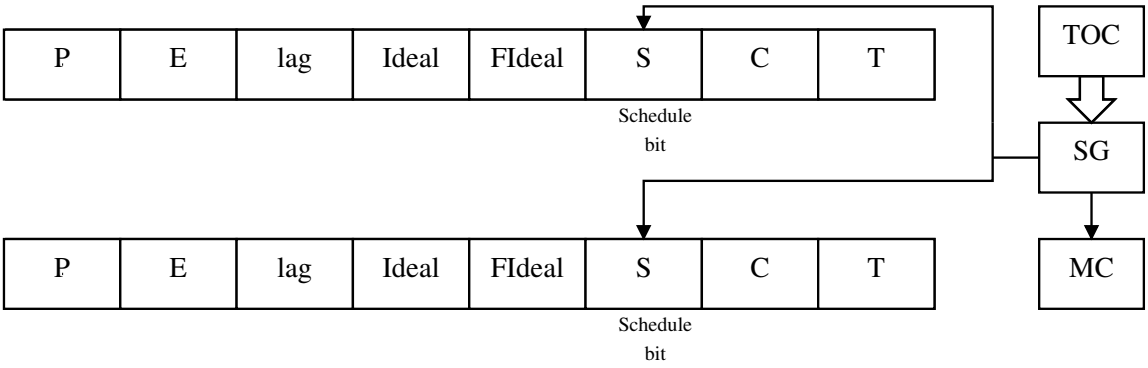


Fig. 6.5. Schedule Generator

This module sets the schedule bits for the tasks that are selected based on the total order, Figure 6.5. It does so by interpreting the scheduling mask generated by the total order calculator at the end of each pass. When all the tasks are scheduled from the contending set or all the processors are allocated a task, the process completes for that slot. The schedule generator notifies the master controller on completion of schedule generation and master controller stores the generated schedule and presents them to the processor interrupt service routine (ISR) as requested.

6.4.5 Master Controller

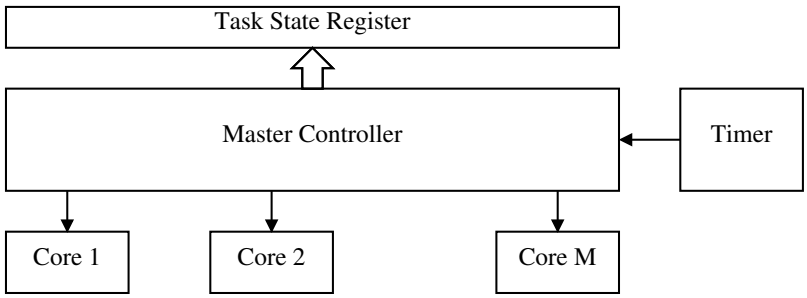


Fig. 6.6. Master Controller

The heart of the master controller is essentially a state machine described in Figure 6.7. In addition to the scheduling state machines, it also implements the interfacing logic to the processors, Figure 6.6. The master controller is woken up using a timer at the beginning of each slot and it performs the scheduling task for next k slots and stores the generated schedule. Upon timer expiration, it checks if the schedule is already computed. If schedule details are available it immediately interrupts the processors that need to do a context switch. Otherwise, it runs the scheduler to compute the schedule for the next k slots. The master controller also provides an interface to program the task set at the time of system startup.

6.4.6 Scheduler Operation

The scheduler is invoked by a timer. The timer can be programmed during the system startup to fire with a period same as the slot duration. The scheduler first checks whether the scheduling decision for that slot is already present or not. If not, it calculates the schedule. Next, it checks which processors need to be interrupted for a task switch. It then sends interrupt signals to those processors to invoke the context switch ISR. At this point the ISRs query the scheduler core using some addressing scheme for the next task to run. This process can be completed in one

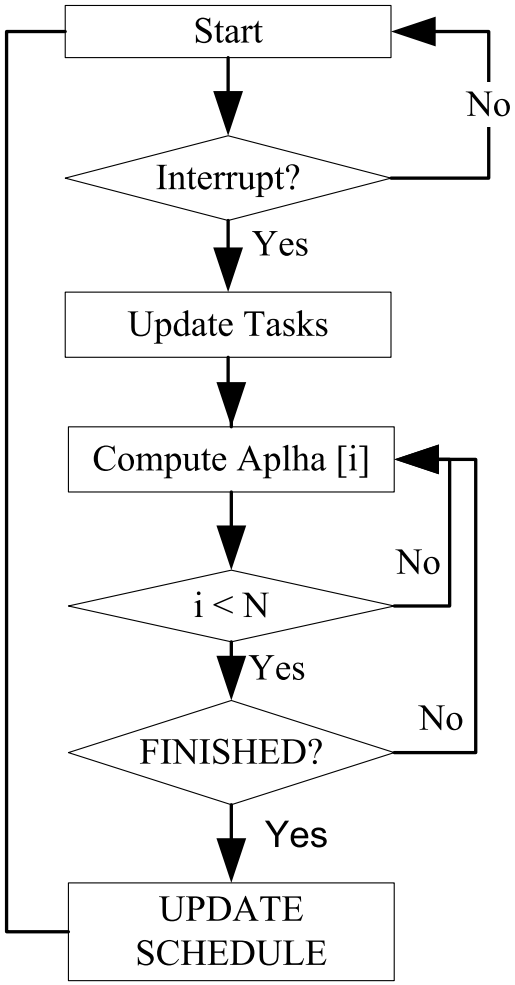


Fig. 6.7. FSM of the Scheduler Core

memory access. The ISR can immediately switch to the next task to run it. The scheduler core only needs to respond to the queries from the ISRs once it has sent the interrupts. So it goes to idle mode after sending the interrupt signals to save energy.

6.5 Results and Discussions

In this section we will compare the performance and predictability of our hardware Pfair scheduler with that of the other two implementation schemes.

6.5.1 Experimental Setup

To obtain time and power estimates of the software version, the scheduling algorithm was implemented in C and was run on an ARM power performance simulator based on SimpleScalar [58] [6]. The ARM simulator can perform a low level power performance simulation of an ARM binary running on an Intel SA1100 processor. For the hardware Pfair scheduler, Synopsys design compiler was used for synthesis power, area and timing results. We use the H.263 benchmark from the DSPstone suite. All the three different implementation options were evaluated. The results of the simulations are reported in section 6.5.2.

Benchmarks

The benchmark selected was H.263 from DSPstone suite [59]. H.263 is a video codec standard originally designed as a low- bitrate compressed format for video conferencing. This requires soft real-time processing. We dissociated a portion of this application into the following subtasks: DCT, Dequantization, IDCT, Quantization and calculation of SAD (sum of absolute division). All these were applied on an 8x8 macroblock in a pipelined fashion. By partitioning H.263, we get the execution time and period for each of the individual subtasks. Each subtask was cross-compiled for the ARM architecture (StrongARM) and simulated with SimpleScalar. The periods were obtained by assuming different frame rates used in real applications as in Table 6.1. We assume an image resolution of 480x240 resulting in 300 macroblocks per frame. Table 6.1 lists all the details of the benchmark that we have used. Four taskset configurations were generated with utilizations varying from 5 to 17. We used the minimum possible number of processors to schedule each taskset.

Table 6.1
DSPStone Based Benchmark Details

Taskset Number	**Frame Rate**	**Utilization**	**Number of Tasks**	**Number of Processors**
1	5	5.25	10	6
2	12	11.15	25	12
3	15	13.25	30	14
4	24	16.57	50	17

Evaluation Criteria

We have evaluated the scheduler core design in terms of speed, area and power. We define each property as follows:

Speed: We measured the number of cycles taken by the scheduler to perform the scheduling task. We also considered the length of the ISR running in the processors while calculating the overall scheduling time. We then use the scheduler frequency to calculate the absolute time required to schedule the tasks.

Area: The scheduler core was synthesized using 90 nm process technology from Synopsys [57]. Synopsys design compiler was used to obtain area estimates using the library. We compared the area of the scheduler core to that of an additional ARM core which can be used in the dedicated software scheduler.

Power: The primary motivation of the scheduler core design being low power, this is the most important metric in our evaluation. We estimate the static and dynamic power consumption in the scheduler core using Synopsys Power Compiler. We compare the power and energy estimates to those of replicated scheduler scheme and dedicated software scheduling scheme. Although instantaneous power of our scheduler core can be higher than the software schedulers, the overall energy consumption is much lower. The corresponding results have been illustrated in section 6.5.2.

6.5.2 Results

In this section we discuss the results on scheduling delay and energy consumption obtained by running real-time benchmark on the three implementations. The scheduling speed is compared in terms of the number of cycles required to schedule a slot. This is followed by synthesis results of the Pfair scheduler core.

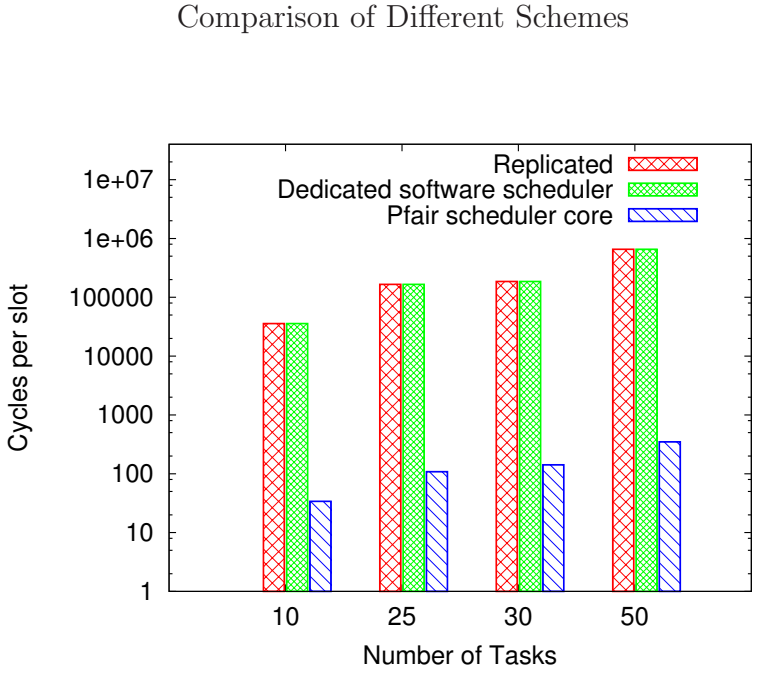


Fig. 6.8. Comparison of Scheduling Delay with Varying Number of Tasks

First we discuss the results on the scheduling delay. The scheduling delay for the replicated, and dedicated software scheduler were found out using the Simplescalar simulator [6]. We have used the Synopsys VCS simulator to measure the scheduling delay of the Pfair scheduler core. Figure 6.8 shows the scheduling delay due to the three different implementation schemes. The replicated and the dedicated software schemes yield the same scheduling delay. But the Pfair scheduler core shows an order of magnitude improvement (10^3) in scheduling delay. For example, in the case

of taskset having 30 tasks, which uses 14 processors, the software schemes completes the scheduling operation for one slot in 185937 cycles whereas the hardware scheme completes the same in 142 cycles. In real-time applications that have tasks with very small periods, the scheduling decisions need to be made much quickly compared to the periods of the task set. Hence, the Pfair scheduler core should be the preferred choice for such systems.

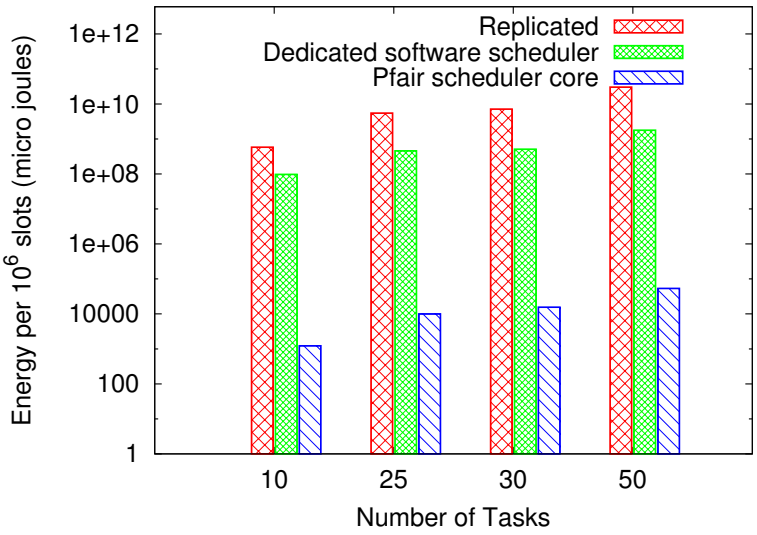


Fig. 6.9. Comparison of Scheduling Energy with Varying Number of Tasks

The energy consumption of the Pfair scheduler core is compared to the dedicated software and replicated implementation schemes in Figure 6.9. To calculate energy consumption of the Pfair scheduler core, we used the power values from the synthesis results. The scheduling delay per slot was obtained using Synopsys VCS simulation. For the dedicated software and replicated implementations, an ARM power performance simulator based on the SimpleScalar was used [58]. The dedicated Pfair scheduler yields an order of magnitude (10^5) improvement over both the schemes. For example, in the case of task set having 30 tasks that uses 14 processors, the replicated scheme consumes 508 micro joules whereas the hardware scheme

consumes only 15 nano joules. The low scheduling delay and the low energy consumption of the Pfair scheduler core make it an attractive choice for use in a low power multiprocessor system on chip.

Pfair Scheduler Core Synthesis Results

We have synthesized the design using TSMC 90nm process technology. As discussed in the section 6.4, the area and power consumption depend on the number of tasks supported. We have synthesized designs to support task sets with 10 to 50 tasks. Figures 6.10 and 6.11 show these synthesis results. As expected, the power and area consumption of the design increases with the number of tasks. This is due to the linear increase in the number of state registers as the number of tasks increases. This replication maintains the scheduling delay as the number of tasks increases (Figure 6.8).

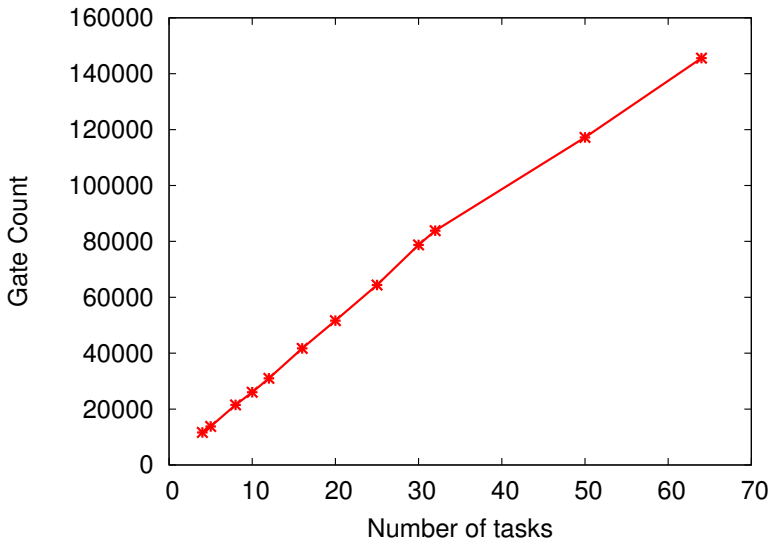


Fig. 6.10. Synthesis Gate Count

The total power consumption of the hardware design is shown in Figure 6.11. We used Synopsys design compiler to get the Dynamic and leakage power of the circuit.

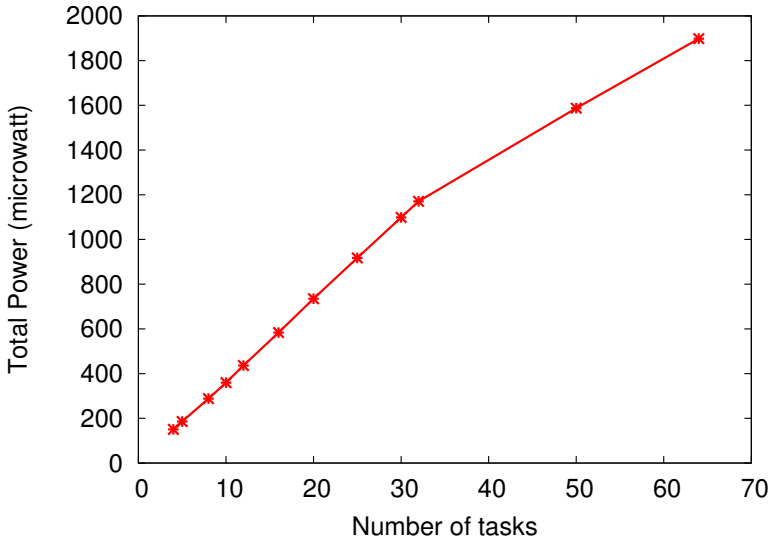


Fig. 6.11. Synthesis Total power

The total power consumption is in microwatts, which is an order of magnitude (10^3) lesser than the power consumption in dedicated software schedulers.

6.6 Conclusions and Future Work

The imminent requirements of high performance embedded systems will require multi core embedded design to be in place. Scheduling real-time tasks on such platform has to be efficient and optimal to obtain maximum performance. The proposed energy efficient hardware scheduler core can provide such performance at a reduced energy cost. Experimental evaluation has shown a 10^3 order improvement in scheduling delay while consuming 10^5 orders less energy. Future work includes incorporating low power techniques such as dynamic voltage/frequency scaling to further increase the efficiency of the system.

7. TEMPERATURE AWARE DYNAMIC POWER MANAGEMENT[1]

In recent times, the power densities of microprocessors have doubled every three years. This increase in power densities has led to two major problems. Firstly, high energy consumption is a limitation for mobile, battery operated devices. Secondly, higher temperatures directly affect reliability and cooling costs. Current estimates predict that cooling costs will rise at $1-$3 per watt of heat dissipated [53]. The power consumption in microprocessors mainly consists of two components; dynamic power consumption, and leakage power consumption. Dynamic power consumption is a result of transistor switching activity whereas the leakage power is due to the leakage current and is dependent on the system temperature. A feedback loop exists between temperature and leakage power. Higher temperature increases leakage power which in turn increases temperature. Hence it is extremely important to manage energy consumption and temperature in current microprocessors. In this work we present an online temperature aware energy management technique for real-time applications with varying execution times.

Past work on temperature aware energy management has mostly focused on tasks with fixed execution times [9, 10, 18, 19, 38, 39, 60]. However, in practical situations, task execution times are rarely fixed. It is possible to estimate a task's worst case execution time (WCET), but the actual execution times can be as less as 50% of the WCET. Therefore, a scheme that takes this variation of execution times into consideration is highly desirable. Past efforts that take the variation of task execution times into consideration have not considered temperature awareness [40, 41, 66]. In this work we combine the past efforts on energy management using feedback control and temperature aware energy management.

Energy consumption is a cumulative measure over a time interval whereas temperature is an instantaneous property of the system. Hence a scheme that minimizes

[1]The work in this section has been derived from the paper [31]

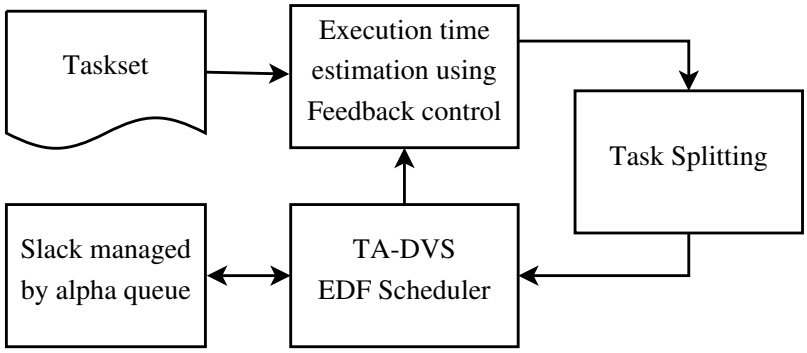


Fig. 7.1. Flowchart of the TA-DVS System

energy consumption might exhibit a large number of temperature constraint violations. We consider a temperature constraint on the system, the violations of which, should be avoided as much as possible. Our primary goal is to reduce energy consumption. The secondary goal is to reduce violations of the temperature constraint.

Dynamic voltage/frequency scaling (DVS) is a widely used technique to reduce energy consumption, in both research and practice. We propose a temperature aware DVS scheme that reduces the number of temperature constraint violation while reducing energy consumption. Our scheme is based on a feedback controller which increases energy savings over basic DVS approach by estimating the execution time for the next job of a task. Based on the feedback controller's estimate we split a task into two portions. We use the α-queue technique to manage the slack available for a task at run time [8]. The TA-DVS technique computes the execution speed by managing the amount of slack available for each portion of a task such that the temperature does not exceed the temperature constraint. Figure 7.1 shows the integration of these different components in our system.

The primary technical contributions of our work are as follows:

1. Experimentally showed that when the canonical speed (defined later) is less than or equal to the equilibrium speed (described later), energy management is sufficient to satisfy the system temperature constraint.

2. Further, when the canonical speed is greater than the equilibrium speed we propose a temperature aware energy management technique, TA-DVS, that reduces the number of temperature constraint violations while still reducing energy consumption.

3. Demonstrated through simulation data that TA-DVS reduces temperature constraint violations by 18.9% on the average compared to existing schemes.

This section is organized as follows: We discuss related work in section 7.1. Section 7.1.1 lays out the thermal model used in the work. We describe the slack management technique used in this section in section 7.2. In section 7.3 we discuss the task splitting approach followed by a discussion of feedback control in section 7.4. Section 7.5 presents our temperature aware dynamic voltage scaling technique. We present the experiments and results in section 7.6. Finally, section 7.7 concludes this section.

7.1 Related Work

In this section, we briefly discuss existing energy management and temperature aware scheduling techniques for real-time systems.

Energy management for tasks with varying execution times has been widely studied in the literature [8, 51, 62, 66]. Pillai et al. considered static and dynamic voltage scaling to reduce energy consumption considering the EDF and RM scheduling techniques [51]. Zhu et al. improved over [51] by using a feedback controller to estimate the actual execution time of tasks [66]. In [8], Aydin et al. propose solutions to reduce energy consumption by using a speculative speed adjustment algorithm that anticipates early completions by tracking average case workload information.

Past work on temperature aware energy management has mostly focused on tasks with fixed execution times [9, 10, 18, 19, 38, 39, 60]. Chen et al. studied the problem of minimizing the maximum temperature. Their work concentrates on approximation

bounds for minimization of maximum temperature by considering continuous and discrete speed levels on uniprocessor and multiprocessor environments [19]. Wang et al. and Bansal et al. studied energy efficient speed scheduling under thermal constraints for a frame based taskset [60], [9]. Bao et al. developed an online temperature aware DVFS technique considering the frequency/temperature dependency to increase energy savings [10].

Temperature aware task scheduling for non real-time applications to reduce thermal gradients and thermal cycles has been studied in [24].

To the best of our knowledge this is the first work considering temperature aware energy management using a feedback controller for real-time tasks with varying execution times.

7.1.1 Energy Model Augmentations and Thermal Model

We use energy and thermal models similar to those in [60]. Our temperature aware scheduling approach concentrates on DVS enabled processors. The processor is assumed to have m discrete speed levels; $s_1, s_2, .., s_l$. The total power consumption of a DVS processor can be represented by:

$$\Psi(s, \Theta) = hs^{\gamma} + \delta\Theta + \rho \tag{7.1}$$

Here, hs^{γ} is the speed dependent power component and $\delta\Theta + \rho$ is the speed independent power component. The speed, $s = s(t)$ and the absolute temperature, $\Theta = \Theta(t)$ are functions of time t. In the above equation, h and γ are constants such that $\gamma \leq 3$. The speed independent power component mainly resulting from leakage current is often dependent on the temperature. Hence, we model it as a linear function of temperature. The energy consumed during the time interval $[t_0, t_1]$ is given by $\int_{t_0}^{t_1} \Psi(s(t), \Theta(t))dt$.

Considering cooling effects and ambient temperature, the variation of temperature with time is given by:

$$\Theta'(t) = \alpha s^{\gamma}(t) - \beta\Theta(t) - \sigma \tag{7.2}$$

where α, β and σ are the constants. To simplify the notation, we set $\theta(t) = \frac{\Theta(t)}{\alpha} - \frac{\sigma}{\alpha\beta}$, where $\theta(t)$ is the adjusted temperature. Equation 7.2 can now be simplified as:

$$\theta'(t) = s^{\gamma}(t) - \beta\theta(t). \tag{7.3}$$

From equation 7.3, the equilibrium temperature for a given speed s is given by:

$$\theta = s^{\gamma}/\beta \tag{7.4}$$

Similarly, for a given temperature constraint, θ^*, the corresponding equilibrium speed is given by:

$$s_e = (\beta\theta^*)^{1/\gamma} \tag{7.5}$$

The energy overhead of changing the processor speed is dependent on the speed levels before and and after the speed change. We assume that changing the processor speed from s_i to s_j incurs an energy overhead given by $E = \xi \mid s_i^2 - s_j^2 \mid$ where ξ is a constant.

7.2 Slack Management

Actual execution times of tasks are often lesser than their worst case execution times. These unused execution cycles are termed as slack. Efficient management of static and dynamic slack is necessary to manage energy consumption in dynamic systems with varying execution times. In our system, we manage static slack by releasing each task with its speed set to the canonical speed of the system. We define canonical speed as minimum available speed greater than or equal to the

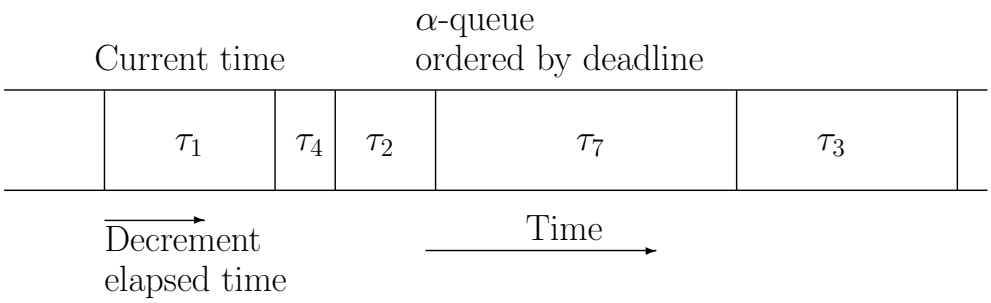


Fig. 7.2. α-Queue for Computation of Dynamic Slack

taskset utilization, U_{tot}. Aydin et al. showed that when the EDF scheduling policy is used, by releasing tasks at canonical speed, we can minimize the energy consumption statically while meeting all deadlines. To account for dynamic slack we use the α-queue technique developed in [8]. The α-queue technique computes the amount of available slack for a task by calculating its earliness compared to the statically optimal schedule. To compute the earliness of a task we maintain two parameters for each task during the schedule.

- rem_i, the remaining execution time of τ_i in the statically optimal schedule.
- $wcet_i$, the remaining worst case execution time of τ_i in the current schedule.

To maintain rem_i, upon arrival, each task pushes its worst case execution time at the canonical speed onto the α-queue, which is totally ordered according to the EDF policy (ties between tasks are broken in a consistent manner). The element at the head of the queue is always decremented by the amount of time elapsed. This results in a dynamic image of the ready queue in the statically optimal schedule, Figure 7.2. It can be shown that at any given time, the dynamic slack available for a task τ_x is no less than $\sum_{d_i<d_x} rem_i + rem_x - wcet_x$, where d_i represents the absolute deadline for task τ_i. This is because tasks with higher priority than τ_x must have already finished in the actual schedule.

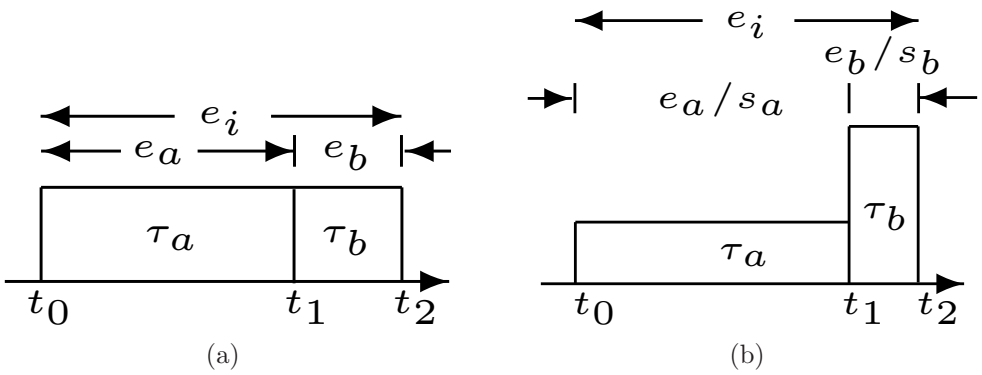


Fig. 7.3. Task Splitting

We use task splitting in conjunction with feedback control to exploit the dynamic slack created in the system. Dynamic slack is created due to the fluctuation in actual execution times of jobs from a given task. Let l_i denote the amount of dynamic slack available when a job of the task τ_i is released. Using this dynamic slack, the speed of τ_i can be reduced to:

$$s_i = \left(\frac{e_i}{e_i + l_i} \right) * s_l \tag{7.6}$$

As in Figure 7.3a, task splitting divides a given task τ_i into two subtasks, τ_a and τ_b having execution times e_a and e_b respectively, such that $e_i = e_a + e_b$. The two subtasks are allowed to execute at different speeds, Figure 7.3b. A feedback controller is used to adjust the value of e_a for each task. We expect that the execution of task τ_i will finish within e_a execution cycles. We reserve enough time in subtask τ_b so that τ_i meets its deadline even if it requires its worst case execution time. This allows us to use an even lower speed for subtask τ_a, given by:

$$s_a = \left(\frac{e_a}{e_a + l_i} \right) * s_l \tag{7.7}$$

Splitting a given tasks' execution time into two subtasks is sufficient for applying feedback control because the controller adjusts the execution time of only a single subtask. Hence we avoid creating more than two subtasks which would lead to unnecessary overheads without any benefits. Equation 7.7 assumes that we use all the available slack for subtask τ_a. In section 7.5 we will show that this choice may violate the temperature constraint. It can be shown that task splitting is necessary to apply feedback control in hard real-time systems.

7.4 Feedback Control

In real systems, the actual execution time of different jobs from a given task often fluctuates over time. Earlier work on dynamic real-time scheduling has shown that feedback control is a useful technique for enhancing the schedule by reacting to fluctuations in execution time. In section 7.3, we described how task splitting can reduce energy consumption if we can estimate the actual execution time of the next job. In this section, we focus on determining a value for e_a using feedback control.

We use a PID-feedback controller in our system to control the execution time of τ_a. A PID-feedback controlled system has a controlled variable, a set point and an output. The feedback controller changes the output so that the value of the controlled variable remains the same as the set point. A PID-controller has three terms, namely, proportional control, integral control and derivative control. The proportional term controls the reaction to the current error from the set point. The integral term controls the reaction to the history of recent errors and the derivative term controls the reaction to the rate of error change. In our system, we use the actual execution time of a task, a_i, as the set point and e_a as the controlled variable. The system error is defined as:

$$e_{ij} = e_{a_{ij}} - a_{ij} \tag{7.8}$$

We use the following feedback control formula to adjust the value of e_a:

$$\begin{aligned} \Delta e_{a_{ij}} &= K_P * e_{ij} + \frac{1}{K_I} \sum_{j \in IW} e_{ij} + K_D * (e_{ij} - e_{i(j-1)}) \\ e_{a_{i(j+1)}} &= e_{a_{ij}} + \Delta e_{a_{ij}} \end{aligned} \tag{7.9}$$

Here K_P, K_I and K_D are the proportional, integral and derivative parameters respectively, and IW is the length of the window for recent histories. In our experiments we use $K_P = 0.9$, $K_I = 0.08$ and $K_D = 0.1$. These values based on trial and error were found to perform best in terms of accurately adjusting the values of e_a.

7.5 Temperature Aware Energy Management

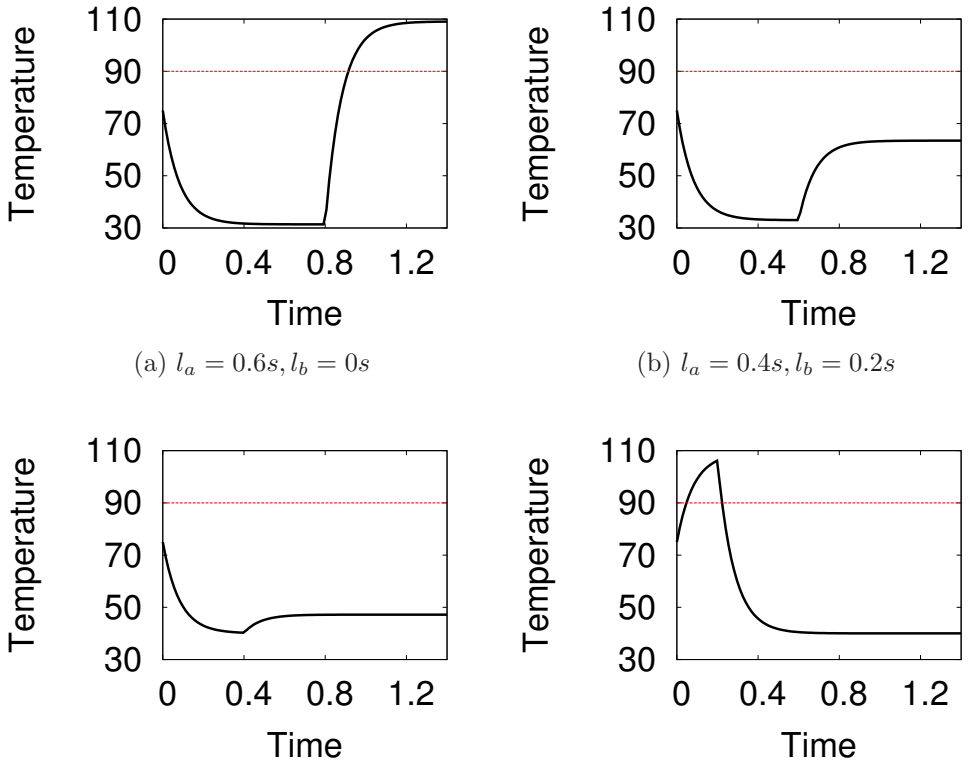
We explore temperature aware energy management. We assume that the system has a temperature constraint $\theta_c(\Theta_c$ adjusted). Violations of this temperature constraint should be avoided as much as possible, even if the subtask τ_b executes. As mentioned in section 7.3, to meet the temperature constraint, it might not be best to use all the slack for subtask τ_a. If energy minimization was the only goal, then we could use all the slack for τ_a and run τ_b at maximum speed if required as in [66]. However, for temperature aware energy minimization we distribute the slack between τ_a and τ_b such that $\theta(t) \leq \theta_c$ during $[t_0, t_2]$. It is worthwhile to note here that, during an interval of time $[t_0, t_2]$ the energy consumption depends only on the speeds used, whereas the temperature during the time interval also depends on the temperature at t_0. Solving equation 7.3 using an integrating factor, we get:

$$\theta(t_1) = s_a^\gamma/\beta + e^{\beta(t_0 - t_1)}(\theta(t_0) - s_a^\gamma/\beta) \tag{7.10}$$

$$\theta(t_2) = s_b^\gamma/\beta + e^{\beta(t_1 - t_2)}(\theta(t_1) - s_b^\gamma/\beta) \tag{7.11}$$

Let l_a and l_b denote the amounts of slack allotted to subtasks τ_a and τ_b respectively such that $l_a + l_b = l_i$. So, we get:

$$s_a = \left(\frac{e_a}{e_a + l_a}\right) * s_l, s_b = \left(\frac{e_b}{e_b + l_b}\right) * s_l, \tag{7.12}$$



(a) $l_a = 0.6s, l_b = 0s$ (b) $l_a = 0.4s, l_b = 0.2s$

(c) $l_a = 0.2s, l_b = 0.4s$ (d) $l_a = 0s, l_b = 0.6s$

Fig. 7.4. Temperature Variation with Slack Distribution

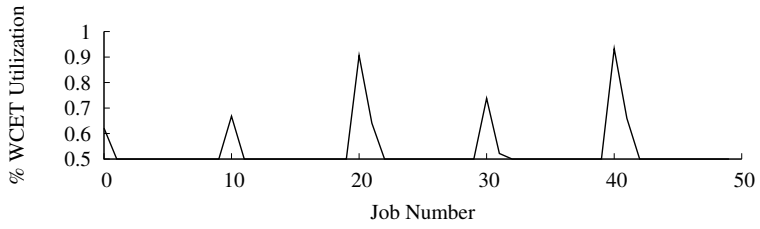
As a motivational example consider a task with $e_i = 0.8s, e_a = 0.2s, e_b = 0.6s$ and $l_i = 0.6s$. Figure 7.4 shows the temperature curves during $[t_0, t_2]$ assuming different values of l_a. The temperature at t_0 is assumed to be 75 °C. Assuming $\Theta_c =$ 90 °C, it can be clearly seen that both Figures 7.4a and 7.4d violate the temperature constraint, while Figures 7.4b and 7.4c obey the same. Here, Figure 7.4b represents a better speed schedule than Figure 7.4c because it uses a lesser speed for

subtask τ_a. Our goal is to use most of the slack for τ_a while reserving slack for τ_b so that the temperature constraint is not violated even if τ_b executes. All the four speed schedules result in comparable energy consumptions.

To minimize the energy consumption while obeying the temperature constraint, we need to determine suitable values for l_a and l_b. The values of l_a and l_b give the speeds to be used for τ_a and τ_b according to equation 7.12. To maximize the energy savings while meeting temperature constraints, we want $\theta(t_2)$ to be as close to θ_c as possible. In this way, the minimum possible slack is used for τ_b while maximizing the slack used for τ_a. It is important to note here that TA-DVS can not guarantee that the temperature constraint will always be satisfied. The amount of available slack may not be enough to obey the temperature constraint. Rather, TA-DVS is a best effort solution to reduce the number of temperature constraint violations.

Algorithm 5
TA-DVS: Calculate l_a and l_b

1: **for** $i = 1$ to l **do**
2: $s_a \leftarrow s_i$
3: $s_b \leftarrow \dfrac{e_b s_l}{e_b + l_i - e_a(s_l/s_a - 1)}$
4: Compute $\theta(t_1)$ and $\theta(t_2)$ according to equation 7.10,7.11
5: **if** $\theta(t_2), \theta(t_1) \leq \theta_c$ **then**
6: break
7: **end if**
8: **if** $i = l$ **then**
9: $s_a \leftarrow$ canonical speed
10: **end if**
11: **end for**
12: $l_a \leftarrow e_a * (s_l/s_a - 1)$
13: $l_b \leftarrow l_i - l_a$



][!h]

Fig. 7.5. Pattern of Variation for Actual Execution Times of a Task

Algorithm 5 shows the pseudo-code for calculating l_a and l_b. We want to use the maximum possible slack for τ_a to maximize the energy savings. Hence, the algorithm iterates through the list of available speeds starting from the lowest. Using this speed for s_a, the values of s_b and $\theta(t_2)$ are calculated. The *for* loop ends when $\theta(t_1)$ and $\theta(t_2)$ are within θ_c for the first time. This speed schedule achieves the maximum possible energy savings while obeying the temperature constraints during both τ_a and τ_b if possible. If none of the available speeds satisfy the condition on line 5, it means that the amount of available slack is insufficient to satisfy the temperature constraint. In this case we choose the canonical speed as the speed for subtask τ_a.

7.6 Results and Discussions

We evaluated the performance of TA-DVS in a simulation environment which implements task splitting, feedback control and α-queue. For comparison purposes, we also implemented two other energy management schemes. The first one, *Multiple Feedback Dynamic Voltage Scaling* approach, referred to as MF-DVS performs temperature unaware energy management using feedback control [66]. The second scheme, *Temperature Aware Speed Control*, referred to as TASC performs temperature constraint aware energy management without taking dynamic slack into account [60]. We generated fifteen tasksets, each with ten tasks, and total worst case utilization varying from [0.85, 1.00] in intervals of 0.01. The periods of the tasks were uniformly distributed in the range [1,100] sec. The worst case execution time

(WCET) of each task was chosen to meet the total utilization of the taskset. The actual execution time of each task was distributed in the range $[0.5, 1.0] \times$ WCET and followed the pattern shown in Figure 7.5. In this pattern, the actual execution time of the task remains at 50% of the WCET and spikes to a peak value every tenth job. The peak value is normally distributed in the range [0.5 $\times$ WCET, 1.0 $\times$ WCET]. After the peak, the actual execution time of the task falls off exponentially.

We assume a DVS capable processor with eight different normalized speed levels {0.25, 0.5, 0.63, 0.75, 0.81, 0.87, 0.93, 1.0}. The maximum frequency of the processor is assumed to be 2.1 GHz. Since the speed levels are discrete, we use the closest speed which is no lesser than that resulting from equation 7.12. We experiment with temperature constraint values in the range of $[80, 95]^{\circ}C$. Hence we assume a fine grained control over the speed at corresponding equilibrium speeds. We set the length of the integral window, IW = 10. Each individual run in our experiments was of length 500 sec. In all our experiments, we use the following values for parameters in the system model: $h = 6,\ \gamma = 3,\ \delta = 0.01,\ \rho = 0.1$ Watt, $\alpha = 105$ K/Joule, $\beta = 12.325$ sec^{-1}, $\sigma = 371.5$ K/sec and $\xi = 2.52\ \mu J$ [60].

In all our experiments, none of the jobs missed a deadline, showing that temperature aware energy management is a safe technique to use in terms of real-time constraints.

In the first experiment, we observed the number of temperature constraint violations with varying taskset utilizations. A temperature constraint of 90°C was used in this experiment which corresponds to an equilibrium speed of 0.91. Figure 7.6 shows the results. For the tasksets with utilization 0.86 and 0.87, the processor can always run at the canonical speed of 0.87 or lower. Since the equilibrium speed is 0.91, there are no constraint violations with any of the schemes. From this result we can directly conclude that when the canonical speeds is less than or equal to the equilibrium speed, energy management using appropriate slack reclamation is sufficient to satisfy temperature constraints. At utilizations of 0.87 and 0.93, we see

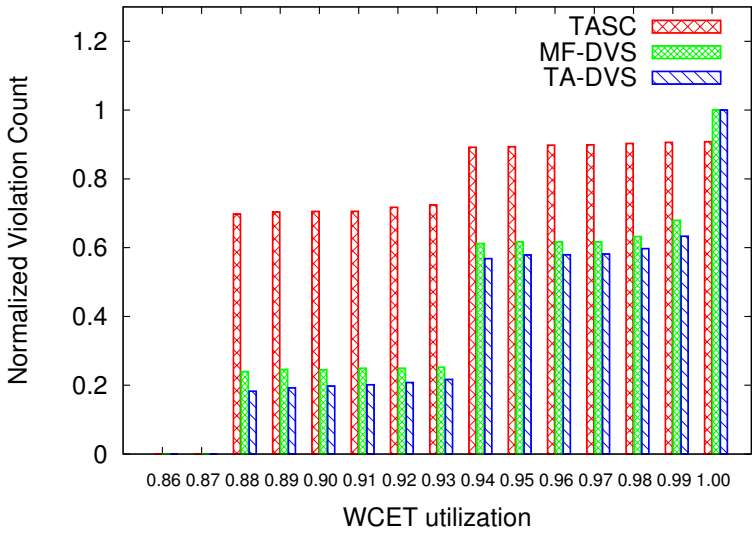


Fig. 7.6. Temperature Constraint Violations for $\Theta^* = 90°C$, with Varying Taskset Utilization

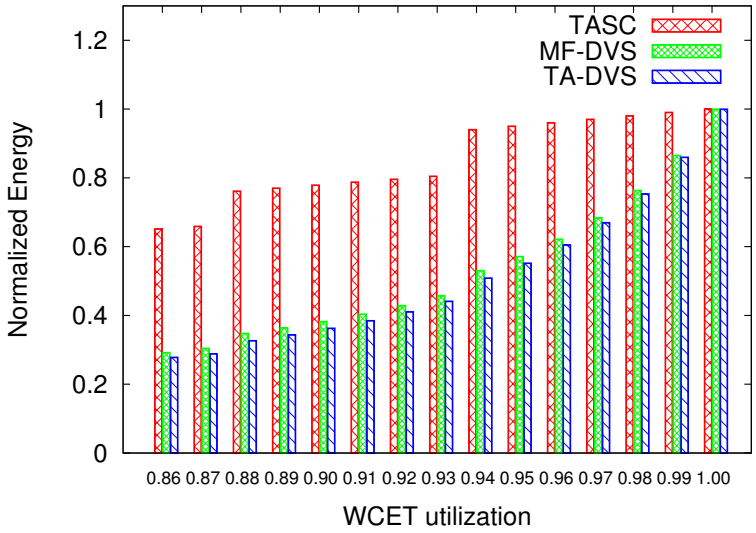


Fig. 7.7. Normalized Energy with Varying Taskset Utilization

an increase in the number of violations. Beyond utilizations of 0.87 and 0.93, the next available speeds are 0.93 and 1.00 which result in equilibrium temperatures of 93.6° and 109°C respectively. Due to the discreteness in the speed levels, we see a

jump in the number of violations for these tasksets. However, TA-DVS is able to reduce the number of violations by an average of 18.9% over MF-DVS. Since TASC does not take advantage of the dynamic slack created, it results in a considerably high number of constraint violations. At utilization of 1.00, wee see that TA-DVS and MF-DVS actually result in a higher number of constraint violations compared to TASC. This is due to the fact TASC does not try to utilize runtime slack to reduce energy consumption. Our results suggest that, at high utilizations, it might be a better idea to let the processor idle than to try and reduce energy consumption which might increase temperature constraint violations.

Figure 7.7 shows the normalized energy consumption for the same tasksets. As expected, with increasing utilization, the normalized energy consumption increases for all three schemes. For all utilizations, the energy consumption of the TA-DVS scheme is comparable to that of MF-DVS. Since TASC does not take advantage of the dynamic slack created, it leads to considerably higher energy consumption.

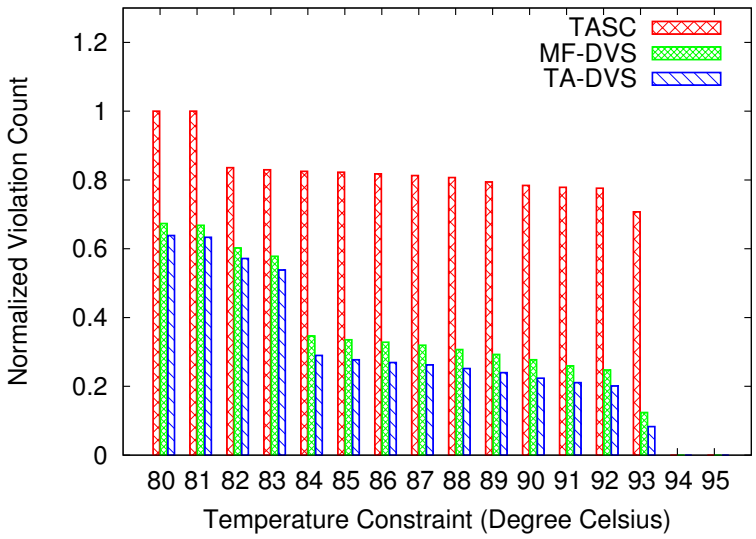


Fig. 7.8. Constraint Violations for WCET Utilization = 0.91 with Varying Temperature Constraints

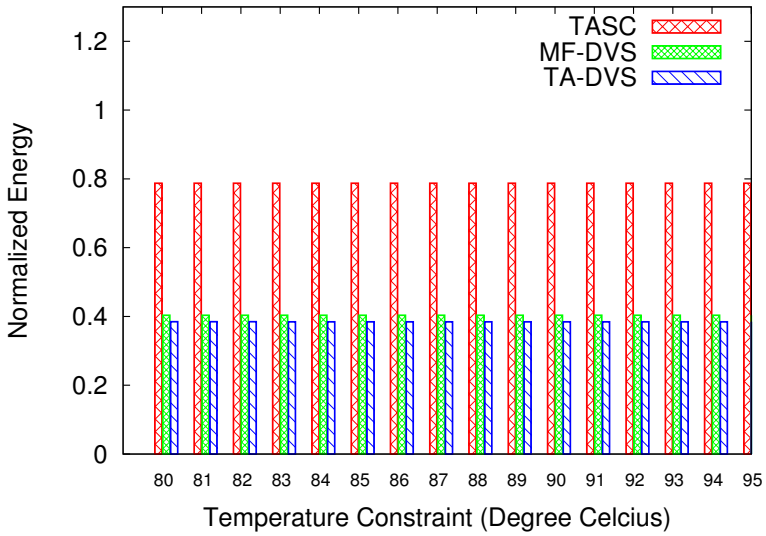


Fig. 7.9. Normalized Energy for WCET Utilization = 0.91 with Varying Temperature Constraints

In the second experiment, we observe the number of constraint violations by varying the temperature constraint itself. Figure 7.8 shows the results for a taskset with utilization 0.91. For utilization 0.91, the canonical speed is 0.93 and the corresponding equilibrium temperature is 93.6°. Hence for constraints beyond 93°, we observe no violations. As the temperature constraint is gradually increased from 80°, the violations fall down gradually with a *jump* at 83°. The *jump* is due to the addition of one more speed to the set of safe speeds that do not violate the given constraint. In all the cases, TA-DVS is able to reduce the number of constraint violations by an average of 14% over MF-DVS. As in the previous experiment, TASC results in a high number of constraint violations. Figure 7.9 shows the normalized energy consumption for the same experiment. The result shows that for TA-DVS and TASC, the energy consumption does not depend on the temperature constraint used, implying that temperature aware energy management does not increase the energy consumption. This result shows that temperature awareness does not adversely affect energy consumption.

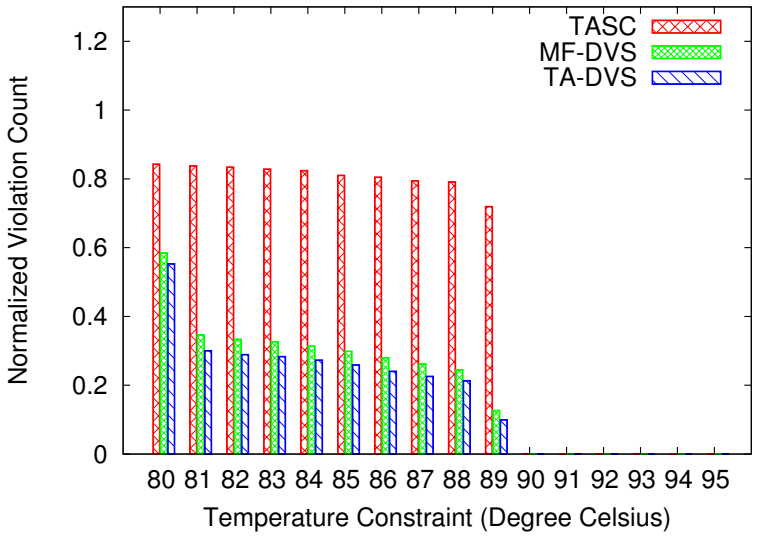


Fig. 7.10. Constraint Violations for WCET Utilization = 0.91 with Varying Temperature Constraints and Fine-Grained Speed Distribution

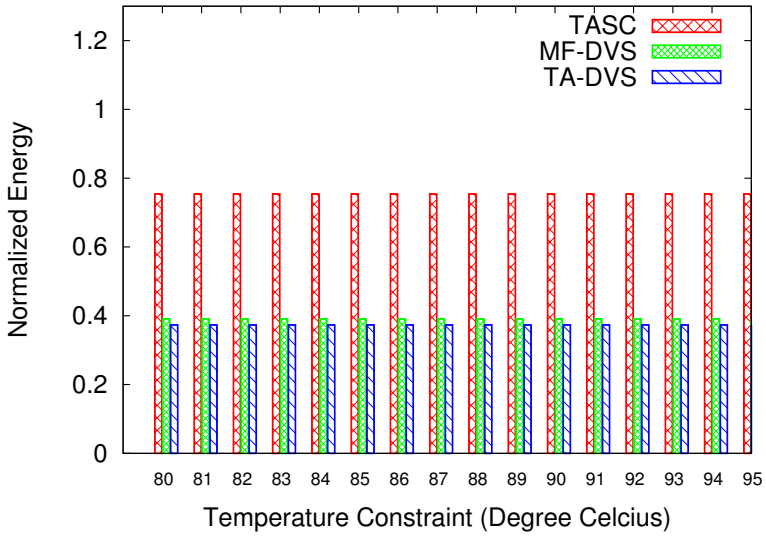


Fig. 7.11. Normalized energy for WCET utilization = 0.91 with varying temperature constraints and fine-grained speed distribution

In the third experiment, we consider a fine-grained speed distribution for the processor by considering hundred speed levels between 0 and 1 with increments of 0.01. We observe the constraint violations by varying the temperature constraint. Figure 7.10 shows the result for a taskset with utilization 0.91. It can be clearly seen that all the three schemes perform better compared to the coarse-grained distribution of speed levels that we assumed earlier. This is due to the increased room for utilizing the available slack. Although the equilibrium temperature for utilization 0.91 is 93.6°, we observe no constraint violations even with constraints of $90° - 95°$. This is due to the fact that early completions of tasks results in an actual utilization which is lesser than 0.91. In this experiment, TA-DVS reduces the number of constraint violations by an average of 14% over MF-DVS. Figure 7.11 shows the normalized energy consumption for this experiment yielding similar results as in Figure 7.9.

Table 7.1
Multimedia Benchmark for TA-DVS Evaluation

Task	Description	Period (sec)	WCET (sec)
mpegplay	MPEG video decoder	30	11
madplay	MP3 audio decoder	30	1
tmn	H263 video encoder	400	165
toast	GSM speech decoder	25	1
adpcm	ADPCM speech decoder	80	7

We also experimented with a set of multimedia tasks with a total worst case utilization of 0.94. The taskset consists of six programs as shown in Table 7.1. The execution times of these tasks were obtained by profiling offline traces. The violation count and energy consumption for the multimedia taskset during a 5000 sec interval are shown in Table 7.2. Similar to the results from the synthetic tasksets, TASC results in a high number of constraint violations and energy consumption. TA-DVS is able to reduce the number of constraint violations over MF-DVS by 8.03% while achieving similar energy consumption.

Table 7.2
Temperature Constraint Violations and Energy Consumption for $\Theta^* = 90°C$ Using the Multimedia Benchmark

Scheme	Violation count	Energy (J)
TASC	6165	138806
MF-DVS	5217	117380
TA-DVS	4798	116757

7.7 Conclusions and Future Work

Reducing temperature and energy consumption are important design constraints for modern computing devices. In this section, we propose a temperature aware energy management technique for real-time tasks with varying execution times. We use feedback control and DVS techniques to reduce energy consumption in a temperature aware manner. We experimentally showed that when the equilibrium speed is lesser than the canonical speed, a energy management that utilizes slack efficiently is sufficient to satisfy the temperature constraint. Further, our proposed scheme, TA-DVS, reduces temperature constraint violations by 18.9% on the average while consuming similar amount of energy, compared to an existing energy management technique. In the future TA-DVS will be extended to multiprocessor platforms. For this, we will extend the thermal model to consider the IC floorplan and the effect of the temperature of neighboring cores. The thermal model will be extended to account for fan controlled cooling.

8. CONCLUSIONS AND FUTURE WORK

Energy management for real-time systems is a challenging problem due to deadline constraints that tasks must obey. This dissertation has explored the usage Dynamic Voltage and Frequency Scaling capabilities of the underlying platform to reduce runtime energy consumption in real-time systems.

Pfair scheduling is an optimal scheduling algorithm for periodically recurrent real-time tasks, but suffers from applicability concerns in real systems. This dissertation has proposed techniques to address some of these issues.

This dissertation has also explore temperature aware dynamic power management for real-time tasks with varying execution times. This work brings out the similarities and differences in the characteristics of energy management and temperature awareness for real-time systems.

8.1 Summary

Section 4 considered the problem of employing DVFS to reduce energy consumption in Pfair scheduled real-time systems. First we showed how DVFS will violate the task deadlines in Pfair scheduling. Then we proposed our weight scaling approach to maintain real-time correctness while reducing energy consumption in Pfair scheduling. Comparison of our approach against the basic Pfair scheduling algorithm showed improvements of upto 66% in energy consumption. This work also proposed techniques to optimize task processor assignment to reduce overheads resulting out of task migrations and frequency switches.

Section 5 explored the problem of choosing a good quantum size for Pfair scheduling to reduce the practical overheads involved in implementing Pfair scheduling. First the overheads involved in Pfair scheduling were analyzed and then the *Quotient Search* heuristic was presented to choose a good quantum size. The proposed

technique was compared against other quantum size selection approached showing improvements in runtime and overheads.

In Section 6 a hardware block for Pfair scheduling was designed and implemented. We proposed the use of a centralized Pfair scheduler in multiprocessing systems to avoid repeated calculations. For hardware implementation, the definitions of Pfair parameters were suitably modified to transform them into the integer domain. Finally, comparison with other software based implementation approaches showed improvements in terms of energy consumption and scheduling delay.

In Section 7 we explored temperature aware DVFS in uni-processor systems using the EDF scheduling policy. This work proposes a solution to temperature aware energy management using task with varying execution time using a feedback controller based approach. The comparison of our approach with other existing approaches showed improvements in system temperature without adversely affecting energy management.

8.2 Future Work

We now describe a few of the challenges that are remaining in this research area. Most of the work on multiprocessor real-time scheduling still focuses on tasksets with independent tasks. The existing research on handling task dependencies with multiprocessor real-time scheduling is still immature. The challenge with handling task dependencies lies in the design of appropriate synchronization protocols that can prevent real-time priority inversions where a task with a higher priority waits for a resource held by a lower priority task. Extensions of PAPF to handle task dependencies will model real systems more accurately.

In the future, the number of processors in the chip may well increase beyond a number where the realization of a shared L2 cache becomes impractical. It will be interesting to study how the overheads in Pfair scheduling change in the absence of a global shared cache. In such a scenario, it might be useful the study restricted

migration of tasks so that a task is only allowed to migrate within cores which share an L2 cache.

Also, the accounting of interrupt overheads within Pfair scheduling has not been handled. It is not clear how interrupts should be modeled because the degree of interruptions is system dependent and varies largely with the workload [36]. However, the interrupt overheads need to be accounted for, and the execution times of tasks need to be appropriately adjusted.

The Hardware Pfair Scheduler presented in Section 6 can be made DVFS aware by integrating the work in Section 4. The challenge in this extension lies in designing the communication protocol between the scheduler core and the processing elements of the multiprocessor platform to keep the implementation fast and predictable. This will further improve the energy efficiency of the hardware scheduler.

It would be interesting to see extensions of the work on Temperature aware energy management for real-time systems. The challenge here lies with extending the thermal model appropriately to handle the temperature dependence of one core on its neighborhood cores. HotSpot is a thermal model which takes the floorplan of the circuit into account and allows the study of thermal evolution by developing an equivalent circuit of thermal resistances and capacitances. However, the suitability of HotSpot in dynamic online settings remains to be studied.

Yet another direction to be explored is the support for multiple quantum sizes in a Pfair scheduled system. In a task set with a mix of small and large tasks (in terms of execution time), the presence of multiple slot sizes can greatly improve implementation efficiency by preferring to execute large tasks on processors which use large quantum sizes; and preferring processors with small quantum sizes for tasks with small execution times.

REFERENCES

[1] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Isolation in commodity multicore processors," *IEEE Transactions on Computers*, vol. 40, no. 6, pp. 49–59, 2007.

[2] J. H. Anderson, "Real-time multiprocessor scheduling: Connecting theory and practice," in *Proceedings of the International Conference on Real-Time and Network Systems RTNS*, ser. 18, Nov. 2010.

[3] J. H. Anderson and A. Srinivasan, "Mixed Pfair/ERfair scheduling of asynchronous periodic tasks," *Journal of Computer and System Sciences*, vol. 68, pp. 157–204, February 2004.

[4] J. Anderson, J. Anderson, and A. Srinivasan, "Pfair scheduling: beyond periodic task systems," in *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications RTCSA*, 2000, pp. 297–306.

[5] J. Anderson, J. Calandrino, and U. Devi, "Real-time scheduling on multicore platforms," in *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium RTAS*, 2006, pp. 179–190.

[6] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *IEEE Transactions on Computers*, vol. 35, no. 2, pp. 59–67, 2002.

[7] T. M. Austin, D. Blaauw, T. N. Mudge, and K. Flautner, "Making typical silicon matter with razor," *IEEE Transactions on Computers*, vol. 37, no. 3, pp. 57–65, 2004.

[8] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez, "Dynamic and aggressive scheduling techniques for power-aware real-time systems," in *Proceedings of the IEEE Real-Time Systems Symposium RTSS*, 2001, pp. 95–105.

[9] N. Bansal, T. Kimbrel, and K. Pruhs, "Dynamic speed scaling to manage energy and temperature," in *Proceedings of the IEEE Symposium on Foundations of Computer Science FOCS*, 2004, pp. 520–529.

[10] M. Bao, A. Andrei, P. Eles, and Z. Peng, "On-line thermal aware dynamic voltage scaling for energy optimization with frequency/temperature dependency consideration," in *Proceedings of the Design Automation Conference DAC*, 2009, pp. 490–495.

[11] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, Jun. 1996.

[12] S. K. Baruah, J. Gehrke, and C. G. Plaxton, "Fast scheduling of periodic tasks on multiple resources," in *Proceedings of the 9th International Parallel Processing Symposium IPPS*, 1995, pp. 280–288.

[13] J. B. Bernstein, M. Gurfinkel, X. Li, J. Walters, Y. Shapira, and M. Talmor, "Electronic circuit reliability modeling," *Microelectronics Reliability*, vol. 46, no. 12, pp. 1957–1979, 2006.

[14] B. Brandenburg, J. Calandrino, and J. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms: A case study," in *Proceedinngs of the IEEE Real-Time Systems Symposium RTSS*, 2008, pp. 157–169.

[15] B. B. Brandenburg and J. H. Anderson, "On the implementation of global real-time schedulers," in *Proceedings of the IEEE Real-Time Systems Symposium RTSS*, 2009, pp. 214–224.

[16] J. M. Calandrino and J. H. Anderson, "Quantum support for multiprocessor Pfair scheduling in Linux," in *Proceedings of the 2nd International Workshop on Operating System Platforms for Embedded Real-Time Applications*, 2006.

[17] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LITMUS^{RT} : A testbed for empirically comparing real-time multiprocessor schedulers," in *Proceedings of the 27th IEEE Real-Time Systems Symposium RTSS*. IEEE, 2006, pp. 111–126.

[18] T. Chantem, R. P. Dick, and X. S. Hu, "Temperature-aware scheduling and assignment for hard real-time applications on MPSoCs," in *Proceedings of the Conference on Design, Automation and Test in Europe DATE*, 2008, pp. 288–293.

[19] J.-J. Chen, C.-Y. Yang, T.-W. Kuo, and S.-Y. Tseng, "Real-time task replication for fault tolerance in identical multiprocessor systems," in *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium RTAS*, 3-6 April 2007, pp. 249–258.

[20] 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 IEEE Real-Time and Embedded Technology and Applications Symposium RTAS*, 2008, pp. 13–23.

[21] L. Clark, E. Hoffman, J. Miller, M. Biyani, L. Liao, S. Strazdus, M. Morrow, K. Velarde, and M. Yarch, "An embedded 32-b microprocessor core for low-power and high-performance applications," *IEEE Journal of Solid-State Circuits*, vol. 36, no. 11, pp. 1599–1608, 2001.

[22] F. Clarke, *Optimization and nonsmooth analysis*. Society for Industrial Mathematics, 1990, vol. 5.

[23] A. K. Coskun, R. Strong, D. M. Tullsen, and T. Simunic Rosing, "Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors," in *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '09. ACM, 2009, pp. 169–180.

[24] A. K. Coskun, T. S. Rosing, and K. Whisnant, "Temperature aware task scheduling in MPSoCs," in *Proceedings of the Conference on Design, Automation and Test in Europe DATE*, 2007, pp. 1659–1664.

[25] K. Danne, R. Miihlenbernd, and M. Platzner, "Executing hardware tasks on dynamically reconfigurable devices under real-time conditions," in *Proceedings of the International Conference on Field Programmable Logic and Applications FPL*. IEEE, 2006, pp. 1–6.

[26] V. Devadas and H. Aydin, "Real-time dynamic power management through device forbidden regions," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium RTAS*, 2008, pp. 34–44.

[27] U. Devi, "Soft real-time scheduling on multiprocessors," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2006.

[28] S. Feng, S. Gupta, A. Ansari, and S. A. Mahlke, "Maestro: Orchestrating lifetime reliability in chip multiprocessors," in *High Performance Embedded Architectures and Compilers HiPEAC*, ser. Lecture Notes in Computer Science, vol. 5952. Springer, 2010, pp. 186–200.

[29] V. W. Freeh, D. K. Lowenthal, F. Pan, N. Kappiah, R. Springer, B. L. Rountree, and M. E. Femal, "Analyzing the energy-time trade-off in high-performance computing applications," *IEEE Transactions on Parallel Distributed Systems*, vol. 18, pp. 835–848, June 2007.

[30] S. Funk, G. Levin, C. Sadowski, I. Pye, and S. Brandt, "DP-fair: a unifying theory for optimal hard real-time multiprocessor scheduling," *Real-Time Systems*, vol. 47, no. 5, pp. 389–429, 2011.

[31] N. Gupta and R. Mahapatra, "Temperature aware energy management for real-time scheduling," in *Proceedings of the 12th International Symposium on Quality Electronic Design ISQED*, March 2011, p. 6.

[32] N. Gupta, S. K. Mandal, J. Malave, A. Mandal, and R. N. Mahapatra, "A hardware scheduler for real time multiprocessor system on chip," in *Proceedings of the 23rd International Conference on VLSI Design*, 2010, pp. 264–269.

[33] V. Gupta, "Finding the optimal quantum size: Sensitivity analysis of the M/G/1 round-robin queue," *SIGMETRICS Performance Evaluation Review*, vol. 36, no. 2, pp. 104–106, 2008.

[34] J. Hildebrandt and D. Timmermann, "An FPGA based scheduling coprocessor for dynamic priority scheduling in hard real-time systems," in *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2000, vol. 1896, ch. 83, pp. 777–780.

[35] P. Holman and J. H. Anderson, "Adapting Pfair scheduling for symmetric multiprocessors," *Journal of Embedded Computing*, vol. 1, no. 4, pp. 543–564, 2005.

[36] P. L. Holman, "On the implementation of Pfair-scheduled multiprocessor systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2004.

[37] T. Horvath and K. Skadron, "Multi-mode energy management for multi-tier server clusters," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques PACT*, 2008, pp. 270–279.

[38] W.-L. Hung, Y. Xie, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Thermal-aware task allocation and scheduling for embedded systems," in *Proceedings of the Conference on Design, Automation and Test in Europe DATE*, 2005, pp. 898–899.

[39] R. Jayaseelan and T. Mitra, "Temperature aware task sequencing and voltage scaling," in *Proceedings of the International Conference on Computer-Aided Design ICCAD*, 2008, pp. 618–623.

[40] R. Jejurikar and R. K. Gupta, "Dynamic slack reclamation with procrastination scheduling in real-time embedded systems," in *Proceedings of the Design Automation Conference DAC*. ACM, 2005, pp. 111–116.

[41] W. Kim, J. Kim, and S. L. Min, "A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis," in *Proceedings of the Conference on Design, Automation and Test in Europe DATE*, 2002, pp. 788–794.

[42] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval Research Logistics*, vol. 52, no. 1, pp. 7–21, 2005.

[43] S. Kumar, C. Hughes, and A. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 162–173, 2007.

[44] K. Li, "Performance analysis of power-aware task scheduling algorithms on multiprocessor computers with dynamic voltage and speed," *IEEE Transactions on Parallel Distributed Systems*, vol. 19, no. 11, pp. 1484–1497, 2008.

[45] J. W. McPherson, "Reliability challenges for 45nm and beyond," in *Proceedings of the 43rd Annual Design Automation Conference DAC*, 2006, pp. 176–181.

[46] R. Mishra, N. Rastogi, D. Zhu, D. Mossé, and R. Melhem, "Energy aware scheduling for distributed real-time systems," in *Proceedings of the 17th International Parallel and Distributed Processing Symposium IPDPS*, 2003, p. 21.

[47] V. Mooney, T. Sakamoto, and G. De Micheli, "Run-time scheduler synthesis for hardware-software systems and application to robot control design," in *Proceedings of the 5th International Workshop on Hardware/Software Codesign CODES/CASHE*, 1997, pp. 95–99.

[48] A. C. Nácul, F. Regazzoni, and M. Lajolo, "Hardware scheduling support in SMP architectures," in *Proceedings of the Conference on Design, Automation and Test in Europe DATE*, 2007, pp. 642–647.

[49] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai, "Hardware implementation of a real-time operating system," in *Proceedings of the 12th TRON Project International Symposium*. IEEE, 1995, pp. 34–42.

[50] F. Paterna, L. Benini, A. Acquaviva, F. Papariello, G. Desoli, and M. Olivieri, "Adaptive idleness distribution for non-uniform aging tolerance in multiprocessor systems-on-chip," in *Proceedings of the Conference on Design, Automation and Test in Europe DATE*, 2009, pp. 906–909.

[51] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001, pp. 89–102.

[52] D. Roberts, R. Dreslinski, E. Karl, T. Mudge, D. Sylvester, and D. Blaauw, "When homogenous becomes heterogenous: Wearout aware task scheduling for streaming applications," in *Proceedings of the Workshop on Operating System Support for Heterogenous Multicore Architectures OSHMA*, 2007.

[53] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," *SIGARCH Comput. Archit. News*, vol. 31, no. 2, pp. 2–13, 9-11 June 2003.

[54] A. Srinivasan, P. Holman, J. Anderson, S. Baruah, and J. Kaur, "Multiprocessor scheduling in processor-based router platforms: Issues and ideas," in *Proceedings of the 2nd Workshop on Network Processors*, 2003, pp. 48–62.

[55] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah, "The case for fair multiprocessor scheduling," in *Proceedings of the International Parallel and Distributed Processing Symposium IPDPS*, 2003, p. 10.

[56] A. Srinivasan and J. H. Anderson, "Fair scheduling of dynamic task systems on multiprocessors," *Journal of Systems and Software*, vol. 77, no. 1, pp. 67–80, Jul. 2005.

[57] Synopsys. (2009) Verilog simulator. [Online]. Available: http://www.synopsys.com/products/simulation/simulation.html

[58] (2009) Sim-panalyzer: The simplescalar arm power modelling project. University of Michigan & the University of Colorado. [Online]. Available: http://www.eecs.umich.edu/~panalyzer/

[59] V. živojnović, J. M. Velarde, C. Schläger, and H. Meyr, "DSPSTONE: A DSP-oriented benchmarking methodology," in *Proceedings of the International Conference on Signal Processing and Technology ICSPAT*, 1994.

[60] S. Wang, J. Chen, Z. Shi, and L. Thiele, "Energy-Efficient speed scheduling for Real-Time tasks under thermal constraints," in *Proceedings of the International Conference on Embedded and Real-Time Computing Systems and Applications RTCSA*, 2009, pp. 201–209.

[61] C. Xian, Y.-H. Lu, and Z. Li, "Energy-aware scheduling for real-time multiprocessor systems with uncertain task execution time," in *Proceedings of the 44th ACM/IEEE Design Automation Conference DAC*, 2007, pp. 664–669.

[62] C. Xian, Y.-H. Lu, and Z. Li, "Dynamic voltage scaling for multitasking real-time systems with uncertain execution time," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 8, pp. 1467–1478, 2008.

[63] R. Xu, C. Xi, R. G. Melhem, and D. Mossé, "Practical PACE for embedded systems," in *Proceedings of the 4th ACM International Conference on Embedded Software*, 2004, pp. 54–63.

[64] W. Yuan and K. Nahrstedt, "Energy-efficient soft real-time CPU scheduling for mobile multimedia systems," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003, pp. 149–163.

[65] D. Zhu, R. Melhem, and B. Childers, "Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 7, pp. 686–700, 2003.

[66] Y. Zhu and F. Mueller, "Feedback EDF scheduling of real-time tasks exploiting dynamic voltage scaling," *Real-Time Systems*, vol. 31, no. 1, pp. 33–63, 2005.

APPENDIX A
SIMULATION BASED TOOLS

Most of the work in this dissertation was evaluated using simulation based tools. This section provides further details on these tools to give greater insight into the evaluation process.

For the evaluation of PAPF in section 4, a Java based simulator was developed which implements PAPF over the basic Pfair scheduling algorithm. The simulator also supports the evaluation of the optimization schemes presented in the section. The Java classes involved in the simulator and their descriptions are mentioned in Table A.1.

Table A.1
List of Classes in PAPF Simulator

Class	**Description**
Simulator	The main Java class that instantiates other classes within the simulator and keeps track of time
Dispatcher	Responsible for Job release, DVFS and weight scaling
ExecutionReport	Notifies whether a job finished in time or earlier than time
Job	Job representation
PFScheduler	The Pfair scheduling algorithm logic
ReadySet	Representation of the set of ready jobs
Report	One ExecutionReport for each Processor in the system
Slack	The representation of a slack element
SlackManager	Responsible for computing slack usable by a given task
SlackSet	The set of available dynamic slack in the system
Task	Task representation
Utils	Utility functions

The evaluation for quantum size selection schemes presented in section 5 was performed in another Java simulator. This simulator incorporates the overhead model and the selection heuristics mention in section 5. Table A.2 lists the classes involved in this simulator.

Table A.2
List of Classes in Quantum Size Selection Scheme Evaluator

Class	Description
Main	The main Java class which instantiates other Java classes and produces output
TaskOb	A task object
Taskset	A set of tasks
OptimizeExhaustive	Exhaustive search optimization scheme
OptimizeMean	Mean execution time based selection
OptimizeMedian	Median execution time based selection
OptimizeRandom	Random execution time based selection
OptimizeQuotient	Our proposed Quotient Search scheme
OptimizeLITMUS	The LITMUS^{RT} quantum size approach

The evaluation of TA-DVS in section 7 was performed on another discrete event Java simulator which implements Feedback control, task splitting and α-queue techniques. The simulator also incorporates the thermal and energy model used in the section. Table A.3 lists the classes involved in the simulator.

Table A.3
List of Classes in TA-DVS Scheduler

Class	Description
Simulator	The main Java class which instantiates other Java classes and tracks time
Task	A task object
SystemModel	Class incorporating the system model for this work
AlphaQueue	Implementation of α-queue
EventType	Enum for representing different event types
Event	An event in the discrete event simulator
EventManager	The class that manages events in the simulator based on event time
FeedbackController	Feedback control implementation
Pattern	Class for producing the run time variation of task execution time
ReadyQueue	The ready queue of the EDF scheduler
RunType	An enum representing whether the CPU is idle or busy

VITA

Nikhil Gupta

ngupta@cse.tamu.edu

514A HRBB, Department of Computer Science & Engineering

Texas A&M University

Publications (complete list at http://students.cs.tamu.edu/ngupta/research.html)

- **N. Gupta** and R. N. Mahapatra, *"Power Aware Pfair Scheduling in Multiprocessor Real Time Systems"*, Cool Work In Progress session at Design and Automation Conference (DAC), 2011.
- **N. Gupta** and R. N. Mahapatra, *"Temperature Aware Energy Management for Real-Time Scheduling"*, Proceedings of the International Symposium on Quality Electronic Design (ISQED), pp. 1-6, 2011.
- **N. Gupta**, S. K. Mandal, A. Mandal, J. Malave & R. N. Mahapatra, *"A Hardware Scheduler for Real Time Multiprocessor System on Chip"*, 23rd International Conference on VLSI Design, pp.264-269, 2010.

Education

Texas A & M University, Texas, USA	**Indian Institute of Technology, India**
PhD in Computer Science August 2006 - December 2011; GPA : 4.0/4.0 **Dissertation Title:** Energy Efficient Scheduling for Real-Time Systems	Bachelor of Technology in Computer Science & Engineering July 2002 - April 2006; Major CGPA (Computer Science) : 8.64/10

Skills

Languages: C, C++, Java, Perl, 80x86 Assembly, VHDL, LaTeX

Operating Systems: Linux/Unix, Windows, Solaris